SLIDE : TRAINING DEEP NEURAL NETWORKS WITH LARGE OUTPUTS ON A CPU FASTER THAN A V100-GPU

A LOCALITY SENSITIVE HASHING

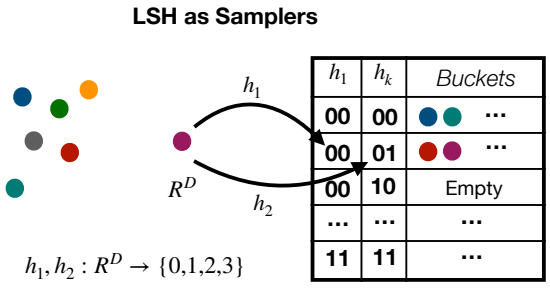

Figure 1. Schematic diagram of LSH. For an input, we obtain multiple hash codes and retrieve candidates from the respective buckets.

In formal terms, consider $\mathcal{H}$ to be a family of hash functions mapping R^D to some set $\mathcal{S}$.

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

- **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, for approximate nearest neighbor search, $p_1 > p_2$ and $c < 1$ is needed. An LSH allows us to construct data structures that give provably efficient query time algorithms for the approximate near-neighbor problem with the associated similarity measure.

One sufficient condition for a hash family $\mathcal{H}$ to be an LSH family is that the **collision probability** $Pr_{\mathcal{H}}(h(x) = h(y))$ should be a monotonically increasing with the similarity, i.e.

$$Pr_{\mathcal{H}}(h(x) = h(y)) = f(Sim(x, y)), \tag{1}$$

where f is a monotonically increasing function. In fact, most of the popular known LSH families, such as Simhash (Gionis et al., 1999) and WTA hash (Yagnik et al., 2011; Chen & Shrivastava, 2018), satisfy this strong property. It can be noted that Equation 1 automatically guarantees the two required conditions in the Definition A for any S_0 and $c < 1$.

It was shown in (Indyk & Motwani, 1998) that having an LSH family for a given similarity measure is sufficient for efficiently solving nearest-neighbor search in sub-linear time. Given a family of (S_0, cS_0, p_1, p_2)-sensitive hash functions, one can construct a data structure for c-NN with $O(n^\rho \log n)$ query time and space $O(n^{1+\rho})$, where $\rho = \frac{\log p_1}{\log p_2} < 1$.

The Algorithm: The LSH algorithm uses two parameters, (K, L). We construct L independent hash tables from the collection $\mathcal{C}$. Each hash table has a meta-hash function H that is formed by concatenating K random independent hash functions from $\mathcal{F}$. 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.

The candidate generation algorithm works in two phases [See (Spring & Shrivastava, 2017a) for details]:

1. **Pre-processing Phase:** We construct L hash tables from the data by storing all elements $x \in \mathcal{C}$. We only store pointers to the vector in the hash tables because storing whole data vectors is very memory inefficient.
2. **Query Phase:** Given a query Q; we search for its nearest-neighbors. We report the union from all of the buckets collected from the L hash tables. Note that we do not scan all the elements in $\mathcal{C}$. Instead, we only probe L different buckets, one bucket for each hash table.

After generating the set of potential candidates, the nearest-neighbor is computed by comparing the distance between each item in the candidate set and the query.

A.1 LSH for Estimations and Sampling

LSH for Estimations and Sampling: Although LSH provides provably fast retrieval in sub-linear time, LSH is known to be very slow for accurate search because it requires very large number of tables, i.e. large L. Also, reducing the overheads of bucket aggregation and candidate filtering is a problem on its own. Consequent research led to the sampling view of LSH (Spring & Shrivastava, 2017b;a; CHEN et al., 2018; Chen et al., 2018; Luo & Shrivastava,

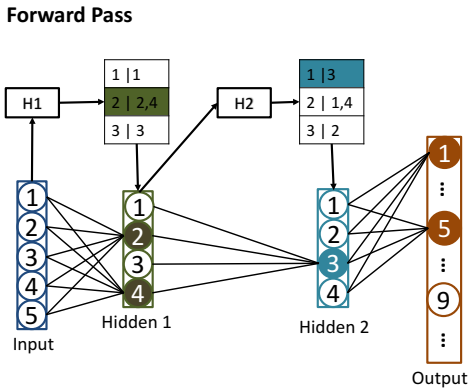

Figure 2. Forward Pass: Given an input, we first get the hash code $H1$ for the input, query the hash table for the first hidden layer, and obtain the active neurons. We get the activations for only this set of active neurons. We do the same for the subsequent layers and obtain a final sparse output. In practice, we use multiple hash tables per layer.

2018) that alleviates costly searching by efficient sampling. It turns out that merely probing a few hash buckets (as low as 1) is sufficient for adaptive sampling. Observe that an item returned as a candidate from a (K, L)-parameterized LSH algorithm is sampled with probability $1-(1-p^K)^L$, where p is the collision probability of LSH function (sampling probability is monotonic in p). Thus, with LSH algorithm, the candidate set is an adaptive sampled where the sampling probability changes with K and L.

This sampling view of LSH was the key ingredient for the algorithm proposed in paper (Spring & Shrivastava, 2017b) that shows the first possibility of adaptive dropouts in near-constant time, leading to efficient backpropagation algorithm.

A.1.1 MIPS Sampling

Recent advances in maximum inner product search (MIPS) using asymmetric locality sensitive hashing has made it possible to sample large inner products.

For the sake of brevity, it is safe to assume that given a collection $\mathcal{C}$ of vectors and query vector Q, using (K, L)-parameterized LSH algorithm with MIPS hashing (Shrivastava & Li, 2014a), we get a candidate set S. Every element in $x_i \in \mathcal{C}$ gets sampled into S with probability p_i, where p_i is a monotonically increasing function of $Q \cdot x_i$. Thus, we can pay a one-time linear cost of preprocessing $\mathcal{C}$ into hash tables, and any further adaptive sampling for query Q only requires few hash lookups.

Algorithm 1 SLIDE Algorithm

Input: $DataX, LabelY$
Output: θ
Weights w_l initialization for each layer l
LSH hash tables HT_l, hash functions h_l initialization for each layer l
Compute $h_l(w_l^a)$ for all neurons
Insert all the neuron ids a, into HT_l according to $h_l(w_l^a)$
for $e = 1 : Iterations$ **do**
  $Input_0 = Batch(X, B)$
  for $l = 1 : Layer$ **do**
    S_l = $Sample(Input_l, HT_l)$ (Algorithm 2)
    activations = Forward Propagation ($Input_l$, S_l)
    $Input_{l+1} = activations$
  end for
  for $l = 1 : Layer$ **do**
    Backpropagation (S_l)
  end for
end for
return θ

Algorithm 2 Algorithm for LSH Sampling

Input: $Input_l$**,** HT_l**,** h_l
Output: S_l**, a set of active neurons on layer** l
Compute$h_l(Input_l)$.
for $t = 1 : L$ **do**
  S = $S \cap$ Query($h_l(Input_l)$, HT_l^t)
end for
return S

B Different Hash Functions

Signed Random Projection (Simhash) : Refer (Gionis et al., 1999) for explanation of the theory behind Simhash. We use $K \times L$ number of random pre-generated vectors with components taking only three values $\{+1, 0, -1\}$. The reason behind using only $+1s$ and $-1s$ is for fast implementation. It requires additions rather than multiplications, thereby reducing the computation and speeding up the hashing process. To further optimize the cost of Simhash in practice, we can adopt the sparse random projection idea (Li et al., 2006). A simple implementation is to treat the random vectors as sparse vectors and store their nonzero indices in addition to the signs. For instance, let the input vector for Simhash be in R^d. Suppose we want to maintain $1/3$ sparsity, we may uniformly generate $K * L$ set of $d/3$ indices from $[0, d-1]$. In this way, the number of multiplications for one inner product operation during the generation of the hash codes would simply reduce from d to $d/3$. Since the random indices are produced from one-time generation, the cost can be safely ignored.

Winner Takes All Hashing (WTA hash) : In SLIDE, we slightly modify the WTA hash algorithm from (Yagnik et al., 2011) for memory optimization. Originally, WTA takes $O(KLd)$ space to store the random permutations Θ given the input vector is in R^d. $m << d$ is a adjustable hyper-parameter. We only generate $\frac{KLm}{d}$ rather than $K * L$ permutations and thereby reducing the space to $O(KLm)$. Every permutation is split into $\frac{d}{m}$ parts (bins) evenly and each of them can be used to generate one WTA hash code. Computing the WTA hash codes also takes $O(KLm)$ operations.

Densified Winner Takes All Hashing (DWTA hash) : As argued in (Chen & Shrivastava, 2018), when the input vector is very sparse, WTA hashing no longer produces representative hash codes. Therefore, we use DWTA hashing, the solution proposed in (Chen & Shrivastava, 2018). Similar to WTA hash, we generate $\frac{KLm}{d}$ number of permutations and every permutation is split into $\frac{d}{m}$ bins. DWTA loops through all the nonzero (NNZ) indices of the sparse input. For each of them, we update the current maximum index of the corresponding bins according to the mapping in each permutation.

It should be noted that the number of comparisons and memory lookups in this step is $O(NNZ * \frac{KLm}{d})$, which is significantly more efficient than simply applying WTA hash to sparse input. For empty bins, the densification scheme proposed in (Chen & Shrivastava, 2018) is applied.

Densified One Permutation Minwise Hashing (DOPH) : The implementation mostly follows the description of DOPH in (Shrivastava & Li, 2014b). DOPH is mainly designed for binary inputs. However, the weights of the inputs for each layer are unlikely to be binary. We use a thresholding heuristic for transforming the input vector to binary representation before applying DOPH. The k highest values among all d dimensions of the input vector are converted to 1s and the rest of them become 0s. Define idx_k as the indices of the top k values for input vector x. Formally,

$$Threshold(x_i) = \begin{cases} 1, & \text{if } i \in idx_k. \\ 0, & \text{otherwise.} \end{cases}$$

We could use sorting algorithms to get the top k indices, but it induces at least $O(dlogd)$ overhead. Therefore, we keep a priority queue with indices as keys and the corresponding data values as values. This requires $O(dlogk)$ operations.

C Reducing the Sampling Overhead

The key idea of using LSH for adaptive sampling of neurons with large activation is sketched in 'Introduction to overall system' section in the main paper. We have designed three strategies to sample large inner products: 1) Vanilla Sampling 2) Topk Sampling 3) Hard Thresholding. We first introduce them one after the other and then discuss their utility and efficiency. Further experiments are reported in section D.

Vanilla Sampling: Denote β_l as the number of active neurons we target to retrieve in layer l. After computing the hash codes of the input, we randomly choose a table and only retrieve the neurons in that table. We continue retrieving neurons from another random table until β_l neurons are selected or all the tables have been looked up. Let us assume we retrieve from τ tables in total. Formally, the probability that a neuron N_l^j gets chosen is,

$$Pr(N_l^j) = (p^K)^\tau (1 - p^K)^{L-\tau}, \tag{2}$$

where p is the collision probability of the LSH function that SLIDE uses. For instance, if Simhash is used,

$$p = 1 - \frac{cos^{-1}\left(\frac{(w_l^j)^T x_l}{||w_l^j||_2 \cdot ||x_l||_2}\right)}{\pi}.$$

From the previous process, we can see that the time complexity of vanilla sampling is $O(\beta_l)$.

TopK Sampling: In this strategy, the basic idea is to obtain those neurons that occur more frequently among all L hash tables. After querying with the input, we first retrieve all the neurons from the corresponding bucket in each hash table. While retrieving, we use a hashmap to keep track of the frequency with which each neuron appears. The hashmap is sorted based on the frequencies, and only the neurons with top β_l frequencies are selected. This requires additional $O(|N_l^a|)$ space for maintaining the hashmap and $O(|N_l^a| + |N_l^a| log |N_l^a|)$ time for both sampling and sorting.

Hard Thresholding: The TopK Sampling could be expensive due to the sorting step. To overcome this, we propose a simple variant that collects all neurons that occur more than a certain frequency. This bypasses the sorting step and also provides a guarantee on the quality of sampled neurons. Suppose we only select neurons that appear at least m times in the retrieved buckets, the probability that a neuron N_l^j gets chosen is,

$$Pr(N_l^j) = \sum_{i=m}^{L} \binom{L}{i} (p^K)^i (1 - p^K)^{L-i}, \tag{3}$$

Figure 3 shows a sweep of curves that present the relation between collision probability of $h_l(w_l^j)$ and $h_l(x_l)$ and the probability that neuron N_l^j is selected under various values of m when $L = 10$. We can visualize the trade-off between collecting more good neurons and omitting bad neurons by tweaking m. For a high threshold like $m = 9$, only the neurons with $p > 0.8$ have more than $Pr > 0.5$ chance of retrieval. This ensures that bad neurons are eliminated but

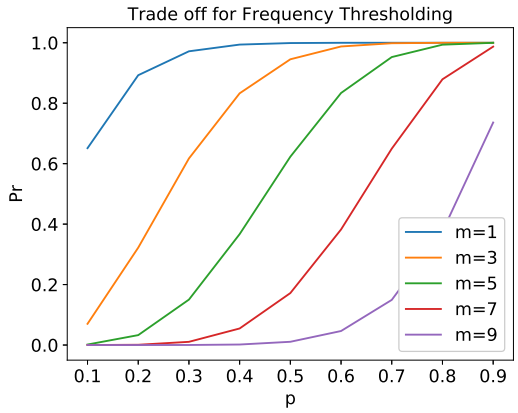

Figure 3. Hard Thresholding: Theoretical selection probability Pr vs the collision probabilities p for various values of frequency threshold m (eqn. 3). High threshold ($m = 9$) gets less number of false positive neurons but misses out on many active neurons. A low threshold ($m = 1$) would select most of the active neurons along with lot of false positives.

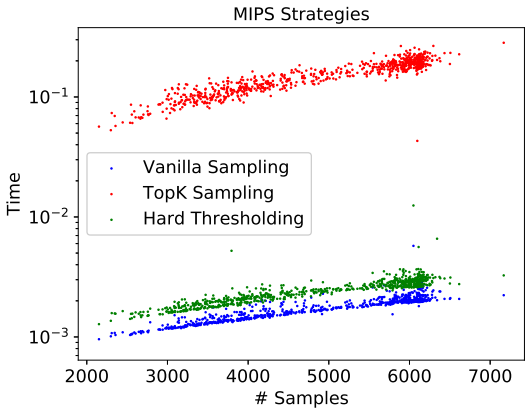

Figure 4. Sampling Strategies: Time consumed (in seconds) for various sampling methods after retrieving active neurons from Hash Tables.

Table 1. Time taken by hash table insertion schemes

	Insertion to HT	Full Insertion
Reservoir Sampling	0.371 s	18 s
FIFO	0.762 s	18 s

the retrieved set might be insufficient. However, for a low threshold like $m = 1$, all good neurons are collected but bad neurons with $p < 0.2$ are also collected with $Pr > 0.8$. Therefore, depending on the tolerance for bad neurons, we choose an intermediate m in practice.

C.1 Reducing the Cost of Updating Hash Tables

We introduce the following heuristics for addressing the expensive costs of updating the hash tables:

1) Recomputing the hash codes after every gradient update is computationally very expensive. Therefore, we dynamically change the update frequency of hash tables to reduce the overhead. Assume N_0 is the initial update frequency and $t - 1$ is the number of times the hash tables have already been updated. We apply exponential decay on the update frequency such that the t^{th} hash table update happens on iteration $\sum_{i=0}^{t-1} N_0 e^{\lambda i}$ where λ is a tunable decay constant. The intuition behind this scheme is that the gradient updates in the initial stage of the training are larger than those in the later stage, especially while close to convergence.

2) SLIDE needs a policy for adding a new neuron to a bucket when it is already full. To solve such a problem, we use the same solution in (Wang et al., 2018) that make use of Vitters reservoir sampling algorithm (Vitter, 1985) as the replacement strategy. It was shown that reservoir sampling retains the adaptive sampling property of LSH tables, making the process sound. In addition, for further speed up, we implement a simpler alternative policy based on FIFO (First In First Out).

3) For Simhash, the hash codes are computed by $h_w^{sign}(x) = sign(w^T x)$. During backpropagation, only the weights connecting the active neurons across layers get updated. Only those weights contribute to the change of $w^T x$. Therefore, we can also memorize the result of $w^T x$ besides the hash codes. When $x \in R^d$ gets updated in only $d^{'}$ out of d dimensions, where $d^{'} \ll d$, we only need $O(d^{'})$ rather than $O(d)$ addition operations to compute the new hash codes for updated x.

D Design Choice Comparisons

In the main paper, we presented several design choices in SLIDE which have different trade-offs and performance behavior, e.g., executing MIPS efficiently to select active neurons, adopting the optimal policies for neurons insertion in hash tables, etc. In this section, we substantiate those design choices with key metrics and insights. In order to better analyze them in more practical settings, we choose to benchmark them in real classification tasks on Delicious-200K dataset.

D.1 Evaluating Sampling Strategies

Sampling is a crucial step in SLIDE. The quality and quantity of selected neurons and the overhead of the selection strategy significantly affect the SLIDE performance. We profile the running time of these strategies, including Vanilla sampling, TopK thresholding, and Hard thresholding, for selecting a different number of neurons from the hash tables during the first epoch of the classification task.

Figure 4 presents the results. The blue, red and green dots represent Vanilla sampling, TopK thresholding, and Hard thresholding respectively. It shows that the TopK thresholding strategy takes magnitudes more time than Vanilla sampling and Hard thresholding across all number of samples consistently. Also, we can see that the green dots are just slightly higher than the blue dots meaning that the time complexity of Hard Thresholding is slightly higher than Vanilla Sampling. Note that the y-axis is in log scale. Therefore when the number of samples increases, the rates of change for the red dots are much more than those of the others. This is not surprising because TopK thresholding strategy is based on sorting algorithms which has $O(nlogn)$ running time. Therefore, in practice, we suggest choosing either of Vanilla Sampling or Hard Thresholding for efficiency. For instance, we use Vanilla Sampling in our extreme classification experiments because it is the most efficient one. Furthermore, the difference between iteration wise convergence of the tasks with TopK Thresholding and Vanilla Sampling are negligible.

D.2 Addition to Hashtables

SLIDE supports two implementations of insertion policies for hash tables described in section 3.1 in main paper. We profile the running time of the two strategies, Reservoir Sampling and FIFO. After the weights and hash tables initialization, we clock the time of both strategies for insertions of all 205,443 neurons in the last layer of the network, where 205,443 is the number of classes for Delicious dataset. Then we also benchmark the time of whole insertion process including generating the hash codes for each neuron before inserting them into hash tables.

The results are shown in Table C. The column "Full Insertion" represents the overall time for the process of adding all neurons to hash tables. The column "Insertion to HT" represents the exact time of adding all the neurons to hash tables excluding the time for computing the hash codes. Reservoir Sampling strategy is more efficient than FIFO. From an algorithmic view, Reservoir Sampling inserts based on some probability, but FIFO guarantees successful insertions. We observe that there are more memory accesses with FIFO. However, compared to the full insertion time, the benefits of Reservoir Sampling are still negligible. Therefore we can choose either strategy based on practical utility. For instance, we use FIFO in our experiments.

REFERENCES

Chen, B. and Shrivastava, A. Densified winner take all (wta) hashing for sparse datasets. In *Uncertainty in artificial intelligence*, 2018.

CHEN, B., SHRIVASTAVA, A., and STEORTS, R. C. Unique entity estimation with application to the syrian conflict. *THE ANNALS*, 2018.

Chen, B., Xu, Y., and Shrivastava, A. Lsh-sampling breaks the computational chicken-and-egg loop in adaptive stochastic gradient estimation. 2018.

Gionis, A., Indyk, P., and Motwani, R. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pp. 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-615-7. URL http://dl.acm.org/citation.cfm?id=645925.671516.

Indyk, P. and Motwani, R. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613. ACM, 1998.

Li, P., Hastie, T. J., and Church, K. W. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 287–296. ACM, 2006.

Luo, C. and Shrivastava, A. Scaling-up split-merge mcmc with locality sensitive sampling (lss). *arXiv preprint arXiv:1802.07444*, 2018.

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. Densifying one permutation hashing via rotation for fast near neighbor search. In *International Conference on Machine Learning*, pp. 557–565, 2014b.

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*, 2017a.

Spring, R. and Shrivastava, A. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 445–454. ACM, 2017b.

Vitter, J. S. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1): 37–57, 1985.

Wang, Y., Shrivastava, A., Wang, J., and Ryu, J. Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search. In *ACM SIGMOD Record*, pp. 889–903. ACM, 2018.

Yagnik, J., Strelow, D., Ross, D. A., and Lin, R.-s. The power of comparative reasoning. In *2011 International Conference on Computer Vision*, pp. 2431–2438. IEEE, 2011.