HFH: HOMOLOGICALLY FUNCTIONAL HASHING FOR COMPRESSING DEEP NEURAL NETWORKS

Lei Shi, Shikun Feng, Zhifan Zhu
Baidu, Inc.
China
{shilei06,fengshikun01,zhuzhifan}@baidu.com

ABSTRACT

As the complexity of deep neural networks (DNNs) trends to grow to absorb the increasing sizes of data, memory and energy consumption has been receiving more and more attentions for industrial applications, especially on mobile devices. This paper presents a novel structure based on *homologically functional hashing* to compress DNNs, shortly named as HFH. For each weight entry in a deep net, HFH uses multiple low-cost hash functions to fetch values in a compression space, and then employs a small reconstruction network to recover that entry. The compression space is homological because all layers fetch hashed values from it. The reconstruction network is plugged into the whole network and trained jointly. On several benchmark datasets, HFH demonstrates high compression ratios with little loss on prediction accuracy. Particularly, HFH includes the recently proposed HashedNets (Chen et al., 2015a) as a degenerated scenario and shows significantly improved performance. Moreover, the homological hashing essence facilitates us to efficiently figure out one single desired compression ratio, instead of exhaustive searching throughout a combinatory space configured by all layers.

1 INTRODUCTION

Deep Neural networks (DNNs) have been receiving ubiquitous success in wide applications, ranging from computer vision (Krizhevsky et al., 2012), to speech recognition (Hinton et al., 2012), natural language processing (Collobert et al., 2011), and domain adaptation (Glorot et al., 2011). As the sizes of data mount up, people usually have to increase the number of parameters in DNNs so as to absorb the vast volume of supervision. High performance computing techniques are studied to speed up training, concerning optimization algorithms, parallel synchronisations on clusters w/o GPUs, and stochastic binarization/ternarization, etc (Courbariaux et al., 2015; Dettmers, 2016; Lin et al., 2016).

On the other hand the memory and energy consumption is usually, if not always, constrained in industrial applications (Kim et al., 2016; Yang et al., 2015). For instance, for commercial search engines (e.g., Google and Baidu) and recommendation systems (e.g., NetFlix and YouTube), the ratio between the increased model size and the improved performance should be considered given limited online resources. Compressing the model size becomes more important for applications on mobile and embedded devices (Han et al., 2016; Kim et al., 2016). Having DNNs running on mobile apps owns many great features such as better privacy, less network bandwidth and real time processing. However, the energy consumption of battery-constrained mobile devices is usually dominated by memory access, which would be greatly saved if a DNN model can fit in on-chip storage rather than DRAM storage (c.f. Han et al. (2016) for details).

A recent trend of studies are thus motivated to focus on compressing the size of DNNs while mostly keeping their predictive performance (Han et al., 2016; Kim et al., 2016; Yang et al., 2015). With different intuitions, there are mainly two types of DNN compression methods, which could be used in conjunction for better parameter savings. The first type tries to revise the training target into more informative supervision using *dark knowledge*. In specific, Hinton et al. (2014) suggested to train a large network ahead, and distill a much smaller model on a combination of the original labels and the soft-output by the large net. The second type observes the redundancy existence in network weights

(Chen et al., 2015a; Denil et al., 2013), and exploits techniques to constrain or reduce the number of free-parameters in DNNs during learning. This paper focuses on the latter type.

To constrain the network redundancy, several efforts (Denil et al., 2013; Le et al., 2013; Yang et al., 2015) formulated an original weight matrix into either low-rank or fast-food decompositions. Moreover Han et al. (2015) and Han et al. (2016) proposed a simple-yet-effective pruning-retraining iteration during training, followed by quantization and fine-tuning. An adaptive pruning strategy was further studied recently in (Guo et al., 2016) under the name of dynamic network surgery. Chen et al. (2015a) proposed HashedNets to efficiently implement parameter sharing prior to learning, and showed notable compression with much less loss of accuracy than low-rank decomposition. More precisely, prior to training, a hash function is used to randomly group (virtual) weights into a small number of buckets, so that all weights mapped into one hash bucket directly share a same value. HashedNets was further deliberated in frequency domain for compressing convolutional NNs in Chen et al. (2015b). It should be noted that, these approaches are not completely conflicted, and methods based on different assumptions may be used jointly for a better compression.

To obtain a desired compression in applications, one key question of most existing compression approaches is how to appropriately choose the compression ratios. Usually, people have to search within finite choices either heuristically or in a Bayesian manner. The searching procedure becomes more and more exhaustive as the network depth increases, since usually compression ratios are various for different (kinds of) layers (Han et al., 2015; Guo et al., 2016). Moreover, we observe HashedNets compresses model sizes greatly at marginal loss of accuracy for many situations, whereas significantly loses accuracy for others. After revisiting its mechanism, we conjecture this instability comes from at least three factors. First, hashing and training are disjoint in a two-phase manner, i.e., once inappropriate collisions exist, there may be no much optimization room left for training. Second, one single hash function is used to fetch a single value in the compression space, whose collision risk is larger than multiple hashes (Broder & Karlin, 1990). Third, parameter sharing within a buckets implicitly uses identity mapping from the hashed value to the virtual entry in HashedNets.

This paper proposes the homologically functional hashing (HFH) to relieve these problems. Specifically, we use *multiple hash functions* to map per virtual entry into multiple values in the compression space. Inspired from the "homology" in biology (Williams & Forey, 2004), the compression space is a *homological* space, meaning that it is accessed by all entries from all (kinds of) layers in a DNN. Then an additional network plays in a *mapping function* role from these hashed values to the virtual entry (before hashing), which can be also regarded as "reconstructing" the virtual entry from its multiple hashed values. Plugged into and jointly trained within the original network, the reconstruction network is of a comparably ignorable size, i.e., at low memory cost.

The homologically functional hashing structure includes HashedNets as a degenerated special case, and facilitates less value collisions and better value reconstruction. Moreover, the homology essence favors high efficiency during searching appropriate compression ratios, because the only value we have to determine is a scalar, i.e., the size of the homological compression space. Since HFH imposes no restriction on other network design choices (e.g. dropout and weight sparsification), it can be considered as a standard tool for DNN compression, and provably used jointly with other compression techniques for a even better compression. Experiments on several datasets demonstrate significantly larger reduction of model sizes and/or less loss on prediction accuracy, compared with HashedNets.

2 Background

Notations. Throughout this paper we express scalars in regular (A or b), vectors in bold ($\mathbf{x}$), and matrices in capital bold ($\mathbf{X}$). Furthermore, we use x_i to represent the i-th dimension of vector $\mathbf{x}$, and use X_{ij} to represent the (i, j)-th entry of matrix $\mathbf{X}$. Occasionally, $[\mathbf{x}]_i$ is also used to represent the i-th dimension of vector $\mathbf{x}$ for specification clarity . Notation $\mathbb{E}[\cdot]$ stands for the expectation operator.

Feed Forward Neural Networks. We define the forward propagation of the ℓ-th layer as

$$a_i^{\ell+1} = f(z_i^{\ell+1}), \quad \text{with} \quad z_i^{\ell+1} = b_i^{\ell+1} + \sum_{j=1}^{d^\ell} V_{ij}^\ell a_j^\ell, \quad \text{for } \forall i \in [1, d^{\ell+1}]. \tag{1}$$

For each ℓ-th layer, d^ℓ is the output dimensionality, $\mathbf{b}^\ell$ is the bias vector, and $\mathbf{V}^\ell$ is the (*virtual*) weight matrix in the ℓ-th layer. Vectors $\mathbf{z}^\ell, \mathbf{a}^\ell \in \mathbb{R}^{d^\ell}$ denote the units before and after the activation function $f(\cdot)$. Typical choices of $f(\cdot)$ include rectified linear unit (ReLU) (Nair & Hinton, 2010), sigmoid and tanh (Bishop, 1995).

Feature Hashing has been studied as a dimension reduction method for reducing model storage size without maintaining the mapping matrices like random projection (Shi et al., 2009; Weinberger et al., 2009). Briefly, it maps an input vector $\mathbf{x} \in \mathbb{R}^n$ to a much smaller feature space via $\boldsymbol{\phi} : \mathbb{R}^n \to \mathbb{R}^K$ with $K \ll n$. Following the definition in Weinberger et al. (2009), the mapping $\boldsymbol{\phi}$ is a composite of two approximate uniform hash functions $h : \mathbb{N} \to \{1, \dots, K\}$ and $\xi : \mathbb{N} \to \{-1, +1\}$. The j-th element of $\boldsymbol{\phi}(\mathbf{x})$ is defined as:

$$[\boldsymbol{\phi}(\mathbf{x})]_j = \sum_{i:h(i)=j} \xi(i) x_i. \tag{2}$$

As shown in (Weinberger et al., 2009), a key property is its inner product preservation, which we quote and restate below.

Lemma [Inner Product Preservation of Original Feature Hashing] With the hash defined by Eq. (2), the hash kernel is unbiased, i.e., $\mathbb{E}_{\boldsymbol{\phi}}[\boldsymbol{\phi}(\mathbf{x})^\top \boldsymbol{\phi}(\mathbf{y})] = \mathbf{x}^\top \mathbf{y}$. Moreover, the variance is $\mathrm{var}_{\mathbf{x},\mathbf{y}} = \frac{1}{K} \sum_{i \neq j} \left(x_i^2 y_j^2 + x_i y_i x_j y_j \right)$, and thus $\mathrm{var}_{\mathbf{x},\mathbf{y}} = \mathcal{O}(\frac{1}{K})$ if $||\mathbf{x}||_2 = ||\mathbf{y}||_2 = \mathrm{const}$.

HashedNets in (Chen et al., 2015a). As illustrated in Figure 1(a), HashedNets randomly maps network weights into a smaller number of groups prior to learning, and the weights in a same group share a same value thereafter. A naive implementation could be trivially achieved by maintaining a secondary matrix that records the group assignment, at the expense of additional memory cost however. HashedNets instead adopts a hash function that requires no storage cost with the model. Assume there is a finite memory budge K^ℓ per layer to represent $\mathbf{V}^\ell$, with $K^\ell \ll (d^\ell + 1) d^{\ell+1}$. We only need to store a weight vector $\mathbf{w}^\ell \in \mathbb{R}^{K^\ell}$, and assign V_{ij}^ℓ an element in $\mathbf{w}^\ell$ indexed by a hash function $h^\ell(i, j)$, namely

$$V_{ij}^\ell = \xi^\ell(i, j) \cdot w_{h^\ell(i,j)}^\ell, \tag{3}$$

where hash function $h^\ell(i, j)$ outputs an integer within $[1, K^\ell]$. Another independent hash function $\xi^\ell(i, j) : (d^{\ell+1} \times d^\ell) \to \pm 1$ outputs a sign factor, aiming to reduce the bias due to hash collisions (Weinberger et al., 2009). The resulting matrix $\mathbf{V}^\ell$ is *virtual*, since d^ℓ could be increased without increasing the *actual* number of parameters in $\mathbf{w}^\ell$ once the compression space size K^ℓ is determined and fixed.

Substituting Eq. (3) into Eq. (1), we have $z_i^{\ell+1} = b_i^{\ell+1} + \sum_{j=1}^{d^\ell} \xi^\ell(i, j) w_{h^\ell(i,j)}^\ell a_j^\ell$. During training, $\mathbf{w}^\ell$ is updated by back propagating the gradient via $\mathbf{z}^{\ell+1}$ (and the virtual $\mathbf{V}^\ell$). Besides, the activation function $f(\cdot)$ in Eq. (1) was kept as ReLU in (Chen et al., 2015a) to further relieve the hash collision effect through a sparse feature space. In both (Chen et al., 2015a) and this paper, the open source *xxHash*[1] is adopted as an approximately uniform hash implementation with low cost.

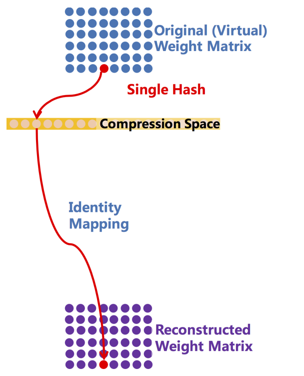
3 HOMOLOGICALLY FUNCTIONAL HASHING

3.1 STRUCTURE FORMULATION

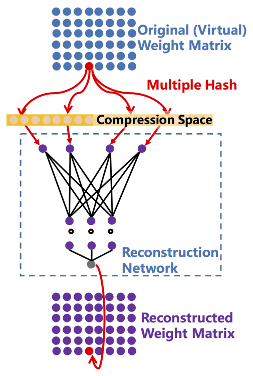
The key differences between our homologically functional hashing and HashedNets (Chen et al., 2015a) are three folded: (i) how to maintain the compression space, (ii) how to employ hash functions, and (iii) how to map from hashed values to a virtual entry. Precisely,

- Reminder in HashedNets the compression space are split into pieces, with one piece $\mathbf{w}^\ell$ for one layer. In contrast, the compression space $\mathbf{w} \in \mathbb{R}^K$ in HFH is taken as a whole, so that any entry in all layers fetches hashed values from a same space, namely the homology.
- Instead of adopting one pair of hash function (h, ξ) in Eq. (3), we use a set of multiple pairs of independent random hash functions. Let's say there are U pairs of mappings $\{h_u, \xi_u\}_{u=1}^U$, each $h_u(i, j)$ outputs an integer within $[1, K]$, and each $\xi_u(i, j)$ selects a sign factor.

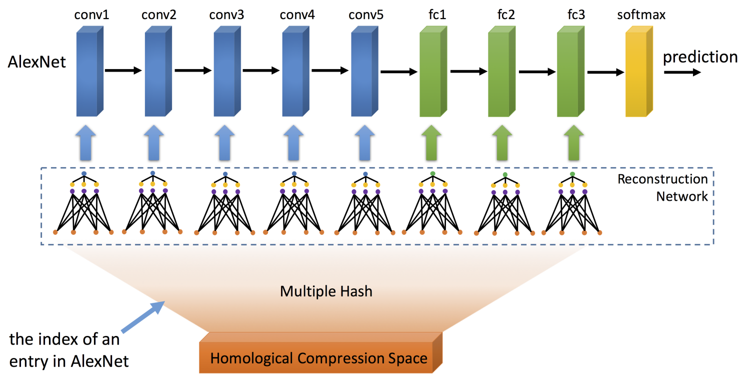
[1]http://cyan4973.github.io/xxHash/



(a) HashedNets on one layer



(b) HFH on one layer



(c) HFH on AlexNet

Figure 1: Illustrations of hashing approaches for neural networks compression. (a) and (b) illustrate HashedNets (Chen et al., 2015a) and our HFH on one layer's weight matrix. (c) shows HFH on AlexNet. (Best viewed in color)

- Eq. (3) of HashedNets assigns an equality between one virtual element and one hashed value, i.e., an implicit identity mapping. In contrast, this paper uses a multivariate function $g(\cdot)$ to map from multiple hashed values $\{\xi_u(i,j) w_{h_u(i,j)}\}_{u=1}^{U}$ to V_{ij}. Specifically,

$$V_{ij} = g\left(\left[\xi_1(i,j) w_{h_1(i,j)}, \ \ldots, \ \xi_U(i,j) w_{h_U(i,j)}\right]; \ \boldsymbol{\alpha}\right). \tag{4}$$

  Therein, $\boldsymbol{\alpha}$ is referred to as the parameters in $g(\cdot)$. Note that the input $\xi_u(i,j) w_{h_u(i,j)}$ is order sensitive from $u = 1$ to U. For optimization convenience, we choose $g(\cdot)$ to be a multi-layer feed forward neural network, and other multivariate functions can be considered as alternatives.

Figure 1(b) illustrates our HFH structure on one layer, which can be easily plugged in any matrices of DNNs. Note that $\boldsymbol{\alpha}$ in the reconstruction network $g(\cdot)$ is of a much smaller size compared to $\mathbf{w}$. For instance, a setting with $U = 4$ and a 1-layer $g(\cdot; \boldsymbol{\alpha})$ of $\boldsymbol{\alpha} \in \mathcal{R}^4$ performs already well enough in experiments. In other words, Eq. (4) just uses an ignorable amount of additional memory to describe a functional $\mathbf{w}$-to-$\mathbf{V}$ mapping, whose properties will be further explained in the sequel. Figure 1(c) further shows the HFH structure on multiple layers, taking AlexNet (Krizhevsky et al., 2012) for instance. Therein, no matter convolutional or fully-connected, all layers fetch hashed values from one single homological compression space.

3.2 Training Procedure

The parameters in need of updating include $\mathbf{w}$ in the compression and $\boldsymbol{\alpha}$ in $g(\cdot)$. Training HFH is equivalent to training a standard neural network, except that we need to forward/backward-propagate values related to $\mathbf{w}$ through $g(\cdot)$ and the virtual entries.

Forward Propagation. Substituting Eq. (4) into Eq. (1), we omit the super-script ℓ and get

$$z_i = b_i + \sum_{j=1}^{d} a_j V_{ij} = b_i + \sum_{j=1}^{d} a_j \cdot g\left(\left[\xi_1(i,j) w_{h_1(i,j)},\ \ldots,\ \xi_U(i,j) w_{h_U(i,j)}\right];\ \boldsymbol{\alpha}\right). \tag{5}$$

Backward Propagation. Denote $\mathcal{L}$ as the final loss function, e.g., cross entropy or squared loss, and suppose $\delta_i = \frac{\partial \mathcal{L}}{\partial z_i}$ is already available by back-propagation from layers above. The derivatives of $\mathcal{L}$ with respect to $\mathbf{w}$ and $\boldsymbol{\alpha}$ are computed by

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \sum_i \sum_j a_j \delta_i \frac{\partial V_{ij}}{\partial \mathbf{w}}, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{\alpha}} = \sum_i \sum_j a_j \delta_i \frac{\partial V_{ij}}{\partial \boldsymbol{\alpha}}, \tag{6}$$

where, since we choose $g(\cdot)$ as a multilayer neural network, derivatives $\frac{\partial V_{ij}}{\partial \mathbf{w}}$ and $\frac{\partial V_{ij}}{\partial \boldsymbol{\alpha}}$ can be calculated through the small network $g(\cdot)$ in a standard back-propagation manner.

Complexity. Concerning time and memory cost, HFH roughly has the same complexity as HashedNets, since the additional small network $g(\cdot)$ is quite light-weighted (e.g., 4 hashed inputs and 2 layers). One key varying factor is the way to implement multiple hash functions. On one hand, if they are calculated online, HFH requires little additional time if tackling them in parallel. On the other, if they are pre-computed and stored in dicts to avoid hashing time cost, the multiple hash functions of HFH demand more storage space. In application, we suggest to pre-compute hashes during offline training for speedup, and to compute hashing in parallel during online prediction for saving memory under limited hardware budget.

3.3 Property Analysis

In this part, we try to depict the properties of our HFH from several aspects to help understanding it, especially in comparison with HashedNets (Chen et al., 2015a).

Value Collision. It should be noted, both HashedNets and HFH conduct hashing prior to training, i.e., in a two-phase manner. Consequently, it would be unsatisfactory if hashing collisions happen among important values. For instance in natural language processing tasks, one may observe wired results if there are many hashing collisions among embeddings (which form a matrix) of frequent words, especially when they are not related at all. In the literature, multiple hash functions are known to perform better than one single function (Azar et al., 1994; Broder & Karlin, 1990; Broder & Mitzenmacher, 2001). In intuition, when we have multiple hash functions, the items colliding in one function are hashed differently by other hash functions.

Value Reconstruction. In both HashedNets and HFH, the hashing trick can be viewed as a reconstruction of the original parameter $\mathbf{V}$ from $\mathbf{w} \in \mathcal{R}^K$. In this sense, the approach with a lower reconstruction error is preferred[2]. Then we have at least the following two observations:

- **The maximum number of possible distinct values** output by hashing intuitively explains the modelling capability (Shi et al., 2009). For HashedNets, considering the sign hashing function $\xi(\cdot)$, we have at most $2K$ possible distinct values of Eq. (3) to represent elements in $\mathbf{V}$. In contrast, since there are multiple ordered hashed inputs, HFH has at most $(2K)^U$ possible distinct values of Eq. (4). Note that the memory size K is the same for both.
- **The reconstruction error** may be difficult to analyzed directly, since the hashing mechanism is trained jointly within the whole network. However, we observe $g\left(\left[\xi_1(i,j) w_{h_1(i,j)},\ \ldots,\ \xi_U(i,j) w_{h_U(i,j)}\right];\ \boldsymbol{\alpha}\right)$ degenerates to $g(\xi_1(i,j) w_{h_1(i,j)})$ if we assign zeros to all entries in $\boldsymbol{\alpha}$ unrelated to the 1st input dimension. Since $g(\xi_1(i,j) w_{h_1(i,j)})$ depends only on one single pair of hash functions, it is conceptually equivalent to HashedNets. Consequently, including HashedNets as a special case, HFH with freely adjustable $\boldsymbol{\alpha}$ is able to reach a lower reconstruction error to fit the final accuracy better.

[2]One might argue that there exists redundancy in $\mathbf{V}$, whereas here we could imagine $\mathbf{V}$ is already structured and filled by values with least redundancy.

Feature Hashing. In line with previous work (Shi et al., 2009; Weinberger et al., 2009), we compare HashedNets and HFH in terms of feature hashing. For specification clarity, we drop the sign hashing functions $\xi(\cdot)$ below for both methods, the analysis with which is straightforward by replacing K hereafter with $2K$.

- For HashedNets, one first defines a hash mapping function $\boldsymbol{\phi}_i^{(1)}(\mathbf{a})$, whose k-th element is

$$\left[\boldsymbol{\phi}_i^{(1)}(\mathbf{a})\right]_k \triangleq \sum_{j:h(i,j)=k} a_j, \quad \text{for} \quad k = 1, \dots, K. \tag{7}$$

  Thus z_i by HashedNets can be computed as the inner product (details c.f. Section 4.3 in (Chen et al., 2015a))

$$z_i = \mathbf{w}^\top \boldsymbol{\phi}_i^{(1)}(\mathbf{a}). \tag{8}$$

- For HFH, we first define a hash mapping function $\boldsymbol{\phi}_i^{(2)}(\mathbf{a})$. Different from a K-dim output in Eq. (7), it is of a much larger size K^U, with $\left(\sum_{u=1}^U k_u K^{(u-1)}\right)$-th element as

$$\left[\boldsymbol{\phi}_i^{(2)}(\mathbf{a})\right]_{\sum_{u=1}^U k_u K^{(u-1)}} \triangleq \sum_{\substack{j:h_1(i,j)=k_1\\ h_2(i,j)=k_2\\ \dots\\ h_U(i,j)=k_U}} a_j, \quad \text{for} \quad \forall u,\ k_u = 1, \dots, K. \tag{9}$$

  Second, we define vector $\mathbf{g}_{\boldsymbol{\alpha}}(\mathbf{w})$ still of length K^U, whose $\left(\sum_{u=1}^U k_u K^{(u-1)}\right)$-th entry is

$$[\mathbf{g}_{\boldsymbol{\alpha}}(\mathbf{w})]_{\sum_{u=1}^U k_u K^{(u-1)}} \triangleq g\left(w_{k_1}, w_{k_2}, \dots, w_{k_U};\ \boldsymbol{\alpha}\right), \quad \text{for} \quad \forall u,\ k_u = 1, \dots, K. \tag{10}$$

  Thus z_i by HFH can be computed as the following inner product

$$z_i = \mathbf{g}_{\boldsymbol{\alpha}}(\mathbf{w})^\top \boldsymbol{\phi}_i^{(2)}(\mathbf{a}). \tag{11}$$

  The difference between Eq. (8) and Eq. (11) further explains the above discussion about "the maximum number of possible distinct values".

Homology. Efficiency during searching appropriate compression ratios is the most important benefit brought by the homological property. Particularly, people usually have to search within candidate compression ratios either heuristically or in a Bayesian manner. The searching procedure becomes more and more exhaustive as the network depth increases, since usually compression ratios are various for different (kinds of) layers. For instance, the compression ratios of convolutional and fully-connected layers in both Han et al. (2015) and Guo et al. (2016) are quite different. In contrast for HFH, we only need to vary one scalar value K concerning the compression ratio. One may argue that these methods can adopt a same pruning ratio throughout layers, while preliminary experiments (not focused on and thus omitted in this paper) indicate equally pruning leads to inferior performances.

4 Related Work

Recent studies have confirmed the redundancy existence in the parameters of deep neural networks. Denil et al. (2013) decomposed a matrix in a fully-connected layers as the product of two low-rank matrices, so that the number of parameters decreases linearly as the latent dimensionality decreases. More structured decompositions Fastfood (Le et al., 2013) and Deep Fried (Yang et al., 2015) were proposed not only to reduce the number of parameters, but also to speed up matrix multiplications. Han et al. (2015) and Han et al. (2016) proposed to iterate pruning-retraining during training DNNs, and used quantization and fine-tuning as a post-processing step. Huffman coding and hardware implementation were also considered. In order to mostly keep accuracy, the authors suggested multiple rounds of pruning-retraining. That is, for little accuracy loss, we have to prune slowly enough and thus suffer from increased training time. More recently, dynamic network surgery was proposed in (Guo et al., 2016), which achieves state-of-the-art compressions by adopting a dynamic strategy to adapt the pruning procedure. Again, the most related work to ours is HashedNets (Chen et al., 2015a), which was then extended in (Chen et al., 2015b) to random hashing in frequency domain for compressing convolutional neural networks. Either HashedNets or HFH could be combined in conjunction with other techniques for better compression.

Extensive studies have been made on constructing and analyzing multiple hash functions, which have shown better performances over one single hash function (Broder & Karlin, 1990). One multi-hashing algorithm, d-random scheme (Azar et al., 1994), uses only one hash table but d hash functions, pretty similar to our settings. One choice alternative to d-random is the d-left algorithm proposed in Broder & Mitzenmacher (2001), used for improving IP lookups. Hashing algorithms for natural language processing are also studied in Goyal et al. (2012). Papers (Shi et al., 2009; Weinberger et al., 2009) investigated feature hashing (a.k.a. the hashing trick), providing useful bounds and feasible results.

5 Experiments

We conduct extensive experiments to evaluate HFH on DNN compression, especially in comparison with HashedNets. Codes for fully reproducibility will be open source in future after necessary polishment.

5.1 Environment Descriptions

Datasets. Five benchmark datasets (Larochelle et al., 2007) are considered here, including (1) the original `MNIST` hand-written digit dataset, (2) dataset `BG-IMG` as a variant to `MNIST`, (3) binary image classification dataset `CONVEX`, (4) dataset `CIFAR-10`, and (5) ImageNet `ILSVRC-2012`. For all datasets, we use prespecified training and testing splits. In particular, the original `MNIST` dataset has #train=60,000 and #test=10,000, while both `BG-IMG` and `CONVEX` have #train=12,000 and #test=50,000. `CIFAR-10` consists of 60k 32×32 color images of 10 object classes, i.e., 6k images per class, and there are 50k training and 10k testing images, respectively. ImageNet `ILSVRC-2012` has 1.2M training images and 50k validation images. Moreover, collected from a commercial search engine, a large scale dataset with billions of samples is used to learn DNNs for pairwise semantic ranking. We randomly split out 20% samples from the training data to form the validation set.

Methods and Settings. Chen et al. (2015a) compared HashedNets against several DNN compression approaches, and showed HashedNets performs consistently the best, including the low-rank decomposition (Denil et al., 2013). Under the same settings, we compare HFH with HashedNets[3] and baseline DNNs without compression. If not defined, all activation functions are chosen as ReLU.

5.2 Feedforward NNs for Classification

Focusing on classification by feedforward NNs, the performances of HFH are tested on `MNIST`, texttttBG-IMG and `CONVEX` in the following two scenarios. First, we compare the effects of varying compression ratio by HFH and HashedNets. Second, we compare different configurations of HFH itself, including the number U of seeds and the layer of reconstruction network $g(\cdot)$. Hidden layers within $g(\cdot)$ keep using tanh as activation functions.

5.2.1 Varying Compression Ratio

To test robustness, we vary the compression ratio with (1) a fixed virtual network size, and then with (2) a fixed memory size (i.e., the size of compression space). Three-layer (1 hidden layer) and five-layer (3 hidden layers) networks are investigated. In experiments, we vary the compression ratio geometrically within $\{1, \frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{64}\}$. For HFH, this comparison sticked to use 4 hash functions, 3-layer $g(\cdot)$, and without dual space hashing.

With Virtual Network Size Fixed. The hidden layer for 3-layer nets initializes at 1000 units, and for 5-layer nets starts at 100 units per layer. As the compression ratio ranges from 1 to 1/64 with a fixed virtual network size, the memory decreases and it becomes increasingly difficult to preserve the classification accuracy. The testing errors are shown in Figure 2, where standard neural networks with equivalent parameter sizes are included in comparison. HFH shows robustly effective compression against the compression ratios, and persistently produces better prediction accuracy than HashedNets. It should be noted, even when the compression ratio equals to one, HFH with the reconstruction network structure is still not equivalent to HashedNets and performs better.

[3]HashedNets code downloaded from http://www.cse.wustl.edu/∼wenlinchen/project/HashedNets/index.html

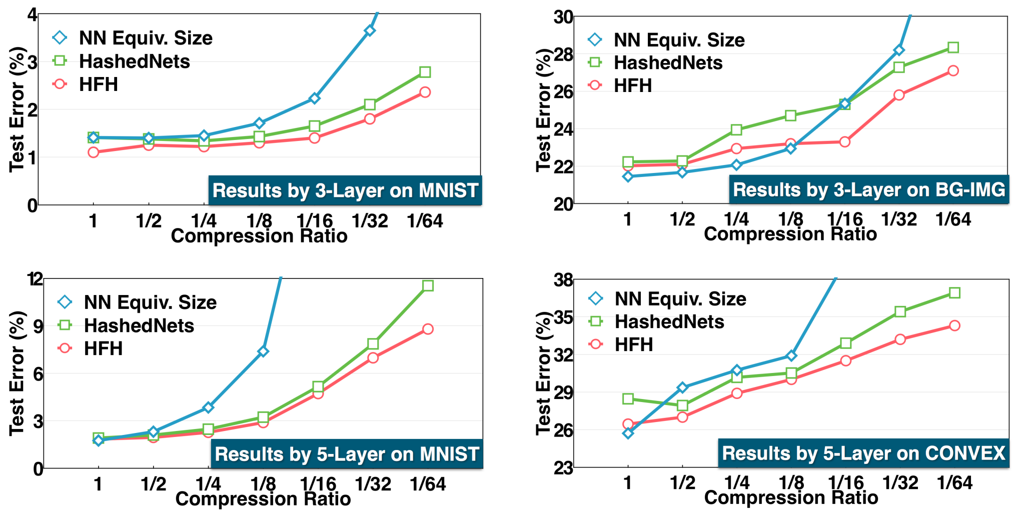


Figure 2: Testing errors by varying compression ratio with a fixed virtual network size.

With Memory Storage Fixed. We change to vary the compression ratio from 1 to 1/64 with a fixed memory storage size, i.e., the size of the virtual network increases while the number of free parameters remains unchanged. In this sense, we'd better call it expansion instead of compression. Both 3-layer and 5-layer nets initialize at 50 units per hidden layer. The testing errors in this scenario are shown in Figure 3. At all compression (expansion) ratios on each dataset, HFH performs better than or at least comparably well compared to HashedNets.

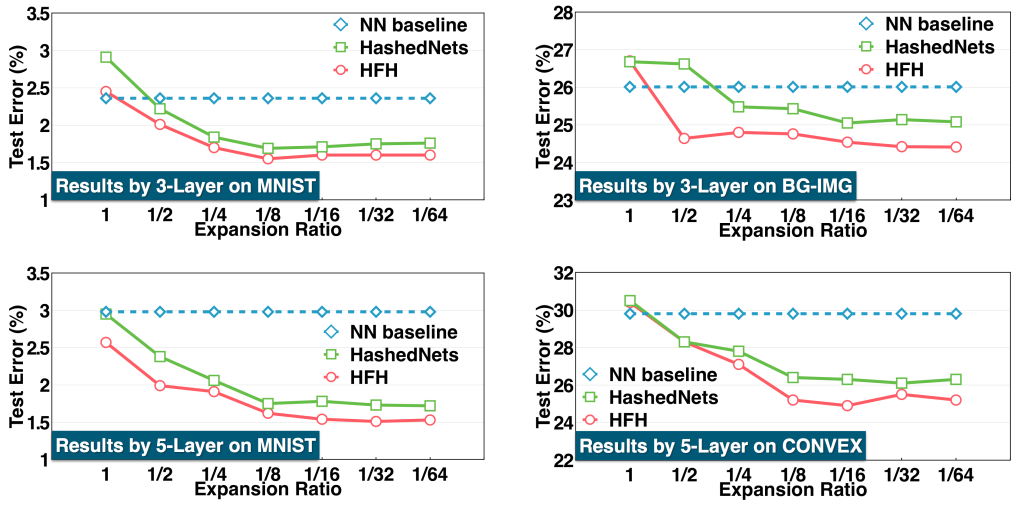


Figure 3: Testing errors by varying compression (expansion) ratio with a fixed memory storage.

5.2.2 Varying Configurations of HFH

On 3-layer nets with compression ratio $1/8$, we vary the configuration dimensions of HFH, including the number of hash functions (U), and the structure of layers of the reconstruction network $g(\cdot)$. Since it is impossible to enumerate all probable choices, U is restricted to vary in $\{2, 4, 8, 16\}$. The structure of $g(\cdot)$ is chosen from $2 \sim 4$ layers, with $U \times 1$, $U \times 0.5U \times 1$, $U \times U \times 0.5U \times 1$ layerwise widths, respectively. We denote Ux-Gy as x hash functions and y layers of $g(\cdot)$. Table 1 shows the performances of HFH with different configurations on `MNIST`.

We summarize key observations below. First, the series from index (0) to (1.x) fixes a 3-layer $g(\cdot)$ and varies the number of hash functions. As listed, more hash functions do not ensure a better accuracy, and instead U4-G3 performs the best, perhaps because too many hash functions potentially brings too many partial collisions. Second, the series from (0) to (2.x) fixes the number of hash functions and varies the layer number in $g(\cdot)$, where three layers performs the best mainly due to its strongest representability.

Table 1: Performances on `MNIST` by various configurations of HFH.

Index	Config	Test Error(%)
(0)	U4-G3	1.32
(1.1)	U2-G3	1.42
(1.2)	U8-G3	1.39
(1.3)	U16-G3	1.40
(2.1)	U4-G2	1.34
(2.2)	U4-G3	1.28

5.3 Convolutional NNs for Image Classification

We further evaluate the performances by hashing compression methods on deep convolutional NNs for image classification. Note that, we treat the convolutional layers same as fully-connected layers during hashing. A more meticulous alternative could be hashing over CNNs on `CIFAR-10` (Krizhevsky & Hinton, 2009) and ImageNet `ILSVRC-2012` datasets are reported here for comparison.

Particularly on `CIFAR-10`, a traditional CNN model with 3 conv layers and 2 fc layers is trained as the baseline, which stores roughly 4.8 million parameters and achieves 11.01% top-1 classification test error. As we vary the compression ratio, again equally over all layers, the top-1 test errors by HashedNets and HFH are illustrated in Figure 4. As shown, HFH can still achieve a test error at 18% with just 1 percent memory size of the baseline model, much better than HashedNets.

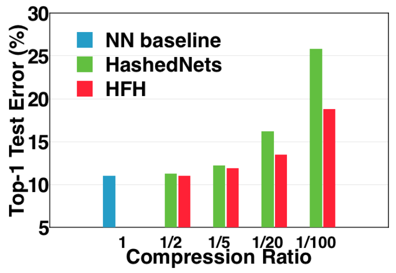


Figure 4: Performances by compressed convolutional NNs on `CIFAR-10` data.

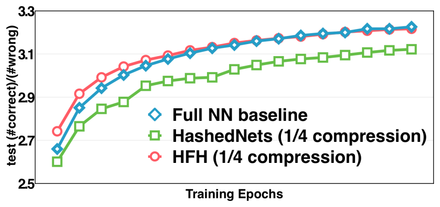


Figure 5: Performances for pairwise semantic ranking. Testing correct-to-wrong pairwise ranking ratios (the larger the better) are plotted versus the number of training epochs.

Table 2: Performances by AlexNet on ImageNet `ILSVRC-2012`. Note HashedNet and HFH are in a same comparable level, holding an equal compression ratio across all layers.

Method	Top-1 Test Error(%)	Parameters	Compression
baseline	42.78%	61.0M	1
HashedNet (Chen et al., 2015a)	61.71%	6.7M	1/9
HashedNet (Chen et al., 2015a)	64.17%	3.45M	1/17.7
HFH (ours)	48.05%	6.7M	1/9
HFH (ours)	51.99%	3.45M	1/17.7
Fastfood 16 (AD) (Yang et al., 2015)	42.90%	16.4M	1/3.7
SVD (Denton et al., 2014)	44.02%	11.9M	1/5
pruning (Han et al., 2015)	42.77%	6.7M	1/9
dynamic surgery (Guo et al., 2016)	43.09%	3.45M	1/17.7

On ImageNet `ILSVRC-2012` data, the standard AlexNet model (Krizhevsky et al., 2012) is chosen as the reference model, which has 61 million parameters across 5 conv layers and 3 fc layers. The top-1 classification test error by this baseline AlexNet is 42.78%. The current state-of-the-art compression results on AlexNet are achieved by pruning in (Han et al., 2015) with overall 1/9 compression ratio, and dynamic network surgery in (Guo et al., 2016) with overall 1/17.7 compression ratio, which are obtained by carefully tuning the compression ratios in different layers.

Applying HashedNets to match the above two compression ratios, the performances are much worse, as listed in Table 2. However, using HFH at the same compression ratios, the test errors are significantly reduced and even comparable to those by (Han et al., 2015) and (Guo et al., 2016). Again it is noticeable that: (1) both HashedNets and HFH hold the compression ratios across all layers equal, and thus the searching time cost is low; (2) the hashing techniques and the pruning strategies are not mutually exclusive, but provably to be used jointly for a better compression.

5.4 Convolutional NNs for Pairwise Semantic Ranking

Finally, we evaluate the performance of HFH on semantic learning-to-rank convolutional NNs. The data are collected from logs of a commercial search engine, with per clicked query-url being a positive sample and per non-clicked being a negative sample. There are totally around 45 billion samples. We adopt a deep convolutional structured semantic model similar to (Huang et al., 2013; Shen et al., 2014), which is of a siamese structure to describe the semantic similarity between a query and the title of an url. Stochastically for each query, the network is trained to optimize the pairwise ranking cross entropy for each randomly sampled pair of positive and negative samples.

The performance is evaluated by correct-to-wrong pairwise ranking ratio on testing set. In Figure 5, we plot the performance by a baseline network as training epoch proceeds. Results by HFH and HashedNets with 1/4 compression ratio are included for comparison. Still, HFH performs better than HashedNets throughout the training epochs, and even comparable to the full network baseline that requires 4 times of memory storage. The deterioration of HashedNets probably comes from many inappropriate collisions on word embeddings, especially for words of high frequencies.

6 Conclusion and Future Work

This paper presents a novel homologically functional hashing for neural network compression. Briefly, after adopting multiple low-cost hash functions to fetch values in a homological compression space, HFH employs a small reconstruction network to recover each entry in an matrix of the original network. The compression space is shared throughout all layers, and the reconstruction network is plugged into the whole network and learned jointly. HFH includes the recently proposed HashedNets (Chen et al., 2015a) as a degenerated special case. On multiple datasets, HFH demonstrates promisingly high compression ratios with little loss on prediction accuracy, especially significantly improved compared with HashedNets. As a simple and effective approach, HFH is expected to be a standard tool for DNN compression.

As future work, we aim for deeper analysis on the properties and bounds of HFH. More industrial applications are also expected, especially on mobile devices. Furthermore, HFH might be extended in a multi-hops hyper-structure. Briefly, imagining $\mathbf{w}$ in the compression space plays a *virtual* role similar to $\mathbf{V}$, we may adopt the homologically functional hashing structure once again on $\mathbf{w}$, so that $\mathbf{w}$ is still virtual and reconstructed by hashed values from another further compressed space.

References

Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. In *STOC*, 1994.

Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., 1995.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

A. Broder and A. Karlin. Multilevel adaptive hashing. In *SODA*, 1990.

Andrei Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. In *INFOCOM*, 2001.

Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *ICML*, 2015a.

Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing convolutional neural networks. In *NIPS*, 2015b.

Ronan Collobert, Jason Weston, Leon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *JMLR*, 12:2493–2537, 2011.

Graham Cormode and S. Muthukrishnan. An improved data stream summary: The Count-Min sketch and its application. *J. Algorithms*, 55:29–38, 2005.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training deep neural networks with binary weights during propagations. In *NIPS*, 2015.

Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. Predicting parameters in deep learning. In *NIPS*, 2013.

Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *NIPS*, 2014.

T. Dettmers. 8-bit approximations for parallelism in deep learning. In *ICLR*, 2016.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Domain adaptation for large scale sentiment classification: a deep learning approach. In *ICML*, 2011.

Amit Goyal, Hal III Daume, and Graham Cormode. Sketch algorithms for estimating point queries in NLP. In *EMNLP/CoNLL*, 2012.

Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *NIPS*, 2016.

Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *NIPS*, 2015.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *ICLR*, 2016.

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

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning Workshop*, 2014.

Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *CIKM*, 2013.

Yoonseop Kang, Saehoon Kim, and Seungjin Choi. Deep learning to hash with multiple representations. In *IEEE ICDM*, 2012.

Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. In *ICLR*, 2016.

A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

Brian Kulis and Trevor Darrell. Learning to hash with binary reconstructive embeddings. In *NIPS*, 2009.

Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluations of deep architectures on problems with many factors of variation. In *ICML*, 2007.

Quoc Viet Le, Tamas Sarlos, and Alexander J. Smola. Fastfood – approximating kernel expansions in loglinear time. In *ICML*, 2013.

Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In *arXiv:1312.4400*, 2013.

Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. In *ICLR*, 2016.

Z. Mariet and S. Sra. Diversity networks. In *ICLR*, 2016.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted Boltzmann machines. In *ICML*, 2010.

Marc'Aurelio Ranzato, Y-Lan Boureau, and Yann LeCun. Sparse feature learning for deep belief networks. In *NIPS*, 2007.

Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Gregoire Mesnil. A latent semantic model with convolutional-pooling structure for information retrieval. In *CIKM*, 2014.

Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and SVN Vishwanathan. Hash kernels for structured data. *JMLR*, 10:2615–2637, 2009.

Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. Learning to hash for indexing big data – a survey. *Proceedings of IEEE*, 104(1):34–57, 2016.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *ICML*, 2009.

David Malcolm Williams and Peter L Forey. *Milestones in Systematics*. CRC Press, 2004.

Zichao Yang, Marcin Moczulski, Misha Denil, Nando de Freitas, Alex Smola, Le Song, and Ziyu Wang. Deep fried convnets. In *ICCV*, 2015.