МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

Факультет мехатроніки та комп'ютерних технологій

Кафедра комп'ютерних наук

Кваліфікаційна робота

на тему

«Розробка програмного забезпечення для покращення якості зображень з використанням нейронних мереж»

Рівень вищої освіти другий (магістерський)

Спеціальності 122 Комп'ютерні науки

Освітня програма Комп'ютерні науки

Виконав: студент групи МгІТ-1-22

Артем Рубан.

Науковий керівник:
к.т.н., доц. Володимир Яхно

Рецензент:

д.т.н., проф .Віктор Чупринка

Київ 2023

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
Факультет мехатроніки та комп'ютерних технологій
Кафедра комп'ютерних наук

Рівень вищої освіти другий (магістерський)
Спеціальність 122 Комп'ютерні науки
Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук та технологій
_____ В.Ю.Щербань
«_____» _____ 2023р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Рубану Артему Олександровичу

1. Тема роботи *Розробка програмного забезпечення для покращення якості зображень з використанням нейронних мереж,*
науковий керівник роботи Яхно Володимир Михайлович, к.т.н., доцент, затверджені наказом КНУТД від " 12 " вересня 2023 року № 210 -уч
2. Строк подання студентом дипломної роботи 10.11.2023 р.
3. Вихідні дані до роботи Розробки кафедри комп'ютерних наук, рекомендована література, додатки.
4. Зміст дипломної бакалаврської роботи: Вступ, Розділ 1. Опис та дослідження нейронних мереж для вирішення проблем покращення якості зображення; Розділ 2. Проєктування та огляд програмної області; Розділ 3. Практична реалізація програмного забезпечення для вирішення проблеми покращення якості зображення; Додатки - програмні коди.

5. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Вступ	Володимир ЯХНО к.т.н., доц.		
Розділ 1	Володимир ЯХНО к.т.н., доц.		
Розділ 2	Володимир ЯХНО к.т.н., доц.		
Розділ 3	Володимир ЯХНО к.т.н., доц.		

6. Дата видачі завдання __.__.2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	15.08.2023	
2	Розділ 1 Опис та дослідження нейронних мереж для вирішення проблем покращення якості зображення.	02.09.2023	
3	Розділ 2 Проєктування та огляд програмної області	16.09.2023	
4	Розділ 3 Практична реалізація програмного забезпечення для вирішення проблеми покращення якості зображення	12.10.2023	
5	Висновки	14.10.2023	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	19.10.2023	
7	Здача кваліфікаційної роботи на кафедру для рецензування (за 14 днів до захисту)		
8	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)		
9	Подання кваліфікаційної роботи у відділ магістратури для перевірки виконання додатку до індивідуального навчального плану (за 10 днів до захисту)		
10	Подання кваліфікаційної роботи на затвердження завідувачу кафедри (з 7 днів до захисту)		

Студент _____ Артем РУБАН

Науковий керівник роботи _____ Володимир ЯХНО

АНОТАЦІЯ

РУБАН А.О. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ПОКРАЩЕННЯ ЯКОСТІ ЗОБРАЖЕННЯ З ВИКОРИСТАННЯМ НЕЙРОННИХ МЕРЕЖ

Кваліфікаційна робота за спеціальністю 122 - «Комп’ютерні науки».

– Київський національний університет технологій та дизайну, Київ, 2023 рік.

Дана кваліфікаційна робота націлена на дослідження, розробку та реалізацію програмного забезпечення, заснованого на генеративно-змагальних мережах (GAN), для підвищення якості зображень. Метою є створення ефективного алгоритму покращення роздільної здатності зображень з використанням передових методів глибокого навчання, а також практичне впровадження даної системи в рамках комп'ютерного зору.

Проведено ретельне дослідженням структури нейронних мереж, їх формування та розвитку в контексті підвищення якості зображень. Також проведено теоретичний аналіз комп'ютерного зору та огляд переваг використання фреймворку PyTorch.

Проведено огляд принципів роботи згорткових мереж (CNN), математична модель генеративно-змагальної мережі (GAN), а також методи вирахування похибки. Особливу увагу приділено функцій генератора та дискримінатора.

Представлено практичну реалізацію програмного забезпечення для покращення якості зображень з використанням генеративно-змагальних мереж. Розглянуто бібліотеки, вибрано набір даних, описано класи генератора та дискримінатора, а також наведено функцію навчання моделі. Розроблено інтерфейс програмного забезпечення із прикладами використання.

Ключові слова: *Штучний інтелект, комп’ютерний зір, CNN, GAN, PyTorch, Python.*

ANNOTATION

Ruban A.O. Software development for improving image quality using neural networks

Master's thesis in specialty 122 - "Computer Science". - Kyiv National University of Technology and Design, Kyiv, 2023.

This master's thesis is aimed at the research, development and implementation of software based on generative adversarial networks (GAN) to improve the quality of images. The goal is to create an effective algorithm for improving the resolution of images using advanced deep learning methods, as well as practical implementation of this system in the framework of computer vision.

A thorough study of the structure of neural networks, their formation and development in the context of image quality improvement was carried out. A theoretical analysis of computer vision and an overview of the advantages of using the PyTorch framework were also carried out.

An overview of the principles of convolutional networks (CNN), the mathematical model of a generative-competitive network (GAN), as well as error calculation methods was conducted. Special attention is paid to the generator and discriminator functions.

The practical implementation of software for improving the quality of images using generative-competitive networks is presented. The libraries are reviewed, the dataset is selected, the generator and discriminator classes are described, and the model training function is given. A software interface with examples of use has been developed.

Keywords: *Artificial intelligence, computer vision, CNN, GAN, PyTorch, Python.*

ЗМІСТ

ВСТУП

В епоху, яка характеризується експоненційним зростанням цифрових зображень і ненаситним попитом на високоякісний візуальний вміст у різних сферах, пошуки покращення якості зображення стають дедалі важливішими. Швидке поширення смартфонів, цифрових камер і різноманітних візуальних даних в Інтернеті вимагають інноваційних рішень для вирішення постійної проблеми, а саме покращення якості зображення. Зважаючи на цей контекст, застосування нейронних мереж стало новаторським і впливовим підходом у сфері покращення зображення та комп'ютерного зору.

Поява нейронних мереж, зокрема архітектур глибокого навчання, революціонізувала сферу штучного інтелекту та спричинила ґрунтовні зміни в різних областях. Нейронні мережі, натхненні взаємопов'язаними нейронами людського мозку, продемонстрували надзвичайні можливості в розумінні та обробці складних даних, що робить їх безцінними інструментами для вирішення складних і багатогранних проблем. У контексті покращення якості зображення нейронні мережі довели свою спроможність підвищити чіткість зображення, зменшити шум і відновлювати деталі зображення, чого раніше було важко досягти за допомогою звичайних методів обробки. Здатність нейронних мереж навчатися на основі даних, адаптуватися до різноманітних типів зображень і забезпечувати автоматизовані контекстно-залежні вдосконалення пропонує багатообіцяючий шлях для задоволення зростаючих потреб у високоякісному візуальному вмісті в різних секторах, починаючи від охорони здоров'я та розваг до спостереження та автономних систем.

Комп'ютерний зір, підгалузь штучного інтелекту, став свідком зміни парадигми з інтеграцією нейронних мереж. Поєднання комп'ютерного зору та методів глибокого навчання відкрило безліч можливостей. Системи комп'ютерного зору, уповноважені нейронними мережами, можуть не тільки виявляти об'єкти та розпізнавати шаблони на зображеннях, але й активно покращувати якість зображення. Ці системи можуть покращувати зображення з

низькою роздільною здатністю, усувати недоліки та реконструювати відсутню інформацію, таким чином розсуваючи межі того, чого можна досягти у сфері обробки візуальних даних. Комп’ютерне бачення, підкріплене нейронними мережами, має змінити визначення галузей, дозволяючи аналізувати, інтерпретувати та маніпулювати візуальними даними з безпрецедентною точністю та ефективністю.

Основною метою цієї кваліфікаційної роботи є використання можливостей нейронних мереж для покращення якості зображення, тим самим сприяючи розвитку комп’ютерного зору. Розробляючи та впроваджуючи програмні рішення, які використовують нейронні мережі, це дослідження спрямоване на вирішення проблем, пов’язаних із покращенням якості зображення, і досліджує потенціал для автоматизації та оптимізації процесу покращення зображення. Також завдяки ретельним експериментам і аналізу це дослідження має на меті запропонувати розуміння ефективності підходів на основі нейронних мереж для покращення зображення та їх застосування до сценаріїв реального світу.

РОЗДІЛ 1. ОПИС ТА ДОСЛІДЖЕННЯ НЕЙРОННИХ МЕРЕЖ ДЛЯ ВИРІШЕННЯ ПРОБЛЕМИ ПОКРАЩЕННЯ ЯКОСТІ ЗОБРАЖЕННЯ

1.1 Структура нейронів у нейронній мережі

Нейрон у нейронній мережі схожий на крихітну одиницю прийняття рішень, ґрунтовану на роботі людського мозком. Він приймає вхідні дані, обробляє їх і створює вихідні дані, і це фундаментальний будівельний блок глибокого навчання. Давайте розберемо його структуру на прикладах із повсякденного життя [1-8].

Розглянемо схему та будову звичайного штучного нейрону (рис1.1):

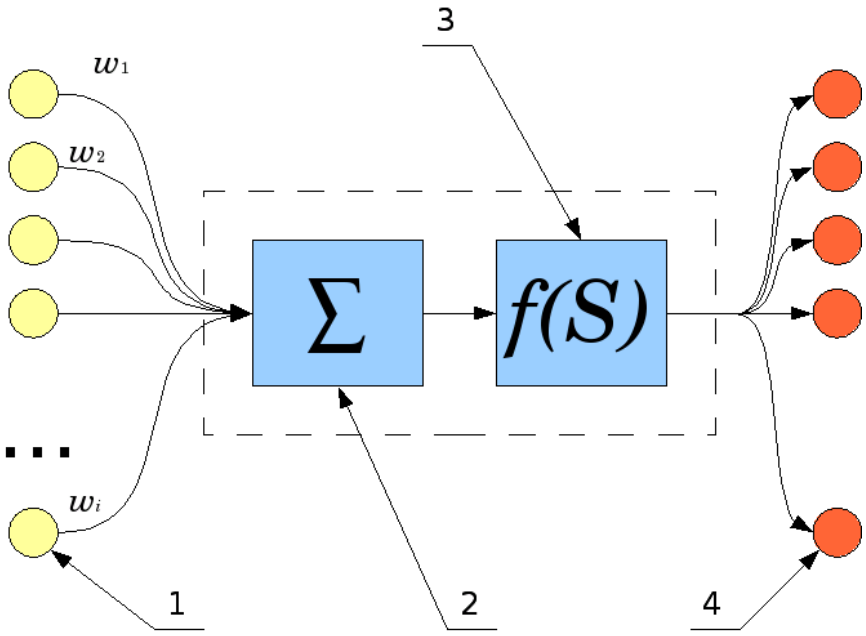

Рис 1.1 Схема штучного нейрону.

1. **Входи (дендрити):**

Нейрони отримують численні вхідні дані, які схожі на сенсорну інформацію, яку збирає наше тіло. Наприклад, у нейронній мережі розпізнавання зображень ці вхідні дані можуть бути рівнями яскравості окремих пікселів. Кожен вхід має вагу, пов’язану з ним, яка представляє його важливість у процесі прийняття рішень. На зображенні яскравіші пікселі

можуть мати більшу вагу, оскільки містять більше інформації про вміст зображення.

2. **Обробка (тіло клітини):**

Тіло клітини нейрона обробляє ці вхідні дані, виконуючи обчислення. Він обчислює зважену суму вхідних даних, подібно до того, як наш мозок обробляє інформацію від органів чуття. Наприклад, під час обробки зображень нейрон може ідентифікувати краї, порівнюючи значення пікселів.

3. **Функція активації:**

Функція активації формує вихід нейрона. Наприклад, функція активації Rectified Linear Unit (ReLU) зазвичай використовується в глибокому навчанні. Якщо оброблене значення від'ємне, ReLU встановлює його на нуль, фактично кажучи: «Я не відповідаю на це». Але якщо він позитивний, це дозволяє значенню пройти без змін, вказуючи, що нейрон активний. Ця нелінійність дозволяє нейрону моделювати складні зв'язки в даних.

4. Вихід (аксон):

Вихід нейрона є результатом його обробки. Його часто передають через функцію активації, як-от сигмоїдна функція, яка звужує вихідні дані в певний діапазон. Цей кінцевий вихід є рішенням або відповіддю нейрона на вхідні дані. Наприклад, у фільтрі електронної пошти спаму нейрон може вивести значення, близьке до 1, якщо він вважає, що електронний лист спам, і близьке до 0, якщо це не так.

Ваги та зміщення(w_i):

Нейрони коригують свої рішення, використовуючи вагові коефіцієнти та зсув, які діють як «ручки гучності» для вхідних даних. Це схоже на надання більшої ваги певним факторам під час прийняття рішення. Наприклад, якщо ви вирішуєте, чи піти на пікнік, важливість погоди (weight) може бути більшою, тоді як інші фактори, як-от доступність (bias), також можуть вплинути на ваше

рішення. Налаштування цих ваг і зміщення дозволяє нейрону адаптувати свої рішення, навчаючись на даних з часом.

З'єднання (синапси):

Нейрони з’єднані між собою через синапси, які схожі на зв’язки між людьми в соціальній мережі. Ці зв’язки забезпечують перетікання інформації між нейронами. Сила цих зв’язків показує, наскільки один нейрон впливає на інший. У разі системи розпізнавання, «мовленнєвий» нейрон, який розпізнає фонеми, може мати міцні зв’язки з нейронами, які розпізнають певні слова, що вказує на те, що фонеми є будівельними блоками слів.

У нейронній мережі численні взаємопов’язані нейрони працюють в унісон для вирішення складних завдань. Кожен нейрон обробляє вхідні дані, вивчає дані, регулюючи свої ваги та зміщення, і співпрацює з іншими нейронами для прийняття складних рішень. Саме колективний інтелект цих нейронів, схожий на взаємодію клітин людського мозку, дозволяє нейронним мережам досягти успіху в різних програмах, від розпізнавання зображень до розуміння природної мови тощо.

1.2 Формування та розвиток нейронних мереж для покращення якості зображення

Нейронні мережі отримали широке визнання як потужні інструменти у сфері покращення якості зображення. Їхній шлях від задуму до того, як стати інструментом у цій галузі, відзначений ключовими розробками та постійним пошуком кращих рішень для покращення зображення.

Основу нейронних мереж для покращення якості зображення було закладено з появою штучних нейронних мереж (ШНМ) у 1940-х роках, натхненних структурою людського мозку. Проте знадобилося кілька десятиліть, щоб ШНМ перетворилися на глибокі нейронні мережі (Deep Neural

Networks, або DNN), які характеризуються кількома шарами взаємопов’язаних нейронів. Ця глибина виявилася необхідною для роботи зі складними даними зображень.

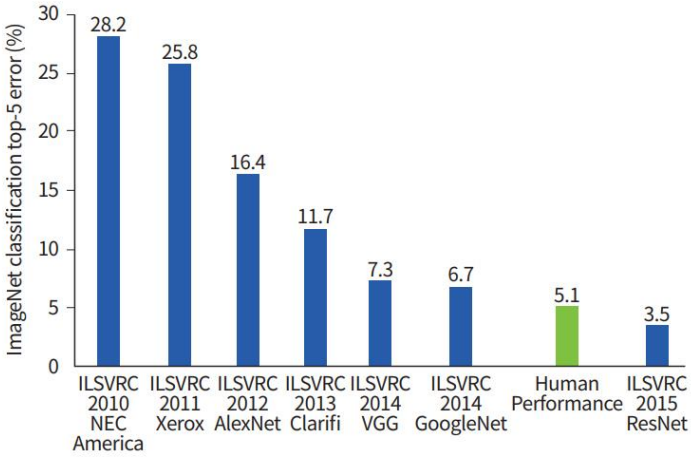

Рис 1.2.1 Графік відсотка помилки серед переможців змагання ImageNet по вирішенню проблеми покращення якості зображення.

Одним із ключових моментів у формуванні нейронних мереж для покращення зображення стала розробка згорткових нейронних мереж (Convolutional Neural Networks, або CNN)(рис 1.2.2). CNN, як-от відомі AlexNet і VGGNet[11-12,., представили згорткові шари, які можуть автоматично вивчати та виявляти особливі характеристики на зображенні. Для прикладу розглянемо процес покращення якості зображення з низькою роздільною здатністю. CNN може навчитися визначати та покращувати важливі деталі зображення, в результаті чого зображення стає чіткішим і привабливішим.

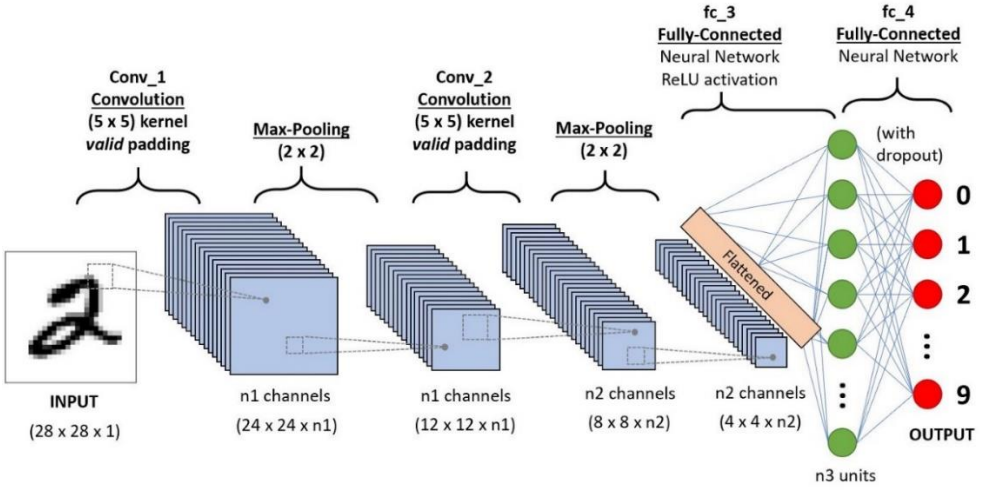

Рис 1.2.2 Схема методу роботи згорткових нейронних мереж.

Поява Generative Adversarial Networks (GAN) стала значним стрибком в еволюції нейронних мереж для покращення якості зображення. GAN складаються з двох нейронних мереж, генератора та дискримінатора, які беруть участь у змагальному процесі навчання. Цю концепцію, використану в GAN, можна порівняти з фальсифікатором, який намагається створити фальшиву валюту, а детектив намагається ідентифікувати підробку. У контексті якості зображення мережа генератора генерує покращені зображення, а мережа дискримінатора відрізняє їх від реальних (рис 1.2.3). З часом цей змагальний процес призводить до створення високоякісних зображень, які важко відрізнити від оригіналу. GAN виявилися дуже ефективними в таких завданнях, як надвисока роздільна здатність зображень, усунення шумів і передача стилів.

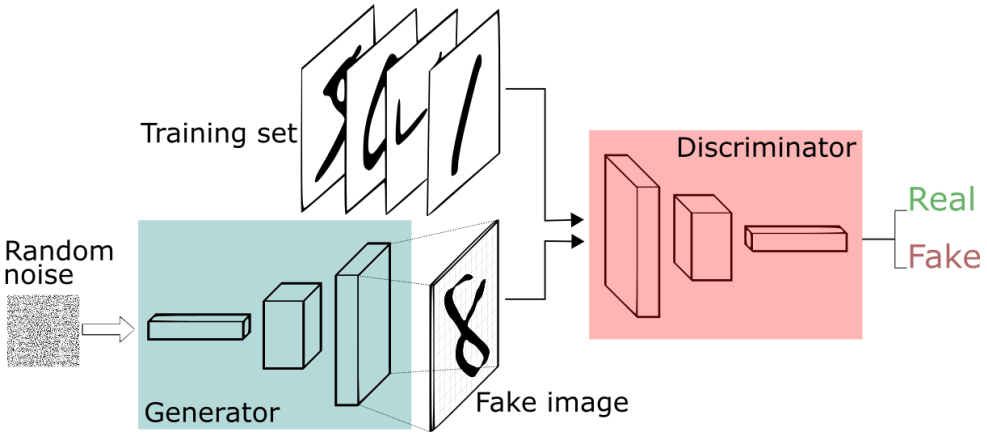

Рис 1.2.3 Схема методу роботи генеративно-змагальних мереж.

Важливим кроком у розвитку нейронних мереж для покращення якості зображення є навчання перенесення. Ця концепція передбачає повторне використання попередньо навчених моделей нейронних мереж, їх тонке налаштування для конкретних завдань покращення зображення. Це схоже на те, як досвідчений художник починає картину, а потім дозволяє іншому художнику закінчити її. Попередньо навчена модель дає цінні знання про особливості зображення, а точне налаштування адаптує ці знання для поточного завдання. Трансферне навчання значно прискорює розробку рішень для покращення зображення, використовуючи мудрість, отриману з різноманітних наборів даних зображень.

Автоенкодери, тип архітектури нейронної мережі, знайшли своє місце в наборі інструментів для покращення якості зображення. Автокодер стискає вхідне зображення в компактне представлення (кодування), а потім реконструює зображення з цього кодування (рис 1.2.4). Цей процес заохочує мережу навчитися зберігати важливі деталі, одночасно зменшуючи шум і артефакти. Автоенкодери особливо корисні в таких програмах, як усунення шумів і малювання зображень. Уявіть собі, що стару подряпану фотографію потрібно реставрувати, щоб розкрити її первісну красу – саме тут краще всього себе показують автоенкодери.

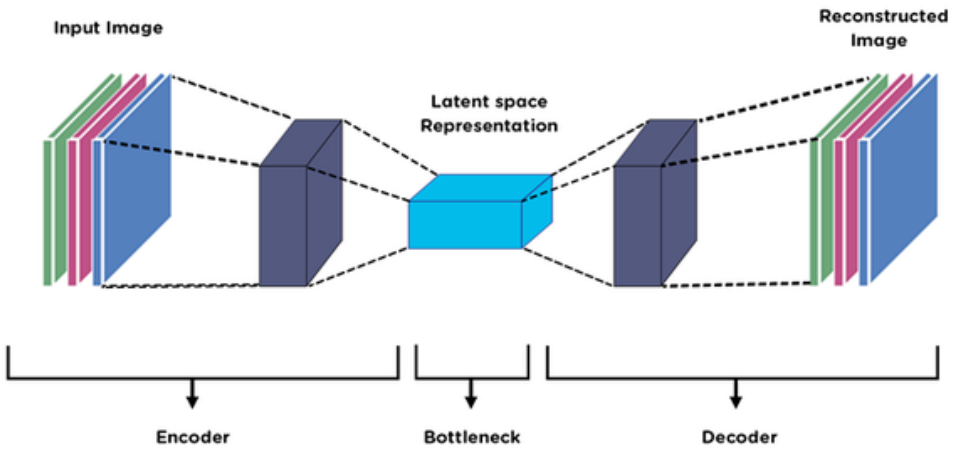

Рис 1.2.4 Схема методу роботи мереж на основі автоенкодерів.

Розробка нейронних мереж для покращення якості зображення приділяє все більше уваги додаткам у реальному часі та апаратному прискоренню. Завдяки використанню графічних процесорів (GPU) і спеціалізованого апаратного забезпечення, наприклад Tensor Processing Units (TPU), ці мережі можуть швидко обробляти й покращувати зображення, забезпечуючи додатки в реальному часі, такі як покращення відео, прямі трансляції та навіть автономне сприйняття транспортних засобів.

Підсумовуючи, шлях нейронних мереж до покращення якості зображення характеризується постійними інноваціями та адаптацією до мінливих вимог різних галузей. Чи йдеться про підвищення чіткості медичних зображень для точного діагнозу, відновлення старих фотографій для збереження спогадів чи забезпечення високоякісного потокового відео, нейронні мережі стали незамінними для створення візуально привабливого та насиченого інформацією вмісту. Їх постійний розвиток обіцяє захоплюючі можливості для покращення якості зображення в багатьох програмах.

1.3 Теоретичний аналіз визначення комп’ютерного зору

Комп’ютерний зір[1-8] — це сфера штучного інтелекту (ШІ), яка дозволяє комп’ютерам і системам отримувати значну інформацію з цифрових зображень, відео та інших візуальних типів даних — і виконувати дії або давати рекомендації на основі цієї інформації. Якщо ШІ дозволяє комп’ютерам мислити, комп’ютерний зір дозволяє їм бачити, спостерігати та розуміти.

Комп’ютерний зір працює майже так само, як і людський, за винятком того, що люди мають перевагу. Людський зір має фору в тривалості ознайомленням з контекстом, щоб навчитися розрізняти об’єкти, як далеко вони знаходяться, чи рухаються вони та чи є щось не так на зображенні.

Комп’ютерний зір навчає машини виконувати ці функції, але робити це потрібно набагато швидше за допомогою камер, даних і алгоритмів, а не

сітківки, зорових нервів і зорової кори. Оскільки система, навчена перевіряти продукти або спостерігати за виробничим активом, може аналізувати тисячі продуктів або процесів за хвилину, помічаючи непомітні дефекти чи проблеми, вона може швидко перевершити людські можливості.

Комп'ютерне бачення використовується в різних галузях промисловості, від енергетики та комунальних послуг до виробництва та автомобілебудування, і ринок продовжує зростати.

Комп'ютерний зір потребує великої кількості даних. Він проводить аналіз даних знову і знову, доки не розпізнає відмінності й остаточно не розпізнає зображення. Наприклад, щоб навчити комп'ютер розпізнавати автомобільні шини, йому потрібно надати величезну кількість зображень шин і предметів, пов'язаних із ними, щоб дізнатися про відмінності та розпізнати її, особливо без дефектів.

Для цього використовуються дві основні технології: тип машинного навчання, що називається глибоким навчанням, і згорткова нейронна мережа (CNN).

Машинне навчання використовує алгоритмічні моделі, які дозволяють комп'ютеру навчитися контексту візуальних даних. Якщо через модель подано достатньо даних, комп'ютер «перегляне» дані та навчиться відрізняти одне зображення від іншого. Алгоритми дозволяють машині навчатися самостійно, а не хтось програмує її розпізнавати зображення.

CNN допомагає моделі машинного або глибокого навчання «бачити», розбиваючи зображення на пікселі, яким присвоюють теги або мітки. Він використовує мітки для виконання згорток (математична операція над двома функціями для отримання третьої функції) і робить прогнози щодо того, що він «бачить». Нейронна мережа запускає згортки та перевіряє точність своїх прогнозів у серії ітерацій, доки прогнози не почнуть збуватися. Тоді це розпізнавання або бачення зображень у спосіб, подібний до людей.

Подібно до того, як людина розпізнає зображення, CNN спочатку розрізняє жорсткі грані та прості форми, а потім заповнює інформацію, коли виконує ітерації своїх прогнозів. Рекурентна нейронна мережа (RNN) використовується подібним чином для відеозастосунків, щоб допомогти комп'ютерам зрозуміти, як зображення в серії кадрів пов'язані одне з одним.

Історія комп'ютерного зору:

Вчені та інженери вже близько 60 років намагаються розробити способи, за допомогою яких машини зможуть бачити та розуміти візуальні дані. Експерименти почалися в 1959 році, коли нейрофізіологи показали кішці низку зображень, намагаючись співвіднести реакцію в її мозку. Вони виявили, що тварина спочатку реагує на жорсткі краї або лінії, і з наукової точки зору це означає, що обробка зображень починається з простих форм.

Приблизно в той же час була розроблена перша технологія комп'ютерного сканування зображень, яка дозволила комп'ютерам оцифровувати та отримувати зображення. Ще одна віха була досягнута в 1963 році, коли комп'ютери змогли перетворити двовимірні зображення в тривимірні форми. У 1960-х роках штучний інтелект став академічною галуззю дослідження, і це також поклало початок пошукам штучного інтелекту щодо вирішення проблеми людського зору.

У 1974 році була запроваджена технологія оптичного розпізнавання символів (Optical Character Recognition, або OCR), яка могла розпізнавати текст, надрукований будь-яким шрифтом[18]. Подібним чином інтелектуальне розпізнавання символів (Intelligent Character Recognition, або ICR) могло розшифровувати рукописний текст за допомогою нейронних мереж. Відтоді OCR та ICR знайшли свій шлях до обробки документів, розпізнавання номерних знаків транспортних засобів, мобільних платежів, машинного перекладу та інших поширених галузей.

У 1982 році нейробіолог Девід Марр встановив, що зір працює ієрархічно, і запровадив алгоритми для машин, щоб виявляти краї, кути, криві та подібні основні форми. Паралельно комп'ютерний вчений Куніхіко Фукусіма розробив мережу клітин, які можуть розпізнавати шаблони. Мережа під назвою Neocognitron включала згорткові шари в нейронну мережу.

До 2000 року основна увага приділялася розпізнаванню об'єктів, а до 2001 року з'явилися перші програми для розпізнавання обличчя в реальному часі. Стандартизація того, як набори візуальних даних позначаються тегами та анотуються з'явилася вже в 2000-х роках. У 2010 році став доступний набір даних для навчання нейронних мереж ImageNet. Він містив мільйони позначених тегами зображень у тисячах класів об'єктів і забезпечує основу для CNN і моделей глибокого навчання, які використовуються сьогодні. У 2012 році команда з Університету Торонто взяла участь у конкурсі CNN на розпізнавання зображень. Модель під назвою AlexNet значно знизила частоту помилок при розпізнаванні зображень. Після цього прориву рівень помилок впав лише до кількох відсотків.

Приклади використання програм на основі комп'ютерного зору:

У галузі комп'ютерного зору проводиться багато досліджень, але це не просто дослідження. Реальні програми демонструють, наскільки важливий комп'ютерний зір для діяльності в бізнесі, розвагах, транспорті, охороні здоров'я та повсякденному житті. Ключовою рушійною силою зростання цих додатків є потік візуальної інформації, що надходить зі смартфонів, систем безпеки, камер дорожнього руху та інших пристроїв з візуальними інструментами. Ці дані можуть відігравати важливу роль у діяльності в різних галузях. Подібна інформація створює тестовий стенд для навчання додатків комп'ютерного зору, щоб вони стали частиною ряду людських дій:

- IBM використала комп'ютерний зір, щоб створити My Moments для гольф-турніру Masters 2018. Компанія IBM Watson переглянула сотні годин

записів Masters і змогла розпізнати зображення (і звуки) найцікавіших та важливих кадрів матчу. Він підібрав ці ключові моменти та представив їх шанувальникам у вигляді персоналізованих відеороликів.

- Перекладач Google дозволяє користувачам навести камеру смартфона на вивіску іншою мовою та майже одразу отримати переклад на бажану мову.
- Розробка безпілотних транспортних засобів покладається на комп'ютерне зір, щоб зрозуміти візуальний вхід від камер автомобіля та інших датчиків. Важливо розпізнавати інші автомобілі, дорожні знаки, смуги руху, пішоходів, велосипедів та всю іншу візуальну інформацію, що зустрічається на дорозі.

Ось кілька прикладів поширених завдань комп'ютерного зору:

Класифікація зображень розпізнає зображення та може його класифікувати (собака, яблуко, обличчя людини). Точніше, він здатний точно передбачити, що дане зображення належить до певного класу. Наприклад, компанія соціальних медіа може використовувати його для автоматичної ідентифікації та відокремлення небажаних зображень, завантажених користувачами.

Розпізнавання об'єктів може використовувати класифікацію зображень для ідентифікації певного класу, а потім виявляти та зводити в таблицю їх появу на зображенні або відео. Наприклад, пошук браків на складальній лінії або ідентифікацію обладнання, яке потребує технічного обслуговування.

Відстеження об'єкта стежить за об'єктом після його виявлення. Це завдання часто виконується за допомогою послідовних зображень або відео в реальному часі. Автономні транспортні засоби, наприклад, повинні не тільки класифікувати та виявляти такі об'єкти, як пішоходи, інші автомобілі та

дорожня інфраструктура, вони повинні відстежувати їх у русі, щоб уникнути зіткнень і дотримуватися правил дорожнього руху.

1.4 Огляд та переваги використання PyTorch

PyTorch [26],.— це платформа машинного навчання з відкритим кодом, яка набула величезної популярності завдяки своїй гнучкості, динамічному графіку обчислень і простоті використання. Він був розроблений дослідницькою лабораторією Facebook AI Research (FAIR) і став популярним вибором для дослідників, розробників і спеціалістів із обробки даних у сфері глибокого навчання. Нижче будуть розглянуті функції та аспекти, які роблять PyTorch кращим інструментом для створення та навчання нейронних мереж.

Фреймворк PyTorch підтримує понад 200 різних математичних операцій. Популярність PyTorch продовжує зростати, оскільки він спрощує створення моделей штучних нейронних мереж. PyTorch в основному використовується дослідниками даних для досліджень і програм штучного інтелекту. PyTorch випущено за модифікованою ліцензією BSD.

Спочатку PyTorch був проєктом стажування для Адама Паске, який на той час був студентом Суміта Чінтали, одного з розробників Torch. Паске та кілька інших працювали з розробниками з різних університетів і компаній, щоб протестувати PyTorch. Зараз Чінтала працює дослідником у Meta, колишній Facebook, яка використовує PyTorch як базову платформу для управління всіма робочими навантаженнями ШІ.

PyTorch є «пітонічним» за своєю природою, що означає, що він дотримується стилю кодування, який використовує унікальні функції Python для написання читабельного коду. Python також популярний завдяки використанню динамічних обчислювальних графіків. Це дозволяє розробникам, науковцям і налагоджувачам нейронних мереж запускати та тестувати частину коду в режимі реального часу, а не чекати, поки буде написана вся програма.

Основні характеристики PyTorch:

- Динамічно-обчислювальний графік:

Однією з видатних особливостей PyTorch є його динамічно-обчислювальний графік. На відміну від деяких інших фреймворків глибокого навчання зі статичними графіками, PyTorch будує графіки під час виконання операцій. Ця властивість забезпечує більш інтуїтивне налагодження, легку модифікацію моделі та динамічний потік керування. Написання коду схоже на Pythonic, що робить його доступним для тих, хто знайомий з програмуванням на Python.

- Автоматична диференціація (Autograd):

PyTorch пропонує вбудовану бібліотеку автоматичного розрізнення, відому як Autograd. Він автоматично обчислює градієнти тензорів відносно деякої функції втрат. Ця функція спрощує зворотне поширення, важливу частину навчання нейронних мереж, і усуває потребу в обчисленнях градієнта вручну. З PyTorch можна зосередитися на розробці й експериментуванні з архітектурою нейронної мережі, а не турбуватися про дрібниці градієнтів.

- TorchScript:

Це виробниче середовище PyTorch, яке дозволяє користувачам плавно переходити між режимами. TorchScript оптимізує функціональність, швидкість, простоту використання та гнучкість.

- Тензори та інтеграція NumPy:

PyTorch легко взаємодіє з NumPy, популярною бібліотекою для чисельних обчислень на Python. Тензори PyTorch, які є фундаментальними структурами даних для нейронних мереж, схожі на масиви NumPy, що полегшує маніпулювання даними та перехід між цими двома бібліотеками. Ця інтеграція спрощує попередню обробку даних і покращує читабельність коду.

- Модульна будова:

Модульний дизайн PyTorch дозволяє створювати моделі нейронної мережі як послідовність модулів або шарів. Можна легко комбінувати готові модулі для створення складних архітектур. Крім того, API PyTorch є розширюваним, що дозволяє дослідникам впроваджувати власні шари, функції втрат і алгоритми оптимізації, що робить його придатним для широкого спектру програм і експериментів.

- Підтримка спільноти та бібліотеки:

PyTorch може похвалитися живою та зростаючою спільнотою дослідників і розробників, які роблять внесок у його екосистему. Це призвело до появи великої кількості попередньо навчених моделей, проєктів з відкритим кодом і бібліотек сторонніх розробників, які розширюють функціональні можливості PyTorch. Наприклад, PyTorch Hub надає доступ до численних попередньо підготовлених моделей для таких завдань, як класифікація зображень і обробка мови.

- Змінні:

Змінні укладені поза тензором, щоб утримувати градієнт без небажаних втручань. Вони представляє вузол на обчислювальному графі.

- Функції:

Це зв’язки між двома змінними. Функції не мають пам’яті для збереження будь-якого стану чи буфера та не мають власної пам’яті.

Як PyTorch спрощує розробку нейронних мереж

Динамічний обчислювальний графік PyTorch сприяє спрощуванню налагодження та експериментуванню. Ви можете перевіряти проміжні значення, змінювати архітектуру мережі в ході справи та відстежувати потік даних через мережу легше, ніж у рамках зі статичними графіками. Це особливо корисно на етапі дослідження, коли ви точно налаштовуєте свою модель.

Модульний і розширюваний дизайн PyTorch дозволяє з легкістю створювати власні архітектури. Ви можете експериментувати з новими структурами нейронних мереж і легко включати їх у свої моделі.

Екосистема PyTorch, керована спільнотою, пропонує доступ до широкого спектру попередньо навчених моделей. Ці моделі можна точно налаштувати для конкретних завдань, що є неймовірно цінним, коли вам потрібно побудувати ефективні моделі з обмеженими даними.

PyTorch може використовувати графічні процесори, що робить його високоефективним для завдань глибокого навчання. Це прискорення є важливим для навчання великих нейронних мереж на складних наборах даних.

Таким чином, динамічний обчислювальний графік PyTorch, бездоганна інтеграція з NumPy, автоматична диференціація та модульний дизайн зробили його видатною структурою для розробки нейронних мереж. Він значно спростив процес створення, навчання та експериментування з моделями глибокого навчання, що зробило його найкращим вибором для дослідників і розробників у цій галузі.

PyTorch в порівнянні з TensorFlow

PyTorch часто порівнюють із TensorFlow, системою глибокого машинного навчання, розробленою Google. Оскільки TensorFlow існує довше, він має більшу спільноту розробників і більше документації.

Проте PyTorch має переваги перед TensorFlow. PyTorch динамічно визначає обчислювальні графіки, на відміну від статичного підходу TensorFlow. Динамічними графіками можна керувати в реальному часі. Крім того, TensorFlow має крутішу криву навчання, оскільки PyTorch базується на інтуїтивно зрозумілому Python.

TensorFlow може краще підходити для проєктів, які вимагають виробничих моделей і масштабованості, оскільки він був створений з наміром

бути готовим до виробництва. Проте з PyTorch легше й легше працювати, що робить його хорошим варіантом для швидкого створення прототипів і проведення досліджень.

Галузі використання PyTorch:

PyTorch є одним із найпопулярніших фреймворків глибокого навчання завдяки своїй гнучкості та потужності обчислень. Його легко освоїти, і він використовується в багатьох програмах, включаючи обробку природної мови (Natural Language Processing, або NLP) і класифікацію зображень.

Нижче наведено кілька типових випадків використання PyTorch:

Natural Language Processing — це технологія, яка дозволяє комп'ютеру розуміти усну чи письмову людську мову. Основні елементи NLP включають машинний переклад, пошук інформації, аналіз настроїв, вилучення інформації та відповіді на запитання.

Глибокі нейронні мережі лежать в основі кількох проривів у машинному розумінні природних мов, таких як Siri та Google Translate. Але більшість із цих моделей використовують метод рекурсивної нейронної мережі, щоб розглядати мову як плоску послідовність слів, тоді як багато лінгвістів підтримують модель рекурсивної нейронної мережі, оскільки вважають, що мову найкраще зрозуміти, коли вона представлена в ієрархічному дереві фраз. PyTorch полегшує розуміння цих складних мовних моделей. Наприклад, у 2018 році Salesforce розробила багатозадачну модель навчання NLP, яка виконує 10 завдань одночасно.

Навчання з підкріпленням (Reinforcement Learning, або RL). Бібліотека Python, відома як Pyqlearning, використовується для виконання навчання з підкріпленням, яке є підмножиною машинного навчання. У RL машина створена для навчання на досвіді, щоб вона могла приймати правильні рішення та отримувати найкращу можливу винагороду. RL в основному використовується для розробки робототехніки для автоматизації, керування

рухом роботів або планування бізнес-стратегії та використовує архітектуру навчання Python Deep Q для створення моделі.

Класифікація зображень — це процес, який класифікує зображення на основі його візуального вмісту за допомогою алгоритму класифікації зображень. Наприклад, алгоритм може повідомити програмі комп'ютерного зору, чи містить певне зображення кішку чи собаку. У той час як виявлення об'єктів не викликає зусиль для людського ока, це може бути складним для програм комп'ютерного зору. Використовуючи PyTorch, розробник може обробляти зображення та відео для створення точної моделі комп'ютерного зору.

Висновок до розділу

У цьому розділі ми розглянули дослідження нейронних мереж, їх життєво важливу роль у покращенні якості зображення, теоретичні основи комп'ютерного зору та переваги використання фреймворку PyTorch. Разом ці теми забезпечують повне розуміння того, як сучасні технології дозволяють нам сприймати, інтерпретувати та маніпулювати візуальними даними, способами, які раніше вважалися неможливими.

Початок взятий з вивчення структури нейронів у нейронній мережі, проводячи паралелі з біологічними аналогами, які їх надихають. Було звернуто увагу на будівельні блоки штучного інтелекту, які прояснили механізми, за допомогою яких нейронні мережі отримують вхідні дані, обробляють їх і створюють результати. Концепція вагових коефіцієнтів і зміщень, що діють як регульовані «ручки гучності» для вхідних сигналів, і зв'язки між нейронами, які служать синапсами в соціальній мережі обміну інформацією, показали складну взаємодію, яка характеризує ці системи.

Далі було розглянуто формування та розвиток нейронних мереж для покращення якості зображення. Від ранніх початків створення штучних

нейронних мереж до появи згорткових нейронних мереж (CNN) і революційного впливу генеративних-суперницьких мереж (GAN). З'ясували, як нейронні мережі еволюціонували, щоб досягти успіху в таких завданнях, як усунення шуму зображення, надвисока роздільна здатність і передача стилю. Підкреслили силу передачі навчання, подібну до обміну знаннями між художниками, і неоціненну роль, яку відіграють автокодери у відновленні зображень до їх первісного стану.

Переходячи до сфери комп'ютерного зору, було проведено теоретичний аналіз, щоб зрозуміти його визначення та наслідки. Було виявлено, що комп'ютерний зір — це не просто розпізнавання об'єктів, це передбачає покращення якості зображення, виявлення закономірностей і навіть відновлення історичних моментів, зафіксованих на фотографіях. Динамічний характер комп'ютерного зору, посилений нейронними мережами, дозволяє нам досліджувати невидиме та отримувати результати, які виходять за межі традиційного людського сприйняття.

Проведено огляд і визначення переваг PyTorch, фреймворку з відкритим кодом, який став основою досліджень і розробок глибокого навчання. Його динамічний обчислювальний графік, автоматичне розрізнення, бездоганна інтеграція з NumPy, модульний дизайн і широка підтримка спільноти спростили створення та навчання нейронних мереж. Роль PyTorch у тому, щоб зробити розробку нейронних мереж доступною, ефективною та універсальною, очевидна у випадках її використання, починаючи від розпізнавання зображень і закінчуючи розумінням природної мови.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТО ОГЛЯД ПРОГРАМНОЇ ОБЛАСТІ

2.1 Принцип роботи Згорткових мереж (CNN)

Згорткова нейронна мережа (ConvNet/CNN) — це алгоритм глибокого навчання, який може приймати вхідне зображення, привласнювати важливість (вивчені ваги та зміщення) аспектам або об'єктам зображення і відрізняти одне від одного. При цьому зображення порівняно з іншими алгоритмами вимагають набагато менше попередньої обробки. У примітивних методах фільтри розробляються вручну, але навчені мережі CNN вчаться застосовувати ці фільтри/характеристики.

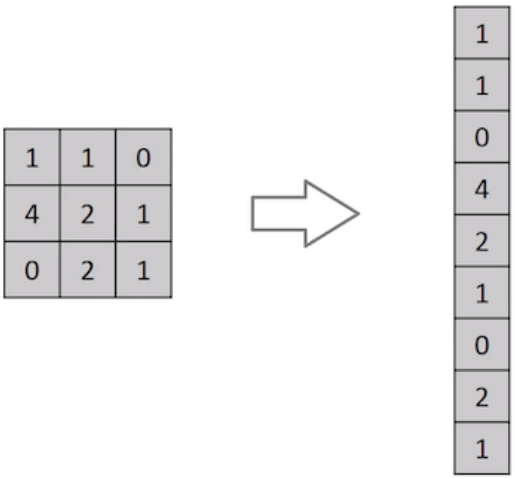

Рис 2.1.1 Матриця 3×3 у вигляді вектора 9×1

Для розуміння та обробки зображення нейронною мережею його потрібно перевести в вектор. У випадках найпростіших двійкових зображень при виконанні прогнозування класів метод може показати середню точність, але на практиці, коли мова піде про складні зображення, в яких піксельні залежності, він виявиться неточним.

Мережа CNN здатна з успіхом схоплювати просторові та часові залежності у зображенні через застосування відповідних фільтрів. Така архітектура за рахунок скорочення кількості задіяних параметрів та можливості повторного використання ваг дає кращу відповідність набору даних зображень.

Іншими словами, мережу можна навчити краще розуміти складність зображення.

Вхідне зображення

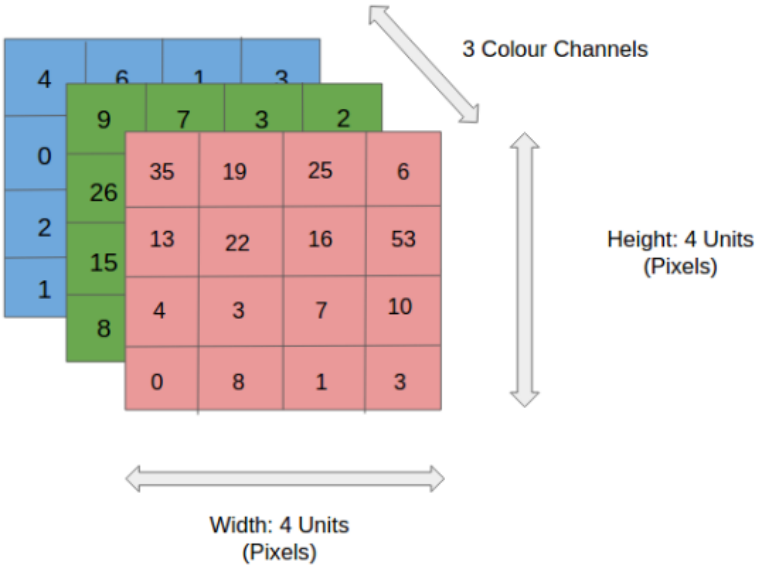

Рис 2.1.2 RGB-зображення 4×4×3

На зображені вище (Рис 2.1.2) ми бачимо розділене на три колірні площини (червону, зелену та синю) RGB-зображення, яке можна описати в різних колірних просторах - у відтінках сірого (Grayscale), RGB, HSV, CMYK і т.п.

Роль CNN полягає в тому, щоб привести зображення у форму, яку легко обробляти, без втрати ознак, що мають вирішальне значення в отриманні хорошого прогнозу. Це важливо при розробці архітектури, яка не тільки добре вивчає функції, а й масштабується для масивних наборів даних.

Шар згортки – ядро(Kernel)

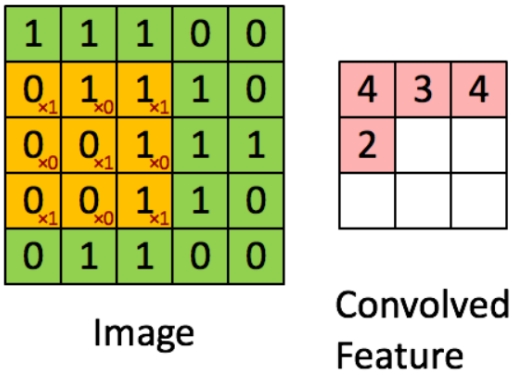

Рис 2.1.3 Згортка зображення 5×5×1 з ядром 3×3×1 для отримання згорнутої ознаки 3×3×1

У демонстрації вище зелена секція нагадує наше вхідне зображення 5×5×1. Елемент, що бере участь у виконанні операції згортки в першій частині шару згортки, називається ядром/фільтром K, представлений жовтим кольором. Нехай K буде матрицею 3×3×1:

```
Kernel/Filter, K =
1 0 1
0 1 0
1 0 1
```

Ядро зміщується 9 разів через довжину кроку в одиницю (тобто кроку немає), щоразу виконуючи операцію множення матриці K на матрицю P, над якою знаходиться ядро.

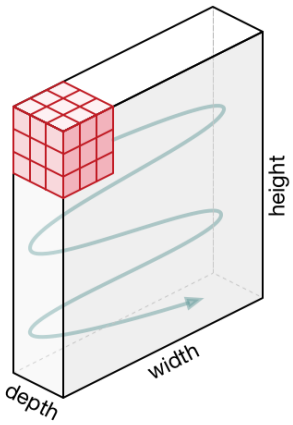

Рис 2.1.4 Переміщення ядра

Фільтр переміщається вправо з певним значенням кроку, доки не проаналізує всю ширину. Рухаючись далі, він переходить до початку зображення (ліворуч) з тим самим значенням кроку і повторює процес доти, доки проходить все зображення.

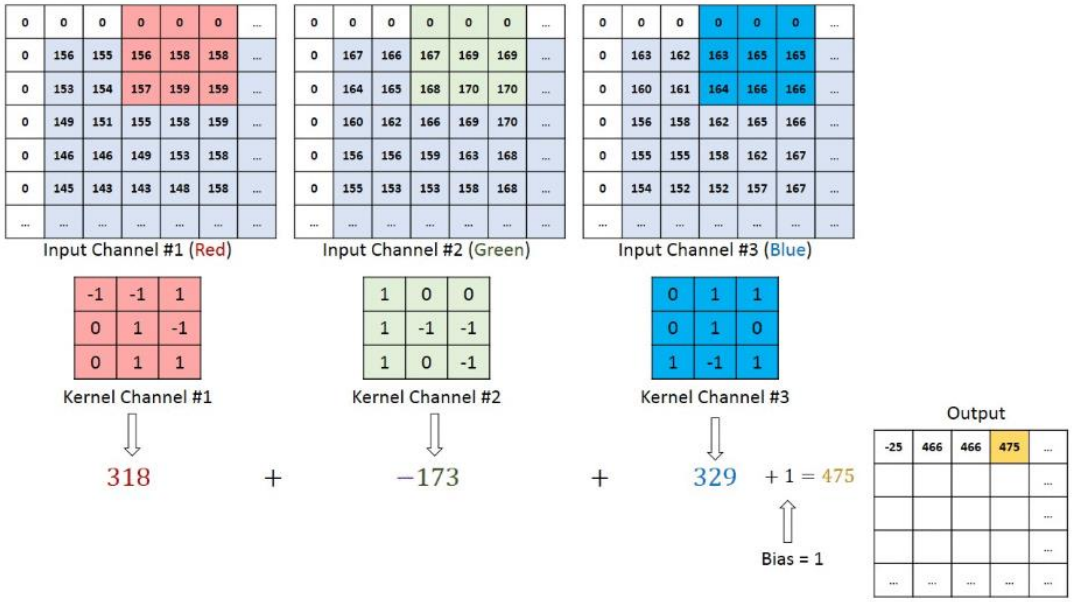

Рис 2.1.5 Операція згортки на матриці зображення M×N×3 з ядром 3×3×3

У разі зображень з кількома каналами (наприклад, RGB) ядро має таку саму глибину, що й у вхідного зображення. Матричне множення виконується між стеками K_n та I_n ($[K_1, I_1]$; $[K_2, I_2]$; $[K_3, I_3]$), всі результати підсумовуються зі зміщенням, щоб отримати сплощений канал виведення згорнутих ознак з глибиною в 1.

Згортка робиться, щоб витягти високорівневі ознаки, наприклад краї вхідного зображення. Мережу не потрібно обмежувати єдиним шаром. Перший шар умовно несе відповідальність за схоплювання ознак низького рівня, таких як краї, колір, орієнтація градієнта і т. д. Через додаткові шари архітектура адаптується до ознак високого рівня, ми отримуємо мережу зі здоровим розумінням зображень у наборі даних, схожих на наше.

У результатів згортки два типи: перший - згорнута ознака зменшується у розмірі порівняно з розміром на вході, другий тип стосується розмірності - вона залишається колишньою, або збільшується. Це робиться шляхом застосування допустимого заповнення у першому випадку або нульового заповнення – у другому.

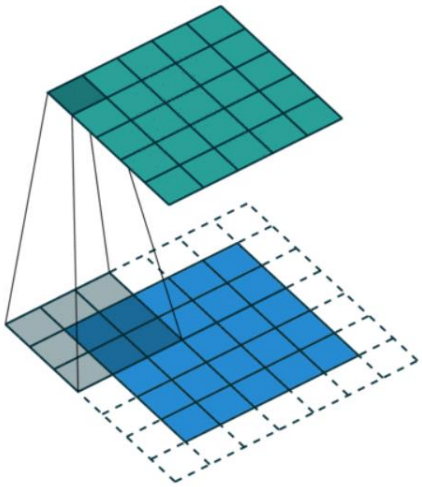

Рис 2.1.6 Нульове заповнення: для створення зображення 6×6×1 зображення 5×5×1 доповнюється нулями

Збільшуючи зображення 5×5×1 до 6×6×1, а потім проходячи над ним ядром 3×3×1, ми виявимо, що згорнута матриця матиме роздільну здатність 5×5×1. Звідси й назва – **нульове наповнення**. З іншого боку, зробивши те саме без заповнення, ми виявимо матрицю з розмірами самого ядра (3×3×1); ця операція називається **допустимим наповненням**.

Шар об'єднання

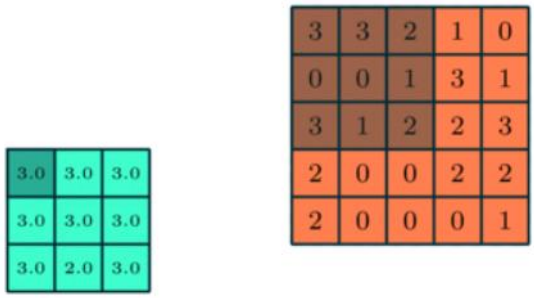

Рис 2.1.7 Об'єднання 3×3 над згорнутою ознакою 5×5

Подібно до згорткового шару, шар об'єднання відповідає за зменшення розміру згорнутого об'єкта в просторі. Це робиться для зменшення необхідної під час обробки даних обчислювальної потужності за рахунок скорочення розмірності. Крім того, це корисно для отримання домінуючих ознак, які позиційними інваріантами, тим самим дозволяючи підтримувати процес ефективного навчання моделі.

Є два типи об'єднання: максимальне та середнє. Перше повертає максимальне значення із покритої ядром частини зображення. А середнє об'єднання повертає середнє значення зі всіх значень покритої ядром частини.

Максимальне поєднання також виконує функцію шумоподавлення. Воно повністю відкидає зашумлені активації, а також усуває шум разом із зменшенням розмірності. З іншого боку, середнє поєднання для придушення шуму просто знижує розмірність. Отже, можна сказати, що максимальне об'єднання працює набагато краще за середнє об'єднання.

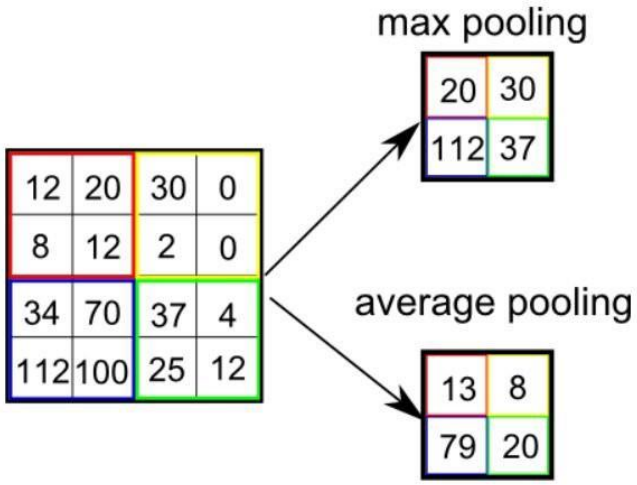

Рис 2.1.8 Типи об’єднань

Шари об'єднання та згортки разом утворюють і-тий шар згорткової нейронної мережі. Кількість таких шарів може бути збільшено залежно від складності зображень, щоб краще схоплювати деталі, але це робиться рахунок збільшення обчислювальної потужності.

Виконання процесу вище дозволяє моделі розуміти особливості зображення. Перетворимо результат на вектор і подамо його до звичайної нейронної мережі, що класифікує.

Класифікація - повнозв'язковий шар

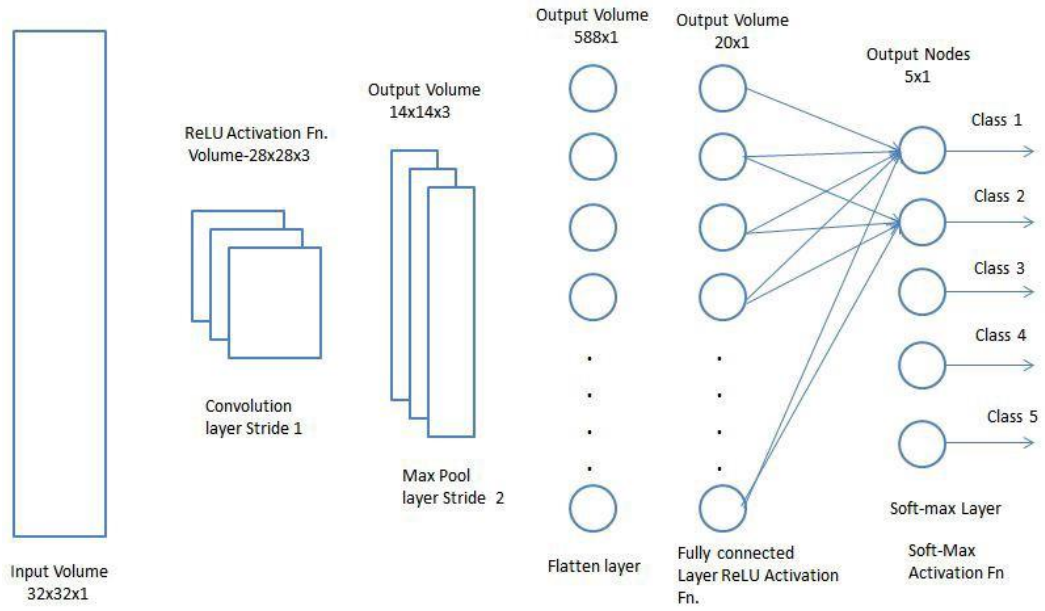

Рис 2.1.9 Використання вектора після згортки в повнозв'язній нейронній мережі

Додавання повнозв'язкового шару - це (зазвичай) обчислювально-недорогий спосіб навчання нелінійним комбінаціям високорівневих ознак, які представлені на виході шару згортки. Повнозв'язковий шар вивчає функцію у цьому просторі, яка може бути нелінійною.

Після перетворення вхідного зображення у придатну для багаторівневого перцептрону форму ми повинні згладити зображення у вектор. Згладжений вихідний сигнал подається на нейронну мережу з прямим зв'язком, при цьому на кожній ітерації навчання застосовується зворотне поширення(**back propagation**). За серію епох модель знаходить здатність розрізняти домінуючі та деякі низькорівневі ознаки у зображеннях та класифікувати їх методом класифікації Softmax.

2.2 Генеративно-змагальна мережа (GAN). Генератор да Дискримінатор

Generative Adversarial Network (GAN) — це потужний клас нейронних мереж, які використовуються для неконтрольованого навчання. Він був розроблений і представлений Яном Дж. Гудфеллоу в 2014 році. GAN в основному складаються із системи двох конкуруючих моделей нейронних мереж, які конкурують одна з одною та здатні аналізувати, фіксувати та копіювати варіації в наборі даних.

Було помічено, що більшість звичайних нейронних мереж можна легко обдурити, щоб неправильно класифікувати речі, додавши лише невелику кількість шуму до вихідних даних. Модель після додавання шуму має більшу впевненість у неправильному прогнозі, ніж коли вона прогнозувала правильно. Причина такого супротиву полягає в тому, що більшість моделей машинного навчання вивчають обмежену кількість даних, що є величезним недоліком, оскільки воно схильне до перенавчання. Крім того, відображення між входом і виходом майже однакове. Хоча може здатися, що межі поділу між різними класами є лінійними, але насправді вони складаються з лінійностей, і навіть невелика зміна в точці в просторі ознак може призвести до неправильної класифікації даних.

Генеративно-змагальні мережі (GAN) можна розбити на три частини:

- **Генеративна -** вивчення генеративної моделі, яка описує, як дані генеруються в термінах імовірнісної моделі.
- **Змагальна -** навчання моделі відбувається в умовах змагання.
- **Мережева -** використання глибоких нейронних мереж як алгоритми штучного інтелекту (ШІ) для навчальних цілей.

У GAN є генератор і дискримінатор. Генератор створює підроблені зразки даних (зображення, аудіо тощо) і намагається обдурити Дискримінатор. Дискримінатор, з іншого боку, намагається відрізнити справжні зразки від

підроблених. Генератор і Дискримінатор є нейронними мережами, і обидва вони конкурують один з одним на етапі навчання. Ці кроки повторюються кілька разів, і після кожного повторення Генератор і Дискримінатор стають все кращими і кращими. Роботу можна наочно представити схемою, наведеною нижче(Рис 2.2.1):

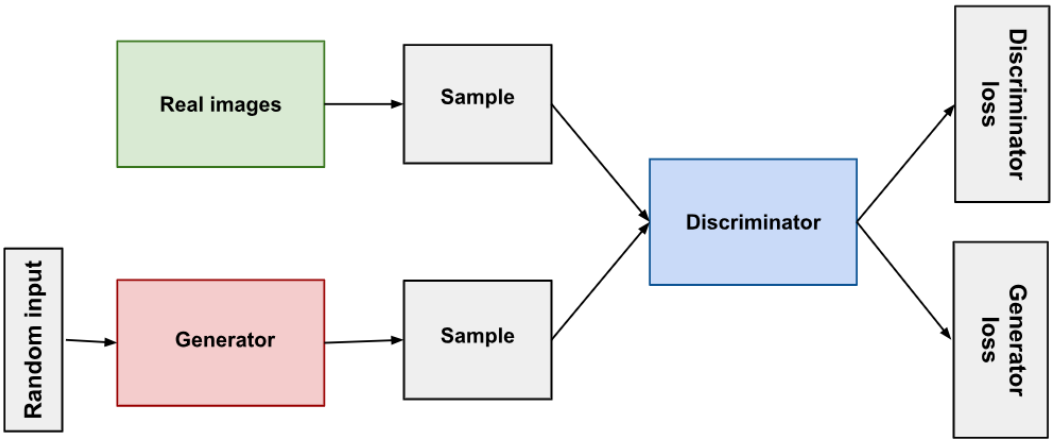

Рис 2.2.1 Схема роботи моделі GAN.

Генеративна модель фіксує розподіл даних і навчається таким чином, щоб намагатися максимізувати ймовірність помилки дискримінатора. Дискримінатор, з іншого боку, заснований на моделі, яка оцінює ймовірність того, що отриманий зразок отримано з навчальних даних, а не з генератора. GAN сформульовано як мінімаксну гру, де Дискримінатор намагається мінімізувати свою винагороду V(D, G), а Генератор намагається мінімізувати винагороду Дискримінатора або, іншими словами, максимізувати свої втрати. Це можна математично описати формулою нижче:

$$\min_G \max_D V(D,G)$$

$$V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

де, G = Генератор;

D = Дискримінатор;

$p_{data}(x)$ = розподіл реальних даних;

p_z = розподіл генератора;

x = вибірка з $p_{data}(x)$;

z = вибірка з p_z;

$D(x)$ = мережа дискримінатора;

$G(z)$ = Генеративна мережа.

Генератор да Дискримінатор

Архітектура генератора в основному є повністю згортковою моделлю (CNN), яка використовується для створення високоякісних зображень із надвисокою роздільною здатністю. Додавання моделі дискримінатора, яка діє як класифікатор зображень, створено для забезпечення того, щоб загальна архітектура налаштовувалася відповідно до якості зображень, а отримані зображення були набагато оптимальнішими. Архітектура GAN створює правдоподібні природні зображення з високою якістю сприйняття.

Генератор:

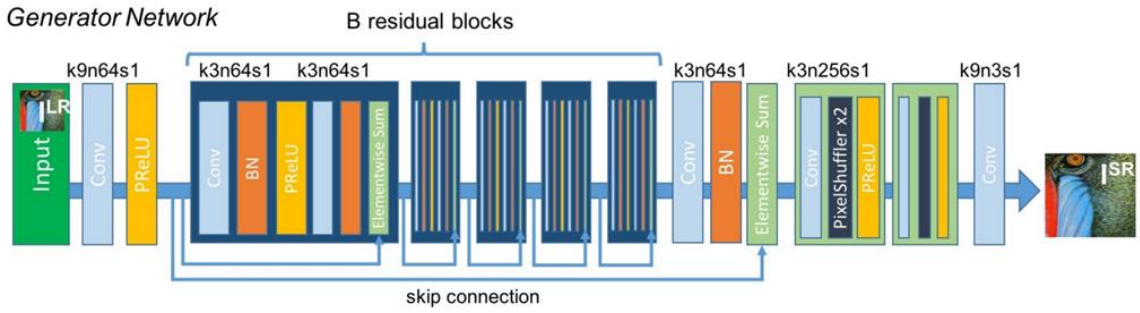

Рис 2.2.2 Схема генератора GAN.

Архітектура генератора генераторної мережі складається з вхідних даних із низькою роздільною здатністю, які проходять через початковий згортковий рівень із ядер (**Kernal**) 9×9 і 64 карт функцій активації, за якими слідує параметричний рівень ReLU. Альтернативним варіантом є Leaky ReLU, де значення, менші за нуль, зіставляються з числом, встановленим користувачем. Однак у випадку параметричного ReLU ми можемо дозволити

нейронній мережі вибрати найкраще значення самостійно, отже, вона є кращою в цьому сценарії.

Наступний рівень повної згорткової моделі із прямою передачею використовує групу залишкових блоків (**residual blocks**). Кожен із залишкових блоків містить згортковий рівень із ядер 3×3 і 64 карти функцій активації, за якими слідує рівень пакетної нормалізації (**BuchNorm, або BN**), параметрична функція активації ReLU, ще один згортковий рівень із пакетною нормалізацією та кінцевий метод поелементної суми. Метод поелементної суми використовує висновок прямої подачі(**feed-forward**) разом із виходом пропуску з'єднань(**skip connection**) для надання остаточного результату.

Після створення залишкових блоків будується решта моделі генератора, як показано на зображенні вище(Рис 2.2.2). Використовується піксельне перемішування в архітектурі моделі генератора після 4-кратної дискретизації згорткового шару для створення зображень із надвисокою роздільною здатністю. Піксельні «тасувальники» (**Pixel Shuffler**) беруть значення з розміру каналу та вставляють їх у розміри висоти та ширини. У цьому випадку і висота, і ширина множаться на два, а канал ділиться на два.

Дискримінатор:

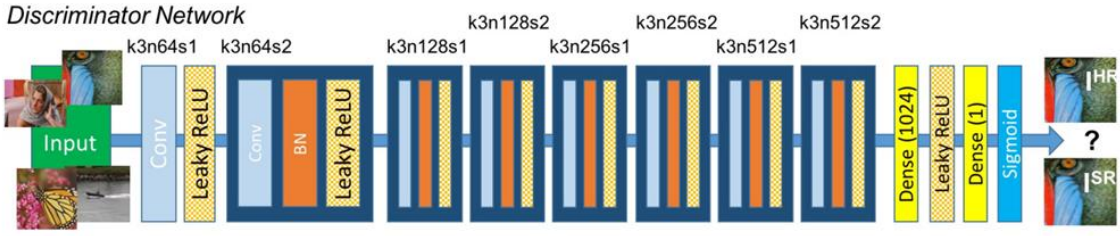

Рис 2.2.3 Схема дискримінатора GAN.

Архітектура дискримінатора побудована найкращим чином для підтримки типової процедури GAN. І генератор, і дискримінатор конкурують між собою, і вони обидва вдосконалюються одночасно. Поки мережа дискримінатора намагається знайти фальшиві зображення, генератор

намагається створювати реалістичні зображення, щоб уникнути виявлення дискримінатора. Робота у випадку GAN також подібна, де генеративна модель *G* з метою обдурити диференційований дискримінатор *D*, який навчений відрізняти зображення з високою роздільною здатністю від реальних зображень.

Таким чином, архітектура дискримінатора, показана на зображенні вище(Рис 2.2.3), працює для розрізнення зображень із надвисокою роздільною здатністю та реальних зображень. Побудована дискримінаторна модель спрямована на розв'язання змагальної проблеми мінімально-максимального значення. Загальну ідею формулювання цього рівняння можна інтерпретувати наступним чином:

$$\min_{\theta_G} \max_{\theta_D} \mathbb{E}_{I^{HR} \sim p_{train}(I^{HR})} \left[\log D_{\theta_D}(I^{HR})\right] + \mathbb{E}_{I^{LR} \sim p_G(I^{LR})}\left[\log(1 - D_{\theta_D}(G_{\theta_G}(I^{LR})))\right]$$

Побудована архітектура дискримінатора є досить інтуїтивно зрозумілою та легкою для розуміння. Ми використовуємо початковий згортковий шар, за яким слідує функція активації Leaky ReLU. Значення альфа(**α**) для Leaky ReLU встановлено на 0,2 для цієї структури. Потім у нас є купа повторюваних блоків згорткових шарів, за якими йде шар пакетної нормалізації та функція активації Leaky ReLU. З п'яти повторюваних блоків складаються щільні шари, за якими слідує функція активації сигмоподібної форми для виконання дії класифікації. Зауважте, що початковий розмір згортки становить 64×64, який множиться на 2 після двох повних блоків кожен, доки ми не досягнемо 8-кратного коефіцієнта масштабування 512×512. Ця модель дискримінатора допомагає генератору навчатися ефективніше та отримувати кращі результати.

2.3 Математична модель GAN

Спочатку введемо необхідну термінологію. Через *X* ми позначатимемо деякий простір об'єктів. Наприклад, зображення 64×64×3 пікселя. На деякому

імовірнісному просторі Ω задана випадкова векторна величина $x: \Omega \to X$ з розподілом ймовірностей, що мають щільність $p(x)$ таку, що підмножина простору X, на якому $p(x)$ приймає ненульові значення - це, наприклад, фотографії людських обличь. Нам дано випадкову i.i.d. вибірка фотографій обличь для величини $\{x_i, i \in [1, N], x_i \sim p(x)\}$. Додатково визначимо допоміжний простір $Z = R^n$ та випадкову величину $z: \Phi \to Z$ з розподілом ймовірностей, що мають щільність $q(z)$. $D: X \to (0,1)$ - функція-дискримінатор. Ця функція приймає на вхід об'єкт $x \in X$ (у нашому прикладі – зображення відповідного розміру) і повертає ймовірність того, що вхідна картинка є фотографією людського обличчя. $G: Z \to X$ - функція-генератор. Вона набуває значення $z \in Z$ і видає об'єкт простору X, тобто, у нашому випадку, картинку.

Припустимо, що ми вже маємо ідеальний дискримінатор D. Для будь-якого прикладу x він видає справжню ймовірність приналежності цього прикладу заданого підмножині X, з якого отримана вибірка $\{x_i\}$. Переформулюючи завдання обману дискримінатора ми отримуємо, що необхідно максимізувати ймовірність, що видається ідеальним дискримінатором на згенерованих прикладах. Таким чином, оптимальний генератор знаходиться як $G^* = \arg\max_G E_{z \sim q(x)} D_k(G(z))$. Так як $\log(x)$ - монотонно зростаюча функція і не змінює положення екстремумів аргументу, цю формулу перепишемо у вигляді $G^* = \arg\max_G E_{z \sim q(x)} \log D_k(G(z))$, що буде зручніше надалі.

Насправді зазвичай ідеального дискримінатора немає і його треба знайти. Оскільки завдання дискримінатора — надавати сигнал на навчання генератора, замість ідеального дискримінатора досить взяти дискримінатор який ідеально відокремлює реальні приклади від згенерованих поточним генератором. Тобто ідеальний тільки на підмножині X, з якого генеруються приклади поточним генератором. Це завдання можна переформулювати як пошук такої функції D, яка максимізує ймовірність правильної класифікації прикладів як справжніх або згенерованих. Це називається завданням бінарної

класифікації і в даному випадку ми маємо нескінченну навчальну вибірку: кінцева кількість реальних прикладів і потенційно нескінченна кількість згенерованих прикладів. Кожен приклад має мітку: справжній він або згенерований.

Отже, наша вибірка $S = \{(x, 1), x{\sim}p(x)\} \cup \{(G(z), 0), z{\sim}q(z)\}$. Визначимо щільність розподілу $f(\xi|\eta = 1) = D(\xi)$, $f(\xi|\eta = 0) = 1 - D(\xi)$, тоді $f(\xi|\eta)$ - це переформулювання дискримінатора D, що надає можливість класу 1 (справжній приклад) у вигляді розподілу на класах $\{0, 1\}$. Так як $D(\xi) \in (0,1)$, це визначення задає коректну щільність ймовірності. Тоді оптимальний дискримінатор можна знайти як:

$$D^* = f^*(\xi|\eta) = \arg\max_f f(\xi_1, \dots|\eta_1, \dots) = \arg\max_f \prod_i f(\xi_i|\eta_i)$$

Згрупуємо множники для $\eta_i = 0$ та $\eta_i = 1$:

$$D^* = \arg\max_f \prod_{i,\eta=1} f(\xi_i|\eta_i = 1) \prod_{i,\eta=0} f(\xi_i|\eta_i = 0) =$$

$$= \arg\max_D \prod_{x_i{\sim}p(x)} D(x_i) \prod_{z_i{\sim}q(z)} \Big(1 - D\big(G(z_i)\big)\Big) =$$

$$= \arg\max_D \sum_{x_i{\sim}p(x)} \log D(x_i) + \sum_{z_i{\sim}q(z)} \log\Big(1 - D\big(G(z_i)\big)\Big)$$

І при прагненні розміру вибірки в нескінченність отримуємо:

$$D^* = \arg\max_D E_{x_i{\sim}p(x)} \log D\,(x_i) + E_{z_i{\sim}q(z)} \log\Big(1 - D\big(G(z_i)\big)\Big)$$

Отже, отримуємо наступний ітераційний процес:

1. Встановлюємо довільний початковий $G_0(z)$.
2. Починається *k*-та ітерація, *k=1…K*.
3. Шукаємо оптимальний для поточного генератора дискримінатор:

$$D_k = \arg\max_D E_{x_i{\sim}p(x)} \log D\,(x_i) + E_{z_i{\sim}q(z)} \log\Big(1 - D_{k-1}\big(G(z_i)\big)\Big)$$

4. Вдосконалюємо генератор, використовуючи оптимальний дискримінатор:
   $G_k = \arg\max_G E_{z\sim q(x)} \log D_k\,(G(z))$. Важливо перебувати на околиці поточного генератора. Якщо відійти далеко від значень поточного генератора, дискримінатор перестане бути оптимальним і алгоритм перестане бути вірним.
5. Завдання навчання генератора вважається вирішеним, коли $D_k(x) = 1/2$ для будь-якого *x*. Якщо процес не зійшовся, переходимо на наступну ітерацію в пункт (2).

Сумаризуємо цей алгоритм в одну формулу, яка в певному сенсі задає мінімакс гру між дискримінатором і генератором:

$$\min_G \max_D L(D, G) = E_{x\sim p(x)} \log D\,(x) + E_{z\sim q(z)} \log\Big(1 - D\big(G(z)\big)\Big)$$

Обидві функції *D, G* можуть бути представлені у вигляді нейромереж: $D(x) = D(x, \theta_1), G(z) = G(z, \theta_2)$, після чого завдання пошуку оптимальних функцій зводиться до задачі оптимізації за параметрами і можна вирішувати за допомогою традиційних методів: **backpropagation** і **SGD**. Додатково, оскільки нейромережа - це універсальний апроксиматор функцій, $G(z, \theta_2)$ може наблизити довільний розподіл ймовірностей, що вирішує питання вибору розподілу *q(z)*. Це може бути будь-який безперервний розподіл у деяких розумних рамках. Наприклад, *Uniform*(-1, 1) чи *N*(0, 1).

2.4 Знаходження цільової функції за допомогою: Середньо-квадратична похибка та перехресної ентропія

Розуміння нюансів різних функцій втрат має вирішальне значення для створення ефективних моделей машинного навчання. Вибір правильної функції втрат може значно вплинути на ефективність моделі та визначити, наскільки добре вона узагальнює дані. Звернемо увагу на дві поширені функції

знаходження втрат: середню квадратичну похибку (**Mean Squared Error, або MSE**) і втрати перехресної ентропії (**Cross-Entropy Loss**).

Функції втрат відіграють ключову роль у навчанні моделей машинного навчання, оскільки вони кількісно визначають різницю між прогнозами моделі та фактичними цільовими значеннями. Добре підібрана функція втрат дозволяє моделі вчитися на своїх помилках і періодично оновлювати свої параметри, щоб мінімізувати помилку. Це зрештою призводить до більш точних і надійних прогнозів. Ці функції втрат використовуються в машинному навчанні для завдань класифікації та регресії відповідно, щоб визначити, наскільки добре модель працює на невидимому наборі даних.

Функція втрат перехресної ентропії

Функція втрат перехресної ентропії, також відома як логарифмічна втрата, є широко використовуваною функцією втрат у машинному навчанні, зокрема для проблем класифікації. Вона кількісно визначає різницю між прогнозованим розподілом ймовірностей і фактичним або справжнім розподілом цільових класів. Функція втрат перехресної ентропії часто використовується під час навчання моделей, які знаходять оцінки ймовірності, наприклад логістичної регресії та нейронних мереж.

Функція втрат перехресної ентропії представляє різницю між прогнозованим розподілом ймовірностей, створеним моделлю, та справжнім розподілом цільових класів. Мінімізація перехресних втрат ентропії представляє мету мінімізації розбіжності або різниці між цими двома розподілами. Мінімізуючи втрату перехресної ентропії, модель заохочується виробляти оцінки ймовірності, які точно відповідають справжнім розподілам класів. Це призводить до кращих прогнозів і ефективності класифікації.

Як для задач бінарної, так і для багатокласової класифікації головною метою є мінімізація перехресної втрати ентропії, що, у свою чергу, максимізує ймовірність призначення правильних міток класу точкам вхідних даних.

Прикладом використання втрат перехресної ентропії для задач багатокласової класифікації є навчання моделі за допомогою набору даних.

Перехресна втрата ентропії для проблеми бінарної класифікації

У задачі бінарної класифікації[21],. є два можливих класи (0 і 1) для кожної точки даних. Перехресні втрати ентропії для двійкової класифікації можна визначити як:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log\big(p(y_i)\big) + (1 - y_i) \cdot \log(1 - p(y_i))$$

де, y - представляє справжню мітку класу (0 або 1);

p - представляє прогнозовану ймовірність класу 1.

Функція втрат «карає» модель сильніше, коли вона призначає низьку ймовірність справжньому класу.

Перехресна втрата ентропії для проблеми багатокласової класифікації

Для задач багатокласової класифікації, де існує більше двох можливих класів, використовується загальна форма втрати перехресної ентропії. Враховуючи класи, багатокласові перехресні втрати ентропії для однієї точки даних можна визначити як:

$$\text{Втрата} = -\sum_{i=1}^{\text{кількість виходів}} y_i \cdot \log \widehat{y_i}$$

де, y_i – представляє справжню мітку класу для класу i (1, якщо справжній клас дорівнює i, 0 інакше);

$\widehat{y_i}$ – прогнозована ймовірність класу i.

Підсумовування проходить по всіх класах. Як і в бінарному випадку, функція втрат штрафує модель, коли вона призначає низькі ймовірності справжнім класам.

Втрата середньої квадратичної похибка (MSE)

Втрата середньої квадратичної похибка – це широко поширена функція втрат у машинному навчанні та статистиці, яка вимірює середню квадратичну різницю між прогнозованими значеннями та фактичними цільовими значеннями. Це особливо корисно для задач регресії, де метою є прогнозування безперервних числових значень.

Втрата середньої квадратичної похибки обчислюється шляхом зведення до квадрату різниці між істинним значенням y і прогнозованим значенням $\hat{y}$. Беремо ці нові отримані числа (зводимо їх до квадрату), додаємо все разом щоб отримати остаточне значення, та знову розділяємо це число на y. Це буде наш остаточний результат.

Формула для розрахунку втрата середньої квадратичної похибки така:

$$MSE = \frac{1}{n}\sum(y - \hat{y})^2$$

Це дасть нам значення втрати від 0 до нескінченності, а більші значення вказують на середню квадратичну помилку.

MSE є популярним вибором для навчання регресійних моделей, оскільки він простий, інтерпретований і диференційований, що робить його придатним для алгоритмів оптимізації на основі градієнта. Однак це може бути не найкращим вибором для всіх ситуацій, оскільки він може погано обробляти певні типи розподілу. У таких випадках альтернативні функції втрат, такі як середня абсолютна похибка (MAE) або втрата Хубера, можуть бути більш доречними.

Висновок до розділу

У рамках цього розділу було сформульовано глибоке розуміння кількох ключових компонентів у сфері глибокого навчання. З'ясування принципу роботи згорткових нейронних мереж (CNN) підкреслює їх фундаментальну роль в обробці зображень і розпізнаванні образів. CNN, завдяки своїм згортковим шарам і можливостям вилучення функцій, зарекомендували себе як дуже важлива частина у сфері комп'ютерного зору.

Крім того, експозиція про Generative Adversarial Networks (GAN) дає змогу зрозуміти інноваційну парадигму генеративно-змагальних мереж. Подвійність Генератора та Дискримінатора, які беруть участь у постійній грі змагального навчання, продемонструвала свій потенціал у різних програмах, таких як генерація зображень, передача стилів і усунення шумів. Математична модель GAN, інкапсулююча взаємодію між цими двома об'єктами, забезпечує формалізовану основу для розуміння та оптимізації змагального процесу навчання.

Обговорення функцій втрат, зокрема середньоквадратичної помилки (MSE) і перехресних втрат ентропії, з'ясовує ключову роль, яку ці показники відіграють у навчанні та оцінці нейронних мереж. У той час як MSE підходить для завдань регресії, Cross-Entropy Loss перевершує проблеми класифікації, підкреслюючи їх значення в оцінці ефективності моделі.

У сукупності ці теми виходять за рамки глибокого навчання та комп'ютерного бачення, прокладаючи шлях для інноваційних розробок і нових застосувань. Принципи CNN і динаміка GAN є прикладом основи передових технологій обробки зображень і генерації даних. Оскільки наукове співтовариство продовжує використовувати ці концепції та методи, можливості для розвитку штучного інтелекту та суміжних галузей залишаються широкими та привабливими.

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ВИРІШЕННЯ ПРОБЛЕМИ ПОКРАЩЕННЯ ЯКОСТІ ЗОБРАЖЕННЯ

3.1 Огляд використаних бібліотек для вирішення поточного завдання

Для написання програмного забезпечення для покращення якості зображення з використанням нейронних мереж було вирішено використовувати мову програмування Python. Python — це високорівнева мова програмування загального призначення з динамічною строгою типізацією та автоматичним управлінням пам'яттю, орієнтована на підвищення продуктивності розробника, читання коду та його якості, а також на забезпечення переносимості написаних на ньому програм. Мова є повністю об'єктно-орієнтованою в тому плані, що все є об'єктами.

В коді написаного для створення програми було використано такий список бібліотек, який підтримує мова Python:

```
import os
import math
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import tkinter as tk
from tkinter import *
from tkinter import ttk
from tkinter import filedialog as fd
from tkinter.messagebox import showinfo
from torch.autograd import Variable
from datetime import datetime
from PIL import Image
from PIL import ImageTk
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torchvision.utils import save_image
from tqdm import tqdm
```

Звернемо увагу на деяких представників цього списку:

OS — цей модуль у Python надає функції для взаємодії з операційною системою. OS поставляється зі стандартними службовими модулями Python.

Цей модуль забезпечує портативний спосіб використання залежних від операційної системи функцій. Модулі *os* і *os.path* містять багато функцій для взаємодії з файловою системою.

Math — це вбудований модуль стандартної бібліотеки Python, який надає стандартні математичні константи та функції. Ви можете використовувати математичний модуль для виконання різноманітних математичних обчислень, таких як числові, тригонометричні, логарифмічні та експоненціальні обчислення.

NumPy — бібліотека з відкритим кодом для мови програмування Python. Можливості: підтримка багатовимірних масивів; підтримка високорівневих математичних функцій, призначених до роботи з багатовимірними масивами.

PyTorch — фреймворк машинного навчання для Python з відкритим вихідним кодом, створений на базі Torch. Використовується на вирішення різних завдань: комп'ютерний зір, обробка природної мови. Розробляється переважно групою штучного інтелекту Facebook.

Tkinter — ця структура Python надає інтерфейс до набору інструментів Tk і працює як тонкий об’єктно-орієнтований рівень поверх Tk. Набір інструментів Tk — це кросплатформна колекція «графічних елементів керування», або віджетів, для створення інтерфейсів програм.

Datetime — ця бібліотека є одним із найпоширеніших модулів для роботи з даними об’єктів дати та часу. Ви можете створювати об’єкти дати та дати й часу, переглядати діапазон дат, аналізувати та форматувати рядки дати тощо.

Python Imaging Library (PIL) — це безкоштовна додаткова бібліотека з відкритим вихідним кодом для мови програмування Python, яка додає підтримку для відкриття, обробки та збереження багатьох різних форматів файлів зображень.

Tqdm — це бібліотека Python для додавання панелі виконання. Це дозволяє налаштувати та відобразити панель прогресу з показниками, які ви хочете відстежувати. Його простота використання та універсальність робить його ідеальним вибором для відстеження експериментів машинного навчання.

3.2 Набір даних для навчання нейронної мережі

Використання набору даних у навчанні Generative Adversarial Network (GAN) є обов'язковим з кількох причин. Він надає різноманітний набір прикладів, що дозволяє GAN вивчати закономірності, особливості та мінливості, наявні в даних реального світу. Добре підібраний набір даних допомагає узагальнити модель, уникаючи таких проблем, як згортання режиму, і сприяючи стабільному навчанню. Крім того, набір даних служить еталоном для оцінки продуктивності GAN, гарантуючи, що згенеровані зразки відповідають характеристикам навчальних даних. Етичні міркування також підкреслюють важливість відповідального пошуку джерел і курування наборів даних для пом'якшення упереджень і етичних проблем у створених результатах. По суті, репрезентативний набір даних формує основу для здатності GAN створювати автентичні, різноманітні та етично обґрунтовані результати.

Для навчання даної нейронної мережі було вирішено використати набір даних зображень DIV2K.

DIV2K — це популярний набір даних із високою роздільною здатністю для одного зображення, який містить 1000 зображень із різними сценами та розділений на 800 для навчання, 100 для перевірки та 100 для тестування. Він був зібраний для NTIRE2017 і NTIRE2018 Super-Resolution Challenges, щоб заохотити дослідження надвисокої роздільної здатності зображень із більш реалістичним погіршенням. Цей набір даних містить зображення низької роздільної здатності з різними типами погіршень.

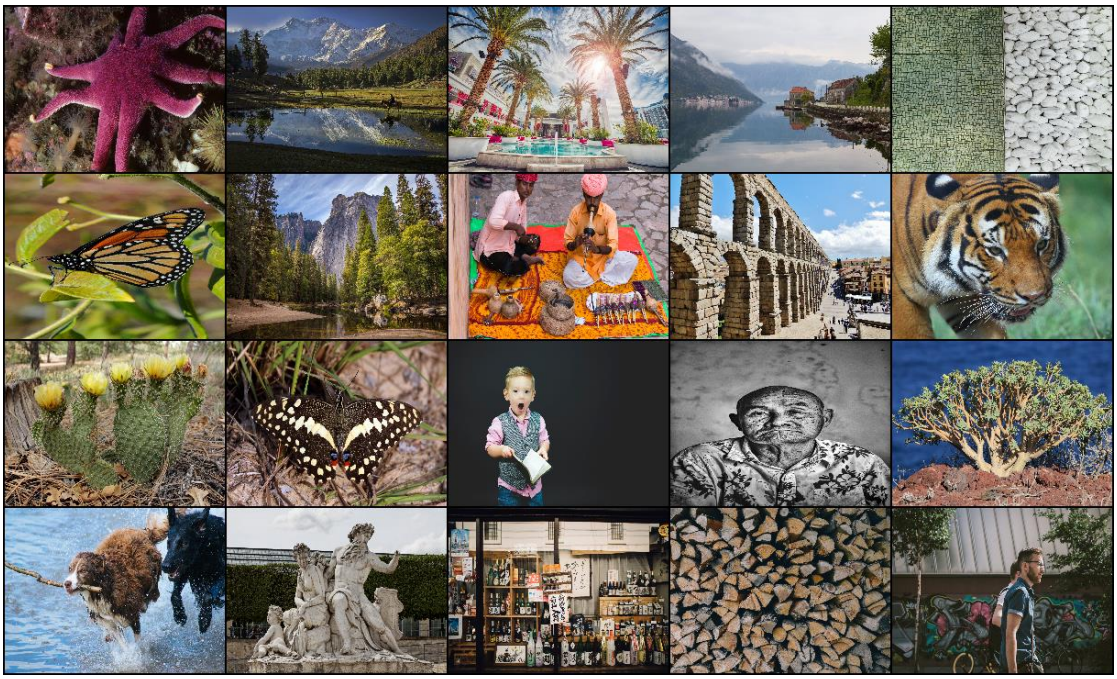

Рис 3.2.1 Приклад зображень з набору даних DIV2K

В коді програми реалізовано клас CustomImageDataset. Цей клас створений для зручної роботи із зображеннями, де у нас є пари низької та високої роздільної здатності. А саме, фотографії одних і тих же сцен у двох папках - однієї із зображеннями низької роздільної здатності, а іншої – високої.

Реалізація класу для роботи з набором даних:

```
class CustomImageDataset(Dataset):
    def __init__(self, low_res_folder, high_res_folder, transform=None):
        self.low_res_folder = low_res_folder
        self.high_res_folder = high_res_folder
        self.transform = transform

        # Отримаємо список файлів в папках
        self.low_res_files = sorted(os.listdir(self.low_res_folder))
        self.high_res_files = sorted(os.listdir(self.high_res_folder))

    def __len__(self):
        return len(self.low_res_files)

    def __getitem__(self, idx):
        # Отримаємо шлях до зображення
        low_res_img_path = os.path.join(self.low_res_folder, self.low_res_files[idx])
        high_res_img_path = os.path.join(self.high_res_folder, self.high_res_files[idx])

        # Відкриваємо зображення
        low_res_img = Image.open(low_res_img_path)
        high_res_img = Image.open(high_res_img_path)

        if self.transform:
            low_res_img = self.transform(low_res_img)
            high_res_img = self.transform(high_res_img)

        #print(f"Loaded images for index {idx}, low_res shape: {low_res_img.shape}, high_res shape: {high_res_img.shape}")

        return low_res_img, high_res_img
```

Цей клас **CustomImageDataset** створений для роботи із зображеннями низької та високої роздільної здатності. Розглянемо як він влаштований:

1. Конструктор **__init__**
   - Приймає три параметри: шлях до папки із зображеннями низької роздільної здатності (**low_res_folder**), шлях до папки із зображеннями високої роздільної здатності (**high_res_folder**), та опціональне перетворення (**transform**).
   - Зберігає передані значення атрибути класу.
   - Отримує та сортує список файлів із папок **low_res_folder** та **high_res_folder**.
2. Метод **__len__**
   - Повертає кількість файлів у папці із зображеннями низької роздільної здатності (припускаючи, що кількість файлів в обох папках однакова).
3. Метод **__getitem__**
   - Приймає індекс (**idx**).
   - Збирає шляхи до зображень низької та високої роздільної здатності.
   - Відкриває зображення за допомогою бібліотеки PIL (**Image.open**).
   - Застосовує трансформації, якщо вони надані.
   - Повертає кортеж зображень низької та високої роздільної здатності.
4. Трансформації (**transform**)
   - Це послідовність трансформацій, що застосовуються до кожного зображення.
   - Включає зміну розміру зображення, перетворення в тензор і нормалізацію.

3.3 Блок із залишковим з'єднанням та блок підвищення дискретизації. Клас Генератора та Дискримінатора.

Residual Block та Upsample Block

Ці два блоки представляють собою ключові компоненти моделі архітектури GAN для завдання створення зображення високої роздільної якості. Давайте розглянемо їх призначення і структуру:

1. Residual Block (Блок з остаточним з'єднанням)

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(channels)
        self.prelu = nn.PReLU()
        self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(channels)

    def forward(self, x):
        residual = self.conv1(x)
        residual = self.bn1(residual)
        residual = self.prelu(residual)
        residual = self.conv2(residual)
        residual = self.bn2(residual)

        return x + residual
```

Residual Block є елементом глибоких нейронних мереж, покликаний справлятися з проблемою загасаючих градієнтів. Він використовує залишкове з'єднання, дозволяючи шарам, що навчаються, відображати зміни щодо вхідного сигналу. Два послідовні згорткові шари з батч-нормалізацією та функцією активації PReLU формують "залишкову карту", яка додається до вхідного тензора. Такий пристрій блоку сприяє ефективнішому навчанню глибоких моделей.

Структура блоку:

- Вхідні дані проходять через два згорткові шари з батч-нормалізацією (BN) та функцією активації PReLU.
- Залишковий блок додає оригінальні вхідні дані до виходу, що формує остаточний вихід блоку.

Батч-нормалізація в Residual Block

Батч-нормалізація - це техніка в глибокому навчанні, яка сприяє стабілізації та прискоренню навчання нейронних мереж. У контексті Residual Block вона застосовується після кожного згорткового шару. Цей процес включає наступні кроки:

- **Нормалізація по батчу (Batch Normalization):** Вхідні дані піддаються стандартній процедурі нормалізації, віднімаючи середнє значення батча і роблячи розподіл на стандартне відхилення батча. Це допомагає стабілізувати розподіл активацій та прискорює навчання, запобігаючи згасанню або вибуху градієнтів.
- **Масштабування і зсув (Scaling and Shifting):** Додаткові параметри, що навчаються (γ і β) масштабують і зсувають нормалізовані значення. Це дає мережі можливість самостійно керувати амплітудою активацій, збагачуючи їхню представницьку здатність.

В Residual Block батч-нормалізація застосовується до залишкової "карти" після першої згортки і до "залишкової" карти перед додаванням до вхідного тензора. Це допомагає більш ефективно передавати градієнти, забезпечуючи стабільність навчання зі збільшенням глибини мережі.

2. Upsample Block (Блок підвищення дискретизації)

```
class UpsampleBLock(nn.Module):
    def __init__(self, in_channels, up_scale):
        super(UpsampleBLock, self).__init__()
        self.conv = nn.Conv2d(in_channels, in_channels * up_scale ** 2, kernel_size=3, padding=1)
        self.pixel_shuffle = nn.PixelShuffle(up_scale)
        self.prelu = nn.PReLU()

    def forward(self, x):
        x = self.conv(x)
        x = self.pixel_shuffle(x)
        x = self.prelu(x)
        return x
```

Upsample Block збільшує просторову роздільну здатність зображення. Це досягається через згортковий шар, який збільшує кількість каналів, а потім застосовується PixelShuffle для збільшення просторової роздільної здатності.

Функція активації PReLU додає нелінійність, сприяючи навчанню та надаючи зображенню природніший зовнішній вигляд. Цей блок дозволяє генератору збільшувати роздільну здатність зображення, зберігаючи важливі структурні характеристики.

Структура блоку:

- Вхідні дані проходять через згортковий шар, який збільшує кількість каналів.
- PixelShuffle переупорядковує пікселі вхідного зображення, збільшуючи його розмір відповідно до заданого масштабу.
- PReLU застосовує нелінійність до отриманого зображення.

Обидва ці блоки активно використовують у генераторах GAN підвищення якості згенерованих зображень. Residual блоки полегшують навчання, а Upsample блоки сприяють генерації більш детальних зображень високої роздільної здатності.

Генератор та Дискримінатор

1. Генератор

Генератор в архітектурі GAN виконує критичну роль у синтезі зображень високої роздільної здатності. Його завдання - трансформувати зображення низької роздільної здатності в більш детальні та якісні зображення.

```
class Generator(nn.Module):
    def __init__(self, scale_factor=1):
        upsample_block_num = int(math.log(scale_factor, 2))

        super(Generator, self).__init__()
        self.block1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=9, padding=4),
            nn.PReLU()
        )
        self.block2 = ResidualBlock(64)
        self.block3 = ResidualBlock(64)
        self.block4 = ResidualBlock(64)
        self.block5 = ResidualBlock(64)
        self.block6 = ResidualBlock(64)
        self.block7 = nn.Sequential(
```

```
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64)
        )
        block8 = [UpsampleBLock(64, 2) for _ in range(upsample_block_num)]
        block8.append(nn.Conv2d(64, 3, kernel_size=9, padding=4))
        self.block8 = nn.Sequential(*block8)

    def forward(self, x):
        block1 = self.block1(x)
        block2 = self.block2(block1)
        block3 = self.block3(block2)
        block4 = self.block4(block3)
        block5 = self.block5(block4)
        block6 = self.block6(block5)
        block7 = self.block7(block6)
        block8 = self.block8(block1 + block7)

        return torch.tanh(block8)
```

Цей процес досягається через декілька ключових компонентів:

1. **Вилучення ознак:** Після введення зображення блоки 2-6 (Residual Blocks) дозволяють генератору витягувати важливі ознаки із зображення. Це важливо для того, щоб зберегти структуру та зміст.

2. **Residual Blocks:** Ці блоки є глибокими шарами, що забезпечують навчання "залишковим" функціям, що сприяє ефективному навчанню змін, які слід внести до зображення.

3. **Upsample Blocks:** Блок 8 містить Upsample-блоки, які збільшують роздільну здатність зображення. Це дозволяє генератору відновлювати деталі та структуру зображення.

2. Дискрмінатор

Дискримінатор виконує роль класифікатора, поділяючи зображення на "реальні" та "згенеровані". Його архітектура орієнтована на ефективне розрізнення між цими двома типами зображень.

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2),
```

```
            nn.Conv2d(64, 64, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
```

```
            nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),

            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2),

            nn.Conv2d(256, 256, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2),

            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2),

            nn.Conv2d(512, 512, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2),

            nn.AdaptiveAvgPool2d(1),
            nn.Conv2d(512, 1024, kernel_size=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(1024, 1, kernel_size=1)
        )

    def forward(self, x):
        batch_size = x.size(0)
        return torch.sigmoid(self.net(x).view(batch_size))
```

Структура Дискримінатора:

- **Згорткові шари:** Шари згортки виконують операції виділення ознак. Вони допомагають дискримінатору розпізнавати рівні абстракції зображення.
- **LeakyReLU:** Використання LeakyReLU у функції активації сприяє запобіганню згасання градієнта, що покращує навчання дискримінатора.
- **Адаптивний пулінг:** Адаптивний пулінг надає фіксоване зображення, скорочуючи його розмірність перед фінальною класифікацією.
- **Шар із повнозв’язаними вузлами:** Шар із повнозв’язаними вузлами стискає просторову інформацію в єдине значення, що представляє ймовірність того, що зображення є "реальним".

3.4 Функція навчання моделі GAN для вирішення проблеми покращення якості зображення

У цій функції реалізовано процес навчання моделі GAN з використанням генератора та дискримінатора. Навчання відбувається протягом декількох епох, де кожна епоха включає навчання та тестування моделей.

```
def train_model(generator, discriminator, dataloader, test_dataloader, num_epochs, device):
    for epoch in range(num_epochs):

        #__Training_____________
        gen_loss, disc_loss = 0, 0
        tqdm_bar = tqdm(dataloader, desc=f'Training Epoch {epoch}', total=int(len(dataloader)))

        for batch_idx, imgs in enumerate(tqdm_bar):
            generator.train()
            discriminator.train()

            # Переносимо дані на пристрій (GPU або CPU)
            imgs_lr = Variable(imgs[0].to(device))
            imgs_hr = Variable(imgs[1].to(device))


            ### Навчання Генератора
            optimizer_G.zero_grad()
            gen_hr = generator(imgs_lr)

            # Створюємо мітки для Дискримінатора
            valid = Variable(Tensor(imgs_lr.size(0)).fill_(1.0), requires_grad=False)
            fake = Variable(Tensor(imgs_lr.size(0)).fill_(0.0), requires_grad=False)

            loss_GAN = criterion_GAN(discriminator(gen_hr), valid)
            loss_content = criterion_content(gen_hr, imgs_hr)
            loss_G = loss_content + 1e-3 * loss_GAN
            loss_G.backward()
            optimizer_G.step()


            ### Навчання Дискримінатра
            optimizer_D.zero_grad()
            loss_real = criterion_GAN(discriminator(imgs_hr), valid)
            loss_fake = criterion_GAN(discriminator(gen_hr.detach()), fake)
            loss_D = (loss_real + loss_fake) / 2
            loss_D.backward()
            optimizer_D.step()

            gen_loss += loss_G.item()
            disc_loss += loss_D.item()
            tqdm_bar.set_postfix(gen_loss=gen_loss / (batch_idx + 1), disc_loss=disc_loss / (batch_idx + 1))

        #__Testing_____________
        gen_loss, disc_loss = 0, 0
        tqdm_bar = tqdm(test_dataloader, desc=f'Testing Epoch {epoch}', total=int(len(test_dataloader)))

        for batch_idx, imgs in enumerate(tqdm_bar):
            generator.eval()
            discriminator.eval()

            # Переносимо дані на прилад (GPU або CPU)
            imgs_lr = Variable(imgs[0].to(device))
            imgs_hr = Variable(imgs[1].to(device))
```

```
        ### Навчання Генератора
        optimizer_G.zero_grad()
        gen_hr = generator(imgs_lr)

        # Створюємо мітки для Дискримінатора
        valid = Variable(Tensor(imgs_lr.size(0)).fill_(1.0), requires_grad=False)
        fake = Variable(Tensor(imgs_lr.size(0)).fill_(0.0), requires_grad=False)

        loss_GAN = criterion_GAN(discriminator(gen_hr), valid)
        loss_content = criterion_content(gen_hr, imgs_hr)
        loss_G = loss_content + 1e-3 * loss_GAN
        loss_G.backward()
        optimizer_G.step()
```

```
        ### Навчання Дискрімінатора
        optimizer_D.zero_grad()
Изменение — не сохранено
        l = criterion_GAN(discriminator(imgs_hr), valid)
        loss_fake = criterion_GAN(discriminator(gen_hr.detach()), fake)
        loss_D = (loss_real + loss_fake) / 2
        loss_D.backward()
        optimizer_D.step()

        gen_loss += loss_G.item()
        disc_loss += loss_D.item()
        tqdm_bar.set_postfix(gen_loss=gen_loss / (batch_idx + 1), disc_loss=disc_loss / (batch_idx + 1))
```

```
    # Зберігаємо зображення після кожної епохи
    save_generated_images(generator, epoch)

    gen_losses.append(gen_loss/len(test_dataloader))
    disc_losses.append(disc_loss/len(test_dataloader))

    # Зберігаємо параметри моделі для подальшого використання
    if np.argmin(gen_losses) == len(gen_losses)-1:
        torch.save(generator.state_dict(), f"saved_models/generator_{size_in}_to_{size_out}.pth")
        torch.save(discriminator.state_dict(), f"saved_models/discriminator_{size_in}_to_{size_out}.pth")
        print("Save model checkpoints")
```

Процес навчання моделі відбувається так:

1. **Підготовка змінних:** Виймаються батчі із завантажувача даних, і дані переносяться на вказаний пристрій (GPU або CPU).
2. **Навчання генератора:**

- Обнулюються градієнти оптимізатора генератора.
- Отримаємо дані (**gen_hr**) з генератора на основі зображення з низькою роздільною здатністю (**imgs_lr**).
- Створюються мітки (**valid** та **fake**) для навчання дискримінатора.
- Обчислюються функції втрат для генератора: GAN loss (**loss_GAN**) та content loss (**loss_content**).
- Загальна функція втрат генератора (**loss_G**) складається з content loss та GAN loss.
- Здійснюється зворотне поширення помилки (**backward()**) та оновлення параметрів генератора.

3. **Навчання дискримінатора:**

- Обнулюються градієнти оптимізатора дискримінатора.
- Обчислюються GAN loss для реальних та згенерованих зображень (**loss_real** та **loss_fake** відповідно).
- Загальна функція втрат для дискримінатора (**loss_D**) є середнім між **loss_real** та **loss_fake**.
- Здійснюється зворотне поширення помилки та оновлення параметрів дискримінатора.

4. **Тестування:**

Процес тестування аналогічний до навчання, але моделі знаходяться в режимі **eval**. Це дозволяє оцінити продуктивність моделей без обчислення градієнтів.

Збереження результатів:

Згенеровані зображення зберігаються після кожної епохи навчання. Також зберігаються параметри моделей, якщо досягнуто кращого значення функції втрат на тестовому наборі.

Ця функція є ключовою частиною навчання GAN і включає послідовні кроки навчання і тестування, а також збереження результатів для подальшого використання.

Також перед навчанням моделі потрібно задати певні характеристики для функції втрат та оптимізаторів.

Визначення функцій втрат

У цій ділянці коду визначаються функції втрат, які використовуватимуться під час навчання генератора і дискримінатора.

```
# Визначення функцій втрат
criterion_GAN = nn.MSELoss()
criterion_content = nn.L1Loss()
```

criterion_GAN = nn.MSELoss() – ця функція втрат Mean Squared Error (MSE) між прогнозами, що генеруються генератором, і фактичними мітками (0 для згенерованих зображень, 1 для реальних). Вона висловлює, як сильно вихід генератора відрізняється від очікуваного.

criterion_content = nn.L1Loss() – ця функція втрат представляє L1 Loss між виходами генератора та зображеннями з високою роздільною здатністю. Вона оцінює абсолютну різницю між згенерованими і цільовими зображеннями, сприяючи збереженню деталей.

Визначення оптимізаторів

Далі йде визначення оптимізаторів для генератора та дискримінатора.

```
# Визначення оптимізаторів для генератора та дискримінатора
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(b1, b2))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(b1, b2))
```

optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(b1, b2)) – Генератор оптимізується з використанням алгоритму Adam. Цей оптимізатор пристосовується до динаміки навчання у процесі навчання та використовує градієнти для оновлення параметрів генератора.

optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(b1, b2)) – Дискримінатор також оптимізується з використанням алгоритму Adam з тими самими параметрами.

Завантаження попередньо навчених моделей

```
# Завантаження попередньо навчених моделей
if load_pretrained_models:
    generator.load_state_dict(torch.load(f"./saved_models/generator_{size_in}_to_{size_out}.pth"))
    discriminator.load_state_dict(torch.load(f"./saved_models/discriminator_{size_in}_to_{size_out}.pth"))
```

Якщо встановлено прапорець **load_pretrained_models**, попередньо навчені моделі генератора та дискримінатора завантажуються зі збережених файлів. Це корисно, якщо ви хочете продовжити навчання або використовувати заздалегідь навчені моделі для генерації зображень.

3.5 Інтерфейс програмного забезпечення та приклад використання

Для того щоб відкрити програму потрібно спочатку задати шлях до фалу в терміналі за допомогою команди: **cd <шлях/до/файлу>**, потім запустити його за допомогою команди: **python3 GANver5.py.**

Після виконаних інструкція користувача зустрічає вікно під назвою «**GAN program**».

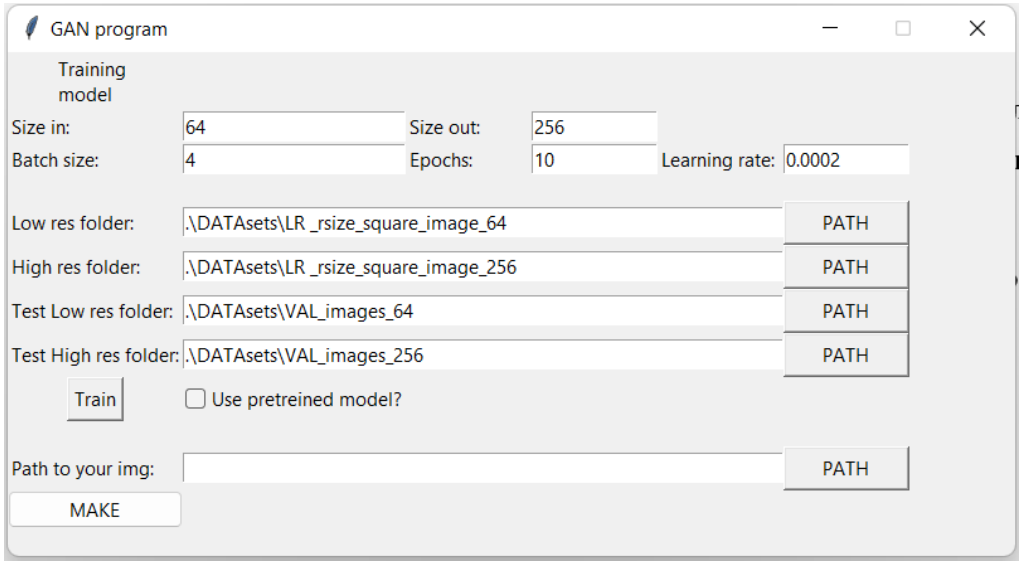

Рис 3.5.1 Інтерфейс програми «GAN program»

В даному вікні ми можемо поміти декілька полів для вводу тексту. В верхній частині програми знаходяться параметри навчання моделі:

- **Size in** – розмір зображення з низькою роздільною якістю n×n (Наприклад, 64×64);
- **Size out** – розмір зображення яке ми збираємося отримати n×n (Наприклад, 256×256);
- **Batch size** – розмір батчу;
- **Epochs** – кількість епох навчання моделі;
- **Learning rate** – темп навчання моделі.

Нижче розташовані поля та кнопка **«PATH»** до кожного з них. При натисканні на яку відкривається менеджер файлів, щоб користувач міг вказати

шлях до папок з власним набором даних до кожного поля відповідно(Наприклад, DIV2K, як в нашому випадку).

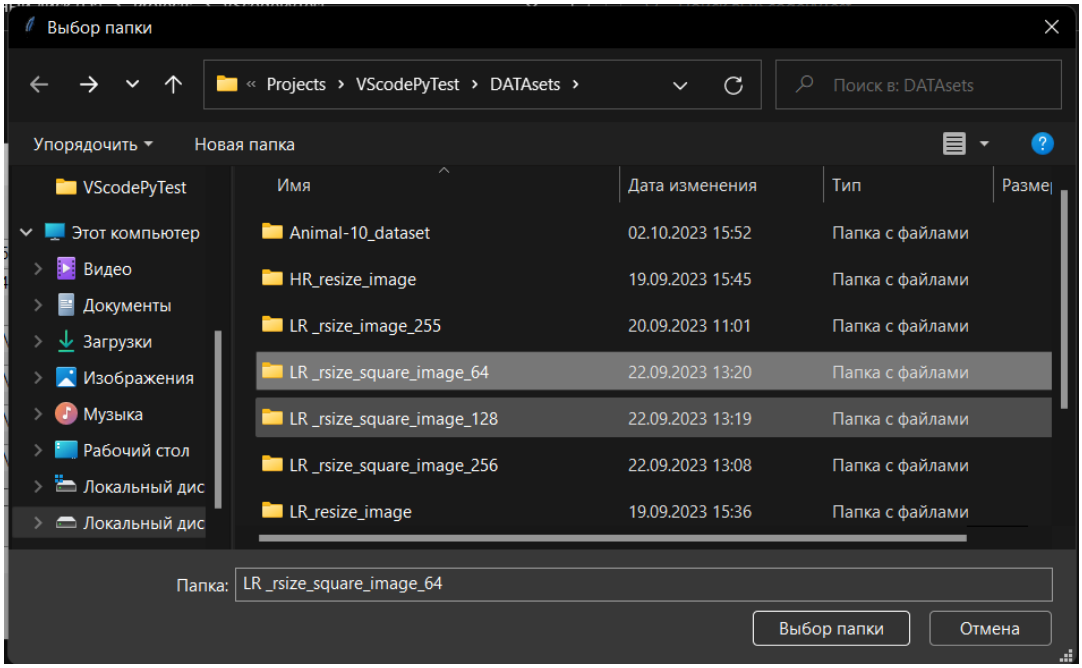

Рис 3.5.2 Менеджер файлів

Далі ми можемо помітити кнопку **«Train»** та маркер з написом **«Use pretreined model»**. Натискання даної кнопки викликає функцію **train_model()**, а положення маркера за те, чи буде під час виконання процесу навчання використовуватися збережені раніше параметри моделі.

Рис 3.5.3 Процес навчання моделі

Після кожної ітерації навчання зберігаються приклади згенерованих зображень для порівняння якості (Рис 3.5.4). Також автоматично зберігаються параметри моделі, якщо вони задовольняють очікування.

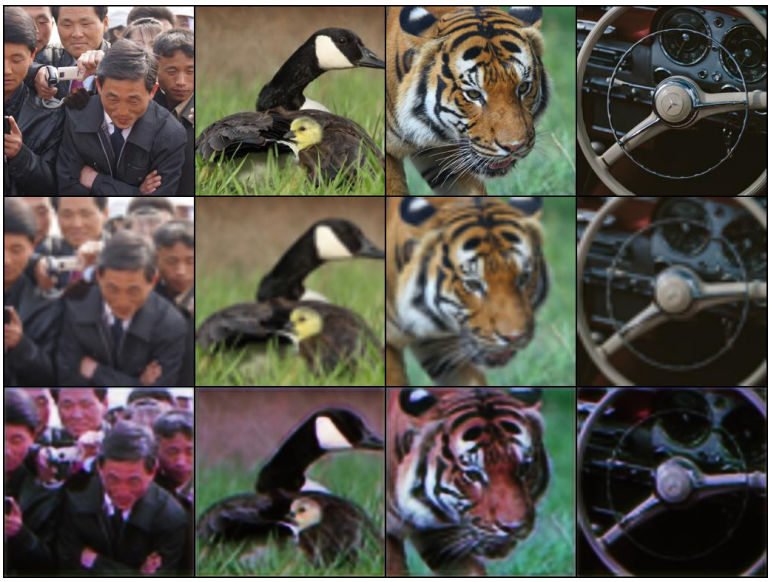

Рис 3.5.4 Приклад генерації зображення під час виконання навчання моделі

В самому низу вікна програми розташовано поле та відповідна кнопка **«PATH»** за допомогою якої користувач може вказати зображення, яке він бажає покращити. Натиснувши кнопку **«Make»**, виконується функція **generate_and_save_images()**, після чого відкриється поверх ще одне вікно, під назвою **«Genereted image»**, в якому буде вхідне зображення порівняно з зображенням, яке було згенеровано на основі навчання мережі GAN.

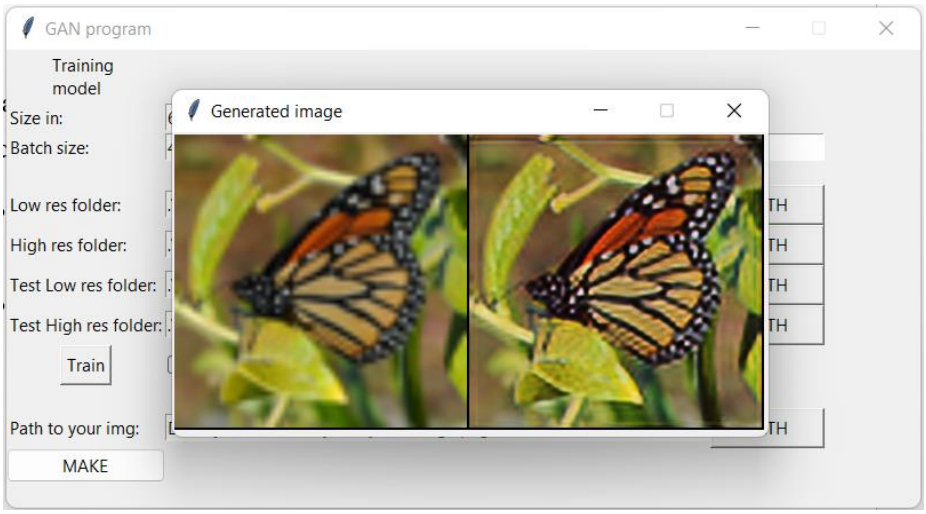

Рис 3.5.5 Вивід вікна разом з згенерованим зображенням(з 64×64, в 256×256)

ВИСНОВКИ

Дана кваліфікаційна робота є комплексним дослідженням та реалізацією методів покращення якості зображень з використанням нейронних мереж, зокрема, генеративно-змагальних мереж (GAN).

Проведено детальний аналіз нейронних мереж та їх структури, виявлено ключові аспекти формування та розвитку в контексті покращення якості зображень. Теоретичний аналіз комп'ютерної зору розкриває важливість та перспективи впровадження нейромережевих підходів до цієї області. В даному руслі фреймворк PyTorch виділяється як потужний засіб для розробки та навчання нейронних мереж.

Проведено огляд програмної області, розглядаючи принципи роботи згорткових мереж (CNN) і математичну модель генеративно-змагальної мережі (GAN). Увага приділяється функціям генератора та дискримінатора, а також методам вимірювання помилок, таким як середньоквадратична похибка та бінарна перехресна ентропія.

Проведена практична реалізація програмного забезпечення для вирішення проблеми покращення якості зображень. Розглянуто використовувані бібліотеки, обрано набір для навчання, розроблено та описано класи генератора та дискримінатора із застосуванням блоків з остаточним з'єднанням та блоків підвищення дискретизації. Наведено функцію навчання GAN та реалізовано користувальницький інтерфейс програмного забезпечення з прикладами використання.

Наприкінці можна відзначити, що ця кваліфікаційна робота, яка дає теоретичний огляд досліджуваної проблеми, та й успішно реалізує програмне забезпечення, здатне вирішувати це завдання. Отримані результати підкреслюють ефективність використання генеративно-змагальних мереж для покращення якості зображень, а розроблений інтерфейс робить це рішення доступним та зручним для застосування в реальних умовах.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

1. ШНМ – штучна нейронна мережа
1. DNN (Deep Neural Network) – мережа глибокого навчання
2. CNN (Convolutional Neural Network) – згорткова нейронна мережа
3. GAN (Generative Adversarial Network) – генеративно-змагальна мережа
4. GPU (Graphics Processing Unit) – графічний процесор
5. TPU (Tensor Processing Units) – тензорний процесор
6. ШІ – штучний інтелект
7. RNN (Recurrent Neural Network) – рекурентна нейронна мережа
8. ICR (Intelligent Character Recognition) – інтелектуальне розпізнавання символів
9. API ()
10. NLP (Neural Network Processing) – обробка природної мови
11. LR (Reinforcing Learning) – навчання з підкріпленням
12. SGD (Stochastic Gradient Descent) – стохастичний градієнтний спуск
13. MSE (Mean Square Error) – середньоквадратична похибка

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. С. О. Субботін, Нейронні мережі: теорія та практика, Навчальний посібник. Житомир, Видавець: О. О. Євенок 2020

2. Електронне мережне навчальне видання: Методи глибокого навчання на різнорідних даних. Київ, КПІ ім. Ігоря Сікорського 2022

3. Шапиро Л. Комп'ютерний зір. Л. Шапиро, Дж. Стокман. (пер. с англ.), 2006. – 752 с.

4. Форсайт Дэвид А., Понс Жан. Комп'ютерний зір. Сучасний підхід. (пер. с англ), 2004. – 928 с.

5. Л. М. Добровська. І. А. Добровська. Теорія та практика нейронних мереж. Навчальний посібник. Київ НТУУ «КПІ» 2015

6. Лейт Ахмед Мустафа Аль Равашдех, Руженцев І. Оцінка похибки динамічних нейронних мереж для вимірювальних систем. 2018

7. Логвін А.О. Типи генеративних нейронних мереж. DOI https://doi.org/10.32838/2663-5941/2021.1-1/17

8. Функції активації: ступінчаста, лінійна, сигмоїда, ReLU та Tanh. https://robotdreams.cc/uk/blog/327-funkciji-aktivaciji-stupinchasta-liniyna-sigmojida-relu-ta-tanh

9. Jonathan Ho, Ajay Jain, Pieter Abbeel. Denoising Diffusion Probabilistic Models. 34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

10. Chitwan Saharia, Jonathan Ho, William Chan, Tim Salimans, David J. Fleet, Mohammad Norouzi. Image Super-Resolution via Iterative Refinement. Google Research, Brain Team. 30 Jun 2021

11. Вікіпедія - Штучний нейрон. https://uk.wikipedia.org/wiki/Штучний_нейрон

12. Вікіпедія - Середньоквадратична похибка. https://uk.wikipedia.org/wiki/Середньоквадратична_похибка

13. Вікіпедія – Перехресна ентропія. https://uk.wikipedia.org/wiki/Перехресна_ентропія

14. PYTORCH DOCUMENTATION. https://pytorch.org/docs/stable/index.html

15. DIVE INTO DEEP LEARNING. https://d2l.ai/

16. Patrice Y. Simard, Dave Steinkraus, John C. Platt. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. https://www.researchgate.net/profile/John-Platt-2/publication/2880624_Best_Practices_for_Convolutional_Neural_Networks/links/00b49524c79b1afb07000000/Best-Practices-for-Convolutional-Neural-Networks.pdf

17. Convolution arithmetic. https://github.com/vdumoulin/conv_arithmetic

18. Orquestra Documentation. Generative Adversarial Networks (GANs). https://docs.orquestra.io/docs/qml-core/tutorials/model_catalog/gan_guide.html

19. Деніел Нельсон. Що таке Generative Adversarial Network (GAN)? https://www.unite.ai/uk/what-is-a-generative-adversarial-network-gan/

20. Goodfellow, Ian; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde-Farley, David; Ozair, Sherjil; Courville, Aaron; Bengio, Joshua (2014). «Generative Adversarial Networks». https://arxiv.org/abs/1406.2661

21. Salimans, Tim; Goodfellow, Ian; Zaremba, Wojciech; Cheung, Vicki; Radford, Alec; Chen, Xi (2016). «Improved Techniques for Training GANs». https://arxiv.org/abs/1606.03498

22. Goodfellow, Ian J.; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde-Farley, David; Ozair, Sherjil; Courville, Aaron; Bengio, Yoshua (2014). «Generative Adversarial Networks». https://arxiv.org/abs/1406.2661

23. Luc, Pauline; Couprie, Camille; Chintala, Soumith; Verbeek, Jakob (25 листопада 2016). Semantic Segmentation using Adversarial Networks. NIPS Workshop on Adversarial Training, Dec , Barcelona, Spain 2016. https://arxiv.org/abs/1611.08408

24. Andrej Karpathy, Pieter Abbeel, Greg Brockman, Peter Chen, Vicki Cheung, Rocky Duan, Ian Goodfellow, Durk Kingma, Jonathan Ho, Rein Houthooft, Tim

Salimans, John Schulman, Ilya Sutskever, And Wojciech Zaremba. Generative Models. OpenAI. Архів оригіналу за 22 квітня 2021.

25. Li, Wei; Gauci, Melvin; Gross, Roderich (July 6, 2013). A Coevolutionary Approach to Learn Animal Behavior Through Controlled Interaction. Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation (GECCO 2013). Amsterdam, The Netherlands: ACM. с. 223–230. https://dl.acm.org/doi/10.1145/2463372.2465801

26. Gross, Roderich; Gu, Yue; Li, Wei; Gauci, Melvin (December 6, 2017). Generalizing GANs: A Turing Perspective. Proceedings of the Thirty-first Annual Conference on Neural Information Processing Systems (NIPS 2017). Long Beach, CA, USA. с. 1–11. https://web.archive.org/web/20180630105112/https://nips.cc/Conferences/2017/Schedule?showEvent=9402

27. Mean Squared Error (MSE). www.probabilitycourse.com.