Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра автоматизації, комп'ютерних наук і технологій

# КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти – бакалавр
за освітньо-професійною програмою
«Комп'ютерні науки»
зі спеціальності
122 – Комп'ютерні науки

тема роботи:

***«Застосування згорткових мереж для дослідження алгоритмів розпізнавання образів»***

Виконав студент гр. КН-21 _____ Мадай В. О.

Керівник _____ Кумченко Ю. О.

Нормоконтроль _____ Маринич І. А.

Завідувач кафедри _____ Рубан С. А.

Кривий Ріг – 2025

**КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**Факультет:** *інформаційних технологій*

**Кафедра:** *автоматизації, комп'ютерних наук і технологій*

**Ступінь вищої освіти:** *Бакалавр*

**Спеціальність:** *122 – Комп'ютерні науки*

**ЗАТВЕРДЖУЮ**

Зав. кафедрою: *к.т.н. Рубан С.А.*

_____

« *29* » *січня* 2025 р.

**ЗАВДАННЯ**

**на кваліфікаційну роботу бакалавра**

студентові групи КН-21 *Мадай Владлену Олександровичу*

**1. Тема кваліфікаційної роботи:** *«Застосування згорткових мереж для дослідження алгоритмів розпізнавання образів»*

затверджено наказом по університету № *254с* від *13.05.2025 р.*

**2. Термін здачі кваліфікаційної роботи:** *06.06.2025 р.*

**3. Склад кваліфікаційної роботи:** *Пояснювальна записка обсягом 69с., додатки, презентація у Microsoft PowerPoint (13 слайдів) в електронному та друкованому вигляді*

**4. Консультанти кваліфікаційної роботи:**

Розділ 1-2 доц. Кумченко Ю. О.

Нормоконтроль доц. Маринич І. А.

**5. Календарний план:**

| № | Етапи роботи | Термін виконання |
|---|---|---|
| 1 | *Вступ* | *01.04.25* |
| 2 | *Розділ 1* | *05.04.25* |
| 3 | *Розділ 2* | *01.05.25* |
| 4 | *Висновки* | *25.05.25* |
| 5 | *Оформлення кваліфікаційної роботи* | *28.05.25* |
| 6 | *Підготовка презентації та графічного матеріалу* | *20.05.25* |
| 7 | *Підготовка доповіді до захисту* | *05.06.25* |

**6. Дата видачі завдання:** *28.01.2025р.*

**Керівник** _____ / Кумченко Ю. О./

**7. Запевнення:** *Я, Мадай Владлен Олександрович, запевняю, що ця кваліфікаційна робота виконана самостійно, не містить академічного плагіату, фабрикації, фальсифікації. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.*

*Із чинним Положенням про академічну доброчесність Криворізького національного університету ознайомлений.*

*Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі умисних порушень робота не допускається до захисту або оцінюється незадовільно.*

**Студент** _____ / Мадай В. О./

# АНОТАЦІЯ

Мадай В. О. Застосування згорткових мереж для дослідження алгоритмів розпізнавання образів : кваліфікаційна робота бакалавра : 122 – Комп’ютерні науки. Кривий Ріг. Криворізький національний університет, 2025. 69 с.

Актуальність даної теми полягає в тому, що розпізнавання образів залишається важливою частиною розвитку штучного інтелекту, і удосконалення алгоритмів для ефективнішої роботи з зображеннями є важливим кроком у розвитку сучасних технологій.

Метою роботи є дослідження застосування згорткових нейронних мереж для розв'язання задач розпізнавання образів, зокрема класифікації зображень, аналізу ефективності різних архітектур та порівняння результатів навчання моделей на різних наборах даних.

У першому розділі було розглянуто основні концепції, які складають основу для розуміння нейронних мереж, а також здійснено огляд ключових архітектур згорткових нейронних мереж (CNN). Для подальшого аналізу були обрані ResNet і Inception: ці архітектури дають змогу досягати високої продуктивності завдяки новим підходам таким як залишкові з'єднання та інцепшн-блоки, і є актуальними для більш складних та масштабних завдань

У другому розділі було було проведено всебічний аналіз двох сучасних архітектур згорткових нейронних мереж - *ResNet34* та *Inception*. Було розглянуто їх внутрішню будову, ключові особливості та переваги у вирішенні задач класифікації зображень. Таким чином, було підтверджено доцільність використання як *ResNet34*, так і *Inception* у задачах комп’ютерного зору, з можливістю подальшої адаптації під інші типи даних і задачі.

АЛГОРИТМ, АРХІТЕКТУРА МЕРЕЖІ, ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ, РОЗПІЗНАВАННЯ ОБРАЗІВ, ЗГОРТКОВИЙ ШАР, CNN, RESNET, INCEPTION

ЗМІСТ

# ВСТУП

Розпізнавання образів - це процес, спрямований на автоматичне виявлення закономірностей та зв'язків у вхідних даних. Після ідентифікації певних закономірностей кожному образу присвоюється відповідна мітка або результат. Структура вхідних даних визначається залежно від основної мети цього процесу і може включати будь-який тип подій, вимірів, процесів або спостережень, які необхідно класифікувати або оцінити.

Попит на застосування методів розпізнавання образів зростає разом з розвитком потужніших процесорів. Виникнення нових концепцій у промислових системах та необхідність обробки великих обсягів інформації стимулюють появу нових тенденцій у галузі розпізнавання образів. Розробка і впровадження різноманітних систем розпізнавання підтверджують популярність цього підходу в практичному застосуванні та наукових дослідженнях. Системи розпізнавання зразків можна класифікувати за сферами використання, наприклад, класифікація зображень документів у сфері оптичного розпізнавання символів, персональна ідентифікація за обличчям або відбитком пальця в біометрії, автоматична діагностика в медицині, розпізнавання цілей у військовій справі, послідовний аналіз у біоінформації, сортування об'єктів у промисловій автоматизації, захист енергетичних систем у електротехніці та багато інших.

Проблеми розпізнавання образів можна вирішувати за допомогою чотирьох основних підходів, які поділяються на чотири класи: статистичні, структурні, гібридні методи, що поєднують статистичний та структурний підходи, а також методи штучного інтелекту. Останній підхід є найбільш доцільним для складних задач, де між вхідними образами та цільовим результатом немає прямого або лінійного зв’язку.

Згорткові мережі здобули популярність завдяки своїй здатності автоматично витягувати характеристики зображень, що робить їх надзвичайно

потужними для задач, пов'язаних з візуальним сприйняттям. Вони дозволяють не лише значно знизити потребу в ручному витяганні ознак, але й забезпечують високу точність і ефективність при обробці великих обсягів даних. Архітектури CNN, такі як LeNet, AlexNet, VGG, ResNet, Inception та інші, продемонстрували чудові результати на завданнях класифікації зображень, детекції об'єктів та семантичному сегментуванні.

Актуальність даної теми полягає в тому, що розпізнавання образів залишається важливою частиною розвитку штучного інтелекту, і удосконалення алгоритмів для ефективнішої роботи з зображеннями є важливим кроком у розвитку сучасних технологій. Також, із розвитком апаратних засобів, таких як графічні процесори (GPU) та спеціалізовані чіпи для глибокого навчання, стало можливим застосовувати складніші та глибші моделі, що дозволяє отримувати ще кращі результати.

На сьогодні найактуальнішим методом для вирішення задачі розпізнавання образів і їх класифікації є використання згорткових нейронних мереж, що застосовують методи глибокого навчання (Deep Learning). Такий підхід дозволяє ефективно враховувати просторове розташування ознак під час їх видобутку з вхідних даних. Прикладами таких мереж є:

- автономні транспортні засоби компанії Tesla;
- системи розпізнавання мови, такі як Google Assistant;
- системи розпізнавання обличчя та відбитків пальців;
- додаток для перекладу Google Translate.

Метою роботи є дослідження застосування згорткових нейронних мереж для розв'язання задач розпізнавання образів, зокрема класифікації зображень, аналізу ефективності різних архітектур та порівняння результатів навчання моделей на різних наборах даних.

Для досягнення поставленої мети необхідно виконати наступні завдання:

– Ознайомитись з основами згорткових нейронних мереж і принципами їх роботи.

– Дослідити популярні архітектури CNN, такі як ResNet і Inception, та їх застосування для задач розпізнавання образів.

– Провести експерименти на стандартних наборах даних, таких як CIFAR-10 або MNIST, та порівняти ефективність різних моделей.

– Виконати оцінку результатів та розробити рекомендації щодо покращення точності моделей.

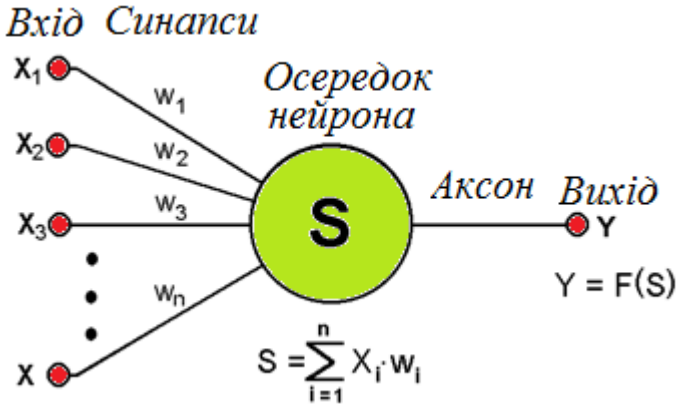
# РОЗДІЛ 1
# ОПИС ОСНОВНИХ ПОНЯТЬ ТА АНАЛІЗ НЕЙРОНИХ МЕРЕЖ

## 1.1 Основні складові нейронної мережі

У сучасному світі комп'ютерного зору розпізнавання образів є однією з найважливіших та найактуальніших задач, яка має широке застосування в багатьох галузях науки та техніки. Від медицини та безпеки до автомобільної промисловості та розваг, технології, що дозволяють комп'ютерам розпізнавати зображення та відео, знаходять все більше застосувань. Одним з найбільш ефективних методів розв'язання задачі розпізнавання образів є використання згорткових нейронних мереж (CNN - Convolutional Neural Networks), які стали основою для багатьох сучасних систем обробки зображень.

### 1.1.1 Штучний нейрон

Штучний нейрон (рис.1.1) є основним елементом нейронних мереж, що імітує функціонування біологічного нейрона, здатного обробляти та передавати інформацію. Кожен нейрон приймає вхідні сигнали, обробляє їх за допомогою активаційної функції та генерує вихідний сигнал, що передається наступним нейронам або вихідному шару




.

Рисунок 1.1 – Структура штучного нейрону

Нейрони, взаємопов’язані певним чином та об'єднані в мережу, утворюють нейронну мережу. Штучні нейрони мають правило для комбінування вхідних сигналів та активаційну функцію, яка дозволяє обчислити вихідний сигнал.

Інформація, що надходить на вхід нейрона, підсумовується з урахуванням вагових коефіцієнтів сигналів (1.1).

$$S = b + \sum_{i=1}^{n} x_i w_i \qquad (1.1)$$

де $x_i$ – вхідні данні у нейрон; $b$ – коефіцієнт зміщення; $w_i$ – ваги зв’язків.

### 1.1.2 Активаційна функція

Активаційна функція відіграє ключову роль у процесі навчання нейронних мереж, оскільки вона визначає, які саме вхідні сигнали будуть активувати нейрон і сприятимуть формуванню моделі. Вибір активаційної функції має великий вплив на ефективність та здатність моделі вирішувати складні завдання.

Функція активації нейрона (рис. 1.2) визначає правило, за яким зважена сума вхідних сигналів $S$ перетворюється у вихідний сигнал нейрона $Y$, який передається наступним нейронам мережі, тобто $Y=F(S)$. На рис. 1.2 наведено найбільш поширені графіки активаційних функцій нейронів.

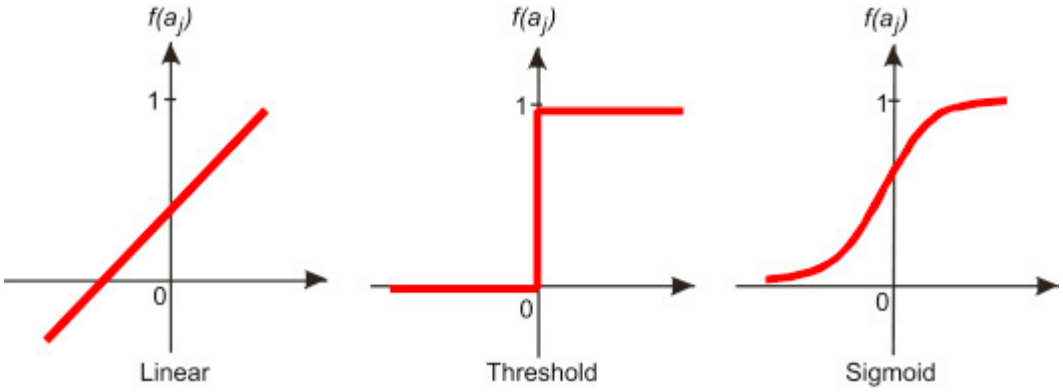



Рисунок 1.2 – Графіки активаційних функцій

Розглянемо математичний опис кожної з них.

– *Лінійна функція активації:* Лінійна функція активації визначає вихідний сигнал як безпосередню пропорцію від вхідного сигналу, тобто:

$$Y = k \cdot S \tag{1.2}$$

Ця функція не додає нелінійності до моделі, що обмежує її здатність до навчання складних патернів.

– *Ступінчаста функція активації:* Ступінчаста функція активації надає вихід значення 0 або 1, залежно від того, чи перевищує вхідний сигнал певний поріг.

$$Y = f(x) = \begin{cases} 1, S > 0 \\ -1, S \leq 0 \end{cases} \tag{1.3}$$

Вона часто використовується в простих моделях для класифікації, але має обмеження, зокрема, через свою дискретність, що ускладнює навчання через відсутність градієнта.

– *Сигмоідна функція активації:* Сигмоідна функція має вигляд

$$Y = \frac{1}{1 + e^{-CS}}, \text{де } C > 0 \tag{1.4}$$

де вихідне значення знаходиться в межах від 0 до 1. Вона часто використовується в задачах класифікації й дає можливість моделі м'якше переходити між класами, але може страждати від проблеми "градієнтного спаду" при великих значеннях вхідного сигналу.

– *Логарифмічна функція активації***:** Логарифмічна функція активації використовує логарифм від вхідного сигналу:

$$Y = \ln (S + \sqrt{S^2 + 1}, \text{де } C > 0 \tag{1.5}$$

де $S$ - це вхід нейрона. Ця функція має властивість згладжувати великі значення вхідного сигналу та може бути корисною у задачах, де потрібно зменшити вплив великих значень. Однак, вона може бути чутливою до

значень близьких до нуля і може потребувати додаткових перетворень для уникнення математичних помилок, таких як логарифм від нуля.

## 1.2 Структура згорткової нейронної мережі

Згорткові нейронні мережі (CNN) є потужним інструментом для обробки та аналізу зображень, відео та інших даних, що мають просторову структуру. Їхня архітектура складається з кількох ключових компонентів, які працюють разом для ефективної обробки вхідних даних. Основними етапами обробки є вхідні дані, де мережа отримує зображення чи інші типи інформації; згортковий шар, який здійснює виявлення особливостей; підвибірковий шар, що зменшує розмірність даних та покращує обчислювальну ефективність; повнозв'язний перцептрон, який виконує класифікацію або регресію на основі виявлених ознак; та процес навчання мережі, що включає оптимізацію ваг через зворотне поширення помилки. Розглянемо більш детально структуру згорткової нейронної мережі та принципи роботи кожного її компоненту.

### 1.2.1 Вхідні дані

*Вхідні дані* в згорткових нейронних мережах подаються у вигляді тривимірної матриці розміром M×N×K, де M×N - це роздільна здатність зображення у пікселях, а K - кількість кольорових каналів. Зображення може бути представлено як у форматі RGB з трьома каналами, так і у відтінках сірого - з одним каналом. Перед подачею до мережі дані підлягають нормалізації, оскільки необроблені значення пікселів можуть мати широкий діапазон, що ускладнює навчання моделі.

Нормалізація - це процес приведення значень ознак до визначеного діапазону, зазвичай від 0 до 1. Для цього використовується формула:

$$f(p, min, max) = \frac{p - min}{max - min} \quad (1.6)$$

де $f$ - нормалізоване значення, $p$ - поточне значення пікселя, $min$ та $max$ – мінімальне та максимальне значення серед усіх пікселів.

Після нормалізації дані подаються на вхід згорткової нейронної мережі для подальшої обробки.

### 1.2.2 Згортковий шар

*Convolution layer (Згортковий шар)* є одним з ключових елементів згорткової нейронної мережі (CNN). Його основне завдання — виконання операції згортки над вхідними даними з використанням набору фільтрів, результатом чого є формування карти активації (рис. 1.3). Кожен фільтр має розміри N×N×K, де K - кількість каналів вхідного зображення. По суті, фільтр - це матриця ваг, значення якої ініціалізуються випадковим чином у межах від -0.5 до 0.5.

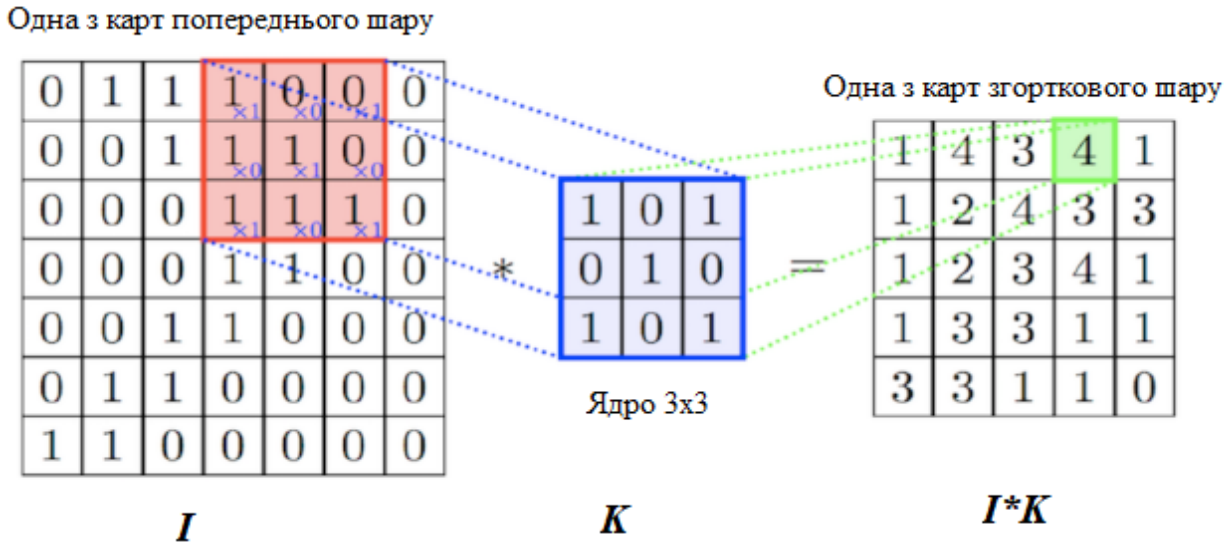



Рисунок 1.3 – Ілюстрація операція згортки

Розмір фільтрів зазвичай знаходиться в межах від 3×3×K до 7×7×K. Чим більший фільтр, тим більшу область вхідного зображення він охоплює, що дозволяє виявляти більш узагальнені ознаки. У процесі обчислень кожен

фільтр переміщується (зі зсувом - stride) по всій вхідній матриці. На кожному кроці здійснюється поелементне множення значень пікселів у вікні з відповідними вагами фільтра, після чого результати підсумовуються та записуються до відповідної позиції карти активації. Отримана карта активації має розмір M×M, який визначається за формулою (1.7).

$$M = \frac{N - K}{S} + 1 \tag{1.7}$$

де N – ширина (висота) вхідних даних; S – крок (stride); K – ширина фільтру.

Під час виконання операції згортки розміри вихідної матриці зазвичай зменшуються. Щоб уникнути цього ефекту, передбачено попередню обробку даних за допомогою методу *ZeroPadding* - доповнення вхідної матриці нульовими значеннями по її краях. Розмір такого доповнення (позначається як $p$) розраховується за формулою (1.8):

$$p = \frac{K - 1}{2} \tag{1.8}$$

де $K$ - розмір фільтра.

Завдяки цій техніці розмір карти активації після згортки залишається незмінним відносно вхідної матриці.

На кінцеві розміри вихідного зображення також впливає параметр кроку (*stride*): чим він більший, тим менше елементів буде у результаті. Розрахунок розміру вихідної матриці виконується за відповідною формулою (1.7).

Завершальним етапом роботи згорткового шару є застосування функції активації до карти активації. Найчастіше для цього використовують функцію *ReLU,* яка застосовується до кожного елемента карти після згортки, формуючи фінальну карту ознак.

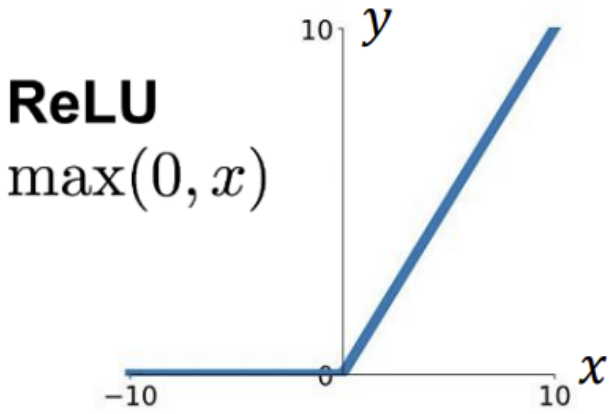



Рисунок 1.4 – Функція активації *ReLU*

1.2.3 Підвибірковий шар

*Pooling layer (Підвибірковий шар)* використовується для зменшення обсягу вхідних даних, зберігаючи при цьому ключову інформацію. Завдяки цьому шар допомагає зменшити обчислювальні витрати та забезпечує стійкість моделі до змін масштабу зображення, одночасно сприяючи виявленню більш загальних ознак.

Існують різні типи операцій *pooling'y*:

- *minimum-pooling* - вибір мінімального значення у вікні;
- *average-pooling* - обчислення середнього значення;
- *max-pooling* - вибір максимального значення (найпоширеніший метод, див. рис. 1.5).

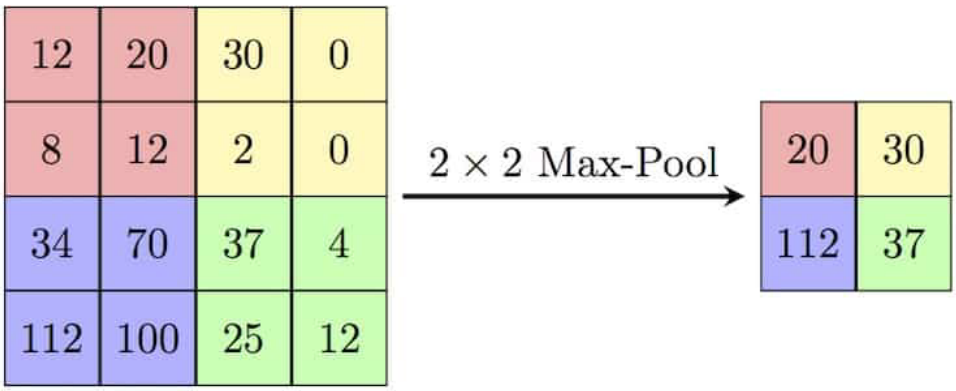



Рисунок 1.5 – Ілюстрація операції *max-pooling* (вікно 2×2)

У випадку *max-pooling* визначається розмір вікна (наприклад, $N \times N$), яке послідовно переміщується по вхідній матриці без перекриттів. Для кожної області вікна обирається максимальне значення, яке записується у відповідну позицію вихідної матриці. Це дозволяє зменшити розмір зображення, зберігаючи при цьому найінформативніші пікселі. Новий розмір результату можна обчислити за формулою (1.7), в якій крок переміщення дорівнює ширині вікна *pooling'y.*

### 1.2.4 Повнозв'язний перцептрон

*Повнозв'язний шар*, або багатошаровий перцептрон (див. рис. 1.6), є завершальним етапом згорткової нейронної мережі. Його основне призначення - виконання класифікації. Цей шар здатен моделювати складні нелінійні залежності, що дозволяє підвищити точність розпізнавання об'єктів.

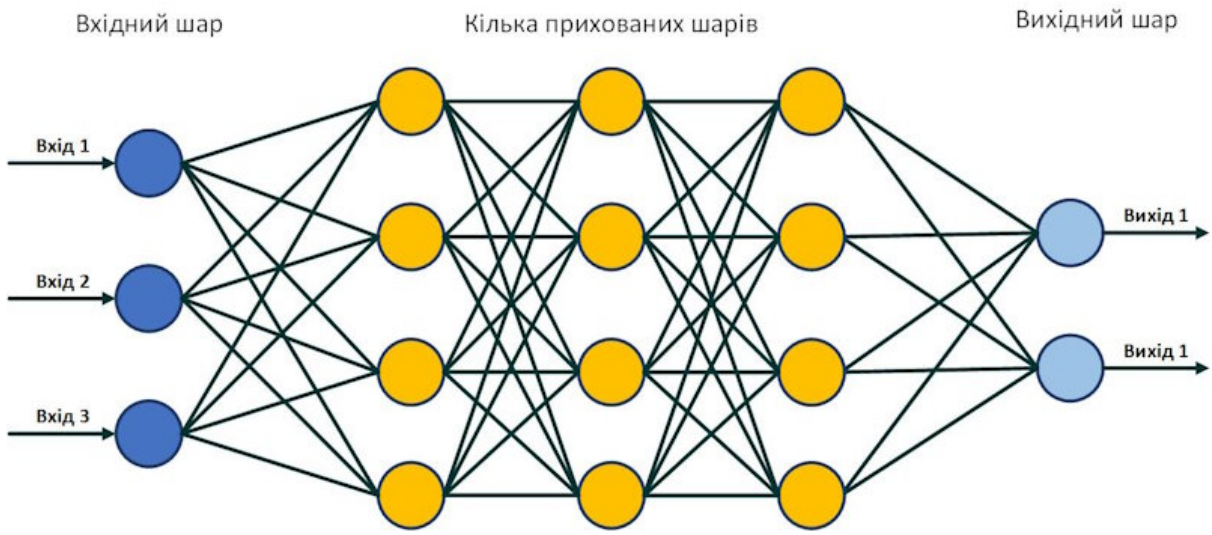



Рисунок 1.6 – Ілюстрація багатошарового перцептрона

На вхід першого шару перцептрона подаються всі значення з $R$ карт ознак розміром $N \times N$, отриманих після обробки в згорткових і підвибіркових шарах. У результаті формується вхідний шар з $N \times N \times R$ нейронів. Після нього мережа може містити довільну кількість прихованих шарів, а завершальним є вихідний шар, у якому кількість нейронів відповідає числу класів, що розпізнаються.

### 1.2.5 Навчання мережі

Навчання нейронної мережі полягає у поступовому поданні даних із навчальної вибірки на її вхід. Мережа обробляє ці дані і генерує відповідь. На кожному етапі навчання обчислюється функція похибки, на основі якої відбувається коригування ваг мережі. Функція похибки визначається як різниця між фактичною відповіддю мережі та правильною (цільовою) відповіддю.

Для нейронів, що знаходяться в останньому шарі, вихідна відповідь та цільова величина відомі. Величина похибки розраховується за допомогою середньоквадратичної помилки:

$$E_t = \frac{1}{2}\sum_i (z_{ti} - y_{ti})^2 \qquad (1.9)$$

де: $E_t$- величина функції похибки для даних $t$; $z_{tj}$ та $y_{tj}$ - вірний та фактичний вихід нейрону для даних $t$ відповідно.

Існує кілька варіантів розташування шарів відносно один одного:

– Якщо підвибірковий шар розташований перед повнозв'язним, тоді похибка передається до шару перед підвибірковим.

– Якщо згортковий шар знаходиться перед повнозв'язним, то обчислення похибки здійснюється через операцію зворотної згортки.

– Якщо підвибірковий шар розміщений перед згортковим, похибка передається підвибірковому шару, який потім передає її наступному шару.

Ковзне вікно (фільтр) можна трактувати (рис. 1.7) як звичайний повнозв'язний шар, але з тією різницею, що в згортковому шарі зв'язки є роздільними. Це означає, що один зв'язок з певними вагами може бути застосований до кількох нейронів.

Після того, як згортка представлена у вигляді шарів нейронів, обчислення похибки здійснюється так само, як і в повнозв'язному шарі. Таким чином, обчисливши похибки для одного шару, можна розрахувати похибки для кожного попереднього шару.

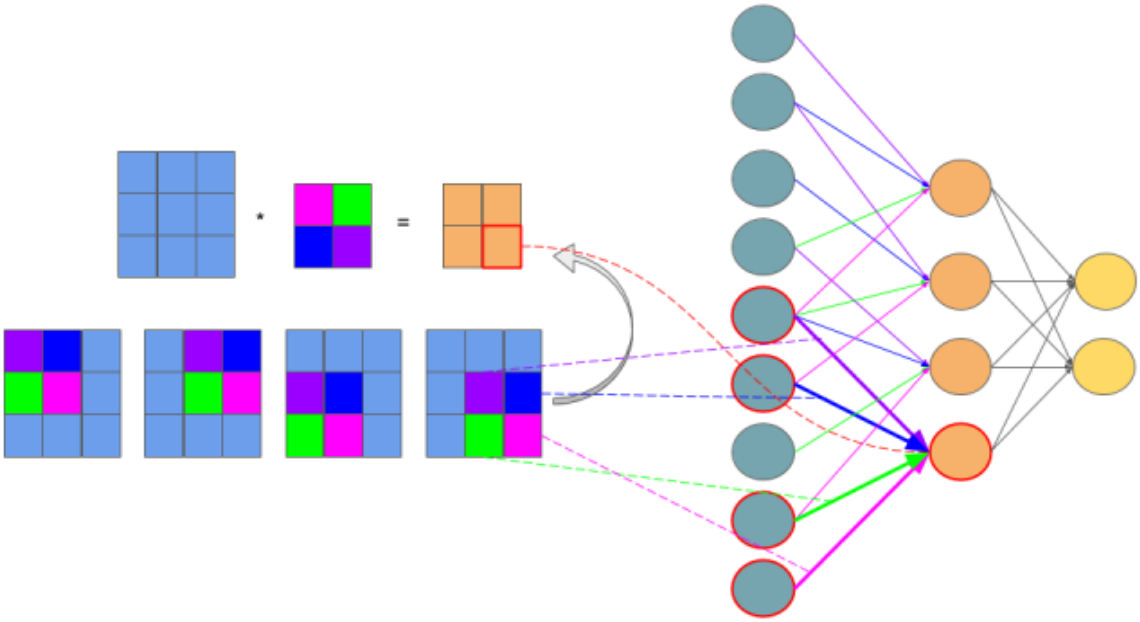


Рисунок 1.7 – Ілюстрація інтерпретації операції багатошарової згортки

Операція зворотної згортки (рис. 1.8) є методом обчислення похибок, який полягає в повороті фільтру на 180 градусів і зміненому процесі ковзання фільтром по карті активації. Це означає, що для обчислення похибок необхідно повернути фільтр на 180 градусів і здійснити звичайну згортку на основі раніше обчислених похибок. Важливою умовою для коректного розміру результату є застосування ZeroPadding.

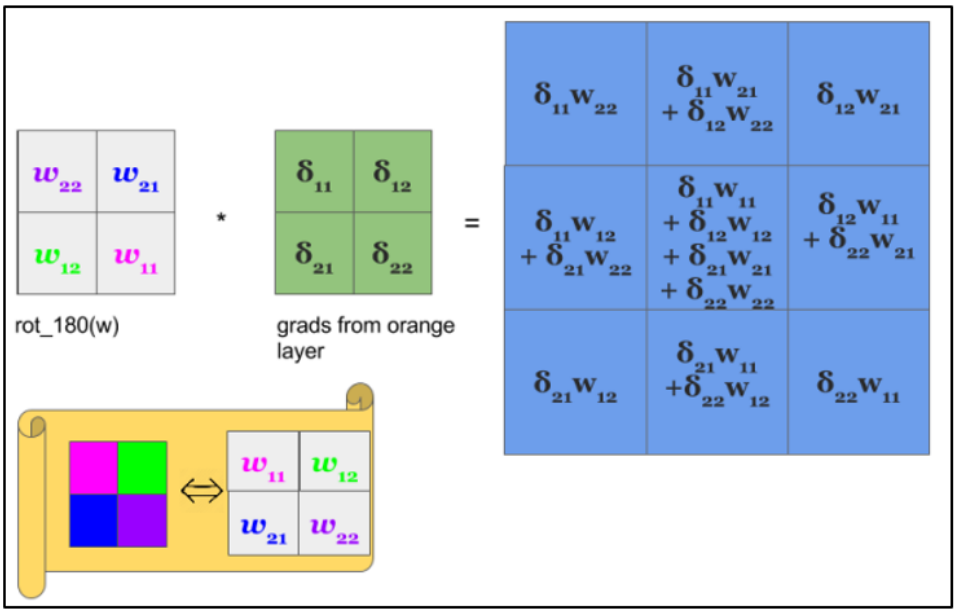



Рисунок 1.8 – Ілюстрація операції результату зворотної згортки

## 1.3 Огляд та аналіз відомих архітектур CNN

Згорткові нейронні мережі (CNN) стали основним інструментом у сфері комп’ютерного зору. Вони дозволяють автоматично виділяти та навчатися важливим ознакам зображень. Наведемо короткий огляд чотирьох ключових архітектур: *AlexNet, VGG, ResNet* та *Inception* і оберемо ті, що будуть використані для подальшого аналізу.

### 1.3.1 Архітектура AlexNet

*AlexNet* є однією з перших глибоких згорткових нейронних мереж, яка принесла прорив у галузі комп’ютерного зору. Вона була розроблена у 2012 році, стала революційною архітектурою, яка перемогла на змаганні *ImageNet*.

Основні характеристики *AlexNet* (рис.1.9):

– Складається з 8 шарів (5 згорткових і 3 повнозв'язні).

– Використовує *ReLU* як функцію активації, що дозволяє мережі навчатися значно швидше.

– Застосовує *dropout* для зменшення переобучення.

– Першою ввела використання *GPU* для прискорення навчання.

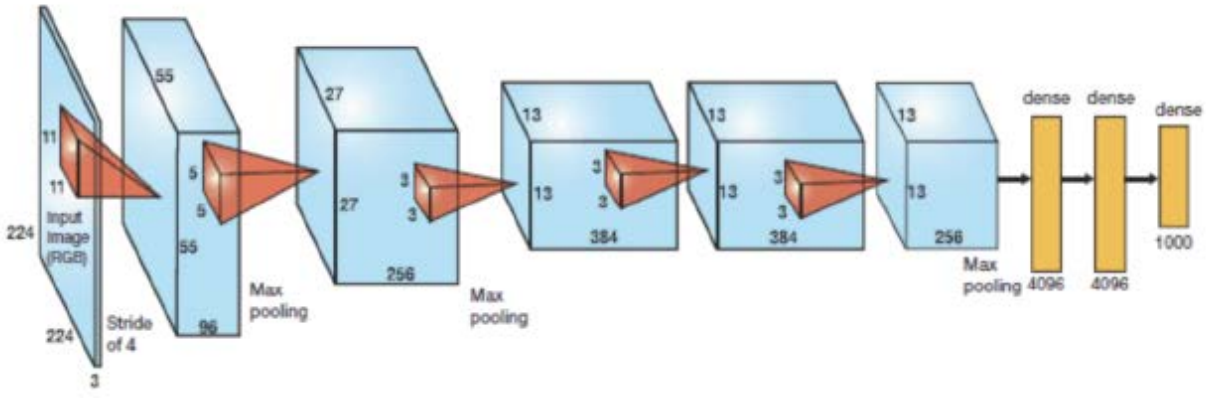



Рисунок 1.9 - Архітектура моделі AlexNet

*AlexNet* показала, що збільшення глибини мережі та правильне налаштування гіперпараметрів можуть значно покращити точність розпізнавання.

Таблиця 1.1 – Опис шарів архітектури *AlexNet*

| № шару | Тип шару | Параметри ядра / Налаштування | Кількість фільтрів | Розмір виходу | Активація |
|---|---|---|---|---|---|
| 1 | Згорточний (Conv) | 11×11, крок 4, заповнення 2 | 96 | 55×55×96 | ReLU |
| 2 | Max Pooling | 3×3, крок 2 | – | 27×27×96 | – |
| 3 | Згорточний (Conv) | 5×5, заповнення 2 | 256 | 27×27×256 | ReLU |
| 4 | Max Pooling | 3×3, крок 2 | – | 13×13×256 | – |
| 5 | Згорточний (Conv) | 3×3, заповнення 1 | 384 | 13×13×384 | ReLU |
| 6 | Згорточний (Conv) | 3×3, заповнення 1 | 384 | 13×13×384 | ReLU |
| 7 | Згорточний (Conv) | 3×3, заповнення 1 | 256 | 13×13×256 | ReLU |
| 8 | Max Pooling | 3×3, крок 2 | – | 6×6×256 | – |
| 9 | Повнозв’язний (FC) | – | 4096 | 4096 | ReLU |
| 10 | Повнозв’язний (FC) | – | 4096 | 4096 | ReLU |
| 11 | Повнозв’язний (FC) | – | 1000 | 1000 (класи ImageNet) | softmax |

Згорткові шари (Conv): Вони застосовують фільтри для виділення ознак (features) з вхідного зображення, таких як контури, текстури тощо.

Max Pooling: Це операція, що зменшує розмірність зображення, зберігаючи найбільш важливі характеристики.

Повнозв'язні шари (Fully Connected): Це шари, де кожен нейрон з'єднаний з усіма нейронами попереднього шару. Вони відповідають за класифікацію та прийняття фінальних рішень.

### 1.3.2 Архітектура *VGG*

Мережа *VGG* була запропонована у 2014 році і стала наступним кроком у розвитку *CNN* після *AlexNet*. Вона відома своєю простою, послідовною структурою - всі згортки мають розмір 3×3. Мережі *VGG* (наприклад, *VGG-16, VGG-19*) демонструють, як глибина мережі впливає на точність.

Основні характеристики *VGG:*

– Відзначається простою та уніфікованою структурою: всі згорткові шари мають фільтри 3×3 з кроком 1, а максимальне підвибіркування - 2×2.

– Існують різні варіації архітектури: *VGG-11, VGG-16, VGG-19*, де число вказує на кількість шарів.

– Мережа значно глибша за *AlexNet*, що дозволяє їй виявляти складніші ознаки.

– Підтримує велику кількість параметрів, що забезпечує високу точність, але потребує значних обчислювальних ресурсів.

| ConvNet Configuration | | | | | |
|---|---|---|---|---|---|
| A | A-LRN | B | C | D | E |
| 11 weight layers | 11 weight layers | 13 weight layers | 16 weight layers | 16 weight layers | 19 weight layers |
| input (224 × 224 RGB image) | | | | | |
| conv3-64 | conv3-64<br>**LRN** | conv3-64<br>**conv3-64** | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 |
| maxpool | | | | | |
| conv3-128 | conv3-128 | conv3-128<br>**conv3-128** | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 |
| maxpool | | | | | |
| conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256<br>**conv1-256** | conv3-256<br>conv3-256<br>**conv3-256** | conv3-256<br>conv3-256<br>conv3-256<br>**conv3-256** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| FC-4096 | | | | | |
| FC-4096 | | | | | |
| FC-1000 | | | | | |
| soft-max | | | | | |

Рисунок 1.10 – Приклад варіанту архітектури *VGG*

*VGG* стала еталонною архітектурою, яку часто використовують як базову модель у багатьох комп’ютерних задачах. Попри ефективність, архітектура має велику кількість параметрів і вимагає значних ресурсів.

### 1.3.3 Архітектура ResNet

*ResNet* (Residual Network), розроблена у 2015 році, ввела концепцію залишкових з'єднань (skip connections), які дозволяють передавати інформацію через кілька шарів. Це вирішує проблему затухання градієнтів і дозволяє створювати надзвичайно глибокі мережі (наприклад, *ResNet-50, ResNet-101*).

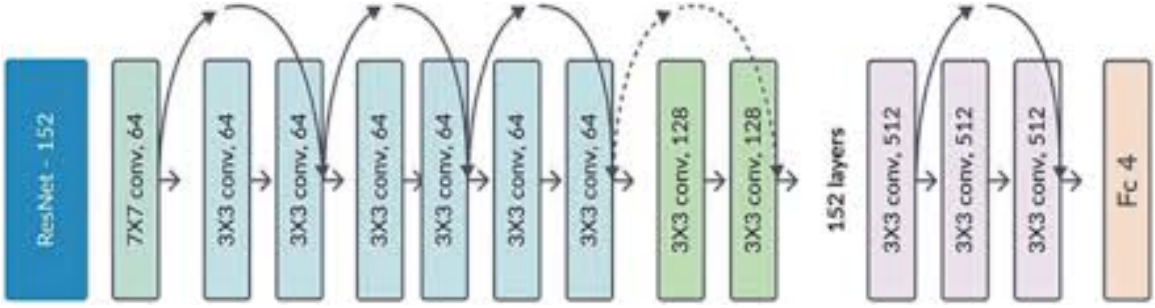



Рисунок 1.11 - Приклад варіанту типової архітектури ResNet

Модель поєднує глибину з ефективним навчанням і високу точність.

### 1.3.4 Архітектура Inception

*Inception* (або GoogleNet), запропонована компанією *Google* у 2014 році, використовує інцепшн-блоки - паралельні згортки з різними фільтрами (1×1, 3×3, 5×5), які дозволяють моделі вловлювати ознаки на різних масштабах. Завдяки згорткам 1×1 значно знижується кількість параметрів.

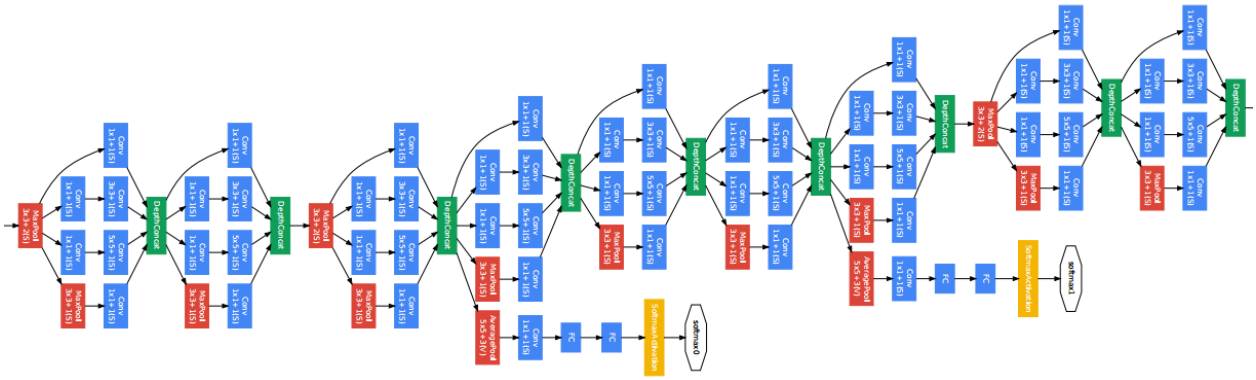


Рисунок 1.12 - Приклад варіанту типової архітектури GoogleNet

Подальші версії (*Inception v3, v4, Inception-ResNet*) забезпечили ще більшу точність та ефективність.

### 1.3.5 Обґрунтування вибору архітектур мереж для аналізу

Для подальшого дослідження обрано архітектури *ResNet* та *Inception*, оскільки вони є представниками сучасного покоління глибоких *CNN* і мають переваги над попередниками (*AlexNet, VGG*). *ResNet* забезпечує ефективне навчання дуже глибоких мереж за допомогою залишкових зв'язків, а *Inception* - оптимальний баланс між точністю та обчислювальною складністю завдяки багатошаровій структурі з різними фільтрами.

Обидві архітектури відзначаються високою продуктивністю, є широко дослідженими та підтримують подальше масштабування і вдосконалення. Їх аналіз дозволяє оцінити можливості глибоких мереж у реальних застосуваннях.

*Висновки до розділу:*

У першому розділі було розглянуто основні концепції, які складають основу для розуміння нейронних мереж, а також здійснено огляд ключових архітектур згорткових нейронних мереж (CNN).

Визначено, що штучний нейрон є основним елементом нейронної мережі, який відповідає за прийом і обробку інформації. Окремо проаналізовано роль активаційних функцій, що дають змогу нейрону генерувати вихід на основі вхідного сигналу, впливаючи на здатність мережі вивчати складні патерни.

Описано, як вхідні дані проходять через кілька шарів мережі, кожен з яких виконує певну задачу: згортка, підвибірка, і повнозв'язний перцептрон. Розкрито роль кожного з шарів у процесі обробки інформації: згорткові шари відповідають за виділення ознак, підвибіркові - за зменшення розмірності, а повнозв'язний перцептрон здійснює кінцеве прийняття рішень. Зазначено важливість процесу навчання, який передбачає оптимізацію параметрів

мережі для досягнення високої точності у завданнях класифікації та розпізнавання.

Детально описано чотири основні архітектури - AlexNet, VGG, ResNet та Inception - які зробили значний внесок у розвиток згорткових нейронних мереж і дослідження їхніх переваг і обмежень. Визначено, що кожна з архітектур має свої особливості, що дозволяють їй ефективно вирішувати різні завдання в галузі комп’ютерного зору. Розглянуто, чому для подальшого аналізу були обрані ResNet і Inception: ці архітектури дають змогу досягати високої продуктивності завдяки новим підходам, таким як залишкові з'єднання та інцепшн-блоки, і є актуальними для більш складних та масштабних завдань.

# РОЗДІЛ 2
# АНАЛІЗ МОЖЛИВОСТЕЙ ОБРАНИХ АРХІТЕКТУР ТА ПРОВЕДЕННЯ ДОСЛІДЖЕННЯ

## 2.1 Архітектура моделі Residual Network

Residual Network (ResNet) — це штучна нейронна мережа, концепція якої натхненна будовою пірамідальних нейронів кори головного мозку. Її основним елементом є залишкові (residual) блоки, які реалізують механізм пропуску сигналу (shortcut connections), дозволяючи передавати інформацію в обхід певних шарів мережі.

Глибокі нейронні мережі відкрили нові можливості у сфері класифікації зображень завдяки здатності поєднувати характеристики різних рівнів - від базових до абстрактних - у межах наскрізної багатошарової архітектури. При цьому розширення глибини мережі, тобто збільшення кількості послідовних шарів, дозволяє покращити якість виділення ознак. Сучасні дослідження підтверджують, що саме глибина архітектури є критичним фактором, що впливає на ефективність моделі.

Хоча збільшення глибини нейронної мережі зазвичай сприяє підвищенню точності, виникає проблема деградації: після досягнення певного рівня глибини подальше збільшення кількості шарів не лише не покращує результат, а й призводить до його погіршення. Такий ефект не пов’язаний із перенавчанням, а з труднощами оптимізації дуже глибоких архітектур (рис.2.1). Згідно з результатами досліджень [8], додавання шарів до вже глибокої мережі може спричинити зростання похибки на тренувальній вибірці, що вказує на обмеження в ефективності навчання складних моделей. Цей феномен підтверджено серією експериментів.

Розглянемо дві нейронні мережі: одну з меншою глибиною та іншу, що є її розширенням за рахунок додавання додаткових шарів. В одному з

архітектурних підходів [8] до глибшої моделі додаються шари, які передають інформацію без змін - ті самі дані, що надходять на вхід, проходять далі, тоді як решта структури копіює архітектуру базової неглибокої мережі. Такий підхід теоретично гарантує, що більш глибока модель не повинна демонструвати гірші результати навчання, ніж її спрощений варіант.

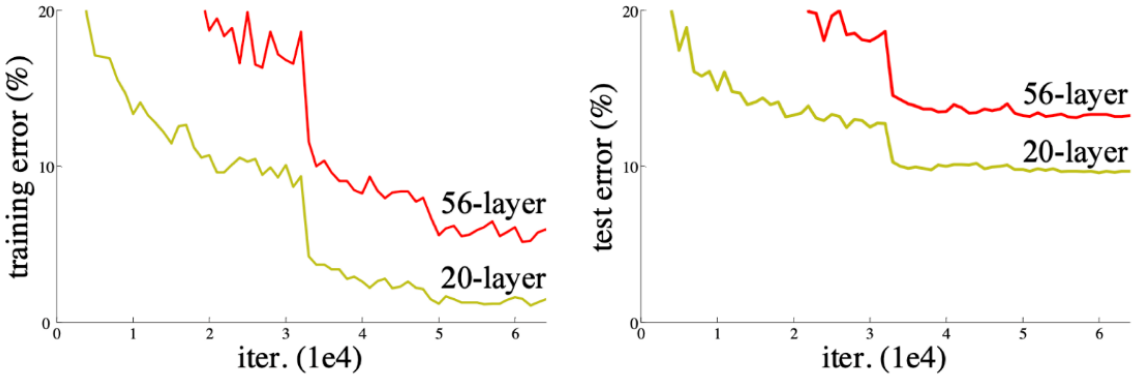



Рисунок 2.1 – Ілюстрація процесу накопичення похибки при збільшенні кількості шарів

Однак практичні дослідження [8] показують інше: сучасні оптимізаційні алгоритми часто не здатні знайти розв’язки, які б хоча б дорівнювали за якістю раніше сконструйованим моделям, або ж обчислення таких розв’язків вимагає занадто багато ресурсів і часу.

Збільшення кількості шарів у нейронній мережі зазвичай сприяє кращій здатності до навчання, однак не завжди це призводить до зростання точності. З певним рівнем глибини, подальше збільшення кількості шарів може почати негативно впливати на ефективність моделі, знижуючи її здатність до правильної генералізації. Причиною цього є складність оптимізації глибших мереж, що виникає через збільшення кількості параметрів і взаємодій між ними. У таких випадках моделі можуть не тільки перестати покращуватися, але й демонструвати погіршення результатів через виникнення проблеми деградації. Такий ефект спостерігається навіть при використанні

найсучасніших підходів і технік навчання, що робить вибір оптимальної архітектури складним завданням.

У дослідженні [5] запропоновано спосіб подолання проблеми деградації шляхом інтеграції в архітектуру нейронної мережі спеціальних блоків залишкового навчання (Residual learning), як показано на рисунку 2.2.

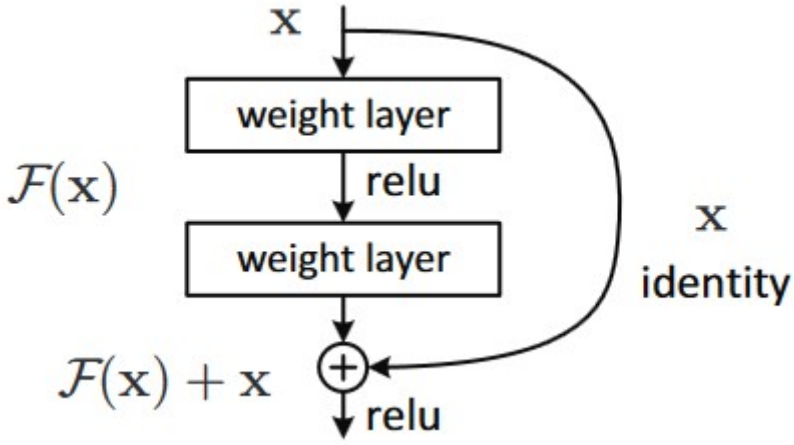



Рисунок 2.2 – Структура блоку Residual learning

Замість того щоб очікувати, що кожен набір послідовних шарів безпосередньо реалізує бажане функціональне відображення, запропоновано дозволити їм навчатися залишковому перетворенню. Якщо позначити цільову функцію як H(x), то відповідні нелінійні шари мають апроксимувати залишкову функцію F(x) := H(x) - x. Тоді вихідна функція записується як F(x) + x. Передбачається, що навчання залишкової функції є простішим, ніж пряме навчання H(x). У разі, якщо оптимальним варіантом є Identity Mapping [10], тоді зведення залишку до нуля є менш складним завданням, ніж побудова ідентичного перетворення через стек нелінійних шарів. Така структура F(x) + x може бути реалізована шляхом використання прямого з'єднання (shortcut connection), яке ілюструється на рисунку 2.2.

Shortcut-з'єднання - це спеціальні зв'язки, що дозволяють пропускати один або кілька шарів у мережі. У цьому випадку вони реалізують просте ідентичне відображення, результат якого додається до виходу складених шарів.

Незважаючи на це, повна модель все ще навчається за допомогою наскрізного стохастичного градієнтного спуску (SGD) [11].

У нашому підході залишкове навчання використовується для кожного блоку, що включає кілька послідовних шарів. Формально, базовий елемент архітектури описується наступним чином:

$$y = F(x, \{W_i\}) + x \tag{2.1}$$

де: $F$ - функція, що представляє залишкове відображення; $y$ - вихідний вектор після обробки; $x$ - вхідний вектор до шару.

Операція $F + x$ реалізується через shortcut-з'єднання, яке виконує просте елементне додавання. Після цієї операції застосовується друга нелінійна активаційна функція. Важливо зазначити, що такі з'єднання не збільшують кількість параметрів і не ускладнюють обчислення, що робить їх ефективними як з теоретичної, так і з практичної точки зору. Це дозволяє об'єктивно порівнювати звичайні та залишкові мережі за такими критеріями, як глибина, кількість параметрів, ширина та обчислювальні витрати - за винятком незначного впливу операції додавання.

Для коректної роботи shortcut-з'єднань необхідно, щоб розміри вхідного вектора $x$ та функції $F(x)$ збігалися. У випадках, коли ці розміри відрізняються, можна застосувати лінійну проекцію через матрицю $W_s$, що дозволяє привести $x$ до необхідного розміру:

$$y = F(x, \{W_i\}) + W_s x \tag{2.2}$$

У рівнянні (2.2) може також використовуватись квадратна матриця $W_s$ для здійснення лінійної проекції. Структура залишкової функції $F$ залишається досить гнучкою: вона може складатися з кількох згорткових шарів. Операція поелементного додавання виконується між відповідними каналами двох згорткових карт, канал за каналом.

У типових архітектурах *ResNet* (див. рис. 1.11) залишкові блоки реалізуються із використанням двох або трьох послідовних шарів, що включають активаційні функції *ReLU* та операції нормалізації партії (*batch normalization*) між ними.

Різновиди мереж із залишковою (Residual) архітектурою представлені на рисунку 2.3.

| layer name | output size | 18-layer | 34-layer | 50-layer | 101-layer | 152-layer |
|---|---|---|---|---|---|---|
| conv1 | 112×112 | 7×7, 64, stride 2 | | | | |
| conv2_x | 56×56 | 3×3 max pool, stride 2 | | | | |
| | | $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$ |
| conv3_x | 28×28 | $\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$ |
| conv4_x | 14×14 | $\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$ |
| conv5_x | 7×7 | $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$ | $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$ |
| | 1×1 | average pool, 1000-d fc, softmax | | | | |
| FLOPs | | $1.8\times10^9$ | $3.6\times10^9$ | $3.8\times10^9$ | $7.6\times10^9$ | $11.3\times10^9$ |

Рисунок 2.3 – Різні варіанти мереж з архітектурою *Residual*

## 2.2 Архітектура моделі Inception

Найпростішим підходом до підвищення ефективності глибоких нейронних мереж є масштабування їх розмірів. Це передбачає як збільшення глибини - тобто кількості шарів у мережі, так і розширення її ширини - кількості нейронів у кожному шарі. Такий підхід часто вважається безпечним і ефективним способом підвищити якість моделей, особливо у випадках, коли доступна велика кількість анотованих навчальних даних. Водночас цей метод має два суттєві недоліки. По-перше, збільшення розмірів моделі зазвичай призводить до зростання кількості параметрів, що підвищує ризик перенавчання, особливо за обмеженого обсягу навчальних прикладів. По-друге, рівномірне масштабування архітектури значно збільшує потребу в обчислювальних ресурсах.

Одним з ключових підходів до вирішення зазначених проблем є перехід від повнозв’язаних архітектур до архітектур із обмеженими зв’язками. Це питання було досліджене компанією Google [7], яка запропонувала використання модулів Inception (див. рис. 2.4).

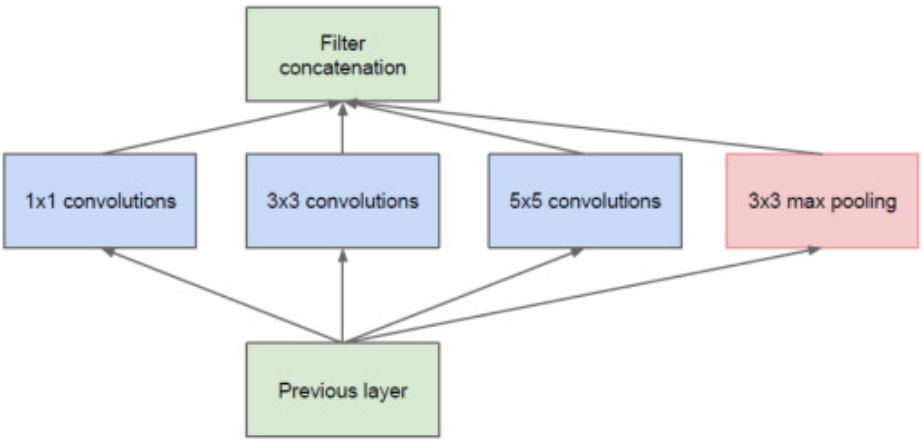



Рисунок 2.4 – Структура початкової версії модулю Inception

Ключова ідея архітектури Inception полягає в наближенні оптимальної локальної розрідженої структури зорової згорткової мережі за допомогою доступних щільних компонентів. Побудова мережі здійснюється шляхом формування її з окремих згорткових модулів. Основне завдання - знайти ефективну локальну структуру та відтворити її просторово по всій мережі. Такі модулі формують одиниці наступного шару, які пов’язані з відповідними одиницями попереднього шару.

Передбачається, що кожна одиниця у шарі відповідає певній ділянці вхідного зображення, і ці одиниці згруповані у фільтри. У нижніх шарах мережі, ближчих до вхідного рівня, корельовані одиниці зосереджуються в локальних регіонах. Це дозволяє покривати подібні скупчення за допомогою згорток із фільтрами розміру 1x1. У той же час очікується наявність меншої кількості розсіяних скупчень, які потребують використання фільтрів більшого розміру, що охоплюють ширші області зображення.

Щоб уникнути складнощів, пов’язаних з узгодженням виходів таких згорток, реалізація Inception-архітектури обмежується використанням фільтрів розмірів 1x1, 3x3 та 5x5 - це рішення було зумовлене радше практичністю, ніж технічною необхідністю. У результаті сформована архітектура (див. рис. 2.5) поєднує згадані шари в єдиний вихідний вектор, що слугує вхідними даними для наступного шару.

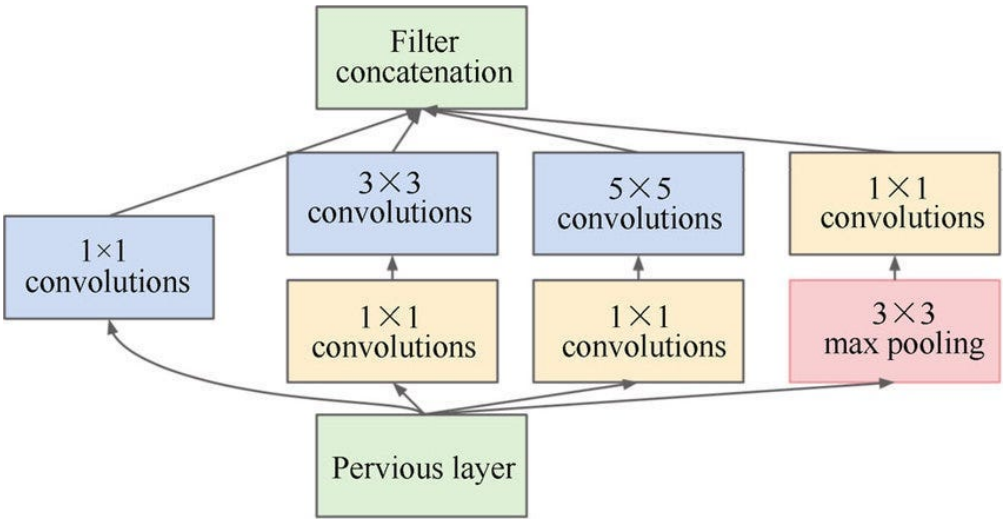



Рисунок 2.5 – Структура початкового модулю зі зменшенням розмірів

Оскільки операції об’єднання є ключовими для ефективності сучасних згорткових нейронних мереж (CNN), введення паралельних шляхів об’єднання на кожному етапі має додаткову корисну цінність.

Архітектура Inception являє собою мережу, що складається з модулів зазначеного типу, які послідовно накладаються один на один, з періодичними шарами підвибірки maxpooling з кроком 2, що дозволяють знижувати роздільну здатність мережі.

Однією з основних переваг цієї архітектури є її здатність значно збільшувати кількість одиниць на кожному шарі без значного зростання обчислювальної складності. Використання зменшення розмірів на кожному етапі дозволяє ефективно передавати велику кількість вхідних фільтрів до наступного шару, зменшуючи їх розмір. Іншим важливим аспектом цієї конструкції є її узгодженість з інтуїтивним розумінням того, що візуальна інформація повинна оброблятися на різних масштабах і потім агрегуватися, щоб наступний етап міг абстрагувати елементи різних масштабів. Покращене використання обчислювальних ресурсів дозволяє збільшити як ширину кожного блоку, так і кількість блоків, не збільшуючи обчислювальних витрат.

Іншим підходом до застосування початкової архітектури є створення її спрощених, але більш обчислювально ефективних варіантів.

На рис. 2.6 наведеґо структуру модуля Inception для нейронної мережі GoogLeNet (див. рис. 1.12), а на рис. 2.7 наведено її різні різновиди.

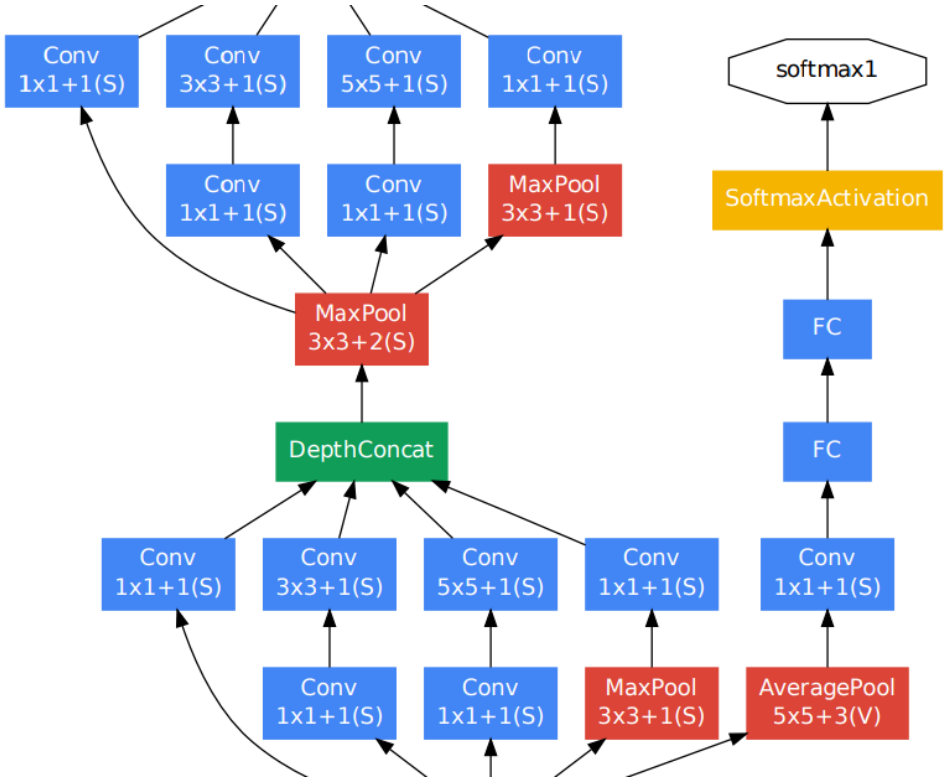



Рисунок 2.6 – Структура модулю Inception нейронної мережі GoogLeNet

| type | patch size/ stride | output size | depth | #1×1 | #3×3 reduce | #3×3 | #5×5 reduce | #5×5 | pool proj | params | ops |
|---|---|---|---|---|---|---|---|---|---|---|---|
| convolution | 7×7/2 | 112×112×64 | 1 | | | | | | | 2.7K | 34M |
| max pool | 3×3/2 | 56×56×64 | 0 | | | | | | | | |
| convolution | 3×3/1 | 56×56×192 | 2 | | 64 | 192 | | | | 112K | 360M |
| max pool | 3×3/2 | 28×28×192 | 0 | | | | | | | | |
| inception (3a) | | 28×28×256 | 2 | 64 | 96 | 128 | 16 | 32 | 32 | 159K | 128M |
| inception (3b) | | 28×28×480 | 2 | 128 | 128 | 192 | 32 | 96 | 64 | 380K | 304M |
| max pool | 3×3/2 | 14×14×480 | 0 | | | | | | | | |
| inception (4a) | | 14×14×512 | 2 | 192 | 96 | 208 | 16 | 48 | 64 | 364K | 73M |
| inception (4b) | | 14×14×512 | 2 | 160 | 112 | 224 | 24 | 64 | 64 | 437K | 88M |
| inception (4c) | | 14×14×512 | 2 | 128 | 128 | 256 | 24 | 64 | 64 | 463K | 100M |
| inception (4d) | | 14×14×528 | 2 | 112 | 144 | 288 | 32 | 64 | 64 | 580K | 119M |
| inception (4e) | | 14×14×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 840K | 170M |
| max pool | 3×3/2 | 7×7×832 | 0 | | | | | | | | |
| inception (5a) | | 7×7×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 1072K | 54M |
| inception (5b) | | 7×7×1024 | 2 | 384 | 192 | 384 | 48 | 128 | 128 | 1388K | 71M |
| avg pool | 7×7/1 | 1×1×1024 | 0 | | | | | | | | |
| dropout (40%) | | 1×1×1024 | 0 | | | | | | | | |
| linear | | 1×1×1000 | 1 | | | | | | | 1000K | 1M |
| softmax | | 1×1×1000 | 0 | | | | | | | | |

Рисунок 2.7 – Різновиди мереж з архітектурою Inception

## 2.3 Дослідження обраних архітектур

Для проведення дослідження було обрано дві архітектури згорткових нейронних мереж - ResNet та Inception, які реалізують різні підходи до оптимізації продуктивності мережі та підвищення точності результатів. У якості базового набору даних для навчання та тестування було використано Cifar10 [16], що включає 50 000 кольорових зображень для тренування та 10 000 - для тестування. Обидві вибірки охоплюють 10 однакових категорій (див. рис. 2.8). Для валідації використовувалося 10% тренувального набору. На вхід нейромереж подавалися зображення з роздільною здатністю 224x224 пікселі.

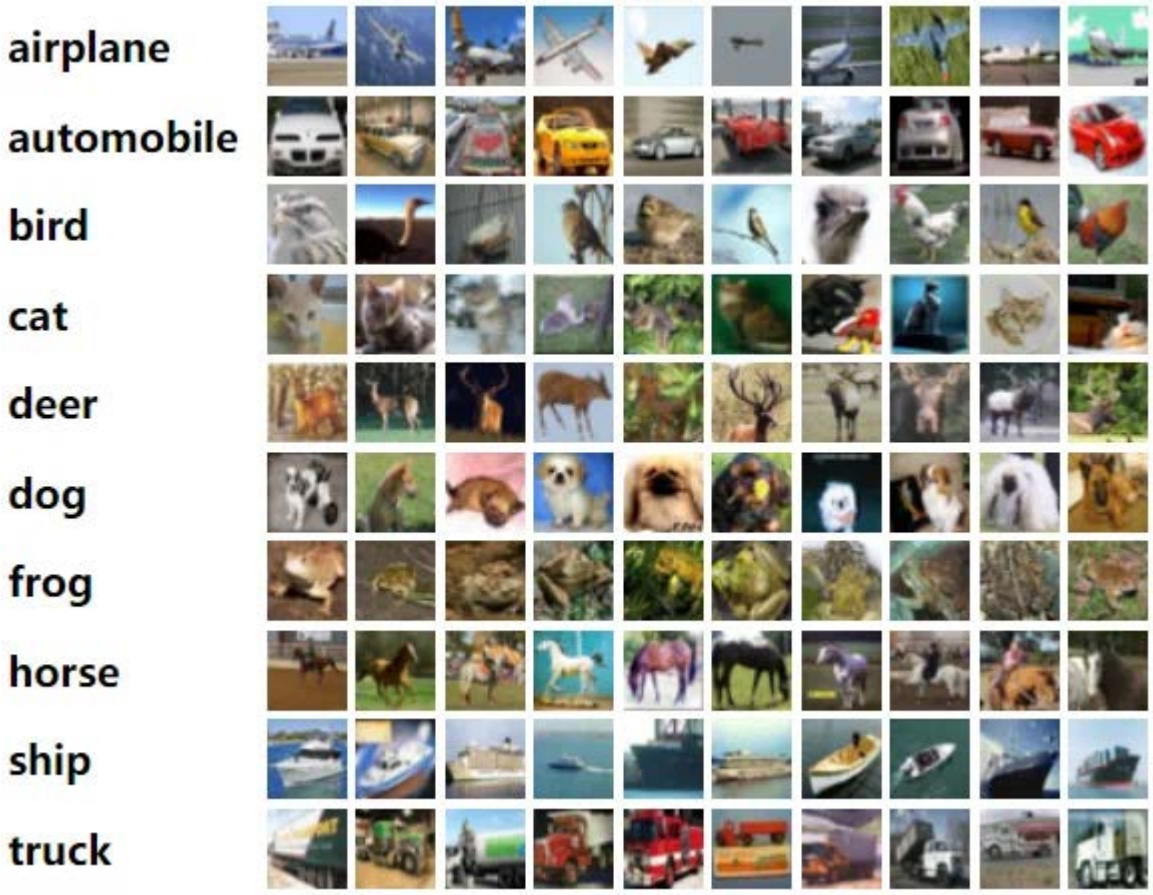



Рисунок 2.8 – Набір класів обраного датасету Cifar10

Було здійснено порівняльне дослідження з метою оцінити вплив розміру партії (*batch size*) на середнє значення похибки під час навчання нейронної мережі, її точність на валідаційній вибірці, а також ефективність розпізнавання на тестових даних. Навчання проводилося протягом 10 епох.

У якості методу оптимізації було обрано алгоритм *Adam* (*Adaptive Moment Estimation*) [17], який поєднує переваги методів *Momentum* та *RMSProp*. На відміну від класичного стохастичного градієнтного спуску (*SGD*), *Adam* автоматично адаптує швидкість навчання для кожного параметра, що дозволяє досягати стабільнішої та швидшої збіжності, особливо на складних задачах з великими об’ємами даних.

Результати експерименту для різних розмірів партії - 10, 100 та 1000 - наведено у таблицях 2.1- 2.3 відповідно.

Таблиця 2.1 - Середня похибка навчання мережі (різні розміри партії)

| Назва мережі\Розмір партії | 10 | 100 | 1000 |
|---|---|---|---|
| ResNet-34 | 0.69 | 0.57 | 0.01 |
| Inception | 1.76 | 1.78 | 0.15 |

Таблиця 2.2 - Середня похибка розпізнавання -вибірка валідації (різні розміри партії)

| Назва мережі\Розмір партії | 10 | 100 | 1000 |
|---|---|---|---|
| ResNet-34 | 1.00 | 1.02 | 1.43 |
| Inception | 1.75 | 2.3 | 0.95 |

Таблиця 2.3 - Якість розпізнавання - вибірка тестова (різні розміри партії)

| Назва мережі\Розмір партії | 10 | 100 | 1000 |
|---|---|---|---|
| ResNet-34 | 70% | 70% | 71% |
| Inception | 61% | 59% | 78% |

У процесі дослідження було встановлено, що збільшення розміру партії позитивно впливає на точність розпізнавання зображень у моделі *Inception*,

тоді як на архітектуру *ResNet* це практично не чинить впливу. Водночас, надмірне збільшення розміру партії може спричинити зниження загальної якості розпізнавання.

Наступний етап експерименту був спрямований на аналіз того, як кількість епох навчання впливає на ефективність навчального процесу та точність класифікації на валідаційній і тестовій вибірках. Розмір партії в цьому випадку складав 100, а в ролі оптимізатора, як і раніше, використовувався алгоритм *Adam*. Результати наведено у таблицях 2.4 - 2.6.

Таблиця 2.4 - Середня похибка навчання мережі (різна кількість епох)

| Назва мережі\Розмір партії | 5 | 10 | 15 |
|---|---|---|---|
| ResNet-34 | 0.78 | 0.67 | 0.60 |
| Inception | 1.90 | 1.72 | 1.53 |

Таблиця 2.5 - Середня похибка розпізнавання - вибірка валідації (різна кількість епох)

| Назва мережі\Розмір партії | 5 | 10 | 15 |
|---|---|---|---|
| ResNet-34 | 0.9 | 0.6 | 0.74 |
| Inception | 1.9 | 1.72 | 0.73 |

Таблиця 2.5 - Якість розпізнавання - вибірка тестова (різна кількість епох)

| Назва мережі\Розмір партії | 5 | 10 | 15 |
|---|---|---|---|
| ResNet-34 | 68% | 70% | 73% |
| Inception | 79% | 86% | 90% |

У процесі дослідження встановлено, що збільшення кількості епох позитивно впливає на точність розпізнавання для обох архітектур. Це

пояснюється тим, що мережі отримують більше можливостей для оптимального налаштування вагових коефіцієнтів.

У межах наступного експерименту досліджувався вплив застосування фільтрів на якість розпізнавання зображень. Для тестування було використано 10 зображень, що належать до класу "Кінь" (результати у таблиці 2.7).

Таблиця 2.7 - Якість розпізнавання одного класу (різні фільтри)

| Назва мережі/фільтр | Без фільтру | Розмиття | Шум | Чорно-білий колір |
|---|---|---|---|---|
| ResNet-34 | 60% | 30% | 60% | 40% |
| Inception | 70% | 10% | 60% | 40% |

У ході експерименту було встановлено, що застосування різноманітних фільтрів до зображень негативно впливає на точність їх розпізнавання. Модель *Inception* продемонструвала вищу точність при обробці зображень без фільтрації, тоді як при розмитті зображень кращі результати показала мережа *ResNet34*. За умови використання інших типів фільтрів результати обох моделей були приблизно однаковими.

У наступному дослідженні аналізувався вплив якості зображення на точність класифікації. Для цього було відібрано 10 зображень класу "Кінь", які поетапно зменшувалися у якості на 95%, 75%, 50% та 35% (результати у таблиці 2.8).

Таблиця 2.7 - Якість розпізнавання одного класу (різна якість)

| Назва мережі/фільтр | 95% | 75% | 50% | 30% |
|---|---|---|---|---|
| ResNet-34 | 40% | 60% | 60% | 60% |
| Inception | 80% | 70% | 70% | 70% |

У межах цього розділу було проаналізовано вплив різних параметрів навчання згорткових нейронних мереж на якість розпізнавання зображень. Було розглянуто дві сучасні архітектури - *ResNet34* та *Inception*, які реалізують різні підходи до обробки вхідних даних.

З’ясовано, що:

– Збільшення розміру партії позитивно впливає на точність моделі *Inception*, тоді як *ResNet34* виявився менш чутливим до цього параметра. Проте надто великі розміри партії можуть знижувати загальну якість розпізнавання.

– Збільшення кількості епох сприяє покращенню результатів обох архітектур завдяки ефективнішому налаштуванню вагових коефіцієнтів.

– При накладанні фільтрів обробки зображень спостерігається зниження точності класифікації. Модель *Inception* краще справляється з незміненими зображеннями, тоді як *ResNet34* демонструє вищу стійкість до розмиття.

– Зниження якості зображення (до 35%) не має суттєвого впливу на результати розпізнавання. Обидві моделі залишаються стабільними, з незначною перевагою *Inception* за точністю.

Усі експерименти проводилися із використанням оптимізатора *Adam*, який забезпечив ефективне та швидке навчання моделей. Результати дослідження можуть бути корисними при виборі архітектури та налаштуванні параметрів для задач класифікації зображень.

## 2.4 Приклад карт ознак зображення

Розглянемо більш детально процес отримання карт ознак зображення, для цього розглянемо процес на прикладі одного зображення (рис. 2.9) з датасету

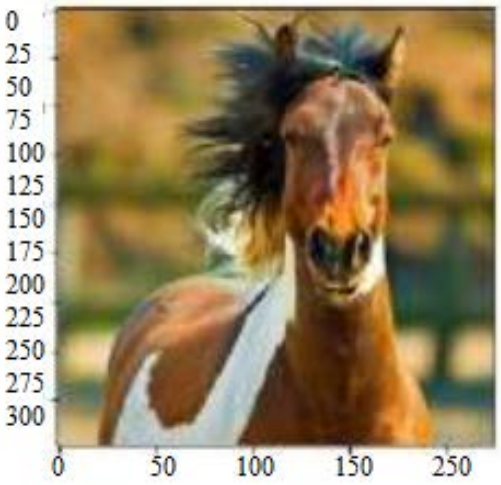



Рисунок 2.9 – Обране зображення

Для візуалізації роботи алгоритмів двох згорткових нейронних мереж були обрані три варіанти обробки зображень, які залежать від розташування шарів у мережі: на початковому етапі, в середині та в кінці роботи алгоритму. Вибір саме трьох випадків зумовлений різною архітектурою згорткових мереж, що можуть містити різну кількість шарів. Оскільки кількість вихідних карт ознак варіюється від 64 до 1024, для зручності візуалізації буде представлено лише частину отриманих карт ознак у вигляді блоку 3x3, що відображає верхній лівий кут кожної карти ознак зображення.

На початковому етапі роботи алгоритму мережі *ResNet* при розмірі фільтру 28x28 і кількості фільтрів 128 отримуємо 128 карт ознак. Результати роботи цієї мережі наведено на рисунку 2.10.

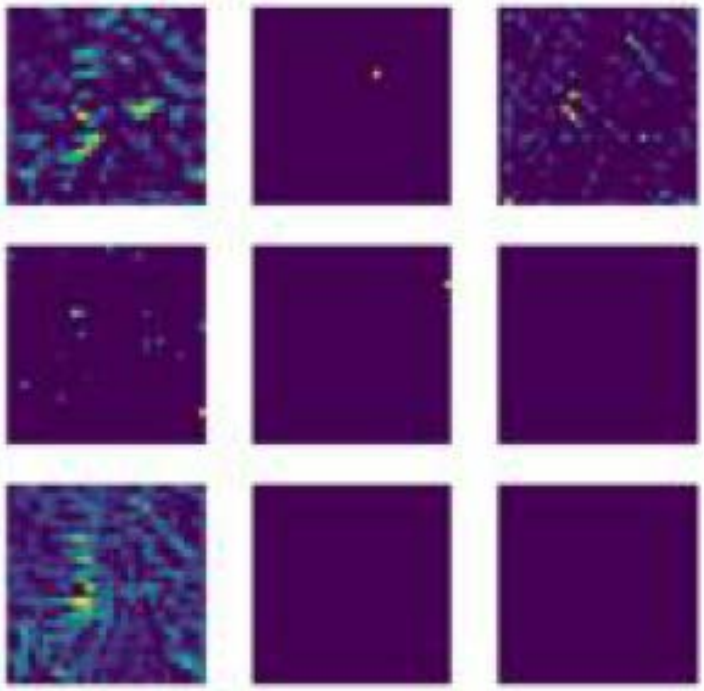


Рисунок 2.10 – Початок роботи *ResNet* (карта ознак)

У середині процесу роботи алгоритму мережі *ResNet*, при використанні фільтру розміром 14x14 та 256 фільтрів, генерується 256 карт ознак. Результати роботи мережі можна побачити на рисунку 2.11.

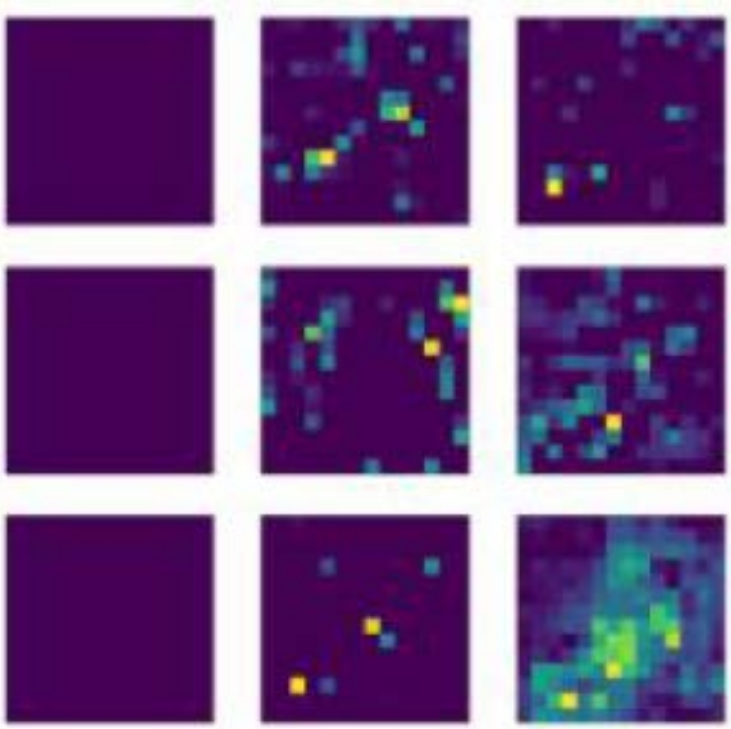


Рисунок 2.11 – Середина роботи *ResNet* (карта ознак)

Наприкінці виконання алгоритму мережі *ResNet*, при розмірі фільтру 7x7 та 512 фільтрах, генерується 512 карт ознак. Результати роботи мережі зображено на рисунку 2.12.

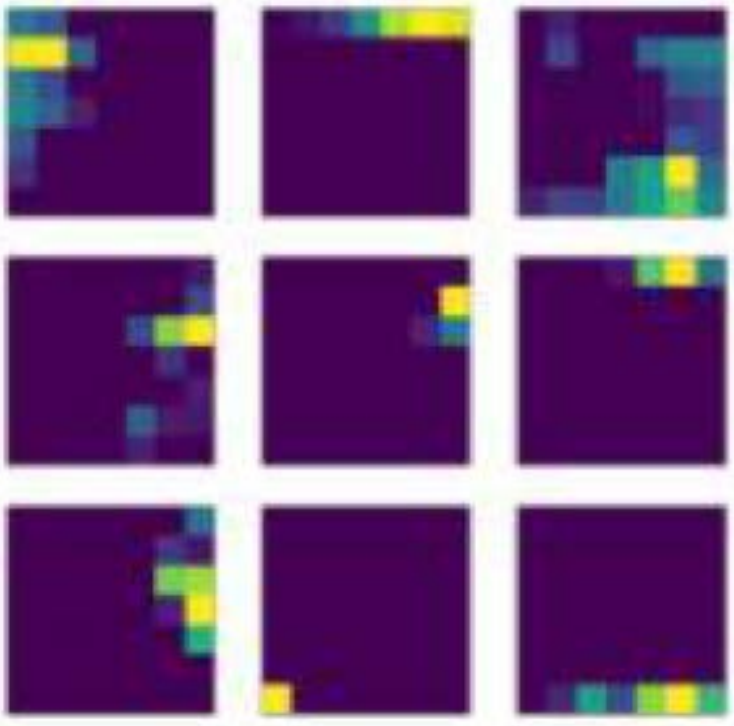


Рисунок 2.12 – Кінець роботи *ResNet* (карта ознак)

На початковому етапі роботи алгоритму мережі *Inception*, при використанні фільтру розміром 28x28 та 480 фільтрів, генерується 480 карт ознак. Результати роботи цієї мережі можна побачити на рисунку 2.13.

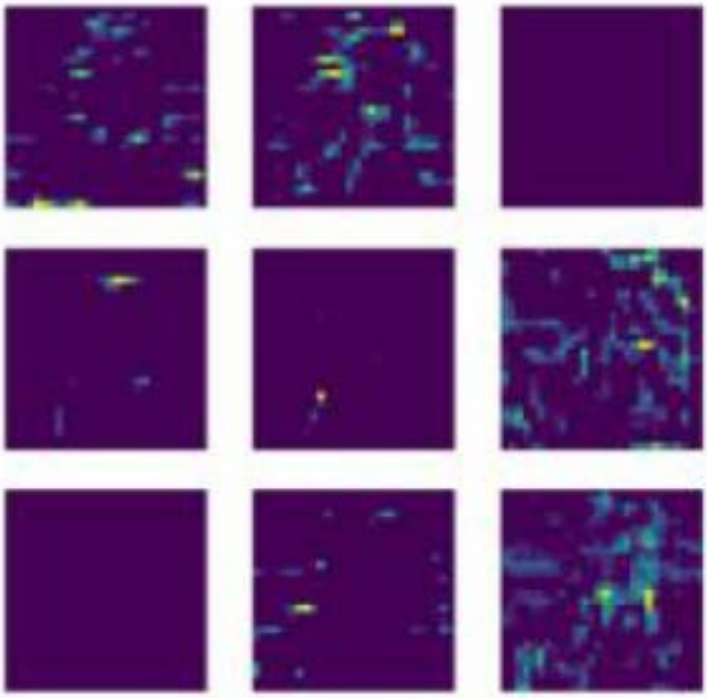


Рисунок 2.13 – Початок роботи *Inception* (карта ознак)

У середині процесу роботи алгоритму мережі *Inception*, при розмірі фільтру 14x14 та 832 фільтрах, генерується 832 карти ознак. Результати роботи мережі представлені на рисунку 2.14.

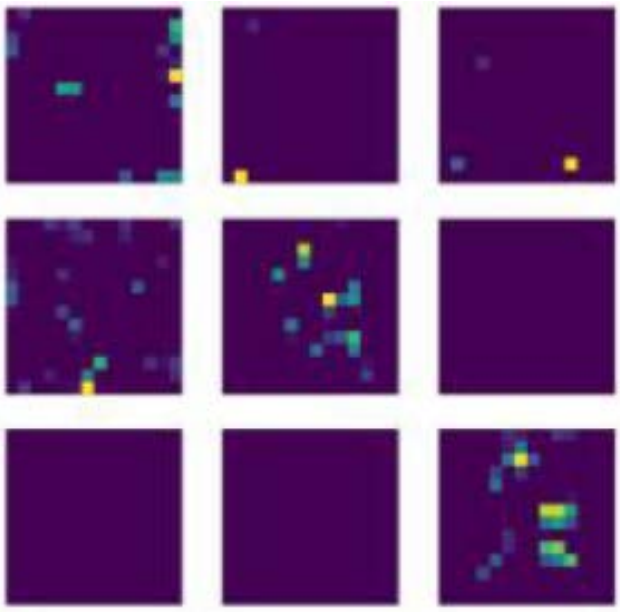


Рисунок 2.14 – Середина роботи *Inception* (карта ознак)

Наприкінці роботи алгоритму мережі *Inception*, при розмірі фільтру 7x7 та 1024 фільтрах, генерується 1024 карти ознак. Результати роботи цієї мережі можна побачити на рисунку 2.15.

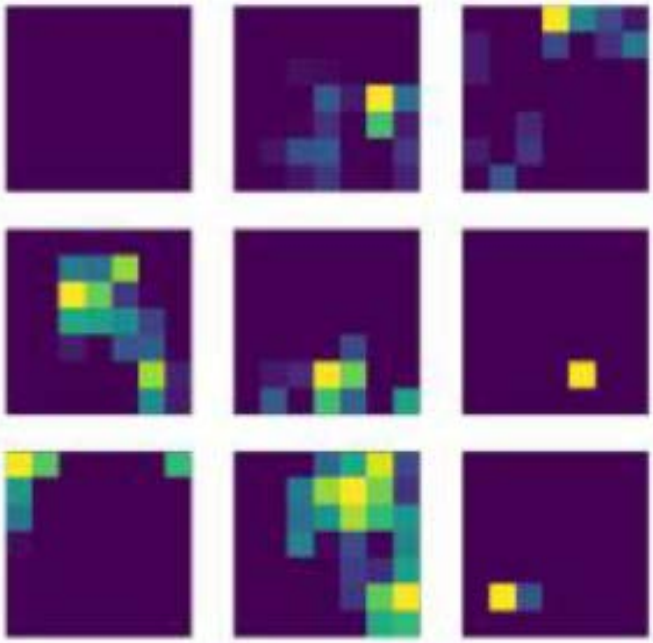


Рисунок 2.15 – Кінець роботи *Inception* (карта ознак)

У цьому розділі було проаналізовано ефективність роботи двох згорткових нейронних мереж, *ResNet* та *Inception*, на різних етапах обробки зображень. Для кожної мережі були визначені три етапи: початковий, середній та кінцевий, де вивчався вплив різних розмірів фільтрів і кількості фільтрів на створення карт ознак.

– *ResNet* показала поступове збільшення кількості карт ознак на кожному етапі: на початку з 128 карт ознак, потім 256 у середині процесу, і 512 на кінцевому етапі.

– *Inception* продемонструвала схожу динаміку: на початку було отримано 480 карт ознак, у середині - 832, а наприкінці - 1024 карти ознак.

Ці результати підтверджують ефективність застосування згорткових фільтрів різного розміру на різних етапах навчання та з'ясування значення глибини нейронної мережі для покращення якості обробки зображень.

## 2.5 Розробка програмного забезпечення

Для реалізації архітектур згорткових нейронних мереж було використано мову програмування *Python* [12], а також безкоштовні відкриті бібліотеки *TensorFlow* [13] та *Keras* [14]. Розробка проводилась у хмарному середовищі *Google Colab* [15].

*Python* - це інтерпретована, багатоплатформна мова програмування, яка відзначається простим синтаксисом, зручністю для навчання та розробки, а також потужною стандартною бібліотекою з великою кількістю вбудованих функцій. Завдяки інтерактивному середовищу *Python* дозволяє швидко тестувати та відлагоджувати код, що є важливим аспектом при розробці складних алгоритмів, таких як нейронні мережі.

*TensorFlow* - це потужна бібліотека з відкритим кодом для розробки та навчання моделей машинного навчання, зокрема нейронних мереж. Вона дозволяє створювати складні моделі для вирішення задач класифікації зображень, обробки тексту та інших завдань. *TensorFlow* забезпечує високопродуктивне обчислення за допомогою графічних процесорів (*GPU*), що значно прискорює процес навчання нейронних мереж, зокрема для великих наборів даних та складних моделей.

*Keras* є високорівневою бібліотекою, що працює поверх *TensorFlow* і спеціалізується на розробці моделей глибокого навчання. Вона значно спрощує створення, налаштування та тренування нейронних мереж завдяки інтуїтивно зрозумілому інтерфейсу та набору абстракцій, що дозволяють швидко створювати складні моделі. *Keras* дозволяє зосередитись на високорівневих аспектах, таких як архітектура мережі, залишаючи низькорівневі деталі *TensorFlow* для автоматичного виконання. Завдяки цьому, *Keras* є дуже популярною серед дослідників та розробників, оскільки спрощує процес створення моделей без втрати гнучкості.

*Google Colab* - це хмарна платформа для програмування на *Python*, що пропонує потужне середовище для створення, виконання та обміну кодом.

*Colab* надає користувачам можливість писати код у вигляді окремих блоків, які можна виконувати по черзі, що сприяє зручному та організованому процесу розробки. Однією з основних переваг є доступ до потужних графічних процесорів (*GPU*) та тензорних процесорів (*TPU*), що значно прискорює навчання моделей, роблячи платформу дуже корисною для обробки великих обсягів даних і складних обчислень.

## 2.6 Опис шарів *Keras*, що будуть використані

### 2.6.1 Шар Conv2D

Conv2D - це базовий згортковий шар, який застосовується до двовимірних зображень і виконує операцію згортки. Він є одним з ключових елементів згорткових нейронних мереж і відповідає за виявлення просторових ознак (*features*), таких як краї, текстури та форми. Цей шар формує фільтри, які навчаються автоматично під час тренування моделі.

*Синтаксис створення шару:*

```
Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', activation=None)
```

*Основні параметри:*

– *filters* - ціле число, яке визначає кількість фільтрів, що будуть застосовані в цьому шарі. Кожен фільтр виводить окрему карту ознак.

– *kernel_size* - розмір фільтра, який може бути заданий як одне число (для квадратного фільтра), або як кортеж (наприклад, (3, 3)).

– *strides* - крок переміщення фільтра по зображенню. Може бути цілим числом або кортежем. Чим більший *stride,* тим менше перекриття між зонами обробки.

– *padding*- параметр, який визначає, як обробляються межі зображення. Значення *'valid'* означає, що обтинання відбувається без доповнення, у той час як *'same'* дозволяє зберегти розмір вхідного зображення, додаючи нулі по краях.

– *activation* - функція активації, яка застосовується до вихідних значень нейронів. Наприклад, *relu, sigmoid* або *tanh.*

Цей шар є фундаментальним для побудови моделей обробки зображень, оскільки дозволяє виявляти локальні залежності та ключові ознаки на різних рівнях глибини нейронної мережі.

### 2.6.2 Шар *MaxPooling2D*

*MaxPooling2D* (*Шар вибіркової агрегації*) - це шар підвибірки, який виконує операцію максимального пулінгу. Його основне завдання - зменшити просторові розміри карти ознак (*height* × *width*), залишаючи найважливішу інформацію. Це дозволяє зменшити обчислювальну складність мережі, зменшити кількість параметрів, а також зменшити ризик перенавчання.

*Синтаксис створення шару:*

```
MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid')
```

*Основні параметри:*

– *pool_size* - розмір вікна, в межах якого буде вибиратися максимальне значення. Задається як кортеж або ціле число (наприклад, (2, 2) означає, що береться максимум із кожного блоку 2×2).

– *strides* - крок переміщення вікна по зображенню. Якщо не вказаний, за замовчуванням дорівнює *pool_size.*

– *padding* - визначає спосіб обробки меж карти ознак:

  – *'valid'* - без доповнення: вихідні розміри зменшуються.

  – *'same'* - з доповненням нулями: розміри карти ознак зберігаються.

Цей шар не має параметрів, які навчаються, і служить виключно для зменшення розмірів і виділення найважливіших ознак. Зазвичай його застосовують після згорткового шару.

### 2.6.3 Шар *AveragePooling2D*

*AveragePooling2D* (Шар усереднювальної підвибірки) - це тип підвибіркового шару, який замість вибору максимального значення у фільтрувальному вікні, як це робить *MaxPooling2D,* обчислює середнє значення всіх елементів у кожному підвибірковому вікні. Це дозволяє згладжувати карту ознак і зменшувати розміри вихідних даних, зберігаючи при цьому загальну інформацію про ознаки.

*Синтаксис створення шару:*

```
AveragePooling2D(pool_size=(2, 2), strides=None, padding='valid')
```

*Основні параметри:*

- *pool_size* - розмір фільтрувального вікна, що визначає область, по якій обчислюється середнє значення. Задається як кортеж або ціле число, наприклад (2, 2) означає обчислення середнього в кожному блоку розміром 2×2.
- *strides* - крок пересування вікна по зображенню. Якщо не заданий явно, за замовчуванням використовується значення *pool_size*.
- *padding* - визначає, як обробляються межі зображення:
  - 'valid' - без доповнення: після операції зменшується розмір вихідного зображення.
  - 'same' - з доповненням: розміри карти ознак залишаються незмінними.

*AveragePooling2D* корисний у випадках, коли важливо зменшити розмірність і при цьому зберегти загальну тенденцію ознак, а не лише найяскравіші з них. Такий підхід іноді забезпечує кращу узагальненість у задачах, де максимальні значення не є критичними.

### 2.6.4 Шар *Activation*

*Activation (*Шар активації попередніх даних*) — це спеціалізований шар, який застосовує* функцію активації до вхідних даних. Активаційна функція

дозволяє нейронній мережі моделювати складні нелінійні залежності, що робить її здатною до вирішення складних задач, таких як класифікація, регресія чи розпізнавання зображень.

Синтаксис створення шару:

```
Activation(activation)
```

*Основний параметр:*

– *activation* - назва функції активації, яка буде застосована до вхідних даних. Найпоширеніші варіанти:

– *'relu' (Rectified Linear Unit)* - найбільш часто використовувана функція у згорткових мережах, добре справляється з проблемою зникнення градієнта.

– *'sigmoid'* - стискає вихід до діапазону [0, 1], часто використовується в задачах бінарної класифікації.

– *'tanh'* - стискає значення до діапазону [-1, 1].

– *'softmax'* - застосовується на останньому шарі для багатокласової класифікації.

Шар *Activation* може бути використаний як окремий етап в побудові моделі або вбудований безпосередньо в інші шари, наприклад *Dense* чи *Conv2D*, через параметр *activation.*

Застосування функцій активації є критично важливим для ефективного навчання нейронної мережі, оскільки саме вони додають мережі здатність до навчання на складних, нелінійних залежностях у даних.

### 2.6.5 Шар *Add*

*Add* - це шар, який виконує операцію підсумовування кількох вхідних даних. Цей шар корисний, коли потрібно комбінувати декілька вхідних даних у єдиний результат. Всі дані, які подаються на вхід, повинні мати однакові розміри, щоб операція підсумовування була коректною.

*Синтаксис створення шару:*

```
Add(inputs)
```

*Основний параметр:*

– *inputs* - список або кортеж з вхідних тензорів однакового розміру, які будуть підсумовані. Після застосування цього шару результатом буде єдиний тензор, що містить суму всіх вхідних тензорів.

Шар Add зазвичай використовується в архітектурах, де потрібно комбінувати кілька різних характеристик або результатів, отриманих від інших шарів. Це може бути корисно для реалізації пропуску зв'язків (*skip connections*) або мостових (*shortcut*) з'єднань між шарами в складних нейронних мережах, таких як *ResNet*, де інформація передається як сума з попереднього шару.

Цей підхід дозволяє моделі вчитися на більшій кількості варіантів даних і зберігати важливі ознаки, навіть якщо певні шари не виявили суттєвої інформації.

### 2.6.6 Шар *Concatenate*

*Concatenate* - це шар, який виконує операцію об’єднання декількох вхідних тензорів у один великий тензор. Операція об'єднання зазвичай застосовується для комбінування різних характеристик або особливостей, отриманих від різних шарів або етапів мережі. Шар повертає єдиний тензор того ж розміру за винятком осі з’єднання, де відбувається об’єднання всіх вхідних тензорів.

*Синтаксис створення шару:*

```
Concatenate(axis=-1, tensors)
```

*Основні параметри:*

– *axis* - ціле число, що визначає ось, за якою буде відбуватись об’єднання вхідних тензорів.

– Значення *axis=-1* вказує на об’єднання по останній осі (по колонках для тензорів *2D*).

– Якщо потрібно об'єднати по іншій осі, слід вказати відповідне значення для *axis,* наприклад, *axis=0* для об’єднання по першій осі (по рядках*).*

– *tensors* - список або кортеж з вхідних тензорів, які потрібно об'єднати. Всі тензори повинні мати однакові розміри по всіх осях, за винятком осі об’єднання.

Шар *Concatenate* корисний в ситуаціях, коли потрібно поєднати різні види інформації, що отримуються з кількох шарів. Наприклад, в деяких архітектурах глибоких нейронних мереж, таких як *Inception* або *DenseNet,* де різні шари можуть генерувати окремі особливості, які потім об’єднуються для подальшої обробки.

Об’єднання тензорів за допомогою цього шару дозволяє моделі зберігати більше інформації, що може сприяти поліпшенню точності та загальної ефективності моделі.

## 2.7 Розробка мережі *ResNet34*

Структура *Residual блоку* для мережі *ResNet34* наведено на рис. 2.16

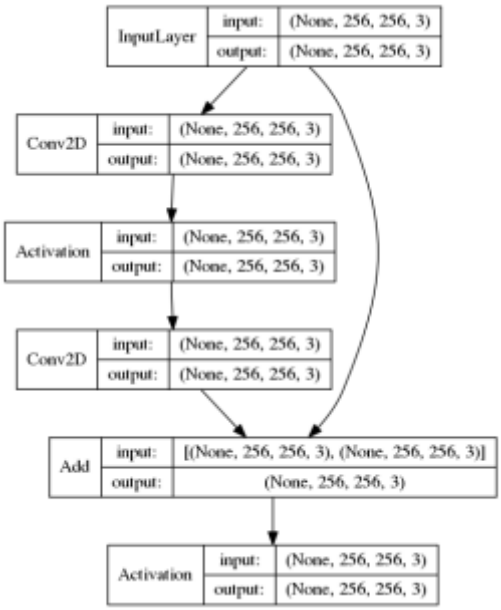



Рисунок 2.16 - Структура *Residual* блоку

На рис. 2.17 наведено лістинг створення даного блоку за допомогою Цей функції *residual_block*

```
#функція створення residual блоку мережі ResNet34
def residual_block(layer_input, filters_count, is_first=False):
merge_input = layer_input
if layer_input.shape[-1]!=filters_count:
if is_first:
merge_input = Conv2D(filters_count, (1,1),strides=(2,2),
activation='linear',kernel_initializer='he_normal')(layer_input)
else:
merge_input = Conv2D(filters_count, (1,1),strides=(1,1),
activation='linear',kernel_initializer='he_normal')(layer_input)
# batchnorm1 = BatchNormalization(axis=3,momentum=0.9,epsilon=0.0001)(laye
r_input)
# activation1 = Activation('relu')(layer_input)
if is_first:
conv1 = Conv2D(filters_count, (1,1), strides=(2,2), padding='same',
activation='relu',kernel_initializer='he_normal')(layer_input)
else:
conv1 = Conv2D(filters_count, (1,1), strides=(1,1), padding='same',
activation='relu',kernel_initializer='he_normal')(layer_input)
#batchnorm2 = BatchNormalization(axis=3)(conv1)
conv2 = Conv2D(filters_count, (3,3), padding='same',
activation='linear',kernel_initializer='he_normal')(conv1)
layer_out = add([conv2, merge_input])
activation2 = Activation('relu')(layer_out)
return activation2
```

Рисунок 2.17 – Лістинг коду створення *Residual* блоку

– *layer_input* - шар мережі, з якого будуть надходити вхідні дані для обробки в блок.

– *filters_count* - кількість фільтрів, які будуть використовуватись у кожній згортці всередині блоку.

– *is_first* - булевий параметр, який визначає, чи є цей блок першим у мережі.

Якщо цей блок є першим, для вхідних даних спочатку застосовується згортка з фільтром розміру 1x1, що має *filters_count* фільтрів. Це дозволяє підготувати дані для подальших операцій в блоці.

Мережа з архітектурою *ResNet34* створюється за допомогою функції *create_ResNet* (Рис. 2.18).

```
def create_ResNet():
ResNet34_input = Input(shape=(224, 224, 3))
conv1 = Conv2D(64, (7,7), strides=(2,2), padding='same', activation='relu',
kernel_initializer='he_normal')(ResNet34_input)
pool1 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(conv1)
residual_block64_1 = residual_block(pool1, 64)
residual_block64_2 = residual_block(residual_block64_1,64)
residual_block64_3 = residual_block(residual_block64_2,64)
residual_block128_1 = residual_block(residual_block64_3,128, is_first=True)
residual_block128_2 = residual_block(residual_block128_1,128 )
residual_block128_3 = residual_block(residual_block128_2,128 )
residual_block128_4 = residual_block(residual_block128_3,128 )
residual_block256_1 = residual_block(residual_block128_4,256, is_first=True)
residual_block256_2 = residual_block(residual_block256_1,256)
residual_block256_3 = residual_block(residual_block256_2,256)
residual_block256_4 = residual_block(residual_block256_3,256)
residual_block256_5 = residual_block(residual_block256_4,256)
residual_block256_6 = residual_block(residual_block256_5,256)
residual_block512_1 = residual_block(residual_block256_6,512, is_first=True )
residual_block512_2 = residual_block(residual_block512_1,512)
residual_block512_3 = residual_block(residual_block512_2,512)
activation = Activation('relu')(residual_block512_3)
pool2 = AveragePooling2D(pool_size=(7,7),strides=(1,1))(residual_block512_3)
ResNet34 = Model(inputs=ResNet34_input, outputs=pool2)
return ResNet34
```

Рисунок 2.18 – Лістинг коду створення мережі *ResNet34*

Ця функція не приймає жодних аргументів, але повертає побудовану мережу. Вона включає всі необхідні шари, включаючи кілька *Residual* блоків, що дозволяють мережі ефективно навчатися через глибину та покращувати продуктивність завдяки механізму пропуску (*skip connections*).

## 2.8 Розробка мережі *Inception*

Структура *Inception блоку* для мережі *Inception* наведено на рис. 2.19. Цей *Inception* блок створюється за допомогою функції *inception_block*, лістинг коду реалізації якого наведено на рисунку 2.20.

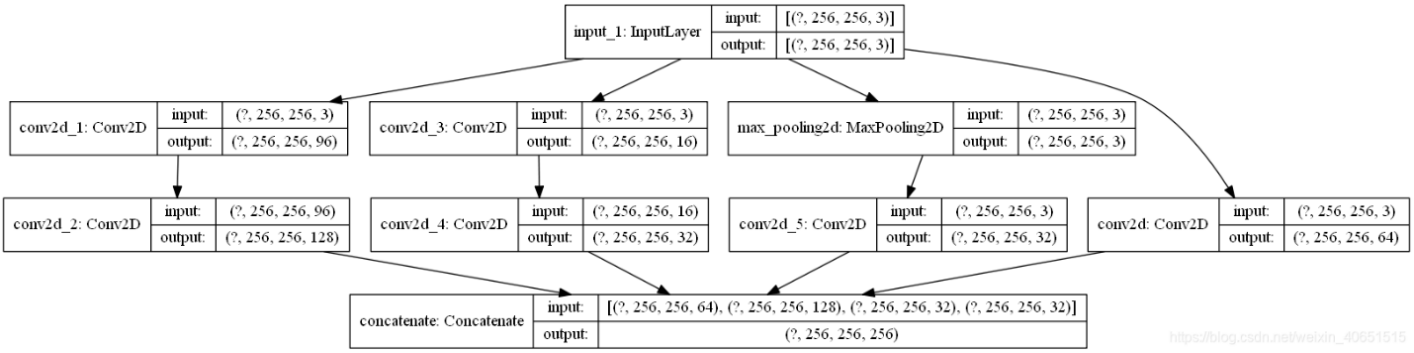



Рисунок 2.19 – Структура *Inception* блоку

```
#функція створення inception блоку мережі Inception
def inception_block(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
conv1 = Conv2D(f1, (1,1), padding='same', activation='relu',
kernel_initializer='he_normal')(layer_in)
conv2 = Conv2D(f2_in, (1,1), padding='same', activation='relu',
kernel_initializer='he_normal')(layer_in)
conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu',
kernel_initializer='he_normal')(conv2)
conv4= Conv2D(f3_in, (1,1), padding='same', activation='relu',
kernel_initializer='he_normal')(layer_in)
conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu',
kernel_initializer='he_normal')(conv4)
pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
conv6 = Conv2D(f4_out, (1,1), padding='same', activation='relu',
kernel_initializer='he_normal')(pool)
layer_out = concatenate([conv1, conv3, conv5, conv6], axis=-1)
return layer_out
```

Рисунок 2.20 - Лістинг коду створення *Inception* блоку

Блок містить наступні аргументи:

– *layer_in* - вхідний шар мережі, з якого будуть надходити дані для подальшої обробки в Inception блоці.

– *f1* - кількість фільтрів, які використовуються в першому підблоці *block(a).*

– *f2_in* - кількість фільтрів, які застосовуються на вході в підблок *block(b).*

– *f2_out* - кількість фільтрів на виході з підблоку *block(b).*

– *f3_in* - кількість фільтрів на вході в підблок *block(c).*

– *f3_out* - кількість фільтрів на виході з підблоку *block(c)*.

– *f4_out* - кількість фільтрів на виході з підблоку *block(d)*.

Важливою особливістю *Inception* блоку є те, що він поєднує кілька різних операцій згортки з різними розмірами фільтрів, що дозволяє мережі *Inception* обирати найкращі характеристики з різних рівнів деталі вхідного зображення.

Мережа з архітектурою *Inception* створюється за допомогою функції *create_Inception* (Рис. 2.21).

```
#функція створення мережі inception
def create_Inception():
Inception_input = Input(shape=(224, 224, 3))
conv1 = Conv2D(64, (7,7), strides=(2,2), padding='same', activation='relu',
kernel_initializer='he_normal')(Inception_input)
pool1 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(conv1)
conv2 = Conv2D(192, (3,3), strides=(1,1), padding='same', activation='relu',
kernel_initializer='he_normal')(pool1)
pool2 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(conv2)
inception_block1 = inception_block(pool2, 64, 96, 128, 16, 32, 32)
inception_block2 = inception_block(inception_block1, 128, 128, 192, 32, 96, 64)
pool3 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(inception_
block2)
inception_block3 = inception_block(pool3, 192, 96, 208, 16, 48, 64)
inception_block4 = inception_block(inception_block3, 160, 112, 224, 24, 64, 64)
inception_block5= inception_block(inception_block4, 128, 128, 256, 24, 64, 64)
inception_block6 = inception_block(inception_block5, 112, 144, 288, 32, 64, 64)
inception_block7 = inception_block(inception_block6, 256, 160, 320, 32, 128, 12
8)
pool4 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(inception_
block7)
inception_block8 = inception_block(pool4, 256, 160, 320, 32, 128, 128)
inception_block9 = inception_block(inception_block8, 384, 192, 384, 48, 128, 12
8)
pool5 = AveragePooling2D(pool_size=(7,7),strides=(1,1))(inception_block9)
dropout = Dropout(0.4)(pool5)
Inception = Model(inputs=Inception_input, outputs=dropout)
return Inception
```

Рисунок 2.21 - Лістинг коду створення мережі *Inception*

## 2.9 Процес навчання мережі

На рисунку 2.22 представлена реалізація функції попередньої обробки даних, яка виконує дві ключові задачі перед передачею зображень до нейронної мережі:

1. *Нормалізація зображень* – значення пікселів у вхідних зображеннях перетворюються в діапазон [0, 1], що досягається шляхом поділу кожного значення на 255. Це забезпечує стабільність і швидкість навчання нейронної мережі, оскільки масштаб вхідних даних стає однорідним.

2. *Кодування міток (label encoding)* – кожна мітка класу перетворюється у вектор з десяти елементів, де елемент з індексом k (що відповідає номеру класу зображення) дорівнює 1, а всі інші - 0. Такий підхід відомий як *one-hot* кодування і широко використовується для задач багатокласової класифікації.

```
# Функція для попередньої обробки даних
def preprocess_input_data(x, y):
x_p = x/255.0
y_p = keras.utils.to_categorical(y, 10)
return x_p, y_p
```

Рисунок 2.22 – Лістинг коду функції попередньої обробки

Ця функція готує дані у відповідному форматі для подальшого ефективного навчання згорткових нейронних мереж.

На рисунку 2.23 зображено лістинг коду, що відповідає за завантаження та підготовку датасету *CIFAR-10* - одного з найпопулярніших наборів зображень, який містить 60 000 кольорових зображень розміром 32×32 пікселі, розподілених на 10 категорій (наприклад: літаки, автомобілі, птахи, коти тощо).

```
# Завантаження датасету та його попередня обробка
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
x_train, y_train = preprocess_input_data(x_train, y_train)
x_test, y_test = preprocess_input_data(x_test, y_test)
```

Рисунок 2.23 – Лістинг коду завантаження та підготовки датасету

Після завантаження дані проходять попередню обробку за допомогою раніше реалізованої функції *preprocess_input_data*. Ця функція:

– нормалізує зображення, перетворюючи значення пікселів у діапазон [0, 1], що сприяє стабільнішому та швидшому навчанню мережі;

– перетворює мітки класів у формат *one-hot* - вектор з 10 елементів, у якому лише одна позиція відповідає певному класу і має значення 1, а всі інші - 0.

Таким чином, блок забезпечує правильну підготовку вхідних даних для подальшого використання у згорткових нейронних мережах.

На рисунку 2.24 представлено лістинг коду, який відповідає за конфігурацію архітектури нейронної мережі. У цьому блоці виконується масштабування вхідних зображень до потрібного розміру (наприклад, 224×224 пікселі), що дозволяє адаптувати вхідні дані до вимог конкретної архітектури моделі, наприклад *ResNet34* або *Inception.*

```
# Задавання розміру вхідних даних
size = 224 #ширина та висота вхідного зображення
model = create_ResNet()
# Налаштування архітектури моделі
train_model = keras.models.Sequential()
train_model.add(keras.layers.Lambda(lambda image: tf.image.resize(image, (size,s
ize))))
train_model.add(model)
train_model.add(Flatten())
train_model.add(Dense(10, activation='softmax'))
plot_model(model, show_shapes=True, to_file='model_arch1.png')
train_model.build(input_shape=(None,224,224,3))
```

Рисунок 2.24 – Лістинг коду налаштування для класифікації

Після цього до основної моделі додаються додаткові шари класифікації, які включають:

– шари згладжування (flatten), що перетворюють багатовимірні тензори у вектори;

– щільні (Dense) шари для побудови логіки класифікації;

– активаційний шар (зазвичай Softmax) для отримання ймовірностей належності зображення до певного класу.

Цей блок дозволяє завершити побудову повної архітектури, яка об’єднує переднавчену модель (або власну базову структуру) з кастомним класифікатором для вирішення задачі розпізнавання зображень.

Лістинг коду, зображений на рисунку 2.25, відповідає за конфігурацію моделі перед процесом навчання.

```
# Налаштування моделі для навчання
train_model.compile(loss='categorical_crossentropy',
optimizer=keras.optimizers.Adam(),
metrics=['accuracy'])
```

Рисунок 2.25 – Лістинг коду навчання моделі

У цьому етапі визначаються ключові компоненти, необхідні для ефективного навчання нейронної мережі:

– Функція втрат (*loss*) - визначає, як модель буде оцінювати різницю між передбаченими значеннями та реальними мітками. Для задач багатокласової класифікації зазвичай використовується функція *categorical_crossentropy*.

– Оптимізатор (*optimizer*) - відповідає за оновлення ваг мережі під час зворотного поширення помилки. У цьому блоці часто використовується оптимізатор *Adam*, який поєднує в собі переваги методів *Adagrad* та *RMSProp*, забезпечуючи швидку та стабільну збіжність.

– Метрики (*metrics*) - набір показників, які обчислюються під час навчання та тестування моделі. Найчастіше використовується метрика *accuracy* (точність), що дозволяє оцінити, наскільки добре модель класифікує вхідні зображення.

Цей блок є обов’язковим перед викликом методу *fit*, оскільки без нього модель не матиме змоги коректно навчатися або оцінювати якість своїх передбачень.

Лістинг коду, представлений на рисунку 2.26, ініціює процес навчання нейронної мережі. Основним методом, що використовується, є *model.fit(...)*, який запускає навчання на підготовленому датасеті згідно з заданими параметрами:

– *batch_size* - визначає кількість зразків, які будуть оброблятися одночасно під час одного кроку навчання. Менші партії дозволяють моделі швидше адаптуватися, але можуть бути менш стабільними, тоді як більші забезпечують більш згладжене оновлення ваг.

– *epochs* - вказує, скільки разів вся навчальна вибірка буде повністю передана через модель. Збільшення кількості епох дозволяє досягти кращих результатів, але може призвести до перенавчання, якщо їх забагато.

– *validation_split* - задає частку (у відсотках) навчальних даних, які тимчасово відокремлюються для перевірки якості навчання моделі після кожної епохи. Це дає змогу відстежувати ефективність моделі на "невідомих" даних, не залучаючи окремий тестовий набір.

```
history = train_model.fit(x_train, y_train, batch_size=100, epochs=15,
validation_split=0.1, callbacks=[save_weights])
train_model = load_model('/content/drive/MyDrive/Inception/Inception_weights.1
0-0.80')
```

Рисунок 2.26 – Запуск навчання моделі

У результаті виконання цього блоку модель починає процес поступового вдосконалення своїх ваг, поступово знижуючи помилку та підвищуючи точність класифікації.

Лістинг коду, зображений на рисунку 2.27, виконує тестування нейронної мережі на незалежній тестовій вибірці (рис. 2.4). Метою цього етапу є оцінка здатності моделі узагальнювати знання на нових, раніше не бачених даних. Після завершення тестування мережа повертає два основні показники:

– Похибку (*loss*) - числове значення функції втрат, яке свідчить про відхилення передбачень моделі від реальних значень.

– Точність (*accuracy*)- відсоткове значення правильних класифікацій на тестовому наборі.

```
#Перевірка точності моделі на тестовій вибірці
test_loss, test_acc = train_model.evaluate(x_test, y_test)
print('Test loss: {}, Test accuracy: {}'.format(test_loss, test_acc * 100))
train_model.summary()
```

Рисунок 2.27– Лістинг коду перевірки точності

Ці метрики дозволяють об'єктивно оцінити ефективність навченої моделі та порівняти її з іншими архітектурами або підходами.

Лістинг коду зображений на рисунку 2.28 відповідає за візуалізацію процесу навчання нейронної мережі. У цьому блоці здійснюється побудова графіків на основі метрик, що були автоматично записані під час тренування моделі. Зокрема, за допомогою бібліотеки *matplotlib* виводяться графіки:

– Точності (*accuracy*) - відображає, наскільки правильно модель класифікує зображення з кожною епохою. Можна простежити, як покращується або погіршується якість класифікації протягом навчання.

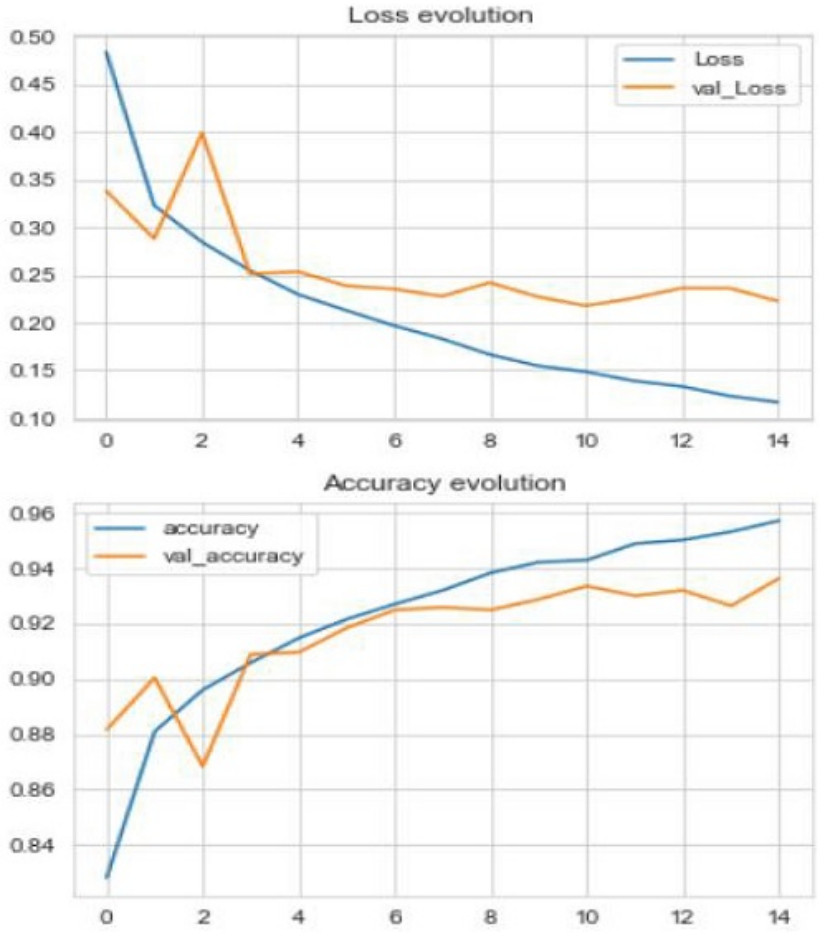
– Втрат (*loss*) - показує, як змінюється функція втрат у процесі навчання, що дозволяє оцінити збіжність моделі та ймовірність перенавчання.

```
145  # Виведення графіку метрик мережі, які записувалися під час навчання
146  plt.figure(figsize=(15, 4))
147  plt.title('Train', fontsize=17)
148  plt.xlabel('epoch')
149  plt.ylabel('value')
150  plt.subplot(1,2,1)
151  plt.plot(history.history['loss'], label='loss')
152  plt.plot(history.history['accuracy'], label='accuracy')
153  plt.legend()
154  plt.grid(True)
155  plt.subplot(1,2,2)
156  plt.plot(history.history['val_loss'],label='val_loss')
157  plt.plot(history.history['val_accuracy'],label='val_accuracy')
158  plt.legend()
159  plt.grid(True)
160  plt.show()
```

Рисунок 2.27– Лістинг коду візуалізації результату

На одному графіку зазвичай зображуються криві як для навчальної вибірки, так і для валідаційної, що дозволяє візуально порівняти ефективність моделі на обох наборах даних.

Приклад типового вигляду таких графіків представлено на рисунку 2.28. Їх аналіз дає змогу зробити висновки про стабільність, узагальнюючу здатність та продуктивність побудованої моделі.




а) похибка і точність на тренувальній виборці; б) похибка і точність на валідаційній виборці

Рисунок 2.28– Візуалізація графіків  метрик навчання мережі

На етапі тестування модель була перевірена на незалежній тестовій вибірці, яка не використовувалася під час навчання. За отриманими результатами мережа продемонструвала задовільні показники якості, зокрема низьке значення похибки та високу точність класифікації. Це свідчить про здатність моделі ефективно узагальнювати знання та забезпечувати коректне розпізнавання образів на нових даних. Показники точності є достатніми для подальшого практичного використання моделі або її інтеграції в більшу систему автоматичної обробки зображень.

У Додатку А наведено повний лістинг коду розробленого ПЗ.

*Висновки до розділу:*

У цьому розділі було проведено всебічний аналіз двох сучасних архітектур згорткових нейронних мереж - *ResNet34* та *Inception*. Було розглянуто їх внутрішню будову, ключові особливості та переваги у вирішенні задач класифікації зображень.

В результаті дослідження:

- Детально проаналізовано принципи функціонування *ResNet*-блоків із залишковими з'єднаннями, що полегшують процес навчання глибоких мереж.
- Розглянуто архітектуру мережі *Inception,* яка поєднує фільтри різного розміру в одному шарі для кращого виявлення ознак на різних масштабах.
- Проведено експерименти з візуалізацією карт ознак на різних етапах роботи мереж, що дозволило побачити, як змінюється представлення даних у глибині нейронної мережі.
- Виконано розробку та реалізацію обох архітектур у середовищі *Google Colab* з використанням бібліотек *TensorFlow* та *Keras*.
- Наведено опис основних шарів, за допомогою яких будуються моделі, зокрема згорткові, pooling-шари, шари об'єднання та активації.
- Реалізовано етапи підготовки та нормалізації даних, а також навчання і тестування моделей.
- Зроблено порівняльний аналіз отриманих результатів, який показав, що мережа *Inception* демонструє дещо кращу точність, однак обидві архітектури є ефективними при роботі з даними *CIFAR-10.*

Таким чином, було підтверджено доцільність використання як *ResNet34,* так і *Inception* у задачах комп’ютерного зору, з можливістю подальшої адаптації під інші типи даних і задачі.

# ВИСНОВКИ

У результаті ретельного аналізу предметної області було здобуто знання про різні алгоритми та архітектури згорткових нейронних мереж, що дозволило сформувати план дослідження методів розпізнавання образів. У ході роботи досліджено дві різні архітектури нейронних мереж і здійснено порівняльний аналіз їхньої роботи за допомогою створеного програмного забезпечення.

Експериментальні результати отримано з використанням бібліотек Keras і TensorFlow, мови програмування Python та безкоштовного хмарного сервісу Google Colab. Для оцінки ефективності архітектур нейронних мереж застосовувалися алгоритми й методи глибокого навчання.

Тестування створених нейронних мереж було виконано на наборі даних CIFAR-10, що включає 50 000 кольорових зображень у тренувальній вибірці та 10 000 — у тестовій. Обидві вибірки охоплюють по 10 однакових класів зображень, при цьому валідаційна вибірка становить 10% від обсягу тренувальної.

У процесі експериментального дослідження проаналізовано вплив ключових параметрів навчання — розміру партії та кількості епох — на якість розпізнавання зображень. Отримані результати свідчать про те, що зі збільшенням розміру партії та кількості епох точність розпізнавання зростає.

У ході експериментальних досліджень було проаналізовано ефективність розпізнавання зображень із застосуванням різноманітних фільтрів. Результати показали, що найкращу точність при обробці зображень без фільтрів продемонструвала архітектура Inception, тоді як при розпізнаванні розмитих зображень найкращі результати показала мережа ResNet34. У випадку використання інших типів фільтрів результати обох мереж були подібними.

У процесі дослідження було проведено порівняльний аналіз різних архітектур нейронних мереж для задачі розпізнавання зображень за різних

умов попередньої обробки даних. Експерименти засвідчили, що мережа Inception забезпечує найвищу точність при роботі з не зміненими зображеннями, тоді як ResNet34 демонструє кращі результати при розпізнаванні зображень із розмиттям. Для інших типів фільтрів різниця між моделями була незначною. Ці результати свідчать про доцільність адаптивного підходу до вибору архітектури нейронної мережі залежно від характеру вхідних даних, що дозволяє підвищити ефективність розв’язання прикладних задач комп’ютерного зору.

Отримані результати можуть бути використані як орієнтир для вибору оптимальної архітектури нейронної мережі під конкретну прикладну задачу розпізнавання образів.

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Artificial neuron URL: https://en.wikipedia.org/wiki/ Artificial_neuron#::text=An%20artificial%20 neuron%20is%20a,in%20an% 20artificial%20 neural% 20network (Дата звернення 20.04.25)

2. RGB COLOR MODEL URL: https://en.wikipedia.org/wiki/ RGB_color_ model (Дата звернення 20.04.25)

3. Grayscale URL:https://ru.wikipedia.org/wiki/%D0%9E%D1%82% D1%82%D0%B5%D0%BD%D0%BA%D0%B8_%D1%81%D0%B5%D1% 80% D0%BE%D0%B3%D0%BE(Дата звернення 20.04.25)

4. Нормализация входных векторов (Normalization) URL: https://wiki.loginom.ru/articles/normalization.html (Дата звернення 20.04.25)

5. Zero Padding In Convolutional Neural Networks Explained URL: https://deeplizard.com/learn/video/qSTv_m-KFk0 (Дата звернення 15.03.25)

6. Convolutional neural network URL: https://en.wikipedia.org/wiki/ Convolutional_neural_network(Дата звернення 15.03.25)

7. Residual neural network URL: https://en.wikipedia.org/wiki/ Residual_neural_network (Дата звернення 15.03.25)

8. Deep Residual Learning for Image Recognition URL: https://arxiv.org/pdf/1512.03385.pdf (Дата звернення 20.04.25)

9. Going deeper with convolutions URL: https://arxiv.org/pdf/1409.4842.pdf (Дата звернення 15.03.25)

10. Identity function URL: https://en.wikipedia.org/wiki/Identity_function (Дата звернення 20.04.25)

11. Stochastic gradient descent URL: https://en.wikipedia.org /wiki/ Stochastic_gradient_descent (Дата звернення 20.04.25)

12. Python (programming language) URL: https://en.wikipedia.org/wiki/Python_(programming_language) (Дата звернення 15.03.25)

13.Бібліотека машинного навчання TensorFlow. URL: https://en. wikipedia.org/wiki/TensorFlow (Дата звернення 06.05.25)

14.Бібліотека підтримки операцій з нейронними мережами Keras URL: https://en.wikipedia.org/wiki/Keras (Дата звернення 06.05.25)

15. Google Colab URL: https://colab.research.google.com/ (Дата звернення 06.05.25)

16. Cifar10 Dataset URL: https://en.wikipedia.org/wiki/RGB_ color_ model (Дата звернення 06.05.25)

17. Adam — latest trends in deep learning optimization URL: https://towardsdatascience. com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

18. Keras functional API URL: https://keras.io/guides/functional_api/ (Дата звернення 06.05.25)

19. Штеньович В. Білоус А. Вступ до машинного навчання. URL: https://dou.ua / lenta/articles/introduction-machine-learning-1

20. N. Bartyzal. Artificial neural networks URL: http://shodhganga. inflibnet.ac. in/bitstream/10603/48/6/chaper%204_c%20b%20bangal (Дата звернення 06.05.25)

21. A. Moawad. Neural networks and back-propagation algorithm. URL: https://medium.com/datathings/neural-networks-and-backpropagationexplained in-a-simple-way-f540a3611f5e (Дата звернення 20.04.25)

22. N. Donges. Transfer Learning URL: https://towardsdatascience.com /transfer-learning-946518f95666

23. J. Deng, W. Dong, R. Socher. ImageNet: A large-scale hierarchical image database URL: https://ieeexplore.ieee.org/document/5206848 (Дата звернення 20.04.25)

24. J. Jordan. Common architectures in convolutional neural networks URL: https://www.jeremyjordan.me/convnet-architectures

25. E. Culurciello. Neural Network Architectures. Inception model URL: https://towardsdatascience.com/neural-network-architectures-156e5bad51ba77

(Дата звернення 06.05.25)

26. V. Fung. An overview of ResNet and its variants URL: https://towardsdatascience.com/an-overview-of-resnet-and-its-variants 5281e2f56035 (Дата звернення 20.04.25)

27. P. Dahal. Classification and Loss Evaluation – Softmax and Cross Entropy Loss/ URL:: https://deepnotes.io/softmax-crossentropy (Дата звернення 06.05.25)

28. Моркун Н. В., Завсегдашня І. В. Методичні вказівки до виконання кваліфікаційної роботи бакалавру для студентів спеціальності 122 “Комп’ютерні науки”. Кривий Ріг : Видавничий центр КНУ, 2021. 50 с.

29. ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура і правила оформлення. Київ, ДП «УкрННЦ», 2015. 26с. (Інформація та документація).

30. ДСТУ 8302:2015. Бібліографічне посилання. Загальні вимоги та правила складання Київ, ДП «УкрННЦ», 2016. 16 с. (Інформація та документація).

31. ДСТУ 3582:2013. Бібліографічний опис. Скорочення слів і словосполучень в українській мові. Загальні вимоги та правила. Київ, ДП «УкрННЦ», 2013. 23 с. (Інформація та документація).

# ДОДАТОК А

## Лістинг коду розробленого програмного забезпечення

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
from keras.layers import *
from keras.models  import Model from
keras.utils import plot_model from
keras.models import load_model
  #функція створення residual блоку мережі ResNet34
  def residual_block(layer_input, filters_count, is_first=False):

merge_input = layer_input
    if layer_input.shape[-1]!=filters_count: if
      is_first:
        merge_input = Conv2D(filters_count, (1,1),strides=(2,2),
                             activation='linear',kernel_initializer='he_normal')(layer_input)
 else:
        merge_input = Conv2D(filters_count, (1,1),strides=(1,1),
                            activation='linear',kernel_initializer='he_normal')(layer_input)
# batchnorm1 = BatchNormalization(axis=3,momentum=0.9,epsilon=0.0001)(laye r_input)
 # activation1 = Activation('relu')(layer_input)
if is_first:
      conv1 = Conv2D(filters_count, (1,1), strides=(2,2), padding='same',
                     activation='relu',kernel_initializer='he_normal')(layer_input)
else:
      conv1 = Conv2D(filters_count, (1,1), strides=(1,1), padding='same',
                     activation='relu',kernel_initializer='he_normal')(layer_input)
#batchnorm2  = BatchNormalization(axis=3)(conv1)
 conv2 = Conv2D(filters_count, (3,3), padding='same',
 activation='linear',kernel_initializer='he_normal')(conv1)
layer_out = add([conv2, merge_input])
activation2 = Activation('relu')(layer_out)
return activation2
```

```
  def create_ResNet():
    ResNet34_input = Input(shape=(224, 224, 3))
        conv1 = Conv2D(64, (7,7), strides=(2,2), padding='same', activation='relu',
                       kernel_initializer='he_normal')(ResNet34_input)
    pool1 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(conv1)
    residual_block64_1 = residual_block(pool1, 64) residual_block64_2 =
        residual_block(residual_block64_1,64) residual_block64_3 =
        residual_block(residual_block64_2,64)
    residual_block128_1 = residual_block(residual_block64_3,128, is_first=True)
        residual_block128_2 = residual_block(residual_block128_1,128 ) residual_block128_3
        = residual_block(residual_block128_2,128 ) residual_block128_4 =
        residual_block(residual_block128_3,128 )
    residual_block256_1 = residual_block(residual_block128_4,256, is_first=True)
        residual_block256_2 = residual_block(residual_block256_1,256) residual_block256_3 =
        residual_block(residual_block256_2,256) residual_block256_4 =
        residual_block(residual_block256_3,256) residual_block256_5 =
        residual_block(residual_block256_4,256) residual_block256_6 =
        residual_block(residual_block256_5,256)
    residual_block512_1 = residual_block(residual_block256_6,512, is_first=True )
        residual_block512_2 = residual_block(residual_block512_1,512) residual_block512_3 =
        residual_block(residual_block512_2,512)
    activation = Activation('relu')(residual_block512_3)
    pool2 = AveragePooling2D(pool_size=(7,7),strides=(1,1))(residual_block512_3)
    ResNet34 = Model(inputs=ResNet34_input, outputs=pool2)
return ResNet34
      #функція створення inception блоку мережі Inception
  def inception_block(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
        conv1 = Conv2D(f1, (1,1), padding='same', activation='relu',
                       kernel_initializer='he_normal')(layer_in)
        conv2 = Conv2D(f2_in, (1,1), padding='same', activation='relu',
                       kernel_initializer='he_normal')(layer_in)
        conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu',
                       kernel_initializer='he_normal')(conv2)
        conv4= Conv2D(f3_in, (1,1), padding='same', activation='relu',
                      kernel_initializer='he_normal')(layer_in)
        conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu',
                       kernel_initializer='he_normal')(conv4)
    pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in) conv6 =
        Conv2D(f4_out, (1,1), padding='same', activation='relu',
```

```
                kernel_initializer='he_normal')(pool)
  layer_out = concatenate([conv1, conv3, conv5, conv6], axis=-1) return
      layer_out
    #Функція ствоення мережі Inception
    def create_Inception():
  Inception_input = Input(shape=(224, 224, 3))
      conv1 = Conv2D(64, (7,7), strides=(2,2), padding='same', activation='relu',
                kernel_initializer='he_normal')(Inception_input)
  pool1 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(conv1)
      conv2 = Conv2D(192, (3,3), strides=(1,1), padding='same', activation='relu',
                kernel_initializer='he_normal')(pool1)
  pool2 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(conv2)
  inception_block1 = inception_block(pool2, 64, 96, 128, 16, 32, 32)
  inception_block2 = inception_block(inception_block1, 128, 128, 192, 32, 96, 64)
   pool3 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(inception_ block2)
  inception_block3 = inception_block(pool3, 192, 96, 208, 16, 48, 64)
  inception_block4 = inception_block(inception_block3, 160, 112, 224, 24, 64, 64)
  inception_block5= inception_block(inception_block4, 128, 128, 256, 24, 64, 64)
  inception_block6 = inception_block(inception_block5, 112, 144, 288, 32, 64, 64)
  inception_block7 = inception_block(inception_block6, 256, 160, 320, 32, 128, 128)
   pool4 = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(inception_ block7)
  inception_block8 = inception_block(pool4, 256, 160, 320, 32, 128, 128)
  inception_block9 = inception_block(inception_block8, 384, 192, 384, 48, 128, 128)
  pool5 = AveragePooling2D(pool_size=(7,7),strides=(1,1))(inception_block9)
  dropout = Dropout(0.4)(pool5)
  Inception = Model(inputs=Inception_input, outputs=dropout)
  return Inception
      # Функція для попередньої обробки даних
      def preprocess_input_data(x, y):
          x_p = x/255.0
          y_p = keras.utils.to_categorical(y, 10)
          return x_p, y_p
      # Завантаження датасету та його попередня обробка
  (x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
  x_train, y_train = preprocess_input_data(x_train, y_train) x_test,
      y_test = preprocess_input_data(x_test, y_test)
    # Задавання розміру вхідних  даних
    size = 224 #ширина та висота вхідного зображення
model = create_ResNet()
```

```
    # Налаштування архітектури моделі
train_model = keras.models.Sequential() train_model.add(keras.layers.Lambda(lambda image:
    tf.image.resize(image, (size,s ize))))
 train_model.add(model) train_model.add(Flatten())
    train_model.add(Dense(10, activation='softmax'))
 plot_model(model, show_shapes=True, to_file='model_arch1.png')
 train_model.build(input_shape=(None,224,224,3))
    # Налаштування моделі для навчання
    train_model.compile(loss='categorical_crossentropy',
                optimizer=keras.optimizers.Adam(),
                metrics=['accuracy'])
 train_model.summary()
 save_weights = tf.keras.callbacks.ModelCheckpoint('./drive/MyDrive/ResNet34/R
    esNet_weights.{epoch:02d}-{val_accuracy:.2f}',
                monitor='val_accuracy',
                mode='max',
                save_best_only=True)
    history = train_model.fit(x_train, y_train, batch_size=100, epochs=15,
                    validation_split=0.1, callbacks=[save_weights])
 train_model = load_model('/content/drive/MyDrive/Inception/Inception_weights.1 0-0.80')
    #Перевірка точності моделі на тестовій вибірці
 test_loss, test_acc = train_model.evaluate(x_test, y_test)
 print('Test loss: {}, Test accuracy: {}'.format(test_loss, test_acc * 100))
 train_model.summary()
    # Виведення графіку метрик мережі, які записувалися під час навчання
 plt.figure(figsize=(15, 4))
    plt.title('Train', fontsize=17)
.xlabel('epoch')
    plt.ylabel('value')
    plt.subplot(1,2,1)
plt.plot(history.history['loss'], label='loss')
    plt.plot(history.history['accuracy'], label='accuracy')
    plt.legend()
plt.grid(True)
plt.subplot(1,2,2) plt.plot(history.history['val_loss'],label='val_loss')
    plt.plot(history.history['val_accuracy'],label='val_accuracy')
    plt.legend()
 plt.grid(True) plt.show()
```