Scalable and Sustainable Deep Learning via Randomized Hashing

Ryan Spring
Rice University
Department of Computer Science
Houston, Texas, USA 43017-6221
rdspring1@rice.edu

Anshumali Shrivastava
Rice University
Department of Computer Science
Houston, Texas, USA 43017-6221
anshumali@rice.edu

ABSTRACT

Current deep learning architectures are growing larger in order to learn from complex datasets. These architectures require giant matrix multiplication operations to train millions of parameters. Conversely, there is another growing trend to bring deep learning to low-power, embedded devices. The matrix operations, associated with the training and testing of deep networks, are very expensive from a computational and energy standpoint. We present a novel hashing-based technique to drastically reduce the amount of computation needed to train and test neural networks. Our approach combines two recent ideas, Adaptive Dropout and Randomized Hashing for Maximum Inner Product Search (MIPS), to select the nodes with the highest activations efficiently. Our new algorithm for deep learning reduces the overall computational cost of the forward and backward propagation steps by operating on significantly fewer nodes. As a consequence, our algorithm uses only 5% of the total multiplications, while keeping within 1% of the accuracy of the original model on average. A unique property of the proposed hashing-based back-propagation is that the updates are always sparse. Due to the sparse gradient updates, our algorithm is ideally suited for asynchronous, parallel training, leading to near-linear speedup, as the number of cores increases. We demonstrate the scalability and sustainability (energy efficiency) of our proposed algorithm via rigorous experimental evaluations on several datasets.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Theory of computation** → **Streaming, sublinear and near linear time algorithms**; **Parallel algorithms**;

KEYWORDS

Neural Networks; Deep Learning; Locality-Sensitive Hashing; Randomized Algorithms; Parallel Computing

KDD'17, , August 13–17, 2017, Halifax, NS, Canada.

ACM ISBN 978-1-4503-4887-4/17/08…$15.00
https://doi.org/10.1145/3097983.3098035

1 INTRODUCTION

Deep learning is revolutionizing big-data applications, after being responsible for groundbreaking improvements in image classification [12] and speech recognition [9]. With the recent upsurge in data, at a much faster rate than our computing capabilities, neural networks are growing larger to process information more effectively. In 2012, state-of-the-art convolutional neural networks contained at most 10 layers. Afterward, each successive year has brought deeper architectures with greater accuracy. Microsoft's deep residual network [8], which won the ILSVRC 2015 competition with a 3.57% error rate, had 152 layers and 11.3 billion FLOPs. To handle such large neural networks, researchers usually train them on large computer clusters with high-performance graphics cards.

Due to the growing size and complexity of networks, efficient algorithms for training massive deep networks in a distributed, parallel environment is currently the most sought-after problem in both academia and the commercial industry. For example, Google [5] used a 1-billion parameter neural network, which took three days to train on a 1000-node cluster, totaling over 16,000 CPU cores. Each instantiation of the network spanned 170 servers. In distributed computing environments, the parameters of giant deep networks are required to be split across multiple nodes. However, this setup requires costly communication and synchronization between the parameter server and processing nodes in order to transfer the gradient and parameter updates. The sequential and dense nature of gradient updates prohibits any efficient splitting (sharding) of the neural network parameters across computer nodes. There is no clear way to avoid the costly synchronization without resorting to some ad-hoc breaking of the network. This ad-hoc breaking of deep networks is not well-understood and is likely to hurt performance. Synchronization is one of the major hurdles in scalability. Asynchronous training is the ideal solution, but it is sensitive to conflicting, overlapping parameter updates, which leads to poor convergence.

While deep networks are growing larger and more complex, there is also push for greater energy efficiency to satisfy the growing popularity of machine learning applications on mobile phones and low-power devices. For example, there is recent work by McMahan et al. [22] aimed at leveraging the vast data of mobile devices. This work has the users train neural networks on their local data, and then periodically transmit their models to a central server. This approach preserves the privacy of the user's personal data, but still allows the central server's model to learn effectively. Their work is dependent on training neural networks locally. Back-propagation is the most popular algorithm for training deep networks. Each iteration of the back-propagation algorithm is composed of giant

matrix multiplications. These matrices are very large, especially for massive networks with millions of nodes in the hidden layer, which are common in industrial practice. Large matrix multiplications are parallelizable on GPUs, but not energy-efficient. Users require their phones and tablets to have long battery life. Reducing the computational costs of neural networks, which directly translates into longer battery life, is a critical issue for the mobile industry.

The current challenges for deep learning illustrate a great demand for algorithms that reduce the amount of computation and energy usage. To reduce the bottleneck matrix multiplications, there has been a flurry of works around reducing the amount of computations associated with them. Most of them revolve around exploiting low-rank matrices or low precision updates (See Section 6.2 for details). However, updates with these techniques are hard to parallelize making them unsuitable for distributed and large scale applications. On the contrary, our proposal capitalizes on the sparsity of the activations to reduce the computation complexity. To the best of our knowledge, this is the first proposal that exploits sparsity to reduce the amount of computation associated with deep networks. We further show that our approach admits asynchronous parallel updates leading to perfect scaling with increasing parallelism.

Recent machine learning research has focused on techniques for dealing with the famous problem of over-fitting with deep networks. A notable line of work [2, 20, 21] improved the accuracy of neural networks by only updating the neurons with the highest activations. Adaptive dropout [2] sampled neurons in proportion to an affine transformation of the neuron's activation. The Winner-Take-All (WTA) approach [20, 21] kept only the top-k% neurons by using a hard threshold. It was found that such a selective choice of nodes and sparse updates provide a natural regularization [30]. However, these approaches rely on inefficient, brute-force techniques to find the best neurons. Thus, these techniques are equally as expensive as the standard back-propagation method leading to no computational savings.

Our idea is to index the neurons (the weights of each neuron as a vector) in a hash table using locality sensitive hashing. These hash tables allow us to select (or sample) the neurons with the highest activations in sub-linear time. Moreover, since our approach results in a sparse active set of neurons randomly, the gradient updates are unlikely to overwrite each other because of their sparsity. Sparse updates are ideal for asynchronous and parallel gradient updates. It is known that asynchronous stochastic gradient descent (ASGD) [23] will converge if the number of simultaneous parameter updates is small. We heavily leverage this sparsity which unique to our proposal. On several deep learning benchmarks, we show that our approach outperforms standard algorithms including vanilla dropout [30] at high sparsity levels and matches the performance of adaptive dropout [2] and winner-take-all [20, 21] while needing less computation (only 5%).

1.1 Our Contributions:

(1) We present a scalable and sustainable algorithm for training and testing fully-connected neural networks. Our idea capitalizes on the recent, successful technique of adaptive dropouts and locality sensitive hashing (LSH) for maximum inner product search (MIPS) [25]. We show significant reductions in the computational requirement for training deep networks without any significant loss in accuracy (within 1% of the accuracy of the original model on average). In particular, our method achieves the performance of other state-of-the-art regularization methods such as Dropout, Adaptive Dropout, and Winner-Take-All when using only 5% of the neurons in a standard neural network.
(2) Our proposal reduces computations associated with both the training and testing (inference) of deep networks by reducing the multiplications needed for the feed-forward and back-propagation operations.
(3) The key idea in our algorithm is to associate LSH hash tables [7, 11] with every layer. These hash tables support constant-time $O(1)$ insertion and deletion operations.
(4) Our scheme naturally leads to sparse gradient updates. Sparse updates are ideally suited for massively parallelizable asynchronous training [23]. We demonstrate that this sparsity opens room for truly asynchronous training without any compromise with accuracy. As a result, we obtain near-linear speedup when increasing number of cores.
(5) The code for training and running our randomized-hashing approach is available online [1]

2 RELATED WORK

There have been several recent advances aimed at improving the performance of neural networks. Courbariaux et al. [4], Lin et al. [17] reduced the number of floating point multiplications by mapping the network's weights stochastically to {-1, 0, 1} during forward propagation. Reducing the precision of the weights and activations is an orthogonal approach. In addition, binary quantization only gives a constant factor of improvement, while our approach is sub-linear in the size of the network. Therefore, the improvements will be significantly more for larger networks.

Sindhwani et al. [28] uses structured matrix transformations with low-rank matrices to reduce the number of parameters for the fully-connected layers of a neural network. This low-rank constraint leads to a smaller memory footprint. However, such an approximation is not well suited for asynchronous, parallel training, limiting its scalability. We instead use random but sparse activations that are easily parallelized by leveraging advances in approximate query processing. (See Section 6.2 for details)

We briefly review Dropout and its variants, which are popular sparsity promoting techniques for neural networks. Although such randomized, sparse activations improve the generalization of neural networks, to the best of our knowledge, this sparsity has not been adequately exploited to make deep networks computationally cheap and parallelizable. We provide first such evidence.

Dropout [30] is primarily a regularization technique that addresses the issue of over-fitting by randomly dropping half of the nodes in a hidden layer while training the network. The nodes are independently sampled for every mini-batch of training data [30]. We reinterpret Dropout as a technique for reducing the number of multiplications during forward and back-propagation phases, by ignoring nodes randomly in the network. It is known that the

[1] https://github.com/rdspring1/LSH_DeepLearning

network's performance becomes worse when too many nodes are dropped from the network. Usually, only 50% of the nodes in the network are dropped when training the network. At test time, the network takes the average of the thinned networks to form a prediction from the input data, which involves computing the activations for all of the nodes in the network.

Adaptive dropout [2] is an enhancement to the dropout technique that adaptively chooses the nodes based on their activations. The methodology samples a small subset of nodes from the network, where the sampling is in proportion to the node activations. Adaptive dropouts demonstrate better performance than vanilla dropout [30]. A notable feature of Adaptive Dropout is that you can drop significantly more nodes than the standard Dropout technique while still retaining superior performance.

Winner-Take-All [20, 21] is an extreme form of Adaptive Dropouts that uses mini-batch statistics to enforce a sparsity constraint. With this technique, only the k% largest, non-zero activations are used during the forward and back-propagation phases of training. This approach requires computing the forward propagation step before selecting the k% nodes with a hard threshold. Unfortunately, all of these techniques require full computation of the activations to sample nodes selectively. Therefore, they are only intended for better generalization and not for reducing computational cost. Our approach uses the insight that selecting a very sparse set of hidden nodes with the highest activations can be reformulated as dynamic approximate query processing problem, which can solve efficiently using locality sensitive hashing. The differentiating factor between our approach and the two other algorithms, Adaptive Dropout and Winner-Take-All (WTA), is that we use sub-linear time, randomized hashing to determine the active set of nodes instead computing the activation for each node individually.

There is also another orthogonal line of work which uses universal hashing to reduce the network's memory footprint [3]. Unlike our objective, theirs was to reduce the number of parameters in a neural network by using a hash function to tie virtual weights together to the same real weight. The HashedNet architecture is more computational expensive than the standard neural network because it incurs an additional overhead when either rebuilding the weight matrix or looking up the value of corresponding weight. Hashed nets are complementary to our approach because we focus on reducing the computational cost of neural networks rather than its memory size.

3 BACKGROUND

3.1 Locality-Sensitive Hashing (LSH)

Locality-Sensitive Hashing (LSH) [6, 7, 10] is a popular, sub-linear time algorithm for approximate nearest-neighbor search. The main idea is to place similar items into the same bucket of a hash table with high probability. An LSH hash function maps an input data vector to an integer key - $h(x) : \mathbb{R}^D \mapsto [0, 1, 2, \ldots, N]$. A collision occurs when the hash values for two elements are equal - $h(x) = h(y)$. The collision probability for an LSH hash function is proportional to the similarity metric between the two elements - $Pr[h(x) = h(y)] \propto sim(x, y)$. Essentially, similar items are more likely to collide with each other under the same hash function.

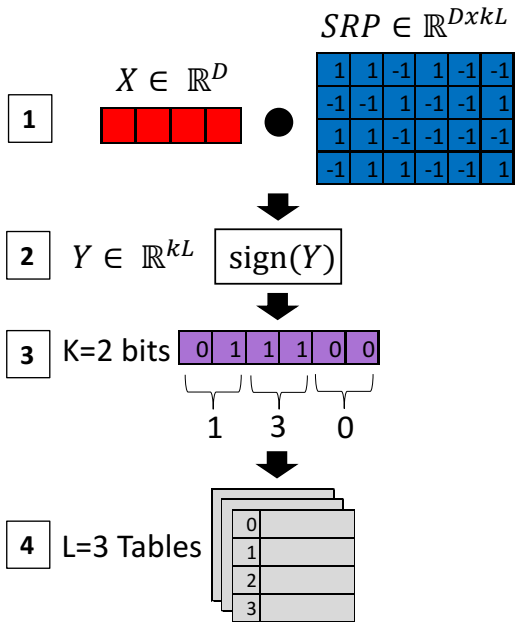

Figure 1: Locality Sensitive Hashing: (1) Compute the projection using a signed, random matrix $\mathcal{R}^{D \times kL}$ and the item $x \in \mathcal{R}^D$. (2) Generate a bit from the sign of each entry in the projection $\mathcal{R}^{kL}$ (3) From the kL bits, we create L integer fingerprints with k bits per fingerprint. (4) Add the item x into each hash table using the corresponding integer key

Sub-linear Time Search using (K, L) LSH Algorithm. To be able to answer approximate nearest-neighbor queries in sub-linear time, the idea is to create hash tables that have constant-time insert and search operations (See Figure 1). Given the collection C, which we are interested in querying for the set of nearest-neighbors, the hash tables are generated using some locality sensitive hash (LSH) family. We assume that we have access to the appropriate locality sensitive hash (LSH) family $\mathcal{F}$ for the given similarity metric.

The classic LSH algorithm uses two parameters - (K, L) to improve the precision and recall of nearest-neighbors for a collection C. Each hash table has a meta-hash function H that is formed by concatenating K random hash functions from $\mathcal{F}$. Now, under the meta-hash function H, all of the K independent hash function values must match in order for two items to have the same fingerprint. $[H(x) = H(y)] \iff [h_i(x) = h_i(y)]$ for all $[i = 0, 1, 2, \ldots, K]$. L hash tables are constructed from the collection C. Given a query, we collect one buckets from each hash table and return the union of L buckets. Intuitively, the meta-hash function reduces the amount of false positives because valid nearest-neighbor items are more 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 probability that at least one of the L meta-hash fingerprints match and the two items form a candidate pair [15] is

$$Pr[H(x) = H(y)] = 1 - (1 - p^k)^L$$

The overall algorithm works in two phases:

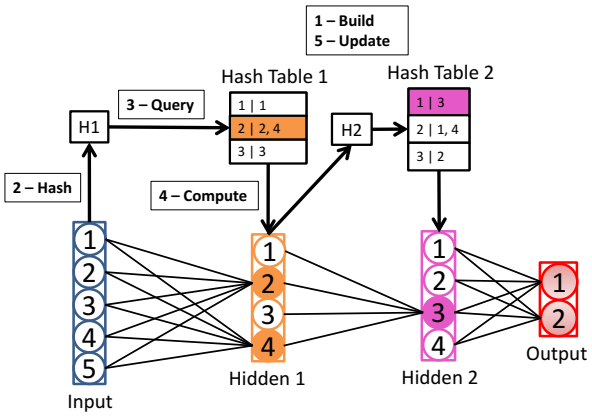

Figure 2: A visual representation of a neural network using randomized hashing: (1) Build the hash tables by hashing the weights of each hidden layer (Pre-processing operation) (2) Hash the layer's input (3) Query the layer's hash table(s) for the active set *AS* (4) Only perform forward and back-propagation on the neurons in the active set. The solid-colored neurons in the hidden layer are the active neurons. (5) Update the *AS* weights via gradient descent and the hash tables by rehashing the updated *AS* weights.

(1) **Pre-processing Phase:** We construct L hash tables from the data by storing all elements $x \in C$. (See Figure 2 for an illustration) 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 will search for its nearest-neighbors. We report the union from all of the buckets collected from the L hash tables. Note, we do not scan all of the elements in C, we only probe L different buckets, one bucket per hash table. **Note:** For nearest-neighbor search, we need to filter these candidates further. However, our algorithm does not require such filtering, because we want to perform adaptive sampling and not exact nearest-neighbor search. (explained in section 6.1)

Multi-Probe LSH. One common complaint with the classical LSH algorithm is that it requires a significant number of hash tables. Large L increases the processing time and memory cost. A simple solution is to probe multiple "close-by" buckets in each hash table rather than probing only one bucket [19]. Thus, for a given query q, in addition to probing bucket $H_j(q)$ in hash table $j \in L$, we also generate several new addresses to probe by slightly perturbing values of $H_j(q)$. This simple idea significantly reduces the number of tables needed with LSH, allowing us to work with only a few hash tables. (See Lv et al. [19] for more details)

4 PROPOSED METHODOLOGY

4.1 Intuition

The Winner-Take-All [20, 21] technique shows that we should only consider a few nodes with large activations (top k%) and to ignore the rest while computing the feed-forward pass. Furthermore, the back-propagation updates should only be performed on those top k% nodes. Let n denote the total number of nodes in the neural network. Let *AS* (Active Set) define the subset of top k% nodes with significant activations where $|AS| \ll n$. For each gradient update, Winner-Take-All needs to first perform $O(n \log n)$ work to sort the activations to find the *AS*, and then to update the $O(AS)$ weights. $O(n \log n)$ seems quite wasteful. In particular, finding the active set *AS* is a search problem that can be solved well using smart data structures. Furthermore, if the data structure is dynamic and efficient, then the gradient updates will also be efficient.

For a node i with weight w_i and input x, its activation is a monotonic function of the inner product $w_i^T \cdot x$. Thus, selecting the active set *AS* is equivalent to searching through a collection of weight vectors for the ones that have large inner products with the input x. Equivalently, from a query processing perspective, if we treat the input x as a query, then the search problem of selecting top k% nodes can be solved in sub-linear time using the recent advances in maximum inner product search (MIPS) [25]. Our proposal is to create hash tables with indexes generated by asymmetric locality sensitive hash functions tailored for inner products. With such hash tables, we can very efficiently approximate the active set *AS* for a given query input x.

One last implementation challenge is how to update the nodes (weights associated with them) in the *AS*, during the gradient update. If we can perform these updates in $O(AS)$ instead of $O(n)$, then we save a significant amount of computation. Therefore, we need a data structure where updates are also efficient. We describe our the details of our system in Section 4.2.

4.2 Hashing-Based Back-Propagation

We use randomized hash functions to build hash tables from the nodes in each hidden layer. We sample nodes from the hash table with probability proportional to the node's activation in sub-linear time. We then perform forward and back propagation only on the active nodes retrieved from the hash tables. We later update the hash tables to reorganize only the modified weights.

Figure 2 illustrates an example neural network with two hidden layers, five input nodes, and two output nodes. Hash tables are built for each hidden layer, where the weighted connections for each node are hashed to place the node in its corresponding bucket. Creating hash tables to store all the initial parameters is a one-time operation which requires cost linear in the number of parameters.

During a forward propagation pass, the input to the hidden layer is hashed with the same hash function used to build the hidden layer's hash table. The input's fingerprint is used to collect the active set *AS* nodes from the hash table. The hash table only contains pointers to the nodes in the hidden layer. Then, forward propagation is performed only on the nodes in the active set *AS*.

Note: As argued, unlike exact nearest-neighbor search, we report everything retrieved from buckets as the active set *AS*, without any filtration of the candidates. Our randomized hashing approach

Algorithm 1 Deep Learning with Randomized Hashing

```
// HF_l - Layer l Hash Function
// HT_l - Layer l Hash Tables
// AS_l - Layer l Active Set
// θ^l_AS ∈ W^l_AS, b^l_AS - Layer l Active Set parameters
Randomly initialize parameters W^l, b^l for each layer l
HF_l = constructHashFunction(k, L)
HT_l = constructHashTable(W^l, HF_l)
while not stopping criteria do
  for each training epoch do
    // Forward Propagation
    for layer l = 1 . . . N do
      fingerprint_l = HF_l(a_l)
      AS_l = collectActiveSet(HT_l, fingerprint_l)
      for each node i in AS_l do
        a^{l+1}_i = f(W^l_i a^l_i + b^l_i)
      end for
    end for
    // Backpropagation
    for layer l = 1 . . . N do
      ΔJ(θ^l_AS) = computeGradient(θ^l_AS, AS_l)
      θ^l_AS = updateParameters(θ^l_AS, ΔJ(θ_AS))
    end for
    for each Layer l -> updateHashTables(HF_l, HT_l, θ^l)
  end for
end while
```

adaptively samples [29] from all of the candidates. (See Section 6.1 for the sampling view of our algorithms)

The rest of the hidden layer's nodes, which are not part of the active set, are ignored and automatically switched off. On the back-propagation pass, the active set is reused to determine the gradient and to update the parameters. We rehash the nodes in each hidden layer to account for the changes in the network during training.

In detail, the hash function for each hidden layer is composed of K randomized hash functions. We use the sign of an asymmetrically transformed random projection to generate the K bits for each data vector. (See Shrivastava and Li [27] for details) The K bits are stored together efficiently as an integer, forming a fingerprint for the data vector. We create a hash table of 2^K buckets, but we only keep the nonempty buckets to minimize the memory footprint (analogous to hash-maps in Java). Each bucket stores pointers to the nodes whose fingerprints match the bucket's id instead of the node itself. In figure 2, we showed only one hash table, which is likely to miss valuable nodes in practice. In our implementation, we generate L hash tables for each hidden layer, and each hash table has an independent set of K random projections. Our final active set from these L hash tables is the union of the buckets selected from each hash table. For each layer, we have L hash tables. Effectively, we have two tunable parameters, K bits and L tables to control the size and the quality of the active sets. The K bits increase the precision of the fingerprint, meaning the nodes in a bucket are more likely to generate higher activation values for a given input. The L tables increase the probability of finding useful nodes that are missed because of the randomness in the hash functions.

Efficient Query and Updates: Our algorithm critically depends on the efficiency of the query and update procedure. The hash table is one of the most efficient data structures, so this is not a difficult challenge. We only store references to the weight vectors, which makes the hash table a very light entity. Furthermore, we reduce the number of hash tables L by using Multi-Probe LSH [19]. A large number of tables L increases the hashing time and memory cost. A simple solution is to probe multiple "nearby" buckets in every hash table rather than probing only a single bucket. Multi-Probe LSH for a binary hash function is quite straightforward. We just have to randomly flip a few bits of the meta-hash fingerprint to generate more addresses.

The gradient update to a weight vector associated with a node may change its location in the hash table. Updating the node's location only requires one insertion and one deletion in the respective buckets. There are plenty of data structures available for representing the buckets that have efficient insert and delete operations. In theory, we can use a red-black-tree to ensure both insertion and deletion cost is logarithmic in the size of the bucket. However, in our implementation, the buckets are represented by simple arrays because they are easy to parallelize, and the buckets are relatively sparse. Arrays have constant-time $O(1)$ insertion and linear-time $O(b)$ deletion, where b is the size of buckets.

Overall Cost: For each layer, during every Stochastic Gradient Descent (SGD) update, we compute $K \times L$ hashes of the input data, probe several buckets per hash table, and then take their union. In our experiments, we use $K = 6$ and $L = 5$ – only 30 hash computations per data point. There are many other techniques to further reduce this hashing cost [1, 16, 24, 26]. The process gives us the active set AS of nodes, which is significantly smaller than the total number of nodes n. During SGD, we update all of the nodes in the AS and then rebuild the hash tables. Overall, the cost is on the order of the number of nodes in the active set AS. For 1000 nodes per layer and an AS containing only 5% of the layer's nodes, we only have to update around 50 nodes. The primary bottleneck is calculating the activation for each node in the AS. The performance benefits will be even more significant for larger neural networks.

Bonus: Sparse Updates can be Parallelized: As mentioned, we only need to update the set of weights associated with nodes in the active set AS. If each AS is very sparse, then it is unlikely that a group of active sets will significantly overlap. Small overlaps imply fewer conflicts while updating the parameters. Fewer conflicts while updating is an ideal ground where SGD updates can be parallelized without any overhead. In fact, it was shown both theoretically and experimentally that random, sparse SGD updates can be parallelized without compromising with the convergence [23]. Vanilla SGD is a sequential operation, and parallel updates lead to poor convergence, due to significant overwrites among the gradient updates. Our experimental results, in Section 5.3, support this known phenomenon. Exploiting this unique property, we show near-linear scaling without hurting convergence using our algorithm, while increasing the number of concurrently running models.

5 EVALUATIONS

We design experiments to answer the following six important questions:

(1) How much can we reduce computation without affecting the neural network's accuracy?
(2) How effective is adaptive sampling compared to a random sampling of nodes?
(3) How does the accuracy of our approximate hashing-based approach compare with the expensive, exact approaches of adaptive dropouts [2] and Winner-Takes-all [20, 21]?
(4) How is the network's convergence effected by increasing number of cores when using asynchronous SGD?
(5) Is the sparse gradient update necessary? Is there any deterioration in performance, if we perform standard, dense updates in parallel?
(6) How much does the walk-clock time decrease, as a function of increasing number of cores?

For evaluation, we implemented the following five approaches to compare and contrast against our approach.

- Standard (NN) : A full-connected neural network
- Dropout (VD) [30]: A neural network that disables the nodes of a hidden layer using a fixed probability threshold
- Adaptive Dropout (AD) [2]: A neural network that disables the nodes of a hidden layer using a probability threshold based on the inner product of the node's weights and the input.
- Winner Take All (WTA) [20, 21]: A neural network that sorts the activations of a hidden layer and selects the k% largest activations
- Randomized Hashing (LSH): A neural network that selects nodes using randomized hashing. A hard threshold limits the active node set to k% sparsity

Dataset	Train Size	Test Size
MNIST8M	8,100,000	10,000
NORB	24,300	24,300
Convex	8,000	50,000
Rectangles	12,000	50,000

Figure 3: Dataset - Training + Test Size

5.1 Datasets

To test our neural network implementation, we used four publicly available datasets - MNIST8M [18], NORB [14], CONVEX, and RECTANGLES [13]. The statistics of these datasets are summarized in Table 3. The MNIST8M, CONVEX, and RECTANGLES datasets contain 28×28 images, forming 784-dimensional feature vectors. The MNIST8M task is to classify each handwritten digit in the image correctly. It is derived by applying random deformations and translations to the MNIST dataset. The CONVEX dataset objective is to identify if a single convex region exists in the image. The goal for the RECTANGLES dataset is to discriminate between tall and wide rectangles overlaid on a black and white background image. The NORB dataset [14] contains images of 50 toys, belonging to 5 categories under various lighting conditions and camera angles.

5.2 Sustainability

5.2.1 Experimental Setting. All of the experiments for our approach and the other techniques were run on a 6-core Intel i7-3930K machine with 16 GB of memory. Our approach uses stochastic gradient descent with Momentum and Adagrad [5]. Since our approach uniquely selects an active set of nodes for each hidden layer, we focused on a CPU-based approach to simplify combining randomized hashing with neural networks. The ReLU activation function was used for all methods. The learning rate for each approach ranged between 10^{-2} and 10^{-4}. The parameters for the randomized hash tables were K = 6 bits and L = 5 tables with multi-probe LSH [19]. For the experiments, we use a fixed threshold to cap the number of active nodes selected from the hash tables to guarantee the amount of computation is within a certain level.

5.2.2 Effect of computation levels. Figures 4, 5 show the accuracy of each method on neural networks with 2 and 3 hidden layers with the percentage of active nodes ranging from [0.05, 0.10, 0.25, 0.5, 0.75, 0.9]. The standard neural network is our baseline in these experiments and is marked with a dashed black line. Each hidden layer contains 1000 nodes. The x-axis represents the average percentage of active nodes per epoch selected by each technique. Our approach only performs the forward and back propagation steps on the nodes selected in each hidden layer. The other baseline techniques except for Dropout (VD) perform the forward propagation step for all of the nodes first, before setting node activations to zero based on the corresponding algorithm. Therefore, Dropout (VD) and our proposal (LSH) require fewer multiplications than the standard neural network training procedure.

Figures 4 and 5 summarizes the accuracy of different approaches at various computations levels.

- Our method (LSH) gives the best overall accuracy with the fewest number of active nodes. The fact that our approximate method is even slightly better than WTA and adaptive dropouts is not surprising, as it is long known that a small amount of random noise leads to better generalization. (See Srivastava et al. [30] for examples)
- As the number of active nodes decreases from 90% to 5%, LSH experiences the smallest drop in performance and less performance volatility.
- VD experiences the greatest drop in performance when reducing the number of active nodes from 50% to 5%.
- WTA performed better than VD when the percentage of active nodes is less than 50%
- As the number of active nodes approaches 100%, the performance stabilizes for each method.

Lowering the computational cost of running neural networks and running fewer operations reduces the energy consumption and heat produced by the processor. However, large neural networks provide better accuracy and arbitrarily reducing the number of active nodes hurts performance. Our experiments show that our method (LSH) provides the best of both worlds - high performance and low processor computation. This approach is ideal for mobile phones because reducing the processor's load directly translates into longer battery life.

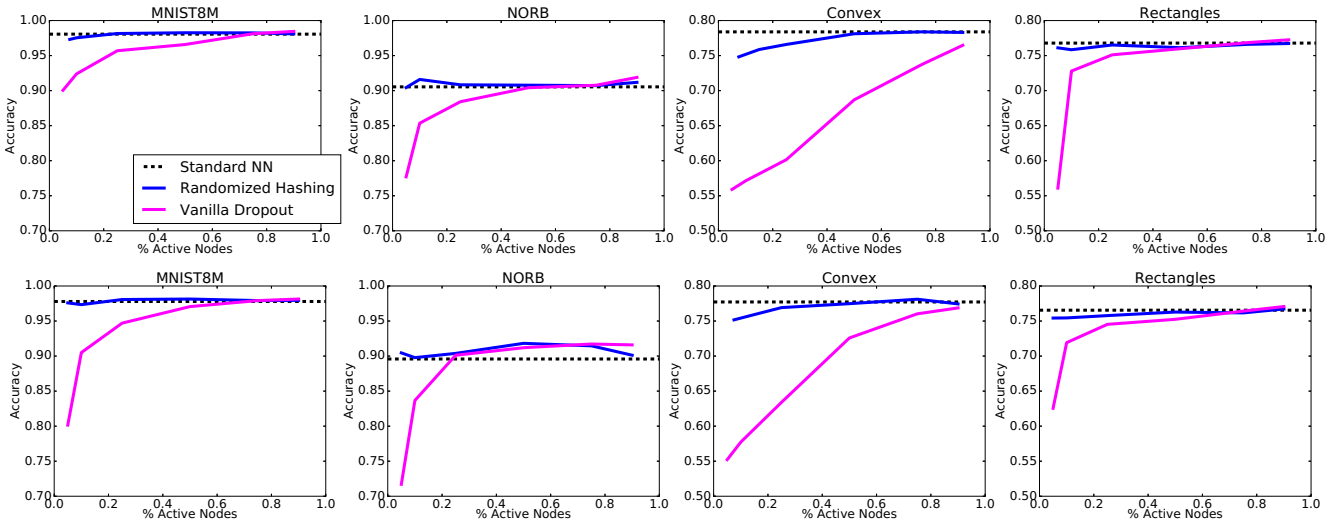

Figure 4: Classification accuracy under different levels of active nodes with networks on the MNIST (1st), NORB (2nd), Convex (3rd) and Rectangles (4th) datasets. The standard neural network (dashed black line) is our baseline accuracy. We can clearly see that adaptive sampling with hashing (LSH) is significantly more effective than random sampling (VD). Top Panels: 2 hidden Layers. Bottom Panels: 3 hidden Layers

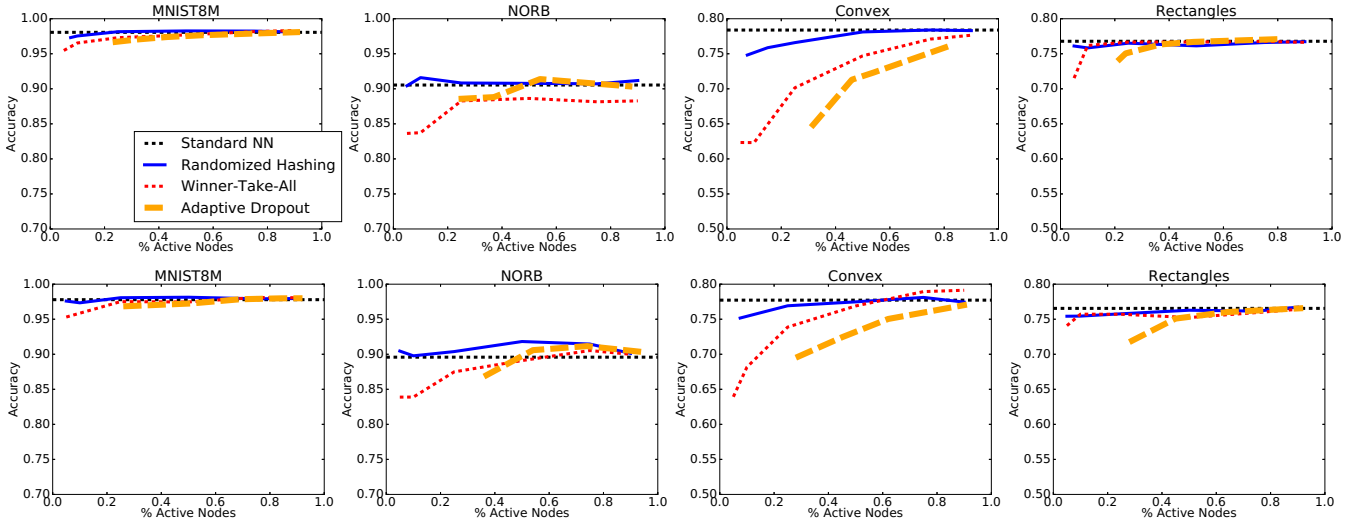

Figure 5: Classification accuracy under different levels of active nodes with networks on the MNIST (1st), NORB (2nd), Convex (3rd) and Rectangles (4th) datasets. The standard neural network (dashed black line) is our baseline accuracy. WTA and AD (dashed red and yellow lines) perform the same amount of computation as the standard neural network. Those two techniques select nodes with high activation values to achieve better accuracy, but they require computing the activation for every node in the hidden layer. We compare our LSH approach to determine whether our randomized algorithm achieves comparable performance, while reducing the total amount of computation. We do not have data for adaptive dropout at the 5% and 10% computation levels because those models diverged when the number of active nodes dropped below 25%. Top Panels: 2 hidden Layers. Bottom Panels: 3 hidden Layers

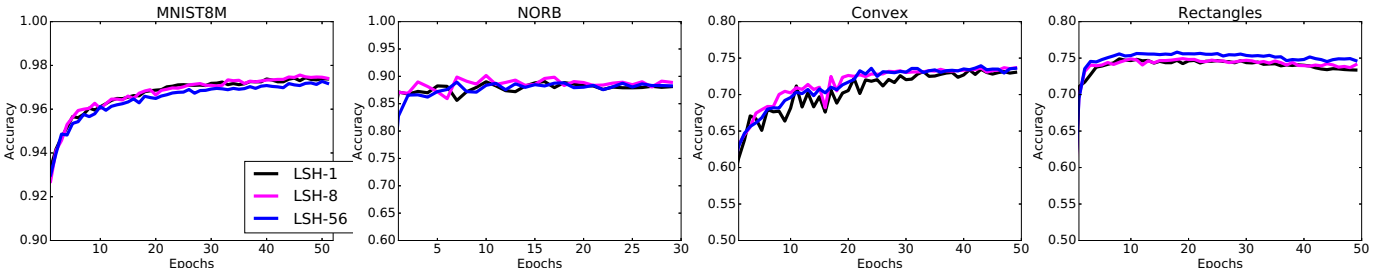

Figure 6: The convergence of our randomized hashing approach (LSH-5%) over several training epochs using asynchronous stochastic gradient (ASGD) with 1, 8, 56 cores. We used a (3 hidden layer) network on the MNIST (1st), NORB (2nd), Convex (3rd) and Rectangles (4th) datasets. Only 5% of the standard network's computation was performed in this experiment. ASGD has no effect on convergence with the sparse, random gradient updates.

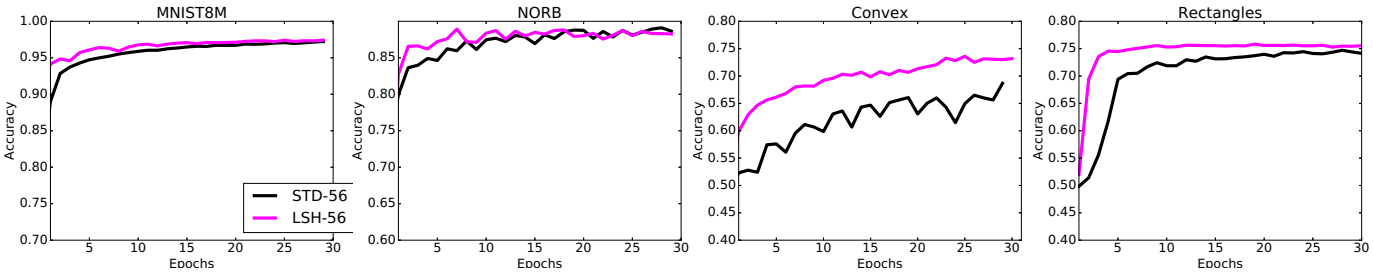

Figure 7: Performance comparison between our randomized hashing approach and a standard network using asynchronous stochastic gradient descent (ASGD) on an Intel Xeon ES-2697 machine with 56-cores. We used (3 hidden layer) networks on MNIST (1st), NORB (2nd), Convex (3rd) and Rectangles (4th). All networks were initialized with the same settings for this experiment. We see that parallelizing the non-sparse gradient updates leads to poor convergence.

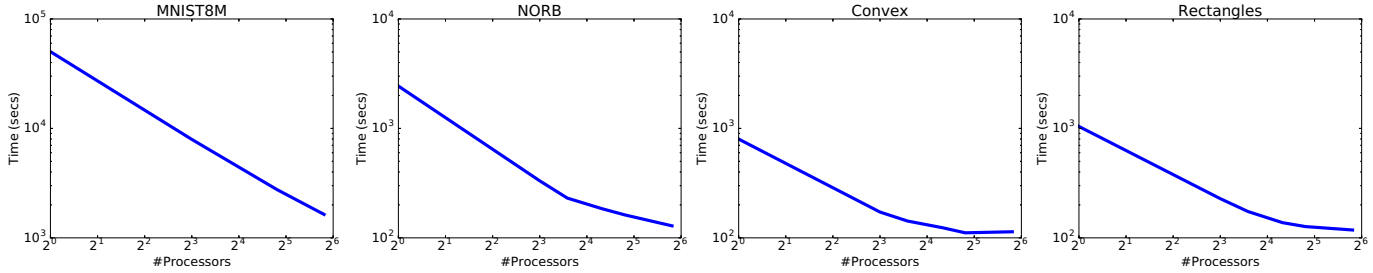

Figure 8: The wall-clock per epoch for our approach (LSH-5%) gained by using asynchronous stochastic gradient descent. We used a (3 hidden layer) network on the MNIST (1st), NORB (2nd), Convex (3rd) and Rectangles (4th). We see smaller performance gains with the Convex and Rectangles datasets because there are not enough training examples to use of all of the cores effectively. Only 5% of the standard network's computation was performed in this experiment.

5.3 Scalability

5.3.1 Experimental Setting. We now show experiments to demonstrate the scalability of our approach to large-scale, distributed computing environments. Specifically, we are testing if our approach maintains accuracy and improves training time, as we increase the number of cores. We use asynchronous stochastic gradient descent with momentum and adagrad [5, 23]. Our implementation utilizes data parallelism by running the same model on multiple training examples concurrently. The gradient is applied without synchronization to maximize performance. We run all of the experiments on an Intel Xeon ES-2697 machine with 56 cores and 256 GB of memory. The ReLU activation was used for all models, and the learning rate ranged between 10^{-2} and 10^{-3}.

5.3.2 Results with different number of cores. Figure 6 shows how our method performs with asynchronous stochastic gradient descent (ASGD) using only 5% of the neurons of a full-sized neural network. The neural network has three hidden layers, and each hidden layer contains 1000 neurons. The x-axis represents the number of epochs completed, and the y-axis shows the test accuracy for the given epoch. We compare how our model converges with multiple cores working concurrently. Since our ASGD implementation does not use locks, it depends on the sparsity of the gradient to ensure the model converges and performs well [23]. From our experiments, we see that our method converges at a similar rate and obtains the same accuracy regardless of the number of cores running ASGD.

Figure 8 illustrates how our method scales with multiple cores. The inherent sparsity of our randomized hashing approach reduces the number of simultaneous updates and allows for more asynchronous models without any performance penalty. We show the corresponding drop in wall-clock computation time per epoch while adding more cores. We achieve roughly a 31x speed up while running ASGD with 56 cores.

5.3.3 ASGD Performance Comparison with Standard Neural Network. Figure 7 compares the performance of our LSH approach against a standard neural network (STD) when running ASGD with 56-cores. We clearly out-perform the standard network for all of our experimental datasets. However, since there is a large number of models running concurrently, their gradients are constantly being overridden, preventing ASGD from converging to an optimal local minimum. Our approach produces a spare gradient that reduces the number of conflicts between the different models, while keeping enough valuable gradients for ASGD to progress towards the local minimum efficiently.

From Figures 6, 7 and 8, we conclude the following:

(1) The gradient updates are quite sparse with 5% LSH and running them in parallel does not affect the convergence rate of our hashing-based approach. Even when running 56 cores in parallel, the convergence is indistinguishable from the sequential update (1 core) on all the four datasets.
(2) If we instead run vanilla SGD in parallel, then the convergence is affected. The convergence is in general slower compared the sparse 5% LSH. This slow convergence is due to dense updates which leads to overwrites. Parallelizing dense updates affects the four datasets differently. For convex dataset, the convergence is very poor.
(3) As expected, we obtain near-perfect decrease in the wall clock times with increasing the number of cores with LSH-5%. Note, if there are too many overwrites, then atomic overwrites are necessary, which will create additional overhead and hurt the parallelism. Thus, the near-perfect scalability also indicates fewer gradient overwrites.
(4) On the largest dataset - MNIST8M, the running time per epoch for the 1-core implementation is 50,254 seconds. The 56-core implementation runs in 1,637 seconds. Since the convergence is not affected, there is a 31x speedup in the training process while using 56 cores.
(5) We see that the performance gains from data parallelism become flat with the Convex and Rectangle datasets, especially while using a large number of cores. This poor scaling occurs because the two datasets have fewer training examples than MNIST8M or NORB, so there is less parallel work for a large number of cores. We do not see such behaviors with MNIST8M which has around 8 million training examples.

6 DISCUSSIONS

Machine learning with a huge parameter space is becoming a common phenomenon. Stochastic Gradient Descent (SGD) remains the most popular optimization algorithm due to its effectiveness and simplicity. Each SGD update is expected to alter only a small subset of the parameters significantly. Identifying that subset of parameters is a search problem. We can exploit the rich literature in approximate query processing to find this active set of parameters efficiently. Of course, the approximate active set contains a small amount of random noise, which is often good for generalization. Sparsity and randomness enhance data parallelism because the gradient updates are unlikely to overwrite each other. In conclusion, we are reformulating the machine learning problem into an approximate query processing problem, and then leveraging the decades of research from the systems and database communities. We have demonstrated one concrete example, by showing how neural networks can be scaled-up using randomized hashing. We believe that the combination of sparsity with approximate query processing is the future of large-scale machine learning.

6.1 Equivalence with Adaptive Dropouts

From a statistical perspective, Asymmetric Locality-Sensitive Hashing (ALSH) [25] for finding nodes with large inner products is equivalent to Adaptive Dropout [2] with a non-trivial sampling distribution.

The Adaptive Dropout technique uses the Bernoulli distribution to sample nodes with large activation. In theory, any distribution assigning probabilities in proportion to the node's activation is sufficient. We argue that the Bernoulli distribution is sub-optimal. There is another non-intuitive, but a very efficient distribution. This distribution comes from the theory of Locality-Sensitive Hashing (LSH) [11], which is primarily considered a black-box technique for fast sub-linear search. Our core idea comes from the observation that, for a given search query q, the LSH algorithm inherently samples, in sub-linear time, points from a distribution with probability proportional to $1-(1-p^K)^L$ [29]. Here, the collision probability p is a monotonic function of the similarity between the query and the retrieved point.

THEOREM 6.1. **Hashing-Based Efficient Samplingt** - *For a given input x to any layer of the neural network, any (K, L) parameterized LSH algorithm selects a node i, associated with weight vector w_i, with probability proportional to $1-(1-p^K)^L$. Here, the collision probability of the associated locality-sensitive hash function is $p = Pr[h(x) = h(w_i)]$. Thus, the sampling probability $1-(1-p^K)^L$ is monotonic with respect to p.*

6.2 Low-Rank vs Sparsity

The low-rank (or structured) assumption is very convenient for reducing the complexity of general matrix operations. However, low-rank, dense updates do not promote sparsity and are not friendly for distributed computing. The same principle holds with deep networks. We illustrate it with a simple example.

Consider a layer of the first network (left) shown in Figure 9. The insight is that if the weight matrix $W \in \mathbb{R}^{mxn}$ for a hidden layer has low-rank structure where rank $r \ll min(m, n)$, then it has a representation $W = UV$ where $U \in \mathbb{R}^{mxr}$ and $V \in \mathbb{R}^{rxn}$. This low-rank structure improves the storage requirements and matrix-multiplication time from $O(mn)$ to $O(mr + rn)$. As shown in Figure 9, there is an equivalent representation of the same network using an intermediate hidden layer that contains r nodes and uses the identity activation function. The weight matrices for the hidden layers in the second network (right) map to the matrix decomposition, U and V.

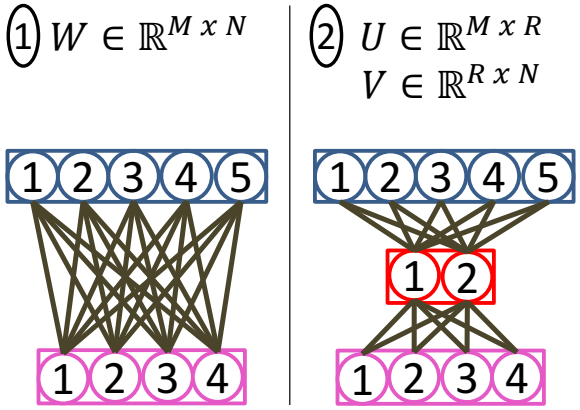

Figure 9: (1) The single-layer network is characterized by the weight matrix $W \in R^{M \times N}$ of constrained rank R such that $W = UV$ with $U \in \mathbb{R}^{M \times R}$ and $V \in \mathbb{R}^{R \times N}$. (2) An equivalent network contains two layers, represented by the matrices, U and V. The first layer uses the identity activation function I. (3) Both networks produce the same output. $a = f(W^T x) = f((UV)^T x) = f((V^T U^T)x) = f(V^T I(U^T x))$

Sindhwani et al. [28] uses structured matrix transformations with low-rank matrices to reduce the number of parameters for the fully-connected layers of a neural network. The low-rank, structured matrices require dense gradient updates, which is not ideally suited for data parallelism [23]. In this work, we move away from the low-rank assumption. Instead, we make use of sparsity to reduce the amount of computation. We showed that due to random sparsity, our approach is well-suited for Asynchronous Stochastic Gradient Descent (ASGD), leading to near-linear scaling.

7 FUTURE WORK

One future direction is to optimize our approach for low-power, mobile processors. There are many platforms including the Nvidia Tegra, Qualcomm Snapdragon, and Movidius Myriad. The other direction is to leverage the power of GPUs and distributed computing. GPUs are commonly used to train deep networks because of their high performance. The next logical step is to harness the computational savings via randomized hashing on GPUs.

8 ACKNOWLEDGMENTS

The work was supported by NSF Awards IIS-1652131 and DMS-1547433. We would like to thank reviewers of ICML 2016, NIPS 2016 and KDD 2017 for their encouraging remarks and suggestions.

REFERENCES

[1] Dimitris Achlioptas. 2003. Database-friendly Random Projections: Johnson-Lindenstrauss with Binary Coins. *J. Comput. Syst. Sci.* 66, 4 (June 2003), 671–687. https://doi.org/10.1016/S0022-0000(03)00025-4

[2] Jimmy Ba and Brendan Frey. 2013. Adaptive dropout for training deep neural networks. In *Advances in Neural Information Processing Systems 26*. 3084–3092.

[3] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. 2015. Compressing Neural Networks with the Hashing Trick. In *Proceedings of the 32nd International Conference on Machine Learning*. 2285–2294.

[4] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[5] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*. 1223–1231.

[6] Jinyang Gao, Hosagrahar Visvesvaraya Jagadish, Wei Lu, and Beng Chin Ooi. 2014. DSH: data sensitive hashing for high-dimensional KNN search. In *Proceedings of the 2014 ACM SIGMOD*. ACM, 1127–1138.

[7] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. *VLDB* 99, 6 (1999), 518–529.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[9] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE* 29, 6 (2012), 82–97.

[10] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 9, 1 (2015), 1–12.

[11] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. Dallas, TX, 604–613.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[13] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. 2007. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*. ACM, 473–480.

[14] Yann LeCun, Fu Jie Huang, and Leon Bottou. 2004. Learning methods for generic object recognition with invariance to pose and lighting. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, Vol. 2. IEEE, II–97.

[15] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. *Mining of massive datasets*. Cambridge University Press.

[16] Ping Li, Trevor J Hastie, and Kenneth W Church. 2006. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 287–296.

[17] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. 2015. Neural Networks with Few Multiplications. *arXiv preprint arXiv:1510.03009* (2015).

[18] Gaëlle Loosli, Stéphane Canu, and Léon Bottou. 2007. Training Invariant Support Vector Machines using Selective Sampling. In *Large Scale Kernel Machines*, Léon Bottou, Olivier Chapelle, Dennis DeCoste, and Jason Weston (Eds.). MIT Press, Cambridge, MA., 301–320. http://leon.bottou.org/papers/loosli-canu-bottou-2006

[19] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 950–961.

[20] Alireza Makhzani and Brendan Frey. 2013. k-Sparse Autoencoders. *arXiv preprint arXiv:1312.5663* (2013).

[21] Alireza Makhzani and Brendan J Frey. 2015. Winner-Take-All Autoencoders. In *Advances in Neural Information Processing Systems 28*. 2791–2799.

[22] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2016. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*. http://arxiv.org/abs/1602.05629

[23] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*. 693–701.

[24] Anshumali Shrivastava. 2016. Simple and Efficient Weighted Minwise Hashing. In *Advances in Neural Information Processing Systems 29*. 1498–1506.

[25] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In *Advances in Neural Information Processing Systems*. 2321–2329.

[26] Anshumali Shrivastava and Ping Li. 2014. Improved Densification of One Permutation Hashing. In *UAI*. Quebec, CA.

[27] Anshumali Shrivastava and Ping Li. 2015. Improved Asymmetric Locality Sensitive Hashing (ALSH) for Maximum Inner Product Search (MIPS). In *Conference on Uncertainty in Artificial Intelligence (UAI)*.

[28] Vikas Sindhwani, Tara Sainath, and Sanjiv Kumar. 2015. Structured transforms for small-footprint deep learning. In *Advances in Neural Information Processing Systems*. 3070–3078.

[29] Ryan Spring and Anshumali Shrivastava. 2017. 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).

[30] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.