НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут прикладного системного аналізу
Кафедра штучного інтелекту

«На правах рукопису»

УДК 004.8.032.26:004.932](043.3)

До захисту допущено:

В.о. завідувачки кафедри

_____ Ірина ДЖИГИРЕЙ

«___»_____2024 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Системи і методи штучного інтелекту»

зі спеціальності 122 «Комп'ютерні науки»

на тему: «Структурно-параметричний синтез капсульних нейронних мереж для розв'язання задачі обробки зображень»

Виконав:
студент ІІ курсу, групи КІ-21мп
Кудрєв Денис Олексійович _____

Науковий керівник:
проф. кафедри ШІ, д. т. н., професор
Синєглазов Віктор Михайлович _____

Рецензент:
проф. кафедри ТКРС,
Національного авіаційного університету,
д. т. н., професор
Заліський Максим Юрійович _____

Засвідчую, що у цій магістерській дисертації
немає запозичень з праць інших авторів
без відповідних посилань
Студент: _____

Київ – 2024

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра штучного інтелекту

Рівень вищої освіти – другий (магістерський)
Спеціальність – 122 «Комп’ютерні науки»
Освітньо-професійна програма «Системи і методи штучного інтелекту»

ЗАТВЕРДЖУЮ
В.о. завідувачки кафедри
_____ Ірина ДЖИГИРЕЙ
«___»_____2023 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

1. Тема дисертації «Структурно-параметричний синтез капсульних нейронних мереж для розв'язання задачі обробки зображень», науковий керівник дисертації Синєглазов Віктор Михайлович, д. т. н., професор, затверджені наказом по університету від «08» листопада 2023 р. №5200-с.
2. Термін подання студентом дисертації – 07 січня 2024 року.
3. Об’єкт дослідження – капсульні нейронні мережі.
4. Предмет дослідження – структурно-параметричний синтез капсульних нейронних мереж.
5. Перелік завдань, які потрібно зробити.
    1) Здійснити огляд технічної літератури за темою роботи.
    2) Дослідити актуальність обраної теми.
    3) Ознайомитись із існуючими моделями та методами реалізації капсульних нейронних мереж.
    4) Здійснити порівняльний аналіз методів реалізації капсульних нейронних мереж.

5) Провести експерименти над топологією капсульних нейронних мереж та проаналізувати результати.
6) Розробити гібридний алгоритм структурно-параметричного синтезу капсульних нейронних мереж.
7) Застосувати алгоритм, проаналізувати та порівняти результати з капсульними мережами знайденими вручну.
8) Провести аналіз ринкових можливостей запуску стартап проєкту.
9) Розробити концептуальні висновки.
10) Підготувати ілюстративний матеріал.
11) Оформити пояснювальну записку.

6. Перелік ілюстративного матеріалу: презентація.

7. Орієнтовний перелік публікацій: заплановано опублікування результатів роботи “STRUCTURAL-PARAMETRIC SYNTHESIS OF CAPSULE NEURAL NETWORKS” авторів Синєглазова В. М., Кудрєва Д. О. у фаховому періодичному виданні категорії Б “Electronics and Control Systems”.

8. Дата видачі завдання – 30 серпня 2023 року.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Здійснити огляд технічної літератури за темою роботи	01.09.2023 – 07.09.2023	Виконано
2.	Дослідити актуальність обраної теми	08.09.2023 – 14.09.2023	Виконано
3.	Ознайомитись із існуючими моделями та методами реалізації капсульних нейронних мереж	15.09.2023 – 21.09.2023	Виконано
4.	Здійснити порівняльний аналіз методів реалізації капсульних нейронних мереж	22.09.2023 – 28.09.2023	Виконано
5.	Провести експерименти над топологією капсульних нейронних мереж та проаналізувати результати	29.09.2023 – 05.10.2023	Виконано
6.	Розробити гібридний алгоритм структурно-параметричного синтезу капсульних нейронних мереж	06.10.2023 – 02.11.2023	Виконано
7.	Застосувати алгоритм, проаналізувати та порівняти результати з капсульними мережами знайденими вручну	03.11.2023 – 07.12.2023	Виконано
8.	Провести аналіз ринкових можливостей запуску стартап проєкту	08.12.2023 – 14.12.2023	Виконано
9.	Розробити концептуальні висновки	15.12.2023 – 22.12.2023	Виконано
10.	Підготувати ілюстративний матеріал	23.12.2023 – 23.12.2023	Виконано
11.	Оформити пояснювальну записку	24.12.2023 – 07.01.2024	Виконано

Студент Денис КУДРЄВ

Науковий керівник Віктор СИНЄГЛАЗОВ

АНОТАЦІЯ

Магістерська дисертація: 126 с., 14 табл., 34 рис., 18 посилань, додаток.

КОМП'ЮТЕРНИЙ ЗІР, СТРУКТУРНО-ПАРАМЕТРИЧНИЙ СИНТЕЗ, КАПСУЛЬНІ НЕЙРОННІ МЕРЕЖІ, ОБРОБКА ЗОБРАЖЕНЬ, КЛАСИФІКАЦІЯ

Цю роботу присвячено структурно-параметричному синтезу капсульних нейронних мереж. Задача структурно-параметричного синтезу полягає у знаходженні оптимальної структури та параметрів алгоритму штучного інтелекту для вирішення поставленої задачі.

Виконано дослідження сучасних методів та реалізацій капсульних нейронних мереж, проаналізовано їх переваги та недоліки, обрано найкращі підходи для вирішення задачі.

У цій роботі розроблено методологію структурно-параметрично синтезу капсульних нейронних мереж, основою якого є гібридний алгоритм. Гібридний алгоритм складається з генетичного алгоритму та градієнтного алгоритму (Adam).

У результаті використання гібридного алгоритму знайдено оптимальну топологію та параметри капсульної нейронної мережі для вирішення задачі класифікації.

Результати роботи прийнято до опублікування у фаховому періодичному виданні категорії Б.

ABSTRACT

Master’s thesis: 126 p., 14 tab., 34 fig., 18 references, appendix.

COMPUTER VISION, STRUCTURAL-PARAMETRIC SYNTHESIS, CAPSULE NEURAL NETWORKS, IMAGE PROCESSING, CLASSIFICATION

This work is dedicated to the structural-parametric synthesis of capsule neural networks. The task of structural-parametric synthesis involves finding the optimal structure and parameters of the artificial intelligence algorithm to solve the specified problem.

Research on modern methods and implementations of capsule neural networks was conducted, analyzing their advantages and disadvantages. The best approaches were chosen to address the task.

In this work, a methodology for the structural-parametric synthesis of capsule neural networks was developed, based on a hybrid algorithm. The hybrid algorithm consists of a genetic algorithm and a gradient algorithm (Adam).

As a result of using the hybrid algorithm, the optimal topology and parameters of the capsule neural network were found to solve the classification task.

The results of the work have been accepted for publication in a specialized B category periodical.

ЗМІСТ

ВСТУП

Штучний інтелект справжньою мірою став найгарячішим трендом у світі технологій і бізнесу сучасності. За останні роки відбулося стрімке розвитку цієї галузі, і вона значно вплинула на різні сфери життя, включаючи медицину, освіту, виробництво, фінанси та багато інших. Все більше і більше підприємств та бізнесів намагаються застосовувати штучний інтелект у своїх процесах та технологіях.

Однією з найширших галузей штучного інтелекту, яка зазнала великого зростання, є комп'ютерний зір – розробка систем з використанням ШІ, здатних розпізнавати аналізувати та розпізнавати візуальні дані, подібно до того, як це робить людський зір. Ця галузь займається включає у себе вирішення завдань таких як розпізнавання об'єктів, розпізнавання облич, визначення руху, сегментація зображень, визначення глибини, тощо. Для вирішення цих завдань у основному застосовуються глибокі згорткові нейронні мережі.

Згорткові нейронні мережі отримали значного розповсюдження й досягли значних успіхів у завданні обробки зображень. Їх принцип роботи натхнений роботою людської системи візуального сприйняття. Але є й деякі розбіжності у тому, як інформація про певні сутності передається між шарами згорткових мереж та тим як воно передається у нашому мозку. У 2011 році, у статті [1], дослідники виклали свої бачення напрямку, у якому повинні розвиватися ці нейромережі. У цій статті було використано термін – капсули, як ще одного рівня абстракції всередині шарів нейромережі. У 2017 році вийшла стаття [2], у якій було запропоновано алгоритм " Routing by agreement ", завдяки якому може працювати так звані капсульні мережі.

Капсульні мережі вирішують багато проблем згорткових нейронних мереж, набагато краще навчаються на обмежених за розміром наборах даних,

краще вивчають ознаки об’єктів на зображеннях. Хоча капсульні нейронні мережі все ще не випередили за своїм розвитком згорткові нейронні мережі на складних та реальних датасетах, вони вже отримують найліпші результати на таких датасетах як MNIST. Капсульні нейронні мережі є активною темою досліджень у галузі сучасного комп'ютерного зору.

У зв’язку з появою цього нового виду нейронних мереж виникає потреба у формалізації підходів й застосуванні алгоритмів для створення капсульних нейронних мереж з оптимальною архітектурою та параметрами для розв’язання задач з обробки зображень.

Метою роботи є визначення найбільш важливих гіперпараметрів й створення алгоритму знаходження оптимальної топології мережі й навчальних параметрів.

У першому розділі роботи надаються загальні відомості про стан штучного інтелекту на поточний рік та наводяться напрямки досліджень, які пов’язані саме з капсульними нейронними мережами та перспективи використання капсульних нейронних мереж.

У другому розділі наводиться топологія капсульних нейронних мереж, математично описуються деякі підходи до застосування капсул та шарів з ними. Наводяться відомості про алгоритми навчання нейронних мереж, які можуть використовуватися для капсульних нейронних мереж.

У третьому розділі описано аналіз підходів до реалізації капсульних нейронних мереж й виконується вибір архітектури, для якої буде проводитися структурно-параметричний синтез.

У четвертому розділі виконується аналіз застосування результатів роботи для початку власного стартап-проєкту.

РОЗДІЛ 1 ОГЛЯД НАПРЯМУ ШТУЧНОГО ІНТЕЛЕКТУ

1.1 Штучний інтелект – світовий тренд

Штучний інтелект — це широке поле, яке стосується використання технологій для створення машин і комп'ютерів, які мають здатність імітувати когнітивні функції, пов'язані з людським інтелектом, наприклад здатність бачити, розуміти та реагувати на усну чи письмову мову, аналізувати дані, давати рекомендації тощо.

Машинне навчання - галузь штучного інтелекту зосереджена на використанні даних і алгоритмів для імітації людського навчання, дозволяючи машинам удосконалюватися з часом, ставати дедалі точнішими під час прогнозування чи класифікації або розкривати інформацію описану даними. Воно працює трьома основними способами, починаючи з використання комбінації даних і алгоритмів для передбачення шаблонів і класифікації наборів даних, функції помилок, яка допомагає оцінити точність, а потім процесу оптимізації, щоб найкраще вписати точки даних у модель.

Машинне навчання передбачає показ великого обсягу даних машині, щоб вона могла вчитися та робити прогнози, знаходити шаблони або класифікувати дані.

В останні роки штучний інтелект отримав значного розвитку, зокрема завдяки створенню нових алгоритмів, появі доступних обчислювальних ресурсів великої продуктивності та доступу до великих обсягів даних.

Штучний інтелект широко використовується для обробки даних - у сфері розпізнавання зображень, звуків, тощо. Також нещодавно справжній буму зазнали так звані великі лінгвістичні моделі, які побудовані за архітектурою трансформер, а також інші моделі для генерування зображень.

Також штучний інтелект починає зазнавати поширення в автономних автомобілях, його використання штучного інтелекту дозволяє поліпшити технології автономних транспортних засобів, зробити їх більш безпечними для оточуючих.

Штучний інтелект використовується у фінансах, його застосовують аналізу фінансових даних та визначення ризиків, прогнозування можливих кризових ситуацій, оцінки ймовірність дефолту позичальників, а також визначення змін у ринкових трендах, що допомагає у прийнятті більш обґрунтованих фінансових рішень.

Особливого вибуху зазнали на даний момент генеративний штучний інтелект - це штучний інтелект, здатний генерувати текст, зображення або інші медіа, використовуючи генеративні моделі. Генеративні моделі ШІ вивчають закономірності та структуру своїх вхідних тренувальних даних і потім генерують нові дані, що мають подібні характеристики.

У 2014 році досягнення, такі як варіаційний автоенкодер та генеративна змагальна мережа, вперше створили практичні глибокі нейронні мережі, здатні вчитися генеративним, а не дискримінативним, моделям складних даних, таких як зображення. Ці глибокі генеративні моделі стали першими, хто міг виводити не лише класові мітки для зображень, а й створювати цілі зображення.

У 2017 році мережа трансформатор зробила стрибок в генеративних моделях у порівнянні зі старими моделями довготривалої та короткотривалої пам'яті, що призвело до створення першого генеративно навченого трансформатора, відомого як GPT-1, у 2018 році. За ним послідував GPT-2 у 2019 році, який продемонстрував можливість узагальнення без нагляду до багатьох різних завдань.

У 2021 році випуск DALL-E, генеративної моделі на основі трансформера, за якими послідували Midjourney та Stable Diffusion, позначили появу

практичного штучного інтелекту високої якості здатного генерувати візуальні зображення за допомогою введення текстових вказівок.

1.2 Класифікація штучних нейронних мереж

Штучна нейронна мережа — це обчислювальна модель, натхненна тим, як функціонують біологічні нейронні мережі у мозку людини. Це ключовий компонент машинного навчання та використовується для різних завдань, таких як розпізнавання образів, класифікація, регресія тощо.

Основним будівельним блоком штучної нейронної мережі є штучний нейрон або просто «нейрон». Нейрони організовані в шари, які зазвичай складаються з вхідного шару, одного або кількох прихованих шарів і вихідного шару. Кожне з’єднання між нейронами має пов’язану з ним вагу, і мережа вчиться регулювати ці параметри-ваги під час навчання для поліпшення результатів виконання поставлених перед мережею завдань.

Процес навчання штучної нейронної мережі включає в себе подачу вхідних даних, передачу їх через мережу, порівняння виходу мережі з очікуваним результатом, коригування тренованих параметрів на основі отриманої похибки.

Існує багато різних видів нейронних мереж. Їх можна класифікувати таким чином (рис. 1.1).

Рисунок 1.1 – Класифікація нейронних мереж

1) На основі напряму поширення інформації:
   - нейронні мережі прямого поширення; інформація рухається в одному напрямку, від вхідного рівня до вихідного рівня, без циклів або петель; це найпростіший вид нейронних мереж;
   - рекуренті нейронні мережі; ці мережі мають з'єднання, які утворюють цикли, що дає змогу їм зберігати пам'ять про попередні вхідні дані; РНМ добре підходять для завдань, де контекст або послідовна інформація є вирішальною.
2) На основі з'єднання:
   - повністю з'єднані мережі: усі нейрони одного шару з'єднані з усіма нейронами наступного рівня;
   - мережі розрідженого з'єднання; лише підмножина нейронів одного шару з'єднана з нейронами наступного шару.
3) За способом налаштування ваг:
   - з фіксованими зв'язками;

- з динамічними зв’язками.

4) За типом логіки, що використовується:
   - чіткі мережі;
   - нечіткі мережі.
5) На основі глибини:
   - неглибокі нейронні мережі; мають невелику кількість прихованих шарів; нейронні мережі прямого зв’язку часто неглибокі;
   - глибокі нейронні мережі; мають велику кількість прихованих рівнів, що дозволяє їм вивчати ієрархічні представлення даних; глибоке навчання було особливо успішним у таких завданнях, як розпізнавання зображень і мови.
6) Залежно від призначення:
   - генеративні змагальні мережі; складаються з генератора та дискримінатора, які використовуються для створення нових екземплярів даних;
   - автокодувальники: призначені для неконтрольованого навчання та стиснення даних;
   - мережі довготривалої короткочасної пам’яті; тип рекурентних нейронних мереж зі спеціалізованими комірками пам’яті для обробки довгострокових залежностей.
7) На основі архітектури:
   - мережі радіальних базисних функцій; містять радіальні базисні функції як функції активації; вони зазвичай використовуються для завдань розпізнавання образів;
   - згорткові нейронні мережі; у цих мережах використовуються згортки або фільтри для обробки вхідних даних; ці фільтри ковзають по вхідному зображенню або іншому типу даних, виконуючи операції

згортки, що дозволяє виявляти різноманітні функціональні особливості (границі, текстури тощо);

- капсульні нейронні мережі; у цих мережах використовуються капсульні шари для у поєданні з операцією згортки для визначення ознак на зображенні; капсульні нейронні мережі набагато краще закодовують у собі властивості зображень, їм необхідно набагато менше даних для вивчення ознак та закономірностей між ними, у цьому плані вони є доволі перспективними.

1.3 Класифікація методів машинного навчання

Зараз розповсюджено 4 методи машинного навчання:

- навчання з учителем;
- навчання без учителя;
- навчання з підкріпленням;
- напівкероване навчання.

1.3.1 Навчання з учителем

Цей тип машинного навчання подає історичні вхідні та вихідні дані в алгоритми машинного навчання з обробкою між кожною парою вводу/виводу, що дозволяє алгоритму зміщувати модель для створення виходів, якомога точніше узгоджених із бажаним результатом. Загальні алгоритми, які

використовуються під час навчання з вчителем, включають нейронні мережі, дерева рішень, лінійну регресію та опорні векторні машини.

1.3.2 Навчання без учителя

У той час як контрольоване навчання вимагає від користувачів допомоги машині в навчанні, неконтрольоване навчання не використовує ті самі позначені навчальні набори та дані. Замість цього машина шукає менш очевидні шаблони в даних. Цей тип машинного навчання дуже корисний, коли вам потрібно визначити шаблони та використовувати дані для прийняття рішень. Загальні алгоритми, що використовуються в неконтрольованому навчанні, включають приховані марковські моделі, k-середні, ієрархічну кластеризацію та моделі суміші Гауса.

1.3.3 Навчання з підкріпленням

Навчання з підкріпленням – це тип машинного навчання, який найближче до того, як навчаються люди. Алгоритм або агент, який використовується, навчається під час взаємодії зі своїм середовищем отримуючи позитивну або негативну винагороду. Загальні алгоритми включають часову різницю, глибокі змагальні мережі та Q-навчання.

1.3.4 Напівкероване навчання

Напівкероване навчання це підход до машинного навчання, який поєднує у собі елементи навчання з учителем і навчання без учителя. У звичайному навчанні з учителем для побудови моделі потрібна велика кількість промаркованих даних, де кожний приклад має вхідні дані та відповідні мітки. З іншого боку, в навчанні без учителя дані можуть не мають міток і використовуються для пошуку прихованих структур або кластеризації.

У напівкерованому навчанні ми використовуємо обидва типи даних - дані з мітками та без міток. Зазвичай маємо обмежену кількість промаркованих прикладів і велику кількість немаркованих прикладів. За допомогою напівкерованого навчання ми можемо скористатись інформацією з обох видів даних для покращення роботи моделі.

Один з підходів напівкерованого навчання - це використання навчання без учителя для створення внутрішнього представлення даних та використання цього представлення для класифікації маркованих даних. Таким чином, модель може використовувати знання, отримані з немаркованих даних, для уточнення і покращення своєї роботи з маркованими даними.

Напівкероване навчання є корисним у випадках, коли отримання маркованих даних може бути витратним або складним процесом, але є велика кількість немаркованих даних, які можуть бути використані для покращення результатів моделі. Воно також може допомогти уникнути проблеми перенавчання, коли модель дуже точно пристосовується до маркованих даних, але показує погані результати на нових, невідомих прикладах.

1.4 Огляд літератури присвяченої капсульним нейронним мережам

Капсульні нейронні мережі почали швидко розвиватися відносно недавно. Хоча ідея використовувати капсули у нейронній мережі існувала вже давно, у 2011 році вийшла стаття [1], у якій було запропоновано капсули як напрям досліджень, лише у 2017 році вийшла робота [2], у якій був запропонований алгоритм “маршрутизація за згодою” (англ. “Routing by agreement”), завдяки якому дана архітектура може використовуватися, мережа отримала назву CapsNet.

Дослідники у статті [3] запропонували інший варіант активаційної функції капсул, яка використовується в оригінальній CapsNet, також вони експериментували з модифікацією топології оригінальної мережі. У статті [4] надали формальний опис оригінального динамічного маршрутизаційного підходу як задачі оптимізації, що мінімізує втрату кластеризації, та запропонували слабко модифіковану версію. У статті [5] висунули концепцію групових капсульних мереж, стверджуючи, що вони зберігають еквіваріантність для позиції виводу та інваріантність для активацій. Ті ж автори оригінального CapsNet у статті [6] адаптували алгоритм очікування-максимізації для кластеризації схожих голосів капсул при маршрутизації. Спектральні капсульні мережі зі статті [7] були засновані на цій останній роботі і модифікували маршрутизацію, ґрунтуючись на декомпозиції до одиничного значення голосів капсул з попередніх шарів. У статті [8] запропонували маршрутизацію, створену на основі варіаційних Баєсівських методів для тренування гауссівської змішаної моделі. Дослідники у статті [9] зосередилися на забезпеченні стійкості капсульних мереж до афінних трансформацій, розділяючи матриці трансформацій між усіма капсулами низького рівня та кожними капсулами високого рівня. У статті [10] дослідники поклали під сумнів ефективність

дотеперішнього алгоритму маршрутизації, стверджуючи, що кращі результати можна отримати взагалі без маршрутизації. З іншого боку, дослідники у статті [11] довели, що механізм "маршрутизації за згодою" є необхідним для забезпечення композиційних структур мереж на основі капсул. Тим не менш, у статті [12] запропонували нову архітектуру на основі варіації оригінальної ідеї капсул, яку вони назвали капсули гомогенного фільтра без маршрутизації між шарами.

Механізм уваги дозволяє динамічно надавати більше важливості конкретним ознакам, які вважаються більш значущими для вирішення конкретної проблеми. Така ідея здобула велику популярність в ряді застосувань глибокого навчання і була реалізована в обробці природної мови та комп'ютерного зору. У статті [13] дослідники застосували механізм уваги до маршрутизації капсул із функцією прямого поширення без ітерацій. Однак вони вибирали капсули низького рівня, множачи їх активації на вектор параметрів, який був вивчений за допомогою зворотного поширення, і не вимірювали узгодженість. Таким чином, оригінальна ідея "маршрутизації за згодою" була дуже викривлена. У статті [14] дослідники незначно змінили оригінальну динамічну маршрутизацію для обчислення узгодження між позою капсули високого рівня та голосами капсул низького рівня за допомогою механізму оберненого скалярного добутку. Вони запропонували паралельну ітеративну маршрутизацію замість послідовної, виконуючи процедуру маршрутизації одночасно на всіх шарах капсул. У статті [15] було застосовано капсули разом із базовим блоком самоуваги для задачі взаємин між сутностями в обробці природної мови.

У статті [16] дослідники об’єднали наробки попередників й застосували механізм самоуваги для капсул, щоби виконувати маршрутизацію інформації до капсул більш високого рівня й отримали легку архітектуру, яка має малу кількість параметрів, що налаштовуються (160К).

Тим не менш, на даний момент, немає жодних досліджень, які би були сфокусовані на знаходженні оптимальних топологій та параметрів капсульних нейронних мереж за допомогою просунутих методів, таких як генетичний алгорти. У статті [3] була виконана лише невелика серія експериментів, яка не призвела до значних поліпшень результатів.

1.5 Перспектива використання капсульних нейронних мереж

Ідея капсульних мереж досить інтуітивна і спрямована на вирішення проблем згорткових нейронних мереж, в першу чергу втрату важливої інформації між шарами. Результат роботи капсули має векторну або матричну форму й дозволяє передавати згруповані важливі ознаки у наступний шар. Тим не менш, капсульні нейронні мережі мають на даний момент певні недоліки, в першу чергу – значне уповільнення навчання через використання операційно-доровартісних алгоритмів.

Недоліки згорткових нейронних мереж за Джефрі Хінтоном [1] представлено нижче.

1) Згорткові мережі ієрархічно занадто прості – нейрони, шари. Інтуітивно, людський мозок має більше складних структур, які використовуються для розпізнавання зображень, а отже нейромережі теж повинні їх мати.
2) Згорткові мережі втрачають багато інформації під час використання операції максимального об’єднання (max pooling) – де в результаті зниження вимірності у наступний шар переходить лише одне найбільш активне значення активації. Уся інша інформація втрачається. Як каже Хінтон: “Операція об’єднання, яка використовується в згорткових

нейронних мережах, є великою помилкою, і той факт, що вона працює так добре, є катастрофою".

3) Згорткові мережі не вважають на "позу" об'єктів, які досліджують – трансформація та траспозиція об'єкта. Для згорткової мережі трохи нахилена на 45 градусів цифра буде здаватися мережі чужою. Це призводить до необхідності використання великої кількості навчальних даних.
4) Ще один результат неврахування пози – нейронна мережа не зважає на те яким чином розташовані відносно одне одного елементи зображення, коли виконує перехід від менш складних елементів до більш складних.

Капсульні нейронні мережі вирішують вище названі проблеми згорткових нейронних мереж.

Зазначимо дещо про комп'ютерну графіку – вона займається побудовою візуального зображення з деякого внутрішнього ієрархічного представлення геометричних даних. Зверніть увагу, що структура цього представлення повинна враховувати взаємне розташування об'єктів. Це внутрішнє представлення зберігається в пам'яті комп'ютера у вигляді масивів геометричних об'єктів і матриць, які представляють відносне положення та орієнтацію цих об'єктів. Потім спеціальне програмне забезпечення бере це представлення та перетворює його на зображення на екрані. Це називається рендерингом.

Натхненний цією ідеєю, Хінтон стверджує, що насправді мозок виконує дії, протилежні рендерингу. Він називає це оберненою графікою: із візуальної інформації, що отримали очі, деконструюється ієрархічне уявлення про навколишній світ і виконується спроба зіставити його з уже вивченими шаблонами та зв'язками, що зберігаються у мозку. Так відбувається розпізнавання. І ключова ідея полягає в тому, що представлення об'єктів у мозку не залежить від кута зору.

Хінтон стверджує, що для правильної класифікації та розпізнавання об’єктів важливо зберегти ієрархічні зв’язки з урахуванням пози між частинами об’єкта. Саме це і повинні виконувати капсули, які мають на виході ймовірність присутності певного об’єкта (активацію) і його позу.

Капсула — це група нейронів, вихідні дані яких представляють різні властивості однієї сутності. Замість того, щоби використовувати операцію об’єднання, яка втрачає багато інформації, ми можемо використовувати капсули, для того щоби не лише показувати ймовірність існування певної ознаки, а й її характеристики, які нейронна мережа вчиться розпізнавати. Кожен шар капсульної мережі містить багато капсул. Активності нейронів у активній капсулі відображають різні властивості конкретної сутності, яка присутня на зображенні. Ці властивості можуть включати в себе багато різних типів параметрів інстанціювання, таких як поза, деформація, швидкість, альбедо, відтінок, текстура і т.д. Однією дуже особливою властивістю є існування інстанційованої сутності на зображенні. Очевидним способом представлення існування є використання окремого логістичного блоку, виходом якого є ймовірність того, що сутність існує. Але ймовірність існування сутності також можна визначати як довжину вектора властивостей.

Активні капсули на одному рівні визначають, за допомогою матриць трансформації, параметри інстанціювання капсул на рівні вище.

Коефіцієнти впливу капсули більш нижнього рівня на капсулу наступного ітеративно оновлюються для кожного зображення за допомогою алгоритму “Routing by agreement” таким чином, що вихідні дані кожної капсули направляються до тої капсули, де її вихідне значення утворює скупчення з виходами інших капсул.

Тобто, у капсулу, яка займається обробкою певних елементів зображення, надаються дані з капсул нижнього рівня, якщо багато капсул мають схожу позу та активацію (очі, губи, ніс розташовані так, щоби утворювати обличчя).

Таким чином, капсульні нейронні мережі на відміну від згорткових нейронних мереж мають такі особливості.

1) Не використовують операцію об'єднання, що не призводить до втрачі інформації
2) Мають векторну репрезентацію ознак, а не скалярну
3) Зважають на властивості ознак, що зменшує кількість необхідність даних для тренування – мережа здатна впізнавати ознаки, якщо вони мають трохи змінену вигляд.
4) Формує уявлення про існування ознаки більш високого рівня враховуючи взаємне розташування ознак низького рівня.
5) Дозволяє використовувати “Routing by agreement” алгоритми між капсулами різних рівнів, що дозволяє більш ефективно передавати дані.

Тополологічно це досягається за допомогою використання двох нових видів шарів у порівннянні зі згортковими нейронними мережами:

- капсульний згортковий шар (у статті [2] відомий як PrimaryCaps), який виконує операцію згортки як і згортковий шар у згортковій нейронній мережі, але змінює виміри вихідних карт ознак, щоби відображати кількість капсул у шарі і довжину їх виходу;
- капсульний шар (у статті [2] відомий як DigitCaps), який замінює шари об’єднання у згортковій нейронній мережі й визначає існування та властивості ознак більш високого рівня.

Висновки до розділу 1

У цьому розділі було виконано загальний огляд стану штучного інтелекту на поточний день, проведено аналіз джерел пов’язаних з капсульними нейронними мережами, зазначено актуальність робіт пов’язаних з капсульними нейронними мережами та новітність роботи по виконанню структурно параметричного синтезу цих мереж.

РОЗДІЛ 2 КАПСУЛЬНІ НЕЙРОННІ МЕРЕЖІ. ТОПОЛОГІЯ ТА МЕТОДИ НАВЧАННЯ

2.1 Топологія капсульних нейронних мереж

Капсульні нейронні мережі мають схожу топологію й на згорткові нейромережі, але замість згорткових шарів використовуються капсульні згорткові шари, а замість шарів об’єднання капсульні шари.

Першим шаром нейромережі тим не менш все одно може виступати звичайний згортковий шар. Таким чином, у капсульній нейронній мережі можуть застосовуватися 3 види шарів:

- згорткові шари;
- згорткові капсульні шари;
- капсульні шари.

Згорткові шари як і у згорткових нейронних мережах застосовують операцію згортки, обробляють вхідні дані для отримання карт ознак, які можуть використовуватися для визначення існування певних ознак (рис. 2.1).

Згорткові капсульні шари аналогічні згортковим шарам, але операції згортки у цьому шарі виконуються стільки разів, скільки є капсул. Також вони використовують додаткову нелінійну векторну функцію активації на отриманих картах ознак.

Капсульні шари виконують об’єднання ознак і їх параметрів з попереднього шару і повертають параметри та ймовірність існування ознаки більш високого рівня.

Розглянемо топологію капсульної нейронної мережі зі статті [2]. Вона має основну частину нейронної мережі, яка складається зі згорткового, капсульного згорткового та капсульного шару й генератора, який повинен перетворювати параметри виходу правильної капсули у останньому шарі на початкове

зображення. Хоча капсульні нейронні мережі використовуються для вирішення задачі класифікації, так як вихідним значенням капсули є вектор, у якому закодовано властивості ознаки або об’єкту, то можна застосовувати нейромережу генератор, як продовження капсульної нейронної мережі. Генератор повинен перетворити цей вектор повторно на вхідне зображення. Він використовується у вигляді регуляризатора, для покращення точності та запобігання перенавчанню капсульної нейромережі на тренувальних даних. Як приємний бонус, маніпуляція елементів вхідного вектора до генератора дозволяє візуалізувати, які саме властивості були закодовані у ці параметри.

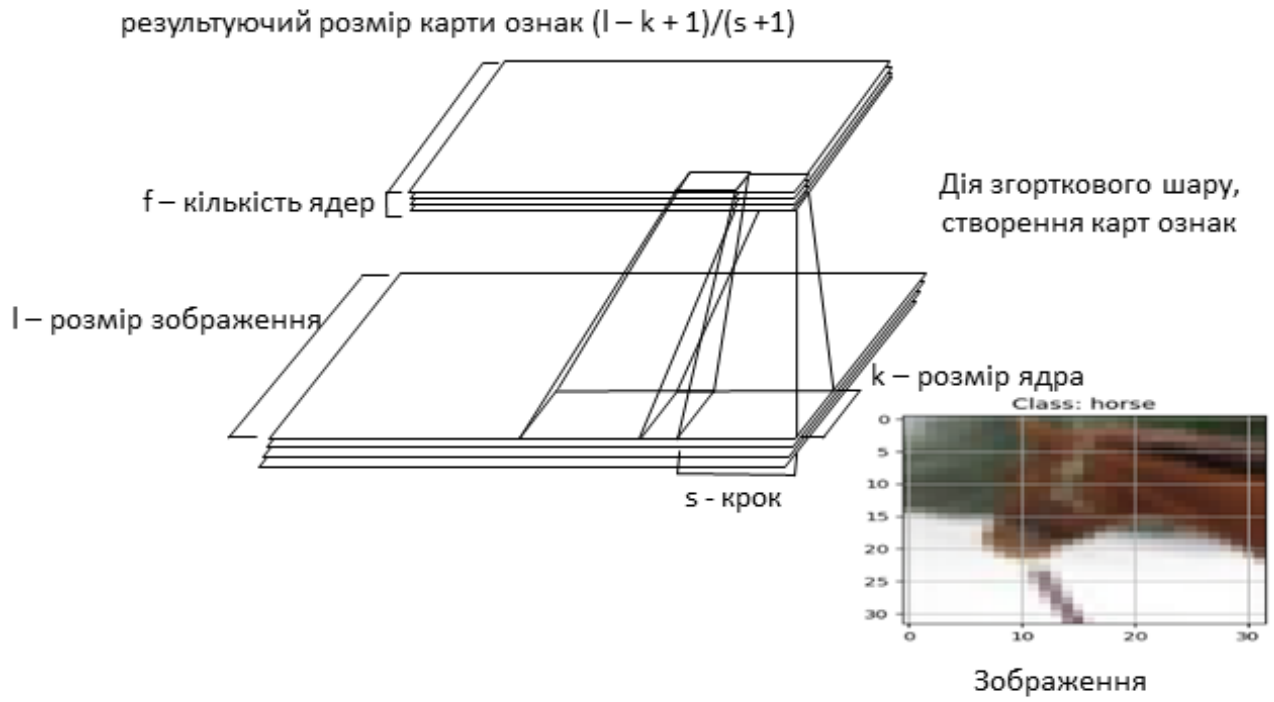

Рисунок 2.1 – Дія згорткового шару

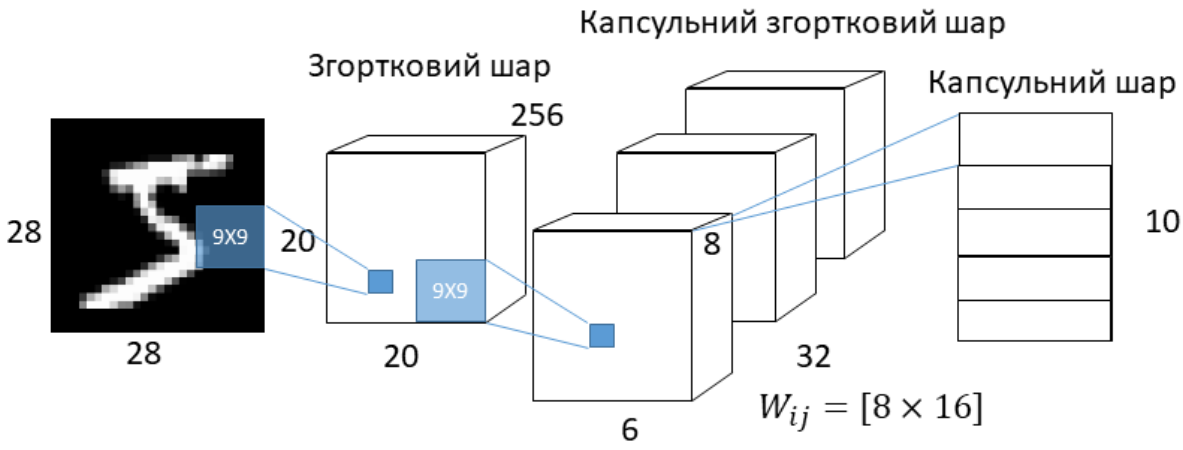

Рисунок 2.2 – Архітектура капсульної нейромережі CapsNet

Маємо такі шари в основній частині як представлено нижче.

1) Згортковий шар: 256 фільтрів, розмірність ядра 9x9x1, крок 1 та активація ReLU. Цей шар має (9*9*1 + 1)*256 = 20992 параметрів, що налаштовуються. Отримує зображення 28x28x1 на вхід, виводить карту ознак розмірністю 20x20x256.
2) Капсульний згортковий шар: цей згортковий шар має у собі 32 капсули, кожна з яких виконує згортку ядром 9x9 кількість фільтрів 8 з кроком 2 на вхідних даних розмірності 20x20x256 й повертає тензор розмірності 6x6x8. Так як таких капсуль 32, то розмірність виходу становить 6x6x8x32. Має 5 308 672 параметрів, що налаштовуються.
3) Капсульний шар: у цьому шарі присутні 10 капсул, одна на кожну цифру датасету MNIST. Кожна капсула приймає тензор розмірності 6x6x8x32, які інакше можна представити як 6x6x32 векторів довжини 8, або 1152 вектори. Кожний з цих векторів має свою вагову матрицю розмірності 8x16, а також свій ваговий коефіцієнт c і b, які застосовуються у алгоритмі “Routing by agreeement”. Таким чином

маємо 1152 x 8 x 16 + 1152 + 1152 параметрів, що налаштовуються, на кожну капсулу, яких загалом 10 – 149760.

Розглянемо тепер декодувальник – він отримує 10 векторів довжини 16 з капсульного шару, з яких лише вектор капсули правильного класу має ненульові значення, й намагається відтворити оригінальне зображення. Він виконує роль регуляризатора – змушує капсули вивчати саме ті ознаки, які корисні для реконструкції оригінального зображення.

Декодувальник складається з трьох повнозв'язних шарів:

- 512 нейронів з функцією активації ReLU – кількість тренованих параметрів (16 * 10 + 1) * 512 = 82432;
- 1024 нейрони з функцією активації ReLU – кількість тренованих параметрів (512 + 1) * 1024 = 525312;
- 784 нейрони (28x28 пікселів) з сигмоїдальною функцією активації – кількість тренованих параметрів (1024 + 1) * 784 = 803600.

Загальна кількість параметрів, що налаштовуються – 8238608.

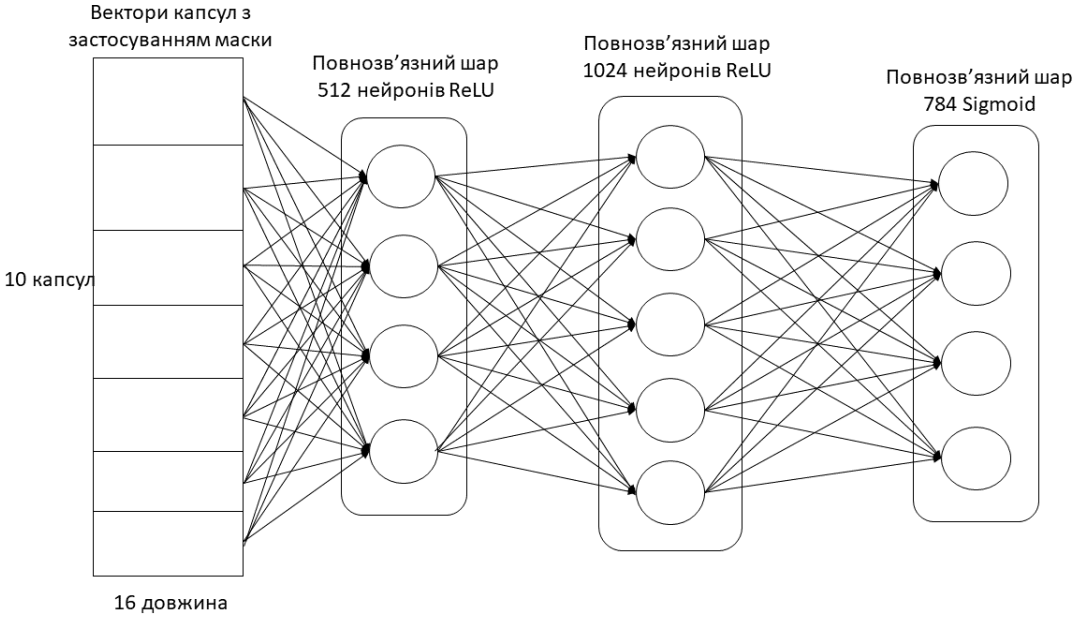

Рисунок 2.3 – Архітектура капсульної нейромережі CapsNet. Частина декодувальник

2.2 Математичні моделі шарів капсульних нейронних мереж

2.2.1 Математична модель нейронних мереж прямого поширення

Нейронна мережа прямого поширення складається з шарів, які у свою чергу складаються з нейронів, які мають зв’язки одне між одним. Кожен нейрон отримує на свій вхід сигнали нейронів з попереднього шару, які зважуються параметрами ваг, що налаштовуються, і до яких також додається параметр зміщення, який теж налаштовується. Результуючий сигнал проходить через нелінійну функції активації, яка стискає його до інтервалу від 0 до 1 та прибирає лінійність, до дозволяє моделювати більш складні відносини між вхідними даним та виходом нейрона.

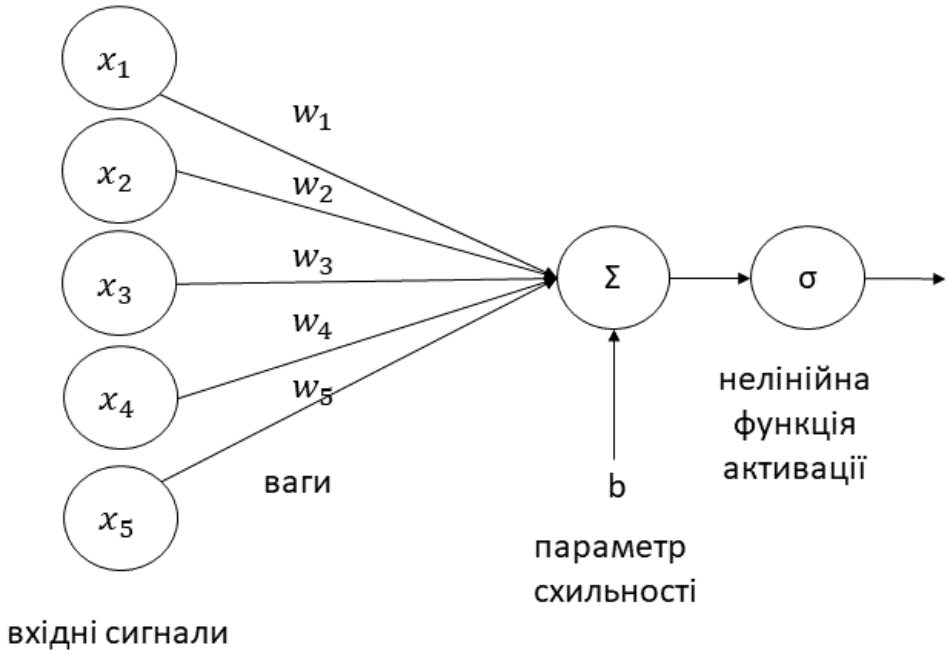

Рисунок 2.4 – Модель нейрону

$$y = \sigma(\sum_i x_i w_i + b), \quad (2.1)$$

де x_i – вхідні сигнали до нейрону, w_i – вагові коефіцієнти вхідних сигналів, b – параметр схильності нейрона.

Якщо ми розглядаємо математичну модель шару нейронної мережі загалом, то тоді вхідні сигнали до шару та параметри схилення нейронів приймають вигляд векторів, вагові коефіцієнти – вигляд матриці.

Маємо таке рівняння:

$$y = \sigma(Wx + b). \quad (2.2)$$

Таким чином, шар нейронної мережі виконує афінне перетворення вхідного вектору з додаванням нелінійності.

2.2.2 Згорткові нейронні мережі

Згорткові нейронні мережі отримали широке поширення у задачі розпізнавання зображень та аудіо. У цих видів даних є такі властивості [4]:

- вони зберігаються як багатовимірні масиви;
- вони мають одну або кілька осей, порядок яких має значення (наприклад, ширина та осі висоти для зображення, вісь часу для звукового кліпу);
- одна вісь, яка називається віссю каналу, використовується для доступа до різних видів дані (наприклад, червоний, зелений і синій канали кольорового зображення або лівий і праві канали стереозвукової доріжки).

Ці властивості жодним чином не приймаються до уваги при застосуванні афінного перетворення, фактично всі осі обробляються однаково, топологічна інформація і порядок даних не враховується. Для того щоби використати неявну структуру даних використовують операцію дискретної згортки. У згортковій нейронній мережі застосовуються 3 види шарів.

Згортковий шар – у цьому шарі, за допомогою операції згортки нейронна мережа може визначати такі частини зображення як краї та зміни кольору [17].

Операція згортки використовує ядро згортки K, яке проходиться по вхідним даним (які також у контексті згорткових нейронних мереж називаються картами ознак) й повертає багатовимірний вектор (результуючу карту ознак), який складається зі скалярних добутків ядра та елементів входу. Ядро є багатовимірним вектором певної розмірності та має певні характеристики. Такі властивості впливають на вихідний розмір o_j згорткового шару вздовж осі j:

- i_j - вхідний розмір уздовж осі j;
- k_j – розмір ядра вздовж осі j;
- s_j - крок (відстань між двома послідовними положеннями ядра) уздовж осі j;
- p_j - заповнення нуля (кількість нулів, об’єднаних на початку та в кінці осі) вздовж осі j.

Приклад операції згортки показано на рис. 2.5.

Операція згортки може бути представлена у вигляді операції матричного множення, з використанням розрідженої матриці.

Наприклад, для операції згортки з використанням 3×3 ядра на вході 4×4, де i = 4, k = 3, s = 1 та p = 0, матриця буде мати такий вигляд:

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}.$$

Також ми все ще використовуємо параметр схильності.

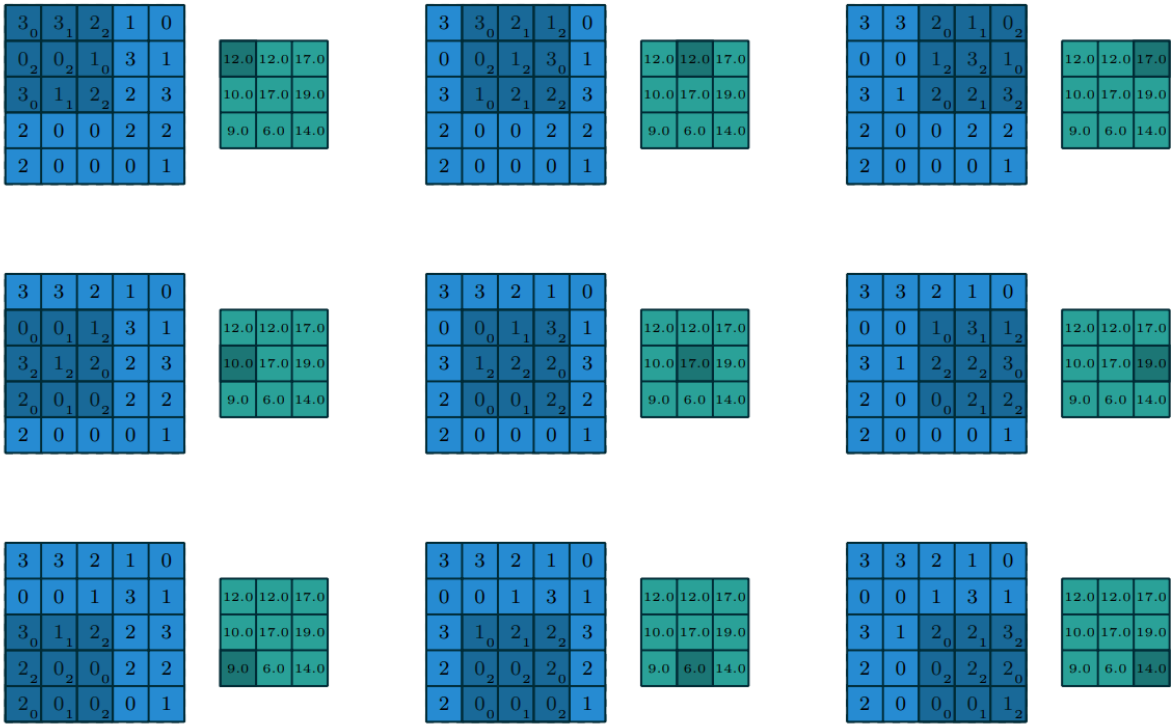

Рисунок 2.5 – Приклад операції згортки на 2D вході з 2D ядром $i_1 = i_2 = 5$, $k_1 = k_1 = 3$, $s_1 = s_2 = 1$ і $p_1 = p_2 = 0$ [17]

Враховуючи цей факт, то вихід згорткового шару, можна записати наступним чином:

$$y = \sigma(Wx + b), \tag{2.3}$$

де матриця W залежить від параметрів ядра згортки, яке використовувалося для її формування.

Шари об’єднання використовуються для зменшення розмірності у нейронній мережі. Операції об’єднання зменшують розмір карт ознак за допомогою певної функції для підсумовування підрегіонів, наприклад взяття середнього або максимального значення (найбільш поширеною функцією на даний момент є саме взяття максимального значення у регіоні).

Об’єднання працює шляхом ковзання вікна по входу та подачі вмісту вікна до функції об’єднання. У певному сенсі об’єднання працює дуже добре як дискретна згортка, але замінює лінійну комбінацію, описану ядро з іншою функцією.

Наступні властивості впливають на вихідний розмір o_j шару об’єднання вісь j:

- i_j - вхідний розмір уздовж осі j;
- k_j - розмір вікна об’єднання вздовж осі j;
- s_j - крок (відстань між двома послідовними положеннями вікна об’єднання) уздовж осі j.

Приклад операції субдискретизації наведено на рис. 2.6.

Шар об’єднання не має параметрів, що налаштовуються.

Повністю поєднані шари в кінці використовують інформацію з попередніх шарів для того щоби вже провести класифікацію за наявними ознаками, вони є звичайними шарами нейронної мережі, які вже були описані у розділі 2.1.

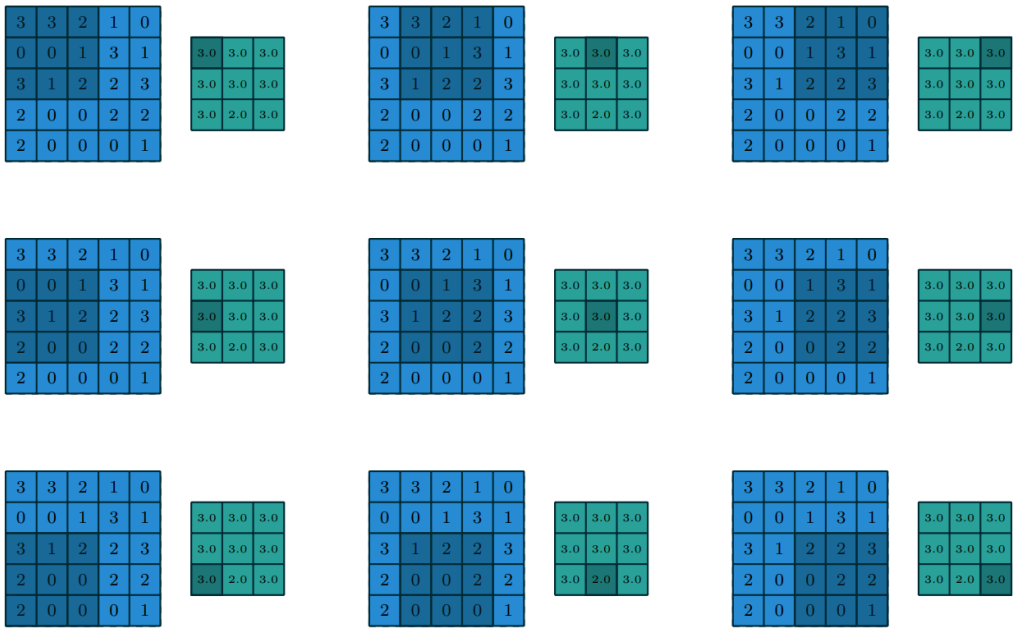

Рисунок 2.6 – Приклад операції об’єднання з функцією максимізації на вхідній матриці 5×5 ядром 3×3 з кроком 1×1 [17]

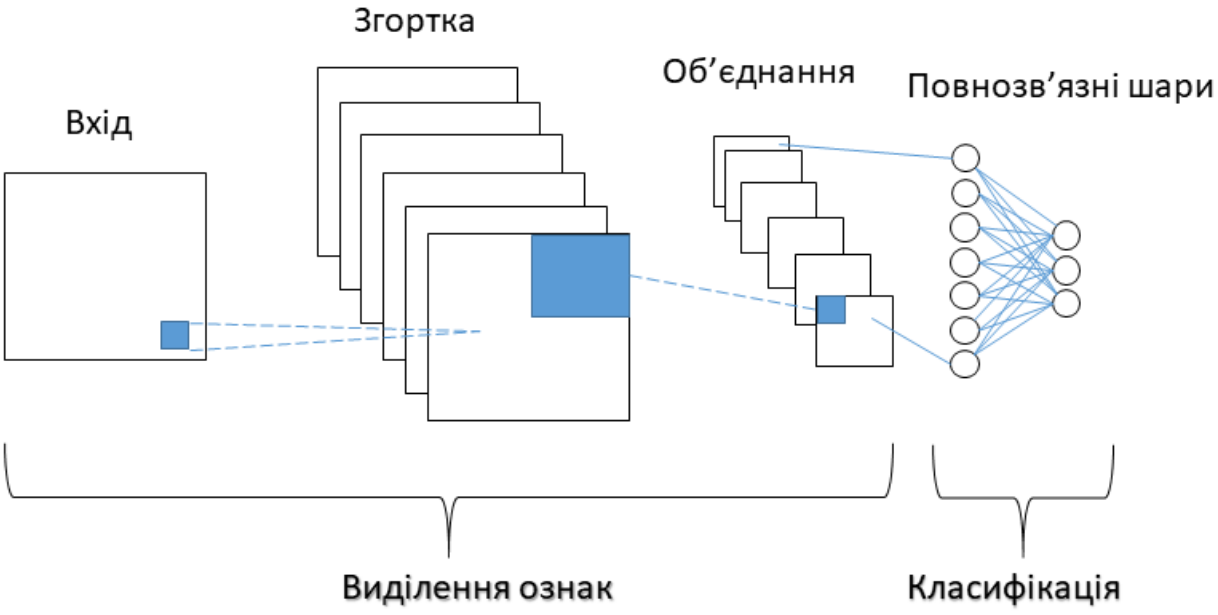

Рисунок 2.7 – Архітектура згорткової нейронної мережі високого рівня

2.2.3 Капсульні нейронні мережі

Шари властиві капсульним нейронним мережам - капсульні нейронні мережі застосовують 2 нових види шарів у порівнянні зі згортковими нейронними мережами – капсульні шари й згорткові капсульні шари.

Капсульні шари - шари, які замінюють шари з операцією об’єднання, вони отримують карти ознак й виводять представлення властивостей ознаки більшого рівня у вигляді вектори, де його довжина – ймовірність, що ця ознака існує.

Розглянемо математичну модель капсульного шару:

На вхід капсули j у шарі l + 1 подаються вектори u_i з шару l. Вони множаться на матрицю ваг W_{ij}, яка виконує трансформацію даних.

$$\hat{u}_{j|i} = W_{ij} u_i. \tag{2.4}$$

Після цього ми обчислюємо зважену суму всіх вхідних векторів

$$s_j = \sum_i c_{ij} \hat{u}_{j|i}, \tag{2.5}$$

де c_{ij} – ваговий коефіцієнт. Чим більше вхідний вектор “погоджується” з вихідним вектором, тим більше значення c_{ij}, яке лежить в інтервалі [0, 1].

Є декілька алгоритмів за якими можна вираховувати ці коефіцієнти, вони будуть розглянути трохи далі.

Насамкінець, застосовується нелінійна функція активації, так зване “стиснення” (squashing) – це було однією з новацій у статті [1], тому що функція активації має векторну форму, а не скалярну як у класичних нейромережах. Після її застосування маємо вихідний вектор капсули.

$$v_j = \frac{\|s_j\|^2}{1+\|s_j\|^2} \frac{s_j}{\|s_j\|^2}. \quad (2.6)$$

Для одновимірного вектору, графік функції активації має вигляд як на рис. 2.8.

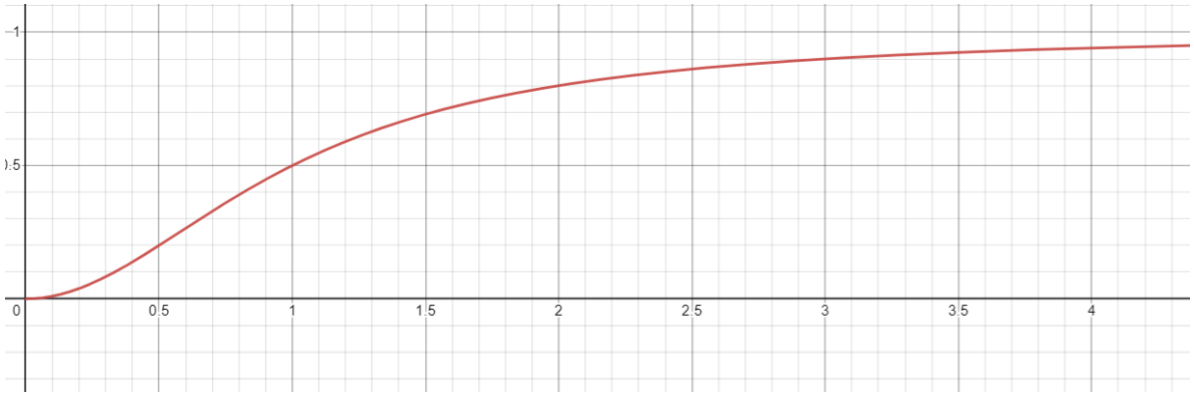

Рисунок 2.8 – Графік функції активації для одновимірного вхідного вектору

Згорткові капсульні шари – шари, які є дуже схожими на звичайні згорткові шари. У шарі виконується операція згортки, з кількістю фільтрів, яка дорівнює добутку кількості капсул на їх розмірність. Після отримання карт ознак, додатково застосовується операція “стиснення” для додавання додаткової блочної нелінійності.

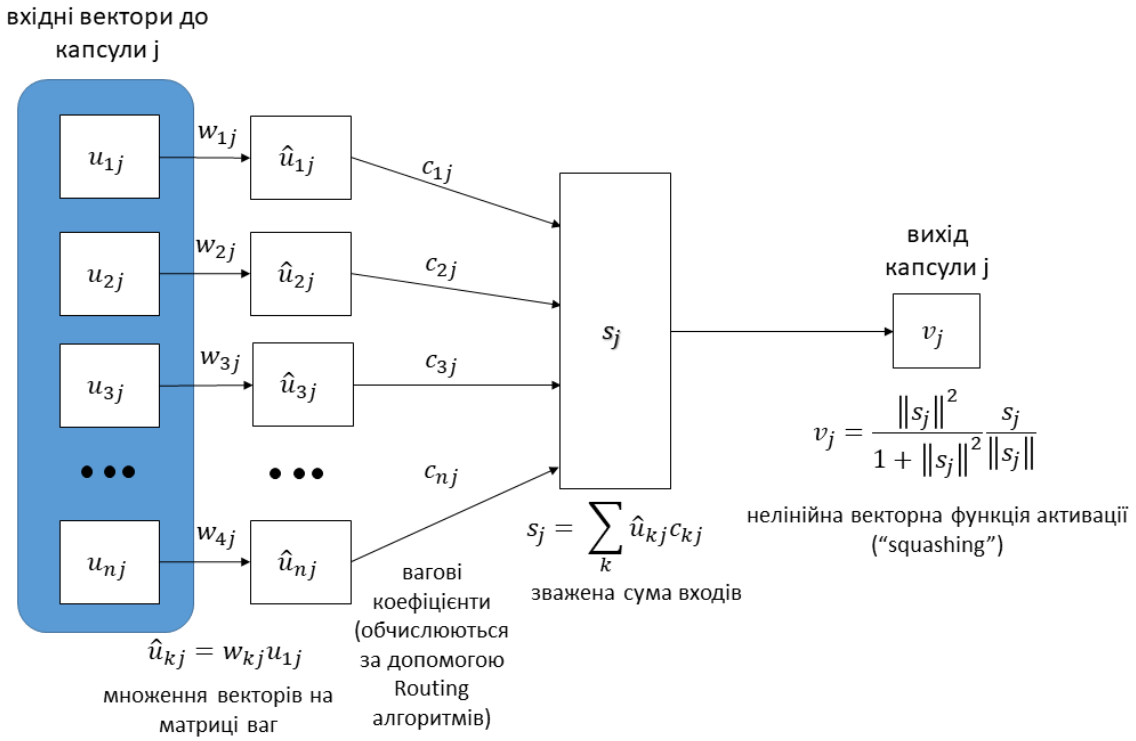

Рисунок 2.9 – Діаграма математичної моделі капсули у капсульному шарі

Розглянемо математичну модель згорткового капсульного шару - математично згортковий капсульний шар еквівалентний до згорткового шару, але з більшою кількістю застосування операцій згортки, по одній операції згортки на капсулу. Також у капсульному згортковому шарі додатково застосування функції стискання по вимірам результуючих ознак.

$$y_i = \sigma(W_i x + b_i), \tag{2.7}$$

де матриця W залежить від параметрів ядра згортки, яке використовувалося для її формування, i = 1…k, k = m * n, де m – кількість капсул, n – розмірність вектора виходу капсули або кількість ядер згортки, які використовуються у капсулі.

Виконується об'єднання усіх результуючих карт ознак й таким чином чи маємо на виході шару вектор розмірності

$$(H, W, m, n), \tag{2.8}$$

де H і W – вимірність карт ознак виходу капсули з урахуванням властивостей ядра згортки та фільтрів. Після цього використовується операція стиснення на останньому вимірі.

2.2.4 Маршрутизація за згодою (Routing by agreement)

У статті [2] був продемонстрований алгоритм, який давав змогу обчислювати коефіцієнти міжкапсульних ваг. Алгоритм має фіксовану ітеративну структуру.

Procedure 1 Routing algorithm.

1: **procedure** ROUTING($\hat{\boldsymbol{u}}_{j|i}, r, l$)
2: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
3: **for** r iterations **do**
4: for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$ ▷ обчислюється рівнянням (2.9)
5: for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij} \hat{\mathbf{u}}_{j|i}$
6: for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$ ▷ обчислюється рівнянням (2.6)
7: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
return $\mathbf{v}_j$

Рисунок 2.10 – Псевдокод алгоритму маршрутизації за згодою [2]

У статті зазначається, що рекомендованою кількістю ітерацій є 3, використання більшої кількості призводить до значного перенавчення [2].

Алгоритм зводиться до обчислення вихідного вектору капсули наступного рівня й модифікації міжкапсульних коефіцієнтів ваг шарів l та l + 1, щоби більше

значення мали ті шляхи між капсулами, капсули яких мають той самий напрям у шарі l, що й напрям вектору виходу капсули у шарі l + 1.

Під операцією softmax(b_i) мається на увазі наступна операція:

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}. \quad (2.9)$$

2.2.5 Маршрутизація на основі самоуваги (Self-Attention Routing)

У капсульних нейронних мережах, які використовують маршрутизацію на основі самоуваги алгоритм маршрутизації не є ітеративним. Оптимальні зв’язки між капсулами різних рівнів вбудовані у вагові матриці W та B. На рис. 2.11, показано, що незважаючи на додатковий вимір, загальна архітектура дуже схожа на повністю з'єднану мережу з додатковим відгалуженням, внесеним механізмом самоуваги. Зазначимо, що загальний вхід капсули на рівні вище s_n^{l+1} є зваженою сумою всіх "векторів передбачення" від капсул u_n^l на рівні нижче. Це досягається матричним множенням кожної капсули, u_n^l, яка належить до вагової матриці $U_{n,d}^l$. Інтуїтивно, весь тензор $W_{(n^l,n^{l+1},d^l,d^{l+1})}^l$, який містить всі вагові матриці, вбирає всі афінні перетворення між капсулами двох сусідніх рівнів.

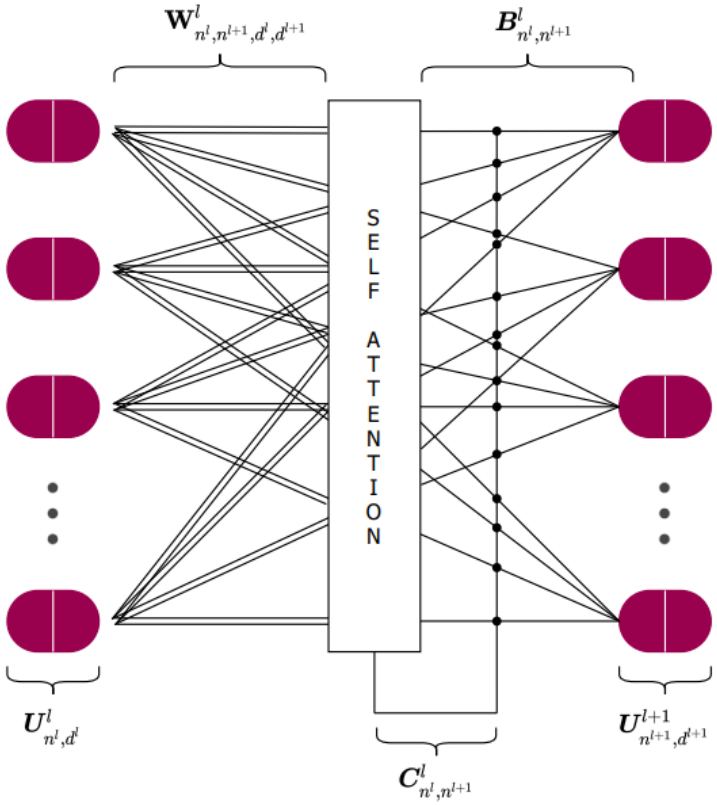

Рисунок 2.11 – Діаграма математичної моделі зв’язку між капсульними шарами з використанням маршрутизації з самоувагою [16]

Це досягається матричним множенням кожної капсули, u_n^l, яка належить до вагової матриці $U_{n,d}^l$. Інтуїтивно, весь тензор $W_{(n^l,n^{l+1},d^l,d^{l+1})}^l$, який містить всі вагові матриці, вбирає всі афінні перетворення між капсулами двох сусідніх рівнів. Таким чином, кожна капсула рівня l, щоби отримати передбачення для рівня вище, виконує наступне рівняння

$$\widehat{U}_{(n^l,n^{l+1},:)}^l = u_n^{Tl} \times W_{(n^l,n^{l+1},:,:)}^l, \tag{2.10}$$

де $\widehat{U}^{l}_{(n^l,n^{l+1},:)}$ містить усі передбачення l-тих капсул, дійсно кожна n^l капсула, за допомогою матриці ваг, передбачує властивості всіх капсул n^{l+1}. Капсули наступного рівня s_n^{l+1} обчислюються наступним чином:

$$s_n^{l+1} = \widehat{U}^{Tl}_{(:,n^{l+1},:)} \times (C^{l}_{(:,n^{l+1})} + B^{l}_{(:,n^{l+1})}), \tag{2.11}$$

де $B^{l}_{(n^l,n^{l+1})}$ – додаткова матриця ваг. З іншого боку $C^{l}_{(n^l,n^{l+1})}$ це матриця, яка містить в собі всі коефіцієнти зваження капсул створених алгоритмом самоуваги.

Коефіцієнти зваження можуть бути обчислені наступним чином. Тензор самоуваги $A^{l}_{(n^l,n^l,n^{l+1})}$

$$A^{l}_{(:,:,n^{l+1})} = \frac{\widehat{U}^{l}_{(:,n^{l+1},:)} \times \widehat{U}^{Tl}_{(:,n^{l+1},:)}}{\sqrt{d^l}}, \tag{2.12}$$

де d^l- розмір капсули у шарі l

Фінальні коефіцієнти, обчислюються за допомогою операції експоненційної нормалізації.

$$C^{l}_{(:,n^{l+1})} = \frac{\exp(\sum_{n^l} A^{l}_{(:,n^l,n^{l+1})})}{\sum_{n^{l+1}} \exp(\sum_{n^l} A^{l}_{(:,n^l,n^{l+1})})} \tag{2.13}$$

2.2.6 Функція втрат капсульної нейронної мережі

Функція втрат капсульної нейронної мережі має такий вигляд (рівняння 2.14).

$$L_c = T_c \max(0, m^+ - \|v_c\|)^2 + \lambda(1 - T_c)\max(0, \|v_c\| - m^-)^2, \quad (2.14)$$

де L_c – втрати для однієї капсули у вихідному капсульному шарі мережі, T_c – індикатор правильності передбачення капсули, 1 коли передбачення правильне, 0, коли неправильне, v_c – норма вихідного значення капсули, m^+ - поріг правильності передбачення (зазвичай 0.9), m^- - поріг неправильності передбачення (зазвичай 0.1)

Тобто, функція втрат має наступний вигляд

$$L = \sum_i L_i \quad (2.15)$$

де i = 1…m – індекси капсул у останньому шарі, який виконує класифікацію.

Додатково, у архітектурах капсульних нейронних мереж з застосуванням генератора, у функції втрат використовується доданок з похибкою реконструції оригінального зображення, яке було створено генератором.

$$L = \sum_i L_i + \gamma \left\| x_{true} - x_{pred} \right\| \quad (2.16)$$

де, x_{true} – вхідне зображення до капсульної нейронної мережі, x_{pred} – зображення створене генератором, γ – коефіцієнт впливу регуляризатора на функцію втрат.

2.3 Найбільш впливові параметри капсульних нейронних мереж

Кількість шарів і їх розташування:

- згорткових шарів;

- капсульних згорткових шарів.

Функція активації капсули (“стиснення”), можуть бути різних видів:

- використана у статті [2] $v_j = \frac{\|s_j\|^2}{1+\|s_j\|^2} \frac{s_j}{\|s_j\|^2}$;
- використана у статті [16] $v_j = (1 - \frac{1}{e^{\|s_j\|}}) \frac{s_j}{\|s_j\|^2}$.

Параметри згорткових шарів на початку моделі:

- розмір ядра, крок, функція активації нейронів, кількість фільтрів.

Параметри згортки у капсульних згорткових шарах:

- аналогічні до параметрів згорткових шарів.

Кількість капсул у капсульних шарах.

- розмір капсул у капсульних шарах.

Принцип реалізації капсул у мережі:

- “Routing by Agreement”;
- “Expectation Maximization”;
- “Attention Routing”;
- “Homogenous Vector Capsules”;
- “Self-Attention Routing”.

2.4 Порівняння капсульних нейронних мереж із сучасними згортковими мережами

Капсульні нейронні мережі з’явилися відносно недавно і є активним предметом досліджень. Тим не менш, вони показують результати високого рівня на еталонних наборах даних, таких як MNIST або Cifar10. Капсульні нейронні

мережі вивчають складні ієрархічні залежності даних, показують себе ліпше при навчанні на обмеженому об’ємі даних.

На відмінність від згорткових мереж, де головною метою є збільшення кількості шарів й великий обсяг досліджень направлений на вирішення проблем, які з цим супроводжуються (вибухи, зникнення градієнтів), капсульні нейронні мережі можуть мати достатньо неглибоку архітектуру, де є декілька згорткових шарів на початку мережі, 1-2 капсульних згорткових шарів (у статті [2] їх 1, у статті [6] їх 2) і один капсульний шар. В той самий час, згорткова нейронна мережа Xception має у собі 71 шар. Навіть з такою малою глибиною, вони все одно отримують гарні результати - капсульна нейронна мережа зі статті [1] мала точність 99.75% на датасеті MNIST, що було найбільш високим результатом на момент її виходу.

Тим не менш, через поскладнення ієрархічної структури капсульної нейронної мережі й використання алгоритмів оптимізації направлення даних між капсулами різних шарів, капсульні нейронні мережі, є значно повільнішими у навчанні за згорткові нейронні мережі. Через це, для вхідних даних великої розмірності, переважними все ще будуть згорткові капсульні мережі. Також у капсульній мережі робиться припущення, що на вхідному зображенні може знаходитися лише один об’єкт класу, який мережа повинна класифікувати [2].

2.5 Тренування капсульної нейронної мережі

Тренування капсульної нейронної мережі відбувається за допомогою методу оберненого поширення похибки.

Метод оберненого поширення похибки є ключовим алгоритмом для тренування штучних нейронних мереж у глибокому навчанні []. Цей метод

дозволяє мережі навчатися і адаптуватися до вхідних даних, зменшуючи помилку між передбаченнями моделі та фактичними результатами.

Основна ідея полягає в тому, щоб зворотно поширювати похибку від виходу мережі до її входу, оновлюючи ваги нейронів таким чином, щоб мінімізувати цю похибку.

Після отримання виходу мережі обчислюється похибка, яка представляє різницю між передбаченими значеннями і фактичними результатами. Похибка потім поширюється від виходу до входу, враховуючи ваги нейронів. Кожен нейрон отримує "зворотній сигнал" про свій внесок у загальну похибку, і ваги коригуються відповідно до цього сигналу.

2.5.1 AdaGrad

Алгоритм AdaGrad (зменшено від Adaptive Gradient Algorithm) є методом оптимізації, який використовується в машинному навчанні для тренування моделей. Він був запропонований Джоном Дючі та Юріем Нестеровим у 2011 році.

Метою AdaGrad є мінімізація математичного сподівання стохастичної цільової функції відносно набору параметрів, заданого послідовністю реалізацій функції. Як і з іншими методами на основі субградієнтів, це досягається шляхом оновлення параметрів в напрямку протилежному до субградієнтів. У стандартних методах на основі субградієнтів використовуються правила оновлення з розмірами кроку, які не враховують інформацію з минулих спостережень. AdaGrad, натомість, адаптує швидкість навчання для кожного параметра індивідуально, використовуючи послідовність оцінок градієнту (рис. 2.12, де $g_\tau -$ градієнт $f_t(x)$ по x).

Algorithm 1: AdaGrad general algorithm

η: Stepsize ;
$f(x)$: Stochastic objective function ;
x_1: Initial parameter vector;
for $t = 1$ *to* T **do**
    Evaluate $f_t(x_t)$;
    Get and save g_t ;
    $G_t \leftarrow \sum_{\tau=1}^{t} g_\tau g_\tau^\top$;
    $x_{t+1} \leftarrow x_t - \eta G_t^{-1/2} g_t$;
end
return x_t

Рисунок 2.12 – Псевдокод алгоритму AdaGrad[1]

2.5.2 Adam

Оптимізатор Adam є розширеною версією стохастичного градієнтного спуску (СГС), яку можна впроваджувати в різноманітні застосування глибокого навчання, такі як комп'ютерний зір та обробка природної мови в майбутні роки.

[1] Джерело зображення: https://optimization.cbe.cornell.edu/images/thumb/d/d1/PseudoCode_tau.jpg/800px-PseudoCode_tau.jpg

Adam був вперше представлений у 2014 році і презентований на відомій конференції для дослідників глибокого навчання під назвою ICLR 2015.

Назва походить від адаптивної оцінки моментів. Оптимізатор називається Adam через використання оцінок першого та другого моментів градієнту для адаптації швидкості навчання для кожної ваги нейромережі.

Adam пропонується як найефективніший стохастичний оптимізатор, який потребує лише градієнтів першого порядку, де вимоги до пам'яті дуже невеликі. Перед введенням Adam було представлено багато адаптивних технік оптимізації, таких як AdaGrad, RMSP, які мають хорошу ефективність порівняно з СГС, але в деяких випадках мають недоліки, такі як загальна ефективність, яка гірша, ніж у СГС в деяких випадках. Таким чином, був представлений Adam, який є кращим з точки зору загальної ефективності. Також в Adam гіперпараметри мають інтуїтивне тлумачення та, отже, вимагають менше налаштувань. Adam показує добрі результати, але в деяких випадках дослідники помічають, що Adam не завжди збігається до оптимального рішення у порівнянні з оптимізатором СГС. У різноманітних задачах глибокого навчання іноді оптимізатори Adam мають низьку загальну ефективність. За словами авторів Нітіша Ширіша Кескара і Річарда Сохера, у деяких випадках перехід до СГС демонструє кращу загальну ефективність, ніж самостійний Adam.

У Adam замість адаптації швидкості навчання на основі середнього першого моменту, як у RMSP, використовується середнє значення других моментів градієнтів. Цей алгоритм обчислює експоненційно згладжене середнє значення градієнтів та квадратів градієнтів. Параметри β_1 та β_2 використовуються для контролю швидкостей згасання цих згладжених середніх значень. Adam є комбінацією двох методів градієнтного спуску: Momentum та RMSP

$$m_t = \beta_1 \cdot m_{t-1} - 1 + (1 - \beta_1) \cdot g_t, \tag{2.19}$$

$$v_t = \beta_2 \cdot v_{t-1} - 1 + (1 - \beta_2) \cdot g_t^2, \quad (2.20)$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad (2.21)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \quad (2.22)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t. \quad (2.23)$$

2.5.3 Алгоритм Нестєрова

Алгоритм Нестєрова - це модифікація стандартного методу градієнтного спуску, яка була розроблена Юрієм Нестеровим. Вона дозволяє більш ефективно збігатися до мінімуму функції в порівнянні з класичним градієнтним спуском, особливо в задачах з великим обсягом даних.

Основна ідея методу полягає в тому, щоб модифікувати оновлення параметрів до мінімуму. Замість того, щоб використовувати градієнт для визначення напрямку оновлення, NAG використовує невеличкий виправний крок вперед, визначений на попередньому кроці.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}), \quad (2.24)$$

$$\theta_t = \theta_{t-1} + v_t, \quad (2.25)$$

де θ_t – поточний вектор параметрів, $\nabla_\theta J(\theta)$ – градієнт функції, v_t – швидкість оновлення параметрів на кроці t, η – швидкість навчання, γ - коефіцієнт згасання, зазвичай має значення 0.9.

Алгоритм використовує прогнозовану позицію $\theta - \gamma v_{t-1}$ для обчислення градієнта. Це дозволяє врахувати інформацію з попереднього кроку і виправити напрямок оновлення.

2.5.4 Генетичний алгоритм як альтернативний спосіб навчання нейронних мереж

Генетичний алгоритм є алгоритмом, натхнення на створення якого дала теорія Дарвінівської еволюції, він можу бути використаним для оптимізації параметрів нейронних мереж. Основна ідея полягає в тому, щоб застосовувати концепції природного відбору і генетики для ефективного пошуку оптимальних ваг та параметрів мережі. Генетичний алгоритм для тренування нейронних мереж загалом виглядає таким чином.

1) Створення початкової популяції генів – випадкового або псевдовипадкового набору параметрів (ваг, зсувів, тощо), з яких створюються нейронні мережі, які й будуть оцінюватися.
2) Оцінка пристосованості – оцінка кожної особини популяції за допомогою функції пристосування, яка визначає, наскільки добре вона вирішує завдання.
3) Селекція – відбір кращих особин на основі їхньої пристосованості. Можуть використовуватися різні методи селекції, вона може відбуватися більш випадково й більше детерміновано, залежно від підходу.
4) Кросовер – обмін частинами параметрів (генами) між відібраними батьківськими особинами для створення нових потомків. Це допомагає зберігати корисні комбінації параметрів.
5) Мутація – випадкова зміна значень параметрів з метою введення різноманітності в популяцію та виявлення нових оптимальних рішень.
6) Оновлення популяції - заміна старих особин новими потомками. Зазвичай відбувається заміна менш пристосованих особин.

7) Повторення – повторення кроків з 2 по 6 протягом кількох поколінь або до того моменту, коли досягається заздалегідь визначений критерій зупинки (наприклад, достатньо низький рівень функції витрат).

Результат – остаточна особина (модель нейронної мережі з топологією й параметрами), яка найкраще вирішує завдання.

Важливо зазначити, що ефективність генетичного алгоритму для тренування нейронних мереж може сильно залежати від вибору параметрів генетичного алгоритму, таких як розмір популяції, ймовірність кросоверу та мутації, критерії зупинки і т. д.

Висновки до розділу 2

У цьому розділі було наведено загальні відомості про топологію капсульних нейронних мереж і математичних моделей їх шарів, функції втрат. Виконано порівняння сучасних капсульних нейронних мереж та згорткових. Описано сучасні методи навчання нейронних мереж, а саме методи градієнтного спуску для оберненого поширення похибки й генетичні алгоритми.

РОЗДІЛ 3 РОЗВ'ЯЗАННЯ ЗАДАЧІ СТРУКТУРНО-ПАРАМЕТРИЧНОГО СИНТЕЗУ КАПСУЛЬНИХ НЕЙРОННИХ МЕРЕЖ

3.1 Вибір архітектури та принципу реалізації капсульних нейронних мереж

Хоча капсульні нейронні мережі виникли достатньо недавно, ідея використання капсул зазнала великої кількості ітерацій та розгалуджень. Перед початком вирішення задачі структурно-параметричного синтезу необхідно визначити, який саме підхід й архітектуру з наведених у сучасній літературі слід використовувати.

3.1.1 Оригінальна капсульна нейронна мережа

Капсульна нейронна мережа зі статті [2] використовує досить неглибоку архітектуру. У ній використовується алгоритм маршрутизації за згодою, у якому капсули нижнього рівня мають більший вплив на капсули вищого рівня, якщо є багато інших капсул, вихідні значення яких є відносно близькими.

Шар PrimaryCaps є згортковим капсульним шаром, у якому N капсул (у статті 32 капсули), застосовують операцію згортки з кількістю ядер D та розміром ядра k, кроком s (у статті 8 ядер розміром 9x9 з кроком 2) на отриманих зі згорткового шару картах ознак. Як результат отримуємо вихід розмірності out × out × N × D, де out – висота або ширина карти ознак після застосування згортки. out × out × N – це кількість векторів, кожний з яких є входом до капсули у наступному шарі. Таким чином маємо наступну кількість вагових параметрів у

цьому шарі N × D × k × k × f, де f – кількість фільтрів/каналів з попереднього шару, всі ці параметри уточнюються шляхом оберненого поширення помилки.

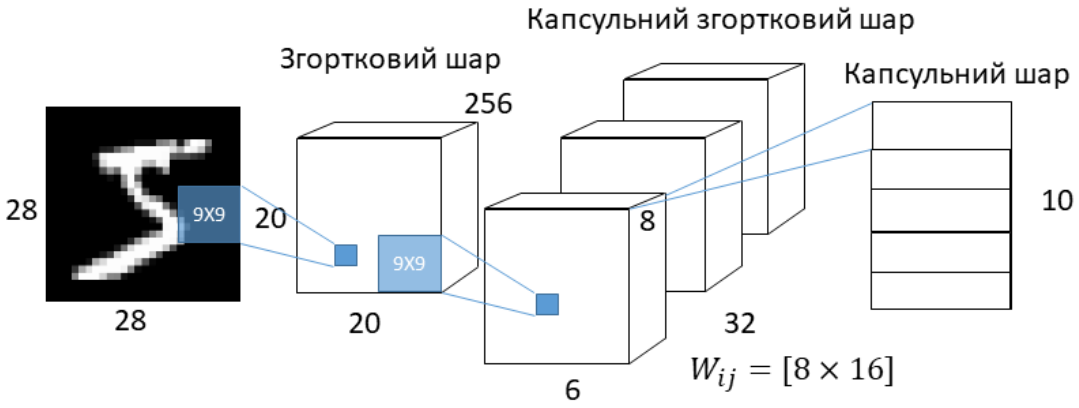

Рисунок 3.1 – Архітектура CapsNet

Для кожного з векторів u_{ij} попереднього шару, для кожної капсули, створена матриця вагових коефіцієнтів W_{ij} розмірності $D \times D^{l+1}$, де D^{l+1} – розмірність виходу капсули наступного шару. Таким чином, кожна з матриць перетворює свій вектор на вектор довжиною вихідного виміру капсули. Після застосування множення вагових матриць та отримання $\hat{u}_{ij} = W_{ij}u_{ij}$, застосовується ітеративний алгоритм динамічної маршрутизації даних за згодою, у якому оцінюється додатковий ваговий коефіцієнт для кожного з отриманих векторів c_{ij}. У алгоритмі також використовується тимчасове значення b_{ij} на кожен з коефіцієнтів c_{ij}. Ці коефіцієнти використовуються у зваженій сумі векторів $s_j = \sum_i c_{ij}\,\hat{u}_{ij}$, до якої після цього застосовується операція активації капсули "стиснення" $v_j = \frac{\|s_j\|^2}{1+\|s_j\|^2}\frac{s_j}{\|s_j\|^2}$, що і залишається вихідним значення цього шару. Таким чином кількість параметрів, які наявні у цьому шарі для однієї капсули - $\text{out} \times \text{out} \times \text{N} \times D \times D^{l+1} + \text{out} \times \text{out} \times \text{N} \times 2$, де $\text{out} \times \text{out} \times \text{N} \times 2$ – це кількість параметрів, які не вточнюються за допомогою

оберненого поширення похибки, навідмінно від всіх інших. Якщо це фінальний шар, то капсул повинно бути, стільки, скільки є класів у датасеті.

Недоліки:

- дуже висока кількість параметрів у мережі - 8.2 мільйонів; 5 мільйонів вагових коефіцієнтів знаходяться лише у капсульному згортковому шарі;
- значне сповільнення навчання через використання ітеративного алгоритму маршрутизації даних між капсулами.

3.1.2 Капсульні мережі з маршрутизацію з максимізацією очікування

Розвитком оригінальної капсульної нейронної мережі стала капсульна нейронна мережа [6], у якій був вдосконалений алгоритм маршрутизації. Також, капсули у статті мають не векторну форму як у попереднику, а матричну, з додатковим окремим параметром, який саме відповідає за наявність ознаки.

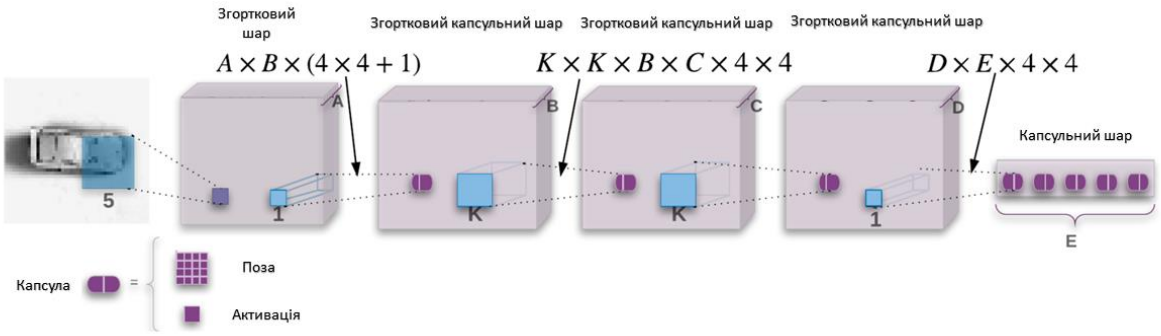

Рисунок 3.2 – Архітектура CapsNet з маршрутизацією максимізації очікування [6]

Кожна капсула містить у собі матрицю пози 4x4 - M, та ймовірність активації - a. Між кожною капсулою i у шарі L та капсулою j у шарі L + 1 знаходиться трансформаційна матриця ваг, що налаштовуються - W_{ij}, а також 2 параметри, що вказують на активацію капсули. Матриця пози капсули i трансформується матрицею W_{ij}, щоби отримати її "голос" $V_{ij} = M_i W_{ij}$ про вигляд матриці j. Пози й активації у шарі L + 1 обчислюються за допомогою нелінійного алгоритму максимізації очікування, який використовує V_{ij} і a_i, для всіх i та j у L та L + 1.

procedure EM ROUTING($\boldsymbol{a}$, V)
  $\forall i \in \Omega_L, j \in \Omega_{L+1}$: $R_{ij} \leftarrow 1/|\Omega_{L+1}|$
  for t iterations **do**
    $\forall j \in \Omega_{L+1}$: M-STEP($\boldsymbol{a}$, R, V, j)
    $\forall i \in \Omega_L$: E-STEP(μ, σ, $\boldsymbol{a}$, V, i)
  return $\boldsymbol{a}$, M

procedure M-STEP($\boldsymbol{a}$, R, V, j) ▷ for one higher-level capsule, j
  $\forall i \in \Omega_L$: $R_{ij} \leftarrow R_{ij} * \boldsymbol{a}_i$
  $\forall h$: $\mu_j^h \leftarrow \frac{\sum_i R_{ij} V_{ij}^h}{\sum_i R_{ij}}$
  $\forall h$: $(\sigma_j^h)^2 \leftarrow \frac{\sum_i R_{ij} (V_{ij}^h - \mu_j^h)^2}{\sum_i R_{ij}}$
  $cost^h \leftarrow \left(\beta_u + log(\sigma_j^h)\right) \sum_i R_{ij}$
  $a_j \leftarrow logistic(\lambda(\beta_a - \sum_h cost^h))$

procedure E-STEP(μ, σ, $\boldsymbol{a}$, V, i) ▷ for one lower-level capsule, i
  $\forall j \in \Omega_{L+1}$: $\boldsymbol{p}_j \leftarrow \frac{1}{\sqrt{\prod_h^H 2\pi(\sigma_j^h)^2}} \exp\left(-\sum_h^H \frac{(V_{ij}^h - \boldsymbol{\mu}_j^h)^2}{2(\sigma_j^h)^2}\right)$
  $\forall j \in \Omega_{L+1}$: $\boldsymbol{R}_{ij} \leftarrow \frac{\boldsymbol{a}_j p_j}{\sum_{k \in \Omega_{L+1}} \boldsymbol{a}_k p_k}$

Рисунок 3.3 – Псевдокод алгоритму [6]

Між згортковим шаром та капсульним згортковим шаром маємо $A \times B \times (4 \times 4 + 1)$ параметрів, що налаштовуються, де A – кількість фільтрів у згортковому шарі, B – кількість капсул у капсульному шарі.

Між згортковими капсульними шарами маємо $К \times С \times D \times 4 \times 4$ параметрів, що налаштовуються, де $К -$ розмір ядра згортки у першому шарі, C – кількість капсул в першому шарі, D – кількість капсул у другому шарі.

Між згортковим капсульним шаром й капсульним шаром, маємо $E \times F \times 4 \times 4$ – параметрів, що налаштовуються, E – кількість капсул у згортковому капсульному шарі, F – кількість капсул у капсульному шарі.

Переваги:

- Новий алгоритм дозволяє капсулам більш ефективно вивчати ознаки й використовує менше параметрів, що налаштовуються (якщо у CapsNet [2] для необхідно пози розміру n, яка має векторну форму у ваговій матриці треба мати n^2 параметрів, то у EM необхідно лише n параметрів, бо має матричну форму).

Недоліки:

- Новий алгоритм хоча й більш ефективно визначає оптимальні зв’язки між капсулами, але є важким з точки зору обчислювальної складності і є все ще ітеративним.
- Використання додаткового елемента для визначення активації капсули йде у розріз з елегантністю оригінального метода, у якому активацією капсули була довжина вектора.

3.1.3 Капсульні мережі з маршрутизацією на основі уваги

Капсульні мережі за даним підходом використовують механізм уваги,який реалізований за допомогою скалярного добутку карт ознак капсул з ядром згортки.

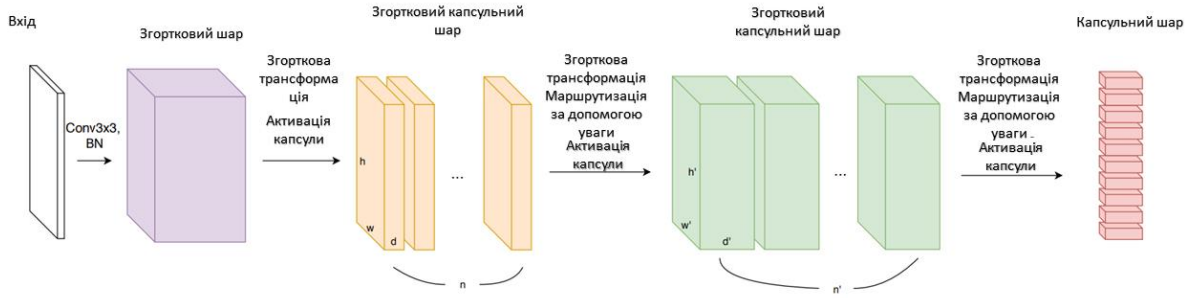

Рисунок 3.4 – Топологія капсульної нейромережі з маршрутизацією на основі уваги [13]

$u^{l}_{w,h,d,n}$ – карта ознак, отримана у шарі l, капсулою d, на рівні каналу n.

$$s^{0}_{(:,:,:,n_0)} = ReLU(Conv_{3\times3}(\tilde{x})), \tag{3.1}$$

$$u_{(:,:,:,n_0)} = \tanh(Conv_{1\times1}(s_{(:,:,:,n_0)})), \tag{3.2}$$

$$\tilde{s}^{l,n}_{(:,:,:,m)} = Conv_{3\times3}(u^{l-1}_{(:,:,:,m)}), \tag{3.3}$$

$$s^{l}_{(w,h,:,n)} = \sum_{m=1,\dots,N^{l-1}} c^{l,n}_{(w,h,m)} \tilde{s}^{l,n}_{(w,h,:,m)}, \tag{3.4}$$

$$b^{l,n} = Conv3D_{1\times1\times D^{l}}(\tilde{s}^{l}), \tag{3.5}$$

$$c^{n,l}_{w,h,:,N^{l-1}} = softmax\left(b^{l,n}_{w,h,:N^{l-1}}\right). \tag{3.6}$$

Замість вагових матриць для трансформації векторів з капсул у даному підході застосовуються згорткові трансформації. Фактично, у даному підході всі капсульні шари є згортковими.

Переваги:

- відсутність ітеративного алгоритму.

Недоліки:

- запропонований підхід не зважає на узгодження передбачень капсул нижнього рівня для створення оцінки про існування ознаки у наступному шарі.

3.1.4 Гомогенні векторні капсули

Даний підхід пропонує взагалі відмовитися від маршрутизації між капсулами. У ньому пропонується використання гілок з капсульними блоками. капс (а) – капсульний шар, який перетворює вхідні значення карт ознак та виміри капсул та їх довжин, за допомогою зміни форми вхідного вектора до вигляду n × d. Далі у капс (б) кожен вектор капсули попереднього шару множиться на відповідну кількість (яка дорівнює кількості класів) векторів з ваговими коефіцієнтами. В результаті повинно бути отримано m × n векторів довжини d, де m – кількість класів, n – кількість капсул, d – довжина вектору капсули. Отримані вектори сумуються по виміру капсул в результаті, буде отримано m × d.

Далі сумуються значення векторів довжин для кожного класу, отримуємо m значень. Вектори з різних гілок складаються, сумуються по класам знову й отриманий вектор передбачень класів експоненційно нормується.

Переваги:

- низька кількість параметрів, що налаштовуються;
- великий простір для знаходження оптимальних архітектур.

Недоліки:

- відмова від маршрутизації між капсулами є значним відходом від оригінальної концепції.

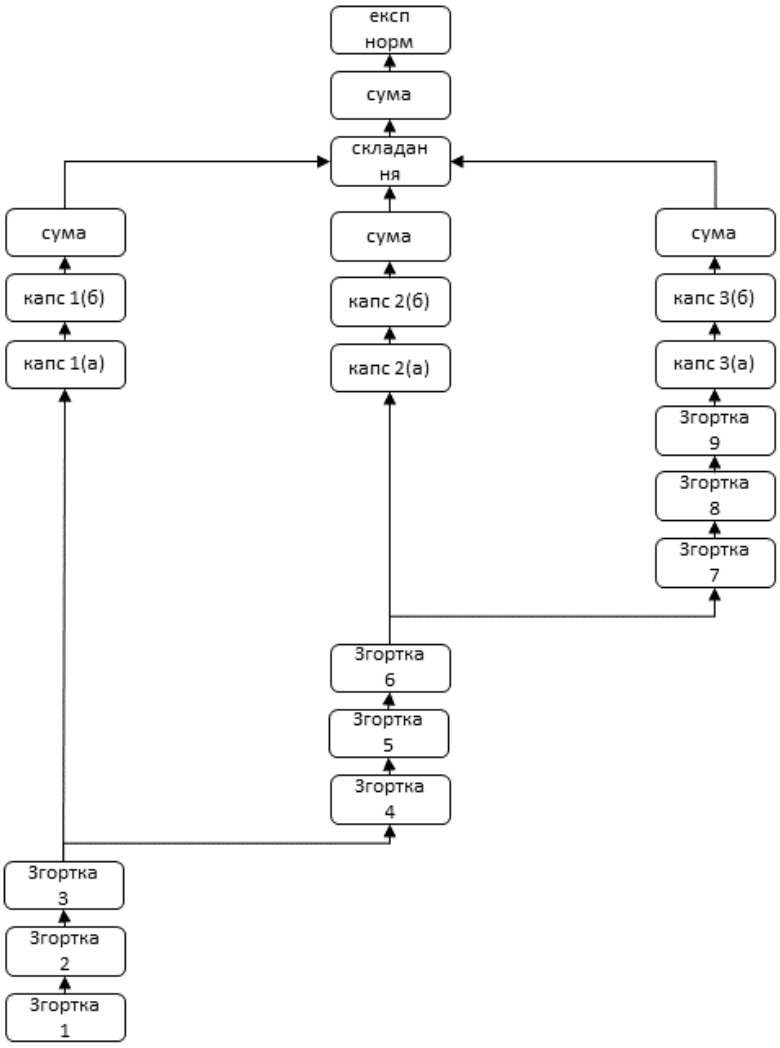

Рисунок 3.5 – Діаграма шарів у нейромережі з гомогенними капсулами

3.1.5 Капсульна мережа з маршрутизацією на основі самоуваги

У даному підході згортковий капсульний шар виконує одну операцію згортки, а після цього перетворює отримані карти ознак на вектори форми N × D, де N – кількість капсул у капсульному згортковому шарі, D –їх розмірність.

Капсульний шар має N капсул розмірності D.

Математичні особливості капсульного шару маршрутизації на основі самоуваги наведено у підрозділі 2.2.5

3.1.6 Вибір підходу та архітектури

Архітектуру капсульної нейронної мережі обрано з використанням генератора, тому що він є органічним засобом регуляризації для капсульних нейронних мереж (вектор передбачення капсули вже має у собі закодовані властивості) й застосовується масово у літературі присвяченій капсулам [2][3][13][16]. Генератор отримує на вхід вектор, який утворений об’єднанням векторів передбачень капсул, при чому, застосовується так звана “маска”, яка присвоює нульові значення векторам капсул, які відповідають за клас відмінний від правдивого. Реалізація капсул обрана зі статті [16], так як вона має наступні переваги:

- є ідейно близькою до оригінальної CapsNet описаної у статті [2];
- має найвищу новизну (стаття вийшла у 2021 році);
- маршрутизація даних між капсул забезпечена механізмом самоуваги з урахуванням узгодження капсул;
- є швидшою за інші реалізації.

3.2 Постановка завдання

3.2.1 Архітектура й топологія капсульної нейронної мережі

Так як у роботі використовується капсульна нейронна мережу з генератором, як підходом для регуляризації, то капсульна нейронна мережа фактично складається з двох нейромереж. Капсульна нейронна мережа складається з:

- згорткових шарів;
- капсульного згорткового шару;
- капсульних повнозв’язних шарів.

Згорткові шари мажуть мати різну кількість ядер, різний крок та розмір ядер, функцію активації. Функції активації будуть розглянуті наступні: сигмоїда, ReLU, Leaky ReLU. Параметрами, що налаштовуються, виступають саме елементи ядер згортки у шарі.

Згортковий капсульний шар може мати ту ж саму кількість ядер, що й у згортковому шарі, що є його попередником, різний крок та розмір ядер. Добуток кількості капсул у шарі й їх розмірності повинен дорівнювати добутку вимірів вихідної карти ознак після застосування операції згортки. Параметрами, що налаштовуються теж виступають ядра згортки у шарі. На рівні векторів виходів капсул використовується функція стиснення капсул.

Капсульні повнозв’язні шари можуть мати довільну кількість капсул та їх розмір. Параметрами, що налаштовуються виступають вагові матриці W афінних перетворень векторних значень капсул з попередніх шарів, матриці B вагів оптимальних зв’язків між капсулами. На рівні векторів виходів капсул використовується функція стиснення капсул.

Шари розташовуються у наступному порядку: спочатку згорткові шари, потім один капсульний згортковий шар, потім капсульні повнозв’язні. Останній повнозв’язний шар повинен мати кількість капсул, скільки міститься класів у датасеті.

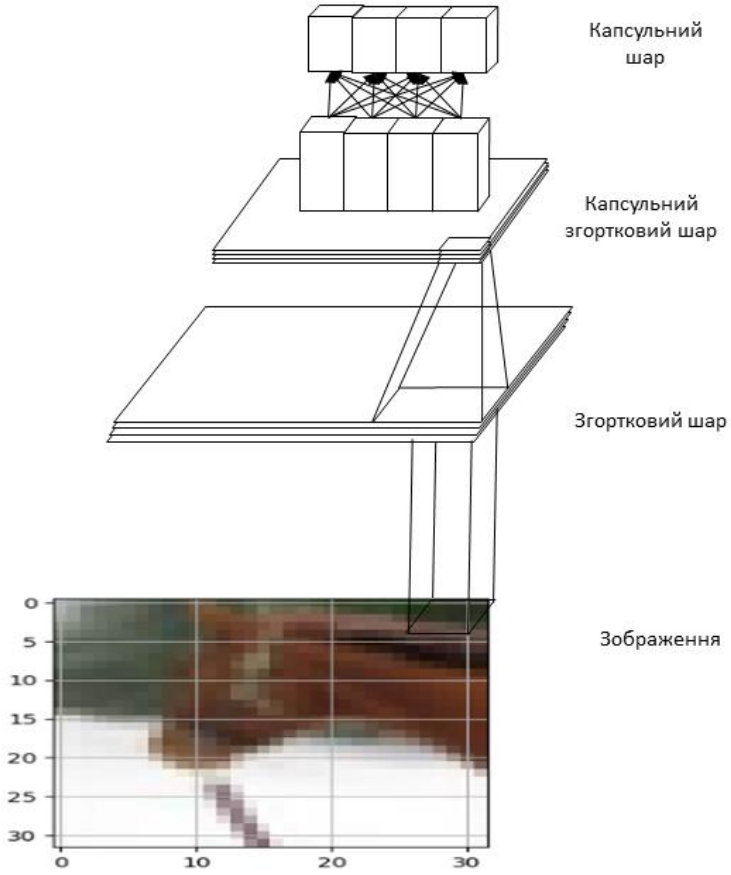

Рисунок 3.6 – Топологія капсульної нейронної мережі

Нейромережа генератор, в свою чергу, може використовувати:

- згорткові шари;
- розгорткові шари;
- звичайні повнозв’язні шари.

У роботі буде використовуватися архітектура генератора, у якій повнозв’язний шар на початку змінює форму на мале зображення з великою кількістю фільтрів, яке з використанням шарів оберненої згортки збільшується до необхідного розміру вихідного зображення, а з додатковий згортковий шар з трьома ядрами згорток переводить зображення у формат RGB.

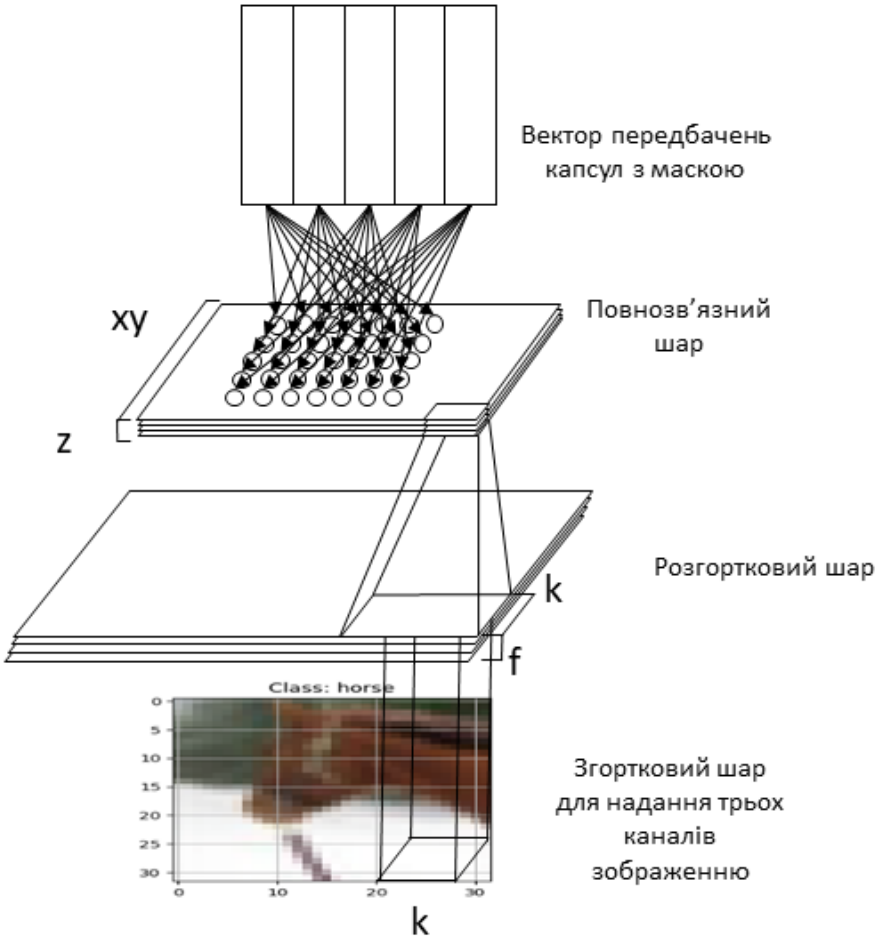

Рисунок 3.7 – Архітектура мережі генератора

Під час навчання $\gamma -$ вплив похибки генератора на загальну похибку мережі, було виставлено як 0.392

3.2.2 Датасет

Задача структурно-параметричного синтезу буде вирішуватися для датасету cifar10. Датасет складається з 60 000 кольорових зображень розміром 32

на 32 пікселі, розподілених на 10 класів. 50 000 зображень є тренувальним, а 10 000 є тестовими.

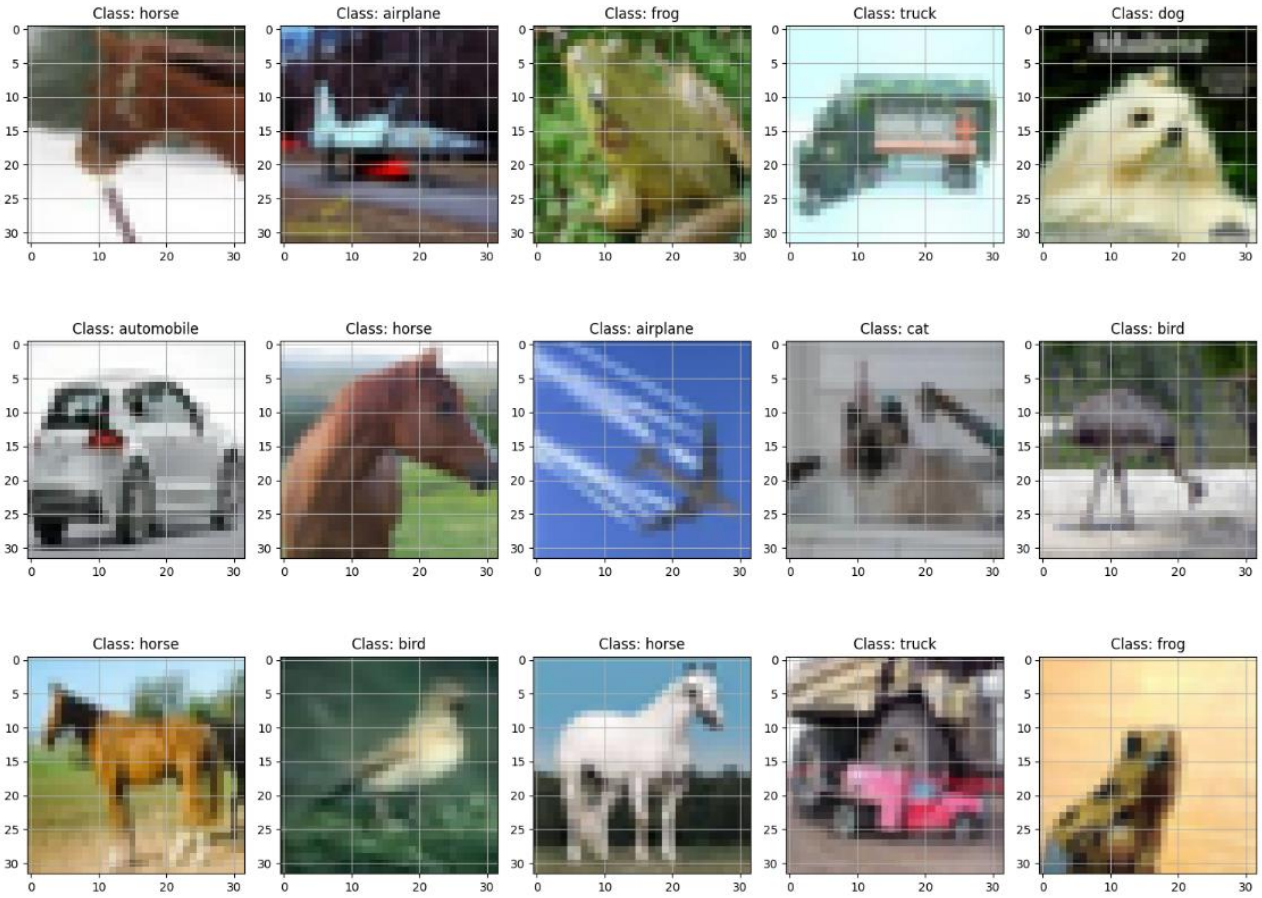

Рисунок 3.8 – Дані з датасету cifar10

Датасет було обрано, тому що капсульні нейронні мережі поки що не досягали на ньому результатів, співставними з сучасними згортковими нейронними мережами й таким чином, простір для знаходження оптимальних архітектур є більшим.

Вхідні дані за стандартною практикою стандартизуються, а також піддаються випадковим змінам яскравості, контрасту та віддзеркаленню.

3.2.3 Математична постановка задачі

Поставимо завдання структурно-параметричного синтезу капсульної нейронної мережі наступним чином.

Задано скінченний набір $J = \{(R_j, Y_j)\}$ j = 1,...,P пар типу «атрибут-значення», де , R_j, Y_j - вхідний і вихідний вектор НМ, відповідно.

Необхідно синтезувати таку оптимальну НМ на основі навчальної вибірки J, яка забезпечувала б ефективне розв'язання задачі класифікації. Критерій оптимальності визначається як

$$I(X) \to opt, \tag{3.7}$$

де $I(X) = E_{acc}(X)$ – точність нейронної мережі при розв'язанні поставленої задачі класифікації на перевірочній вибірці. Хоча дуже важливою характеристикою нейромережі є її складність, яку можна визначити кількістю вагових коефіцієнтів або швидкістю обробки вхідних даних, у даній роботі цією метрикою знехтувано, адже на меті в першу чергу побачити, яку точність можуть отримувати капсульні нейронні мережі зі знайденими оптимальною структурою та параметрами.

$$X = (\bar{x}_{11}, \bar{x}_{12}, \dots, \bar{x}_{1x_2}, x_2, \bar{x}_3, \bar{x}_4, \bar{x}_5, \bar{x}_{61}, \bar{x}_{62}, \dots, \bar{x}_{6x_7}, x_7, x_8, \bar{x}_9)^T, \tag{3.8}$$

де X - вектор, який визначає топологію, структуру та параметри мережі, які необхідно знайти під час вирішення задачі структурно-параметричного синтезу.

$\bar{x}_{1i}$ –вектор властивостей згорткового шару i на початку нейронної мережі.

$$\bar{x}_{1i} = (f_{1i}, k_{1i}, s_{1i}, a_{1i})^T, \tag{3.9}$$

де f_{1i} – кількість ядер у згортковому шарі, $f_{1i} \in \{32, 64, 96, 128, 160, 192, 224, 256\}$, k_{1i} – розмір ядер згорткового шару, $k_{1i} \in \{3, 5, 7, 9\}$, s_{1i} – розмір кроку у згортковому шарі, $s_{1i} \in \{1, 2, 3, 4, 5\}$, a_{1i} – функція активації нейронів у згортковому шарі, $a_{1i} \in$ {'relu', 'sigmoid', 'leaky_relu'}.

x_2 – кількість згорткових шарів.

$\bar{x}_3$ – вектор параметрів капсульного згорткового шару нейронної мережі

$$\bar{x}_3 = (f_3, k_3, s_3, N_3, D_3)^T, \tag{3.10}$$

де f_3 – кількість ядер у згортковому капсульному шарі, $f_3 \in \{32, 64, 96, 128, 160, 192, 224, 256\}$, k_3 – розмір ядер у капсульному згортковому шарі, $k_3 \in \{3, 5, 7, 9\}$ s_3 – розмір кроку у капсульному згортковому шарі, $s_3 \in \{1, 2, 3, 4, 5\}$, N_3 – кількість капсул, $N_3 \in \mathbb{N}$, D_3 – розмір капсули, $D_3 \in \mathbb{N}$, $N_3 \times D_3 = f_3$.

$\bar{x}_4$ – вектор параметрів капсульного шару

$$\bar{x}_4 = (N_4, D_4)^T, \tag{3.11}$$

де N_4 – кількість капсул, $N_4 = m$, де m – кількість класів у навчальній вибірці, D_4 – розмір капсули, $D_4 \in \{8, 16, 32, 48, 64\}$.

$\bar{x}_5$ – вихідні виміри повнозв'язного шару на початку генератора після зміни форми.

$$\bar{x}_5 = (x_5, z_5)^T, \tag{3.12}$$

де x_5 – висота й ширина, $x_5 \in \{2, 4, 8\}$, z_5 – кількість каналів, $z_5 \in \{128, 256, 512\}$.

$\bar{x}_{6i}$ –вектор параметрів оберненого згорткового шару *i* у генераторі. i=1…x_7

$$\bar{x}_{6i} = (f_{6i}, k_{6i}, a_{6i})^T, \quad (3.13)$$

де f_{6i} – кількість ядер, f_{6i} є $\{32, 64, 96, 128, 160, 192, 224, 256\}$, k_{6i} – розмір ядер, k_{6i} є $\{2, 4, 8\}$, a_{6i} – функція активації нейронів, a_{6i} є {'relu', 'sigmoid', 'leaky_relu'}.

x_7 – кількість обернених згорткових шарів.

x_8 – розмір ядра у фінальному згортковому шарі, який повинен повернути зображенню кількість фільтрів, x_8 є $\{1, 2, 3, 4, 5\}$.

$\bar{x}_9$ – вектор вагових параметрів.

Таким чином задача структурно-параметричного синтезу капсульної нейронної мережі представляє собою задачу багатокритеріальної оптимізації.

3.3 Метод (алгоритм) структурно-параметричного синтезу капсульних нейронних мереж

Структурно-параметричний синтез капсульної нейронної мережі складається з двох етапів – структурного синтезу та параметричного синтезу.

Структурний синтез досягається шляхом знаходження оптимальної кількості шарів, їх видів, взаємного розташування, а також параметрів, які цим шарам властиві.

Параметричний синтез ставить собі на мету знайти оптимальні вагові коефіцієнти для нейромережі знайденої під час структурно синтезу, завдяки яким нейромережа буде мати найкращий оптимальний результат.

3.4 Структура гібридного алгоритму структурно-параметричного синтезу

Гібридний алгоритм структурно-параметричного синтезу виконує знаходження оптимальних топологій нейронних мереж й знаходження оптимальних вагових коефіцієнтів.

У ньому поєднується дія двох алгоритмів – генетичного та оптимізаційного алгоритму градієнтного спуску.

Порядок дій представлено нижче.

1. Проініціалізувати популяцію нейронних мереж.
2. Виконати нетривалу оптимізацію ваг для отримання початкових значень точності нейромереж.
3. Створити новий набір нейромереж на основі попереднього, де отримане значення точності використовується для визначення більш ефективних нейромереж, властивості яких ми хочемо зберігати.
4. Повторювати 2-3 певну кількість поколінь.
5. Використати алгоритм градієнтного спуску на мережах з найвищими показниками для дознаходження вагів, які даватимуть найвищу точність.

Для того, щоби зрозуміти, яка структура та гіперпараметри у мережі є більш перспективним необов'язково знаходити оптимальні ваги до кінця, достатньо навчати нейромережу за допомогою градієнтного методу до точки, у якій навчання стабілізується. Це незначне тренування дозволяє отримати порівняльну точність мережі на тестовій виборці, яку після цього можна використати у генетичному алгоритмі для подальшого пошуку вдалих топологій. Після закінчення роботи генетичного алгоритму, найбільш точні мережі можна дотренувати, щоби отримати оптимальні параметри ваг.

3.5 Генетична складова гібридного алгоритму

Генетичний алгоритм застосовується для знаходження оптимальної топології нейронної мережі. Загальну структуру алгоритму було вже описано у підпункті 2.5.4. Тому опишемо структуру хромосоми капсульної нейронної мережі та особливості операцій, які у ньому застосовуються для задачі структурно-параметричного синтезу, а також певні обмеження, які необхідно закласти. Також зазначимо, що для зручності, капсульну нейронну мережу з застосуванням генератора можна поділити на 3 блоки: згортковий блок, капсульний блок, блок-генератор.

3.5.1 Структура хромосоми капсульної нейронної мережі

Хромосома складається з трьох блоків: згорткового, капсульного, блоку генератора.

У згортковому блоці знаходяться гени згорткових шарів та капсульного згорткового шару. Кількість генів згорткових шарів є довільною, але обмежена результуючими вимірами блоку - не дозволяються блоки, у яких вихідний розмір або менший 2 (щоби не отримати помилку або занадто мало карту ознак), або більший за 8 (для уникнення вибуху кількості капсул у капсульному згортковому шарі та зв’язків між ними та капсулами повнозв’язного шару). Гени відповідають розписам векторів у математичній постановці задачі.

У капсульному блоці знаходяться повнозв’язні капсульні шари. Планувалося дослідження існування декількох повнозв’язних шарів, але після певних експериментів, було прийняте рішення обмежитися лише одним.

Використання декількох шарів ускладнило мережу, чим зменшило швидкість тренування, а також трохи погіршило результат.

Блок-генератор складається з гену форми повнозв'язного шару і, як результату, кількості нейронів у ньому, з генів шарів оберненої згортки та гену розміру ядра у фінальному шарі згортки. Ген форми повнозв'язного шару має в собі два параметри горизонтальні виміри та вимір кількості фільтрів. Горизонтальний вимір також визначає, яка буде кількість шарів оберненої згортки, якщо кожен з них використовує доповнення і крок 2, то кількість шарів n визначається рівнянням 3.14.

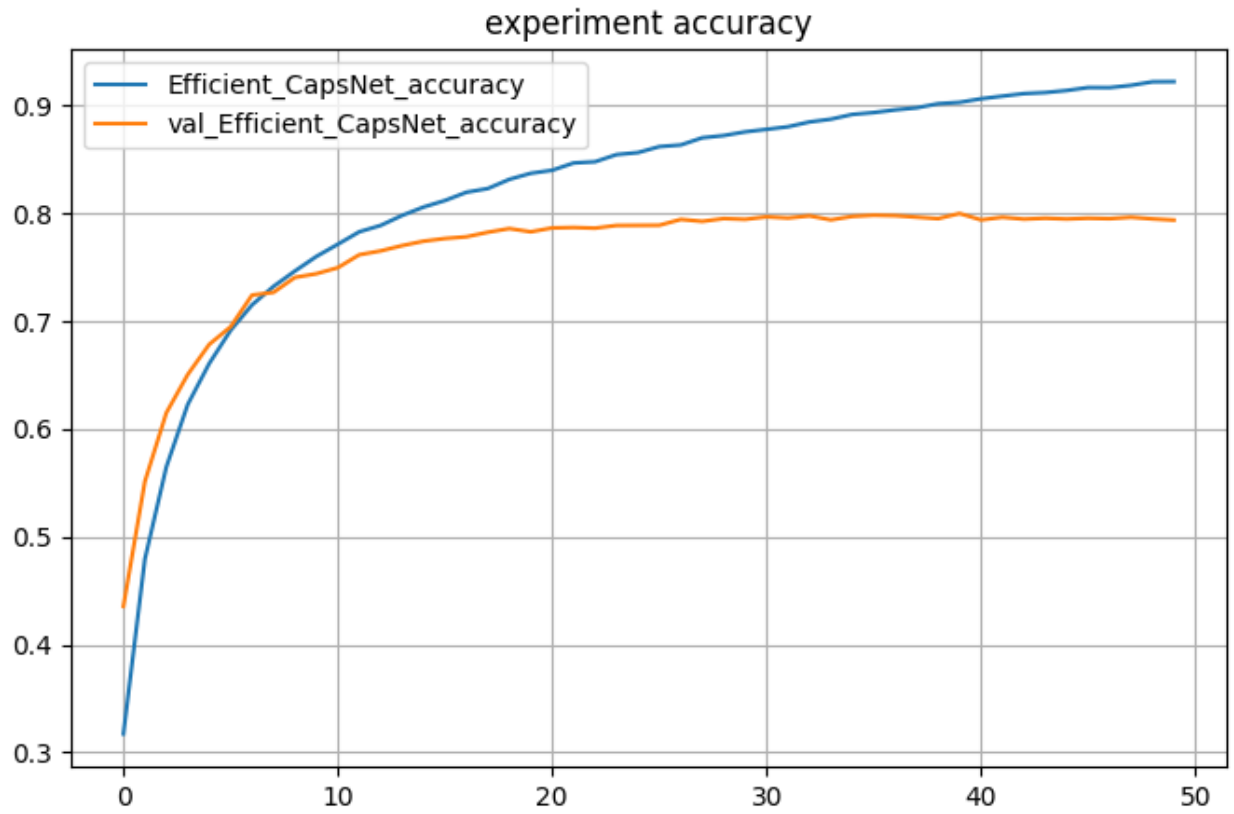

Рисунок 3.9 – Точність експериментальної моделі протягом навчання у 50 епох

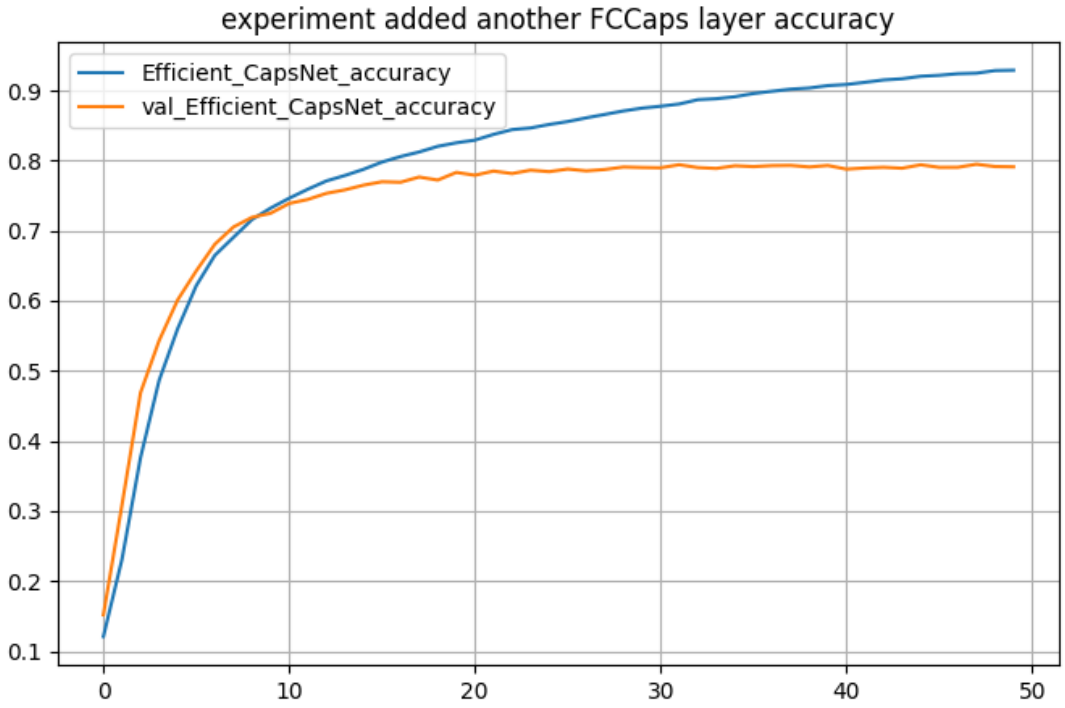

Рисунок 3.10 – Точність експериментальної моделі з додаванням повнозв'язного капсульного шару протягом навчання у 50 епох

$$n = log_2(32/xy), \tag{3.14}$$

де xy – горизонтальний вимір.

Гени обернених згорткових шарів також мають ті ж самі параметри, що описані у математичній постановці задачі

Ген останнього згорткового шару включає в себе лише один параметр – розмір ядра згортки.

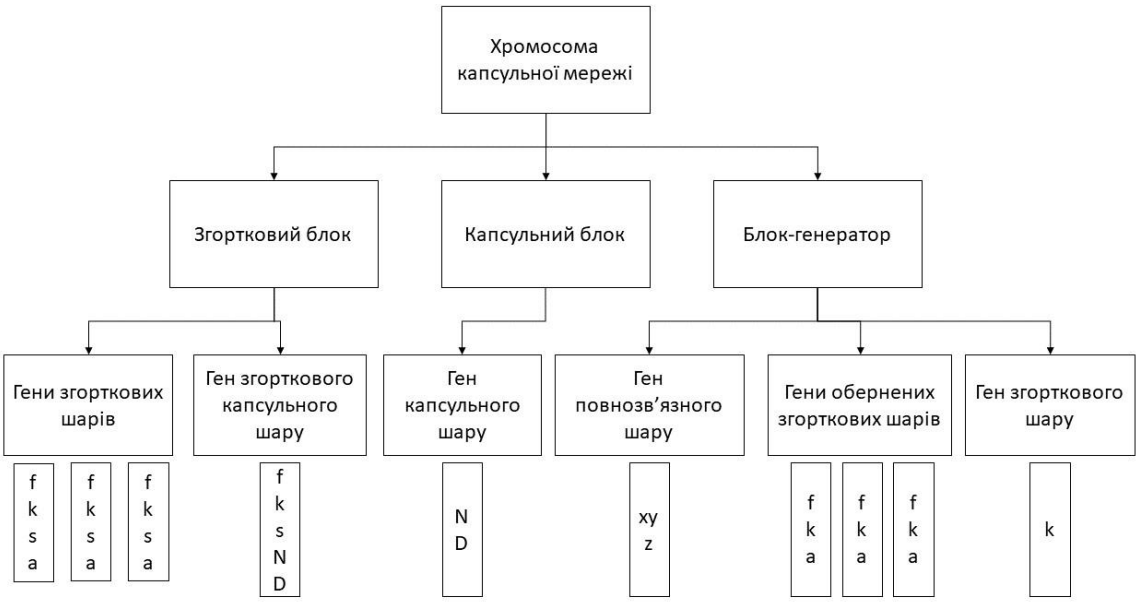

Рисунок 3.11 – Діаграма хромосоми капсульної нейронної мережі

Бінарно хромосому можна закодувати як зображено на рис. 3.12.

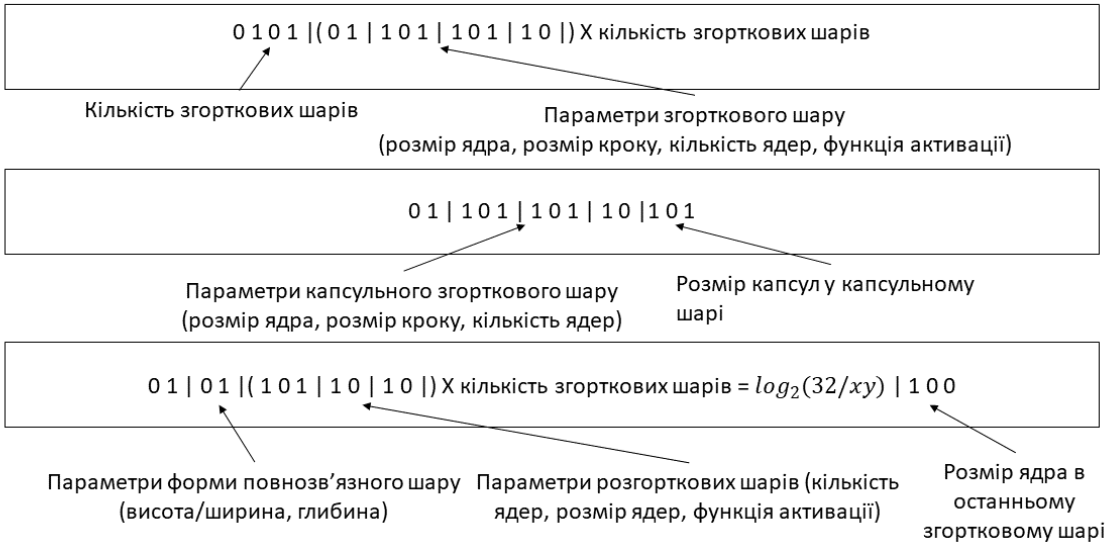

Рисунок 3.12 – Бінарна форма хромосоми капсульної нейронної мережі для генетичного алгоритму

Максимально можлива кількість згорткових шарів – 14, максимальна кількість розгорткових шарів – 4. З урахуванням цього, на закодування хромосоми максимальної довжини може знадобитися до 24 + 10 × 14 + 7 × 4 = 192 біти.

3.5.2 Ініціалізація популяції

Для ініціалізації популяції використовується базова капсульна нейронна мережа, яка мутується по 100 разів для отримання кожного з індивідів популяції.

3.5.3 Мутація

Мутація застосовується до одного з трьох блоків, ймовірність обрати блок нейромережі залежить від кількісті шарів у ньому.

Для згорткового блоку можливі такі мутації після обрання випадкового шару:

- зміна параметрів згорткового шару;
- видалення згорткового шару;
- додавання додаткового згорткового шару шляхом дуплікації вже інсуюючого;
- зміна параметрів капсульного згорткового шару.

Для капсульного блоку мутація полягає у зміні розмірності капсули.

Для блоку-генератора мутація має наступні можливі події:

- зміна вимірів нейронів повнозв’язного шару;
- зміна параметрів випадкового оберненого згорткового шару;
- зміна параметрів фінального згорткового шару.

3.5.4 Операція схрещування

Операція схрещування двох хромосом капсульних нейронних мереж проходить таким чином.

1) Одна з хромосом випадковим чином визначається лівою, а інша правою.
2) Всередині кожного блоку почергово беруться шари з лівої та правої хромосоми.

Таким чином, операція схрещування зберігає відносний порядок шарів у результуючій хромосомі.

3.5.5 Оцінка пристосованості

Єдиним фактором, що враховується в оцінці пристосованості є точність на тестовій вибірці.

$$\text{Fitness(X)} = \text{accuracy}(Y_{pred}, Y_{true}), \tag{3.15}$$

де Y_{pred} – значення передбачені нейромережею, Y_{true} – правдиві значення.

3.5.6 Селекція

Селекція відбувається шляхом бінарного турніру: обираються дві випадкових хромосом. Та хромосома, яка має кращу пристосованість за іншу обирається як більш пристосована.

3.5.7 Оновлення популяції

Особи, які були обрані, схрещуються одне з одним з ймовірністю 0.9, поки їх діти не утворять собою нову популяцію. З ймовірністю 0.2 також застосовується мутація на дітях. Ці коефіцієнти були запозичені зі статті [18], у якій використовувався генетичний алгоритм для знаходження оптимальної структури згорткових нейронних мереж.

3.5.8 Обмеження вкладені в генетичний алгоритм

Є певні обмеження, які варто закладати у структуру капсульних нейронних мереж, які також відображені у капсульному нейронному шарі:

- добуток вихідних вимірів згорткового блоку та кількості фільтрів повинен дорівнювати добутку кількості капсул у згортковому капсульному шарі та довжині цих капсул;

- кількість фільтрів у останньому згортковому шарі згорткового блоку повинна дорівнювати кількості фільтрів у капсульному згортковому шарі;
- завелика вимірність на виході згорткового блоку може призвести до необхідності у завеликій кількості капсул і зв’язків між ними, що може призвести до помилки виділення пам’яті.

У зв’язку з цим, при операціях схрещування та мутаціях необхідно оновлювати кількість капсул і їх довжину у згортковому капсульному шарі. Також встановлено обов’язкове обмеження на розміри виходу капсульного блоку. Якщо у результаті операції схрещування утворити хромосому з дозволеною розмірністю не вдалося, то всі шари дитина отримує від правої хромосоми.

3.6 Градієнтно-оптимізаційна складова гібридного алгоритму

Знаходження вагових коефіцієнтів відбувалося за допомогою алгоритму градієнтного спуску Adam. Початкова швидкість навчання була виставлена 0.0005, вона експоненційно спадає з кожною наступною епохою, фактор експоненційного спадання - 0.97.

Навчання капсульних нейронних мереж для порівняльної точності відбувається 10 епох, тому що приблизно в цей момент воно стабілізується і йде на спад. Донавчання відбувається 40 епох, тому що приблизно саме стільки епох мережа навчається до досягнення абсолютного плато.

3.7 Результати та їх аналіз

3.7.1 Результати застосування генетичного алгоритму для знаходження топологій

Розмір популяції було виставлено у 30 особин. Кількість поколінь з урахуванням початкового – 5. Навчання оптимізатором Adam відбувалося 10 епох, після чого найвища точність за період навчання мережі, присвоювалася відповідній особині та її хромосомі.

Таким чином було оцінено 150 топологій капсульних нейронних мереж. Середній час оцінки одного покоління складав 10 годин. Хромосоми та ваги зберігаються у папку покоління. Формат збереження хромосом – json, механізм запису бібліотека jsonpickle. Також при утворення нового покоління, файли хромосом з інших поколінь використовуються як “кеш”, якщо існує хромосома такого самого вигляду, то особині одразу присвоюється точність, для уникнення зайвих тренувань нейромереж.

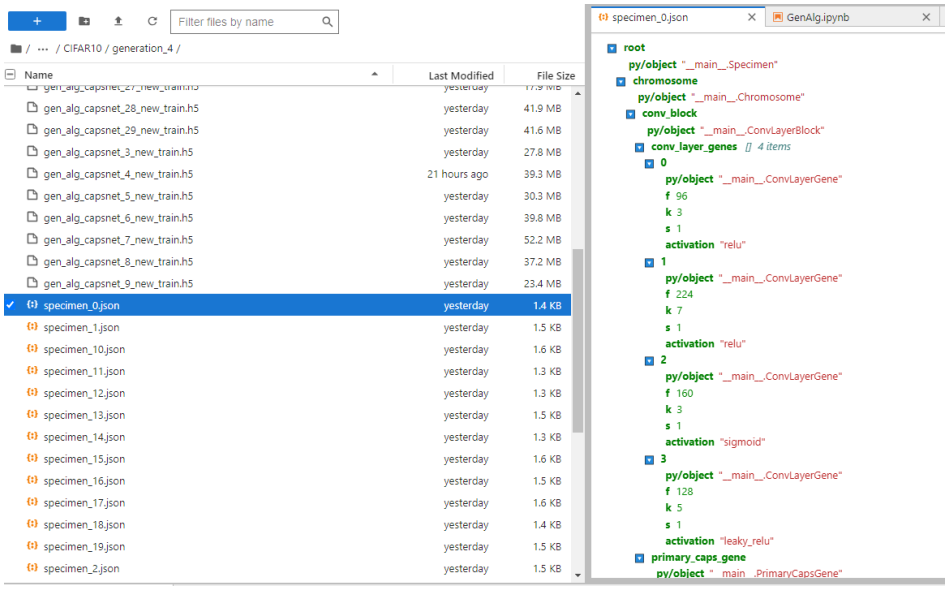

Рисунок 3.13 – Вигляд файлової системи, після виконання алгоритму протягом 5 епох

Еволюційну траєкторію виконання алгоритму можна побачити на рис. 3.14.

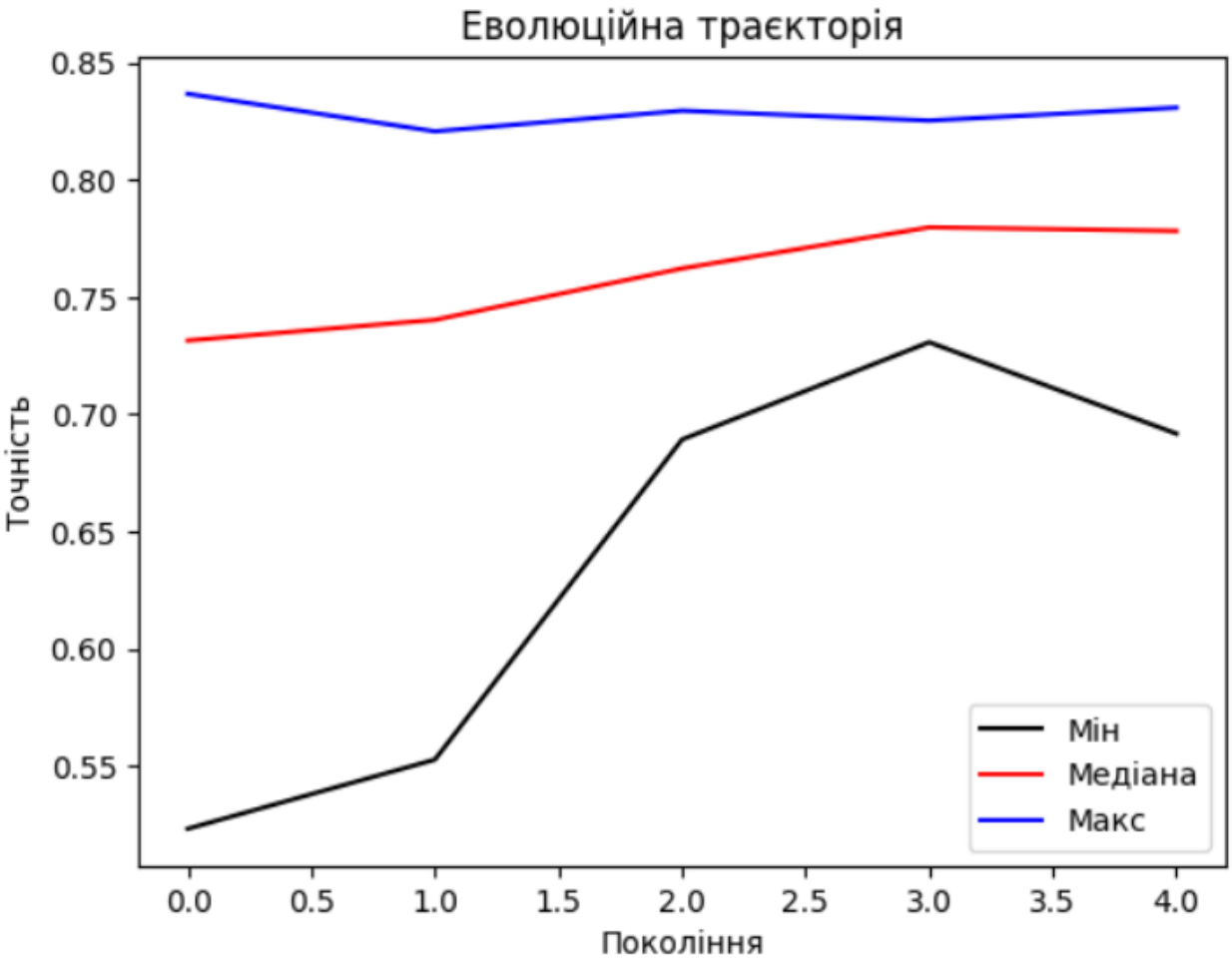

Рисунок 3.14 – Еволюційна траєкторія

Як бачимо з еволюційної траєкторії, загальна якість нейромереж поліпшувалася, окрім останнього покоління, у якому мінімальна якість знизилася. Цьому є пояснення. Так як вхідне зображення попередньо стандартизоване, то значення його пікселів лежать у діапазоні [-1,1]. У початковій версії генетичного алгоритму у гені останнього згорткового шару, також була функція активації, яка могла приймати значення гіперболічного тангенсу, ReLU, Sigmoid, Leaky ReLU. Лише гіперболічний тангенс має свої

вихідні значення у діапазоні [-1,1]. Після виправлення цієї помилки, багато нейромереж почали отримувати коректні значення функції втрат генератора. Архітектури деяких з них, певно були пристосовані до втрат генератора, які жодних чином не змінювалися і це погіршило результати навчання.

Отримано такі п'ять нейромереж з найліпшою точністю (табл. 3.1).

Таблиця 3.1 – Нейромережі з найвищою точністю

Покоління	*Номер особи*	*Точність*	*Кількість параметрів мережі*	*Кількість параметрів генератора*
0	8	0.837	3 798 064	5 645 059
4	4	0.831	5 011 312	5 270 243
2	10	0.830	2 870 192	2 906 275
3	5	0.825	3 548 816	7 887 747
2	14	0.824	3 548 816	2 013 859

Загалом навіть у результаті порівняльного пошуку, було знайдено нейромережі, які отримували ліпші результати за ті, що я знайшов власноруч. Ця базова модель мала точність 0.831 та наступну криву навчання протягом 20 епох (рис. 3.15).

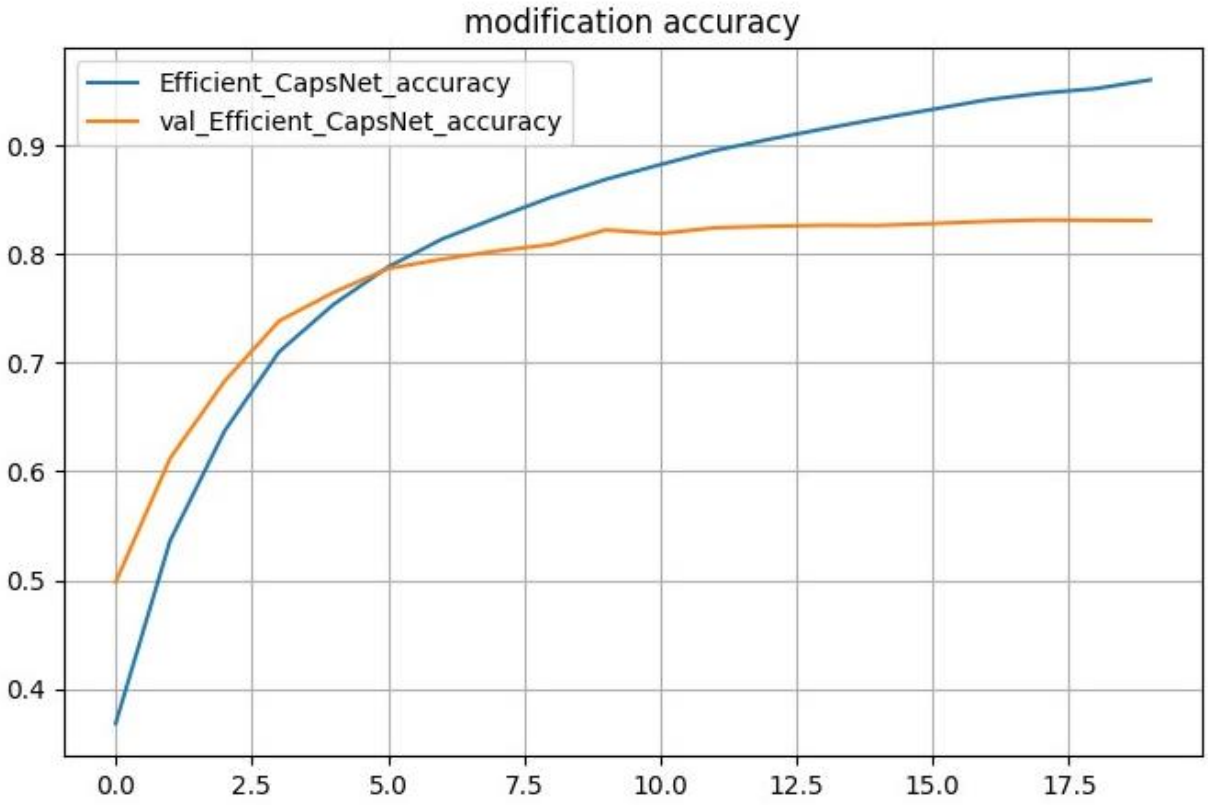

Рисунок 3.15 – Крива навчання нейромережі знайденої власноруч

Наведемо топології найліпших капсульних мереж.

Покоління 0. Особина 8.

Згортковий шар f = 64, k = 3, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 256, k = 3, s = 1, a = ‘ReLU’.

Згортковий шар f = 160, k = 5, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 192, k = 7, s = 1, a = ‘ReLU’.

Капсульний згортковий шар f = 192, k = 7, s = 5, N = 72, D = 24.

Капсульний шар N = 10, D = 64.

Повнозв’язний шар x = 2, z = 256.

Розгортковий шар f = 256, k = 8, s = 2, a = ‘Sigmoid’.

Розгортковий шар f = 256, k = 2, s = 2, a = ‘Leaky ReLU’.

Розгортковий шар f = 256, k = 2, s = 2, a = ‘Leaky ReLU’.

Розгортковий шар f = 256, k = 2, s = 2, a = ‘Leaky ReLU’.

Фінальний згортковий шар f = 3, k = 3, s = 1, a = ‘Sigmoid’.

Покоління 4. Особина 4.

Згортковий шар f = 64, k = 3, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 256, k = 3, s = 1, a = ‘ReLU’.

Згортковий шар f = 224, k = 7, s = 1, a = ‘ReLU’.

Згортковий шар f = 192, k = 3, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 224, k = 5, s = 2, a = ‘ReLU’.

Капсульний згортковий шар f = 224, k = 7, s = 1, N = 56, D = 16.

Капсульний шар N = 10, D = 64.

Повнозв’язний шар x = 2, z = 256.

Розгортковий шар f = 256, k = 8, s = 2, a = ‘Sigmoid’.

Розгортковий шар f = 192, k = 2, s = 2, a = ‘Leaky ReLU’.

Розгортковий шар f = 64, k = 4, s = 2, a = ‘Leaky ReLU’.

Розгортковий шар f = 96, k = 2, s = 2, a = ‘Sigmoid’.

Фінальний згортковий шар f = 3, k = 2, s = 1, a = ‘tanh’.

Покоління 2. Особина 10.

Згортковий шар f = 192, k = 3, s = 1, a = ‘ReLU’.

Згортковий шар f = 64, k = 3, s = 1, a = ‘Sigmoid’.

Згортковий шар f = 224, k = 7, s = 1, a = ‘ReLU’.

Згортковий шар f = 192, k = 3, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 224, k = 5, s = 2, a = ‘ReLU’.

Капсульний згортковий шар f = 224, k = 7, s = 1, N = 56, D = 16.

Капсульний шар N = 10, D = 64.

Повнозв’язний шар x = 2, z = 128.

Розгортковий шар f = 64, k = 2, s = 2, a = ‘Leaky ReLU.

Розгортковий шар f = 160, k = 4, s = 2, a = ‘Leaky ReLU’.

Розгортковий шар f = 224, k = 8, s = 2, a = ‘Sigmoid’.

Розгортковий шар f = 96, k = 2, s = 2, a = ‘Sigmoid’.

Фінальний згортковий шар f = 3, k = 2, s = 1, a = ‘ReLU.

Покоління 3. Особина 5.

Згортковий шар f = 64, k = 3, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 256, k = 3, s = 1, a = ‘ReLU’.

Згортковий шар f = 160, k = 5, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 224, k = 3, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 192, k = 5, s = 2, a = ‘ReLU’.

Капсульний згортковий шар f = 224, k = 7, s = 1, N = 56, D = 16.

Капсульний шар N = 10, D = 64.

Повнозв’язний шар x = 2, z = 256.

Розгортковий шар f = 256, k = 8, s = 2, a = ‘Sigmoid’.

Розгортковий шар f = 192, k = 2, s = 2, a = ‘Leaky ReLU’.

Розгортковий шар f = 224, k = 8, s = 2, a = ‘Sigmoid’.

Розгортковий шар f = 96, k = 2, s = 2, a = ‘Sigmoid’.

Фінальний згортковий шар f = 3, k = 2, s = 1, a = ‘ReLU’.

Покоління 2. Особина 14.

Згортковий шар f = 64, k = 3, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 256, k = 3, s = 1, a = ‘ReLU’.

Згортковий шар f = 160, k = 5, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 224, k = 3, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 192, k = 3, s = 1, a = ‘Leaky ReLU’.

Згортковий шар f = 224, k = 5, s = 2, a = ‘ReLU’.

Капсульний згортковий шар f = 224, k = 7, s = 1, N = 56, D = 16.

Капсульний шар N = 10, D = 64.

Повнозв’язний шар x = 2, z = 256.

Розгортковий шар f = 192, k = 4, s = 2, a = ‘Sigmoid’.

Розгортковий шар f = 32, k = 2, s = 2, a = ‘Leaky ReLU’.

Розгортковий шар f = 224, k = 8, s = 2, a = ‘Sigmoid’.

Розгортковий шар f = 96, k = 2, s = 2, a = ‘Sigmoid’.

Фінальний згортковий шар f = 3, k = 2, s = 1, a = ‘ReLU’.

3.7.2 Результати застосування Adam для дооптимізації ваг

Дооптимізація проводилася проводилася додаткові 40 епох (тобто загальна кількість епох навчання - 50).

Таблиця 3.2 – Точність найкращих нейромереж після донавчання.

Покоління	Номер особи	Точність	Точність після дотренування
0	8	0.837	0.849
4	4	0.831	0.861
2	10	0.830	0.850
3	5	0.825	0.862
2	14	0.824	0.861

Найліпшу точність отримала нейронна мережа 5 з покоління 3, далі буде наведено графіки її дотренування.

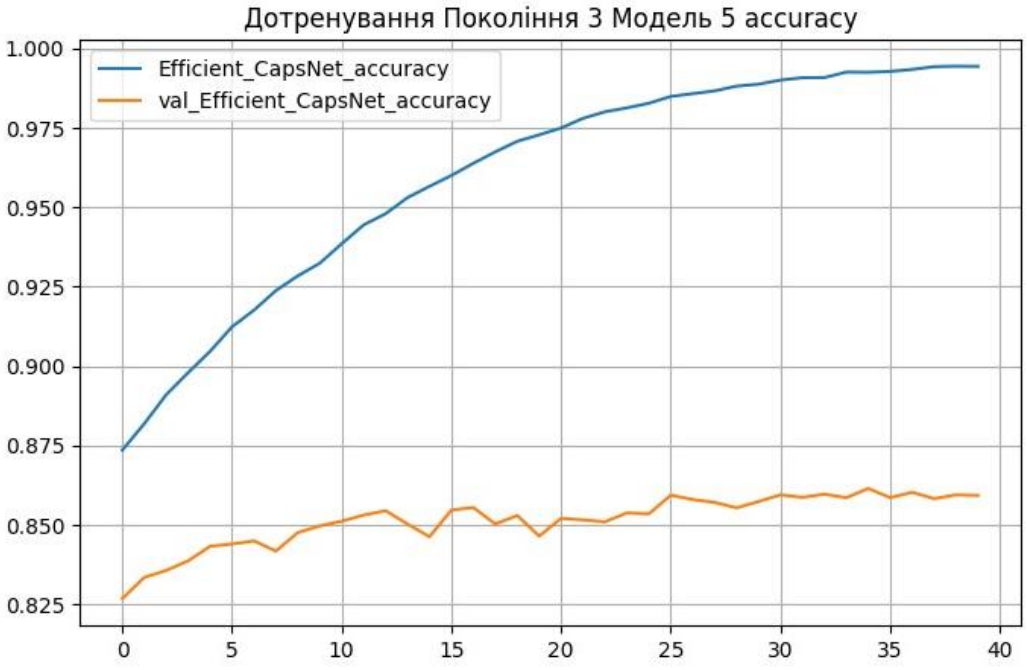

Рисунок 3.16 – Точність

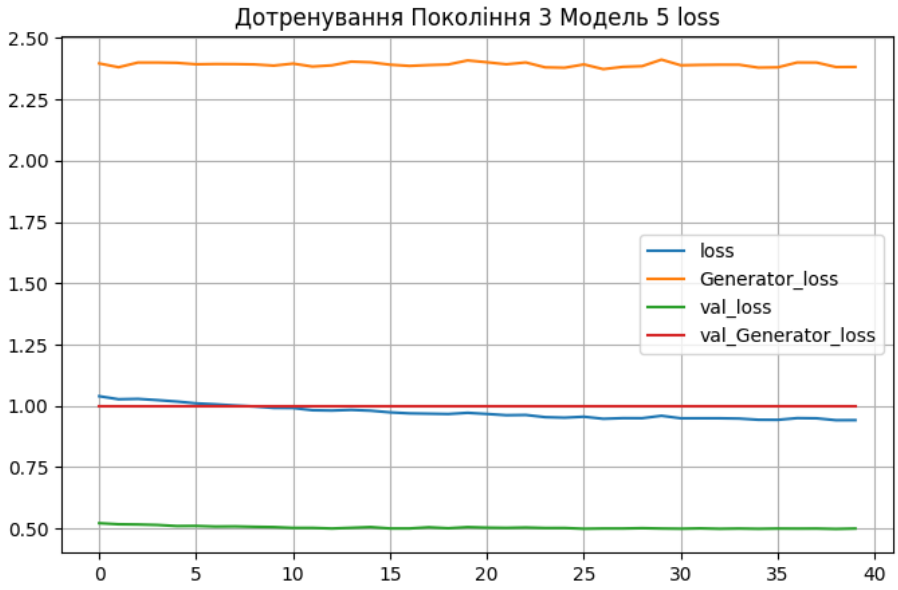

Рисунок 3.17 – Функції втрат

Таким чином, було отримано нейронну мережу, яка має покращені показники у порівнянні зі знайденими вручну. Тим не менш, результати не дуже

задовольняють, згорткові нейронні мережі мають значно кращі результати на цьому датасеті. Причин чому капсульна нейромережа показала себе не дуже гарно є декілька:

1) Датасет – cifar10 є дуже зашумленим датасетом, у якому задні плани сильно відрізняються, що плутає капсули, які впершу чергу повинні закодовувати властивості об’єкту класу й історично капсульні нейронні мережі випередити згорткові на ньому датасеті ще не змогли.
2) Капсульні нейронні мережі сильно залежать від регуляризації мережі генератора. Через помилку з функцією активації останнього шара й відсутність окремої метрики, яка б вказувала на ефективність генератора у створенні зображень та була врахована у генетичному алгоритмі, генератори були малорозвинутими.
3) Недостатня кількість поколінь – процес застосування гібридного алгоритму дуже довгий, хотілося б провести його навчання хоча б 20 поколінь, але забракло часу.

3.8 Структура програмного забезпечення

Програмне забезпечення написане на базі репозиторію гітхаб реалізації капсульних нейронних мереж на основі механізму самоуваги. Використано мову програмування python 3.11, фреймфорк tensorflow, бібліотеку jsonpickly та веб-середовище інтерактивної розробки JupyterLab для. Було створено 4 додаткових файли – GenAlg.ipynb у корневій папці проєкту, genalg.py у папці utils, gen_alg_graph_cifar10.py, gen_alg_model.py у папці models. Також було додано декілька функцій у файл visualization.py, що у папці utils.

Файл GenAlg.ipynb – місце входу користувача, де він може виконувати код та спостерігати результати його виконання.

Файл genalg.py – містить у собі реалізацію генетичного алгоритму та надбудови для дотренування моделей за допомогою градієнтного алгоритму.

Файл gen_alg_graph_cifar10.py – містить у собі граф моделі капсульної нейронної мережі заданої хромосомою для датасету cifar10.

Файл gen_alg_model.py – містить у собі реалізацію моделі капсульної нейронної мережі на основі графу з gen_alg_graph_cifar10.py.

3.9 Інтерфейс користувача

Після запуску JupyterLab у папці проєкту, користувачу належно відкрити файл GenAlg.ipynb.

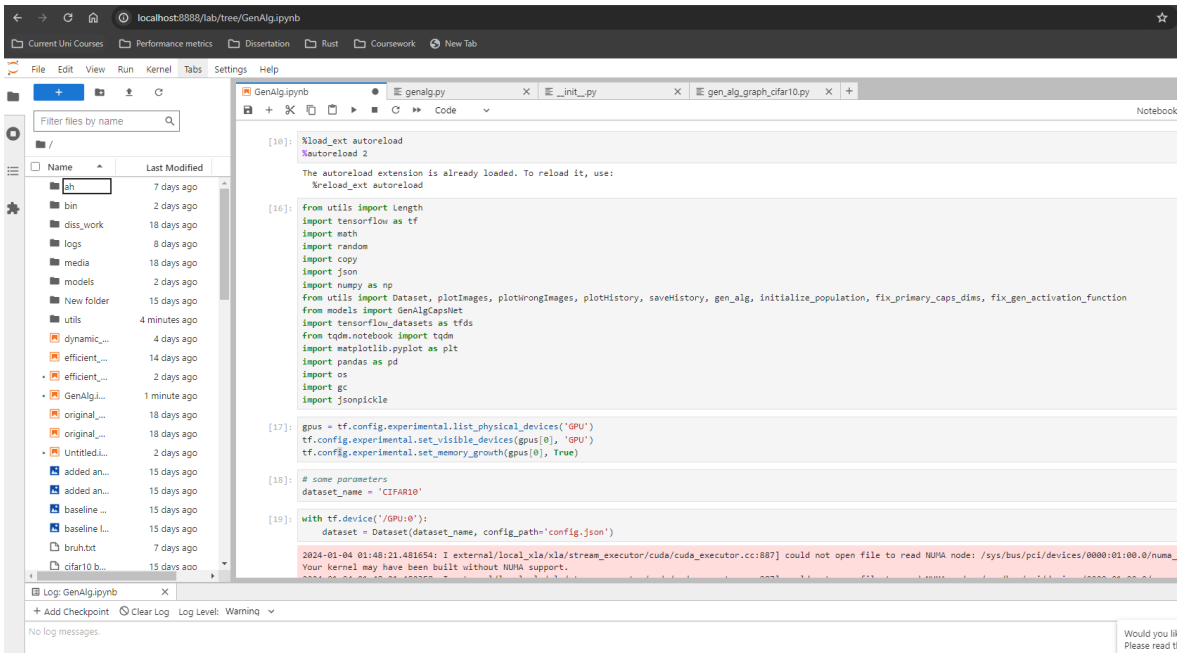

Рисунок 3.18 – Вид на програму у середовищі JupyterLab

Після цього він повинен виконати код у комірках. Якщо користувач бажає проініціалізувати перше покоління, то він повинен розкоментувати наступний блок й виконати його.

```
[20]: # uncomment the line under this one and run this if you wish to initialize your population
      #initialize_population(POPULATION_SIZE, dataset_name)
```

Рисунок 3.19 – Блок ініціалізації популяції

POPULATION_SIZE - змінна, яка визначає розмір популяції при застосування генетичного алгоритму. Її можна змінювати.

Для того щоби запустити генетичний алгоритм необхідно запустити наступні два блоки на рис. 3.20.

```
[21]: STARTING_GENERATION = 0
      GENERATIONS_TOTAL = 5

*[67]: gen_alg(dataset_name, STARTING_GENERATION, GENERATIONS_TOTAL, dataset)

782/782 [==============================] - 88s 113ms/step - loss: 0.5586 - Efficient_CapsNet_loss: 0.1923 - Generator_loss: 0.9344 - Efficient_CapsNet_accuracy: 0.7376 - val_loss: 0.3614 - val_Eff
icient_CapsNet_loss: 0.1909 - val_Generator_loss: 0.4351 - val_Efficient_CapsNet_accuracy: 0.7435 - lr: 3.9187e-04
Epoch 10/10
781/782 [==============================>.] - ETA: 0s - loss: 0.5457 - Efficient_CapsNet_loss: 0.1834 - Generator_loss: 0.9243 - Efficient_CapsNet_accuracy: 0.7532
Epoch 10: val_Efficient_CapsNet_accuracy improved from 0.74350 to 0.74660, saving model to bin/genalg/CIFAR10/generation_4/gen_alg_capsnet_27_new_train.h5
782/782 [==============================] - 88s 113ms/step - loss: 0.5457 - Efficient_CapsNet_loss: 0.1834 - Generator_loss: 0.9243 - Efficient_CapsNet_accuracy: 0.7532 - val_loss: 0.3528 - val_Eff
icient_CapsNet_loss: 0.1841 - val_Generator_loss: 0.4304 - val_Efficient_CapsNet_accuracy: 0.7466 - lr: 3.8012e-04
------------------------------CIFAR10 train------------------------------
Epoch 1/10
782/782 [==============================] - ETA: 0s - loss: 1.0053 - Efficient_CapsNet_loss: 0.4445 - Generator_loss: 1.4308 - Efficient_CapsNet_accuracy: 0.3202
Epoch 1: val_Efficient_CapsNet_accuracy improved from -inf to 0.44880, saving model to bin/genalg/CIFAR10/generation_4/gen_alg_capsnet_28_new_train.h5
782/782 [==============================] - 161s 190ms/step - loss: 1.0053 - Efficient_CapsNet_loss: 0.4445 - Generator_loss: 1.4308 - Efficient_CapsNet_accuracy: 0.3202 - val_loss: 0.6301 - val_Ef
ficient_CapsNet_loss: 0.3691 - val_Generator_loss: 0.6658 - val_Efficient_CapsNet_accuracy: 0.4488 - lr: 5.0000e-04
Epoch 2/10
782/782 [==============================] - ETA: 0s - loss: 0.7662 - Efficient_CapsNet_loss: 0.3506 - Generator_loss: 1.0602 - Efficient_CapsNet_accuracy: 0.4920
Epoch 2: val_Efficient_CapsNet_accuracy improved from 0.44880 to 0.55410, saving model to bin/genalg/CIFAR10/generation_4/gen_alg_capsnet_28_new_train.h5
782/782 [==============================] - 148s 189ms/step - loss: 0.7662 - Efficient_CapsNet_loss: 0.3506 - Generator_loss: 1.0602 - Efficient_CapsNet_accuracy: 0.4920 - val_loss: 0.5456 - val_Ef
ficient_CapsNet_loss: 0.3053 - val_Generator_loss: 0.6130 - val_Efficient_CapsNet_accuracy: 0.5541 - lr: 4.8500e-04
```

Рисунок 3.20 – Блоки запуску генетичного алгоритму

Для того щоби дотренувати 5 найліпших моделей необхідно запустити блок коду на рис. 3.21.

```
[90]: best_caps_nets = get_best_capsnets(caps_nets, 5)

[134]: best_caps_nets

[134]: [[0, 8, 0.8367999792098999],
        [4, 4, 0.8309000134468079],
        [2, 10, 0.8295999765396118],
        [3, 5, 0.8253999948501587],
        [2, 14, 0.8241000175476074]]

[138]: finish_training_best_caps_nets(best_caps_nets, dataset)
```

Рисунок 3.21 – Блоки коду для дотренування найкращих капсульних нейронних мереж знайдених генетичним алгоритмом

Висновки до розділу 3

У цьому розділі було наведено основні особливості різних підходів до реалізації капсульних нейронних мереж й було обрано підхід з капсульними мережами на основі механізму само-уваги. Було описано постановку датасет та постановку задачі структурно-параметричного синтезу капсульних нейронних мереж для задачі класифікації. Після цього було описано реалізацію гібридного алгоритму структурно-параметричного синтезу капсульних нейронних мереж із застосуванням генетичного та градієнтного алгоритмів. Гібридний алгоритм було застосовано, отримані результати були наведені у розділі.

РОЗДІЛ 4 РОЗРОБКА СТАРТАП-ПРОЄКТУ

Стартапи на сьогоднішній день є не тільки ключовим елементом економічного ландшафту, але і каталізатором для інновацій та змін в суспільстві. Стартапи є необхідним інгредієнтом для сталого розвитку сучасного суспільства, забезпечуючи інновації, економічний розвиток та вирішення складних проблем.

4.1 Опис ідеї проєкту

Таблиця 4.1 – Опис ідеї стартап-проєкту

Зміст ідеї	*Напрямки застосування*	*Вигоди для користувача*
Система дозволяє знаходити оптимальну топологію і параметри капсульної НМ для поставленої задачі	Автоматичне знаходження топології нейронної мережі	Не треба бути експертом у проєктування капсульних НМ
	Знаходження оптимальних параметрів мережі для вирішення поставленої задачі	Можливість мати найкращі результати для вирішення задачі

Аналіз потенційних техніко-економічних переваг ідеї (чим відрізняється від існуючих аналогів та замінників) порівняно із пропозиціями конкурентів передбачає:

- визначення переліку техніко-економічних властивостей та характеристик ідеї;
- визначення попереднього кола конкурентів (проєктів конкурентів) або товарів-замінників чи товарів-аналогів, що вже існують на ринку,

та проводиться збір інформації щодо значень техніко-економічних показників для ідеї власного проєкту та проєктів-конкурентів відповідно до визначеного вище переліку.

Проводиться порівняльний аналіз показників: для власної ідеї визначаються показники, що мають:

- гірші значення (W, слабкі);
- аналогічні (N, нейтральні) значення;
- кращі значення (S, сильні).

Таблиця 4.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проєкту

№ п/п	*Технічноекономічні характеристики ідеї*	*Товари/концепції конкурентів*			*W (слабка сторона)*	*N (нейтральна сторона)*	*S (сильна сторона)*
		Мій проєкт	*Немає*	*Немає*			
1.	Форма виконання	Надавання послуг					+
2.	Собівартість	Низька					+
3.	Функціонал	Широкий					+

Визначений перелік слабких, сильних та нейтральних характеристик та властивостей ідеї потенційного товару є підґрунтям для формування його конкурентоспроможності.

4.2 Технологічний аудит проєкту

В межах даного підрозділу було проведено аудит технології, за допомогою якої можна реалізувати ідею проєкту (технології створення товару). Визначення технологічної здійсненності ідеї проєкту передбачає аналіз таких складових (табл. 4.3):

1) За якою технологією буде виготовлено товар згідно ідеї проєкту?
2) Чи існують такі технології, чи їх потрібно розробити/доробити?
3) Чи доступні такі технології авторам проєкту?

Таблиця 4.3 – Аналіз технічної реалізації проєкту

№ п/п	*Ідея проєкту*	*Технології реалізації*	*Наявність технологій*	*Доступність технологій*
1	Створення системи синтезу оптимальної капсульної нейромережі	Мова Python	Наявна	Доступні
		Мова Rust	Відсутні	-
		Мова Go	Відсутні	-
Обрана технологія реалізації мова програмування Python				

За результатами аналізу таблиці зроблено висновок щодо можливості технологічної реалізації проєкту. Технологічним шляхом реалізації проєкту було обрано такі технології як Python через доступність та безкоштовність.

Визначення ринкових можливостей, які можна використати під час ринкового впровадження проєкту, та ринкових загроз, які можуть перешкодити реалізації проєкту, дозволяє спланувати напрями розвитку проєкту із

урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проєктів-конкурентів.

Проведено аналіз попиту: наявність попиту, обсяг, динаміка розвитку ринку (табл. 4.4).

Таблиця 4.4 – Попередня характеристика потенційного ринку стартап-проєкту

№ п/п	*Показники стану ринку (найменування)*	*Характеристика*
1	Кількість головних гравців, од	0
2	Загальний обсяг продаж, грн/ум.од	1 000 000 000 грн./ум. од
3	Динаміка ринку	Зростає
4	Наявність обмежень для входу	Немає
5	Специфічні вимоги до стандартизації та сертифікації	Немає
6	Середня норма рентабельності в галузі, %	20 %

Отже, можна зазначити, що ринок є привабливим до входження.

Проведено аналіз потенційних клієнтів стартап-проєкту (табл. 4.5).

Таблиця 4.5 – Характеристика потенційних клієнтів стартап-проєкту

Потреба, що формує ринок	*Цільова аудиторія*	*Відмінності у поведінці різних потенційних цільових груп клієнтів*	*Вимоги споживачів до товару*
Мати здатність використовувати оптимальну нейронну мережу для задачі класифікації	Інші стартапи, компанії, хоббісти	Стартапи будуть більше схильні до залучення послуги, через обмежену кількість ресурсів Компанії мають більше ресурсів, тому здатні витрачати їх на власні дослідження оптимальних нейромереж Хоббісти мають бажання роботи все власноруч	Нейронна мережа для поставленої замовником задачі повинна бути надана в обмежені терміни. Нейронна мережа повинна досягати дуже високих результатів.

Після визначення потенційних груп клієнтів було проведено аналіз ринкового середовища: складено таблиці факторів, що сприяють ринковому впровадженню проєкту, та факторів, що йому перешкоджають (табл. 4.6, 4.7).

Таблиця 4.6 – Фактори загроз

№ п/п	*Фактор*	*Зміст загрози*	*Можлива реакція компанії*
1	Конкуренція	Вихід на ринок подібних продуктів, що мають ліпші характеристики	Постійно інвестувати у покращення власних методів та робити ціну більш доступною
2	Зміна потреб замовників	Замовники може знадобитися нейромережа, яка повинна розв'язувати іншу задачу	Дослідження різних задач й нейромереж, які можуть їх вирішувати

Таблиця 4.7 – Фактори можливостей

№ п/п	*Фактор*	*Зміст можливості*	*Можлива реакція компанії*
1	Гнучкі ціни	Зменшення ціни товару задля збільшення попиту	Введення власних гнучких цін
2	Поява нових методів синтезу нейронних мереж	З'являються нові методи, що будуть швидше створювати нейромереж, які будуть отримувати кращі результати на поставлених задачах	Покращити власні алгоритми з урахуванням нових методів

Проведено аналіз конкуренції на ринку визначено вплив на підприємство (табл. 4.8).

Таблиця 4.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	*В чому проявляється дана характеристика*	*Вплив на діяльність підприємства*
1. Тип конкуренції - чиста	Дуже багато конкурентів	Якісно провести рекламну кампанію
2. За рівнем конкурентної боротьби - міжнародний	Компанії-конкуренти з інших країн	Робити сервіс доступним для якнайбільшої кількості країн
3. За галузевою ознакою - міжгалузева	Сервс може використовуватись для різних галузей	Розширення рекламної кампанії на різні сфери, де сервіс може бути використаним
4. Конкуренція за видами товарів - товарно-видова	Конкуренція за видами товарів	Уникати помилки конкурентів
5. За характером конкурентних переваг - нецінова	Вдосконалення технологій для зменшення вартості сервіса	Удосконалення моделі. Користуватися можливостями зменшити вартість, окрім як коли це може зменшити якість
6. За інтенсивністю - не марочна	Незначний вплив бренду	Проводити рекламну кампанію

Проведено SWOT-аналіз стартап-проєкту (табл. 4.9).

Таблиця 4.9 – SWOT-аналіз стартап-проєкту

Сильні сторони: Великий попит	Слабкі сторони: Дуже насичений ринок, необхідність застосування високої кількості обчислювальних ресурсів
Можливості: Оптимізація системи, знаходження нових методів, які можна у ній застосовувати.	Загрози: Створення схожих систем конкурентами

4.3 Аналіз ринкової стратегії проєкту

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: було проведено опис цільових груп потенційних споживачів (табл. 4.10).

Таблиця 4.10 – вибір цільових груп потенційних споживачів

№ п/п	*Опис профілю цільової групи потенційних клієнтів*	*Готовність споживачів сприйняти продукт*	*Орієнтовний попит в межах цільової групи*	*Інтенсивність конкуренції в сегменті*	*Простота входу у сегмент*
1	Стартапи	Висока	Високий	Низька	Проста
2	Великі компанії	Середня	Високий	Низька	Складна
3	Хоббісти	Низька	Середній	Низька	Середня
Які цільові групи було обрано: 1, 2, 3					

За результатами аналізу потенційних груп споживачів було обрано цільові групи, для яких буде запропоновано даний товар та визначено стратегію охоплення ринку - стратегію диференційованого маркетингу (компанія працює з декількома сегментами).

Для роботи в обраних сегментах ринку сформовано базову стратегію розвитку (табл. 4.11).

Таблиця 4.11 – Визначення базової стратегії розвитку

№ п/п	*Обрана альтернатива розвитку проєкту*	*Стратегія охоплення ринку*	*Ключові конкурентоспроможні позиції відповідно до обраної альтернативи*	*Базова стратегія розвитку*
1	Розвиток методів синтезу нейронних мереж	Диференційований маркетинг	Якість продукту	Стратегія спеціалізації

4.4 Розроблення маркетингової програми стартап-проєкту

Сформовано маркетингову концепцію товару, який отримає споживач. Для цього підсумовано результати попереднього аналізу конкурентоспроможності товар, визначено його ключові переваги над аналогами та можливості, які він надає (табл. 4.12).

Таблиця 4.12 – Визначення ключових переваг концепції потенційного товару

№ п/п	*Потреба*	*Вигода, яку пропонує товар*	*Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)*
1	Отримати оптимальну нейронну мережу	Синтезована нейронна мережа дуже гарно виконує поставлене завдання	Необхідно покращувати методі знаходження оптимальних мереж
2	Отримати мережу швидко	Процес синтезу може тривати набагато менше часу ніж робота експерта	Необхідно покращувати методи для їх пришвидшення
3	Отримати мережу дешево	Придбати оптимальну синтезовану мережу дешевше, ніж утворювати її за допомогою експертів	Необхідно покращувати методи для зменшення вартості мережі

Висновки до розділу 4

У цьому розділі було проведено аналіз реалізації стартап-проєкту на основі теми роботи. Було проведено опис ідеї стартап-проєкту, технологічний аудит проєкту, аналіз ринкової стратегії проєкту та розробка маркетингової програми стартап-проєкту.

ВИСНОВКИ

У роботі було виконано завдання структурно-параметричного синтезу капсульних нейронних мереж. Було проведено детальний огляд сучасних математичних реалізацій капсул та обрано оптимальний підхід під назвою “Капсульні нейронні мережі з маршрутизацією та основі самоуваги”. Було застосовано гібридний алгоритм, що складається з генетичного та алгоритму градієнтного спуску для знаходження оптимальної топології капсульної нейронної мережі та оптимальних її параметрів, що налаштовуються. Алгоритм оцінив 150 топологій капсульних нейронних мереж, 5 з яких були дотреновані й отримали результати кращі, за результати отримані нейромережою створеною вручну.

У подальшому варто удосконалювати алгоритм, щоби знаходити кращі версії генераторів, які зможуть виступати ефективним регуляризатором навчання нейромережі класифікатора. Для цього у алгоритмі, хромосоми можуть додатково оцінюватися за похибкою навчання генератора. Ще одним напрямком, завдяки якому можна було б покращити результати – це використання декількох генераторів, замість одного, що покращило би їх здатність навчатися.

Також необхідно зазначити досить низьку кількість осіб, які було проаналізовано, й кількість поколінь у генетичному алгоритмі. Це пов’язано з обмеженими обчислювальними ресурсами та часом. Оцінка одного покоління займала 12 годин. Цей час можна було б лінійно скоротити за допомогою розподілення процесу навчання на декілька відеокарт.

Результати роботи прийнято до опублікування у журналі “Electronics and Control Systems”, 2024, №78, “ STRUCTURAL-PARAMETRIC SYNTHESIS OF CAPSULE NEURAL NETWORKS”, авторів Синєглазова В. М., Кудрєва Д. О.

ПЕРЕЛІК ПОСИЛАНЬ

1. Hinton G., Krizhevsky A., Wang S. Transforming Auto-Encoders. *Artificial Neural Networks and Machine Learning*: ICANN 2011, 21st International Conference on Artificial Neural Networks, Espoo, Finland, June 14-17, 2011, Proceedings, Part I. 44-51. 10.1007/978-3-642-21735-7_6.
2. Sabour S., Frosst N., Hinton G.E. Dynamic Routing Between Capsules. arXiv:1710.09829. DOI: https://doi.org/10.48550/arXiv.1710.09829
3. Edgar Xi, Selina Bing, and Yang Jin. Capsule network performance on complex data. arXiv: 1712.03480. DOI: https://doi.org/10.48550/arXiv.1712.03480
4. Dilin Wang and Qiang Liu. An optimization view on dynamic routing between capsules, 2018. URL: https://openreview.net/forum?id=HJjtFYJDf (дата звернення: 01.10.2023)
5. Jan Eric Lenssen, Matthias Fey, and Pascal Libuschewski. Group equivariant capsule networks. arXiv: 1806.05086. DOI: https://doi.org/10.48550/arXiv.1806.05086
6. Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with EM routing, 2018. URL: https://openreview.net/pdf?id=HJWLfGWRb (дата звернення: 01.10.2023)
7. Mohammad Taha Bahadori. Spectral capsule networks, 2018. URL: https://openreview.net/pdf?id=HJuMvYPaM (дата звернення: 01.10.2023)
8. Fabio De Sousa Ribeiro, Georgios Leontidis, and Stefanos D Kollias. Capsule routing via variational bayes. arXiv: 1905.11455. DOI: https://doi.org/10.48550/arXiv.1905.11455
9. Jindong Gu and Volker Tresp. Improving the robustness of capsule networks to image affine transformations. arXiv: 1911.07968. DOI: https://doi.org/10.48550/arXiv.1911.07968

10. Inyoung Paik, Taeyeong Kwak, and Injung Kim. Capsule networks need an improved routing algorithm. arXiv: 1907.13327. DOI: https://doi.org/10.48550/arXiv.1907.13327
11. Sai Raam Venkatraman, Ankit Anand, S Balasubramanian, and R Raghunatha Sarma. Learning compositional structures for deep learning: Why routing-by-agreement is necessary. arXiv: 2010.01488. DOI: https://doi.org/10.48550/arXiv.2010.01488
12. Adam Byerly, Tatiana Kalganova, and Ian Dear. No Routing Needed Between Capsules. arXiv: 2001.09136. DOI: https://doi.org/10.48550/arXiv.2001.09136
13. Jaewoong Choi, Hyun Seo, Suii Im, and Myungjoo Kang. Attention routing between capsules. arXiv: 1907.01750. DOI: https://doi.org/10.48550/arXiv.1907.01750
14. Yao-Hung Hubert Tsai, Nitish Srivastava, Hanlin Goh, and Ruslan Salakhutdinov. Capsules with inverted dot-product attention routing. arXiv: 2002.04764. DOI: https://doi.org/10.48550/arXiv.2002.04764
15. Dunlu Peng, Dongdong Zhang, Cong Liu, and Jing Lu. Bg-sac: Entity relationship classification model based on self-attention supported capsule networks. Appl. Sof Comput. 91, 106186 (2020).
16. Mazzia, V., Salvetti, F. & Chiaberge, M. Efficient-CapsNet: capsule network with self-attention routing. Sci Rep 11, 14634 (2021). DOI: https://doi.org/10.1038/s41598-021-93977-0
17. Vincent Dumoulin F and Francesco Visin. A guide to convolution arithmetic for deep learning. arXiv: 1603.07285. DOI: https://doi.org/10.48550/arXiv.1603.07285
18. Y. Sun, B. Xue, M. Zhang, G. G. Yen and J. Lv, Automatically Designing CNN Architectures Using the Genetic Algorithm for Image Classification. IEEE Transactions on Cybernetics, vol. 50, no. 9, pp. 3840-3854, Sept. 2020, doi: 10.1109/TCYB.2020.2983860.

ДОДАТОК А ЛІСТИНГ ПРОГРАМИ

genalg.py

```
from utils import Length
import tensorflow as tf
import math
import random
import copy
import json
import numpy as np
from utils import Dataset, plotImages, plotWrongImages, plotHistory, saveHistory
from models import GenAlgCapsNet
import tensorflow_datasets as tfds
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
import pandas as pd
import os
import gc
import jsonpickle
from statistics import median


INPUT_SIZE = 32
NEURAL_NET_EPOCHS_TRAINING = 20
GENERATIONS_TOTAL = 10
POPULATION_SIZE = 30
MUTATION_CHANCE = 0.2
CROSSOVER_CHANCE = 0.9

CONV_LAYER_ADD_CHANCE = 0.1
CONV_LAYER_REMOVE_CHANCE = 0.1
```

```
        MIN_CONV_OUTPUT_SIZE = 2
        MAX_CONV_OUTPUT_SIZE = 8
        CONV_LAYER_K = [3, 5, 7, 9]
        CONV_LAYER_S = [1, 2, 3, 4, 5]
        CONV_LAYER_F = [32, 64, 96, 128, 160, 192, 224, 256]
        CONV_ACTIVATION_FUNCTIONS = ['relu', 'sigmoid', 'leaky_relu']

        CAPS_OUTPUT_N = 10
        CAPS_OUTPUT_D = [8, 16, 32, 48, 64]
        #ADDITIONAL_CAPS_LAYER_D = [4, 8, 16, 32, 48, 64, 128]
        #ADDITONAL_CAPS_LAYER_N = [4, 8, 16, 32, 48]

        DECONV_DENSE_XY_DIMS = [2, 4, 8]
        DECONV_DENSE_Z_DIMS = [128, 256, 512]
        DECONV_LAYER_S = 2
        DECONV_LAYER_K = [2, 4, 8]
        DECONV_LAYER_F = [32, 64, 96, 128, 160, 192, 224, 256]
        DECONV_ACTIVATION_FUNCTIONS = ['relu', 'sigmoid', 'leaky_relu']
        FINAL_CONV_LAYER_ACTIVATION_FUNCTIONS = ['tanh']
        FINAL_CONV_LAYER_K = [1, 2, 3, 4, 5]

        def is_valid_output_size(output):
            return output >= MIN_CONV_OUTPUT_SIZE and output <=
MAX_CONV_OUTPUT_SIZE



        def calculate_conv_layer_size_output(input_size, conv_layer):
            return math.ceil((input_size - conv_layer[0] + 1)/float(conv_layer[1]))



        def calculate_chromosome_size_output(input_size, conv_layers):
```

```
    current_output = input_size
    if len(conv_layers) == 0:
        return input_size
    for conv_layer in conv_layers:
        current_output = calculate_conv_layer_size_output(current_output, conv_layer)
    return current_output


class Specimen():
    def __init__(self, chromosome, accuracy):
        self.chromosome = chromosome
        self.accuracy = accuracy


class Chromosome():
    def __init__(self, conv_block, caps_block, gen_block):
        self.conv_block = conv_block
        self.caps_block = caps_block
        self.gen_block = gen_block


class Generation():
    def __init__(self, chromosomes):
        self.chromosomes = chromosomes


class CapsBlock():
    def __init__(self, output_fc_caps):
        #self.additional_fc_caps = additional_fc_caps
        self.output_fc_caps = output_fc_caps
```

```
class FCCapsGene():
    def __init__(self, N, D):
        self.N = N
        self.D = D


class GenBlock():
    def __init__(self, dense_layer, deconv_layers, final_conv_layer):
        self.dense_layer = dense_layer
        self.deconv_layers = deconv_layers
        self.final_conv_layer = final_conv_layer


class DenseLayerGene():
    def __init__(self, xy, z):
        self.xy = xy
        self.z = z


class DeconvLayerGene():
    def __init__(self, f, k, s, activation):
        self.f = f
        self.k = k
        self.s = s
        self.activation = activation


class FinalConvLayerGene():
    def __init__(self, k, activation):
        self.k = k
```

```
            self.activation = activation


        class ConvLayerBlock():
            def __init__(self, conv_layer_genes, primary_caps_gene):
                self.conv_layer_genes = conv_layer_genes
                self.primary_caps_gene = primary_caps_gene

            def get_block_conv_params(self):
                params = []
                for conv_layer_gene in self.conv_layer_genes:
                    layer_params = [conv_layer_gene.k, conv_layer_gene.s]
                    params.append(layer_params)
                if self.primary_caps_gene is not None:
                    params.append([self.primary_caps_gene.k, self.primary_caps_gene.s])
                return params

            def get_block_output_size(self):
                return calculate_chromosome_size_output(INPUT_SIZE,
self.get_block_conv_params())

            def is_possible_to_mutate_add_layer(self):
                current_output_size = self.get_block_output_size()
                room_to_spare = current_output_size - MIN_CONV_OUTPUT_SIZE


        class ConvLayerGene():
            def __init__(self, f, k, s, activation):
                self.f = f
                self.k = k
                self.s = s
```

```
            self.activation = activation



        class PrimaryCapsGene():
            def __init__(self, f, k, s, N, D):
                self.f = f
                self.k = k
                self.s = s
                self.N = N
                self.D = D



        def crossover(chromosome1, chromosome2):
            is_left_chromosome_first = random.random() < 0.5
            resulting_chromosome = Chromosome(None, None, None)
            resulting_conv_block = conv_crossover(chromosome1.conv_block,
chromosome2.conv_block, is_left_chromosome_first)
            resulting_chromosome.conv_block = resulting_conv_block
            resulting_caps_block = caps_crossover(chromosome1.caps_block,
chromosome2.caps_block, is_left_chromosome_first)
            resulting_chromosome.caps_block = resulting_caps_block
            resulting_gen_block = gen_crossover(chromosome1.gen_block, chromosome2.gen_block,
is_left_chromosome_first)
            resulting_chromosome.gen_block = resulting_gen_block
            return resulting_chromosome



        def conv_crossover(conv_block1, conv_block2, is_left_chromosome_first):
            desired_width = (conv_block1.get_block_output_size() +
conv_block2.get_block_output_size())/2
            output_conv_layer_block = ConvLayerBlock([], None)
```

```
        if is_left_chromosome_first:
            output_conv_layer_block.primary_caps_gene =
copy.deepcopy(conv_block2.primary_caps_gene)
            is_crossover_successful = produce_conv_layers_crossover(conv_block1, conv_block2,
output_conv_layer_block, desired_width)
        else:
            output_conv_layer_block.primary_caps_gene =
copy.deepcopy(conv_block1.primary_caps_gene)
            is_crossover_successful = produce_conv_layers_crossover(conv_block2, conv_block1,
output_conv_layer_block, desired_width)

        if not is_crossover_successful:
            output_conv_layer_block = copy.deepcopy(conv_block2 if is_left_chromosome_first
else conv_block1)

        return output_conv_layer_block


    # return True if could perform crossover on these 2 blocks
    def produce_conv_layers_crossover(conv_block_left, conv_block_right,
output_conv_layer_block, desired_width):
        is_left_turn = True
        left_layers = []
        right_layers = []
        resulting_layers = []
        index_left = 0
        index_right = len(conv_block_right.conv_layer_genes) - 2

right_layers.append(copy.deepcopy(conv_block_right.conv_layer_genes[len(conv_block_right.conv
_layer_genes) - 1]))
        while output_conv_layer_block.get_block_output_size() >=
MIN_CONV_OUTPUT_SIZE:
            resulting_layers = []
            resulting_layers.extend(left_layers)
```

```
            resulting_layers.extend(right_layers)
            output_conv_layer_block.conv_layer_genes = resulting_layers
            if output_conv_layer_block.get_block_output_size() <= desired_width:
                size_output = output_conv_layer_block.get_block_output_size()
                filter_size = output_conv_layer_block.primary_caps_gene.f
                output_conv_layer_block.primary_caps_gene.N = int(size_output * (filter_size / 8))
                output_conv_layer_block.primary_caps_gene.D = size_output * 8
                break
            if is_left_turn and index_left < len(conv_block_left.conv_layer_genes):
                left_gene_to_append =
copy.deepcopy(conv_block_left.conv_layer_genes[index_left])
                left_layers.append(left_gene_to_append)
                index_left += 1
            elif not is_left_turn and index_right > -1:
                right_gene_to_append =
copy.deepcopy(conv_block_right.conv_layer_genes[index_right])
                right_layers.insert(0, right_gene_to_append)
                index_right -= 1
            is_left_turn = not is_left_turn

        if output_conv_layer_block.get_block_output_size() < MIN_CONV_OUTPUT_SIZE:
            return False
        else:
            return True



    def caps_crossover(conv_block1, conv_block2, is_left_chromosome_first):
        output_caps_layer_block = CapsBlock(None)

        if is_left_chromosome_first:
            produce_caps_crossover(conv_block1, conv_block2, output_caps_layer_block)
        else:
```

```
            produce_caps_crossover(conv_block2, conv_block1, output_caps_layer_block)

        return output_caps_layer_block



    def produce_caps_crossover(caps_block_left, caps_block_right, output_caps_layer_block):
        #if caps_block_left.additional_fc_caps is not None:
        #    output_caps_layer_block.additional_fc_caps =
copy.deepcopy(caps_block_left.additional_fc_caps)
        output_caps_layer_block.output_fc_caps =
copy.deepcopy(caps_block_right.output_fc_caps)



    def gen_crossover(gen_block1, gen_block2, is_left_chromosome_first):
        output_gen_block = GenBlock(None, None, None)

        if is_left_chromosome_first:
            produce_gen_crossover(gen_block1, gen_block2, output_gen_block)
        else:
            produce_gen_crossover(gen_block2, gen_block1, output_gen_block)

        return output_gen_block

    def produce_gen_crossover(gen_block_left, gen_block_right, output_gen_layer_block):
        output_gen_layer_block.dense_layer = gen_block_left.dense_layer
        output_gen_layer_block.final_conv_layer = gen_block_right.final_conv_layer

        is_left_turn = True
        left_layers = []
        right_layers = []
        resulting_layers = []
        start_dim = output_gen_layer_block.dense_layer.xy
```

```
    deconv_layer_count = int(math.log2(INPUT_SIZE/start_dim))
    index_left = 0
    index_right = len(gen_block_right.deconv_layers) - 1

    while len(left_layers) + len(right_layers) != deconv_layer_count:
        if is_left_turn and index_left < len(gen_block_left.deconv_layers):
            left_layers.append(copy.deepcopy(gen_block_left.deconv_layers[index_left]))
            index_left += 1
        elif not is_left_turn and index_right > -1:
            right_layers.insert(0, copy.deepcopy(gen_block_right.deconv_layers[index_right]))
            index_right -= 1
        is_left_turn = not is_left_turn

    output_layers = []
    output_layers.extend(left_layers)
    output_layers.extend(right_layers)
    output_gen_layer_block.deconv_layers = output_layers


def get_generation_dir(dataset, generation_index):
    return os.path.join('bin', 'genalg', f'{dataset}',f'generation_{generation_index}')


def get_specimen_path(generation_folder_path, model_index):
    return os.path.join(generation_folder_path, f'specimen_{model_index}.json')


def write_specimen_to_json(specimen, dataset, generation_index, model_index):
    curpath = os.path.abspath(os.curdir)
    specimen_json = jsonpickle.encode(specimen)
    generation_folder_path = get_generation_dir(dataset, generation_index)
```

```
            if not os.path.exists(generation_folder_path):
                os.mkdir(generation_folder_path)
            with open(get_specimen_path(generation_folder_path, model_index), "w") as outfile:
                outfile.write(specimen_json)


        def read_specimen_from_json(dataset, generation_index, model_index):
            generation_folder_path = get_generation_dir(dataset, generation_index)
            specimen_path = get_specimen_path(generation_folder_path, model_index)
            specimen = read_object_from_file(specimen_path)
            return specimen


        def read_object_from_file(file_path):
            with open(file_path, 'r') as f:
                file_json = f.read()
                file_object = jsonpickle.decode(file_json)
            return file_object


        def are_specimen_chromosomes_equal(specimen1, specimen2):
            return jsonpickle.encode(specimen1.chromosome) ==
jsonpickle.encode(specimen2.chromosome)


        def get_all_file_specimens(dataset):
            dataset_directory = os.path.join('bin', 'genalg', f'{dataset}')
            generation_directories = [os.path.join(dataset_directory, x) for x in
os.listdir(dataset_directory) if 'generation' in x]
            all_specimen = []
            for generation_dir in generation_directories:
```

```
            specimens_in_gen = [read_object_from_file(os.path.join(generation_dir, x)) for x in
os.listdir(generation_dir) if 'specimen' in x and '.json' in x]
            all_specimen.extend(specimens_in_gen)


        return all_specimen



    def get_generation_dir(dataset, generation_index):
        dataset_directory = os.path.join('bin', 'genalg', f'{dataset}')
        generation_dir = os.path.join(dataset_directory, f'generation_{generation_index}')
        return generation_dir



    def get_specimens_in_generation(dataset, generation_index):
        generation_dir = get_generation_dir(dataset, generation_index)
        specimen_file_list = [x for x in os.listdir(generation_dir) if 'specimen' in x and '.json' in x]
        specimens_in_gen = [read_object_from_file(os.path.join(generation_dir, x)) for x in
specimen_file_list]
        specimen_count = len(specimen_file_list)
        specimens = [None] * specimen_count
        for i in range(len(specimens)):
            specimens[int(specimen_file_list[i].replace('specimen_', '').replace('.json', ''))] =
specimens_in_gen[i]
        return specimens



    def update_accuracy_if_specimen_present(dataset, specimen, generation_index,
specimen_index):
        file_specimens = get_all_file_specimens(dataset)
        for file_specimen in file_specimens:
            if are_specimen_chromosomes_equal(file_specimen, specimen) and
file_specimen.accuracy is not None:
                specimen.accuracy = file_specimen.accuracy
```

```
        write_specimen_to_json(specimen, dataset, generation_index, specimen_index)


def initialize_population(population_size, dataset_name):
    conv_layer_genes = [
        ConvLayerGene(32, 5, 1, 'leaky_relu'),
        ConvLayerGene(64, 3, 1, 'leaky_relu'),
        ConvLayerGene(64, 3, 1, 'leaky_relu'),
        ConvLayerGene(64, 3, 1, 'leaky_relu'),
        ConvLayerGene(128, 3, 2, 'leaky_relu')
    ]
    primary_caps_gene = PrimaryCapsGene(128, 9, 1, 32, 16)
    conv_layer_block = ConvLayerBlock(conv_layer_genes, primary_caps_gene)
    output_fc_caps = FCCapsGene(10, 16)
    caps_block = CapsBlock(output_fc_caps)
    dense_gene = DenseLayerGene(4, 256)
    deconv_layer_genes = [
        DeconvLayerGene(128, 4, 2, 'leaky_relu'),
        DeconvLayerGene(128, 4, 2, 'leaky_relu'),
        DeconvLayerGene(128, 4, 2, 'leaky_relu')
    ]
    final_conv_layer = FinalConvLayerGene(3, 'tanh')
    gen_block = GenBlock(dense_gene, deconv_layer_genes, final_conv_layer)
    starting_chromosome = Chromosome(conv_layer_block, caps_block, gen_block)
    chromosomes = [copy.deepcopy(starting_chromosome) for x in range(population_size)]
    for chromosome in chromosomes:
        for i in range(100):
            mutate_chromosome(chromosome)
    for i in range(population_size):
        specimen = Specimen(chromosomes[i], None)
        write_specimen_to_json(specimen, dataset_name, 0, i)
```

```
    def get_maximum_accuracy_from_model_history(history):
        max_val = max(list(history.history['val_Efficient_CapsNet_accuracy']))
        return max_val


    def gen_alg(dataset_name, starting_generation_index, max_gen_index, dataset):
        for generation_index in range(starting_generation_index, max_gen_index):
            generation_specimens = get_specimens_in_generation(dataset_name,
generation_index)
            all_file_specimens = get_all_file_specimens(dataset_name)
            for specimen_index in range(len(generation_specimens)):
                gc.collect()
                specimen = generation_specimens[specimen_index]
                update_accuracy_if_specimen_present(dataset_name, specimen, generation_index,
specimen_index)
                if specimen.accuracy is None:
                    model_train = GenAlgCapsNet(dataset_name, mode='train', verbose=False,
chromosome=specimen.chromosome, index = specimen_index,
generation_index=generation_index)
                    with tf.device('/GPU:0'):
                        history = model_train.train(dataset, initial_epoch=0)
                    model_accuracy = get_maximum_accuracy_from_model_history(history)
                    specimen.accuracy = model_accuracy
                    write_specimen_to_json(specimen, dataset_name, generation_index,
specimen_index)


            next_generation_specimens = create_next_generation(generation_specimens,
generation_index)
            for specimen_index in range(len(next_generation_specimens)):
                specimen = next_generation_specimens[specimen_index]
                write_specimen_to_json(specimen, dataset_name, generation_index + 1,
specimen_index)
```

```
            fix_primary_caps_dims(dataset_name, generation_index + 1)


        def create_next_generation(previous_generation_specimens, previous_generation_index):
            resulting_specimens = []
            while len(resulting_specimens) < len(previous_generation_specimens):
                first_specimen_index = random.choice(range(len(previous_generation_specimens)))
                second_specimen_index = random.choice([x for x in
range(len(previous_generation_specimens)) if x != first_specimen_index])
                first_specimen_accuracy =
previous_generation_specimens[first_specimen_index].accuracy
                second_specimen_accuracy =
previous_generation_specimens[second_specimen_index].accuracy
                chosen_specimen_index1 = first_specimen_index if first_specimen_accuracy >
second_specimen_accuracy else second_specimen_index

                first_specimen_index = random.choice(range(len(previous_generation_specimens)))
                second_specimen_index = random.choice([x for x in
range(len(previous_generation_specimens)) if x != first_specimen_index])
                first_specimen_accuracy =
previous_generation_specimens[first_specimen_index].accuracy
                second_specimen_accuracy =
previous_generation_specimens[second_specimen_index].accuracy
                chosen_specimen_index2 = first_specimen_index if first_specimen_accuracy >
second_specimen_accuracy else second_specimen_index

                if chosen_specimen_index1 == chosen_specimen_index2:
                    continue

                random_number = random.random()

                if random_number < CROSSOVER_CHANCE:
```

```
                resulting_chromosome =
crossover(previous_generation_specimens[chosen_specimen_index1].chromosome,
previous_generation_specimens[chosen_specimen_index2].chromosome)
                resulting_specimen = Specimen(resulting_chromosome, None)
                resulting_specimens.append(resulting_specimen)

            for new_specimen in resulting_specimens:
                random_number = random.random()
                if random_number < MUTATION_CHANCE:
                    mutate_chromosome(new_specimen.chromosome)

            return resulting_specimens


        def mutate_chromosome(chromosome):
            conv_block_layer_count = len(chromosome.conv_block.conv_layer_genes) + 1
            caps_block_layer_count = 1
            gen_block_layer_count = len(chromosome.gen_block.deconv_layers) + 2
            weights = [conv_block_layer_count, caps_block_layer_count, gen_block_layer_count]
            mutation = random.choices([mutate_randomly_conv_block,
mutate_randomly_caps_block, mutate_randomly_gen_block], weights=weights)
            mutation[0](chromosome)


        def mutate_randomly_conv_block(chromosome):
            conv_block_layer_count = len(chromosome.conv_block.conv_layer_genes)
            conv_block_primary_caps_count = 1
            weights = [conv_block_layer_count, conv_block_primary_caps_count]
            mutation = random.choices([mutate_randomly_conv_layers,
mutate_randomly_primary_caps], weights=weights)
            mutation[0](chromosome)
```

```
        def mutate_randomly_conv_layers(chromosome):
            grow_layers_indeces =
get_possible_to_mutate_number_of_conv_layers(chromosome.conv_block, grow_conv_layer)
            remove_layers_indeces =
get_possible_to_mutate_number_of_conv_layers(chromosome.conv_block, remove_conv_layer)
            actual_add_layers_chance = CONV_LAYER_ADD_CHANCE if
any(grow_layers_indeces) else 0.0
            actual_remove_layers_chance = CONV_LAYER_REMOVE_CHANCE if
any(remove_layers_indeces) else 0.0
            actual_mutate_layer_chance = 1 - actual_add_layers_chance - actual_add_layers_chance
            random_number = random.random()
            if random_number < actual_add_layers_chance:
                grow_index = random.choice(grow_layers_indeces)
                grow_conv_layer(chromosome.conv_block, grow_index)
            elif random_number > actual_add_layers_chance and random_number <
actual_add_layers_chance + actual_remove_layers_chance:
                remove_index = random.choice(remove_layers_indeces)
                remove_conv_layer(chromosome.conv_block, remove_index)
            else:
                update_index = random.choice(range(len(chromosome.conv_block.conv_layer_genes)))
                randomly_update_conv_layer(chromosome, update_index)



        def randomly_update_conv_layer(chromosome, index):
            f = random.choice(CONV_LAYER_F)
            chromosome.conv_block.conv_layer_genes[index].f = f
            k_options = []
            for k in CONV_LAYER_K:
                copy_conv_block = copy.deepcopy(chromosome.conv_block)
                copy_conv_block.conv_layer_genes[index].k =  k
                if is_valid_output_size(copy_conv_block.get_block_output_size()):
                    k_options.append(k)
```

```
            k = random.choice(k_options)
            chromosome.conv_block.conv_layer_genes[index].k = k
            s_options = []
            for s in CONV_LAYER_S:
                copy_conv_block = copy.deepcopy(chromosome.conv_block)
                copy_conv_block.conv_layer_genes[index].s = s
                if is_valid_output_size(copy_conv_block.get_block_output_size()):
                    s_options.append(s)
            s = random.choice(s_options)
            chromosome.conv_block.conv_layer_genes[index].s = s
            chromosome.conv_block.conv_layer_genes[index].activation =
random.choice(CONV_ACTIVATION_FUNCTIONS)
            if index == len(chromosome.conv_block.conv_layer_genes) - 1:
                output_size = chromosome.conv_block.get_block_output_size()
                chromosome.conv_block.primary_caps_gene.f = f
                chromosome.conv_block.primary_caps_gene.N = int(output_size * (f / 8))
                chromosome.conv_block.primary_caps_gene.D = output_size * 8


        def grow_conv_layer(conv_block, index):
            grown_layer = copy.deepcopy(conv_block.conv_layer_genes[index])
            conv_block.conv_layer_genes.insert(index, grown_layer)


        def remove_conv_layer(conv_block, index):
            layer_to_remove = conv_block.conv_layer_genes[index]
            conv_block.conv_layer_genes.remove(layer_to_remove)
            if index == len(conv_block.conv_layer_genes):
                output_size = conv_block.get_block_output_size()
                f = conv_block.conv_layer_genes[len(conv_block.conv_layer_genes) - 1].f
                conv_block.primary_caps_gene.f = f
```

```
            conv_block.primary_caps_gene.N = int(output_size * (f / 8))
            conv_block.primary_caps_gene.D = output_size * 8



        def get_possible_to_mutate_number_of_conv_layers(conv_block, mutator_func):
            index = len(conv_block.conv_layer_genes) - 1
            possible_index_list = []
            while index > -1:
                test_block = copy.deepcopy(conv_block)
                mutator_func(test_block, index)
                if is_valid_output_size(test_block.get_block_output_size()):
                    possible_index_list.append(index)
                index -= 1
            return possible_index_list



        def mutate_randomly_primary_caps(chromosome):
            f = random.choice(CONV_LAYER_F)
            chromosome.conv_block.primary_caps_gene.f = f

chromosome.conv_block.conv_layer_genes[len(chromosome.conv_block.conv_layer_genes) - 1].f
= f
            k_options = []
            for k in CONV_LAYER_K:
                copy_conv_block = copy.deepcopy(chromosome.conv_block)
                copy_conv_block.primary_caps_gene.k = k
                if is_valid_output_size(copy_conv_block.get_block_output_size()):
                    k_options.append(k)
            k = random.choice(k_options)
            chromosome.conv_block.primary_caps_gene.k = k
            s_options = []
            for s in CONV_LAYER_S:
```

```
            copy_conv_block = copy.deepcopy(chromosome.conv_block)
            copy_conv_block.primary_caps_gene.s = s
            if is_valid_output_size(copy_conv_block.get_block_output_size()):
                s_options.append(s)
        s = random.choice(s_options)
        chromosome.conv_block.primary_caps_gene.s = s
        output_size = chromosome.conv_block.get_block_output_size()
        chromosome.conv_block.primary_caps_gene.N = int(output_size * (f / 8))
        chromosome.conv_block.primary_caps_gene.D = output_size * 8


    def mutate_randomly_caps_block(chromosome):
        chromosome.caps_block.output_fc_caps.D = random.choice(CAPS_OUTPUT_D)


    def mutate_randomly_gen_block(chromosome):
        dense_layer_count = 1
        deconv_layer_count = len(chromosome.gen_block.deconv_layers)
        final_conv_layer_count = 1
        weights = [dense_layer_count, deconv_layer_count, final_conv_layer_count]
        mutation = random.choices([mutate_randomly_dense_layer,
mutate_randomly_deconv_layers, mutate_randomly_final_conv_layer], weights=weights)
        mutation[0](chromosome)


    def mutate_randomly_dense_layer(chromosome):
        xy = random.choice(DECONV_DENSE_XY_DIMS)
        z = random.choice(DECONV_DENSE_Z_DIMS)
        chromosome.gen_block.dense_layer.xy = xy
        chromosome.gen_block.dense_layer.z = z
        desired_deconv_layer_count = int(math.log2(INPUT_SIZE/xy))
```

```
            index = len(chromosome.gen_block.deconv_layers) - 1
            while desired_deconv_layer_count < len(chromosome.gen_block.deconv_layers):

chromosome.gen_block.deconv_layers.remove(chromosome.gen_block.deconv_layers[index])
                index -= 1
            index = len(chromosome.gen_block.deconv_layers) - 1
            while desired_deconv_layer_count > len(chromosome.gen_block.deconv_layers):

chromosome.gen_block.deconv_layers.append(copy.deepcopy(chromosome.gen_block.deconv_laye
rs[index]))



        def mutate_randomly_deconv_layers(chromosome):
            index = random.choice(range(len(chromosome.gen_block.deconv_layers)))
            deconv_layer = chromosome.gen_block.deconv_layers[index]
            k = random.choice(DECONV_LAYER_K)
            f = random.choice(DECONV_LAYER_F)
            activation = random.choice(DECONV_ACTIVATION_FUNCTIONS)
            deconv_layer.k = k
            deconv_layer.f = f
            deconv_layer.activation = activation



        def mutate_randomly_final_conv_layer(chromosome):
            k = random.choice(FINAL_CONV_LAYER_K)
            #activation = random.choice(FINAL_CONV_LAYER_ACTIVATION_FUNCTIONS)
            chromosome.gen_block.final_conv_layer.k = k
            chromosome.gen_block.final_conv_layer.activation = activation



        def fix_primary_caps_dims(dataset_name, generation_index):
            specimens = get_specimens_in_generation(dataset_name, generation_index)
```

```
        for specimen_index in range(len(specimens)):
            specimen = specimens[specimen_index]
            conv_block = specimen.chromosome.conv_block
            output_size = conv_block.get_block_output_size()
            f = conv_block.primary_caps_gene.f
            conv_block.primary_caps_gene.N = int(output_size * (f / 8))
            conv_block.primary_caps_gene.D = output_size * 8
            write_specimen_to_json(specimen, dataset_name, generation_index, specimen_index)


    # wow this is a real facepalm
    def fix_gen_activation_function(dataset_name, generation_index):
        specimens = get_specimens_in_generation(dataset_name, generation_index)
        for specimen_index in range(len(specimens)):
            specimen = specimens[specimen_index]
            gen_block = specimen.chromosome.gen_block
            gen_block.final_conv_layer.activation = 'tanh'
            write_specimen_to_json(specimen, dataset_name, generation_index, specimen_index)


    def get_genalg_accuracy_list(starting_gen, end_gen):
        accuracy_list = []
        for gen_index in range(starting_gen, end_gen + 1):
            gen_specimens = get_specimens_in_generation('CIFAR10', gen_index)
            for specimen_index in range(len(gen_specimens)):
                accuracy_list.append([gen_index, specimen_index,
gen_specimens[specimen_index].accuracy])
        return accuracy_list


    def get_accuracies_by_generation(caps_nets_list, gen_count):
```

```
    by_generation_list = [[], [], [], [], []]
    for caps in caps_nets_list:
        by_generation_list[caps[0]].append(caps[2])
    return by_generation_list


def plot_generations_trajectory(caps_nets_list):
    by_generation = get_accuracies_by_generation(caps_nets_list, 5)
    gen_stats = []
    for generation in by_generation:
        gen_median = median(generation)
        gen_max = max(generation)
        gen_min = min(generation)
        gen_stats.append([gen_min, gen_median, gen_max])
    gen_stats_arr = np.array(gen_stats)
    print
    X = range(5)
    # Plotting both the curves simultaneously
    plt.plot(X, gen_stats_arr[:,0], color='black', label='Мін')
    plt.plot(X, gen_stats_arr[:,1], color='red', label='Медіана')
    plt.plot(X, gen_stats_arr[:,2], color='blue', label='Макс')

    # Naming the x-axis, y-axis and the whole graph
    plt.xlabel("Покоління")
    plt.ylabel("Точність")
    plt.title("Еволюційна траєкторія")

    # Adding legend, which helps us recognize the curve according to it's color
    plt.legend()

    # To load the display window
```

```
    plt.show()


def sort_func(caps_net):
    return caps_net[2]


def get_best_capsnets(caps_nets, best_caps_net_count):
    best_caps_nets = copy.deepcopy(caps_nets)
    best_caps_nets.sort(reverse=True, key=sort_func)
    return best_caps_nets[:best_caps_net_count]


def finish_training_best_caps_nets(best_caps_nets, dataset):
    for best_caps_net in best_caps_nets:
        finish_training_capsnet(best_caps_net[0], best_caps_net[1], dataset, 'CIFAR10')


def finish_training_capsnet(generation, index, dataset, dataset_name):
    specimen = read_specimen_from_json(dataset_name, generation, index)
    model_train = GenAlgCapsNet(dataset_name,
                                mode='train',
                                verbose=True,
                                chromosome=specimen.chromosome,
                                index=index,
                                generation_index=generation,

custom_path=f'bin/genalg/CIFAR10/generation_{generation}/gen_alg_capsnet_{index}_new_train.
h5')
    model_train.load_graph_weights()
    with tf.device('/GPU:0'):
        history = model_train.train(dataset, initial_epoch=10)
```

```
saveHistory(history, f'Дотренування Покоління {generation} Модель {index}')
```