2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

# Gist: Efficient Data Encoding for Deep Neural Network Training

Animesh Jain[†⋆], Amar Phanishayee[§], Jason Mars[†], Lingjia Tang[†] and Gennady Pekhimenko[‡]
*Microsoft Research[§], Univerity of Michigan, Ann Arbor[†], University of Toronto[‡]*
*anijain@umich.edu, amar@microsoft.com, profmars@umich.edu, lingjia@umich.edu, pekhimenko@cs.toronto.edu*

***Abstract*—Modern deep neural networks (DNNs) training typically relies on GPUs to train complex hundred-layer deep networks. A significant problem facing both researchers and industry practitioners is that, as the networks get deeper, the available GPU main memory becomes a primary bottleneck, limiting the size of networks it can train.**

**In this paper, we investigate widely used DNNs and find that the major contributors to memory footprint are intermediate layer outputs (feature maps). We then introduce a framework for DNN-layer-specific optimizations (e.g., convolution, ReLU, pool) that significantly reduce this source of main memory pressure on GPUs. We find that a feature map typically has two uses that are spread far apart temporally. Our key approach is to store an encoded representation of feature maps for this temporal gap and decode this data for use in the backward pass; the full-fidelity feature maps are used in the forward pass and relinquished immediately.**

**Based on this approach, we present *Gist*, our system that employs two classes of layer-specific encoding schemes – lossless and lossy – to exploit existing value redundancy in DNN training to significantly reduce the memory consumption of targeted feature maps. For example, one insight is by taking advantage of the computational nature of back propagation from pool to ReLU layer, we can store the intermediate feature map using just 1 bit instead of 32 bits per value. We deploy these mechanisms in a state-of-the-art DNN framework (CNTK) and observe that *Gist* reduces the memory footprint to upto 2× across 5 state-of-the-art image classification DNNs, with an average of 1.8× with only 4% performance overhead. We also show that further software (e.g., CuDNN) and hardware (e.g., dynamic allocation) optimizations can result in even larger footprint reduction (upto 4.1×).**

***Keywords*-DNN Training; Data Encodings; Compression;**

## I. Introduction

The availability of large datasets and powerful computing resources has enabled a new breed of deep neural networks (DNNs) to solve hitherto hard problems such as image classification, translation, and speech processing [22], [24], [31], [48]. These DNNs are *trained* by repeatedly iterating over datasets. The DNN training process has large compute and memory requirements and primarily relies on modern GPUs as the compute platform. Unfortunately, as DNN models are getting larger and deeper, the size of available GPU main memory quickly becomes the primary bottleneck[1], thus limiting the size of the DNNs that a GPU can support [39], [40]. Modern DNNs are already facing this issue, prompting researchers to develop memory-efficient implementations of the networks [37].

Many researchers have recognized this shortcoming and proposed approaches to reduce the memory footprint of DNN training. However, prior approaches are not able to simultaneously achieve all of the following three desirable properties: (i) *provide high memory footprint reduction, (ii) low-performance overhead, and (iii) minimal effect on training accuracy*. Most prior works propose efficient techniques to reduce the memory footprint in DNN *inference* with an emphasis on reducing the model size (also referred to as weights) [18], [19], [20], [17]. However for DNN training, weights are only a small fraction of total memory footprint. In training, intermediate computed values (usually called *feature maps*) need to be stored/stashed in the forward pass so that they can be used later in the backward pass. These feature maps are the primary contributor to the significant increase in memory footprint in DNN training compared to inference. This important factor renders prior efforts, that target weights for memory footprint reduction, ineffective for training. State-of-the-art memory footprint reduction approaches for training transfer data structures back and forth between CPU and GPU memory but pay a performance cost in doing so [39]. Finally, approaches that explore lower precision computations for DNN training, primarily in the context of ASICs and FPGAs, either do not target feature maps (and thus unable to achieve high memory footprint reduction) or, when used aggressively, result in reduced training accuracy [11], [15], [8].

The key insight of this work is in acknowledging that a feature map typically has two uses in the computation timeline and that these uses are spread far apart temporally. Its first use is in the forward pass and second is much later in the backward pass. Despite these uses being spread far apart, the feature map is still stashed in single precision format (32-bits) when they are unused between these accesses. We find that we can store the feature map data with efficient encodings that result in a much smaller footprint between the two temporal uses. Furthermore, we propose that if we take layer types and interactions into account, we can enable highly efficient *layer-specific encodings* – these opportunities are missed if we limit ourselves to a layer-agnostic view. Using these key insights, we design two

⋆Work done during an internship at Microsoft Research, Redmond.

[1]For GPU main memory (GDDR5/GDDR5X), the first order concern is bandwidth as many GPU applications are bandwidth-bound. It is hard to get both high bandwidth and high-density DRAM-based memory at low cost [33].

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00070

layer-specific lossless encodings and one lossy encoding that are *fast, efficient in reducing memory footprint, and have minimal effect on training accuracy*.

Our first lossless encoding, Binarize, specifically targets ReLU layers followed by a pooling layer. Upon careful examination of ReLU's backward pass calculation, we observe that the ReLU output, that has to be stashed for the backward pass, can be encoded using just 1-bit values, leading to 32× compression for the ReLU outputs. Our second lossless encoding, Sparse Storage and Dense Compute (SSDC), that specifically targets ReLU followed by convolution layer, is based on the observation that ReLU outputs have high sparsity. SSDC facilitates storage in memory-space efficient sparse format but performs computation in dense format, retaining the performance benefits of highly optimized cuDNN dense computation, while exploiting sparsity to achieve high reduction in memory footprint. Finally, in the lossy domain, our key insight of representing the stashed feature maps in smaller format *only* between the two temporal uses enables us to be very aggressive with precision reduction without any loss in accuracy compared to baseline. This lossy encoding, Delayed Precision Reduction (DPR), delays precision reduction to the point where values are no longer needed in the forward pass and achieves significant bit savings (as small as 8 bits).

Utilizing all these encodings, we present *Gist* that specifically targets feature maps to reduce the training memory footprint. It performs a static analysis on the DNN execution graph, identifies the applicable encodings, and creates a new execution graph with relevant encode and decode functions inserted. *Gist* also performs a static liveness analysis of the affected feature maps and newly generated encoded representations to assist the DNN framework's memory allocator, CNTK [43] in our case, to achieve an efficient memory allocation strategy.

This paper makes the following contributions:

- **Systematic Memory Breakdown Analysis.** We perform a systematic memory footprint analysis, revealing that *feature maps* are the major memory consumers in the DNN training process. We also make a new observation that the feature maps have high data redundancy and can be stored in much more efficient formats between their forward and backward use.
- **Layer-specific Lossless Encodings.** We present two layer-specific encodings – (1) Binarize that achieves 32× compression for ReLU outputs for layer combination of ReLU followed by Pool, and (2) Sparse Storage and Dense Compute that exploits high sparsity exhibited in ReLU outputs for ReLU followed by convolution layers.
- **Aggressive Lossy Encoding.** We present DPR, that applies precision reduction only for the backward use of the feature maps – values in the forward pass are kept in full precision – leading to aggressive bit savings without affecting accuracy.
- **Footprint Reduction on a Real System.** We observe that *Gist* reduces the memory footprint by 2× across 5 state-of-the-art image classification DNNs, with an average of 1.8× with only 4% performance overhead. By reducing memory footprint, *Gist* can fit larger minibatches in the GPU memory, improving GPU utilization and speeding up the training for very deep networks, e.g., a speedup of 22% for Resnet-1202. We also show that further optimizations to existing DNN libraries and memory allocation can result in even larger memory footprint reductions (upto 4.1×).

## II. Background and Motivation

DNNs typically consist of an input and an output layer with multiple hidden layers in between. Recently, convolution neural networks (CNNs), a class of DNNs, have been shown to achieve significantly better accuracy compared to previous state-of-the-art algorithms for image classification [32], [45], [46], [22], [34]. CNNs have been growing deeper with every iteration, starting with few convolution layers in the beginning (AlexNet) to hundreds of convolution layers in recent ones (Inception).

### A. Training vs. Inference

DNNs have two distinct modes of operation: (i) *training*, when a model is trained based on a set of inputs (training set) and a corresponding set of expected outputs, and (ii) *inference*, when an already trained network is used to generate predictions for new inputs.

For this paper, we focus on two main differences between training and inference. First, training consists of two phases: *forward* and *backward* passes [41], [10]; inference only involves a forward pass. The goal of the backward pass in DNN training is to backpropagate error and find weight error gradients that can be applied to the weights to steer the parameters in the *right* direction. Training is performed in batches of input samples, commonly known as a mini-batch [3]. Training on minibatches as opposed to training on an image-by-image basis has been shown to achieve better accuracy and better hardware utilization [12], [21], [9].

Second, in inference, the major part of storage overhead comes from *weights*. These weights are fixed after training, and hence many different optimizations can be applied to reduce their storage requirements [17], [2], [18], [5]. In contrast, training has many distinct data structures, e.g., weights that change over the course of training, weight gradients, feature maps (intermediate layer outputs) that need to be stashed in the forward pass for use in the backward pass, and backward gradient maps.

*1) Why Memory Can Be a Problem in Training?:* To understand the memory requirements of GPU-based training, we study the breakdown of memory footprint on five state-of-the-art CNNs in CNTK. While using FPGAs [50] and ASICs [28], [25] is also possible, most of these designs are either proprietary or in relatively early development stages.

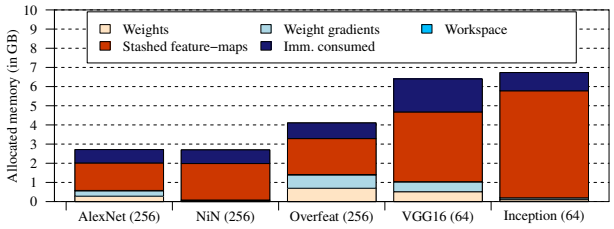



**Figure 1:** Breakdown of memory footprint in DNN training amongst different data structures

Hence, we conduct our study on a modern GPU (Maxwell GTX Titan X in our case). However, our approaches are applicable to optimized hardware as well (Section V-H).

Figure 1 shows the breakdown of total memory footprint across different data structures. Feature maps are the intermediate layer outputs that are passed on as an input to the following layer. Gradient maps are the intermediate gradients generated in the backward pass and passed as an input to the previous layer. In CNNs, not every feature map has to be saved for the backward pass. We thus distinguish *stashed* feature maps (generated in the forward and used in both forward and backward passes) from *immediately consumed* feature maps (generated in the forward pass and consumed immediately in the forward pass) and gradient maps (generated in the backward pass and consumed immediately). Stashed feature maps are required in the backward pass and thus stored for a long time in a minibatch processing. In contrast, immediately consumed feature maps and gradient maps can be discarded as soon as they are used. Finally, *workspace* is deep learning library's (cuDNN in this case) intra-layer storage to support layer computations [7]. cuDNN provides a choice between memory-optimal and performance-optimal implementations, translating to a tradeoff between algorithm performance and workspace storage requirements. In this work, we choose its memory-optimal implementation as an optimized baseline.

We draw two major conclusions from this figure. First, larger (deeper) DNNs consume a large amount of memory even with relatively small minibatch sizes (64). VGG16 and Inception can only fit in our GPU memory if the minibatch size is 64 and start exceeding the 12 GB GPU memory limit at higher minibatch size. Higher minibatch size is desirable as it leads to better GPU utilization [12], [51]. Second, training memory footprint tends to be dominated primarily by stashed feature maps, followed by immediately consumed data structures. For example, in VGG16, 83% of memory is consumed by stashed feature maps and immediately consumed data; this number grows to 97% for Inception. This result stands in stark contrast to inference where feature maps don't need to be stashed and memory consumption is dominated by weights. We conclude that the stashed feature maps and immediately consumed data structures (in that order of importance) are key for optimizing GPU memory consumption in CNN training.

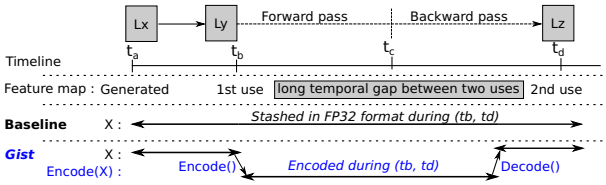



**Figure 2:** Long temporal gap between uses of a feature map

### B. Limitations of Prior Work

In this paper, we develop techniques to reduce the DNN training memory footprint. Here, we briefly describe the limitations of existing approaches.

**Prefetch and Swap-out.** One potential approach is to move parts of the working set between CPU and GPU memory using PCIe links and smart prefetching analysis [39]. However, this approach still suffers from significant overheads of data transfer with respect to power/energy and their inability to completely mask the performance cost of swapping data in and out of GPU memory (upto 27% for Inception). In addition, it uses a shared resource, PCIe links, which sometimes can be of critical importance in distributed DNN training [9].

**Reducing Minibatch Size.** While reducing minibatch size is effective at reducing the memory footprint during training, it adversely affects the runtime of the training process because smaller minibatches lead to GPU underutilization [12]. This performance hit can be recovered by using more GPUs, where each GPU works on a smaller minibatch. But this is also an inefficient solution as GPU machines are both costly and power hungry, and might also result in sub-linear scaling due to stragglers and transfer across workers [9].

**Recompute.** Instead of a saving the output of a large layer, prior work has considered recomputing the output of a layer's forward pass again in the backward pass [4]. Unfortunately, we observe that the largest layers are usually the ones that also take the longest to recompute, that can cause significant performance overhead. Yet, this technique is still applicable for some specific layers (like batch normalization) and can be used in conjunction with our work.

## III. Gist: Key Ideas

In this work, we design techniques to reduce DNN training memory footprint by focusing on the primary contributors – feature maps. We find that a feature map typically has two uses in the computation timeline and these uses are spread far apart temporally, as shown in Figure 2. Its first use is in the forward pass and the second use is much later in the backward pass. In the baseline, the data is stashed in single precision (FP32) even though these uses are far apart. In our approach, we represent the data in a much smaller encoded format in the temporal gap and decode it just before it is needed again in the backward

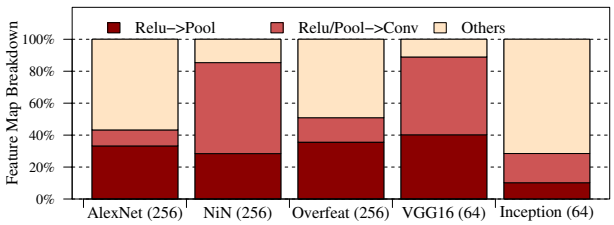



**Figure 3:** Breakdown of memory within stashed feature maps. ReLU layer consumes a major portion of the footprint.

pass; the forward use still gets the data in FP32 format, but the memory space is relinquished as soon as the forward use is complete. This results in an efficient memory sharing strategy (Section IV-C), reducing total memory footprint.

| Target data structures | Footprint Reduction Technique | Type |
|---|---|---|
| ReLU-Pool feature map | *Binarize* | Lossless |
| ReLU-Conv feature map | *Sparse Storage and Dense Compute* | Lossless |
| Other feature map | *Delayed Precision Reduction* | Lossy |
| Immediately consumed | *Inplace computation* | Lossless |

Table I: Summary of *Gist* techniques

Our second key insight is that taking layer types and interactions into account opens up opportunities for designing highly aggressive encoding schemes, which were earlier hidden due to the layer agnostic nature of prior work. In this section, we first identify such opportunities that let us design two lossless and one lossy encodings specifically targeted to reduce memory footprint of stashed feature maps. Next, we observe that inplace computations can reduce the size of immediately consumed data structures. Table I shows a brief outline of our techniques and their target data structures.

### A. Opportunities For Lossless Encodings

Among feature maps, we first target ReLU activation functions which are heavily used in all major CNNs targeted for vision-related tasks [22], [48], [13], [31]. In these CNNs, the convolution layers are typically followed by ReLU activations. This group of a Conv-ReLU pair is either followed by the same group or by a pool layer, resulting in many ReLU-Conv and ReLU-Pool layer pairs. Consequently, we observe that ReLU feature maps form a major fraction of total memory footprint.

This is shown in Figure 3 which zooms in on the stashed feature maps (in Figure 1) and analyze their breakdown across different CNN layers, with an emphasis on three different categories: (i) ReLU outputs followed by a Pool layer (*ReLU-Pool*), (ii) ReLU/Pool outputs followed by a conv layer (*ReLU-Conv*), and (iii) remaining stashed feature maps (*Others*). We observe that significant portion of memory footprint is attributed to *ReLU outputs* (Pool outputs have very low contribution). For example, VGG16 has 40% and 49% of the stashed feature maps for ReLU-Pool and ReLU-Conv respectively (89% total used for ReLU outputs).

We make two key observations for these ReLU outputs that let us store the stashed feature map with much fewer bits. First, when carefully examining the backward pass computation of ReLU layer, we observe that ReLU outputs for ReLU-Pool combination can be stored in just 1-bit. Second, we observe that ReLU outputs typically have high sparsity that can be exploited to encode the feature map in much smaller sparse format. Next, we expand on these opportunities.

**ReLU-Pool.** Typically, in a backward pass calculation, a layer uses its stashed input feature map (X), stashed output feature map (Y) and output gradient map (dY) to calculate input gradient map (dX), as shown in Figure 4(a). However, every layer does not require all this data for the backward pass. Upon further investigation, we discover that although feature maps are stashed with the same precision across layer types, it is mostly for convenience, not out of computation necessity. We show backward pass calculation for *ReLU* in Figure 4(b). It only requires Y and dY to calculate dX. Moreover, an element of dY is passed to dX, only if the corresponding element in Y is positive, else dX is set to 0. With this observation in mind, it is natural to consider replacing Y with 1-bit values. Unfortunately, it is not always possible, because the next layer might require its stashed input feature map X (ReLU output in this case) for the backward pass calculation. However, upon further examination, we observe that in the case of ReLU-Pool, the pool backward pass does not require the actual values of ReLU output as shown in Figure 4(c) (described in more details in Section IV-A), resulting in significant encoding opportunities. This observation becomes the basis of our first lossless encoding, called Binarize.

**ReLU-Conv.** Binarize is not applicable to ReLU-Conv pair, because convolution requires its stashed input feature map for the backward pass calculation (as shown in Figure 4(d)). However, upon careful data analysis, we observe that ReLU outputs have high sparsity (large number of zeroes) induced by the ReLU calculations in the forward pass. For example, for VGG16, we observe high sparsity, going even over 80%, for all the ReLU outputs, motivating us to apply sparse compression and computation for these feature maps. However, switching both compute and memory to sparse domain results in significant performance degradation [49], [23]. Building on this observation, we present Sparse Storage and Dense Compute (SSDC) encoding that stores the data in Compressed Sparse Row (CSR) encoding format while keeping the computation in dense format.

### B. Opportunities For Lossy Encodings

For lossy encoding, we investigate precision reduction as it is amenable to GPU architecture compared to prior offline approaches like quantization and Huffman encoding [18]. We make two observations that let us achieve aggressive bit savings without any loss in accuracy.

**Non-Uniform Precision Reduction.** We observe that *uniform* precision reduction to even 16 bits (IEEE half precision

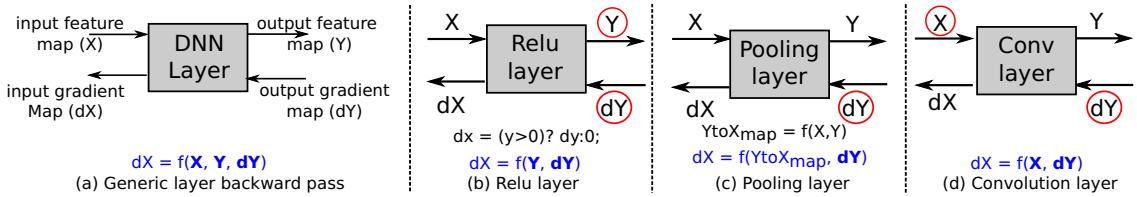



**Figure 4:** Backward pass computation. Only circled data structures are needed in the backward pass

format), where all the data structures are represented in 16 bits while the computation happens on 32 bits, leads to severe accuracy losses. We observe that if instead, we restrict the precision reduction to only gradient maps, then the training accuracy is not affected. Note that, although this type of selective precision reduction has been studied in a recent work [11], it does not study the effect of reducing precision in stashed feature maps.

**Delayed Precision Reduction.** But more importantly, we find that the way the precision reduction is applied in training should be significantly changed. Currently, the conventional wisdom [11], [15], [8], [16] is to apply precision reduction right after the value is generated by a particular layer. This design choice leads to the situation when the error generated by the precision reduction is injected directly into the next layer and is then propagated (potentially increasing in magnitude) to all future layers in the forward pass. In our work, we observe that it is better to separate the two uses of every output layer in the forward and backward pass (as previously shown in Figure 2). The first immediate use (by the next layer) significantly benefits from the more precise (usually FP32) representation, avoiding any error injection due to precision reduction in the forward pass. While the second use, much later in the backward pass, can tolerate lower precision. This separation, implemented in our Delayed Precision Reduction encoding, push the bit lengths to a very small value like 8 bits for multiple DNNs (unseen in prior work).

### C. Opportunities For Inplace Computation

Next, we shift our focus from stashed feature maps to immediately consumed data structures. We observe that a good portion of immediately consumed data can be removed by performing inplace computation. As discussed in a previous work [4], this optimization is applicable for the layers (specifically ReLU) that have a *read-once and write-once* property between each element of input and output. In the absence of inplace optimization, convolution output shows up in the immediately consumed category. With inplace computation, the memory space for convolution is reused by ReLU, reducing immediately consumed memory footprint.

## IV. Design and Implementation

Based on the observations in the previous section, we present the design of our system *Gist*. Figure 5 shows our system architecture. Typically, DNN frameworks like

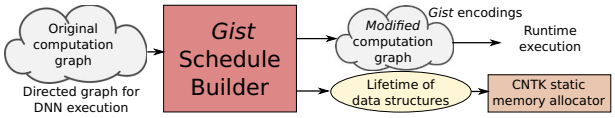



**Figure 5:** *Gist* System Architecture - Schedule Builder finds applicable *Gist* encodings and performs liveness analysis.

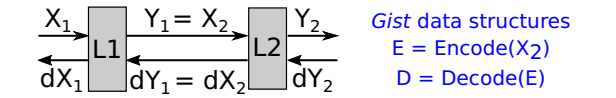
CNTK [43] and TensorFlow [1] represent DNNs as a series of computational steps via a directed execution graph. *Gist*'s *Schedule Builder* takes the original execution graph, identifies the edges where new encodings/decodings are needed, and creates a new execution graph with encode/decode functions inserted.

In isolation, encoded representations only add to the memory footprint. To solve this problem, we utilize the CNTK memory allocator that uses the lifetimes of various data structures to find an efficient static memory *sharing* strategy. Schedule Builder performs liveness analysis, infers the lifetime of affected stashed feature maps and encoded representations, and presents them to the CNTK memory allocator for optimization. *Gist* encodings reduce the lifetime of FP32 stashed feature maps, opening up more opportunities for memory sharing, and thus reducing the total memory footprint. In this section, we present the details of *Gist* encodings and design of *Gist*'s Schedule Builder and its interaction with CNTK memory allocator.

### A. Encodings

We present a generic view of encodings in Figure 6, illustrating the difference between baseline and *Gist* encodings in terms of new data structures and changes in backward pass calculations (shown in blue color). For the baseline, backward pass calculation is a function of intermediate feature map ($Y_1$ in the figure) along with other data structures. *Gist* introduces two new data structures – *E*, the Encoded intermediate feature map in the forward pass, and *D*, the decoded intermediate feature map in the backward pass. These new data structures change the backward pass calculation, which are now dependent on D instead of $Y_1$. We now present the details of these encodings.

**Lossless Encoding - Binarize.** As discussed in Section III, we observe that for ReLU-Pool layer combination, ReLU output can be encoded much more efficiently, because (i) the backward pass of ReLU only needs to know if the stashed feature map is positive (1 bit), and (ii) the pool layer backward pass can be optimized so that it does not




| | |
|---|---|
| *Baseline:* | $dX_1 = f(\mathbf{Y_1}, X_1, dY_1)$, $dX_2 = f(\mathbf{X_2}, Y_2, dY_2)$ |
| *Gist:* | $dX_1 = f(\mathbf{D}, X_1, dY_1)$, $dX_2 = f(\mathbf{D}, Y_2, dY_2)$ |

| *Layer pairs* | *Encode function* | *Decode function* |
|---|---|---|
| ReLU - Pool | FP32 to 1-bit | Work on encoded data |
| ReLU/Pool - Conv | FP32 to CSR-like | CSR-like to FP32 |
| Others | FP32 to FP8/10/16 | FP8/10/16 to FP32 |

**Figure 6:** *Gist* encodings

need ReLU outputs.

CNNs typically use MaxPool layer to subsample the input matrix. MaxPool's forward pass slides a window of a specific size over the input matrix X, finds the maximum value in this window, and passes it on to the output Y. For the backward pass, it passes on dY to that location of dX in the window, from where the maximum value of X was chosen in the forward pass. In baseline CNTK implementation, the MaxPool layer stashes both input and output feature maps for the backward pass to find the location of maximum values. We instead, create a mapping from Y to X in the forward pass that keeps track of these locations ($YToX_{map}$ in Figure 4 (b)). We use this mapping in the MaxPool backward pass calculation, removing the dependence on its input and output stashed feature maps.

With this optimization, Binarize encoding lets us achieve a significant reduction in memory footprint for ReLU-Pool layer combination. Binarize adds two encoded feature maps. First, a 1-bit data structure that replaces the ReLU feature map, storing information whether the stashed feature map value was positive. Second, for Pool layer, Binarize stores a Y to X mapping of the location of maximum values. This data structure has as many elements as the Pool output (typically, one-fourth or one-ninth of preceding ReLU output), where each element is stored in 4 bits (the largest sliding window in our application suite is $3 \times 3$). Therefore, these encoded data structures result in a compression of close to $16\times$ ($32\times$ for ReLU output and $8\times$ for MaxPool output) for ReLU-Pool stashed feature maps.

Tying back to Figure 6, both encode and decode functions are implemented within ReLU and pool layers using CUDA. Pool and ReLU backward pass have been updated to perform computation on the encoded data structures itself. We observe small performance improvements with Binarize encoding, because Binarize encoding significantly increases effective memory bandwidth for ReLU layers, improving memory-bandwidth bound ReLU backward pass computation.

**Lossless Encoding - Sparse Storage and Dense Compute.** As discussed in Section II, other set of ReLU layers – ReLU-Conv, exhibits high amount of sparsity induced by ReLU calculations, making it suitable to store these feature maps in sparse format. We also observe high sparsity for a few Pool-Conv layer combinations if the preceding ReLU layer has high sparsity. SSDC encoding is applicable to both layer combinations.

However, switching to sparse computation on GPUs shows performance improvements only when the sparsity is very high ($> 97.5\%$), which is typically not the case in CNNs [49], [23]. To tackle this problem, we present Sparse Storage and Dense Compute (SSDC) encoding, that isolates computation and storage, facilitating storage of data in sparse format and computation in dense (FP32) format. SSDC stores the data in a sparse format for the majority of its lifetime, and converts the data back into dense format only before it is actually required for computation. This achieves significant memory footprint reduction, while retaining the performance benefits of highly optimized cuDNN library.

For choosing a suitable sparse format, we compare 3 commonly used formats - ELL, Hybrid and Compressed Sparse Row (CSR). We observe that CSR achieves lowest format-conversion latency among these options, achieving the best compression-performance overhead tradeoff. This format stores all the non-zero values, along with a meta array that holds the column indices of the non-zero values in each row. (There is an extra meta array which is very small in size, and, thus omitted for the rest of the discussion). Most DNN frameworks store data structures in an n-dimensional matrix, which can always be collapsed into two dimensions. We take these 2D matrices and convert them into a CSR format.

We use Nvidia cuSPARSE library to perform the encodings/decodings, listed in Figure 6. However, the original implementation stores each index as a 4-byte value, resulting in no improvement with compression if the sparsity is below 50%. This is due to cuSPARSE conservative assumption that the number of elements in a row of the matrix can be high, allotting 4 bytes for every column index. We perform *Narrow Value Optimization*, where we reshape the 2D matrix and restrict the number of columns to 256, requiring only 1 byte per column index [36]. This reduces the minimal sparsity requirement for compression to be effective from 50% to 20%, resulting in both wider applicability and higher compression ratios.

**Lossy Encoding - Delayed Precision Reduction.** *Gist*'s third encoding uses precision reduction to exploit CNN error tolerance to reduce the memory footprint of remaining stashed feature maps. Similar to SSDC encoding, we isolate the storage from the computation. The computation still happens in FP32 format while the data is stored with lower precision for most of its lifetime. Though backward pass implementation can be modified to work directly on precision-reduced values, we convert back to FP32 because cuDNN is closed-source library. Nevertheless, we discuss the impact of such optimization on compression ratio in Section V-H.

Our usage of precision reduction differs significantly from the previous research that applies it immediately after computation finishes. We delay the precision reduction until the feature map has been consumed in the forward pass, thus naming it Delayed Precision Reduction (DPR). This lets us achieve more aggressive precision reduction on GPUs. DPR is applicable to any layer combination. We also apply it over SSDC encoding, compressing the non-zero values array in the CSR format. We do not touch the meta array in CSR format and Binarize encoded data structures as these affect control, and thus are not suitable for lossy encoding.

Figure 6 lists the encode and decode functions for DPR encoding. We use three smaller representations of 16, 10 and 8 bits, packing 2, 3 and 4 values, respectively, into 4 bytes. For packing 3 values into 4 bytes, 10 bits is the largest length possible (9 bits leave 5 bits unused, 11 bits requires one extra bit). For 16 bits, we use IEEE half precision floating point format (1 sign, 5 exponent and 10 mantissa bits), referred to as FP16. For 8-bits (FP8), we choose 1 bit for sign, 4 for exponent and 3 for mantissa, and for 10-bits (FP10), we use 1 sign, 5 exponent and 4 mantissa bits. In FP10, three 10-bit values are stored in a 4-byte space, rendering 2-bits useless. We ignore denormalized numbers as they have a negligible effect on CNNs accuracy. We use *round-to-nearest* rounding strategy for these conversions. The value is clamped at maximum/minimum value if the FP32 value is larger/smaller than the range of the smaller format. We write CUDA implementations to perform these conversions. Since conversions can happen in parallel, DPR results in minimal performance overhead.

### B. Schedule Builder

In *Gist*, the values in the forward pass are in FP32 format (for both lossless and lossy encodings) while only the data that is required for the backward pass is stored in an encoded format. However, since the feature maps are still generated in the original FP32 format in the forward pass before they are encoded, without any further optimization, the encodings will result in increased memory footprint. This gives rise to the question that how does *Gist* leads to memory footprint reduction.

This task is handled by *Gist*'s Schedule Builder that has two responsibilities. First, identifying the applicable layer encodings from the CNTK execution graph, performing a static analysis to distinguish between forward and backward use of a feature map, and inserting the encode and decode functions in the execution graph, thus creating a new execution graph that is used at runtime. And, second, performing a static liveness analysis for the affected stashed feature maps and newly generated encoded/decoded representations, and pass it on to the CNTK static memory allocator that finds an efficient memory allocation strategy (Section IV-C).

Figure 2 illustrates the liveness analysis performed by the Schedule Builder. The figure shows the two uses of a feature map that are temporally far apart in the computation timeline – one in the forward pass and one much later in the backward pass. In the baseline, the lifetime of this feature map is very long and it is stored in FP32 format for this whole duration. *Gist* breaks this lifetime into three regions – FP32 format that is live only for the immediate forward use, encoded (much smaller) format that is live for the long temporal gap between the two uses, and FP32 format for the decoded value that is live only for the immediate backward use. The figure also shows the points at which Schedule Builder inserts encode and decode functions.

### C. CNTK Memory Allocator

Schedule Builder passes this liveness analysis to the CNTK static memory allocator that finds an efficient strategy to allocate the memory. CNTK, similar to other deep learning frameworks, performs static memory allocation. The other alternative, dynamic allocation, results in a lower footprint but at the expense of many expensive cudaMalloc calls for each minibatch, resulting in performance overhead. Nevertheless, we discuss the impact of dynamic memory allocation on compression ratio in Section V-H.

The key idea employed in the CNTK memory allocator is *memory sharing*. It takes lifetimes of different data structures and their sizes as input, and finds an efficient memory sharing strategy. The memory allocator creates groups of data structures whose lifetimes do not overlap and thus can share the same memory space. Therefore, the size of this group is the largest size of the member within the group, as opposed to the sum of size of the members of the group. To come up with an efficient strategy, it first sorts the data structures on the basis of size, and then forms these groups, so that large data structures can share the same memory space. At the end of this process, memory allocator has multiple groups which are either dominated by feature maps that are stored for the backward pass or by immediately consumed feature maps or gradient maps. *Gist* encodings, by reducing the lifetime of FP32 stashed feature map, create higher opportunities for memory sharing, resulting in lower memory footprint.

**Example - Putting it all together.** We present an example in Figure 7 to illustrate the interactions between static memory manager and *Gist* encodings. The example shows lifetimes of five variables (X, A, B, C and D). In part (a), we show the output of CNTK memory allocator for the baseline. Memory allocator forms 2 groups, resulting in a total size of 18 MB, 10 for stashed feature map (X) and 8 for immediately consumed. In part (b), we apply SSDC encoding, which breaks the lifetime of original X into three separate timelines. In this case, CNTK memory allocator again forms two groups, however, the total size is reduced to 12, 2 for (encoded) stashed feature map and 10 for immediately consumed. As shown in the figure, *Gist encodings convert the original FP32 stashed feature maps*

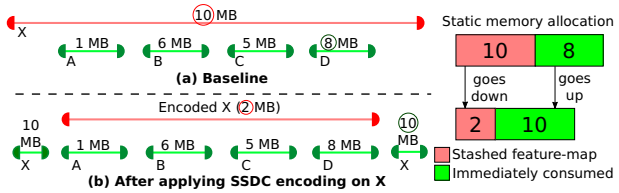



**Figure 7:** Example illustrating the interaction between *Gist* encodings and CNTK static memory allocator

*into immediately consumed data, and the encoded (and much smaller) data structure is now stashed for the backward pass.* This might increase the total immediately consumed data (8 to 10 MB here), but reduces the stashed feature map significantly (10 to 2 MB), resulting in overall reduction in memory footprint (18 to 12 MB).

## V. Evaluation

### A. Methodology

**Infrastructure.** We evaluate *Gist* memory reduction capabilities on Microsoft CNTK deep learning framework. We implement *Gist* encodings, inplace optimization, Schedule Builder, and make necessary changes in CNTK static memory allocator. The evaluation is performed on an Nvidia Maxwell GTX Titan X [26] card with 12 GB of GDDR5 memory using cuDNN v6.0 and CUDA 8.0.

**Applications.** We evaluate *Gist* on 6 state-of-the-art image classification CNNs: AlexNet [32], NiN [34], Overfeat [44], VGG16 [45], Inception [46] and Resnet [22], using ImageNet training dataset [42]. These CNNs present a wide range of layer shapes and sizes, while also capturing the evolution of CNNs in past few years.

**Baselines.** Our first baseline, referred to as *CNTK baseline*, is CNTK original static memory allocation strategy, without any of our optimizations. In Section II, we show that stashed feature maps and immediately consumed data structures are the major contributors to the total memory footprint, hence CNTK baseline consists of only these two data structures. It does not include weights, weight gradients, and workspace (also in line with previous work on DNN training [39], [40]).

We also use a second baseline to study the effect of different encodings in isolation. This baseline, referred to as *investigation baseline*, is modified CNTK baseline where memory sharing is not allowed for stashed feature maps. Other data structures are shared exactly the same way as in the CNTK baseline. This baseline allows us to study the impact of our encodings on different data structures in isolation. For end-to-end memory reduction numbers, we still use CNTK baseline.

For performance overhead evaluation, we use memory-optimized cuDNN configuration as the focus of our work is memory footprint reduction. Memory-optimized cuDNN presents an optimized baseline for comparison. Note that CNTK baseline and investigation baseline have the same performance as they do not affect computation.

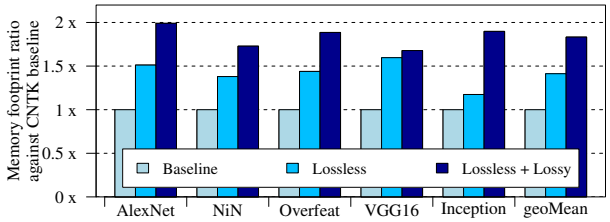



**Figure 8:** Evaluation of memory footprint reduction - *Gist* cuts down total memory footprint significantly

**Comparison Metric.** We use *Memory Footprint Ratio (MFR)* to evaluate the efficacy of *Gist* on reducing the memory footprint. MFR is described as follows.

$$MFR = \frac{Memory\ Footprint\ of\ Baseline}{Memory\ Footprint\ after\ encoding} \quad (1)$$

### B. Gist's Memory Footprint Reduction

*Gist* is designed to tackle the increasing memory footprint in DNN training. In this section, we evaluate *Gist*'s efficacy from this aspect. In this experiment, we, first, apply all lossless optimizations – Binarize, SSDC and inplace – and measure the total memory footprint (stashed feature maps and immediately consumed data structures). Then, we apply *Gist*'s lossy encoding – DPR – on top of lossless optimizations and measure additional reduction in memory footprint. For lossy encoding, we choose the smallest floating point representation that does not affect the training accuracy (detailed in Section V-D1). The findings of this experiment are presented in Figure 8.

Figure 8 shows the Memory Footprint Ratio (MFR) achieved by *Lossless* and *Lossy* optimizations when compared to CNTK *Baseline*. We observe that the lossless optimizations result in a MFR of more than 1.5× for AlexNet and VGG16 (1.4× on average). DPR, on top of lossless, further reduces the total memory footprint, achieving MFR of upto 2× for AlexNet, with an average of 1.8×. This experiment shows that *Gist* optimizations result in significant memory footprint reductions, making it possible to fit a network that can be larger in depth or wider with respect to larger minibatch size.

**Performance Overhead.** Next, we measure the performance overhead introduced by *Gist* encodings. We run the same experiment again, measuring the execution time of processing a minibatch, averaged over 5 minutes of training time (hundreds of minibatches). The findings of this experiment are presented in Figure 9. The figure shows the performance degradation for *Lossless* and *Lossy* encodings, with error bars capturing the performance variation. We observe minimal performance degradation across CNNs, resulting in an average 3% degradation for lossless and 4% for lossy and lossless optimizations combined, with a maximum overhead of 7% for VGG16 when both lossy and lossless

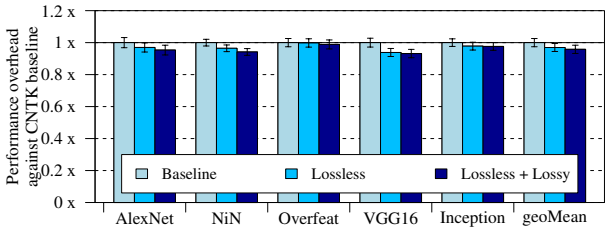



**Figure 9:** Performance overhead of *Gist* encodings. *Gist* results in minimal performance overhead

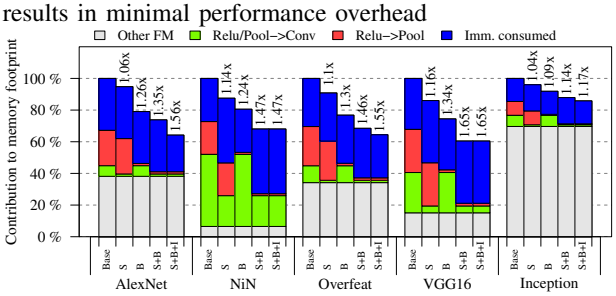



**Figure 10:** Impact of *Gist* lossless techniques (S - SSDC, B - Binarize, I - Inplace) on memory footprint of different data structures. Total MFR for each configuration is present at the top of each bar

optimizations are applied. This shows that *Gist* achieves significant MFR with minimal performance overhead.

### C. Lossless Encodings

In this section, we evaluate the impact of lossless techniques on memory footprint and performance.

*1) Impact on Memory Footprint:* In this experiment, we apply *Gist* lossless encodings – Binarize and SSDC – in isolation and evaluate how they affect the memory consumed by stashed feature maps and immediately consumed data structures. Then, we study the same effect when both encodings are applied together. Finally, we evaluate inplace optimization, that targets the immediately consumed data structures. The findings of this experiment are presented in Figure 10.

We perform this study on the *investigation baseline*, where stashed feature maps are not allowed in memory sharing, allowing us to study the impact of encodings in isolation. When an encoding is applied, the stashed feature map is converted to an immediately consumed data structure (possibly increasing its footprint), and this much smaller encoded data structure is now stashed for the backward pass, as discussed in Section IV-B. The figure shows this effect by breaking down the total memory footprint into 4 regions: ReLU/Pool-Conv (suitable for SSDC), ReLU-Pool (Binarize), other feature maps (untouched in this experiment as they are suitable for DPR), and immediately consumed.

The first bar shows the breakdown across these categories for the baseline. Then, we apply SSDC encoding, that reduces ReLU/Pool-Conv footprint significantly and slightly

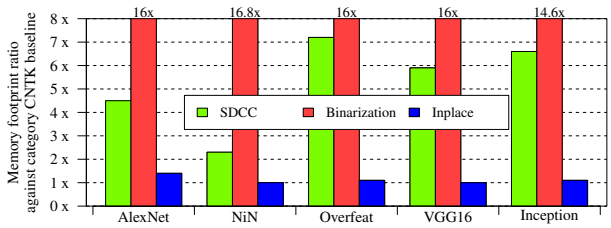



**Figure 11:** *Gist* lossless encodings MFR on target categories

increases the immediately consumed memory footprint. For example, for AlexNet, it results in the total MFR of 1.06×. Similarly, the third bar is for Binarize encoding, targeting ReLU-Pool category, resulting in MFR of 1.26×. Then, we apply both these encodings together, shown in the fourth bar, resulting in total of 1.35× MFR. Finally, we apply inplace optimization that targets immediately consumed data structure, further increasing the MFR to 1.56×. These techniques result in different footprint reduction for CNNs, as the proportion changes across different categories.

Note that Figure 10 only shows the total MFR, however *Gist* encodings reduce memory consumption of different categories to a different extent. We show this effect in Figure 11, presenting MFR for different optimizations on their target data structures. We observe that, as expected, Binarize results in significant memory savings, reaching close to 16× MFR (32× for ReLU output and at least 8× for pool output). Reduction for SSDC varies significantly across CNNs, providing upto 7× MFR for Overfeat. Finally, inplace optimization results in upto 1.4× MFR for AlexNet. Inplace optimization does not always reduce the total memory footprint, because the affected immediately consumed data structure might not be a heavy hitter in the memory groups formed by CNTK memory allocator (Section IV-C).

*2) Impact on Performance:* To evaluate the performance overhead of lossless techniques, we run the previous experiment and measure the performance overhead. We observe that Binarize, instead of showing performance degradation, results in small performance improvement. This is because ReLU backward pass calculations are memory bandwidth bound and Binarize encoding increases effective memory bandwidth by representing the data in 1-bit format. We observe that SSDC encoding results in small performance overhead – upto 4% on average. The combination of two lossless encodings result in slightly better performance than just SSDC encoding alone, because of the better performing Binarize encoding. And, finally, inplace optimization has no effect on performance as it does not incur any encoding or decoding overhead.

### D. Lossy Encodings

In this section, we study the impact of lossy encodings on accuracy, memory footprint, and performance.

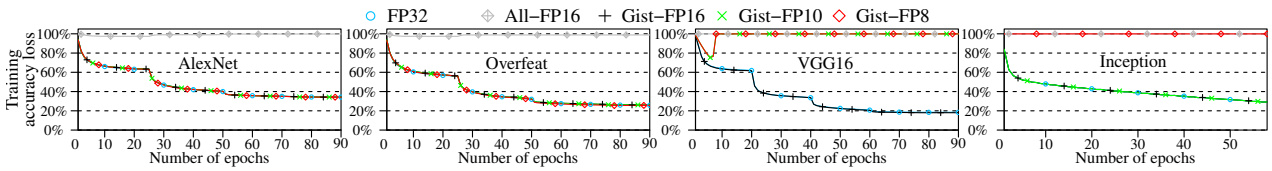



**Figure 12:** Impact of DPR Encoding on accuracy. The smallest format, having same accuracy compared to FP32 format, for AlexNet and Overfeat is FP8, for Inception is FP10 and for VGG16 is FP16; DPR achieves aggressive bit savings.

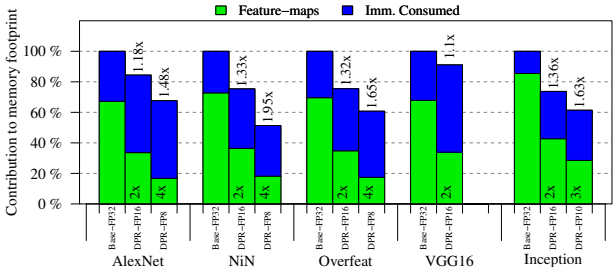



**Figure 13:** Impact of DPR encodings. Total MFR is present at the top and MFR achieved on the stashed feature maps is present at the bottom of each bar. Lowest precision for VGG16 with no loss in accuracy is FP16.

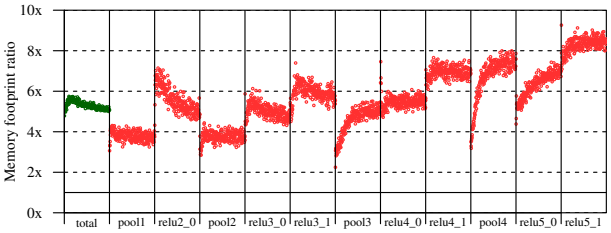



**Figure 14:** SSDC sensitivity to Sparsity (15 epochs, VGG16)

*1) Impact on Accuracy:* First, we study the effect of applying precision reduction on the training accuracy. In this experiment, we, first, train the network in FP32 precision (shown as *Baseline-FP32*). Second, in line with previous research [15], [8], [16], we represent all the data structures throughout the network in FP16 format and then train the network (shown as *All-FP16*). Finally, we train the network using FP16, FP10 and FP8 DPR encodings (shown as *Gist-FP**). The computation is still performed with FP32 precision. The values are converted back to FP32 format just before the computation.

The findings of this experiment are shown in Figure 12, showing the training accuracy loss for different networks. Training accuracy loss curve is a method of capturing training accuracy over time. At the start, the network has nearly 100% accuracy loss (almost 0% accuracy). Over time, the accuracy improves and correspondingly the accuracy loss reduces. For example, at 90th epoch, Overfeat achieves 22% accuracy loss i.e. Overfeats accuracy is 78% (100% - 22%). The figure compares the deviation between the accuracy of CNTK baseline (FP32 blue circles) and Gist-FP16/10/8 DPR encodings. Note that the y-axis is not the accuracy loss from Gist encodings. Instead, it measures the networks achieved accuracy as the training epoch increases. Therefore, for DPR encoding to be as accurate as the FP32 baseline, the curves should overlap.

There are two key observations from this figure. First, representing all data structures in 16 bits leads to severe accuracy losses. This is because the precision reduction is applied immediately after each layer output is computed, propagating the error in the forward pass and resulting in severe accuracy losses. Second, applying precision reduction only for the backward use of stashed feature maps, as done in DPR, can result in aggressive bit savings (as low as 8 bits for AlexNet and Overfeat). For Inception, we observe that when FP8 is applied, the network stops training, but FP10 has enough precision to result in no accuracy losses. VGG16 needs the highest precision, and does not train with representation smaller than FP16, showing that the minimum acceptable precision is network dependent.

*2) Impact on Footprint Reduction:* In this section, we evaluate how much MFR these efficient representations achieve. The experiment involves running DPR FP16 and the next smallest representation (FP10/FP8) that has no accuracy losses, and measuring the total memory footprint. The findings of this experiment are presented in Figure 13, showing the MFR against investigation baseline.

DPR encoding converts the stashed feature maps into immediately consumed data and stashes the encoded feature map for the backward pass. To see this effect, we break the total memory consumption into stashed feature maps and immediately consumed. When DPR encoding uses FP16, the stashed feature maps are compressed 2×, with some increase in immediately consumed footprint, resulting in the total MFR of 1.18× for AlexNet as an example. FP8 further cuts down the memory footprint, resulting in MFR of 4× for stashed feature maps and a total of 1.48× MFR for AlexNet. As shown previously, FP8 leads to bad accuracy for VGG16, and thus we omit results for FP8 for VGG16.

*3) Performance Overhead:* Next, we evaluate the performance overhead of DPR encoding. For this study, we run the last experiment again and measure the execution time of a minibatch processing, averaged over 5 minutes of training. We observe that extremely parallel DPR encoding implementation has minimal performance overhead, with an average of 1%.

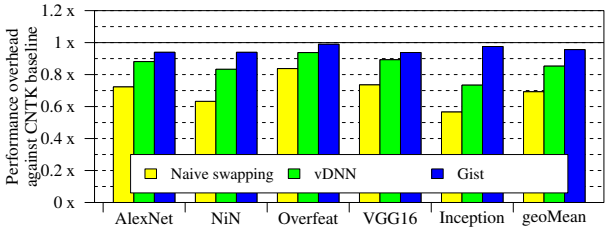



**Figure 15:** Performance comparison of *Gist* against naive swapping and vDNN

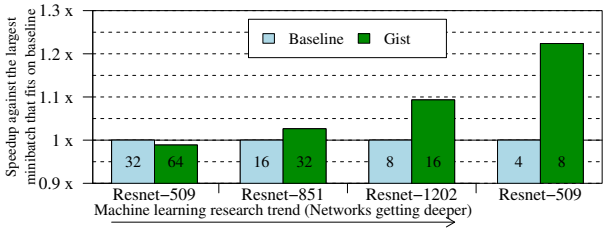



**Figure 16:** *Gist* enables training deeper networks with larger minibatch, achieving better performance. The largest minibatch that fits in the GPU memory is present at the bottom of the bar

### *E. Sensitivity Study*

Most of the *Gist* encodings result in fixed MFR, as they are agnostic to data values. However, SSDC encoding depends on the sparsity of data structure that changes during the training. In this section, we perform a sensitivity study for SSDC encoding on VGG16. The experiment involves applying the SSDC encoding, training the network for 15 epochs, while recording the achieved compression ratio for the applicable feature maps after every 1000th minibatch. The findings of this experiment are presented in Figure 14.

This figure shows the MFR achieved for each applicable layer (each block is a single layer) over time. We observe significant MFR across all layers, varying across the layers, and also across time within a single layer. The MFR is typically much larger than 1, except only for a small duration of first few minibatches (close to 200) of the first epoch (one epoch for VGG16 has 20K minibatches). This happens because at the start of the training, the network weights are initialized randomly. It takes few minibatches for weights to change and for sparsity to come into effect.

### *F. Comparison with Prior Work*

Another way to reduce the memory footprint is to *swap* the parts of working set between the CPU and GPU memory using PCIe links. In this section, we compare Gist's performance with such approaches. vDNN, the most relevant prior work to Gist, is built upon this approach and uses a prefetching analysis to find a suitable overlap between the data transfer and computation time [39]. We implement vDNN in CNTK and present the comparison in Figure 15, showing the performance overhead of naive swapping (no prefetching), vDNN and *Gist* against CNTK baseline.

We observe that naive swapping results in a heavy performance loss, averaging 30%, because there is no overlap between kernel execution and data transfer. vDNN improves this performance by performing prefetching analysis. However, vDNN still has high overhead as the data transfer time cannot be completely hidden, resulting in an average slowdown of 15%, with a maximum of 27% for Inception. By keeping the data within GPU, *Gist* is not limited by the PCIe bandwidth and observes an average slowdown of only 4% (upto 7%).

### *G. Impact on Machine Learning Trend*

A look at past ImageNet challenge winners show that networks are getting deeper over time [42]. Thus, training them incurs a higher memory footprint potentially exceeding the limited size of GPU DRAM (16 GB for the most expensive card). This means that to train deeper networks, one has to use smaller minibatch sizes that fit in GPU memory, resulting in underutilized hardware and high training time. Next, in the context of training deep networks with small minibatch sizes, we show how *Gist* allows for faster training by enabling the use of larger minibatch sizes.

We use Resnet [22], the winner of 2015 ImageNet challenge, for this study. Resnet presents a highly composable structure, enabling us to vary the depth of the network and project this trend of deeper networks. The original Resnet paper evaluates the accuracy to the maximum depth of 1202 layers [22]. In line with the paper, we vary the number of layers to 509, 851 and 1202. We present the findings of this experiment in Figure 16, showing the speedup achieved by the largest minibatch that fits with *Gist* compared to the largest minibatch that fits with CNTK Baseline. We observe that by enabling larger minibatches, *Gist* increases GPU utilization and improves training time, e.g., a speedup of 22% for Resnet-1202. In general, due to better utilization of GPU resources with larger minibatch sizes, Gist's performance improvements positively correlate with the existing machine learning trend of deeper modern networks.

### *H. Discussion – Memory Allocation*

Deep learning frameworks typically perform static memory allocation on GPUs to avoid expensive *cudaMalloc* calls while processing a minibatch. However, there are ongoing efforts to accelerate training on FPGAs [50] and ASICs [28], [25]. In such scenarios, dynamic memory allocation can be a preferable choice if the memory allocation is itself implemented in hardware and results in minimal performance overhead. The question then arises, in the presence of such *optimized hardware*, how much footprint reduction does dynamic memory allocation achieve, and what is the impact of the *Gist* encodings when memory is allocated dynamically.

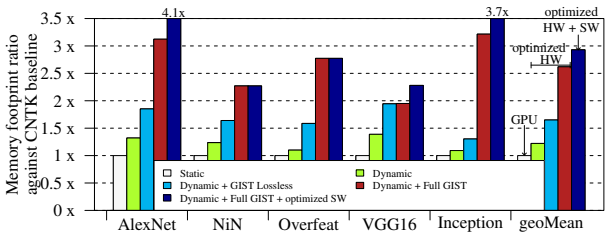



**Figure 17:** Impact of optimized hardware (dynamic allocation), *Gist* and optimized software (cuDNN)

Second, for DPR encodings, we decode the data back into FP32 format, because cuDNN library requires data to be in the FP32 format. This requires an extra memory block (though only for immediate use) for the decoded FP32 representation. We believe that most of this decoding can be pulled inside cuDNN implementation, which can either execute directly on the encoded data, or decode only the data that is required in near future, for example, the data required for the near-future tile in a tiled matrix multiplication. Such *optimized software* can remove the need for decoded data structure, potentially resulting in higher MFR.

In this section, we discuss the impact of such optimized hardware and software on MFR. For dynamic memory allocation, we modify the liveness analysis module from CNTK and simulate dynamic memory management, allocating a region only when it is required and relinquishing it as soon as it is dead. We find the peak memory consumed in this scheme and compare it against static memory allocation. Similarly, for *optimized software*, we modify the liveness analysis module to remove the decoded FP32 values and reallocate the memory. We present the findings in Figure 17.

The figure shows the achieved MFR for dynamic memory allocation, *Gist* encodings in presence of dynamic allocation, and optimized software with *Gist* encodings and dynamic allocation, against CNTK baseline. There are three key observations from this graph. First, dynamic memory allocation results in good MFR, going over 1.5× for Overfeat, with an average of 1.2× across all CNNs. Second, *Gist* encodings are still applicable in the presence of dynamic memory allocation. We observe that *Gist* lossless and lossy encodings achieve MFR of 1.7× and 2.6× respectively. Finally, optimized software can further cut down the memory footprint, resulting in a MFR of upto 4.1× for AlexNet against CNTK baseline, with an average of 2.9× across all CNNs.

## VI. Related Work

Our paper presents a systematic analysis of breakdown of total memory footprint across different data structures in DNN training, showing that stashed feature maps and immediately consumed data structures are the major contributors in modern DNN frameworks (as opposed to weights in DNN inference). We present layer-specific lossless encodings, targeting different categories of stashed feature maps, which to our knowledge has not been proposed before. *Gist* lossy encoding is specifically designed for DNN training and stands in stark contrast with the prior body of similar work on DNN inference. Our work presents a unique way of applying precision reduction in which the data in the forward pass is kept in full FP32 format, while only the data that is stashed for the backward pass is represented with fewer bits, resulting in more aggressive bit savings which is unseen in the previous work.

**Generic Approaches.** vDNN transfers the data between CPU and GPU memory using smart prefetching analysis [39], fitting large networks in the GPU memory, but at the expense of (1) performance cost (15% on average, and upto 27%), and (2) energy cost of using PCIe and GPU DRAM bus constantly for the data transfer, and (3) using PCIe, which is a shared critical resource in a distributed training [9], potentially causing performance issues in distributed setting. CDMA, designed on top of vDNN, leverages sparsity to compress the data sent between CPU and GPU [40]. We found that CNTK memory allocator already implements this memory sharing (our CNKT baseline already has these optimizations). [4], [14] presents details of MxNet's memory sharing and inplace optimizations. It also proposes layer re-computation to trade off large memory space with re-computing fast DNN layers. This work is orthogonal and can achieve additional speedup with *Gist* encodings.

**Encodings.** Lossy encodings have been studied rigorously in the domain of DNN inference. These works apply network pruning, quantization, Huffman encoding and precision reduction to reduce the model size (*weights*) [18], [30], [19], [20], [27]. Many HW accelerators have been designed employing limited precision and leveraging sparsity to reduce computational and memory requirements [17], [5], [6], [35], [29], [38], [47], [2]. However, these techniques do not apply directly for training as weights change frequently during the training process, and weights are not a major contributor to total memory footprint.

Most of the other works for DNN training have looked into the reducing precision requirements for *computation*. These works do not focus on reducing memory footprint and, thus, do not optimize memory for stashed feature maps. For example, BUCKWILD! breaks down memory footprint into four categories (DMGC in the paper), but ignore stashed feature maps, as it does not play significant role in computational precision study [11]. Similarly, [15], [8], [16] show that 16-bits dynamic fixed point computation is enough for training small DNNs on CIFAR-10 and do not focus on primary contributor to memory footprint. We share an observation with this work that uniform precision reduction results in severe accuracy losses. These works keep a shadow copy of weights in full precision, which is updated at the end of each minibatch and then quantized for next minibatch, to keep accuracy in check.

## VII. Conclusion

In this paper, we investigate approaches to reduce the memory footprint of DNN training, enabling training of deeper DNNs on GPUs. We present, *Gist*, that employs two layer-specific lossless and one aggressive lossy encoding schemes, targeting the primary contributor to total memory footprint (feature maps). A common approach in our encodings is to store an encoded representation of feature maps and decode this data in the backward pass; the full-fidelity feature maps are used in the forward pass and relinquished immediately. *Gist* reduces the memory footprint by 2× across 5 state-of-the-art image classification DNNs, with an average of 1.8× with only 4% performance overhead and no effect on training accuracy.

## Acknowledgement

We thank our anonymous reviewers for their feedback and suggestions. This work was done in the context of Project Fiddle at Microsoft Research (MSR). We thank MSR's Special Internship Project Program for sponsoring this effort. Animesh Jain was also partially supported by the National Science Foundation (NSF) under grants IISVEC1539011 and CAREER SHF-1553485.

## References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Operating Systems Design and Implementation (OSDI)*, 2016.

[2] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[3] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *International Conference on Computational Statistics (COMPSTAT)*, 2010.

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. In *arxiv*, 2016.

[5] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. In *arxiv*, 2014.

[8] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. In *arxiv*, 2014.

[9] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. GeePS: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *European Conference on Computer Systems (EuroSys)*, 2016.

[10] Y. Le Cun, B. Boser, J. S. Denker, R. E. Howard, W. Habbard, L. D. Jackel, and D. Henderson. Handwritten digit recognition with a back-propagation network. In *Neural Information Processing Systems (NIPS)*, 1990.

[11] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[12] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. In *arxiv*, 2017.

[13] Edward Grefenstette, Phil Blunsom, Nando de Freitas, and Karl Moritz Hermann. A deep architecture for semantic parsing. In *arxiv*, 2014.

[14] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *arxiv*, 2016.

[15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on International Conference on Machine Learning (ICML)*, 2015.

[16] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. In *arxiv*, 2016.

[17] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[18] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding. In *arxiv*, 2015.

[19] Stephen José Hanson and Lorien Pratt. Comparing biases for minimal network construction with back-propagation. In *Neural Information Processing Systems (NIPS)*, 1989.

[20] Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *in Neural Information Processing Systems (NIPS)*, 1993.

[21] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: DNN as a service and its implications for future warehouse scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *arxiv*, 2015.

[23] Parker Hill, Animesh Jain, Mason Hill, Babak Zamirai, Michael Laurenzano, Chang-Hong Hsu, Scott Mahlke, Lingjia Tang, and Jason Mars. Addressing compute and memory bottlenecks for dnn execution on GPUs. In *International Symposium on Microarchitecture (MICRO)*, 2017.

[24] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. In *IEEE Signal Processing Magazine*, 2012.

[25] https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine learning/. Build and train machine learning models on our new google cloud TPUs, 2017.

[26] https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan x/specifications. Nvidia GeForce GTX Titan X, 2017.

[27] A. Jain, P. Hill, S. C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[28] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[29] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[30] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *International Conference on Supercomputing (SC)*, 2016.

[31] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR)*, 2014.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*, 2012.

[33] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. Adaptive-latency DRAM: Optimizing DRAM timing for the common-case. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[34] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In *arxiv*, 2013.

[35] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[36] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[37] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q. Weinberger. Memory-efficient implementation of densenets. In *arxiv*, 2017.

[38] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[39] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[40] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, and Stephen W. Keckler. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *arxiv*, 2017.

[41] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. 1988.

[42] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. In *arxiv*, 2014.

[43] Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[44] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. In *arxiv*, 2013.

[45] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *arxiv*, 2014.

[46] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[47] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[48] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *arxiv*, 2014.

[49] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[50] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *International Conference on Computer-Aided Design (ICCAD)*, 2016.

[51] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Tbd: Benchmarking and analyzing deep neural network training. In *arxiv*, 2018.