Learned Compression for Vector Databases

John A. Wright* Sloke Shrestha* Vibhor Srivastava*
University of Texas at Austin
{johnawright, sloke, vibhor.srivastava}@utexas.edu

Abstract

Recent advancements in representation learning have made vector database management systems (VDBMS) feasible. VDBMS are used to store, search, and retrieve various ML-derived vector representations of data, which allows for quick and accurate search and retrievals of data based on their vector distance or similarity. Because embedding vectors can occupy a fairly large space in memory, operating in an uncompressed space can result in extreme memory and bandwidth requirements. To ease these tight requirements, practitioners have been using several compression and indexing methods like product quantization (PQ) [5], vector quantization (VQ), multi-index hashing, Hierarchical Navigable Small Worlds (HNSW) [8], and more [11]. However, with progress in learned neural compression [12], this presents a promising alternative in the form of learning a custom compression algorithm as part of the database's normal operations [14]. In this paper, we explore the Vector Quantized Variational Autoencoder (VQVAE) model as well as a learned compression network with an entropy bottleneck as alternatives for compressing the embeddings. We benchmark popular existing compression and indexing methods, PQ and HNSW, on the SIFT1M dataset and observe how our VQVAE and learned compression model compare to the existing state-of-the-art solutions using our curated list of metrics. We show in this paper that our learned compression model increases the accuracy and speed of indexing at the cost of a bit of compression, however, there is much that can be done in future work to further improve the compression. This paper serves as a good starting point for exploring learned compression in the context of model embedding compression for vector databases.

1. Introduction

Vector databases are becoming increasingly popular these days with the advances in machine learning, particularly in the field of Natural Language Processing (NLP). With new applications of these models emerging like deep learning recommendation model (DLRM) [10] and retrieval augmented generation (RAG) [7], serving these applications is going to need more optimization. These applications typically require a large amount of vectors to learn the semantic relationships and similarities between other vectors. For example, Meta proposed DLRM [10], equipped with billions of embedding vectors that can take 96 terabytes of memory to serve. The huge amount of vectors needed to deploy these kinds of applications poses several storage, accuracy, and throughput challenges and trade-offs. Another popular application for vector databases is similarity search, which searches the database for a given vector's nearest neighbors. This has a plethora of use cases like image retrieval and recommendation-based searches.

To make performance optimizations for storage and throughput, several researchers have come up a variety of vector compression and indexing algorithms. In this work, we focus on two state-of-the-art compression and indexing algorithms which are widely used in production: product quantization (PQ) and Hierarchical Navigable Small World (HNSW). We benchmark these existing algorithms using our curated list of metrics against the SIFT1M dataset, which is a list of image embeddings used in similarity search evaluation. Then, we explore using Vector Quantized Variational AutoEncoder (VQVAE) [13] and end-to-end learned compressible features [12], to compress these vectors as an alternative.

Our goal with this project is to determine if learned compression models are viable for vector databases. In this paper, we make the following contributions:

- An exploration of the current state of learned compression models for vector database use cases and current solutions for this application
- We propose a neural network for learned compression that can achieve comparable accuracy and speed with state-of-the-art
- We also define a set of benchmarks and metrics to be used when comparing solutions

*Equal contribution

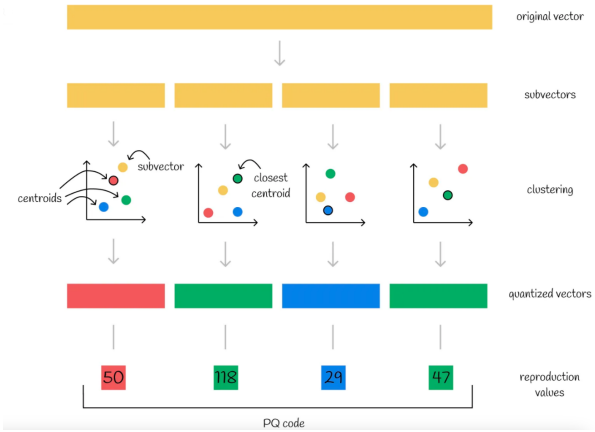

Figure 1. Product Quantization pipeline [2]

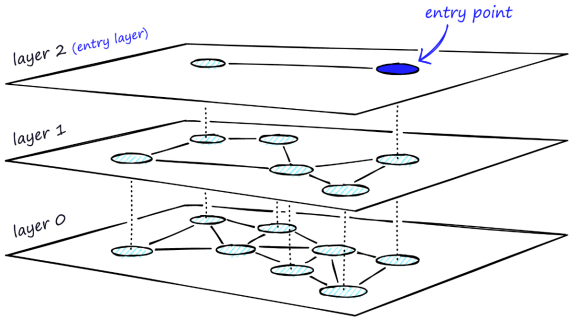

Figure 2. HNSW Example [1]

2. Background

In this section, we describe some of the baseline compression and indexing algorithms that are standard in the industry: PQ and HNSW. We'll also cover the basics of the VQVAE and Learned Compressible Features architectures.

2.1. Product Quantization

Product quantization (PQ) is a compression technique that is employed in vector databases [5]. At a high level, in product quantization, the vectors are divided into different sub-vectors. Each of these sub-vectors occupies a certain subspace in the vector space. These sub-vectors form different clusters. Each of these sub-vectors is then assigned to a cluster centroid which is encoded with the ID of the centroid, and the coordinates of the centroid are saved for later use. The number of created centroids is typically powers of 2 for efficient memory usage.

There are some trade-offs that exist between accuracy and memory requirements which are determined by choosing the number of clusters (n_c) and number of splits for each vector (M) which are hyper-parameters you can select. Specifically, as we decrease the n_c, we increase the quantization error. As we decrease M, the codebook size decreases, which eases up the memory requirement. However, the quantization would be done in higher dimensions, which can suffer from the curse of dimensionality, and lead to higher quantization errors. Section **??** shows some benchmarks performed on PQ.

2.2. Hierarchical Navigable Small Worlds

Hierarchical Navigable Small Worlds (HNSW) is a popular hierarchical graph structure method to approximate the data space. Navigating this graph structure can give the approximate nearest neighbors (ANN) efficiently with low computation cost and high speed (fig 2) [8]. There is a big trade-off with memory as this method must store not only the vectors, but copies of vectors, layers, links, and entry points to the graph.

Using various probability functions based on the data space, the vectors are split and replicated among several layers. On the top layers, there are very few vectors with long edges between them. Conversely, the lower layers have more vectors and shorter edges, with the bottom layer having all of the vectors in the space. The long edges at the top give speed during traversal to minimize the search space. The short edges at the bottom are to refine and get an accurate search. It is important to note that the layer below contains all of the vectors from the layer above it as well. This is why the bottom layer contains all of the vectors.

Essentially, the user enters the top layer at a starting vector and greedily moves along the list of neighbors and finds the nearest vector to the query vector on the layer. Once a local minima is hit (cannot get any closer on the current layer), the user moves down to the same vertex on the lower layer. This process is repeated until layer 0, where the nearest neighbor to the query on this layer is finally returned. Each vector has a neighbor of itself on the layer below it so that the search on the next layer can start from the best point from the top layer. Having more links and neighbors allows for the algorithm to not terminate prematurely. However, while increased links give a more accurate search, they also increase network complexity and search time.

2.3. Vector Quantized Variational Autoencoder

Vector Quantized Variational Auto-encoder (VQVAE) is a type of variational auto-encoder where the latent space is discretized [13]. The discrete space is achieved through the integration of vector quantization, where latent variables are mapped to a learned codebook. Figure 3 shows the pipeline for a VQVAE.

Because of the design of VQVAE, it is intuitive to imagine a compressor for vector databases. The VQVAE is trained on a dataset to reconstruct the input embedding. All the embeddings, before storing them in the database, are run through the encoder and the quantizer. In the case of a sim-

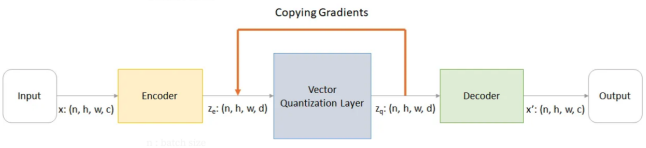

Figure 3. VQVAE architecture [3]

ilarity search given a query vector, the query vector is first run through the encoder and the quantizer. Hopefully, the quantized vectors, which are of shorter length, will have enough semantic properties like the uncompressed embeddings. Then, approximate nearest neighbors (ANN) algorithms would be used to find the top k similar vectors with respect to the query vector.

2.3.1 Finite Scale Quantization

Finite Scale Quantization (FSQ) [9] aims to vastly simplify the way vector quantization is done for generative modeling, removing the need for commitment losses, EMA updating of the codebook, as well as tackling the issues with codebook collapse or insufficient utilization. They simply round each scalar into discrete levels with straight-through gradients; the codes become uniform points in a hypercube.

The VAE representation is projected down to a few dimensions (typically less than 10). Each dimension is quantized to a small set of fixed values, leading to an (implicit) codebook given by the product of these sets. However, as we will mention later, going from 128 dimensions in the SIFT1M dataset to the order of 10 is a tall task, and increasing the dimensions of the projected representation blows up the memory footprint of the implicit codebook. This is the reason why we pivot to training a network which learns compressible features which is described in 2.4.

2.4. Learned Compressible Features

Singh et al. in their paper, "End-to-End Learning of Compressible Features" described and end-to-end network to learn compressible features [12]. This work introduces entropy bottleneck that maximizes the entropy of the latent representation, which in turn helps the latent representation to be compressed using entropy coding techniques.

Figure 4 shows the generic architecture of the network used for learning a compressible latent representation.

$$\theta^*, \phi^* = \arg\min_{\theta,\phi} \sum_{x \in D} L(\hat{x}, x) + \lambda. - \log_2(\hat{z}; \phi) \quad (1)$$

Equation 1 shows the optimization formulation. The second term maximizes the entropy of the latent representation. The function $L(\hat{x}, x)$ is the reconstruction loss between the input, x and $\hat{x}$. we use mean squared error (MSE) for the

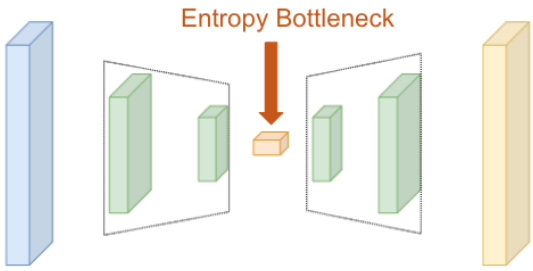

Figure 4. Learning of Compressible Features

reconstruction loss. The hyperparameter, λ weights optimization of compressibility vs. reconstruction loss.

2.4.1 Generalized Divisive Normalization Layers

Johannes Balle et al. [6] introduced a parametric nonlinear transformation that is well-suited for gaussianizing data from natural images. We use the GDN layers after convolution layers in the hopes of making latent representation sparser.

3. Methodology

In this section, we'll discuss the methodology for our project, starting with the dataset and the benchmarking metrics we chose to compare the solutions with. Following that will be the architecture and training results of our learned compression models.

3.1. SIFT1M Dataset

The dataset we chose for the project is the SIFT1M Dataset [5], which is a set of image embeddings used for nearest neighbor search evaluation. We chose this dataset because a common use case for vector databases is finding a given embedding's neighbors for recommendations or other uses. For example, think about Spotify's algorithm. They have audio saved as vectors, and in order to give you recommendations, they'll take an input vector from a song, and find it's nearest neighbors.

This dataset allows us to have a standard set of embeddings with pre-calculated nearest neighbors to perform our evaluation on. The test/train split is 1 million/10 thousand, and the format of the dataset can be seen in Figure 5. Each entry in the list has a tensor which holds the values for the embedding. The test split entries also hold a list of 100 nearest neighbors and their distances, which is what is used to evaluate accuracy.

3.2. Benchmarks Metrics

We chose four benchmarking metrics to evaluate the solutions. These cover the majority of uses that similarity search and vector compression would have and encompass what

```
FeaturesDict({
    'embedding': Tensor(shape=(128,), dtype=float32),
    'index': Scalar(shape=(), dtype=int64),
    'neighbors': Sequence({
        'distance': Scalar(shape=(), dtype=float32),
        'index': Scalar(shape=(), dtype=int64),
    }),
})
```

Figure 5. Format of the Sift1M Dataset [5]

people would be looking at to compare. The first is Recall@1, which is if the correct nearest neighbor is returned as the nearest neighbor in the search result. This is counted up and shown as a percentage, which represents that some number of embeddings out of 10 thousand return the correct nearest neighbor. The next metric is a custom metric that we are calling 'Relevancy'. This is what percent of the K neighbor search results are in the top T correct nearest neighbors. In other words, Relevancy captures the percentage of search results that are correct, with respect to the top amount of correct nearest neighbors. We chose 20 as T since this strikes a good balance between the size of the dataset while maintaining a competitive number to compare results against. If T is too large, then results will all be similar due to every solution performing well due to the wide margin for error. Whereas if T is too small, all solutions may perform badly, with little comparison benefit once again. Relevancy is displayed as a percent, which is the average of the 10 thousand individual Relevancy percents calculated for each test split embedding.

Our last two benchmarking metrics are more related to performance efficiency. Firstly, we evaluate search speed between the solutions, which is measured in milliseconds. This is performed using the 'time' Python module, and we start and stop the timer after the line of code executing the search. The final number is calculated from the average of a thousand iterations. For our results below, an amount of ten results is used, since this is a good middle ground between the possible search amounts. The last metric we chose is the compression ratio. This is simply how much space is saved given the compression algorithm and is computed by dividing the uncompressed size by the compressed size.

3.3. Hardware Resources

All the benchmark testing and model training was done on RTX Titan XP and similarity search evaluations were done with on i9 14900k and RTX 4090.

3.4. Models

We tried training a VQVAE and network learned compressible features. This section describes different iterations of training we performed. The models were trained with a batch size of 32 and for 40 epochs.

3.4.1 VQVAE with FSQ

We first tried training a VQVAE with FSQ. The network had a standard encoder and decoder architecture. Where the encoder was made up of convolutional layers with ReLU activation. At the end, the features were averaged spatial-wise. The decoder was made up of convolution transpose layers instead of convolutional layers with everything else remaining the same. The bottleneck was of size 6 with the levels : [8, 8, 8, 5, 5, 5]. Figure 6 shows the log loss curve for training the VQVAE with FSQ. Unfortunately, our implementation of this model did not converge, so we moved on to explore a learned compression network with an entropy bottleneck.

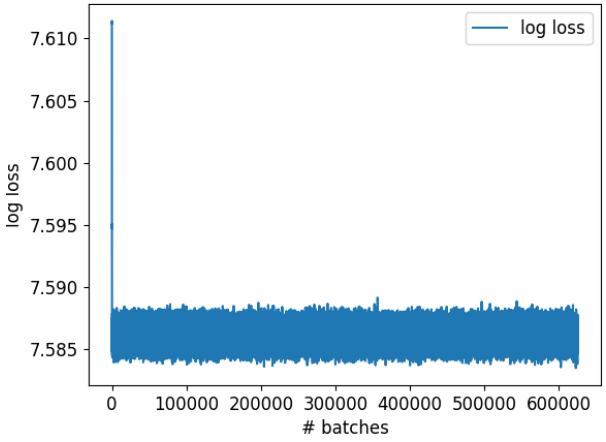

Figure 6. Loss curve for training VQVAE with FSQ.

3.4.2 Network to learn compressible representation: Only ReLU activation

We tried training a network with entropy bottleneck as described in section 2.4. The encoder and decoder had the same architecture as the VQVAE with FSQ. The only difference was that the bottleneck was optimized for higher entropy instead of implementing quantized representations with FSQ. The model was trained with a lambda parameter of 1. Figure 7 shows the loss curve for the network trained to learn compressible representation. The figure shows a decent convergence of the model. This is the model we ultimately use for benchmarking purposes.

3.4.3 Network to learn compressible representation: ReLU + batch normalization

We added some batch normalization layers after convolutional layers on the encoder to hopefully learn sparser representations. We also got rid of the spatial-wise averaging and used MLP layers instead to learn the latent representations.

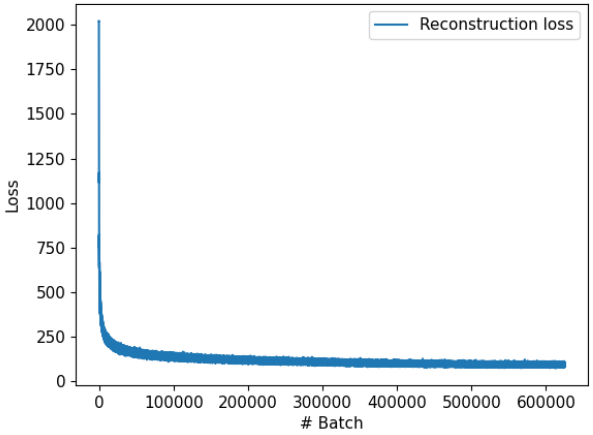

Figure 7. Loss curve for compressible representation: only ReLU activation. The blue curve represents the MSE loss.

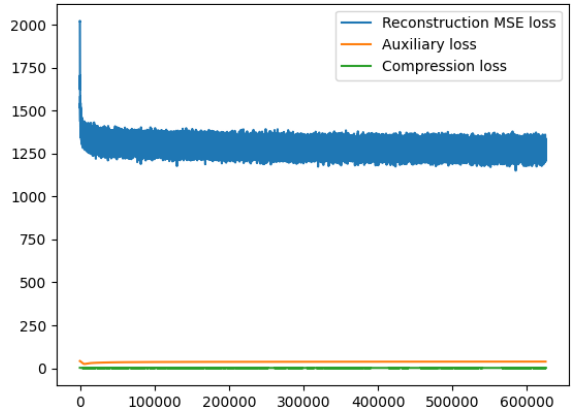

Figure 8. Loss curve for compressible representation: Relu + batch normalization

3.4.4 Network to learn compressible representation: ReLU + GDN activations

We added some GDN (introduced in section 2.4.1) layers after convolutional layers on the encoder to hopefully learn sparser representations. We also got rid of the spatial-wise averaging and used MLP layers instead to learn the latent representations.

Looking at figures 7, 9, and 9, we tried benchmarking the compressed representation obtained by the network with GDN activations explained on section 3.4.4. However, it was only able to score a relevancy score of 20%.

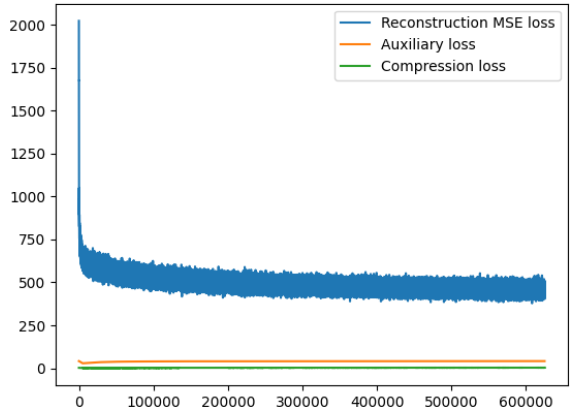

Figure 9. Loss curve for compressible representation: Relu + GDN activations

4. Evaluation and Results

In order to gather the benchmark results evaluated using our metrics discussed in 3.2, we implemented and ran each of the benchmark algorithms using Facebook's AI-powered Similarity Search (faiss) [4]. The inferences were run on the CPU since we opted to use the CPU version of the faiss library. This is because the GPU version was not up to date, and we wanted to ensure that both test environments were the same. As for our learned compression model we ended up with from section 3.4.2, we train the model on the GPU and inference using the same CPU to ensure a level playing field. We'll start by discussing the benchmark results first to show the current landscape of solutions, and then we'll follow up with the results from our learned compression.

4.1. Benchmark Results

To start off the results of our benchmarks, we'll take a look at flat indexing, which you can think of as a lookup table. This acts as our baseline for comparing everything against. As a quick note, recall@1 is shown in the following four graphs, each corresponding to an algorithm, beginning with Figure 10 for Flat Indexing. The other three metrics are shown in Table 1. As shown in Figure 10, the Recall@1 for the flat index is 100%, which makes sense as it's just a simple lookup. Relevancy is also 100% for the same reason. What we wanted to highlight here is that the search speed and compression ratio are both very poor, but these serve as the baseline for comparison and computing metrics such as compression ratio. Speaking of which, the uncompressed size of the dataset indexing is 512 MB.

Moving onto Product Quantization (PQ), Figure 11 shows the recall@1 results for PQ with a subvector length of 8. PQ sees a massive improvement in compression and speed

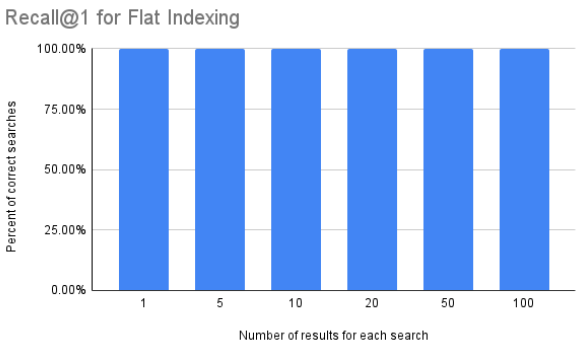

Figure 10. Recall@1 for Flat Indexing.

Algorithm	Relevancy	Speed	Compression
Flat Index	100%	184 ms	1.00x
PQ with 8Sv	58.5%	24.8 ms	63.2x
PQ with 16Sv	88.4%	33.7 ms	32.8x
HNSW + PQ8	68.4%	1.35 ms	11.8x

Table 1. Relevancy, Search Speed (Speed), and Compression Ratio (Compression) results for the four benchmarks tested.

shown in Table 1, but at the cost of search result accuracy, as demonstrated in the decrease in recall@1 and relevancy. PQ with a subvector length of 16 however, improves the accuracy quite a bit at the cost of search speed and compression, as shown in Figure 12 and Table 1. This makes sense as a longer subvector length would increase the dimensionality of the space, increasing search time as well as making the index use more memory. This does make the indexing more accurate though as the search is more fine-grained. The key takeaway here is that we now have a good understanding of what modern compression looks like, and the expected accuracy and search speed with varying degrees of that compression.

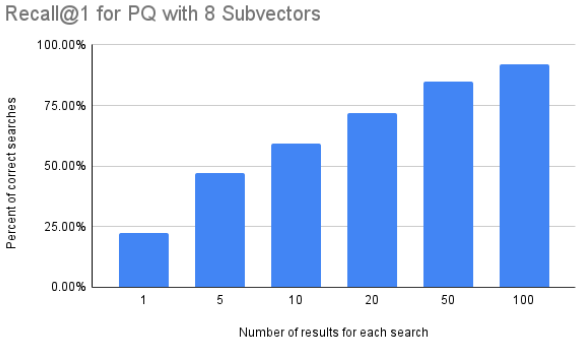

Figure 11. Recall@1 for Product Quantization with subvector length of 8.

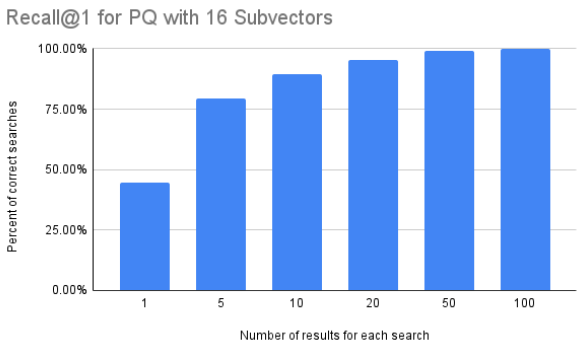

Figure 12. Recall@1 for Product Quantization with subvector length of 16.

Lastly, we also implemented and ran a composite index of HNSW plus PQ, where HNSW was used to build a graph of the PQ-encoded vectors. This combines the fast search speed of HNSW with the compression of PQ. The faiss index factory was used to combine them, which made the implementation straightforward. Figure 13 shows the recall@1 results, which plateau as the number of search results increases. This is likely because we ran an unoptimized HNSW with no hyperparameter training. This isn't a problem though as we mainly want to see what the upper bound for search speed looks like when indexing as HNSW is the current state-of-the-art, and tuning would not affect speed to our knowledge. In addition, just using the out-of-the-box HNSW makes for a fairer comparison for the next section. The search speed increase is quite substantial though, as seen in Table 1. However, this does reduce compression since HNSW uses a lot of memory to store the graphs.

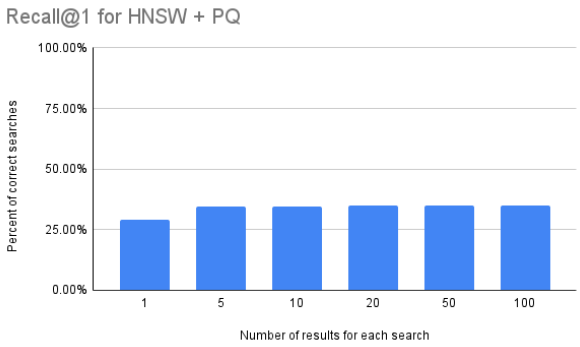

Figure 13. Recall@1 for Hierarchical Navigable Small Worlds indexing with Product Quantization compression.

4.2. Learned Compression Results

Now that we've covered the benchmark results, we can discuss how our learned compression model fits into that land-

scape. The actual compression is done through our model, and then we index into the resulting latent representation with HNSW. We again used the unoptimized, out-of-the-box HNSW from faiss to try and get a fair comparison between PQ and our model. In the last row of Table 2, you can see how our model did. We were able to achieve a slightly more accurate model with a decent increase in search speed as well when compared to PQ8. The reason for the speed increase is that while PQ and PQ + HNSW versions indexed on a length 128 vector, our model reduced the length down to 64, so HNSW was able to compute the neighbors quicker. In Figure 14, our learned compression was also able to return the correct closest neighbor a significant percent more than PQ8 with HNSW. This is also without optimization, so the difference could even be greater.

Algorithm	Relevancy	Speed	Compression
Flat Index	100%	18.9 ms	1.00x
PQ with 8Sv	58.5%	2.58 ms	63.2x
PQ with 16Sv	88.4%	3.38 ms	32.8x
HNSW + PQ8	68.4%	0.14 ms	11.8x
Learn + HNSW	70.6%	0.10 ms	4x

Table 2. Relevancy, Search Speed (Speed), and Compression Ratio (Compression) results for all algorithms. Learn + HNSW is our learned compression with HNSW indexing.

While we were able to achieve better accuracy and speed, our compression did take a hit. We explored some normalized techniques as discussed earlier that would help in compression, however, we didn't find much luck in the time we spent working on it. Normalization is helpful as tightening the distribution of values of the latent representation can add compression in the form of what data types we are able to use. For example, if we can reduce the range of numbers to be between -128 and 127, we can use 8-bit integers rather than something larger. Out of the three metrics, compression is the easiest to improve though, so we were mainly focused on making sure that accuracy and speed were good, as compression can be improved later.

There were some trade-offs we were trying to balance as well, and there is room for further exploration. Firstly, if you compress a vector more, you may lose some semantic meaning. This is not only bad for trying to reconstruct the vector, but also when performing an algorithm like similarity search, any semantic loss may negatively affect performance in terms of accuracy. Also, we wanted to try and keep our model architecture and layer setup relatively simple so that encode and decode time would be faster. While encoding and decoding aren't on the critical path for similarity search, these are still important times to track as any input to the database will need to be encoded, and if you want to get back the original embedding, you will need to decode.

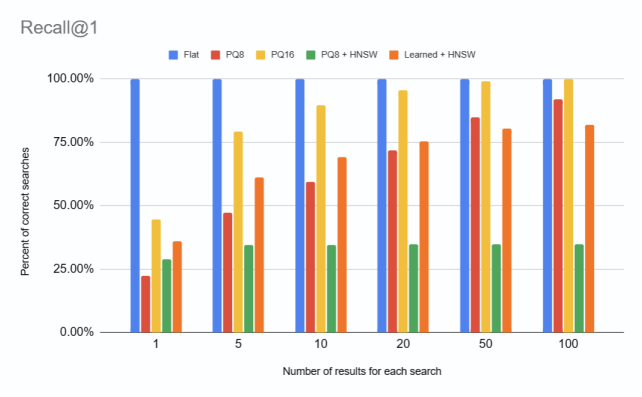

Figure 14. Recall@1 for Hierarchical Navigable Small Worlds indexing with Product Quantization compression.

4.3. GitHub

All of the code developed during this project can be found in this GitHub repository.

5. Future work

5.1. Split vectors

In PQ, we see that the vectors are turned into smaller sub-vectors. Clusters are computed in each sub-vectors. Later, each quantized vectors are concatenated together. In the future we aim to learn compression network for each sub-vectors just like in PQ. This architecture might enable us to use VQVAE with FSQ because we would be using smaller vector sizes.

5.2. Entropy coding

In this work, we did not perform entropy coding after learning compressible representations. However, we plan to perform entropy coding and learn how much compression can be achieved with and without learning compressible representations. This will show if learning compressible representation is really valuable.

5.3. More media types

In our current work, we work with image embedding, namely the SIFT1M dataset. However, we will explore the challenges and complexities associated with working with multiple embedding sources which might come from different encoders and different media types (audio, text, IMU, etc.).

6. Conclusion

With vector databases continuing to advance technologically and increase in popularity due to the explosion of use cases for machine learning, figuring out optimal storage solutions that enable compact compression with great performance is

critical to the industry. In this project, we investigate whether learned compression is viable for these vector database management systems, specifically within the context of similarity search. We also formulate a series of metrics to use for comparison and evaluate existing solutions as benchmarks with those metrics. Our results show that while there is future work to be done, learned compression definitely has the potential to be one of the best-performing solutions when it comes to compressing model embeddings. We were able to achieve better accuracy and speed than the popular algorithm combination of Product Quantization (subvector length 8) with Hierarchical Navigable Small Worlds while taking a small hit to compression. However, with compression being the easiest of the metrics to improve, we believe that there is a definite path to improve the compression ratio of our model and exceed the performance of the previous solutions.

References

[1] Hierarchical navigable small worlds (hnsw). https://www.pinecone.io/learn/series/faiss/hnsw/. Accessed: 2024-03-25.

[2] Similarity search, part 2: Product quantization. https://towardsdatascience.com/similarity-search-product-quantization-b2a1a6397701. Accessed: 2024-03-25.

[3] Understanding vector quantized variational autoencoders (vq-vae). https://shashank7-iitd.medium.com/understanding-vector-quantized-variational-autoencoders-vq-vae-323d710a888a. Accessed: 2024-03-25.

[4] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[5] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[6] Nick Johnston, Elad Eban, Ariel Gordon, and Johannes Ballé. Computationally efficient neural image compression. *arXiv preprint arXiv:1912.08771*, 2019.

[7] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[8] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[9] Fabian Mentzer, David Minnen, Eirikur Agustsson, and Michael Tschannen. Finite scalar quantization: Vq-vae made simple. *arXiv preprint arXiv:2309.15505*, 2023.

[10] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 993–1011, 2022.

[11] James Jie Pan, Jianguo Wang, and Guoliang Li. Survey of vector database management systems. *arXiv preprint arXiv:2310.14021*, 2023.

[12] Saurabh Singh, Sami Abu-El-Haija, Nick Johnston, Johannes Ballé, Abhinav Shrivastava, and George Toderici. End-to-end learning of compressible features. In *2020 IEEE International Conference on Image Processing (ICIP)*, pages 3349–3353. IEEE, 2020.

[13] Aaron Van Den Oord, Oriol Vinyals, et al. Neural discrete representation learning. *Advances in neural information processing systems*, 30, 2017.

[14] Hailin Zhang, Penghao Zhao, Xupeng Miao, Yingxia Shao, Zirui Liu, Tong Yang, and Bin Cui. Experimental analysis of large-scale learnable vector storage compression. *arXiv preprint arXiv:2311.15578*, 2023.