МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 30.00.00.000 ПЗ

Група ІПм-23-2

Кулинич Олег

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Кулинич Олег Миколайович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі, методи та алгоритми оптимізації нейронних мереж для задач комп'ютерного зору

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Кулинич О.М.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Лютак Ігор Зіновійович, д.т.н., професор

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітньо-кваліфікований рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

***ЗАТВЕРДЖУЮ*:**
Зав. кафедрою ***ІПЗ***
доц. В.В. Бандура
“ 04 ” вересня 2022 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Кулинич Олегу Миколайовичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) “Моделі, методи та алгоритми оптимізації нейронних мереж для задач комп’ютерного зору”
керівник проекту (роботи) Лютак Ігор Зіновійович, д.т.н., професор
затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Порівняння та опис методів оптимізації нейронних мереж, розгляд нейронних моделей для задач комп’ютерного зору

4. Зміст розрахунково - пояснювальної записки (перелік питань, які потрібно розробити)

1. Теоретичні основи комп’ютерного зору та аналіз предметної області
2. Моделі нейронних мереж для задач комп’ютерного зору
3. Методи та алгоритми оптимізації нейронних мереж
4. Експериментальне дослідження та порівняння алгоритмів оптимізації нейронних мереж

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Порівняння функції втрат градієнтних методів по ітераціях (Рис. 4.1. ст. 85)
2. Точність методів графічної оптимізації за епохами (Рис. 4.3. ст. 86)
3. Функція втрат градієнтних методів по ітераціях (ResNet18) (Рис. 4.4. ст. 87)
4. Результати порівняння методів Adam та Nadam (Рис. 4.7. ст. 90)
5. Порівняння точності навчання з Dropout (Рис. 4.15. ст. 97)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц. к.т.н. Вовк Р. Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів проекту	Примітка
1	Вибір теми, пошук та вивчення літератури	21.09.2024	Виконано
2	Опис предметної області, аналіз сучасних рішень комп'ютерного зору	03.10.2024	Виконано
3	Розгляд популярних моделей для задач CV	09.10.2024	Виконано
4	Вивчення методів оптимізації нейронних моделей	27.10.2024	Виконано
5	Експериментальне дослідження різних методів оптимізації для визначення оптимальних	10.11.2024	Виконано
6	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	Виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 104 с., 26 рис., 5 табл., 41 джерело.

Тема: Моделі, методи та алгоритми оптимізації нейронних мереж для задач комп’ютерного зору.

Об’єктом дослідження є нейронні мережі для виконання задач комп’ютерного зору та методи їхньої оптимізації.

Мета роботи полягає в дослідженні методів та алгоритмів оптимізації моделей нейронних мереж у реальних задачах комп’ютерного зору..

Предметом дослідження є методи оптимізації нейронних мереж, для зменшення ресурсозатратності моделі, без втрати точності роботи.

Результати дослідження: Виконано експериментальне порівняння методів оптимізації та проведено їхній аналіз.

Висновок: В результаті дослідження реалізовано методи оптимізації нейронних мереж та оптимізовано нейронну модель ResNet18, для визначення оптимальних методів, які вирішують проблеми оптимізації для різних нейронних моделей, показано способи регуляризації та її вплив на навчання мережі.

КОМП’ЮТЕРНИЙ ЗІР, НЕЙРОННА МОДЕЛЬ, НЕЙРОННА МЕРЕЖА, ОПТИМІЗАЦІЯ, CNN, РЕГУЛЯРИЗАЦІЯ, ГІПЕРПАРАМЕТРИ, МЕТОД, ГРАДІЄНТНИЙ СПУСК

ANNOTATION

Master's thesis: 104 p., 26 fig., 5 tab., 41 sources.

Theme: Models, methods and algorithms for optimising neural networks for computer vision tasks.

The **object of research** is neural networks for computer vision tasks and methods of their optimisation.

The **purpose** of the study is to investigate methods and algorithms for optimising neural network models in real computer vision problems.

The **subject of research** is the methods of optimising neural networks to reduce the resource consumption of the model without losing accuracy.

Research results: An experimental comparison of optimisation methods was performed and their analysis was carried out.

Conclusion: As a result of the study, neural network optimisation methods were implemented and the ResNet18 neural model was optimised to determine the optimal methods that solve optimisation problems for different neural models, and methods of regularisation and its impact on network training were shown.

COMPUTER VISION, NEURAL MODEL, NEURAL NETWORK, OPTIMISATION, CNN, REGULARISATION, HYPERPARAMETERS, METHOD, GRADIENT DESCENT

ЗМІСТ

Стр.

РОЗДІЛ 4

ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ПОРІВНЯННЯ АЛГОРИТМІВ ОПТИМІЗАЦІЇ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CV – Computer Vision, комп’ютерний зір

CNN - Convolutional Neural Networks, згорткові нейронні мережі

ШІ – Штучний ітелект

YOLO – You Only Look Once, алгоритм для виявлення об’єктів

ViT – Vision Transformer, трансформер для задач комп’ютерного зору

GD – Gradient Descent, базовий метод градієнтного спуску

SGD – Stochastic Gradient Descent, стохастичний градієнтний спуск

NAG – Nesterov Accelerated Gradient, вдосконалення методу імпульсу з додаванням передбачення

NAS – Neural Architecture Search, метод для оптимізації пошуку гіперпараметрів

ВСТУП

Актуальність роботи

Темп розвитку технологій в сучасному світі є дуже швидким, охоплюючи все більше й більше сфер людського життя. Дещо лиш декілька років назад здавалось чимось неможливим. Серед таких технологій значуще місце займає штучний інтелект (ШІ), його розвиток почався ще в 50-х роках минулого століття, але він був дуже нерівномірним [1]. Але останніми роками інтерес відновився й все більше компаній почали займатись його розвитком, та кількість таких компаній чи команд розробників лиш збільшується. Через це ми можемо помітити шалені темпи розвитку різних ШІ-моделей, які створені для різних сфер: фінансова, освітня, медицина й тд. Однією з ключових областей штучного інтелекту є комп'ютерний зір – технологія, що дозволяє машинам «бачити» й аналізувати «побачену» інформацію та видавати якійсь результати чи поради, саме її я вибрав для дослідження [2].

На мою думку, комп'ютерний зір, на даний момент, є однією із найбільш перспективних та потужних галузей сучасних технологій. Перелічувати сфери в яких він використовується, або може використовуватись, можна дуже довго, для прикладу: у медицині він допомагає виявляти захворювання на ранніх стадіях, аналізуючи зображення рентгенівських знімків, МРТ й тд; в сучасних автомобілях він дозволяє розпізнавати пішоходів, дорожні знаки, розмітку та інші тз в реальному часі, це й є ключем для повного авто-пілоту в майбутньому транспорті; розпізнавання людей чи предметів в різних системах безпеки. Всі ці системи мають за основу нейронні мережі – це математичні моделі, які працюють за принципом людського мозку. [3-5]

Але, незважаючи на високу ефективність, сучасні нейронні мережі мають деякі проблеми. Ці моделі потребують дуже багато ресурсів для навчання чи виконання завдань, особливо тоді, коли вона працює з величезними обсягами даних. Часто навіть для виконання простих завдань вони вимагають потужного апаратного забезпечення. А це викликає великі складнощі, особливо, коли їх

планують використовувати на пристроях, які мають обмежені ресурси, наприклад, як смартфони чи інші мобільні пристрої. Часто в таких ситуаціях використання таких моделей є неможливим, адже тут важливо не тільки точність її роботи, а й її швидкість обчислень та енергоефективність [6].

Тому ключовим завданням в таких випадках є оптимізація нейронних мереж. Саме оптимізація допоможе вирішити проблеми з роботою моделей в умовах обмежених ресурсів, вона дозволить зробити моделі більш швидшими, менш ресурсозатратними й це все без втрати точності й ефективності їхньої роботи.

Таким чином, дослідження моделей, методів та алгоритмів оптимізації моделей нейронних мереж є дуже актуальною темою. Вона допоможе краще розробляти такі моделі, зменшувати споживання ресурсів, що дозволить інтегрувати їх в більшу кількість різних систем. Це відкриє доступ до технологій комп'ютерного зору більшому колу користувачів і відкриє нові можливості його застосування та покращення ефективності.

Мета і задачі дослідження

Мета роботи полягає в дослідженні методів та алгоритмів оптимізації моделей нейронних мереж у реальних задачах комп'ютерного зору.

Для досягнення цієї мети потрібно провести огляд сучасних методів та моделей, які використовуються для вирішення задач комп'ютерного зору, проаналізувати їхні переваги та недоліки. Також проаналізувати існуючі методи та алгоритми оптимізації нейронних мереж. Окрім того, розглянути застосування методів оптимізації у комп'ютерному зорі. Провести дослідження ефективності різних методів та порівняти їх. Сформулювати рекомендації щодо застосування у різних умовах та визначити перспективи подальших досліджень.

Об'єктом дослідження є нейронні мережі для виконання задач комп'ютерного зору та методи їхньої оптимізації.

Предметом дослідження є методи оптимізації нейронних мереж, для зменшення ресурсозатратності моделі, без втрати точності роботи.

Методи дослідження

Було проведено аналіз сучасних моделей для задач комп'ютерного зору та можливостей використання методів оптимізації Також проведено порівняльний аналіз між існуючими рішеннями оптимізації нейронних мережю.

Наукова новизна одержаних результатів

У роботі досліджено різні сучасні методи оптимізації нейронних мереж для задач комп'ютерного зору, проведено їхній порівняльний аналіз, що дозволяє вдосконалити розробку більш ефективних й компактних нейронних мереж.

Практичне значення одержаних результатів

В результаті проведеного дослідження було реалізовано різні методи оптимізації, з їхньою допомогою було оптимізовано нейронну мережу, також визначені оптимальні методи для виконання цього завдання, тому результати дослідження можуть використовуватись для розробки систем комп'ютерного зору, для збільшення їхньої швидкості та зменшення вимог до обчислювальних ресурсів, зокрема для інтеграції в мобільні пристрої.

Структура магістерської роботи.

Магістерська робота викладена в 110 сторінках друкованого тексту, що складається із вступу, чотирьох розділів, висновків та списку використаних джерел (41 джерело). Робота містить 28 рисунків та 3 таблиці.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ КОМП'ЮТЕРНОГО ЗОРУ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Теоретичні відомості про комп'ютерний зір

1.1.1. Історія розвитку комп'ютерного зору

Історія розвитку технологій комп'ютерного зору є досить захопливою, адже вона охоплює не одне десятиліття. Початки даної технології можна простежити аж в 1950-х роках, ще тоді дослідники й інженери почали досліджувати можливість навчання комп'ютерів розуміти й інтерпретувати візуальні дані [1].

Одним із перших кроків зробив Френк Розенблат, саме він розробив примітивну штучну нейронну мережу «Перцептрон», саме вона заклала основу для систем комп'ютерного зору на основі нейронних мереж. Ця модель складалась із трьох елементів: S-елементи – це сигнали, які надходять від сенсорів (або ж рецепторів), A-елементи – асоціативні елементи та R-елементи – реагуючі. [7]

Першими включаються S-елементи, вони передаються A-елементам по зв'язкам S-A (ці зв'язки мають ваги, які можуть бути рівні -1, 0, 1, ці ваги генеруються рандомно і не змінюються), якщо сигнали, які потрапили до A в сумі перебільшують, якийсь певний поріг, то цей A-елемент збуджується і видає сигнал 1, в противному випадку 0. Далі збуджені елементи A йдуть до R (зв'язки A-R також мають ваги, але в даному випадку вони можуть мати абсолютно різні значення). В кінцевому етапі цього процесу підсумовуються зважені сигнали A-елементів й якщо буде пройдений певний поріг, тоді генерується сигнал 1 – це означає, що елемент розпізнано, в противному випадку генерується -1. Цей процес показаний на рисунку 1.1.

Навчання в цьому процесі полягає саме в зміні ваг A-R зв'язків, повністю навчена модель повністю готова працювати в режимі розпізнавання, для

прикладу перша така модель розроблена Розенблатом могла розпізнавати букви англійського алфавіту [7].

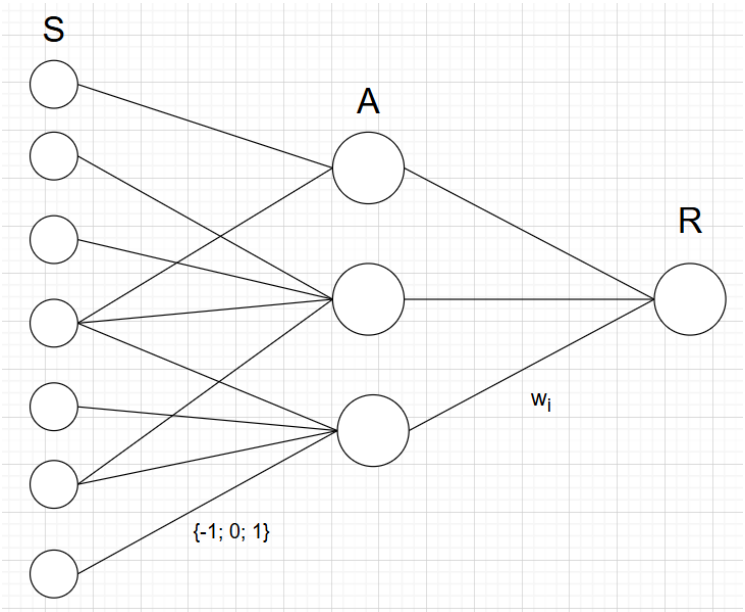

Рис. 1.1. Модель роботи перцептрона.

Всі сильно надіялись на перцептрона, але як виявилось, він мав багато недоліків та обмежень, через це дослідники відійшли від цього.

Наступний період – це 70-80-ті, саме тут були досліджені та розроблені алгоритми для виявлення країв і вилучення ознак із зображення. Це основна техніка, завдяки якій комп’ютер може «побачити» різницю між небом і травою на фото.

Ці алгоритми виявляють межі й краї різних об’єктів чи областей в зображенні, краї відображають значні зміни інтенсивності або кольору та надають інформацію про структуру та вміст зображення. А краї в свою чергу є підказкою для класифікації та ідентифікації об’єктів [8].

В 80-90-ті почали досліджувати розпізнавання об’єктів за допомогою методів машинного навчання. І в ці роки ця сфера пережила стрімке зростання, яке відбулось через активний розвиток технологій й збільшення обчислювальних потужностей, а це означало, що можна переходити до складніших алгоритмів, адже попередні були ефективними для базових завдань.

У 2001 році відбулась революція в цій сфері, а саме розробка алгоритму Віоли-Джонса, яка дозволяла розпізнавати обличчя в реальному часі. Він був дійсно революційним й ефективним не тільки для свого часу, а й в теперішній час. Справедливо врахувати, що все-таки сучасні алгоритми є більш ефективними та точними (для прикладу CNN), але взявши до уваги його компактність його досі використовують у випадках, якщо обчислювальна потужність є невеликою. Цей алгоритм дуже повільно навчається, але може виявляти обличчя з вражаючою швидкістю в таких умовах [9].

Алгоритм отримує зображення, яке розділяє на багато менших областей і намагається знайти в них обличчя, шукаючи конкретні ознаки в кожній області.

Наступна революція сталась після прориву в CNN, адже вони виявились дуже ефективними для сфери комп'ютерного зору. А потім вже почали появлятись інші відомі архітектури, які використовуються в сучасності AlexNet, VGGNet, ResNet і тд.

Розвиток в розробці глибоких нейронних мереж почав активніше використовуватись в сфері комп'ютерного зору й відкривати нові «горизонти» для неї [10].

1.1.2. Поняття та основні задачі комп'ютерного зору

Зір – є одним із основних органів відчуття людини, саме через нього людина отримує більшість інформації з навколишнього світу, він дозволяє нам сприймати та інтерпретувати її. Підсвідомо ми аналізуємо всю інформацію й можемо, наприклад, впізнавати предмети чи людей на фото, при зустрічі, більшість людей ніколи не замислюється, а як їй це вдається. Але ось для машини навчитись виконувати такі речі є дуже складною задачею, але це й є основним завданням комп'ютерного зору.

Комп'ютерний зір – це технологія ШІ, яка використовує нейронні мережі для того, щоб навчити машини отримувати інформацію із фото, відео або інших візуальних даних. Він дозволяє комп'ютерам аналізувати дану інформацію та

видавати певні результати [2].

Якщо так глянути, то комп’ютерний зір працює на основі нашого, він імітує його, тільки замість очей в нього камери, за допомогою нейронних мереж він навчається виконувати ці функції. Для цього йому потрібно «згодувати» велику кількість даних, він багато разів аналізує ці дані, поки не помітить різницю й не зможе точно розпізнати об’єкт на зображенні.

Для прикладу, щоб навчити комп’ютер розпізнавати велосипед на фото, йому потрібно дати велику кількість зображень велосипедів і об’єктів, які якось пов’язані із велосипедами, щоб навчитись «бачити» відмінності та точно сказати, де саме на фото велосипед. Для виконання цих завдань використовують глибоке навчання («Deep Learning») та згорткову нейронну мережу (CNN), але про це далі детальніше [10].

Є основні задачі, які повинен виконувати комп’ютерний зір:

1. Виявлення об’єктів – це вміння точно розпізнавати об’єкти на будь якому зображенні чи відео та визначати їх точне розташування за допомогою прямокутних рамок, які містять предмет, які називаються обмежувальними рамками (рис. 1.2.) [11].

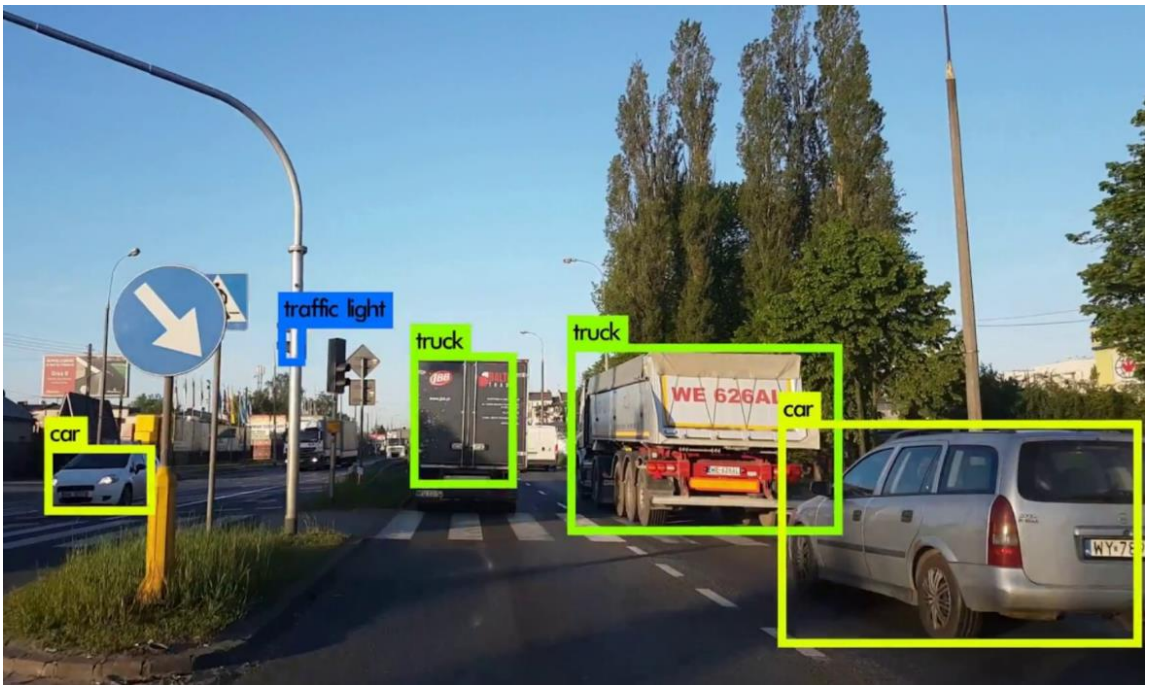

Рис. 1.2. Виявлення об’єктів на фото

Коли хтось говорить за комп’ютерний зір, то кожен зразу уявляє собі подібну картину в голові, як на рисунку 1.2. На фото можна помітити, що комп’ютер розмежовує рамками автомобілі (жовтий), вантажівки (зелений) та світлофор (синій).

2. Класифікація – визначення класу, до якого належить об’єкт. Тут все просто, коли машина отримує зображення машини, то вона видає результат, ймовірність того, що це машина чи якийсь інший об’єкт. Виявлення об’єктів, про яке писав вище, є більш просунутою версією класифікації [12].
3. Сегментація – це ще більш просунута еволюція класифікації та виявлення об’єктів. Сама сегментація на піксельному рівні ідентифікує об’єкти на зображенні, які належать до одного класу, щоб вказати, що вони належать до одного класу, вона зафарбовує їх однаково, або ж позначає (рис. 1.3.).

Рис. 1.3. Сегментація

Існують декілька видів сегментації:

- Семантична сегментація: призначає категорію кожному пікселю.

- Сегментація екземплярів: ідентифікує кожен екземпляр в межах певної категорії.
- Сегментація країв: визначає форму об’єкта, шляхом визначення країв.
- Сегментація за регіонами: використовує такі характеристики, щоб групувати пікселі відповідно до схожості.
- Сегментація на основі кластеризації: групує пікселі за допомогою методів кластеризації.

Тобто підсумуємо основні задачі, на явному прикладі, шукаємо машини на фото. Класифікація – комп’ютер визначає чи є на зображенні предмети класу «Машина». Виявлення об’єктів – це не тільки класифікація, а й визначення їхнього розташування на фото. Семантична сегментація – визначає кожен піксель до конкретного класу, але він не розрізняє кожен об’єкт, як окремий екземпляр, тобто, якщо в нас на фото дві машини поруч, то результатом буде одна довга машина. А ось сегментація екземплярів – визначає кожен екземпляр певного класу, його межі [12-13].

1.2. Аналіз сучасних рішень у задачах комп’ютерного зору

Як вже можна зрозуміти, що комп’ютерний зір має дуже насичену історію. Все перейшло від основ виявлення країв до методів глибокого навчання та CNN. Але незважаючи на те, що зараз зазвичай використовуються саме методи глибокого навчання, в особливих випадках досі залишаються актуальними традиційні методи.

1.2.1. Використання базових методів обробки зображення.

Класичні методи обробки зображень зосереджені на математичних й алгоритмічних процедурах, без використання навчання моделей.

Однією із задач таких методів є визначення країв, вона дозволяє виділяти контури об’єктів, що є вирішальним завданням для аналізу зображень.

Найпоширенішим таким алгоритмом є Canny, який був розроблений Джоном. Ф. Кенні в 1986 році. Він став поширеним завдяки своїй надійності та точності [8].

Canny використовує багатоетапний метод для визначення різноманітних країв на зображеннях. Насправді схожих методів є багато, одними із популярних ще є Sobel та Laplacian, але саме Canny дав найбільший вплив.

Взагальному алгоритм складається із таких етапів:

1. Перетворення в відтінки сірого. Зображення з кольорового перетворюється на відтінки сірого, це можна робити різними методами.
2. Наступним кроком є зниження шуму. Тут також є багато методів, але в зазвичай використовують шумоподавлення за допомогою фільтра Гауса. Це є важливим кроком для алгоритму, адже шум створює зайву інформацію для алгоритму, а також може бути сприйнятий як контур.
3. Розрахунок градієнта. В даному кроці ми шукаємо градієнти інтенсивності зображення, адже якщо так подумати, то можна сказати, що край можна визначити як місце, де різко змінюється колір або інтенсивність.
4. Придушення немаксимумів, використовується, щоб отримати тонку лінію контуру. Ми проходимо циклом по всіх пікселях і вибираємо два сусідні пікселі, щоб визначити, чи є інтенсивність пікселя більшою, ніж двох сусідніх, якщо так ми продовжуємо далі, якщо ні, то ставимо значення 0.
5. Подвійний поріг. Використовуються два порогові значення, перше нижче за друге, кінцеві краї включають пікселі градієнтів, які перевищують вищий поріг, їх називають сильними, нижчі за нижній поріг пригнічуються. Ті, що знаходяться між враховуються на наступному етапі, це слабкі.
6. Гістерезис. Перевіряється чи слабкі пікселі є зв’язані із сильними, якщо ні, вони відкидаються, якщо так, це формує кінцеві краї об’єкта (рис.

1.4.).

Даний алгоритм також може включати алгоритми Sobel та інші, щоб пришвидшувати та покращувати його. Також він використовується й для сегментації об’єктів на зображенні за краями [8].

Рис. 1.4. Результат виконання Canny

Задача виявлення ознак є важливим кроком у багатьох програмах комп’ютерного зору. Вона передбачає пошук важливих деталей чи характеристик на зображенні. Тут також є декілька основних методів, які зазвичай використовуються, такі, як SIFT, SURF, HOG.

SIFT – це метод, який визначає та описує ознаки на зображенні, він був розроблений так, що на нього не впливає обертання, масштаб та частково освітлення. Метод є повільним, але він надзвичайно точний. Але це не залишилось таким, адже появився алгоритм SURF – це швидша версія SIFT, яка зберігає інваріантність масштабу і повороту, зменшує кількість обчислень, з невеликими втратами точності [14].

Метод HOG відстежує випадки орієнтації градієнта в певних ділянках зображення. Після поділу зображення на комірки він нормалізує локальний

контраст у блоках, що перекриваються, і обчислює гістограму напрямків градієнта в кожній комірці (рис. 1.5.) [15]. Цей метод є дуже ефективним в медицині.

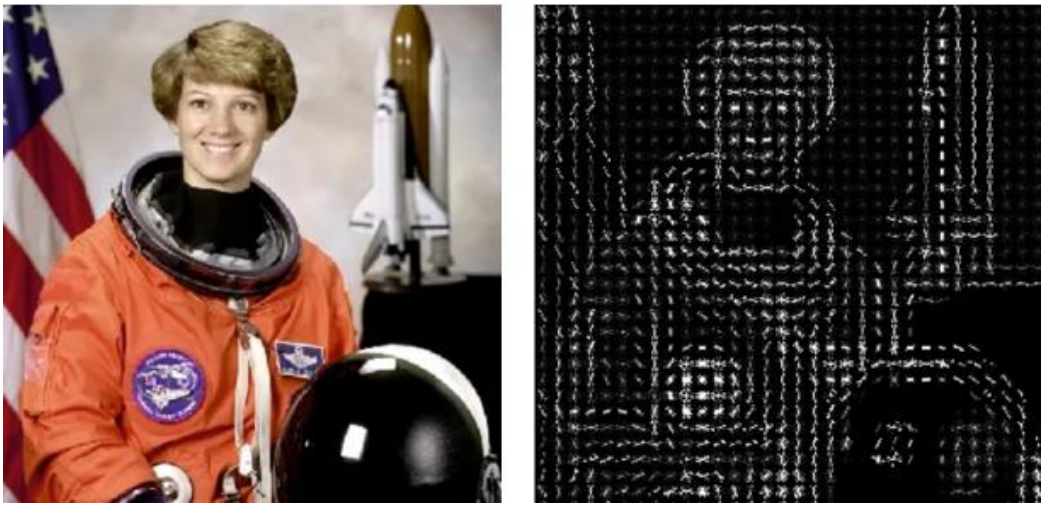

Рис. 1.5. Приклад виконання HOG.

Якраз через свою простоту та ефективність ці методи досі знаходять місце в розробці сучасного комп’ютерного зору, особливо для оптимізації, адже вони гарно працюють в умовах обмежених обчислювальних ресурсів, також для задач, які не вимагають високої складності.

Але ці методи також мають й недоліки, вони загнані в «алгоритмічні рамки», через цю відсутність адаптивності вони не здатні підлаштовуватись під різні специфічні вимоги завдання, для прикладу, як нерівномірне освітлення. Тому в більш складніших задачах використовують саме глибоке машинне навчання.

1.2.2. Вплив Deep Learning

З початку появи машинного навчання його старались застосовувати в сфері комп’ютерного зору й це спричинило велику революцію. Більшість сучасних рішень беруть за основу саме нейронні моделі, адже порівняно із минулими класичними методами вони є більш адаптивними, зручними та точними й показують себе краще в складних завданнях. Активне використання глибокого

навчання також зумовлене тим, що обчислювальні потужності сучасних комп’ютерів чи інших машин збільшились в багато разів [10].

Основним елементом глибокого навчання є самі нейронні мережі, вони імітують процес обробки інформації людським зором, сама нейронна мережа зазвичай складається із шарів.

Шари поділяються на три основні групи:

1. Вхідний шар.

У даному шарі необроблені дані передаються в модель.

2. Приховані шари.

У даних шарах якраз й відбуваються всі операції щодо перетворення вхідних даних, визначаються ознаки та закономірності.

3. Вихідний шар.

В останньому шарі й генерується кінцевий результат.

Ключова особливість нейронних мереж в тому, що вони навчаються. Процес навчання називається зворотним поширенням.

Зворотне поширення – це процес налаштування ваг нейронної мережі на основі коефіцієнта похибки, яка визначається в минулій ітерації, між прогнозованим і отриманим результатом. Цей процес повторюється поки мережа не досягне бажаної продуктивності [16].

Він має певні переваги, через які й отримав своє поширення:

1. Він швидкий і простий в програмуванні.
2. Гнучкість.
3. Не має параметрів для навчання.

Найбільшим недоліком даного методу є те, що він може бути чутливим до шуму та його ефективність й продуктивність залежить від вхідних даних.

В сучасних завданнях найчастіше використовують CNN, саме з винаходом їх стався значний прогрес в розвитку комп’ютерного зору.

CNN (Згорткова нейронна мережа) – це вид алгоритму глибокого навчання, який був спеціально розроблений для обробки великих вхідних даних, схожих на

сітку, такі як зображення (рис. 1.6.) [17].

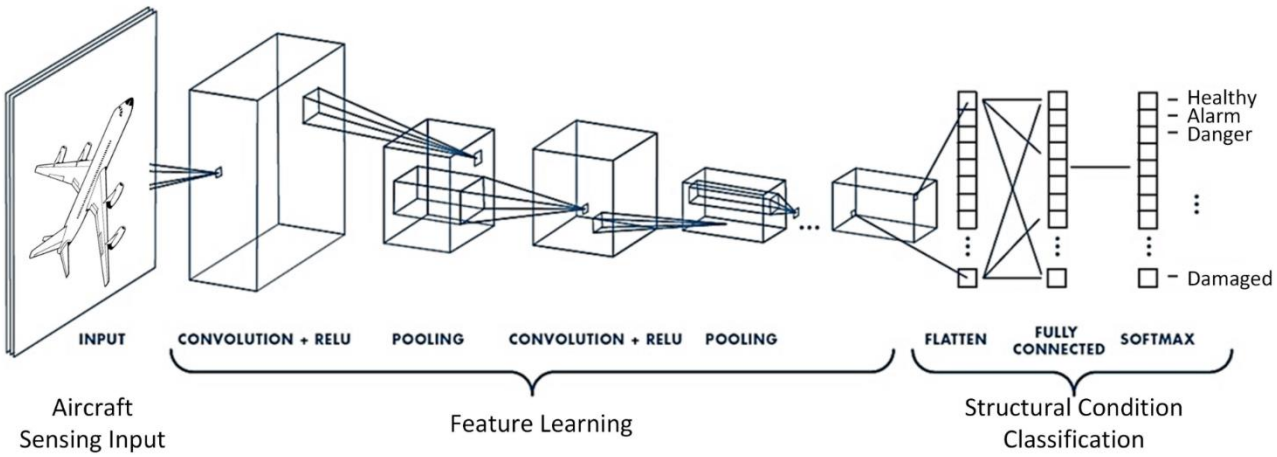

Рис. 1.6. Загальне зображення роботи CNN

CNN складається із декількох основних компонентів [10,17,18]:

1. Згорткові шари. Ці шари застосовують операції згортання за допомогою фільтрів для того, щоб ідентифікувати ознаки, такі як краї, текстури, форми.
2. Об’єднання шарів. Щоб зберегти важливу інформацію та знизити обчислювальні витрати, об’єднання шарів зменшують просторові розміри карти згорткових ознак.
3. Повністю пов’язані шари. Це завершальні шари CNN, після кількох згорткових і об’єднаних, дані шари ідентифікують об’єкти та роблять кінцеві прогнози, на основі минулого навчання.

Є кілька етапів навчання CNN: Спочатку мережа навчається на позначеному наборі даних. Мережа робить прогнози щодо виходу для певного введення під час навчання та порівнює їх із фактичною позначкою. Для кількісного визначення цього порівняння використовується функція втрат, наприклад середня квадратична помилка або категорійна крос-ентропія. Змінюючи вагові коефіцієнти та зміщення мережі за допомогою підходу до оптимізації, наприклад градієнтного спуску, мета полягає в тому, щоб мінімізувати втрати. CNN навчаються за допомогою базового підходу, який називається зворотним

поширенням. Це дозволяє мережі мінімізувати втрати шляхом оновлення своїх ваг і зміщень.

Саме CNN стали найкращим вибором для виконання задач комп'ютерного зору, адже вони точно виконують всі основні завдання. Але їх також не оминули недоліки.

Головним недоліком CNN є потреба в великій кількості навчальних позначених даних, а це несе за собою багато часу на отримання такої кількості даних та навчання, та великих витрат коштів. Також згорткові нейронні мережі є чутливими до змін масштабу, повороту, освітлення та інших аспектів вхідних даних. Ця чутливість впливає на продуктивність моделі, що вимагає використання додаткових методів для адаптації або ж розширення вхідних даних й додаткового навчання. Також навчання великих моделей вимагає значної кількості обчислювальних потужностей.

1.3. Основні проблеми використання нейронних моделей

Основна проблема, яка вже була згадана вище є проблема великого обсягу даних для навчання. Збір і позначення таких даних є дуже довгим й недешевим процесом. Для виконання складних завдань часто обсяг даних може дорівнювати декількох терабайт [18].

Існують великі набори даних, такі як ImageNet чи COCO. ImageNet – це велика БД візуальних даних, які призначені для використання в дослідженні комп'ютерного зору. На момент 2017 року в базі було більше 14 млн. посилань на вручну анотовані зображення. Ці дані вимагають багато місця для зберігання, а це ускладнює організацію локального або хмарного сховища, адже його потрібно з кожним разом все більше.

Також важливою проблемою є те, що для Deep Learning потрібне важке апаратне забезпечення, це призводить до великих фінансових витрат, й маленькі дослідницькі групи чи малий бізнес можуть мати проблем з апаратним

забезпеченням.

Також через велику кількість даних в таких наборах часто трапляються «шумні» дані, які містять помилкові мітки чи нерелевантну інформацію. Це знижує ефективність навчання та може призвести до зниження точності висновків моделі.

До вище згаданих проблем також варто врахувати анотацію даних, як вже казав в базі ImageNet містяться вручну анотовані дані, а це займає велику кількість часу та коштів, адже автоматизовані методи для даних задач часто не видають результати необхідної точності.

Анотація даних – це процес маркування даних за допомогою тексту чи інших інструментів, щоб визначити ознаки, які модель повинна навчитись визначати самостійно. За допомогою наборів таких даних нейронна мережа навчається певним завданням. Тут важливо, щоб всі маркування були точними і без помилок, адже якщо вони будуть ефективність моделі буде падати.

Існує кілька способів створення таких анотацій, як було вказано вище, то це може бути як й вручну, так й автоматично. Зазвичай автоматична анотація використовується для складних завдань, це спеціально попередньо навчені алгоритми, адже анотування таких речей може зайняти багато часу. Але навіть такий спосіб вимагає людської уваги, адже існує можливість неточних анотацій, а це може коштувати дуже дорого.

Зазвичай в великих компаніях анотацією займаються спеціально наймані для цього працівники, які отримують зарплатню, але для невеличких команд чи стартапу – це може бути проблемою.

Необхідно згадати про етичні проблеми використання таких даних, адже нерідко дані збираються через камери та інші пристрої, що порушує питання конфіденційності, адже використання зображень людей без їхньої згоди порушує норми та закони. Окрім того, такі дані можуть містити певні упередження щодо статі, раси чи соціального класу людини, а це призводить до моделей, які навчені видавати такі ж упереджені результати.

Логічним рішенням проблем великого набору даних є зменшення їхньої кількості, але як це зробити?

Одним із способом є визначення певної кількості «еталонних» даних, які повинні бути містити ключові приклади, такі скорочені набори збільшать швидкість навчання, але зменшують точність та ефективність. Щоб це виправити використовується аугментація даних.

Аугментація даних – це технологія, яка створює нові дані за рахунок невеликих перетворень вже наявних даних. Тобто, за допомогою поворотів, зміни масштабування, додавання шуму, зміни яскравості можна згенерувати нові дані, на яких зможе навчатись мережа (рис. 1.7.). А це в свою чергу буде економити витрати й час на анотування нових даних [19].

Також необхідно оптимізовувати алгоритми, які будуть потребувати менші набори даних, але видавати високу точність.

Рис. 1.7. Аугментація даних

Як зазначалось вище, є проблема із обчислювальними ресурсами. Використання складних моделей такі як ResNet, ViTs та інших, потребує багато ресурсів. Це включає велику кількість потужних графічних процесорів, CPU та спеціальне обладнання, що є недешевим задоволенням. Також для роботи із великим набором даних потрібно багато оперативної пам’яті, що також збільшує

витрати. Для прикладу, для навчання моделей іноді потрібно сотні потужних графічних процесорів на кілька тижнів роботи. Це значно обмежує доступність для стартапів чи малих компаній [18].

Така висока залежність від обчислювальних ресурсів обмежує їхнє використання на деяких кінцевих пристроях, такі як смартфони чи інші мобільні пристрої.

Одним із ефективних способів вирішення цих проблем є оптимізація моделей, саме цю проблему я буду описувати в даній роботі. Використання деяких методів, такі як квантування, прунінг дозволяє зменшувати розмір і вимоги до обчислювальних ресурсів без значної втрати точності [20-21].

Також активно почались розроблятись моделі, які спеціально розроблені для роботи в умовах обмежених ресурсів, не втрачаючи високу точність.

Коли модель навчається також варто пильнувати, щоб не виникла ще одна проблема, а саме проблема перенавчання [23].

Проблема з перенавчанням виникає, коли модель занадто добре вивчає набір навчальних даних, включаючи шум й неважливі деталі. Це призводить до того, що модель показує високу ефективність при роботі з цими даними, але з реальними – низьку ефективність та точність. Це виникає через те, що замість того, щоб шукати та вивчати ознаки, модель запам’ятовує точні закономірності, й результаті, після завершення навчання, така модель не працює точно із новими даними. Для прикладу, модель запам’ятовує точні кольори, контури, форми, які характерні лише для навчальних даних, а не опиратись на загальні ознаки (рис. 1.8.).

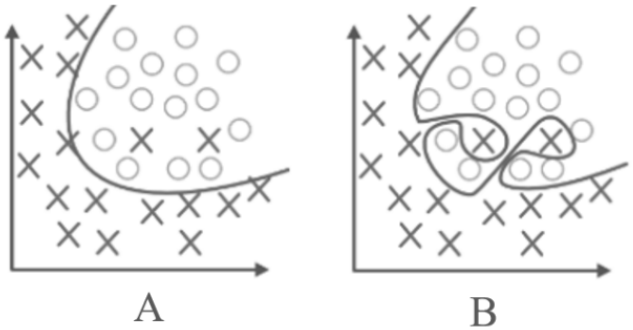

Рис. 1.8. Приклад перенавчання

На рисунку 1.8. можна помітити різницю між правильним навчанням під пунктом А та перенавчанням – В.

Причини виникнення перенавчання:

1. Основною причиною є неправильні дані для навчання, або мала кількість даних. Для правильного навчання нейронні мережі потребують велику кількість різноманітних даних.
2. Складна архітектура моделі також є причиною, адже часто великі моделі мають велику кількість параметрів, що призводить до того, що мережа починає «запам’ятовувати» неважливі деталі, замість якихось загальних ознак.
3. Відсутність регуляризації та аугментації. Ці речі допомагають уникнути перенавчання, адже допомагають більш урізноманітнювати навчальні дані. А регуляризація допомагає запобігати перенавчанню, вводячи обмеження, які не дають вивчати занадто специфічні ознаки.

Рішення цієї проблеми вже є давно відомі. Для початку варто збільшити набір навчальних даних, але тут я говорю не тільки новими даними, також можна урізноманітнити вже наявні шляхом аугментації. Для прикладу, якщо ми навчаємо мережу на виявлення автомобілів, то варто «годувати» її зображеннями автомобілів різних форм, кольорів, з різних ракурсів, масштабу й тд.

Також як й було вказано допомагають методи регуляризації, такі як L1 чи L2, ці методи гарно підходять, якщо ми маємо велику кількість ознак. Ці методи вводять «штраф» до функції машинного навчання, це надає перевагу важливим ознакам, й не дає моделі занадто точно «запам’ятовувати» навчальні дані.

Сюди також варто згадати ще один метод регуляризації, такий як Dropout, суть цього методу полягає в рандомному ігноруванні нейронів під час навчання, це примушує модель навчатись на різних даних [22-23].

Ще є така річ, як EarlyStopping (рання зупинка), вона допомагає нам зупинити перенавчання моделі вчасно. Ми перевіряємо її на тестовому дані, й якщо точність перестає збільшуватись, та починає лиш зменшуватись, це

означає, що мережа починає перенавчатись на навчальних даних й ми зупиняємо навчання, щоб уникнути більшого перенавчання. Для профілактики перенавчанню варто регулярно проводити оцінку моделі.

Як би дивно не звучало, але зменшення складності моделі допоможе стати їх більш точною, адже це допоможе уникати перенавчання, адже з більшою кількістю параметрів є більша можливість, що модель почне навчатись на неважливих даних. Саме зменшення цієї кількості допоможе уникати таких даних.

У висновку до цього, можу сказати, що незважаючи на велику кількість проблем дослідники активно шукають їхнє рішення, я загально згадав лиш основні, вирішення цих проблем робить штучний інтелект більш доступним.

1.4. Застосування комп’ютерного зору в різних сферах

Комп’ютерний зір ще від початку свого зародження почав активно використовуватись й з його розвитком він почав охоплювати все більше сфер, в сучасному світі він не є чимось дивним, кількість яких буде лиш збільшуватись, а саме ШІ дозволить прискорити цей процес.

В даному підрозділі опишу декілька поширених використання комп’ютерного зору.

Однією з таких сфер є автомобільна індустрія, комп’ютерний зір є ключовим для винайдення автопілоту в автомобілях, й зараз саме так він й використовується. Яскравим прикладом такого використання є компанія Tesla, вона використовує комп’ютерний зір для оцінки оточення за допомогою камер в своїх автопілотах. Їхні автомобілі здатні розпізнавати пішоходів, дорожні знаки та смуги руху й це допомагає уникати різних аварійних ситуацій, також дозволяє розвивати технологію автопілоту, але на даний момент це не є повністю автоматизовано, тому потребує «живого» водія, якщо автомобіль бачить реакції водія, він припиняє свій рух [5].

Щодо автомобільної індустрії, комп’ютерний зір також допомагає забезпечувати безпеку дорожнього руху, та відстежувати порушників. Тут варто згадати камери відстежування перевищування швидкості, які вимірюють швидкість учасників руху, й в випадку порушення вони автоматично виписують штраф, адже розпізнають номер автомобіля, також вони здатні відстежувати не тільки перевищення швидкості. Також такі системи здатні відстежувати автомобілі злочинців й повідомляти про їхню появу. Окрім такого використання, також можна відстежувати транспорт, який перевозить небезпечний вантаж, прокладаючи безпечніший шлях. Також ця система може використовуватись для керування світлофорами, відстежуючи кількість автомобілів та пішоходів й прораховуючи кращі дії, зменшуючи кількість заторів.

Не потрібно забувати, який вклад комп’ютерного зору в сферу медицини та охорони здоров’я. Як вже розказував вище, ШІ допомагає в діагностиці захворювань через аналіз знімків МРТ, рентгенівські знімки та КТ. Для прикладу, виявлення ракових пухлин на ранніх стадіях. Це здатне стати хорошим допоміжним інструментом для лікарів й дуже прискорити винесення діагнозу. Великою перевагою комп’ютера є те, що він здатен обробити величезну кількість інформації набагато швидше, ніж людина, але без людського втручання тут на даний момент ніяк [3].

В промисловості також починають активно використовувати комп’ютерний зір, адже це здатне зекономити витрати, а також пришвидшити виробництво. Комп’ютери здатні швидко виявляти дефекти при виробництві, відмінно від людської праці, яка є дорожчою. Варто врахувати те, що також така робота може виконуватись в особливо небезпечних умовах, в такому випадку робота комп’ютера буде набагато кращою. Також при виробництві він здатен в режимі реального часу знаходити дефекти та моментально на них реагувати, що тільки підвищує якість та продуктивність [4].

Мабуть кожен чув за магазини Amazon Go, в даному випадку через камери комп’ютер розпізнає, які товари клієнти беруть з полиць й автоматично

списуються кошти. Таке рішення зменшує витрати на працівників кас та є зручними для клієнтів, адже уникаються черги. Окрім цього в сфері торгівлі та маркетингу через камери комп'ютер може аналізувати емоції клієнтів для оцінки задоволення обслуговуванням.

Окрім безпеки дорожнього руху також комп'ютерний зір використовується й для інших систем безпеки. На даний момент системи розпізнавання обличчя є невід'ємною частиною потужних систем безпеки. Для прикладу, розпізнавання обличчя замість використання ключів чи карток для доступу в певні приміщення, або ж впізнавання злочинців й їхнє відстежування в режимі реального часу через аналіз відео і бази даних правоохоронних органів. Але це залишає відкритим питання конфіденційності.

Ще є багато сфер в яких використовується комп'ютерний зір, адже він активно розвивається. На даний момент одним із трендом в розвитку цієї сфери є генеративний штучний інтелект. Він вже спричинив багато шуму навколо себе, оскільки, в даному випадку, дозволяє комп'ютеру створювати реалістичні зображення чи відео, а це означає, що при досягненні високої точності, ми зможемо генерувати такий навчальний набір даних, й така нейронна модель зможе сама себе вдосконалювати. Також активним трендом є периферійні обчислення, адже це дозволить збільшити швидкість такої мережі, а також покращить конфіденційність даних, адже всі обчислення будуть проводитись на вашому пристрої. Також це допоможе працювати таким моделям без обов'язкового стабільного інтернет-з'єднання.

Важливими є дослідження в області AR (доповнена реальність) та VR (віртуальна реальність), адже за допомогою комп'ютерного зору можна буде покращувати такі системи, через розуміння реального оточення, наприклад, розпізнавання жестів, рухів, міміки без наявності багатьох маркерів, а лиш через одну камеру. Все це дозволить віртуальним об'єктам реалістично взаємодіяти із світом, що покращить досвід використання цих технологій в різних сферах, від сфери розваг до освіти [24].

До цього можна навести приклад можливого використання таких технологій в кіноіндустрії та індустрії відеоігор, адже, для прикладу, можна зчитувати міміку й рухи акторів для якихось сцен, без маркерів, що здешевить виробництво.

Також ведуться дослідження в можливості навчання без учителя, адже це дозволить нейронним моделям навчатись на неанотованих даних. Окрім цього всього починають створюватись мультимодальні нейронні моделі, колись кожна модель виконувала й навчалась для якогось конкретного завдання, а зараз починають появлятись моделі, які об’єднують в собі декілька, для прикладу можна згадати ChatGPT, який окрім генерації тексту, також може генерувати зображення, аналізувати фото й тд.

1.5. Висновок до розділу

У даному розділі було проведено аналіз предметної області комп’ютерного зору. Було описано історію розвитку цієї сфери, для розуміння як вона розвивалась, які були ключові етами, також розглянув класичні алгоритми обробки зображень й проаналізовано сучасні рішення, щоб зрозуміти принципи роботи даних методів й як вони використовуються в сучасних умовах, їхні переваги та недоліки.

Окрім того було описано основні проблеми та виклики, які зустрічаються в комп’ютерного зору й розглянуто їхні причини та можливі рішення. В кінці було також розглянуто сфери застосування технології в сучасних умовах та її майбутні тенденції розвитку.

РОЗДІЛ 2
МОДЕЛІ НЕЙРОННИХ МЕРЕЖ ДЛЯ ЗАДАЧ КОМП'ЮТЕРНОГО ЗОРУ

2.1. Згорткові нейронні мережі (CNN)

2.1.1. Основні принципи роботи згорткової нейронної мережі

У минулому розділі я вже згадував за згорткові нейронні мережі й було сказано, що вони стали дуже важливими для розвитку комп'ютерного зору, тут розберемось в них більш детальніше.

Основним елементом CNN є згорткові шари, як говорив вище, вони виконують операції згортки, щоб виявляти ознаки з вхідних даних, в нашому випадку зображення. Згортка – це операція додавання кожного компонента зображення до його сусідів, які зважені ядром, можна сказати, що спочатку відбувається транспонування матриці ядра, з наступним множенням та додаваням (рис. 2.1.) [17-18].

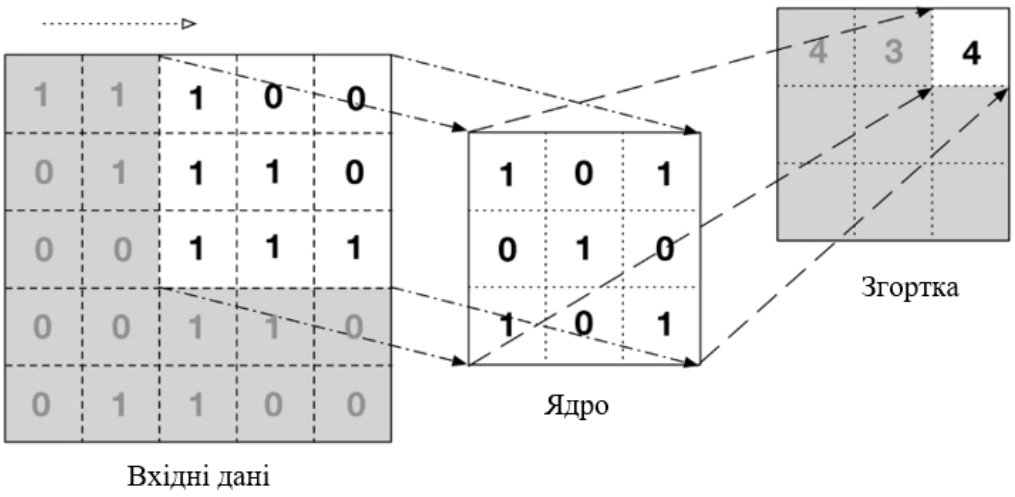

Рис. 2.1. Згортка в CNN

Взагальному описати формулу згортки можна як

$$(I * K)(x, y) = \sum_m \sum_n I(m, n) K(x - m, y - n) \qquad (2.1)$$

де I – це вхідний сигнал;

K – це ядро.

Ядром в даному випадку називається невеличка матриця чисел, наприклад 3×3, ці матриці ковзають по зображенню виконуючи операції згортки, витягуючи певні ознаки з зображення, в залежності від ознак фільтр (ядро) змінюється.

В результаті такої операції отримуємо карту ознак, на кожну карту ознак є своє ядро. Таких згорткових шарів може бути декілька, залежно, які ознаки нам потрібно витягнути, ну й наскільки складне завдання.

Варто розуміти, що в CNN існує ієрархія ознак, на самому початку витягуються прості ознаки, такі як краї, контури. На наступних йдуть вже складніші ознаки: форми, текстури. А на кінцевих шарах вже різні семантичні ознаки: об’єкти, категорії й тд.

Існують ще деякі параметри, які необхідні для коректної роботи нейромережі, такими параметрами є крок і zero-padding, ці параметри впливають на результат згортки.

Крок – це кількість пікселів, на яку ядро переміщується за кожну операцію, тут все просто, збільшення кроку буде більш корисним для того, щоб визначити якійсь глобальні об’єкти, а ось його зменшення дозволить краще визначати якійсь дрібні деталі. Це все впливає на швидкість роботи, адже з більшим кроком модель буде працювати швидше, але вона втратить свою ефективність, з меншим кроком все навпаки.

Zero-padding – це параметр, який відповідає за те, скільки пікселів буде додано до зображення відступом. Тут варто пояснити детальніше, й згадати те, що ядро це матриця певного розміру, яка рухається по зображенню, але якщо уявити цей процес, то можна зрозуміти, що деякі пікселі ядро буде проходити більше разів, ніж інші, ті, що розташовані по краях. Щоб уникнути цього до вхідного зображення додається «рамка» із нульових пікселів, число цього відступу якраз й є цим параметром. Це зазвичай використовується, коли краї

можуть містити якусь важливу інформацію. До речі, на практиці не обов'язково завжди відступ має бути однаковим із усіх сторін.

З його допомогою збільшується площа обробки ядром, а це забезпечує більшу точність.

Це все впливає на кінцевий розмір карти ознак, цей розмір можна визначити за формулою

$$W_{out} = \frac{W_{in} - F + 2P}{S} + 1 \quad (2.2)$$

де W_{in} – це розмір вхідного зображення;

F – це розмір ядра, він множиться на 2, адже враховує додавання відступу на праву та ліву сторони вводу;

S – крок.

Вкінці формули додається одиниця, щоб врахувати початкове розміщення ядра.

Для зменшення обчислювальної складності застосовують ще один важливий елемент під назвою шари об'єднання. Зниження складності відбувається шляхом зменшення розміру карти ознак, адже зменшується кількість параметрів і обчислень, також це є профілактикою перенавчанню нейронної мережі. Якщо ж ми не використовуємо об'єднання, то вихід буде мати такий же розмір, як й вхід [18].

Існує два способи:

1. Max Pooling: виконується шляхом зберігання максимального значення з карт ознак.
2. Average Pooling: Це збереження середнього значення.

Серед них на практиці кращим виступає перший, тому найчастіше використовується він.

Важливим елементом нейронної мережі є функція активації, адже без неї нейронна мережа буде лише лінійною моделлю, вона дозволяє моделювати

складні залежності між вхідними й вихідними даними. Саме ця функція вирішує чи має нейрон бути активованим.

Є декілька поширених функцій, в CNN та й не тільки, використовується саме ReLu. Через свою ефективність та вона найчастіше використовується в глибокому навчанні.

Сама формула виглядає просто

$$f(x) = \max(0, x) \tag{2.3}$$

Тобто, ця функція замінює всі мінусові значення на 0, а плюсові залишає незмінними (рис. 2.2.). Якраз через те, що вона утримує всі від'ємні нейрони неактивованими, вона швидше навчається.

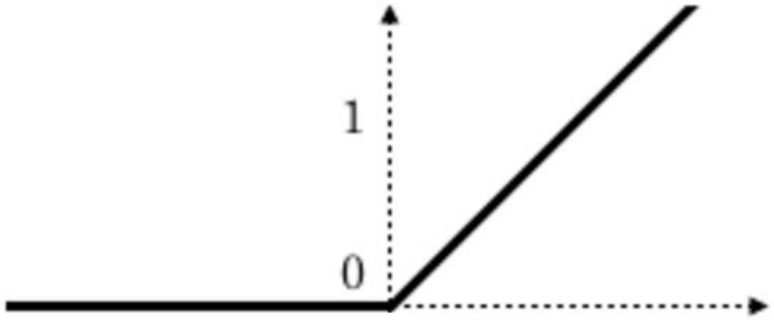

Рис. 2.2. Графік функції ReLu

Існує й покращена версія ReLu, під назвою Leaky ReLu. В даному випадку при від'ємному значенні повертається ненульовий результат, зазвичай це 0.01*x. Дана функція збільшує діапазон значень, адже ми не відкидуємо від'ємні значення [25].

Окрім цього існує багато різних функцій активації, такі як Sigmoid, Tanh, але саме при глибокому навчанні вони не отримали великого поширення. Sigmoid не часто використовується, через проблему градієнтного затухання, адже присвоює значення для виходу від 0 до 1, але вона все-таки використовується в повністю зв'язаному шарі для бінарної класифікації. Друга

схожа на першу, але здатен дати кращу ефективність, адже працює в діапазоні від -1 до 1.

Після цього всього переходимо до останнього елементу, це повністю зв’язаний шар, якраз в цьому шарі мережа приймає рішення на основі ознак з попередніх шарів, з’єднуючи різні нейрони в даному шарі мережа має можливість приймати рішення на основі комбінації ознак, а не лиш по окремих ознаках. Окрім цього цей шар може містити функцію активації, в залежності від завдання. Але варто враховувати те, що даний шар працює лише з одновимірними масивами, тобто перед цим нам потрібно перетворити матрицю в нього. Також для регуляризації моделі в даному шарі можуть використовувати методи, які були згадані в попередньому розділі.

2.1.2. AlexNet

AlexNet – стала революційною нейронною моделлю для класифікації зображень, саме через те, що вона перша почала використовувати для цього глибокі нейронні мережі, адже тоді навчання моделі із величезною кількістю параметрів вважалось неможливим. Через це саме вона задала майбутні «тренди» в розробці моделей для комп’ютерного зору, й повернула інтерес до глибокого навчання [26].

Модель в 2012 році перемогла конкурс ImageNet з показником помилки 15,3%, це стало проривом, адже до цього найкращий показник був 26,2%, це й вернуло популярність глибоким нейронним мережам, й буквально через декілька років вже були розроблені нейронні моделі, які перевершили людину в точності.

Ця мережа внесла дуже багато нових інновацій, які постараємось детальніше розглянути.

AlexNet стала першою архітектурою, яка використовувала графічний процесор для збільшення продуктивності навчання, сама оригінальна модель, через обмеження графічного процесора того часу, була розділена на 2 процесора.

Сама архітектура моделі є простою, якщо порівнювати із сучасними, але

тоді вона була проривом, саме модель складалась із 5 згорткових шарів, 3 шари об’єднання (Max-Pooling), 2 шари нормалізації, 2 повністю об’єднані шари та 1 шар SoftMax (рис. 2.3.), за ці шари було детально розказано раніше.

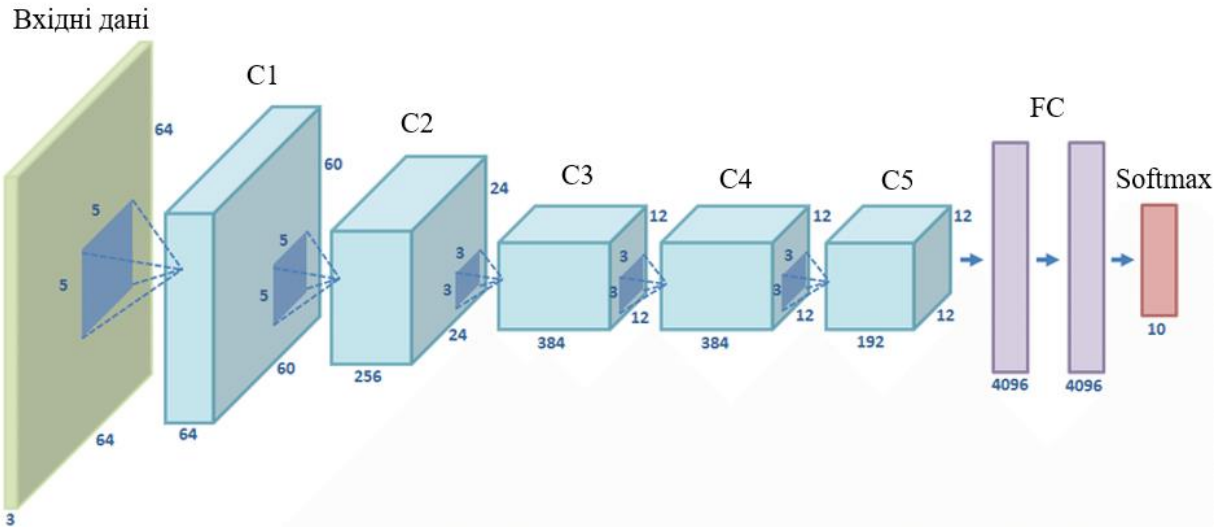

Рис. 2.3. Архітектура AlexNet

Однією із таких особливостей є використання ReLu для функції активації, до цього використовувались сигмоїда і тан, але як було сказано раніше, вони гальмують процес навчання.

Також AlexNet використали вікна об’єднання 3 на 3 з кроком 2, на відміну від звичайних шарів Max-Pooling, коли таке вікно переміщується, то воно перекривається із попереднім вікном, це дозволило покращити точність моделі.

Також для уникнення проблеми перенавчання використовували аугментацію, вона допомогла збільшити загальну кількість даних.

В даній нейронній моделі використовували аугментацію за допомогою дзеркального відображення, що вже дозволить подвоїти кількість даних, обертання та шляхом рандомного обрізання зображень, що дозволило дуже сильно збільшити кількість навчальних даних. Що привело, до збільшення точності.

Для уникнення цієї ж проблеми використовувався Dropout, нейрон, який відкидався не робив ніякого внеску ні в прямому поширенні, ні в зворотному, це

збільшує час навчання, але в результаті покращує його. Показник відкидання в мережі складав 50%, тобто половина нейронів відкидалась у кожному шарі випадковим чином.

Але незважаючи на всі ці переваги даної мережі, вона не залишилась без недоліків, окрім недоліків CNN її недоліком є те, що порівняно із сучасними нейронними моделями вона не є достатньо глибокою, отже вона не підходить до вивчення складніших ознак зображень, також їй необхідно більше часу для досягнення необхідної точності, порівняно із майбутніми.

2.1.3. VGGNet

Як було сказано, після представлення AlexNet всі знову зацікавились глибокими нейронними мережами й через 2 роки була представлена нова модель VGGNet, яка враховувала недоліки AlexNet, і досягла точності 92,7%. Також мережа збільшила свою глибину, якщо попередня мала 8 шарів, то VGG – 16 шарів, вона називається VGG16, існує ще VGG19 із 19 шарами відповідно [26].

Взагальному архітектура складається із 5 блоків, в кожному із них по 2-3 згорткові шари, після кожного блоку йде шар об’єднання (Max-Pooling), і 3 повністю з’єднані шари. В VGG19 додаються три додаткові згорткові шари (рис. 2.4.).

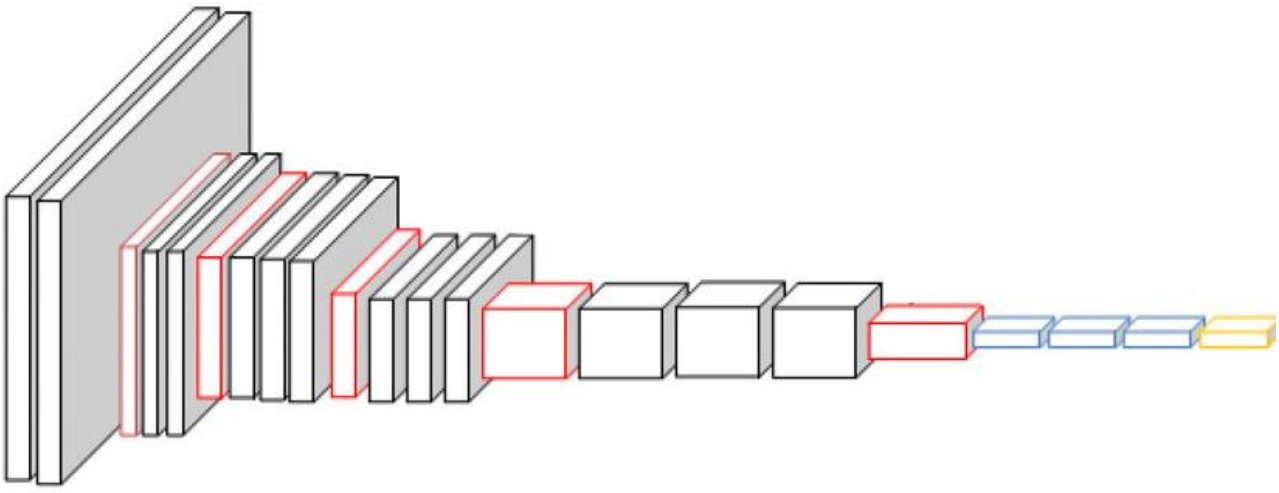

Рис. 2.4. Архітектура VGG

На рис. 2.4. чорним контуром позначено згорткові шари, червоним – шари об’єднання, синій – повністю з’єднані шари, а жовтий – Softmax. Сама мережа приймає на вхід зображення 244×244.

Однією з особливостей цієї мережі є те, що використовується мале ядро 3×3 у всіх згорткових шарах із кроком 1 і також відступ 1, це використовується для збереження просторову роздільну здатність, це також дозволило збільшити точність. Окрім цього використання більш глибокої архітектури дало можливість визначати більш складні ознаки. Також ще більш вдосконалилась аугментація даних. На відміну від попередньої моделі зазвичай не використовується локальна нормалізація, адже вона збільшує використання ресурсів, а на загальну точність не впливає.

Простота архітектури й ефективність вплинула на популярність цієї моделі, адже навіть сьогодні вона залишається однією із найчастіше використовуваних моделей, також вона вплинула на тенденції розробки інших моделей.

В іншому вона має багато спільного із AlexNet, тільки вирішила деякі проблеми цієї моделі. Але незважаючи на те, вона також має недоліки.

Через те, що використовується мале ядро й через свою глибину, ця модель вимагає великої обчислювальної потужності, що не підходить для використання в умовах обмеженої потужності, також навчання мережі займає дуже багато часу. Недоліком є велика кількість параметрів, а саме 138 мільйонів, через це виникає проблема вибухових градієнтів – це коли в процесі зворотного поширення градієнти стають все більшими, це приводить до великих оновлень ваг, що заважає правильно навчатись мережі.

2.1.4. ResNet

Після створення AlexNet для зменшення відсотку помилок наступні моделі почали використовувати все більше шарів. Але суть в тому, що це працює лише для невеликої кількості шарів, а далі збільшуючи кількість шарів навпаки збільшується відсоток помилки. Це виникає через те, що градієнти функції

втрати стають дуже малими по відношенню до ваги, або навпаки градієнти можуть стати дуже великими, що призводить до проблем під час навчання моделі.

Дослідники виявили, що в згорткової нейронної мережі є граничне значення шарів, для прикладу мережа із 20 шарами працює точніше, ніж із 56, дехто думав, що це все проблема перенавчання, але вона себе показувала погано, що на навчальних даних, що на реальних.

Щоб уникнути усіх проблем зв'язаних із цим розробники ResNet впровадили таку річ, як резидуальний (залишковий) блок [26].

Звичайний шар спочатку приймає вхідні дані, обробляє їх й передає результат далі. А при використанні залишкового блоку спочатку все як і завжди, але далі до оброблених даних додається їхній оригінал, додавання відбувається перед передачею результату на наступні шари, утворюючи залишкове відображення, це дає можливість мережі відповідати залишковому зображенню, а не в тому, щоб вивчати основне відображення. Skip connection обходить деякі рівні між активаціями та наступними шарами, що покращує оптимізацію мережі.

Наприклад його формула

$$y = F(x) + x \tag{2.4}$$

де x – це вхідні дані;

$F(x)$ – оброблені дані;

y – дані, які передаються далі.

Значним плюсом такої архітектури є те, що якщо якийсь шар погіршує продуктивність, то він буде пропущений. Тому в результаті виникає можливість створювати надзвичайно глибокі мережі, без попередніх проблем з градієнтами, наприклад була представлена модель на 152 шари.

Інновації, які були введені в цю модель допомогли перевершити попередні модель із відсотком помилки 3,57% в 2015.

ResNet зробили революцію в створенні надглибоких нейронних мереж, вирішивши проблеми попередніх мереж, але також містить й недоліки.

Основним недоліком, як й в попередніх мережах, є необхідність в великих обчислювальних потужностей й довгий час навчання. Наступним недоліком є необхідність в великому наборі навчальних даних, адже при використанні недостатнього об’єму даних виникає помилка з перенавчанням моделі.

2.1.5. YOLO

Модель, яка стала популярною через свою швидкість та точність, вона стала проривом в завданні виявлення об’єктів в реальному часі, й навіть зараз залишається однією з найбільш використовуваних моделей. В загальному за весь час свого існування вона оновлювалась декілька разів, покращуючи точність та додаючи різні функції.

Як говорить назва You Only Look Once модель аналізує зображення лиш один раз, а не по декілька разів чи частинами, як роблять інші моделі. Модель прогнозує координати коробок об’єктів та класи одночасно [27].

При виконанні завдання зображення розбивається на сітку, в якій кожна клітинка відповідає за прогнозування об’єктів. Кожна клітинка сітки прогнозує фіксовану кількість коробок й для кожної коробки декілька параметрів – координати центру коробки відносно клітинки, ширина та висота коробки та оцінку впевненості – це показник, наскільки модель впевнена, що коробка містить об’єкт. Також для коробки прогнозується клас об’єкта. Починаючи із другої версії, модель використовує якірні коробки – це задані форми й пропорції коробок, які допомогли краще працювати.

На виході ми отримуємо результат, що на один об’єкт може бути декілька різних обмежувальних коробок, які перекривають одне одного. Тут використовується метом NMS (не максимальне придушення) для того, щоб прибрати всі зайві коробки й залишити лиш ту, яка має найбільшу ймовірність.

Взагальному архітектура моделі мінялась із різними версіями, коротко

можна проаналізувати різні версії.

Перша батьківська версія вражала своєю швидкість, але не була дуже точною, тому в другій версії була покращена точність та додано якірні коробки.

YOLOv3 ще більше покращила продуктивність та точність виявлення об’єктів, модель більш точніше почала працювати із об’єктами в різних масштабах та роздільній здатності.

Наступні версії вносили зміни, щоб покращити точність та оптимізацію моделі, щоб видавати гарні результати навіть на пристроях з низьким рівнем ресурсів.

2.1.6. Порівняння популярних моделей CNN

Проаналізувавши моделі згорткових нейронних мереж можна скласти порівняльну характеристику (табл. 2.1).

Таблиця 2.1

Порівняльна характеристика моделей CNN

Параметр	AlexNet	VGGNet	ResNet	YOLO
Рік випуску	2012	2014	2015	2016
Точність на ImageNet	84,7%	92,7%	93,6%	-
Глибина (кількість шарів)	8	16-19	50-152 і більше	24-53
Кількість параметрів	~60 млн	~138 млн	~25,5 млн	~62 млн
Розмір вхідних даних	227 × 227	224 × 224	224 × 224	Залежно від версії
Вимоги обчислювальних ресурсів	Нижче середнього	Високі	Високі, але ефективніші	Залежно від версії, середні

Продовження таблиці 2.1.

Розмір ядра	11×11, 5×5, 3×3	3×3	3×3, 1×1	3×3, 1×1
Основні недоліки	Великі фільтри	Велика кількість параметрів	Складність навчання дуже глибоких мереж	Обмежена точність для малих об’єктів

Всі моделі активно використовуються в сучасних умовах, в залежності від завдання. Після проведення порівняльного аналізу можна зробити висновок, що для завдань де є невеликий набір даних й обмежені ресурси краще підходить AlexNet. Якщо наші ресурси не є обмеженими й нам потрібна висока точність, то тут підходить VGGNet. ResNet використовується, коли нам необхідно визначати дуже дрібні ознаки зображення та велика кількість навчальних даних, адже вона є ефективною для дуже глибоких мереж, адже вона здатна забезпечити стабільне навчання й поруч із наявними в неї найбільша точність.

Після аналізу даних популярних моделей по завданнях їх можна розділити так:

У випадку, якщо нашим ключовим завданням є виявлення об’єктів на зображеннях найкраще підійде YOLO, адже вона демонструє високу швидкість та точність, але вона не підійде, якщо необхідно визначати дрібні деталі, як вище сказано в такому випадку ефективнішою буде ResNet. YOLO теж підійде для обробки відео та відстеження об’єктів в реальному часі.

При завданні з сегментації ефективнішою буде ResNet через її особливості. Це також підходить для використання в завданні класифікації.

В висновку можна сказати, що в сучасних умовах більш універсальною буде ResNet через свою гнучкість, але це за умови наявності великих обчислювальних потужностей, але якщо нашою ціллю є детекція об’єктів, то в більшості сучасних завдань використовувати ефективніше YOLO.

2.2. Трансформери в комп’ютерному зорі

2.2.1. Робота трансформерів.

Після того, як трансформери почали показувати свою продуктивність в моделях з обробки та генерації природної мови, почали досліджувати їхню ефективність для задач комп’ютерного зору. Й це стало одним із радикальних проривів.

Трансформери – модель глибокого навчання, яка спочатку була розроблена для NPL, що використовує механізм самоуваги для зважування кожного компонента вхідних даних, й замість того, щоб обробляти всі дані покроково, він дивиться на них й визначає, які частини важливіші за інші, для прикладу із зображенням він визначає, які частини зображення впливають на розпізнавання об’єкта [28].

На початку роботи трансформера зображення ділиться на невеликі квадратики фіксованого розміру, які називають патчами, і кожен із цих квадратиків обробляється, як вектор чисел, які описують інформацію патча, також зберігається інформація про позицію кожного патча, адже модель не знає про розміщення кожного квадратика.

Після цього трансформер аналізує, які патчі впливають один на одного, тобто якщо ми маємо патч із автомобільною фарою, то значить, що швидше за все, інші мають описувати автомобіль, це й є механізм уваги. Це робиться шляхом визначення значення уваги, вона обчислюється за формулою

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.5}$$

де Q – це вектор, що представляє конкретний елемент входу;

K – це вектор, що описує наскільки кожен елемент є корисним;

V – вектор, який містить значення, пов’язану з елементами входу. Ці вектори формуються за допомогою навчальних матриць;

QK^T- тут обчислюється подібність між Q і K, для нормалізації, щоб уникнути занадто великих ваг;

d_k – це розмірність вектора K.

Далі зважується з V, щоб зрозуміти важливість патчу.

Після кількох шарів, вся інформація збирається модель видає результат.

Це якщо по простому описати, як працюють трансформери, те, що ми працюємо із зображеннями, ніби із текстом, ділячи їх на частини, то це означає, що така мережа простіше розробляється для роботи із текстом та зображеннями одночасно, це є однією з переваг трансформерів.

Це все стосується ViT, але існують інші архітектури трансформерів, іншою популярною є Swin-T.

Ключовою інновацією в Swin-T було використання вікна ковзання, тобто замість того, щоб використовувати фіксовану сітку для поділу зображення, він використовує вікна з рандомними зсувами, які перекриваються, для обчислення локального значення уваги [29].

Замість того, щоб аналізувати всі патчі, він працює лише з цими вікнами, там обчислюється їхня увага. Це знижує обчислювальну складність, адже увага обмежується тільки невеликою кількістю патчів (рис. 2.5.).

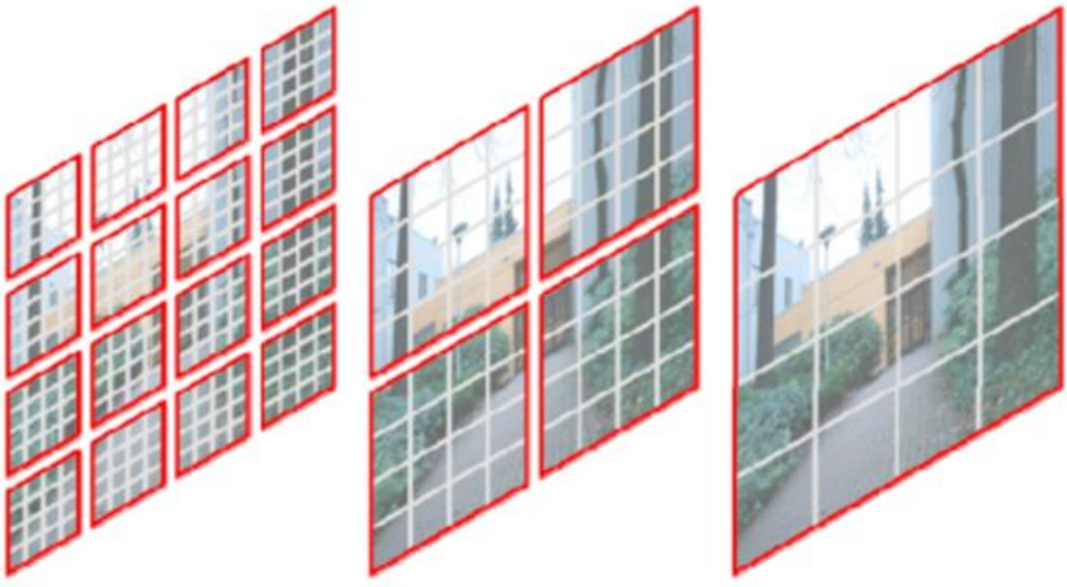

Рис. 2.5. Зміна вікон із шарами

Так модель проходиться по всьому зображенню, але, щоб врахувати зв’язки патчів, які в різних вікнах використовується це ковзання. Після кожного шару вікна трішки ковзають, й обчислюється увага. В результаті ми обробляємо не тільки локально, а й все зображення.

З кожним шаром моделі розмір вікна збільшується, а ось кількість патчів зменшується, адже вони разом групуються. Приклад виконання можна побачити на рис. 2.5.

Ось так спочатку від аналізу деталей ми переходимо до аналізу цілого зображення, тому мережа показує чудові результати для класифікації та сегментації.

2.2.2. Переваги та недоліки трансформерів.

На даний момент використання трансформерів в сфері комп’ютерного зору є дуже активним, деякі дослідники говорять, що вони вже починають домінувати на ринку, або їхня популярність на рівні із CNN. Їхнє використання стало популярним саме через їхні переваги.

Основною перевагою використання трансформерів є здатність до глобальної уваги. За допомогою механізму уваги модель може визначати зв’язки між різними частинами зображення, незалежно від їхньої відстані одне від одного. А це допомагає зрозуміти контекст й структуру зображення, така можливість є корисною при виконанні складних завдань, адже ми можемо розуміти загальну картину, а не локальні ознаки [28].

Ще дуже значною перевагою трансформерів є їхня гнучкість та універсальність. Вони є не залежними від попередньо визначених фільтрів, це дає змогу легко їх налаштовувати на виконання різних задач, без великих змін в структурі. Також трансформери легко масштабуються для виконання багатьох завдань й для роботи із різними типами даних, саме це є однією із ключових особливостей, через які їх більше використають для певних областей ШІ. Такі мережі здатні працювати із зображеннями, текстом, аудіо, відео, це допомагає

ефективніше створювати такі мережі, як генерація зображень по текстовому промту.

Окрім того, продуктивність таких моделей покращується із збільшенням розмірів моделі та об’єму навчальних даних, тому в завданнях, де присутня необхідність використовувати велику кількість даних вони показують точні результати, які не рідко перевершують всі традиційні моделі.

Але варто врахувати те, що незважаючи на всі їхні переваги, вони також мають недоліки.

Звісно, що зразу на думку спадає проблема із використанням обчислювальних ресурсів, й це так. Для ефективного навчання таких моделей потрібні потужні графічні процесори, що вимагає великої кількості коштів. Сам механізм уваги має квадратичну складність відносно розміру вектора. Це означає, що із зростанням роздільної здатності зображення, кількості патчів, збільшення необхідних обчислювальних ресурсів збільшується експоненціально. А це робить складним використання таких моделей для роботи із великими зображеннями або в режимі реального часу, особливо для пристроїв із дуже обмеженими ресурсами.

Недоліком трансформерів є й складність навчання та складність оптимізації, адже вони мають велику кількість різних гіперпараметрів. Також варто врахувати необхідність в великій кількості даних, теж вони складно інтерпретуються, тому важко зрозуміти, чому мережа зробила такий прогноз.

Щоб вирішити деякі недоліки й покращити ефективність трансформерів був розроблений Swin-T.

Для зменшення обчислювальної складності почались використовуватись вікна, детальну їхню роботу описав вище. Це рішення дозволило зменшити обчислювальну складність від квадратичної до лінійної відносно розміру зображення, а це робить ефективним використання Swin-T для зображень із високою роздільною здатністю.

Використання вікон також дозволило краще визначати різні локальні

ознаки, порівняно із ViT.

Окрім цього модель здатна більш повно розуміти загальну картину. Всі її особливості дозволили показати свою ефективність та продуктивність, досягаючи дуже точних результатів.

2.2.3. Аналіз гібридних підходів.

З розвитком моделей для задач CV, для забезпечення більшої ефективності почались дослідження з створення гібридних моделей, а саме об’єднання різних архітектур, які зможуть доповнити одне одного своїми сильними сторонами.

Сильною стороною CNN є їхня здатність визначати дрібні локальні ознаки та їхня оптимізація, а ось недоліком – глобальний контекст, а перевагою трансформерів є якраз здатність розпізнавати його. Об’єднання CNN і трансформерів дозволить використовувати кращі аспекти з кожної архітектури.

Однією з таких моделей є Convolutional Vision Transformers (CvT).

При роботі цієї моделі зображення спочатку обробляється згортковими шарами, які дозволяють ефективно працювати із великою кількістю дрібних деталей, що забезпечує більшу точність. Після проходження через згорткові шари, зображення перетворюється патчі, ці патчі формують вектор, який передається в трансформер, який відповідає за обробку складних зв’язків між різними елементами зображення [30].

Використання такого підходу дозволило досягти вищої точності, ефективніше працювати із великими зображеннями та складішими завданнями, окрім того забезпечує кращу гнучкість налаштування під конкретні завдання.

Ще яскравим прикладом використання гібридних підходів є Swin-T, який описав раніше, він взяв від CNN ієрархічну структуру.

Інноваційною моделлю для розпізнавання об’єктів на зображеннях є Detection Transformer (DETR), модель замінює традиційні методи більш простим підходом.

Спочатку зображення проходить звичайну обробку згортковими шарами

ResNet, вона витягує важливі ознаки із зображення. Далі до кожних ознак додається інформація про їхнє розташування, перед передачею даних дальше, адже для трансформери не мають вбудованого поняття порядку [31].

Трансформер одночасно аналізує всі частини зображення, визначає зв'язки між різними частинами зображення, й визначає до якого класу належить кожен об'єкт, чи він є фоном. А далі формується всім знайома рамка.

Основною перевагою цієї моделі є її проста архітектура, але вона потребує довшого навчання, ніж традиційні методи.

Взагальному гібридні підходи страждають від схожих проблем, основною проблемою є необхідність великих обчислювальних ресурсів, потреба в великій кількості навчальних даних. Також вони не є необхідними для багатьох задач, адже будуть складними, в таких задачах більш продуктивними будуть CNN.

2.3. Порівняльна характеристика CNN та трансформерів

Після дослідження цих популярних підходів можна скласти порівняльну характеристику (табл. 2.2).

Таблиця 2.2

Порівняння CNN та трансформерів

Характеристика	CNN (Convolutional Neural Networks)	Трансформери (Transformers)
Рік появи	2014	2017
Обчислювальна складність	$O(N^2)$	$O(N^2 d)$
Тип даних	Працюють найкраще із візуальними даними	Працюють із різними типами даних
Вивчення ознак	Ефективно визначають локальні ознаки	Ефективно визначають глобальний контекст

Продовження таблиці 2.2

Потреба в кількості даних	Залежно від складності завдання, але порівняно менша	Висока, адже краще працюють з великою кількістю даних
Масштабованість	Обмежена, важка	Висока, легша
Обчислювальні ресурси	Менші вимоги	Високі вимоги
Адаптація	Середня	Висока
Точність ImageNet	80-90%	90-95%
Робота з шумом	Ефективна	Менш ефективна
Гнучкість	Середня	Висока

Ми можемо помітити, що різниця між ними є досить помітною, й в дечому трансформери випереджають згорткові мережі, але все залежить від типу й складності завдання.

Завдання розпізнаванння об’єктів краще виконують CNN, якраз через їхню особливість краще розпізнавати дрібні деталі, вони можуть швидко й точно визначити об’єкти різного розміру на зображенні. Також ефективними будуть й гібридні підходи, але вони мають більшу складність й потребу в ресурсах.

Через цю саму особливість вони підходять для виконання завдання з розпізнавання обличчя, адже краще вловлюють особливості, такі як ніч чи структуру очей.

Для завдань в сфері медицини також ефективніше використовувати саме згорткові, адже виконання таких завдань часто потребує точного аналізу дрібних деталей, щоб виявити різні аномалії. Але якщо потрібно об’єднати різні завдання, тоді гібридний підхід.

Для виконання завдань в реальному часі трансформери можуть бути трішки повільними.

Щодо завдання генерації зображень краще себе показують саме трансформери, адже вони краще аналізують глобальні ознаки для створення реалістичних зображень. Та й для інших саме мультимодальних завдань кращим підходом будуть вони, адже їхня архітектура дає можливість працювати із різними типами даних, без великих складних змін, краще адаптується й більш гнучка.

Щодо виконання завдань класифікації зображень, то обидві архітектури показують себе гарно, але тут все залежить від їхніх особливостей. CNN – стандарт класифікації, через локальність, та їхня архітектура дозволяє працювати добре навіть на невеликих наборах даних. Але трансформери показують кращі результати на складних задачах чи з великими наборами даних. Тому в випадку невеликих наборів даних чи малих обчилювальних ресурсів краще використовувати CNN, а для великих наборів – трансформери.

Така сама ситуація при завданні сегментації зображень. Також в цьому завданні ефективно працюють гібридні підходи, адже в такому випадку точно виявляються локальні та глобальні особливості.

Після проведення аналізу можна зробити висновок, що тут все залежить від конкретного завдання та можливостей техніки. У випадку, якщо нам необхідно працювати із невеликими наборами даних чи існує потреба в аналізі дрібних локальних деталей, краще підходять CNN, особливо якщо в нас система із малими обчислювальними потужностями.

Трансформери будуть більш ефективними та точними для задач із визначення глобальних зв’язків. Якщо ж нам необхідно враховувати все, то краще працюватимуть гібридні підходи, але вимагатимуть великих обчислювальних ресурсів.

2.4. Висновок до розділу

У даному розділі було розглянуто та досліджено архітектури для виконання

завдань CV та було проведено порівняльний аналіз, який показав для яких завдань вони будуть ефективними.

Було розглянуто різні види CNN на прикладі AlexNet, VGGNet, ResNet та YOLO, проведено опис їхньої роботи, щоб зрозуміти їхні відмінності, визначити переваги та недоліки. Це дало можливість зрозуміти, чому вони досі залишаються домінуючими при виконанні деяких завдань. Також була складена порівняльна характеристика даних моделей.

Також проведено огляд роботи трансформерів, проаналізовано їхні особливості й відмінності від згорткових нейронних мереж на прикладі ViT. Окрім того було описано їхні переваги та недоліки. Розглянув поняття гібридних підходів на прикладі декількох моделей, їхній принцип роботи.

Розгляд переваг та недоліків допомагає визначити оптимальну модель для виконання потрібного завдання.

РОЗДІЛ 3
МЕТОДИ ТА АЛГОРИТМИ ОПТИМІЗАЦІЇ НЕЙРОННИХ МЕРЕЖ

3.1. Вибір нейронної моделі для оптимізації

Для виконання даного завдання я вибрав модель середньої складності, а саме ResNet18. Ця модель досі залишається дуже популярною через свою продуктивність та відносно невелику складність.

Модель легко підлаштовується під будь-яку задачу й розмір вхідних даних, тому вона є хорошим вибором для дослідження методів оптимізації.

Властивості моделі, яку буду використовувати подані в таблиці 3.1.

Таблиця 3.1.

Властивості нейронної мережі ResNet18

Властивість	Опис/Значення
Кількість шарів	18
Число ваг	~11 млн
Розмір вхідних даних	224×224 (свою модель модифікую під 32×32)
Активатор	ReLU
Нормалізація	Batch Normalization
Розмір ядра	3×3, 1×1
Розмірність виходу	10, 100
Функція втрат	Перехресна ентропія

У даній таблиці описується модель, яку будемо використовувати. Більш детальних опис роботи даної моделі, можна побачити в розділі 2, де виділено також її переваги та недоліки.

Окрім даної таблиці, в певних завданнях використовується для наглядної

різниці буде використовуватись проста CNN із 5 шарів, в більшості вона є схожою із попередньою.

3.2. Класифікація методів оптимізації

Якщо по простому сказати, то нейромережа – це дуже складна функція, що складається із величезної кількості параметрів. Оптимізація нейронної моделі є одним із основних завдань при розробці, адже це процес покращення її продуктивності та ефективності. Метою оптимізації є мінімізація функції втрат та зменшення розміру моделі, що дозволить мережі краще навчатись та точно визначати результати [32-33].

В загальному це різні методи, які спрямовані на зменшення витрати обчислювальних ресурсів із збереженням або навіть іноді покращенням точності моделі.

Методи оптимізації можна умовно розділити на декілька груп: градієнтні методи, глобальна оптимізація, регуляризація, оптимізація гіперпараметрів.

Градієнтні методи оптимізації є найбільш популярними, вони оновлюють параметри моделі за допомогою похідних. Один із найпростіших таких методів, та й часто використовуваний – це градієнтний спуск. Його суть в тому, що він змінює ваги в протилежному напрямку градієнту функції втрати. Але його недоліком є те, що він не підходить для великих наборів даних, які є поширеними в задачах комп’ютерного зору. Тому замість нього використовують стохастичний градієнтний спуск (SGD), він змінює ваги після кожного окремого навчального прикладу. Є декілька різних модифікацій цього градієнтного спуску [34].

Ці методи є одним із способом оновлення функції втрат, в своїй моделі я буду використовувати Adam для більшості завдань.

Еволюція градієнтних методів дуже просунулась з розробкою адаптивних методів, таких як AdaGrad, RMSProp та Adam, які показали свою ефективність,

особливо в завданнях комп’ютерного зору.

Наступним ключовим інструментом в оптимізації будуть методи глобальної оптимізації. Серед таких найбільш часто використовуються еволюційні алгоритми, які використовують різні механізми для імітації природнього відбору, ці методи часто використовуються для налаштування ідеальних гіперпараметрів [35].

Додатковою стратегією є імітація відпалу, тут йде імітація охолодження металів. Для того, щоб уникнути локальних мінімумів на початку і зосередити увагу на останніх етапах, він поступово знижує інтенсивність пошуку. Також використовуються методи «рою» частинок, вони грунтуються на ідеї взаємодії між частинками, які представляють можливі розв’язки, й забезпечують ефективний пошук у складних функціональних просторах.

Ще важливим елементом оптимізації є регуляризація, про неї я згадував раніше, в цьому розділі ми розберемо її роль в оптимізації моделі. Адже використовуючи методи регуляризації, такі як Dropout, L1, L2 ми не тільки уникаємо проблему перенавчання, а також й зменшуємо складність самої моделі, забезпечуючи покращення результатів [23-24].

Оптимізація гіперпараметрів є не менш важливою частиною оптимізації моделей, адже окрім того, щоб оптимізовувати ваги необхідно правильно підібрати гіперпараметри, а саме кількість шарів, нейронів, швидкість навчання й тд. Класичним методом для вирішення цієї проблеми є Grid Search, але він є малоефективним для великих моделей, такі які використовуються для комп’ютерного зору.

Останнім часом дедалі частіше для оптимізації моделей комп’ютерного зору використовують автоматизовані системи, такі як AutoML чи NAS. Тут ми використовуємо алгоритми та навчання для пошуку оптимальних гіперпараметрів та й навіть архітектури моделі [36].

Також важливим є оптимізація структури моделі, типовими рішеннями цієї проблеми є квантування та проріджування.

Кожна архітектура CNN чи трансформери вимагають різного підходу до оптимізації. Для першої дуже важливо оптимізовувати фільтри та шари, а другі вимагають налаштування механізмів уваги та нормалізації.

Це можна буде помітити на прикладі різниці між простою CNN та ResNet18.

Для ефективної оптимізації найчастіше використовують поєднання всіх методів оптимізації, вибір правильних кращих алгоритмів дає можливість досягати дуже високих результатів у складних завданнях й робити технологію комп'ютерного зору більш доступною.

Більш детально опишу роботу цих методів в цьому розділі.

3.3. Методи градієнтної оптимізації

Як вже говорив вище градієнтні методи є дуже популярним підходом для оптимізації. Вони отримали свою популярність через свою простоту для реалізації та розуміння. В загальному ці методи використовуються для того, щоб підбирати параметри для мінімізації функції втрат, в виконанні задач комп'ютерного зору це є дуже важливим завданням, адже вони часто містять велику кількість параметрів й працюють із великими наборами даних.

Основним методом є градієнтний спуск, цей метод за основу бере опуклу функцію й ітеративно підлаштовує параметри в напрямку зменшення функції втрат до її локального мінімуму [34].

Щоб зрозуміти роботу даного методу можна розглянути його формулу

$$a_{i+1} = a_i - \gamma \nabla f(a) \tag{3.1}$$

де a_i – це інформація про поточний стан моделі на ітерації (параметри);

a_{i+1} – це наступна позиція;

γ – це параметр швидкості навчання;

$\nabla f(a)$ – це градієнт функції втрат.

Ця формула описує наступну позицію в напрямку до локального мінімуму. Тобто алгоритм працює так спочатку визначаються початкові параметри, й далі на кожній ітерації алгоритм шукає такі параметри, які відповідають за найбільш крутий напрямок до точки в якій функція втрати буде найменшою, й це робиться крок за кроком. Це простіше зрозуміти, якщо уявити опуклу вниз площину, і вам потрібно спуститись до найнижчої точки з найменшою кількістю кроків, щоб це зробити ви будете рухатись в найкрутішому напрямку.

Але такі обчислювання вимагають більшої обчислювальної потужності й займає багато часу, особливо для великих моделей, через цю причину даних метод рідко використовується в своєму класичному вигляді, адже появились інші версії, які збільшили ефективність.

Одним із таких алгоритмів є SGD – Stochastic Gradient Descent (Стохастичний градієнтний спуск). Він дуже покращив ефективність звичайного градієнтного спуску через те, що замість того, щоб проходити весь набір даних на кожному кроці, вибирається один випадковий приклад, або декілька прикладів формуючи міні-пакет, це дозволило зменшити кількість обчислень [33].

Формула оновлення параметрів на наступній ітерації залишається такою ж самою, але тепер градієнт обчислюється для невеликої вибірки.

Перевагою цього методу є те, що для великого набору даних він показує високу обчислювальну ефективність, адже в порівнянні із минулим методом обчислювальні витрати дуже зменшуються, адже нам непотрібно проходити всі дані.

Але цей алгоритм додає долю випадковості в процес, що робить процес оптимізації менш стабільним, через що шлях до мінімуму є більш шумним, но цей шум немає великого значення, якщо алгоритм дійде до мінімуму швидше. Тут можна зробити логічний висновок, що якщо шлях є шумним, то він вимагає більшої кількості кроків, ніж традиційні методи, але незважаючи на те, що кількість ітерацій збільшується, він всерівно потребує меншої кількості

обчислювальних ресурсів.

Недоліками є те, що через випадковість й шум ускладнюється шлях до збіжності з глобальним мінімумом й може призвести до коливань навколо оптимальної точки.

Для прикладу в використанні ResNet18 він може показувати себе набагато гірші, ніж інші.

Також є метод який називається Batch Gradient Descent (Пакетний градієнтний спуск), його відмінність від традиційного методу є в тому, що він оновлює параметри, лиш після того, як буде оцінено всі приклади.

Перевагою є те, що він створює стабільний шлях, але на великому наборі даних він буде вимагати велику обчислювальну потужність, особливо пам’ять, адже вимагає того, щоб весь набір даних містився в пам’яті.

Але повернемось до SGD, виявили його недоліки й розробили покращення, одним із таких є Momentum (Метод імпульсу).

Уявіть собі, що ви рухаєтесь до найнижчої точки спуску, наближаючись до цієї точки ви використовуєте повільніший темп, щоб не переступити її, тому варто підібрати повільнішу швидкість, але якщо вона буде занадто маленькою, то шлях для досягнення цієї точки займе дуже багато часу. Так й працює в цих методах, взагалі, якщо швидкість занадто маленька, то модель може подумати, що функція втрат взагалі не покращується й припинити тренування.

Метод імпульсу допомагає прискорити градієнти в правильних напрямках, що дозволяє швидше досягнути мінімуму.

Основним вдосконаленням цього методу є те, що додається «пам’ять» про попередні оновлення, а це допомагає зменшити кількість шуму, більш згладжуючи шлях оптимізації. Алгоритм дозволяє швидше рухатись до мінімуму, зменшуючи коливання [38].

Формули даного алгоритму виглядає ось так

$$v_i = \eta v_{i-1} + \gamma \nabla f(a_i) \tag{3.2}$$

$$a_{i+1} = a_i - v_i \tag{3.3}$$

де v_i – це накопичений імпульс, швидкість;

η – це коефіцієнт демпфування (імпульсу), який визначає, наскільки попередні оновлення беруться до уваги. Зазвичай цей коефіцієнт на початку рівне 0,5, а потім поступово йде до 0,9.

Метод значно покращив попередній, адже прискорив рух й покращив роботу в випадках, коли функція втрат має довгі витягнуті мінімальні області.

Окрім Momentum є алгоритм Nesterov Accelerated Gradient (NAG), який також відомий під ім'ям Nesterov Momentum. Цей алгоритм базується на попередньому, але дещо змінений, а саме додаванням передбачення руху [33].

При роботі цього алгоритму ми обчислюємо значення градієнту не в поточній точці, а в точці на яку вказує імпульс, тобто яка була передбачена ним.

Цей алгоритм також можна показати формулами

$$v_i = \eta v_{i-1} + \gamma \nabla f(a_i - \eta v_{i-1}) \tag{3.4}$$

$$a_{i+1} = a_i - v_i \tag{3.5}$$

Цей алгоритм дозволяє отримати більш точніше оновлення параметрів й уникає зайвого прискорення, такий метод показує ефективність в оптимізації, тому й користується популярністю.

Великий прорив в розвитку градієнтних алгоритмів оптимізації зробила розробка адаптивних алгоритмів. Особливість адаптивних алгоритмів в тому, що вони автоматично змінюють швидкість навчання для параметрів залежно від їхніх попередніх градієнтів.

Попередні методи використовують фіксовану швидкість, що є не хорошим рішенням для всіх параметрів, адже для деяких параметрів можуть бути потрібними частіші оновлення, а іншим навпаки менше оновлень, тому для вирішення цієї проблеми почали використовувати адаптивні методи.

Одним із перших таких алгоритмів став AdaGrad, основа цього методу полягає в накопиченні квадрат градієнтів для кожного параметра, з допомогою цього він може змінювати швидкість навчання для кожного параметра по-різному, залежно від того, наскільки часто він оновлювався. Тобто параметри, які часто оновлюються отримують менші швидкості [33].

Формула цього алгоритму виглядає ось так

$$a_{i+i} = a_i - \frac{\gamma}{\sqrt{G_i + \epsilon}} \nabla f(a_i) \tag{3.6}$$

де G_i – це сума квадратів градієнтів до даної ітерації;

ϵ – це мале чисто, яке буде запобігати діленню на 0.

Даний метод має описані вище переваги, він дозволяє більш автоматизувати темпи навчання, також він гарно працює для розріджених даних, але поступово швидкість навчання можу задатно низько впасти, що уповільнить навчання. Та й він менш ефективний в глибокому навчанні.

Але знову було розроблено наступний алгоритм, який вирішив проблеми AdaGrad, цим алгоритмом став RMSProp.

Основною особливістю було те, що замість того, щоб зберігати суму квадратів градієнтів, зберігати ковзне середнє значення квадратів градієнтів для кожного параметра. За допомогою цього уникається проблема занадто маленьких значень швидкості.

Математично обчислення ковзного середнього значення квадратів градієнтів виглядає ось так

$$E[g_i^2] = \beta E[g_{i-1}^2] + (1 - \beta)\big(\nabla \mathrm{f}(a_i)\big)^2 \tag{3.7}$$

де $E[g_i^2]$ – це поточна оцінка середнього квадратів градієнтів;

β – це коефіцієнт згладжування, він визначає як сильно беруться до уваги

попередні значення, за замовчуванням беруть 0,9.

Після цього йдуть оновлення параметрів моделі:

$$a_{i+1} = a_i - \frac{\gamma}{\sqrt{E[g_i^2]+\epsilon}} \nabla f(a_i) \tag{3.8}$$

Алгоритм вирішив проблему попереднього й став більш ефективним для використання в глибоких нейронних мережах.

Але його наступник покращив його результати через те й став найпоширенішим методом оптимізації. Цим методом став Adaptive Moment Estimation, також відомий коротко Adam [33].

Якраз його я виберу основним методом оптимізації функції втрат нейронної мережі для виконання свого завдання.

В загальному алгоритм виглядає як поєднання ідей та особливостей Momentum та RMSProp. Він використовує два параметри імпульс та ковзне середнього значення квадратів.

Але замість того, щоб змінювати швидкості на основі першого моменту, як в попередньому методі, він використовує другий момент.

Розберемо математичне представлення алгоритму. Спочатку йде оцінка середнього значення градієнтів на перший момент.

$$m_i = \beta_1 m_{i-1} + (1 - \beta_1)\nabla \mathrm{f}(a_i) \tag{3.9}$$

де m_i – це поточна оцінка середнього градієнту, перший момент;

β_1 – це коефіцієнт згладжування для першого моменту, типове значення 0,9.

Після цього відбувається оцінка середнього квадрату градієнтів, другого моменту.

$$v_i = \beta_2 v_{i-1} + (1 - \beta_2)\big(\nabla \mathrm{f}(a_i)\big)^2 \tag{3.10}$$

Відповідно змінні в даній формулі є для другого моменту.

Після цього обчислення відбувається оновлення параметрів, за формулою.

$$a_{i+1} = a_i - \frac{\gamma}{\sqrt{\hat{v}_i + \epsilon}} \hat{m}_i \tag{3.11}$$

де $\hat{m}_i = \frac{m_i}{1-\beta_1^i}$ – це зміщена оцінка середнього градієнта;

$\hat{v}_i = \frac{v_i}{1-\beta_2^i}$ – це зміщена оцінка середнього квадрату градієнтів.

Adam є методом, який став дуже ефективним рішенням для складних задач з великою кількістю параметрів та наборів даних, він з відривом випереджає попередні методи по ефективності.

Також варто згадати, що в сучасних дослідженнях часто використовують різні комбінації методів, розробляючи гібридні методи.

Проаналізувавши попередній опис градієнтних методів оптимізації можна в висновку скласти їхню порівняльну характеристику та описати переваги та недоліки.

Серед звичайних методів лідирує NAG, адже він дозволяє уникнути проблеми попередніх, особливо значно відстає базовий метод градієнтного спуску чи метод пакетного градієнтного спуску через те, що вимагають більшу кількість обчислювальних ресурсів, дані методи можна використовувати, якщо ми працюємо з невеликою кількістю параметрів й коли в нас невеликий набір даних, але для моделей комп’ютерного зору вони не підходять.

Momentum та NAG є ефективними для задач комп’ютерного зору, але програють адаптивним методам, адже потребують налаштування швидкості навчання, якщо налаштувати неправильно, то вони стають нестабільними.

AdaGrad вводить адаптивну швидкість навчання, але через свої проблеми він не є ефективним для навчання глибоких нейронних мереж, які зазвичай використовуються для моделей комп’ютерного зору.

RMSProp виправив проблеми попереднього й став популярним для

глибокого навчання.

Але якщо зробити загальний висновок, то найбільш універсальним методом, який показує свою ефективність для задач різної складності є Adam, тому не дивно що він активно використовується при розробці моделей для комп'ютерного зору й не тільки.

Вибір методу оптимізації може сильно вплинути на якість та швидкість навчання нейронної моделі, тому варто враховувати всі особливості мережі та даних методів.

3.4. Глобальна оптимізація

Глобальна оптимізація є дуже важливою частиною оптимізації, особливо в моделях комп'ютерного зору, які містять дуже багато параметрів. Основна відмінність від локальної оптимізації є в тому, що ми шукаємо найефективніше рішення у всьому просторі параметрів, а не як в локальній – в межах локального простору параметрів, знайти глобальний мінімум набагато складніше.

Глобальна оптимізація використовується не в усіх випадках, її варто використовувати якщо ми знаємо, що функція містить локальні мінімуми, або взагалі структура площини функції втрат. В випадках, якщо ми знаємо, що глобальний мінімум являє собою локальний мінімум варто використовувати локальну оптимізацію, адже її пошук є набагато простішим [32].

Але також варто врахувати, що якщо використовувати лише локальну оптимізацію в випадках, які вимагають глобальної приведе до нижчої продуктивності моделі.

В загальному, часто ми не знаємо як виглядає поверхня функції втрат, чи є там багато локальних мінімумів й тд. В такому випадку варто спочатку реалізувати локальну оптимізацію й визначити базовий рівень продуктивності, після того додати метод глобальної оптимізації й визначити чи покращується продуктивність. Таким чином ми визначимо чи необхідна глобальна

оптимізація, якщо ні, то швидше за все функція є унімодальною, тобто в неї є лише один мінімум, який є й локальним й глобальним.

Існує декілька популярних методів глобальної оптимізації, одними із них є генетичні алгоритми. Основною ідею генетичних алгоритмів була імітація природного відбору, еволюції [35].

Він включає основні поняття, такі як селекція, мутація, схрещення, хромосоми, популяція й тд. Всі ці поняття імітують роботу реальних цих понять з теорії еволюції.

Сама робота методу починається із створення популяції – це група рішень, або саме їх називають хромосомами, кожна таке рішення – це набір параметрів для моделі. Початкова популяція генерується випадково.

Наступним етапом алгоритму є оцінка придатності кожної хромосоми. За допомогою функції придатності, ця функція приймає на вхід параметри та видає їхню придатність на основі функції втрат. Чим менше значення функції втрат, тим більша оцінка.

Після оцінки настає етап селекції. Під час цього етапу вибирають рішення з кращими оцінками, які передають свої гени наступному поколінню, тобто на їх основі створюється наступна популяція.

Також існують процеси схрещення та мутації. У першому випадку вибираються два кращі рішення для об’єднання, комбінуючи параметри, створюючи нові рішення, нащадків. Відповідно в наступному методі додаються випадкові зміни до параметрів, щоб уникати застій роботи методу, додаючи різноманітність.

Метод завершує свою роботу, коли досягає певну кількість популяцій (ітерацій) або коли зміни в функції втрат є несуттєвими.

Перевагами цього методу є те, що він підходить для визначення глобального мінімуму в площині із багатьма локальними мінімумами.

Але недоліком є висока обчислювальна вартість, адже необхідно обчислювати велику кількість популяцій.

Ще одним популярним методом глобальної оптимізації є метод рою частинок (PSO).

Якщо попередній метод базувався на ідеї еволюції, то даний імітує поведінку рою птахів чи комах в природі. Якщо простіше, то кожна комаха рухається в просторі, шукаючи найкраще місце й вони обмінюються цією інформацією з іншими.

Сам метод працює так, що спочатку випадково генеруються частинки й випадково розміщуються в просторі. Кожна частинка може мати свої значення параметрів [39].

Після цього оцінюється розміщення кожної частинки за допомогою функції втрат. Знову таки, чим менше значення цієї функції, тим краще розміщення.

Кожна частинка рухається й враховує декілька параметрів: особистий досвід, тобто вона спирається на найкраще положення, яке знайшла сама, й колективний досвід – це найкраще розміщення, яке знайшли всі частинки.

Швидкість руху частинки обчислюється за формулою

$$v_{i,t+1} = \omega v_{i,t} + c_1 r_1 (p_{best} - x_{i,t)} + c_2 r_2 (g_{best} - x_{i,t)} \qquad (3.12)$$

де $v_{i,t+1}$ – це швидкість частинки і на кроці t;

$x_{i,t}$ – це розміщення частинки;

p_{best} – це найкраще своє розміщення;

g_{best} – це найкраще колективне розміщення.

с – це параметри, які відповідають за те наскільки сильно орієнтується на своє та колективне розміщення, типове значення 2;

r – це випадкові числа в діапазоні від 0 до 1, вони визначають силу тяжіння до розміщення;

ω – це початковий коефіцієнт інерції, він може зменшуватись із кожною ітерацією.

Після цього частинки рухаються із оновленою швидкістю. Сам процес

повторюється поки не буде досягнуто мінімуму або ж не досягне певної кількості ітерацій.

Перевагою даного алгоритму є те, що він простий в реалізіції та те, що він є ефективним в задачах з великою кількістю параметрів. Але він не лишився без недоліків, основним недоліком є те, що частинки можуть «блукати» поблизу локального мінімуму.

Наступним методом є метод імітації відпалу. Він базується на фізичному процесі охолодження металів, коли спочатку метал нагрівають, а потім повільно охолоджують для того, щоб виявити дефекти [40].

При роботі даного алгоритму спочатку вибирається високе значення температури, чим більше значення тим більша ймовірність того, що алгоритм прийме гірші рішення.

На кожному кроці відбувається оцінка розміщення за допомогою функції втрат, тобто, якщо нове рішення є краще за наявне, то воно стає наявним. Але існує ймовірність того, що гірше значення буде також прийняте, ця ймовірність обчислюється за формулою

$$P = e(-\frac{\Delta f}{T}) \tag{3.13}$$

де Δf – це дельта функції втрат;

T – температура.

Висока температура дозволяє виходити із локальних мінімумів.

Під час роботи алгоритму значення температури поступово знижується, тобто відбувається процес охолодження, а це означає, що ймовірність прийняття кращих рішень стає більшою.

Сама робота алгоритму закінчується, коли температура опускається до мінімуму або ж коли зміна функції втрат зупиняється.

Даний алгоритм має перевагу в тому, що він також здатний ефективно виходити із локальних мінімумів та має просту реалізацію, але його недоліками

є те, що він має повільну збіжність на останніх етапах, порівняно із іншими методами. Також варто врахувати те, що він вимагає багато часу на обчислення та дуже чутливий до налаштувань параметрів.

Існує також й байєсівська оптимізація, він для пошуку глобального мінімуму використовує ймовірнісну модель для передбачення значення функції втрат в нових точках [41].

Спочатку виконується обчислення функції втрат в випадкових точках. Після того використовується ймовірнісна модель, для прикладу гаусовий процес, що передбачає значення функції втрат у всьому просторі.

Далі обчислюється функція придатності, яка визначає в яких наступних точках буде обчислюватись функція втрат. Після обчислення результати додаються до гаусового процесу для уточнення передбачення.

Процес повторюється поки не буде досягнуто мінімального значення функції втрат або ж після певної кількості ітерацій.

Перевагами є те, що він зменшує кількість обчислень функції втрат, що важливо в завданнях, коли обчислення вимагають великою кількості ресурсів.

Але алгоритм погано працює для задач з великою кількістю параметрів, адже тоді обчислювальні витрати стають дуже високими, також він має складну реалізацію, особливо порівняно із попередніми.

Ці методи є дуже перспективними в використанні в моделях комп’ютерного зору, для багатьох різних завдань. В залежності від завдання й можливостей вибирають підходящі алгоритми. Для прикладу в задачах оптимізації гіперпараметрів та вибору архітектури використовують й генетичні алгоритми й байєсівську оптимізацію, залежно від складності моделі. Також часто використовують гібридні підходи для виконання різних специфічних випадках.

Но отримувати починають більшу популярність генетичні алгоритми через їхню універсальність.

Їх зазвичай використовують саме для дуже складних функцій втрат, що не підходить для нашого завдання.

3.5. Оптимізація гіперпараметрів та структури моделі

Оптимізація гіперпараметрів є одним із дуже важливих складових для оптимізації навчання нейромереж. Раніше всі ці завдання виконувались вручну, але це вимагає дуже багато часу та знань. Тому дослідження почались в напрямку, щоб автоматизувати даних процес. Це допомагає ефективніше налаштовувати їх та покращувати навчання мережі.

Для цієї задачі найчастіше використовується метод, який був розглянутий раніше – байєсівська оптимізація.

Але зараз детальніше розберемо саме методи для автоматизації процесу.

Одним із методів є NAS – Neural Architecture Search, із назви можна зрозуміти, що він призначений для автоматизованого пошуку архітектури нейронної мережі. Його ідея заключається в тому, щоб замість налаштування архітектури вручну, він використовує алгоритми для пошуку оптимального рішення [37].

Спочатку формується простір пошуку, визначається набір можливих архітектур й їхні налаштування – кількість шарів, типи цих шарів, розміри ядра й тд. Цей простір для складних моделей можу бути дуже великим.

Після цього починає працювати метод пошуку, це можуть бути ті методи, які ми описали вище для локальної чи глобальної оптимізації, а саме часто використовуються генетичні алгоритми, байєсівська оптимізація чи градієнтні методи.

Згенеровані мережі навчаються, зазвичай на невеликому наборі даних, а потім відбувається їхня оцінка за точністю або ж функцією втрат, й кращі архітектури використовуються для наступних поколінь.

Після закінчення роботи методу NAS повертає архітектуру, яка є найефективнішою для виконання даного завдання.

Тобто ми можемо автоматично визначити оптимальні налаштування для дуже глибоких мереж, для прикладу, для виконання завдання сегментації, тобто

оптимальну глибину, розміри ядра й тд.

Даним метод має багато переваг, адже ми отримуємо оптимізовані гіперпараметри для нейронної моделі, замість того, щоб вручну їх налаштовувати, що зайняло б дуже багато часу, особливо для надзвичайно складних нейронних мереж.

Але незважаючи на переваги метод має й недоліки, найбільшим недоліком є висока обчислювальна вартість.

Зважаючи на цей недолік можна зробити висновок, що він не підходить для всіх задач, варто зрозуміти чи є доцільним його використання. Коли налаштування параметрів вручну є дуже складним завданням, тоді він підійде.

Також існує метод AutoML, його ідеєю є автоматизація повністю всього процесу навчання. Це дозволяє створювати моделі без необхідності глибоких знань [37].

AutoML складається із декількох основних компонентів.

Одним із таких компонентів є оптимізація гіперпараметрів, а саме підбір швидкості навчання, регуляризації, розміру й тд, він визначає набір значень для кожного такого параметра.

Наступним компонентом буде вибір архітектури, тут може використовуватись вище описаний NAS.

Також важливим є вибір методів нормалізації, регуляризації обробки ознак й тд.

Для того, щоб знайти оптимальні комбінації гіперпараметрів AutoML використовує різні алгоритми, серед них є вже вищезгадані генетичні алгоритми та байєсівська оптимізація. Але також іноді використовується Grid Search – це покроковий перебір усіх комбінацій параметрів, він потребує багато обчислювальних ресурсів, але найпростіший із усіх. Також ще використовується Random Search – це вже використання випадкових комбінацій, він є менш ефективним, але набагато швидший за попередній.

Як й усіх цих випадках виконується оцінка моделі для кожної комбінації

параметрів, вона перевіряється на певному валідаційному наборі даних, це може оцінюватись точність або функція втрат.

Результатом виконання методу буде найкраща модель із найбільш оптимальними гіперпараметрами.

У його використанні також є багато переваг, такі як мінімальна потреба в участі людей, які мають знання в галузі, також перевагою буде його універсальність, адже він підходить до багатьох різних завдань.

Але він має той самий недолік, що й NAS – потребу великих обчислювальних ресурсів.

Окрім оптимізації гіперпараметрів також важливо оптимізувати саме модель.

Для виконання даної задачі використовуються різні методи, одним із них є Pruning (Метод прорідження). Сучасні нейронні моделі для задач комп'ютерного зору можуть містити величезну кількість параметрів (нейронів чи фільтрів), які збільшують кількість й складність обчислення. Тому ідея методу полягає в тому, щоб видаляти зайві параметри [21].

Спочатку метод оцінює важливість кожного параметра чи нейрона, аналізує його внесок в функцію втрат. Для прикладу, якщо в параметра є малі ваги він може вважатись не дуже важливим.

Після того, як відбулась оцінка він видаляє параметри чи нейрони, які вважаються неважливими через незначний вплив.

Далі модель перенавчається для відновлення точності, яка може знизитись через виконання методу. Й відбувається оцінка, яка перевіряє, чи збереглась точність моделі після зниження складності.

Існує два типи проріджения – локальна та узагальне, в першому випадку викидаються неважливі параметри лише для деяких шарів, а в другому – по всій моделі.

Перевагами цього методу є зменшення розміру моделі й складності, що спрощує її розгортання, а також він прискорює обчислення. Але крім цього

існують й недоліки, основними з яких є зниження точності, якщо метод налаштований дуже «агресивно», також недоліком є те, що метод потребує додаткового перенавчання, яких може бути декілька.

Для оптимізації ResNet18 я буду використовувати саме загальний тип обрізки.

Наступним методом для оптимізації самої моделі є метод Quantization (Квантування).

Основною ідеєю цього методу є зменшення розміру моделі шляхом зниження точності числових представлень ваг і активацій моделі. Для прикладу перетворення 32-бітних чисел до 8-бітних або ж менших [21].

Самі методи квантування поділяються на дві категорії – квантування після навчання та під час навчання.

У випадку квантування після навчання метод застосовується після того, як модель вже повністю навчена. Тут параметри перетворюються на числа з нижчими бітами, але перенавчання моделі не відбувається.

Така версія методу використовується для розгортання моделей на пристроях з обмеженими обчислювальними ресурсами, але недоліком цього методу є те, що втрачається точність, а також накопичується помилка апроксимації, що впливає на складні завдання, для прикладу розпізнавання дрібних деталей зображення.

Для того щоб пом’якшити дані недоліки використовуються дані калібрування – це певна під-набір даних, за допомогою якого буде проведена оцінка моделі.

Ці дані допомагають знайти оптимальні параметри квантування, щоб максимально зменшити похибку й зберегти максимально можливу точність моделі.

Під час квантування під час навчання воно впроваджується в процес навчання. Такий підхід дає можливість моделі адаптуватись до зменшення формату, що дозволяє мінімізувати втрати в продуктивності моделі.

Окрім того, таке навчання дозволяє налаштувати процес квантування для

кожного параметру.

Результатом цього є модель, яка є більш точною, порівняно із квантуванням після навчання.

Перевагами цього методу оптимізації є зменшення розміру, прискорення обчислення, зниження складності, а це дозволяє використовувати таку модель на пристроях з обмеженими ресурсами.

Але варто враховувати недоліки, адже такий метод не підходить для завдань, які вимагають високої точності, для прикладу завдання в медичній сфері.

Використовуватиму квантування після навчання, адже воно уникає більшості недоліків при оптимізації ResNet18.

Метод який також підходить для оптимізації моделі є метод дистиляції знань, основна ідея в цьому методі є передача знань від потужної моделі до меншої, щоб менша модель навчилась повторювати поведінку більшої [22].

Для роботи даного методу нам потрібна вже навчена потужна модель, яка показує високу точність, це буде нашим вчителем, та менша модель, вона може бути взагалі не навчена, це учень.

Спочатку вчитель генерує результат, який може бути в вигляді м’яких міток, тобто ймовірності замість жорстких міток.

Після того йде навчання моделі-учня, яка для цього використовує дві функції втрат, класичну, яка порівнює результат учня з правильними мітками, та функцію дистиляції, яка порівнює результат учня з результатом вчителя, змушуючи імітувати його поведінку.

Також враховується параметр температури дистиляції, який використовується для згладжування результатів моделі-вчителя.

Для того, щоб модель-учень могла ефективно вчитись в учителя потрібно, щоб в них була архітектурна сумісність. Тому необхідно знайти таку модель-учня, яка б була ефективною та могла б найбільш точніше імітувати вчителя.

Перевагами використання такого методу є створення менших моделей, які

мають високу точність, також ефективність на мобільних пристроях, та гнучкість цього методу, адже він може працювати з різними архітектурами.

Але варто взяти до уваги також й недоліки використання дистиляції, адже він дуже залежить від вчителя, адже якщо він є недостатньо точним, то метод не дасть великого покращення точності, також навчання вимагає додаткових обчислень, адже необхідно обчислювати дві моделі.

Дані методи є дуже важливими для оптимізації моделей комп'ютерного зору. Адже оптимізація гіперпараметрів допомагає створити модель з оптимальними параметрами без ручного налаштування, особливо в складних задачах, прикладами таких Google AutoML Vision,, H2O AutoML та NAS-FPN.

Також методи квантування, прорідження та дистиляції допомагає при створенні моделей менших розмірів для розгортання їх на мобільних пристроях зберігаючи максимально можливу високу точність.

3.6. Роль регуляризації в оптимізації

Регуляризація – це методи, які допомагають моделі уникати переначання, такі методи вже були згадані раніше. Але також такі методи відносяться й до оптимізації, адже вони зменшують складність моделі, роблячи навчання більш ефективним й підвищуючи стабільність моделі.

Регуляризація додає до функції втрат штраф, для прикладу за великі ваги.

Існують декілька видів регуляризації, для прикладу L1-регуляризація [24]. Даний тип додає до функції втрат штаф за суму абсолютних значень втрат. Такі штрафи змушують зменшувати деякі ваги, іноді зменшуючи їх до обнулення, цей процес допомагає створювати розріджувані моделі, бо таким способом він усуває менш важливі ознаки. Це робить дану регуляризацію особливо корисною в завданнях з їхньою великою кількістю, адже вона дозволяє ефективно вибирати важливі ознаки, що зменшує складність обчислень.

Але недоліком даної регуляризації є те, що вона може бути нестабільшою в

складних моделях, вона також не підходить для завдань, якщо всі ознаки є важливими.

Схожим типом є L2-регуляризація, але вона додає штраф за суму квадратів ваг. Цей тип також зменшує великі ваги, але не обнуляє їх. Та якраз вона не надає переваги одних ознак над іншими, через те модель стає стабільшішою.

Недоліком даної регуляризації є те, що вона не видаляє непотрібні ознаки, лиш просто зменшує їх вплив, тому її зазвичай використовують в випадках, якщо ми вважаємо, що всі ознаки є важливими.

Техніка регуляризації Dropout з певною ймовірністю виключає деякі нейрони під час кожної ітерації навчання. Зазвичай типовим значенням цієї ймовірності є 0,5.

Цю техніку регуляризації можна розглядати як компонент оптимізації через те, що вона зменшує складність моделі, адже через виключення частини нейронів модель стає простішою в кожній ітерації.

Dropout знижує залежність моделі від окремих нейронів, через це покращуючи узагальнення.

Випадковість методу допомагає досліджувати ширший простір параметрів, що також допомагає покращувати навчання.

Для ResNet18 буде краще використання L1-регуляризації та Dropout, адже використовувані набори тестових даних не містять однакових за важливістю ознак.

3.7. Висновок до розділу

У даному розділі було проведено детальний аналіз методів оптимізації, описано їхні переваги та недоліки. Розглянуто класифікацію методів оптимізації для нейронних мереж.

Було проведено аналіз понять глобальної та локальної оптимізації для того, щоб зрозуміти в яких завданнях які методи ефективніше використовувати.

Описано різновиди методів градієнтного спуску й визначено найбільш оптимальний для задач комп’ютерного зору.

Також було проведено аналіз оптимізації гіперпараметрів, описано методи для автоматизації визначення оптимальних параметрів та архітектури, розказано про їхні головні переваги, та для яких завдань ефективніше використовувати.

Визначено важливість методів оптимізації структури моделі та переваги їхнього використання для пристроїв з обмеженими обчислювальними ресурсами.

Також проаналізовано роль регуляризації в оптимізації, розглянуто способи регуляризації, які зменшують складність моделі.

РОЗДІЛ 4
ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ПОРІВНЯННЯ АЛГОРИТМІВ ОПТИМІЗАЦІЇ

4.1. Підготовка до дослідження

В попередніх розділах було детально описано роботу нейронних модель для задач комп’ютерного зору, а також методи оптимізації, які використовують, від примітивних до більш складніших.

Для реалізації методів я використовував мову програмування Python. Вибрав цю мову, адже вважаю її однією із найкращих для дослідження в галузі нейронних мереж та їхньої оптимізації. Вона має багато переваг в цьому, однією з яких є популярність, через це вона є одним із стандартів для машинного навчання, також доступна велика кількість навчальних ресурсів та документації. Окрім того наявна підтримка багатьох бібліотек та фреймворків для роботи із ШІ, такі як PyTorch, TensorFlow, Keras, які дають більше можливостей для дослідження та експериментів. Також наявні бібліотеки для легкої візуалізації даних в графіках, таблицях чи діаграмах.

Для створення моделей нейронних мереж та обчислень зв’язаних з ними використовував бібліотеку PyTorch. Вона підходить для виконання завдання, адже вже здобула свою популярність через простоту та гнучкість роботи. Окрім цього для задач комп’ютерного зору вона підтримує бібліотеку Torchvision, яка містить готові набори даних для навчання та тестування нейронної мережі, через що не потрібно буде збирати ці дані. Також PyTorch дає можливість простішої реалізації складних архітектур згорткових нейронних мереж чи трансформерів. Бібліотека підтримує можливість навчання нейронної моделі через використання графічного процесора, що значно прискорить навчання.

Також використовував бібліотеку MatPlotLib для створення візуалізації даних через графіки та таблиці, для простішого сприйняття та наглядного

зображення різниці.

Також для тренування й тестування власної мережі використовував набір даних CIFAR-10 та для більшого набору CIFAR-100, вони є досить популярними для навчання невеликих мереж, містять зображення 32 × 32, що дозволить швидко тестувати. В бібліотеці CIFAR-10 містяться зображення 10 різних класів, що дає можливість навчити модель для простих завдань, щодо CIFAR-100 – це 100 класів, що дає можливість тестувати модель на складних задачах. Також невелика роздільна здатність також дає певну складність моделі, адже покаже як вони пристосовані до шуму. Цей набір доступний в бібліотеці Torchvision, тому це спростить його використання для нашого завдання.

Для написання коду використовував безкоштовний текстовий редактор Microsoft Visual Studio Code, через його простоту, швидкість та легкість використання.

Метою дослідження є порівняння методів оптимізації нейронних мереж, визначення оптимальних для задач комп'ютерного зору, аналіз того, як вони впливають на точність нейронної моделі та швидкість навчання.

Для порівняння градієнтних методів оптимізації використовував методи: SGD, Momentum, Nesterov Momentum, Adagrad, RMSProp, Adam, та комбінацію Nadam, вибрав ці методи адже вони користуються популярністю для виконання різних завдань, також для порівняння з сучасними методи вибрав базовий SGD.

Також порівняв методи структури нейронної моделі, а саме Pruning та квантування, визначив ефективність даних методів на практиці.

Окрім цього необхідно було порівняти методи регуляризації нейронних мереж, як вони справляються із оптимізацією та профілактикою для проблеми перенавчання моделі.

Використовував моделі різної складності з різною кількістю параметрів, для демонстрації оптимальних алгоритмів оптимізації для неї. А саме використовував просту CNN з різною кількістю шарів та ResNet18. Це допоможе краще порівняти різні методи.

Для оцінки роботи методів використовувались різні метрики залежно від завдання й ідеєю роботи методів. Основними метриками звісно, що є точність на тестових чи тренувальних даних, зміна функції втрат, розмір моделі й тд.

4.2. Порівняння методів градієнтного спуску

При описі градієнтної оптимізації було описано детальну роботу всіх використовуваних при дослідженні методів, їхньою головною ціллю є зменшення функції втрат, для більш ефективного й швидшого навчання нейронної мережі.

Спочатку ми ці методи будемо порівнювати для завдання класифікації з простою CNN, код представлений в лістингу 4.1.

Лістинг 4.1.

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(32 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Згідно коду можна побачити, що я використовую мережу із 2 згортковими шарами та 2 повнозв’язними.

Перший згортковий шар містить 3 входи, для прикладу для трьох-канального зображення, та 16 фільтрів (виходів) розміром 3×3, та містить відступ 1.

Другий згортковий шар містить 16 входів та 32 фільтри такого ж розміру та з таким же відступом.

Перший повнозв’язний шар приймає на вхід дані розміром 32*8*8 та 128 виходів, відповідно другий такий шар приймає 128 входів та має 10 виходів – класів, відповідно до використовуваного набору даних.

Також для активації шарів використовується функція ReLu. Окрім цього виконуються операції Max Pooling для зменшення розміру зображення. Варто помітити, що перед передачею даних в повнозв’язний шар зображення формуються в одновимірний вектор.

Як говорив вище для цієї задачі використовую набір даних із 10 класами – CIFAR-10. Тому спочатку потрібно завантажити ці дані, а саме тестовий та навчальний набір, приклад цього показано в лістингу 4.2.

Лістинг 4.2.

```
    train_dataset = torchvision.datasets.CIFAR10(root='./data', 
train=True, download=True, transform=transform)
    train_loader = torch.utils.data.DataLoader(train_dataset, 
batch_size=64, shuffle=True)

    test_dataset = torchvision.datasets.CIFAR10(root='./data', 
train=False, download=True, transform=transform)
    test_loader = torch.utils.data.DataLoader(test_dataset, 
batch_size=64, shuffle=False)
```

В даному коді можемо помітити, що ми завантажуємо тестовий та навчальний набір. Також ми перетворюємо дані та нормалізуємо їх в методі transform.

Навчальний набір містить 60000 зображень, а тестовий 10000, ці фото ми завантажуємо, окрім цього вказаний розмір батчу в 64, це означає, що наші методи не будуть проходити по кожному зображенню, а використовувати пакети по 64 випадкові зображення.

Окрім цього нам необхідно реалізувати методи градієнтної оптимізації, в бібліотеці містяться вбудовані ці методи, але для цього завдання я вирішив

реалізувати їх вручну.

Розпочнемо із базового SGD, код його можна помітити в лістингу 4.3.

Лістинг 4.3.

```
class SGD:
def __init__(self, params, lr=0.001):
    self.lr = lr
    self.params = list(params)

def step(self):
    with torch.no_grad():
        for param in self.params:
            if param.grad is not None:
                param -= self.lr * param.grad

def zero_grad(self):
    for param in self.params:
        if param.grad is not None:
            param.grad.zero_()
```

Даний метод є найпростішим в реалізації, обчислюється все за формулою, яка була описана в попередньому розділі. В даному випадку, як сказав вище його краще назвати пакетним SGD, адже використовує пакети.

Створений окремий клас, який містить метод ініціалізації, кроку оптимізатора та скидання градієнтів. Метод перебирає всі параметри моделі в яких є градієнти, якщо так, то параметр оновлюється шляхом його віднімання, градієнт множиться на швидкість навчання, в нашому випадку її значення 0.001.

Наступним методом буде метод Momentum, приклад його реалізації можна помітити в лістингу 4.4.

Лістинг 4.4.

```
class Momentum:
def __init__(self, params, lr=0.001, momentum=0.9):
    self.lr = lr
    self.momentum = momentum
    self.params = list(params)
    self.velocity = [torch.zeros_like(p) for p in self.params]

def step(self):
```

```
        with torch.no_grad():
            for i, param in enumerate(self.params):
                if param.grad is not None:
                    self.velocity[i] = self.momentum *
self.velocity[i] - self.lr * param.grad
                    param += self.velocity[i]

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

Можна помітити, що сюди додається параметр імпульсу, який в даному випадку дорівнює типовому значенню 0.9. Всі обчислення виконуються за формулами. Даний метод використовує імпульс для оновлення параметрів мережі.

Після цього методу описується метод Nesterov Momentum, його вигляд в лістингу 4.5.

Лістинг 4.5.

```
    class Nesterov:
    def __init__(self, params, lr=0.001, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.params = list(params)
        self.velocity = [torch.zeros_like(p) for p in self.params]

    def step(self):
        with torch.no_grad():
            for i, param in enumerate(self.params):
                if param.grad is not None:
                    prev_velocity = self.velocity[i].clone()
                    self.velocity[i] = self.momentum *
self.velocity[i] - self.lr * param.grad
                    param += -self.momentum * prev_velocity + (1 +
self.momentum) * self.velocity[i]
```

Даний метод відрізняється від попереднього тим, що тут ми використовуємо передбачення. Цей метод також містить функцію zero_grad(), лиш у всіх він має однакову реалізацію, тому не буду його вказувати.

Це про звичайні методи, далі реалізовувались адаптивні методи

градієнтного спуску.

Спочатку буде метод під назвою AdaGrad, його реалізація показана в лістингу 4.6.

Лістинг 4.6.

```
    class AdaGrad:
    def __init__(self, params, lr=0.01, eps=1e-8):
        self.lr = lr
        self.eps = eps
        self.params = list(params)
        self.sum_of_squares = [torch.zeros_like(p) for p in
self.params]

    def step(self):
        with torch.no_grad():
            for i, param in enumerate(self.params):
                if param.grad is not None:
                    self.sum_of_squares[i] += param.grad ** 2
                    param -= self.lr * param.grad /
(self.sum_of_squares[i].sqrt() + self.eps)
```

Можна помітити значення епсилон в параметрах, а також особливість цього методу, яка полягає в накопиченні квадрату градієнтів параметра.

Ще цікавим адаптивним методом є RMSProp, код алгоритму в лістингу 4.7.

Лістинг 4.7.

```
    class RMSProp:
    def __init__(self, params, lr=0.001, alpha=0.99, eps=1e-8):
        self.lr = lr
        self.alpha = alpha
        self.eps = eps
        self.params = list(params)
        self.mean_square = [torch.zeros_like(p) for p in
self.params]

    def step(self):
        with torch.no_grad():
            for i, param in enumerate(self.params):
                if param.grad is not None:
                    self.mean_square[i] = self.alpha *
self.mean_square[i] + (1 - self.alpha) * param.grad ** 2
                    param -= self.lr * param.grad /
(self.mean_square[i].sqrt() + self.eps)
```

В коді можна помітити типове значення альфа, а саме коефіцієнт згладжування. Також ми зберігаємо середнє значення квадратів, порівняно із минулим.

Останнім методом буде метод Adam, який є одним із найчастіше застосовуваних. В лістингу 4.8. можна побачити приклад його реалізації.

Лістинг 4.8.

```
    class Adam:
    def __init__(self, params, lr=0.001, beta1=0.9, beta2=0.999,
eps=1e-8):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.eps = eps
        self.params = list(params)
        self.m = [torch.zeros_like(p) for p in self.params]
        self.v = [torch.zeros_like(p) for p in self.params]
        self.t = 0

    def step(self):
        with torch.no_grad():
            self.t += 1
            for i, param in enumerate(self.params):
                if param.grad is not None:
                    self.m[i] = self.beta1 * self.m[i] + (1 -
self.beta1) * param.grad
                    self.v[i] = self.beta2 * self.v[i] + (1 -
self.beta2) * (param.grad ** 2)
                    m_hat = self.m[i] / (1 - self.beta1 ** self.t)
                    v_hat = self.v[i] / (1 - self.beta2 ** self.t)
                    param -= self.lr * m_hat / (v_hat.sqrt() +
self.eps)
```

Даний метод помітно відрізняється кількістю параметрів та обчислення порівняно із попередніми, вони збільшують час виконання методу, порівняно із іншими.

Після опису методів й мережі ми перейдемо до її тренування та тестування.

Порівнювати ці методи будемо за значенням функції втрат за ітераціями та епохами, та також за точністю мереж навчених з даними оптимізаторами.

Епоха – це час за який мережа пройде весь навчальний набір даних, їх ми

будемо використовувати декілька.

Для точного порівняння всі оптимізатори будуть починати для нових моделей, щоб не було так, що мережа продовжує своє навчання замість того, щоб почати заново.

Результати виконання коду можна побачити за рис. 4.1. – 4.3.

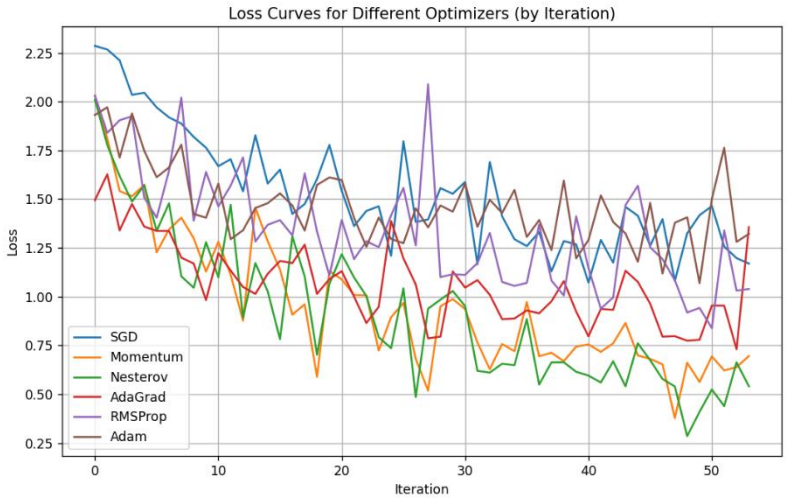

Рис. 4.1. Порівняння функції втрат градієнтних методів по ітераціях

Згідно рис. 4.1. можна помітити, що зміна функції втрат по ітераціях (значення записувалось кожні 100 ітерацій) є досить шумною, що для цих методів є нормальним явищем, порівняно із базовим GD, але він не використовується для завдань із великою кількістю даних, адже вимагає великої кількості пам’яті та обчислювальних ресурсів.

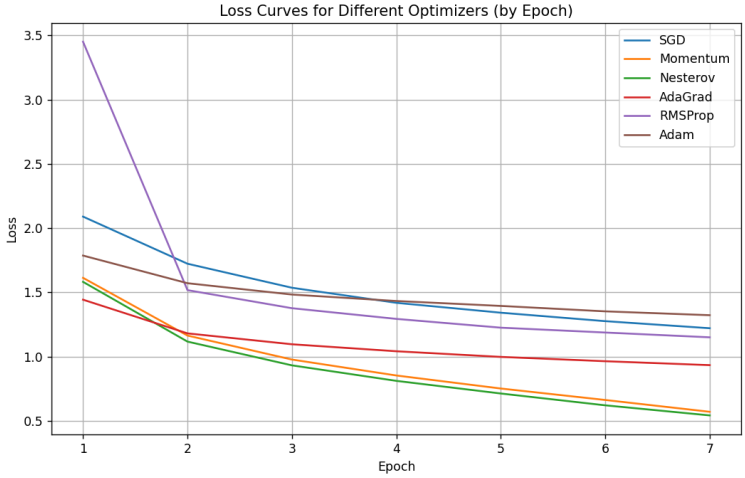

Рис. 4.2. Функція втрат градієнтних методів по епохах

Для кращої наглядності я зробив окремий графік (рис. 4.2.), де записував значення функції втрат по епохах, тут можна помітити, що методи Momentum та Nesterov вирвались вперед порівняно із іншими.

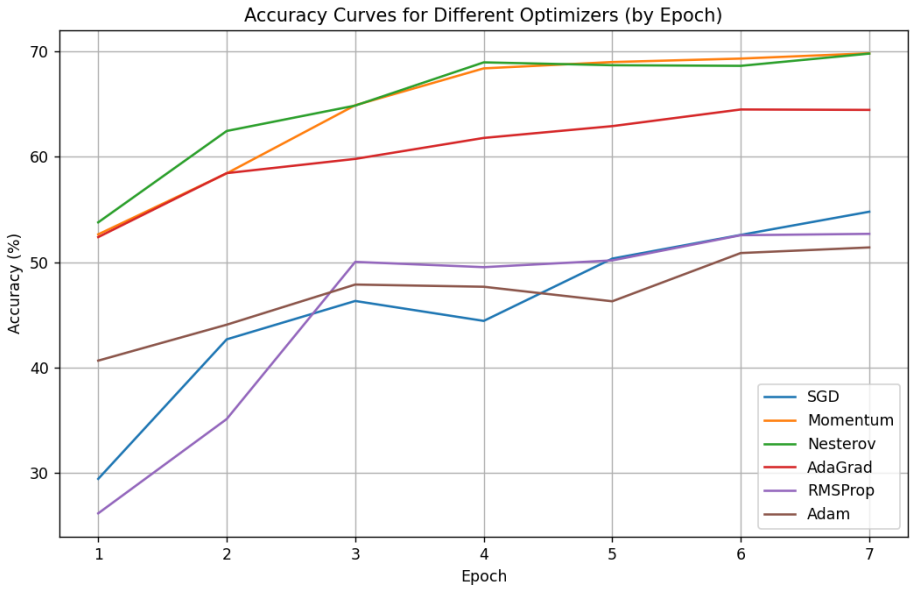

Рис. 4.3. Точність методів градієнтної оптимізації за епохами

Після проходження епохи обчислювалась точність, на рис. 4.3. також можна побачити, що графіки методів Nesterov та Momentum вирвались вперед.

Після проведення аналізів даного дослідження можна побачити, що найбільш ефективними методами будуть Nesterov та Momentum, а саме найкращим буде саме першим, адже показує найбільшу точність та найменшу функцію втрат. Така різниця між всіма методами є через те, що ми використовуємо дуже просту модель, на якій адаптивні методи показують низьку продуктивність.

Найменш точним виявився метод Adam, окрім цього він виконує більше обчислень, тому їхній час також більший.

Далі для кращого порівняння я виконав дослідження цих методів за допомогою більш складної мережі, а саме ResNet18 також для завдання класифікації.

Для створення даної мережі я використав вже визначену мережу в PyTorch.

Код створення цієї мережі представлено в лістингу 4.9.

Лістинг 4.9.

```
    class ResNet18(nn.Module):
        def __init__(self):
                    super(ResNet18, self).__init__()
                    self.model =
torchvision.models.resnet18(weights = None)
                    self.model.fc =
nn.Linear(self.model.fc.in_features, 100)

                def forward(self, x):
                    return self.model(x)
```

Використовується базова модель ResNet18 із відповідно 18 шарами, модель створює пустою, без тренування наперед.

Також для більшої складності завдання я використовував не CIFAR-10, а більший набір даних CIFAR-100. Розмір міні-пакету такий же, як й в попередньому.

Після навчання ми отримали результати, які відрізняються від попередніх (рис. 4.4. – 4.6.).

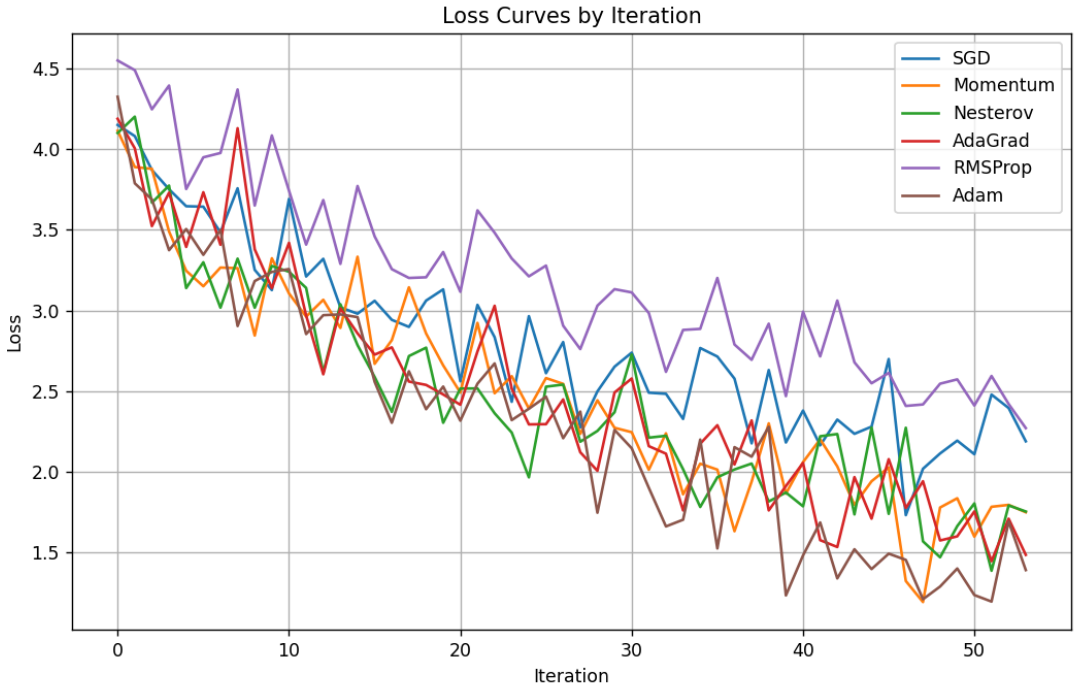

Рис. 4.4. Функція втрат градієнтних методів по ітераціях (ResNet18)

Зразу стає видно, що Adam, який програвав в попередній задачі, в даному випадку показує найменші результати функції втрат, для кращого вигляду сформував графік по епохам (рис. 4.5.).

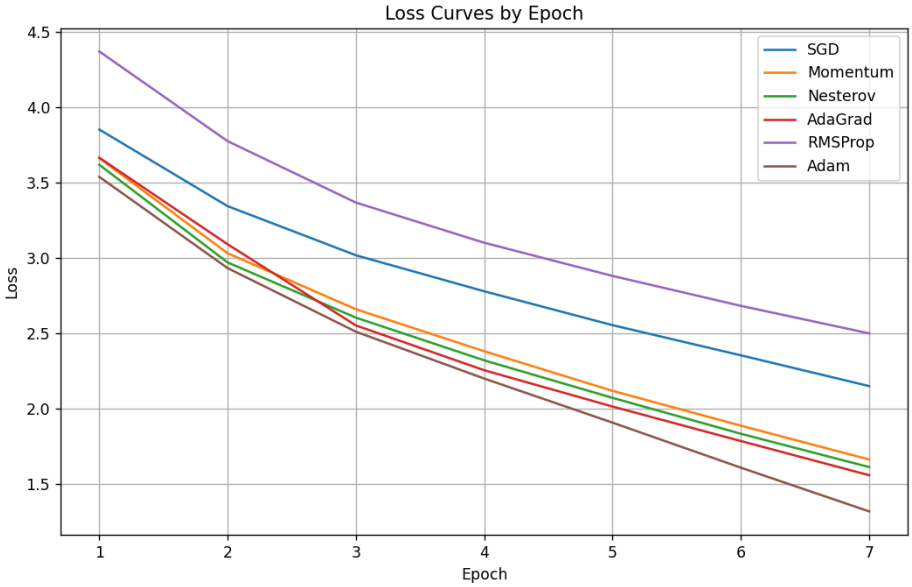

Рис. 4.5. Функція втрат градієнтних методів по епохах (ResNet18)

По епохах можна помітити, що адаптивні методи вирвались більше вперед. Особливо порівняно із SGD, також не ефективним є метод RMSProp.

Також потрібно оглянути графік точності (рис. 4.6.).

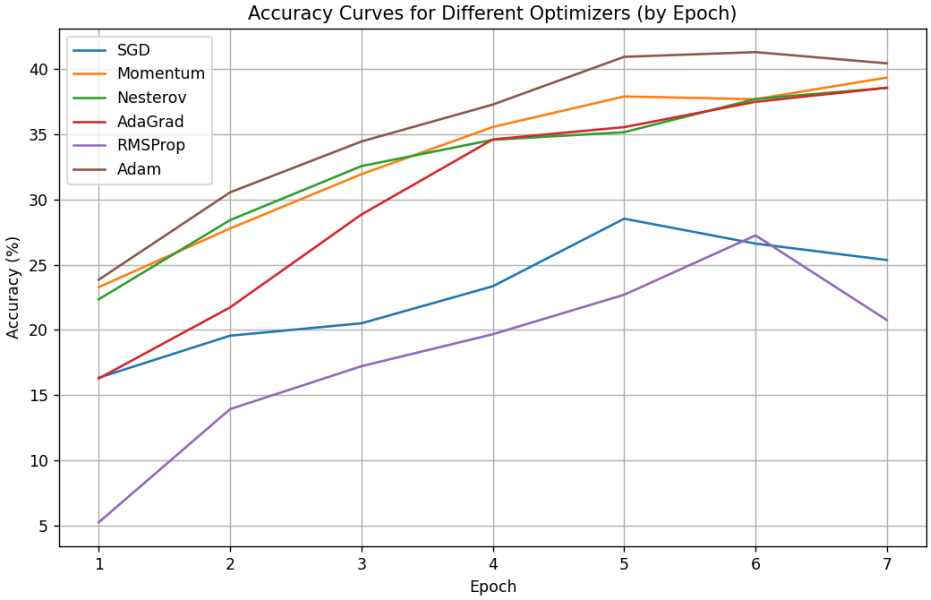

Рис. 4.6. Точність градієнтних методів по епохах (ResNet18)

Після аналізу цього завдання стає видно, що порівняно із простою нейронною моделлю, використання адаптивних методів є більш ефективним й кращим варіантом, тут виявилось, що найкращим методом для складної мережі буде саме Adam, адже він показав найбільшу точність та найменшу функцію втрат.

Окрім звичайних методів градієнтної оптимізації я порівняв метод Adam та його гібрид з Nesterov під назвою Nadam, реалізація методу в лістингу 4.10.

Лістинг 4.10.

```
    class NadamOptimizer:
    def __init__(self, params, lr=0.001, beta1=0.9, beta2=0.999,
eps=1e-8):
        self.params = list(params)
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.eps = eps
        self.m = [torch.zeros_like(param).to(device) for param in
self.params]
        self.v = [torch.zeros_like(param).to(device) for param in
self.params]
        self.t = 0
    def step(self):
        self.t += 1
        for i, param in enumerate(self.params):
            if param.grad is None:
                continue
            grad = param.grad
            self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1)
* grad
            self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2)
* grad**2
            m_hat = self.m[i] / (1 - self.beta1 ** self.t)
            v_hat = self.v[i] / (1 - self.beta2 ** self.t)
            m_hat_prime = self.beta1 * m_hat + (1 - self.beta1) *
grad
            param.data -= self.lr * m_hat_prime /
(torch.sqrt(v_hat) + self.eps)
```

Різниця даного методу від попереднього в тому, що до класичного Adam додається модифікований імпульс Nesterov, він обчислюється в m_hat_prime, а далі за його допомогою оновлює параметри моделі.

Результат порівняння зображений на рис. 4.7.

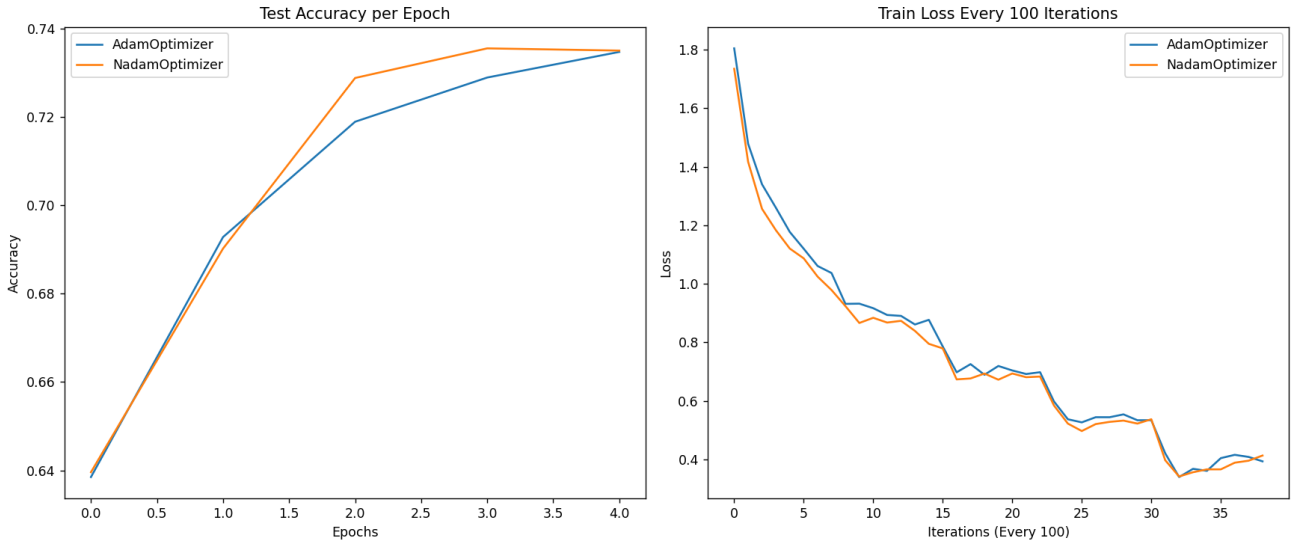

Рис. 4.7. Результат порівняння градієнтних методів Adam та Nadam

На графіках можна помітити те, що Nadam за допомогою імпульсу швидше досягяє збіжності й показує кращі результати.

Після всіх порівнянь можна зробити висновок, що оптимальний метод градієнтної оптимізації залежить від самого завдання. Адже на дуже простих нейронних мережах із маленькою кількістю параметрів адаптивні методи будуть себе показувати гірше, тому варто проаналізувати складність своєї моделі, також гіперпараметри й тд. Це все дозволить вибрати дійсно ефективний метод оптимізації навчання нейронної мережі.

4.3. Дослідження методів оптимізації структури моделі

Як вже говорив раніше, окрім оптимізації функції ваг важливо оптимізовувати саму структуру моделі. Найчастіше використовуються два методи, а саме Pruning та квантування.

Спочатку розберемо перший метод.

Для дослідження цього методу використовував ResNet18, також використовую CIFAR-10, та методом градієнтної оптимізації Adam, адже як

показало попереднє дослідження він є найбільш оптимальним.

Метриками для оцінки роботи буде точність та кількість параметрів, тобто розмір мережі.

Для початку здійснимо тренування мережі й оцінимо її до Pruning. Після здійснення оцінки переходимо до виконання методу, для цього я використовував вже наявний метод в бібліотеці PyTorch.

Визначаємо параметри, за якими саме ми будемо обрізати нашу мережу, зазвичай це є ваги (лістинг 4.11.).

Лістинг 4.11.

```
prune_parameters = []
for name, module in model.named_modules():
    if isinstance(module, (nn.Conv2d, nn.Linear)):
        prune_parameters.append((module, 'weight'))

prune.global_unstructured(
    prune_parameters,
    pruning_method=prune.L1Unstructured,
    amount=0.4,
)

for module, name in prune_parameters:
    prune.remove(module, name)
```

В даному коді можна помітити, що для кожного шару ми визначаємо параметр для обрізки після чого викликається обрізка global_unstuctured, це означає, що обрізка буде для цілої мережі, із методом L1Unstructured, також передається параметр amount, який відповідає за відсоток «відрізаних» параметрів, в нас значення 40%, його можна змінювати, залежно від потрібної задачі.

Після виконання обрізки, ми видаляємо маски обрізаних ваг, адже після

обрізки утворюються маски параметрів, які будуть лиш збільшувати розмір моделі, тому вони більше не потрібні й ми їх видаляємо.

Наступним кроком ми оцінюємо модель після обрізки. Також я вирішив додати Fine Tuning, це точна настройка мережі, а саме після обрізки ми ще навчаємо мережу, в моєму випадку на 1 епосі, щоб мережа точно визначила потрібні параметри та змогла збільшити більше свою точність.

Після того знову оцінюємо модель й бачимо результат (табл. 4.1.).

Таблиця 4.1.

Оцінка Pruning

Метрика	Модель до обрізання	Після обрізання	Після тонкого налаштування
Точність (%)	90.490	89.360	89.690
Кількість параметрів	11176832	6708019	11171048

Можна побачити, що після обрізання кількість параметрів зменшилась, але, як би це дивно не звучало, але точність моделі ResNet18 майже не змінилась. Це відбулось через те, що менш важливі ваги відкинулись й це дозволило уникнути втрат точності та зменшити розмір мережі.

Після тонкого налаштування кількість параметрів майже відновилась, так само як й точність. Така оптимізації дозволить робити моделі більш компактними, що дозволить використовувати нейронні мережі на пристроях з обмеженими ресурсами.

Наступним методом було квантування.

Під час виконання цього завдання я також використовував трішки складну модель, але також, щоб краще порівняти відбулось дослідження й за допомогою ResNet18, а метод градієнтної оптимізації Adam.

Знову так само спочатку тренується звичайна модель та оцінюється після неї переходимо до квантування, для того, щоб його виконати використовував вже визначений функціонал в бібліотеці PyTorch.

Використовував саме динамічне квантування це показано на лістингу 4.12.

Лістинг 4.12.

```
quantized_model = torch.quantization.quantize_dynamic(
quantized_model,
{torch.nn.Linear},
dtype=torch.qint8)
```

В коді видно, що метод приймає декілька параметрів, а саме модель, яку необхідно квантування, далі, які саме шари потрібно квантувати, після того останнім параметром є тип для квантизованих ваг.

В нашому випадку ми квантизуємо лиш шари Linear, для згорткових треба вказати Conv2d.

Після квантування ми оцінюємо модель й все те саме ми повторюємо із ResNet.

Результат дослідження представлений в таблиці на табл. 4.2.

Таблиця 4.2.

Порівняння квантування

Модель	Точність (%)	Розмір моделі (КБ)
Базова модель	98.840	947.221
Квантизована модель	98.850	512.783
ResNet18	98.570	43729.430
Квантизований ResNet18	98.570	43715.244

На таблиці можна побачити різницю між моделлю після квантування й до. Можна помітити, що в обох варіантах точність не змінилась, або як в першому варіанти трішки збільшилась, але це дуже мізерне значення, але розмір моделі зменшився. В першому варіанті можна побачити, що мережа зменшилась майже вдвічі, що є дуже хорошим результатом, адже це дозволить використовувати її краще на мобільних пристроях. В другому варіанти не така велика різниця, адже складніша мережа вимагає трішки іншого налаштування квантування.

Можна помітити, що ці два методи є дуже ефективними для оптимізації, зазвичай для складних моделей їх використовують в парі, для набагато кращих результатів й досягнення балансу в розмірі й точності моделі. Але обидва методи

вимагають налаштування під конкретне завдання та структуру мережі.

4.4. Порівняння регуляризації нейронної мережі

Наступним етапом для кращої роботи нейронної мережі є регуляризація нейромережі, адже дуже важливо вирішувати проблему перенавчання та може оптимізувати навчання нейронної мережі.

Одним із таких методів є L1-регуляризація.

Для виконання цього порівняння використовував просту нейронну мережу та велику кількість епох, щоб зробити перенавчання й побачити як регуляризація з нею справляється.

Сам метод регуляризації можна побачити в лістингу 4.13.

Лістинг 4.13.

```
def l1_regularization(model, lambda_l1):
l1_norm = 0.0
for param in model.parameters():
    l1_norm += torch.norm(param, 1)
return lambda_l1 * l1_norm
```

В даному методі можна помітити, як вона працює, детально описав її в попередньому розділі.

Після цього модель тренується без неї та з нею й вони тестуються, результати порівняння є на рис. 4.10 – 4.13.

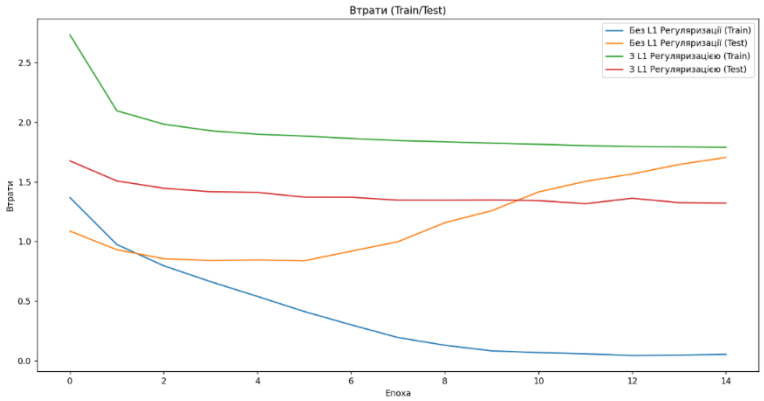

Рис. 4.10. Порівняння функції втрат з та без L1

На рис. 4.10 можна помітити, як синій графік (функція втрат на навчальних даних без регуляризації) падає до нуля, але в той час на 5 епосі функція втрат на тестових даних (оранжевий) починає йти дороги, це на практиці виглядає перенавчання.

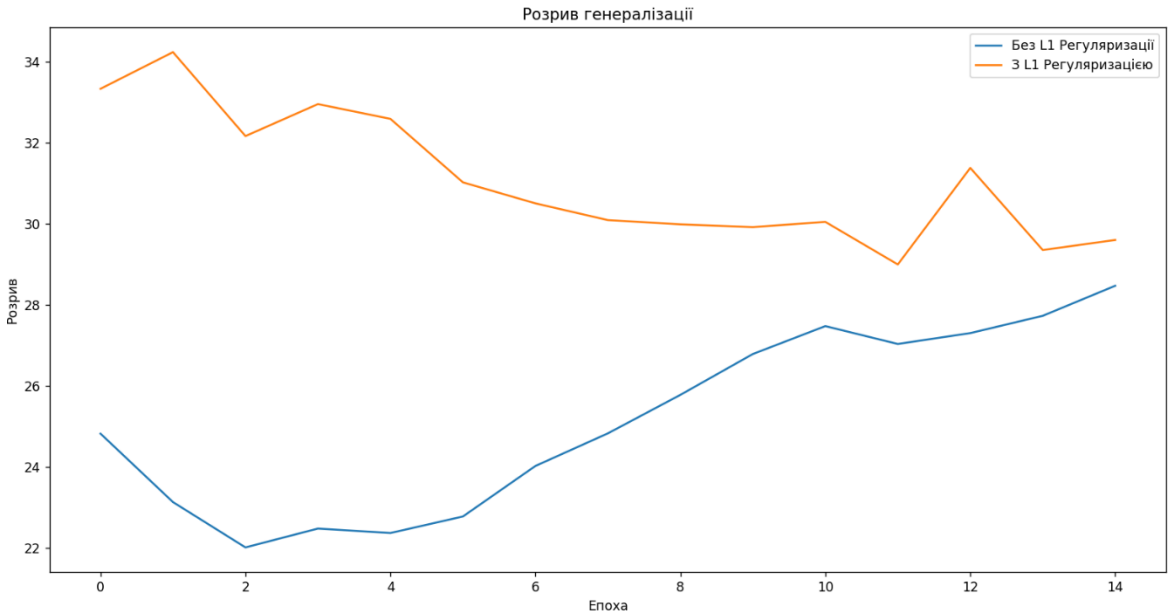

Рис. 4.11. Порівняння розриву генералізації L1

Розрив генералізації – це різниця між ефективністю методу на навчальних даних та тестових.

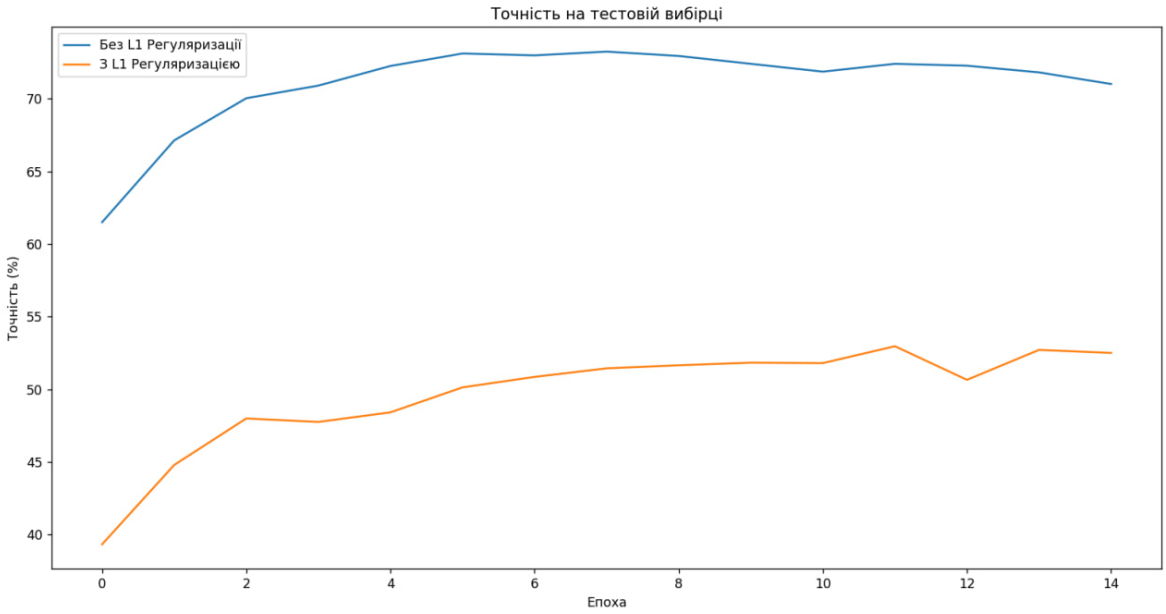

Рис. 4.12. Порівняння точності L1

Можна помітити, як на графіку рис. 4.11. розрив з L1 йде на спад, в той час як без регуляризації він починає збільшуватись через перенавчання, адже починає втрачати точність.

Але можна побачити, що точність без регуляризації є більшою, але починає йти на спад, порівняно з тим, що з регуляризацією, вона нижча, але зростає.

З цього можна зробити висновок, що на початкових етапах модель з цією регуляризацією є менш ефективною й вимагає більше часу на навчання, але в результаті навчання буде більш стабільним й ефективнішим, адже ми уникнемо помилки перенавчання.

Далі для порівняння використовую метод Dropout, який випадково відключає певні нейрони, що покращує навчання мережі.

Для цього методу використовував таку ж мережу та метод оптимізації Adam. Також спочатку тренується звичайна модель, а потім модель з використанням Dropout.

Для Dropout використовував вже визначений бібліотекою функціонал, з ймовірністю 50%, це означає, що 50% параметрів на шарах будуть відкидуватись випадково.

Після цього порівняв їх за такими ж метриками, як при L1. Графіки результатів показані на рис. 4.13. – 4.15.

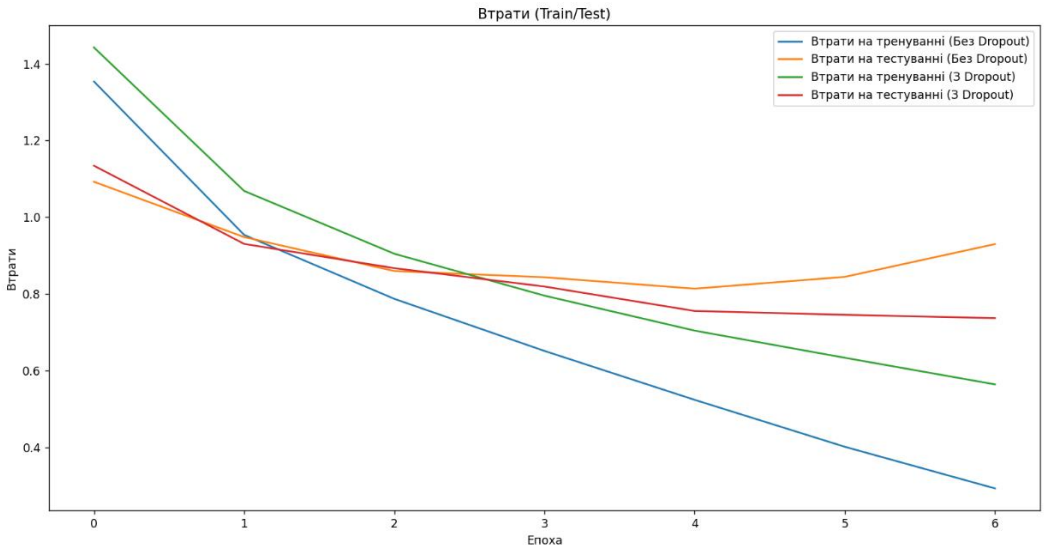

Рис. 4.13. Порівняння функції втрат Dropout

На графіку знову помітно, що функція втрат на тестуванні без регуляризації починає рости.

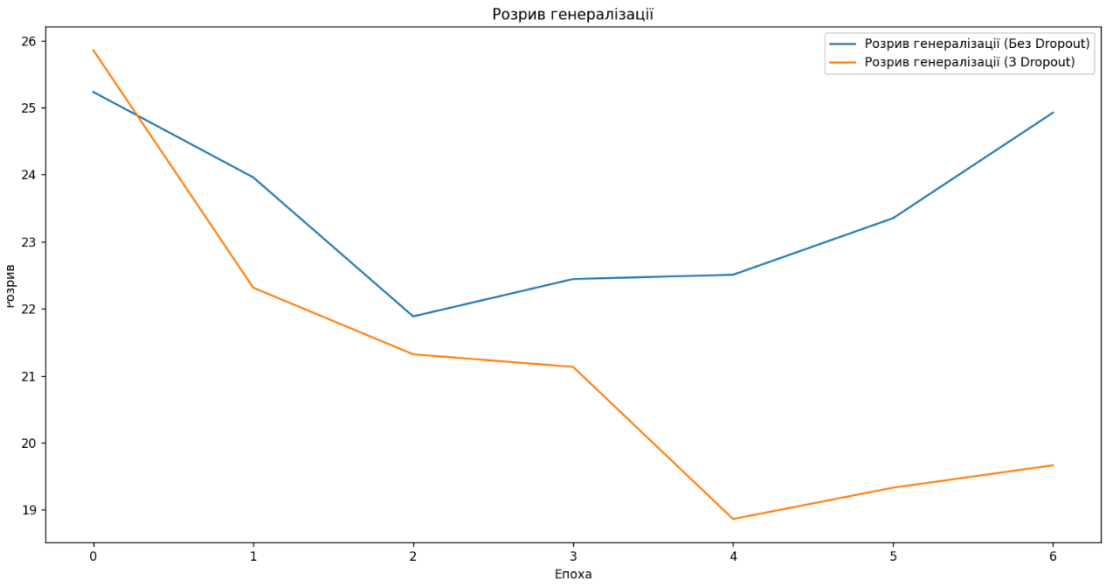

Рис. 4.14. Порівняння розриву генералізації Dropout

Таке саме можна помітити на графіку порівняння розриву генералізації (рис. 4.14.), це свідчить про явне перенавчання мережі.

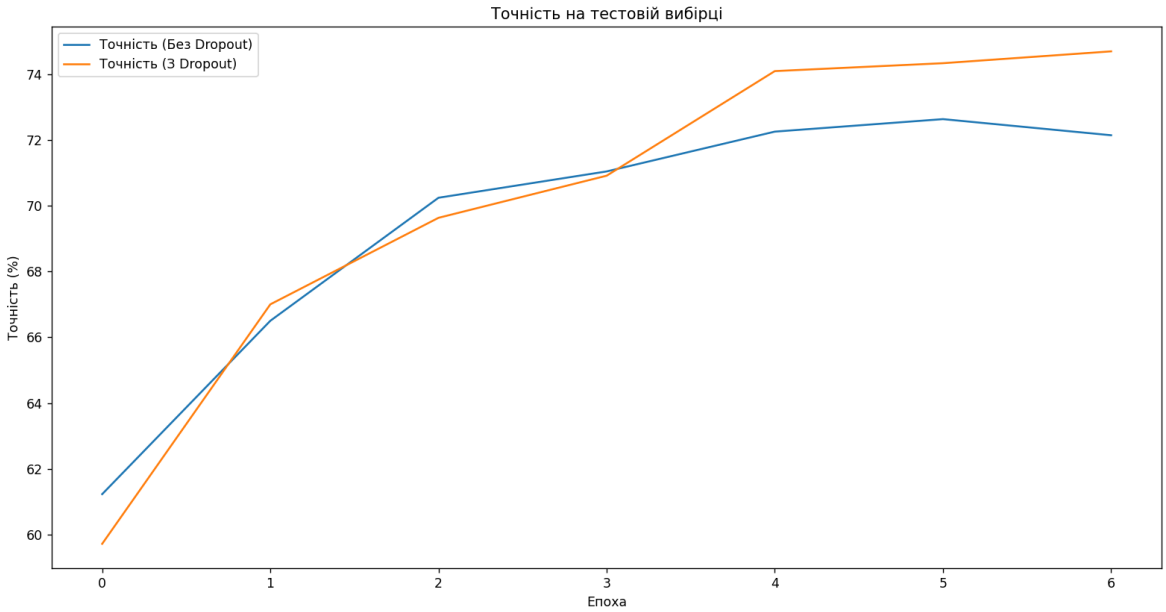

Рис. 4. 15. Порівняння точності навчання з Dropout

На рис. 4.15. можна побачити, що точність нейронної мережі з методом

регуляризації росте, в той час як без неї починає падати.

Все це показує, що метод дійсно працює, адже допомагає уникати проблеми перенавчання. Окрім того, він використовує меншу кількість параметрів, що зменшує кількість обчислення.

Метод Dropout порівняно із L1 не потребує більшого часу навчання, щоб догнати модель по точності без регуляризації. Але метод L1 краще підходить для задач, в яких використовується велика кількість ознак, й потрібно розділити неважливі та важливі. Обидва методи зменшують складність моделі, але роблять це по різному, іноді використовуються разом, особливо для дуже складних моделях, адже обоє будуть доповнювати один одного посилюючи регуляризацію.

4.5. Висновок до розділу

У даному розділі було описано підготовку до дослідження порівняння різних підходів до оптимізації нейронних мереж для задач комп’ютерного зору. Описано, які технології використовував для виконання завдання. Розглянуто різні методи оптимізації на практиці та показав способи їхньої реалізації на практиці, для порівняння.

Було проведено порівняння градієнтних методів оптимізації в умовах різної складності мережі, а саме SGD, Momentum, Nesterov, AdaGrad, RMSProp, Adam Було виявлено найбільш оптимальні варіанти для цих мереж та пояснив чому результати відрізняються й на що варто звертати увагу при виборі оптимізатора для своєї мережі.

Також проведено аналіз методів оптимізації структури нейронної мережі, для того, щоб показати як вони впливають на точність та розмір моделі, результати показали, що вони є ефективним рішенням для задач в умовах обмежених ресурсів.

Окрім цього досліджено вплив регуляризації на навчання нейронної мережі.

ВИСНОВКИ

Під час виконання даної магістерської роботи було розглянуто проблему оптимізації нейронних мереж для задач комп’ютерного зору.

Для вирішення цієї проблеми було проведено дослідження наявних методів оптимізації та порівняно їх, щоб визначити, які з них більш ефективніше справляються із задачею оптимізації нейронної моделі ResNet18.

В останньому розділі магістерської роботи провів експериментальне дослідження алгоритмів, для визначення найбільш оптимальних. Описав підготовку та пояснив вибір технологій, які використовував для виконання задачі.

У результаті дослідження визначив, що серед звичайних градієнтних методів для оптимізації найкращі результати показують себе Nesterov – для простих моделей та Adam – для складних, в нашому випадку ResNet18, адже вони показали найвищі результати точності під час навчання. Також виявив перспективи використання гібридних методів, які поєднали в собі властивості адаптивних та звичайних, на прикладі Nadam. У висновку порівняння можна сказати, що використання цих методів залежить від складності завдання та моделі.

Дослідження методів оптимізації структури показали, що методи квантування та обрізки дуже ефективно роблять модель більш компактною в розмірі шляхом зменшення формату числових значень в першому методі та обрізки неважливих ваг в другому, без великої втрати точності. В деяких випадках метод обрізки може навіть й збільшити точність, адже забере непотрібні ваги, які лиш заважають виконувати роботу моделі. Методи однаково гарно працюють й для складних та легких завдань, але якщо необхідна висока точність на дрібних деталях, то вони можуть негативно вплинути. Також ці методи можна використовувати разом, особливо часто це роблять для дуже глибоких мереж.

Регуляризація допомагає уникати проблеми перенавчання, що дозволяє стабільніше навчати модель. Результати порівняння алгоритму регуляризації L1 показали, що мережа повинна витратити більше часу на тренування, але це допоможе досягти кращих результатів. Відмінно від попереднього методу Dropout показав, що йому не потрібно більше часу на навчання, але він також справляється із виконанням завдання. Через це його частіше використовують, але також їх використовують разом, адже вони дозволяють доповнити одне одного.

Загалом ці всі різні підходи зазвичай використовуються разом, для досягнення більшої продуктивності.

Перспективи розвитку методів оптимізації для задач CV є дуже великими, адже ця технологія все активніше починає використовуватись й створення більш компактних, гнучких моделей допоможе лиш збільшити кількість сфер її використання. Вони будуть все більше використовуватись для мобільних пристроїв та розширять функціонал вже відомих нам нейронних моделей. Також вони сильно впливають на генеративні нейронні мережі, адже допоможуть генерувати більш реалістичні відео та фото з меншою кількістю ресурсів. Зменшення потреб в обчислювальних ресурсів без втрати точності дасть можливість стати цій технології більш доступною для сфер, які вимагають високої точності, але не мають обчислювальних потужностей. Розвиток сучасних технологій дуже сильно впливає на дане завдання, адже дослідники знаходять нові рішення, які раніше не були доступними.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mijwel, M. M. (2015). History of Artificial Intelligence Yapay Zekânın T arihi. Computer Science,(April 2015), 3-4.
2. Khan, A. I., & Al-Habsi, S. (2020). Machine learning in computer vision. Procedia Computer Science, 167, 1444-1451.
3. Gao, J., Yang, Y., Lin, P., & Park, D. S. (2018). Computer vision in healthcare applications. Journal of healthcare engineering, 2018.
4. Zhou, L., Zhang, L., & Konz, N. (2022). Computer vision techniques in manufacturing. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 53(1), 105-117.
5. Janai, J., Güney, F., Behl, A., & Geiger, A. (2020). Computer vision for autonomous vehicles: Problems, datasets and state of the art. Foundations and Trends® in Computer Graphics and Vision, 12(1–3), 1-308.
6. Tan, M., & Le, Q. (2019, May). Efficientnet: Rethinking model scaling for convolutional neural networks. In International conference on machine learning (pp. 6105-6114). PMLR.
7. Gallant, S. I. (1990). Perceptron-based learning algorithms. IEEE Transactions on neural networks, 1(2), 179-191.
8. Ding, L., & Goshtasby, A. (2001). On the Canny edge detector. Pattern recognition, 34(3), 721-725.
9. Chaudhari, M. N., Deshmukh, M., Ramrakhiani, G., & Parvatikar, R. (2018, August). Face detection using viola jones algorithm and neural networks. In 2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA) (pp. 1-6). IEEE.
10. Voulodimos, A., Doulamis, N., Doulamis, A., & Protopapadakis, E. (2018). Deep learning for computer vision: A brief review. Computational intelligence and neuroscience, 2018(1), 7068349.
11. Zou, Z., Chen, K., Shi, Z., Guo, Y., & Ye, J. (2023). Object detection in 20 years:

A survey. Proceedings of the IEEE, 111(3), 257-276.

12. Khan, A. A., Laghari, A. A., & Awan, S. A. (2021). Machine learning in computer vision: a review. *EAI Endorsed Transactions on Scalable Information Systems*, *8*(32), e4-e4.
13. Garcia-Garcia, A., Orts-Escolano, S., Oprea, S., Villena-Martinez, V., Martinez-Gonzalez, P., & Garcia-Rodriguez, J. (2018). A survey on deep learning techniques for image and video semantic segmentation. Applied Soft Computing, 70, 41-65.
14. Zheng, L., Yang, Y., & Tian, Q. (2017). SIFT meets CNN: A decade survey of instance retrieval. IEEE transactions on pattern analysis and machine intelligence, 40(5), 1224-1244.
15. Ahamed, H., Alam, I., & Islam, M. M. (2018, November). HOG-CNN based real time face recognition. In 2018 International conference on advancement in electrical and electronic engineering (ICAEEE) (pp. 1-4). IEEE.
16. Dongare, A. D., Kharde, R. R., & Kachare, A. D. (2012). Introduction to artificial neural network. International Journal of Engineering and Innovative Technology (IJEIT), 2(1), 189-194.
17. Thoma, M. (2017). Analysis and optimization of convolutional neural network architectures. arXiv preprint arXiv:1707.09725.
18. Bhatt, D., Patel, C., Talsania, H., Patel, J., Vaghela, R., Pandya, S., ... & Ghayvat, H. (2021). CNN variants for computer vision: History, architecture, application, challenges and future scope. Electronics, 10(20), 2470.
19. Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. Journal of big data, 6(1), 1-48.
20. Kim, J., Chang, S., & Kwak, N. (2021). PQK: model compression via pruning, quantization, and knowledge distillation. arXiv preprint arXiv:2106.14681.
21. Aghli, N., & Ribeiro, E. (2021). Combining weight pruning and knowledge distillation for cnn compression. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 3191-3198).
22. Santos, C. F. G. D., & Papa, J. P. (2022). Avoiding overfitting: A survey on

regularization methods for convolutional neural networks. ACM Computing Surveys (CSUR), 54(10s), 1-25.

23. Vasicek, D. (2019). Artificial intelligence and machine learning: Practical aspects of overfitting and regularization. Information Services & Use, 39(4), 281-289.
24. Lepetit, V. (2008, July). On computer vision for augmented reality. In 2008 international symposium on ubiquitous virtual reality (pp. 13-16). IEEE.
25. Rasamoelina, A. D., Adjailia, F., & Sinčák, P. (2020, January). A review of activation function for artificial neural network. In 2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI) (pp. 281-286). IEEE.
26. Swapna, M., Sharma, Y. K., & Prasadh, B. M. G. (2020). CNN Architectures: Alex Net, Le Net, VGG, Google Net, Res Net. Int. J. Recent Technol. Eng, 8(6), 953-960.
27. Diwan, T., Anirudh, G., & Tembhurne, J. V. (2023). Object detection using YOLO: Challenges, architectural successors, datasets and applications. multimedia Tools and Applications, 82(6), 9243-9275.
28. Jamil, S., Jalil Piran, M., & Kwon, O. J. (2023). A comprehensive survey of transformers for computer vision. Drones, 7(5), 287.
29. Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., ... & Guo, B. (2021). Swin transformer: Hierarchical vision transformer using shifted windows. In Proceedings of the IEEE/CVF international conference on computer vision (pp. 10012-10022).
30. Guo, J., Han, K., Wu, H., Tang, Y., Chen, X., Wang, Y., & Xu, C. (2022). Cmt: Convolutional neural networks meet vision transformers. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 12175-12185).
31. Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., & Zagoruyko, S. (2020, August). End-to-end object detection with transformers. In European conference on computer vision (pp. 213-229). Cham: Springer International Publishing.

32. Sun, R. Y. (2020). Optimization for deep learning: An overview. Journal of the Operations Research Society of China, 8(2), 249-294.
33. Soydaner, D. (2020). A comparison of optimization algorithms for deep learning. International Journal of Pattern Recognition and Artificial Intelligence, 34(13), 2052013.
34. Dogo, E. M., Afolabi, O. J., Nwulu, N. I., Twala, B., & Aigbavboa, C. O. (2018, December). A comparative analysis of gradient descent-based optimization algorithms on convolutional neural networks. In 2018 international conference on computational techniques, electronics and mechanical systems (CTEMS) (pp. 92-99). IEEE.
35. Young, S. R., Rose, D. C., Karnowski, T. P., Lim, S. H., & Patton, R. M. (2015, November). Optimizing deep learning hyper-parameters through an evolutionary algorithm. In Proceedings of the workshop on machine learning in high-performance computing environments (pp. 1-5).
36. He, X., Zhao, K., & Chu, X. (2021). AutoML: A survey of the state-of-the-art. Knowledge-based systems, 212, 106622.
37. Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., ... & De Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. Advances in neural information processing systems, 29.
38. Duda, J. (2019). SGD momentum optimizer with step estimation by online parabola model. arXiv preprint arXiv:1907.07063.
39. Qolomany, B., Maabreh, M., Al-Fuqaha, A., Gupta, A., & Benhaddou, D. (2017, June). Parameters optimization of deep learning models using particle swarm optimization. In 2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC) (pp. 1285-1290). IEEE
40. Brooks, S. P., & Morgan, B. J. (1995). Optimization using simulated annealing. Journal of the Royal Statistical Society Series D: The Statistician, 44(2), 241-257.
41. Wu, J., Poloczek, M., Wilson, A. G., & Frazier, P. (2017). Bayesian optimization with gradients. Advances in neural information processing systems, 30.

ДОДАТКИ

Додаток А

Приклад фрагменту коду реалізації порівняння методів градієнтної оптимізації

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import time
import matplotlib.pyplot as plt

device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(32 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)
    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, 2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, 2)
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

train_dataset = torchvision.datasets.CIFAR10(root='./data',
train=True, download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=64, shuffle=True)

test_dataset = torchvision.datasets.CIFAR10(root='./data',
train=False, download=True, transform=transform)
test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=64, shuffle=False)

class SGD:
    def __init__(self, params, lr=0.001):
        self.lr = lr
```

```
        self.params = list(params)

    def step(self):
        with torch.no_grad():
            for param in self.params:
                if param.grad is not None:
                    param -= self.lr * param.grad
    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()

class Momentum:
    def __init__(self, params, lr=0.001, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.params = list(params)
        self.velocity = [torch.zeros_like(p) for p in self.params]
    def step(self):
        with torch.no_grad():
            for i, param in enumerate(self.params):
                if param.grad is not None:
                    self.velocity[i] = self.momentum *
self.velocity[i] - self.lr * param.grad
                    param += self.velocity[i]
    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()

class Nesterov:
    def __init__(self, params, lr=0.001, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.params = list(params)
        self.velocity = [torch.zeros_like(p) for p in self.params]
    def step(self):
        with torch.no_grad():
            for i, param in enumerate(self.params):
                if param.grad is not None:
                    prev_velocity = self.velocity[i].clone()
                    self.velocity[i] = self.momentum *
self.velocity[i] - self.lr * param.grad
                    param += -self.momentum * prev_velocity + (1 +
self.momentum) * self.velocity[i]
    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()

class AdaGrad:
```

```
    def __init__(self, params, lr=0.01, eps=1e-8):
        self.lr = lr
        self.eps = eps
        self.params = list(params)
        self.sum_of_squares = [torch.zeros_like(p) for p in
self.params]
    def step(self):
        with torch.no_grad():
            for i, param in enumerate(self.params):
                if param.grad is not None:
                    self.sum_of_squares[i] += param.grad ** 2
                    param -= self.lr * param.grad /
(self.sum_of_squares[i].sqrt() + self.eps)
    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()

class RMSProp:
    def __init__(self, params, lr=0.001, alpha=0.99, eps=1e-8):
        self.lr = lr
        self.alpha = alpha
        self.eps = eps
        self.params = list(params)
        self.mean_square = [torch.zeros_like(p) for p in
self.params]
    def step(self):
        with torch.no_grad():
            for i, param in enumerate(self.params):
                if param.grad is not None:
                    self.mean_square[i] = self.alpha *
self.mean_square[i] + (1 - self.alpha) * param.grad ** 2
                    param -= self.lr * param.grad /
(self.mean_square[i].sqrt() + self.eps)
    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()

class Adam:
    def __init__(self, params, lr=0.001, beta1=0.9, beta2=0.999,
eps=1e-8):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.eps = eps
        self.params = list(params)
        self.m = [torch.zeros_like(p) for p in self.params]
        self.v = [torch.zeros_like(p) for p in self.params]
```

Продовження додатку А

```
        self.t = 0

    def step(self):
        with torch.no_grad():
            self.t += 1
            for i, param in enumerate(self.params):
                if param.grad is not None:
                    self.m[i] = self.beta1 * self.m[i] + (1 -
self.beta1) * param.grad
                    self.v[i] = self.beta2 * self.v[i] + (1 -
self.beta2) * (param.grad ** 2)
                    m_hat = self.m[i] / (1 - self.beta1 ** self.t)
                    v_hat = self.v[i] / (1 - self.beta2 ** self.t)
                    param -= self.lr * m_hat / (v_hat.sqrt() +
self.eps)
    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()

def evaluate_model(model):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    return accuracy

def train_model(optimizer_class, model, criterion, lr, epochs=7,
**kwargs):
    model.train()
    optimizer = optimizer_class(model.parameters(), lr=lr,
**kwargs)
    loss_history_iter = []
    loss_history_epoch = []
    accuracy_history = []
    iteration_counter = 0
    start_time = time.time()

    for epoch in range(epochs):
        running_loss = 0.0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
```

```
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            if (iteration_counter + 1) % 100 == 0:
                print(f"{optimizer_class.__name__} - Epoch [{epoch
+ 1}/{epochs}], Iteration [{iteration_counter + 1}], Loss:
{loss.item():.4f}")
                loss_history_iter.append(loss.item())
            iteration_counter += 1
        loss_history_epoch.append(running_loss /
len(train_loader))
        print(f"{optimizer_class.__name__} - Epoch [{epoch +
1}/{epochs}], Loss: {running_loss / len(train_loader):.4f}")
        accuracy = evaluate_model(model)
        accuracy_history.append(accuracy)
    end_time = time.time()
    training_time = end_time - start_time
    print(f"{optimizer_class.__name__} - Training Time:
{training_time:.2f} seconds")
    return loss_history_iter, loss_history_epoch,
accuracy_history, accuracy

criterion = nn.CrossEntropyLoss()
optimizers = [SGD, Momentum, Nesterov, AdaGrad, RMSProp, Adam]
results = {}
loss_histories_iter = {}
loss_histories_epoch = {}
accuracy_histories = {}

for optimizer_class in optimizers:
    print(f"\nTraining with {optimizer_class.__name__}")
    model = SimpleCNN().to(device)
    lr = 0.01
    loss_iter, loss_epoch, acc_history,acc =
train_model(optimizer_class, model, criterion, lr)
    accuracy = evaluate_model(model)
    print(f"Accuracy: {acc:.2f}%")
    results[optimizer_class.__name__] = accuracy
    loss_histories_iter[optimizer_class.__name__] = loss_iter
    loss_histories_epoch[optimizer_class.__name__] = loss_epoch
    accuracy_histories[optimizer_class.__name__] = acc_history

plt.figure(figsize=(10, 6))
for optimizer_name, losses in loss_histories_iter.items():
    plt.plot(losses, label=optimizer_name)
plt.title("Loss Curves for Different Optimizers (by Iteration)")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend()
plt.grid()
plt.show()
```

Продовження додатку А

```
plt.figure(figsize=(10, 6))
for optimizer_name, losses in loss_histories_epoch.items():
    plt.plot(range(1, len(losses) + 1), losses,
label=optimizer_name)
plt.title("Loss Curves for Different Optimizers (by Epoch)")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.grid()
plt.show()

fig, ax = plt.subplots(figsize=(8, 6))
ax.axis('tight')
ax.axis('off')

num_epochs = len(next(iter(accuracy_histories.values())))
data = [[f"Epoch {epoch + 1}"] +
[f"{accuracy_histories[opt][epoch]:.2f}" for opt in
results.keys()] for epoch in range(num_epochs)]
column_labels = ["Epoch"] + list(results.keys())
ax.table(cellText=data, colLabels=column_labels, loc='center',
cellLoc='center')
plt.title("Accuracy Table")
plt.show()

plt.figure(figsize=(10, 6))
for optimizer_name, accuracies in accuracy_histories.items():
    plt.plot(range(1, len(accuracies) + 1), accuracies,
label=optimizer_name)
plt.title("Accuracy Curves for Different Optimizers (by Epoch)")
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.legend()
plt.grid()
plt.show()
```