КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

Кваліфікаційна робота
на здобуття ступеня магістра

за освітньо-науковою програмою «Штучний інтелект»
спеціальності 122 «Комп'ютерні науки» на тему:

АСОЦІАТИВНА МЕТАПАМ'ЯТЬ

Виконав студент 2-го курсу
Винник Дмитро Олександрович _____
(підпис)

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Панченко Тарас Володимирович _____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри математичної інформатики
«_____»_____ 2023 р.,
протокол №_____
Завідувач кафедри
Терещенко В. М. _____
(підпис)

Київ – 2023

ЗМІСТ

РЕФЕРАТ

Магістерська робота складається із вступу, трьох розділів, висновків, списку використаних джерел (45 найменувань) та 1 додатку. Робота містить 15 рисунків та 5 таблиць. Загальний обсяг роботи становить 32 сторінки, основний текст роботи викладено на 19 сторінках.

O(1) KNN, НЕЙРОХЕШУВАННЯ, ІНДЕКСИ, ДЕРЕВА ВАН ЕМБДЕ БОАРСА, ІНФОРМАЦІЙНИЙ ПОШУК, АСОЦІАТИВНА ПАМ'ЯТЬ, МЕТАПАМ'ЯТЬ, ШВИДКІ СТРУКТУРИ ДАНИХ, SOURCE KNOWLEDGE, RETRIEVAL-AUGMENTED, SELF-SUPERVISED.

Об'єктом дослідження є штучна епізодична пам'ять. Предметом дослідження є асоціативна метапам'ять.

Метою роботи є розробка підходу для швидкого пошуку і збереження багатовимірного многовиду спогадів у одновимірну (лінійну) пам'ять, яка притаманна всім комп'ютерам, що реалізують архітектуру фон Неймана.

Методи розроблення: мультимодальні штучні нейронні мережі, методи пошуку подібності щільних векторів (dense vectors), методи квантування кодів (code quantization), методи індексації графами близькості (proximity graphs). Інструменти розроблення: середовища розробки Visual Studio та PyCharm, мови програмування C++ та Python 3, профайлер VS Performance Profiler.

Результати роботи: доведена необхідність використання нейрохешування та неспроможність звичайних алгоритмічних методів індексації до пошуку та додавання у реальному часі великої кількості векторних представлень, коли база досягає великих розмірів; натреновано модель, що повертає двійкове число чим формує лінійний порядок; втілено структуру даних для швидкого пошуку сусідів.

СКОРОЧЕННЯ

k-NN	–	*K*-nearest neighbors algorithm Алгоритм *k* найближчих сусідів
Faiss	–	Facebook AI Similarity Search Пошук подібності Facebook AI
Glass	–	Graph Library for Approximate Nearest Search Бібліотека графів для приблизного пошуку сусідів
GPU	–	Graphics Processing Unit Модуль обробки графіки (графічний процесор)
HNSW	–	Hierarchical Navigable Small World Ієрархічний курсований малий світ
IVF	–	Inverted File Index Інвертований файловий індекс
LLaMA	–	Large Language Model Велика модель мови
LSH	–	Locality-Sensitive Hashing Локальночутливе хешування
mAP	–	Mean Average Precision Середня усереднена точність
MIPS	–	Maximum Inner-Product Search Пошук максимального скалярного добутку
NGT	–	Neighborhood Graph and Tree Граф сусідства та дерево
NMSLIB	–	Non-Metric Space Library Бібліотека неметричних просторів
OPQ	–	Optimized Product Quantization Оптимізоване добуткове квантування
ScaNN	–	Scalable Nearest Neighbors Масштабовані найближчі сусіди

ВСТУП

Оцінка сучасного стану об’єкта розробки

Трансформери [1] є ведучою архітектурою штучних нейронних мереж для моделювання і передбачення послідовностей. В найпростішому випадку генеративні трансформери вчаться передбачувати наступний токен використовуючи інформацію всіх попередніх токенів у межах вікна. Наприклад, токенами можуть слугувати слова вкладені у багатовимірний векторний простір чи невеликі прямокутні фрагменти зображення (n, m). Їх розкладають в одновимірні вектори ($n * m$) і вкладають у векторний простір за принципом схожим зі словами [2].

Одним з підходів для покращення якості виводу генеративних трансформерів є використання зовнішньої бази знань (retrieval-augmented) з якої беруться початкові знання (source knowledge). У випадку текстових трансформерів, із бази беруться релевантні речення. Релевантність визначається за скалярним добутком (MIPS), косинусом подібності або Евклідовою відстанню між вкладеними векторами цих речень. Яскравими представниками такого підходу є REALM [3], DPR [4], RAG [5], FiD [6], RETRO [7], R2-D2 [8] та Atlas [9]. Ці архітектури переважно розв’язують задачу відповіді на відкриті запитання.

В загальному випадку для пошуку релевантних елементів шукають k найближчих сусідів (k-NN). Когнітивним аналогом бази знань виступає епізодична пам’ять, аналогом k-NN – метапам’ять. Метапам’ять відіграє одну з ключових ролей у мисленні: людина може одразу, без тривалих роздумів сказати, що не знає п’ятого за розміром динозавра. Це дозволяє суттєво економити ресурси мозку, бо інакше потрібно було б згадувати усе, що знає людина, щоб зробити висновок, що вона чогось не знає.

Окрім знань у базі можуть зберігатися вкладені вектори з попередніх контекстів. Так роблять в архітектурах Memorizing Transformers [10] та Unlimiformer [11]. Це неявно розширює поточний контекст до розміру обмеженого лише якістю пошуку.

Актуальність роботи та підстави для її виконання

Кількість векторів у базі знань може сягати мільярдів. Для швидкого пошуку сучасні архітектури використовують багаторівневі індекси подібності, що базуються на квантуванні та пошуку у графах. Їхній час запиту зростає лінійно при збільшенні k. Емпірично, це дозволяє віднайти порядку тисячі речень по 64 токени без значного впливу на час тренування. Такого обсягу тексту недостатньо для деяких задач, наприклад, сумаризації (реферування). В області комп'ютерного зору це ще помітніше, бо тисяча зображень це лише ~17 секунд відео при частоті 60 кадрів в секунду.

Вочевидь людський мозок не витрачає багато часу на засвоєння сенсорної інформації. Іншими словами, індексація відбувається миттєво під час інтеграції нової інформації. Блукання розуму (mind-wandering) теж не потребують довгого пошуку. Натомість загальновживані алгоритми k-NN є сублінійними у кращому випадку, наприклад LSH [12] – $O(n^{p<1})$.

Мета та завдання роботи

Метою роботи є розробка підходу для додавання та пошуку по базі знань за константний час, використовуючи нейрохешування та швидкі структури даних. Для досягнення цієї мети поставлено такі завдання:

- заміряти час роботи найбільш вживаних індексів подібності на синтетичних даних
- проаналізувати доступні методи нейрохешування
- адаптувати один з них під мультимодальний режим
- порівняти популярні набори даних
- індексувати один з наборів даних
- підібрати швидку структуру даних для цього індексу

РОЗДІЛ 1. ІНДЕКСИ ПОДІБНОСТІ

1.1. Порівняння методів пошуку подібних векторів

З бібліотек пошуку, що ґрунтуються на рецензованих роботах, найвідомішими є Meta Faiss [13] та Google ScaNN [14], з неакадемічних – Spotify Annoy, NMSLIB, Kakao N2, Yahoo NGT. Окрім відкритих бібліотек існує багато напівкомерційних векторних баз даних. Серед них: Milvus > Weaviate > Qdrant (ранжування за швидкістю) та Pinecone (не тестувалася).

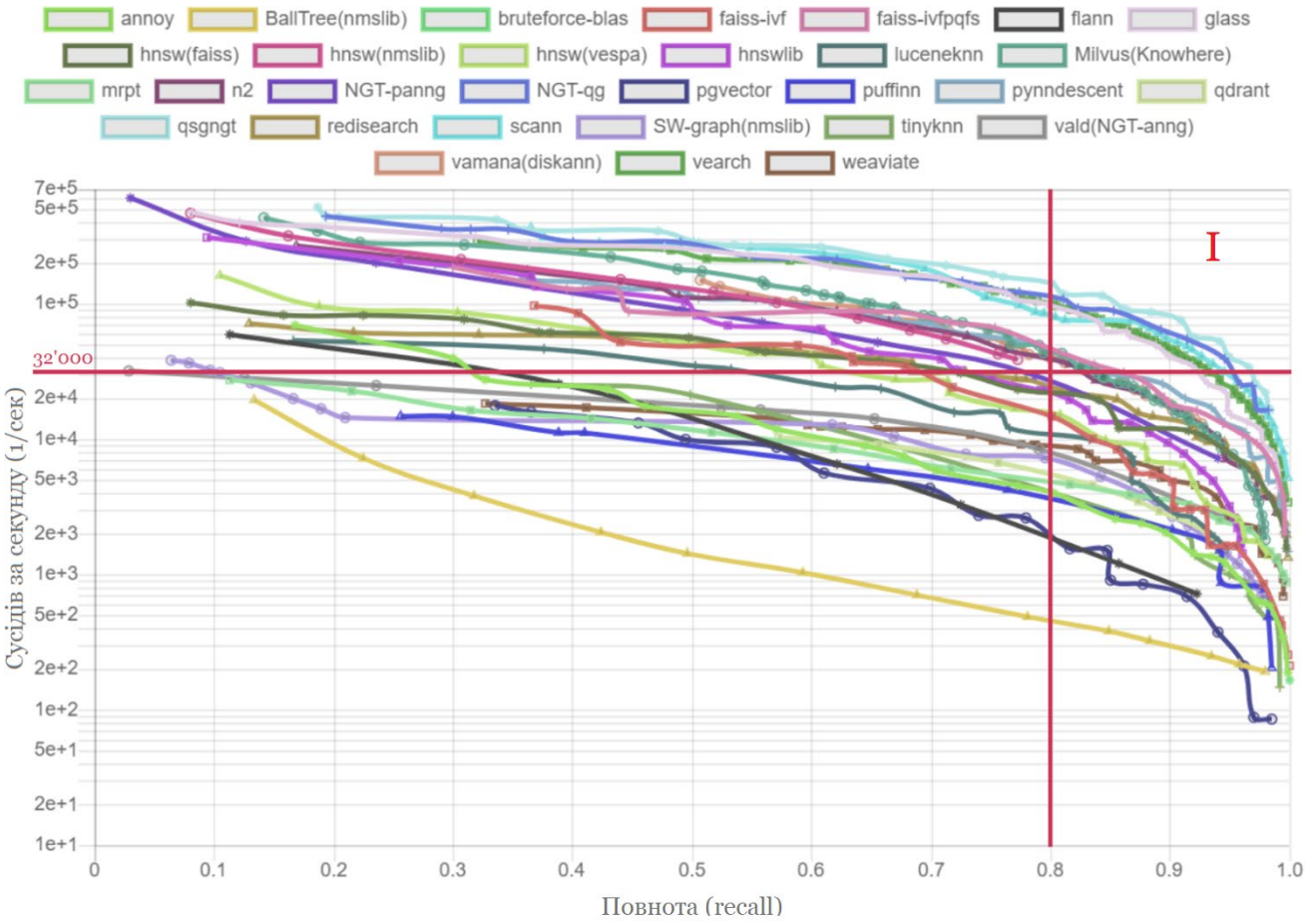

Рисунок 1 – Заміри продуктивності великої кількості методів з ann-benchmarks[1]. Червоними лініями відмежований I-й квадрант: кільк. зап. > 32′000, повн. > 0.8

Апріорі задамо нижню межу продуктивності у 32 тисячі запитів за секунду з повнотою щонайменше у 0.8. За невеликих k, такої кількості достатньо для виконання у реальному часі (real-time). Тоді після фільтрації залишиться 11 методів:

[1] https://github.com/erikbern/ann-benchmarks/
https://ann-benchmarks.com/

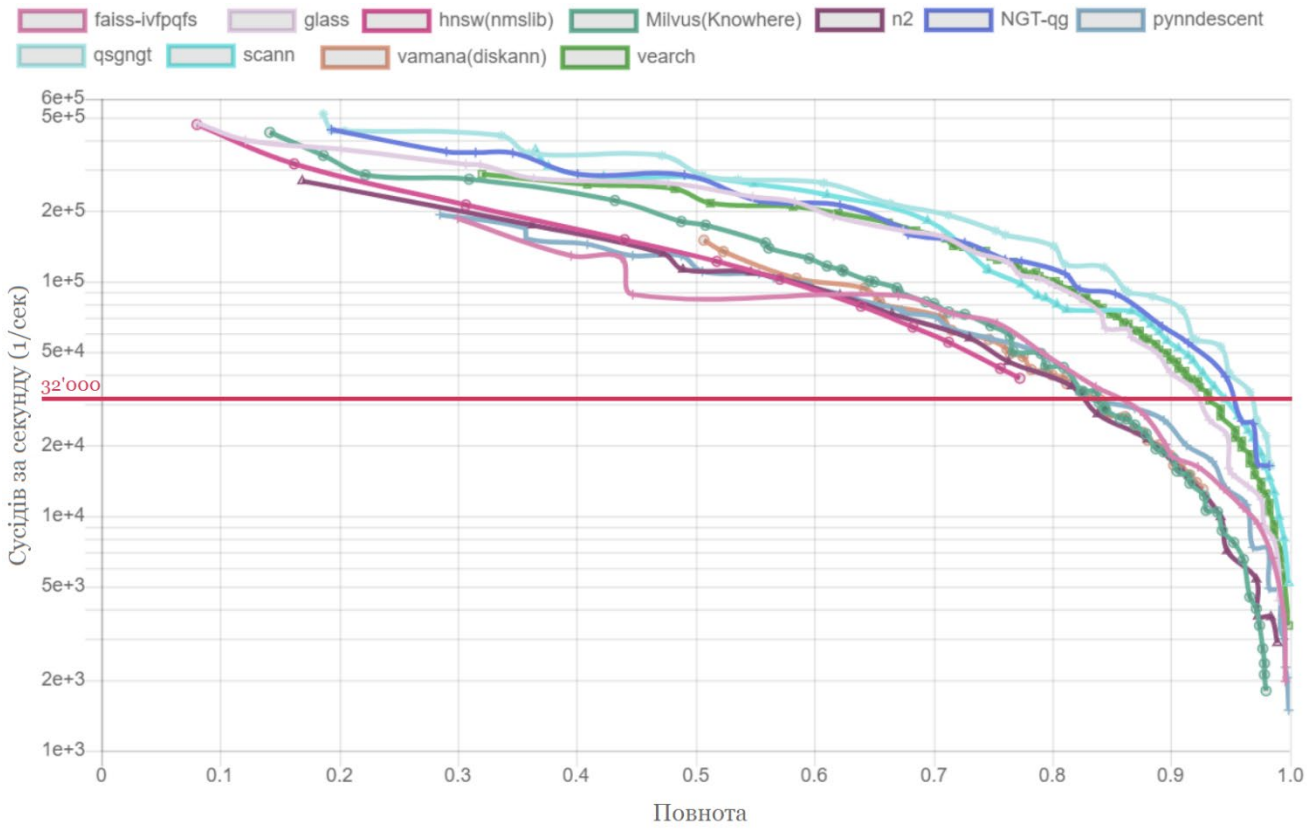

Рисунок 2 – Заміри продуктивності real-time методів

До переліку real-time методів входять зазначені раніше рецензовані Faiss та ScaNN. На графіку помітно виділяється група найшвидших методів: qsgngt > NGT > Glass > ScaNN. З них широковживаними є NGT та ScaNN.

1.2. Опис бібліотеки Faiss

Індекси Faiss є менш затратними по пам’яті ніж ScaNN, бо ScaNN окрім зберігання квантованого індексу потребує зберігання оригінальних векторів. Також Faiss має кращу документацію. Тому доцільно почати дослідження з Faiss.

Опис Faiss з репозиторію: “Faiss – це бібліотека для ефективного пошуку подібності та кластеризації щільних векторів (з високою щільністю інформації). Вона містить алгоритми, які здійснюють пошук у наборах векторів будь-якого розміру, аж до тих, які не вміщуються в оперативній пам’яті. Вона також містить допоміжний код для оцінки та налаштування параметрів. Faiss написано мовою C++ із повними обгортками для Python/numpy. Деякі з найбільш корисних алгоритмів реалізовані на GPU. Переважно її розроблює дослідницька група Meta Fundamental AI Research”.

Одним з основних методів для формування індексу є квантування. Воно використовується для ефективного пошуку подібності у даних великої розмірності. Мета квантування – зменшити розмірність набору даних, зберігаючи важливу інформацію про дані. Це забезпечує швидші й ефективніші алгоритми пошуку, які можна використовувати для прискорення таких завдань, як пошук зображень і тексту.

У Faiss квантування виконується шляхом поділу набору даних на кілька менших підпросторів [15], які називаються комірками Вороного. Кожна комірка Вороного відповідає кластеру подібних точок даних. Далі центроїди цих кластерів обчислюються та зберігаються як таблиця пошуку. Коли запит виконується до набору даних, він спочатку квантується в одну з комірок Вороного, а потім знаходить найближчий центроїд.

Основна перевага квантування полягає в тому, що воно зменшує обсяг даних, які потрібно зберігати та шукати, завдяки чому воно набагато швидше, ніж інші методи. Крім того, квантування можна легко паралелізувати, що робить його придатним для широкомасштабного аналізу даних.

Faiss підтримує порівняння Евклідових відстаней, скалярних добутків та косинусів подібності. Всі три метрики є пропорційними одна одній:

$$\cos(\theta_{xy}) = \frac{\langle x, y \rangle}{\|x\|\|y\|}$$

$$\|x - y\|_2^2 = \sum_{i=1}^{n} (x_i - y_i)^2 = \|x\|_2^2 + \|y\|_2^2 - 2x^T y = 2\left(1 - \cos(\theta_{xy})\right)$$

$$\langle x, y \rangle \propto \cos(\theta_{xy}) \propto \|x - y\|_2^2$$

Евклідову відстань слід використовувати коли норми векторів мають сенс.

1.3. Фабрика індексів Faiss

Фабрика індексів – це допоміжний модуль у Faiss, який забезпечує зручний спосіб створення та налаштування об’єктів індексу. Фабрика піклується про ініціалізацію, навчання та збереження індексу, а також надає простий інтерфейс для налаштування різних параметрів індексу, таких як кількість кластерів, кількість байтів на код у квантуванні та кількість рівнів.

У цій роботі для тестування використовувалася наступна конфігурація: *OPQ16_64,IVF1000_HNSW32,PQ16x4fs*

OPQ [16] має на меті зменшити спотворення внесені векторним квантуванням, шляхом застосування ортогонального перетворення до вхідних векторів перед квантуванням. Трансформація обертає та масштабує вектори таким чином, щоб мінімізувати відстань між трансформованими векторами та центроїдами кластерів, що призводить до більш точного квантування.

Алгоритм OPQ працює, спочатку розбиваючи вхідні вектори на кілька блоків, а потім ітеративно оптимізуючи обертання та масштабування кожного блоку, щоб мінімізувати помилку квантування. Оптимізація виконується за допомогою градієнтного спуску, при цьому градієнти обчислюються з аналітичного виразу.

OPQ16 на початку конфігурації вказує лише на матрицю повороту, тому в кінці обов'язково маємо *PQ16x4fs*, що відповідає саме за процес квантування [15] і означає 16 кодів по 4 біти у режимі *fast scan*. Таким чином, розмір коду 16 * 4 / 8 = 8 байтів. *_64* в *OPQ16_64* – це додаткова операція зменшення розмірності до 64. *IVF1000_HNSW32* вказує на комбінацію IVF з 1000 комірок та HNSW [17], де кожен центроїд пов'язаний з 32 іншими.

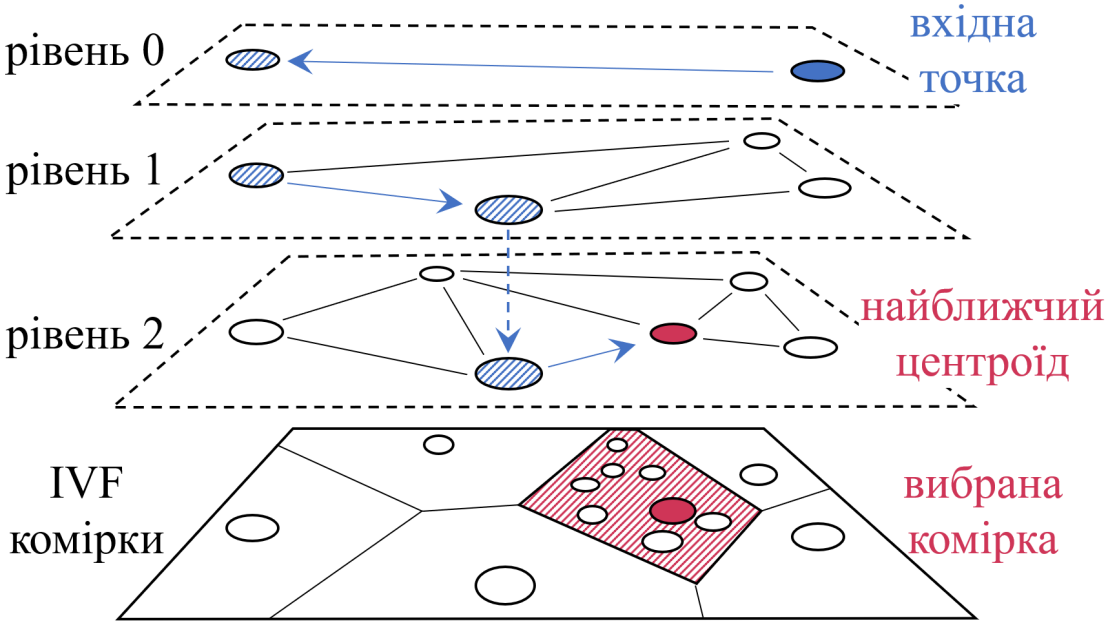

Рисунок 3 – HNSW та IVF. Репліка з блогу Pinecone

1.4. Тестування індексів

Для гарантованого отримання k сусідів у IVF треба робити проби по сусідніх комірках, бо метод є приблизним. Причому зі збільшенням k кількість необхідних проб теж зростає. У Faiss за це відповідає параметр nprobe.

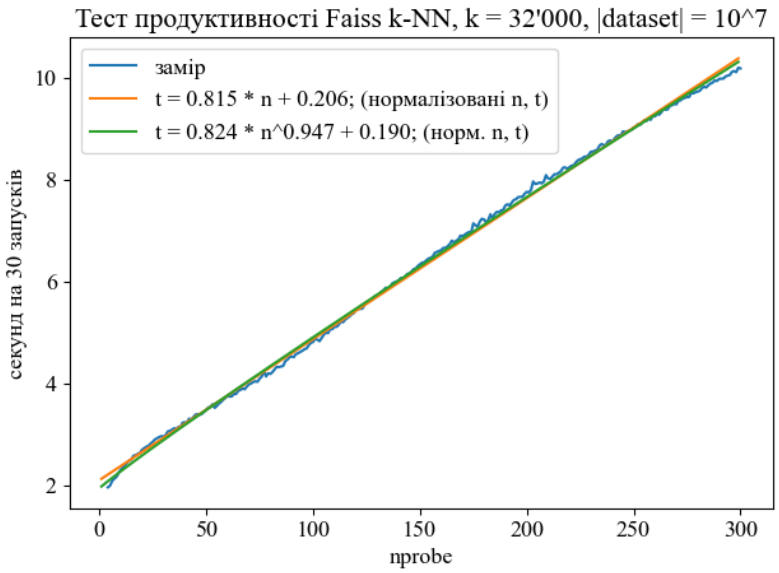

Рисунок 4 – Залежність часу запиту від nprobe

– У легенді графіку наведені апроксимації заміру многочленами першого порядку та степеневими функціями зі зсувом. Як можна побачити час запиту зростає лінійно (вважатимемо природу степеня 0.947 помилковою).

Наступним кроком перевіримо залежність часу запиту від k:

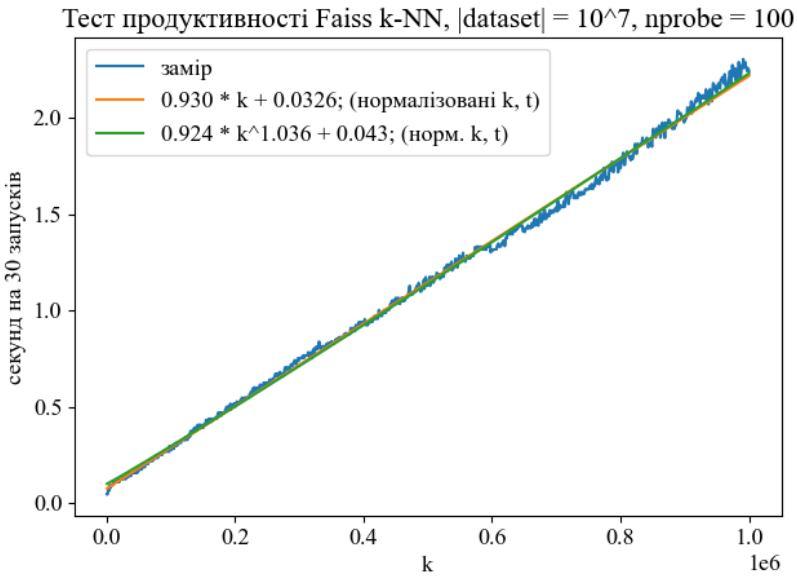

Рисунок 5 – Залежність часу запиту від кількості сусідів у k-NN

Ця залежність теж виявилася близькою до лінійної (вважатимемо природу степеня 1.036 помилковою). Натомість залежність від розміру бази даних переходить із сублінійної в супер лінійну при збільшенні k (0.639 → 0.755 → 0.911 → 1.01 → 1.14):

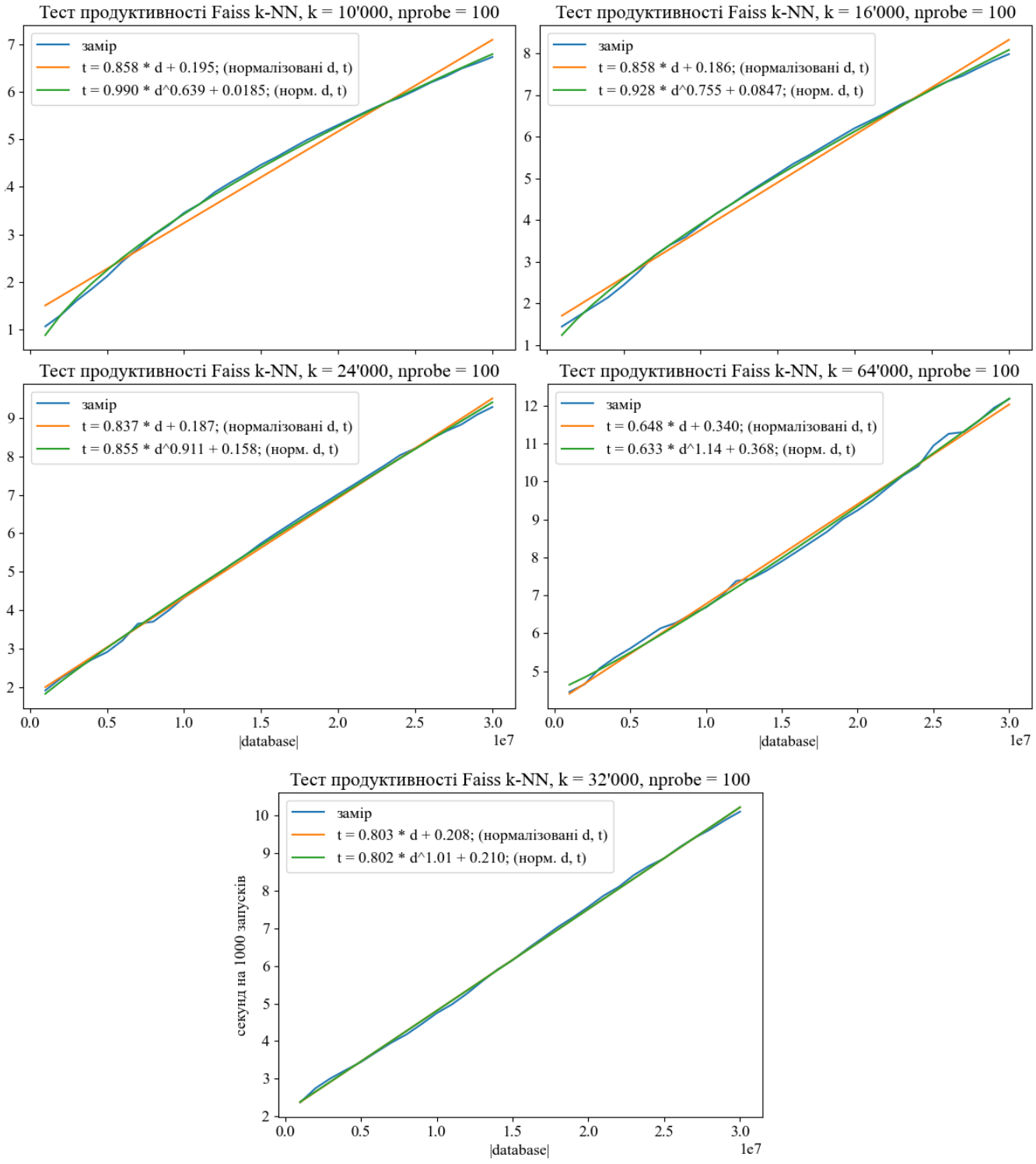

Рисунок 6 – Залежність часу запиту від розміру бази даних

Враховуючи останні досягнення у збільшенні розміру контексту до ~2 мільйонів токенів [18], обчислимо скільки часу буде займати запит у випадку набору текстових даних The Pile [19], що після обробки містить 5.8 мільярдів фрагментів по 64 токени. Це еквівалентно k-NN з $k = 2'048'000 / 64 = 32'000$.
Тобто можемо використати не нормалізований аналог t = 0.803 * d + 0.208:

t = 2.7e-07 * 5.8e9 + 2.1 ≈ 1568 секунд/1000 запусків = 1.568 секунди/запит
Це в точності відповідає графіку faiss-ivfpqfs на Рис. 2 та заміряній повноті 0.89. З цього ж графіку можна побачити, що найшвидші методи ScaNN та NGT-qg випереджують Faiss до трьох разів.

Така тривалість суттєво уповільнює тренування. У статті яка вводить генеративний трансформер LLaMA [20] автори зазначили, що пропускна здатність моделі на 65 мільярдів параметрів – 380 токенів/секунду/GPU на 2048 A100 GPU з 80 ГБ відеопам’яті. Тобто пропускна здатність кластера 778′240 токенів/секунду. Якщо повернутися до контексту довжиною ~2M, то це 2.634 секунд/запит.

Окрім уповільнення тренування на 1.568 / (1.568 + 2.634) = 37% у випадку Faiss і 17% у швидших методах, індекси подібності мають чимало неінтуїтивних гіперпараметрів. Водночас штучні нейромережі доволі часто випереджують класичні алгоритми. Їхнє пряме розповсюдження має константний час у межах кількох мілісекунд. Це наштовхує на думку, що має бути нейромережевий підхід, який працює за k * O(const) = O(k).

РОЗДІЛ 2. НЕЙРОХЕШУВАННЯ

2.1. Інтуїція

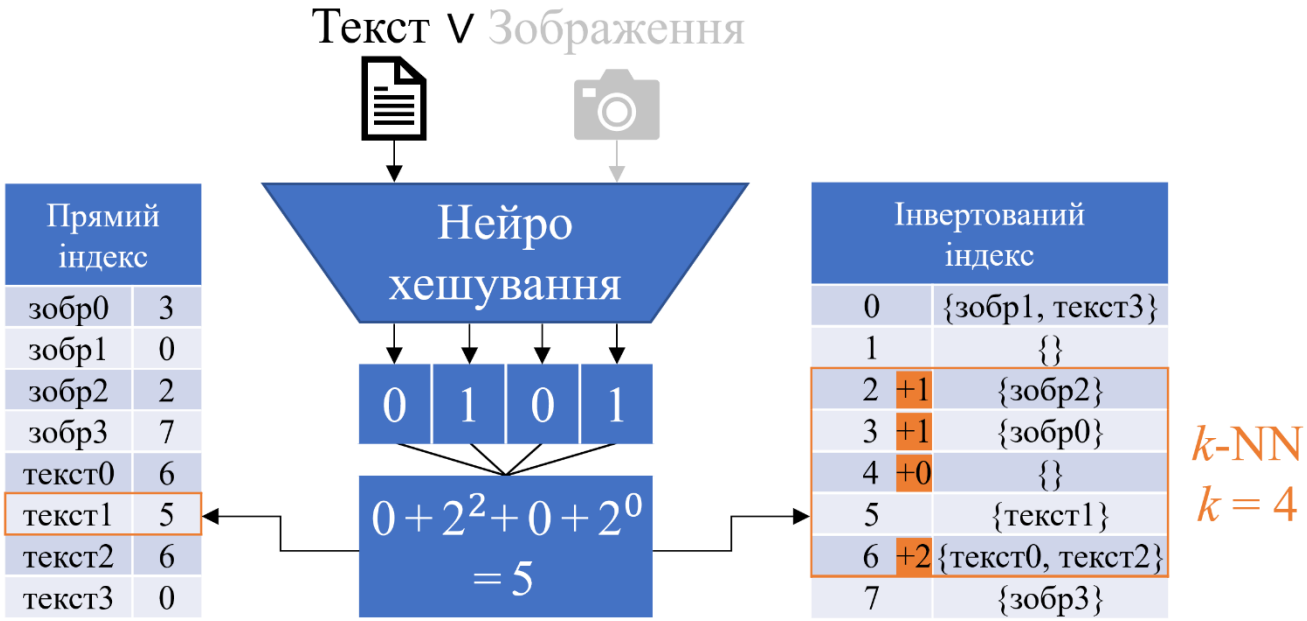

Рисунок 7 – Концепт архітектури. «текст1» додано в прямий та інвертований індекс під номером 5. Для *k*-NN залишається жадібно пройти по інв. індексу

Ідея полягає у тому, щоб моделювати ієрархію універсалій (ознак) за допомогою ієрархії чисел. Тобто, щоб бінарне число 0100 (4) було семантично ближче до 0110 (5) ніж до 1100 (12). Щоб покази, що це можливо, розглянемо випадок одновимірного *k*-NN:

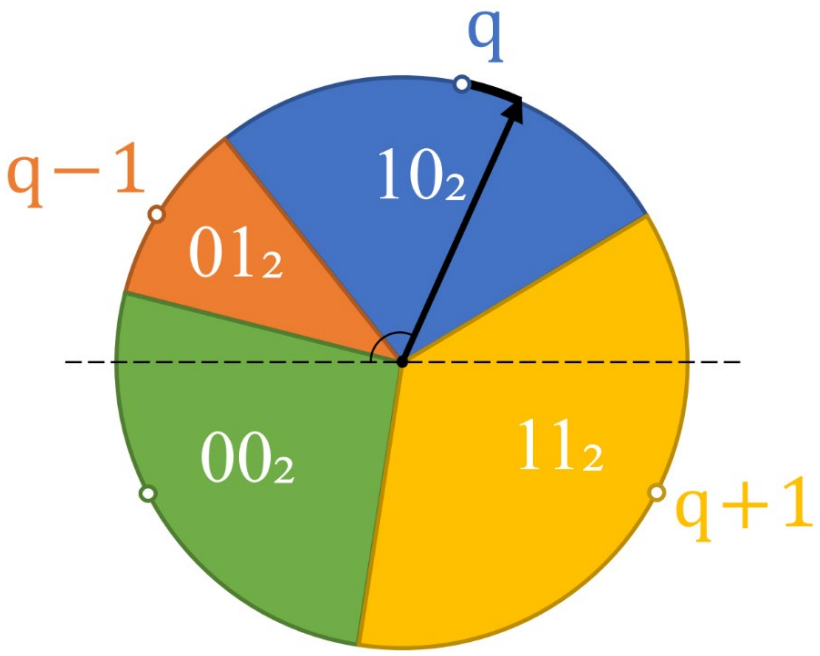

Рисунок 8 – Двовимірна проєкція одновимірної діаграми Вороного

Кольорові зони відповідають за приналежність до певної вершини. Наприклад, чорний вектор запиту попав у комірку синьої вершини. Нейронна мережа може легко вивчити залежність індексу (кольору) від кута вектору. Бінарні виходи мережі є натуральним способом вивчити бінарний числовий індекс. Якщо точок багато, а номери комірок впорядковані за часовою стрілкою, то шукати *k* найближчих сусідів за індексом не складно – треба взяти вершини з індексами від q – k / 2 до q + k / 2, де q це індекс вектора запиту.

У двовимірному випадку все значно складніше, бо істинний (ground truth) порядок вершин за відстанню є розривним, а отже маємо межу рішень (decision boundary) з самоперетинами. Це ускладнює навчання та погіршує узагальнювальні здібності нейронної мережі. Можна якоюсь мірою компенсувати це обширним збалансованим набором даних під час претренування, бо такий набір буде змушувати нейронну мережу ефективно стискати представлення.

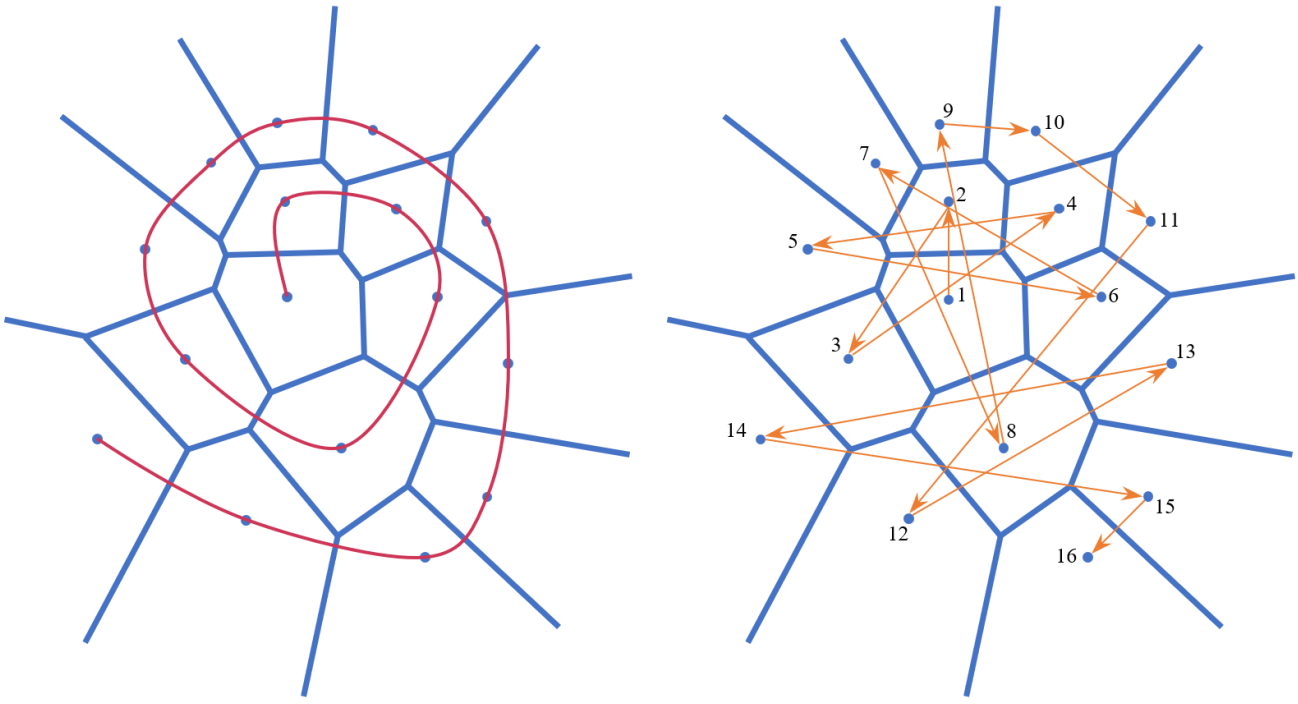

Рисунок 9 – Двовимірна діаграма Вороного.
Зліва бажаний плавний порядок вершин за відстанню від першої,
справа розривний істинний порядок. Нейронна мережа вчить істинний порядок

2.2. Опис даних

Більшість робіт використовують три набори зображень: CIFAR-10 [21], NUS-WIDE [22], MS COCO [23]. Існують різні відфільтровані підмножини та розбиття цих наборів. У CIFAR-10 кожне зображення має один з 10 класів, у NUS-WIDE-21 у кожного зображення до 21 класу, у MS-COCO-2014 до 80 класів. Набір заведено розбивати на три частини: тренувальну частину (train), тестову частину (query) та базу даних (database). Під час тестування для зображення запиту генерується бінарний код (хеш) який шукається у базі даних. Таким чином база даних може містити тренувальні зображення, бо вони не перетинаються з тестовими.

Балансування вибірок зазнавало змін за час розвитку напрямку. Останні роботи TBH [24], NSH [25], WCH [26] зійшлися до одного принципу: CIFAR-10 – вибірка із рівномірним розподілом, NUS-WIDE-10 – у кожного класу щонайменше 100 зображень, MS-COCO-2014 – незбалансована вибірка. Деякі минулі роботи не звертали уваги на балансування набору даних, або містили помилки. Наприклад, у HashNet [27] вибірка database містить query, що може призвести до ідеальних результатів на певних елементах. Тому код для розбиття був взятий з публічного репозиторію DeepHash[2] після перевірки на коректність.

Архітектури; **Набір даних**	**Query (Test)**	**Train**	**Database**	**Усього**
HashNet; NUS-WIDE-81-m	5′000 (незбаланс.)	10′000 (незбаланс.)	Train + Database0 = 218′491	223′496
HashNet; MS-COCO-2014	5′000 (незбаланс.)	10′000 (незбаланс.)	Test! + Database0 = 112′218	122′218
CIBHash; CIFAR-10	5′000 (незбал.) (+5′000 валід.)	500 * 10	Train + Database0 = 50′000	60′000
TBH, NSH, WCH; CIFAR-10	1′000 * 10	5′000 *10	Train = Database = 50′000	60′000
TBH, CIBHash, NSH, WCH; NUS-WIDE-21	2100 (мін. 100/клас)	10′500 (мін. 100/клас)	Train + Database0 = 193′734	195′834
TBH, CIBHash, NSH, WCH; MS-COCO-2014	5′000 (незбаланс.)	10′000 (незбаланс.)	Train + Database0 = 117′218	122′218

Таблиця 1 – Порівняння вибірок

[2] https://github.com/swuxyj/DeepHash-pytorch

2.3. Бінарне Сіамське нейрохешування CIBHash

У зазначених наборах даних ми маємо істинні мітки (класи зображень), але використовувати їх будемо лише для тестування. Бо ми ставимо на меті натренувати модель без нагляду (контролю). Такі методи машинного навчання називаються самоконтрольованими (self-supervised). Одним з популярних self-supervised підходів для вивчення векторних представлень є використання дуальних архітектур. Їх також називають Сіамськими архітектурами, дистиляційними методами, архітектурами суміщення вкладних векторів (joint embeddings architectures) та методами максимізації взаємної інформації (methods of mutual information maximization).

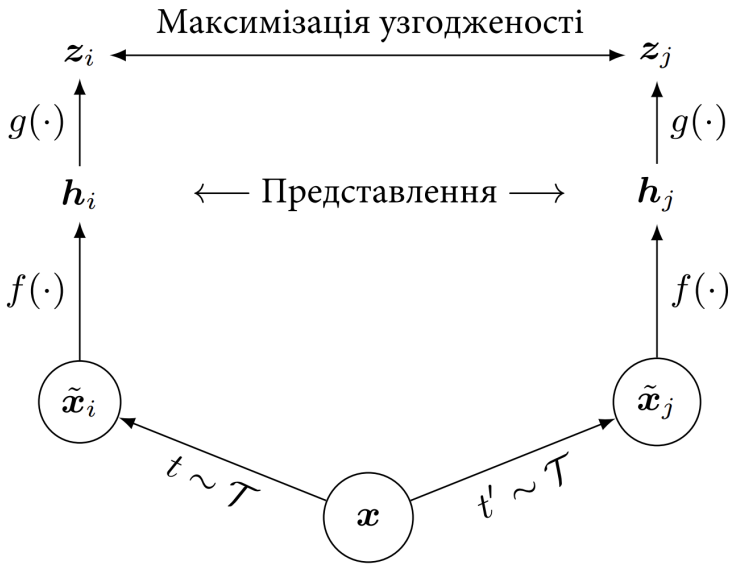

Рисунок 10 [28] – Архітектура контрастного навчання SimCLR.
t та *t'* це аугментації отримані з одного сімейства *T*; *f(·)* – це кодувальна мережа, *g(·)* – проєкційна голова, яка використовується лише під час навчання

Більшість з них використовує контрастне навчання для уникнення колапсу до тривіального рішення. Однак для цього потрібна велика вибірка негативних елементів [29]. Натомість існують методи яким не потрібна негативна вибірка. Замість цього вони використовують різні регуляризації, щоб запобігти колапсу. Наведемо ранжування цих методів за точністю класифікації на ImageNet-1k [30]:

DINOv2 [31] > EsViT [32] = I-JEPA [33] > DINO [34] >= VICReg [35] > Whitening-based methods [36] = Barlow Twins [37] > BYOL [38]

Причому більш точними себе показують методи що орієнтовані на модальність набору даних (у випадку ImageNet це зображення): DINOv2..DINO. Їх немає сенсу адаптувати під текст через те, що їх архітектура ґрунтується на специфічних аугментаціях зображень та інших аспектах не притаманних текстам. Мультимодальні методи VICReg..BYOL є менш точними, але більш універсальними. Практика показує, що з часом надто специфічні методи уступають універсальним[3], але таких методів по всій видимості поки не існує.

Метод	Архітек.	Епохи	Точність
Методи без аугментації даних			
data2vec	ViT-L/16	1600	77.3
MAE	ViT-B/16	1600	68.0
	ViT-L/16	1600	76.0
	ViT-H/14	1600	77.2
CAE	ViT-B/16	1600	70.4
	ViT-L/16	1600	78.1
I-JEPA	ViT-B/16	600	72.9
	ViT-L/16	600	77.5
	ViT-H/14	300	79.3
	ViT-H/16_{448}	300	**81.1**
Методи з аугментацією даних			
SimCLR v2	RN152 (2×)	800	79.1
DINO	ViT-B/8	300	80.1
iBOT	ViT-L/16	250	**81.0**

Метод	Архітек.	Дані	Точність
MAE	ViT-H/14	INet-1k	76.6
DINO	ViT-S/8	INet-1k	79.2
SEERv2	RG10B	IG2B	79.8
MSN	ViT-L/7	INet-1k	80.7
EsViT	Swin-B/W=14	INet-1k	81.3
Mugs	ViT-L/16	INet-1k	82.1
iBOT	ViT-L/16	INet-22k	82.3
DINOv2	ViT-S/14	LVD-142M	81.1
	ViT-B/14	LVD-142M	84.5
	ViT-L/14	LVD-142M	86.3
	ViT-g/14	LVD-142M	**86.5**

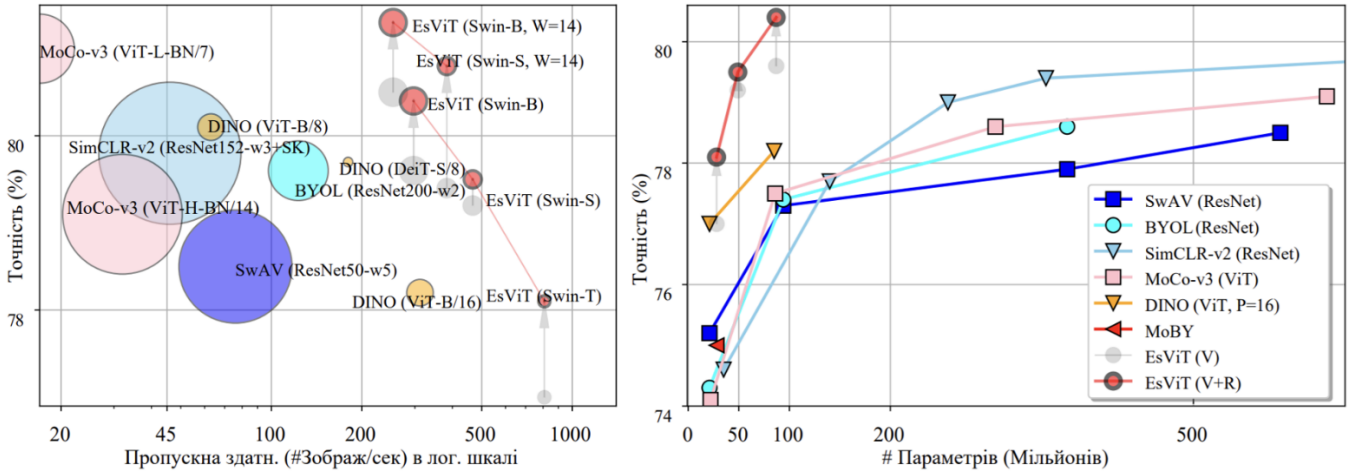

Рисунок 11 – Підтвердження ранжування. Витяг зі статей [33], [31], [32]

На жаль ми не можемо використовувати перелічені дуальні self-supervised методи, бо їхні вкладені вектори складаються з чисел з рухомою точкою. Порівнювати такі вектори це надто повільно. Тому методи нейрохешування

[3] http://www.incompleteideas.net/IncIdeas/BitterLesson.html

використовують бінарні коди. Звичайні штучні нейронні мережі по своїй природі не здатні повертати дискретні значення. Однак існує багато математичних підходів, які дозволяють оминути це обмеження. Порівняння методів бінарного нейрохешування наведено в огляді [39]. Серед останніх варто зазначити CIBHash [40], NSH [25], WCH [26] та SDC [41]. Чим далі ці методи відходять від оригінальної архітектури SimCLR [28], тим більше з'являється невиправданої схильності до ненатуральних статистичних підходів і різноманітних математичних трюків.

У цій роботі ми використаємо CIBHash, бо NSH не має відкритого коду, у WCH дотреновують базову модель виділення ознак, що є нечесним, а SDC має занадто складні конфігурації. CIBHash це доволі проста архітектура: базова претренована модель виділення ознак VGG [42] яку заморожують під час тренування та два лінійних шари. Бінаризація досягається використанням наскрізного оцінювача градієнта (straight-through gradient estimator): при зворотному поширенні помилки *sign* функція просто пропускається.

Щоб досягти лінійного порядку, у функцію помилки було додано переведення із двійкового коду в десяткове число. Тобто обидві частини дуальної моделі повертають по одному числу, далі ці числа віднімаються для позитивних і негативних елементів вибірки. Позитивні елементи є фрагментами одного зображення і мусять мати меншу різницю. Попри простоту підходу він має добре працювати, бо така операція добре пропускає градієнт.

Оригінальну модель виділення ознак або радше вкладених векторів було замінено на мультимодальну модель BEiT-3 [43]. Потенційно, якщо додати текстові аугментації у CIBHash це дозволить індексувати текст окрім зображень. Також BEiT має кращі результати на класифікації зображень ніж VGG, а отже його вкладені вектори містять більше інформації про вхідні дані. Це має підвищити показники CIBHash.

✓	✗	✗	✓	✓	mAP@5 = 1/2 * (AP_1 + AP_2) ≈ 1/2 * 1.62 = 0.81
P@1 = 1/1	P@2 = 1/2	P@3 = 1/3	P@4 = 2/4	P@5 = 3/5	
rel@1 = 1	rel@2 = 0	rel@3 = 0	rel@4 = 1	rel@5 = 1	
P@1 * rel@1 = 1/1	P@2 * rel@2 = 0/2	P@3 * rel@3 = 0/3	P@4 * rel@4 = 2/4	P@5 * rel@5 = 3/5	
AP_1@5 = 1/3 * (1/1 + 0/2 + 0/3 + 2/4 + 3/5) = 0.7					
✓	✓	✗	✓	✗	
P@1 = 1/1	P@2 = 2/2	P@3 = 2/3	P@4 = 3/4	P@5 = 3/5	
rel@1 = 1	rel@2 = 1	rel@3 = 0	rel@4 = 1	rel@5 = 0	
P@1 * rel@1 = 1/1	P@2 * rel@2 = 2/2	P@3 * rel@3 = 0/3	P@4 * rel@4 = 3/4	P@5 * rel@5 = 0/5	
AP_2@5 = 1/3 * (1/1 + 2/2 + 0/3 + 3/4 + 0/5) ≈ 0.92					

Таблиця 2 – Приклад обрахунку mAP@5

Метод	**CIFAR-10**			**NUS-WIDE**			**MS COCO**		
	16 біт	32 біти	64 біти	16 біт	32 біти	64 біти	16 біт	32 біти	64 біти
AGH	0.333	0.357	0.358	0.592	0.615	0.616	0.596	0.625	0.631
ITQ	0.305	0.325	0.349	0.627	0.645	0.664	0.598	0.624	0.648
DGH	0.335	0.353	0.361	0.572	0.607	0.627	0.613	0.631	0.638
SGH	0.435	0.437	0.433	0.593	0.590	0.607	0.594	0.610	0.618
BGAN	0.525	0.531	0.562	0.684	0.714	0.730	0.645	0.682	0.707
BinGAN	0.476	0.512	0.520	0.654	0.709	0.713	0.651	0.673	0.696
GreedyHash	0.448	0.473	0.501	0.633	0.691	0.731	0.582	0.668	0.710
HashGAN	0.447	0.463	0.481	-	-	-	-	-	-
DVB	0.403	0.422	0.446	0.604	0.632	0.665	0.570	0.629	0.623
DistillHash	0.284	0.285	0.288	0.667	0.675	0.677	-	-	-
TBH	0.532	0.573	0.578	0.717	0.725	0.735	0.706	0.735	0.722
MLS^3RDUH	0.369	0.394	0.412	0.713	0.727	0.750	0.607	0.622	0.641
DATE	0.577	0.629	0.647	0.793	0.809	0.815	-	-	-
MBE	0.561	0.576	0.595	0.651	0.663	0.673	-	-	-
CIMON	0.451	0.472	0.494	-	-	-	-	-	-
CIBHash	0.590	0.622	0.641	0.790	0.807	0.815	0.737	0.760	0.775
SPQ	0.768	0.793	0.812	0.766	0.774	0.785	-	-	-
NSH	0.706	0.733	0.756	0.758	0.811	0.824	0.746	0.774	0.783

Таблиця 3 [26] – CIFAR-10: mAP@1000; NUS-WIDE, MS COCO: mAP@5000

2.4. Мультимодальний трансформер BEiT

Дослідники машинного навчання нещодавно зосередилися на розробці базових моделей загального призначення, які можуть працювати з кількома модальностями та адаптуватися до різноманітних подальших завдань. Дослідницька група Microsoft представила BEiT-3 – мультимодальну базову модель для зорових і мовно-зорових задач. BEiT-3 інтерпретує візуальні токени як текст іноземною мовою яку автори називають «Imglish» та виконує уніфіковане масковане моделювання «мови» на зображеннях, текстах і парах зображення-текст. Модель вчиться на мономодальних і мультимодальних даних через спільну мережу Multiway Transformer, яка використовує спільний модуль самоуваги (self-attention) і різні мережі прямого розповсюдження для кожної модальності. BEiT-3 досяг передової продуктивності у різних тестах:

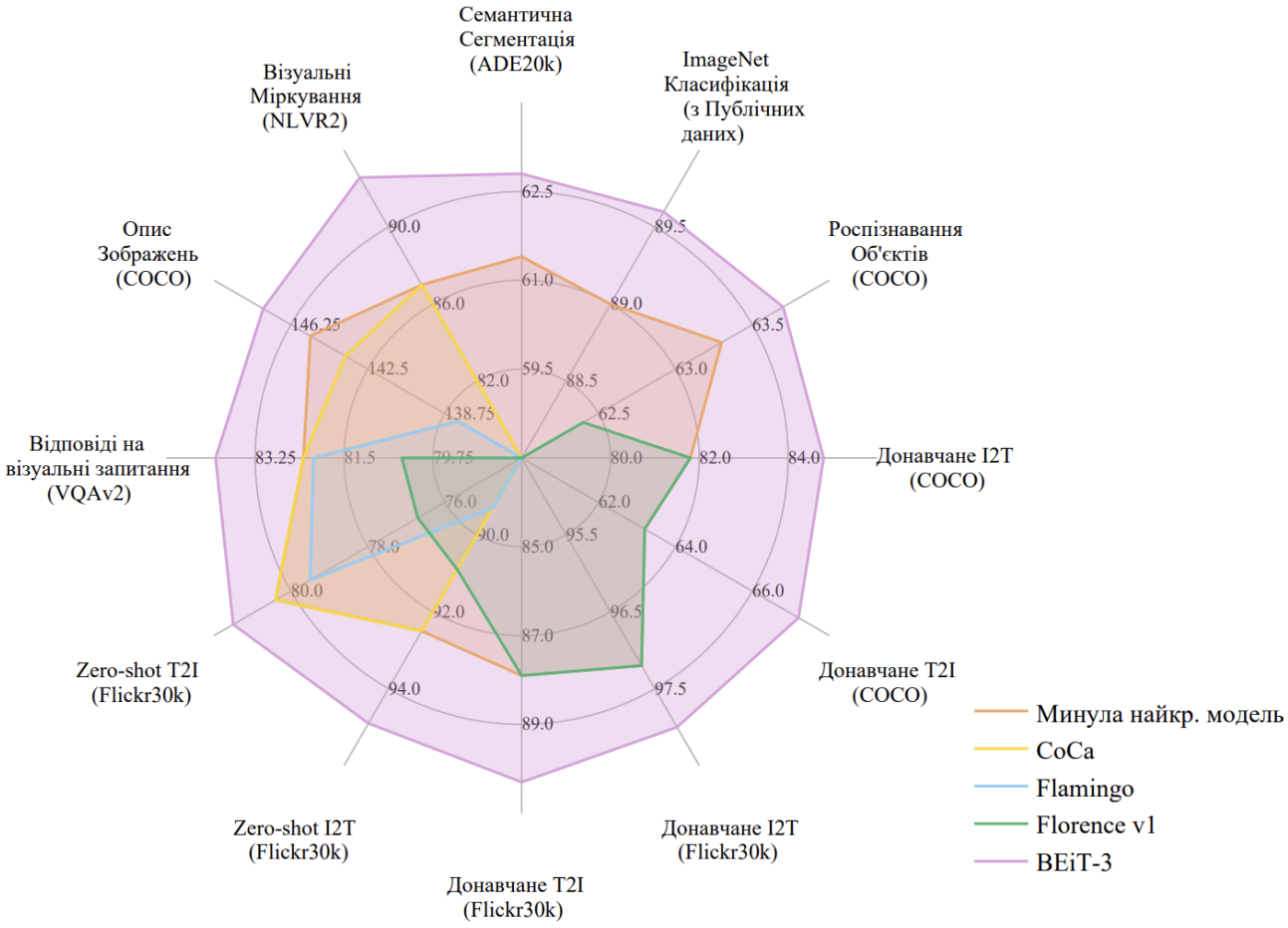

Рисунок 12 [43] – BEiT-3 випереджує інші базові моделі.

I2T/T2I = пошук зображення-по-тексту/тексту-по-зображенням

2.5. Результати тренування

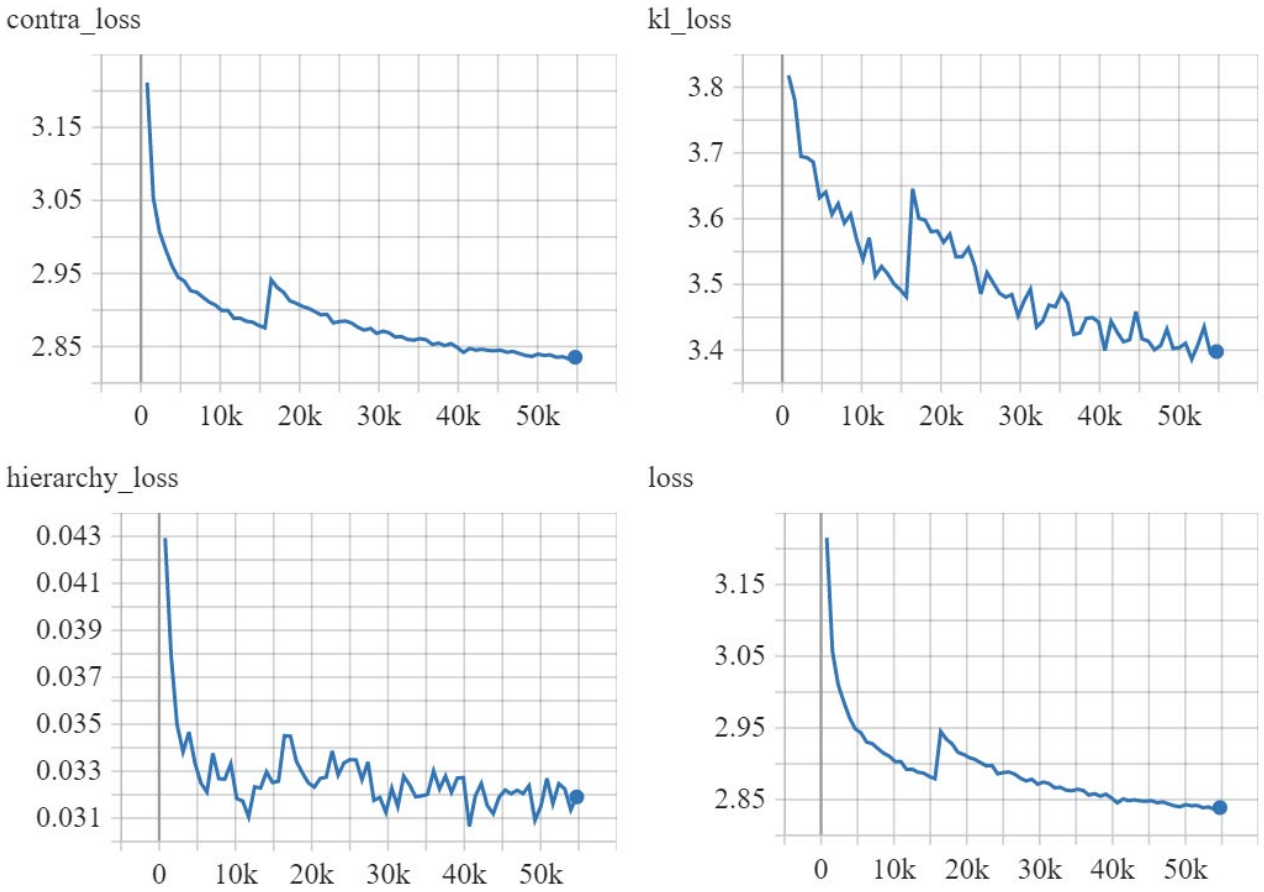

Рисунок 13 – Криві навчання. Функції втрат Contra та KL були введені у CIBHash, Hierarchy – це різниця десяткових чисел введена у цій роботі. Зсув біля 15k викликаний перезапуском

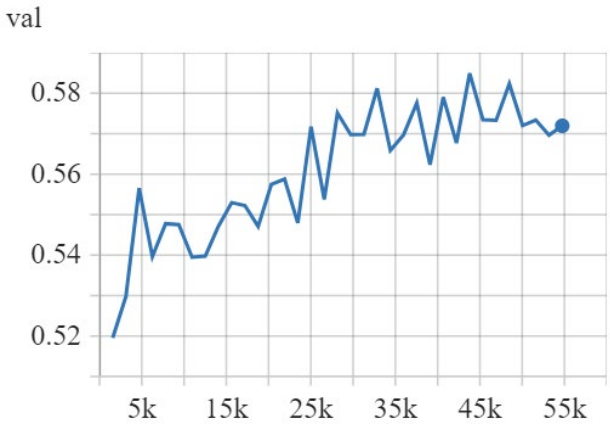

Рисунок 14 – Крива валідації: mAP@1000

РОЗДІЛ 3. ДЕРЕВА ВАН ЕМДЕ БОАРСА

3.1. Опис вЕБ дерев

Припустимо, що ми можемо отримати номер для кожного елемента в наборі даних. Немає гарантій, що ці номери не будуть однаковими для різних елементів або гарантій, що кожному номеру відповідає хоча б один елемент. Тобто маємо велику булеву маску ~ розріджений масив. Шукати сусідів у ньому не так просто, як у повністю заповненому масиві, бо жадібний пошук може зайняти O(n). Щоб уникнути цього, можна використати дерева ван Емде Боаса [44]. Вони здатні робити додавання та пошук сусідів за O(log(log(u))), де u ($u$niverse) – це максимальна кількість елементів. В нашому випадку довжина коду, що повертає нейрохешування, є константною, а отже операції додавання та пошуку у дереві теж константні. Причому константа є дуже маленькою.

На жаль, зменшення складності по часу неминуче веде до збільшення складності по пам’яті. Можна провести аналогію між цими деревами та сортуванням підрахунком. У випадках коли швидкість стоїть на першому місці, на пам’ять не звертають увагу.

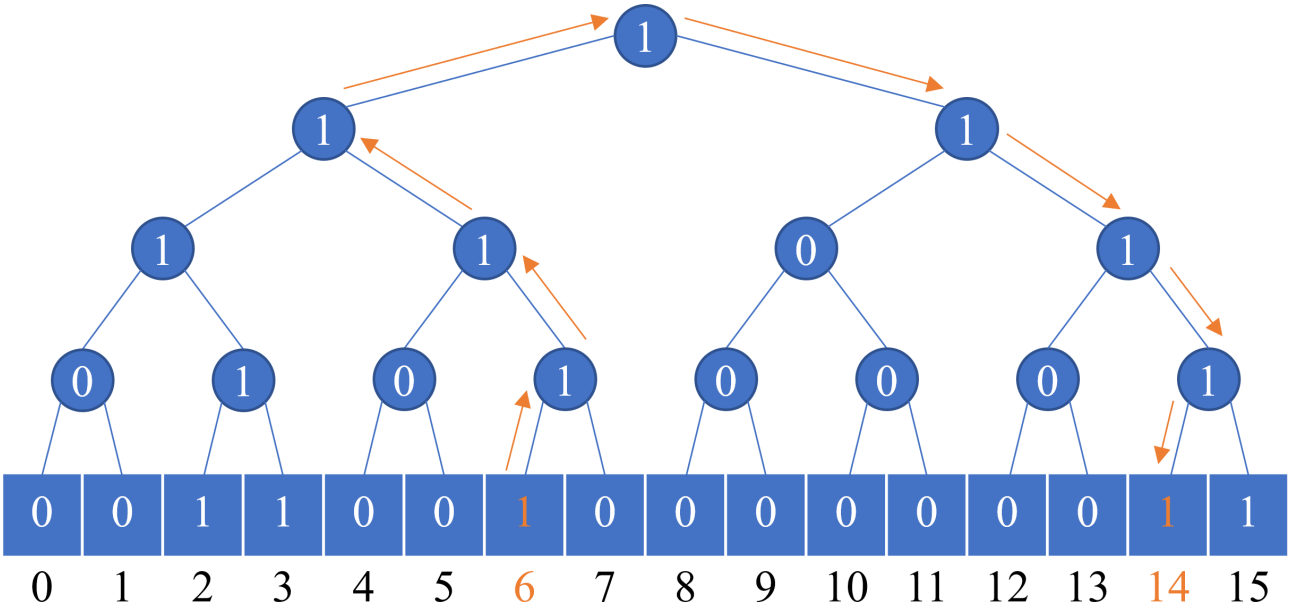

Рисунок 15 – Прототип вЕБ дерева, що містить елементи 2, 3, 6, 14, 15. Помаранчевим кольором виділено пошук наступника 6. Ним є елемент 14

Насправді дерева складніші, тому що мають більше ніж два розгалуження у кожній вершині для економії пам’яті. Вони дуже складні в реалізації, бо операції над ними мають нетривіальну рекурсивну природу. Також вЕБ дерева не паралелізуються, бо пошук наступного сусіда залежить від результату попереднього пошуку.

Щоб зменшити константи часу пошуку та додавання, доцільно доповнити вЕБ дерева двозв’язним списком та наступним відображенням (C++ map, Python dict): {номер_елементу: вказівник_на_вузол_списку}.

Остаточно маємо такий порядок дій при додаванні:

1. Знаходимо номер елемента *N* за допомогою нейрохешування
2. За допомогою дерева встановлюємо номер попередника і наступника *N*
3. За допомогою відображення знаходимо вузли попер. і наступника *N* у списку
4. Замінюємо їм відповідні вказівники на вузол *N*

При пошуку порядок дій той самий, але на кроці 4 треба запустити два жадібні пошуки по *k*/2 сусідів для попереднього і наступного вузла списку.

3.2. Тестування вЕБ дерев

Як було зазначено раніше, дерева складні в реалізації. Їхні показники пам’яті та швидкості різняться від імплементації до імплементації. Щоб зрозуміти межі їхньої продуктивності, були проведені тести різних імплементацій знайдених в інтернеті. Деякі з них (opengenus, github/Petar) виявилися непрацездатними. github/Julian написана мовою Python, усі інші мовою C++. Зовнішньо github/Julian має дуже якісний код і покриття тестами, але його результати виявилися гіршими за будь-яку з C++ імплементацій.

Швидкість додавання та пошуку має інтервали, бо вона залежить від кількості елементів у дереві. Для тестування бралися дві кількості: мінімально можлива 32′000 та максимально можлива 2^24. Код тестування наведений у додатку А. Отримані результати підтверджують, що за допомогою вЕБ дерев у реальному часі можливо робити запити великої кількості сусідів.

Імплементація	Обсяг пам'яті depth=24, Гб	Швидкість додавання k=32′000, 1/сек ~ Гц	Швидкість пошуку k=32′000, 1/сек
opengenus[4]	–		
github/Petar[5]	–		
github/Julian[6]	27.3	[1; 1]	[2; 11]
github/TISparta[7]	3.7	[38; 48]	[39; 96]
github/dragoun[8]	1.3	[15; 24]	[26; 49]
geeksforgeeks[9]	3.0	[34; 77]	[43; 85]

Таблиця 4 – Заміри різних вЕБ дерев. За пам'яттю оптимальною є імплементація dragoun, за швидкістю і пам'яттю – geeksforgeeks

Метод	Розмір елемента	Додав. k= 32′000, сек	Запит k= 32′000, сек	Обсяг пам'яті |db|=5.8e9, Гб
Наївний: argsort(norm(database – query))	768 f32	1′254	10′535	17′817
argsort(hamming_dist(database – query))	32 b8	40	345	185.6
argsort(abs(decimal_database – d_query))	1 i32	4.7	740	23.2
Python булева маска	1 b8	1/350	18.6	5.8
Python вЕБ дерево (github/Julian)	–	1.09	0.35	6′978
C++ булева маска	1 b8	1/1′000	[2.6e-4; 4]	5.8
C++ вЕБ дерево (github/dragoun) + Відображення, двозв'язний список + BEiT CIBHash	–	1/13 + 3.2e-7 + 3.84e-3 = 1/12	1/15 + 4.3e-4 + 3.84e-3 = 1/14	332.8 + 562.6 = 895.4
Faiss GPU (OPQ16_64,IVF1000HNS…)	8 i8	1.25	1.57	88.7

Таблиця 5 – Заміри різних *k*-NN підходів.
Червоним відмічені неприйнятні показники, помаранчевим – посередні.
768 f32 – vector[768] of 32 bit float, 32 b8 – vector[32] of 8 bit boolean і т. д.

Цікавим є перехід між Faiss та запропонованим підходом з двозв'язним списком. Збільшення швидкості у ~10 разів призводить до збільшення у ~10 разів затрат по пам'яті. Може здатися, що ~900 Гб оперативної пам'яті це недоступна кількість, але наразі у Google Cloud є віртуальні машини до 11776 Гб. У нашому випадку ВМ з 976 Гб коштує від 6$ за годину[10].

[4] https://iq.opengenus.org/van-emde-boas-tree
[5] https://github.com/PetarV-/Algorithms
[6] https://github.com/Julian/veb
[7] https://github.com/TISparta/Van-Emde-Boas-tree
[8] https://github.com/dragoun/veb-tree
[9] https://geeksforgeeks.org/proto-van-emde-boas-tree-set-6-query-successor-and-predecessor
[10] https://cloud.google.com/compute/vm-instance-pricing#m3_machine_types

ВИСНОВКИ

Був запропонований нейромережевий підхід який обходить наявні методи пошуку сусідів у швидкості. Це може зменшити час виводу для високонавантаженої питально-відповідної системи або дати змогу використовувати зорову пам’ять у реальному часі агентові під час навчання з підкріпленням. Бо найкращі бібліотеки пошуку витрачають щонайменше пів секунди на один запит при k=32′000 на The Pile. Але цей набір даних далеко не найбільший з наявних. Якщо візуальні дані будуть безперервно поступати з частотою людського ока із віртуального середовища при навчанні з підкріпленням, то швидкості жодної з бібліотек не вистачить на пошук у такому обсязі даних.

Попри високу швидкість, точність пошуку залишається доволі низкою: 0.59 проти Faiss 0.89. Скоріш за все, це пов’язано з тим, що все нейрохешування зводиться до тренування двох лінійних шарів. Має сенс збільшити кількість шарів та замінити їх на Gated MLP [45], бо такі шари показують схожі з механізмом уваги результати залишаючись простими у структурі та швидкими у виконанні.

Специфічний до зображень метод нейрохешування WCH показує порівняну із Faiss точність. Це говорить, що такий потенціал мають і мультимодальні методи.

СПИСОК ЛІТЕРАТУРИ

1. Vaswani A., Shazeer N.M., Parmar N., Uszkoreit J., Jones L., Gomez A.N., Kaiser L., та Polosukhin I. Attention is All you Need // NIPS. 2017.
2. Dosovitskiy A., Beyer L., Kolesnikov A., Weissenborn D., Zhai X., Unterthiner T., Dehghani M., Minderer M., Heigold G., Gelly S., Uszkoreit J., та Houlsby N., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," // ArXiv, Vol. abs/2010.11929, 2020.
3. Guu K., Lee K., Tung Z., Pasupat P., та Chang M.W., "REALM: Retrieval-Augmented Language Model Pre-Training," // ArXiv, Vol. abs/2002.08909, 2020.
4. Karpukhin V., Oğuz B., Min S., Lewis P., Wu L.Y., Edunov S., Chen D., та Yih W.T. Dense Passage Retrieval for Open-Domain Question Answering // Conference on Empirical Methods in Natural Language Processing. 2020.
5. Lewis P., Perez E., Piktus A., Petroni F., Karpukhin V., Goyal N., Kuttler H., Lewis M., Yih W.T., Rocktäschel T., Riedel S., та Kiela D., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," // ArXiv, Vol. abs/2005.11401, 2020.
6. Izacard G., Grave E. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering // Conference of the European Chapter of the Association for Computational Linguistics. 2020.
7. Borgeaud S., Mensch A., Hoffmann J., Cai T., Rutherford E., Millican K., van den Driessche G., Lespiau J.B., Damoc B., Clark A., та ін. Improving language models by retrieving from trillions of tokens // International Conference on Machine Learning. 2021.
8. Fajcik M., Docekal M., Ondrej K., та Smrz P. R2-D2: A Modular Baseline for Open-Domain Question Answering // Conference on Empirical Methods in Natural Language Processing. 2021.

9. Izacard G., Lewis P., Lomeli M., Hosseini L., Petroni F., Schick T., Yu J.A., Joulin A., Riedel S., та Grave E., "Few-shot Learning with Retrieval Augmented Language Models," // ArXiv, Vol. abs/2208.03299, 2022.
10. Wu Y., Rabe M.N., Hutchins D.S., та Szegedy C., "Memorizing Transformers," // ArXiv, Vol. abs/2203.08913, 2022.
11. Bertsch A., Alon U., Neubig G., та Gormley M.R., "Unlimiformer: Long-Range Transformers with Unlimited Length Input," // ArXiv, Vol. abs/2305.01625, 2023.
12. Gionis A., Indyk P., та Motwani R. Similarity Search in High Dimensions via Hashing // Very Large Data Bases Conference. 1999.
13. Johnson J., Douze M., та Jégou H., "Billion-Scale Similarity Search with GPUs," // IEEE Transactions on Big Data, Vol. 7, 2017. с. 535-547.
14. Guo R., Sun P., Lindgren E., Geng Q., Simcha D., Chern F., та Kumar S. Accelerating Large-Scale Inference with Anisotropic Vector Quantization // International Conference on Machine Learning. 2020.
15. Jégou H., Douze M., та Schmid C., "Product Quantization for Nearest Neighbor Search," // IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 33, 2011. с. 117-128.
16. Ge T., He K., Ke Q., та Sun J., "Optimized Product Quantization," // IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 36, 2014. с. 744-755.
17. Malkov Y.A., Yashunin D.A., "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," // IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 42, 2016. с. 824-836.
18. Bulatov A., Kuratov Y., та Burtsev M.S., "Scaling Transformer to 1M tokens and beyond with RMT," // ArXiv, Vol. abs/2304.11062, 2023.

19. Gao L., Biderman S.R., Black S., Golding L., Hoppe T., Foster C., Phang J., He H., Thite A., Nabeshima N., Presser S., та Leahy C., "The Pile: An 800GB Dataset of Diverse Text for Language Modeling," // ArXiv, Vol. abs/2101.00027, 2020.
20. Touvron H., Lavril T., Izacard G., Martinet X., Lachaux M.A., Lacroix T., Rozière B., Goyal N., Hambro E., Azhar F., та ін., "LLaMA: Open and Efficient Foundation Language Models," // ArXiv, Vol. abs/2302.13971, 2023.
21. Krizhevsky A. Learning Multiple Layers of Features from Tiny Images 2009.
22. Chua T.S., Tang J., Hong R., Li H., Luo Z., та Zheng Y. NUS-WIDE: a real-world web image database from National University of Singapore // ACM International Conference on Image and Video Retrieval. 2009.
23. Lin T.Y., Maire M., Belongie S.J., Hays J., Perona P., Ramanan D., Dollár P., та Zitnick C.L. Microsoft COCO: Common Objects in Context // European Conference on Computer Vision. 2014.
24. Shen Y., Qin J., Chen J., Yu M., Liu L., Zhu F., Shen F., та Shao L., "Auto-Encoding Twin-Bottleneck Hashing," // 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020. с. 2815-2824.
25. Shen Y., Yu J., Zhang H., Torr P.H.S., та Wang M. Learning to Hash Naturally Sorts // International Joint Conference on Artificial Intelligence. 2022.
26. Yu J., Qiu H., Chen D., та Zhang H. Weighted Contrastive Hashing // Asian Conference on Computer Vision. 2022.
27. Cao Z., Long M., Wang J., та Yu P.S., "HashNet: Deep Learning to Hash by Continuation," // 2017 IEEE International Conference on Computer Vision (ICCV), 2017. с. 5609-5618.
28. Chen T., Kornblith S., Norouzi M., та Hinton G.E., "A Simple Framework for Contrastive Learning of Visual Representations," // ArXiv, Vol. abs/2002.05709, 2020.
29. LeCun Y., Courant. A Path Towards Autonomous Machine Intelligence Version 0.9.2, 2022-06-27 2022.

30. Deng J., Dong W., Socher R., Li L.J., Li K., та Fei-Fei L., "ImageNet: A large-scale hierarchical image database," // 2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009. с. 248-255.
31. Oquab M., Darcet T., Moutakanni T., Vo H.Q., Szafraniec M., Khalidov V., Fernandez P., Haziza D., Massa F., El-Nouby A., та ін., "DINOv2: Learning Robust Visual Features without Supervision," // ArXiv, Vol. abs/2304.07193, 2023.
32. Li C., Yang J., Zhang P., Gao M., Xiao B., Dai X., Yuan L., та Gao J., "Efficient Self-supervised Vision Transformers for Representation Learning," // ArXiv, Vol. abs/2106.09785, 2021.
33. Assran M., Duval Q., Misra I., Bojanowski P., Vincent P., Rabbat M.G., LeCun Y., та Ballas N., "Self-Supervised Learning from Images with a Joint-Embedding Predictive Architecture," // ArXiv, Vol. abs/2301.08243, 2023.
34. Caron M., Touvron H., Misra I., J'egou H., Mairal J., Bojanowski P., та Joulin A., "Emerging Properties in Self-Supervised Vision Transformers," // 2021 IEEE/CVF International Conference on Computer Vision (ICCV), 2021. с. 9630-9640.
35. Bardes A., Ponce J., та LeCun Y., "VICReg: Variance-Invariance-Covariance Regularization for Self-Supervised Learning," // ArXiv, Vol. abs/2105.04906, 2021.
36. Ermolov A., Siarohin A., Sangineto E., та Sebe N. Whitening for Self-Supervised Representation Learning // International Conference on Machine Learning. 2020.
37. Zbontar J., Jing L., Misra I., LeCun Y., та Deny S. Barlow Twins: Self-Supervised Learning via Redundancy Reduction // International Conference on Machine Learning. 2021.
38. Grill J.B., Strub F., Altch'e F., Tallec C., Richemond P.H., Buchatskaya E., Doersch C., Pires B.Á., Guo Z.D., Azar M.G., та ін., "Bootstrap Your Own Latent:

A New Approach to Self-Supervised Learning," // ArXiv, Vol. abs/2006.07733, 2020.

39. Luo X., Chen C., Zhong H., Zhang H., Deng M., Huang J., та Hua X., "A Survey on Deep Hashing Methods," // ACM Transactions on Knowledge Discovery from Data, Vol. 17, 2020. с. 1-50.
40. Qiu Z., Su Q., Ou Z., Yu J., та Chen C. Unsupervised Hashing with Contrastive Information Bottleneck // International Joint Conference on Artificial Intelligence. 2021.
41. Ng K., Zhu X., Hoe J.T., Chan C.S., Zhang T., Song Y.Z., та Xiang T., "Unsupervised Hashing via Similarity Distribution Calibration," // ArXiv, Vol. abs/2302.07669, 2023.
42. Simonyan K., Zisserman A., "Very Deep Convolutional Networks for Large-Scale Image Recognition," // CoRR, Vol. abs/1409.1556, 2014.
43. Wang W., Bao H., Dong L., Bjorck J., Peng Z., Liu Q., Aggarwal K., Mohammed O.K., Singhal S., Som S., та Wei F., "Image as a Foreign Language: BEiT Pretraining for All Vision and Vision-Language Tasks," // ArXiv, Vol. abs/2208.10442, 2022.
44. van Emde Boas P., "Preserving order in a forest in less than logarithmic time," // 16th Annual Symposium on Foundations of Computer Science (sfcs 1975), 1975. с. 75-84.
45. Liu H., Dai Z., So D.R., та Le Q.V. Pay Attention to MLPs // Neural Information Processing Systems. 2021.

ДОДАТОК А. КОД ТЕСТУВАННЯ ДЕРЕВ

```cpp
#include <cmath>
#include <iostream>
#include <vector>
#include <numeric>
#include <chrono>
#include <assert.h>
#include <cstdint>
#include <random>

#include "vEBTree.hpp"
#include "util.hpp"

using namespace std;

int main() {
    const int32_t sz = 1 << 24;
    const int32_t filled = sz;
    const int32_t nk = 32000;
    const int32_t log_every_nk = 500;

    assert(nk <= filled);
    assert(filled <= sz);

    vEBTree tree(sz);

    int32_t* v = new int32_t[sz];
    iota(v, v + sz, 0);
    auto rng = default_random_engine{};
    shuffle(v, v + sz, rng);

    auto a = chrono::high_resolution_clock::now();
    for (int32_t i = 0; i < filled; ++i) {
        tree.insert(v[i]);
    }
    auto b = chrono::high_resolution_clock::now() - a;

    float seconds = chrono::duration_cast<chrono::milliseconds>(b).count() / 1000.;
    cout << float(filled) / nk / seconds << endl;

    int32_t query, succ;
    a = chrono::high_resolution_clock::now();
    for (uint32_t i = 0; true; ++i) {
        query = v[rand() % (filled - 1)];
        succ = tree.successor(query);

        if (i % (log_every_nk * nk) == 0 && i) {
            b = chrono::high_resolution_clock::now() - a;

            if (filled == sz) {
                assert(succ == query + 1);
            }

            seconds = chrono::duration_cast<chrono::milliseconds>(b).count() / 1000.;
            cout << log_every_nk / seconds << ' ' << query << ' ' << succ << '\n';
            a = chrono::high_resolution_clock::now();
        }
    }

    return 0;
}
```