Split-CNN: Splitting Window-based Operations in Convolutional Neural Networks for Memory System Optimization

Tian Jin
IBM Thomas J. Watson Research Center
Yorktown Heights, New York, USA
tian.jin1@ibm.com

Seokin Hong*
Kyungpook National University
Daegu, South Korea
seokin@knu.ac.kr

Abstract

We present an interdisciplinary study to tackle the memory bottleneck of training deep convolutional neural networks (CNN). Firstly, we introduce Split Convolutional Neural Network (Split-CNN) that is derived from the automatic transformation of the state-of-the-art CNN models. The main distinction between Split-CNN and regular CNN is that Split-CNN splits the input images into small patches and operates on these patches independently before entering later stages of the CNN model. Secondly, we propose a novel heterogeneous memory management system (HMMS) to utilize the memory-friendly properties of Split-CNN. Through experiments, we demonstrate that Split-CNN achieves significantly higher training scalability by dramatically reducing the memory requirements of training algorithms on GPU accelerators. Furthermore, we provide empirical evidence that splitting at randomly chosen boundaries can even result in accuracy gains over baseline CNN due to its regularization effect.

ACM Reference format:
Tian Jin and Seokin Hong. 2019. Split-CNN: Splitting Window-based Operations in Convolutional Neural Networks for Memory System Optimization. In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems, Providence, RI, USA, April 13–17, 2019 (ASPLOS'19),* 13 pages.
https://doi.org/10.1145/3297858.3304038

*This work was performed when Seokin Hong was affiliated with IBM Thomas J. Watson Research Center.

1 Introduction

In recent years, a wealth of deep convolutional neural networks have been proposed to achieve superior results in tasks such as image classification, segmentation and captioning, among which there are AlexNet [28], VGG [36], Overfeat[35], and ResNet [20], just to name a few. While these networks become more accurate, their memory requirements are growing increasingly.

Unfortunately, today's GPUs have considerably small device memory. The memory capacity of most NVIDIA GPUs is less than or equal to 16 GB. Even the latest GPU (e.g., NVIDIA GPU Tesla V100) has only 32 GB of memory capacity. Thus, even with the most recent GPUs, popular machine learning frameworks will nevertheless require a multi-GPU system to train state of the art models with a large batch size, which significantly increases the cost and complexity of the deep neural networks training systems. Since the trend of deep learning is toward deeper and larger networks, the bottleneck on the GPU's memory capacity will only become worse and must be overcome. Specifically, there are **three challenging trends** contributing to the memory bottleneck of training deep neural networks: prevalence of memory-bound layers, increasing batch sizes, and increasing model complexity. We will elaborate on the cause and impact of these trends on DNN training systems in the coming section.

To tackle the three challenges, we make three key contributions in this paper as follows:

- Firstly, we propose Split-CNN, a generic instrumentation of the regular CNN models, that breaks apart memory bottlenecks of the CNN models while retaining or even enhancing the application-specific metrics of the model (i.e., classification accuracy). We evaluated Split-CNN and quantified its effect on application metrics across multiple model/dataset combinations.
- Secondly, we introduce a new heterogeneous memory management system (HMMS) to optimally schedule memory allocation, deallocation, offloading and prefetching without any required tuning on the network models. We then show the evaluation results on the efficacy of its offloading/prefetch plan to demonstrate its capability of solving the challenge with the memory-bound layers.

- Thirdly, we combine Split-CNN and HMMS to showcase that Split-CNN can be trained with significantly larger batch size, thus effectively solving the challenges with the increasing demands on large batch size.

Our experiments show that Split-CNN significantly increases the scalability of the training algorithm and our HMMS can schedule memory allocations and host-device data movements optimally with no tuning required. By combining these two techniques, we can train VGG-19 models with 6 times larger batch size and ResNet-18 models with 2 times larger batch size than baseline method. Furthermore, since Split-CNN enables model training with large batch sizes, it can improve the performance of the distributed training by reducing the network traffics caused by parameter updates. Based on our analysis, Split-CNN achieves 2.1x speedup in the distributed training for VGG-19 model with typical cloud network bandwidth.

2 Background and Motivation

2.1 Training a Deep Neural Network

To train a deep neural network made of many layers of mathematical functions, one must use back-propagation algorithm [10, 34]. This is a two-stage process. In the first stage, a *forward pass* is executed to compute the set of outputs Y (e.g. predicted labels) of a model F for a given *mini-batch* of inputs (e.g., images). Then a loss function L is computed based on some notion of distance between the output label predictions Y and the ground truth. In the second stage, the loss is propagated backwards to each weight w in the form of $\frac{\partial L}{\partial w}$ by applying chain rules to each layer. The gradient of the loss function with respect to each parameter is then subtracted from the current value of the parameter to perform optimization. Crucially, in order to compute such gradients, intermediate values computed in the forward pass are often required again in the backward pass. This is the primary reason behind the high memory capacity requirement of training deep neural networks.

2.2 Challenges to the Memory Capacity Constraint

In this section, we elaborate on the **three challenges** emerging from the deep learning community that will pose challenges to the trainability of CNN models given the hardware memory capacity constraint.

2.2.1 More Memory Bound Layers

Recent advances in the area of deep learning have driven the layer computation of deep convolutional neural networks away from computation bound and more towards memory bound. One such advancement is batch normalization layer [26] whose computation mainly involves calculating aggregate statistics of intermediate layer results. Today, such layer has been used extensively in recently proposed neural networks like [20, 23]. Another such improvement is the fast convolution algorithm [29] that trades memory space for faster computation using Winograd algorithm. This algorithm has been integrated into the NVIDIA cuDNN library, the de facto standard library for deep learning computations on NVIDIA GPUs.

Since intermediate results are often kept alive throughout the forward propagation pass, having more memory bound layers implies that more intermediate results need to be kept in memory even though the total execution time of the layer is shorter. As a result, both advances and their quick and wide adoption across the deep learning community has made traditional techniques like memory offloading [32], and CUDA managed unified memory the victim of heavy performance degradation. This is because both techniques involve paging device data in and out of device memory, and both techniques fail to have sufficient time to make costly data transfer as a result of the low computation to memory consumption ratio of the current neural network models.

2.2.2 Larger Batch Size

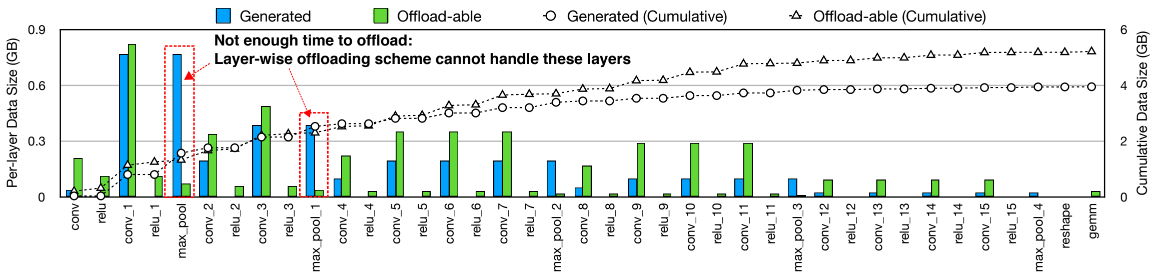
Recent studies such as [3, 11, 37] emphasizes the importance of using very large batch sizes during training of deep neural networks for at least a significant part of training time to improve the quality of the gradients and thus accelerate convergence. Notably, [37] trained a ResNet-50 network on a TPU pod with an initial batch size of 8192, and then increased the batch size to 16384 soon after. It is often very inefficient, if not downright impossible, to train deep neural networks with large batch sizes, using only a small number of GPUs. For instance, previous work by [32] enabled training of VGG-16 network with batch size equal to mere 256 on a single Titan X GPU with significant (18%) training throughput degradation.

2.2.3 Higher Model Complexity

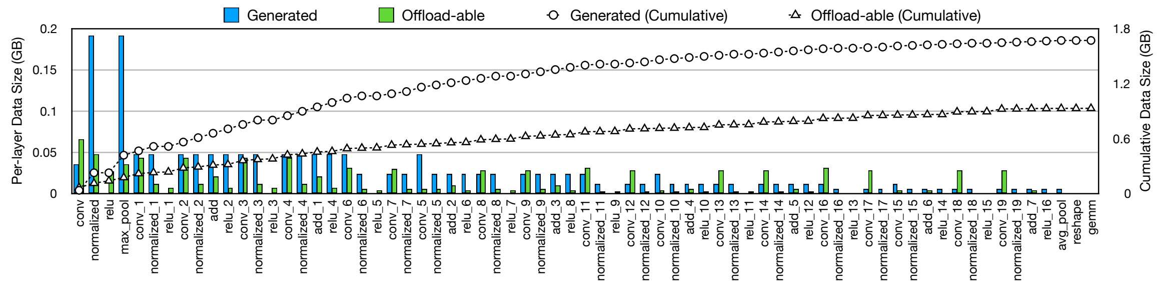
The deep learning community has gone a long way since the proposal of the first deep CNN named AlexNet [28] and the models are only growing ever more complex. For instance, Mask R-CNN [21] involves jointly training neural network models to perform multiple tasks including image classification, object detection, and object mask generation. Triplet network [22] trains neural networks using batches of triplets of image instances, effectively training three neural networks at the same time for deep metric learning. The growing complexities of model designs translate directly into increased demand for higher accelerator memory capacity.

2.3 Limitation of Layer-wise Memory Allocation

To address the increasing memory requirements, many techniques have been proposed. One of the most relevant state-of-the-art technique, vDNN, employs a layer-wise memory allocation scheme [32] to train deep convolutional neural networks with significantly less memory requirements and



(a) VGG-19 with batch-size=64



(b) ResNet-18 with batch-size=64

Figure 1. The amount of generated data vs. the amount of data that can be offloaded without performance loss

some performance degradation. vDNN offloads intermediate results right after they are computed, freeing them immediately after consumption by ensuing layer(s). The legality of such aggressive memory deallocation is enforced by synchronization between compute stream and memory stream after the execution of the consumer layer. However, this technique is restricted in many ways; it requires a complex and multi-stage tuning process. Despite the complexity of such procedure, it will nonetheless incur noticeable performance degradation when training networks with large batch sizes. Moreover, the throughput and trainability of a neural network are nevertheless bottlenecked by the layer who produces the largest intermediate results.

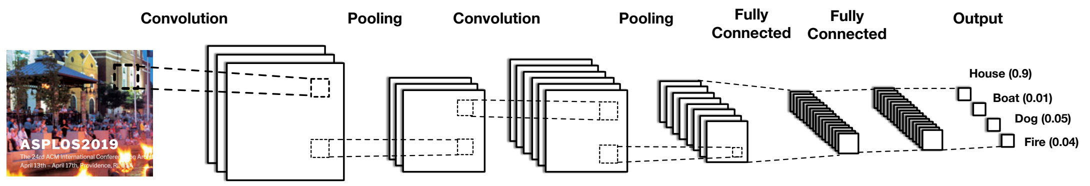
2.4 Opportunities and Challenges with NVLink

The advent of NVLink-enabled DNN training systems like IBM Power System S822LC with 4 NVIDIA Tesla P100 GPUs and IBM Power System AC9228 with 4 NVIDIA Tesla V100 GPUs significantly help with the memory capacity constraints by enabling faster memory offloading. However, NVLink alone is not the full answer.

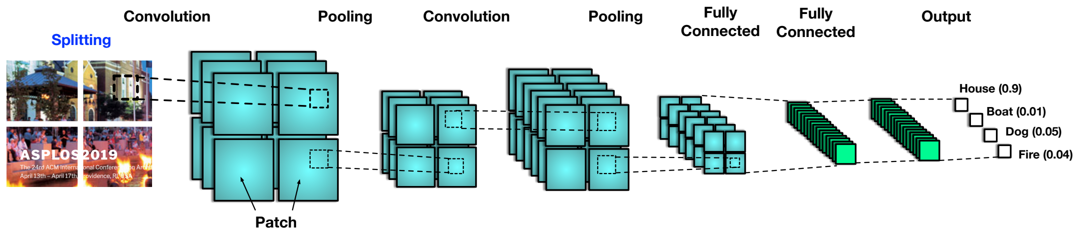
Figure 1 shows the memory usage of the forward propagation training pass of the two most commonly used neural networks, VGG-19 [36] and ResNet-18 [20]. To obtain the figure, we profiled the forward training pass of these neural networks and calculated two values for each layer operation: **generated data size** and **offload-able data size**. Generated data size is the size of intermediate results generated by this layer (and therefore can be offloaded during the execution of the next layer) that will need to be consumed again in the backward propagation pass. Offload-able data size is the size of data we can offload during the execution of this operation. This is obtained by multiplying the measured NVLink peak bandwidth with the profiled execution time of this layer.

Their cumulative values are plotted on the graph as well. Several observations are made: firstly, intermediate results for VGG-19 can potentially be **completely offloaded** without performance penalty because cumulative offload-able data size *eventually* exceeds cumulative generated data size, whereas for ResNet-18, only 55% of intermediate results can be offloaded. Secondly, in both networks, not all layers have enough time to offload its input intermediate results. Most notably, memory bound layers like pooling layers and batch normalization layers almost never have enough time to offload. In the context of two networks profiled, this means that to offload all of VGG's intermediate results with the previous technique of layer-wise offloading will incur the heavy performance penalty as confirmed by our experimental results in later sections. This also means that partial layer-wise offloading of ResNet will be ineffective and inefficient because layers producing the largest intermediate results often have the smallest budget to offload memory.

Our observations call for two improvements over the prior art of layer-wise memory offloading technique. The first improvement is the ability to spread memory offloading and prefetching across multiple layers. This allows for more time



(a) Conventional Convolutional Neural Network



(b) Split Convolutional Neural Network with Four Patches

Figure 2. Conventional and Split Convolutional Neural Network

to offload data before synchronizing memory stream with compute stream, effectively preventing the blocking of computation by memory transfer. We propose a new heterogeneous memory management system (HMMS) with an intelligent offloading/prefetch planning algorithm to achieve this goal. However, spreading memory offloading and prefetching runs the risk of aggravating the effect of memory bottleneck as seen at the beginning of ResNet in Figure 1 (b). This can be solved by breaking down the memory bottleneck into smaller pieces and spreading them across the forward propagation pass such that they are far away from each other. We propose Split-CNN, a generic instrumentation of CNNs to achieve this goal as the second improvement.

3 Split Convolutional Neural Network

The problem of physical memory limitations did not exist solely within the confines of the machine learning community. In fact, the problem of fitting application data into device cache has been well-studied. Many have appealed to loop tiling [4, 5, 16] to keep active data within a small cache. Ideally, one could apply loop tiling to layer computations, and by executing computations tile by tile, memory saving could be achieved. However, traditional loop transformations are semantic-preserving, making it difficult to apply on the whole network mainly because the prevalence of window-based operations such as pooling and convolutional layers makes data-dependencies expensive to satisfy across layers.

In light of the limitation of existing solutions and the predicament of applying traditional loop transformations, we explored an alternative and intrusive memory optimization technique, called Split-CNN (Split Convolutional Neural Network), in similar nature. Split-CNN splits the computation of window-based operations into multiple small pieces; such splitting does not respect the data-dependencies across layers, thus changing the semantics of the neural network.

In the following subsection, we will discuss the steps to instrument any window-based operation layer so that the operation can be performed in a split way. For simplicity, we will be working with a 1-dimensional example.

3.1 Single Layer Formulation

Firstly we define the following concepts and notations:

- A *tensor* is a generalized matrix. Within our work, a tensor is equivalent to a multi-dimensional array.
- $Op\,(X, k, s, p)$ denotes a window-based operation. It is usually one of convolution or pooling operations. Such operations are characterized by its window of operation k, stride s and padding p. Padding p consists of 2 elements p_b, p_e, which, respectively, denotes the padding in the beginning and the end of the spatial dimension W. To avoid unnecessary complications in our formulation, we mandate that $k >= s$. This excludes certain down-sampling convolutions from our scope of consideration.
- $Split_D\,(T, (s_0, \cdots, s_{N-1}))$ denotes partitioning the tensor T along dimension D into N parts following the specification of a N-tuple $(s_0, \cdots, s_{N-1})$, where s_i denotes the index of the starting element of the ith part. We use index notation $Split_D\,(T, (s_0, \cdots, s_{N-1}))\,[i]$ to address the ith partition.
- $[T_0, \cdots, T_n]_D$ denotes the concatenation of multiple tensors along dimension D.

To split the operation of a single layer, one first has to decide a partition on the output of such a layer. This choice is arbitrary and a good choice for load balance is to partition the input as evenly as possible, we will denote such a split as $\boldsymbol{O} = (O_0, \cdots, O_{N-1})$ where O_i denotes the position of the starting element in the ith partition along the spatial dimension of the input tensor.

Then we proceed to find an input split $\boldsymbol{I} = (I_0, \cdots, I_{N-1})$ such that with proper paddings $\boldsymbol{p} = (p_0, \cdots, p_{N-1})$, $Op\,(Split_W\,(T, \boldsymbol{I})\,[i]\,, k, s, p_i)$ produces an output tensor of the desired spatial dimension size $O_{i+1} - O_i$. Formally, to ensure that all input feature maps are consumed by the split-operation [1], I_i must be within the closed interval $[lb\,(I_i)\,, ub\,(I_i)]$ where $lb\,(I_i)\,, ub\,(I_i)$ are calculated using the following two equations, which, respectively, corresponds to splitting right before the first element of the window that, after the application of the unsplit operation, produces the first element of the next output patch O_{i+1}, and right after the last element of a window that, after the application of the unsplit operation, produces the last element of the current output patch O_i. To aid interpretation, $lb\,(I_i) = ub\,(I_i)$, if the kernel shape equals the stride, in which case the splitting is natural and non-intrusive.

$$lb\,(I_i) = O_i s - p_b \tag{1}$$

$$ub\,(I_i) = (O_i - 1)\,s + k - p_b \tag{2}$$

Now we proceed to compute the proper padding p_i for each input patch X_i:

$$p_{i,b} = \begin{cases} p_b & i = 0 \\ I_i + p_b - (O_i - 1)\,s & otherwise \end{cases}$$

$$p_{i,e} = \begin{cases} p_e & i = N - 1 \\ (O_{i+1} - 1)\,s + k - (I_{i+1} + p_b) & otherwise \end{cases}$$

Thus arbitrary window-based operation $X = Op\,(X, k, s, p)$ can be formulated, with an arbitrarily chosen output partition scheme $\boldsymbol{O}$ and a suitably computed input scheme $\boldsymbol{I}$ based on Equation. 1 and Equation. 2, instead as:

$$\boldsymbol{I} = ComputeInputSplitScheme\,(k, s, p, \boldsymbol{O}) \tag{3}$$

$$X_0, \cdots, X_{N-1} = Split_W\,(X, \boldsymbol{I}) \tag{4}$$

$$\boldsymbol{p} = (p_0, \cdots, p_{N-1}) = ComputePadding\,(k, s, p, \boldsymbol{O}, \boldsymbol{I}) \tag{5}$$

$$\forall n \in \{0, \cdots, N-1\}\; Y_n = Op\,(X_n, k, s, p_i) \tag{6}$$

$$Y = [Y_0, \cdots, Y_{N-1}]_W \tag{7}$$

It is easy to generalize our technique to higher dimensions. In Figure 2, we provide an illustration of our technique for the 2-dimensional case with four intermediate patches.

[1]Hence, picking an I_i outside $[lb\,(I_i)\,, ub\,(I_i)]$ is also workable, but this may degrade the model performance (e.g., accuracy) because some features will be abandoned with our formulation by means of negative padding.

3.2 Multi-Layer Formulation

An interesting and desirable scenario arises when the output split scheme of the mth layer, $\boldsymbol{O}^m$ coincides with the input split scheme of its consumer layer (the $(m+1)$th layer), $\boldsymbol{I}^{m+1}$. In this case, patches of input of the mth layer can be operated upon independently through layers m and $(m+1)$ without being concatenated/split in between. In this case, splitting becomes a multi-layer construct with no communication between individual patches during forward and backward execution of intermediate layers, yielding flexibility of scheduling at our disposal. Due to the simple and intuitive formulation of Split-CNN, we refrain from the unnecessary formalization of the multi-layer splitting algorithm.

3.3 Stochastic Splitting

Drawing inspiration from DropOut [39], DropConnect [41] and Deep Networks with Stochastic Depth [24], we explore an avenue in similar nature. Specifically, in our experiments, we explore splitting intermediate feature maps at randomly chosen boundaries. Specifically, for each minibatch, a separate output split scheme $\boldsymbol{O} = (s_0, s_1, \cdots, s_{N-1})$ for each feature map spatial dimension of size L is computed and it satisfies the following equation when $i > 0$:

$$s_i \sim DiscreteUniform\left(\lceil \frac{(i-\omega)\cdot L}{N} \rceil, \lfloor \frac{(i+\omega)\cdot L}{N} \rfloor\right)$$

Where $\omega \in [0, 0.5)$ can be interpreted as the *wiggle room* allowed in each direction when splitting a tensor spatial-wise. The intuition behind stochastic splitting is to prevent the network architecture from utilizing the split structure of Split-CNN during training to obtain a set of trained weights that can perform well with the original unsplit network architecture upon testing/inferencing in production. The ability to use unsplit network for inference is a desirable trait as it removes the burden of adapting existing model inference/deployment infrastructure to work well with split network. In our experiments, we do not tune ω and use a constant ω of 0.2.

4 Heterogeneous Memory Management System

To intelligently schedule data offloading and prefetching on NVLink-enabled devices, and to utilize the memory-friendliness of Split-CNN, we propose the design of a heterogeneous memory management system (HMMS). Before describing HMMS, we define two key terms as follows:

- **Computation Graph** : a computation graph $G = (N, E)$ is a common way of specifying neural network computations. In this graph, each node $n \in N$ represents a mathematical operation (e.g., convolution); each node will have a set of inputs $X = \{x_0, x_1 \cdots x_N\}$

and a single output y. Each directed edge $e \in E$ represents a producer-consumer data flow; in the scope of our concern, all computation graphs are directed acyclic graphs.

- **Tensor Storage Object (TSO)**: Following established practices, it is often useful to separate the conceptual presence of a tensor in the computation graph from the physical appearance of the underlying data in the memory system. This allows our HMMS to perform some simple memory optimizations by allowing multiple tensors to occupy the same tensor storage space if specific conditions are met. Therefore, we introduce the concept of Tensor Storage Object (TSO) to represent a contiguous region of memory storage space used by one or more tensors to store its results. More details regarding these optimizations will appear in the ensuing sections.

In the following subsections, we describe HMMS's five-step method of planning memory usage for the computation graph as illustrated in Figure 3.

4.1 Splitting and Graph Generation

In the first step, we split the training model. With splitting depth d specified by the percentage of convolutional layers to break apart and a 2-tuple of integers (h, w) specifying the number of splits in each spatial dimensions (height and width), HMMS automatically transforms a regular convolutional neural network (CNN) to a Split-CNN. It is worth noting that, further evaluations of Split-CNN in later sections show that degradation of final model accuracy occurs slowly as one chooses more aggressive splitting depths and the number of splits. Therefore, searching for the minimal d, h, w can happen with very coarse granularity; furthermore, for the same reason, parameter values for d, h, w can be overestimated to simplify or eliminate the tuning process.

In the second step, we serialize the computation by topologically sorting compute nodes in the dataflow graph. We then generate the corresponding serialized back-propagation graph. Operations in the back-propagation graph are appended; the order such operations appear in the backward graph is the reverse of serialized forward order.

4.2 Storage Assignment and Optimization

In the third step, we assign and optimize tensor storage. We assign each tensor in the computation graph a tensor storage objects. As the assignment proceeds, we also keep a reference counter for each tensor storage object. We then perform two simple memory optimizations. 1) **In-place ReLU (Rectified Linear Unit)**: Rectified linear unit layers can be computed in place because the input of such layer is not needed in the backward error propagation for this layer. Therefore, if the reference counter indicates that no other tensor references the TSO of such layer's input tensor, we can safely replace

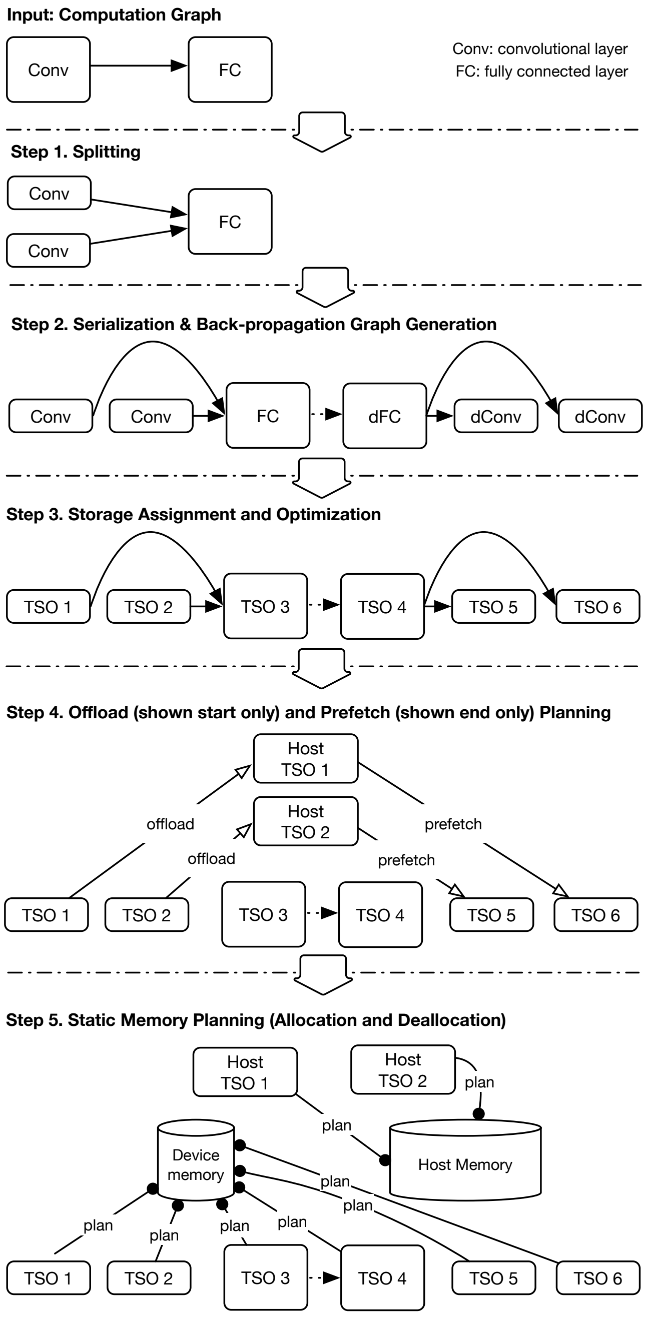


Figure 3. Heterogeneous Memory Management System (HMMS)

the input tensor's TSO with that of the output tensor. 2) **Summation Error Storage Object Sharing**: Summation is defined as $y = \Sigma_i x_i$. Thus $\frac{\partial y}{\partial x_i} = 1$ for all i. Thus when chain rule is applied to obtain the back-propagated error terms with the formulae $\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial x_i}$, all error terms have the same value. We, therefore, allow all error terms to occupy the same TSO. Further memory optimization can be easily added to optimize the mapping between tensors to TSO's.

4.3 Offload and Prefetch Planning

In the fourth step, we plan the offload and prefetch. We derive the optimal offloading and prefetching scheme to offload the most amount of memory without hurting the performance (at least theoretically). Importantly, memory transfer happens in parallel with computations because they are issued to dedicated memory streams to overlap computation and memory transfer.

For each tensor storage object being offloaded, four critical moments will occur during runtime:

1. *Start of the offload*: HMMS kicks off the device to host memory transfer through an idle memory stream *m* right after there is no more write to the TSO. Note that the host TSO is statically allocated offline.
2. *End of the offload*: HMMS will synchronize the computation stream with the memory stream *m* through which the TSO of interest is transferred. Such synchronization ensures that later computations can safely overwrite the memory address within this TSO. Again, the TSO is statically planned offline to be de-allocated immediately from the device memory pool right after the synchronization.
3. *Start of the prefetch*: HMMS will begin to retrieve the content of TSO from the host back to the device via a new idle memory stream m'. Again, a new TSO is statically planned offline and available to store the content retrieved onto the device.
4. *End of the prefetch*: HMMS will synchronize compute stream and the memory stream m' right before the TSO appear as the storage object for an input tensor for some operation in the backward propagation pass to ensure that prefetch is complete before its usage.

Since the *start of the offload* should happen as soon as a tensor is computed and the *end of the prefetch* should happen immediately before such a tensor is needed, we only need to plan for the *end of the offload* and the *start of the prefetch*. The planning process is inspired by the analysis of Figure 1. It consists of two stages: profiling stage and memory planning stage. In the profiling stage, we obtain the profiled execution time for each layer/operation. Execution time measurements are done by measuring, using *high_resolution_clock* in C++, the total execution time of 20 repeated executions and then dividing it by 20. In the memory planning stage, we decide when to end the offload and start the prefetch by calculating the size of the memory transfer allowed without slowing down the computation; we denote this calculated quantity as offload-able/prefetch-able memory capacity.

To plan the *end of offload*, we keep track of the offload-able memory capacity balance by recording any loss and gain in the offload-able memory capacity. A **loss** in offload-able memory capacity is incurred whenever we decide to offload a TSO; the value of the loss is equal to the size of the TSO. Whenever an operation is executed, a **gain** in offload-able memory capacity will occur, the value of which is determined by the product of profiled execution time and system-specific NVLink memory transfer bandwidth.

Algorithm 1: Offload Planning Algorithm

```
Data: serialized list of (fwd) operations in the CNN : ops
Initialize offload_capacity_balance = 0 ;
Initialize TSO_to_free = {} ;
Initialize profile_exec_time as described ;
Initialize nvlink_bandwidth as described ;
// Initialize memory and computation streams.
Initialize mem_stream[], comp_stream ;
for Operation op ∈ ops do
    input_TSO = TSO of input feature map of op ;
    if no further write happens to input_TSO then
        Get an idle memory stream m.
        Plan to allocate host TSO for input_TSO
          immediately before op starts executing. ;
        Plan to transfer input_TSO to host via m
          immediately after op starts executing. ;
        input_TSO.stream = m. ;
        Append input_TSO to TSO_to_free. ;
        offload_capacity_balance -= input_TSO.size ;
    end
    // Compute the increase of offloading capacity by
      multiplying op execution time with nvlink
      bandwidth.
    increase = profile_exec_time[op] *
      nvlink_bandwidth;
    offload_capacity_balance += increase ;
    if offload_capacity_balance ≥ 0 or op is the last then
        for TSO tso ∈ TSO_to_free do
            Plan to synchronize with tso.stream
              immediately after op starts executing. ;
            Plan to free tso immediately after the above
              synchronization. ;
        end
        if TSO_to_free is not empty then
            Plan to synchronize with comp_stream after
              above synchronizations with memory
              stream. ;
            offload_capacity_balance = 0 ;
            Clear TSO_to_free.
        end
    end
end
```

With the information of the memory capacity balance, we synchronize the compute stream with memory stream (at the *end of offload*) only if the balance is positive, implying that the computation will not be slowed down by synchronizing with memory streams. This is because, by construction, when such balance is positive, there will be no outstanding memory transfer. The algorithm is described in detail in Algorithm 1. We intentionally omit the simple algorithmic logic to keep the ratio of offloaded and non-offloaded TSO's under the theoretical limit to avoid distraction.

Planning the start of prefetch is similar to planning the end of offload, but the analysis proceeds in the opposite direction from the last operation in the backward propagation graph to the first one. Gains and losses to the prefetch-able memory capacity are similarly recorded and the start of the prefetch is planned whenever such balance turns positive.

4.4 Static Memory Planing

In the fifth step, we statically plan memory allocation and deallocation. In this stage, three memory pools are created to accommodate memory storage space requirements of the computation graph and its corresponding offloading or prefetching plan. All memory pools are contiguous.

1. Host general purpose memory pool is a chunk of pinned memory allocated on the host machine to store the intermediate results offloaded from the GPU.
2. Device parameter memory pool is allocated on the device to store the network parameters and their corresponding gradients. The primary motivation of having a dedicated parameter memory pool is to reduce memory fragmentation as parameter tensors tend to be very small.
3. Device general purpose memory pool is allocated on the device to store intermediate results for the forward and backward propagation of the neural network. It is also used to provide workspace for various cuDNN convolution algorithms.

The static memory planning algorithm steps through the list of operations in the serialized computation graph to allocate memory space to each tensor storage object (TSO). It uses first-fit memory allocation strategy where the first contiguous chunk of memory that the TSO object can fit in is allocated to the TSO object. Memory is allocated to TSO only for the minimum duration required mandated both by the reference counter and the previously computed offloading and prefetching scheme. Since memory planning happens entirely offline, there will be no runtime overhead for such aggressive static memory management policy.

5 Split-CNN Evaluation

To quantify the effect of Split-CNN on application-specific performance metrics (e.g., accuracy), we evaluated the effect of various hyperparameters on the test accuracy and convergence schedule of the convolutional neural network training. We also evaluated Split-CNN on CIFAR-10 [27] and ImageNet [12] datasets using AlexNet, VGG-19, ResNet-18, ResNet-50 architectures. We kept global batch sizes [2] of all experiments the same at 256 for ImageNet and 128 for CIFAR.

5.1 Datasets

In our experiments, we evaluated our proposed technique on two datasets: CIFAR-10 and ImageNet. CIFAR-10 is a 10-class dataset containing 60,000 colored images of size 32x32. Within CIFAR-10, there are 50,000 training samples and 10,000 testing samples. Similarly, ImageNet is a 1000-class dataset consisting of 1.2 million training samples and 50,000 validation samples. All the error rates reported are classification errors on the test or validation samples.

5.2 Impact of Hyperparameters on Accuracy

In Split-CNN, there are three tunable hyperparameters: the depth of the split, the number of splits and the extent to which stochastic splitting is allowed to deviate from splitting in the center. Splitting depth corresponds to the number of convolutional layers that are split from the input of the neural network to the layer where spatial patches are joined. The number of splits refers to the number of intermediate spatial patches used in a Split-CNN. In our experiments, the splitting depth is characterized by the percentage of convolutional layers that are split.

5.2.1 Splitting Depth

Firstly, we implemented Split-CNN versions of VGG-19 and ResNet-18 adapted for the CIFAR dataset. In each case, we split approximately [3] 0%, 12.5%, 25.0%, 37.5% and 50% of the convolutional layers into four equally sized spatial patches before joining. We then train each of the neural networks for 350 epochs with a starting learning rate of 0.1 and decrease the learning rate by a factor of 10 at epoch 150 and 250. It can be observed from Figure 4 that test accuracy degrades approximately linearly with increasing depth of the split.

5.2.2 Number of Splits

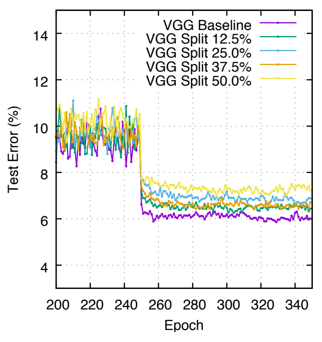
Secondly, using Split-CNN versions of VGG-19 and ResNet-18 on CIFAR dataset, we split about 25% the convolutional layers into 1, 2, 3, 4, 6 and 9 spatial patches. As can be observed from Figure 5, the accuracy degrades slowly with the increasing number of splits. ResNet-18 is less sensitive than VGG-19 to the increased breakage of spatial communication.

5.2.3 Stochasticity of Splitting

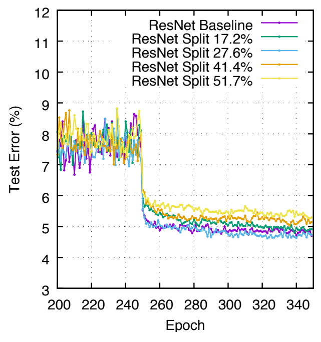
Thirdly, we evaluated the stochastic splitting as a way to improve final model accuracy of Split-CNNs. Using the Split-CNN versions of VGG-19 with 50% of the convolutional layers split into four patches and ResNet-18 with 51.7% of the convolutional layers split into four patches, we trained

[2] sum of local batch sizes across 4 GPUs within one machine

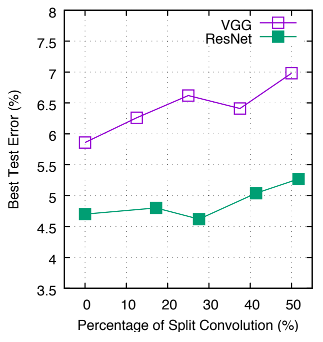
[3] The percentages are approximate because we only join at the residual block boundaries in ResNet.



(a) Split-VGG-19 w/ Various Depths

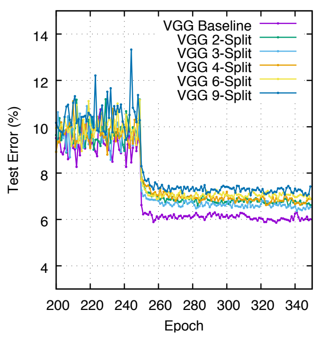


(b) Split-ResNet-18 w/ Various Depths

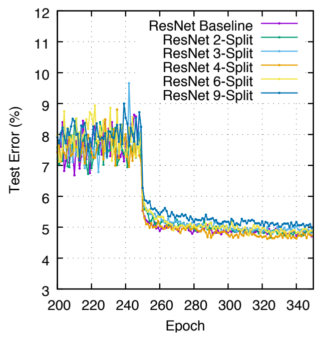


(c) Best Test Error vs. % Split

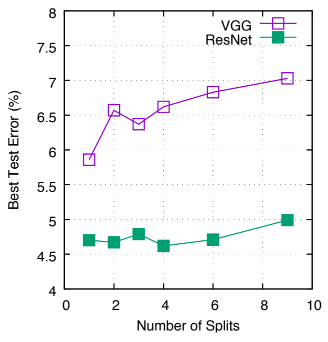
Figure 4. Effects of Splitting Depth on Test Error (lower is better)



(a) Split-VGG-19 w/ Various No. Splits

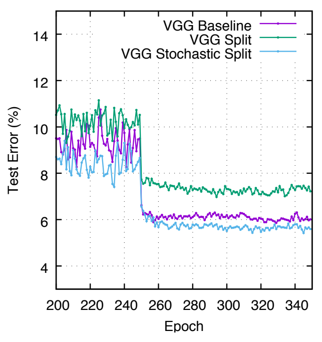


(b) Split-ResNet-18 w/ Various No. Splits

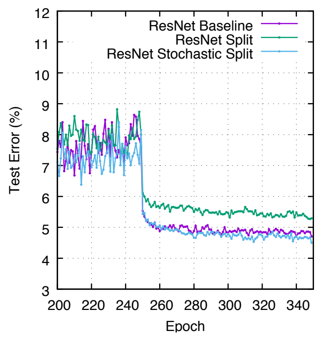


(c) Best Test Error vs. % Split

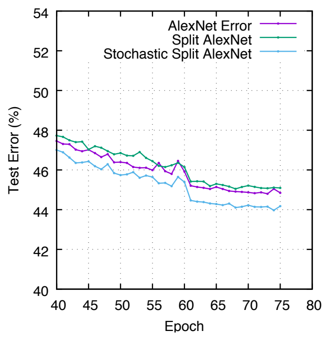
Figure 5. Effects of Number of Splits on Test Error (lower is better)



(a) VGG-19



(b) ResNet-18



(c) AlexNet

Figure 6. Effects of Stochasticity of Splitting on Test Error (lower is better)

models using Stochastic Split-CNN and evaluated the classification performance of Stochastic Split-CNN using the unsplit network. As can be observed from Figure 6, Stochastic Split-CNN is very competitive with the baseline (i.e., unsplit network) and even outperforms the baseline in many cases.

5.3 Convergence Speed

To ensure that Split-CNN architecture can be generalized to datasets of heavier workloads, we performed additional experiments using AlexNet and ResNet-50 on the challenging ImageNet dataset in addition to our extensive testing on CIFAR dataset. In these experiments, we also observed the convergence speed of Split-CNN over a longer training duration. Following established practice documented in PyTorch, we adopted an initial learning rate of 0.01 for AlexNet and 0.1 for ResNet, following a weight decay of 1e-4, a momentum of 0.9 and a global mini-batch size of 256 using 4 GPUs. The learning rate is decayed by a factor of 10 every 30

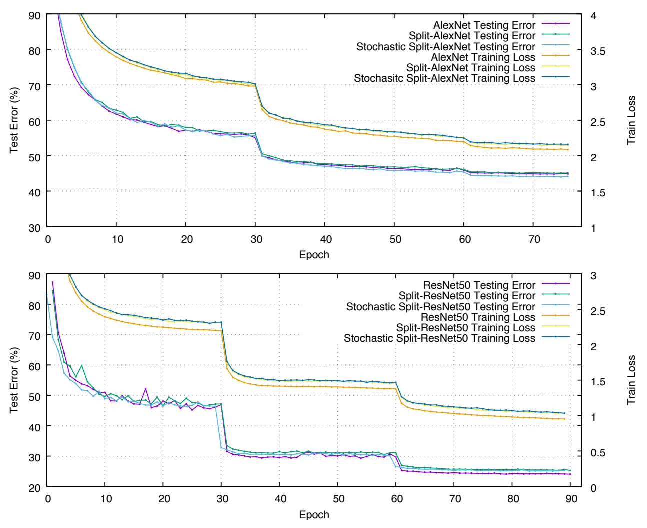


Figure 7. Split-CNN Classification Performance on ImageNet

epochs. The results and related Split-CNN hyperparameters are shown in Table 1. Validation errors throughout the entire training process are plotted in Figure 7. As can be observed, even with very aggressive splitting configurations (4 spatial patches and 81.2% of the convolutional layers split in the case of ResNet-50), the percentage of accuracy degradation is within 2% in all cases; even for accuracy-minded audience, stochastic splitting can be used to close the gap between Split-CNN and regular CNN further.

6 Evaluation of Heterogeneous Memory Management System (HMMS)

6.1 Experimental Setup

Our experiments are conducted on IBM Power System S822LC which is a widely used server-grade machine in the area of High-Performance Computing. This system uses NVIDIA Tesla P100 GPUs with a memory capacity of 16 GB. We used IBM Power8 CPU as our host processor. We restrict processors to only utilize its local memory node when allocating pinned memory. The GPU is connected to the host machine via an NVLink 1.0 interconnect with a peak measured bandwidth of 34.1 GB/sec.

Classification Accuracy of Split-CNN				
Architecture	AlexNet	ResNet50	VGG19	ResNet18
Dataset	ImageNet	ImageNet	CIFAR	CIFAR
Splitting Depth	60%	81.2%	50 %	50 %
No. of Splits	4	4	4	4
Baseline Acc.	55.2 %	**75.9 %**	94.14 %	95.3 %
SCNN Acc.	55.0 %	74.7 %	93.02 %	94.8 %
SSCNN Acc.	**55.9 %**	74.9 %	**94.58 %**	**95.5 %**

Table 1. Classification Performance of Split-CNN

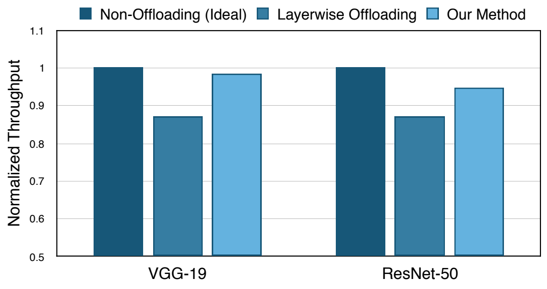


Figure 8. Training Throughput with Three Scheduling Methods

In this section, we demonstrate the efficacy of our memory planning algorithm and the memory scalability of Split-CNN. We implemented the HMMS and Split-CNN on top of a customized machine learning framework that uses NVIDIA cuDNN V7 to perform most neural network operations.

6.2 Efficacy of Memory Scheduling Algorithm

We evaluated the scheduling plan derived from our offloading/prefetch planning algorithm. Showcasing robustness, we tested our algorithm on a much deeper ResNet-50 network instead of the ResNet-18 one. Specifically, using the same profiling techniques and the cumulative generated data size and the cumulative offload-able data size calculated as in Figure 1, we deduced that ResNet-50 can offload 40% of its intermediate results without hurting training throughput. In our previous discussions, we have already established that VGG-19 can offload all its intermediate results without hurting performance. To validate such claims, we then use three schemes to construct memory plans for the training of such networks with a shared batch size of 64:

1. **Baseline memory plan** does not offload or prefetch, therefore should result in best training throughput.
2. **Layer-wise memory allocation scheme** offloads and prefetches the intermediate results in a layer-wise way. This memory planning scheme is constrained to only offload so much data that the percentage of offloaded data does not exceed the theoretical limit we calculated beforehand.
3. **HMMS** automatically plans the synchronization point of offloading and prefetching memory copies using the algorithm described above. Again, we actively prevent offloading more percentage of the data than we can theoretically handle.

The results as shown in Figure 8 demonstrate that our method for scheduling memory offloading and prefetch always allow the training procedure to execute at no discernible performance degradation. For VGG and ResNet, HMMS results in throughput degradation of only 1.3% and 5.1%, far less than 13.0% and 12.9% experienced by layer-wise memory management policy. The reason for our superior offloading and prefetch performance is that our method can

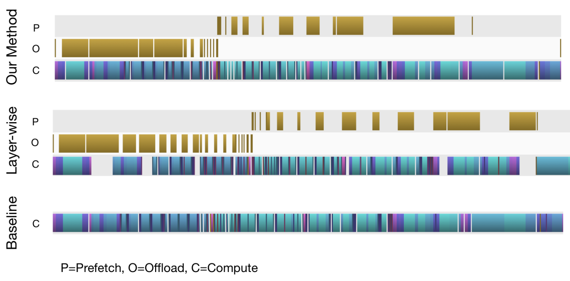


Figure 9. Profiling Results for VGG-19 with Three Offload-Scheduling Methods

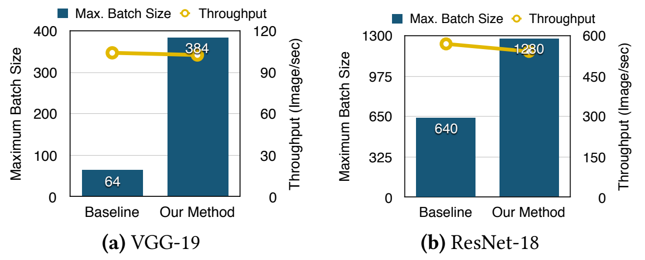


Figure 10. Impact on the Maximum Batch Size and Throughput with number of splits = 4, depth ≈ 75%

plan a longer duration of offloading time without eagerly synchronizing with the computation stream, which often results in the slowdown. We also collected profiling diagrams generated by NVIDIA CUDA profiler (nvprof) to corroborate our conjecture in Figure 9.

With experimental results demonstrating the effectiveness of our method in scheduling memory offloading and prefetch across layers with high variations of memory to compute ratio, we conclude that our scheduling algorithm can resolve challenges of memory scheduling involving increasingly memory-bound layer computations.

6.3 Split-CNN with HMMS

We then demonstrate that Split-CNN, when combined with the HMMS, can significantly increase the trainable batch sizes in ResNet and VGG. One caveat is that, although the HMMS is capable of offloading data close the theoretical limit, significant batch size improvement still cannot occur in ResNet due to its very high memory consumption to computation ratio. A simple trick is used to skew this ratio slightly to our favor: specifically, we adopted a recently proposed technique [6] to re-compute some of the intermediate results needed by the batch normalization layer. After adopting this memory efficient ResNet-18 variant, the percentage of offload-able data grows to 70%, which enabled our techniques to enlarge the batch size even further. Note that the total computation time of the memory efficient ResNet-18 networks is still far from enough to offload all the data.

Our techniques achieved 6x larger batch size on VGG and 2x larger batch size on ResNet with a mere 1.5% and 4.9% throughput degradation respectively as shown in Figure 10. Notably, unlike prior techniques like vDNN [32], our method finds the optimal schedule without a trial-and-error process. We recognized that two factors, enabled by our techniques, contribute the most to the increased train-ability of DNN's:

1. cuDNN convolutions often require a large amount of workspace for each layer computations to compute efficiently. The partitioning of larger convolutional layers into smaller one allows each smaller convolution to re-use the same workspace, thus reducing the memory capacity pressure caused by workspace requirements.
2. Usually, when a deep neural network fails to fit inside the GPU memory, the culprit is usually a very small subset of the neural network layers. Split-CNN enables the breakdown of such a sudden increase of memory capacity requirements and therefore eliminate the possibility of a small subset of layers blocking the train-ability of the entire network.

6.4 Accelerating Distributed Training

Split-CNN can be used to accelerate distributed training because by increasing the batch size, Split-CNN can decrease the number of parameter updates (thus network communication) of distributed training. To demonstrate such benefits, we extrapolated the performance of distributed Split-CNN training based on the measured single-node performance of Split-CNN and established analytical models [31] quantifying the performance of multi-node distributed training.

[31] proved that gradient aggregation using distributed allreduce operation has a lower bound time complexity of $\frac{2|G|}{B_{min}}$ where $|G|$ is the size of the gradient and B_{min} is the minimum network bandwidth between any learner node.

Assuming that bandwidth-bound network communication is the only communication overhead, we can derive the time to compute a single training epoch to be:

$$T_{epoch} = \frac{|D|}{N}\left(T_{forward} + max\left(T_{backward}, \frac{2|G|}{\alpha B_{min}}\right)\right)$$

where $|D|$ refers to the number of training samples in the training dataset, N refers to the mini-batch size, $T_{forward}$, $T_{backward}$ refers to forward, backward propagation computation time and α is the bandwidth utilization efficiency coefficient. In practice, distributed training algorithm usually pipelines backward propagation with gradient aggregation as in [15], thus the *max* function applied to backward propagation computation time and network communication time.

By evaluating the aforementioned performance model with measured parameters from VGG-19, we project the speedup achievable by distributed training using Split-CNN for VGG-19 model in bandwidth constraint environment. In our evaluation, we use an optimistic bandwidth utilization efficiency $\alpha = 0.8$ and vary the network bandwidth from 32

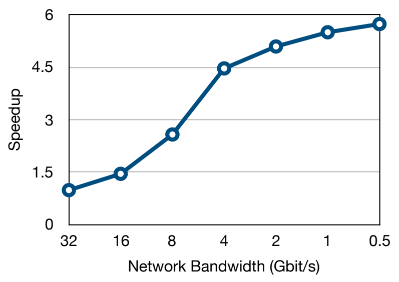


Figure 11. Speedup of Distributed Training with Split-CNN

Gbit/s to 0.5 Gbit/s. As shown in Figure 11, with common high-performance cloud network bandwidth [1] (10 Gbit/s), Split-CNN-based distributed training algorithm can achieve 2.1 speedup factor compared with baseline distributed training algorithms for VGG-19. Moreover, it is worth to note that since gradient aggregation is not the only source of network communication overhead, our model underestimates the cost of network communication and therefore the projected speedup factor is only a lower bound. In a non-bandwidth limited environment, distributed training (e.g., [15]) can still incur roughly 10% network communication overheads.

7 Related Work

There have been several proposals to reduce the memory usage of the deep neural network. Deep neural networks are over-parameterized and have large amounts of redundancy in the network models, resulting in inefficient memory usage. One approach to address this inefficiency is to use quantization or a reduced precision (e.g., 16-bit float instead of 32-bit one) when representing data. [14] proposed vector quantization techniques to reduce the storage size of the models by compressing the weights of the fully connected layer. [40] proposed to use fixed-point representation to quantize the weights and activations. Another network with fixed-point data representation was proposed to use ternary weights and 3-bits activations in [25]. Highly quantized networks have been proposed to improve the performance and energy efficiency [8, 9, 13, 38].

Another approach to reducing network complexity and size is network pruning. It is used to prune redundant and non-informative weights in the network models [18, 19, 30]. Recent work [7] showed that the network pruning can reduce the number of parameters by orders of magnitude in several state-of-the-art neural networks. Deep Compression [17] removes the redundant connections and quantizes weights, and then compresses the weights with Huffman coding to reduce the required memory for inference on large networks.

To resolve the bottleneck caused by the limited GPU memory capacity, vDNN [32] proposed to virtualize the memory usage of deep neural networks through a prefetching and offloading technique that utilizes the CPU DRAM as an external buffer. By dynamically offloading the feature maps from GPU memory to CPU memory during the forward propagation and prefetching them from CPU memory to GPU memory during the backward propagation, vDNN can reduce the required capacity of GPU memory. The performance of this technique largely depends on the communication bandwidth between GPU and CPU and the extent to which memory transfer can be overlapped with computation. Recently, [33] showed that the communication overhead can be mitigated with a compressing DMA engine. Moreover, [2] used convolution tiling on FPGA to reduce off-chip memory access of DNN, but their work also claims difficulties in applying such technique to general purpose processors.

These prior approaches are largely orthogonal to the ideas proposed in this paper, and hence can be used together to reduce the GPU memory usage of the deep neural networks further.

8 Discussion and Conclusion

We started with three challenges to resolve: the difficulties of memory schedule planning when facing increasing memory-bound layers; the increasing demand for larger batch sizes and the ever-growing neural network complexities.

To address these challenges, we presented Split-CNN, an automatic instrumentation that enables new system optimizations to improve the memory scalability of training deep convolutional neural network (CNN). Split-CNN divides the computations across layers of CNN into multiple parts by splitting the input image into smaller patches and performing the computations on these patches independently. We rigorously evaluated the effect of Split-CNN on application-specific performance metrics like classification accuracy to ensure no significant degradation occurs. Moreover, we discovered that a variant of Split-CNN called Stochastic Split-CNN can even enhance such performance metrics.

To fully unleash the potential of Split-CNN, we proposed a heterogeneous memory management system (HMMS) to statically plan the memory allocation, deallocation, offloading and prefetching of deep neural network training. The combination of Split-CNN and HMMS has shown greatly increased scalability of state-of-the-art neural networks (VGG and ResNet) at a much-reduced cost to training throughput. Through experimentation, we demonstrated the ease of use and efficacy of this memory management system, effectively resolving the first challenge we identified.

We then enabled training VGG-19 with 6x larger batch sizes, ResNet-18 with 2x larger batch sizes than training without our proposed techniques; hence we have made big strides towards solving the demand for larger batch sizes.

Demonstrating the effectiveness of our solutions to the first two challenges, we envision that deep learning application developers working with models with high complexities will be able to utilize our proposed techniques to solve the third challenge as they face it.

References

[1] IBM Cloud bandwidth package. https://www.ibm.com/cloud/bandwidth. Accessed: 2019-01-24.

[2] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[3] Lukas Balles, Javier Romero, and Philipp Hennig. Coupling adaptive batch sizes with learning rates. *CoRR*, abs/1612.05086, 2016.

[4] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In Rajiv Gupta, editor, *Compiler Construction*, pages 283–303, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. Parameterized diamond tiling for stencil computations with chapel parallel iterators. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 197–206, New York, NY, USA, 2015. ACM.

[6] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. In-place activated batchnorm for memory-optimized training of dnns. *CoRR*, abs/1712.02616, 2017.

[7] C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors. *Learning both Weights and Connections for Efficient Neural Network*. Curran Associates, Inc., 2015.

[8] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

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

[10] Yann Le Cun. A theoretical framework for back-propagation, 1988.

[11] Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. Automated Inference with Adaptive Batches. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1504–1513, Fort Lauderdale, FL, USA, 20–22 Apr 2017. PMLR.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[13] Steven K. Esser, Rathinakumar Appuswamy, Paul Merolla, John V. Arthur, and Dharmendra S. Modha. Backpropagation for energy-efficient neuromorphic computing. In *NIPS*, 2015.

[14] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir D. Bourdev. Compressing deep convolutional networks using vector quantization. *CoRR*, abs/1412.6115, 2014.

[15] 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. *CoRR*, abs/1706.02677, 2017.

[16] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, New York, USA, 2014.

[17] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[18] Stephen José Hanson and Lorien Pratt. Advances in neural information processing systems 1. chapter Comparing Biases for Minimal Network Construction with Back-propagation, pages 177–185. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.

[19] Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*, pages 164–171, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[20] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[21] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[22] Elad Hoffer and Nir Ailon. Deep metric learning using triplet network. *CoRR*, abs/1412.6622, 2014.

[23] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[24] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. Deep networks with stochastic depth. In *ECCV*, 2016.

[25] K. Hwang and W. Sung. Fixed-point feedforward deep neural network design using weights x002b;1, 0, and x2212;1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, Oct 2014.

[26] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[27] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, USA, 2012. Curran Associates Inc.

[29] Andrew Lavin. Fast algorithms for convolutional neural networks. *CoRR*, abs/1509.09308, 2015.

[30] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *NIPS*, 1989.

[31] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, February 2009.

[32] 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 *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[33] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, and S. W. Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *The 24th International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[34] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

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

[36] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[37] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017.

[38] Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *NIPS*, 2014.

[39] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[40] Vincent Vanhoucke and Mark Z. Mao. Improving the speed of neural networks on cpus. 2011.

[41] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1058–1066, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.