Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра математики

Кваліфікаційна робота
освітній ступінь – бакалавр

на тему: **«Класифікація зображень за допомогою мереж Колмогорова-Арнольда»**

Виконав: студент 4-го року навчання
освітньої програми «Прикладна математика»,
спеціальності 113 Прикладна математика

Вітіск Владислав Олександрович

Керівник: Швай Н. О.
к.ф.-м.н., доцент, доцент кафедри математики

Рецензент:
____

Кваліфікаційна робота захищена
з оцінкою ___________________________

Секретар ЕК___________________________
(підпис)

«______» _________________ 20___р.

Київ - 2024

Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра математики

ЗАТВЕРДЖУЮ
Зав.кафедри математики,
доцент, кандидат фіз.-мат. наук
______________ *Чорней Р.К.*
(підпис)
"____"________________ 2024

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
для кваліфікаційної роботи
студенту 4-го курсу, факультету інформатики
Вітіску Владиславу Олександровичу

Тема: «Класифікація зображень за допомогою мереж Колмогорова-Арнольда»
Зміст кваліфікаційної роботи:

Анотація
1. Вступ
2. Огляд основних означень, тверджень та принципів,
що пов'язані з мережеми Колмогорова-Арнольда
3. Тренування мереж для задачі класифікації зображень
4. Огляд отриманих результатів
Висновки
Список літератури

Дата видачі "____"__________ 2024 Керівник ______________
(підпис)
Завдання отримав ______________
(підпис)

Графік підготовки кваліфікаційної роботи до захисту

Графік узгоджено «______» ________________ 2024р.

№ з/п	Перелік робіт	Термін виконання етапу	Підпис наукового керівника	Дата ознайомлення наукового керівника	Примітка
1.	Отримання теми кваліфікаційної роботи.	25.09.2024			
2.	Ознайомлення з темою кваліфікаційної роботи.	08.10.2024			
3.	Розробка плану та структури роботи.	13.10.2024			
4.	Попередній огляд наукової літератури. Написання вступу та анотації.	14.11.2024			
5.	Робота з науковою літературою, опис основних означень.	14.12.2024			
6.	Дослідження властивостей мереж Колмогорова-Арнольда в контексті задачі класифікації зображень.	14.02.2024			
7.	Робота над текстовим оформленням теоретичної частини та отриманих результатів.	14.03.2025			
8.	Попередній аналіз кваліфікаційної роботи. Виправлення помилок.	14.04.2025			

№ з/п	Перелік робіт	Термін виконання етапу	Підпис наукового керівника	Дата ознайомлення наукового керівника	Примітка
9.	Попередній захист кваліфікаційної роботи.	23.05.2025			
10.	Захист кваліфікаційної роботи.	05.06.2025			

Науковий керівник ____________________________
(ПІБ)

Виконавець кваліфікаційної роботи ____________________________
(ПІБ)

Зміст

Анотація **6**

1 Вступ **7**

2 Огляд основних означень, тверджень та принципів, що пов'язані з мережами Колмогорова-Арнольда **9**

2.1 Машинне навчання та нейронні мережі 9

2.2 Мережі Колмогорова-Арнольда 11

2.3 Згорткові мережі 14

3 Тренування мереж для задачі класифікації зображень **17**

3.1 Постановка задачі точної класифікації зображень 17

3.2 Умови тренування та оцінки результатів 17

3.3 Опис наборів даних 18

3.4 Опис структур моделей 20

4 Огляд отриманих результатів **24**

4.1 Здатність KAN до роботи з зашумленими даними 24

4.2 Порівняння здатностей KAN до роботи з трнеувальними даними різного розміру 29

Висновки **35**

Література **36**

Додатки **38**

Додаток А. Код наборів даних 38

Додаток Б. Код моделей 46

Анотація

Метою даної кваліфікаційної роботи є дослідження мереж Колмогорова-Арнольда в контексті задач точної класифікації зображень. Здійснено тренування моделей на різних наборах даних за допомогою мови програмування Python та відповідних бібліотек. Оцінено ефективність та масштабованість відповідних нейронних мереж.

1 Вступ

У наш час, алгоритми машинного навчання швидко змінюють інформаційну сферу та впливають на роботу людей. Журналісти та дизайнери часто використовують генерацію зображень за текстом, офісні працівники спрощують роботу над формальною частиною звітів та документів за допомогою великих мовних моделей. В основі цих технологій лежать нейронні мережі, над основами яких працювали з 1940-их років, та розквіт яких припадає на другу половину 2010-их.

З того часу моделі покращували свої здібності, наближаючи згенеровані результати до близьких до людських. В основі поточних покращень результатів нейронних мереж лежать два основні фактори: збільшення розміру моделей та збільшення об'єму тренувальних даних, що пов'язані між собою. З кожним роком кількість параметрів нових мереж збільшуються(рис. 1), що вимагає розширення обчислювальних потужностей, що в свою чергу потребує як додаткових інвестицій у нове обладнання, так і витрат електроенергії, що негативно впливає на екологічний та кліматичний стан планети.

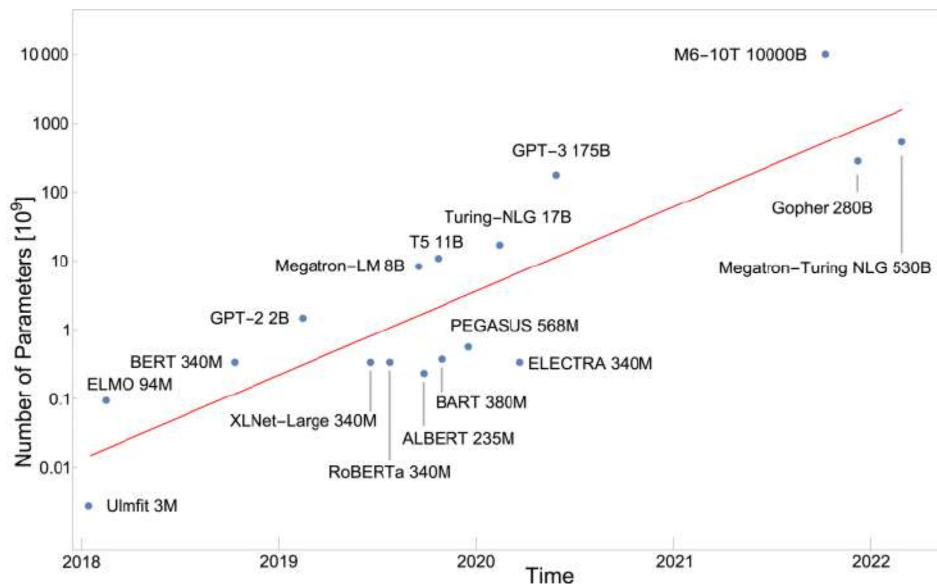

Рис. 1: Кількість параметрів нейронних мереж в залежності від року [1]

В 2-му кварталі 2024 група дослідників з Масачусетського Інституту технологій, Каліфорнійського інституту технологій та Північно-Східного університету опублікували роботу, що запропонувала рішення проблем класи-

чних моделей з багатошаровими перцептронами. Це рішення полягає в розробці нової нейронної мережі, в основі якої лежить теорема Колмогорова-Арнольда, що потенційно зменшить розміри моделей та об'єм тренувальних даних, і, відповідно, потреби в обчислювальних потужностях, а також може підвищити точність передбачень.

Перші результати авторів концепції підтверджують переваги мереж Колмогорова-Арнольда, втім, потребується подальша робота над дослідженням сфер застосувань нового алгоритму.

Метою даної роботи є:

1. Розглянути теоретичне підґрунтя мереж Колмогорова-Арнольда
2. Створити моделі для задач класифікації зображень.
3. Оцінити ефективність і здатність до масштабування цих мереж для точної класифікації зображень у різноманітних наборах даних, що сприятиме прогресу в машинному навчанні та комп'ютерному зорі.

Робота складається з трьох основних розділів.

Перший розділ присвячений теоретичному підґрунтю мереж Колмогорова-Арнольда. В ньому були розглянуті основні твердження та ідеї нейронних мереж загалом, та більш детальний огляд цього алгоритму.

Другий розділ присвячений створенню мереж Колмогорова-Арнольда для задачі класифікації зображень за допомогою мови програмування Python.

В третьому розділі було розглянуто результати моделей, отриманих в другому розділі.

2 Огляд основних означень, тверджень та принципів, що пов'язані з мережами Колмогорова-Арнольда

2.1 Машинне навчання та нейронні мережі

Машинне навчання - одна з гілок комп'ютерних наук, що зосереджена на розробці алгоритмів та структур, що здатні за допомогою тренувальної інформації створювати математичні моделі для передбачення результату на основі вхідних даних, структурно подібних до навчальних.

Найпростішою моделлю передбачень є наступне рівняння:

$$f = W * x, x \in \mathbb{R}^D, W \in \mathbb{R}^{C*D} \tag{1}$$

де x - вектор вхідних даних, а W - матриця ваг для отримання передбачення.

Для подальшого ускладнення моделі, що необхідне для прогнозування більш складних процесів, введемо поняття функції активації.

Означення 1. *Функція активації - це функція перетворення, яка визначає вихід нейронної мережі на основі вхідного сигналу.*

Ускладнивши найпростішу модель(1) додавши нелінійну функцію активації, наприклад $ReLU(x) = max(0, x)$, та ще одну матрицю ваг, ми отримаємо найпростішу двошарову нейронну мережу

$$f = W_2 * max(0, W_1 x), x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H*D}, W_2 \in \mathbb{R}^{C*H} \tag{2}$$

Використання функції активації в даному рівнянні(2) необхідне, щоб уникнути лінійного перетворення $W_1 * W_2$, що потрібно для здатності мережі виявляти закономірності. Функція, що буде використовуватись в нейронних мережах, має бути нелінійною, неперервною, диференційовною та зростаючою, що необхідно для того, щоб модель могла навчатися.

Прикладом функцій активацій, що використовуються в машинному навчанні, є:

- $ReLU(x) = max(0, x)$;

- $\sigma(x) = \dfrac{1}{1+e^{-x}}$;
- $tanh(x) = \dfrac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$.

Для прогнозування нейронним мережам необхідно мати здібність апроксимувати складні функції, що доведено теоремою Цибенка(Універсальної апроксимації), що стверджує, що будь-яку функцію багатьох змінних можна апроксимувати за допомогою двошарової нейронної мережі з довільною точністю за умови достатньої кількості ваг та правильного їх підбору.

Також, нейронні мережі можна представити в інший спосіб - у вигляді обчислювального графа(рис. 2). Вершинами цього графу є значення змінних, а ребрами - ваги W_n. Даний граф поділений на шари:

- Вхідний, що містить вершини, що представляють вхідні данні;
- Приховані, кожен з яких складається з результатів функції активації результатів попереднього шару;
- Вихідний, що містить результати передбачення моделі.

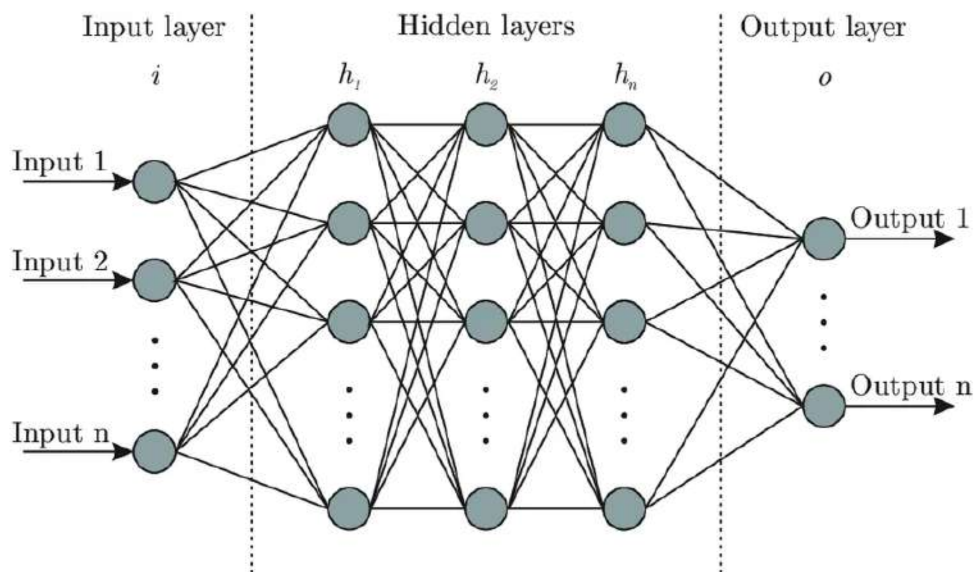

Рис. 2: Представлення нейронної мережі у вигляді графа[2]

На цьому зображенні ми бачимо, що кожна вершина з'єднана з кожною з сусідніх шарів.

Означення 2. *Нейронна мережа називається повнозв'язною, якщо кожна вершина обчислювального графу пов'язана(має ребра) з усіма вершинами наступного шару.*

Повнозв'язні нейронні мережі також мають іншу назву - багатошаровий перцептрон (Multi-Layer Perceptron, MLP), що отримали таку назву через кількість шарів та схожість роботи окремої вершини на спрощений принцип роботи нейрона(рис. 3).

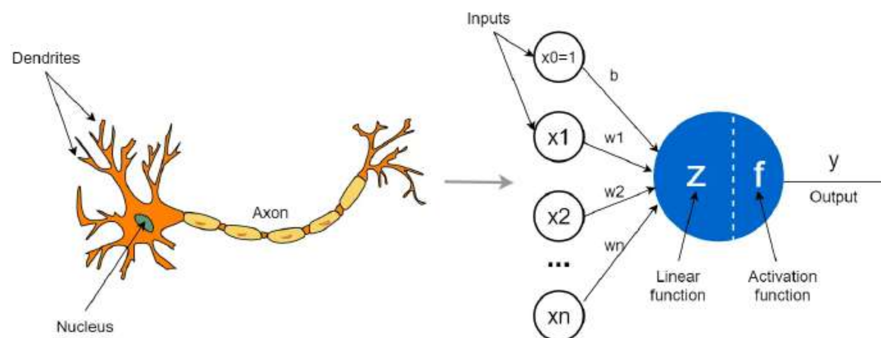

Рис. 3: Представлення нейронної мережі у вигляді графу [3]

MLP сьогодні є найпопулярнішим та найуживанішим варіантом нейронної мережі. Задача тренування багатошарового перцептрона зводиться до обрахунку ваг W_n, що дають найкращі результати передбачень. Для виконання цієї задачі використовується метод зворотного поширення помилки. Для застосування цього способу необхідно, щоб функція активації була диференційовною.

Алгоритм тренування складається з двох частин:

- Прямої - обчислення передбачення на основі поточних даних та ваг.
- Зворотної - стохастичний градієнтний спуск, що застосовується для покращення ваг за допомогою обчислення градієнтів помилки рухаючись від вихідного шару мережі до вхідного.

2.2 Мережі Колмогорова-Арнольда

Основою для нової моделі машинного навчання, що має конкурувати з Багатошаровим перцептроном, стала теорема Колмогорова-Арнольда, в честь

чого і була названа мережа.

Теорема (Колмогорова-Арнольда). *[4] Якщо f - неперервна функція багатьох змінних, то f можна записати як скінченну композицію функцій однієї змінної та бінарної операції додавання.*

$$f(x) = f(x_1, \cdots, x_n) = \sum_{q=1}^{2n+1} \Phi_q(\sum_{p=1}^{n} \phi_{q,p}(x_p)) \tag{3}$$

де, $\phi_{q,p} : [0, 1] \to \mathbb{R}$ *та* $\Phi_q : \mathbb{R} \to \mathbb{R}$

Мережі Колмогорова-Арнольда(KAN) пропонують відмовитись від лінійних ваг W_n, що є у MLP, зображених у вигляді ребер обчислювального графа, і фіксованих функцій активації, що є вершинами. Натомість, KAN пропонує перейти до натренованих одновимірних функцій активації, що будуть знаходитись на ребрах, а у вершинах буде відбуватись додавання результатів, отриманих після застосування натренованої функції активації, залишаючись повнозв'язною(рис. 4).

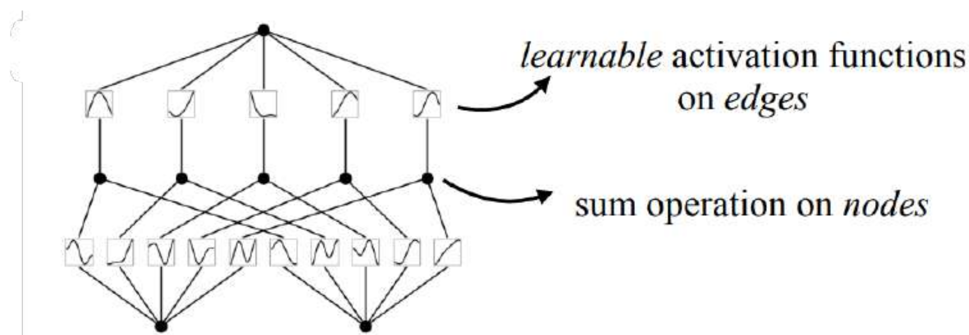

Рис. 4: Представлення мережі Колмогорова-Арнольда у вигляді графа[4]

Використовуючи теорему Колмогорова-Арнольда, замість вивчення однієї функції багатьох змінних, необхідно дослідити поліноміальне число одновимірних функцій, кожну з яких можна параметризувати за допомогою B-сплайнів, що і будуть навченими функціями активації. Використання таких ваг дає можливість інтерпретувати результати тренувань краще, ніж матриця з числами.

Подібно до нейронних мереж, KAN можна розділити на шари. Шар, що приймає n_{in} - мірні вхідні дані та повертає n_{out}-мірний результат, може бути

визначений як матриця функцій однієї змінної [4]:

$$\Phi = \{\phi_{q,p}\}, p = 1, 2, \cdots, n_{in}, q = 1, 2, \cdots, n_{out},$$

де функція $\phi_{q,p}$ має параметри, що підлягають навчанню.

Рівняння 3 є рівнянням двошарової мережі Колмогорова-Арнольда, де внутрішня функція є шаром з n вхідними значеннями та $2n+1$ вихідними, а зовнішня з $2n+1$ вхідними та одним результуючим. На рисунку 4 зображено відповідну KAN, де n=2, відповідно рухаючись знизу вгору шари містять 2, 5 та 1 вершини.

У загальному вигляді рівняння результату передбачення $\hat{y}$ n-шаровою мережею Колмогорова-Арнольда з вхідним x можна записати у вигляді наступного рівняння:

$$\hat{y} = (\Phi_{n-1} \circ \Phi_{n-2} \circ \cdots \circ \Phi_1 \circ \Phi_0)x \tag{4}$$

Задача тренування мережі Колмогорова-Арнольда полягає у підборі оптимальних функцій активації для апроксимації функції, якою ми можемо описати явище, результати якого ми хочемо передбачити.

Введемо наступні позначення[4]. Форма мережі Колмогорова-Арнольда позначається масивом цілих чисел

$$[n_0, n_1, \cdots, n_L],$$

де n_l - число вершин у l-тому шарі. Позначимо i-тий нейрон l-того шару за допомогою (l, i), а результат активації нейроном (l, i) через $x_{l,i}$. Функцію активації, що з'єднує (l, i) та $(l + 1, j)$ запишемо так:

$$\phi_{l,j,i}, \quad l = 0, \cdots, L-1, \quad i = 1, \cdots, n_l, \quad j = 1, \cdots, n_{l+1}.$$

Оскільки KAN повнозв'язна, а кожна вершина обчислювального графа є сумою операцій результатів функцій активації, то результат, отриманий на вершині $(l + 1, j)$, можна записати наступним чином

$$x_{l+1,j} = \sum_{i=1}^{n_l} \phi_{l,j,i}(x_{l,i}), \qquad j = 1, \cdots, n_{l+1}. \tag{5}$$

За допомогою рівняння 5, ми можемо представити рівняння 4, припускаючи, що розмірність вихідного шару дорівнює 1, у вигляді подібному до

рівняння 3:

$$\hat{y} = \sum_{i_{L-1}=1}^{n_{L-1}} \phi_{L-1,i_L,i_{L-1}} \left(\sum_{i_{L-2}=1}^{n_{L-2}} \cdots \left(\sum_{i_1=1}^{n_1} \phi_{1,i_2,i_1} \left(\sum_{i_0=1}^{n_0} \phi_{0,i_1,i_0}(x_{i_0}) \right) \right) \cdots \right), \quad (6)$$

З рівняння 6 ми бачимо, що всі операції KAN диференційовні, тому подібно до багатошарових перцептронів ми можемо застосувати метод зворотного поширення помилки для тренування мережі.

2.3 Згорткові мережі

Робота з повнозв'язними моделями для комп'ютерного зору ускладнюється великим розміром вектору вхідних даних, оскільки зображення складаються зі значної кількості пікселів, які мають, зазвичай, один або три кольорові канали. Відповідно, для здатності моделі показувати хороші результати ефективності у відповідній задачі необхідно, щоб модель мала значну кількість параметрів, що, в свою чергу, створює потребу у значній кількості тренувальних даних для підбору оптимальних ваг.

Вирішення цієї проблеми також було знайдено у фундаментальних принципах роботи головного мозку: Девід Гантер Г'юбел та Торстен Нільс Візел помітили, що нервова система опрацьовує не кожен фоторецептор окремо, а певні групи, що відповідають за різні зорові ділянки. На основі цього Куніхіко Фукушима запропонував концепцію неокогнітрона, що стала основою для згорткових нейронних мереж[5].

Згорткові нейронні мережі(Convolutional Neural Networks, CNN) - один з найпоширеніших видів моделі для задач комп'ютерного зору. Продовжуючи ідею шарів мережі, прості CNN можна поділити на наступні основні блоки: згорткові, агрегувальні та повнозв'язні шари.

Основним типом шару для CNN є згортковий. Саме він, за допомогою фільтру, формує "зорові ділянки" - рецептивні поля(рис. 5), що дає можливість розпізнавати певні шаблони, які допомагають розпізнати об'єкт на зображенні, в той же час зменшуючи розмірність виходу шару. Для налаштування цього шару використовуються наступні гіперпараметри:

- розмір фільтру - визначає розмір вікна згортки, що рухається зображення

зліва-направо, згори-вниз. Найчастіше використовується розмір 3x3 чи 5x5;

- кількість фільтрів - впливає на кількість матриць, що зі збільшенням можуть краще виявляти шаблони, але потребує більше ресурсів для навчання;
- крок фільтру - регулює відступ вправо від попередньої згортки;
- відступ - створює зовнішню пусту рамку зображення для збереження вхідного розміру на виході.

Тренування цього шару використовує метод зворотного поширення помилки та полягає в оптимальному підборі значень фільтрів.

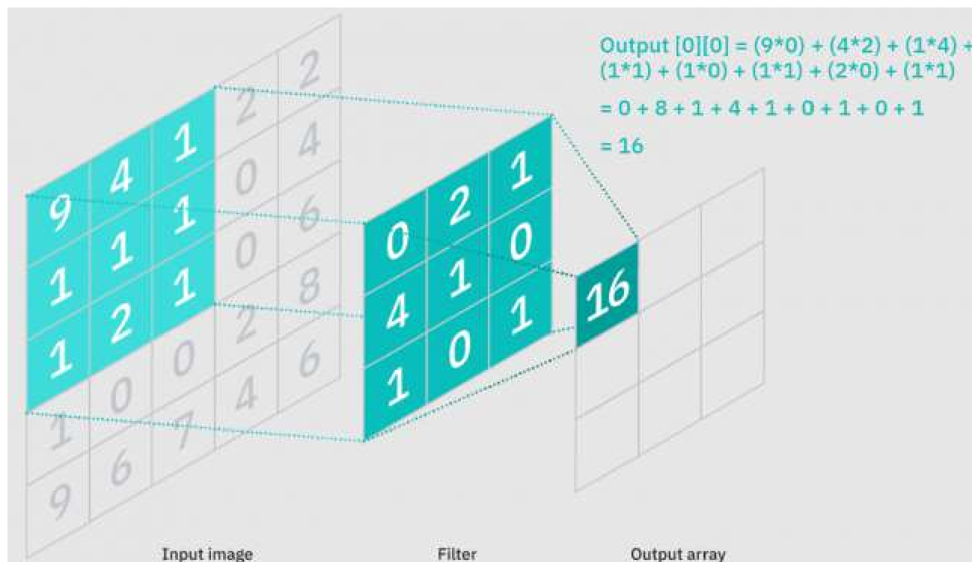

Рис. 5: Приклад обрахунку згортки[6]

Наступним видом є агрегаційний, що виконує перетворення над певною групою нейронів для кращого узагальнення та зменшення розмірності виходу цього шару. Найвживанішими агрегаціями є отримання максимального або середнього значення з групи, розмір якої може регулюватись.

Останнім видом є повнозв'язні шари, що працюють аналогічно до подібних в інших мережах.

Згорткова нейронна мережа, зазвичай, складається з кількох почергових згорткових та агрегаційних шарів, після яких застосовуються повнозв'язні.

Прикладами подібної архітектури є мережі сімейств LeNet(рис. 6), VGGNet чи AlexNet.

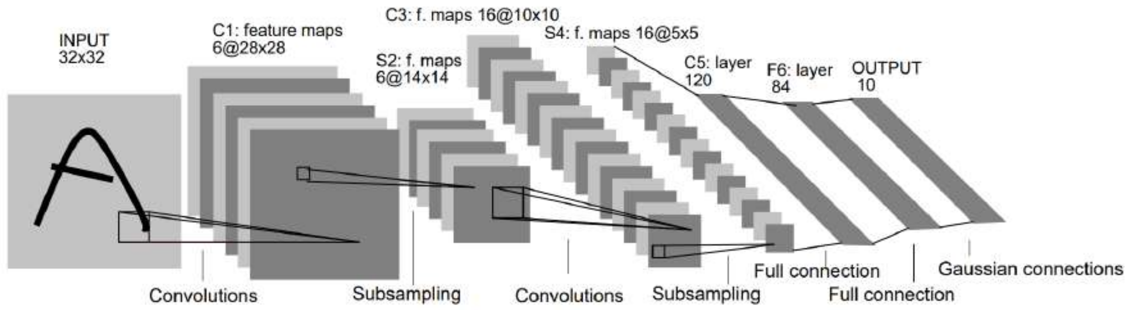

Рис. 6: Архітектура LeNet-5, що складається з двох згорткових, двох агрегаційних та трьох повнозв'язних шарів[7]

3 Тренування мереж для задачі класифікації зображень

3.1 Постановка задачі точної класифікації зображень

Задача точної класифікації зображень є основною проблемою комп'ютерного зору та полягає у розробці моделі для автоматичного віднесення вхідного зображення до одного з попередньо визначених класів з високою точністю. Для тренування розпізнавання категорії зображеного об'єкту використовується набір даних(датасет), що містить в собі зображення, співвіднесені до певного класу. Як для тренування, так і подальшого використання зображення мають мати однакову розмірність - однакову кількість кольорових каналів, кількість ступенів яскравості кожного каналу та піксельний розмір.

Для перевірки ефективності створеної моделі відбувається порівняння дійсних класів тестових зображень та передбачених, та обчислення різних метрик, що будуть описані у розділі про оцінку результатів.

У цій роботі буде проаналізовано та порівняно з класичним підходом кілька ключових характеристик функціонування KAN, а саме:

1. Вплив розміру моделі на продуктивність і здатність до узагальнення;
2. Стійкість мережі до зашумлених даних;
3. Залежність якості навчання від обсягу тренувальної вибірки та ефективність використання даних різного розміру.

3.2 Умови тренування та оцінки результатів

Для роботи з нейронними мережами в даній роботі було вирішено використовувати Python та бібліотеку torch.

Для оцінки здатностей моделей будуть використані наступні метрики:

- Точність на тестовій вибірці - частка правильно класифікованих спостережень відносно загальної їх кількості.
- Тренувальна функція втрат (Train Loss) - середнє значення функції втрат, обчислене для всіх прикладів у тренувальному наборі даних.

- Тестова функція втрат (Test Loss) - середнє значення функції втрат, обчислене для всіх прикладів у тестувальному наборі даних.

- Час тренування однієї епохи. Для підвищення надійності оцінки використовується медіанне значення часу проходження епохи, що мінімізує вплив поодиноких аномальних затримок, не пов'язаних безпосередньо з процесом навчання чи тестування.

Для порівнюваності результатів моделі тренувались за однакових умов:

- Усі моделі тренувались на одній і тій же самій відеокарті, з мінімізацією навантаження та кількості фонових процесів процесів.

- Для усіх моделей використовувались однакові гіперпараметри: оптимізатор, кількість епох, batch size.

3.3 Опис наборів даних

Датасети, розглянуті в цій роботі, можна розбити на 3 сімейства:

- MNIST-подібні;

- CIFAR-10-подібні;

- CIFAR-100.

MNIST - це набір даних, що містить у собі 70 тисяч зображень рукописних цифр від 0 до 9 розміром 28×28 пікселів з одним кольоровим каналом[8]. Усі зображення розподілені на 60 тисяч для тренувальної підвибірки та 10 тисяч - для тестувальної.

Для дослідження здатностей мереж Колмогорова-Арнольда до роботи з зашумленими даними, окрім стандартної версії датасету, буде використано три варіації з різними шумами[9]:

- MNIST з адитивним білим гаусівським шумом(NMNIST_AWGN);

- MNIST з розмиттям у русі(NMNIST_MB);

- MNIST зі зниженим контрастом та адитивним білим гаусівським шумом(NMNIST_AWGN_RC).

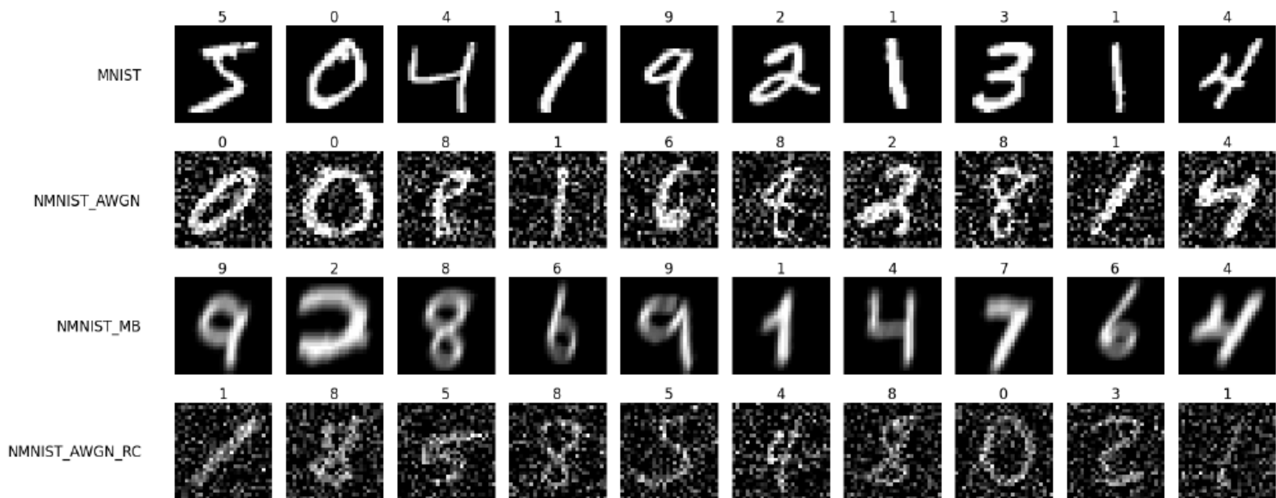

Рис. 7: Приклади зображень для наборів даних MNIST, NMNIST_AWGN, NMNIST_MB, NMNIST_AWGN_RC

CIFAR-10 та CIFAR-100 - набори даних, створені Алексом Крижевським, Вінодом Наїром і Джеффрі Хінтоном на основі датасету, що складається з 80 мільйонів маленьких зображень розміром 32×32 пікселі з трьома кольоровими каналами[10]. Кожен з цих наборів складається з 50 тисяч тренувальних записів та 10 тисяч тестувальних. Число поряд з назвою датасету відповідає за кількість класів, на які поділені всі зображення, тобто в даному випадку це 10 або 100 категорій.

CINIC-10 - набір даних, що також зібраний на основі того ж більшого датасету, але на відміну від CIFAR-10, має більшу кількість зображень: по 90 тисяч зображень для тренувальної, тестової та валідаційної частини[11]. В цій роботі буде використовуватись тільки тренувальна та тестова підмножини.

Для дослідження впливу кількості даних буде використано CIFAR-10-подібні датасети, до яких віднесемо сам CIFAR-10 та CINIC-10, щоб порівняти, як моделі працюють при збільшенні кількості даних. В той же час, за допомогою CIFAR-100, можна буде побачити вплив на метрики збільшення кількості класів порівняно з CIFAR-10 при однаковій кількості даних.

Рис. 8: Приклади зображень для наборів даних CIFAR-10, CINIC-10 та CIFAR-100

Для подальшої роботи всі дані були масштабовані так, щоб середнє значення та стандартне відхилення для кожного з каналів було 0.5. Після нормалізації дані було конвертовано у 16-бітний формат з плаваючою комою float16, що зменшує обсяг пам'яті та пришвидшує навчання й тестування моделей. Детальніше про реалізацію класів для роботи з наборами даних можна побачити в Додатку А.

3.4 Опис структур моделей

В даній роботі буде розглянуто 4 варіанти моделей:

- Повнозв'язна нейронна мережа (MLP)
- Мережі Колмогорова-Арнольда (KAN)
- Класична згорткова нейронна мережа (CNN)
- Згорткова мережа Колмогорова-Арнольда (CKAN)

Для дослідження впливу розміру, було натреновано для всіх датасетів кілька різних варіантів моделей за кількістю шарів, а відповідно і за кількістю параметрів, для кожного виду. Повнозв'язні мережі та KAN складаються з різної кількості лінійних шарів: оскільки перші мають складатися не менше

ніж з двох шарів, не рахуючи вхідного, то для них буде розглянуто варіант з двома або трьома шарами, і аналогічні розміри були використані для мереж Колмогорова-Арнольда. Тестування показало, що додавання шарів не призводило до подальшого підвищення точності.

Під час тренувань KAN було розглянуто також випадок з лише вхідним та вихідним шаром, втім результати виявились гіршими, ніж будь-яка більша повнозв'язна модель, але кращими, ніж для лінійної моделі, що складалася з аналогічних шарів.

Для роботи з мережами Колмогорова-Арнольда було використано реалізацію EfficientKAN[12], що в значній мірі повторює логіку роботи бібліотеки pykan, створеної З. Ліу та іншими, з покращеною роботою алгоритму. В подальших версіях та дослідженнях автори класичної реалізації перейняли покращення, втілені в EfficientKAN, для подальшого дослідження властивостей нового підходу. Через зміщення фокусу дослідників на інтерпретаційність мереж Колмогорова-Арнольда, саме модифікована версія початкового алгоритму стала основною для нових розробок інших авторів.

CKAN та CNN мають сталу згорткову частину, подібну до LeNet-5, та складається з двох згорткових шарів з ядром 5*5 та MaxPooling шаром за кожним з них. Для додаткового регулювання кількості параметрів було додано варіанти з різною кількістю повнозв'язних шарів, а саме, одним або двома. Додавання ще одного шару не приводило до жодних вагомих змін.

```
self.conv1 = nn.Conv2d(color_channels, 6, 5)
self.conv2 = nn.Conv2d(6, 16, 5)
self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2)
```

Лістинг 1: Згорткова частина CNN

```
self.conv1 = KAN_Convolutional_Layer(in_channels = color_channels,
                                     out_channels = 6,
                                     kernel_size = (5, 5),
                                     grid_size = grid_size,
                                     padding = (0, 0),
                                     device = "cuda"
                                     )

self.conv2 = KAN_Convolutional_Layer(in_channels = 6,
                                     out_channels = 16,
                                     kernel_size = (5, 5),
```

```
                                          grid_size = grid_size,
                                          padding = (0, 0),
                                          device = "cuda"
                                          )
self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2)
```

Лістинг 2: Згорткова частина CKAN

Для CKAN було обрано реалізацію згорток, створену Александром Боднером та командою, що базується на EficcientKAN[13]. В своїх дослідженнях автори розглядали різні варіанти застосування тренованих ваг:

- Класичні лінійні шари та згортки з тренованими функціями активації
- KAN лінійні шари та звичайні згортки
- KAN лінійні та згорткові шари

В даній роботі буде використано останній варіант.

Для спрощення створення моделей різного розміру, в EfficientKAN реалізовано можливість автоматичного створення моделі за допомогою переданого масиву, перший елемент якого відповідає за розмір вхідного шару, останній - за розмір вихідного класу, що в контексті цієї роботи означає кількість класів. Проміжні елементи відповідають за відповідні приховані шари.

Подібний підхід в цій роботі було додано для MLP, CNN та CKAN.

Для генерації простих повнозв'язних шарів в клас моделі було додано наступну функцію:

```
def __generate_stack(self, size):
    layers = []
    for i in range(len(size) - 1):
        layers.append(nn.Linear(size[i], size[i + 1]))
        if i < len(size) - 2:
            layers.append(nn.ReLU())
    return nn.Sequential(*layers)
```

Лістинг 3: Метод для генерації повнозв'язної частини

Реалізація для CKAN, в зв'язку з використанням KAN-лінійних шарів, відрізняється:

```
def __generate_stack(self, size):
    layers = []
```

```
    for i in range(len(size) - 1):
        layer = KANLinear(
            size[i],
            size[i + 1],
            grid_size = 5,
            spline_order = 3,
            scale_noise = 0.01,
            scale_base = 1,
            scale_spline = 1,
            base_activation = nn.SiLU,
            grid_eps = 0.02,
            grid_range = [0, 1],
        )
        layers.append(layer)
    return nn.Sequential(*layers)
```

Лістинг 4: Метод для генерації повнозв'язної частини з використанням теореми Колмогорова-Арнольда

Ще одним місцем для автоматизації став обрахунок розміру вхідного лінійного шару після згорткової частини CNN та CKAN, оскільки це значення буде відрізнятись в залежності від розміру сторони квадратного зображення.

```
def __calculate_flatten_size(self, image_size):
    after_fst_conv = image_size - 5 + 1
    after_fst_pool = after_fst_conv / 2
    after_snd_conv = after_fst_pool - 5 + 1
    after_snd_pool = after_snd_conv / 2
    self.flatten_size = int(after_snd_pool * after_snd_pool * 16)
```

Лістинг 5: Метод для обрахунку розміру вхідного лінійного шару

Після чого, отримане значення використовується в вище згаданій функції для побудови лінійних шарів для обох видів згорткових мереж.

Детальну реалізацію MLP, CNN та CKAN можна побачити в додатку Б.

4 Огляд отриманих результатів

4.1 Здатність KAN до роботи з зашумленими даними

У результаті тренування моделей MLP, KAN, CNN та CKAN різного розміру на датасеті MNIST було встановлено, що всі моделі демонструють високу точність, яка перевищує 97.93%. У загальному вигляді їх можна впорядкувати за зростанням предикативних здібностей у такій послідовності:

$$\text{MLP} \quad \rightarrow \quad \text{KAN} \quad \rightarrow \quad \text{CNN} \quad \rightarrow \quad \text{CKAN}$$

Втім, варто відзначити, що результати CNN дуже близькі до CKAN.

Подальші порівняння доцільно здійснювати попарно: MLP-KAN та CNN-CKAN, оскільки ці пари мають аналогічні структури.

В парі MLP-KAN, мережі Колмогорова-Арнольда проявили себе краще порівняно з відповідними за розмірами багатошаровими перцептронами, що можна побачити на Рис. 9. Окрім того, на рисунку помітно, що MLP повільніше досягає своєї максимальної точності, що підтверджується графіками тренувальних та тестувальних втрат(Рис. 10 та Рис. 11).

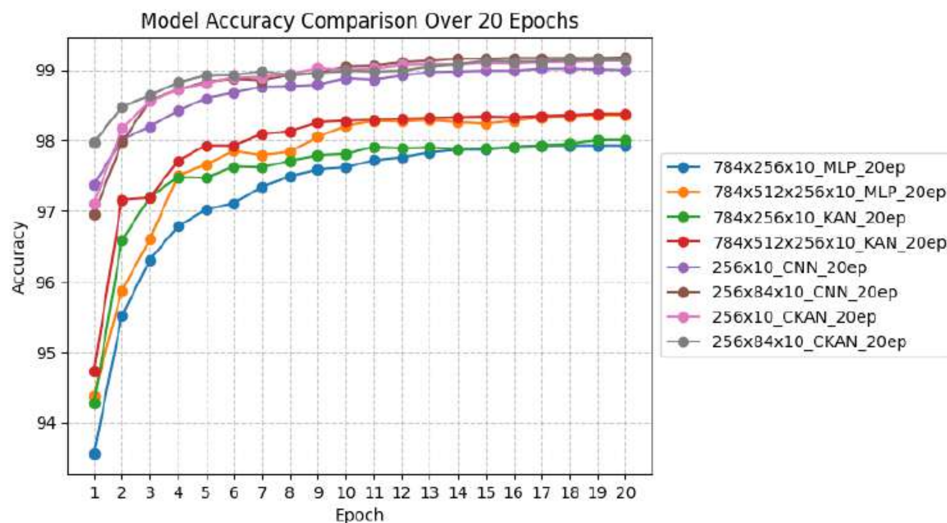

Рис. 9: Порівняння точності моделей на датасеті MNIST

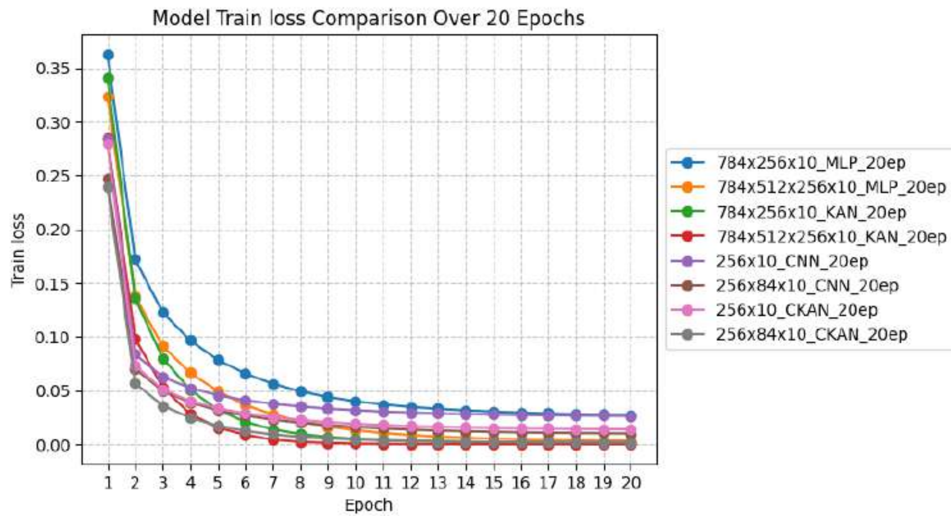

Рис. 10: Порівняння тренувальних втрат моделей на датасеті MNIST

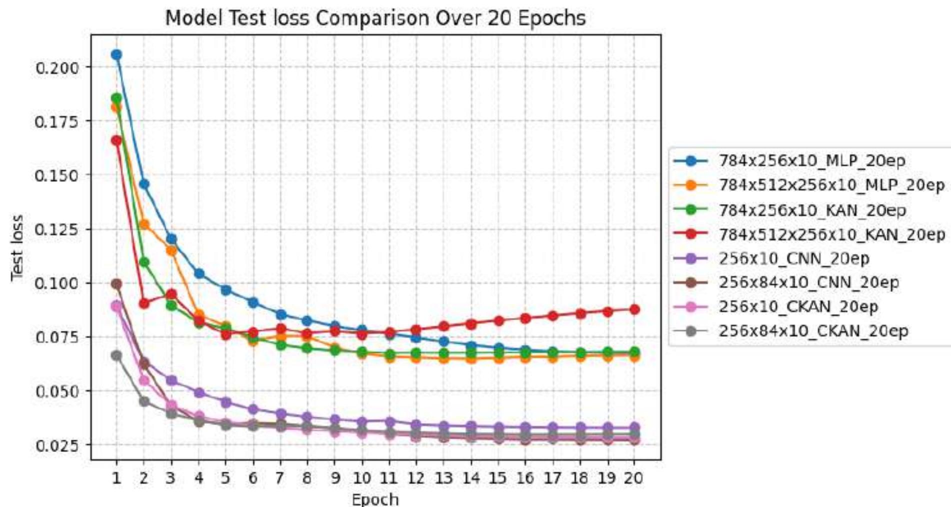

Рис. 11: Порівняння тестувальних втрат моделей на датасеті MNIST

В парі CNN-CKAN, модель, в якій використана теорема Колмогорова-Арнольда, проявила себе краще в випадках з одним повнозв'язним шаром, і трохи гірше, порівняно з класичним підходом, для двох повнозв'язних шарів. Попри це, поведінка точності моделей в різних епохах(Рис. 9) схожа. З графіків тренувальних втрат(Рис. 10) маємо, що CKAN швидше навчається, що частково підтверджується і поведінкою тестувальних втрат(Рис. 11).

Попри трохи вищу точність для двох випадків, значним недоліком CKAN є на порядок більший час проходження однієї епохи. Ця проблема буде присутня для всіх датасетів, та вона пов'язана з великою кількістю обрахунків,

потрібних для роботи зі згортками, що використовують функції активації, що навчаються.

Аналогічно, KAN виявився "повільнішим"відносно MLP, проте різниця виявилась не такою значною. Також було відмічено, що різниця збільшується непропорційно відносно кількості шарів.

Тип моделі	**Розмір FC шарів**	**Класичний підхід (MLP/CNN)**	**З застосуванням теореми Колмогорова-Арнольда (KAN/CKAN)**	**Різниця у %**
Повнозв'язна	784*256*10	6.79	9.24	36.08 %
	784*512*256*10	7.21	10.57	46.60 %
Згорткова	256*10	8.51	200.83	2259.93 %
	256*84*10	8.63	201.51	2234.99 %

Табл. 2: Порівняння медіанного часу моделей на MNIST

Використання наборів даних з шумами знизило точність для всіх моделей, але в різній мірі для різних датасетів(Табл. 3). В той же час, через трохи відмінний формат, в якому зберігаються дані, тренувальний час для MLP, KAN та CNN відчутно знизився, чого не було помічено для CKAN.

Модель	**Розмір FC шарів**	**Втрата точності відносно MNIST, %**		
		MNIST AWGN	MNIST MB	MNIST AWGN + RC
MLP	784*256*10	2.06	0.33	6.56
	784*512*256*10	1.77	0.11	4.59
KAN	784*256*10	2.01	-0.22	5.26
	784*512*256*10	1.51	-0.20	4.09
CNN	400*10	0.45	-0.09	2.26
	400*84*10	0.85	0.11	2.03
CKAN	400*10	0.63	0.13	2.41
	400*84*10	0.74	0.12	1.94

Табл. 3: Втрати точності моделей відносно базового MNIST на варіаціях з шумами: AWGN, MB та AWGN+RC

Застосування гаусового білого шуму знизило точність усіх моделей: вплив на KAN та MLP виявився найбільшим, і варто відмітити, що на найменшому розмірі KAN втратив відчутно більше точності, натомість з більшими моделями мережі Колмогорова-Арнольда проявили себе краще, ніж класичний варіант повнозв'язної нейронної мережі. CNN та CKAN показали значно ближчі

результати, і втрата точності не перевищувала 0.85%. Згорткова мережа, що використовує треновані функції активації, показала кращий результат тільки для моделі з двома повнозв'язними шарами.

Використання розмиття у русі, як одного з варіантів зашумлення, найменше вплинуло на результати всіх моделей. KAN змогли перевершити класичний MLP для усіх розмірів, та навіть показали зростання точності відносно базової версії MNIST. CNN, в свою чергу, в усіх випадках перевершили CKAN.

Додавання гаусового білого шуму в комбінації зі зниженням контрастності зображень найбільше погіршило точність усіх моделей. Подібно до ситуації з варіантом без зниження контрастності, мережі Колмогорова-Арнольда найменшого розміру показали найбільше падіння якості передбачень серед усіх моделей, але для більших розмірів результат значно покращився і вони змогли перевершити MLP. Пара CNN-CKAN одні відносно інших показали схожі результати до тих, що були помічені з першим варіантом шумів: класичні згорткові мережі втратили менше точності у випадках використання одного або трьох повнозв'язних шарів.

Варто відмітити, що для датасетів, де був присутній Гаусів білий шум, для KAN, CNN та CKAN було помічено відчутне перенавчання, що можна побачити на графіку тестувальних втрат(Рис. 12), чого не було, наприклад, для NMNIST_MB (Рис. 13).

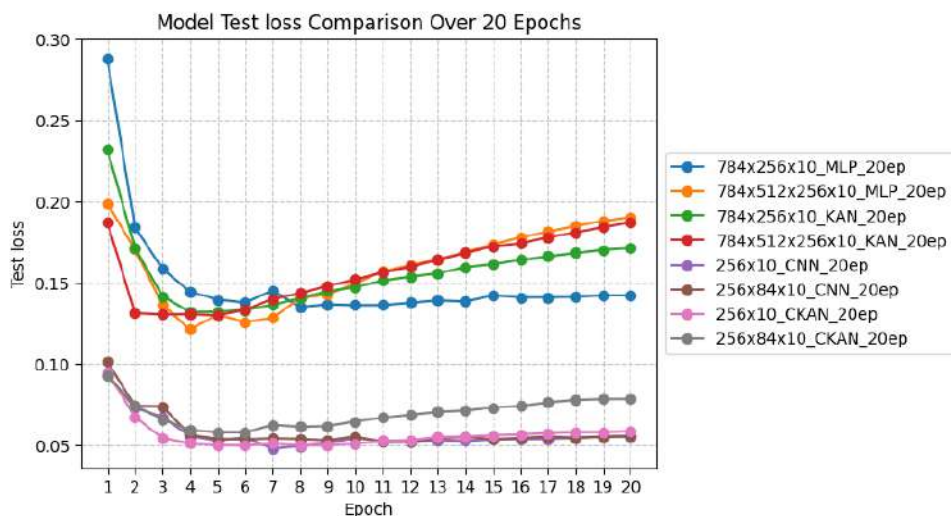

Рис. 12: Порівняння тестувальних втрат на датасеті NMNIST_AWGN

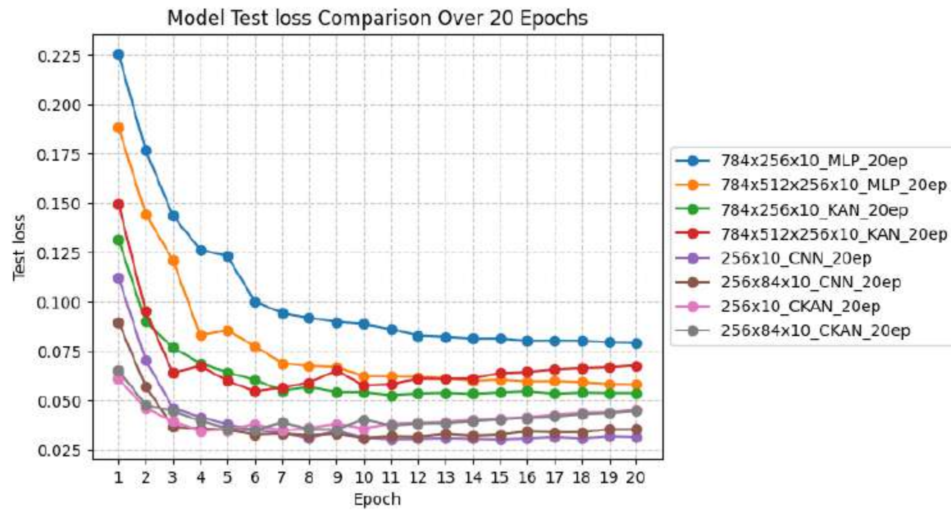

Рис. 13: Порівняння тестувальних втрат моделей на датасеті NMNIST_MB

Узагальнені результати можна побачити в таблиці 4, що містить у собі точність на 20 епосі та медіанний час проходження однієї епохи.

Модель	**Розмір FC шарів**	**MNIST**		**Mnist AWGN**		**Mnist MB**		**Mnist AWGN + RC**	
		Точність	Час	Точність	Час	Точність	Час	Точність	Час
MLP	784*256*10	97.93	6.79	95.91	1.98	97.61	1.95	91.51	2.02
	784*512*256*10	98.36	7.21	96.62	2.19	98.25	2.29	93.85	2.27
KAN	784*256*10	98.01	9.24	96.04	4.89	98.23	4.94	92.85	4.99
	784*512*256*10	98.38	10.57	96.89	7.27	98.58	7.47	94.36	7.45
CNN	256*10	99.00	8.51	98.55	2.79	99.09	3.24	96.76	3.35
	256*84*10	99.18	8.63	98.34	3.19	99.07	3.47	97.17	3.71
CKAN	256*10	99.13	200.83	98.51	201.62	99.00	202.40	96.74	198.02
	256*84*10	99.14	201.51	98.41	203.07	99.02	204.27	97.22	202.05

Табл. 4: Порівняння моделей за точністю та медіанним часом епохи на різних варіантах MNIST

Отже, мережі Колмогорова-Арнольда показали в середньому кращу точність, ніж MLP, але трохи поступаються в часі проходження однієї епохи. В той же час, CNN та CKAN, що перевершили попередні два види в здатності передбачень, показали схожі результати, хоча тривалість для CKAN на порядок виросла.

4.2 Порівняння здатностей KAN до роботи з трнеувальними даними різного розміру

Після тренування моделей на наборі даних CIFAR-10, порядок видів моделей за предикативними здатностями не змінився.

В цьому випадку, різниця між KAN та MLP стала більш вагомою на обох розмірах створених моделей, і, що варто відмітити, мережі Колмогорова-Арнольда проявили себе майже однаково, не зважаючи на додатковий шар, на відміну від класичного підходу, де це дало покращення точності(Рис. 14). Причиною цього стало сильне перенавчання KAN з трьома шарами, що можна побачити на графіках тренувальних(Рис. 15) та тестувальних втрат(Рис. 16).

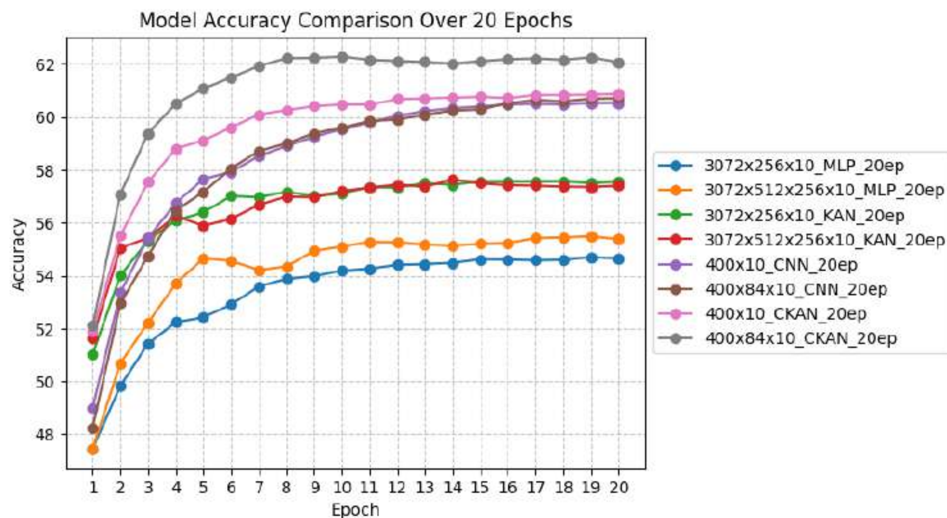

Рис. 14: Порівняння точності моделей на датасеті CIFAR-10

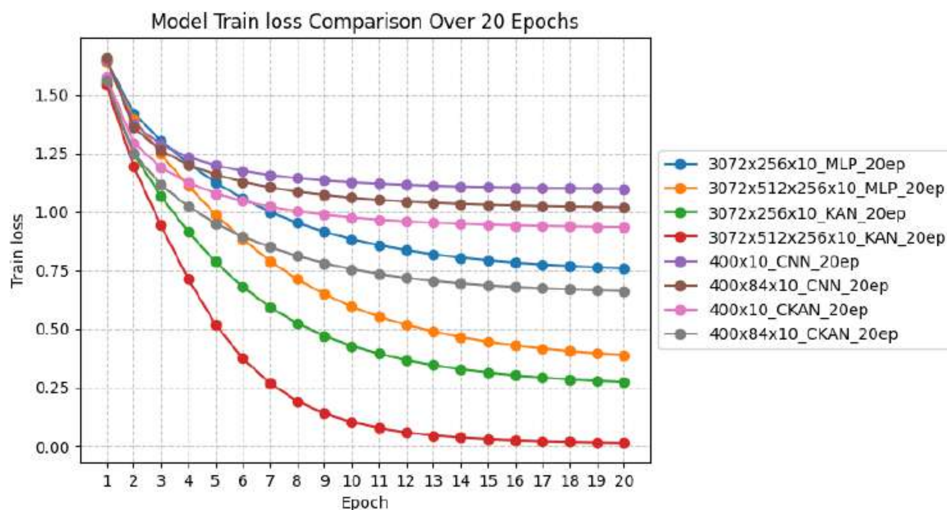

Рис. 15: Порівняння тренувальних втрат моделей на датасеті CIFAR-10

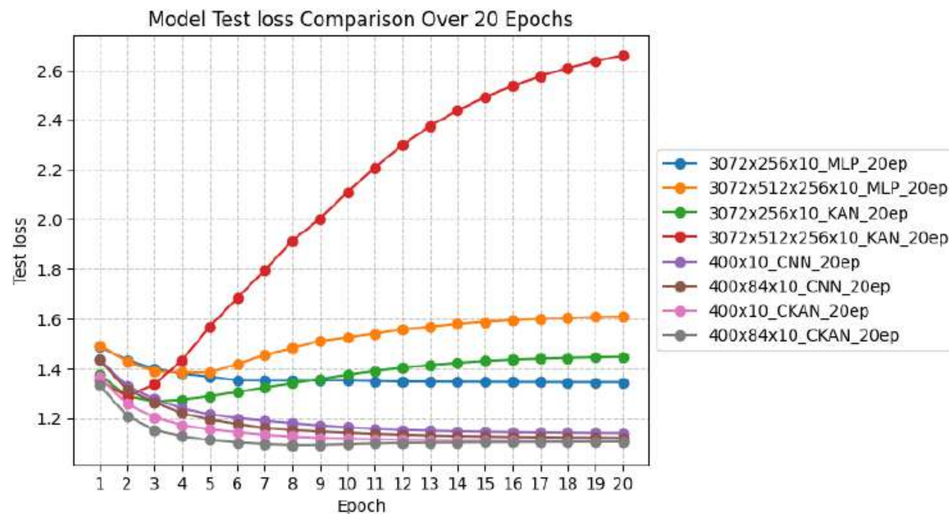

Рис. 16: Порівняння тестувальних втрат моделей на датасеті CIFAR-10

В парі CNN-CKAN ситуація виявилась подібною до попередніх двох видів: моделі, що використовують теорему Колмогорова-Арнольда, перевершили класичний підхід на усіх розмірах, хоч відрив цього разу виявився значно меншим. Втім, на відміну від KAN, згортковий варіант не проявив такого значного перенавчання.

Тренування на розширеному, відносно CIFAR-10, наборі даних CINIC-10 показало зменшення точності, та очікуване збільшення часу на проходження однієї епохи, що складається з тренування та тестування, оскільки набір тренувальних даних збільшився майже вдвічі, а тестувальна вибірка - в 9 разів.

В першій парі KAN проявили себе краще, особливо на варіанті з двома шарами, де мережі Колмогорова-Арнольда змогли отримати на 10% вищу точність відносно іншого методу, хоч різниця виявилась не такою великою, як для CINIC-10(Рис. 17). Втім, з графіків тестувальних втрат(Рис. 19) було помічено велике перенавчання для трьохшарової KAN, чого не було помічено для звичайної повнозв'язної нейронної мережі.

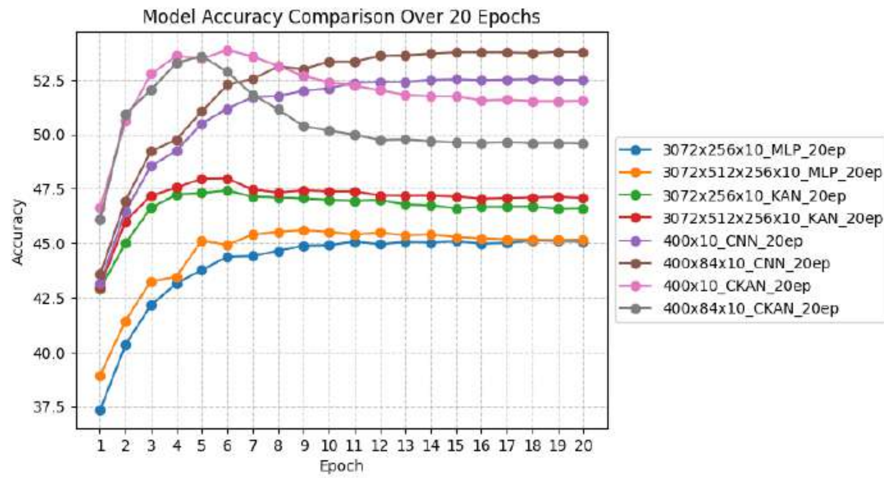

Рис. 17: Порівняння точностей моделей на датасеті CINIC-10

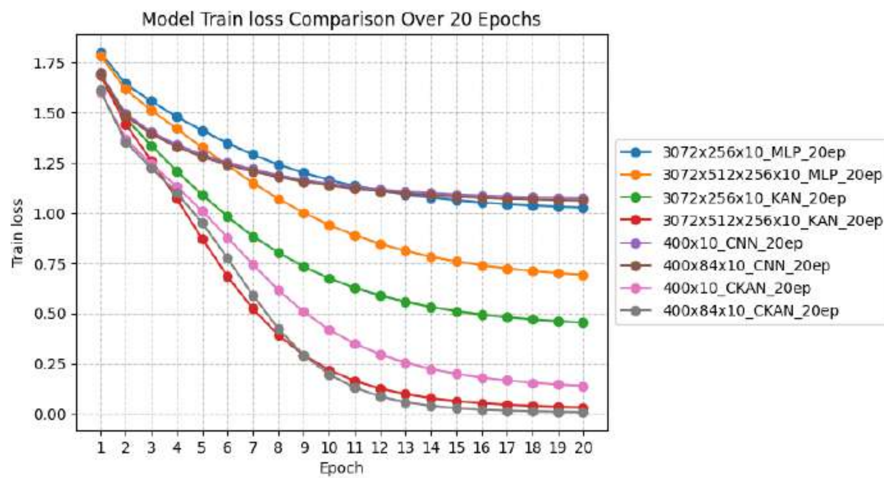

Рис. 18: Порівняння тренувальних втрат моделей на датасеті CINIC-10

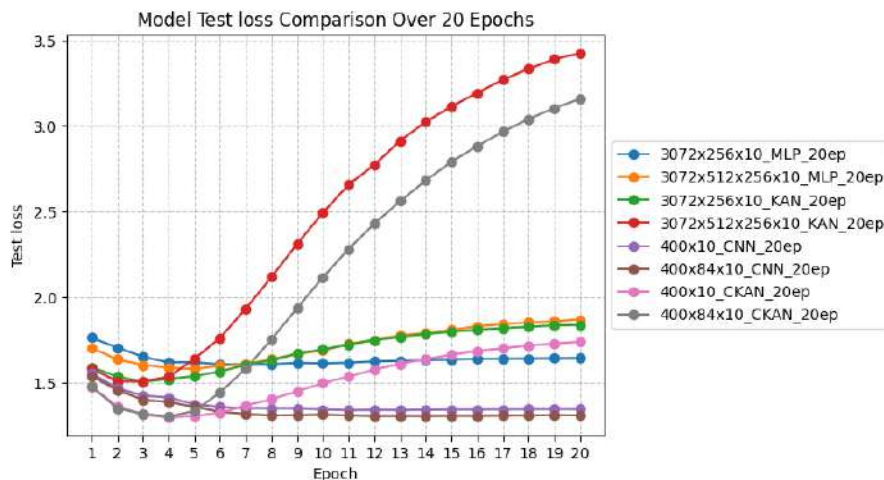

Рис. 19: Порівняння тестувальних втрат моделей на датасеті CINIC-10

Варто відмітити, що у даному випадку відносна різниця в часі стала меншою: для MLP вона коливалась в межах 26.69-26.96 секунд для різних розмірів, а для KAN - 30.89-34.73, з чого видно, що при збільшенні кількості шарів зміна в часі для різних методів не пропорційна.

В парі CNN-CKAN друга вперше сильно поступилась першій, при чому збільшення кількості повнозв'язних шарів не тільки не покращило точність, так ще й відчутно знизило точність через перенавчання, що можна побачити на рисунку 19. До 5-6 епохи CKAN показали гарну швидкість навчання та високу точність, що перевершувала точність відповідних за розміром CNN(Рис. 17).

В таблиці 5 зображені узагальнені результати для датасетів CIFAR-10 та CINIC-10.

Модель	**Розмір FC шарів**	**CIFAR-10**		**CINIC-10**	
		Точність	Час	Точність	Час
MLP	3072*256*10	54.65	6.10	45.07	26.69
	3072*512*256*10	55.38	6.36	45.16	26.96
KAN	3072*256*10	57.56	9.45	46.59	30.89
	3072*512*256*10	57.41	12.08	47.08	34.73
CNN	400*10	60.53	7.95	52.48	31.56
	400*84*10	60.72	8.46	53.78	32.21
CKAN	400*10	60.88	327.23	51.54	606.13
	400*84*10	62.08	325.47	49.58	602.50

Табл. 5: Порівняння моделей на CIFAR-10 та CINIC-10:
точність (Accuracy) і медіанний час епохи

Як можна побачити, застосування теореми Колмогорова-Арнольда дає більшу перевагу над стандартним підходом, коли кількість даних обмежена. У випадку, коли датасет збільшується, розрив звужується, або досягається гірший результат.

Тренування моделей на датасеті CIFAR-100 показало перевагу підходу тренованих ваг у випадку малої кількості тренувальних даних для кожного класу, як в цьому наборі даних: KAN для всіх розмірів отримали кращу точність порівняно з MLP, та в деяких випадках показали результат кращий

за CNN(Рис. 20). В свою чергу CKAN показали найкращі результати: навіть для найменшої моделі було отримано кращі результати, ніж для інших типів моделей.

Під час тренування було відмічене значне перенавчання для KAN, що складається з трьох шарів, що можна побачити на рисунку 21.

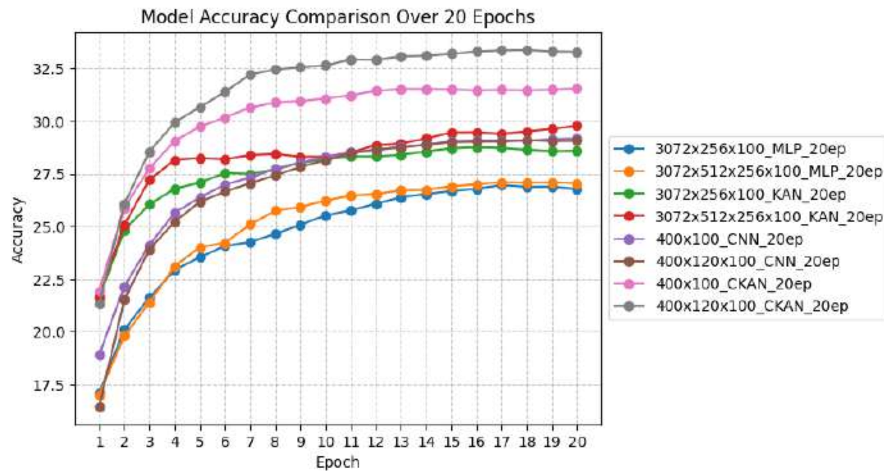

Рис. 20: Порівняння тестувальних втрат моделей на датасеті CIFAR-100

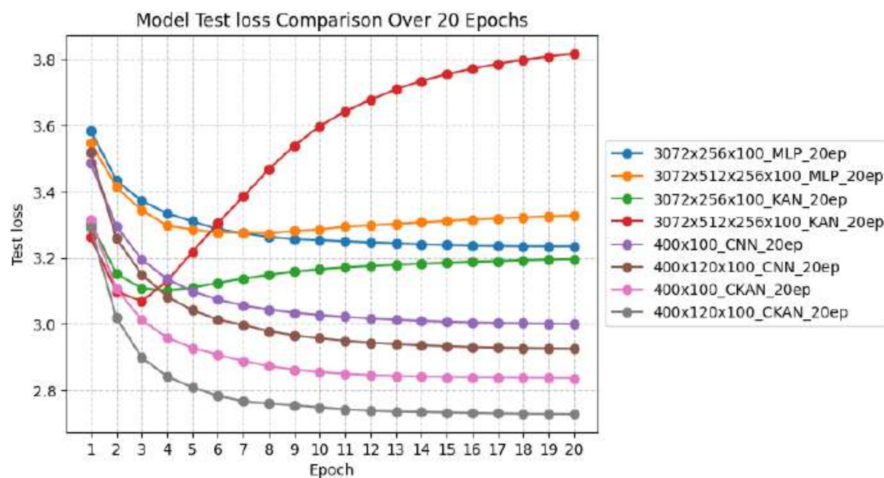

Рис. 21: Порівняння тестувальних втрат моделей на датасеті CIFAR-100

В таблиці 6 наведені узагальнені дані для цього набору даних.

Модель	**Розмір FC шарів**	**CIFAR-100**	
		Точність	Час
MLP	3072*256*100	26.77	6.54
	3072*512*256*100	27.05	7.03
KAN	3072*256*100	29.02	8.71
	3072*512*256*100	30.02	12.87
CNN	400*100	29.18	6.31
	400*84*100	29.07	6.61
CKAN	400*100	31.56	331.23
	400*84*100	33.29	326.63

Табл. 6: Результати моделей на CIFAR-100

Як бачимо, мережі Колмогорова-Арнольда та їх згортковий варіант проявляють себе тим краще, чим більш обмеженою є кількість доступних для тренування даних.

Висновки

У цій роботі було розглянуто мережі Колмогорова-Арнольда в контексті задач точної класифікації зображень. Було розглянуто теоретичні аспекти нової концепції машинного навчання, створено моделі за допомогою мови програмування Python та розглянуто результати, отримані в ході роботи.

Другий розділ присвячений теоретичному підґрунтю мереж Колмогорова-Арнольда. В ньому були розглянуті основні твердження та ідеї нейронних і згорткових мереж загалом, та більш детальний огляд нового алгоритму.

Третій розділ присвячений створенню мереж Колмогорова-Арнольда для задачі класифікації зображень за допомогою мови програмування Python та бібліотеки torch, в якому розглянуто задачу, досліджувані метрики, набори даних та види моделей.

В четвертому розділі було розглянуто результати, отримані моделями, в якому, узагальнюючи, можна побачити перевагу мереж, в основі яких лежить теорема Колмогорова-Арнольда, на малих, співрозмірних до класичного підходу, моделях в точності, або ж кращі результати при роботі з набором даних з обмеженою кількістю зображень на один клас. Також зміна тренованих ваг на треновані функції активації не вплинула на здатність до роботи з шумами. Попри певні переваги, мережі Колмогорова-Арнольда мають суттєвий недолік - великий час тренування, особливо для CKAN, в яких медіанна тривалість епохи виявилась на порядок вищою, ніж для CNN.

Література

[1] Göbel U. Big, bigger, giant. The rise of giant AI models. CONTACT Software Blog. URL: https://blog.contact-software.com/en/2022/10/gross-groesser-gigantisch-die-folgen-der-riesenmodelle-in-der-ki/ (date of access: 07.11.2024).

[2] Shukla L. Designing Your Neural Networks. Medium. URL: https://towardsdatascience.com/designing-your-neural-networks-a5e4617027ed (date of access: 08.11.2024).

[3] Pramoditha R. The Concept of Artificial Neurons (Perceptrons) in Neural Networks. Medium. URL: https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc (date of access: 08.11.2024).

[4] KAN: Kolmogorov–Arnold networks / Z. Liu et al. Massachusetts, 2024. 50 p. (Preprint. Massachusetts Institute of Technology; arXiv:2404.19756). URL: https://doi.org/10.48550/arXiv.2404.19756.

[5] Fukushima K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics. 1980. Vol. 36, no. 4. P. 193–202. URL: https://doi.org/10.1007/bf00344251 (date of access: 16.01.2025).

[6] Ajitesh Kumar. Real-World Applications of Convolutional Neural Networks. Analytics Yogi. URL: https://vitalflux.com/real-world-applications-of-convolutional-neural-networks/ (date of access: 16.01.2025).

[7] Gradient-based learning applied to document recognition / Y. LeCun et al. Proceedings of the IEEE. 1998. Vol. 86, no. 11. P. 2278–2324. URL: https://doi.org/10.1109/5.726791 (date of access: 16.01.2025).

[8] Papers with Code - MNIST Dataset. The latest in Machine Learning | Papers With Code. URL: https://paperswithcode.com/dataset/mnist (date of access: 18.05.2025).

[9] Learning Sparse Feature Representations Using Probabilistic Quadtrees and Deep Belief Nets / S. Basu et al. Neural Processing Letters. 2016. Vol. 45, no. 3. P. 855–867. URL: https://doi.org/10.1007/s11063-016-9556-4 (date of access: 18.05.2025).

[10] Krizhevsky A. Learning Multiple Layers of Features from Tiny Images. 2009.

[11] CINIC-10 Is Not ImageNet or CIFAR-10 / L. N. Darlow et al. 2018. URL: https://doi.org/10.48550/arXiv.1810.03505.

[12] GitHub - Blealtan/efficient-kan: An efficient pure-PyTorch implementation of Kolmogorov-Arnold Network (KAN). GitHub. URL: https://github.com/Blealtan/efficient-kan (date of access: 18.05.2025).

[13] Convolutional Kolmogorov-Arnold Networks / A. D. Bodner et al. 2024. URL: https://doi.org/10.48550/arXiv.2406.13155.

Додатки

Додаток А. Код наборів даних

Батьківський клас Loader(Loader.py):

```
class Loader:
    def __init__(self, dataset_name, image_size, channels, batch_size):
        self.name = dataset_name
        self.image_size = image_size
        self.channels = channels
        self.batch_size = batch_size
        self.__load_train_dataset()
        self.__load_test_dataset()

    def __load_train_dataset(self):
        self.train_dataset = None

    def __load_test_dataset(self):
        self.test_dataset = None

    def get_train_dataset(self):
        return self.train_dataset

    def get_test_dataset(self):
        return self.test_dataset

    def get_channels(self):
        return self.channels

    def get_batch_size(self):
        return self.batch_size

    def get_input_size(self):
        return self.image_size*self.channels*self.image_size

    def get_side_size(self):
        return self.image_size

    def get_dataset(self):
        return self.train_dataset, self.test_dataset, self.image_size * self
.image_size * self.channels, self.batch_size
```

Клас для набору даних MNIST(MNIST.py):

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from DatasetUtil.Loader import Loader

class MNIST(Loader):
    def __init__(self, batch_size):
        super().__init__("MNIST", 28, 1, batch_size)
        self.__load_train_dataset()
        self.__load_test_dataset()

    def __load_train_dataset(self):
        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean = [0.5], std = [0.5]),
            transforms.ConvertImageDtype(torch.float16)
        ])
        training_data = datasets.MNIST(
            root = "data",
            train = True,
            download = True,
            transform = transform,
        )
        self.train_dataset = DataLoader(training_data, batch_size = self.
    batch_size, pin_memory=True)

    def __load_test_dataset(self):
        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean = [0.5], std = [0.5]),
            transforms.ConvertImageDtype(torch.float16)
        ])
        test_data = datasets.MNIST(
            root = "data",
            train = False,
            download = True,
            transform = transform,
        )
        self.test_dataset = DataLoader(test_data, batch_size = self.
    batch_size, pin_memory=True)
```

Клас для набору даних MNIST з гаусовим білим шумом (NMNIST_AWGN.py):

```
from torch.utils.data import TensorDataset, DataLoader
from DatasetUtil.Loader import Loader
import scipy as sp
import torch

class NMNIST_AWGN(Loader):
    def __init__(self, batch_size):
        super().__init__("MNIST_AWGN", 28, 1, batch_size)
        self.loaded = sp.io.loadmat('data/MNIST-AWGN/mnist-with-awgn.mat')
        self.__load_train_dataset()
        self.__load_test_dataset()

    def __load_train_dataset(self):
        train_images = self.loaded['train_x']
        train_labels = self.loaded['train_y']
        tensor_train_x = torch.tensor(train_images, dtype = torch.float32).
    view(60000, 1, 28, 28)
        tensor_train_x = (tensor_train_x / 255.0 - 0.5) / 0.5
        tensor_train_x = tensor_train_x.to(torch.float16)
        tensor_train_y = torch.tensor(train_labels, dtype = torch.long)
        tensor_train_y = tensor_train_y.nonzero(as_tuple = True)[1]
        train_set = TensorDataset(tensor_train_x, tensor_train_y)
        self.train_dataset = DataLoader(train_set, batch_size = self.
    batch_size, shuffle = True, pin_memory=True)

    def __load_test_dataset(self):
        test_images = self.loaded['test_x']
        test_labels = self.loaded['test_y']
        tensor_test_x = torch.tensor(test_images, dtype = torch.float32).
    view(10000, 1, 28, 28)
        tensor_test_x = (tensor_test_x / 255.0 - 0.5) / 0.5
        tensor_test_x = tensor_test_x.to(torch.float16)
        tensor_test_y = torch.tensor(test_labels, dtype = torch.long)
        tensor_test_y = tensor_test_y.nonzero(as_tuple = True)[1]
        test_set = TensorDataset(tensor_test_x, tensor_test_y)
        self.test_dataset = DataLoader(test_set, batch_size = self.
    batch_size, pin_memory=True)
```

Клас для набору даних MNIST з розмиттям у русі (NMNIST_MB.py):

```
from torch.utils.data import TensorDataset, DataLoader
from DatasetUtil.Loader import Loader
import scipy as sp
import torch

class NMNIST_MB(Loader):
    def __init__(self, batch_size):
        super().__init__("MNIST_MB", 28, 1, batch_size)
        self.loaded = sp.io.loadmat('data/MNIST-MB/mnist-with-motion-blur.mat')
        self.__load_train_dataset()
        self.__load_test_dataset()

    def __load_train_dataset(self):
        train_images = self.loaded['train_x']
        train_labels = self.loaded['train_y']
        tensor_train_x = torch.tensor(train_images, dtype = torch.float32).view(60000, 1, 28, 28)
        tensor_train_x = (tensor_train_x / 255.0 - 0.5) / 0.5
        tensor_train_x = tensor_train_x.to(torch.float16)
        tensor_train_y = torch.tensor(train_labels, dtype = torch.long)
        tensor_train_y = tensor_train_y.nonzero(as_tuple = True)[1]
        train_set = TensorDataset(tensor_train_x, tensor_train_y)
        self.train_dataset = DataLoader(train_set, batch_size = self.batch_size, shuffle = True, pin_memory=True)

    def __load_test_dataset(self):
        test_images = self.loaded['test_x']
        test_labels = self.loaded['test_y']
        tensor_test_x = torch.tensor(test_images, dtype = torch.float32).view(10000, 1, 28, 28)
        tensor_test_x = (tensor_test_x / 255.0 - 0.5) / 0.5
        tensor_test_x = tensor_test_x.to(torch.float16)
        tensor_test_y = torch.tensor(test_labels, dtype = torch.long)
        tensor_test_y = tensor_test_y.nonzero(as_tuple = True)[1]
        test_set = TensorDataset(tensor_test_x, tensor_test_y)
        self.test_dataset = DataLoader(test_set, batch_size = self.batch_size, pin_memory=True)
```

Клас для набору даних MNIST з додаванням гаусового білого шуму та зменшеною контрастністю (NMNIST_AWGNRC.py):

```
from torch.utils.data import TensorDataset, DataLoader
from DatasetUtil.Loader import Loader
import scipy as sp
import torch


class NMNIST_AWGN_RC(Loader):
    def __init__(self, batch_size):
        super().__init__("MNIST_AWGN_RC", 28, 1, batch_size)
        self.loaded = sp.io.loadmat('data/MNIST-AWGN_and_RC/mnist-with-reduced-contrast-and-awgn.mat')
        self.__load_train_dataset()
        self.__load_test_dataset()

    def __load_train_dataset(self):
        train_images = self.loaded['train_x']
        train_labels = self.loaded['train_y']
        tensor_train_x = torch.tensor(train_images, dtype = torch.float32).view(60000, 1, 28, 28)
        tensor_train_x = (tensor_train_x / 255.0 - 0.5) / 0.5
        tensor_train_x = tensor_train_x.to(torch.float16)
        tensor_train_y = torch.tensor(train_labels, dtype = torch.long)
        tensor_train_y = tensor_train_y.nonzero(as_tuple = True)[1]
        train_set = TensorDataset(tensor_train_x, tensor_train_y)
        self.train_dataset = DataLoader(train_set, batch_size = self.batch_size, shuffle = True, pin_memory=True)

    def __load_test_dataset(self):
        test_images = self.loaded['test_x']
        test_labels = self.loaded['test_y']
        tensor_test_x = torch.tensor(test_images, dtype = torch.float32).view(10000, 1, 28, 28)
        tensor_test_x = (tensor_test_x / 255.0 - 0.5) / 0.5
        tensor_test_x = tensor_test_x.to(torch.float16)
        tensor_test_y = torch.tensor(test_labels, dtype = torch.long)
        tensor_test_y = tensor_test_y.nonzero(as_tuple = True)[1]
        test_set = TensorDataset(tensor_test_x, tensor_test_y)
        self.test_dataset = DataLoader(test_set, batch_size = self.batch_size, pin_memory=True)
```

Клас для набору даних CIFAR-10(CIFAR10.py):

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from DatasetUtil.Loader import Loader

class CIFAR10(Loader):
    def __init__(self, batch_size):
        super().__init__("CIFAR-10", 32, 3, batch_size)
        self.__load_train_dataset()
        self.__load_test_dataset()

    def __load_train_dataset(self):
        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
            transforms.ConvertImageDtype(torch.float16)
        ])
        training_data = datasets.CIFAR10(
            root = "data",
            train = True,
            download = True,
            transform = transform,
        )
        self.train_dataset = DataLoader(training_data, batch_size = self.
    batch_size, pin_memory=True)

    def __load_test_dataset(self):
        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
            transforms.ConvertImageDtype(torch.float16)
        ])
        test_data = datasets.CIFAR10(
            root = "data",
            train = False,
            download = True,
            transform = transform,
        )
        self.test_dataset = DataLoader(test_data, batch_size = self.
    batch_size, pin_memory=True)
```

Клас для набору даних CINIC-10(CINIC10.py):

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from DatasetUtil.Loader import Loader

class CINIC10(Loader):
    def __init__(self, batch_size):
        super().__init__("CINIC-10", 32, 3, batch_size)
        self.__load_train_dataset()
        self.__load_test_dataset()

    def __load_train_dataset(self):
        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean = [0.5, 0.5, 0.5], std = [0.5, 0.5,
    0.5]),
            transforms.ConvertImageDtype(torch.float16)
        ])
        image_folder = datasets.ImageFolder('data/cinic/train', transform =
    transform)
        self.train_dataset = DataLoader(image_folder, batch_size = self.
    batch_size, shuffle = True, pin_memory=True)

    def __load_test_dataset(self):
        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean = [0.5, 0.5, 0.5], std = [0.5, 0.5,
    0.5]),
            transforms.ConvertImageDtype(torch.float16)
        ])
        image_folder = datasets.ImageFolder('data/cinic/test', transform =
    transform)
        self.test_dataset = DataLoader(image_folder, batch_size = self.
    batch_size, shuffle = True, pin_memory=True)
```

Клас для набору даних CIFAR-100(CIFAR100.py):

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from DatasetUtil.Loader import Loader

class CIFAR100(Loader):
    def __init__(self, batch_size):
        super().__init__("CIFAR-100", 32, 3, batch_size)
        self.__load_train_dataset()
        self.__load_test_dataset()

    def __load_train_dataset(self):
        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
            transforms.ConvertImageDtype(torch.float16)
        ])
        training_data = datasets.CIFAR100(
            root = "data",
            train = True,
            download = True,
            transform = transform,
        )
        self.train_dataset = DataLoader(training_data, batch_size = self.
    batch_size, pin_memory=True)

    def __load_test_dataset(self):
        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
            transforms.ConvertImageDtype(torch.float16)
        ])
        test_data = datasets.CIFAR100(
            root = "data",
            train = False,
            download = True,
            transform = transform,
        )
        self.test_dataset = DataLoader(test_data, batch_size = self.
    batch_size, pin_memory=True)
```

Додаток Б. Код моделей

Реалізація класу повнозв'язних мереж(MLP.py):

```
from torch import nn


class MLP(nn.Module):
    def __init__(self, size):
        super().__init__()
        self.kan_transform_flag = False
        self.flatten = nn.Flatten()
        self.linear_relu_stack = self.__generate_stack(size)
        self.model_type = "MLP"

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

    def __generate_stack(self, size):
        layers = []
        for i in range(len(size) - 1):
            layers.append(nn.Linear(size[i], size[i + 1]))
            if i < len(size) - 2:
                layers.append(nn.ReLU())
        return nn.Sequential(*layers)
```

Реалізація класу згорткових мереж(CNN.py):

```
from torch import nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self, color_channels, image_size, fc_layers):
        super().__init__()
        self.model_type = "CNN"
        self.conv1 = nn.Conv2d(color_channels, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.__calculate_flatten_size(image_size)
        fc_layers.insert(0, self.flatten_size)
        self.size = fc_layers
        self.linear_relu_stack = self.__generate_stack(fc_layers)
        self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
```

```
        x = x.view(-1, self.flatten_size)
        logits = self.linear_relu_stack(x)
        return logits

    def __calculate_flatten_size(self, image_size):
        after_fst_conv = image_size - 5 + 1
        after_fst_pool = after_fst_conv / 2
        after_snd_conv = after_fst_pool - 5 + 1
        after_snd_pool = after_snd_conv / 2
        self.flatten_size = int(after_snd_pool * after_snd_pool * 16)

    def __generate_stack(self, size):
        layers = []
        for i in range(len(size) - 1):
            layers.append(nn.Linear(size[i], size[i + 1]))
            if i < len(size) - 2:
                layers.append(nn.ReLU())
        return nn.Sequential(*layers)
```

Реалізація класу згорткових мереж Колмогорова-Арнольда(CKAN.py):

```
from torch import nn

from model.KANs.EfficientKAN import KANLinear
from model.KANs.KANConv import KAN_Convolutional_Layer


class CKAN(nn.Module):
    def __init__(self, image_size, color_channels, fc_layers, grid_size: int
    = 5):
        super().__init__()
        self.model_type = "CKAN"
        self.kan_transform_flag = False
        self.conv1 = KAN_Convolutional_Layer(in_channels = color_channels,
                                             out_channels = 6,
                                             kernel_size = (5, 5),
                                             grid_size = grid_size,
                                             padding = (0, 0),
                                             device = "cuda"
                                             )
        self.conv2 = KAN_Convolutional_Layer(in_channels = 6,
                                             out_channels = 16,
                                             kernel_size = (5, 5),
                                             grid_size = grid_size,
                                             padding = (0, 0),
                                             device = "cuda"
                                             )
        self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2)
```

```
27        self.flat = nn.Flatten()
28        self.__calculate_flatten_size(image_size)
29        fc_layers.insert(0, self.flatten_size)
30        self.size = fc_layers
31        self.kan_layers = self.__generate_stack(fc_layers)
32
33    def forward(self, x):
34        x = self.pool(self.conv1(x))
35        x = self.pool(self.conv2(x))
36        x = self.flat(x)
37        x = self.kan_layers(x)
38        return x
39
40    def __calculate_flatten_size(self, image_size):
41        after_fst_conv = image_size - 5 + 1
42        after_fst_pool = after_fst_conv / 2
43        after_snd_conv = after_fst_pool - 5 + 1
44        after_snd_pool = after_snd_conv / 2
45        self.flatten_size = int(after_snd_pool * after_snd_pool * 16)
46
47    def __generate_stack(self, size):
48        layers = []
49        for i in range(len(size) - 1):
50            layer = KANLinear(
51                size[i],
52                size[i + 1],
53                grid_size = 5,
54                spline_order = 3,
55                scale_noise = 0.01,
56                scale_base = 1,
57                scale_spline = 1,
58                base_activation = nn.SiLU,
59                grid_eps = 0.02,
60                grid_range = [0, 1],
61            )
62            layers.append(layer)
63        return nn.Sequential(*layers)
```