An Evaluation of Deep Hashing for Nearest Neighbor Search on General Embedding Data: First Steps Towards the Meta4 Engine

AMAR SHIVARAM, Otto Von Guericke University, Magdeburg, Germany
CHINMAYA HEGDE, Otto Von Guericke University, Magdeburg, Germany
DISHA SETLUR, Otto Von Guericke University, Magdeburg, Germany
PRAFULLA DIWESH, Otto Von Guericke University, Magdeburg, Germany
SHIVALIKA SUMAN, Otto Von Guericke University, Magdeburg, Germany
VINAYAK KUMAR, Otto Von Guericke University, Magdeburg, Germany

ABSTRACT: Nowadays, a great variety of everyday information retrieval tasks rely on searching efficiently over a large amount of data. This paper explores techniques to improve data retrieval speed, without affecting search results quality. In specific, we study a model to create similarity-preserving hash codes from embedding data, in order to store effectively their large vectorized information. The problem of managing large dimensional embeddings and identifying an accurate similarity measure to organize these embeddings is addressed here. There are various approaches to this problem. In our study, we consider the data-dependent Triplet-based deep learning hashing approach. We implement this on the GloVe Dataset and the ImageNet dataset for which we use the popular EfficientNet architecture to obtain the embeddings. After training our model for the hash code assignation, we evaluate to what extent this data organization helps in supporting efficient Approximate Nearest Search, comparing the runtime and mean recall with respect to a full search. While using a large scale processing framework, we find that on both datasets we are able to train our triplet network model to provide hash codes that accelerate the average search for top-25 most similar items, without compromising the mean recall obtained. We observe that this is achieved while using models that are still not trained to their full potential, which suggests that our approach might still offer further improvements.

Additional Key Words and Phrases: Deep Hashing, Approximate Nearest Neighbour Search, Triplet Loss, ImageNet, GloVe

ACM Reference Format:
Amar Shivaram, Chinmaya Hegde, Disha Setlur, Prafulla Diwesh, Shivalika Suman, and Vinayak Kumar. 2020. An Evaluation of Deep Hashing for Nearest Neighbor Search on General Embedding Data: First Steps Towards the Meta4 Engine. 1, 1 (January 2020), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Applications of Machine Learning and Artificial Intelligence are driving the progress of data management technologies today. These applications require meaningful information from expanding amounts of business data. Feature selection is one of the most important steps in the development of AI models for these applications. The most relevant features are created for a specific domain. Creating hand-crafted features takes a lot of time and it becomes harder to select appropriate features when we have a massive amount of high dimensional data. Also, as the size of the features increase, the performance of the model might deteriorate, since there might be scarcity of observations on all features. To obtain features in the most optimal way which is error-free and reduces time and with the desirable cost is a major challenge. One solution to this problem are embeddings.

Embedding is a representation learning technique, which consists of creating a model that can map entities of data into a continuous dense vector that represents the entity. Such vector might have smaller dimensions than the original data, or not. Embeddings are useful too when handling high dimensional data that is non-Euclidean in nature, for example, Images or Texts. Embeddings help overcome the curse of dimensionality. Furthermore, embeddings can be trained to capture many aspects of the entities under study.

All embeddings for a particular entity, are stored in what is known as an Embedding Space. The embedding space contains all the entities of a data set. This helps us infer that all similar entities will be placed relatively close to each other in this embedding space, when compared to dissimilar entities. Even though there is a lot of progress on embeddings, there are fewer contributions to proper data management solutions of embeddings. Given the usefulness of embeddings to help in similarity search, or relational reasoning, we believe that this will be one of the most active research areas in the coming days. Key related works on the use cases are highlighted in the background section of this paper.

Embeddings are being used for similarity search, recommender systems to recommend relevant and similar items, image search and many other cognitive application[2] areas. But to load and manage such a large number of entities, can be problematic as some of the data management pipelines are not designed to handle large vector data. A proposed solution that would overcome this obstacle and effectively utilize and manage embedding data is similarity-preserving hashing.

A hash function is one that maps a certain vector of data having an arbitrary size to compact fixed-size values. Similarity-preserving hashing is one that apart from hashing, is able to make that hash codes of similar data are also similar. The use of hashing to accelerate exact or similar retrieval could assist in storage issues of embeddings, since it could enable us to group these vectors in a way that is coherent with the expected similarity-queries that they are likely to receive. The hash values are stored in a table called the hash table, which behaves like an index over the large-sized vector data. Each hash code is unique to an entity.

In similarity-preserving hashing, we set a top-k similarity search criteria. We have used the ImageNet and GloVe dataset. We also use a standard pre-trained embedding model GloVe, and we have used Apache Spark's Parquet file format for storing the data in a partitioned format after hashing.

An interesting approach used in this paper is, to obtain similar embeddings based on Hashcodes by using approximate Nearest Neighbour Algorithm. It is an operation prevalent in many domains such as Computer Vision, Databases, multimedia, etc. The regular algorithm searches for the nearest neighbors based on the smallest distance between them. This method is less expensive when used with hash codes as the entry point to the data is no longer high dimensional, and after a first search on the hash table, we need a smaller number of comparisons to fetch the top-k results.

The main work and contributions of this paper are summarized as follows:

(1) We propose functions that help to understand the training of a deep hashing model such as a) Coverage b) Computation c) Number of hash codes, and observe how the model trains over different datasets and hyper-parameters, to help build a robust and usable deep hashing network. We report successful training, though it achieves less than expected hash codes.
(2) We evaluate the performance of Approximate Nearest Neighbours(ANN) with hash codes v/s normal Nearest Neighbours with a similarity search criteria for top-k, on two datasets and report the potential of deep hashing to accelerate this process, on an Apache Spark cluster, using partitioned Parquet files.

The remainder of our paper is structured as follows: We present the theoretical work and literature survey for the background in Section 2. This is followed by the design where we describe the architecture and the tools used in this project in Section 3. Section 4 deals with Evaluation measures and experimental setup. The works related to what we have discussed in this paper are discussed in Section 5 and we conclude the paper in Section 6, summarizing our findings and proposing future work in Section 7.

2 BACKGROUND

This section deals with the early findings about Embeddings and Hashing. Embeddings are used in an extensive manner in this paper. The embeddings are the ones which are sent through the hash functions and thereby a hash code is obtained. These topics are discussed in detail in the upcoming sections. The Hashing part gives an overview of the different types of hashing approaches available which can help us to obtain similarity-preserving hash codes; and the one suited for the approach we follow.

2.1 Embeddings

We presented the problem of feature selection concisely in the introduction section. Apart from this, AI poses many other problems that revolve around data. A. L'Heureux et al. contribute a survey of these challenges in the paper [7].

Embedding is a technique to represent discrete variables as continuous values in a vector. These vectors are mapped to a vector space in N dimensions. With the advent of Natural language processing, embeddings gained popularity where it translates non-euclidian textual data to vectors in relatively low dimensional space. The evolution of embeddings dates back to traditional language models where the interest is on a probability distribution over a sequence of words. Neural networks were trained with input text corpus and the results were the probability of the word randomly chosen nearby to the input word. The objective is to learn similar embeddings for words/tokens in similar contexts and preserving the relationship between them. The survey from S. Wang et al.[14] explores the field of word embeddings from initial language models to recent advancements in it.

Furthermore, embeddings are used to convert graph data into low dimensional space by preserving the graph properties and aiding graph analytics. A comprehensive survey of graph embeddings [3] contributes towards detailed analysis and taxonomy of graph embeddings.

2.2 Hashing

Hashing has been widely used in multimedia data retrieval due to its efficient storage and retrieval. The idea of hashing is to convert each document into a small signature using a hashing function. This paper throws light on a popular method of hashing known as Locality Sensitive Hashing (LSH), which is also named similarity-preserving hashing. LSH stands for any kind of algorithm to identify approximate nearest neighbors. The goal is to assign a high similarity hash code for documents that have a high similarity, and dissimilar codes for documents that are very dissimilar. It reduces the high dimensional features into smaller dimensions while preserving the differentiability.

LSH has many applications. Some of them are mentioned below.

- Recommender systems
- Near duplicate detection
- Image similarity identification

Approaches to LSH can be broadly classified into two categories

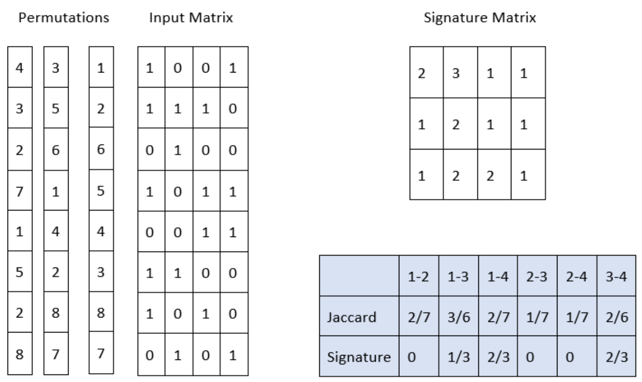
(1) Data Independent Methods
(2) Data Dependent Methods

2.2.1 Data Independent Methods. These methods are where the hash functions are defined independently of the training dataset.*Min Hashing* is a type of data-independent method which is popular. In this method, the rows are permuted randomly and we select the *min-hash* function once and apply the same to each of the columns. The evaluation metric of min-hash is *Jaccard similarity.* The hash function is represented as follows,

$$h_\pi(C) = \min_\pi \ \pi(C) \tag{1}$$

where C represents the column

For min-hash as described in Figure 1, we have an input matrix, which is a sequence of documents (represented along the columns), represented by a series of shingles which indicates if it is present or not present along the rows (1 if present, 0 otherwise). We have the permutations that give the signature based on each permutation that tells you the first position that is non-zero for each permutation in the matrix. The signature matrix represents the hash-code of the


Permutations
Input Matrix
Signature Matrix

Permutations			Input Matrix			
4	3	1	1	0	0	1
3	5	2	1	1	1	0
2	6	6	0	1	0	0
7	1	5	1	0	1	1
1	4	4	0	0	1	1
5	2	3	1	1	0	0
2	8	8	1	0	1	0
8	7	7	0	1	0	1

Signature Matrix			
2	3	1	1
1	2	1	1
1	2	2	1

	1-2	1-3	1-4	2-3	2-4	3-4
Jaccard	2/7	3/6	2/7	1/7	1/7	2/6
Signature	0	1/3	2/3	0	0	2/3

Fig. 1. Min Hashing

corresponding documents along the columns. These codes preserve similarity between the documents. There isn't a fixed number of permutations and these permutations can be done any number of times.

The similarity, by the end of the process, is evaluated based on Jaccard Similarity which is calculated based on the input matrix and the signature which is calculated based on the signature matrix.

The shortcomings of a data independent method like min-hashing, are as follows:

- It's hard to pick random permutations of large data sets
- Representing the random permutations takes a lot of space
- It's a time consuming process as we access the input matrix each time for every permutation

Apart from these, overall, data independent methods make assumptions on the data structure which might not be adequate. Therefore methods that consider the data itself have been widely studied in the recent years.

2.2.2 Data Dependent Methods. As the name implies, in these methods the hash functions are learned from the dataset. Compared to the Data Independent methods, Data Dependent methods achieve better output characteristics. The data-dependent methods can be classified into three types.

(1) Unsupervised Hashing
(2) Supervised Hashing
(3) Deep Hashing

Unsupervised Hashing. For an unsupervised hashing, the input is a feature vector x_i and the output is set of binary codes b_i. The instances similar in the original feature space should have similar binary codes. x_i is a compact matrix for all training points in X. The process is as, when x_i is close to x_j, the hamming distance between b_i and b_j should be low. When x_i is far off to x_j, the hamming distance between b_i and b_j should be high. An example of unsupervised hashing is *PCA Hashing*.

Supervised Hashing. The input for a supervised hashing is a feature vector x_i which is a compact matrix for all training points in X and class labels y_i or pairwise constraints s_{ij}. The instances similar in the original feature space have similar binary codes (ie) when s_{ij} = 1, the hamming distance between b_i and b_j should be low. When s_{ij} = 0, the hamming distance between b_i and b_j should be high. Eg. *Kernel based Supervised Hashing*. By evaluating the technique mentioned as done in [12], we find that Supervised hashing has better output characteristics in comparison to Unsupervised hashing and the Data Independent methods.

Deep Hashing. The deep hashing uses deep learning techniques (artificial neural networks) for hashing the data. The data sent for hashing can be both structured and unstructured. The method uses a deep neural network to output approximate hash codes for the data. If the labels for the data are available, it can be added to the output layer to learn a better-shared representation of the data. Eg. *Convolution Neural Network Hashing*(CNNH). By evaluating CNNH as done in [17], it is observed that Deep hashing techniques perform better than Supervised hashing techniques.

Hence it is inferred that the data-dependent methods improve over the data independent methods for LSH. And amongst the data-dependent methods, the deep hashing methods stands out prominently, with better evaluation results. The systems which use deep learning for LSH are still in early-stage as this is a field that is still in research. Given its high speed and low memory cost, similarity-preserving hashing can be an integral part of Approximate Nearest Neighbour search.[6]

3 DESIGN- DEEP HASHING FOR HIGH DIMENSIONAL SIMILARITY SEARCH ON EMBEDDED DATA

The aim in doing this project is to research the potential of speeding-up the retrieval of multimedia data with efficacy and robustness using Deep Hashing. The multimedia data which we have is embedded and these embeddings are then passed on through a hash function and thereby hash codes are retrieved for each of these embeddings. These hash codes are then checked for similarity.

Research Questions: Our experiments are designed to address the following research questions.

(1) What hyper-parameters have a big impact on the Deep Neural Network for efficiently learning hash codes over different datasets?
(2) Does Hashing improve the speed in top-k similarity search in image and text data, if so, how is mean recall compromised?

3.1 Architecture

3.1.1 Physical Architecture. The Figure 2 describes the high-level overview of the architecture followed in this paper. The data is of many input formats- Structured and Unstructured. The ones we concentrate in this paper is mostly on unstructured data. We use both text data and image data as our input data and performed the depicted process. Using embedding methods on these data formats, we obtain their vector representation in embedding space. Using these embeddings, we can find how similar the data are, in a way that does not require feature engineering or complex similarity measures (i.e., cosine similarity applies quite well). To store and manage these embeddings efficiently and have a comparison in a faster approach, we learn hash codes for these embeddings using a neural network with triplet training, which we describe in a later part of the paper. These hash codes seek to provide improved query speeds, by grouping under a common hash code similar items with a high probability. These hash codes are then added to the original data, which is partitioned by these codes and stored with snappy compression in parquet files. A similarity search is performed by querying the data with their hash-codes first, and then comparing the embedding for the query with all those in the same bucket, retrieving then the top-K similar items. Finally, we evaluate the runtime and recall, when compared to a full search.

The big data ecosystem of Spark is used in the architecture to carry out the process of similarity search as the data to be compared is huge. Hence we need a mechanism which can handle huge data as well as support fast retrieval of similarity. The right way to achieve this mix was to use the ecosystem of big data by storing the files as partitioned parquet files and processing them in spark thereby achieving fast retrieval.

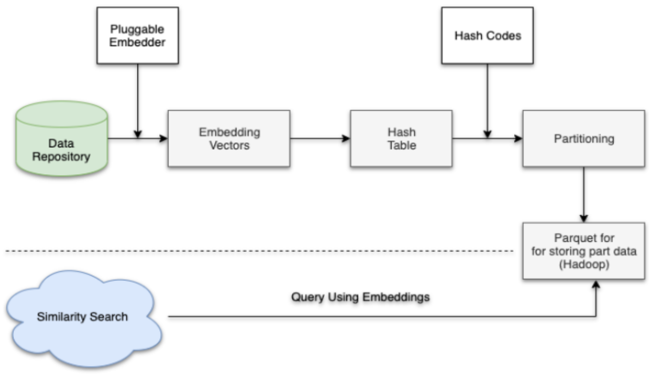


Fig. 2. Physical Architecture

3.1.2 Architecture of the triplet-based hashing. Wang et al in [15] proposed this triplet-based hashing method. It performs hash code learning in a manner that aims to maximize the likelihood of the input triplet labels. The image as discussed in Figure 3 has three components, a deep neural network to learn features from images, a fully connected layer to learn the hash codes from the features and a loss function.

The input fed to the network is a query, a positive data example, and a negative data example. Here the positive example is one that is labeled as similar to the query, while the negative example is labeled as dissimilar. The embeddings of these inputs are passed to the neural network. This network outputs hash codes and these hash codes are used for triplet label similarity and loss function. The loss function is discussed in a later part of the paper. The embeddings which are similar to each other produce similar hash codes.

3.2 Model

The Network Architecture used for Image Data and Text data are different.The hash function is calculated using the following formula.

$$\mathcal{L} = -\sum_{s_{ij}}(3 \times \theta_{P1_m P2_m} - \log(1 + e^{\theta_{P1_n P2_n}}) - \log(1 + e^{\theta_{P1_n N_n}}) - \log(1 + e^{\theta_{P2_n N_n}}))$$

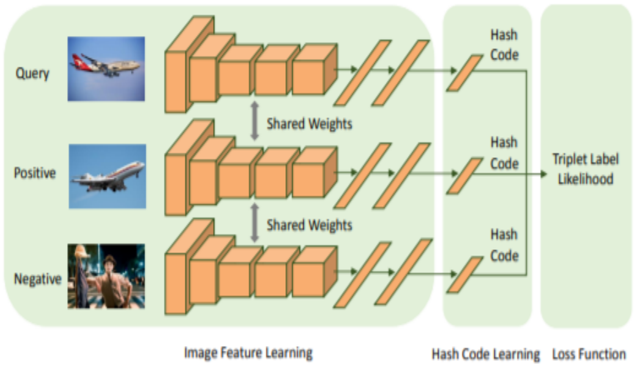


Fig. 3. Architecture of triplet-based hashing[15]

3.2.1 Network Architecture - Image . A fully Connected Neural Network has been used for our model on image data. The dimension of the input is 40960. The network we used for our evaluation, after some architecture exploration, has 8 hidden layers followed by a Dropout of 0.5 and 0.4. The Activation Function is leaky Relu for the hidden units and tanh for the output layer. The code length as the output layer for hash codes were 7, 8 and 9.

3.2.2 Network Architecture- Text. A fully Connected Neural Network has been used for our model on textual data. The input dimension is 199. The network has 3 hidden layers. The number of activation units starts from 128 in the first layer and decreases till 32 in the last layer. The code length for hash codes were 3 and 4. The Activation Function is leaky Relu for the hidden units and tanh for the output layer. The code length used as the output layer for hash codes were 3 and 4.

3.3 Approximate Nearest Neighbour Search

The Approximate Nearest Neighbour(ANN) Search is the key part of our design model. This search provides the top-k search results. This is carried out in Apache Spark since it can handle large amounts of data. The neighbours for each data item is computed in the following manner as explained in Algorithm 1. The data item is taken and a cosine similarity of the data item and all the other embeddings in the repository is done using pairwise cosine similarity measure. The *top-k* similarities is taken and presented. This gives us similar items to the given data. This is done for all the data item and hence this involves a great amount of data processing, and calculation as the data is voluminous.

The Figure 4 represents a brief overview of the process followed in the ANN Search. It is divided into 3 blocks.

(1) Data Conversion
(2) Nearest Neighbour Search
(3) Evaluation

Data Conversion. The Data Conversion block is responsible for the conversion of the file to a parquet file so that it is easy to retrieve and efficiently process the data. The conversion is done using Spark from a *csv* file to a *parquet* file. These files are partitioned by the hash codes along the columns.

Algorithm 1 ANN with Hashcode and Embeddings

```
1: procedure PARTBYHASHCODE($part$)
2:     for each hashcode $i$ in $part$ do
3:         $top - k\ search$
4:             if  $top - k\ notFound$ :
5:             $bitwiseSearch$
6:         end if
7:     end for
8:         $Compare\ with\ groundtruth$
9:         $Calc\ recall25$
10:    Return execTime, recall25
11: end procedure
```

Nearest Neighbour Search. The Nearest Neighbour Search searches for *top–k* ANN for each hash code. There might occur a condition that *k* data items are not retrieved. In these cases, a `1 Bit Search` is performed retrieve *k* data items. A *top-k* search based on the embedding is also performed for each data item to get the ground truth of the data item upon which evaluations can be conducted.

Evaluation. The Evaluation block of the ANN Search performs two main operations. It reports the time for each Hash code search. It also evaluates the predicted value with the actual values obtained. The evaluation metric used in our case is `Recall-K`. Recall-K computes the Recall values on the retrieved K approximate nearest neighbour data items. The actual values are computed when a top-k search is performed on the embeddings and the predicted values are computed when a top-k search is performed on the hash codes. These values are fed and a Recall-K metric is obtained. For each data item, the search is done on the entire item set and the hashed bucket item set. The recall is calculated individually for each data item in consideration and the mean value across the K examples is taken as the recall-K value.

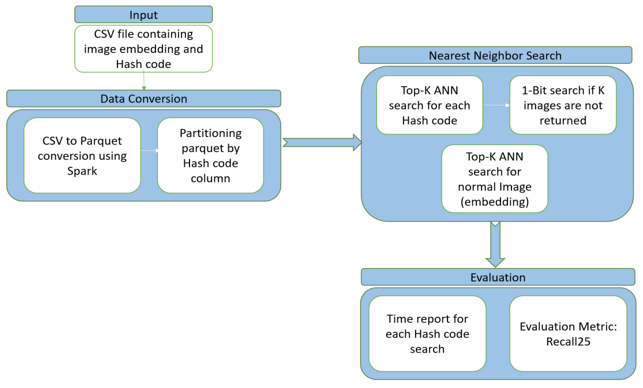

Input
CSV file containing image embedding and Hash code
Data Conversion
CSV to Parquet conversion using Spark
Partitioning parquet by Hash code column
Nearest Neighbor Search
Top-K ANN search for each Hash code
1-Bit search if K images are not returned
Top-K ANN search for normal Image (embedding)
Evaluation
Time report for each Hash code search
Evaluation Metric: Recall25

Fig. 4. ANN Search

4 EVALUATION

4.1 Experimental Setup

4.1.1 Image Data set. The ImageNet DataSet has been taken to generate the embeddings as our example dataset for learning the

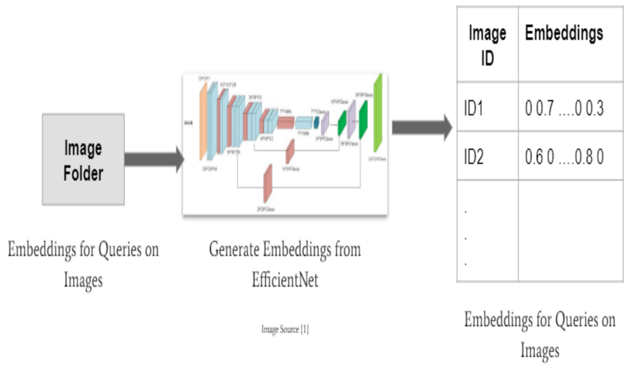


Image Folder
Embeddings for Queries on Images
Generate Embeddings from EfficientNet
Image Source [1]
Image ID
Embeddings
ID1
0 0.7 ....0 0.3
ID2
0.6 0 ....0.8 0
Embeddings for Queries on Images

Fig. 5. Generating Embeddings from Efficient Net [9]

hash codes. ImageNet Dataset originally has 14 million Image data. But, for our Implementation, used 100,000 images from 200 classes, to reduce the computational efforts. To generate the embeddings for these images, the EfficientNet pre-trained model on Imagenet has been used, by removing the classification layer and taking as an embedding the output of the remaining last layer. For training, we have used a Fully Connected Neural Network, having leaky Relu as activation functions in non-classifying layer and tanh in the output layer. We have explained this previously.

4.1.2 Text Data set. The GloVe pre-trained word embeddings are used as our text dataset for learning the hash codes. Due to the lack of labels, the pre-trained word embeddings are made to undergo a clustering process with a `DB Scan Algorithm` so that we can obtain clusters in the dataset and assign these clusters as class labels (4 number of classes in our case). A total of 199 examples were extracted to train the network. The word, its cluster label which now is our class label and the word embeddings are fed into the hash functions in order to retrieve hash codes. The hash codes are similar to the data belonging to the same class label.

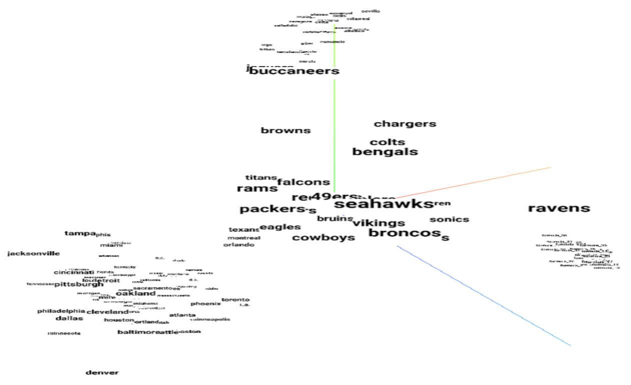


Fig. 6. Words from GloVe represented in Feature Space

A sample set of words are taken along with their class labels and embeddings and they are visualised in the feature space as presented in Figure 6. This figure represents the unique split in the embeddings and they are projected into the feature. The pattern observed is such that all team names are aligned towards one side and all city names are aligned towards another side.

4.1.3 Implementation Details. The work for the paper was done in `Python 3.7`. There were several software libraries used to do the tasks for this paper. `Tensorflow 1.0` was used for obtaining the embeddings for the images and hashing the data available. `Pandas` was used for the various Data Transformation tasks to be done. The numerical operations were performed with `Numpy`. `Scikit-learn` was used for the various machine learning tasks such as clustering and calculating the similarity. Handling of large data files in Spark was done by `PySpark`. The word similarity to retrieve the ground truth for words was done using `Gensim`. `Matplotlib` and `Pyplot` were used for the various plotting tasks in the tasks.

4.1.4 Parquet. The processing of huge data to check for similarity is done using Spark. The data is sent as a *csv* file. These files contain information about the id, embeddings and the hash code of the data. The hash codes are obtained after passing the embeddings through the hash function. The files are then converted to parquet files as this file system supports to store huge data in a convenient way. They also have fast retrieval properties. The files are converted to parquet using the spark system and these parquet files are partitioned by the hash codes. We also select snappy compression. They are stored in the repository with respect to hashcodes. For evaluation purposes, each repository contains information about the data having similar hash codes.

Apache Parquets are columnar storage format available in the Hadoop ecosystem. These formats compress the data efficiently and encodes them in such a manner that it can handle complex data when provided in bulk. The compression in these data is performed column by column. It has different encoding schemes for text and numerical data. Thus this puts forward a mechanism to implement new encoding schemes according to the variety of the data.

4.1.5 Hardware Configurations of the System. The ANN Search is done in a Spark cluster. The cluster has ten Spark clients, which includes one node to access the cluster, two nodes as HDFS NameNodes (one active node and the other a standby node) and seven executor nodes as HDFS DataNodes, which can also be used to run Spark applications. Each of them runs on virtual machines, with VMWare ESXi as a hypervisor.

In the following, we introduce hosts' hardware for seven executor nodes. Each of them has four CPU cores, 16 GB RAM (6 GB is available for executor nodes to run Spark applications) and 150 GB hard disk. The cluster is heterogeneous since four of the nodes are with four Intel Xeon E5-2650 @2.00 GHz cores, two nodes are with four Intel Xeon E5-2650v2 @2.60 GHz cores, the last node with four Intel Xeon E5520 @2.27 GHz cores. All hosts are connected by a 10 GBit Ethernet in a star topology.

4.2 Implementation

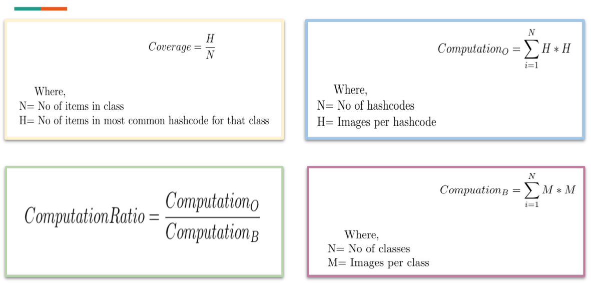
In our experiment here, we have used the ImageNet dataset which consists of 100000 images. The images are categorized into 200 classes with approx 500 images belonging to each class. Our objective here is to generate hashcodes for these images. To simplify our task, we first extract the embeddings from these images. The existing EfficientNet neural network as shown in Figure 5 is used here to extract these embeddings. EfficientNet has proven to work well on the ImageNet dataset with an accuracy of 84.4%. Hence we have used this network, and the last layer is removed, to obtain the embeddings for each image.

The embeddings are then trained based on the triplet loss function to obtain the hashcode where a loss function is a key component in the training procedure. The triplet loss is a loss function in Artificial Neural Network which evaluates the likelihood of the learned hashcode to that of the triplet labeled inputs i.e baseline input is compared with a positive input(true) and a negative input(false). The objective is to optimize the weights in the network in a manner such that the distance between the baseline and the positive examples is minimized and the distance between the baseline and the negative examples is maximized.

Here we train batch-wise with a batch size of 32 input and repeat this over 4 epochs with 10,000 iterations.

4.3 Results

4.3.1 Hyperparameter tuning. Initially, training of the Deep Hash Model, a smaller network with 1 hidden layer with 5120 activation units was chosen. By doing a trial and error method, we have trained the model on the current architecture. Deciding the hash code length to be generated was also a challenge, hence we ended up training the model with three different numbers of activation units (7,8 and 9) in the output layer for Image Data and two(3 and 4) for textual Data. Also, we used three different Activation Functions in the output layer: Tanh, Sigmoid and None. In the graphs below, the evaluation metrics for hashing, i.e; Computation and Coverage for all the activation function and code length combination used while training, have been shown. Also the number of Hash codes generated while training has been shown for different combinations of Activation Functions and Hash code length. Our goal here was to Minimize the computation and maximize the coverage and the Ideal hash codes would have been the number of classes being used for training. The formulae for Computation and Coverage is given in Figure 7



$$Coverage = \frac{H}{N}$$

Where,
N= No of items in class
H= No of items in most common hashcode for that class

$$Computation_O = \sum_{i=1}^{N} H * H$$

Where,
N= No of hashcodes
H= Images per hashcode

$$ComputationRatio = \frac{Computation_O}{Computation_B}$$

$$Compuation_B = \sum_{i=1}^{N} M * M$$

Where,
N= No of classes
M= Images per class

Fig. 7. Computation and Coverage

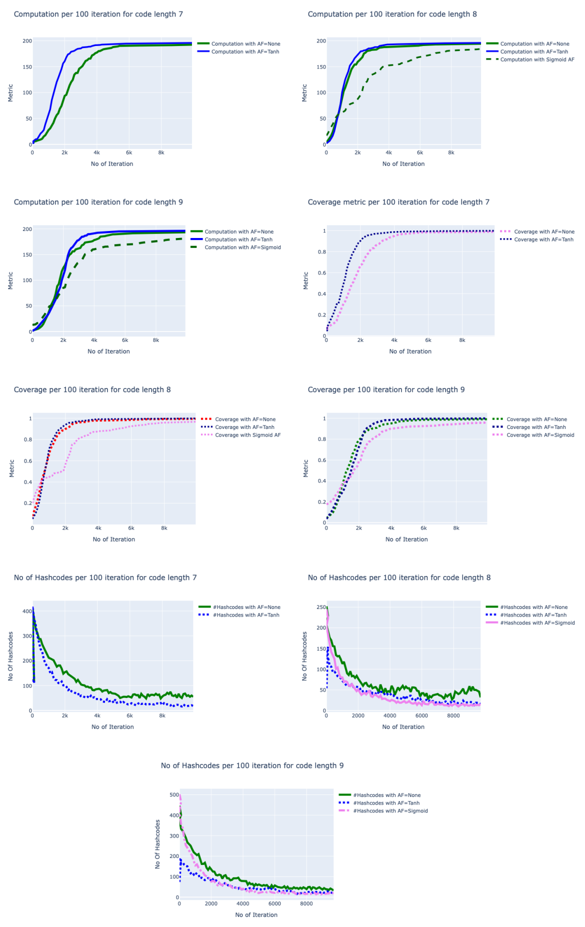


Fig. 8. Computation, Coverage and No of Hash code Plots for ImageNet

The parameters combination that performed really well on Image data training is : Activation Function as None and code length 7, which gave us 55 Hash codes for 200 classes.And the best results on text data is for the combination: Activation Function as None, Code Length as 4.

ImageNet Results: Figure 8 shows graphically, the computation, coverage and number of hash codes per 100 iterations for a range of code lengths namely 7,8,9. The plots were drawn with Activation function as *tanh, sigmoid* and none.

GloVe Results: Figure 9 shows graphically, the computation, coverage and the number of hash codes per 100 iterations for code length of 3 and 4 for the GloVe dataset. The plots are plotted with no and *tanh* activation function.

For both datasets our best solution do not achieve an ideal computation of 1. This highlights that there is still room for improvement.

4.3.2 Approximate Nearest Neighbour(ANN) Search. ANN Search is done on both ImageNet and GloVe dataset. As the data in consideration is voluminous in nature, we use Spark to efficiently process the data. The similarity measure used is *cosine similarity.* The Recall

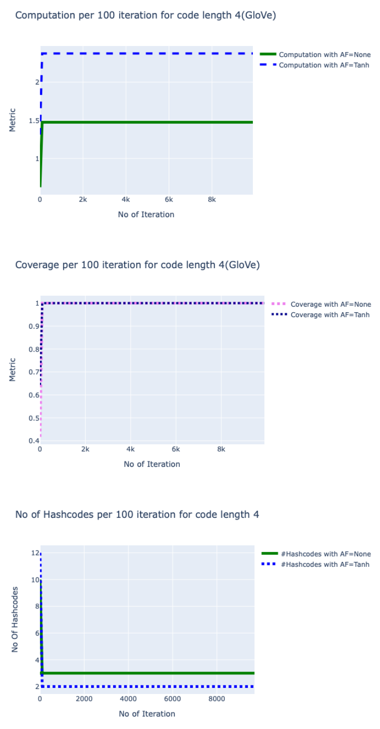
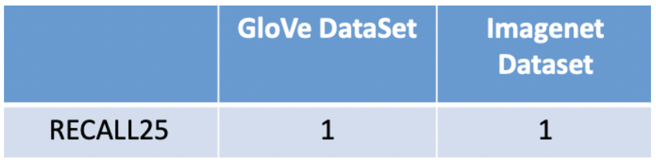


Fig. 9. Computation, Coverage and No of Hash code Plots for GloVe

values obtained are mentioned in Figure 10. For the selected number of examples we achieve a perfect recall, which mean that the 25 results for each image are the same when using the learned hash code to filter the search, as when not using it.



	GloVe DataSet	Imagenet Dataset
RECALL25	1	1

Fig. 10. Recall25 Evaluation Matrix

Results for ImageNet. The Figure 11 gives us the time taken by the ANN search to run on the Complete Embedded Data, Hash Partitioned Data and the mean time of the complete ANN Search. These runs are made for K=25 Images. The figures describes time taken for finding nearest neighbour of each image. So for K=25 images,the search time are shown. Time here are taken in context of finding the cosine similarities between the images and the top-K retrieved images. The mean time taken for searching top-K images in the complete data set approximately is 0.21472 seconds when compared to the mean time taken to search for top-K images in hashed bucket which took approximately 0.16488 seconds. So the hashed bucket search is faster in comparison to the normal one, by 1.3x times faster.

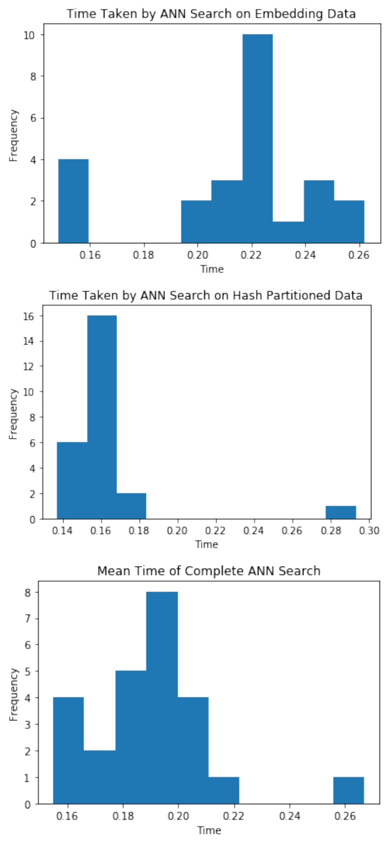


Fig. 11. Search Time for K=25 images

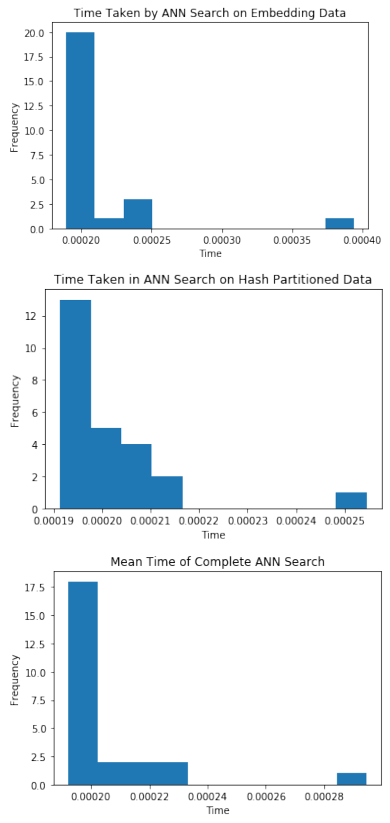


Fig. 12. Search Time for K=25 text items

Results for GloVe. The Figure 12 gives us the time taken by the ANN search to run on the Complete Embedded Data, Hash Partitioned Data and the mean time of the complete ANN Search. These runs are made for K=25 Text items. Similar to image search the Top-K nearest neighbour search is done on complete text data set and hash bucket. The mean time taken for searching top-K texts in the complete data set is approximately 0.00021 seconds when compared to the mean time taken to search for top-K texts in hashed bucket which took approximately 0.00015 seconds. So the hashed bucket search is faster in comparison to the normal one, by around 1.4x times.

Finally, concerning the run time obtained for ANN, we should observe that since the models did not train to the best computation, and a number of hash code that matches or is more fine-grained than the classes given, we consider that there is still room for improvement in the run time.

5 RELATED WORK

The semantic similarity was highlighted using embedded data for texts in the thesis work of Rutuja Pawar [8] with the triplet labels[16], loss function, the fastText[5][1] pre-trained model and entity resolution datasets. A high dimensional similarity search was carried out using Locality Sensitive Hashing(LSH) and Learning to Hash(L2H) technique. Evaluation and comparison were done using the benchmark metrics for LSH[10] and L2H[13]. Our paper contributes by using the embeddings of Images from Imagenet[4] and helps to build a robust deep hashing network using EfficientNet[11] and standard metrics for evaluation.

ANN search was used with Hamming Distance in comparison with a series of deep hashing methods to show the potential of deep learning networks in the paper by Li et al. [6]. Our paper contributes in comparison and report of computation time between deep hashing and approximate nearest neighbours on top-k search on embeddings and hash codes. Bitwise search has been used if the top-k results are none. Recall-K and execution time are also included among the evaluation metrics.

6 CONCLUSION

Our work shows that hash codes can be learned using deep neural networks and the results show the potential of hashing techniques using embeddings compared to traditional Nearest Neighbour search. We also highlight the impact of hyper-parameters, reporting on the training of models that though successful (reducing the loss), do not achieve a number of hash codes close the the optima. Hence, we believe that there is still a large room for improvement in model training. Still, using the resuling models we evaluate and performance of approximate nearest neighbors vs.

nearest neighbors on the complete dataset on this paper, finding some improvements thanks to the use of learned hashes without deterioration of the mean recall.

7 FUTURE WORK

User personalization applications are driving the demand for data management systems that can help queries related to user-level or semantic-level similarity between items. Models like topic models, latent semantic hashing and embeddings, are able to learn a continuous dense vector that can be added to items to represent how it relates to others according to its dimensions. Smoothly interfacing between traditional data queries, and queries that can use such kind of embedding data is likely to greatly enhance what data management systems can offer to their end-users.

We envision a pluggable system, which we call the Meta4 Engine, which could facilitate the process of learning embeddings, augmenting data with such embeddings, creating indexes or partitions for efficient data access, and offering embedding-based queries such as top-k similar search, comparison queries (e.g. Berlin is to Germany as Paris is to?), or many others. Such a system stands at the intersection between databases and search engines, hence we argue that it should be pluggable to both kinds of systems. In this paper, we studied hashing as a building block for efficient data access and we validated that it can help accelerate nearest neighbor searches without a notable loss in accuracy. We found that architectural searches are needed to adapt the model to different datasets. Hence, we identify a need for a solution that could assist in the model adaptation, like AutoML. In future work, we thus aim to consider AutoML for deep hashing, query definitions to cover all that can be queried-for related to embeddings, the integration into a database and a search engine with support for in-system embedding learning, and further possibilities (apart from deep hashing) for storage engine improvements.

ACKNOWLEDGMENTS

We would like to express our deep gratitude and sincere thanks to M.Sc. Gabriel Campero Durand for his valuable guidance, encouragement and support throughout this project. We also want to thank Prof. Dr. rer. nat. habil. Gunter Saake for giving us an opportunity to work on this project.

REFERENCES

[1] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching Word Vectors with Subword Information. *CoRR* abs/1607.04606 (2016). arXiv:1607.04606 http://arxiv.org/abs/1607.04606

[2] Rajesh Bordawekar and Oded Shmueli. 2017. Using Word Embedding to Enable Semantic Queries in Relational Databases. In *DEEM'17*.

[3] Hongyun Cai, Vincent Zheng, and Kevin Chang. 2017. A Comprehensive Survey of Graph Embedding: Problems, Techniques and Applications. *IEEE Transactions on Knowledge and Data Engineering* (09 2017). https://doi.org/10.1109/TKDE.2018.2807452

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[5] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of Tricks for Efficient Text Classification. *CoRR* abs/1607.01759 (2016). arXiv:1607.01759 http://arxiv.org/abs/1607.01759

[6] Mingyong Li, Ziye An, Qinmin Wei, Kaiyue Xiang, and Yan Ma. 2019. Triplet Deep Hashing with Joint Supervised Loss Based on Deep Neural Networks. *Computational intelligence and neuroscience* 2019 (2019).

[7] A. L'Heureux, K. Grolinger, H. F. Elyamany, and M. A. M. Capretz. 2017. Machine Learning With Big Data: Challenges and Approaches. *IEEE Access* 5 (2017), 7776–7797. https://doi.org/10.1109/ACCESS.2017.2696365

[8] Rutuja Pawar. 2019. An Evaluation of Deep Hashing for High-Dimensional Similarity Search on Embedded Data, Masters' Thesis, Otto von Guericke University, Magdeburg.

[9] Sankaranarayanan Piramanayagam, Eli Saber, Wade Schwartzkopf, and Frederick W Koehler. 2018. Supervised classification of multisensor remotely sensed images using a deep learning framework. *Remote Sensing* 10, 9 (2018), 1429.

[10] Kohei Sugawara, Hayato Kobayashi, and Masajiro Iwasaki. 2016. On approximately searching for similar word embeddings. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2265–2275.

[11] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *CoRR* abs/1905.11946 (2019). arXiv:1905.11946 http://arxiv.org/abs/1905.11946

[12] Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2010. Semi-supervised hashing for scalable image retrieval. (2010).

[13] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. 2015. Learning to hash for indexing big data—A survey. *Proc. IEEE* 104, 1 (2015), 34–57.

[14] Shirui Wang, Wenan Zhou, and Chao Jiang. 2019. A survey of word embeddings based on deep learning. *Computing* (11 2019). https://doi.org/10.1007/s00607-019-00768-7

[15] Xiaofang Wang, Yi Shi, and Kris M Kitani. 2016. Deep supervised hashing with triplet labels. In *Asian conference on computer vision*. Springer, 70–84.

[16] Xiaofang Wang, Yi Shi, and Kris M. Kitani. 2016. Deep Supervised Hashing with Triplet Labels. *CoRR* abs/1612.03900 (2016). arXiv:1612.03900 http://arxiv.org/abs/1612.03900

[17] Rongkai Xia, Yan Pan, Hanjiang Lai, Cong Liu, and Shuicheng Yan. 2014. Supervised hashing for image retrieval via image representation learning. In *Twenty-eighth AAAI conference on artificial intelligence*.