Similarities in Neuron Vectors and The Implications to CNN Inferences

Lin Ning
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
lning@ncsu.edu

Xipeng Shen
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
xshen5@ncsu.edu

***Abstract*—This technical report presents *deep reuse*, a method for speeding up CNN inferences by detecting and exploiting deep reusable computations on the fly. It empirically reveals the massive similarities among neuron vectors in activation maps, both within CNN inferences on an input and across inputs. It gives an in-depth study on how to effectively turn the similarities into beneficial computation reuse to speed up CNN inferences. The investigation covers various factors, ranging from the clustering methods for similarity detection, to clustering scopes, similarity metrics, and neuron vector granularities. The insights help create *deep reuse*. As an on-line method, *deep reuse* is easy to apply and adaptive to each CNN (compressed or not), and its input. Using no special hardware support or CNN model changes, this method speeds up inferences by 1.77-2X (up to 4.3X layer-wise) on the fly with virtually no (<0.0005) loss in accuracy.**

***Index Terms*—CNN, neuron vector, similarity, training, adaptive, deep reuse,**

I. Introduction

Deep Convolutional Neural Networks (CNN) have shown successes in many machine learning applications. However, inferences by CNN are compute-intensive. Recent years have seen numerous efforts in speeding up CNN inferences. Some propose special hardware accelerators [1]–[4], others build high performance libraries (e.g., CUDNN [1], MKL-DNN [2]), methods to compress models [5]–[7], Tensor graph optimizations [3], and other software optimizations.

However, despite the many efforts, faster CNN inference remains a pressing need, especially for many emerging CNN applications in latency or throughput sensitive domains. Real-time detection of objects, for instance, is essential for minimizing the latency of the autonomous vehicle control loop, which is crucial for driving safety. Surveillance image analysis gives relentless demands for higher inference speeds to reduce the time needed for analyzing millions of images streaming in from thousands of cameras.

To meet the demands, this work proposes *deep reuse*, a new technique for speeding up CNN inferences by discovering and exploiting deep reusable computations on the fly. *Deep reuse* is effective, halving the inference time of CNNs implemented on state-of-the-art high performance libraries and compression techniques, while causing virtually no (<0.0005) accuracy loss. It is meanwhile easy to use, requiring no special hardware support or CNN model changes, ready to be applied on today's systems.

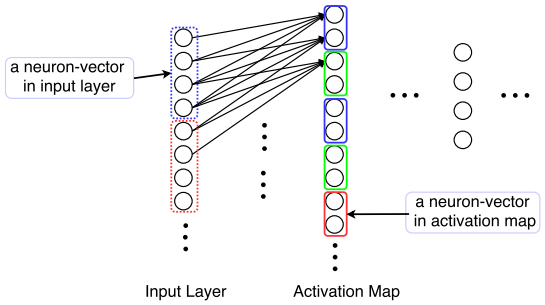


Figure 1. Illustration of neuron-vectors using a simple 1-D CNN with kernel size of 4 and one weight filter. Neurons in the same block form a neuron-vector. Block colors indicate the similarity of the neuron-vector values.

Deep reuse centers around similarities among neuron vectors. A *neuron vector* is made up of values carried by some consecutive neurons at a CNN layer. As Figure 1 illustrates, if the layer is an input image layer, a neuron vector contains the values of a segment of input image pixels; if the layer is a hidden layer, it contains a segment in its activation map.

The basic idea of *deep reuse* is to leverage similarities among neuron vectors, such that computation results attained on one neuron vector can be effectively reused for some other neuron vectors in CNN inferences. Figure 2 illustrates the basic form of such reuses. The eight 3-neuron vectors, represented by $\vec{x_{ij}}$, form four groups. Neuron vectors in a group are similar to each other. In this example, when the dot product of one of them is reused for all others in the group (e.g., $\vec{x_{11}} \cdot \vec{w_{11}}$ for $\vec{x_{31}} \cdot \vec{w_{11}}$ and $\vec{x_{41}} \cdot \vec{w_{11}}$), half of the computations in $X \times W$ could be saved.

Although the basic idea is straightforward to understand, a series of open questions must be answered for it to work beneficially for CNN:

- Are there strong similarities among neuron vectors in practice?
- How to effectively detect the similarities and leverage them?

[1]https://developer.nvidia.com/cudnn
[2]https://github.com/intel/mkl-dnn
[3]https://www.tensorflow.org/lite/

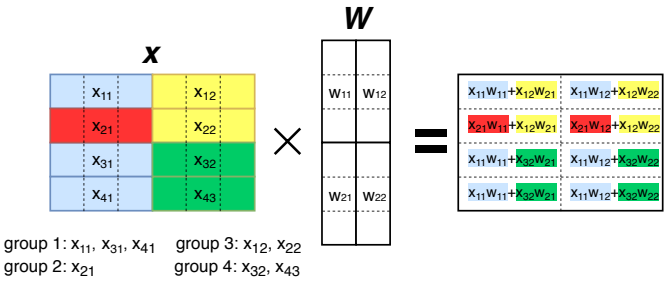


Figure 2. An example of the basic form of computation reuse across neuron vectors in convolution $X \times W$.

- Because activation maps change with inputs, finding similar neuron vectors must be done at inference time. The overhead is hence essential. How to minimize the overhead while maximizing the reuse benefits?
- Can the reuse bring significant speedups with no or little accuracy loss? Can it still apply if the CNNs are compressed?

In this work, we give a systematic exploration to these questions, and create *deep reuse* runtime optimization for CNN. The exploration is five-fold.

First, we conduct a series of measurements and confirm that a large amount of similarities exist among neuron vectors. Further, we find that, to fully uncover the similarities, one needs to consider the relations among neuron vectors not only inside an activation map, but also across the activation maps generated in different runs of the CNN.

Second, we experiment with several clustering methods, including K-means, Hyper-Cube, and Locality Sensitive Hashing (LSH), for detecting similarities among neuron vectors to form groups. The exploration identifies LSH as the most appealing choice for its low overhead and high clustering quality for neuron vectors.

Third, we investigate three clustering scopes to find deep reuse opportunities, including neuron vectors within the execution on one input, within the executions of a batch of inputs, and across executions in different batches. Through the process, we develop a *cluster reuse* algorithm to maximize the benefits of LSH-based clustering for all inputs.

Fourth, we experiment with two kinds of similarity distances, and a spectrum of neuron vector granularities by adjusting the length of neuron-vectors for clustering. We identify *angular cosine distance* as a better choice over Euclidean distance for *deep reuse*, and unveil the cost-benefit tradeoffs incurred by different neuron vector granularities.

Finally, we integrate all findings into *deep reuse* and apply this method to three popular CNN networks, CifarNet, AlexNet [8] and VGG-19 [9]. We measure both the end-to-end performance and accuracy, and provide detailed layer-wise performance analysis results in various settings. Results show that, *deep reuse* gives 3.19-4.32X layer-wise speedups and 1.77-2X whole network speedups with virtually no (<0.0005) accuracy loss.

To the best of our knowledge, this is the first study on systematically leveraging neuron vector-level computation reuses for speeding up CNN inferences. The produced *deep reuse* has several appealing properties: (1) All its optimizations happen at inference time on the fly, adaptive to every input to CNN. (2) It is compatible with model compression and other existing CNN optimization techniques. Its reuse across neuron vectors applies regardless whether the model is pruned or quantized. In Section V-D, we demonstrate that the method remains effective on compressed CNN models. (3) It is easy to apply, requiring no special hardware support or CNN model changes, and meanwhile, it is compatible with most exiting hardware or software accelerations, as its optimized CNN still has matrix multiplications (on smaller matrices) as its core computations. (4) It offers simple knobs (neuron vector granularity) allowing users to tune to adjust the tradeoff between accuracy and time savings. (5) Finally, it brings significant performance benefits with no or little accuracy loss.

II. Background and Notations

The convolutional layer of CNN takes an input tensor with size $N_b \times I_w \times I_h \times I_c$ and outputs a tensor with size $N_b \times O_w \times O_h \times M$. Here, N_b is the batch size. I_w, I_h and I_c are the width, height and channel size of the input to the convolutional layer. The input could be an input image or an activation map. O_w, O_h and M are the width, height and channel size of the corresponding output.

Given a stride size of s, a kernel width of k_w and a kernel height of k_h, the input tensor is unfolded into a large input matrix $\mathbf{x}$ with dimension $N \times K$, where, when the stride s is 1, $N = N_b \cdot (I_w - k_w + 1) \cdot (I_h - k_h + 1)$ is the number of rows for a batch of inputs and $K = I_c \cdot k_h \cdot k_w$ is the kernel weight matrix size. The number of rows corresponding to one input is $N_{img} = \frac{N}{N_b}$. The weight of the convolutional layer is represented with a tensor W with size $K \times M$, where M is the number of weight filters. The output $\mathbf{y}$ without adding the bias is then computed with $\mathbf{y} = \mathbf{x} \cdot W$. The main computation comes from the matrix-matrix multiplication, which has a complexity of $O(N \cdot K \cdot M)$.

III. Deep Reuse for CNN

This section starts with the basic idea of *deep reuse* and the key conditions for the idea to work beneficially for CNN, then describes our detailed design of *deep reuse*, and finally concludes with a discussion on the properties of *deep reuse* and its relationship with some other common CNN optimizations.

A. Basic Idea and Key Conditions

The basic idea of *deep reuse* is grouping similar neuron vectors into clusters and using the cluster centroids as the representatives for computations. For example, as illustrated in Fig. 3, the original computation is $\mathbf{y} = \mathbf{x} \cdot W$. With *deep reuse*, we may consider each row of $\mathbf{x}$ as a neuron vector denoted with $\mathbf{x}_i$. First, we group the 4 neuron vectors into 3 clusters and compute the centroid vectors $\mathbf{x}_c$. The centroid vectors are

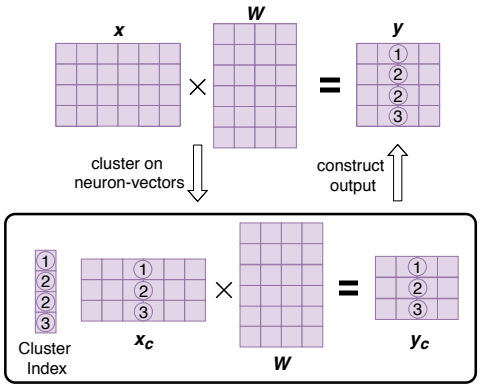


Figure 3. Illustration of using *deep reuse* to reduce the computation cost (*whole-vector clustering*). Numbers 1,2 and 3 are the cluster IDs.

taken as representatives. In this example, both $\mathbf{x}_2$ and $\mathbf{x}_3$ are represented by the value of $\mathbf{x}_{c,2}$ (the centroid vector of cluster 2). The next step is to do the computation using the centroids $\mathbf{y}_c = \mathbf{x}_c \cdot W$. The full results are then attained by reusing the outputs of the centroid vectors for each cluster member; that is, $\mathbf{y}_2 = \mathbf{y}_3 = \mathbf{y}_{c,2}$ in this example.

Computation Savings: In a general case, given an input matrix $\mathbf{x}$, we could group all the neuron vectors into $|C|$ clusters. The corresponding centroid vectors form a new matrix $\mathbf{x}_c$ with size of $|C| \times K$. Since we only need to compute $\mathbf{y}_c = \mathbf{x}_c \cdot W$, the computation complexity becomes $O(|C| \cdot K \cdot M)$. If $|C| << N$, we could save a large number of computations. In the rest of the technical report, we use *remaining ratio* to indicate the fraction of computations left after the optimization. It is defined as

Remaining ratio: $r_c = \frac{|C|}{N}$.

The smaller r_c is, the more computations are saved.

Key Conditions: For the idea to actually benefit CNN inferences, three conditions must hold.

1) There is a substantial amount of strong similarities among neuron vectors.
2) The time needed by detecting and leveraging the similarities should be much smaller than the time savings it brings to CNN. It is important to notice that *deep reuse* is an on-line process. Because activation maps change with each input, the detection of similarities among the neuron vectors in an activation map must happen on the fly at the inference time. The same is the operations for saving the dot products of cluster centroids and for retrieving them for reuse. Therefore, it is essential that the overhead of these introduced operations is kept much smaller than the time savings they bring to CNN.
3) The reuses cause no or negligible loss of inference accuracy.

The first condition needs empirical studies on actually CNNs to check. A brief summary of our observations is that on three popular CNNs (CifarNet, AlexNet, VGG-16) and two datasets (Cifar10, ImageNet), our study consistently finds strong similarities among neuron vectors across every convolution layer both within the inference on one input and across inputs. We put the details into Section V and will elaborate them later. In this section, we concentrate on our design of *deep reuse* for effectively finding the similarities on the fly and turning them into better inference speed.

B. *Design of* Deep Reuse

To fully capitalize on neuron vector similarities and at the same time achieve good trade-off between runtime overhead and the gains, the design of *deep reuse* employs a set of features, including an efficient runtime clustering algorithm, the capability in harnessing deep reuse opportunities in three scopes, the flexibility in accommodating various neuron vector granularities, and the use of a similarity metric that empirically proves effective. We explain each of the features next.

1) Clustering Method: Choosing an appropriate clustering method is essential for the effectiveness of *deep reuse*. First, the method should be able to give good clustering results for effectively capturing the similarities between neuron vectors. Second, it must be lightweight such that it does not introduce too much overhead at runtime.

In this work, we studied several different methods, and identified *Locality Sensitive Hashing* (LSH) as the clustering method for *deep reuse*.

LSH is widely used as an algorithm for solving the approximate or exact Nearest Neighbor problem in high dimension space [10]–[14]. For each input vector $\mathbf{x}$, a hashing function h is determined by a random vector $\mathbf{v}$ in the following way:

$$h_{\mathbf{v}}(\mathbf{x}) = \begin{cases} 1 & if \quad \mathbf{v} \cdot \mathbf{x} > 0 \\ 0 & if \quad \mathbf{v} \cdot \mathbf{x} \leq 0 \end{cases} \tag{1}$$

Given a series of random vectors, LSH maps an input vector into a bit vector.

Using LSH, input vectors with smaller distances have a high probability to be hashed into the same bit vector. Thus, when applying LSH into our context, we consider each bit vector as a cluster ID and all the neuron vectors mapped to the same bit vector form a cluster.

Our experiments (Section V) show that LSH can be applied to both short and long vectors while achieving good accuracy. The hashing itself takes some time. With LSH applied, the operations at a convolution layer now consist of two parts: hashing and the centroid-weight multiplication. If having $|H|$ hashing functions, the computation complexity is $O(N \cdot K \cdot |H| + |C| \cdot K \cdot M)$. Comparing to the original complexity of $O(N \cdot K \cdot M)$, LSH brings benefit only if $|H| << M(1 - r_c)$, where r_c is the *remaining ratio* N_C/N.

In addition to LSH, we have explored two other clustering algorithms: K-means, and Hyper-Cube clustering. As one of the most classical clustering algorithm, K-means could give us relatively good clustering results, which makes it a good choice for studying the similarity between neuron vectors. However, K-means is not practically useful for reducing computations because of its large clustering overhead. Even though in some cases, we could recover the accuracy of the original

network with a very small remaining ratio ($r_c < 0.1$), the computation cost of running K-means itself is even larger than the original matrix-matrix multiplication. Therefore, we only use K-means to study the similarity between neuron-vectors and explore the potential of our approach.

The other alternative method we explored is Hyper-Cube clustering. It regards the data space as a D-dimension hyper-cube, and clusters neuron vectors by applying simple linear algebra operations to each of the selected D primary dimensions of each neuron vector. Let $x_i^{(j)}$ be the j^{th} ($j = 1, 2, \cdots, D$) element of a neuron vector $\vec{x_i}$. Hyper-cube clustering derives a bin number $b_i^{(j)}$ for it, equaling

$$b_i^{(j)} = B \cdot (x_i^{(j)} - \min_{i' \leq N} x_{i'}^{(j)}) / (\max_{i' \leq N} x_{i'}^{(j)} - \min_{i' \leq N} x_{i'}^{(j)}),$$

where, B is the total number of bins for each dimension. The cluster ID of the neuron vector $\vec{x_i}$ is set as $C_{\vec{x_i}} = [b_i^{(1)}, b_i^{(2)}, \cdots, b_i^{(D)}]$. The number of clusters, D^B, could be large, depending on D and B. Our experiments show that in practice, often many bins are empty and the total number of real clusters are much smaller than D^B.

Hyper-Cube is lightweight since the cluster assignment is simple and the complexity of computing the cluster ID for each neuron-vector is only $O(D)$. However, our experiments (Section V) show that this method only works well for short neuron vectors. Reuse on short neuron vectors involves many adding operations to sum the partial products together. As a result, computation savings by Hyper-Cube are less significant than by LSH as our experiments in Section V will report.

LSH has an additional distinctive advantage over the other two clustering algorithms. It applies seamlessly to all scopes of similarity detection, as explained next.

2) Clustering Scopes: To detect the full reuse opportunities among neuron vectors, *deep reuse* supports the detection of similarities of neuron vectors in three levels of clustering scopes: within one input, within a batch of inputs, and across batches.

For the single-input or single-batch level, the detection can be done simply by applying the clustering algorithm to all the neuron-vectors within an input or within a batch directly. There are extra complexities when the scope expands across batches. Because inputs from different batches come at different times, it is often impractical to wait for all the inputs to apply the clustering. *Deep reuse* addresses the complexity through *cluster reuse*.

Cluster Reuse: The purpose of *cluster reuse* is to allow for neuron-vectors from different input batches to share the computation results of the same cluster centroid. If K-means or Hyper-Cube clustering are used, it is hard to reuse the clusters attained on one batch for another batch as they build different clusters for different batches. But with LSH, it can be achieved naturally.

With LSH, we can reuse an existing cluster if a new neuron vector is hashed to a bit vector that has appeared before. No matter which batches two neuron vectors belong to, if they map to the same bit vector, they are assigned with the same cluster ID and thus to the same cluster. We need to use the same family of hash function $\mathcal{H}$ to do the hashing for all the neuron vectors across batches.

Algorithm 1 Cluster Reuse

Input: input matrix $\mathbf{x}$ with dimension $N \times K$; a set of cluster ID S_{id}; the set of outputs O_{id} corresponding to S_{id}.
Algorithm:
for all row vectors $\mathbf{x}_i$ **do**
  Apply LSH to get the cluster id ID_i
end for
for $i = 1$ **to** N **do**
  if $ID_i \in S_{id}$ **then**
    reuse $O_{id=ID_i}$
  else
    insert ID_i into S_{id}
    $O_{id=ID_i} = \mathbf{x_i} \cdot W$
    insert $O_{id=ID_i}$ into O_{id}
  end if
end for

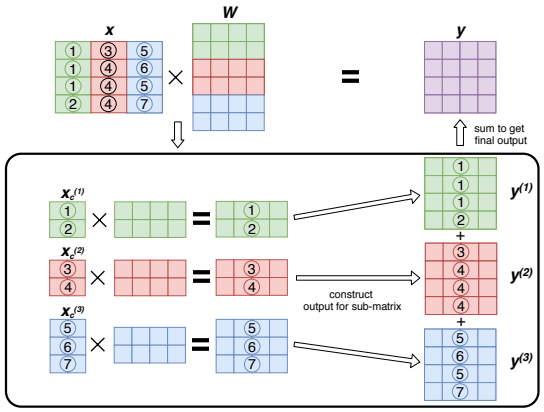


Figure 4. Illustration of *deep reuse* with a smaller clustering granularity (*sub-vector clustering*).

Algorithm 1 provides some details on how to reuse the clusters and the corresponding results with LSH. The algorithm employs a set S_{id} to store all previously appeared bit vectors (the cluster IDs) and an array O_{id} to store all the outputs computed with those cluster centroids. When a new batch of inputs come, it first maps all the neuron vectors to bit vectors using LSH. Then for neuron vectors mapped to the existing clusters, it can reuse the corresponding outputs. For those mapped to a new cluster, it first computes the centroid $\mathbf{x}_c$ and calculates the output of $\mathbf{x}_c \cdot W$, which are used in updating S_{id} and O_{id}. Let R be the averaged cluster reuse rate for a batch. The computation complexity becomes $O(N \cdot K \cdot |H| + (1 - R) \cdot |C| \cdot K \cdot M)$ (if one neuron vector is a whole row in an activation map.) A larger cluster reuse rate helps save more computations.

3) Clustering Granularity: In the basic scheme shown in Figure 3, each row vector in matrix $\mathbf{X}$ is taken as a neuron

vector. Our experiments indicate that a smaller clustering granularity with a shorter neuron-vector length can often expose more reuse opportunities. We refer to the first case as the *whole-vector clustering* and the second case as the *sub-vector clustering*. *Deep reuse* supports both cases, allowing a flexible adjustment of the granularity, useful for users to attain a desired cost-benefit tradeoff.

Fig. 4 illustrates the procedures of *deep reuse* with *sub-vector clustering*. The input matrix $\mathbf{x}$ is divided into three sub-matrices $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$ and $\mathbf{x}^{(3)}$. The neuron vectors used for clustering have a length of 2. For each sub-matrix, *deep reuse* groups the neuron vectors into clusters, and computes the centroids matrix $\mathbf{x}_c^{(i)}$ and the corresponding output $\mathbf{y}_c^{(i)}$. Then it reconstructs the output $\mathbf{y}^{(i)}$ for each sub-matrix. In comparison to the *whole-vector clustering* (Fig. 3), the *sub-vector clustering* has one more step: the result $\mathbf{y}$ is computed by adding the partial results together, as $\mathbf{y} = \mathbf{y}^{(1)} + \mathbf{y}^{(2)} + \mathbf{y}^{(3)}$.

Since the clustering algorithms usually work better on low dimension data, we see better clustering results with a smaller clustering granularity. However, a smaller neuron-vector length results more neuron vectors, and hence more adding operations. Hence, it does not always save more computations. Assuming each input row vector is divided into N_{nv} neuron vectors and the size of each neuron vector is L. We have $N_{nv} \cdot L = K$; the computation introduced by all the adding operations is $O(N \cdot \frac{K}{L} \cdot M)$, where K, M, N are the length of a weight filter, the number of weights filters and the number of rows for a batch of inputs. The average number of clusters when using the *sub-vector clustering* is $|C|_{nv,avg} = \frac{1}{N_{nv}} \sum_{j=1}^{N_{nv}} |C|_{nv,j}$. So the *remaining ratio* is $r_c = \frac{|C|_{nv,avg}}{N}$. The computation complexity of using the *sub-vector clustering* becomes $O((r_{c,nv} + \frac{1}{L}) \cdot N \cdot K \cdot M)$. With a smaller clustering granularity, we are more likely to have a smaller $r_{c,nv}$ but a larger $\frac{1}{L}$. A balance between these two parts is needed to minimize the overall computations.

Deep reuse exposes the clustering granularity as a user definable parameter. Its default value is the channel size of the corresponding activation map, but users can set it differently. One possible way users may use is to simply include it as one of the hyper-parameters of the CNN to tune during the CNN model training stage.

4) Similarity Metric: In this work, we experimented with two different similarity metrics between neuron vectors: the Euclidean distance and the angular cosine distance. For Euclidean distance, the clustering result is decided by evaluating $\|\mathbf{x}_i - \mathbf{x}_j\|$ of any two vectors $\mathbf{x_i}$ and $\mathbf{x_j}$. For the angular cosine distance, the vectors are first normalized ($\hat{\mathbf{x}}_\mathbf{i} = \frac{\mathbf{x_i}}{\|\mathbf{x_i}\|}$) before the distance ($\|\hat{\mathbf{x}}_i - \hat{\mathbf{x}}_j\|$) is computed. We find that clustering using angular cosine distance usually performs better than clustering using Euclidean distance (Section V-C). *Deep reuse* hence uses angular cosine distance by default.

C. Properties of Deep Reuse

As an optimization technique, *deep reuse* features several appealing properties:

First, because it detects similarities on the fly, it is adaptive to every CNN and each of its inputs. The clusters are not built on offline training inputs, but formed continuously as the CNN processes its inputs. This adaptivity helps *deep reuse* effectively discover reuse opportunities in actual inferences.

Second, *deep reuse* is generally applicable. It works on CNNs despite their structural differences or compression status. As Section V reports, it gives consistent speedups on compressed and uncompressed CNNs.

Third, it is easy to apply. It does not require special hardware support or CNN model changes, but at the same time, is compatible with common CNN accelerators—hardware or software based—as its optimized CNN still has matrix multiplications as its core computations.

Fourth, it offers simple knobs, through which users can easily adjust the tradeoff between accuracy and time savings. The knobs include the neuron vector granularity and the strength of the clustering (i.e., the size of the hashing function family used in LSH). Users can simply include these knobs as part of the hyperparameters of the CNN to tune in the training stage.

Finally, it brings significant speedups with no or little accuracy loss, as Section V will report.

IV. Error Analysis

This section analyzes the influence brought to the output layer by the errors introduced by *deep reuse* at a hidden or input layer. Let $F^{(n)}$ be a neural network with n layers. For a layer i, let $x_j^{(i)}$ be the input row vector in row j, $W^{(i)}$ be the model parameter matrix and $y_{j0}^{(n)}$ be the final output in the original network. *Deep reuse* uses the centroid $x_{jc}^{(i)}$ to replace $x_j^{(i)}$. The introduced error is $Err_c^{(i)} = \sum_j \|x_{jc}^{(i)} - x_j^{(i)}\|^2$. The final output becomes $y_j^{(n)}$ and the corresponding error is $\delta(y^{(n)}) = \sum_j \|y_j^{(n)} - y_{j0}^{(n)}\|^2$. If we only apply the reuse on a single layer i, the final output error is bounded by

$$\delta(y^{(n)}) \leq Err_c^{(i)} \prod_{j=i}^{n} \|W_j\|^2 \tag{2}$$

If the reuse applies to all the layers, the final output error bound is

$$\delta(y^{(n)}) \leq \sum_{i=1}^{n} Err_c^{(i)} \prod_{j=i}^{n} \|W_j\|^2 \tag{3}$$

The analysis shows that the influence from the error at one layer to the output layer is a linear relation to the error made at that layer. Next section will show that, in practice, the introduced errors have only marginal influence on CNN inference accuracy.

V. Experimental Results

To examine the existence of neuron vector similarities and to evaluate the efficacy of the *deep reuse*, we experiment with three different networks: CifarNet, AlexNet [8] and VGG-19 [9]. As shown in Table I and the first four columns of Table II, these three networks have a range of sizes and complexities.

Table I
BENCHMARK NETWORKS

NETWORK	DATASET	# CONVLAYERS	IMAGE ORDER
CIFARNET	CIFAR10	2	RANDOM
ALEXNET	IMAGENET	5	RANDOM
VGG-19	IMAGENET	16	RANDOM

The first network works on small images of size 32×32, the other two work on images of 224×224. For all the experiments, the input images are randomly shuffled before being fed into the network.

The baseline network implementation we use to measure the speedups comes from the *slim* model [4] in the TensorFlow framework [5]. We implement our optimized CNNs by incorporating *deep reuse* into the TensorFlow code. Both the original and our optimized CNNs automatically leverage the state-of-the-art GPU DNN library cuDNN [6] and other libraries that TensorFlow uses in default. All the experiments are done on a machine with an Intel(R) Xeon(R) CPU E5-1607 v2 and a GTX1080 GPU.

For each of the networks, we first apply our approach to only a single convolutional layer to measure the single layer speedups and the corresponding accuracy. Then we measure the end-to-end speedups for the full networks. The neuron-vector length L and the number of hashing functions H used in *deep reuse* are determined for each convolution layer as part of the hyperparameters tuning process of CNN training. Sections V-A and V-B presents all the speedup results. In Section V-C, we use the results of applying K-means clustering on CifarNet as examples to demonstrate how different scopes, granularities and similarity distances affect the performance of the *deep reuse* in terms of the r_c-accuracy relationship. Here $r_c = |C|/N$ is the remaining ratio as defined in Section III-A.

Section V-D reports the speedups when *deep reuse* applies to CNNs after model compression [5], demonstrating its complementary relations with model quantization and compression. Section V-E gives a head-to-head comparison with *perforated CNN*, the work most closely related to this study.

All timing results are the average of 20 repeated measurements; variances across repeated runs are marginal unless noted otherwise.

A. Single Layer Speedup

For every single convolutional layer of the three networks, we run experiments using all the three clustering methods with a range of different clustering configurations and collect the r_c-accuracy relationship. For the purpose of study, for both of the Hyper-Cube and LSH clustering methods, we select the configurations that can recover the accuracy while reducing the maximum amount of computations according to the computation complexity analysis. We measure the speedups of every single layer using these configurations.

For example, when using LSH with *sub-vector clustering*, the computation complexity is $O(N \cdot K \cdot |H| + r_c \cdot N \cdot K \cdot M + \frac{1}{L} \cdot N \cdot K \cdot M)$. The number of hashing functions $|H|$ and the neuron-vector length L are the parameters for clustering configurations. For each pair of the $|H|$ and L, there is a corresponding r_c. Given the r_c-accuracy relationship, we find the $|H|$ and L pairs that can recover the accuracy or give the highest accuracy if no configurations recover the full accuracy. Among these configurations, we then use the one that gives the maximum computations savings ($M/(|H| + r_c \cdot M + M/L)$) to measure the speedup.

a) Speedups from intra-batch reuse: Columns $5-11$ in Table II report the speedups that the reuse method produces for each convolutional layer when the reuse applies within a batch (i.e., *cluster reuse* is not used). On average, the method obtains up to 1.63X speedups with Hyper-Cube clustering and 2.41X with LSH clustering. The speedups come with no accuracy loss.

The result shows that on all the layers except the first convolutional layer of VGG-19, LSH brings larger speedups than the Hyper-Cube clustering does. Since LSH recovers the accuracy with longer neuron-vectors as shown in column 9 of Table II, it introduces less adding operations, making *deep reuse* more efficient. Therefore, LSH always has a higher *remaining ratio* and gives more speedups.

Extra Benefits from inter-batch *Cluster Reuse* Column 12 in Table II shows that cluster reuse could bring even more speedups. Although it introduces small accuracy loss (less than 3% if only quantizing one of the convolutional layers), it is still attractive to tasks that could tolerate such accuracy loss.

Fig 5 shows the cluster reuse rate (R) for each convolutional layer of CifarNet across batches. The reuse rate (the fraction of neuron-vectors in current batch that falls into the existing clusters) increases from 0 to around 0.98 after processing 20 batches. We also observed similar patterns in the convolutional layers of AlexNet and VGG-19. The reuse rates all reach over 0.95. This high cluster reuse rate is the main reason for the large increases of the speedups (from an average of 2.4X to 3.6X for AlexNet and from an average of 2.3X to 3.4X for VGG-19).

For CifarNet, cluster reuse brings only modest extra speedups. It is because the *remaining ratio* of the two convolutional layers are already very small (about 0.01). There are few computations left that can be saved by cluster reuse in this case.

Based on the previous computational complexity analysis, the computations being saved by cluster reuse-based LSH is $M/(|H|+R\cdot r_c\cdot M+M/L)$. Therefore, when r_c plays a more major role than $|H|$ and M/L in the computational complexity, cluster reuse increases speedups more. This conclusion is confirmed by the results in Table II.

[4]https://github.com/tensorflow/models/tree/master/research/slim
[5]https://github.com/tensorflow/tensorflow
[6]https://developer.nvidia.com/cudnn

Table II
SINGLE LAYER SPEEDUPS. K IS THE KERNEL SIZE AND M IS THE NUMBER OF WEIGHT FILTERS. L REFERS TO THE NEURON-VECTOR LENGTH. $r_c = |C|/N$ IS THE REMAINING RATIO.

NETWORK	CONVLAYER	K	M	NO CLUSTER REUSE							CLUSTER REUSE
				HYPERCUBE			LSH				LSH
				L	r_c	SPEEDUP	H	L	r_c	SPEEDUP	SPEEDUP
CIFARNET	CONV1	75	64	3	0.03	1.57X	15	5	0.01	1.58X	1.59X
	CONV2	1600	64	10	0.11	1.68X	10	10	0.01	2.51X	2.58X
			AVG			1.63X				2.05X	2.09X
ALEXNET	CONV1	363	64	11	0.14	0.94X	15	11	0.13	1.63X	1.96X
	CONV2	1600	192	5	0.11	2.13X	15	20	0.18	2.84X	4.23X
	CONV3	1728	384	6	0.11	1.22X	15	12	0.15	2.58X	3.92X
	CONV4	3456	384	6	0.13	1.14X	15	12	0.17	2.76X	3.99X
	CONV5	3456	256	6	0.11	1.14X	15	24	0.15	2.23X	4.12X
			AVG			1.31X				2.41X	3.64X
VGG-16	CONV1	27	64	9	0.05	2.89X	15	9	0.08	2.35	2.83
	CONV2	576	64	6	0.05	1.37X	15	16	0.11	2.06X	2.59X
	CONV3	576	128	3	0.03	1.07X	15	16	0.13	1.83X	2.48X
	CONV4	1152	128	3	0.03	0.91X	15	16	0.11	1.95X	2.49X
	CONV5	1152	256	3	0.02	0.88X	10	16	0.09	2.22X	3.39X
	CONV6	2304	256	3	0.02	0.89X	10	16	0.11	2.03X	3.38X
	CONV7	2304	256	3	0.02	0.84X	10	16	0.06	2.79X	3.31X
	CONV8	2304	256	3	0.02	0.85X	10	16	0.09	2.52X	3.40X
	CONV9	2304	512	3	0.03	0.91X	8	16	0.05	3.19X	4.05X
	CONV10	4068	512	3	0.03	0.85X	8	24	0.1	2.85X	4.32X
	CONV11	4068	512	3	0.03	0.92X	8	24	0.11	2.37X	4.16X
	CONV12	4068	512	3	0.03	0.89X	8	24	0.13	2.44X	4.13X
	CONV13	4068	512	3	0.02	0.88X	8	24	0.2	1.86X	3.26X
	CONV14	4068	512	3	0.02	0.91X	8	24	0.18	1.81X	3.28X
	CONV15	4068	512	3	0.02	0.91X	8	24	0.18	1.81X	3.26X
	CONV16	4068	512	3	0.02	0.85X	8	24	0.16	2.02X	3.31X
			AVG			1.05X				2.26X	3.35X

Table III
END-TO-END FULL NETWORK SPEEDUPS AND ACCURACY LOSS. (NEGATIVE ERRORS MEANS IMPROVEMENTS OF ACCURACY)

NETWORK	LSH WITH NO CLUSTER REUSE		
	SPEEDUP	ACCURACY	ACCURACY LOSS
CIFARNET	1.77X	0.7892	-0.0011
ALEXNET	2.00X	0.5360	-0.0002
VGG-19	1.89X	0.7118	+0.0005

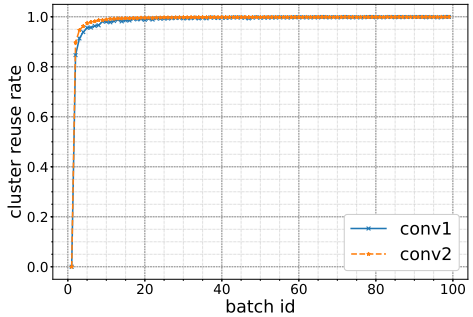
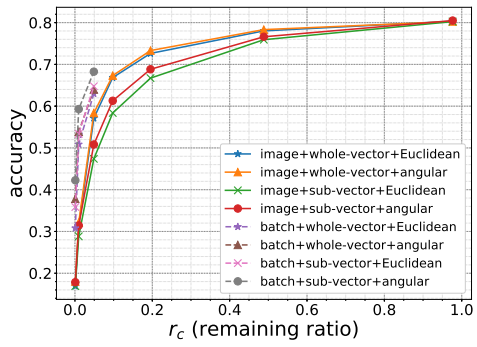


Figure 5. Cluster reuse rate in CifarNet

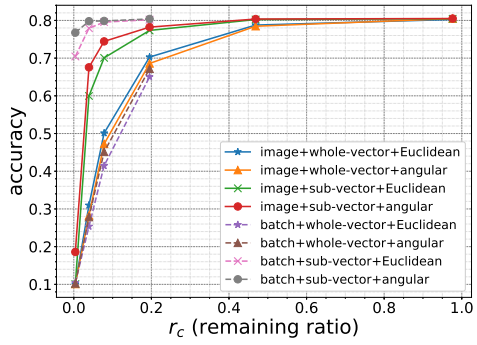
B. End-to-End Speedup

In measuring the end-to-end speedups of the full network, for better accuracy, we use LSH-based —em deep reuse without cluster reuse. We determine the clustering configurations of each convolutional layer in the network by simply adopting the configurations from the single layer experiments since they cause no accuracy loss.

As shown in Table III, our approach obtains up to 2X speedups on the full network. The maximum extra error it brings is 0.0005. The speedups of the full network is relatively smaller than those of a single convolutional layer as there are other layers (e.g., ReLU, pooling) in a CNN.



(a) conv1



(b) conv2

Figure 6. Comparison of the r_c−accuracy relationships for K-means clustering with different configurations on CifarNet at different scopes ('image', 'batch'), granularities (sub-vector or whole-vector), distances (angular or Euclidean). The legend has the patten of scope+granularity+distance.

C. Discussion on Clustering Scope, Granularity and Similarity Distance

Experiments show that we could recover the accuracy with a small *remaining ratio* r_c. This validates the existence of substantial neuron vector similarities and their potential for effective reuse. Besides clustering methods, clustering scope, granularity and similarity distance also affect the efficacy of *deep reuse* in detecting such similarities. This section discusses these connections, using the r_c−accuracy results of applying K-means based clustering on CifarNet as an example. Given the same r_c value, a higher accuracy means better identification of the similarities.

Scope: Section V-A has already reported the substantially more saving opportunities that inter-batch reuse can bring and the corresponding speedups. In this part, we provide a detailed study on the effects when the reuse scope expands from the inference on one image to inferences across images in a batch.

Our discussion draws on the detailed results on the first two convolution layers of CifarNet, as shown in the two graphs in Fig. 6, where, "image" is for reuse within the run on each individual image, while "batch" is for cross images in a batch. In both graphs, the batch-level clustering gives the highest accuracy for a given r_c (*remaining ratio*), for the more reuse opportunities the clustering brings. The curves of the batch-level clustering are shorter than the image-level ones because there are no data when r_c exceeds 0.05 in the batch-level case. The reason is that K-means clustering at batch level requires a large amount of memory, causing memory errors on the machine.

Granularity: To study how granularity affects the performance, we experiment with the *whole-vector clustering* and the *sub-vector clustering* with a neuron-vector size of 25 for both the convolutional layers of CifarNet. In the first layer (Fig. 6a), the *sub-vector clustering* doesn't perform as well as the whole-vector clustering when the scope is small. However, when applying the *sub-vector clustering* with a larger scope, it becomes the best. For the second layer (Fig. 6b), clustering at a smaller granularity always gives better results.

Distance: Fig. 6a shows that on the first layer, clustering based on angular cosine distance is consistently better in identifying the similarities compared to clustering on Euclidean distance. For the second layer (Fig. 6b), the same results hold for all the experiments except one. When performing the *whole-vector clustering* within a single input, using the angular cosine distance gives a slightly worse results than using the Euclidean distance. However, the best clustering quality on the second convolutional layer is still achieved by the angular cosine distance.

In a nutshell, as indicated in Fig. 6, a combination of larger scope (batch-level clustering), smaller granularity (*sub-vector clustering*) and angular cosine distance gives the best clustering results, better accuracy and smaller r_c. The same conclusion holds for the convolutional layers of the other two CNNs.

D. Results on Compressed Models

Network compression is a common method for minimizing the size of CNN models. Through quantization, pruning or compact network designs [5], [6], a CNN model can become much smaller without much quality loss. *Deep reuse* is complementary to these techniques in the sense that it tries to minimize CNN computations through online computation reuse rather than model size through offline weights compression. It can be applied to a compressed model to speed up its inference, just as how it helps uncompressed models.

Table V reports the speedups when we apply *deep reuse* to the compressed AlexNet model from an earlier work [5]. *Deep reuse* gives up to 3.64X speedups on the convolutional layers, quantitatively demonstrating its complementary relationship with model compression, as well as its general applicability.

E. Comparison with Perforated CNN

The work most closely related to this study is the proposal of perforated CNN [15]. It proposes to reduce computations by performing calculations with a small fraction of input patches. The evaluation of the skipped positions is done via interpolation on the computed results. Even though it may avoid some computations, it does not capitalize on dynamically

Table IV
COMPARISON WITH PERFORATED CNN (*deep reuse* NEEDS NO FINE TUNING)

METHOD	NETWORK	COMPUTATION SAVINGS	ACCURACY LOSS	
			BEFORE FINE-TUNING	AFTER FINE-TUNING
PERFORATED CNN	ALEXNET	2.0X	8.5	2
	VGG	1.9X	23.1	2.5
DEEP REUSE	ALEXNET	3.3X	-0.02	-
	VGG	4.5X	0.05	-

Table V
SPEEDUP OF APPLYING *deep reuse* TO THE COMPRESSED ALEXNET MODEL GENERATED BY PRUNING AND WEIGHT QUANTIZATION.

NETWORK	SPEEDUPS
CONV1	1.81X
CONV2	3.29X
CONV3	3.64X
CONV4	3.45X
CONV5	2.71X

discovered similarities of neuron vectors, but uses some prefixed perforation mask to pick the input rows for computations. The corresponding input rows chosen by their perforation mask are fixed for all inputs.

Deep reuse offers a more systematic way to identify computations to skip, adaptive to each input and every run. It enables neuron vector sharing and chooses the shared centroid vectors based on the similarities of neuron vectors measured at inference time. These shared vectors vary from input to input, and from run to run. In addition, it reuses the clusters and computation results from previous batches to further reduce the computation cost. Moreover, perforated CNN requires a fine-tuning process for the quantized model to recover the prediction accuracy. The use of *deep reuse* needs no such fine-tuning process.

We provide a quantitative comparison. As mentioned, perforated CNN causes significant accuracy loss and hence requires a fine-tuning process to recover the prediction accuracy. In our comparison, we use the most accurate cases reported in the previous work [15]. As Table IV reports, *deep reuse* achieves much better accuracies in all the cases. It meanwhile saves many more computations (3.3X versus 2.0X for AlexNet and 4.5X versus 1.9X for VGG) compared to the numbers reported in the previous work [15]. We cannot compare the execution times with the previous paper because the previous implementation was on a different DNN framework and their code is not available to us. However, given that the runtime overhead of our method is small as the previous subsections have shown, we expect that our method shall outperform perforated CNN in a degree similar to the rates in computation savings. The results confirm the significant benefits from the more principled approach taken by *deep reuse* for saving computations.

VI. RELATED WORK

This section discusses other related work besides the aforementioned Perforated CNN [15].

Network quantization [5], [6], [16], [17] also uses clustering, but mostly for offline compression of model parameters rather than online computation reuse on activation maps. RedCNN [18] is another work trying to reduce the model size. It does it by applying a transform matrix to the activation maps of each layer and fine tune the network. It also works offline, working during the training time. In contrast to these techniques, *deep reuse* is an online technique, with a purpose for speeding up CNN inferences. It is complementary to those offline model compression techniques, as Section V-D has empirically shown.

LSH, as a cluster method, has been used in prior CNN studies [19]–[21]. But their purposes differ from ours. For example, in the Scalable and Sustainable Deep Learning work [19], the authors apply LSH to both the weight vector and the input vector, trying to find collisions between a pair of weight and input vectors, which are regarded as a weight-input pair that may give the largest activation. In our work, we use LSH for efficiently detecting similarities among neuron vectors to expose reuse opportunities.

VII. CONCLUSION

This technical report has presented *deep reuse* as a technique to reduce computation cost of CNN inference. Experiments show that massive similarities exist among neuron vectors within and across CNN inferences. *Deep reuse* is designed to efficiently discover such similarities on the fly and turn them into reuse benefits for CNN inferences. It produces up to 3.19X speedups without accuracy loss at a convolutional layer, and up to 4.32X speedups when allowing a 3% accuracy loss. It speeds up the full network by up to 2X with virtually no ($<$0.0005) accuracy loss. *Deep reuse* features the use of an efficient clustering algorithm, a capability to harness deep reuse opportunities in three levels of scopes, a flexibility in accommodating various neuron vector granularities, and a compatibility with common model compression and other existing optimizations. It shows the promise to serve as a ready-to-use general method for accelerating CNN inferences.

VIII. ACKNOWLEDGMENT

This material is based upon work supported by DOE Early Career Award (DE-SC0013700), the National Science Foundation (NSF) under Grant No. CCF-1455404, CCF-1525609, CNS-1717425, CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF.

REFERENCES

[1] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, New York, NY, USA, 2015.

[2] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, New York, NY, USA, 2016.

[3] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, 2016.

[4] L. Du, Y. Du, Y. Li, J. Su, Y. Kuan, C. Liu, and M. F. Chang, "A reconfigurable streaming deep convolutional neural network accelerator for internet of things," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2018.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[6] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016.

[7] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. USA: Curran Associates Inc., 2012, pp. 1097–1105.

[9] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations*, 2015.

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

[11] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*. New York, NY, USA: ACM, 2004, pp. 253–262.

[12] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, 2006, pp. 459–468.

[13] K. Terasawa and Y. Tanaka, "Spherical lsh for approximate nearest neighbor search on unit hypersphere," in *Workshop on Algorithms and Data Structures*, 2007, pp. 27–38.

[14] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. Cambridge, MA, USA: MIT Press, 2015, pp. 1225–1233.

[15] M. Figurnov, A. Ibraimova, D. P. Vetrov, and P. Kohli, "Perforated-cnns: Acceleration through elimination of redundant convolutions," in *Advances in Neural Information Processing Systems*, 2016, pp. 947–955.

[16] Y. Zhou, S.-M. Moosavi-Dezfooli, N.-M. Cheung, and P. Frossard, "Adaptive Quantization for Deep Neural Network," *ArXiv e-prints*, dec 2017.

[17] Y. Choi, M. El-Khamy, and J. Lee, "Towards the limit of network quantization," in *5th International Conference on Learning Representations*, 2017.

[18] Y. Wang, C. Xu, C. Xu, and D. Tao, "Beyond filters: Compact feature map for portable deep model," in *Proceedings of the 34th International Conference on Machine Learning*, Sydney, Australia, 2017.

[19] R. Spring and A. Shrivastava, "Scalable and sustainable deep learning via randomized hashing," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Halifax, NS, Canada: ACM, 2017, pp. 445–454.

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

[21] R. Spring and A. Shrivastava, "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.