Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра математики факультету інформатики

Класифікація зображень з використанням нейронних мереж

Текстова частина до атестаційної роботи

за спеціальністю „Прикладна Математика” 113

Керівник атестаційної роботи,
ст.викладач Бучко О. А.

_____

(підпис)

“____”_____ 2023 р.

Виконав студент 4 р.н.
Кирпа М.Г.
“12” травня 2023 р.

м. Київ – 2023 рік

Календарний план виконання курсової роботи

Тема: Розробка нейронної мережі для розпізнавання графічних об’єктів

Календарний план виконання роботи:

№ п\п	Назва етапу дипломного проекту(роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на атестаційну роботу	Жовтень - листопад 2022р.	
2.	Огляд літератури за темою роботи	Листопад-лютий 2022р.	
3.	Аналіз теоритичної складової при створенні нейронних мереж	Лютий-березнь 2022р.	
4.	Аналіз практичного застосування нейронних мереж та огляд предметної області	Березень 2022р.	
5.	Написання текстової частини	Березень 2023р.	
6.	Перегляд змісту роботи керівником	Квітень 2023р.	
7.	Створення презентації	Травень 2023р.	
8.	Захист атестаційної роботи	Травень 2023р.	

Студент Кирпа М.Г. _____

Керівник Бучко О.А. _____

“_____”_____2023 р.

Зміст

ВСТУП

Сучасний світ генерує надзвичайно великі обсяги інформації за дуже короткі проміжки часу, тож все більше людство стикається з необхідністю швидко і якісно аналізувати нові дані. Однією з найбільш перспективних аналітичних технік є машинне навчання, особливо нейронні мережі. Вони можуть самостійно вивчати залежності в даних та робити прогнози на основі аналізу.

Нейронні мережі мають широкий спектр напрямків застосування у сьогоднішній сфері інформаційних технологій. Одним з найбільш популярних і важливих напрямків є обробка зображень. Класифікація зображень є одним з найбільш простих та широко використовуваних завдань у цій галузі. Мета такого типу завдань постає в відношенні зображення до певного класу або категорії, наприклад, класифікація тварин, визначення різних типів транспорту, розпізнавання рухів людини на картинці тощо.

У даній роботі буде представлено розв’язання задачі класифікації зображення за допомогою нейронних мереж. Визначено основні теоретичні і практичні підходи до обробки зображень, побудови та навчання нейронної мережі, оцінки результатів та порівняння різних алгоритмів зі своїми особливостями. Також буде розглянуто реалізацію нейронної мережі для класифікації зображень на мові програмування Python за допомогою бібліотек з відкритим кодом, таких як TensorFlow, OpenCV та Keras.

Застосування методів класифікації зображень має значний вплив на розвиток багатьох галузей, таких як промисловість, наука, медицина, охоронні системи, медицина тощо.

РОЗДІЛ 1: Огляд історії розвитку класифікації зображень та предметної області.

Близько 540 мільйонів років тому перші організми отримали здатність бачити, що спричинило величезний сплеск у еволюції. Буквально за 10 млн років з'явилася величезна кількість нових видів, що є пращурами багатьох видів тварин які населяють нашу планету сьогодні.

Зір людини є найбільшою сенсорною системою у організмі, майже 50% нервових клітин обробляють інформацію що надходить через неї. Це дозволяє нам виживати, працювати, пересуватися у просторі, взаємодіяти з сторонніми предметами та багато чого іншого.

Перший механічний зір, який зробила людина з'явився приблизно у 16 ст. - камера Обскура, її принцип описує примітивний зір який формувався у наших пращурів і представляє собою маленьку коробку з діркою на одній з стінок, крізь яку проходить світло і проектує об'єкт у перевернутому вигляді на сусідню стіну яка зроблена матового скла або ж тонкого паперу. Цей винахід дав поштовх до створення камер, які ми використовуємо у повсякденному житті.

Вже у 20 сторіччі, а саме в 1959 році науковці Девід Г'юбел і Торстен Вейзел поставили собі мету: розібратись як людський мозок обробляє візуальну інформацію на нейронному рівні. Так як мозок котів дуже схожий з точки зору сприйняття зорових образів на людський, вони провели експеримент, за допомогою електрофізіологічних приладів було відслідковано реакцію окремих нейронів у мозку кота на картинки що були представлені. В результаті вчені з'ясували що мозок ссавців розглядає зображення як сукупність надзвичайно великої кількості орієнтованих ліній і на основі цього вже добудовує інші більш складні образи.

Однією з найвідоміших, і ймовірно першою, науковою роботою яка стосуються комп'ютерного зору є "Block World" представлена профессором Массачусетський технологічного інституту Ларрі Робертсом у 1963 році. Її ідея полягає в представленні

об’єктів на зображенні у вигляді простих геометричних фігур і з’ясуванні як саме можна розбивати об’єкти за цією технікою. Саме вона дала поштовх у розвитку в галузі комп’ютерного зору в такому вигляді, як ми можемо побачити зараз.

Першою практичною роботою прийнято вважати літнє завдання що дав студентам профессор MIT Сеймур Пайперт у 1966 році, постановка задачі виглядала наступним чином: попрацювати над візуальною системою, яка могла б розділяти зображення на «ймовірні об’єкти, ймовірні фонові області та хаос». Саме це завдання й досі намагаються вирішити всі дослідники даної області по всьому світу.

У 1980-х роках доктор Куніхіко Фукусіма розробив штучну нейронну мережу, натхненну роботами Губеля та Вайзеля та імітуючу функціонування простих та складних клітин. Ця мережа використовувала S-клітини для імітації простих клітин та C-клітини для імітації складних клітин. Хоча ці клітини були штучними, вони імітували алгоритмічну структуру біологічних клітин. Основна ідея Неокогнітрона(так він назвав свою розробку) Фукусіми полягала в тому, щоб фіксувати складні патерни, такі як собака, за допомогою складних клітин, які отримують інформацію від інших складних клітин на нижчому рівні, або від простих клітин, які виявляють простіші патерни.

Вже у 1990-х, французький вчений інформатик Ян ЛеКун запропонував свою реалізацію першої сучасної згорткової нейронної мережі у своїй роботі “Gradient-Based Learning Applied to Document Recognition”. Там він описав її роботу на основі одного з найвідоміших датасетів MNIST, що складається з фотографій 70 тисяч написаних вручну цифр від 0 до 9. Логіка моделі наслідувала ідеї Куніхіко Фукусіми, процес тренування складався з декількох ітеративних кроків, а саме:

- Моделі надається приклад зображення
- Модель намагається передбачити клас до якого належить картинка
- Ваги моделі оновлюються спираючись на результат порівняння передбаченого класу і фактичного
- Кроки 1-3 повторюються доки не буде мінімізовано функцію втрат

Спосіб тренування моделі запропонований ЛеКуном є стандартом у сучасному тренуванні нейронних мереж і в загальному вигляді залишається незмінним.

Починаючи з 90-х років 20 століття щорічно галузь поповнювалась новими розробками, датасетами, архітектурами нейронних мереж. З 2010 року було створено проект ImageNet з мільйонами зображень і тисячами класів що поповнюються щорічно, більшість нових архітектур нейронних мереж змагаються у досягненні найкращого результату у прогнозуванні на основі цього датасету. Також необхідно згадати про мережу з архітектурою AlexNet, що у 2012 році перевищила рекордний результат аж на 10% в порівнянні з тодішніми конкурентами, її особливість полягає в тому що НМ використовує графічну пам'ять для тренування, що значно пришвидшує процес і покращує точність. Більшість архітектур нейронних мереж, що працюють з зображеннями, використовують принципи запропоновані ЛеКуном у поєднанні з архітектурними рішеннями AlexNet, але мають значно більшу кількість шарів і гіпер параметрів.

Наразі нейронні мережі є невід'ємною частиною багатьох галузей, найпрогресивніші корпорації світу розробляють інформаційні продукти з використанням цієї технологій. При огляді предметної області можна навести наступні приклади застосування моделей:

- Починаючи з 2014 року в автомобілі компанії Tesla почали встановлювати системи автопілоту що спрощують водіям процес пересування. За допомогою камер, що встановлені вздовж корпусу автомобіля, інформація про стан зовнішніх перешкод передається у керувальну систему в реальному часі, програмне забезпечення обробляє надані дані за допомогою нейронних мереж і допоміжних алгоритмів. На основі обробленої інформації керувальний модуль авто приймає рішення про корекцію швидкості і траєкторії транспортного засобу. Звичайно ці системи й досі залишаються не досконалими і потребують фізичної присутності водія, проте, у далеких поїздках це значно зменшує навантаження на людину.

- У 2020 році в Масачусетському Технічному Університеті, група науковців розробила платформу яка з надзвичайною точністю може малювати картини за заданими параметрами наприклад стиль конкретного художника, також модель здатна покроково відтворювати процес написання художнього виробу. Під час навчання моделі було використано декілька сотень відео з поетапним малюванням елементів картини, таким чином дослідникам вдалось досягти приблизно 90% точності в порівнянні з реальним процесом написання художником. Також платформа має здатність ілюструвати загальні прийоми і може бути використана для навчання недосвідчених художників.
- Також нейронні мережі часто застосовуються у медицині для діагностування складних захворювань, на противагу складних систем що розроблялися переважно у кінці 20 сторіччя, нейронні мережі навчаються на реальних прикладах хвороб, що дозволяє їм виокремлювати залежності непідвладні людському оку, або великій кількості складних правил. Найвідомішою системою подібного типу є система кардіодіагностики від RES Informatica та Центру кардіологічних досліджень в Мілані. Ця нейромережа аналізує тахограми серцебиття людини і виокремлює нестандартні відхилення від норми, за наявності таких, на основі яких класифікує конкретне захворювання пацієнта.

Тож предметна область нейронних мереж наразі охоплює величезну кількість галузей критичних для існування і розвитку людства, у наступних розділах роботи буде докладно оглянуто теоретичну складову згорткових нейронних мереж, що переважно працюють з зображеннями, та порівняно декілька підходів до класифікації зображень зібраних з вільних джерел у мережі інтернет.

РОЗДІЛ 2: Архітектура згорткової нейронної мережі з математичної точки зору.

Згорткова нейронна мережа (англійською Convolutional Neural Network), надалі будемо називати її як ЗНМ - це тип нейронної мережі який є одним з найпопулярніших для вирішення проблем пов'язаних з обробкою відео та зображень. Основна ідея ЗНМ при роботі з картинками - це виявлення певних ознак таких як кути, лінії, форми та ін. на зображенні за допомогою фільтрів і математичних операцій, що представлені шарами ЗНМ.

Основна складова одиниця картинки - це піксель, тобто числова інтерпретація кольору у кольоровій гамі, тобто якщо на вході подається картинка у чорно-білому спектрі то кожен піксель на картинці це число від 0 до 255, що означає близькість до одного з полюсів. І з іншого боку кількість пікселів залежить від висоти та ширини зображення, тож будь-яку картинку можна представити у вигляді числової матриці, з розмірністю ширина × висота × кількість кольорів у гамі, і саме це дозволяє перетворювати зображення у ЗНМ для виявлення особливостей певних класів.

Класична архітектура ЗНМ складається з деякої сукупності різноманітних шарів з унікальними функціями і особливостями. Загалом вони поділяються на:

- Згортковий шар(англійською Convolutional Layer) - є основою нейронної мережі, за допомогою цих шарів на зображення накладаються фільтри невеликого розміру що виявляють різні ознаки, або ж прості геометричні фігури необхідні для подальшого передбачення.
- Шар активації(англійською Activation Layer) - зазвичай слідує одразу після згорткового шару і представляє собою нелінійну обробку даних отриманих з попереднього перетворення, зазвичай його використовують для покращення рівня точності передбачень, зменшення кількості параметрів що зберігає нейронна мережа і уникнення проблем з перенавчанням. Перенавчання може виникнути, коли мережа пам'ятає

надто багато даних з тренувального набору, що може призвести до поганої точності передбачень на нових даних.

- Пулінг шар(англійською Pooling Layer) - відповідає за фільтрацію не інформативних, виявлених раніше ознак за допомогою зменшення розмірності зображення, дозволяє відкинути менш важливі з них і таким чином збільшити ефективність нейронної мережі.
- Повнозв’язний шар(анлійською Fully connected Layer) - обов’язково слідують у кінці архітектури нейронної мережі(не можуть використовуватись перед згорковими або пулінг шарами) і відповідають за безпосередньо класифікацію зображення до певної групи на основі виявлених у попередніх шарах ознак. В них містяться ваги які у процесі навчання мережі оптимізуються для досягнення найвищої точності прогнозування.

У цьому розділі буде розглянуто кожен тип шару ЗНМ, його різновиди, особливості і теоретичні властивості.

2.1 Згортковий шар

Як вже було згадано вище згортковий шар ЗНМ є основним блоком у роботі з зображеннями. Його основна особливість полягає в тому що він складається з скінченного набору фільтрів або ж ядер. Кожен фільтр має висоту і ширину, та випадково ініціалізовану вагу що в процесі навчання оптимізується і набуває обгрунтованого вигляду, також ядро розповсюджується на всю глибину(кількість кольорів у структурі зображення), картинки. Зазвичай фільтри мають квадратну форму, тобто їх висота і ширина співпадають. Метою застосування кожного ядра є виокремлення певних закономірностей на зображенні у процесі навчання.

Результатом застосування фільтрів до початкового зображення буде мапа активізації що складається з матриць згорток, тобто результатів накладання окремих ядер на картинку.

Окрема матриця згортки - це результат накладання конкретного ядра на картинку, а також - це фізична інтерпретація шуканої закономірності на зображенні. Вона представлена за допомогою матриці розмірністю $\frac{(K-L)}{S} + 1$, де K - висота або ширина початкового зображення, L - ширина або висота ядра і S - це крок фільтру який задано для згорткового шару. У випадку коли висота і ширина фільтру чи зображення відрізняються - висота та ширина матриці згортки розраховуються окремо за тією ж формулою.

Елементи цієї матриці розраховуються за допомогою формули:

$$y_{ij} = \sum_{k=0}^{m-1} \sum_{l=0}^{n-1} x_{i \cdot s+k,\, j \cdot s+l} \times w_{k,l},$$ де:

- y_{ij} - значення елемента вихідної матриці згортки з координатами (i, j);
- $x_{i \cdot s+k,\, j \cdot s+l}$ - значення елемента вхідної матриці згортки з координатами $(i \cdot s + k,\, j \cdot s + l)$;
- $w_{k,l}$ - значення відповідного елемента фільтру згортки з координатами (k, l);
- m- розмір фільтру згортки по висоті;
- n - розмір фільтру згортки по ширині;
- s - значення зсуву(stride)

З практичної точки зору це виглядає наступним чином:

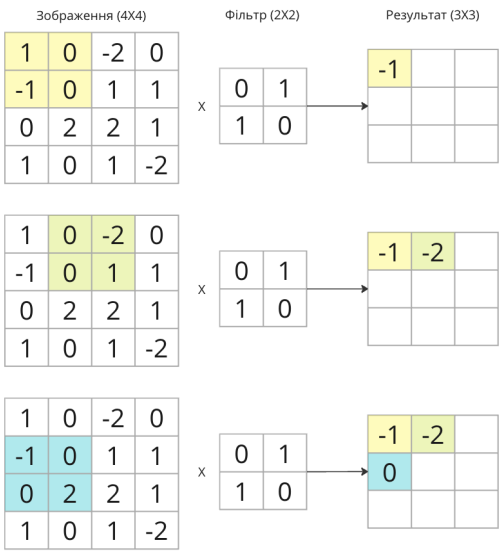

Рис.1 Застосування фільтру згортки до зображення на практиці.

На рисунку номер один представлено зображення 4 × 4 пікселі і ядро 2 × 2 з випадково ініціалізованими вагами і кроком 1. Тож за формулою розмірність матриці дорівнює 3, а перші три елементи: -1, -2, 1 відповідно.

Також не менш важливим є параметр крок(англійською stride), який вже зустрічався у формулах вище, він означає з яким кроком від попереднього лівого верхнього елементу на оригінальне зображення буде накладатись фільтр на прикладі вище крок дорівнює 1, але якщо наприклад крок буде 2, то згорткова матриця буде розмірністю 2 × 2 і перші 2 елементи будуть -1, 1 відповідно. Чим вище крок тим менше буде розмірність згорткової матриці і перекриття зображення відповідно, але швидкість навчання буде підвищено.

Додаткова техніка що може бути застосована для підвищення точності згорткової матриці - це відступ або ж доповнення(англійською padding). Дуже часто зустрічаються зображення які містять важливі елементи по краям, ця інформації може бути додана у згорткову матрицю лише через зсування фільтру, але цього часто може бути недостатньо щоб ознака вважалася важливою на етапі тренування. Тож для протидії подібним проблемам можна додавати на зображення умовну “рамку” з нульовими значеннями щоб розширити область охоплення ядер і не втрачати ймовірних важливих елементів.

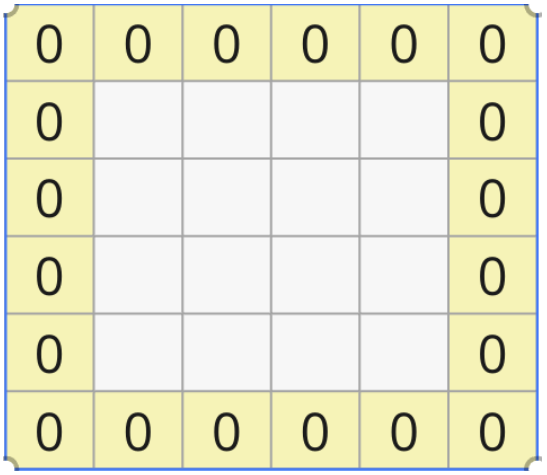

Рис. 2 Приклад доповнення нульової інформації до вхідного зображення.

Кількість ядер у згортковому шарі задається як гіпер параметр при проектуванні ЗНМ. У високорівневих бібліотеках на мові Python таких як Tensorflow або PyTorch цей параметр дуже просто конфігурувати. Зазвичай він задається на основі емпіричних методів за допомогою експериментального підходу до валідації результатів на тестовий даних, і дуже залежить від конкретної задачі. У класичних ЗНМ кількість фільтрів зростає з кожним наступним згортковим шаром у моделі, що дозволяє отримати більш складну та потужну архітектуру мережі.

У підсумку, після застосування згорткового шару до вхідних даних утворюється матриця, яку можна використовувати у наступних шарах, з розмірністю $O = N \times M \times K$, де N - висота згорткової матриці, M - ширина згорткової матриці, K - кількість фільтрів запрограмованих у шарі.

2.2 Шар активації

Важливою складовою архітектури ЗНМ є шари активації або ж шари нелінійної активації. Такі шари не містять у собі вагів, які можна оптимізувати під час навчання моделі, тож їх часто можна не побачити на схемах що візуалізують нейронну мережу. Проте вони є важливою складовою структури і як можна зрозуміти з назви, забезпечують ЗНМ вивчати нелінійні залежності, що в свою чергу впливає на точність прогнозування.

Шари активації у ЗНМ прийнято використовувати після всіх шарів що містять в собі ваги для того щоб позбутися лінійної структури присутньої у попередньому шарі(оскільки згорткова операція є лінійною). Це перетворення відбувається за допомогою застосування диференційованої математичної функції до вхідних даних. Функція обов'язково має бути диференційованою для можливості оптимізації вагів ЗНМ при навчанні.

У різноманітних архітектурах зустрічається велика кількість функцій активації зі своїми перевагами та недоліками, але можна виділити 4 основні та найпопулярніші з них: ReLU, Sigmoid, Tanh та Softmax.

2.2.1 ReLU

Функція активації ReLU була вперше застосована у ЗНМ в 2011 році і виявилася значно практичнішою за запозичену з теорії імовірності логістичну(sigmoid) та гіперболічний тангенс(tanh), що широко використовувались у моделях до цього. Вона є найбільш поширеною та ефективною функцією активації у нейронних мережах, оскільки її простота та швидкість обчислень дозволяють моделі працювати швидше та ефективніше.

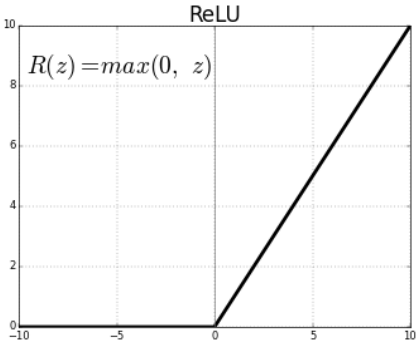

Рис. 3 Принцип роботи функції активації ReLU на графіку

Принцип роботи цієї функції активації, як показано на графіку, полягає в тому що вона відображає вхідний набір даних X у вихідний набір даних Y за формулою $R(z) = max(0, z)$, іншими словами, якщо значення вхідного сигналу менше або дорівнює нулю, то ReLU повертає нуль. Якщо ж значення вхідного сигналу більше

нуля, то ReLU повертає саме це значення. Таке перетворення ще називають нелінійним відсіюванням, що дозволяє моделі бути більш стійкою до шуму та викидів у вхідних даних.

2.2.2 Логістична функція активації(sigmoid)

Також часто у архітектурах ЗНМ часто можна зустріти логістичну функцію активації, її особливість в тому що вона є гладкою, несиметричною “S” подібною кривою з діапазоном значень вихідних даних від 0 до 1. Формула:

$f(z) = \frac{1}{1-e^{-z}}$, де z - це значення вхідних даних

Логістична функція активації виконує функцію активації, що перетворює зважену суму входів нейрону у вихідний сигнал, який передається наступному шару нейронів. Простими словами вона дозволяє “вмикати” і “вимикати” нейрони, коли значення наближається до мінус нескінченності, значення функції прямує до нуля тож цей нейрон не буде врахований у наступному шарі, коли ж значення прямує до нескінченності $\Rightarrow f(z) \to 1$, тож нейрон буде мати більше ваги у наступному шарі.

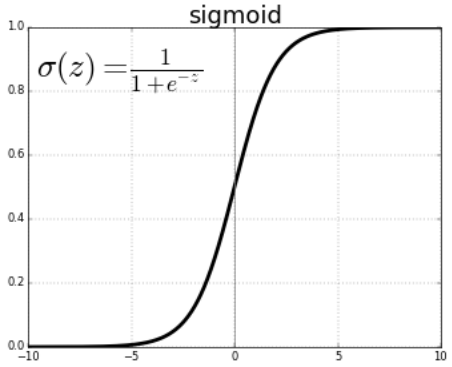

Рис. 4 Принцип роботи функції активації sigmoid на графіку

2.2.3 Гіперболічний тангенс(tanh)

Гіперболічний тангенс як функція активації використовується зазвичай у випадках коли є ймовірність “перенавчання” моделі, він дуже корисний через те що діапазон вихідних даних після застосування функції від -1 до 1, тож у середньому дані будуть нормалізовані біля 0, формула виглядає наступним чином:

$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$, де z - вхідні дані

Великим недоліком tanh є велика чутливість до викидів в даних, тобто у випадках коли значення даних завелике результат може виходити за межі [-1, 1], що призведе до проблем з ініціалізацією ваг у наступному шарі.

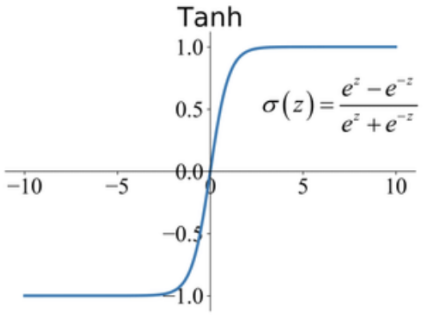

Рис. 5 Принцип роботи функції активації tanh на графіку

2.2.4 Нормована експоненційна функція(softmax)

Функція активації softmax зазвичай застосовується після останнього повнозв'язного шару, для того щоб перетворити вхідні дані з вектору значень на вектор ймовірностей, які в свою чергу означають належність зображення до відповідного класу. Застосувати активацію можна за формулою:

$$f(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- де z_i - і-й елемент вектора
- K - кількість всіх елементів вектора вхідних даних

Нормована експоненційна функція, за допомогою експоненти, перетворює вхідні значення в додатні числа та нормалізує їх, щоб сума дорівнювала 1. Вихідний вектор - це вектор ймовірностей розподілений належності картинки до певного класу.

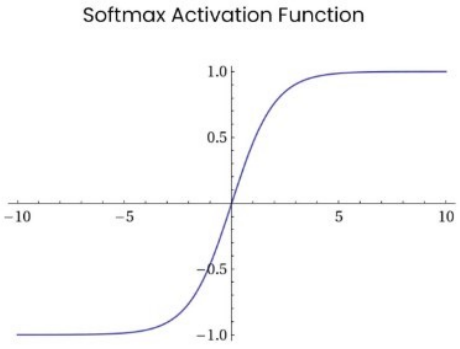

Рис. 6 Принцип роботи функції активації Softmax на графіку

2.3 Пулінг шар

Шар пулінгу також є однією з ключових компонент структури ЗНМ, його призначення полягає у зменшенні розмірності і виокремленні основних ознак з вхідних даних. Цей шар отримує на вхід область входу розміром N×N(отриману в результаті перетворень у попередньому шарі) і застосовує до неї пулінг функцію. На виході ми отримуємо дані меншої розмірності з більш значущими ознаками що призводить до зменшення кількості гіпер параметрів у мережі і більшій стабільності до перенавчання. У різних архітектурах можна зустріти велику кількість функцій пулінгу, але найпопулярнішими та найбільш ефективними вважаються: функція максимального пулінгу(англ. max pooling) та функція середньозваженого пулінгу(англ. avg pooling).

2.3.1 Функція максимального пулінгу

Серед багатьох технік максимальний пулінг є найбільш розповсюдженим у архітектурах ЗНМ, через те, що якщо його застосовувати до вхідного набору, вихідний сигнал буде містити тільки найбільш значущі ознаки. Принцип роботи полягає в тому що на деяку область накладається пулінг фільтр(з заданими висотою та шириною) і з цієї області вибирається найбільший елемент(максимальний), що є елементом вихідного шару.

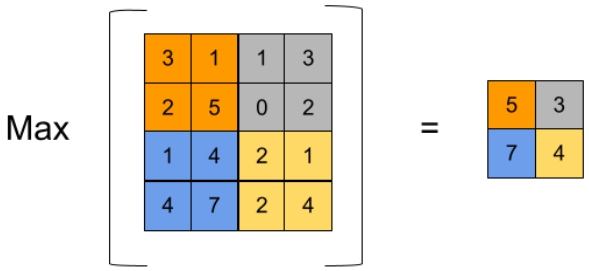

Рис. 7 Принцип роботи функції max pooling

Для кожної області вхідної матриці, яка відповідає пулінг фільтру, максимальний елемент знаходиться за допомогою формули:

$$h_{x,y} = max_{i=0,..,K,\, j=0,...,K} l_{(x+i),\,(y+j)},$$

- де $h_{x,y}$ - елемент вихідної матриці пулінгу, який відповідає області пулінгу (x, y) вхідної матриці
- $l_{(x+i),\,(y+j)}$ - елемент вхідної матриці, який знаходиться в $(x + i)$-ому рядку та $(y + j)$ - ому стовпці
- K - розмір пулінг фільтру.

Найчастіше області фільтру пулінгу дорівнює 2×2, це є оптимальним значенням для збереження всіх ключових ознак у вхідних даних і одночасно зменшення їх розмірності, але зустрічаються і більші розміри фільтрів. Також можна використовувати параметр зсуву(stride) аналогічний до зсуву в згортковому шарі, це дозволить не пропустити важливих ознак, але уповільнить процес тренування моделі.

2.3.2 Функція середнього пулінгу

Функція середнього пулінгу має аналогічну структуру і логіку до максимальної функції, але через свої властивості середнього значення дозволяє зберігати більше контекстної інформації на зображенні на відміну від попереднього методу.

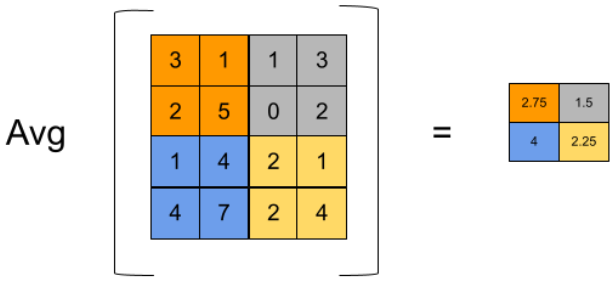

Рис. 8 Принцип роботи функції average pooling

Для кожної області вхідної матриці, яка відповідає пулінг фільтру, середній елемент знаходиться за допомогою формули:

$$h_{x,y} = \frac{1}{K^2} \sum_{i=0, j=0}^{K, K} l_{(x+i), (y+j)}$$

- де $h_{x,y}$ - елемент вихідної матриці пулінгу, який відповідає області пулінгу (x, y) вхідної матриці
- $l_{(x+i), (y+j)}$ - елемент вхідної матриці, який знаходиться в $(x + i)$-ому рядку та $(y + j)$ - ому стовпці
- K - розмір пулінг фільтру.

2.4 Повнозв'язний шар

Останній шар, перед нормованою експоненційною функцією активації, у архітектувах ЗНМ - це повнозв'язний шар(нерідко можна зустріти комбінацію з декількох таких шарів у мережі), його основна функція полягає у остаточній обробці ознак зібраних на попередніх шарах моделі. Кожен нейрон повнозв'язного шару безпосередньо пов'язаний з кожним нейроном з попереднього шару, і на основі цих даних формує вектор ознак що може бути використаний для класифікації зображення.

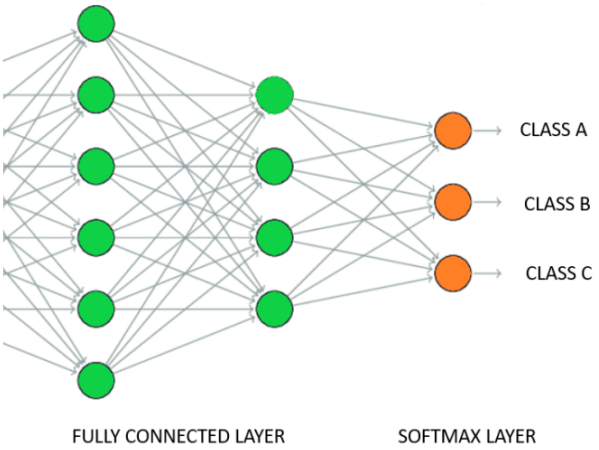

Рис. 9 Принцип роботи повнозв'язного шару

Кожен нейрон повнозв'язного шару виконує лінійну комбінацію вхідних ознак з вагами і додає зсув (англ. bias) для створення нового значення, яке потім може бути передано до функції активації. Тобто з математичної точки зору шар приймає вектор X-вихідні дані з минулого шару, домножає до них вектор вагів W, та додає зсув(шум) b, тобто результатом застосування шару буде Y, що розраховується за формулою:

$$Y = F(WX + b), \text{ де } F \text{ - функція активації}$$

Отже, повнозв'язний шар допомагає згортковій нейронній мережі робити заключні висновки про те, до якого класу чи категорії належать вхідні дані.

2.5 Регуляризація ЗНМ

При тренуванні ЗНМ на новому наборі даних існує дуже розповсюджена проблема - перенавчання(англ. over-fitting), простими словами це процес при якому на тренувальній вибірці модель показує високу точність, проте при прогнозуванні на незнайомих даних(тестовій вибірці) результати моделі значно відрізняються у якості. З архітектурної точки зору є декілька методик які обмежують нейронну мережу і запобігають розповсюдженню даного процесу під час оптимізації вагів, ці методики прийнято називати регуляризація.

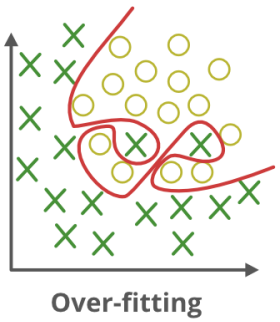

Рис. 10 Приклад перенавчання на тренувальних даних

Найрозповсюдженішими методами прийнято вважати:

- L1 регуляризація - додає до функції втрат моделі суму абсолютних значень параметрів помножену на заздалегідь визначений коефіцієнт лямбда. З математичної точки зору функція втрат(середньоквадратичного відхилення) з L1 регуляризацією буде мати наступний вигляд:

$$\theta = \frac{1}{n}\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}|\beta_j|$$

Де θ - функція втрат, n - кількість спостережень у наборі даних, y_i - спрогнозоване значення, p - кількість параметрів у рядку що дорівнює значенню i , x_{ij} - значення окремого параметру у вхідних даних, β_j - ваги що відповідають параметру x_{ij}, λ - гіпер параметр регуляризації. Чим ближчий гіпер параметр λ до 0 тим менше впливу регуляризація на ваги моделі і тим самим тим більша ймовірність перенавчання, і навпаки якщо ж λ дорівнює 1 всі ваги набувають приблизно однакових значень і модель не зможе виокремити необхідних ознак з зображення. Цей метод створює велику кількість нульових значень у векторі ознак, що призводить до виділення найбільш важливих з них і одночасно зменшення кількості гіпер параметрів моделі.

- L2 регуляризація - додає до функції втрат моделі суму квадратів параметрів параметрів помножену на заздалегідь визначений коефіцієнт лямбда. З математичної точки зору функція втрат(середньоквадратичного відхилення) з L2 регуляризацією буде мати наступний вигляд:

$$\theta = \frac{1}{n}\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}\beta^2{}_j$$

Де θ - функція втрат, n - кількість спостережень у наборі даних, y_i - спрогнозоване значення, p - кількість параметрів у рядку що дорівнює значенню i , x_{ij} - значення окремого параметру у вхідних даних, β_j - ваги що відповідають параметру x_{ij}, λ - гіпер параметр регуляризації. На відміну від L1 регуляризації цей метод не обнуляє більшість вагів, але наближає їх до мінімальних значень що дозволяє залишити в моделі всі, навіть найменш значущі, ознаки.

- Також нерідко можна зустріти поєднання L1 і L2 регуляризацій - це називається еластичною мережею, вона поєднує обидві регуляризації і додає ще один гіпер параметр α - що відповідає за розподіл ваги регуляризації між L1 і L2 методами. Якщо альфа дорівнює 0.6 - це означає що L1 × 0.6 і L2 × 0.4.
- Відкидання нейронів(англ. dropout) - принцип роботи цієї методики полягає в тому, що на деяких шарах ЗНМ випадково вибирається деяка заздалегідь задана гіпер параметром кількість ознак(нейронів), які викидиються з ЗНМ(їх ваги обнуляются) і у наступному шарі модель має відтворити ці ознаки за допомогою наявних даних, що в свою чергу дозволяє позбутися залежності між вагами в шарах ЗНМ та згенерувати велику кількість незалежних між собою ознак, що покращують точність прогнозування.

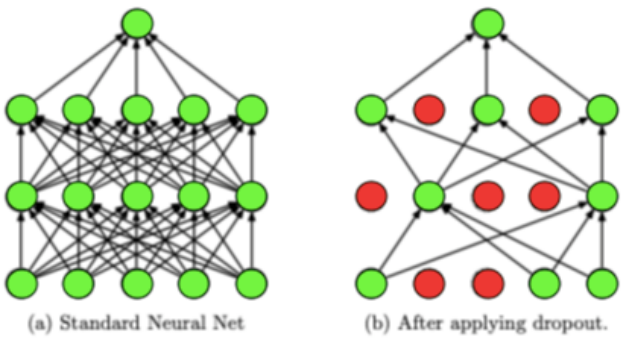

Рис. 11 Приклад архітектури ЗНМ з застосуванням регуляризації Dropout

Під час використання техніки відкидання нейронів модель стає більш точною у прогнозуванні, проте ця методика значно збільшує швидкість оптимізації.

В більшості сучасних ЗНМ вище перелічені техніки комбінують для досягнення найбільшої точності прогнозування. Також, окрім архітектурних рішень, для протидії перенавчанню можна використовувати техніку аугментації тренувальної вибірки даних.

Аугментація зображень навчальної вибірки - це процес створення нових прикладів за допомогою застосування трансформацій до наявних картинок, з метою розширення навчального датасету та зменшення чутливості моделі до перенавчання. Зазвичай такими трансформаціями є зміна розміру, повороту, зсуву, яскравості або насиченості зображення, або застосування інших перетворень, які не змінюють суті зображення. Через те що після такого процесу тренувальна вибірка насичена різноманітними прикладами, ймовірність зустріти небачені раніше ознаки у тестовій вибірці зменшується, тож і проблема з перенавчанням нівелюється.

РОЗДІЛ 3: Тренування(оптимізація вагів) ЗНМ

У розділі вище було детально оглянуто особливості архітектури ЗНМ та її складових частин, неодноразово згадано про ваги у шарах ЗНМ та їх калібровку для отримання точних результатів класифікації. Саме цей процес буде оглянуто в цьому розділі.

Тож, оптимізація вагів ЗНМ є ключовою складовою у створенні якісної моделі прогнозування. Основна мета полягає у пошуку таких значень вагів у шарах ЗНМ, які будуть давати найбільш точні та стабільні передбачення на реальних даних. З теоретичної точки зору є декілька методів калібровки нейронних мереж, проте на практиці найпоширеніший та найвдаліший підхід - це метод зворотного розповсюдження помилки(англ. backpropagation). Цей процес є ітеративним і включає в себе наступні кроки:

1. Пряме розповсюдження(англ. forward propagation) - на цьому етапі на вхід до ЗНМ подаються тренувальні зображення, кожне з них проходить через шари моделі, формуються певні ознаки та ініціалізуються ваги за допомогою яких модель робить прогноз. В результаті модель видає вектори значень, що позначають ймовірності належності до кожного класу для кожного екземпляру
2. Обчислення функції втрат(англ. loss function) - на основі передбачень моделі та реальних значень класів кожного члена вибірки розраховується значення функції втрат, що відображає числову різницю між передбачуваним та правильним результатом.
3. Зворотне розповсюдження(англ. backpropagation) - з метою покращення передбачення, за допомогою застосування похідних, помилка розповсюджується крізь кожен шар у зворотному напрямку, шари отримують інформацію про помилку наступного шару і коригують ваги згідно з отриманими даними.

4. Оновлення вагів - після зворотного розповсюдження ваги в кожному шарі оновлюються відповідно до визначеної функції оптимізації моделі(наприклад, методом градієнтного спуску), задля мінімізації значення функції втрат.

Цей процес повторюється для кожного тренувального зразка, доки не буде вирішено що мережа достатньо точна на тестових даних, або не будуть вичерпані всі можливості для покращення якості прогнозування.

3.1 Пряме розповсюдження(forward propagation)

Пряме розповсюдження або прямий прохід є фундаментальним кроком під час тренування нейронних мереж, це процес передбачає що на вхід модель отримує дані, які проходять крізь кожен шар нейромережі, обробляються за допомогою ваг та активаційних функцій і в результаті формується вектор ймовірностей належності до класів.

Прямий прохід може бути досить простим і складним процесом, залежно від кількості шарів, фільтрів у шарах, активаційних та регуляризаційних функцій, і також необхідно зазначити що всі вище перераховані математичні операції обов’язково мають бути диференційованими для можливості застосування зворотного розповсюдження особливості роботи якого оглянуті нижче.

Якщо розглянути один крок прямого проходу з одного шару до іншого, математично цей процес можна описати наступним чином:

$$a_i^l = f(z_i^l) = f(\sum_j w_{ij}^l \cdot a_j^{l-1} + b_i^l)$$

Де:

- a_i^l - значення нейрону з індексом i, шару з індексом l після застосування функції активації
- f - функція активації що слідує після шару з індексом l у нейромережі
- z_i^l - значення нейрону з індексом i, шару з індексом l
- w_{ij}^l - зв'язуюча вага між нейроном індексом j, шару з індексом l - 1 та нейроном індексом j, шару з індексом l
- b_i^l - випадкова похибка нейрону з індексом i, шару з індексом l

Ця формула є генералізованою і може відрізнятися в залежності від застосовуваного шару, але в будь-якому випадку в результаті для кожного тренувального зображення генерується вектор який надає ймовірність належності картинки до кожного з запрограмованих у модель класів, і за допомогою функції втрат можна визначити наскільки точно були підібрані і оптимізовані ваги w для кожного нейрону.

3.2 Функції втрат(loss function)

Основна мета функції втрат визначити якість передбачення нейронної мережі на основі різниці між очікуваним значенням і прогнозованим, це дозволяє як оцінювати саму модель на тестових даних, так і оптимізувати ваги моделі задля досягнення вищої точності.

Для різних задач машинного навчання існує велика кількість функцій втрат зі своїми перевагами та недоліками, проте не всі з них можна використовувати для задач класифікації зображень, тож найпопулярніші функції що використовують у роботі з класифікацією картинок представлені нижче.

3.2.1 Функція завісних втрат(Hinge loss)

Початково функція завісних втрато була часто вживаною в задачах з моделями бінарних класифікацій на основі опорних векторів, але у задачах пов’язаних з бінарною класифікацією з використанням ЗНМ цей метод теж можна зустріти. Він не є універсальним через те що для використання у мультикласових моделях, в загальному випадку і без додаткових перетворень, цю функцію застосувати не можливо. Проте він значно швидший за інші функції втрат, тому в окремих випадках розробники надають йому значну перевагу.

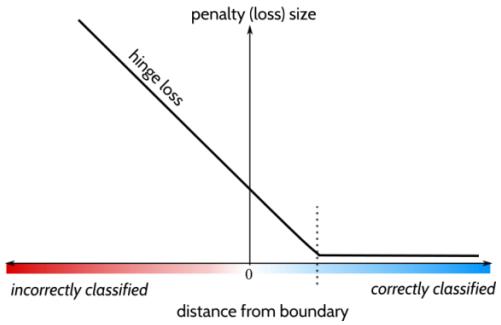

Рис. 12 Приклад розподілу значень функції завісних втрат

З математичної точки зору функція завісних втрат представлена формулою:

$$H(p,\ y)\ =\ \sum_{i\,=\,1}^{N} max(0,\ 1\ -\ (2\ \cdot y_{i}\ -\ 1)\ \cdot\ p_{i})$$

Де:

- $H(p,\ y)$ - значення функції втрат
- p - вектор прогнозованих значень для одного зображення
- y - вектор прогнозованих значень для одного зображення(буде містити 1 для індексу реального класу, і N - 1 нулів для інших класів)
- N - кількість класів зображень
- y_{i} - реальне значення класу i
- p_{i} - ймовірність належності до класу i

Результатом функції буде 0, якщо зразок класифіковано правильно, або ж значення що пропорційне відстані від правильної відповіді до ймовірності у двовимірному просторі, якщо екземпляр класифіковано не вірно.

3.2.2 Крос-ентропія

Крос-ентропія - це функція, яка є ймовірно найчастіше вживаною у роботі з представленим типом задач, її також можна зустріти під назвою логарифмічна функція втрат, особливість крос-ентропії полягає в тому, що на відміну від представленої вище функції завісних втрат, вона може одночасно включати в себе всі класи представлені в моделі і її результат залежить від ймовірностей кожного з них. Результатом логарифмічної функція втрат є ймовірність у межах від 0 до 1(чим ближче до 1, тим краще пронозує модель). Зазвичай ця функція використовується у тандемі з функцією активації Softmax на останньому шарі моделі. Математично крос-ентропія описана формулою:

$$H(p,\ y) \ = \ \ - \sum_{i=1}^{N} y_i \cdot \ log(p_i)$$

Де:

- $H(p,\ y)$ - значення функції втрат
- p - вектор прогнозованих значень для одного зображення
- y - вектор прогнозованих значень для одного зображення(буде містити 1 для індексу реального класу, і N - 1 нулів для інших класів)
- N - кількість класів зображень
- y_i - реальне значення класу i
- p_i - ймовірність належності до класу i

Функція крос-ентропії найчастіше використовується у випадках коли в задачі наявні 3 і більше класів, через те що функція здатна враховувати всі ймовірності з вектору прогнозування.

3.3 Зворотне розповсюдження(backpropagation)

Процес зворотного поширення є фундаментальним при оптимізації вагів ЗНМ, з теоретичної точки зору він полягає у обчисленні похідних функції втрат по вагам нейронної мережі. В основі його лежить диференціювання складних функцій за допомогою ланцюгового правила. Простими словами спочатку обчислюється похідна від функції втрат за відношенням до вихідних значень мережі, далі, ця помилка поширюється назад через мережу за допомогою ланцюгового правила диференціювання, знаходячи помилки на кожному шарі мережі, поки процес не дійде до першого шару ЗНМ. Таким чином формуються градієнти та зсуви які, за допомогою алгоритмів оптимізації, змінюють ваги мережі і покращують її точність.

3.3.1 Ланцюгове правило диференціювання в загальному вигляді

У загальних випадках, в математичних задачах, ланцюгове правило диференціювання використовується для обчислення похідної складеної функції, якщо ж представити пряме розповсюдження у ЗНМ як сукупність великої кількості складених функції, то їх похідні(градієнти) розраховуються за допомогою цієї техніки. У простому вигляді, ланцюгове правило можна інтерпретувати наступним чином: припустимо що існує дві функції $f(x)$ та $g(x)$ їх складена функція дорівнює $h(x) = f(g(x))$, за допомогою ланцюгового правила можна знайти похідну цієї функції відносно змінної x за допомогою формули: $\frac{dh}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$, що означає що похідна складеної функції h може бути знайдена як добуток похідної функції f відносно змінної g та похідної функції g відносно змінної x.

3.3.2 Ланцюгове правило диференціювання у ЗНМ

У згортковому шарі кожну згорку можна представити у вигляді матричної функції яка накладає фільтр(матрицю) на вхідні дані. Основна ціль зворотного розповсюдження підлаштувати значення фільтрів в згортці для мінімізації помилки на

виході ЗНМ, таким чином якщо у ЗНМ є деяка кількість згорткових шарів, то у загальному випадку результат мережі буде мати вигляд h(x) = f(g(m(.....(x))), де f, g, m,... - шари згортки і повнозв'язні шари. І для розрахунку похідної від функції h необхідно застосувати ланцюгове правило для підрахунку диференціалу на кожному з цих шарів.

Зворотне розповсюдження на одному згортковому шарі має наступний вигляд:

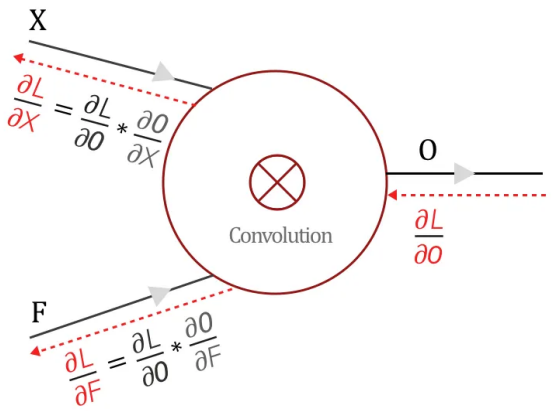

Рис. 13 Приклад зворотного розповсюдження у шарі ЗНМ

При процесі зворотного розповсюдження, після проходу крізь шар утворюється дві матриці похідних, одна матриця використовується для калібровки вагів у фільтрі згортки, інша йде у попередній шар для підрахунку похідної з огляду на результат моделі, для підлаштування вагів у попередньому шарі. Для розрахунку матриці градієнтів фільтру F зважаючи на вхідні дані X використовується формула: $\frac{dL}{dF} = Conv(X, \frac{dL}{dO})$, де X - це вхідні дані у шар згортки, $\frac{dL}{dO}$ - градієнт помилки на вихідних даних наступного шару, $Conv$ - функція згортки. Цю матрицю використовують для оптимізації вагів фільтрів згортки за допомого градієнтних алгоритмів про які йдеться нижче. Також для попереднього шару необхідно розрахувати матрицю похідних для вхідних даних $\frac{dL}{dX}$, це можна зробити за формулою: $\frac{dL}{dX} = Full\ Conv(F^T, \frac{dL}{dO})$, де F^T - транспонована матриця значень

фільтру, $\frac{dL}{dO}$ - градієнт помилки на вихідних даних наступного шару, $Full\ Conv$ - функція повної згортки.

Функція повної згортки є оберненою до функції згортки і означає що кожний елемент матриці $\frac{dL}{dX}$ дорівнює сумі добутків між похідними від вихідного шару що відповідають і-му елементу матриці $\frac{dL}{dO}$ і значення і-го елементу з транспонованої матриці фільтру F^T. Матриця $\frac{dL}{dX}$ використовується як $\frac{dL}{dO}$ у попередньому шарі для розрахунку градієнтів.

Процес розрахунку матриць похідних для фільтрів згорткових шарів відбувається до моменту поки не досягне найпершого шару і не буде згенеровано всі матриці градієнтів для всіх ваг у ЗНМ. Далі ці матриці використовуються для оптимізації оригінальних фільтрів у мережі за допомогою різноманітних алгоритмів розроблених для вирішення проблем градієнтної оптимізації.

3.4 Градієнтні алгоритми для оновлення вагів у шарах ЗНМ

Як вже було згадано, розраховані градієнти використовуються для оновлення вагів у ЗНМ за допомогою використання градієнтів отриманих внаслідок розрахунку помилки між реальними і спрогнозованими значеннями датасету. У цьому розділі детально розглянуто основні найпопулярніші алгоритми оптимізації, які використовуються для навчання нейронних мереж, їх особливості та математичне підґрунтя. Використання ефективних алгоритмів оптимізації допомагає збільшити швидкість збіжності і покращити точність навчання моделі.

3.4.1 Градієнтний спуск

Градієнтний спуск - це найпростіший і ймовірно найпоширеніший алгоритм оптимізації що використовується у нейронних мережах. Його основна ідея полягає в ітеративному оновленні ваг моделі у напрямку протилежному до градієнту функції

втрат, що дозволяє зменшити вплив марно виділених ознак моделі і зменшити помилку. Градієнтний спуск оновлює параметри згідно з наступною формулою:

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla J(\theta_t)$$

- де θ_t, θ_{t+1} - значення параметрів ЗНМ на кроці t
- α - навчальний крок
- $\nabla J(\theta_t)$ - градієнт функцій втрат для кожного параметру моделі

Навчальний крок регулює швидкість спуску(оптимізації) вагів, зазвичай значення навчального кроку близьке до 10^{-3}, занадто великий навчальний крок може призвести до того, що алгоритм буде "перестрибувати" оптимальну точку та не збере достатньо інформації про локальне середовище, що може призвести до нестабільної збіжності або навіть розбіжності. З іншого боку, занадто малий навчальний крок може призвести до повільної збіжності або затримки у навчанні. Зазвичай гіпер параметр швидкості навчання підбирається експериментальним шляхом і може варіюватися в залежності від вибраної архітектури і навчального набору даних.

Градієнтний спуск може застосовуватись як до всього набору даних (пакетний градієнтний спуск), так і до окремих прикладів (стохастичний градієнтний спуск).

3.4.2 Стохастичний градієнтний спуск

Стохастичний градієнтний спуск є покращеною версією класичного градієнтного спуску, замість розрахунку градієнтів на всьому наборі даних, стохастичний градієнтний спуск обчислює градієнт на основі окремих прикладів. Це значно пришвидшує роботу алгоритму і дає можливість працювати з великими наборами даних.

Формула оновлення параметрів у стохастичному градієнтному спуску має наступний вигляд:

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla J(\theta_t; x^{(i)}, y^{(i)})$$

- де θ_t, θ_{t+1} - значення параметрів ЗНМ на кроці t
- α - навчальний крок
- $x^{(i)}, y^{(i)}$ - і-й приклад з набору даних
- $\nabla J(\theta_t; x^{(i)}, y^{(i)})$ - градієнт функцій втрат для і-го - елементу набору даних

Стохастичний градієнтний спуск потребує набагато менше пам'яті і є швидшим за градієнтний спуск за рахунок обчислення градієнту для одного вибраного прикладу з набору даних за раз, проте з іншої точки зору його значення мають більшу дисперсію і варіацію, що може спричинити більшу кількість кроків для збіжності моделі.

Великим недоліком у градієнтному та стохастичному градієнтному спусках є проблема "застрягання" у локальних мінімумах при оптимізації вагів, це відбувається, коли оптимізаційний алгоритм зупиняється в точці, де градієнт наближається до нуля, і не може продовжити знаходити глобальний мінімум функції втрат. Цю проблему можна подолати за допомогою алгоритмів що використовують історичні дані про оптимізацію вагів на попередніх кроках.

3.4.3 Алгоритм з імпульсом

Алгоритм з імпульсом, також відомай як момент(англ. Momentum) відноситься до алгоритмів оптимізації що застосовують історію навчання для покращення якості.

Цей алгоритм використовує накопичувальну швидкість для збільшення швидкості збіжності та подолання проблеми "застрягання" в локальних мінімумах.

У моменті оновлення вагів враховує не тільки поточний градієнт, але й попередні оновлення параметрів. Це дозволяє алгоритму "ускорюватись" у напрямку попереднього оновлення та зменшує його коливання навколо локальних мінімумів. Формула оновлення параметрів моменту виглядає наступним чином:

$$\theta_{t+1} = \theta_t - \alpha \cdot (\beta \cdot v_{t-1} + (1 - \beta)\nabla J(\theta_t)$$

- де θ_t, θ_{t+1} - значення параметрів ЗНМ на кроці t
- α - навчальний крок
- v_{t-1} - накопичувальна швидкість на кроці t - 1
- β - константа(коефіцієнт згасання) що контролює вплив швидкості на попередньому кроці
- $\nabla J(\theta_t)$ - градієнт функцій втрат

Момент дозволяє більш стабільно рухатись у напрямку градієнта та зменшує ймовірність застрягання в локальних мінімумах. Він може бути особливо корисним при оптимізації глибоких нейронних мереж, де є багато локальних мінімумів.

3.4.4 RMSprop

Основна ідея RMSprop полягає в адаптації навчального кроку для кожного параметра моделі в залежності від швидкості зміни градієнтів цього параметра. Це дозволяє більш агресивно змінювати параметри з низькими градієнтами і менш агресивно змінювати параметри з великими градієнтами, що допомагає уникнути проблеми з перескакуванням через оптимальні точки.

Математична формула оновлення параметрів з використанням RMSprop має наступний вигляд:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{((\beta \cdot v_{t-1} + (1-\beta)) * \nabla J(\theta_{t+1}))^2 + \epsilon}}$$

- де θ_t, θ_{t+1} - значення параметрів ЗНМ на кроці t
- α - навчальний крок
- v_t - накопичений квадрат градієнту на кроці t

- β - експоненціально зменшуваний коефіцієнт, який зазвичай встановлюється у діапазоні від 0.9 до 0.999
- $\nabla J(\theta_t)$ - градієнт функцій втрат
- ϵ - малий додаток для числової стійкості

3.4.5 Adam

Adam(англ. Adaptive Momentum Estimation) є комбінацією двох перелічених вище алгоритмів, його прийнято вважати найпопулярнішим алгоритмом оптимізації що використовується в найсучасніших архітектурах ЗНМ. Він комбінує ідеї моменту та адаптивного кроку навчання для досягнення швидкої збіжності та стабільності.

Adam використовує дві експоненційно зважені середні для оцінювання градієнту та його квадрату. Перша експоненційно зважена середня, m_t, оцінює градієнт, а друга,v_t, оцінює квадрат градієнту:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla J(\theta t)$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla J(\theta t))^2$$

- β_1, β_2 - гіпер параметри що контролюють експоненційну зваженість
- $\nabla J(\theta_t)$ - градієнт функцій втрат

Звідси формула для оновлення вагів: $\theta_{t+1} = \theta_t - \alpha \cdot \frac{m_t}{\sqrt{v_t + \epsilon}}$

Adam демонструє хорошу швидкість збіжності та стійкість до великих градієнтів та шуму. Він також автоматично адаптує крок навчання для кожного параметра на основі оцінок градієнту та його дисперсії.

Тож алгоритмі оптимізації є надзвичайно важливим елементом у навчанні ЗНМ, вибір алгоритму дуже сильно впливає на остаточний результат та швидкість навчання.

3.5 Техніка тонкого налаштування нейромережі

Зазвичай тренування нейронної мережі з нуля потребує надзвичайно великої кількості часу та технічних можливостей, сучасні ЗНМ мають десятки шарів і сотні мільйонів гіпер параметрів для оптимізації, саме тому існує дуже простий та швидкий спосіб налаштувати ЗНМ під власні потребі - тонка настройка(англ. fine-tuning).

Fine-tuning Згорткової Нейронної Мережі є підходом до доналаштування попередньо навчених моделей з метою вирішення нових задач та роботи з новими даними. Цей підхід широко використовується в глибинному навчанні, оскільки дозволяє значно зменшити зусилля, необхідні для навчання моделі на нових даних. Fine-tuning передбачає замороження параметрів моделі та зміну вихідного шару для вирішення нової задачі.

Одним з основних використань fine-tuning є робота з великими навчальними наборами даних, коли навчання моделі з нуля є не реалістичним. Замість цього, можна використовувати попередньо навчену модель, що була підготовлена на схожих даних, та доналаштовувати її під нові умови та задачі. Це значно зменшує кількість даних, які необхідні для досягнення достатньої точності моделі, та спрощує процес навчання.

Одним з прикладів успішного використання fine-tuning є розпізнавання облич в зображеннях. В даному випадку, використовуючи попередньо навчену модель, яка вміє розпізнавати обличчя, можна досягнути значно кращих результатів у роботі зі змінними умовами, такими як різке освітлення або наявність інших об'єктів на зображенні. Для досягнення цього, необхідно заморозити ваги моделі, яка вміє розпізнавати обличчя, та доналаштувати вихідний шар моделі для вирішення конкретної задачі, наприклад, розпізнавання облич у темряві або на зображеннях з багатьма об'єктами.

Існують різні підходи до fine-tuning, наприклад, зміна глибини моделі, додавання або видалення шарів, зміна гіпер параметрів та інші. Однак, важливо дотримуватися певних правил, щоб забезпечити ефективність та стабільність моделі

після доналаштування. Найперше, необхідно забезпечити достатню кількість даних для тонкого налаштування моделі. Це може бути досягнуто шляхом використання техніки аугментації даних для збільшення кількості даних. Друге, необхідно підібрати правильний навчальний крок для досягнення оптимальної точності та уникнення проблем зі збіжністю. Третє, необхідно враховувати специфіку нової задачі та відповідність до попередньої моделі.

Узагальнюючи, fine-tuning є ефективним підходом до доналаштування попередньо навчених моделей для вирішення нових задач та роботи з новими даними.

РОЗДІЛ 4: Огляд популярних архітектур ЗНМ

Згорткові нейронні мережі стали основним інструментом для обробки зображень, оскільки вони дозволяють досягти вражаючих результатів у задачах класифікації, виявлення об'єктів та сегментації. Одним з ключових елементів успіху ЗНМ є їх архітектура, яка включає в себе шари згортки, пулінгу та повнозв'язаних шарів, огляд яких вже було описано вище. У цьому розділі розглянемо найпопулярніші архітектури ЗНМ та їх особливості.

4.1 LeNet

Першою архітектурою ЗНМ, у сучасному вигляді прийнято вважати LeNet, вона складається з трьох згорткових шарів та двох повнозв’язних шарів. Ця модель була розроблена для розпізнавання рукописних цифр та є однією з найбільш простих архітектур ЗНМ, але з огляду на час створення і технології минулого створення цієї ЗНМ стало відправною точкою для подальшого розвитку галузі.

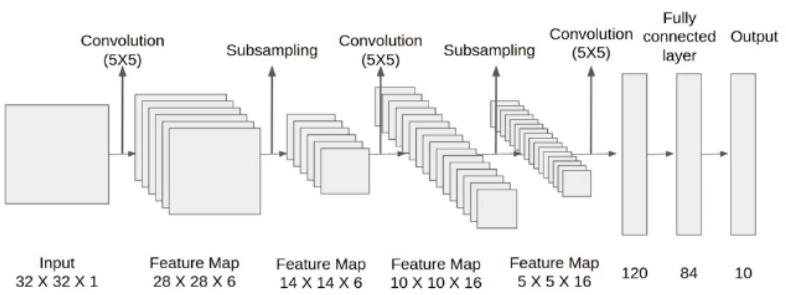

Рис. 14 Візуалізація архітектури LeNet

Як видно з зображення модель приймає чорно-біле зображення розміром 32×32 пікселі, пропускає крізь шари, і на останньому шарі генерує 10 значень належності до певного класу.

4.2 AlexNet

Запропонована в 2012 році командою дослідників з Університету Торонто та компанії Google. AlexNet складається з п'яти згорткових шарів та трьох повнозв'язаних шарів. Архітектура була розроблена для класифікації зображень на даних ImageNet з 1,2 мільйонами зображень та 1000 класів. AlexNet використовує техніку дропаут для регуляризації та активаційну функцію ReLU, що дозволяє досягти набагато кращих результатів порівняно зі звичайними нейронними мережами.

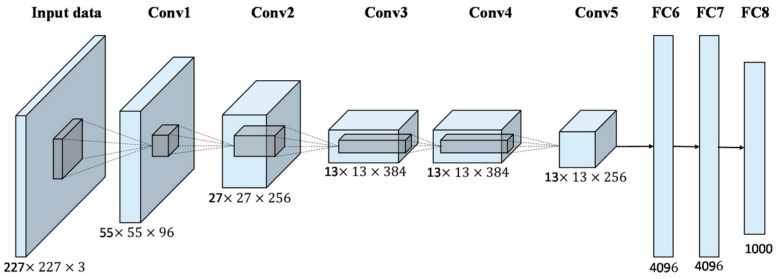

Рис. 15 Візуалізація архітектури AlexNet

На відміну від моделі запропонованої Ле Куном, AlexNet здатна приймати зображення в звичному форматі ЧЗС(червоний, зелений, синій), тобто обробляти кольорові зображення.

4.3 GoogLeNet(Inception)

Запропонована в 2014 році командою дослідників з Google. GoogLeNet складається з 22 згорткових шарів та повнозв'язаних шарів. Особливістю архітектури є використання модулей Inception, які використовують згортки різних розмірів та пулінгу для отримання більш різноманітних функцій. GoogLeNet є дуже ефективною

архітектурою, яка досягає високої точності класифікації зображень за допомогою орієнтовно малої кількості параметрів.

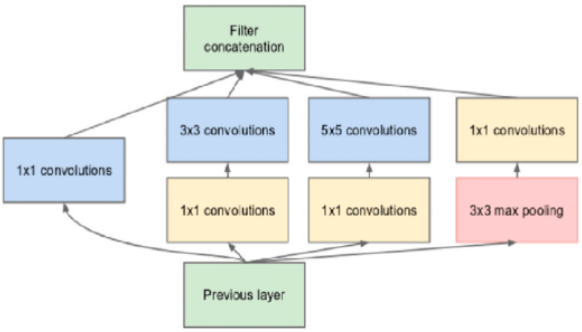

Рис. 16 Візуалізація шару Inception

Так як замість однакових фільтрів у шарі, модель Inception використовує паралельний розрахунок різних за своїм складом згорток, це дозволяє виокремлювати відмінні за наповненням ознаки, що значно прискорює процес навчання і покращує точність моделі.

4.4 ResNet

Запропонована в 2015 році командою дослідників з Microsoft Research. ResNet складається з глибокої мережі з 152 згорткових шарів. Особливістю архітектури є використання модулів зі зв'язками "skip connection", що дозволяють передавати вхідні дані в наступний шар без обробки.

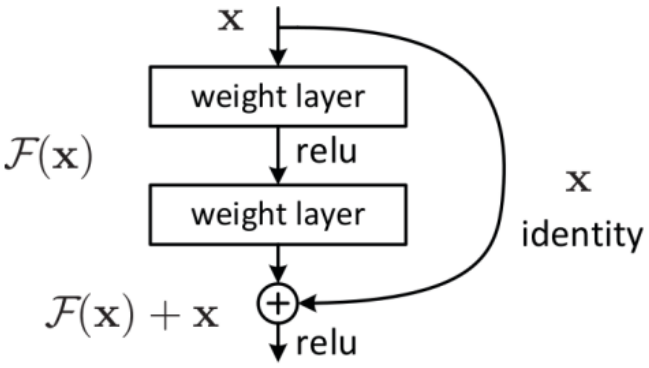

Рис. 17 Візуалізація процесу "Skip Connection"

Суть процесу "skip connection" полягає в тому, що до значень на виході глибших шарів, додаються значення попередніх шарів помножені на деякий коефіцієнт. Такий

підхід дозволяє запобігти проблемі зникаючого градієнту, проблема стосується випадків з виокремленими ознаками з низькими значеннями, при навчанні глибоких мереж та дозволяє створити ще більш глибокі мережі з високою точністю класифікації.

4.5 Efficient Net

Запропонована в 2019 році командою дослідників з Google. EfficientNet є архітектурою, яка досягає високої точності класифікації зображень за допомогою меншої кількості параметрів порівняно з іншими архітектурами.

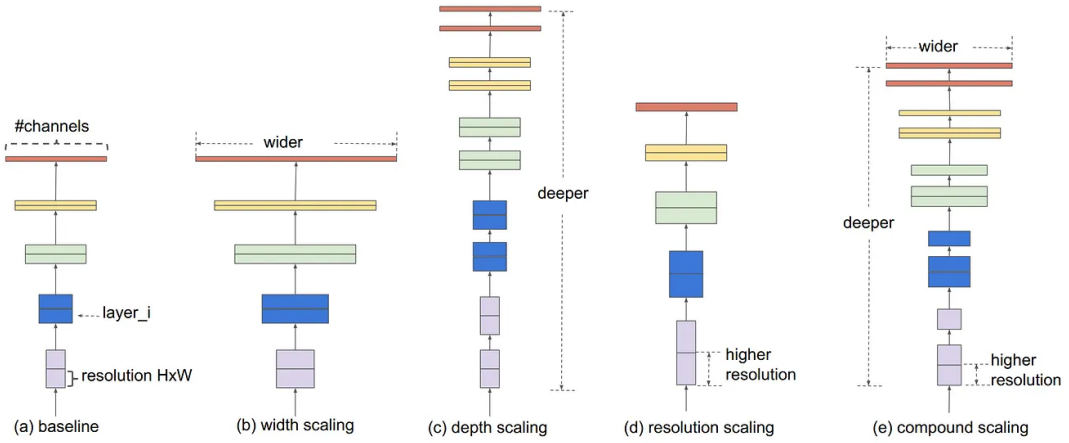

Рис. 18 Порівняння різних маніпулятивних технік до структури зображення

Особливістю архітектури є використання компаунд масштабування, яке дозволяє ефективно масштабувати мережу за рахунок зміни розмірів фільтрів, глибини та ширини мережі. Під час розробки попередніх архітектур моделей, дослідники визначили що для досягнення найкращого результату в шарах ЗНМ необхідно зберігати баланс між шириною, висотою та роздільною здатністю моделі, тож саме в архітектурах моделей Efficient Net використовується техніка збереження співвідношення між цими параметрами зображення для досягнення найвищої точності моделі.

Кожна з вищезгаданих архітектур має свої переваги та недоліки. Вибір архітектури для задачі залежить від ресурсів, доступних для тренування та

застосування мережі, а також від конкретних вимог до точності класифікації. Для більшості задач можна знайти вже попередньо натреновану модель на великому наборі даних, яку можна дотренувати (fine-tune) для виконання конкретної задачі. Також можна створити власну архітектуру, але для створення складної та якісної архітектури необхідно витрати значно більше часу та ресурсів.

РОЗДІЛ 5: Експериментальна частина

Метою експериментальної роботи на тему класифікація зображень за допомогою нейронних мереж є розробка та тестування алгоритмів, які будуть використовуватись для автоматичного розпізнавання об'єктів на зображеннях, також основна ціль роботи застосувати на практиці теоретичні складові ЗНМ що були представлені у попередніх розділах.

Для реалізації експериментальної частини використано мову програмування Python та ряд бібліотек, що розроблені для роботи з багатовимірними масивами. У мережі інтернет зберігається надзвичайно велика кількість даних у вільному доступі. При написанні цієї наукової роботи було зібрано 2248 зображень автомобілів з платформи для продажу транспортних засобів auto ria, що розподілені на 55 класів(марок авто).

5.1 Аугментація датасету та ділення на тренувальну та валідаційну вибірки

Для досягнення найкращих результатів під час тренування нейронних мереж, що вирішують проблему класифікації, необхідно зберігати баланс класів(кількість екземплярів кожного класу має бути однакова або близька до однакового), а також , необхідно додати на деякі зображення додаткові фільтри, що змінюють якість або

кольорову гамму зображення, для урахування всіх особливостей реального життя. Обидві проблеми можна вирішити за допомогою аугментації тренувальної вибірки.

За допомогою бібліотеки Albumentations на мові Python було додатково створено деяку кількість зображень, таким чином, щоб кожен клас навчальної вибірки налічував 250 екземплярів. Для цього було перенесено всі оригінальні зображення і для кількості 250 - n, де n це кількість оригінальних зображень класу, створено картинки які походять від оригінальних і до яких випадковим чином застосовані наступні трансформації:

```
# функція трансформації зображень
transform = A.Compose([
# фільтр блюрінгу застосовується з ймовірністю 50%
A.Blur(blur_limit=(2, 5), p=0.5),
# фільтр з додаванням ознак дощу застосовується з ймовірністю 50%
A.RandomRain(brightness_coefficient=0.9, drop_width=1, blur_value=1, p=0.5),
# фільтр контрасту та яркості зображення застосовується з ймовірністю 50%
A.RandomBrightnessContrast(contrast_limit=(0.4, 0.5), brightness_limit=(-0.4, 0.2), p=0.5),
# фільтр з випадковим поворотом по горизонталі до 30% застосовується з ймовірністю 50%
A.Rotate(limit=30, p = 0.5),
# фільтр з погіршенням якості від 50 до 75% застосовується з ймовірністю 50%
A.ImageCompression(quality_lower=50, quality_upper=75, p=0.5),
# фільтр з додаванням Гауссовго шуму застосовується з ймовірністю 50%
A.GaussNoise(var_limit=(10.0, 50.0), per_channel=True, p=0.5),
# фільтр відзеркаленням по горизонталі застосовується з ймовірністю 50%
A.HorizontalFlip(p = 0.5)
])
```

Так як зображення представлені у вигляді багатовимірних масивів ширина × висоту × кількість кольорів, кожна трансформація, з ймовірністю в 50%, застосовується до кожного значення кожного пікселю картинки і за власним запрограмованим правилом змінює значення від 0 до 255, що впливає на вигляд зображення.

Саме ці фільтри були обрані через те що вони не змінюють загальні риси фотографії, на відміну від випадкового відрізання(наприклад), але повністю відкидають ймовірність наявності повторів при генерації тренувальної вибірки. Цей процес вібувається за домогою наступного блоку коду:

```
# З кожної папки класів
for folder in glob.glob('/Users/mykyta/Dyploma/car_images/*'):
    # Генеруэмо шляхи до оригінальних зображень
    images_path = glob.glob(f'{folder}/*')
    # Створюємо шлях для аугментованих зображень
    folder_path_list = folder.split('/')
    folder_path_list[-2] = folder_path_list[-2] + '_augmented'
    # Якщо папка з створеними зображеннями існує, пропускаємо
    if os.path.exists("/".join(folder_path_list)):
        continue
    else:
        os.mkdir("/".join(folder_path_list))
    # Для недостаючої кількості оригінальних зображень
    for i in range(250 - len(images_path)):
        # Випадковим чином вибираємо оригінальне зображення
        img_path = random.choice(images_path)
        image = load_img(img_path)
        # Накладаємо трансформації
        transformed_img = transform(image=image)['image']
        # Зберігаємо трансформоване оригінальне зображення
        Image.fromarray(transformed_img).save("/".join(folder_path_list) +
                                              f'/augmented_{i}_' + img_path.split('/')[-1])
    # Зберігаємо всі оригінальні зображення в потрібну папку
    for original_img_path in images_path:
        image = load_img(original_img_path)
        Image.fromarray(image).save("/".join(folder_path_list) + '/' +
original_img_path.split('/')[-1])
```

Таким чином в кожній папці класів маємо по 250 зображень підготовлених для процесу тренування і валідації моделей.

Рис. 19 Візуалізація процесу аугментації зображення

На рисунку 19 зображено приклад трансформованих даних, у лівому верхньому куті представлено оригінальне зображення авто Seat і 8 зображень з фільтрами накладеними на оригінал.

Наступним кроком, на основі аугментованих даних, було створено 2 вибірки даних тренувальна та валідаційна за допомогою бібліотеки tensorflow для оптимізації та тестування моделей, з наступним розподілом: 11 тисяч зображень потрапило у тренувальний датасет і 2750 у тестувальний, що є 80 і 20 відсотків від усіх картинок відповідно. Також кожен з датасетів поділений на підгрупи по 96 зображень для того щоб ітеративно навчати модель і не перевантажувати пристрій на якому відбувається процес тренування.

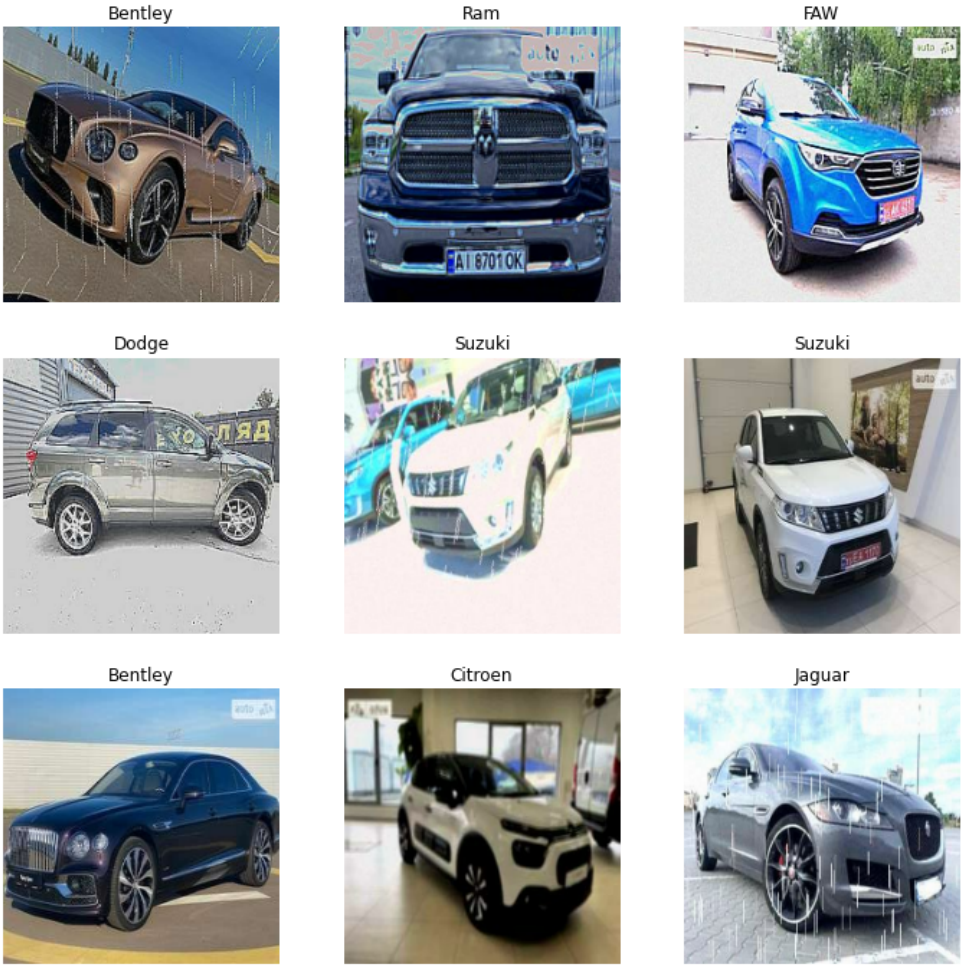

Рис. 20 Приклади зображень з тренувальної вибірки

На рисунку 20 зображено 9 випадково взятих екземплярів з тренувальної вибірки та їх класи у вигляді написів над картинками.

5.2 Створення власної архітектури моделі

За допомогою бібліотек tensorflow та keras було запрограмовано пошарову ініціалізацію власної архітектури моделі.

```
Model: "Власна Архітектура моделі"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (InputLayer)    [(None, 224, 224, 3)]     0

 resizing_layer (Resizing)   (None, 224, 224, 3)       0

 rescaling_layer (Rescaling)  (None, 224, 224, 3)      0

 conv_layer_1 (Conv2D)       (None, 224, 224, 16)      448

 max_pooling_1 (MaxPooling2D  (None, 112, 112, 16)     0
 )

 conv_layer_2 (Conv2D)       (None, 112, 112, 32)      4640

 max_pooling_2 (MaxPooling2D  (None, 56, 56, 32)       0
 )

 conv_layer_3 (Conv2D)       (None, 56, 56, 64)        18496

 avg_pool (AveragePooling2D)  (None, 28, 28, 64)       0

 batch_normalization_2 (Batc  (None, 28, 28, 64)       256
 hNormalization)

 top_dropout_3 (Dropout)     (None, 28, 28, 64)        0

 flatten_24 (Flatten)        (None, 50176)             0

 dense_48 (Dense)            (None, 64)                3211328

 dense_49 (Dense)            (None, 55)                3575

=================================================================
Total params: 3,238,743
Trainable params: 3,238,615
Non-trainable params: 128
_________________________________________________________________
```

Рис. 21 Зображення власної архітектури моделі

Перший шар моделі ініціалізує зображення з розмірністю 224×224×3 пікселів, тобто з висотою і шириною 224, і кількістю кольорів що дорівнює 3. У наступному шарі зображення приводиться до необхідного розміру по висоті та ширині у випадку якщо вони відрізняються від вищезазначених. У шарі під назвою rescaling_layer значення кожного пікселю нормалізуються у діапазоні від 0 до 1(оригінальне значення пікселів дорівнює від 0 до 255), таким чином спрощуються задача оптимізації через відсутність великих значень пікселів. Шари з назвами conv_layer_1, conv_layer_2, conv_layer_3 є шарами згортками і накладають відповідно 16, 32 та 64 фільтри на зображення. Так само і шари max_pool_1, max_pool_2 та avg_pool

відповідають за операції пулінгу після згорткових шарів. Шар з назвою top_dropout_3 автоматично відкидає 25% нейронів між останнім шаром згортки та першим повнозв'язним шаром. Шар flatten перетворює дані розмірністю 28×28×64 у єдиний вектор розмірністю 50176, задля зручної обробки даних у повнозв'язних шарах. Останні шари dense_48, dense_49 є повнозв'язними і генералізують виокремлені на попередніх шарах ознаки для побудови вектору класів розмірністю 55 для зображення. Як видно з малюнку модель налічує 3238743 ваги з яких всі окрім 128 підлягають оптимізації під час тренування моделі. Також до кожного згорткового і повнозв'язного шарів застосовано L1 регуляризацію, що обмежує модель від перенавчання.

Код для ініціалізації моделі виглядає наступним чином:

```
input = tf.keras.layers.Input((img_height, img_width, 3), name = "input_layer")
my_model = tf.keras.layers.Resizing(img_height, img_width, name = "resizing_layer")(input)
my_model = tf.keras.layers.Rescaling(1./255, name = "rescaling_layer")(my_model)
my_model = tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu',
                                  kernel_regularizer=tf.keras.regularizers.l1(0.01),
                                  bias_regularizer=tf.keras.regularizers.l2(0.01),
                                  name = "conv_layer_1")(my_model)
my_model = tf.keras.layers.MaxPooling2D(name = "max_pooling_1")(my_model)
my_model = tf.keras.layers.Conv2D(32, 3, padding='same', activation='relu',
                                  kernel_regularizer=tf.keras.regularizers.l1(0.01),
                                  bias_regularizer=tf.keras.regularizers.l2(0.01),
                                  name = "conv_layer_2")(my_model)
my_model = tf.keras.layers.MaxPooling2D(name = "max_pooling_2")(my_model)
my_model = tf.keras.layers.Conv2D(64, 3, padding='same', activation='relu',
                                  kernel_regularizer=tf.keras.regularizers.l1(0.025),
                                  bias_regularizer=tf.keras.regularizers.l2(0.025),
                                  name = "conv_layer_3")(my_model)
my_model = tf.keras.layers.AveragePooling2D(name="avg_pool")(my_model)
my_model = tf.keras.layers.BatchNormalization(name = "batch_normalization_2")(my_model)
my_model = tf.keras.layers.Dropout(0.25, name="top_dropout_3")(my_model)
my_model = tf.keras.layers.Flatten()(my_model)
my_model = tf.keras.layers.Dense(64, activation='relu')(my_model)
output = tf.keras.layers.Dense(55,
                                  kernel_initializer='ones',
                                  kernel_regularizer=tf.keras.regularizers.L1(0.01),
                                  activity_regularizer=tf.keras.regularizers.L2(0.01),
                                  activation='softmax')(my_model)
my_model = Model(inputs=[input], outputs=[output], name = 'Власна Архітектура моделі')
my_model.compile(loss = 'categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

5.3 Тренування моделі з власною архітектурою

Тренування моделей ЗНМ відбувається за допомогою епох, 1 епоха включає в себе послідовне проходження всіх зображень тренувальної вибірки крізь модель, таким чином для кожного зображення з моделі відбувається процес прямого розповсюдження, підрахунок помилки за допомогою функції втрат, зворотне розповсюдження і оновлення вагів нейронної мережі. За допомогою графічної пам'яті відеокарти пристрою, для кожного зображення ці процеси можна запустити паралельно, проте, зазвичай, пам'яті не вистачає одразу на всі зображення, саме з цієї причини тренувальну та валідаційну вибірки було поділено на підгрупи(батчі) по 96 екземплярів, таким чином під час кожної епохи крізь модель пропускають 11000 / 96 ~ 115 батчів даних на основі яких підраховується помилка і метрика точності моделі, оновлюються ваги і починається нова епоха тренування, також для зручності і контролю перенавчання на кожній епосі можна підраховувати точність моделі на валідаційних даних.

У процесі експерименту було запущено 15 епох тренування на основі тренувального датасету з наступними результатами:

```
Epoch 1/15
115/115 [==========] - 47s 380ms/step - loss: 44.2438 - accuracy: 0.2154 - val_loss: 38.7444 - val_accuracy: 0.1247
Epoch 2/15
115/115 [==========] - 43s 372ms/step - loss: 36.7208 - accuracy: 0.4146 - val_loss: 37.5840 - val_accuracy: 0.2280
Epoch 3/15
115/115 [==========] - 42s 360ms/step - loss: 35.4571 - accuracy: 0.5326 - val_loss: 36.4299 - val_accuracy: 0.2996
Epoch 4/15
115/115 [==========] - 43s 371ms/step - loss: 34.3737 - accuracy: 0.6320 - val_loss: 35.2886 - val_accuracy: 0.3629
Epoch 5/15
115/115 [==========] - 42s 359ms/step - loss: 33.3836 - accuracy: 0.7075 - val_loss: 34.2996 - val_accuracy: 0.4564
Epoch 6/15
115/115 [==========] - 43s 375ms/step - loss: 32.4187 - accuracy: 0.7732 - val_loss: 34.5438 - val_accuracy: 0.4036
Epoch 7/15
115/115 [==========] - 42s 367ms/step - loss: 31.4881 - accuracy: 0.8265 - val_loss: 35.5980 - val_accuracy: 0.3236
Epoch 8/15
115/115 [==========] - 43s 372ms/step - loss: 30.6260 - accuracy: 0.8498 - val_loss: 34.6052 - val_accuracy: 0.3836
Epoch 9/15
115/115 [==========] - 42s 365ms/step - loss: 29.6851 - accuracy: 0.8866 - val_loss: 33.4204 - val_accuracy: 0.3909
Epoch 10/15
115/115 [==========] - 43s 372ms/step - loss: 28.8432 - accuracy: 0.8848 - val_loss: 33.9634 - val_accuracy: 0.3796
Epoch 11/15
115/115 [==========] - 43s 371ms/step - loss: 27.9232 - accuracy: 0.9064 - val_loss: 31.3092 - val_accuracy: 0.4760
Epoch 12/15
115/115 [==========] - 42s 365ms/step - loss: 27.0716 - accuracy: 0.9047 - val_loss: 34.3938 - val_accuracy: 0.3495
```

```
Epoch 13/15
115/115 [==========] - 42s 366ms/step - loss: 26.1509 - accuracy: 0.9223 - val_loss: 31.2132 - val_accuracy: 0.4433
Epoch 14/15
115/115 [==========] - 43s 368ms/step - loss: 25.2905 - accuracy: 0.9286 - val_loss: 31.5239 - val_accuracy: 0.4295
Epoch 15/15
115/115 [==========] - 41s 358ms/step - loss: 24.4021 - accuracy: 0.9380 - val_loss: 28.4410 - val_accuracy: 0.5193
```

Видно, що процес тренування не є лінійним, тож модель іноді потрапляє у локальні мінімуми, також на останніх епохах різниця між тренувальними і валідаційними оцінками точності дуже відрізняється, тож нейронна мережа схильна до перенавчання, навіть за наявності регуляризуючих компонентів. Загалом запропонована модель може впоратись з завданням і виконувати задачу класифікації, проте як і було згадано вище, задля створення власної, якісної моделі необхідно значно розширити датасет, і мати змогу використовувати більшу кількість технічних ресурсів. Тож для продовження експерименту було вирішено використати вже натреновану модель Efficient Net B0 для порівняння результатів з власною мережею.

5.4 Ініціалізація та дотренування моделі Efficient Net B0

Бібліотека tensorflow, розроблена дослідниками з компанії Google, є широко використовуваною у галузі розробки нейронних мереж, з її допомогою можна як створювати як власні нейронні мережі, так і завантажувати вже натреновані моделі для тонкого налаштування на своїх даних. Так як найкращою моделлю наразі прийнято вважати Efficient Net, саме її було використано у експерименті.

Для дотренування моделі Efficient Net B0 необхідно додати декілька шарів у кінець завантаженої моделі для налаштування необхідної кількості класів. Таким чином до архітектури моделі було додано шар середнього пулінгу для виокремлення середніх значень з мапи ознак після останнього запрограмованого згорткового шару, шар випадкового відкидання 20% відсотків нейронів, для регуляризації вагів моделі та повнозв’язний шар з 55 класами для побудови вектору класів у наявній задачі. Завантаження моделі та її архітектурні зміни відбуваються за допомогою наступного коду:

```
# Функція для завантаження моделі EfficientNetB0
def build_model(num_classes):
    # Додамо шар з ініціалізацією зображення розміром 2242243 пікселів
    inputs = tf.keras.layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
    # Додамо шар що змінює розмір зображення до потрібних параметрів якщо це
    # необхідно
    inputs = tf.keras.layers.Resizing(IMG_SIZE, IMG_SIZE)(inputs)
    # Завантажимо модель з бібліотеки Tensorflow натреновану на основі датасету
    # imagenet
    model = EfficientNetB0(include_top=False, input_tensor=inputs, weights="imagenet")

    # Зберігаємо вже начені ваги
    model.trainable = False

    # Додамо декілька шарів і змінимо кількість класів запрограмованих у оригінальній моделі
    x = tf.keras.layers.GlobalAveragePooling2D(name="avg_pool")(model.output)
    x = tf.keras.layers.BatchNormalization()(x)
    top_dropout_rate = 0.2
    x = tf.keras.layers.Dropout(top_dropout_rate, name="top_dropout")(x)
    outputs = tf.keras.layers.Dense(num_classes, activation="softmax", name="pred")(x)

    # Конфігуруємо модель
    model = tf.keras.Model(inputs, outputs, name="EfficientNet")
    # Додаємо оптимізаційни алгоритм Adam з початковим навчальним кроком 0.01
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-2)
    model.compile(
        optimizer=optimizer, loss="categorical_crossentropy", metrics=["accuracy"]
    )
    return model
```

Загальна кількість вагів у моделі дорівнює 4125146, але через те що основна кількість вагів вже зафіксована і тренуванню не підлягає, маємо 73015 параметрів, згенерованих у останніх власноруч доданих шарів, які необхідно оптимізувати для класифікації зображень на власному датасеті. Нескладно помітити, що в порівнянні з минулою моделлю кількість ваг, що потребують навчання значно зменшилась, а отже і задача оптимізації моделі може бути виконана з меншу кількість часу і ресурсів.

Оптимізація моделі зайняла 12 епох з наступними результатами:

```
Epoch 1/12
115/115 [==========] - 124s 997ms/step - loss: 2.5652 - accuracy: 0.4987 - val_loss: 2.1995 - val_accuracy: 0.3862
Epoch 2/12
115/115 [==========] - 109s 939ms/step - loss: 1.2081 - accuracy: 0.6925 - val_loss: 1.3203 - val_accuracy: 0.6673
Epoch 3/12
115/115 [==========] - 112s 968ms/step - loss: 0.9849 - accuracy: 0.7415 - val_loss: 1.1101 - val_accuracy: 0.7149
Epoch 4/12
115/115 [==========] - 109s 939ms/step - loss: 0.8417 - accuracy: 0.7695 - val_loss: 1.0709 - val_accuracy: 0.7371
Epoch 5/12
115/115 [==========] - 108s 926ms/step - loss: 0.8369 - accuracy: 0.7675 - val_loss: 1.1048 - val_accuracy: 0.7436
Epoch 6/12
115/115 [==========] - 107s 927ms/step - loss: 0.7702 - accuracy: 0.7925 - val_loss: 1.0750 - val_accuracy: 0.7502
Epoch 7/12
```

115/115 [==========] - 108s 930ms/step - loss: 0.7441 - accuracy: 0.7919 - val_loss: 1.1071 - val_accuracy: 0.7545
Epoch 8/12
115/115 [==========] - 107s 920ms/step - loss: 0.7047 - accuracy: 0.8007 - val_loss: 1.0573 - val_accuracy: 0.7705
Epoch 9/12
115/115 [==========] - 109s 936ms/step - loss: 0.6658 - accuracy: 0.8132 - val_loss: 1.0660 - val_accuracy: 0.7662
Epoch 10/12
115/115 [==========] - 108s 931ms/step - loss: 0.6867 - accuracy: 0.8102 - val_loss: 1.0639 - val_accuracy: 0.7709
Epoch 11/12
115/115 [==========] - 106s 911ms/step - loss: 0.6588 - accuracy: 0.8135 - val_loss: 1.0370 - val_accuracy: 0.7735
Epoch 12/12
115/115 [==========] - 108s 932ms/step - loss: 0.6634 - accuracy: 0.8144 - val_loss: 1.1026 - val_accuracy: 0.7593

Як видно з задокументованого процесу тренування, хоч кількість епох і зменшилась, середній час однієї епохи є більшим за очікуване значення, цьому в першу чергу сприяє більша кількість гіпер параметрів моделі, а також кількість перетворень(шарів) ЗНМ. Efficient Net B0 в загальному вигляді складається з 7-ми блоків шарів, в кожному з яких є шар згортки, шар активації, функція середнього пулінгу, та допоміжні шари, що допомагають зберегти пропорцію між глибиною, шириною та висотою зображення, це значно уповільнює процес обробки одного зображення, але регуляризує модель і призводить до вражаючої точності. Можна помітити, що точність на тренувальних даних значно менша ніж у порівнюваної моделі, і подекуди різниця перевищує 10 відсотків, проте ця модель є набагато стійкішою на валідаційному наборі даних і на останніх епохах різниця між моделями досягає 20-25% відсотків.

5.5 Порівняння результатів власної та попередньо натренованої моделей

У багато класових задачах машинного навчання результат моделей прийнято вимірювати метрикою зваженої точності. Формула для підрахунку зваженої точності виглядає наступним чином: $\frac{1}{K}\sum_{n=0}^{K}\frac{TP}{(TP+TN)}$, де K - це кількість класів даних, TP - це кількість вірно класифікованих зображень певного класу, і TN - кількість невірно класифікованих зображень певного класу. Тож після підрахунку цієї метрики для обох моделей на валідаційних даних, маємо наступні результати:

- Зважена точність для моделі з власною архітектурою дорівнює 0.53

- Зважена точність для дотренованої моделі Efficient Net дорівнює 0.77

Це означає, що перша мережа з ймовірністю в середньому в 53% правильно класифікує представлену автівку, коли друга модель надасть правильну відповідь з ймовірністю 77%, різниця на валідаційних даних є колосальною.

Також розглянемо результати класифікації моделей на конкретних класах

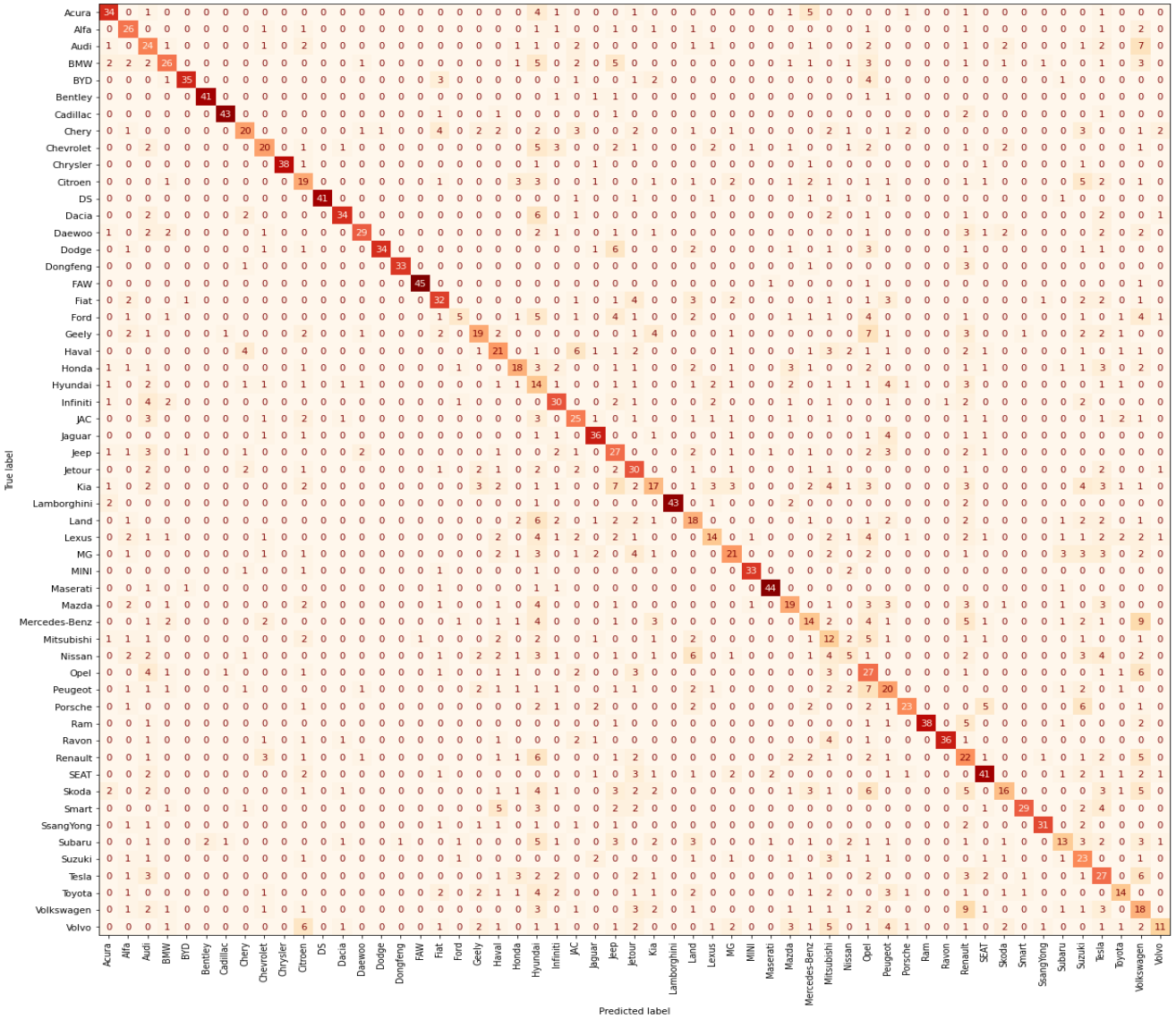

Рис. 22 Зображення результатів моделі з власною архітектурою

На рисунку 22 по вертикалі вказані всі 55 класів автомобілів з валідаційного набору даних, і по горизонталі 55 класів передбачених авто, в кожній комірці вказана кількість авто з реальними значеннями класу і передбаченими. По діагоналі з верхнього лівого кутка до правого нижнього кількість в кожній комірці вказана

кількість правильно класифікованих екземплярів. Тож як бачимо в загальному, модель дійсно непогано класифікує зображення транспортних засобів, з класами FAW та SEAT точність досягає 80% і вище, проте наприклад з прогнозуванням авто марки Ford, Nissan та Volvo, модель справляється менше ніж у 20% випадків.

Подивимось на результат моделі Efficient Net:

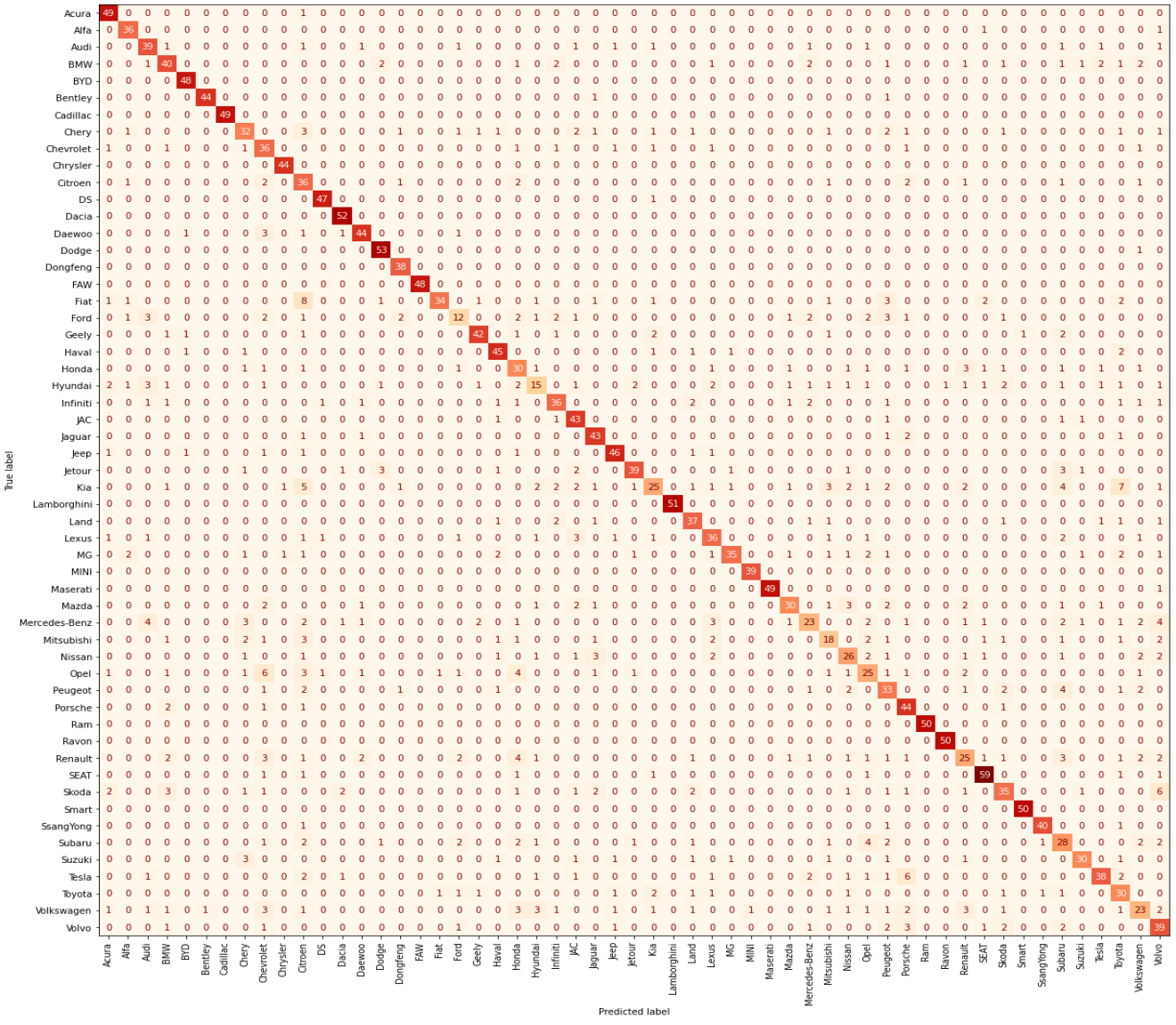

Рис. 23 Зображення результатів моделі Efficient Net B0

З рисунку 23 видно, що модель значно перевершує результати попередньої, проблеми з класифікацією автомобілів марки Volvo та Nissan майже зникли, а з Ford-ами хоч і наявні, проте все одно перевищують результат попередньої моделі.

Тож як висновок можна сказати, що дотренована модель Efficient Net є значно кращою за модель з власною архітектурою, як за загальними середніми показниками, так і за класифікацією конкретних груп транспортних засобів. Це в першу чергу зумовлено більш складною архітектурою моделі створеної групою дослідників з компанії Google, по друге набагато більшими технічними ресурсами, що були використані під час тренування Efficient Net, а також різноманітнішим початковим датасетом що використовувався під час навчання.

Висновок

У атестаційній роботі було розглянуто теоретичні та практичні підходи до розв’язання проблеми класифікації зображень за допомогою нейронних мереж. Докладно описано складну архітектуру та математичні процеси, що відбуваються всередині різноманітних за складом алгоритмів. В ході дослідження було зібрано підготовлено тренувальний набір даних, порівняно декілька підходів до створення та оптимізації моделей з високою точністю класифікації. Також виявлено, що успішність класифікації залежить від вибору архітектури, параметрів нейронної мережі та якості і кількості навчальних даних.

Отримані результати демонструють великий потенціал застосування нейронних мереж для класифікації зображень в різних галузях. Розроблені моделі можуть бути використані для автоматичного розпізнавання об'єктів на зображеннях в різних галузях. Доведено що застосування нейронних мереж дозволяє значно зменшити час та зусилля, необхідні для ручної обробки зображень, та збільшити точність результатів.

Список джерел

1. Mohammad Mustafa Taye, Theoretical Understanding of Convolutional Neural Network: Concepts, Architectures, Applications, Future Directions, 2023
2. Aurelien Geron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems 2nd Edition, 2019
3. Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, J. Santamaría, Mohammed A. Fadhel, Muthana Al-Amidie & Laith Farhan, Review of deep learning: concepts, CNN architectures, challenges, applications, future directions 2021
4. Juan José Cabrera, Sergio Cebollada, María Flores, Óscar Reinoso & Luis Payá, Training, Optimization and Validation of a CNN for Room Retrieval and Description of Omnidirectional Images, 2022
5. Fei-Fei Li, Yunzhu Li, Ruohan Gao, CS231n: Deep Learning for Computer Vision Stanford Courses, 2017
6. Magnus Ekman, Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow 1st Edition, 2021
7. An Introduction to Neural Network, Kevin Gurney, 1997