Learning to Emulate Locality Sensitive Hashing for Nearest Neighbor Search: An Empirical Comparison

Matthew Siper[1], **Zehua Jiang**[1]

[1] New York University, New York, USA

Abstract

Abstract In this paper we seek to answer the questions: *can we train neural networks to emulate locality sensitive hashing (LSH) schemes to perform similarity search? How competitive would such a system be to current large scale state-of-the-art implementations such as Google's ScaNN?* To answer these questions we frame similarity search as a hierarchical clustering problem. We introduce a machine learning (ML) system [1] consisting of 2 layers trained on data annotated from fitting a k-means clustering algorithm on word embeddings from the GloVe dataset. This ML system is then compared to ScaNN and a random benchmark across 6 different GloVe datasets and show that our system predicts nearest neighbor vectors closer to the actual nearest neighbor vectors as compared to ScaNN in 4 out of the 6 tests.

Keywords: *ANN, nearest neighbor; machine learning; k-means; clustering; deep learning; similarity search; locality sensitive hashing.*

1. Introduction

The problem of similarity search for objects, such as text or images, has been widely studied. Many enterprise-grade systems implement similarity search via LSH, or some variant, as the underlying algorithm. At its core, LSH is a dimensionality reduction technique that seeks to maximize the likelihood of hash collisions such that similar objects are hashed to the same bucket with high probability. Since these buckets contain similar objects with high probability, they can be viewed as clusters. Specifically, LSH has been shown to be a dominant technique to solve the (R,c)-NN problem [Indyk and Motwani(1998)]. For some domain S of points set with distance measure D, we can define an LSH family as, H:

$$H = \{h : SU\} \tag{1}$$

where H is called (r_1, r_2, p_1, p_2) -sensitive for D if for any v,q ∈ S

1. if v ∈ $B(q,\mathrm{r}_1)$ then $Pr_H[h(q) = h(v)] \geq p_1$,
2. if v B(q, r_2) then $Pr_H[h(q) = h(v)] \leq p_2$.

[1]`https://github.com/matt-quant-heads-io/algorithmic_ml_sandbox`

note that in order for an LSH family to be of use, it must satisfy that $r_1 < r_2$ and $p_1 > p_2$.

A scheme for employing an LSH family to solve the (R,c)-NN problem is formulated as: Choose some $r_1 = R$ and $r_2 = cR$, for some constant c. Then, given a family H of hash functions that is (r_1, r_2, p_1, p_2) -sensitive maximize $p_1 - p_2$ by concatenating several functions [Datar et al.(2004)]. Specifically, for a given family of functions G where $G = g : SU^k$ such that $g(v) = (h_1(v), ...h_k(v))$, where $h_i \in H$, for some value k. Then, for some integer L, choose L functions $g_1, ..., g_L$ from G, independently, and uniformly at random. For each $v \in P$, where P is a set of inputs, store v in the bucket $g_j(v)$, for j = 1,...,L.

An important note here is to retain only the non-empty buckets using a hashing mechanism. To process a given query vector, q, search all buckets in $\{g_1(v), ..., g_L(v)\}$. Terminate the search when mL items have been found, for some value m. Finally, return any vectors, v_j, where $v_j \in B(q, r_2)$. Note that the tunable parameters k and L are chosen to ensure with constant probability that the following properties hold (to ensure correctness),

1. $\sum_{j=1}^{L} |g_j^{-1}(g_j(q)) \cap (P - B(q, r_2))| < mL$, and
2. If $\exists$v$^* \in B(q, r_1) \iff$ g$_j(v^*) = g_j(q)$ for some j = 1, ...,L.

Clustering algorithms, in general, can be used to describe LSH. Viewed in this light, members of a cluster are *similar* to other members of the same cluster consistent with the notion that buckets in an LSH system contain similar items with high probability. This is in fact similar to Google's ScaNN system which employs LSH to perform maximum inner product search in order to identify nearest neighbors relative to a query vector.

2. Similarity Search

2.1. Main task of accelerating MIPS

In recent years, there has been significant research on methods for accelerating the Maximum Inner Product Search (MIPS) problem. As proposed by [Guo et al.(2020)], the main goal of these methods is to efficiently find the vector in a given set that has the maximum inner product with a query vector. There are two main tasks required to develop an efficient MIPS system: reducing the number of items that are scored, and improving the rate at which items are scored. One approach to reducing the number of items that are scored is to use a space partitioning method. This involves dividing the vector space into smaller regions, and only considering vectors that fall within a particular region when searching for the top result.

This can significantly reduce the number of vectors that need to be considered, and can therefore improve the efficiency of the search. Another important task in developing an efficient MIPS system is improving the rate at which items are scored. This is typically done with quantization, which involves representing vectors with a fixed number of bits. By using quantization, it is possible to score multiple vectors simultaneously, which can significantly improve the rate at which items are scored. The main contribution of Scann's work lies in this area, with the development of efficient quantization methods for MIPS. Overall, successful implementation of MIPS systems require good performance in both tasks: reducing the number of items that are scored, and improving the rate at which items are scored.

2.2. ScaNN

In [Guo et al.(2020)], researchers present a new anisotropic loss function termed score-aware quantization loss, that is shown to be more performant than the standard reconstruction error on vector quantization tasks. Quantization techniques often involve minimizing the reconstruction error when a value x is approximated by a (compressed) quantized analogue $\tilde{x}$. It has been shown that minimizing the reconstruction error is equivalent to minimizing the expected inner product quantization error under certain conditions of the query distribution, without any assumptions on the distribution of data points in the database. Score-aware quantization loss weights the approximation error for data points with higher inner product more as these are deemed to be more similar than data points that produce lower inner product values. They argue that this leads to a more applicable objective function since certain (x, q) pairs will naturally be more similar than others. This natural phenomenon enables score-aware quantization to more accurately quantize vectors. It was shown that regardless of the choice of w, a score-aware quantization loss $l(x_i, \tilde{x}_i, w)$ always decomposes into an anisotropic weighted sum of the magnitudes of the parallel and orthogonal residual errors.

While our paper doesn't investigate or perform any empirical studies pertaining to score-aware quantization loss, we are interested in the underlying ScaNN system used in the paper. Specifically, the underlying ScaNN system uses a tree structure to organize nodes such that a node's leaves contain a similar embedding relative to the embeddings of other leaves from the same parent. The tree structure facilitates faster search times when finding the closest node by reducing the number of nodes that need to be checked, such that only T need to be visited where $T = O(log(n) + m)$, where m is the number of leaves to be visited under node t and n is the number of nodes in the tree. We use the ScaNN system as a model to implement our ML pipeline to perform similarity search.

2.3. The Machine Learning System

As previously mentioned, we train an ML system to emulate ScaNN using 2 layers of deep 1-dimensional convolutional neural networks. The system can be theorized as a hierarchical clustering model where the first layer predicts the cluster containing the nearest neighbor (analagous to the parent node containing the nearest neighbor leaf in the ScaNN system), and the second layer predicts the nearest neighbor in the cluster predicted by layer 1. Specifically, layer 1 is trained to map a given query vector (i.e. a 100-dimensional word embedding of floating point values) to the cluster containing the nearest neighbor, while layer 2 is trained to map a query vector and the centroid vector corresponding to the cluster predicted by layer 1, to the nearest neighbor contained in that cluster. The networks are trained using the word embeddings from the GloVe dataset combined with cluster identifier targets and centroid vectors computed from fitting a k-means clustering algorithm on the dataset during preprocessing.

The output of the layer 1 model is an integer identifier that constitutes a cluster id and is also the key in the hash map that corresponds to the layer 2 model trained to predict the nearest neighbor within that specific cluster. It follows then that each layer 2 model is trained on the intra-cluster data points of that specific cluster. Therefore, one layer 2 model is trained for each cluster in layer 1. Each layer 2 model is trained using centroid vector and query vector pairs drawn from the corresponding cluster from layer 1. The output from each layer 2 model is an id corresponding to the nearest neighbor vector. The system is represented in figure 1.

We generate training datasets across 10,000, 20,000, and 50,000 totals words, respectively, from GloVe. For each dataset, we implement layer 2 model hash maps (i.e. maps where the key is the cluster id and the value is the layer 2 model trained on the data within that cluster), consisting of 10 cluster-to-model key-value pairs and 100 cluster-to-model key-value pairs, respectively, for a total of 6 different systems. We measure the performance of each system against its ScaNN counterpart as described in the experiments section.

3. Methods

3.1. Training Data Generation

Since we frame ANN search as a hierarchical clustering problem, to generate the training data, we first fit a k-means clustering algorithm on the data and output an integer cluster id for each data point. This data was used to train the layer 1 model such that it learned to map a given query vector to its corresponding cluster (i.e. the cluster containing the query vector's nearest neighbor). Since the input data for each layer 2 model consists of a given query vector and the centroid vector of the corresponding cluster, the training data used to train the layer 2 models was generated by computing the nearest neighbor for each query vector in the dataset. An id was then assigned to each nearest neighbor vector and this was used as the target in the training step.

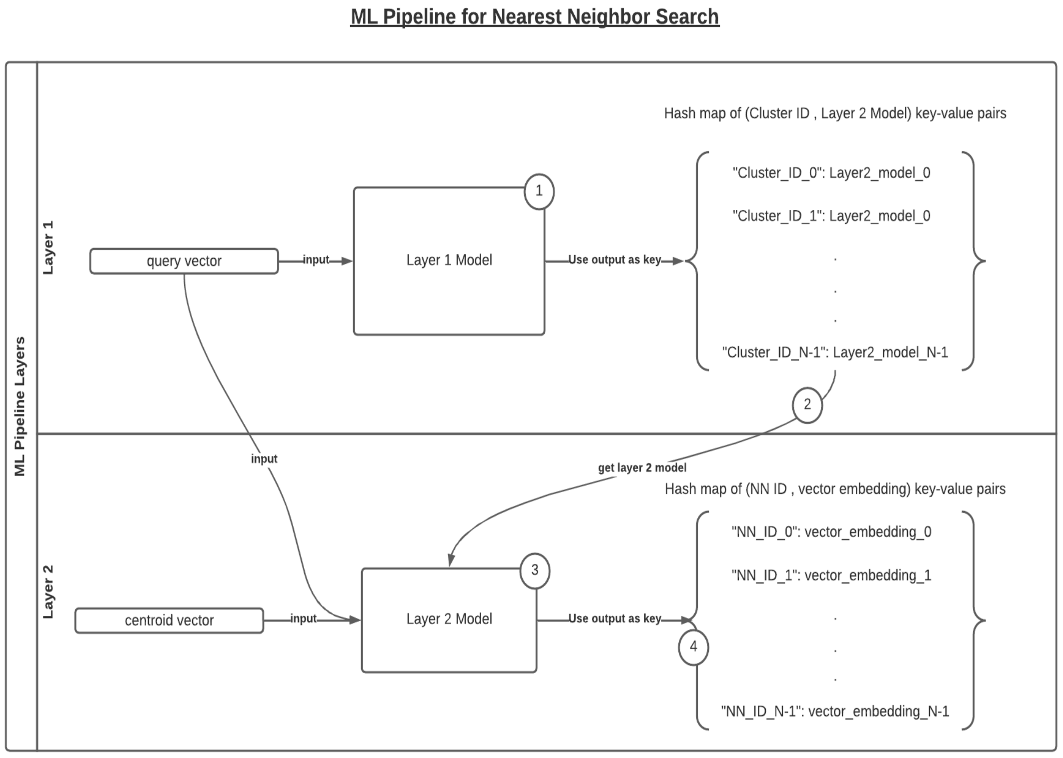
ML Pipeline for Nearest Neighbor Search
ML Pipeline Layers
Layer 1
Layer 2
query vector
input
Layer 1 Model
1
Use output as key
Hash map of (Cluster ID , Layer 2 Model) key-value pairs
"Cluster_ID_0": Layer2_model_0
"Cluster_ID_1": Layer2_model_0
"Cluster_ID_N-1": Layer2_model_N-1
2
get layer 2 model
input
Hash map of (NN ID , vector embedding) key-value pairs
"NN_ID_0": vector_embedding_0
"NN_ID_1": vector_embedding_1
"NN_ID_N-1": vector_embedding_N-1
centroid vector
input
Layer 2 Model
3
Use output as key
4

Figure 1. System diagram explaining the ML Pipeline for nearest neighbor search. Our ML pipeline consists of two layers of deep neural networks that are trained to predict the nearest neighbor vector for a given query vector. The first layer maps the query vector to the closest cluster using word embeddings and cluster identifiers computed via k-means clustering. The second layer predicts the nearest neighbor label within the cluster, using the query vector and the centroid vector of the cluster as inputs. We generate datasets with different sizes and implement layer 2 model hash maps with different numbers of cluster-to-model key-value pairs. These systems are used in the experiments detailed in the following section. The ML pipeline emulates ScaNN in terms of input and output.

Therefore, the layer 2 models learned a mapping of (query vector, centroid vector) pairs to nearest neighbor ids, which correspond to nearest neighbor vectors. We then split each dataset generated to be 90% training and 10% test data. We generated data sets consisting of the following 6 parameter combinations: (10 clusters, 10,000 words), (100 clusters, 10,000 words), (10 clusters, 20,000 words), (100 clusters, 20,000 words), (10 clusters, 50,000 words), and (100 clusters, 50,000 words). These datasets were used to train the models and run the experiments.

3.2. Model Architecture

We used the same model architecture for both layer 1 and layer 2 models. Each model consisted of 3 1-dimensional convolutional layers, each utilizing a ReLu activation function. The first 2 contolutional layers were comprised of 128 neurons. The third convolutional layer consisted of 256 neurons. This layer was followed by a flattening layer which then fed into a densely connected layer to output the probability distribution of targets via a softmax activation function. The model architecture is evident in figure 2 and figure 3.

3.3. Model Training

For each combination of number of clusters and number of tokens in the datasets used, we fit a k-means clustering algorithm on the dataset using the implementation from scikit-learn. We

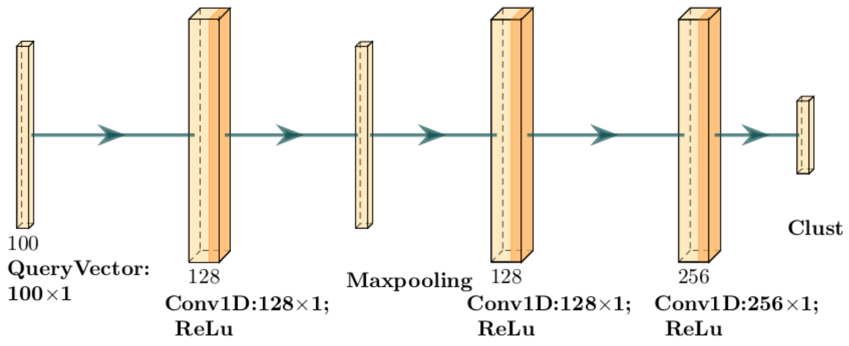

Figure 2. This diagram shows the inter-cluster model architecture (layer 1 model), which takes a query vector as input, which is 100×1. The model consists of a 1D convolutional layer with size 128×1, followed by a maxpooling layer and two additional 1D convolutional layers with sizes 128×1 and 256×1, respectively. The activation function for all convolutional layers is ReLU. After being flattened and passed through a softmax layer, the model produces an output vector of size $n_{cluster}$, which predicts the cluster to which the query belongs.

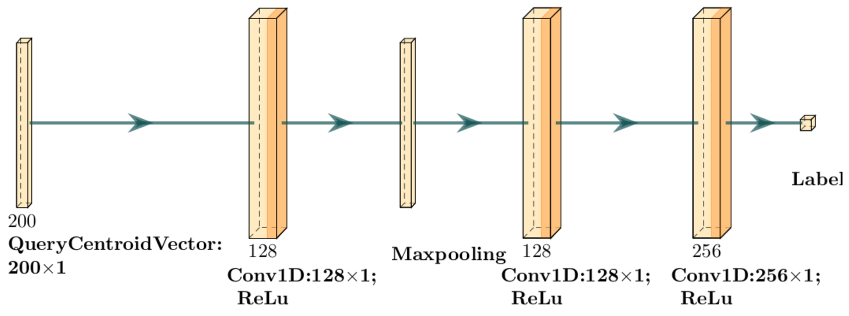

Figure 3. The diagram shows the intra-cluster model architecture (layer 2 model), which is similar to the inter-cluster model, except for the input and output. The input is the concatenated vector of the query vector and the centroid vector produced by the inter-cluster model. After processing the input through a series of 1D convolutional layers, a maxpooling layer, and two additional convolutional layers, the result is flattened and passed through a softmax activation to produce the final prediction, which is a nearest neighbor identifier.

capped the number of times the algorithm recomputed the centroids at 1000 times. Further, each network was optimized with stochastic gradient descent using categorical cross entropy for the objective function. Model training consisted of 1024 steps per epoch for 500 total possible epochs.

We implemented two fitting procedures and applied them during model training. The first was used to save the updated network weights each time a new high watermark prediction accuracy was achieved. The second was an early stopping procedure that terminated model training when the fitting routine failed to achieve a new high watermark accuracy over 3 consecutive epochs. For each combination of number of words and number of clusters used to train a given dataset, 1 layer 1 model was trained and C intra-cluster models were trained where C is the number of clusters used to generate the dataset.

4. Experiments

4.1. Experiment 1: Measuring System Performance via Average Distance

In this experiment, we measured the performance of ScaNN, our ML pipeline, and a random benchmark. We obtained the performance across the 3 models via computing the average distance between the predicted nearest neighbor vector and the actual nearest neighbor vector over 1000 sampled query vectors from the test set. Smaller distance values indicate predicted nearest neighbor vectors that are closer to actual nearest neighbor vectors. For the random model, the predicted nearest neighbor vector was sampled uniformly from the corresponding dataset. The experiments were run across the 6 generated GloVe datasets. The results for each dataset are detailed in table 1.

4.2. Experiment 2: Measuring Classification Accuracy

For this experiment, we sought to better understand the strengths and weaknesses of the ML pipeline by measuring the prediction accuracy in layer 1 and layer 2, respectively. To measure the accuracy of the layer 1 model, the model prediction was compared to the actual cluster label for each of the 1000 sampled query vectors of each test set. The accuracy was computed for each of the 6 generated GloVe datasets. The results of this experiment are shown in table 2. To measure the prediction accuracy for the layer 2 models, we sampled 1000 query vectors from each of the 6 tests. In this experiment, a model prediction was deemed correct if the prediction was in the set of the 5 nearest neighbors for each query vector. Table 3 shows the results for this experiment.

4.3. Experiment 3: Measuring System Prediction Times

In order to better understand any performance trade-offs between ScaNN and the ML pipeline, we measured the average prediction times for both systems across the 6 test sets. To measure the ML system, for each given query vector, we added the prediction time for layer 1 to the prediction time for layer 2. We averaged the prediction times across each model over 1000 query vectors sampled from each of the 6 test sets. The results of this experiment are displayed in table 4.

5. Results

As is evident from the results in table 1 below, the ML system outperformed ScaNN in 4 out of the 6 experiments. The ML system underperformed noticeably in the experiments involving 10 clusters and 10,000 words and 100 clusters and 20,000 words, respectively. This doesn't seem to be related to a scalability issue since the ML system outperformed ScaNN for the datasets of 10 clusters and 50,000 words and 100 clusters and 50,000 words, respectively. A closer analysis of table 3 below in experiment 2 points to the root cause in underperformance being attributed to layer 2. This is evident since the corresponding layer 2 prediction accuracy for these experiments is only 5% and 2%, respectively. Moreover, it is unclear exactly why the layer 2 prediction accuracy is so low since the layer 2 accuracy for datasets 100 clusters and 10,000 words and 10 clusters and 20,000 words, respectively, were 90% and 86%. A closer look at the underlying distributions of

Experiment (clusters, words)	ScaNN	Random	ML
(10,10000)	3.062	7.185	5.723
(100,10000)	**2.975**	**7.034**	**0.540**
(10,20000)	**3.202**	**7.241**	**0.950**
(100,20000)	3.226	7.214	5.89
(10,50000)	**3.346**	**7.540**	**2.001**
(100,50000)	**3.424**	**7.459**	**1.104**

Table 1. Average distance between predicted and actual nearest neighbor vectors

Experiment (clusters, words)	Accuracy
(10,10000)	0.98
(100,10000)	0.97
(10,20000)	0.91
(100,20000)	0.99
(10,50000)	0.87
(100,50000)	0.99

Table 2. Layer 1 accuracy

Experiment (clusters, words)	Accuracy
(10,10000)	0.05
(100,10000)	0.94
(10,20000)	0.86
(100,20000)	0.02
(10,50000)	0.6
(100,50000)	0.76

Table 3. Layer 2 accuracy

cluster membership across the datasets suggest a potential root cause of layer 2 underperformance could be attributed to data imbalance.

The results for the performance of each system in terms of their respective prediction times are displayed in table 4 below. As is evident, the ScaNN system greatly outperforms our ML implementation in terms of raw prediction speed by a factor of about 100. This can largely be attributed to the fact that ScaNN uses C++/Python language bindings such that the actual algorithmic and data structure-related logic is executed in C/C++. We believe this is definitely an area of potential improvement for the ML system. Exploring a C/C++ implementation of our ML system constitutes a necessary step to both more accurately compare both systems and to extend our set of experiments to include larger and more complex datasets.

Experiment (clusters, words)	ScaNN (s)	ML Pipeline (s)
(10,10000)	0.000313	0.080705
(100,10000)	0.000325	0.074219
(10,20000)	0.000307	0.095159
(100,20000)	0.000325	0.070852
(10,50000)	0.000304	0.087592
(100,50000)	0.000323	0.077589

Table 4. Average prediction times of ScaNN and the ML Pipeline

6. Conclusion

In this paper, we explored the fundamental question of whether deep neural networks could learn to emulate a large-scale LSH system for similarity search, such as ScaNN. To address this question, we proposed a machine learning system composed of 2 layers that collectively modelled a hierarchical clustering algorithm. The K-means algorithm was used to generate 6 GloVe datasets, which we used in model training and to conduct the experiments. We showed in 4 out of the 6 experiments that our system outperformed the large scale and state-of-the-art ScaNN system in similarity search where each datapoint consisted of 100-dimensional word embeddings.

Furthermore, we assessed that the underperformance by our ML system in 2 of 6 datasets for experiment 1 was due to underperformance of the respective layer 2 models. We theorized and proposed a data augmentation technique for generating more training exmaples for underperforming layer 2 models. However, due to time constraints we couldn't properly test the usefulness of this technique. Similarly, we intended to conduct an experiment to compare the efficiency and requirements for the ML system and the ScaNN system using the number of floating point operations during inference as a heuristic for space costs; however, we were unable to implement the necessary tools to properly measure this across both systems. In addition to testing our novel data augmentation technique, and conducting an experiment to measure floating point operations across both systems, we see various potentially fruitful paths of future work that are worth exploring.

One such path is to evaluate the performance of the proposed method on other datasets, such as images and documents. This would help to determine the generalizability of the approach, identify limitations of the current system that need to be addressed, and identify any dataset-specific characteristics that may affect system performance. Another potential path of future work would be to compare our implementation to other techniques such as k-NN graphs, where each data point is represented as a node and edges between points preserve similarity. Exploring more complex network architectures, such as graph neural networks or transformers, might prove beneficial if they are able to learn similarity functions and representations well. Lastly, exploring the use of active learning and reinforcement learning techniques to more efficiently and accurately label the training data might also improve system performance and generality. This may also reduce the amount of labeled data required to train the model, while still maintaining performance.

References

[Datar et al.(2004)] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[Guo et al.(2020)] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 3887–3896.

[Indyk and Motwani(1998)] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.