TARAS SHEVCHENKO NATIONAL UNIVERSITY OF KYIV
Faculty of Computer Science and Cybernetics
Department of Mathematical Informatics

Qualification work

to obtain a Master's degree

in the educational and scientific program "Artificial Intelligence"
specialty 122 "Computer Science" on the topic:

ASSOCIATIVE METAMEMORY

Prepared by a 4th year student
Vynnyk Dmytro Oleksandrovych ___________
(signature)

Supervisor:
Associate professor, PhD
Panchenko Taras Volodymyrovych ___________
(signature)

I certify that in this work there are no borrowings from the works of other authors without appropriate references.

Student ___________
(signature)

The work was reviewed and accepted for the defense at the meeting of the Department of Mathematical Informatics
"_____"_______________ 2023,
protocol No. _____
Head of Department
Tereshchenko V. M. ___________
(signature)

Kyiv – 2023

CONTENTS

ABSTRACT

The master's thesis consists of an introduction, three chapters, conclusions, a list of used sources (45 items), and 1 appendix. The work contains 15 figures and 5 tables. The total volume of the work is 32 pages, the main text of the work is laid out on 19 pages.

O(1) KNN, NEUROHASHING, INDEXES, VAN EMBDE BOARS TREE, INFORMATION RETRIEVAL, ASSOCIATIVE MEMORY, METAMEMORY, FAST DATA STRUCTURES, SOURCE KNOWLEDGE, RETRIEVAL-AUGMENTED, SELF-SUPERVISED.

The object of research is artificial episodic memory. The subject of research is associative metamemory.

The objective of the work is to develop an approach for quick searching and saving a multidimensional manifold of memories in a one-dimensional (linear) memory, which is inherent in all computers implementing the von Neumann architecture.

Development methods: multimodal artificial neural networks, similarity search methods for dense vectors, code quantization methods, proximity graph indexing methods. Development tools: Visual Studio and PyCharm development environments, C++ and Python 3 programming languages, VS Performance Profiler.

The results of the work: the necessity of using neurohashing has been proven, along with the inability of conventional algorithmic indexing methods to find and add a large number of vector representations in real-time as the database grows in size, a model has been trained, which returns a binary number that forms a linear order; a data structure has been implemented for fast neighbor search.

ABBREVIATIONS

k-NN	–	*k*-nearest neighbors algorithm
Faiss	–	Facebook AI Similarity Search
FPS	–	Frames per second
Glass	–	Graph Library for Approximate Nearest Search
GPU	–	Graphics Processing Unit
HNSW	–	Hierarchical Navigable Small World
IVF	–	Inverted File Index
LLaMA	–	Large Language Model
LSH	–	Locality-Sensitive Hashing
mAP	–	Mean Average Precision
MIPS	–	Maximum Inner Product Search
NGT	–	Neighborhood Graph and Tree
NMSLIB	–	Non-Metric Space Library
OPQ	–	Optimized Product Quantization
ScaNN	–	Scalable Nearest Neighbors

INTRODUCTION

Assessment of the current state of the development object

Transformers [1] are the leading architecture of artificial neural networks for modeling and predicting sequences. In the simplest case, generative transformers learn to predict the next token using the information from all previous tokens within the window. For example, tokens can be words embedded in a multidimensional vector space or small rectangular (*n, m*) fragments of an image. They are decomposed into one-dimensional vectors (*n* * *m*) and inserted into the vector space according to the principle similar to words [2].

One of the approaches to improving the output quality of generative transformers is the use of an external knowledge base (retrieval-augmented) from which source knowledge is taken. In the case of text transformers, relevant sentences are taken from the database. Relevance is determined by the scalar product (MIPS), cosine similarity, or Euclidean distance between embeddings of these sentences. The prominent representatives of this approach are REALM [3], DPR [4], RAG [5], FiD [6], RETRO [7], R2-D2 [8], and Atlas [9]. These architectures mainly solve the open-domain question answering problem.

In general, for the retrieval of relevant elements, *k* nearest neighbors (*k*-NN) are searched. The knowledge base serves as a cognitive analog of episodic memory, while *k*-NN acts as an analog of metamemory. Metamemory plays a key role in thinking: a person can immediately, without much thought, say that they do not know the fifth largest dinosaur. This ability allows us to save significant brain resources because otherwise, it would be necessary to recall everything a person knows to conclude that they do not possess specific knowledge.

In addition to knowledge, embeddings from previous contexts can be stored in the database. This is done in Memorizing Transformers [10] and Unlimiformer [11] architectures. This implicitly expands the current context to a size limited only by the quality of the search.

Relevance of the work and the reasons for its implementation

The number of vectors in the knowledge base can reach billions. For fast search, modern architectures use multi-level similarity indexes based on quantization and graph search. Their query time increases linearly as *k* increases. Empirically, this allows us to retrieve the order of a thousand sentences of 64 tokens without a significant impact on training time. This amount of text is not enough for some tasks, for example, summarization. In the field of computer vision, this is even more noticeable, because a thousand images represent only ~17 seconds of video at a frequency of 60 FPS.

Evidently, the human brain does not spend a lot of time on the assimilation of sensory information. In other words, indexing happens instantaneously during the integration of new information. In addition, mind-wandering also does not require a long search. Instead, commonly used *k*-NN algorithms are sublinear at best, such as LSH [12] – $O(n^{p < 1})$.

Objective and tasks of the work

The objective of the work is to develop an approach for adding to and searching within the knowledge base in constant time, using neurohashing and fast data structures. To achieve this goal, the following tasks have been set:

- measure the running time of the most used similarity indexes on synthetic data
- analyze available neurohashing methods
- adapt one of them for multimodal mode
- compare popular datasets
- index one of the datasets
- select a fast data structure for this index

SECTION 1. SIMILARITY INDEXES

1.1. Comparison of similarity search methods

From search libraries based on peer-reviewed papers, the most famous are Meta Faiss [13] and Google ScaNN [14], from non-academic – Spotify Annoy, NMSLIB, Kakao N2, Yahoo NGT. In addition to open libraries, there are many semi-commercial vector databases. Among them: Milvus > Weaviate > Qdrant (ranked by speed) and Pinecone (not tested).

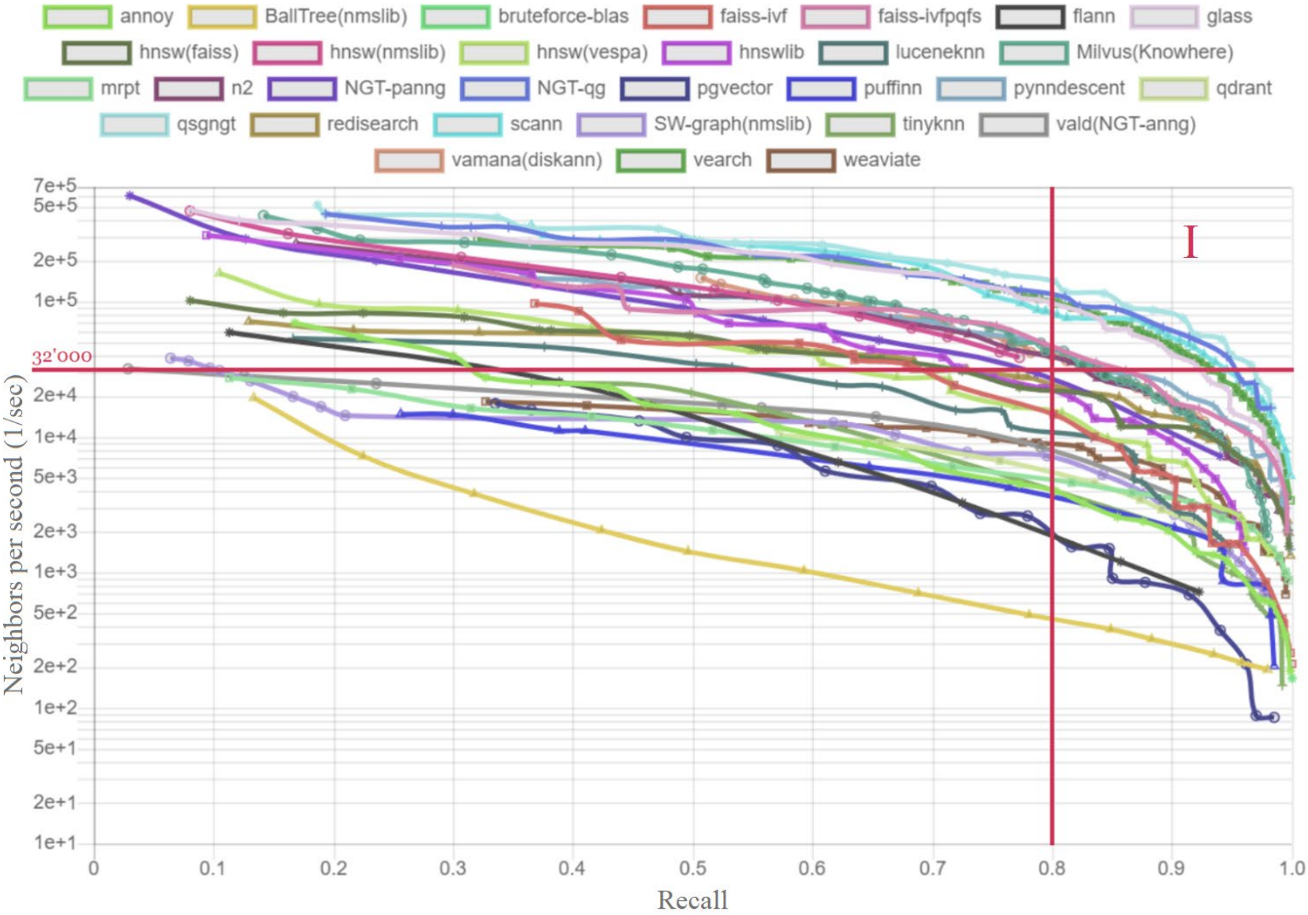


Figure 1 – Performance measurements of a large number of methods[1].
Red lines demarcate the 1st quadrant: number of queries > 32′000, recall > 0.8

A priori, we will set a lower performance limit of 32,000 requests per second with a recall of at least 0.8. For small *k,* this amount is sufficient for real-time execution. Then, after filtering, 11 methods will remain:

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

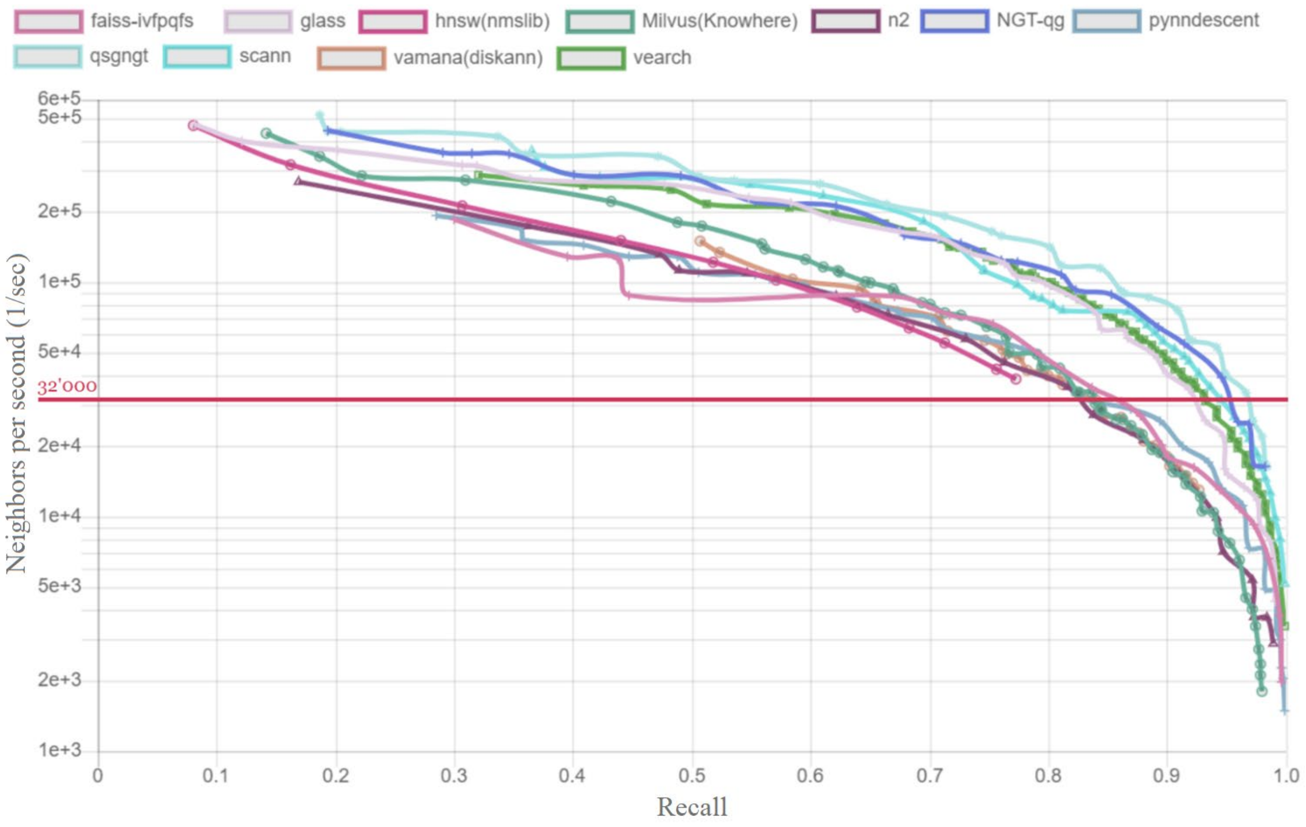


Figure 2 – Performance measurements of real-time methods

The list of real-time methods includes the previously mentioned peer-reviewed Faiss and ScaNN. A group of the fastest methods clearly stands out on the chart: qsgngt > NGT > Glass > ScaNN. Among them, NGT and ScaNN are widely used.

1.2. Description of the Faiss library

Faiss indexes are less memory-intensive than ScaNN, because ScaNN, in addition to storing the quantized index, needs to store the original vectors. Also, Faiss has better documentation. Therefore, it is advisable to start research with Faiss.

The description of Faiss from the repository: "Faiss is a library for efficient similarity search and clustering of dense vectors. It contains algorithms that search in sets of vectors of any size, up to ones that possibly do not fit in RAM. It also contains supporting code for evaluation and parameter tuning. Faiss is written in C++ with complete wrappers for Python/numpy. Some of the most useful algorithms are implemented on the GPU. It is developed primarily at Meta's Fundamental AI Research group".

One of the main methods for index formation is quantization. It is used to efficiently find similarities in high-dimensional data. The goal of quantization is to reduce the dimensionality of a dataset while preserving important information about the data. This provides faster and more efficient search algorithms that can be used to speed up tasks such as image and text searches.

In Faiss, the quantization is performed by dividing the dataset into several smaller subspaces [15], which are called Voronoi cells. Each Voronoi cell corresponds to a cluster of similar data points. Next, the centroids of these clusters are calculated and stored as a lookup table. When a query is performed on a dataset, it is first quantized into one of the Voronoi cells and then finds the nearest centroid.

The main advantage of quantization is that it reduces the amount of data that needs to be stored and searched, making it much faster than other methods. In addition, quantization can be easily parallelized, making it suitable for large-scale data analysis.

Faiss supports the comparison of Euclidean distances, scalar products, and cosine similarities. All three metrics are proportional to each other:

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

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

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

Euclidean distance should be used when the norms of the vectors make sense.

1.3. Faiss index factory

The Index Factory is a helper module in Faiss that provides a convenient way to create and configure index objects. The factory takes care of index initialization, training, and persistence, and provides a simple interface to configure various index parameters, such as the number of clusters, the number of bytes per code in quantization, and the number of levels.

In this work, the following configuration was used for testing: *OPQ16_64,IVF1000_HNSW32,PQ16x4fs*

OPQ [16] aims to reduce the distortion introduced by vector quantization by applying an orthogonal transformation to the input vectors before quantization. The transformation rotates and scales the vectors in such a way as to minimize the distance between the transformed vectors and the centroids of the clusters, resulting in more accurate quantization.

The OPQ algorithm works by first partitioning the input vectors into multiple blocks and then iteratively optimizing the rotation and scaling of each block to minimize the quantization error. Optimization is performed using gradient descent, with gradients calculated from a closed-form expression.

OPQ16 at the beginning of the configuration indicates only the rotation matrix, so at the end, we must have *PQ16x4fs,* which is responsible for the quantization process [15] and means 16 codes of 4 bits in *fast scan* mode. Thus, the code size is 16 * 4 / 8 = 8 bytes. The *_64* in *OPQ16_64* is an additional dimensionality reduction operation to 64. *IVF1000_HNSW32* indicates a combination of 1000-cell IVF and HNSW [17], where each centroid is connected to 32 others.

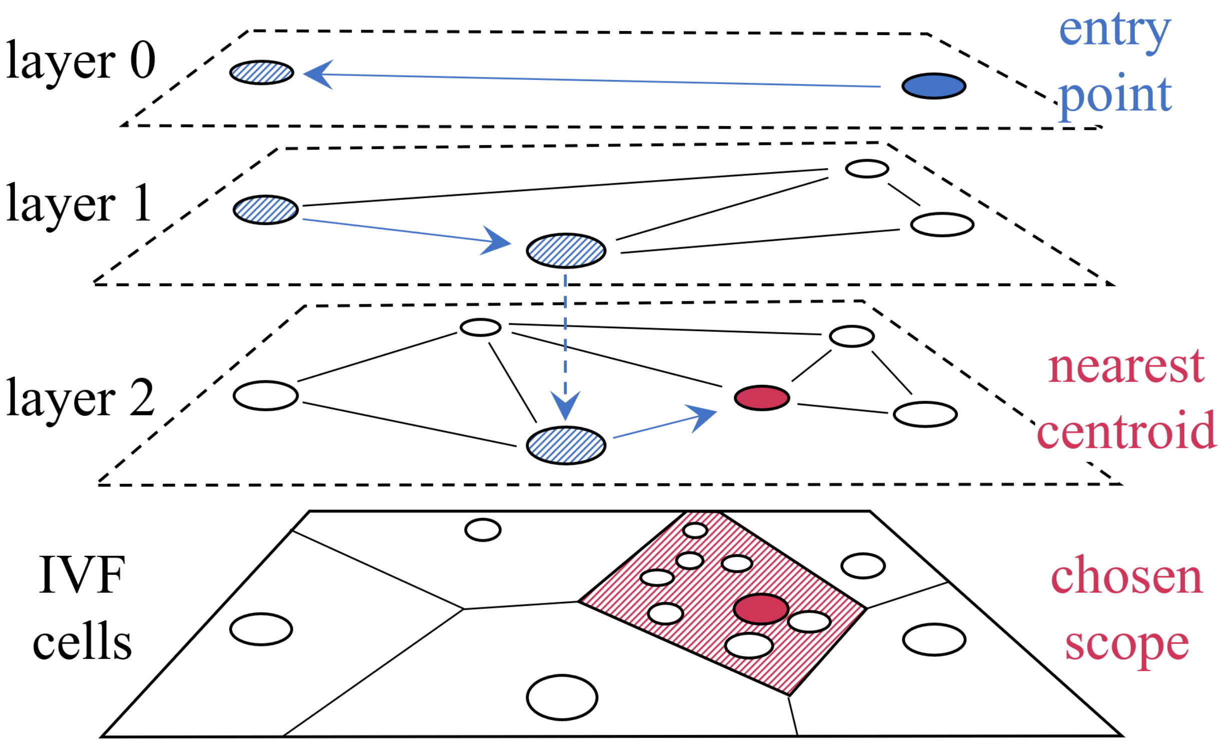


Figure 3 – HNSW and IVF. A replica from the Pinecone blog

1.4. Testing of indexes

To guarantee obtaining *k* neighbors in IVF, it is necessary to make samples on neighboring cells because the method is approximate. Moreover, with increasing *k* the number of required samples is also increasing. In Faiss, the nprobe parameter is responsible for the number of samples.

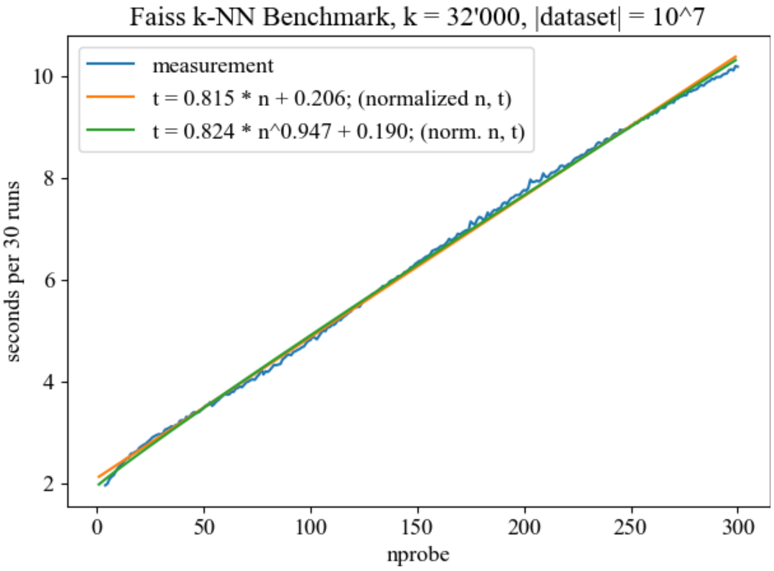


Figure 4 – Dependence of query time on nprobe

– In the legend of the chart, we provide approximations of measurements using first order polynomials and power functions with a shift. As we can see, the query time increases linearly (we consider the nature of the power of 0.947 to be erroneous).

The next step is to check the dependence of the request time on *k*:

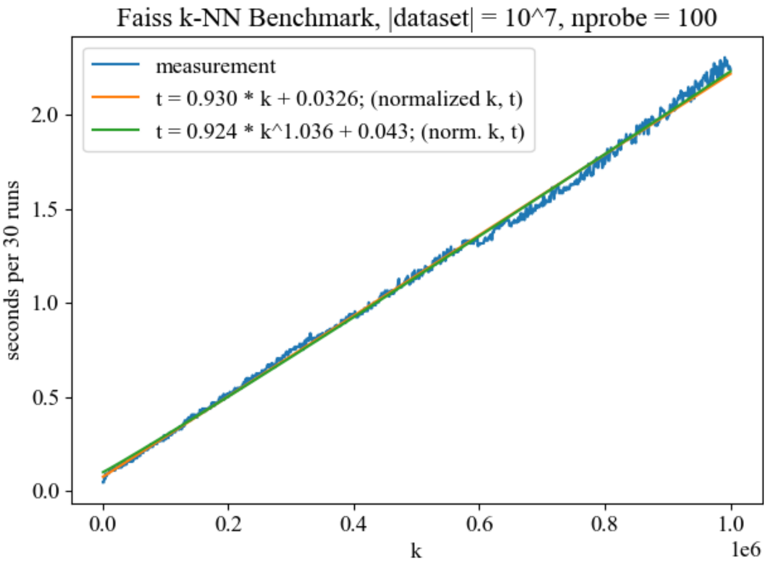


Figure 5 – Dependence of the request time on the number of neighbors in *k*-NN

This dependence also turned out to be close to linear (we will consider the nature of the power of 1.036 to be erroneous). Instead, the dependence on the size of the database goes from sublinear to superlinear as *k* increases (0.639 → 0.755 → 0.911 → 1.01 → 1.14):

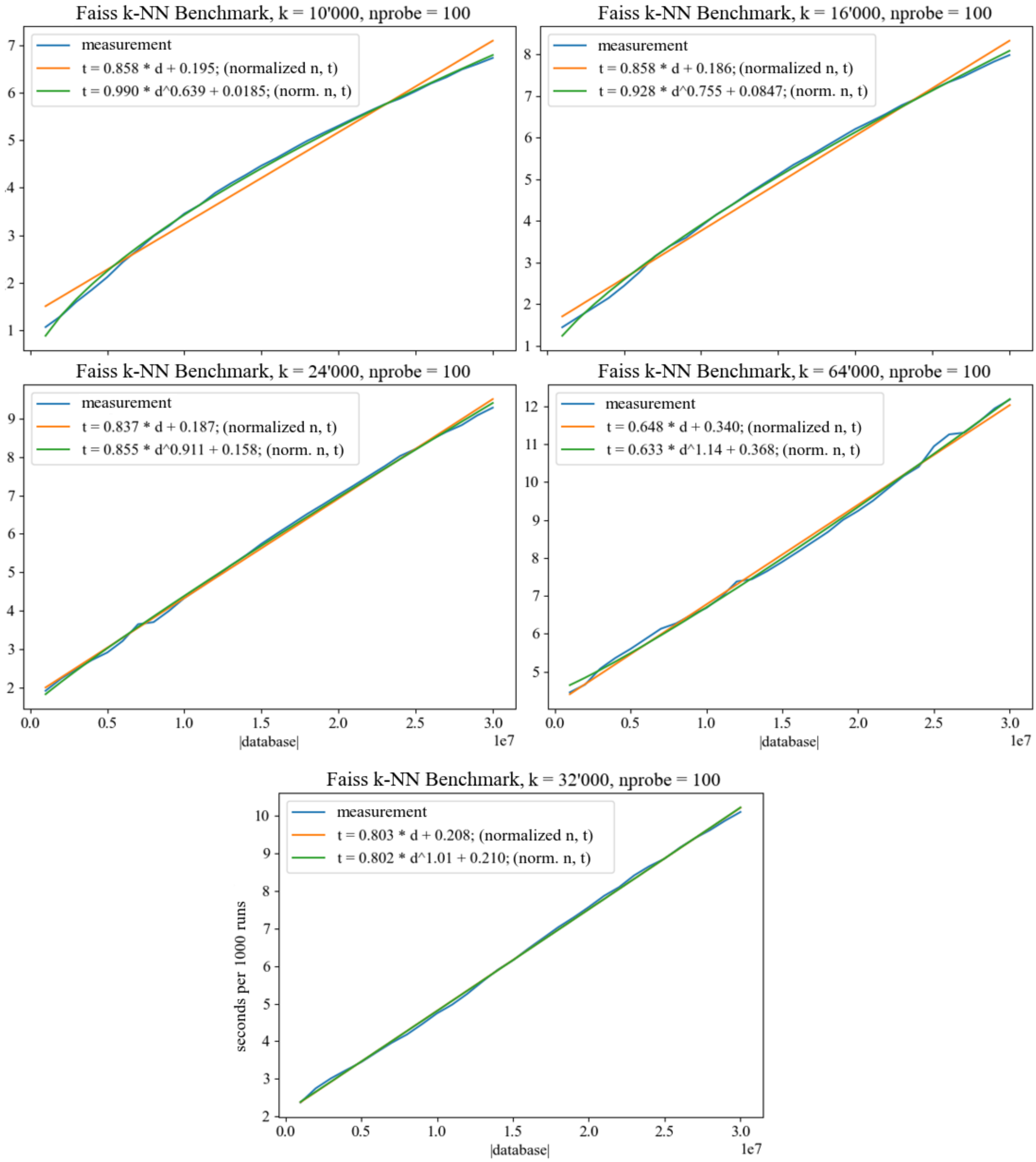


Figure 6 – Dependence of query time on database size

Given the recent advances in increasing the size of the context to ~2 million tokens [18], let us calculate how long the query will take in the case of the text dataset The Pile [19], which after processing contains 5.8 billion fragments of 64 tokens each. This is equivalent to *k*-NN with *k* = 2'048'000 / 64 = 32'000. That is, we can use the non-normalized analog t = 0.803 * d + 0.208:

t = 2.7e-07 * 5.8e9 + 2.1 ≈ 1568 seconds/1000 runs = 1.568 seconds/request

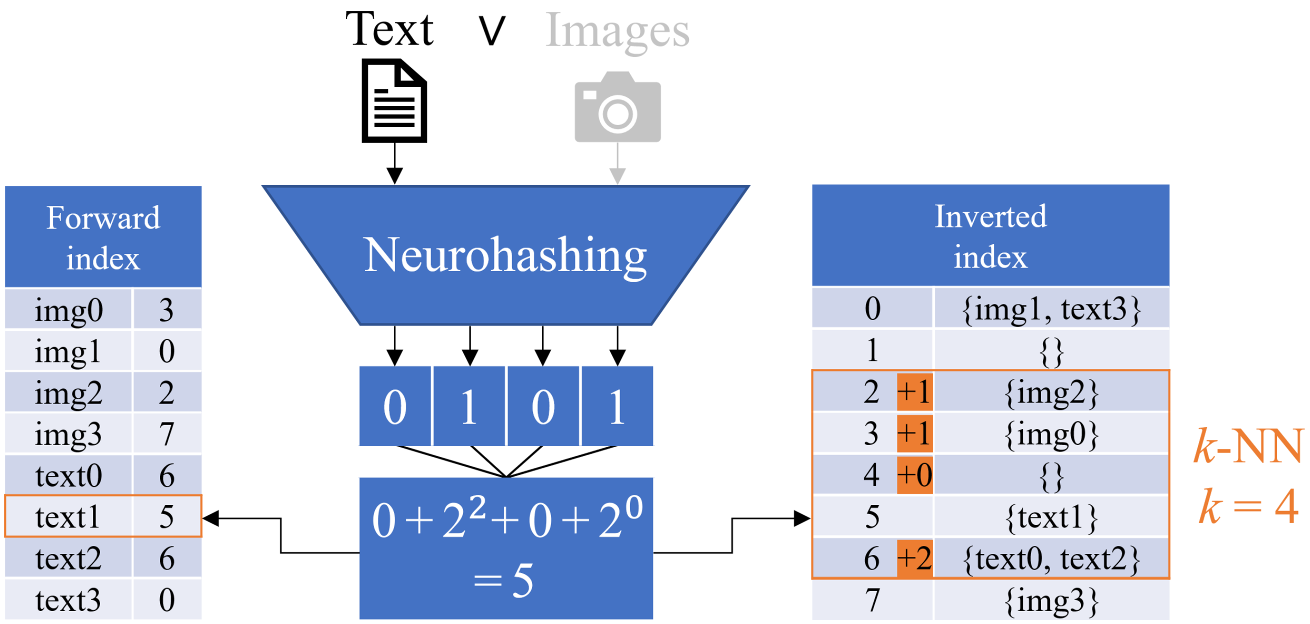
This exactly corresponds to the faiss-ivfpqfs chart in Fig. 2 and measured recall 0.89. From the same chart, it can be seen that the fastest methods ScaNN and NGT-qg are ahead of Faiss by up to three times.

This duration significantly slows down training. In the article that introduces the LLaMA generative transformer [20], the authors noted that the throughput of the model for 65 billion parameters is 380 tokens/second/GPU on 2048 A100 GPUs with 80 GB of video memory. That is, the bandwidth of the cluster is 778′240 tokens/second. If we go back to the ~2M long context, this is 2.634 seconds/request.

Besides slowing down training by 1.568 / (1.568 + 2.634) = 37% in the case of Faiss and 17% in the faster methods, the similarity indexes have quite a few counterintuitive hyperparameters. At the same time, artificial neural networks are quite often ahead of classical algorithms. Their forward propagation has a constant time within a few milliseconds. This suggests that there should be a neural network approach that works in *k* * O(const) = O(*k*).

SECTION 2. NEUROHASHING

2.1. Intuition



Forward index	
img0	3
img1	0
img2	2
img3	7
text0	6
text1	5
text2	6
text3	0

Inverted index		
0		{img1, text3}
1		{}
2	+1	{img2}
3	+1	{img0}
4	+0	{}
5		{text1}
6	+2	{text0, text2}
7		{img3}

Figure 7 – Concept of architecture. "text1" is added to the direct and inverted index under number 5. For *k*-NN, it remains to greedily go through the inv. index

The idea is to model a hierarchy of universals (features) using a hierarchy of numbers. So that the binary number 0100 (4) is semantically closer to 0110 (5) than to 1100 (12). To show that this is possible, consider the case of one-dimensional *k*-NN:

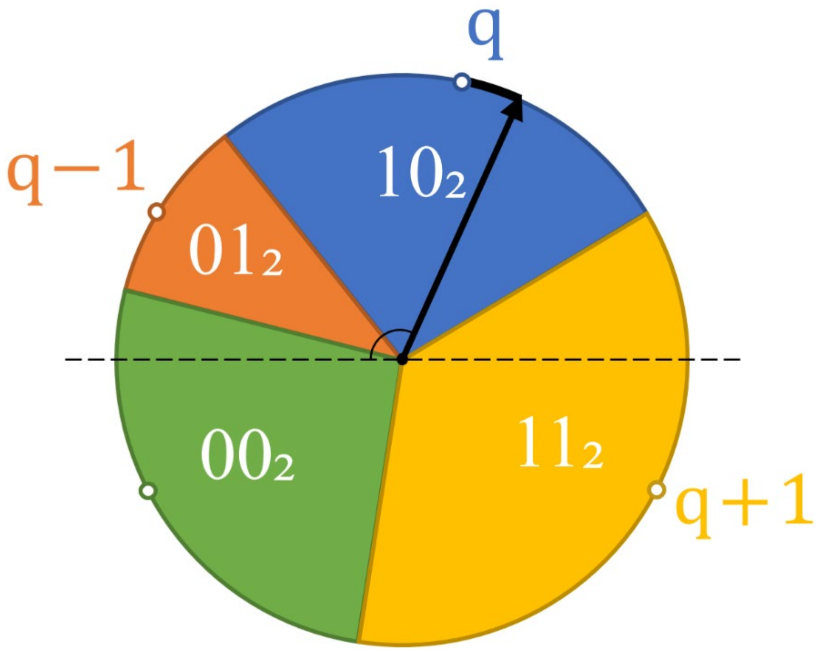


Figure 8 – Two-dimensional projection of a one-dimensional Voronoi diagram

The colored zones are responsible for belonging to a certain vertex. For example, the black query vector has entered the cell of the blue vertex. A neural network can easily learn the dependence of the index (color) on the angle of the vector. Binary outputs of the network are a natural way to learn a binary numerical index. If there are many points, and the cell numbers are arranged clockwise, then it is not difficult to use the index for searching *k* nearest neighbors – we need to take vertices with indices from q – k / 2 to q + k / 2, where q is the index of the query vector.

In the two-dimensional case, everything is much more complicated, because the ground truth order of vertices by distance is discontinuous, and therefore we have a decision boundary with self-intersections. This complicates learning and impairs the generalization abilities of the neural network. This can be compensated to some extent by having a large, balanced dataset during pretraining because such a set will force the neural network to efficiently compress the representation.

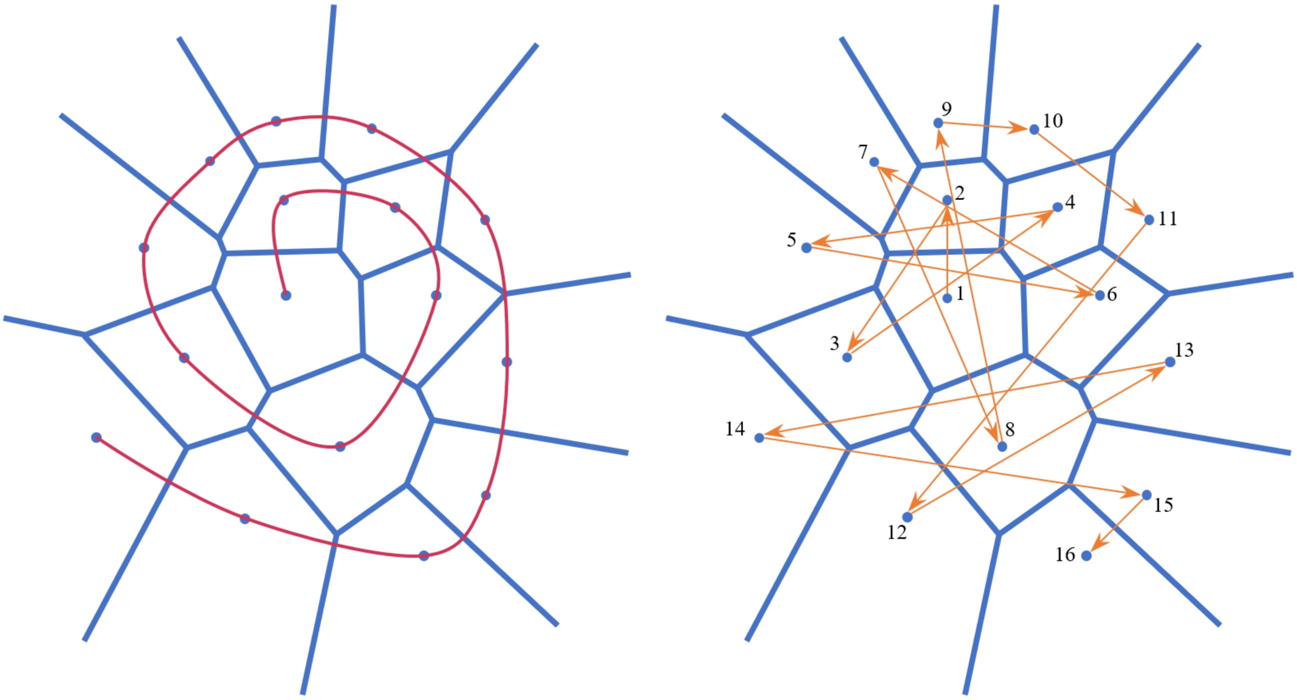


Figure 9 – Two-dimensional Voronoi diagram.
On the left is the desired smooth order of the vertices by distance from the first one, on the right is the discontinuous true order. A neural network learns the true order

2.2. Data description

Most works use three sets of images: CIFAR-10 [21], NUS-WIDE [22], MS COCO [23]. There are various filtered subsets and partitions of these sets. In CIFAR-10 each image has one of 10 classes, in NUS-WIDE-21 each image has up to 21 classes, in MS-COCO-2014 up to 80 classes. The set is divided into three parts: a train part, a test part (query), and a database. During testing a binary code (hash) is generated for the request image, which is searched in the database. Thus, the database can contain training images, because they do not overlap with test images.

The balancing of samples underwent changes during the development of the direction. Recent works TBH [24], NSH [25], WCH [26] converged on the same principle: CIFAR-10 – sampling with uniform distribution, NUS-WIDE-10 – each class has at least 100 images, MS-COCO-2014 – unbalanced sampling. Some past work either neglected to balance the dataset or contained errors. For example, in HashNet [27] the database sample contains query, which may result in perfect results on certain elements. Therefore, the hashing code was taken from the public DeepHash repository[2] after being checked for correctness.

Architectures; Dataset	**Query (Test)**	**Train**	**Database**	**Total**
HashNet; NUS-WIDE-81-m	5'000 (unbalanced)	10'000 (unbalanced)	Train + Database0 = 218'491	223'496
HashNet; MS-COCO-2014	5'000 (unbalanced)	10'000 (unbalanced)	Test! + Database0 = 112'218	122'218
CIBHash; CIFAR-10	5'000 (unbal.) (+ 5'000 valid)	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 (min. 100/class)	10'500 (min. 100/class)	Train + Database0 = 193'734	195'834
TBH, CIBHash, NSH, WCH; MS-COCO-2014	5'000 (unbalanced)	10'000 (unbalanced)	Train + Database0 = 117'218	122'218

Table 1 – Comparison of samples

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

2.3. CIBHash Binary Siamese Neurohashing

In these datasets, we have true labels (image classes), but we will use them only for testing. Because we aim to train a model without supervision. Such methods of machine learning are called self-supervised. One of the popular self-supervised approaches for learning vector representations is the use of dual architectures. They are also called Siamese architectures, distillation approaches, joint embedding architectures, and methods of mutual information maximization.

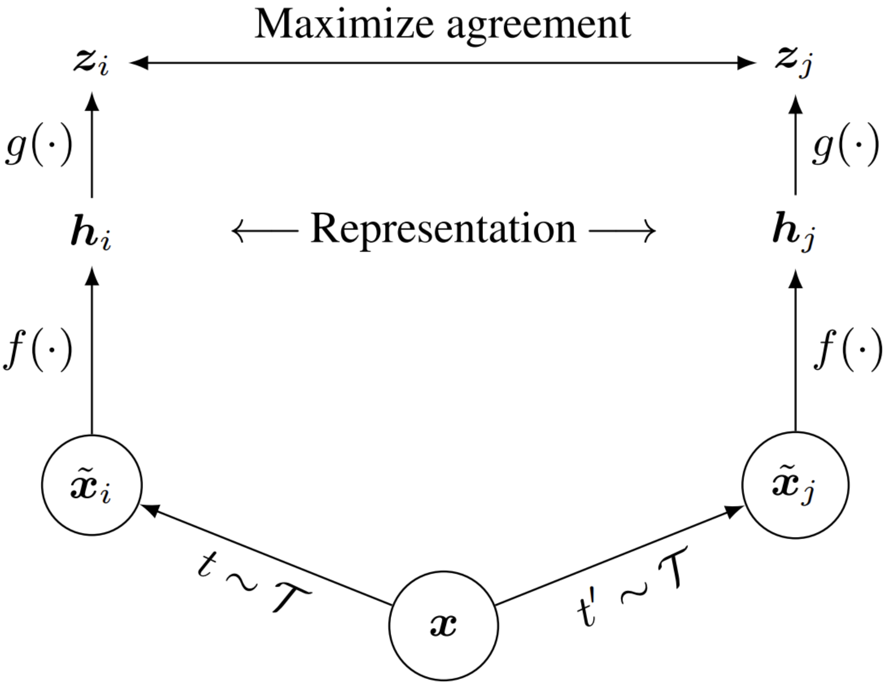


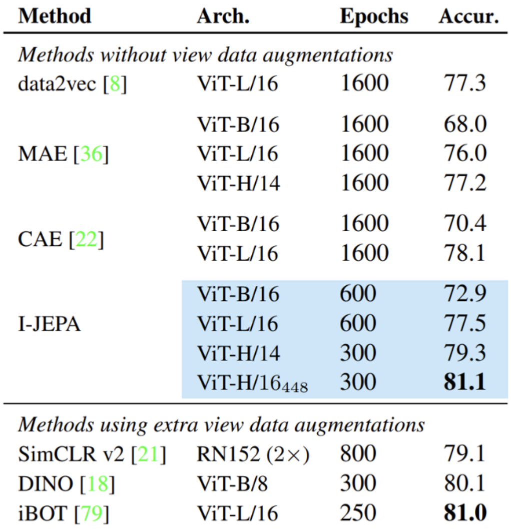
Figure 10 [28] – Contrast learning architecture SimCLR.

t and *t'* are augmentations obtained from one family *T*; *f(·)* is a coding network, *g(·)* is a projection head used only during training

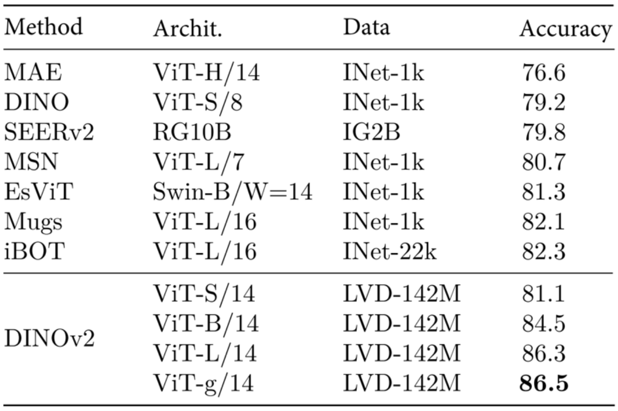
Most of them use contrast learning to avoid collapsing to a trivial solution. However, this requires a large selection of negative elements [29]. Instead, there are methods that do not require a negative sample. Instead, they use different regularizations to prevent collapse. Let us bring the ranking of these methods by classification accuracy on ImageNet-1k [30]:

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

Moreover, the methods that are oriented to the modality of the dataset (in the case of ImageNet, these are images) show themselves to be more accurate: DINOv2.. DINO. It makes no sense to adapt them to the text because their architecture is based on specific augmentations of images and other aspects not inherent in texts. VICReg..BYOL multimodal methods are less precise but more versatile. Practice shows that over time too specific methods give way to universal ones[3], but such methods apparently do not exist yet.



Method	Arch.	Epochs	Accur.
Methods without view data augmentations			
data2vec [8]	ViT-L/16	1600	77.3
MAE [36]	ViT-B/16	1600	68.0
	ViT-L/16	1600	76.0
	ViT-H/14	1600	77.2
CAE [22]	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**
Methods using extra view data augmentations			
SimCLR v2 [21]	RN152 (2×)	800	79.1
DINO [18]	ViT-B/8	300	80.1
iBOT [79]	ViT-L/16	250	**81.0**



Method	Archit.	Data	Accuracy
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**

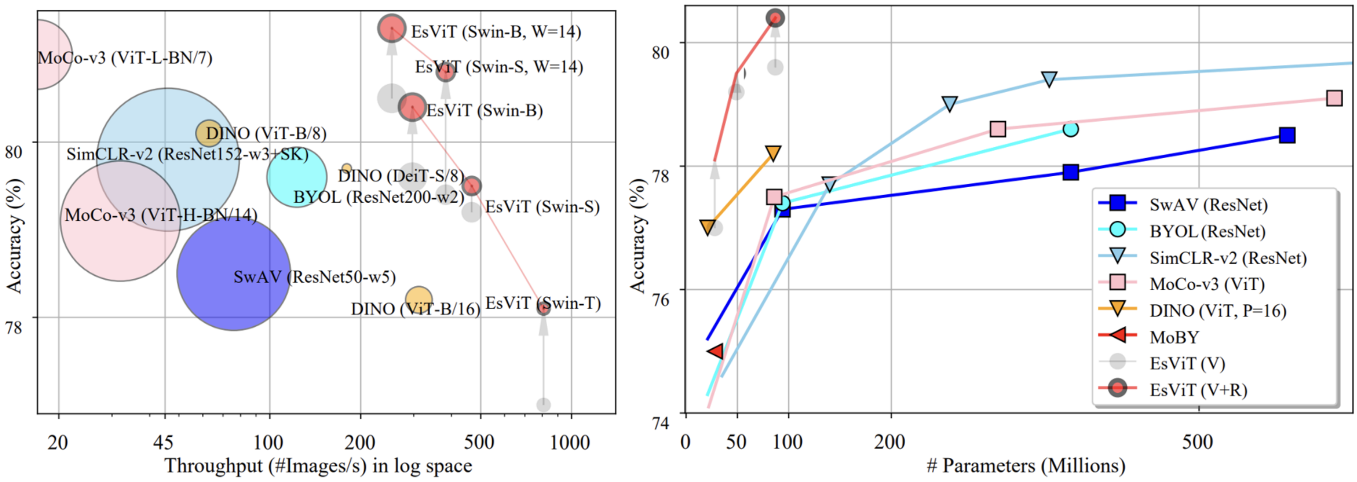


Figure 11 – Confirmation of ranking. Extract from articles [33], [31], [32]

Unfortunately, we cannot use the listed dual self-supervised methods, because their embeddings consist of floating-point numbers. Comparing such vectors is too slow. Therefore, neurohashing methods use binary codes. Conventional artificial

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

neural networks, by their very nature, are not capable of returning discrete values. However, there are many mathematical approaches that allow us to bypass this limitation. A comparison of binary neurohashing methods is provided in the review [39]. Among the latest, it is worth mentioning CIBHash [40], NSH [25], WCH [26], and SDC [41]. The further these methods depart from the original SimCLR architecture [28], the more there is an unjustified tendency to unnatural statistical approaches and various mathematical tricks.

In this work, we will use CIBHash, because NSH does not have an open-source code, in WCH they train the basic feature extraction model, which is unfair, and SDC has too complex configurations. CIBHash is a fairly simple architecture: a basic pretrained VGG feature extraction model [42] which is frozen during training, and two linear layers. Binarization is achieved using a straight-through gradient estimator: when backpropagating error, the *sign* function is simply skipped.

To achieve linear order, an error function was added converting from a binary code to a decimal number. That is, both parts of the dual model return one number, then these numbers are subtracted for the positive and negative elements of the sample. Positive elements are fragments of the same image and should have a smaller difference. Despite the simplicity of the approach, it should work well, because such an operation passes the gradient well.

The original feature or rather embedding extraction model, was replaced by the BEiT-3 multimodal model [43]. Potentially if text augmentations are added to CIBHash this will allow text to be indexed in addition to images. Also, BEiT performs better on image classification than VGG, and therefore its embeddings contain more information about the input data. This should increase CIBHash's performance.

✓	✗	✗	✓	✓	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					

Table 2 – Example of mAP@5 calculation

Method	CIFAR-10			NUS-WIDE			MS COCO		
	16 bit	32 bit	64 bit	16 bit	32 bit	64 bit	16 bit	32 bit	64 bit
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

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

2.4. BEiT multimodal transformer

Machine learning researchers have recently focused on developing general-purpose foundation models that can work across multiple modalities and adapt to a variety of downstream tasks. The Microsoft research group presented BEiT-3 – a multimodal foundation model for visual and speech-visual tasks. BEiT-3 interprets visual tokens as text in a foreign language that the authors call "Imglish" and performs unified masked "language" modeling on images, texts, and image-text pairs. The model learns on monomodal and multimodal data through a joint Multiway Transformer network, which uses a shared self-attention module and different forward propagation networks for each modality. BEiT-3 achieved superior performance in various tests:

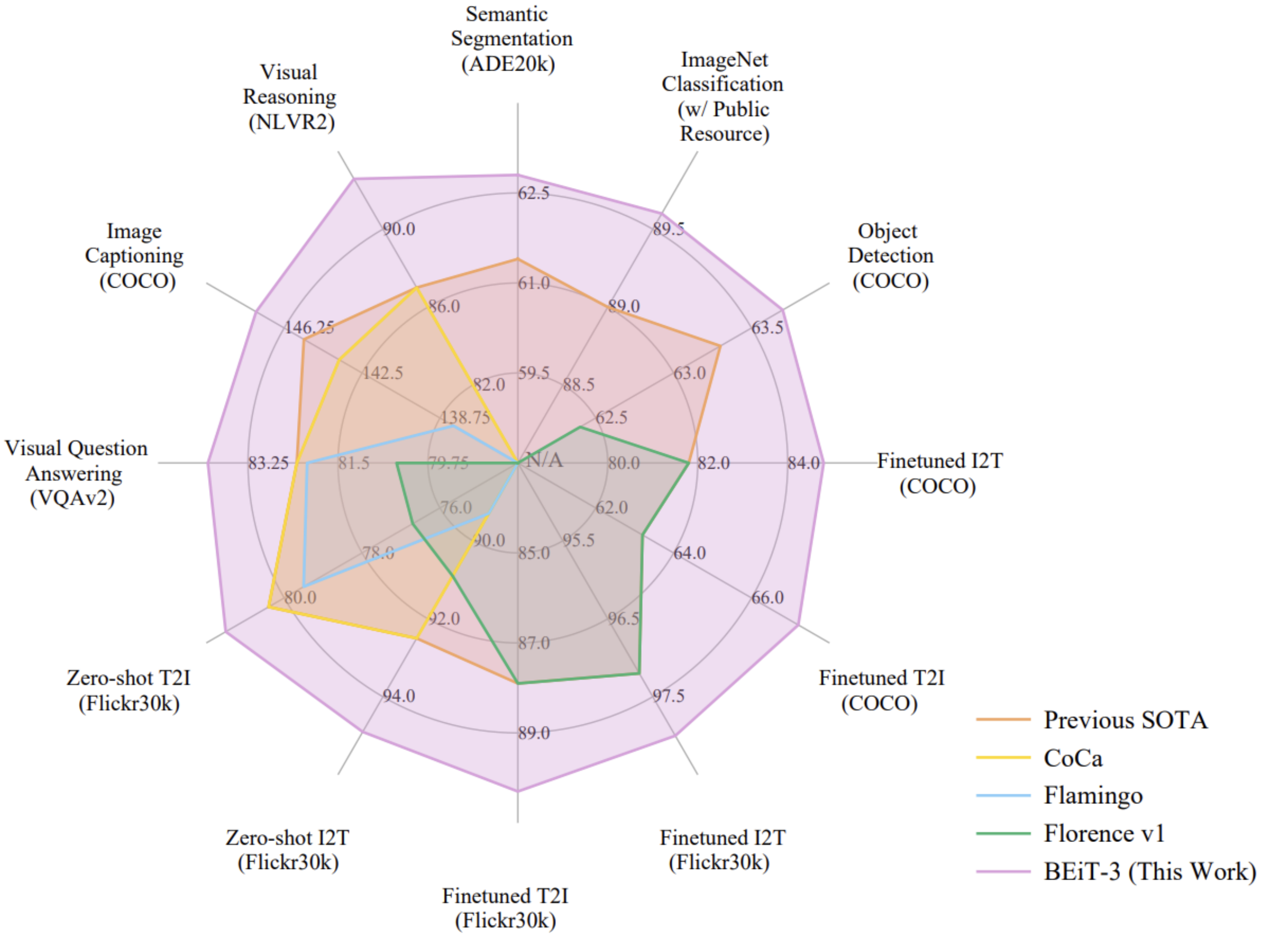


Figure 12 [43] – BEiT-3 is ahead of other foundation models.

I2T/T2I = image-to-text/text-to-image search

2.5. Results of training

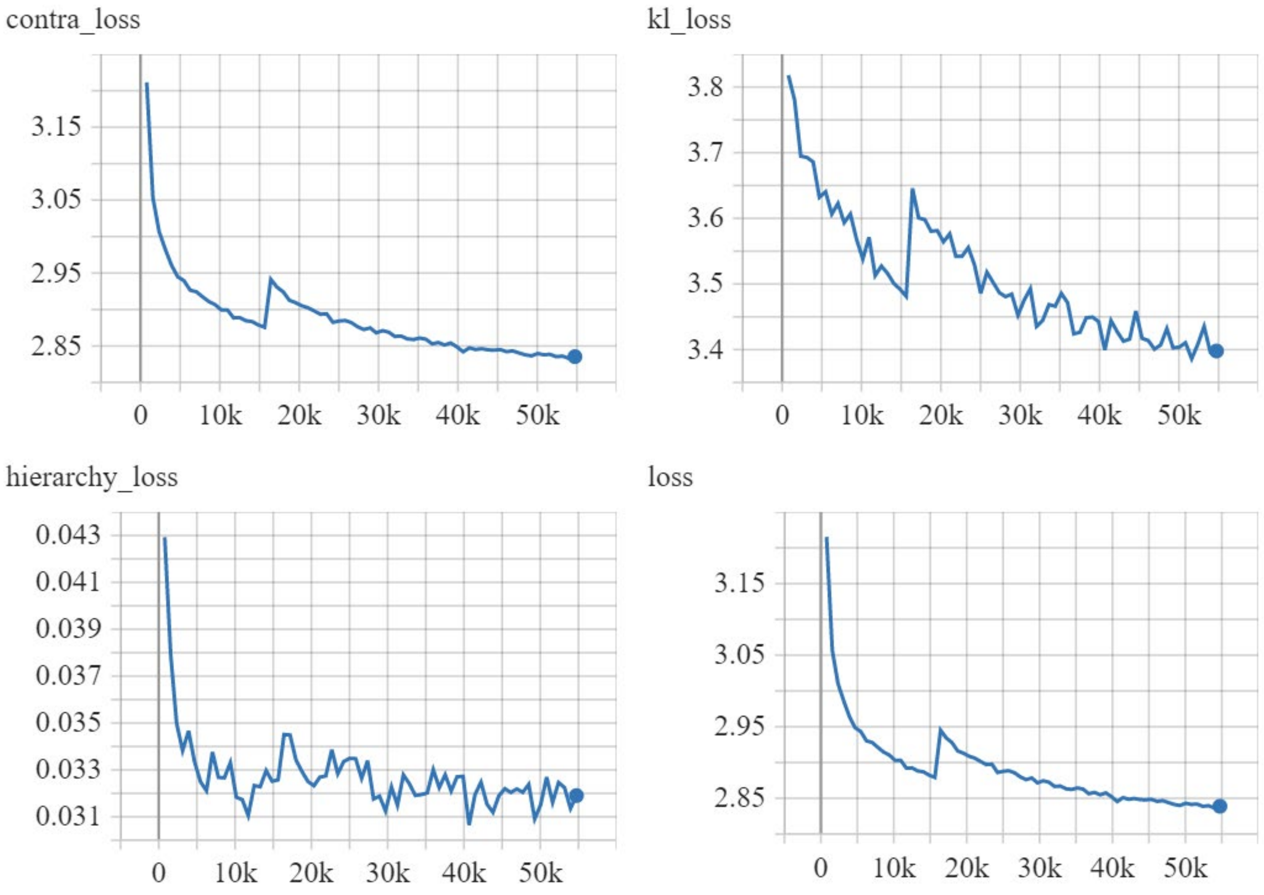


Figure 13 – Learning curves. Contra and KL loss functions were introduced in CIBHash, Hierarchy is the difference of decimal numbers introduced in this work. The shift around 15k is caused by the restart

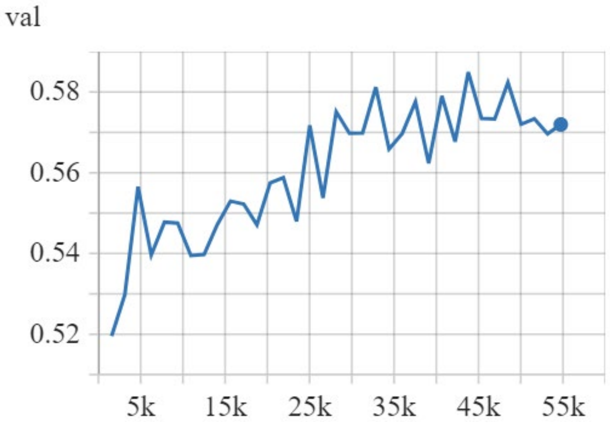


Figure 14 – Validation curve: mAP @1000

SECTION 3. VAN EMDE BOARS TREES

3.1. Description of vEB trees

Let us say we can get a number for each item in the dataset. There is no guarantee that these numbers will not be the same for different items or guarantees that at least one item will match each number. That is, we have a large boolean mask ~ a sparse array. Searching for neighbors in it is not as easy as in a completely filled array, because the greedy search can take O(n). To avoid this, we can use van Emde Boas trees [44]. They are capable of doing addition and neighbor search in O(log(log(u))), where u ($u$niverse) is the maximum number of elements. In our case, the length of the code returned by neurohashing is constant, and therefore the addition and search operations in the tree are also constant. Moreover, the constant is very small.

Unfortunately, decreasing time complexity inevitably leads to increasing memory complexity. An analogy can be drawn between these trees and counting sort. In cases where speed is in the first place, memory is not paid attention to.

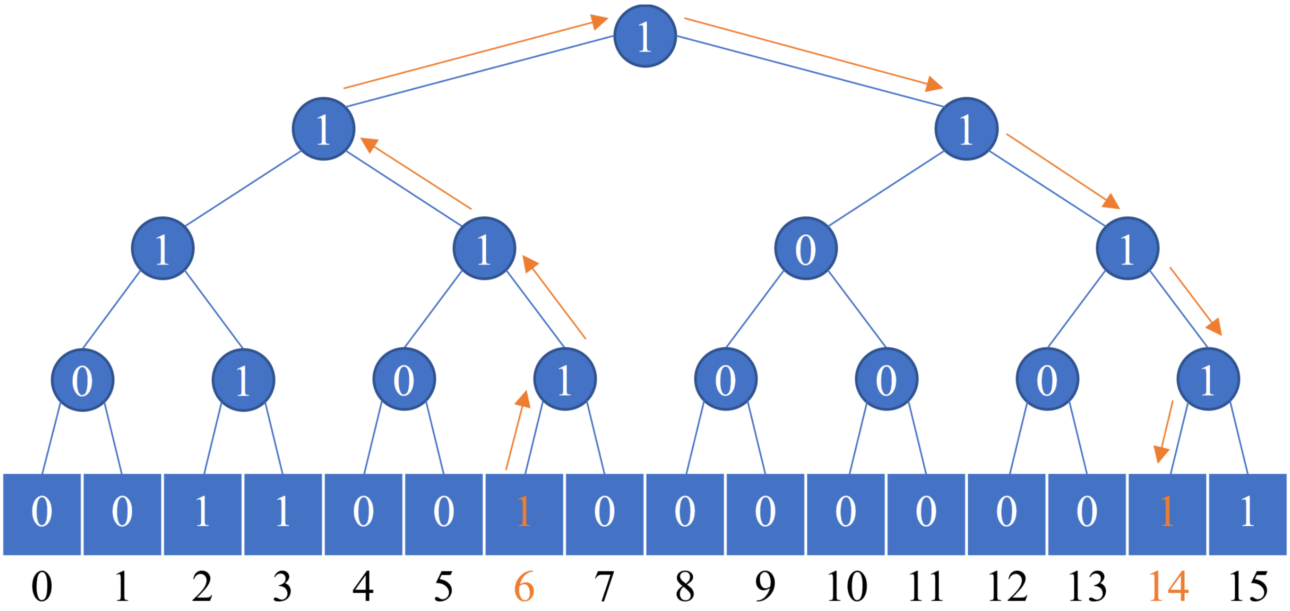


Figure 15 – Prototype of the vEB tree containing elements 2, 3, 6, 14, 15. The search for the successor of 6 is highlighted in orange. It is element 14

In fact, trees are more complex, because they have more than two branches at each node to save memory. They are very difficult to implement because operations on them have a non-trivial recursive nature. Also, vEB trees are not parallelizable since the search for the next neighbor depends on the result of the previous search.

To reduce the query and addition time constants, it is advisable to supplement the vEB tree with a doubly linked list and the following mapping (C++ map, Python dict): {element_number: pointer_to_list_node}.

Finally, we have the following procedure for adding:

1. We find the element number *N* using neurohashing
2. Using the tree, we determine the predecessor and successor number *N*
3. With the help of mapping, we find predecessor and successor nodes of *N* in the list
4. We replace them with the corresponding pointers to node *N*

When searching, the procedure is the same, but in step 4, we need to run two greedy searches for *k*/2 neighbors for the previous and next node of the list.

3.2. Testing of vEB trees

As mentioned earlier, trees are difficult to implement. Their memory and speed performance varies from implementation to implementation. To understand the limits of their performance, tests of various implementations found on the Internet were conducted. Some of them (opengenus, github/Petar) turned out to be inoperable. github/Julian is written in Python, all others in C++. Externally, github/Julian has very high-quality code and test coverage, but its results are worse than any of the C++ implementations.

The speed of addition and search has intervals because it depends on the number of elements in the tree. Two quantities were taken for testing: the minimum possible 32'000 and the maximum possible 2^24. The test code is given in Appendix A. The obtained results confirm that with the help of vEB trees it is possible to query a large number of neighbors in real time.

Implementation	Memory used depth=24, GB	Addition speed k=32'000, 1/sec ~ Hz	Search speed k =32'000, 1/sec
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]

Table 4 – Measurements of different vEB trees. In terms of memory, the dragoun implementation is optimal, in terms of speed and memory – geeksforgeeks

Method	Element size	Addition k= 32'000, sec	Query k= 32'000, sec	Memory usage |db|=5.8e9, GB
Naive: 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 boolean mask (argmin(cumsum(·)))	1 b8	1/350	18.6	5.8
Python vEB tree (github/Julian)	–	1.09	0.35	6'978
C++ boolean mask (greedy [max, min] fill)	1 b8	1/1'000	[2.6e-4; 4]	5.8
C++ vEB tree (github/dragoun) + Mapping, doubly linked list + 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

Table 5 – Measurements of different *k*-NN approaches.
Unacceptable indicators are marked in red, mediocre ones in orange.
768 f32 – vector [768] of 32 bits float, 32 b8 – vector [32] of 8 bits boolean etc.

The transition between Faiss and the proposed doubly-linked list approach is interesting. An increase in speed by ~10 times leads to an increase in memory costs by ~10 times. It may seem that ~900 GB of RAM is an unaffordable amount, but at the moment Google has it Cloud has virtual machines up to 11776 GB. In our case, a VM with 976 GB costs from $6 per hour[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

CONCLUSIONS

A neural network approach was proposed that bypasses the existing methods of finding neighbors in terms of speed. This can reduce the output time for a highly loaded question answering system or enable real-time visual memory to be used by the agent during reinforcement learning. Because the best search libraries spend at least half a second for one query at *k*=32'000 on The Pile. But this dataset is far from the largest available. If visual data will continuously flow with the frequency of the human eye from the virtual environment during reinforcement learning, then the speed of none of the libraries will be sufficient to search in such a volume of data.

Despite the high speed, the search accuracy remains quite low: 0.59 against Faiss' 0.89. Most likely, this is due to the fact that all neurohashing is reduced to training two linear layers. It makes sense to increase the number of layers and replace them with Gated MLP [45] because such layers show results similar to the attention mechanism while remaining simple in structure and fast in execution.

The image-specific neurohashing method WCH shows comparable accuracy to Faiss. This suggests that multimodal methods also have such potential.

REFERENCES

1. Vaswani A., Shazeer NM, Parmar N., Uszkoreit J., Jones L., Gomez AN, Kaiser L., and 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., and 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., and Chang MW, "REALM: Retrieval-Augmented Language Model Pre-Training", // ArXiv, Vol. abs/2002.08909, 2020.
4. Karpukhin V., Oğuz B., Min S., Lewis P., Wu LY, Edunov S., Chen D., and Yih WT 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 WT., Rocktäschel T., Riedel S., and 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 JB, Damoc B., Clark A., et al. Improving language models by retrieving from trillions of tokens // International Conference on Machine Learning. 2021.
8. Fajcik M., Docekal M., Ondrej K., and 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 JA, Joulin A., Riedel S., and Grave E., "Few-shot Learning with Retrieval Augmented Language Models", // ArXiv, Vol. abs/2208.03299, 2022.
10. Wu Y., Rabe MN, Hutchins DS, and Szegedy C., "Memorizing Transformers", // ArXiv, Vol. abs/2203.08913, 2022.
11. Bertsch A., Alon U., Neubig G., and Gormley MR, "Unlimiformer: Long-Range Transformers with Unlimited Length Input", // ArXiv, Vol. abs/2305.01625, 2023.
12. Gionis A., Indyk P., and Motwani R. Similarity Search in High Dimensions via Hashing // Very Large Data Bases Conference. 1999.
13. Johnson J., Douze M., and Jégou H., "Billion-Scale Similarity Search with GPUs", // IEEE Transactions on Big Data, Vol. 7, 2017. p. 535-547.
14. Guo R., Sun P., Lindgren E., Geng Q., Simcha D., Chern F., and Kumar S. Accelerating Large-Scale Inference with Anisotropic Vector Quantization // International Conference on Machine Learning. 2020.
15. Jégou H., Douze M., and Schmid C., "Product Quantization for Nearest Neighbor Search", // IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 33, 2011. p. 117-128.
16. Ge T., He K., Ke Q., and Sun J., "Optimized Product Quantization", // IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 36, 2014. p. 744-755.
17. Malkov YA, Yashunin DA, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs", // IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 42, 2016. p. 824-836.
18. Bulatov A., Kuratov Y., and Burtsev MS, "Scaling Transformer to 1M tokens and beyond with RMT", // ArXiv, Vol. abs/2304.11062, 2023.
19. Gao L., Biderman SR., Black S., Golding L., Hoppe T., Foster C., Phang J., He H., Thite A., Nabeshima N., Presser S., and 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 MA, Lacroix T., Rozière B., Goyal N., Hambro E., Azhar F., et al., "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 TS, Tang J., Hong R., Li H., Luo Z., and Zheng Y. NUS-WIDE: a real-world web image database from the National University of Singapore // ACM International Conference on Image and Video Retrieval. 2009.
23. Lin TY, Maire M., Belongie SJ, Hays J., Perona P., Ramanan D., Dollár P., and Zitnick CL 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., and Shao L., "Auto-Encoding Twin-Bottleneck Hashing", // 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020. p. 2815-2824.
25. Shen Y., Yu J., Zhang H., Torr PHS, and Wang M. Learning to Hash Naturally Sorts // International Joint Conference on Artificial Intelligence. 2022.
26. Yu J., Qiu H., Chen D., and Zhang H. Weighted Contrastive Hashing // Asian Conference on Computer Vision. 2022.
27. Cao Z., Long M., Wang J., and Yu PS, "HashNet: Deep Learning to Hash by Continuation", // 2017 IEEE International Conference on Computer Vision (ICCV), 2017. p. 5609-5618.
28. Chen T., Kornblith S., Norouzi M., and Hinton GE, "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 LJ, Li K., and Fei-Fei L., "ImageNet: A large-scale hierarchical image database", // 2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009. p. 248-255.

31. Oquab M., Darcet T., Moutakanni T., Vo HQ, Szafraniec M., Khalidov V., Fernandez P., Haziza D., Massa F., El-Nouby A., et al., "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., and 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 MG, LeCun Y., and 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., and Joulin A., "Emerging Properties in Self-Supervised Vision Transformers", // 2021 IEEE/CVF International Conference on Computer Vision (ICCV), 2021. p. 9630-9640.

35. Bardes A., Ponce J., and LeCun Y., "VICReg: Variance-Invariance-Covariance Regularization for Self-Supervised Learning", // ArXiv, Vol. abs/2105.04906, 2021.

36. Ermolov A., Siarohin A., Sangineto E., and Sebe N. Whitening for Self-Supervised Representation Learning // International Conference on Machine Learning. 2020.

37. Zbontar J., Jing L., Misra I., LeCun Y., and Deny S. Barlow Twins: Self-Supervised Learning via Redundancy Reduction // International Conference on Machine Learning. 2021.

38. Grill JB, Strub F., Altch'e F., Tallec C., Richemond PH, Buchatskaya E., Doersch C., Pires B.Á., Guo ZD, Azar MG, et al., "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., and Hua X., "A Survey on Deep Hashing Methods", // ACM Transactions on Knowledge Discovery from Data, Vol. 17, 2020. p. 1-50.

40. Qiu Z., Su Q., Ou Z., Yu J., and Chen C. Unsupervised Hashing with Contrastive Information Bottleneck // International Joint Conference on Artificial Intelligence. 2021.

41. Ng K., Zhu X., Hoe JT, Chan CS, Zhang T., Song YZ, and 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 OK., Singhal S., Som S., and 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. p. 75-84.

45. Liu H., Dai Z., So DR, and Le QV Pay Attention to MLPs // Neural Information Processing Systems. 2021.

APPENDIX A. TREE TESTING CODE

```
#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;
}
```