

# Deep Reuse: Streamline CNN Inference On the Fly via Coarse-Grained Computation Reuse

Lin Ning, Xipeng Shen

{lning,xshen5}@ncsu.edu

North Carolina State University

## ABSTRACT

This paper 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 adapts 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.

## KEYWORDS

Deep neural networks, Program Optimizations, GPU

### ACM Reference Format:

Lin Ning, Xipeng Shen. 2019. Deep Reuse: Streamline CNN Inference On the Fly via Coarse-Grained Computation Reuse. In *ICS'19: International Conference on Supercomputing, June 26–28, 2019, Phoenix, AZ*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/000000.000000>

## 1 INTRODUCTION

Deep Convolutional Neural Networks (CNN) have shown successes in many machine learning applications. The speed of inferences (or predictions) by CNN is important for many of its uses, which has prompted numerous recent efforts in speeding up CNN inferences. Some propose special hardware accelerators [9, 13, 29, 36], others build high performance libraries (e.g., CUDNN <sup>1</sup>, MKL-DNN <sup>2</sup>), methods to compress models [14, 15, 34], Tensor graph optimizations <sup>3</sup>, and other techniques.

However, despite the many efforts, faster CNN inference remains a pressing need, especially for many emerging CNN applications in

<sup>1</sup><https://developer.nvidia.com/cudnn>

<sup>2</sup><https://github.com/intel/mkl-dnn>

<sup>3</sup><https://www.tensorflow.org/lite/>

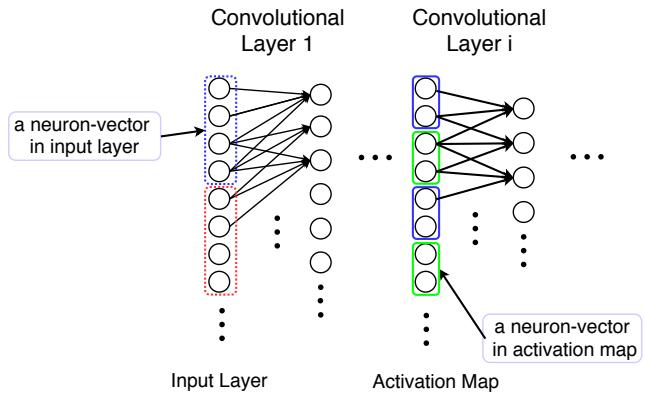
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*ICS'19, June 26–28, 2019, Phoenix, AZ*

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9999-9/18/06...\$15.00

<https://doi.org/10.1145/000000.000000>

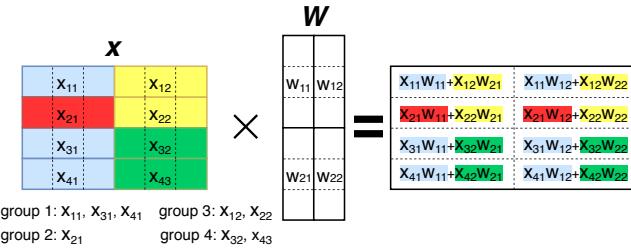


**Figure 1: Illustration of a simple 1-D CNN.** The input for convolutional layer 1 is called the input layer while the input for convolutional layer  $i$  with  $i \neq 1$  is called the activation map. Neurons in the same block form a neuron-vector. Block colors indicate the similarity of the neuron-vector values.

throughput or latency sensitive domains. Surveillance image analysis, for instance, continuously faces demands for higher processing throughput as more surveillance images of higher resolutions are streaming from thousands of cameras into the servers in a growing speed. In autonomous vehicles, real-time detection of objects is essential for minimizing the control latency, which is crucial for driving safety.

In this work, we approach the problem from a different perspective. Rather than proposing new hardware or more thorough ways to apply traditional code optimization techniques (e.g., loop tiling), we investigate the opportunities of *computation reuse* in CNN. Specifically, we propose *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.

*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.



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

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?
- 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 exists 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 [17] and VGG-19 [26]. 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.

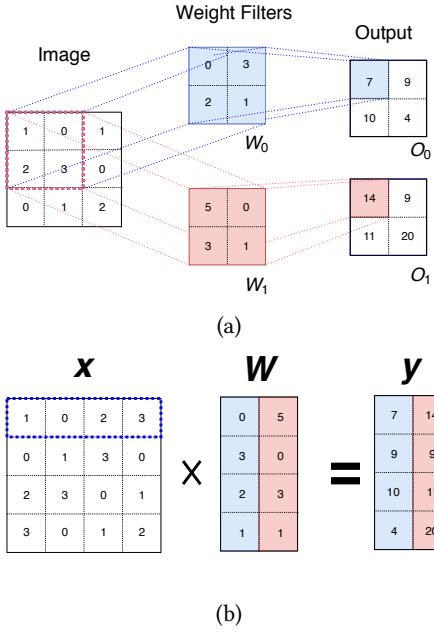
The produced *deep reuse* has several appealing properties: (1) All its optimizations happen at inference time on the fly, adaptive to every input to CNN. (2) It is compatible with model compression and other existing CNN optimization techniques. Its reuse across neuron vectors applies regardless whether the model is pruned or quantized. In Section 4.5, 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.

To the best of our knowledge, this is the first study on systematically leveraging neuron vector-level computation reuses for speeding up CNN *inferences*. Our follow-up study, published recently [21], shows that *deep reuse* can be further extended with adaptivity to speed up CNN *training* as well.

## 2 BACKGROUND AND TERMINOLOGY

CNN is a type of deep neural networks for classification, which takes in an input (e.g. an image) and predicts a class label for it. Usually a CNN consists of multiple layers, including convolutional layers, RELU layers, pooling layers, normalization layers, and fully connected layers. Each layer has its own input (input image for the first layer and activation maps for the following layers) and output, mapping the input to the output through either a linear or non-linear transformation. Among all these types of layers, the convolutional layer is the most compute intensive one and consumes a large portion of the execution time. Figure 1 illustrates the concepts of some basic components with a simple 1-D CNN, including the input layer, the activation maps and the neuron vector.

To further clarify the terms and the computations in CNN, Figure 3 illustrates the computation in a 2D convolutional layer. As shown in Figure 3 (a), the convolution operation shifts a window across the input and computes the dot product between elements in a window and a weight filter in order to produce one neuron in the output. There could be more than one filter and each filter corresponds to an output channel. In the example, we have two weight filters  $W_0$  and  $W_1$ . Therefore, the output has two channels  $O_1$  and  $O_2$ . A practical implementation for the convolution operation is unfolding the input into a larger matrix  $x$ , putting all weight filters into one matrix  $W$ , and calculating the output matrix  $y$  as the dot product between  $x$  and  $W$  as  $y = x \cdot W$ . With Figure 3 (b), we see that all



**Figure 3: Illustration of a 2D convolutional layer. (a) shows the mechanism of a convolution operation. (b) shows the resulting matrix matrix multiplication between the unfolded input matrix  $x$  and the weight matrix  $W$ .**

the input neurons in a window in Figure 3 (a) form a row of  $x$  and each weight filter becomes a column of  $W$ . As a result, an output channel is mapped to a column of  $y$ .

Suppose that  $x$  has  $N$  rows and  $K$  columns and  $W$  has  $K$  rows and  $M$  columns. The computation complexity of the matrix-matrix multiplication for one convolutional layer is  $O(N \cdot K \cdot M)$ . Sometimes, instead of processing one image, a CNN takes a number of inputs as a batch and processing them together. A batch of  $N_b$  images turns into a larger input matrix with  $N \cdot N_b$  rows and  $K$  columns.

Here we list some terminologies that are essential for understanding the idea of *deep reuse*.

**Input layer:** The input to the first convolutional in a CNN is called the *input layer*.

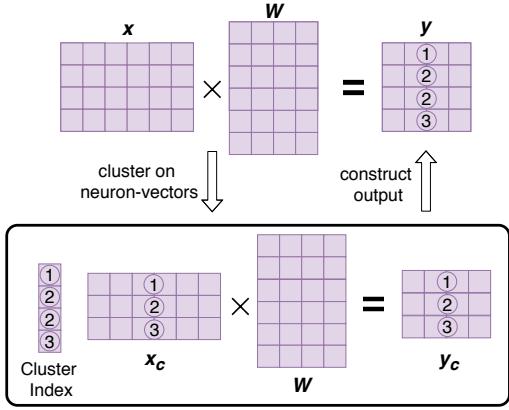
**Activation map:** The input to the convolutional layers other than the first one in a CNN is an *activation map*.

**Neuron vector:** A number of consecutive neurons in the input layer or the activation maps.

**Batch:** A number of inputs that are processed together within a same iteration is called an *input batch*.

### 3 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.



**Figure 4: Illustration of using to reduce the computation cost**  
**Numbers ①,② and ③ are the cluster IDs.**

### 3.1 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. 4, the original computation is  $y = x \cdot W$ . With *deep reuse*, we may consider each row of  $x$  as a neuron vector denoted with  $x_i$ . First, we group the 4 neuron vectors into 3 clusters and compute the centroid vectors  $x_c$ . The centroid vectors are taken as representatives. In this example, both  $x_2$  and  $x_3$  are represented by the value of  $x_{c,2}$  (the centroid vector of cluster 2). The next step is to do the computation using the centroids  $y_c = x_c \cdot W$ . The full results are then attained by reusing the outputs of the centroid vectors for each cluster member; that is,  $y_2 = y_3 = y_{c,2}$  in this example.

**Computation Savings:** In a general case, given an input matrix  $x$ , we could group all the neuron vectors into  $|C|$  clusters. The corresponding centroid vectors form a new matrix  $x_c$  with size of  $|C| \times K$ . Since we only need to compute  $y_c = x_c \cdot W$ , 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 paper, we use *remaining ratio* to indicate the fraction of computations left after the optimization. It is defined as

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

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 4 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.

## 3.2 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.

**3.2.1 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 spaces [2, 3, 7, 16, 30]. For each input vector  $\mathbf{x}$ , a hashing function  $h$  is determined by a random vector  $\mathbf{v}$  in the following way:

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

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 4) 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  $|H| \ll M(1 - r_c)$ , 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  $j^{th}$  ( $j = 1, 2, \dots, D$ ) 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  $C_{\vec{x}_i} = [b_i^{(1)}, b_i^{(2)}, \dots, b_i^{(D)}]$ . 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 4) 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 4 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.

**3.2.2 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

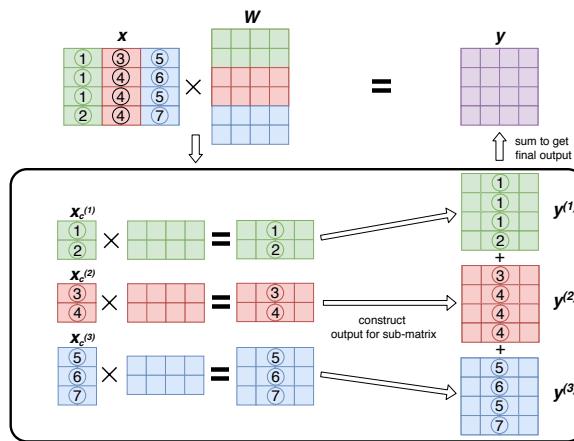
---

**Algorithm 1** Cluster Reuse

---

- 1: **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  $S_{id}$ .
- 2: **Algorithm:**
- 3: **for** all row vectors  $\mathbf{x}_i$  **do**
- 4:   Apply LSH to get the cluster id  $ID_i$
- 5: **end for**
- 6: **for**  $i = 1$  to  $N$  **do**
- 7:   **if**  $ID_i \in S_{id}$  **then**
- 8:     reuse  $O_{id=ID_i}$
- 9:   **else**
- 10:    insert  $ID_i$  into  $S_{id}$
- 11:     $O_{id=ID_i} = \mathbf{x}_i \cdot W$
- 12:    insert  $O_{id=ID_i}$  into  $O_{id}$
- 13:   **end if**
- 14: **end for**

---



**Figure 5: Illustration of *deep reuse* with a smaller clustering granularity (*sub-vector clustering*).**

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 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.

**3.2.3 Clustering Granularity.** In the basic scheme shown in Figure 4, each row vector in matrix  $X$  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. 5 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. 4), 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.

**3.2.4 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 ( $\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i}{\|\mathbf{x}_i\|}$ ) 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 4.3). *Deep reuse* hence uses angular cosine distance by default.

### 3.3 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 4 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 4 will report.

## 4 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 [17] and VGG-19 [26]. As shown in Table 1 and the first four columns of Table 2, these three networks have a range of sizes and complexities. The first network works on small images of size  $32 \times 32$ , the other two work on images of  $224 \times 224$ . The Cifar10 datasets<sup>4</sup> has 60000 color images of size  $32 \times 32$ . ImageNet<sup>5</sup> is a large dataset containing over 14 million images of size  $224 \times 224$  in 1000 classes. Both of the datasets are widely used for studying CNN performance. For all the experiments, the input images are randomly shuffled before being fed into the network. To fully demonstrate the potential of our optimization, we study the performance of *deep reuse* on both GPU servers and mobile devices.

The baseline network implementation we use to measure the speedups comes from the *slim* model<sup>6</sup> in the TensorFlow framework<sup>7</sup>. We implement our optimized CNNs by incorporating *deep reuse* into the TensorFlow code for GPU experiments and TensorFlow Lite<sup>8</sup> code for mobile device experiments. For the set of GPU experiments, both the original and our optimized CNNs automatically leverage the state-of-the-art GPU DNN library cuDNN<sup>9</sup> and other libraries that TensorFlow uses in default. The experiments are done on a machine with an Intel(R) Xeon(R) CPU E5-1607 v2 and a GTX1080 GPU. For the set of mobile device experiments, we convert the models to Tensorflow Lite files and run them on a Huawei SE mate mobile phone, which has a Huawei HiSilicon KIRIN 659 processor and a 4 GB memory.

On GPUs, we first apply our approach to only a single convolutional layer to measure the single layer speedups and the corresponding accuracy for each of the networks. 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*

<sup>4</sup><https://www.cs.toronto.edu/~kriz/cifar.html>

<sup>5</sup><http://image-net.org/about-overview>

<sup>6</sup><https://github.com/tensorflow/models/tree/master/research.slim>

<sup>7</sup><https://github.com/tensorflow/tensorflow>

<sup>8</sup><https://www.tensorflow.org/lite/>

<sup>9</sup><https://developer.nvidia.com/cudnn>

**Table 1: Benchmark networks**

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

are determined for each convolution layer as part of the hyperparameters tuning process of CNN training. Sections 4.1 and 4.2 presents the speedup results. In Section 4.3, we report some insights on 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 3.1.)

Section 4.5 reports the speedups when *deep reuse* applies to CNNs after model compression [14], demonstrating its complementary relations with model quantization and compression. The speedups achieved on mobile device for each convolutional layer of CifarNet and AlexNet are presented in section 4.6. Finally, section 4.4 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.

### 4.1 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.

*Speedups from intra-batch reuse.* Columns 5–11 in Table 2 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 2, it introduces

**Table 2: Single Layer speedups.**  $K$  is the kernel size and  $M$  is the number of weight filters.  $T_b$  is the running time of the baseline model.  $L$  refers to the neuron-vector length.  $r_c = |C|/N$  is the remaining ratio.

NETWORK	CONVLAYER	K	M	$T_b(\mu s)$	NO CLUSTER REUSE						CLUSTER REUSE		
					HYPERCUBE			LSH			LSH		
					L	$r_c$	SPEEDUP	H	L	$r_c$	SPEEDUP	SPEEDUP	
CIFARNET	CONV1	75	64	286	3	0.03	1.57X	15	5	0.01	1.58X	1.59X	
	CONV2	1600	64	139	10	0.11	1.68X	10	10	0.01	2.51X	2.58X	
AVG							1.63X				2.05X	2.09X	
ALEXNET	CONV1	363	64	279	11	0.14	0.94X	16	11	0.13	1.63X	1.96X	
	CONV2	1600	192	269	5	0.11	2.13X	15	20	0.18	2.84X	4.23X	
	CONV3	1728	384	144	6	0.11	1.22X	15	12	0.15	2.58X	3.92X	
	CONV4	3456	384	137	6	0.13	1.14X	15	12	0.17	2.76X	3.99X	
	CONV5	3456	256	115	6	0.11	1.14X	15	24	0.15	2.23X	4.12X	
	AVG						1.31X				2.41X	3.64X	
VGG-16	CONV1-1	27	64	3535	9	0.05	2.89X	20	9	0.08	2.35X	2.83X	
	CONV1-2	576	64	10023	6	0.05	1.37X	20	16	0.11	2.06X	2.59X	
	CONV2-1	576	128	3880	3	0.03	1.07X	18	16	0.13	1.83X	2.48X	
	CONV2-2	1152	128	6143	3	0.03	0.91X	18	16	0.11	1.95X	2.49X	
	CONV3-1	1152	256	3176	3	0.02	0.88X	16	16	0.09	2.22X	3.39X	
	CONV3-2	2304	256	5680	3	0.02	0.89X	16	16	0.11	2.03X	3.38X	
	CONV3-3	2304	256	5801	3	0.02	0.84X	16	16	0.06	2.79X	3.31X	
	CONV3-4	2304	256	5853	3	0.02	0.85X	16	16	0.09	2.52X	3.40X	
	CONV4-1	2304	512	2943	3	0.03	0.91X	15	16	0.05	3.19X	4.05X	
	CONV4-2	4068	512	5333	3	0.03	0.85X	15	24	0.1	2.85X	4.32X	
	CONV4-3	4068	512	5373	3	0.03	0.92X	15	24	0.11	2.37X	4.16X	
	CONV4-4	4068	512	5439	3	0.03	0.89X	15	24	0.13	2.44X	4.13X	
	CONV5-1	4068	512	1688	3	0.02	0.88X	12	24	0.2	1.86X	3.26X	
	CONV5-2	4068	512	1689	3	0.02	0.91X	12	24	0.18	1.81X	3.28X	
	CONV5-3	4068	512	1687	3	0.02	0.91X	12	24	0.18	1.81X	3.26X	
	CONV5-4	4068	512	1693	3	0.02	0.85X	12	24	0.16	2.02X	3.31X	
AVG							1.05X				2.26X	3.35X	

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 2 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 6 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

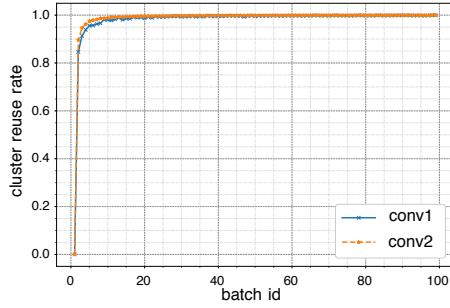
already very small (about 0.01). There are few computations left that can be saved by cluster reuse in this case.

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 2.

## 4.2 End-to-End Speedup and Runtime Overhead

In measuring the end-to-end speedups of the full network, for better accuracy, we use LSH-based *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 3, our approach obtains up to 2X speedups on the end-to-end running time of the full network, including all the runtime overhead of *deep reuse*. Figure 7 shows the execution



**Figure 6: Cluster reuse rate in CifarNet**

**Table 3: End-to-End Full Network Performance (accuracy loss  $\Delta Acc$ , average speedup  $S$  and standard derivation of the speedups  $\sigma(S)$ ) and overhead (applying LSH and reconstructing the output matrix) with *deep reuse*. (Negative errors means improvements of accuracy)**

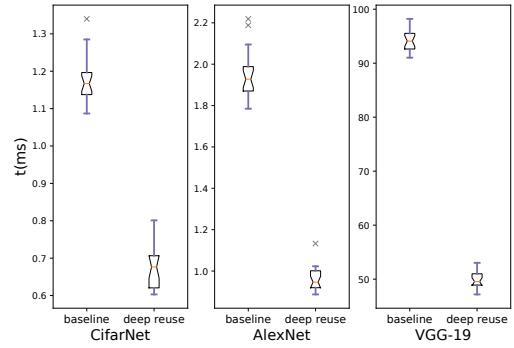
NETWORK	Acc	PERFORMANCE		OVERHEAD	
		$\Delta Acc$	$S$	LSH	RECONS
CIFARNET	0.7892	-0.0011	1.75X	43.6%	45.8%
ALEXNET	0.5360	-0.0002	2.02X	29.7%	35.3%
VGG-19	0.7118	+0.0005	1.89X	23.9%	28.8%

time of both the baseline version and the *deep reuse* version of the three networks. As indicated by the  $r_c$  column in Table 2, *deep reuse* needs to conduct only 1-18% of the original convolution calculations. The end-to-end speedups are not as much for two reasons. First, besides convolution layers, there are other layers (e.g., ReLU, pooling) in a CNN. Second, as a runtime technique, *deep reuse* introduces extra operations. There are mainly two kinds of extra operations: LSH, and the reconstruction of the activation maps after the optimized convolution. As the convolutions are dramatically accelerated, these two kinds of operations become a major portion of the end-to-end time, as shown in Table 3. A potential direction to explore in the future is to further accelerate these operations (e.g., through hardware support), which may materialize the benefits of *deep reuse* even more.

Table 3 also reports the overall prediction accuracies of the network inferences and the differences from those by the original networks. The maximum extra error our technique causes is 0.0005, marginal compared to the 54-78% overall inference accuracies.

### 4.3 Insights 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.



**Figure 7: Execution time of baseline and deep reuse version for CifarNet, AlexNet and VGG-19.**

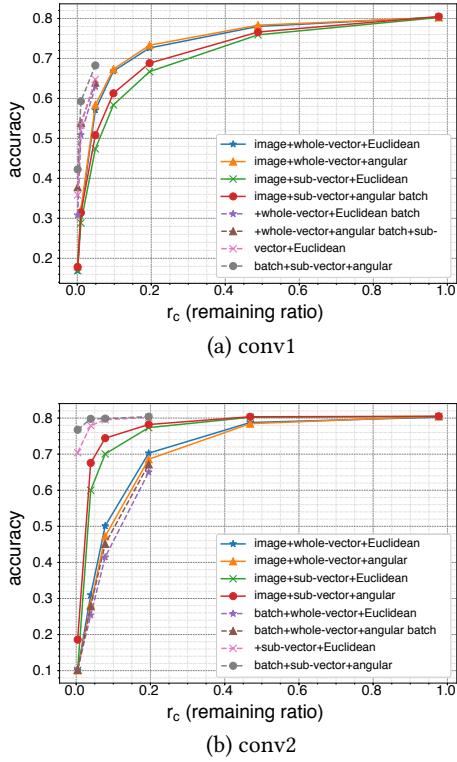
To understand the influence, we conducted a series of experiments. Because this part of exploration is for gaining the insights of the relations between these factors and the full reuse opportunities (rather than for runtime inferences), it is important to minimize the influence from other factors (e.g., clustering errors), whereas runtime clustering overhead is not a concern. Therefore, we use the most accurate clustering method, K-means, for neuron vector clustering in this part of experiments. Specifically, we take the  $r_c$ -accuracy results of applying K-means based clustering on CifarNet for a focused study. (Given the same  $r_c$  value, a higher accuracy means better identification of the similarities.)

**Scope:** Section 4.1 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. 8, 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 when  $r_c$  exceeds 0.05, in the batch-level case, K-means clustering runs out of memory.

**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. 8a), 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. 8b), clustering at a smaller granularity always gives better results.

**Distance:** Fig. 8a 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. 8b), 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



**Figure 8: Comparison of the  $r_c$ -accuracy relationships on CifarNet at different configurations in scopes (‘image’, ‘batch’), granularities (sub-vector or whole-vector), and distances (angular or Euclidean). The legend has the pattern of scope+granularity+distance.**

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. 8, 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.

#### 4.4 Comparison with Perforated CNN

The work most closely related to this study is the proposal of perforated CNN [10]. 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 discovered similarities of neuron vectors, but uses some pre-fixed 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 [10]. As Table 4 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 [10]. 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.

#### 4.5 Results on Compressed Models

Network compression is a common method for minimizing the size of CNN models. Through quantization, pruning or compact network designs [14, 34], 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 5 reports the speedups when we apply *deep reuse* to the compressed AlexNet model from an earlier work [14]. *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.

#### 4.6 Speedup on Mobile Devices

On mobile devices, the computation resources are limited. Therefore, it is more critical to have an optimized CNN inference algorithm which saves both the computation time and energy consumption. We demonstrate the potential of *deep reuse* by measuring the performance of CifarNet and AlexNet on a mobile phone. The size of the VGG-19 model is too large to run on the mobile device. As shown in Table 6, *deep reuse* achieves an average of 2.12X speedup for CifarNet and 2.55X for AlexNet. The speedups are a little bit larger comparing to those on a GPU for most of the layers.

### 5 RELATED WORK

This section discusses other related works beside the aforementioned Perforated CNN [10].

**Table 4: 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 5: Speedup of applying *deep reuse* to the compressed AlexNet generated by pruning and weight quantization.**

LAYER	CONV1	CONV2	CONV3	CONV4	CONV5
SPEEDUP	1.81X	3.29X	3.64X	3.45X	2.71X

**Table 6: End-to-End Network Speedups for CifarNet and AlexNet on the mobile device.  $T_b$  is the running time of baseline model.  $S$  is the average speedup and  $\sigma(S)$  is the standard derivation of the speedups.**

NETWORK	$CT_b(ms)$	$S$	$\sigma(S)$
CIFARNET	29.42	2.12X	0.163
ALEXNET	326.15	2.55X	0.336

Network quantization [5, 14, 34, 37] also uses clustering, but mostly for offline compression of model parameters rather than online computation reuse on activation maps. RedCNN [33] 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 tuning 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 4.5 has empirically shown.

LSH, as a cluster method, has been used in prior CNN studies [27, 28, 31] as a algorithm level optimization. But their purposes differ from ours. For example, in the Scalable and Sustainable Deep Learning work [28], 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.

There are a bunch of work on hardware DNN accelerators. Scapel [35] takes advantage of both the CNN sparsity and the hardware parallelism by customizing the DNN pruning to the underlying hardware with SIMD-aware weight and node pruning. EIE [13] uses an energy-efficient engine designed specifically to compressed CNN model. It performs well by leveraging the sparsity of both weight and input, fetching the small model from SRAM instead of DRAM, and taking advantage of weight sharing and quantization.

DaDianNao [4] presents a multi-chip machine learning architecture which achieves large speedups and significant energy savings. It explores the intra-layer parallelism in a CNN with a tiled architecture and on-chip eDRAM. FPGA based accelerators [1, 19, 22, 24, 25, 36] also show great potential on accelerating DNN inference. However, all these work obtain speedups with special purpose hardware support. On the contrast, *deep reuse* is a pure software solution and it doesn't rely on any specific hardware design.

As the neural networks go deeper and larger, some recent studies focus on optimizing the DNN memory usage. Gao et al. [11] proposes Spotlight, a reinforcement learning algorithm, to find an optimal device placement for training DNNs on a mixture of GPU and CPU devices. The dynamic GPU memory scheduling runtime SuperNeuron [32] enables the network training far beyond the GPU DRAM capacity by reducing the network-wide peak memory usage with liveness analysis, unified Tensor Pool and Cost-Aware Recomputation. Other work, such as AccUDNN [12] and TFLMS [18], try to speedup the training of very deep DNN by offloading some data to host memory with either a memory optimizer or a revised computation graph. All these work help relieve the pressure on GPU memory caused by large models.

There are some studies on how to train a deep neural network efficiently with the use of distributed environments [6, 8, 20]. Pittman and others [23] create a flexible framework for ensemble DNN training on clusters. Google's DistBelief [8] is a software framework that can accelerate the training of a single large network by scaling it to over thousands of machines. Li et al. [20] propose a distribute framework to distribute both the data and workloads over a number of worker nodes and use the server node to maintain a set of globally shared parameters. The framework provides flexible consistency, elastic scalability and fault tolerance. Another recent work [6] implements a data-parallel algorithm that is designed specifically to scale well on a large number of loosely connected processors and tests the implementation on the IBM Blue Gene/Q cluster. Comparing to these frameworks, *deep reuse* aims on accelerating CNN on a single machine without using extra resources.

This current work focuses on CNN inferences. Our follow-up study, published recently [21], shows that *deep reuse* can be extended with adaptivity to substantially accelerate CNN training as well.

## 6 CONCLUSION

This paper 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.

## ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 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 NSF.

## REFERENCES

- [1] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN Accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*.
- [2] Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. 459–468.
- [3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. MIT Press, Cambridge, MA, USA, 1225–1233.
- [4] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*.
- [5] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. 2017. Towards the Limit of Network Quantization. In *5th International Conference on Learning Representations*.
- [6] I. Chung, T. N. Sainath, B. Ramabhadran, M. Picheny, J. Gunnels, V. Austel, U. Chauhari, and B. Kingsbury. 2017. Parallel Deep Neural Network Training for Big Data on Blue Gene/Q. *IEEE Transactions on Parallel and Distributed Systems* (2017).
- [7] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive Hashing Scheme Based on P-stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*. ACM, New York, NY, USA, 253–262.
- [8] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *NIPS*.
- [9] L. Du, Y. Du, Y. Li, J. Su, Y. Kuan, C. Liu, and M. F. Chang. 2018. A Reconfigurable Streaming Deep Convolutional Neural Network Accelerator for Internet of Things. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2018).
- [10] Mikhail Figurnov, Aizhan Ibrahimova, Dmitry P. Vetrov, and Pushmeet Kohli. 2016. PerfectedCNNs: Acceleration through elimination of redundant convolutions. In *Advances in Neural Information Processing Systems*. 947–955.
- [11] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Spotlight: Optimizing Device Placement for Training Deep Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning*.
- [12] Jinrong Guo, Wantao Liu, Wang Wang, Qu Lu, Songlin Hu, Jizhong Han, and Ruixuan Li. [n. d.]. AccUDNN: A GPU Memory Efficient Accelerator for Training Ultra-deep Deep Neural Networks. *journal=arXiv preprint arXiv:1901:06773, year = 2019 ([n. d.])*.
- [13] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*.
- [14] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
- [15] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size. *arXiv preprint arXiv:1602.07360* (2016).
- [16] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.
- [17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. Curran Associates Inc., USA, 1097–1105.
- [18] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. [n. d.]. TFLMS: Large Model Support in TensorFlow by Graph Rewriting. *journal=arXiv preprint arXiv:1807.02037, year = 2018 ([n. d.])*.
- [19] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*.
- [20] Mu Li. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 2014 International Conference on Big Data Science and Computing*.
- [21] Lin Ning, Hui Guan, and Xipeng Shen. 2019. Adaptive Deep Reuse: Accelerating CNN Training on the Fly. In *Proceedings of the 35th IEEE International Conference on Data Engineering*.
- [22] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal. 2013. Memory-centric accelerator design for Convolutional Neural Networks. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*.
- [23] Randall Pittman, Hui Guan, Xipeng Shen, Seung-Hwan Lim, and Robert M. Patton. 2018. Exploring Flexible Communications for Streamlining DNN Ensemble Training Pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*.
- [24] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
- [25] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.
- [26] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations*.
- [27] Ryan Spring and Anshumali Shrivastava. 2017. A New Unbiased and Efficient Class of LSH-Based Samplers and Estimators for Partition Function Computation in Log-Linear Models. *arXiv preprint arXiv:1703.05160* (2017).
- [28] Ryan Spring and Anshumali Shrivastava. 2017. Scalable and Sustainable Deep Learning via Randomized Hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, Halifax, NS, Canada, 445–454.
- [29] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. 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.
- [30] Kengo Terasawa and Yuzuru Tanaka. 2007. Spherical lsh for approximate nearest neighbor search on unit hypersphere. In *Workshop on Algorithms and Data Structures*. 27–38.
- [31] Sudheendra Vijayanarasimhan, Jonathon Shlens, Rajat Monga, and Jay Yagnik. 2014. Deep Networks With Large Output Spaces. *arXiv preprint arXiv:1412.7479* (2014).
- [32] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. (2018).
- [33] Yunhe Wang, Chang Xu, Chao Xu, and Dacheng Tao. 2017. Beyond Filters: Compact Feature Map for Portable Deep Model. In *Proceedings of the 34th International Conference on Machine Learning*. Sydney, Australia.
- [34] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized Convolutional Neural Networks for Mobile Devices. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA.
- [35] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.
- [36] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.
- [37] Y. Zhou, S.-M. Moosavi-Dezfooli, N.-M. Cheung, and P. Frossard. 2017. Adaptive Quantization for Deep Neural Network. *ArXiv e-prints* (dec 2017).