HALOS: HASHING LARGE OUTPUT SPACE FOR CHEAP INFERENCE

Zichang Liu [*1] **Zhaozhuo Xu** [*1] **Alan Ji** [1] **Junyan Zhang** [1] **Jonathan Li** [2] **Beidi Chen** [2]
Anshumali Shrivastava [1 3]

ABSTRACT

Efficient inference in large output space is an essential yet challenging task in large scale machine learning. Previous approaches reduce this problem to Approximate Maximum Inner Product Search (AMIPS), which is based on the observation that the prediction of a given model corresponds to the logit with the largest value. However, models are not perfect in accuracy, and the successful retrievals of the largest logit may not lead to the correct predictions. We argue that approximate MIPS approaches are sub-optimal because they are tailored for retrieving largest inner products class instead of retrieving the correct class. Moreover, the logits generated from neural networks with large output space lead to extra challenges for the AMIPS method to achieve a high recall rate within the computation budget of efficient inference. In this paper, we propose *HALOS*, which reduces inference into sub-linear computation by selectively activating a small set of output layer neurons that are likely to correspond to the correct classes rather than to yield the largest logit. Our extensive evaluations show that *HALOS* matches or even outperforms the accuracy of given models with $21\times$ speed up and 87% energy reduction.

1 INTRODUCTION

In recent years, neural networks(NN) with large output space have obtained promising results in recommendation systems (Xue et al., 2017; Fan et al., 2019) and language processing (Bengio et al., 2003; Mikolov et al., 2010; 2013; You et al., 2021b;c). For example, the output space of the recommendation system corresponds to the item catalog. At the scale of the Amazon product catalog, the output dimension can easily surpass millions of neurons (Bhatia et al., 2016). To deploy such networks in the real world and perform online inference with low latency, one main challenge lies in the expensive computational of giant matrix multiplications in the large output layer.

Existing methods reduce computations by formulating the problem as Approximate Maximum Inner Product Search (AMIPS) (Shrivastava & Li, 2014a; 2015; Guo et al., 2016; Zhang et al., 2018). Specifically, neurons from the output layer are pre-processed and treated as indexed data. During inference, input embedding from the last hidden layer serves as a query to retrieve a small number of candidate neurons for output layer computation. This formulation is natural because NN treats the class with the largest score as prediction. And the score is calculated by a monotonic function given the inner products between input embedding and output layer neurons. The goal is to search for neurons that have the maximum inner product with the query in sub-linear time. Current AMIPS algorithms focus on indexing the data via different data structures including graphs, trees or quantized dictionary to greatly reduce computation cost.

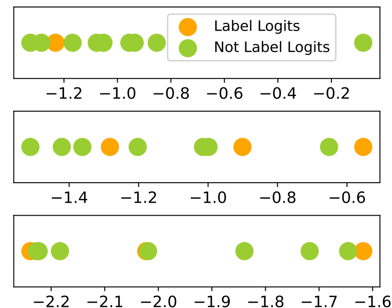

Figure 1: Top 10 logit values for three randomly sampled testing data from extreme classification dataset Wiki10-31k. Logit dots correspond to label class is marked in yellow. The gap between label logits and non-label is narrow, creating difficulties for AMIPS algorithms

Shortcomings of AMIPS Formalism: AMIPS approaches degrade model performance in practice for two reasons. First, AMIPS methods are optimized for retrieving the maximum inner product, which is different from the NN task's objective. NNs cannot generalize perfectly to testing data in practice. The maximum inner product might not lead to the correct class. For example, on Delicious-200K dataset(Wetzker et al., 2008), the average rank for correct

[*]Equal contribution [1]Department of Computer Science, Rice University, Texas, USA [2]Department of Computer Science, Stanford University, California, USA [3]ThirdAI Corp. Texas, USA. Correspondence to: Anshumali Shrivastava <anshumali@rice.edu>.

Proceedings of the 5th *MLSys Conference*, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).

class neuron in logits is only 498.14 out of 205443 for a converged state-of-art NN trained with Softmax function. Second, AMIPS is inaccurate when the gaps between the maximum inner product value and the rest of values are narrow. With a large number of classes, the maximum score is not significantly larger than the rest, and many classes may have roughly the same score, as shown in Figure 1.

An ideal solution would (1) approximate last layer output with sub-linear computation without incurring much overhead, and (2) further optimize for NN task accuracy. Suppose we have an oracle that retrieves a set of neurons from the output layer and only activates retrieved neurons for last layer computation. If such an oracle retrieves a tiny number of neurons, we expect a considerable computation saving, assume retrieval is efficient itself. Moreover, if the retrieved set has the following two properties: 1) The neuron representing the correct label is in the set and, 2) all other neurons in the retrieved set have a smaller score than the correct label. We expect the correct class to have the highest score in this retrieved set, even it may not have the highest score in the full last layer computation, leading to even better accuracy than regular inference.

Therefore, based on the above observation, we design a hashing based neuron retrieval mechanism, which enforces the two objectives mentioned above. We summarize our contribution as following:

- We identify an objective gap between efficient inference and the classical AMIPS formulation. Moreover, to bridge this gap, we observe that inference over a perfect subset of output layer neurons is both efficient and accurate.
- Based on the above observation, we propose *HALOS*, a hashing-based method that uses a learning mechanism to incorporate ground truth information and adapt to logit distribution in the retrieval function. This retrieval mechanism can sample a small subset from last layer neurons with a high probability of including label neurons.
- We provide rigorous evaluations on four large benchmark datasets using two different NN architectures in recommendation systems and language modeling. We show that *HALOS* achieves up to $\mathbf{5}\times$ speed up and at most $\mathbf{87}\%$ energy reduction without any loss in accuracy compared to full computation inference.
- We provide extensive ablation studies on *HALOS* considering efficiency, accuracy, and tuning parameters. We show that *HALOS* outperforms graph and quantization AMIPS methods in query efficiency. Specifically, we achieve at most 2.2 times speedup over state-of-the-art baselines at 0.9 recall.

2 Related Work

2.1 Efficient Inference in Large Output Spaces

Many approaches have been developed for efficient inference in large output spaces (Vijayanarasimhan et al., 2014). Most of these methods can be categorized as an approximate MIPS approach (Levy et al., 2018). Therefore, their accuracy-efficiency trade-offs are determined by the approximation quality. (Zhang et al., 2018) proposed a graph-based method that maps the database vectors in a proximity graph (Tan et al., 2019; Zhou et al., 2019) and outperforms traditional PCA (Bachrach et al., 2014) or SVD (Shim et al., 2017) approaches in language modelling tasks. However, graph-based methods suffer severe performance degradation's in parallel settings because of the difficulty in batching the greedy walks over the graph. Meanwhile, (Morozov & Babenko, 2018) also mentioned the potential risks in the asymmetric transformation in (Zhang et al., 2018) . On the other hand, several AMIPS solvers(Guo et al., 2016; Wu et al., 2017; Chen et al., 2018; Guo et al., 2020) have been proposed for inference over WOLs. However, these solvers trade plenty of computation for accuracy and are both energy and time consuming, even with full parallelism. We provide a detailed literature review in Appendix A.1.

2.2 Hashing Algorithms for Large Scale Machine Learning

Hashing based data structures (details in Appendix A.2) are widely applied in machine learning tasks at scale (Chen et al., 2019b; Spring & Shrivastava, 2020; Chen et al., 2020a; Xu et al., 2021a;b). The general idea of Locality Sensitive Hashing(LSH) is to pre-partition the dataset into buckets where vectors within the same bucket are similar (Shrivastava & Li, 2014a; Indyk & Motwani, 1998; Indyk & Woodruff, 2006). Therefore, given a query vector, the computation can be focused on a tiny subset of the large database. Taking advantages of this massive computation reduction, hashing methods have been applied in: (1) Feature representation: (Li et al., 2011; 2012) demonstrates an efficient way of performing linear learning via permutation-based hashing that preserves Jaccard Similarity preserved (2) Fast nearest neighbor search: (Shrivastava & Li, 2014b; Wang et al., 2018) provides algorithms that tackle efficiency bottlenecks in metric similarity search on ultra high dimensional space. (3) Neural network training: (Chen et al., 2020b) proposes SLIDE, a sub-linear deep learning engine that use LSH to select neurons in forward and backward pass of NN training and achieve outperforming efficiency on CPU. (Chen et al., 2021a) proposes MONGOOSE that combines a scheduler and learnable hash function to improve training efficiency. However, both SLIDE and MONGOOSE focus on NN training and we show that naively apply it in inference causes a significant accuracy drop in Section 4.1.

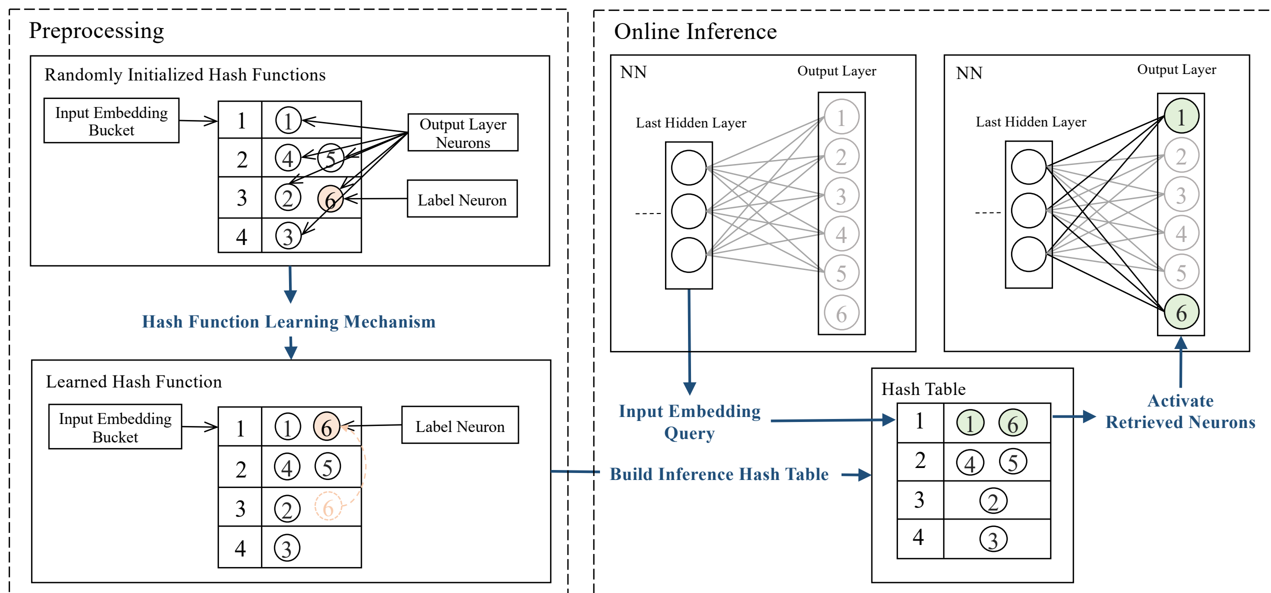

Figure 2: HALOS works in two stages: (1) Offline Processing: We first build hash table with randomly initialize hash functions. Output layer neurons are treated as data, and their indexes are stored in the hash table. Input embedding from last hidden layer is treated as query. Based on current hash table and groud truth information from NN training dataset, we train the randomly initialized hash functions so that input embedding query can find its according label neuron(marked in orange). We rebuild hash tables with the updated hash function for online inference. (2) Online Inference: NN prediction is computed on a subset of neurons retrieved by the input embedding from hash tables rather than the entire last layer neurons.

2.3 Learning to Index vs. Learning to Hash

Learning to index (Dong et al., 2019) focuses on the indexing dynamics while learning to hash (Cao et al., 2017; Shen et al., 2015) focuses on learning the binary representation, which will be used for the task of interests. Learning to hash targets at reducing the cost of each representation distance computation in the linear scan, while learning to index aims to avoid the linear scan via probing a subset of items. Learning to hash ranks the items through hamming distance and calculates the percentage of relevant items in a hamming ball. However, item lying in the hamming ball of a query is not a guarantee for retrieval. For instance, if a query q and its relevant item x are transformed into $K \cot L$-bits hashcodes. The hamming distance of x and q are L. In other words, x lies on the edge of q's hamming ball with a radius L. Learning to hash methods will mark it as a success of retrieval. However, x may lie to the neighbor bucket of q in each hash table, and x cannot be retrieved by hash table. Besides, learning to hash approaches (Cao et al., 2017; Shen et al., 2015) generally report precision and recall with image class information as ground-truth labels. On the contrary, learning to index approaches report the query time versus Top-K similarities recall. For the above reasons, we consider our learning mechanism as learning to index algorithm.

3 Bridging the Gap Between AMIPS and NN Inference

In this section, we first present problem formulation for NN inference with Large Output Spaces. Then, we demonstrate two major challenges for AMIPS methods. At last, we introduce *HALOS* to solve the above challenges.

3.1 Notation and Formulation

We denote the output layer weight matrix as $W \in \mathbb{R}^{m \times d}$ and its bias vector as $b \in \mathbb{R}^m$, where m is the size of output layer (number of classes) and d is the embedding dimension. The last layer can be represented by a set of neurons $\mathcal{C} = \{c_i \,|\, 0 \leq i < m\}$, where each neuron constitutes w_i, the ith row of W, and b_i the ith element of b. Typically, $m \gg d$ in wide output layer setting. During inference, given an input embedding $q \in \mathbb{R}^d$ from the last hidden layer, the output of NN's forward pass is $\sigma(qW^T + b)$, where σ is some activation function that translates the logits into probabilities; then, the indices of the largest logits are returned as the predicted classes. This formulation can be applied to the softmax output layer in language modeling (Zhang et al., 2018), extreme multi-label classification, as well as the matrix factorization in collaborative filtering (Xu et al., 2018).

3.2 The Hardness of Inference by AMIPS

Naively adopting AMIPS for NN inference introduces an objective gap between AMIPS and NN tasks. We all know that NN models cannot generalize perfectly to testing data in practice. AMIPS methods are optimized to retrieve the highest inner product neuron, while the highest inner product neuron may not be the correct prediction.

Further, when highest inner product corresponds to the correct class, the logits distribution from a NN with large output space creates extra difficulties for AMIPS methods to retrieve the highest inner product item within a tight computation budget. We randomly select three test data from Delicious200K (Wetzker et al., 2008). With a converged, state-of-art NN model, we plot the top 10 logits for each test data in Figure 1. We observe that for inputs that NN models make correct predictions, the gap between their label logits and the following non-label logits can be narrow. In this situation, separating the correct label logits from the rest of top logits becomes challenging for AMIPS algorithms. The primary reason behind this is that these indexing approaches have limitations in dividing a cluster. Therefore, a learning approach is necessary to partition the prediction efficiently.

3.3 The Retrieval Oracle

Our goal is to construct the retrieval oracle introduced in Section1. We formulate the objective of the retrieval oracle as the construction of a **Perfect Retrieval Set**. For each input embedding (query), we retrieve a subset of neurons $\mathcal{S}$ such that $k = |\mathcal{S}|$ and $k << m$. k decides the amount of computation saving, as only k number of neurons will be activated instead of m. In the retrieval process, we want to maximize the probability of retrieving label neurons. Moreover, label neurons should have the highest inner products within the subset. Formally,

Definition 1 (Perfect Retrieval Set)**.** *Given a large output layer with neurons $\mathcal{C} = \{c_i \mid i = 1, 2, ...m\}$, for each input embedding q, with labels Y in the multi-label setting, we want to retrieve a subset of neurons $\mathcal{S} \subset \mathcal{C}$ with $|\mathcal{S}| \ll m$ such that,*

$$\underset{c_j \in \mathcal{S}}{\operatorname{argmax}}\, q^T w_j + b_j \in Y.$$

3.4 Algorithm Overview

To construct the ideal retrieval oracle, which returns a Perfect Retrieval Set, we introduce *HALOS*. *HALOS* exploits Locality Sensitive Hashing (LSH) along with an efficient hash function learning procedure to approximate the computation in large output layer using label information from training dataset. Note that we choose a particular variant of LSH called SimHash (Charikar, 2002), which is parameterized by hyperplanes in the dimensionality of the query. The full workflow is illustrated in Figure 2.

Algorithm 1 Offline Preprocessing

1: **Input:** Q**,** Y**,** W**,** HT, H**,** t_1, t_2
2: $P_+ = \{\}$, $P_- = \{\}$
3: **for** $i = 1 : N$ **do**
4: Compute $H(q_i)$.
5: S = $\{\}$
6: **for** $l = 1 : L$ **do**
7: $S = S\cup$ **Query**$(H_l(q_i), HT_l)$
8: **end for**
9: $P_+ = P_+ \cup \{(q_i, w_{y_i}) | y_i \notin S, q_i^T w_{y_i} > t_1\}$
10: $P_- = P_- \cup \{(q_i, w_j) | w_j \in S \setminus y_i, q_i^T w_j < t_2\}$
11: **end for**
12: **shuffle** P_+, P_-
13: $g = min(|P_+|, |P_-|)$
14: $H^{'} \leftarrow$ **IUL**$(H, Pair_p[: g], Pair_n[: m])$
15: $HT^{'} \leftarrow$ **Rebuild**$(HT, H^{'})$
16: **Return** $HT^{'}$, $H^{'}$

Algorithm 2 Online Inference

1: **Input:** q**,** W**,** b**,** $HT^{'}$**,** $H^{'}$
2: Compute $H^{'}(q)$
3: S = $\{\}$
4: **for** $l = 1 : L$ **do**
5: $S = S\cup$ **Query**$(H_l^{'}(q), HT_t^{'})$
6: **end for**
7: **Return** $\operatorname{argmax} qW_S^T$

HALOS works in two separate stages: an offline preprocessing stage (Section 3.5) and an online inference stage (Section 3.6). We summarize the *offline* preprocessing as the following three steps: (1) Given any trained NN, we index last layer weights in hash tables(HT) via hash functions (H), which are randomly generated hyperplanes. (2)We leverage the NN training data to iteratively update H using **Index Updating Loss**(introduce in Section 3.5) based on the neurons retrieved from HT by each training data embedding query. (3) We rebuilt HT using new hash functions and store HT for online inference stage. We summarize the *online* inference phase as the following two steps: (1) Given input from the test set, we compute the forward pass until the output layer and get an embedding q. Then, we query the hash tables with q. (2) We set retrieved neurons as "active", and all other neurons as "inactive" for this input. Finally, we perform matrix multiplication on the "active" neurons and the top-ranked neurons (with highest logits) are returned as the prediction.

3.5 Offline Preprocessing: Index Output Layer Neurons

Following the standard LSH scheme, we construct L hash tables and each hash table has a capacity of 2^K bucket,

where K is the number of binary hash functions per table (Each hash table key is concatenated by the K binary codes together). In total, we have $K \times L$ hash functions. Each neuron c_i can be represented by the concatenation of its weight and bias parameters $c_i = [w_i, b_i]$. For each $[w_i, b_i]$, we generate $K \times L$ hash codes, which constitute L hash table keys, and insert neuron's index i into L hash tables.

For each data in the training set, we collect its output from last hidden layer as embedding q and $[q, 1]$ is used as the embedding query to account for bias. For each embedding query, we generate L hash table keys and retrieve neuron indexes from L hash tables. For simplicity, we omit $[w, b]$, and $[q, 1]$ and directly use w and q in the following sections.

Every binary hash code for an input x is generated by function $f(x) = sign(\theta^T x)$, where θ is drawn i.i.d. from $\mathcal{N}(0, 1)$. This is equivalent to the method used in Simhash (Charikar, 2002). Geometrically, each θ represents a projection hyperplane in $\mathcal{R}^{d+1}$, such that the space is partitioned by K hyperplanes.

In order to possess the properties of a Perfect Retrieval Set, the hash functions should have the following properties: (1) the collision probability between the input embedding query and its ground truth label neuron is high, (2) the collision probability between the input embedding query and its non-label neurons is low (3) neurons are distributed evenly over all buckets for better load-balancing, which leads towards lower query overhead for efficiency purpose. We formally define our ideal hash function as the following:

Definition 2 (**Label Sensitive Hash Family).** *A hash family $\mathcal{H}$ is called $(1, 0, p1, p2)$-sensitive if for a triplet $(q, x, y) \in \left(\mathbb{R}^{d+1} \times \mathbb{R}^{d+1} \times \{0, 1\}\right)$, a hash function h chosen uniformly from $\mathcal{H}$ satisfies:*

- *if $y = 1$ then $Pr(h(q) = h(x)) \geq p_1$*
- *if $y = 0$ then $Pr(h(q) = h(x)) \leq p_2$*

We approximate hash functions from such a family using an iterative learning mechanism, which encourages the above three properties through **Index Updating Loss (IUL)** on pairwise data. Inspired by contrastive learning (Chen et al., 2020d), the key to this learning process is the collection of positive and negative pairwise training samples. For each data embedding q from the NN training dataset, we retrieve its corresponding set of neurons $\mathcal{S}$ from the existing L hash tables. Then, pairwise training samples are collected according to the following criterion: (1)Positive pair consists of the input embedding, and its label class neuron which was not retrieved from current hash table. (2) Negative pair consists of the input embedding, and a retrieved, non-label neuron. Formally,

- Positive Pair $P_+ = (q, w_i)$
  if $c_i \in C \setminus S$ and $c_i \in Y$ and $q^T w_i > t_1$
- Negative Pair $P_- = (q, w_i)$
  if $c_i \in S$ and $c_i \notin Y$ and $q^T w_i < t_2$

Index Update Loss (IUL): IUL is based on the classic triplet loss (Chechik et al., 2010) and we customize it specifically for updating hash table bucket assignment. The intuition behind IUL is that positive pairs are encouraged to land in the same bucket while negative pairs are allocated towards different buckets. We use Hamming distance as an approximation of the difference between the hash codes. Since $sign$ is a discrete function, we use $\tanh$ as a differentiable approximation. We know that $\text{dist}_{\text{hamming}}(q, w) = \frac{1}{2}(K - q^T w)$ (Cao et al., 2017).Formally,

$$\begin{aligned} \mathcal{IUL}(P_+, P_-) = &\sum_{q_i, w_i \in P_+} -log(\sigma(\mathcal{K}(w_i)^T \mathcal{K}(q_i))) \\ &- \sum_{q_j, w_j \in P_-} log(1 - \sigma(\mathcal{K}(w_j)^T \mathcal{K}(q_j))), \end{aligned}$$

where $\mathcal{K}(w) = \tanh(\theta^T w)$, $\mathcal{K}(q) = \tanh(\theta^T q)$, and $\sigma(x) = 1/1 + e^{-x}$.

Positive samples are used to maximize the probabilities of retrieving correct label neurons. Concurrently, it increases the relative ranking of label neurons on the inner product by decreasing the probabilities of retrieving other non-essential neurons. Negative samples are used to maintain a relatively small bucket size by pushing out low inner product neurons from the bucket. Otherwise, all the neurons would ultimately converge to the same bucket in each table. We collect the pairwise training data based on each retrieved set because it directly reflects the circumstances on how last layer neurons are separated by the current hyperplanes. t_1 and t_2 are two inner product ranking thresholds, that control the inner product quality of positive and negative pairs. Usually, we have $t_1 > t_2$ in any valid settings. Otherwise, in the situation that the NN predict a certain label neuron with a small logit, due to the nature of LSH, it would be challenging to train hash functions in the manner that low inner product neurons are retrieved while high inner product neurons are excluded.

Difference from Standard Learning for AMIPS: The positive and negative pair construction is essential. It should be observed that standard learning approaches, focused on AMIPS objective and use every positive and negative pair for training the hash function, potentially solving a harder problem. Instead, we only use the negative pairs arising from the buckets, and positive pairs missed by buckets if they are the correct labels. Overall, our training is aware of the current retrieval mechanism and only enforces what is needed for classification.

3.6 Online Inference

In this section, we introduce the online inference process. The NN parameters, hash functions, and hash tables from processing stages are frozen for deployment. For each test data, forward computation until the last layer is computed and the input embedding from last hidden layer is used for hash table query. Specifically, the $K \times L$ hash codes of the input embedding are generated using the stored hash functions. Based the hashcodes, a subset of output layer neurons is retrieved from the store hashed table. Here, we have a smaller output layer consisted of only retrieved neurons. Specifically, instead of performing the full computation over the output layer, we only compute the logits for each retrieved class, and the highest probability class among all retrieved classes are returned as the prediction. Note that unlike pruning, which set weights as zeros, we construct a much smaller output layer weight matrix and perform a smaller matrix multiplication.

Analysis on Computation Efficiency We focus on the output layer computation for the discussion here as computation before the output layer are not changed. With regular NN inference, the computation at output layer can be expressed as $\sigma(qW^T + b)$. Output layer weight matrix W is of dimension $m \times d$ where m is the size of output layer. With *HALOS* inference, the weight matrix is of dimension $k \times d$ where k is the number of neurons retrieved. The computation overhead *HALOS* introduced is the cost to compute $K \times L$ hash codes and hash table look up, which are known to be cheap. *HALOS* reduce inference cost by computing a much smaller matrix multiplication and nonlinear function, and usually, we have $\frac{k}{m} \leq 5\%$.

4 Evaluation

In this section, we evaluate the effectiveness of *HALOS* method for efficient inference with large output spaces on two large scale extreme classification and two language modeling datasets. Specifically, we investigate from the following three perspectives (1) How does *HALOS* perform compared with established efficient inference baselines in terms of the accuracy and efficiency trade-off? (2) How does *HALOS* perform in terms of optimizing for NN accuracy compared with AMIPS baselines? (3) How does *HALOS* perform compared to AMIPS baselines in terms of both query efficiency and inference efficiency?

Datasets and NN Models As we focus on NN with large output space, we choose two applications with such characteristics: language modeling and extreme classification, which concerns problems in webpage and production categorization, as well as webpage-to-webpage and product-to-product recommendation tasks (Bhatia et al., 2016). For extreme classification, we use a standard fully connected neural network with one hidden layer of size 128. We evaluate on two datasets: Wiki10-31K (Zubiaga, 2012) and Delicious200K (Wetzker et al., 2008). For language modeling, we use a standard fully connected network with one hidden layer of size 128 for Text8 (Mahoney, 2011). A two-layer LSTM network with hidden dimension of 200 is used for Wiki-Text2 (Merity et al., 2016),denotes as Wiki2-LSTM in the results. For mask language modeling, a 6 layer, 4 head Transformer model for Wiki-Text2,(denoted as Wiki2-Trans) and PTB (Mikolov & Zweig, 2012). The datasets are under MIT license. We refer the readers to Section B.2 for more details.

Baselines: We compare *HALOS* with following approaches:(1) **Full** is the regular and paralleled NN inference using all neurons in NN last layer. (2) **SLIDE** (Chen et al., 2020b) is a deep learning system utilizing locality-sensitive hashing for faster training, written in C++. We implement this method for inference. (3) **Graph Decoder** (GD) is an AMIPS method proposed for efficient Softmax inference in (Zhang et al., 2018) that combines the asymmetric transform in (Bachrach et al., 2014) with HNSW (Malkov & Yashunin, 2018). Here we use the original implementation of HNSW (Boytsov & Naidan, 2013) and pre-process the data according to (Zhang et al., 2018). (4) **ip-NSW** is a state-of-the-art graph-based AMIPS algorithm proposed in (Morozov & Babenko, 2018). It belongs to the direct MIPS category and shows performance improvement over GD. (5) **Product Quantization** (PQ) (Johnson et al., 2017) is an AMIPS solver with K-means and asymmetric transformation. We implement this method following the popular open-source ANNS platform from Facebook (Johnson et al., 2019). (6) **NeuralLSH** (Dong et al., 2019) is a LSH based indexing with partition learning from the k-NN graph.

Implementation and Experiment Setting: Following standard NN inference procedure, we set test batch size as 1. All the experiments are conducted on a machine equipped with two 20-core/40-thread processors (Intel Xeon(R) E5-2698 v4 2.20GHz). The machine is installed with Ubuntu 16.04.5 LTS. HALOS for the output layer is written in C++ and compiled under GCC7 with OpenMP. The **Full** inference baseline is implemented in PyTorch. **GD**, **ip-NSW**, **PQ** are implemented in C++ with OpenMP. All implementation is parallelized with multi-threading with full usage of CPU cores. All baselines report the best results after a hyperparameter search.

Energy Measurement We further investigate efficiency from the energy consumption perspective. We measure the energy usage of each method using a monitoring tool. Command line utility tools, including *s-tui*, were used to monitor the CPU power consumption, in Watts (Joules / second), over the inference times for each dataset and each method, in intervals of 1 second. The base power consump-

Method	Dataset	Accuracy	Label Recall	Avg.Time(ms)	Avg.Energy(10^{-3} J)
Full	Wiki2-LSTM	0.4044	1	0.63	9.31
HALOS.	Wiki-2LSTM	**0.4265**	**1**	**0.36 (1.7x)**	**3.20 (2.9x)**
PQ	Wiki2-LSTM	0.2234	0.6654	10.57	128.76
ip-NSW	Wiki2-LSTM	0.3995	0.8705	1.60	23.92
GD	Wiki2-LSTM	0.1369	0.9215	1.76	27.07
NeuralLSH	Wiki2-LSTM	0.2299	0.8826	1.99	39.98
SLIDE	Wiki2-LSTM	0.3309	0.8695	3.33	45.54
Full	Text8	0.9129	1	1.88	31.99
HALOS.	Text8	**0.9132**	**1**	**0.56 (3.3x)**	**4.98 (6.4x)**
PQ	Text8	0.6138	0.6084	12.99	100.26
ip-NSW	Text8	0.8299	0.8977	2.07	22.66
GD	Text8	0.9129	0.9908	2.09	20.40
NeuralLSH	Text8	0.8990	0.9011	2.55	28.44
SLIDE	Text8	0.6517	0.9799	3.92	29.33
Full	Wiki2-Trans	0.8538	1	5.66	8.77
HALOS.	Wiki2-Trans	**0.8519**	**0.9993**	**0.28(20x)**	**1.66(5.28x)**
PQ	Wiki2-Trans	0.7631	0.8566	7.02	10.33
ip-NSW	Wiki2-Trans	0.8164	0.9661	0.37	4.02
GD	Wiki2-Trans	0.8221	0.9787	0.39	3.94
NeuralLSH	Wiki2-Trans	0.8300	0.9804	0.36	3.98
SLIDE	Wiki2-Trans	0.8056	0.9511	0.49	2.33
Full	PTB	0.8534	1	0.55	7.42
HALOS.	PTB	**0.8531**	**0.9997**	**0.32(1.7x)**	**1.33(5.6x)**
PQ	PTB	0.7442	0.8777	2.67	8.98
ip-NSW	PTB	0.8325	0.9903	0.52	2.34
GD	PTB	0.8401	0.9908	0.54	2.19
NeuralLSH	PTB	0.8231	0.9775	0.51	2.47
SLIDE	PTB	0.7799	0.8994	0.60	2.66
Full	Delicious200K	0.4391	1	4.16	71.34
HALOS	Delicious200K	0.4245	**0.889**	**0.81 (5.1x)**	**8.7 (8.2x)**
PQ	Delicious200K	0.3547	0.6677	9.33	113.22
ip-NSW	Delicious200K	0.4122	0.6900	2.66	31.66
GD	Delicious200K	**0.4362**	0.7000	2.29	29.05
NeuralLSH	Delicious200K	0.3928	0.6450	3.33	30.21
SLIDE	Delicious200K	0.4024	0.8542984	5.12	37.75
Full	Wiki10-30K	0.8232	1	0.76	10.69
HALOS.	Wiki10-30K	0.8018	0.9779	**0.39 (1.9x)**	**3.53 (3.0x)**
PQ	Wiki10-30K	0.6766	0.8905	4.06	39.28
ip-NSW	Wiki10-30K	0.7703	0.8854	1.65	15.75
GD	Wiki10-30K	0.7636	0.9163	1.69	15.80
NeuralLSH	Wiki10-30K	0.7001	0.9000	0.99	19.77
SLIDE	Wiki10-30K	**0.8079**	**0.9995**	6.22	25.45

Table 1: This table summarizes the performance of HALOS and other baselines. Compared to full inference, HALOS achieves up to 21× time reduction and 8× energy reduction. HALOS enjoys the highest label recall rate on most datasets. Meanwhile, HALOS outperforms all AMIPS baselines on both accuracy, time and energy usage on most datasets.

tion would be subtracted from the average power of the inference time, in order to measure and compare the energy expenditure of only the inference step of each method.

Evaluation Metric: We compare HALOS against other baselines on the following evaluation metrics: (1) **p@k** for top k accuracy classification tasks. (2) **Label recall** is the times of retrieval set including label neurons divided by total number of testing data.(3) **Time** is measured as the average wall-clock time for passing one testing data through the last layer in milliseconds. (4) **Energy**, measured in Joules, is the average CPU power over the inference period, multiplied

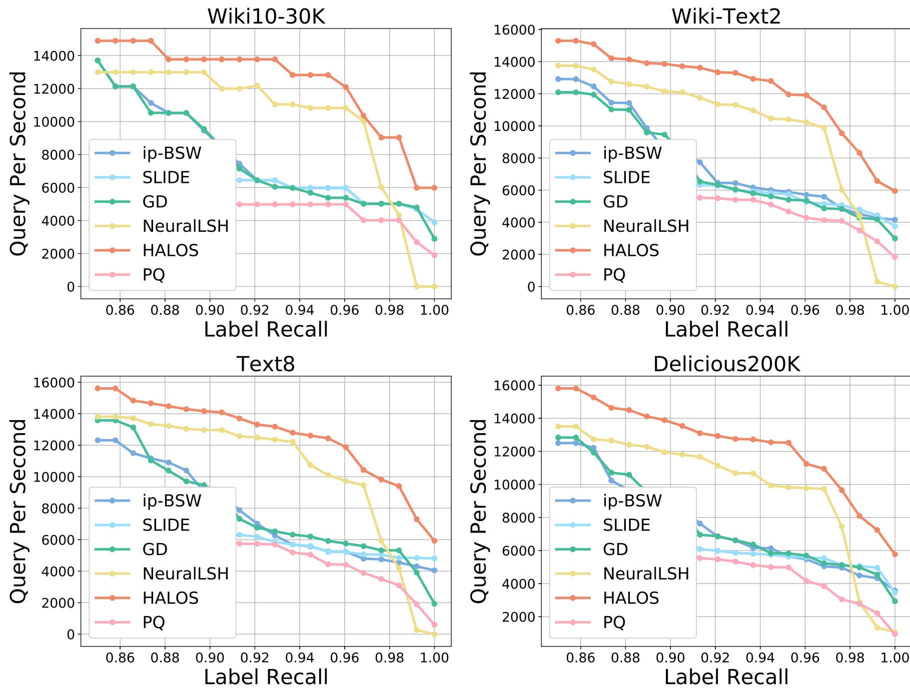

Figure 3: Plot of Label Recall versus Query Per Second. On all four dataset, the line for *HALOS*(in red) is higher than all AMIPS baselines. Specifically, at every query speed, *HALOS* recalls more label neurons comparing to AMIPS baselines. This validates our arguments that naively applying AMIPS on the output layer is not sensitive to the label class, which may hurt model accuracy as shown in Table 1. Wiki-Text2 refers to the LSTN network.

by the inference time. It is then averaged over each testing data.

4.1 Main Result

Table 1 summarize inference accuracy, wall-clock time and energy usage for HALOS and all other baselines. We report the performance in Table 1 when each method achieves maximum retrieval efficiency, denoted as the ratio between $p@1$ and time usage. This selection criterion is chosen to reflect the balance between accuracy and efficiency. We observe that: (1) Comparing to full inference, HALOS achieves up to $20\times$ reduction in time and $8.2\times$ reduction in energy consumption with no or minor accuracy drops. Note that on Text8 and Wiki2-LSTM, HALOS achieves both higher than full-inference accuracy and significant time and energy reduction.(2) HALOS achieves the highest label recall compared to all other baselines on most datasets. (3) HALOS achieves the highest accuracy, lowest time, and energy usage compared to other AMIPS baselines.

Discussion: (Zhang et al., 2018; Chen et al., 2018) compared the performances of different methods under a single CPU thread setting, which is not a practical simulation for real-world cloud systems. Moreover, despite their decent performance on a single thread, methods such as ip-NSW or GD are ill-suited to exploit the full parallelization offered by multi-core CPUs and tend to have a large number of irregular memory accesses. This limitation significantly degrades their performance even compared to exact MIPS computation on CPU with the current deep learning framework such as PyTorch.

4.2 Study on *HALOS* for Label Sensitivity

Both *HALOS* and baselines have tuning parameters that control the accuracy-efficiency trade-offs. Figure 3 plots Label Recall vs. Query Per Second(QPS) for *HALOS* and AMIPS baselines. At every QPS level, *HALOS* retrieves more label neurons. Meanwhile, from Table 1, AMIPS baselines, which target on the largest inner product, have a lower label recall rate on all datasets. This phenomenon validates our arguments regarding the shortcomings of applying AMIPS methods in NN Inference. AMIPS methods struggle in retrieving the label neuron at a given time budget. And failing to retrieve the labels directly leads to incorrect prediction.

4.3 Study on *HALOS* for Efficiency

The major overhead introduced by *HALOS* in inference efficiency lies in the query time. Large overhead may cancel out the computation savings from smaller matrix multiplication and even result in longer than regular inference time, as we observe in Table 1 for some baselines. In this section, we

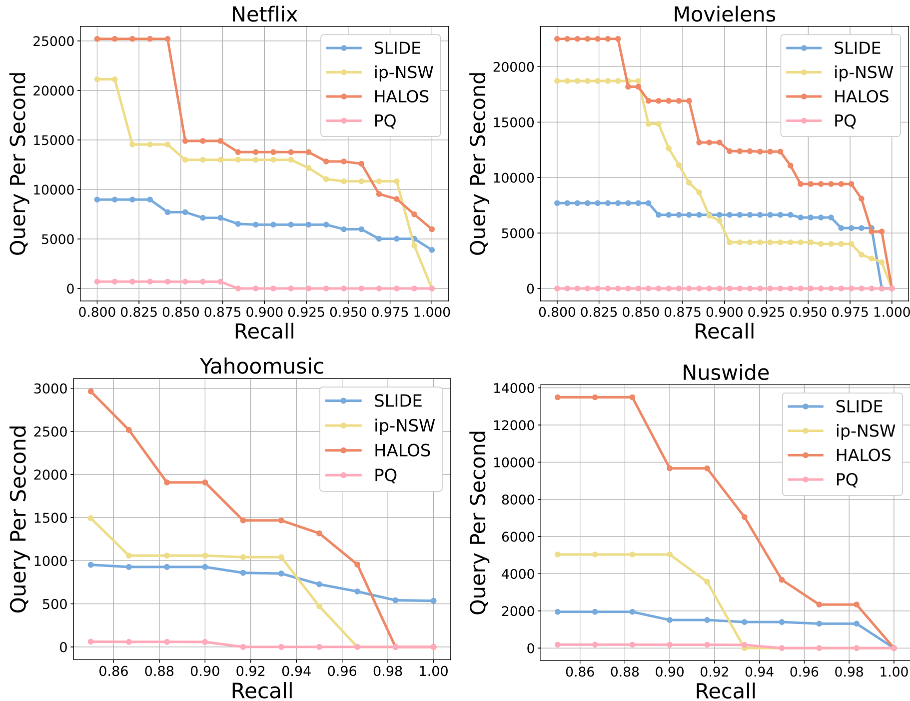

Figure 4: Plot of Recall versus Query Per Second. The lines representing *HALOS*(red) is the highest across datasets. At every recall level, *HALOS* processes more data, leading to better query efficiency comparing to AMIPS baselines.

purely focus on *HALOS*'s query efficiency in perfect model settings, where the MIPS results correspond to the label.

Setting: We use four recommendation datasets and the task is to directly retrieving the item that yields the largest inner product between user and item embedding. To generate user and item embedding, we use deep matrix factorization (DMF) (Xue et al., 2017) for Movielens and Netflix dataset, and alternating least squares (ALS) based MF method (Hu et al., 2008) for Yahoomusic and Nuswide dataset. Then, the preference of an user towards an item is described by their inner product in the embedding space.

Analysis: Figure 4 shows the standard Recall vs. Query Per Second(QPS) plots. The line for *HALOS* lies consistently above other baselines, which means *HALOS* processes more data at every recall level. *HALOS* outperforms RANDOM LSH on all datasets, which validates our arguments that learned hashing functions could better adapt to data distribution and improve query efficiency. *HALOS* also achieves up to $4\times$ QPS compared to IP-NSW and PQ, which validates our choice of hashing tables as indexing data structures. Despite lower time complexity during the preprocessing phase, hash table look-up operations are faster than the greedy walk on tree or graph structures in the query phase. Moreover, hashing methods are more amenable towards less computation and multi-threading (Wang et al., 2018) compared to MIPS solvers (Guo et al., 2016) and graph methods (Morozov & Babenko, 2018; Zhang et al., 2018).

One common concern about the hash table is its memory usage. However, since we only store neuron indices, memory usage is limited. For example, the output dimension of Delicious-200K is 205,443. Each integer cost 4 bytes, and assuming we set L as 10 (same parameters as our main result), the total memory usage is only 8MB.

4.4 Ablation Studies on *HALOS*

Collision Probability: Collision probability denotes the probability that a pair of inputs are hashed to the same bucket. Figure 5 shows the collision probability for both positive and negative pairs we collect during hash function training. On Text8 and Delicious200K, collision probability between positive pairs increases and converges to a level above 0.9. Meanwhile, collision probability between negative pairs decreases throughout the training process. On Delicious200K, the collision probability of negative pairs converges to around 0.1. On Text8, it converges close to zero. These plots confirm that the proposed hash functions learning mechanism adjusts the randomized initialized hyperplanes to index input embedding and its label neurons into the same bucket with a much higher probability. This observation explains *HALOS*'s high label recall in Table 1.
Influence of Hyper Parameters: K, the number of hashes, and L, the number of hash tables, are two key parameters to balance *HALOS*'s retrieval quality and efficiency. In general, fewer hash tables improve efficiency by reducing query time

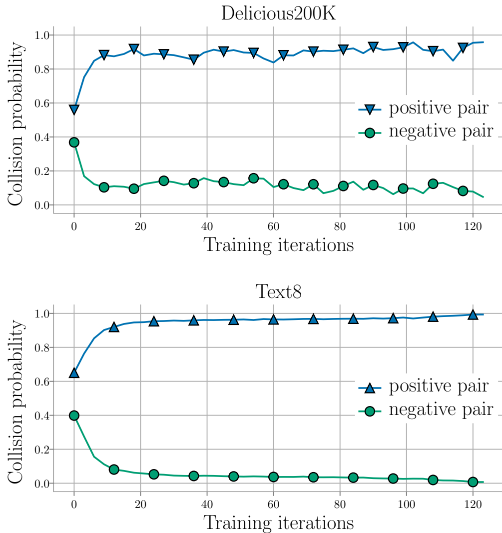

Figure 5: Collision probability is measured as the probability a pair of inputs hashed in the same bucket. Blue line plots the collision probability between positive pairs. Green line plots the collision probability between negative pairs. The starting point represents the probabilities with randomized initialized hashing functions. It is evident that the proposed learning mechanism pushes positive pairs to land in the same bucket while seperates negative pairs. And since positive pairs are collected as input embedding with its label neurons, *HALOS* is more sensitive to label class and achieve high label recall.

		K = 4	K = 6	K = 8
L = 1	P@1	0.4245	NA	NA
	P@5	0.3473	NA	NA
	Retrieval Size	424	0	0
L = 10	P@1	0.4602	0.4488	0.4408
	P@5	0.3676	0.3733	0.3598
	Retrieval Size	2560	875.53	153.31
L = 50	P@1	0.4405	0.4455	0.4457
	P@5	0.3659	0.3599	0.3615
	Retrieval Size	15568	2122.47	360

Table 2: Effect of L and K on $P@1$ and $P@5$ on Delicious-200K. K is the number of hash functions, L is the number of hash functions. Retrieval Size measures the number of output layer neurons retrieved from hash tables.

Dataset	Wiki10-31k	Delicious200k	Text8	Wiki2-LSTM
HALOS p@1	0.82	**0.46**	0.913	**0.427**
HALOS p@5	0.48	**0.37**	0.740	**0.084**
Retrieval Size	2372	2560	965	3071
Full p@1	0.82	0.44	0.913	0.404
Full p@5	0.57	0.36	0.740	0.077

Table 3: This table summarizes the highest accuracy of *HALOS* on four dataset. Bold indicates that *HALOS* is higher than the full accuracy shown in Table 1.

and retrieval size, while more hash table improves retrieval quality and help with accuracy. Here retrieval size represents the average number of items returned by the hash tables in each query. Fewer hashes decrease hash code computation but increase retrieval size(including unnecessary neurons). More hashes increase hash code computation but decrease retrieval size(missing important neurons with high probability). In Table 2, we summarize the effects of varying K and L on Delicious200K. $L = 1$ and $K = 4$ (parameters for our main result) have the lowest accuracy but lead to the smallest overhead and the most computation reduction: it requires less hash code computations (determined by K), fewer hash table lookups (determined by L), and smaller last layer matrix multiplication (determined by retrieval size). Different choices of K and L have limited effect on $P@1$ and $P@5$((up to 3.5% on $P@1$ and 1.3% on $P@5$).

4.5 Study on *HALOS* for Accuracy

Our hashing learning mechanism was designed for the purpose of constructing the Perfect Retrievel Set as described in SectionSection 3.4. In this section, we deviates from focusing on efficiency, purely investigate the possibility of achieving higher accuracy than regular Softmax inference.

Table 3 summarizes the highest accuracy *HALOS* achieves during our hyper parameter search. On Delicious200K and Wiki-Text2 for language modeling, *HALOS* outperforms full inference accuracy by up to 2%. The major reason behind this increase is that the probability of predicting the correct class from a subset of neurons is naturally higher than the probability of predicting the correct class from the entire output space. Specifically, we observe a larger retrieval size to surpass regular inference accuracy. A larger retrieval size along with the query overhead may not lead to efficiency gain but may be of the separate interest for accuracy.

5 Conclusion

In this paper, we introduce *HALOS*, a hashing based approach that performs energy and time efficient inference on a large output layer neural networks. We show a novel problem formulation that identifies and bridges the gap between efficient inference and approximate maximum inner product search (AMIPS). We propose a combination of hashing based data structure and hyperplane learning objectives that dynamically adapts the index functions to reduce the retrieval size while preserve the prediction accuracy. We show that *HALOS* substantially outperforms other AMIPS baselines on both accuracy and efficiency metrics, with up to 87% energy reduction and up to 21x speedup compared to ideally paralleled full inference.

ACKNOWLEDGEMENTS

This work was supported by National Science Foundation IIS-1652131, BIGDATA-1838177, AFOSR-YIP FA9550-18-1-0152, ONR DURIP Grant, the ONR BRC grant on Randomized Numerical Linear Algebra, and gift grants from Intel and VMware. Zhaozhuo Xu was supported by Ken Kennedy Institute BP fellowship.

REFERENCES

Bachrach, Y., Finkelstein, Y., Gilad-Bachrach, R., Katzir, L., Koenigstein, N., Nice, N., and Paquet, U. Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces. In *Proceedings of the 8th ACM Conference on Recommender systems*, pp. 257–264, 2014.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

Bhatia, K., Dahiya, K., Jain, H., Mittal, A., Prabhu, Y., and Varma, M. The extreme classification repository: Multi-label datasets and code, 2016. URL http://manikvarma.org/downloads/XC/XMLRepository.html.

Boytsov, L. and Naidan, B. Engineering efficient and effective non-metric space library. In Brisaboa, N. R., Pedreira, O., and Zezula, P. (eds.), *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings*, volume 8199 of *Lecture Notes in Computer Science*, pp. 280–293. Springer, 2013. doi: 10.1007/978-3-642-41062-8_28. URL https://doi.org/10.1007/978-3-642-41062-8_28.

Cao, Z., Long, M., Wang, J., and Yu, P. S. Hashnet: Deep learning to hash by continuation. *CoRR*, abs/1702.00758, 2017. URL http://arxiv.org/abs/1702.00758.

Charikar, M. S. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pp. 380–388. ACM, 2002.

Chechik, G., Sharma, V., Shalit, U., and Bengio, S. Large scale online learning of image similarity through ranking. *Journal of Machine Learning Research (JMLR)*, 11(3), 2010.

Chen, B., Xu, Y., and Shrivastava, A. Lsh-sampling breaks the computation chicken-and-egg loop in adaptive stochastic gradient estimation. *arXiv preprint arXiv:1910.14162*, 2019a.

Chen, B., Xu, Y., and Shrivastava, A. Fast and accurate stochastic gradient estimation. In *NeurIPS*, 2019b.

Chen, B., Liu, Z., Peng, B., Xu, Z., Li, J. L., Dao, T., Song, Z., Shrivastava, A., and Re, C. Mongoose: A learnable lsh framework for efficient neural network training. In *International Conference on Learning Representations*, 2020a.

Chen, B., Medini, T., and Shrivastava, A. Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. In *Proceedings of the 3rd Conference on Systems and Machine Learning (MLSys)*, 2020b.

Chen, B., Liu, Z., Peng, B., Xu, Z., Li, J. L., Dao, T., Song, Z., Shrivastava, A., and Re, C. {MONGOOSE}: A learnable {lsh} framework for efficient neural network training. In *International Conference on Learning Representations*, 2021a. URL https://openreview.net/forum?id=wWK7yXkULyh.

Chen, N., Liu, F., You, C., Zhou, P., and Zou, Y. Adaptive bi-directional attention: Exploring multi-granularity representations for machine reading comprehension. In *ICASSP*, 2020c.

Chen, N., You, C., and Zou, Y. Self-supervised dialogue learning for spoken conversational question answering. In *INTERSPEECH*, 2021b.

Chen, P. H., Si, S., Kumar, S., Li, Y., and Hsieh, C.-J. Learning to screen for fast softmax inference on large vocabulary neural networks. *arXiv preprint arXiv:1810.12406*, 2018.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pp. 1597–1607. PMLR, 2020d.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Dong, Y., Indyk, P., Razenshteyn, I., and Wagner, T. Learning space partitions for nearest neighbor search. *arXiv preprint arXiv:1901.08544*, 2019.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Fan, M., Guo, J., Zhu, S., Miao, S., Sun, M., and Li, P. Mobius: Towards the next generation of query-ad matching

in baidu's sponsored search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2509–2517, 2019.

Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., and Kumar, S. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, pp. 3887–3896. PMLR, 2020.

Guo, R. et al. Quantization based fast inner product search. In *Artificial Intelligence and Statistics*, pp. 482–490, 2016.

Han, K., Wang, Y., Chen, H., Chen, X., Guo, J., Liu, Z., Tang, Y., Xiao, A., Xu, C., Xu, Y., et al. A survey on visual transformer. *arXiv e-prints*, pp. arXiv–2012, 2020.

Hu, Y., Koren, Y., and Volinsky, C. Collaborative filtering for implicit feedback datasets. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pp. 263–272. Ieee, 2008.

Indyk, P. and Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

Indyk, P. and Woodruff, D. Polylogarithmic private approximations and efficient matching. In *Theory of Cryptography Conference*, pp. 245–264. Springer, 2006.

Jegou, H., Douze, M., and Schmid, C. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

Johnson, J., Douze, M., and Jégou, H. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

Johnson, J., Douze, M., and Jégou, H. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

Levy, D., Chen, D., and Ermon, S. Lsh softmax: Sublinear learning and inference of the softmax layer in deep architectures. 2018.

Li, P., Shrivastava, A., Moore, J. L., and König, A. C. Hashing algorithms for large-scale learning. In *Advances in neural information processing systems*, pp. 2672–2680, 2011.

Li, P., Owen, A., and Zhang, C.-H. One permutation hashing for efficient search and learning. *arXiv preprint arXiv:1208.1259*, 2012.

Liu, F., You, C., Wu, X., Ge, S., Sun, X., et al. Auto-encoding knowledge graph for unsupervised medical report generation. *Advances in Neural Information Processing Systems*, 34, 2021.

Mahoney, M. Text8. http://mattmahoney.net/dc/textdata.html, 2011.

Malkov, Y., Ponomarenko, A., Logvinov, A., and Krylov, V. Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces. In *International Conference on Similarity Search and Applications*, pp. 132–147. Springer, 2012.

Malkov, Y., Ponomarenko, A., Logvinov, A., and Krylov, V. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45: 61–68, 2014.

Malkov, Y. A. and Yashunin, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

Medsker, L. R. and Jain, L. Recurrent neural networks. *Design and Applications*, 5:64–67, 2001.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *CoRR*, abs/1609.07843, 2016. URL http://arxiv.org/abs/1609.07843.

Mikolov, T. and Zweig, G. Context dependent recurrent neural network language model. In *2012 IEEE Spoken Language Technology Workshop (SLT)*, pp. 234–239. IEEE, 2012.

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Morozov, S. and Babenko, A. Non-metric similarity graphs for maximum inner product search. In *Advances in Neural Information Processing Systems*, pp. 4726–4735, 2018.

Shen, F., Liu, W., Zhang, S., Yang, Y., and Tao Shen, H. Learning binary codes for maximum inner product search. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4148–4156, 2015.

Shim, K., Lee, M., Choi, I., Boo, Y., and Sung, W. Svd-softmax: Fast softmax approximation on large vocabulary neural networks. In *Advances in Neural Information Processing Systems*, pp. 5463–5473, 2017.

Shrivastava, A. and Li, P. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). In *Advances in Neural Information Processing Systems*, pp. 2321–2329, 2014a.

Shrivastava, A. and Li, P. In defense of minhash over simhash. In *Artificial Intelligence and Statistics*, pp. 886–894, 2014b.

Shrivastava, A. and Li, P. Improved asymmetric locality sensitive hashing (alsh) for maximum inner product search (mips). In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, UAI'15, pp. 812–821, Arlington, Virginia, USA, 2015. AUAI Press. ISBN 9780996643108.

Spring, R. and Shrivastava, A. A new unbiased and efficient class of lsh-based samplers and estimators for partition function computation in log-linear models. *arXiv preprint arXiv:1703.05160*, 2017.

Spring, R. and Shrivastava, A. Mutual information estimation using lsh sampling. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence, AAAI Press*, 2020.

Tan, S., Zhou, Z., Xu, Z., and Li, P. On efficient retrieval of top similarity vectors. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 5239–5249, 2019.

Vijayanarasimhan, S., Shlens, J., Monga, R., and Yagnik, J. Deep networks with large output spaces. *arXiv preprint arXiv:1412.7479*, 2014.

Wang, Y., Shrivastava, A., Wang, J., and Ryu, J. Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, 2018. ISBN 9781450347037. doi: 10.1145/3183713.3196925. URL https://doi.org/10.1145/3183713.3196925.

Wetzker, R., Zimmermann, C., and Bauckhage, C. Analyzing social bookmarking systems: A del.icio.us cookbook. *Proceedings of the ECAI 2008 Mining Social Data Workshop*, pp. 26–30, 01 2008.

Wu, X., Guo, R., Suresh, A. T., Kumar, S., Holtmann-Rice, D. N., Simcha, D., and Yu, F. Multiscale quantization for fast similarity search. In *Advances in Neural Information Processing Systems*, pp. 5745–5755, 2017.

Xu, J., He, X., and Li, H. Deep learning for matching in search and recommendation. In *WWW Tutorials*, 2018.

Xu, Z., Chen, B., Li, C., Liu, W., Song, L., Lin, Y., and Shrivastava, A. Locality sensitive teaching. *Advances in Neural Information Processing Systems*, 34, 2021a.

Xu, Z., Song, Z., and Shrivastava, A. Breaking the linear iteration cost barrier for some well-known conditional gradient methods using maxip data-structures. *Advances in Neural Information Processing Systems*, 34, 2021b.

Xue, H.-J., Dai, X., Zhang, J., Huang, S., and Chen, J. Deep matrix factorization models for recommender systems. In *IJCAI*, pp. 3203–3209, 2017.

You, C., Chen, N., Liu, F., Yang, D., and Zou, Y. Towards data distillation for end-to-end spoken conversational question answering. *arXiv preprint arXiv:2010.08923*, 2020.

You, C., Chen, N., and Zou, Y. Contextualized attention-based knowledge transfer for spoken conversational question answering. In *INTERSPEECH*, 2021a.

You, C., Chen, N., and Zou, Y. MRD-Net: Multi-Modal Residual Knowledge Distillation for Spoken Question Answering. In *IJCAI*, 2021b.

You, C., Chen, N., and Zou, Y. Self-supervised contrastive cross-modality representation learning for spoken question answering. In *Findings of the Association for Computational Linguistics: EMNLP*, 2021c.

You, C., Zhao, R., Liu, F., Chinchali, S., Topcu, U., Staib, L., and Duncan, J. S. Class-aware generative adversarial transformers for medical image segmentation. *arXiv preprint arXiv:2201.10737*, 2022.

Zhang, M., Wang, W., Liu, X., Gao, J., and He, Y. Navigating with graph representations for fast and scalable decoding of neural language models. In *Advances in Neural Information Processing Systems*, pp. 6308–6319, 2018.

Zhou, Z., Tan, S., Xu, Z., and Li, P. Möbius transformation for fast inner product search on graph. In *Advances in Neural Information Processing Systems*, pp. 8216–8227, 2019.

Zubiaga, A. Enhancing navigation on wikipedia with social tags. *CoRR*, abs/1202.5469, 2012. URL http://arxiv.org/abs/1202.5469.

A Related Literature

A.1 Maximum Inner Product Search for Efficient Inference

The application of deep neural network (NN) models in cloud services is usually associated with a Wide Output Layer (WOL). For deep recommendation models (Bhatia et al., 2016; Xue et al., 2017), the size of the WOL is equal to the number of items to be recommended. For language modeling, the size of WOL is equal to the vocabulary size. This large output space becomes the computation bottleneck, and for this paper, we specifically focus on inference efficiency, which is a major concern for deployment in a cloud computing setting.

To tackle this inefficiency, various methods focus on efficient retrieval of the Top-K logits generated by the NN model. Most of these methods can be categorized as an *Approximate Maximum Inner Product Search* (AMIPS) approaches. Formally, we aim to solve the following problem: given a set S containing all neurons in the WOL as high dimensional vectors and each input embedding to the output layer as query q, we aim to develop an efficient algorithm for computing.

$$w = \arg\max_{x \in S} x^\top q. \quad (1)$$

There are two main categories for efficient inference via AMIPS. The first category of methods aims to reduce the MIPS to classical approximate nearest neighbor search (ANNS) method (Shrivastava & Li, 2014a; 2015), which can be summarized as two steps: (1) pre-processing the data vector $x \in S$ to x^* and query vector q to q^* asymmetrically so that $x^T q \approx f(x^*, q^*)$. Here $f(x, y)$ is cosine distance or Euclidean distance. (2) perform ANNS via indexing structures such as quantization (Jegou et al., 2011), or small world graph (Malkov et al., 2012; 2014; Malkov & Yashunin, 2018). Another category of MIPS-based methods targets at directly performing inner product search without reduction. This direct inner product search can be performed via graph due to the flexibility of modifying the edge definitions (Morozov & Babenko, 2018; Zhou et al., 2019; Tan et al., 2019).

A.2 Locality Sensitive Hashing

In this section, we briefly describe the recent development of Locality Sensitive Hashing(LSH) (Indyk & Motwani, 1998; Indyk & Woodruff, 2006). The high-level idea of LSH is to place similar items into the same bucket of a hash table with high probability. In formal terms, we consider $\mathcal{H}$ as a family of hash functions that maps $\mathbb{R}^D$ to some set $\mathcal{S}$.

Definition 3 (LSH Family). *A family $\mathcal{H}$ is called (S_0, cS_0, p_1, p_2)-sensitive if for any two points $x, y \in \mathbb{R}^D$ and h chosen uniformly from $\mathcal{H}$ satisfies:*

- *if $Sim(x, y) \geq S_0$ then $Pr(h(x) = h(y)) \geq p_1$*
- *if $Sim(x, y) \leq cS_0$ then $Pr(h(x) = h(y)) \leq p_2$*

Typically, $p_1 > p_2$ and $c < 1$ is needed. Moreover, the algorithm uses two parameters, (K, L). We construct L independent hash tables from the collection. Each hash table has a meta-hash function H formed by concatenating K random independent hash functions. Given a query, we collect one bucket from each hash table and return the union of L buckets. Intuitively, the meta-hash function makes the buckets sparse and reduces the number of false positives, because only valid nearest-neighbor items are likely to match all K hash values for a given query. The union of the L buckets decreases the number of false negatives by increasing the number of potential buckets that could hold valid nearest-neighbor items. One sufficient condition for a hash family $\mathcal{H}$ to be a LSH family is that the *collision probability* $Pr_{\mathcal{H}}(h(x) = h(y))$ is a monotonically increasing function of the similarity, i.e.

$$Pr_{\mathcal{H}}(h(x) = h(y)) = f(Sim(x, y)), \quad (2)$$

where f is a monotonically increasing function.

The overall generation algorithm of nearest neighbor candidates works in two phases (See (Spring & Shrivastava, 2017; Chen et al., 2019a) for details):

Pre-processing Phase: Constructing L hash tables from the data by storing all elements $x \in \mathcal{C}$. We only store pointers to the vectors in the hash tables because storing whole data vectors is very memory inefficient.

Query Phase: Given a query Q, we search for its nearest neighbors. We obtain the union from all buckets collected from the L hash tables. Note that we do not scan all the elements in $\mathcal{C}$, we only probe L different buckets, one bucket for each hash table. After generating the set of potential candidates, we compute the distance between the query and each item in the candidate set and sort to find the nearest neighbor.

B Experiment Details

B.1 Dataset Statistic

In our work, we present an experiment on 4 datasets. The first two datasets, Wiki10-31k and Delicious-200K are obtained from the Extreme Classification (EC) Repository(Bhatia et al., 2016), which is a benchmark for various recommendation systems. Each Extreme Classification dataset uses Bag-of-words (BoW) features as input and multi-hot label vector as output. Note that it is possible that some label class appears only in the training/testing set of the EC datasets. Therefore, it is possible to improve over full inference accuracy as we are filtering out irrelevant label

Dataset	Wiki10-31k	Delicious-200K	Text8	Wiki-Text-2	PTB
Output Dimension	30938	205443	1355336	50000	10002
Training Samples	14146	6616	11903644	725434	929536
Testing Samples	196606	100095	5101563	245550	73728

Table 4: This table summarizes the statistics for each dataset.

classes by hashing. For language modeling, we introduce 2 datasets from two models. For Word2vec model, we use the text8 dataset from (Mahoney, 2011). We conduct three preprocessing steps on the dataset: (1) Remove the words with a frequency less than 2 from the vocabulary and mark the removed word as 'UNK'. (2) Represent each word in the document as an input one-hot vector. (3) For each input word, represent its previous 25 words and after 25 words as a multi-hot vector. Then, use the vector as the label. We also introduce a RNN based language models that use the Wiki-Text-2 dataset (Merity et al., 2016). In this dataset, we were given a 35 word sequence as a multi-hot vector input, we would like to predict the next 35 words sequence. Therefore, the label vector is also multi-hot. Details about each dataset are summarized in Table 4.

B.2 Tasks and Models

Extreme Classifications The Extreme Classifications model targets at predicting the labels with ultra-high label dimensionality given the input BoW features. The network architecture is summarized as: (1) An embedding layer that maps multi-hot input vector into a dense 128 dimension vector. (2) Nonlinear activation function. (3) An output layer with number of neurons equals to label dimensionality.

Word2vec The Word2vec model targets at predicting the neighbor words given the central word. The network architecture is summarized as: (1) An embedding layer that maps one-hot input vector into a dense 128 dimension vector. (2) Nonlinear activation function. (3) An output layer with number of neurons equals to vocabulary size.

RNN Recurrent neural networks (RNN) is a widely applied network structure (Medsker & Jain, 2001; Chen et al., 2020c; You et al., 2021a). The RNN language model targets at predicting the next words given the current sequence of words. The network architecture is summarized from input to output as: (1) An embedding layer that maps multi-hot input vector into a dense 200 dimension vector. (2) 1st Dropout function. (3) 2 LSTM layers with a hidden size equivalent to 200. (4) 2st Dropout function. (3) An output layer with the number of neurons equals vocabulary size.

Transformer Transformer models are widely applied in vision (Dosovitskiy et al., 2020; Han et al., 2020; You et al., 2022), language (Devlin et al., 2018; Liu et al., 2021) and acoustic tasks (You et al., 2020; Chen et al., 2021b). We use the transformer to perform the mask language modeling task, which targets at predicting the mask word from the context. we set number of layers to 6, number of heads to 4, word embedding dimension to 256, and hidden dimension to 256. We train both models using Adam optimizer and set the learning rate to be 1e-5.

B.3 When at Same Accuray

For efficient NN inference, both *HALOS* and other baselines has a trade-off between computation and accuracy. Here, we control the accuracy. Specifically, we tune parameters for each method and report the shortest time usage with roughly the same P@1. We summarize the results in Table 5.

The time usage of *HALOS* is the shortest comparing to all other baselines on all datasets. Moreover, the gap between *HALOS* and the best competitor is significant: On Delicious200K, *HALOS* achieves 34% time reduction; On Wiki10-30K, *HALOS* achieves 50% time reduction; On Text8, *HALOS* achieves 36% time reduction; On Wiki-Text2, *HALOS* achieves 50% time reduction.

B.4 Implementation

We build a demo implementation for LSS with link `https://github.com/Ottovonxu/HALOS`. The complete system would be released in the future.

Method	Dataset	$p@1$	Avg.Time(ms)
HALOS	Delicious200K	0.4 ± 0.01	**0.60**
PQ	Delicious200K	0.4 ± 0.01	3.12
ip-NSW	Delicious200K	0.4 ± 0.01	1.02
GD	Delicious200K	0.4 ± 0.01	0.91
NeuralLSH	Delicious200K	0.4 ± 0.01	1.33
SLIDE	Delicious200K	0.4 ± 0.01	2.33
HALOS	Wiki10-30K	0.7 ± 0.01	**0.14**
PQ	Wiki10-30K	0.7 ± 0.01	4.17
ip-NSW	Wiki10-30K	0.7 ± 0.01	0.36
GD	Wiki10-30K	0.7 ± 0.01	0.32
NeuralLSH	Wiki10-30K	0.7 ± 0.01	0.28
SLIDE	Wiki10-30K	0.7 ± 0.01	2.99
HALOS	Text8	0.7 ± 0.01	**0.13**
PQ	Text8	0.7 ± 0.01	13.77
ip-NSW	Text8	0.7 ± 0.01	0.25
GD	Text8	0.7 ± 0.01	0.22
NeuralLSH	Text8	0.7 ± 0.01	0.29
SLIDE	Text8	0.7 ± 0.01	9.99
HALOS	Wiki2-LSTM	0.25 ± 0.01	**0.09**
PQ	Wiki2-LSTM	0.25 ± 0.01	11.33
ip-NSW	Wiki2-LSTM	0.25 ± 0.01	0.18
GD	Wiki2-LSTM	0.25 ± 0.01	3.13
NeuralLSH	Wiki2-LSTM	0.25 ± 0.01	3.13
SLIDE	Wiki2-LSTM	0.25 ± 0.01	0.59

Table 5: This table summarizes the time usage of *HALOS* and other baselines at roughly the same accuracy on four dataset.