Adaptive Deep Reuse: Accelerating CNN Training on the Fly

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

Hui Guan
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
hguan2@ncsu.edu

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

***Abstract*—This work proposes *adaptive deep reuse*, a method for accelerating CNN training by identifying and avoiding the unnecessary computations contained in each specific training on the fly. It makes two-fold major contributions. (1) It empirically proves the existence of a lot of similarities among neuron vectors in both forward and backward propagation of CNN. (2) It introduces the first adaptive strategy for translating the similarities into computation reuse in CNN training. The strategy adaptively adjusts the strength of reuse based on the different tolerance of precision relaxation in different CNN training stages. Experiments show that *adaptive deep reuse* saves 69% CNN training time with no accuracy loss.**

***Keywords*-CNN; neuron vector; similarity; training; adaptive; deep reuse;**

I. Introduction

Recent years have witnessed successes of Convolution Neural Networks (CNNs) in many data mining and data engineering domains. CNNs have given the state-of-the-art prediction accuracy in many tasks, but are known to be compute-expensive and subject to a long training process.

Many efforts have been taken to accelerate CNN training, including removing weight redundancy [1]–[3], using low precision [4], [5], hashing [6] and utilizing sparsity [7]–[9]. Most of these techniques focus on identify the weight redundancy and reduce the number of computations of the convolutional layer. In this paper, we propose *adaptive deep reuse* for accelerating CNN training. Instead of focusing on the weight parameters, this paper points out new opportunities for accelerating CNN training through computation reuse based on properties in convolutional layers' inputs. Here, inputs refer to the input images for the first layer and activation maps for the following hidden layers.

The insight comes from the common existence of similarities among neuron vectors that we recently observed in CNN executions [10]. Take the forward propagation of the first convolutional layer of a CNN as an example. To compute the convolution between an input image and the weight filters, the common practice is to unfold the input image into a large input matrix $\mathbf{x}$, and then multiply $\mathbf{x}$ with the weight matrix W as illustrated in Figure 1 (a). Usually, the size of $\mathbf{x}$ is much larger than the size of W. So if there are many similarities in $\mathbf{x}$ between neuron vectors, it could give some opportunities for computation reuse. Here a **neuron vector** is any number of consecutive elements in a row of the unfolded input matrix $\mathbf{x}$. For example, as shown in Figure 1 (a) and (b), $\vec{x}_{41} = [x_{41}x_{42}]$ is a neuron vector with 2 elements. If the layer is the input layer of a CNN, the vector corresponds to the pixel values of a segment of the input image; if the layer is a hidden layer, the vector corresponds to the values of a segment of the activation map at that layer.

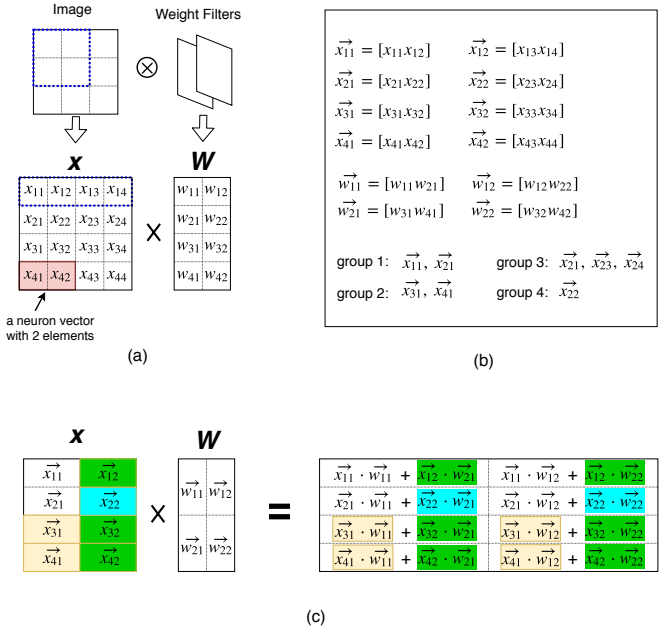


Figure 1. (a) Illustration of the common practice of computing the convolution between an image and the weight filters for the first convolutional layer. (b) and (c) Illustration of the basic idea of computation reuse across neuron vectors on calculating $\mathbf{x} \cdot W$. Instead of calculating 16 dot products, we only need to compute 8 of them: $\vec{x}_{11} \cdot \vec{w}_{11}$, $\vec{x}_{11} \cdot \vec{w}_{12}$, $\vec{x}_{31} \cdot \vec{w}_{11}$, $\vec{x}_{31} \cdot \vec{w}_{12}$, $\vec{x}_{12} \cdot \vec{w}_{21}$, $\vec{x}_{12} \cdot \vec{w}_{22}$, $\vec{x}_{22} \cdot \vec{w}_{21}$and $\vec{x}_{22} \cdot \vec{w}_{22}$.

To exploit the similarities and the reuse, we can group the neuron vectors in $\mathbf{x}$ into a small number of groups. For each group, we only need to compute the multiplications between one neuron vector and the corresponding weight segments. When calculating the multiplications between the same weight segments and the remaining neuron vectors in the same group, we could reuse previous results. For example, as shown in Figure 1 (b) and (c), we represent $\mathbf{x}$ with eight neuron vectors. These eight vectors are grouped into four groups and vectors

in the same group are similar to each other. Group one has two vectors $\vec{x}_{11}$ and $\vec{x}_{21}$. There are four dot products using these two vectors: $\vec{x}_{11} \cdot \vec{w}_{11}$, $\vec{x}_{21} \cdot \vec{w}_{11}$, $\vec{x}_{11} \cdot \vec{w}_{12}$ and $\vec{x}_{21} \cdot \vec{w}_{12}$. To leverage the similarity among neuron vectors within a group, the result of $\vec{x}_{11} \cdot \vec{w}_{11}$ can be reused for $\vec{x}_{21} \cdot \vec{w}_{11}$ and $\vec{x}_{11} \cdot \vec{w}_{12}$ for $\vec{x}_{21} \cdot \vec{w}_{12}$. With these computation reuses, only two rather than four dot products need to be computed. Half of the computations can be saved.

The goal of this current paper is to create ways to effectively exploit the neuron vector similarities to accelerate CNN training. To that end, we strive to answer four major questions:

- CNN training consists of both forward propagation and backward propagation. The backward propagation particularly involves more complicated operations than forward does. Those operations are to propagate errors from the output layer all the way down to the input layer for guiding weight updates. Do neuron vector similarity based reuse applies to both forward and backward propagation? How to integrate the reuse into backward propagation? Do we need to repeat the similarity identification for the two directions of propagation?
- Reusing cluster centers for cluster members incurs errors. How do the errors influence CNN training quality and convergence rate?
- Given that CNN training goes through an iterative process with training errors decreasing gradually, does it make sense to evolve the aggressiveness of the reuse (in terms of allowed reuse-incurred errors) through the training process? How to do that to shorten the training time as much as possible while compromising no quality of the final trained CNN?
- How much ultimate benefits can the reuse bring to real-world CNNs?

To answer these open questions, this paper proposes *adaptive deep reuse* and systematically explores its integration in CNN training and its effects.

We start with some brief background on CNN training and list a set of notations used throughout the paper in Section II. We then explain, in Section III, neuron vector similarities that we have recently observed [10], and how they can be recognized through Locality Sensitive Hashing (LSH), an online clustering method.

Next, in Section IV, we analyze the key computation of the forward and backward propagation, and explain why similarity detection is needed to do in only forward propagation. We explain how the computation of the backward propagation can directly reuse the similarity and the clustering results attained in the forward propagation. The insight lays the foundation for efficient integration of neuron vector-level reuse in CNN training.

Section V explores the relations between reuse aggressiveness and computation accuracy. Based on the progressive trend of training accuracy in CNN training, we introduce two strategies for dynamically adjusting the strength of computation reuse. They try to align the aggressiveness of the reuse with the evolving degree of error tolerance in CNN training. One strategy adjusts the resolution of hashing functions and the number of clusters, the other strategy changes the scope of clustering (across input batches or not). The dynamic nature equips *adaptive deep reuse* the capability to detect and exploit the reuse in each specific CNN training on the fly.

Section VI reports our experimental results. On three commonly used CNNs, *adaptive deep reuse* saves up to 69% training time while giving the same training accuracy as the original CNN training does. The significant savings come from two properties of *adaptive deep reuse*. By clustering neuron vectors at runtime, it finds the redundant computations in each specific run of a CNN; by adapting to the progressive trajectory of the CNN through the training process, it strikes a balance between the aggressiveness in computation savings and the training accuracy.

Overall, this work makes the following main contributions:

- To our best knowledge, this work is the first study that systematically explores neuron vector similarities for speeding up CNN training.
- This work proves that the backward propagation could benefit directly from the neuron vector similarity detected in the forward propagation, which is the key point for efficient computation reuse in the backward propagation.
- The proposed *adaptive deep reuse* is the first method that adaptively and effectively turns the similarities into substantial savings of CNN training times.

II. Background and Notations

CNN training contains two parts: the forward propagation and the backward propagation.

For the forward pass, the formula that a convolutional layer uses to compute the output for a given input $\mathbf{x}$ and model parameters W, b is as follows:

$$\mathbf{y} = \mathbf{x} \cdot W + b, \tag{1}$$

where $\mathbf{x}$ is the unfolded input matrix, $\mathbf{y}$ is the output matrix, W is the weight matrix and b is the bias.

When performing the computation, the convolutional layer takes an input tensor with size $N_b \times I_w \times I_h \times I_c$ and outputs an output 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 the number of channels 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 the number of channels of the corresponding output.

The input is unfolded into a large input matrix $\mathbf{x}$ with a dimension of $N \times K$ using a stride size of s, a kernel width of k_w and a kernel height of k_h. 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 size of a weight kernel. The number of rows corresponding to one input is $N_{img} = \frac{N}{N_b}$. The weight of the convolutional layer is represented as a matrix W with size $K \times M$, where M is the number of weight filters. The output $\mathbf{y}$ has a dimension of $N \times M$ and is computed using Equation 1. The main computation comes

Table I
NOTATIONS USED IN THIS PAPER

NOTATION	MEANING
N_b	BATCH SIZE
I_w	WIDTH OF AN INPUT CHANNEL
I_h	HEIGHT OF AN INPUT CHANNEL
I_c	# OF CHANNELS OF THE INPUTS
O_w	WIDTH OF AN OUTPUT CHANNEL
O_h	HEIGHT OF AN OUTPUT CHANNEL
N	# OF ROWS FOR A BATCH OF INPUTS
K	THE SIZE OF A WEIGHT KERNEL
M	# OF WEIGHT KERNELS
s	STRIDE
k_w	THE KERNEL WIDTH
k_h	THE KERNEL HEIGHT
N_{img}	# OF ROWS CORRESPONDING TO ONE IMAGE
L	THE LENGTH OF A *sub-vector*
H	# OF HASHING FUNCTIONS
$\|C\|$	# OF CLUSTERS
r_c	THE REMAINING RATIO $\frac{\|C\|}{N}$

from the matrix-matrix multiplication, which has a complexity of $O(N \cdot K \cdot M)$.

For the backward pass, there are two key computations to perform: one is computing the gradient of the weight ∇W; the other is computing the deltas of the inputs $\delta \mathbf{x}$. Let $\mathcal{L}$ be the loss function, $\delta \mathbf{y} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$, $\delta \mathbf{x} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}}$ and $\nabla W = \frac{\partial \mathcal{L}}{\partial W}$. Given the chain rule, formulas of the two key computations are

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial W} = \mathbf{x}^T \cdot \delta \mathbf{y}, \tag{2}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \delta \mathbf{y} \cdot W^T. \tag{3}$$

The main computations are two matrix multiplications. Since the dimension of $\delta \mathbf{y}$ is the same as $\mathbf{y}$, the complexity of the backward pass is $O(2 \cdot N \cdot K \cdot M)$.

Table I gives a list of all the notations that are mentioned in this paper.

III. SIMILARITY IDENTIFICATION AND DEEP REUSE

This section explains the basic relation between neuron vector similarity and computation reuse in CNN forward propagation, and how to identify it through online clustering. The knowledge offers the foundation for *adaptive deep reuse* for CNN training.

A. Overview

Fig. 2 illustrates the basic idea of how to employ neuron vector similarity for *deep reuse*. The way to identify the similarities is to group the neuron vectors into clusters. Neuron vectors in the same group are similar to each other. Then we use the cluster centroid to represent all the neuron vectors in that cluster for the computation. In Fig. 2, the four input row vectors are first grouped into two clusters. Vectors $\vec{x}_1$ and $\vec{x}_3$ are in cluster one while vectors $\vec{x}_2$ and $\vec{x}_4$ are in cluster two. The centroid matrix is $\mathbf{x}_c = [\vec{x}_{c1}^T \quad \vec{x}_{c2}^T]^T$, where

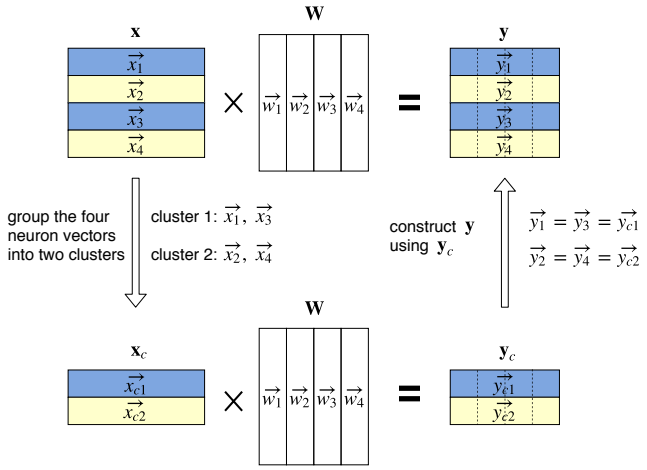


Figure 2. The basic idea of employing neuron vector similarity for *deep reuse*. In matrix $\mathbf{x}$, vectors in the same color belong to a same cluster.

$\vec{x}_{c1}$ and $\vec{x}_{c2}$ are the cluster centroids of cluster 1 and cluster 2. To compute $\mathbf{y} = \mathbf{x} \cdot W$, we first calculate the output of the centroids ($\mathbf{y}_c = [\vec{y}_{c1}^T \quad \vec{y}_{c2}^T]^T$) using $\mathbf{y_c} = \mathbf{x_c} \cdot W$. Then we could reconstruct $\mathbf{y} = [\vec{y}_1^T \ \vec{y}_2^T \ \vec{y}_3^T \ \vec{y}_4^T]^T$ using $\vec{y}_1 = \vec{y}_{c1}$, $\vec{y}_2 = \vec{y}_{c2}$, $\vec{y}_3 = \vec{y}_{c1}$ and $\vec{y}_4 = \vec{y}_{c2}$.

Computation Savings: For an input matrix $\mathbf{x}$ with a dimension of $N \times K$, the original computation complexity of the forward pass $\mathbf{y} = \mathbf{x} \cdot W$ is $O(N \cdot K \cdot M)$. If we could group all the neuron vectors into $|C|$ clusters, we only need to compute the matrix multiplication between the centroid matrix $\mathbf{x_c}$ and the weight matrix W. The computation complexity becomes $O(|C| \cdot K \cdot M)$. The remaining fraction of computation is $\frac{|C|}{N}$. We use r_c to represent this ratio. It is defined as

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

The smaller r_c is, the less computations are left and the more theoretical speedups we could achieve. If $|C| << N$, we could save almost all computations.

For *deep reuse* to generate actual benefits, the following conditions must hold:

- There is a large amount of similarities among neuron vectors.
- The overhead of detecting and leveraging these similarities should be much smaller than the time saving it brings to the CNN training. Since the inputs, especially the activation maps, change during the runtime, the clustering method we use to identify the similarities must be light weighted.

To verify our assumption on neuron vector similarity, we take the trained model of CifarNet, AlexNet and VGG-19 and run their inferences on two datasets (Cifar10 and ImageNet). Our empirical study shows that with K-means clustering (which helps produce high-quality clustering results), we consistently find strong similarities among neuron vectors within a batch of images for each convolutional layer. Section VI provides the results.

As for the second condition, we have tried several clustering methods and found LSH the one that fulfills our requirement.

The remaining of this section will describe how to use LSH to detect the similarities, and describe several designs related to the selection of clustering parameters.

B. Design of Similarity Detection for Deep Reuse

It is important to choose an appropriate clustering method for similarity identification. First, it needs to be able to give good clustering results so that the optimization of *adaptive deep reuse* does not cause accuracy loss of the training. Second, the clustering method should not introduce too much overhead during the training so that the computation savings could be efficiently converted into time savings.

Clustering Method (LSH): After a thorough exploration of several different methods, we identified LSH as the clustering method for the *adaptive deep reuse*. LSH is widely used for solving the approximate or exact Nearest Neighbor problem in high dimension space [11]–[15]. With a random vector $\mathbf{v}$, we could use the following equations to determine a hashing function h for each input vector $\mathbf{x}$

$$h_{\mathbf{v}}(\mathbf{x}) = \begin{cases} 1 & if \quad \mathbf{v} \cdot \mathbf{x} > 0 \\ 0 & if \quad \mathbf{v} \cdot \mathbf{x} \leq 0. \end{cases} \tag{4}$$

If we have a number of random vectors, say H, we will have H hashing functions according to Equation 4. With these hashing functions, LSH could map an input vector into a bit vector with a length of H. A property of LSH is that the input vector with smaller distances have a high probability to be hashed into the same bit vector. Therefore, it is a good candidate algorithm for clustering in our context. When applying it, we use each hashed bit vector as a cluster ID and all the neuron vectors mapped to the same bit vector form a cluster.

To measure the effectiveness of LSH, we run LSH on the neuron vectors during the inference runs of trained CNN models. Experiment results (Section VI) show that when choosing an appropriate number of hashing functions, LSH can be applied to vectors with different lengths and achieve good inference accuracy. With LSH applied, the operations of a convolutional layer now consist of two parts: hashing and the centroid-weight multiplication. The hashing itself takes some time. If having H hashing functions, the computation complexity is $O(N \cdot K \cdot H + |C| \cdot K \cdot M)$. The first item $N \cdot K \cdot H$ is the hashing overhead. 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* $|C|/N$.

Similarity Metric: The metric we use to measure the similarity between any two neuron vectors is the angular cosine distance. The input vectors are first normalized as $\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i}{\|\mathbf{x}_i\|}$. Then the distance is measured with $\|\hat{\mathbf{x}}_i - \hat{\mathbf{x}}_j\|$. Therefore, the input to LSH is $\hat{\mathbf{x}}$ instead of $\mathbf{x}$.

Cluster Scope: *Adaptive deep reuse* supports the detection of similarities among neuron vectors in three levels of clustering scopes: the neuron vectors in a run on one CNN input (single-input level), those in the runs on a batch of inputs (single-batch level), and those across batches (across-batch level). With a larger scope, the pool in which the neuron vectors being clustered is larger and there are more reuse opportunities among neuron vectors. The default scope setting is the single-batch level. The user could change the setting into a single-input or across-batch level according to their demands.

For the single-input or single-batch level, we can simply apply the clustering algorithm to all the neuron vectors within an input or within a batch directly. Some further complexity exists when the scope goes across batches. Since inputs from different batches come at different times, it is impractical to wait until all the inputs arrive to do clustering.

We address the complexity with *cluster reuse* by leveraging the properties of LSH. The idea is to allow neuron vectors from different input batches to be assigned to the same cluster and to share the value and computation result of the same cluster centroid. 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 are mapped to the same bit vector, they are assigned with a same cluster ID and thus to the same cluster. To do that, the same family of hash functions $\mathcal{H}$ has to be used for all batches.

Algorithm 1 Cluster Reuse

Input: input matrix $\mathbf{x}$ with dimension $N \times K$; the set $\mathcal{IDX}$ contains the bit vectors representing the cluster ID; the set of outputs $\mathcal{Y}$ corresponding to $\mathcal{IDX}$.
Algorithm:
Initialize with $\mathcal{IDX} = \{\}$, $\mathcal{Y} = \{\}$
for each iteration **do**
  take a batch of input with a batch size of N_b
  for each row vectors x_i **do**
    $\mathcal{ID}(x_i) = \mathcal{H}(x_i)$
    if $\mathcal{ID}(x_i) \in \mathcal{IDX}$ **then**
      $y(x_i) = \mathcal{Y}_{id=\mathcal{ID}(x_i)}$
    else
      $\mathcal{IDX} = \mathcal{IDX} \cup \mathcal{ID}(x_i)$
      $y(x_i) = x_i \cdot W$
      $\mathcal{Y} = \mathcal{Y} \cup y(x_i)$
    end if
  end for
end for

Algorithm 1 illustrates how to reuse the clusters and the corresponding results with LSH. A set $\mathcal{IDX}$ is used to store all previously appeared bit vectors (the cluster IDs) and a set $\mathcal{Y}$ is used to store all the outputs computed with those cluster centroids. When a new batch of inputs comes, we map each neuron vector to a bit vector using LSH. For neuron vectors being mapped to existing clusters, we could reuse the corresponding outputs. If a neuron vector is mapped to a new cluster, we calculate the output as $y(x_i) = x_i \cdot W$. After that, we could update $\mathcal{IDX}$ and $\mathcal{Y}$ accordingly. The average cluster reuse rate for each batch is represented as R. The computation complexity when using *cluster reuse* becomes $O(N \cdot K \cdot H + (1 - R) \cdot |C| \cdot K \cdot M)$ if using the whole row vector for clustering. Therefore, a larger cluster reuse rate could help saving more computations.

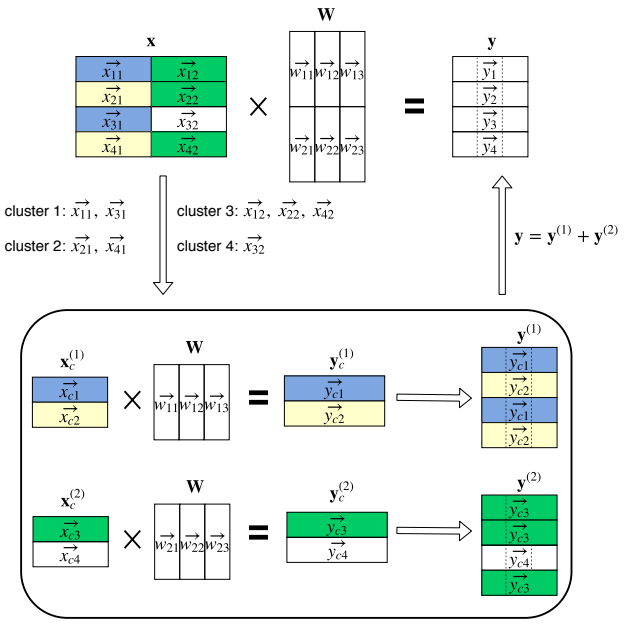


Figure 3. Illustration of *adaptive deep reuse* when clustering over *sub-vectors*.

Cluster Granularity: In the basic scheme shown in Figure 2, 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 the neuron vector which is a consecutive segment of a row vector as a *sub-vector*. Our design allows a flexible adjustment of the clustering granularity by changing the length (L) of the *sub-vector*.

Fig. 3 illustrates the procedures of *adaptive deep reuse* while clustering over *sub-vectors*. The input matrix $\mathbf{x}$ is divided into two sub-matrices $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$. Where $\mathbf{x}^{(1)} = [\vec{x}_{11}^T\ \vec{x}_{21}^T\ \vec{x}_{31}^T\ \vec{x}_{41}^T]^T$ and $\mathbf{x}^{(2)} = [\vec{x}_{12}^T\ \vec{x}_{22}^T\ \vec{x}_{32}^T\ \vec{x}_{42}^T]^T$. For each sub-matrix, *adaptive deep reuse* groups the neuron vectors into clusters, computing the centroid matrices $\mathbf{x}_c^{(i)}$ and the corresponding outputs $\mathbf{y}_c^{(i)}$. Then it reconstructs the partial output $\mathbf{y}^{(i)}$ for each sub-matrix. To compute the final output $\mathbf{y}$, it adds the partial result together as $\mathbf{y} = \mathbf{y}^{(1)} + \mathbf{y}^{(2)}$.

As clustering algorithms usually work better on low dimension data, we see better clustering results when a smaller clustering granularity is used. However, a smaller neuron vector length results more neuron vectors, and hence more adding operations. Therefore, it does not always save more computations. Assume each input row vector is divided into N_{nv} neuron vectors and the length 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 size of a weight filter, the number of weights filters and the number of rows for a batch of inputs. The average number of clusters is $|C|_{nv,avg} = \frac{1}{N_{nv}} \sum_{j=1}^{N_{nv}} |C|_{nv,j}$. For simplicity of notations, we use r_c to also represent the average remaining ratio in this part of discussion ($r_c = r_{c,avg} = \frac{|C|_{nv,avg}}{N}$). The computation complexity of clustering over *sub-vectors* becomes $O((r_c + \frac{1}{L}) \cdot N \cdot K \cdot M)$. With a smaller clustering granularity, we are more likely to have a smaller r_c but a larger $\frac{1}{L}$. A balance between these two parts is needed to minimize the overall computations.

Adaptive 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 to attain a desired cost-benefit trade-off.

C. Overall Computation Complexity

Now taking everything into consideration, the overall computation complexity of using LSH clustering method on *sub-vectors* without *cluster reuse* is

$$\mathcal{C}_f = O((\frac{H}{M} + r_c + \frac{1}{L}) \cdot N \cdot K \cdot M). \quad (5)$$

If using *cluster reuse*, the complexity becomes

$$\mathcal{C}_{f,cr} = O((\frac{H}{M} + (1 - R) \cdot r_c + \frac{1}{L}) \cdot N \cdot K \cdot M). \quad (6)$$

The expected execution time is proportional to the computation complexities.

IV. Reuse of Similarity Detection Results for Backward Propagation

The previous section describes how to use LSH to detect similarities among neuron vectors in the forward propagation. The other part of the CNN training is the backward propagation. The backward propagation accounts for around 2/3 of the computations for each convolutional layer as shown in Section II. Speeding up backward propagation is hence essential for accelerating the CNN training.

To apply *adaptive deep reuse* to the backward propagation, a question we need to answer is whether we can reuse the similarity detection results from the forward propagation. This question arises because of two concerns:

- The neuron vector similarity based computation reuse on the forward propagation already introduces approximation errors to the CNN training process. If we apply LSH to the backward propagation again, it would introduce even more approximation errors, which may make it harder to recover the original training accuracy.
- The LSH clustering method itself introduces computation overhead. As shown in Section II, the main computation of the backward pass includes two matrix multiplications. Applying LSH twice for these two matrix multiplications will bring even more overhead.

A close examination of the computations of backward pass shows that the clustering results attained in the forward pass could be applied directly for computing the weights gradient ∇W and the deltas of the inputs $\delta \mathbf{x}$. The remaining part of this section explains how it works.

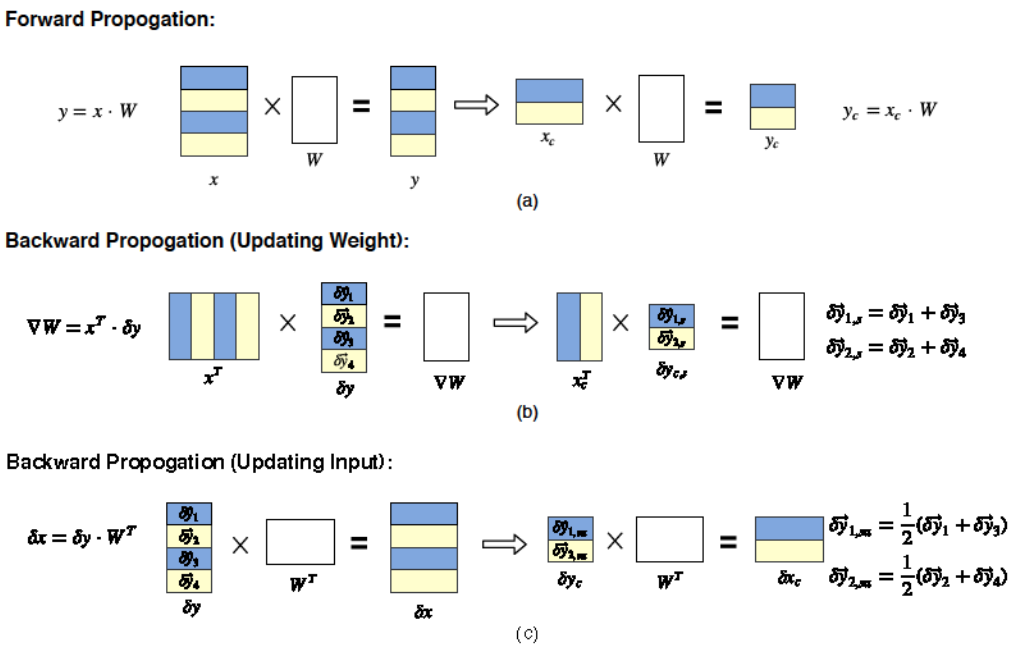


Figure 4. An illustration of the reuse scheme on the forward and backward propagation. (b) and (c) show how to reuse the similarity results found in the forward propagation to compute the weight gradient and the delta of the input.

A. Weight Gradient

Let $\mathcal{L}$ be the loss function. The delta of the output is $\delta \mathbf{y} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$, which has a dimension of $N \times M$. The centroid matrix of the input obtained from the forward propagation is $\mathbf{x}_c$ as shown in Figure 4 (a). The weight gradient is computed using Equation 2. Therefore, we have

$$\frac{\partial \mathcal{L}}{\partial W_{ij}} = \sum_{k=1}^{N} x_{ik} \delta y_{kj} = \sum_{l=1}^{|C|} x_{il} \sum_{k \in l} \delta y_{kj}. \quad (7)$$

For each cluster l, where $l = 1, \ldots, |C|$, let

$$\delta \vec{y}_{l,s} = \sum_{k \in l} \delta \vec{y}_k \quad (8)$$

to represent the resulting vector of adding the values of all corresponding row vectors in $\delta \mathbf{y}$. All the summed vectors $\delta \vec{y}_{l,s}$ form a matrix $\delta \mathbf{y}_{c,s}$ as shown in Figure 4 (b). Then the previous formula becomes

$$\frac{\partial \mathcal{L}}{\partial W} = \mathbf{x}^T \cdot \delta \mathbf{y} = \mathbf{x}_c^T \cdot \delta \mathbf{y}_{c,s}, \quad (9)$$

where $\delta \mathbf{y}_{c,s}$ has a dimension of $|C| \times M$.

Figure 5 gives an illustration of calculating the weight gradient when clustering on *sub-vectors* with length $L = K/2$. First, the input matrix $\mathbf{x}$ is divided into two sub-matrices, denoted as $\mathbf{x}_1$ and $\mathbf{x}_2$. The centroid matrices of each input sub-matrices are $\mathbf{x}_{c,1}$ and $\mathbf{x}_{c,2}$. The corresponding weight gradient matrix can also be splitted into two blocks ∇W_1 and ∇W_2. Second, we compute the corresponding $\delta \mathbf{y}_{c,1,s}$ and $\delta \mathbf{y}_{c,2,s}$ according to Equation 8. Finally, for each block, the weight gradient matrix is computed separately as

$$\frac{\partial \mathcal{L}}{\partial W_I} = \mathbf{x}_I^T \cdot \delta \mathbf{y} = \mathbf{x}_{c,I}^T \cdot \delta \mathbf{y}_{c,I,s}. \quad (10)$$

Here $I = 1, 2$ are the block IDs.

Computation Complexity: If using the whole row vector for clustering, the computation complexity of calculating $\delta \mathbf{y}_{c,s}$ is $O((N - |C|) \cdot M)$ and the complexity of computing $\mathbf{x}_c^T \cdot \delta \mathbf{y}_{c,s}$ is $O(K \cdot |C| \cdot M)$. Combining them gives us the overall complexity of $O((1 - r_c) \cdot N \cdot M + r_c \cdot N \cdot K \cdot M)$, where $r_c = \frac{|C|}{N}$ is the remaining ratio. Given a sub-vector length of L, the average computation complexity of calculating the weight gradient using the forward pass clustering results is

$$C_{b,w} = O(\sum_{I=1}^{K/L} (N - |C_I|) \cdot M + L \cdot |C_I| \cdot M)) \quad (11)$$

$$= O((\frac{1 - r_c}{L} + r_c) \cdot N \cdot K \cdot M), \quad (12)$$

here, for simplicity, we use r_c to represent the averaged remaining ratio across all sub-matrices of $\mathbf{x}$.

B. Delta of the Input

Let l be the cluster ID, where $l = 1, \ldots, |C|$ and N_l be the number of vectors in cluster l. To compute the delta of the input, We first point out that for all $i \in l$, $\mathbf{x}_i = \mathbf{x}_l$. Therefore,

$$\frac{\partial \mathcal{L}}{\partial x_{l,j}} = \frac{1}{N_l} \sum_{i \in l} \frac{\partial \mathcal{L}}{\partial x_{i,j}} \frac{\partial x_{i,j}}{\partial x_{l,j}} = \frac{1}{N_l} \sum_{i \in l} \frac{\partial \mathcal{L}}{\partial x_{i,j}}. \quad (13)$$

Now we have

$$\frac{\partial \mathcal{L}}{\partial x_{l,j}} = \frac{1}{N_l} \sum_{i \in l} (\sum_{k=1}^{M} \delta y_{ik} \cdot W_{kj}) \quad (14)$$

$$= \sum_{k=1}^{M} (\frac{1}{N_l} \sum_{i \in l} \delta y_{ik}) \cdot W_{kj}. \quad (15)$$

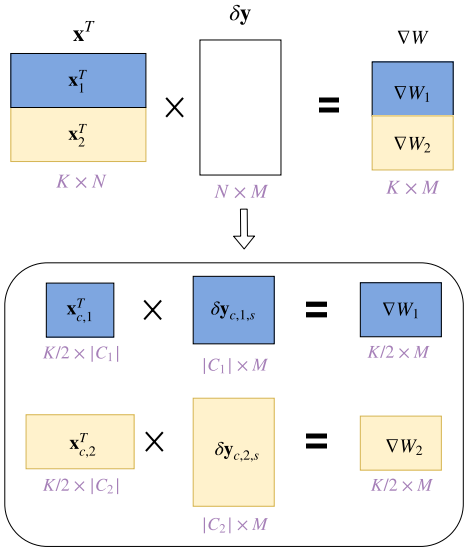


Figure 5. An illustration of calculating the weight gradient when clustering on *sub-vectors*. The length of each *sub-vector* is $L = K/2$ where K is the size of a weight kernel.

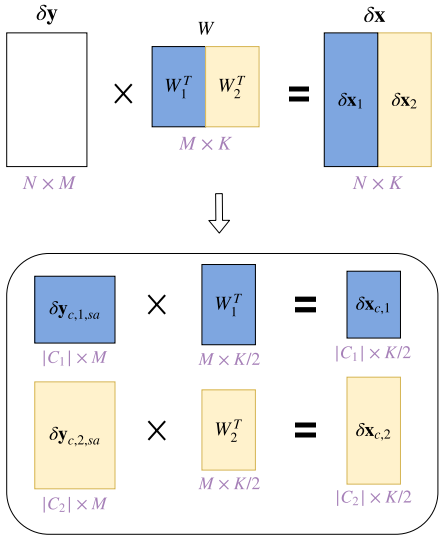


Figure 6. An illustration of calculating the delta of the input $\mathbf{x}$ when clustering on *sub-vectors*. The length of each *sub-vector* is $L = K/2$ where K is the size of a weight kernel.

Let $\delta y_{l,k,sa} = \frac{1}{N_l} \sum_{i \in l} \delta y_{ik}$, the formula becomes

$$\frac{\partial \mathcal{L}}{\partial x_{l,j}} = \sum_{k=1}^{M} \delta y_{l,k,sa} \cdot W_{kj}. \tag{16}$$

Therefore,

$$\frac{\partial \mathcal{L}}{\partial x_c} = \delta \mathbf{y}_{c,sa} \cdot W^T, \tag{17}$$

where calculating $\delta \mathbf{y}_{c,sa}$ is based on the calculation of $\delta \mathbf{y}_{c,s}$ for weight gradient computation. The gradient of the centroid is then used for all the neuron vectors in the same cluster.

When clustering over *sub-vectors*, as shown in Figure 6, both $\delta\mathbf{x}$ and W are divided into two sub-matrices. They are $\delta\mathbf{x}_1$, $\delta\mathbf{x}_2$ and W_1, W_2. The sub-matrices of the input delta are computed as

$$\delta \mathbf{x}_{c,I} = \delta \mathbf{y}_{c,I,sa} \cdot W_I^T. \tag{18}$$

Computation Complexity: When clustering over the row vectors of the input, as shown in Figure 4 (c), the computation complexity is $O(|C| \cdot M \cdot K)$. When using *sub-vectors*, the complexity becomes

$$\mathcal{C}_{b,i} = O(\sum_{I=1}^{K/L} |C_I| \cdot M \cdot L)) \tag{19}$$

$$= O(r_c \cdot N \cdot K \cdot M), \tag{20}$$

where r_c is again the averaged remaining ratio across all sub-matrices of $\mathbf{x}$.

Using Equation 10 and Equation 18, we could directly use the clustering results attained in the forward propagation to compute the weight gradient and input delta. It is easy to see that when clustering over *sub-vectors*, for each sub-matrix of $\delta\mathbf{y}$, we need to compute multiple copies of $\delta\mathbf{y}_{c,s}$. Grouping these output deltas introduces extra overhead. Therefore, even though smaller granularities could lead to better clustering results, it also brings larger computation overhead. It again leads to a trade-off between the reuse-caused accuracy loss and computation overhead.

V. Adaptive Deep Reuse for Training

This section gives a discussion on how to adaptively adjust the clustering designs for different training stages. With these adaptive adjustment, we could leverage the similarities for CNN training more efficiently and achieve more computation savings.

Different CNN training stages have different degrees of tolerance of precision relaxation. Usually at early training iterations, since the model is very rough, the training of the model is hence less sensitive to approximation errors than in later stages. In later training stages when the model gets close to convergence, the model is well learned. A small change of the input matrix may lead to substantial errors in the model updates, causing the training slow to converge. Therefore, the basic idea of *adaptive deep reuse* is to be more aggressive on computation reuse in early stages and adjust the clustering parameters gradually so that we have less computation reuse but better precision in later stages.

There are three clustering parameters to adjust given the description in Section III. They are the clustering granularity (the sub-vector length L), the number of hashing functions (H) and the flag of cluster reuse (CR, $CR = 1$ for turning on the cluster reuse). To study how these clustering parameters affect the strength of reuse and the reuse-caused accuracy loss, we experiment with different combination of parameters and gain the following observations (the detailed experiment results is shown in Section VI):

- When H and CR stay unchanged, a smaller granularity (smaller L) always leads to smaller reuse-caused accuracy loss.

- When L and CR stay unchanged, more hashing functions (larger H) gives smaller reuse-caused accuracy loss. Meanwhile, a larger H gives a larger number of clusters, thus a larger r_c.
- Assume that center reuse is not turned on ($CR = 0$). When L is large, H affects the reuse-caused accuracy loss and r_c more than L does. When L is small, the change of L affects the reuse-caused accuracy loss and r_c more than H does.
- The convolutional layers that are close to the output layer could use larger L and smaller H while achieving the same reuse-caused accuracy loss comparing to the convolutional layers that are close to the input images.
- In the selection of an appropriate combination of L and H, turning on the cluster reuse flag ($CR = 1$) always reduces the remaining ratio r_c. However, it also introduces more errors and larger reuse-caused accuracy loss.

Given these observations, we propose two adaptive strategies. The first one adjusts the combination of clustering granularity and the number of hashing functions. It uses large L and small H at the beginning of the training process. In theory, this setting may lead to large amounts of computation savings but also large clusters and hence approximation errors. As the model learns from the input images, this strategy gradually decreases the value of L and increases H. The reuse becomes less aggressive, computation savings become less, but the perturbance to the learning quality also decreases. The second strategy is about clustering scopes. It sets the cluster reuse flag CR to either 0 or 1 for different training stages.

A. Strategy of Adjusting L and H

To make this strategy work effectively, there are several questions to be answered. We list these questions and our solutions as follows.

a) How to determine the ranges of L and H we are going to use during the training?

At the beginning of CNN training, the adaptive strategy needs to be more aggressive in order to save more computations when the training process could tolerate large precision relaxation. Therefore, we should use the largest L and the smallest H for the initial setting. At the end of the training, we need to have little reuse-caused accuracy loss. Thus we use the smallest L and the largest H at this stage. We empirically set the ranges of L and H based on the following policies:

Policy 1: For each layer, set the lower bound of L as $L_{min} = k_w$ and the upper bound as $L_{max} = \lceil\sqrt{I_c}\rceil \cdot k_w$. k_w is the width of the weight kernel and I_c is the number of input channels.

Amendment 1: For layers other than the first convolutional layer, if k_w is very small (e.g. 3), and $k_w \cdot k_w < 10$, set $L_{min} = k_w \cdot k_w$.

Policy 2: Given the observation that the remaining ratio r_c is always larger than 0.01, we set the lower bound of H by finding the minimum H that $2^{H_{min}} > 0.01N$ and the upper bound of H by $2^{H_{max}} < N$.

Given these two policies, the actual ranges of L and H are determined by the size of a convolutional layer. Therefore, even at the same training stage, different convolutional layers may have different ranges of L and H.

b) When switching from one combination to the other, how to decide the combination of L and H to use next?

There are two factors that affect the choice of the clustering parameters. One is the expected computation time, the other is the corresponding reuse-caused accuracy loss. When switching from one set of parameters to the other, we always expect to choose the one that gives the minimum expected execution time and the smallest reuse-caused accuracy loss.

Because the expected computation time is proportional to the computation complexity, Equations 5, 10 and 18 could help us determine the expected computation time $\mathcal{E}(t)$. Since the similarity detection only happens in the forward propagation, we only use Equation 5 at this stage. We have

$$\mathcal{E}_f(t) \sim (\frac{H}{M} + r_c + \frac{1}{L}). \tag{21}$$

Given $\{L_1, H_1\}$, if we only change the clustering granularity from L_1 to L_2, the change of the expected computation time would be

$$\Delta\mathcal{E}_f(t, \{L_1, H_1\} \rightarrow \{L_2, H_1\}) = \frac{1}{L_2} - \frac{1}{L_1}. \tag{22}$$

On the other hand, if we only change the number of hashing functions from H_1 to H_2, we have

$$\Delta\mathcal{E}_f(t, \{L_1, H_1\} \rightarrow \{L_1, H_2\}) = \frac{H_2 - H_1}{M}. \tag{23}$$

With Equations 22 and 23 and the ranges of L and H, we can place all possible sets of $\{L, H\}$ into an ordered candidate list $[\{L, H\}]$ based on the following policy:

Policy 3: Given the ranges of L and H, create two lists $[L]$ and $[H]$, where $[L]$ is sorted with an decreasing order and $[H]$ is sorted with an ascending order. After using the parameter setting of $\{L_i, H_j\}$, the next possible setting is either $\{L_{i+1}, H_j\}$ or $\{L_i, H_{j+1}\}$. Putting the one that gives a smaller $\Delta\mathcal{E}(t)$ according to Equation 22 and Equation 23 as the next candidate into $[\{L, H\}]$.

This is an offline process and it gives the candidates for runtime examination. The runtime selection of the parameters follows the following strategy. When finishing training with the current set of parameters $\{L_{cur}, H_{cur}\} = \{L_i, H_i\}$, where i is the position of $\{L_{cur}, H_{cur}\}$ in the candidate list, the strategy runs inference on a batch of inputs with $\{L_{cur}, H_{cur}\}$ as the parameters to get an accuracy value A_{cur}. It then applies $\{L_{i+1}, H_{i+1}\}$ to the same batch of inputs for inference and get another accuracy A_{i+1}. It selects the next candidate $\{L_{i+1}, H_{i+1}\}$ to use as $\{L_{cur+1}, H_{cur+1}\}$ for the next stage based on the following conditions:

Amendment 3.1: When the training accuracy is less than 0.5, if $A_{i+1}/A_{cur} \geq 1.5$, $\{L_{i+1}, H_{i+1}\}$ is chosen as $\{L_{cur+1}, H_{cur+1}\}$. Otherwise, apply the same checking process for the next candidate parameter set $\{L_{i+2}, H_{i+2}\}$.

Amendment 3.2: When the training accuracy is larger than 0.5, if $A_{i+1} - A_{cur} \geq 0.1$, $\{L_{i+1}, H_{i+1}\}$ is chosen as $\{L_{cur+1}, H_{cur+1}\}$. Otherwise, check $\{L_{i+2}, H_{i+2}\}$.

Amendment 3.3: If all settings after $\{L_i, H_i\}$ cannot satisfy the conditions in the previous two amendments, we simply chose $\{L_{i+1}, H_{i+1}\}$ as $\{L_{cur+1}, H_{cur+1}\}$ as long as $A_{i+1}/A_{cur} \geq 1.1$. If $A_{i+1}/A_{cur} < 1.1$, skip this set of parameters and go to the next one.

c) how to determine when to switch the clustering parameters?

Given a set of $\{L_{cur}, H_{cur}\}$, we train the network until the loss value stops decreasing. Then we begin to find the next set of parameters to continue training the network.

B. Strategy Based on Cluster Reuse

This second strategy is much simpler than the first one. It only adjusts the decision on turning on or off cluster reuse. We start the training with cluster reuse. When the loss value stops dropping, we set $CR = 0$ and continue training without cluster reuse. It leaves L and H unchanged; they are set as certain manually tuned values (more details in Section VI-B) and stay unchanged throughout the training process.

VI. Evaluation

To validate the hypothesis on neuron vector similarity and to evaluate the efficacy of the *adaptive deep reuse*, we experiment with three different networks: CifarNet, AlexNet [16] and VGG-19 [17]. Table II gives the details of the networks and datasets. These three networks have a range of sizes and complexities. The number of convolutional layers ranges from 2 to 16. The first network works on small images of size 32×32 while 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 [1] in the TensorFlow framework [2]. We implement our *adaptive deep reuse* optimization by incorporating the clustering and reuse strategies into the TensorFlow code. Both the original and our optimized CNNs automatically leverage the state-of-the-art GPU DNN library cuDNN [3] and other libraries that TensorFlow uses in default.

We use policy 1, policy 2 and amendment 1.1 in Section V-A(a) to determine the ranges of *adaptive deep reuse* parameters L and H for each convolutional layer. During the training, we follow policy 3 and amendment 3.1, 3.2, 3.3 in Section V-A(b) to determine how to change the values of L and H for each convolutional layer. The same rules are applied to all the two datasets and three networks in our experiments.

All the experiments are done on a machine with an Intel(R) Xeon(R) CPU E5-1607 v2 and a GTX1080 GPU.

The metric we use to evaluate the influence on the CNN from the clustering based reuse is *reuse-caused accuracy loss*. As *adaptive deep reuse* uses the centroid of a cluster of neuron vectors as the representative of other neuron vectors in the same cluster in computations, there could be a loss on the inference accuracy of the neural network compared to the inference accuracy of the default network. This loss is referred as the "reuse-caused accuracy loss". If the resulting inference accuracy is close to the original inference accuracy, the reuse-caused accuracy loss is small. Then the corresponding clustering method, together with the set of parameters, is considered to have given good clustering results.

In the remaining of this section, we first verify our assumption of neuron vector similarity by applying the K-means clustering method to the inputs neuron vectors on CNN inference. This set of experiments takes a CNN model trained by the default training method, and applies our optimization only to the inference process. The results on the three networks show similar trends, confirming that there are strong similarities among neuron vectors across inputs when CNN runs on real-world datasets. Details are discussed in Section VI-A.

We then apply LSH to CNN inference to study the relationship between the clustering parameters, the remaining ratio and the inference accuracy. Similarly, the experiments only apply our optimization to the inference process. Section VI-B1 gives a more detailed discussion on these relations on all three networks.

Finally, we evaluate the efficiency of different deep reuse strategies in Section VI-B. This set of experiments applies our technique to both the training and the inference processes.

A. Verification of Neuron Vector Similarity

Figure 7 shows the r_c-accuracy relationships when k-means clustering is applied to CifarNet. We use k-means for this measurement because this slower clustering method produces better clustering results and hence can more fully expose the potential. The results on the three networks show simliar trends. Figure 7(a) shows the result for the first convolutional layer of CifarNet, while Figure 7(b) gives the result on the third convolutional layer of AlexNet. The results of two different scopes (single-input level and single-batch level) are shown. The inference accuracy of the original CifarNet is around 0.81 while the inference accuracy of the original AlexNet is around 0.54.

We could see that, by grouping the row vectors into clusters and reusing the computation results of the centroid vectors, we can reach an accuracy close or equal to the original accuracy with a relatively small remaining ratio r_c. If only applying k-means to the first convolutional layer of CifarNet, as shown in Figure 7(a), the accuracy reaches 0.76 with $r_c = 0.5$ when using single-input level clustering. As for the third convolutional layer of AlexNet, the accuracy reaches close to the original one with $r_c \sim 0.5$ for single-input level clustering and $r_c \sim 0.15$ for single-batch level clustering (Figure 7(b)). This observation verifies that there is a large amount of similarities among neuron vectors, hence the potential for computation savings.

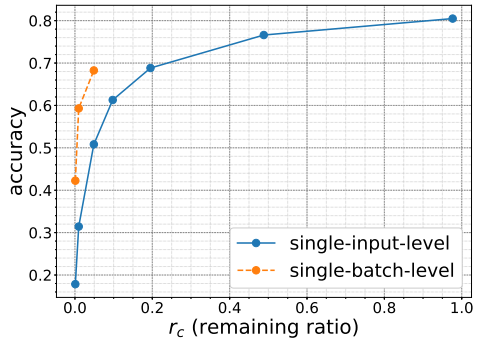
[1] https://github.com/tensorflow/models/tree/master/research/slim

[2] https://github.com/tensorflow/tensorflow

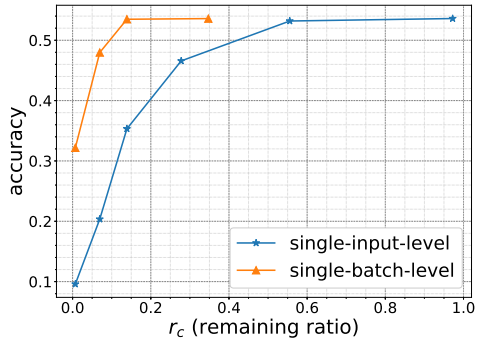
[3] https://developer.nvidia.com/cudnn

Table II
BENCHMARK NETWORKS

NETWORK	DATASET	# CONVLAYERS	K	M	IMAGE ORDER	IMAGE SIZE
CIFARNET	CIFAR10	2	$75 \sim 1600$	64	RANDOM	32×32
ALEXNET	IMAGENET	5	$363 \sim 3456$	$64 \sim 384$	RANDOM	224×224
VGG-19	IMAGENET	16	$27 \sim 4068$	$64 \sim 512$	RANDOM	224×224



(a) CifarNet conv1



(b) AlexNet conv3

Figure 7. The r_c-accuracy relationship of applying k-means clustering to CifarNet and AlexNet. Two clustering scopes are used: the single-input-level clustering and the single-batch-level clustering.

Comparing the curve of the single-batch level clustering and that of the single-input level clustering, it is easy to see that, with a larger clustering scope, the optimized network could recover the original accuracy with a smaller r_c. For the first convolutional layer of CifarNet (Figure 7 (a)), the curve of the single-batch level clustering are shorter than the single-input level one because there are no data when r_c exceeds 0.1 in the single-batch case. The reason is that K-means clustering at batch level requires a large amount of memory, causing memory errors on the machine.

B. *Efficiency of the Adaptive Strategies*

This part reports the relationship among the clustering parameters of LSH, the remaining ratio r_c, and the inference accuracy. It also reports the comparison between the computation time savings of adaptive strategies and analyzes the influence of *adaptive deep reuse* on CNN convergence rate.

Table III
COMPARISON OF ACCURACY BETWEEN INFERENCES WITH CLUSTER REUSE (CR=1) AND WITHOUT CLUSTER REUSE (CR=0).

LAYER	L	H	ACCURACY	
			CR=0	CR=1
CONV1	5	15	0.813	0.799
CONV2	10	10	0.816	0.784

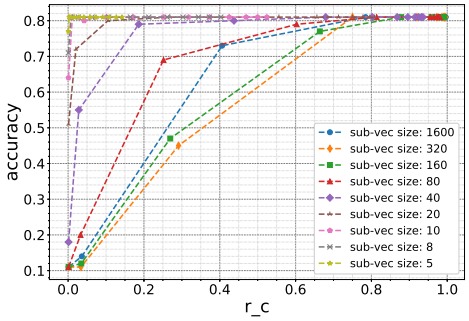
1) Relation between Clustering Parameters, Remaining Ratio, and Inference Accuracy: There are three clustering parameters for LSH clustering: the sub-vector length L, the number of hashing functions H and the flag of turning on cluster reuse CR.

Figure 8 illustrates the r_c-accuracy relationship of using different sub-vector lengths and different numbers of hashing functions. Each curve in the Figure corresponds to a sub-vector length. For example, in Figure 8 (a), the length varies from 5 to 1600 for the second convolutional layer of CifarNet. Each dot on the curve corresponds to a certain number of hashing functions. In Figure 8 (a), it varies from 5 to 60.

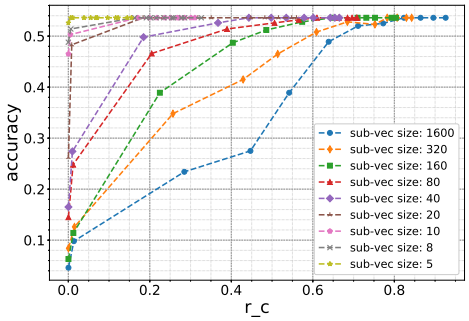
The results show that LSH is effective in identifying the neuron vector similarities. It can recover the original inference accuracy with a very small remaining ratio r_c. We can also tell that with the same remaining ratio r_c, a smaller sub-vector length L tends to give higher accuracy. For a fixed sub-vector length, a larger number of hashing functions are necessary to provide a higher accuracy, which incurs large remaining ratio r_c and hence many remaining computations.

Table III shows the effects of cluster reuse. The results are from the experiments performed on the two convolutional layers of CifarNet. For each layer, the selected set of $\{L, H\}$ is the one that performs the best in the previous experiments of studying the relation between clustering parameters and the inference accuracy. Results in Table III show that, for the optimal sets of $\{L, H\}$, using cluster reuse results in a lower accuracy for both of the two convolutional layers. However, based on our experiments result, cluster reuse helps remove most of the computations when processing later batches. For example, the reuse rate R increases from 0 to around 0.98 after processing 20 batches when applying cluster reuse on CifarNet [10]. It shows a trade-off between computation savings and inference accuracy.

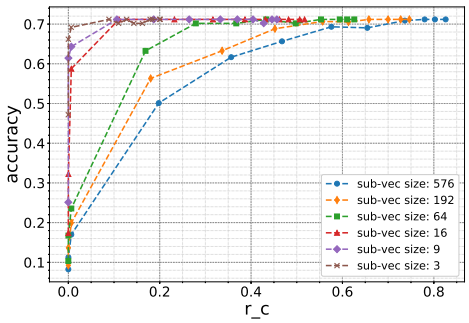
2) Comparison of Computation Time Savings between Three Different Strategies: In this section, we compare the computation savings of using three different strategies.



(a) conv2 in CifarNet



(b) conv2 in AlexNet



(c) conv2 in VGG-19

Figure 8. The r_c-accuracy relationship of applying LSH to CifarNet, AlexNet and VGG-19. Each curve corresponds to a certain sub-vector length. Each dot on a curve reflects the result of using a certain number of hashing functions.

The first strategy uses a fixed set of clustering parameters $\{L, H\}$ and it doesn't enable the cluster reuse. The $\{L, H\}$ set is the optimal one chosen from experiments result shown in previous section. With this strategy, we could save up to 49% CNN training time.

The second strategy is described in Section V-A. It automatically adjusts the parameter set $\{L, H\}$ for different training stages. It turns out that this strategy is very effective. For all the three networks, it could save more than 60% training time. The largest time saving is on AlexNet, which is 69%.

Comparing these two strategies, we find that the second one is more effective, giving larger speedups. For the first strategy, since it uses only one set of parameters, this set of $\{L, H\}$ must introduce little reuse-caused accuracy loss in order to reach the same training accuracy as the original network does. Therefore, the computation saving is limited. For the second strategy, the initial set of $\{L, H\}$ used at the beginning of the training actually gives large reuse-caused accuracy loss. However, it saves a huge amount of computations for the early training iterations. After several training iterations, the adjustment to $\{L, H\}$ gradually leads to smaller reuse-caused accuracy loss, but also less computation savings. Overall, the computation savings for the whole training process is larger than that of using the first strategy. This results in larger savings of computation time. We also experimented with the strategy of adjusting cluster reuse (Sec V-B); it is not as effective as the second strategy as Table IV shows.

Table IV
END-TO-END FULL NETWORK SPEEDUPS

NETWORK	SAVINGS OF THE CNN TRAINING TIME		
	STRATEGY 1	STRATEGY 2	STRATEGY 3
CIFARNET	38%	63%	46%
ALEXNET	49%	69%	58%
VGG-19	45%	68%	54%

It is worth noting that the speedups from *adaptive deep reuse* are significant, but not as significant as the computations savings it brings. The reason is that the reuse could lead to more epochs in training for reaching the same accuracy as the default training does: 28K versus 24K iterations for CifarNet, 820K versus 700K for AlexNet, and 500K versus 400K for VGG-19. The speedups we have reported have already taken into consideration of these extra training epochs.

VII. RELATED WORK

Training DNN with SGD involves a large number of computations for each training iteration and also many training iterations to converge. Prior works have adopted two main strategies to accelerate DNN training: 1) reducing the number of computations per iteration such as stochastic depth to remove some layers during training [18], randomized hashing to reduce the number of multiplications [6], approximate computations [19]; 2) reducing the number of iterations required to converge such as large-batch data parallelism [20], batch normalization to reduce internal covariate shift [21], importance sampling to reduce variance of gradient estimates [22], [23], adaptive learning rate [24]. We focus on the first strategy as our proposed *adaptive deep reuse* falls into this category.

Several recent works take advantage of the sparsity of activation maps to reduce computation cost in the forward and backward propagation. In [6], randomized hashing is combined with adaptive dropout [25] to predict the important neurons and conduct multiplications only for those important ones. Another work [8] uses the sparsity of ReLUs to avoid calculating zero-valued neurons. The most recent work [9] uses random projection to predict important neurons. These approaches usually require a high level of sparsity in activation maps to achieve speedups.

Approximate tensor operations are also able to speed up DNN training. One way for approximation is to use low precision. In [4], deep networks can be trained using only 16-bit wide fixed-point number representation using stochastic rounding, and incur little to no degradation in the inference accuracy. Speedups are also expected using mixed precision training proposed in [5]. Another popular approximation is to enforce a low-rank structure on the layers [2], [3]. These methods are all different from ours and can potentially be combined with *adaptive deep reuse*.

LSH, as a clustering method, has been used in some prior CNN studies [26]–[28]. But their purposes of using LSH differ from ours. For example, in the Scalable and Sustainable Deep Learning work [26], the authors apply LSH to both the weight vector and the input vector and find the collision between a pair of weight and input vectors. In this way they estimate the weight-input pairs that give the highest activation. In our work, we use the collision of hashing results of neuron vectors to figure out similarities among neuron vectors, and reuse the computing results of the neuron vector-weight vector products across similar neuron vectors to save computations.

VIII. Conclusion

This paper presents *adaptive deep reuse* as a technique to reduce the computation cost of the CNN training process. Experiments show that there is a large amount of similarities existing among neuron vectors across the inputs of each convolutional layer. By identifying these similarities using LSH in the forward prorogation and reusing the similarity results in the backward propagation, *adaptive deep reuse* efficiently leverages the similarities and enables deep computation reuses between neuron vectors that are similar to each other. *Adaptive deep reuse* also introduces adaptive strategies that adjust the clustering parameters throughout the CNN training to strike a good balance between computation savings and training errors. Experiments show that *adaptive deep reuse* can save up to 69% training time while causing no accuracy loss to the final training results.

Acknowledgement

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] S. V. Kamarthi and S. Pittner, "Accelerating neural network training using weight extrapolations," *Neural networks*, vol. 12, no. 9, pp. 1285–1299, 1999.

[2] F. Mamalet and C. Garcia, "Simplifying convnets for fast learning," in *International Conference on Artificial Neural Networks*. Springer, 2012, pp. 58–65.

[3] H. Bagherinezhad, M. Rastegari, and A. Farhadi, "Lcnn: Lookup-based convolutional neural network," in *Proc. IEEE CVPR*, 2017.

[4] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

[5] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[6] 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*. ACM, 2017, pp. 445–454.

[7] X. Sun, X. Ren, S. Ma, and H. Wang, "meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting," *arXiv preprint arXiv:1706.06197*, 2017.

[8] S. Shi and X. Chu, "Speeding up convolutional neural networks by exploiting the sparsity of rectifier units," *arXiv preprint arXiv:1704.07724*, 2017.

[9] L. Liu, L. Deng, X. Hu, M. Zhu, G. Li, Y. Ding, and Y. Xie, "Dynamic sparse graph for efficient deep learning," *arXiv preprint arXiv:1810.00859*, 2018.

[10] N. Lin and S. Xipeng, "Similarities in Neuron Vectors and The Implications to CNN Inferences," Tech. Rep. TR-2018-2.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in *European Conference on Computer Vision*. Springer, 2016, pp. 646–661.

[19] M. Adelman and M. Silberstein, "Faster neural network training with approximate tensor operations," *arXiv preprint arXiv:1805.08079*, 2018.

[20] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet training in minutes," in *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 2018, p. 1.

[21] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[22] A. Katharopoulos and F. Fleuret, "Biased importance sampling for deep neural network training," *arXiv preprint arXiv:1706.00043*, 2017.

[23] ——, "Not all samples are created equal: Deep learning with importance sampling," *arXiv preprint arXiv:1803.00942*, 2018.

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[25] J. Ba and B. Frey, "Adaptive dropout for training deep neural networks," in *Advances in Neural Information Processing Systems*, 2013, pp. 3084–3092.

[26] 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.

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

[28] 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.