PipeDream: Fast and Efficient Pipeline Parallel DNN Training

Aaron Harlap[†]* Deepak Narayanan[‡]*
Amar Phanishayee[⋆] Vivek Seshadri[⋆] Nikhil Devanur[⋆] Greg Ganger[†] Phil Gibbons[†]

[⋆]*Microsoft Research* [†]*Carnegie Mellon University* [‡]*Stanford University*

Abstract

PipeDream is a Deep Neural Network (DNN) training system for GPUs that parallelizes computation by pipelining execution across multiple machines. Its *pipeline parallel* computing model avoids the slowdowns faced by data-parallel training when large models and/or limited network bandwidth induce high communication-to-computation ratios. PipeDream reduces communication by up to 95% for large DNNs relative to data-parallel training, and allows perfect overlap of communication and computation. PipeDream keeps all available GPUs productive by systematically partitioning DNN layers among them to balance work and minimize communication, versions model parameters for backward pass correctness, and schedules the forward and backward passes of different inputs in round-robin fashion to optimize "time to target accuracy". Experiments with five different DNNs on two different clusters show that PipeDream is up to 5x faster in time-to-accuracy compared to data-parallel training.

1 Introduction

The last five years has seen a rapid increase in the use of Deep Neural Networks (DNNs), with researchers and practitioners applying these models to great effect across a wide range of applications, including image and video classification, speech recognition, and language translation [16, 17, 21, 22, 44]. As DNNs have become more widely developed and used, model sizes have grown to increase effectiveness—models today have tens to hundreds of layers totaling 10–20 million parameters. Such growth not only stresses the already time- and resource-intensive DNN training processes, but also causes the commonly used parallelization approaches to break down.

The most common approach is *data parallelism*, where the DNN model is replicated on multiple worker machines, with each worker processing a subset of the training data. Weight updates computed on individual workers are aggregated to obtain a final weight update that reflects updates across all inputs. The amount of data communicated per aggregation is proportional to the size of the model. Although data-parallel training works well with some popular models that have high computation-to-communication ratios, two important trends threaten its efficacy. First, growing model sizes increase per-aggregation communication. Indeed, some widely-used models are large enough that the communication overheads already eclipse computation time, limiting scaling and dominating total training time (e.g., up to 85% of training time for VGG-16 [36]). Second, rapid increases in GPU compute capacity further shift the bottleneck of training towards communication across models. Our results show these effects quantitatively (Figure 1) for three generations of NVIDIA GPUs (Kepler, Pascal, and Volta), across five different DNN models.

Another approach to distributed training, *model parallelism*, is used traditionally for models that are too large to keep in a worker's memory or cache during training [10, 25, 7]. Model-parallel training involves partitioning the model among workers such that each worker evaluates and performs updates for only a subset of the model's parameters. However, even though model parallelism enables training of very large models, traditional model parallelism can lead to severe underutilization of compute resources since it either actively uses only one worker at a time (if each layer is assigned to a worker) or cannot overlap computation and communication (if each layer is partitioned). In addition, determining how best to partition a DNN model among workers is a challenging task even for the most experienced machine learning practitioners [30], often leading to additional inefficiency.

This paper describes PipeDream, a new distributed training system specialized for DNNs. Like model parallelism, it partitions the DNN and assigns subsets of layers to each worker machine. But, unlike traditional model parallelism, PipeDream aggressively pipelines minibatch processing, with different workers processing different inputs at any instant of time. This is accomplished by injecting multiple inputs into the worker with the first DNN layer, thereby keeping the pipeline full and ensur-

*Work started as part of an internship at Microsoft Research.

ing concurrent processing on all workers. It also uses data parallelism for selected subsets of layers to balance computation load among workers. We refer to this combination of pipelining, model parallelism, and data parallelism as *pipeline-parallel training*.

Pipeline-parallel training has the potential to provide high DNN training performance when data parallelism struggles. In particular, inter-worker communication can be limited to activations (on the forward pass) and gradients (backward) between adjacent layers assigned to different workers. We observe such communication to be up to 95% less than that for data-parallel training.

PipeDream is the first system to combine pipelining, model parallelism, and data parallelism in a general and automated way. It realizes the potential of pipeline parallelism with a design that addresses several challenges. First, like pipelining in processors, achieving high efficiency requires the right partitioning of the DNN into "stages" (layer sub-sequences) that are each executed on a different worker; this depends on both the model architecture and the hardware deployment. Bad partitionings, where stages have widely skewed amounts of work, can lead to workers spending significant time idle. PipeDream automatically determines how to partition the layers of the DNN based on a short profiling run, using an algorithm that balances computation load among the different stages while minimizing communication. Second, since DNNs will not always divide evenly among available workers, PipeDream can use data parallelism for some stages—multiple workers can be assigned to a given stage, processing different minibatches in parallel. Third, unlike traditional uni-directional pipelines, DNN training is bi-directional—the forward pass is followed by a backward pass through the same layers in *reverse* order. PipeDream interleaves forward and backward minibatch processing on each worker, while making sure to route minibatches through the same workers on the backward pass. This helps to keep all workers busy without pipeline stalls, while preventing excessive in-progress minibatches and ensuring model convergence. Fourth, weight versions need to be managed carefully to obtain a high-quality model at the end of training. We find that allowing the backward pass for a given minibatch to use more up-to-date parameters than those used in the corresponding forward pass can be a significant problem. PipeDream maintains parameter value versions for each in-flight minibatch to combat this problem.

Experiments with PipeDream confirm its effectiveness for each of the five models we evaluated and that cover two important DNN classes – CNNs and RNNs (seq-to-seq). When training Inception-v3 [19], VGG16 [36], Resnet-50 [16], and AlexNet [26] on the ILSVRC12 [32] dataset, PipeDream speeds up training by up to 1.45x, 5.12x, 1.21x and 6.76x respectively compared to data-parallel BSP. When training the S2VT [43] model on the MSVD [3] dataset, PipeDream speeds up training by 3x compared to data-parallel BSP.

To summarize, this paper makes four primary contributions. First, it introduces a parallelization approach specialized to DNN training to address communication bottlenecks by combining model parallelism with aggressive pipelining and data parallelism where appropriate. Second, it identifies the key challenges in realizing the performance potential of this idea and details solutions for each. Third, it describes a system (PipeDream) that efficiently implements pipeline-parallel DNN training. Fourth, it experimentally demonstrates that PipeDream allows parallel DNN training in circumstances where communication overheads limit data-parallel training, including cases where data-parallel is slower than single-machine training.

2 Background & Related Work

This section discusses distributed DNN training, including terminology, common approaches and their limitations, and related work, using training of image classification models as a concrete example.

2.1 DNN Training

A DNN model consists of a sequence of layers of different types (e.g., convolutional, fully connected, pooling). DNN models are typically trained using a dataset of labeled images. Training consists of multiple *epochs*, where an epoch is one iteration through all images in the dataset. In each epoch, the model trains over all images in the dataset in *steps*. In each step, the current model first makes a prediction for a small set of training samples, also known as a *minibatch*. This process is referred to as a *forward pass*. To make a prediction, input data from the minibatch is fed to the first layer of the model. Each layer then computes a function over its inputs, often using *learned* parameters (or weights), to produce an output for the next layer. The output of the last layer is the class prediction. Based on the model's predicted label and the actual label of each image, the output layer computes a loss (or error). In the ensuing *backward pass*, each layer computes 1) the error for the previous layer, and 2) the weight update (gradient of the loss) for all relevant layers, which move the model's predictions toward the desired output.

The goal of DNN training is to obtain a high-accuracy model in as little time as possible. This goal can be captured with two metrics: 1) *statistical efficiency*, the number of epochs needed to reach a desired level of accuracy, and 2) *hardware efficiency*, the time required to complete a single epoch. The total training time to reach a desired accuracy level is simply the product of these two metrics [15]. To train large models in a reasonable

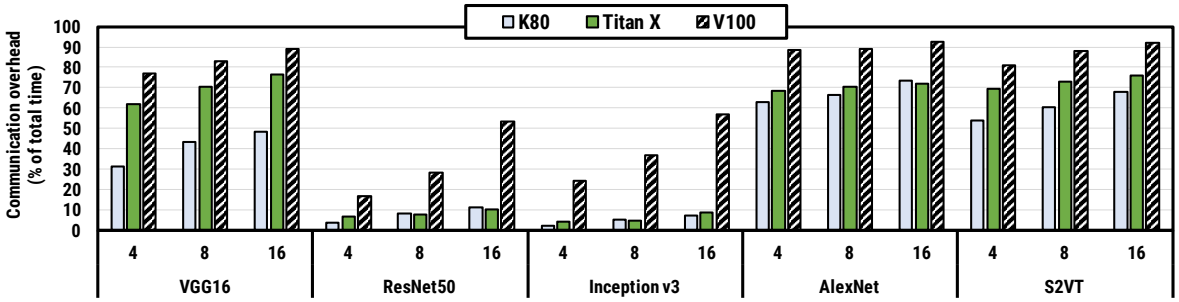


Figure 1: Communication overhead as a percentage of total training time for different hardware configurations. Many models (AlexNet, VGG16, S2VT) have a high communication overhead, even on the relatively slow K80. Two factors contribute to an increase in the communication overhead across *all* models: (i) an increase in the number of data-parallel workers, and (ii) an increase in GPU compute capacity.

amount of time, training is distributed across multiple GPUs[1], usually using one of two approaches: data or model parallelism.

Data Parallelism. With data parallelism, the input dataset is partitioned across multiple GPUs. Each GPU maintains a full copy of the model and trains on its own partition of data while periodically synchronizing weights with other GPUs, using either collective communication primitives [14] or Parameter Servers [28, 9]. The frequency of parameter synchronization affects both statistical and hardware efficiency.

On one end, synchronizing at the end of every minibatch (referred to as *bulk synchronous parallel* or BSP [42]) reduces the staleness of weights used to compute gradients, ensuring good statistical efficiency. However, as shown in Figure 2, BSP requires each GPU to *wait* or *stall* for gradients from other GPUs, thus significantly lowering hardware efficiency. Despite optimizations such as Wait-free Backpropagation [47], where weight gradients are sent as soon as they are available, (common in modern parameter servers), communication stalls are inevitable in data-parallel training due to the structure of the DNN computation, and the fact that communication can often dominate total execution time. Furthermore, rapid increases in computation speeds further shift the training bottleneck towards communication.

Figure 1 quantitatively shows the fraction of training time spent in communication stalls for five different DNN models run on "commodity" public cloud servers using three different generations of NVIDIA GPUs—Kepler (K80), Pascal (Titan X), and Volta (V100)—linked by a 10Gbps network. We focus on three takeaways. First, starting with slower GPUs such as the K80s, we note that some CNNs (VGG16 and AlexNet) and the sequence-to-sequence model for video transcription

[1]For the rest of this paper, we use the terms "GPU", "worker", and "machine" interchangeably, although a machine may have multiple GPUs each running a worker thread.

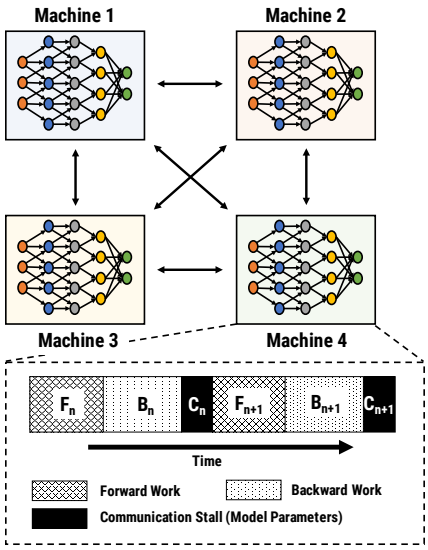


Figure 2: Example data-parallel setup with 4 machines. Timeline at one of the machines shows communication stalls during model parameter exchanges.

(S2VT) spend significant time communicating. Networks such as ResNet50 and Inception-v3 have comparatively low communication overhead. Second, as the number of data parallel workers increases, communication overheads increase for all models. Third, as GPU compute speeds increase (K80s to V100s), communication stalls also increase for all five models.

Prior work has proposed more relaxed synchronization models, where each GPU proceeds with the computation for the next minibatch without stalling for the gradients from the other GPUs. This approach, which we refer to as *asynchronous parallel* or ASP, reduces GPU idle time, and as a result, improves hardware efficiency compared to BSP. However, this process can lead to gradients being computed on stale weights, thereby lowering statistical efficiency. Our experimental results corroborate recent findings that show that ASP does not reduce end-to-end

DNN training time [9, 1, 4].

Model Parallelism. With model parallelism, the model is partitioned across multiple GPUs, with each GPU responsible for only a portion of the model. For machine learning (ML) problems such as matrix factorization, topic modeling, and linear regression, prior work [29, 27, 45, 23] has shown that model parallelism can often achieve faster training times than data parallelism because of the improved statistical efficiency that arises from not using extremely large minibatch sizes; the STRADS framework [23] shows that pipelining multiple minibatches can further improve training times for these ML problems. Model parallelism has also been used for DNNs, but traditionally only as a last resort when the working set of model training is too large to fit in a single worker's memory or cache [25, 7, 10] (making data parallelism not an option). This is because traditional model-parallel DNN training suffers from two major limitations.

First, model-parallel DNN training results in severe under-utilization of GPU resources, as illustrated in Figure 3. The figure shows a partitioning of the DNN layers across four machines, such that each machine is responsible for a group of consecutive layers; in this regime, the inter-layer values (activations and gradients) between these groups are the only parameters that need to be communicated across machines. [2] For each minibatch, only a single stage is active at any instant of time. Pipelining multiple minibatches back-to-back would improve utilization, but is traditionally not done because, 1) the bidirectionality of DNNs (the forward pass is followed by a backward pass through the same layers in reverse order) makes pipelining challenging, and more importantly 2) a naive pipelining mechanism introduces weight update computations on *stale* weights, leading to the final model achieving a lower accuracy than in the data-parallel training.

Second, the burden of partitioning a model across multiple GPUs is left to the programmer [25], resulting in point solutions. Recent work explores the use of reinforcement learning to automatically determine device placement for model parallelism [30]. Unfortunately, such online decision making techniques are time- and resource-intensive; they also don't seamlessly combine pipelining, data-, and model- parallelism.

Other Related Work. Recent work on fast training of Convolutional Neural Networks (CNNs) make use of highly optimized and expensive clusters with high-speed intra- and inter-machine interconnects [14, 12]. Many public cloud providers do not yet offer such optimized server SKUs and one can expect such offerings to be prohibitively expensive when they are offered. In contrast, in our work we investigate the use of commodity SKUs available in public cloud offerings; this is training infrastructure readily accessible to the masses.

[2]While other partitioning schemes are possible, this is the most common, and the one we will use for model parallelism in this paper.

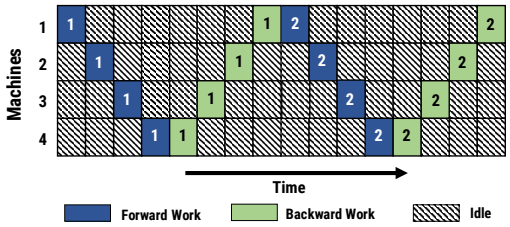


Figure 3: Model parallel training with 4 machines. Numbers indicate minibatch ID. For simplicity, here we assume that forward and backward work in every stage takes one time unit, and communicating activations across machines has no overhead.

CNTK's [33] 1-bit quantization technique tackles the problem of communication bottlenecks in data parallelism training [35]. This approximation strategy lacks generality and is effective for limited scenarios; it does not hurt convergence for some speech models [34], but hurts statistical performance due to noisy gradients in many others [9, 1].

Goyal et al. [14] uses more efficient implementations of `all_reduce`, like the recursive halving-and-doubling algorithm and the bucket algorithm to reduce the amount of data being sent over the network [39]. Others have explored techniques from the HPC literature to reduce the overhead of communication [2, 41]. But all these reduction approaches still involve synchronous communication patterns, resulting in smaller network stalls that only slightly alleviate the communication bottleneck introduced due to ever growing model sizes and faster compute capabilities.

Chen et al. [6] briefly explore the potential benefits of pipelining minibatches in model parallel training, but do not address the conditions for good statistical efficiency, scale, and generality as applicable to large real-world models. In fact, our work shows that pipelining computation naively *is not enough* for real-world models. In our proposed solution (Section 3), we address key issues ignored in prior work, and offer a general and automated solution.

3 Parallel Training in PipeDream

PipeDream combines traditional data parallelism with model parallelism enhanced with pipelining. We call this scheme *pipeline parallelism* (PP). In this section, we first describe PP, and then describe PipeDream's design that addresses the challenges associated with making pipeline-parallel training work effectively.

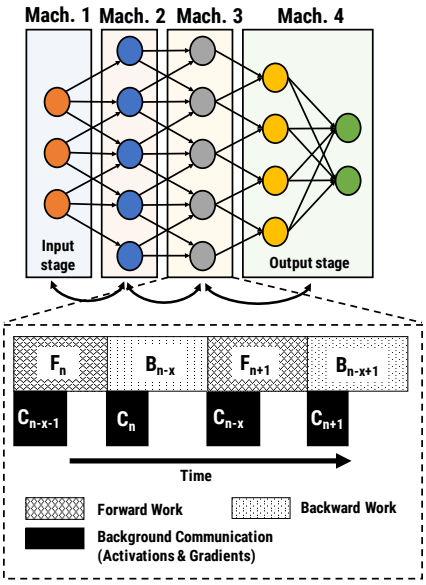


Figure 4: An example pipeline-parallel assignment with four machines and an example timeline at one of the machines, highlighting the temporal overlap of computation and activation / gradient communication.

3.1 Pipeline Parallelism

Pipeline-parallel training partitions the layers of the model being trained into multiple *stages* – each stage contains a *consecutive* set of layers in the model. Each stage is mapped to a separate GPU that performs both the forward and backward pass for all the layers in that stage. We refer to the stage that contains the input layer as the *input stage*, and the one that contains the output layer as the *output stage*. Figure 4 shows a simple example of a pipeline-parallel assignment, where the DNN is split across four machines.

In the simplest case, only one minibatch is active in the system, as in traditional model-parallel training. Figure 3 shows the computation timeline of an example configuration with four machines and one active minibatch in the pipeline. In the forward phase, each stage performs the forward pass for the minibatch for the layers in that stage and sends the results to the next stage. The output stage, after completing its forward pass, computes the loss for the minibatch. In the backward phase, each stage performs the backward pass and propagates the loss to the previous stage. With only one active minibatch, at most one GPU is active at any given point in time.

To ensure that no GPU is idle at any point in time, we inject multiple minibatches into the pipeline one after the other, thus enhancing model-parallel training with pipelining. On completing the forward pass for a minibatch, each stage asynchronously sends the output activations to the next stage, while simultaneously starting to process another minibatch. Similarly, after completing the backward pass for a minibatch, each stage asynchronously sends the gradient to the previous stage, while starting computation for another minibatch. Pipelining has two main advantages over data-parallel training:

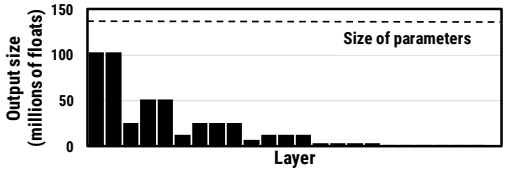


Figure 5: Sizes of layer output data for VGG16 with a minibatch size of 32 on ImageNet1K data. The black dotted line indicates the size of the model parameters.

Pipelining communicates less. PP requires far less communication compared to BSP. Figure 5 compares the size of each layer's output to the overall size of model parameters for VGG16. Instead of having to communicate all the parameters, as is done in BSP, each machine in a PP execution only has to communicate the output data of *one* of the layers. This often results in large reductions in communication (e.g., >90% reduction for VGG16).

Pipelining overlaps computation and communication. Asynchronous communication of forward output activations and backward gradients across stages results in a significant overlap of communication with computation of a subsequent minibatch, thus achieving better hardware efficiency compared to BSP (Figure 4).

While pipelining by itself reduces training time compared to data parallelism, we observe model parallelism and data parallelism work best for different types of layers [25]. As a result, PipeDream aims to combine pipelined model parallelism and data parallelism in a manner that minimizes overall training time. Figure 6 shows how pipeline parallelism might partition layers of a hypothetical model across stages on 8 machines. However, there are three challenges that need to be addressed to make this approach effective for large real-world DNN models:

1. Automatic partitioning of work across available compute resources.
2. Scheduling of computation to maximize throughput while ensuring forward progress in the learning task.
3. Ensuring that learning is effective in the face of asynchrony introduced by pipelining.

The remainder of this section describes these challenges and PipeDream's approach to address them.

3.2 Partitioning Layers Across Machines

Given a model and a set of machines, PipeDream's first challenge is to automatically partition layers of the model across available machines so as to minimize overall training time. Figure 7 shows the workflow adopted by PipeDream to partition the layers of the DNN among

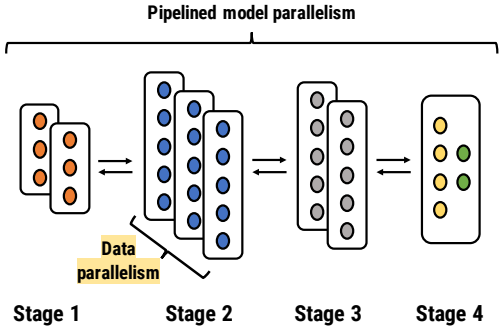


Figure 6: Pipeline Parallel training in PipeDream combines pipelining, model- and data-parallel training.

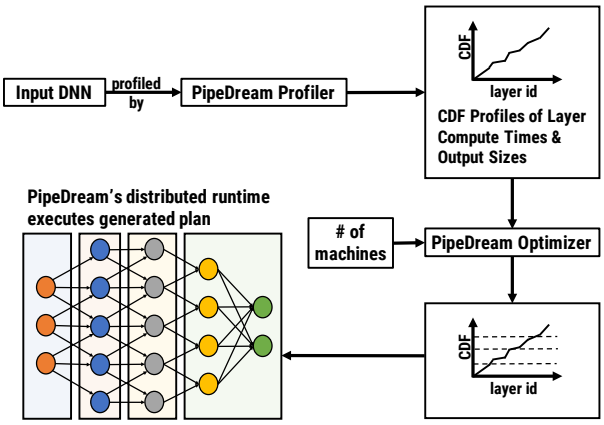


Figure 7: PipeDream's automated mechanism to partition DNN layers into stages. PipeDream first profiles the input DNN, to get estimates for each layer's compute time and output size. Using these estimates, PipeDream's optimizer partitions layers across available machines.

the available machines. When partitioning layers into different stages across machines, PipeDream's partitioning algorithm must ensure that each stage roughly performs the same amount of total work. At the same time, the partitioning algorithm must also ensure that the amount of data communicated across stages is as small as possible, to avoid communication stalls. Load imbalance across machines or excessive communication between machines can lower hardware efficiency (throughput).

Taking these factors into account, given a DNN with N layers and M available machines, PipeDream first profiles the model on a single machine, and then runs a partitioning algorithm that groups layers into stages, while also determining the replication factor for each stage that minimizes the overall training time for the model.

Profiling the DNN Model. Our profiling mechanism exploits the fact that DNN training shows little variance in the computation and communication time across mini-batches. PipeDream records three quantities for each layer l: 1) T_l, the total computation time across the forward and backward pass for the layer, 2) a_l, the size of the output activations of the layer (also the size of input gradients in the backward pass), and 3) w_l, the size of parameters for layer l.

To determine T_l for all layers, PipeDream profiles a short run of the DNN model using 1000 minibatches on one of the machines.[3] Using this profile, PipeDream computes T_l as the sum of the forward and backward computation times for the layer l.

All communication happens in three steps: 1) move data from the GPU to the CPU of the sender, 2) send data from sender to receiver over the network, and 3) move data from the CPU to the GPU of the receiver. PipeDream estimates the time taken for communication as the amount of data that needs to be transferred divided by the network bandwidth on the communication link. C_l, the time taken to communicate the activations from layer l to $l+1$ in a pipeline, is estimated using a_l. The amount of data communicated per worker in data-parallel configurations with m machines is $4 \times (m-1) \times |w_l|/m$; this is used to estimate W_l^m, the time for weight synchronization for the layer when using a distributed parameter server. 4表示fp32对应的4 Bytes

PipeDream's Partitioning Algorithm. Our partitioning algorithm takes the output of the profiling step, and computes: 1) a partitioning of layers into stages, 2) the replication factor for each stage, and 3) optimal number of minibatches to keep the training pipeline busy.

The partitioning algorithm tries to minimize the overall training time of the model. For a pipelined system, this problem is equivalent to minimizing the time taken by the slowest stage of the pipeline. This problem has the optimal sub-problem property; a pipeline that maximizes throughput given a machine count is composed of sub-pipelines that maximize throughput for smaller machine counts. Consequently, we can find the optimal solution to this problem using Dynamic Programming.

Let $A(j,m)$ denote the time taken by the slowest stage in the optimal pipeline between layers 1 and j using m machines. The goal of our algorithm is to find $A(N,M)$, and the corresponding partitioning. Let $T(i \rightarrow j, m)$ denote the time taken by a single stage spanning layers i through j, replicated over m machines.

$$T(i \rightarrow j, m) = \frac{1}{m} \max \left(\sum_{l=i}^{j} T_l, \sum_{l=i}^{j} W_l^m \right)$$

where the left term inside the max is the total computation time for all the layers in the stage, and the right term is the total communication time for all the layers in the stage.

The optimal pipeline consisting of layers from 1 through j using m machines could either be a single stage replicated m times, or be composed of multiple stages.

[3] All the GPUs used in individual experiments are identical. As a result, it is sufficient to profile performance on a single GPU.

Case 1: The optimal pipeline contains only one stage, replicated m times. In this case,

$$A(j, m) = T(1 \rightarrow j, m)$$

Case 2: The optimal pipeline contains more than one stage. In this case, it can be broken into an optimal sub-pipeline consisting of layers from 1 through i with $m - m'$ machines followed by a single stage with layers $i + 1$ through j replicated over m' machines. Then, using the optimal sub-problem property, we have

$$A(j, m) = \min_{1 \leq i < j} \min_{1 \leq m' < m} \max \begin{cases} A(i, m - m') \\ 2 \cdot C_i \\ T(i + 1 \rightarrow j, m') \end{cases}$$

where the first term inside the max is the time taken by the slowest stage of the optimal sub-pipeline between layers 1 and i with $m - m'$ machines, the second term is the time taken to communicate the activations and gradients between layers i and $i + 1$, and the third term is the time taken by the single stage containing the remaining layers in a data-parallel configuration of m' machines.

Initialization. $A(1, m) := T(1 \rightarrow 1, m)$, where $T(.)$ is as defined above, and m is varied from 1 through M (the total number of machines). $A(i, 1) := T(1 \rightarrow i, 1)$, where i is varied from 1 through N (the total number of layers in the model).

Runtime Analysis. The total number of sub-problems is $O(NM)$. Time complexity per sub-problem is also $O(NM)$, leading to a total time complexity of $O(N^2M^2)$.

Based on the partitioning generated by our algorithm, the optimal number of minibatches admitted per input stage to keep the pipeline full in steady state is given by

⌈ (# machines) / (# machines in the input stage) ⌉.

We refer to this quantity as the `NUM_OPT_ACTIVE_MINIBATCHES` (NOAM).

3.3 Work Scheduling

Unlike traditional uni-directional pipelines, pipelined DNN training involves a bi-directional pipeline. The forward pass for a minibatch starts at the input layer and the backward pass ends at the input layer. Consequently, each active minibatch in the pipeline may be in a different layer, either in the forward pass or backward pass. As a result, each machine in the system has to make a choice between two options: i) perform the forward pass for a minibatch, thus pushing the minibatch to downstream machines, and ii) perform the backward pass for a different minibatch, thus ensuring forward progress in learning.

A simple scheduling mechanism that always prioritizes forward work hinders overall forward progress as weight updates can be applied only once backward passes complete. Similarly, always prioritizing backward work may periodically result in idle machines with no available work. We propose a scheduling mechanism that avoids these problems.

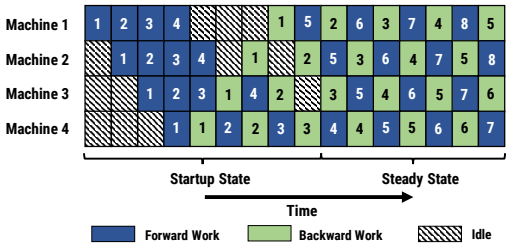


Figure 8: An example pipeline with 4 machines, showing startup and steady states.

In the startup phase, the input stage admits NOAM minibatches to keep the pipeline full in steady state. Once in steady state, each stage *alternates* between performing the forward and backward pass for a minibatch. We call this mechanism *one-forward-one-backward* (1F1B). In a balanced pipeline, 1F1B ensures that no GPU is idle in steady state and that we make forward progress in learning from each minibatch.

Figure 8 shows the corresponding compute timeline for a pipeline with 4 stages each running on one machine. The NOAM for this configuration is 4. In the startup phase, the input stage admits exactly four minibatches that propagate their way to the output stage. As soon as the output stage completes the forward pass for the first minibatch, it performs the backward pass for the same minibatch, and then starts alternating between performing forward and backward passes for subsequent minibatches. As the backward pass starts propagating to earlier stages in the pipeline, every stage starts alternating between forward and backward pass for different minibatches. As shown in the figure, in the steady state, every machine is busy either doing the forward pass or backward pass for a minibatch. For 1F1B to be effective, it is *not necessary* for the forward pass to take as long as the backward pass. In fact, we observe that in practice, the backward pass is always larger than the forward pass, and 1F1B remains an effective scheduling mechanism.

When stages run in a data-parallel configuration, replicated across multiple GPUs, we use deterministic round-robin load balancing (`minibatchID` mod `stageReplicaID`) to spread work from the previous stages across the replicas. Such deterministic load-balancing ensures that the backward pass for a minibatch is performed on the machine responsible for the minibatch's forward pass.

Both the 1F1B scheduling policy for stages in a pipeline and the round-robin scheduling policy for load balancing across replicated stages are *static* policies. Thus, they can be executed by each machine independently without requiring expensive distributed coordination.

3.4 Effective Learning

In a naively pipelined system, the forward pass for each minibatch is performed using one version of parameters and the backward pass using a different version of parameters. Figure 8 illustrates this using a partitioning with no data parallelism. If we observe stage 1 (machine 1), the forward pass for minibatch 5 is performed after the updates from minibatch 1 are applied, whereas the backward pass for minibatch 5 is performed after updates from minibatches 2, 3, and 4 are applied. As a result, in the backward pass for minibatch 5 on stage 1, the gradient is computed using a different set of weights than the ones used in the corresponding forward pass; this discrepancy in weight versions can prevent the model from converging.

Furthermore, different stages in the DNN model suffer from different degrees of staleness. For example, in the third stage, each minibatch has only one interleaved update between its forward and backward pass, while the output stage has no interleaved updates. This asymmetry across layers can further impact model convergence. Our experimental results show that naive pipelining *does not* achieve the same accuracy as data-parallel training. To address this problem, PipeDream uses two techniques.

Weight Stashing. Weight Stashing maintains multiple versions of the weights, one for each active minibatch. When performing the forward pass, each stage processes a minibatch using the latest version of weights available. After completing the forward pass, PipeDream stores the weights used as part of the *intermediate* state for that minibatch. When performing the minibatch's backward pass, the *same version* of the weights is used to compute the weight gradient.

Weight stashing ensures that *within a stage*, the same version of model parameters are used for the forward and backward pass of a given minibatch. For example, in Figure 8, minibatch 5 uses parameters updates from batch 1 on machine 1 and from 2 on machine 2. Weight stashing says nothing about the consistency of parameter versions used for a given minibatch *across* stages.

Vertical Sync. Vertical Sync eliminates the potential inconsistency *across stages*. For example, in Figure 8, using vertical sync, minibatch 5 uses parameters updated by minibatch 1 on all machines for both its forward and backward passes. Each minibatch (m_i) that enters the pipeline is associated with the latest weight version ($w^{(i-x)}$) seen at the input stage. This information is propagated along with the activations and gradients as the minibatch m_i flows through the pipeline in the forward direction. Across all stages, the forward pass for m_i uses the stashed weights $w^{(i-x)}$, as opposed to the latest weight update. After performing the backward pass for m_i (using stashed weights $w^{(i-x)}$), each stage independently applies weight updates to create the latest weights ($w^{(i)}$), and can then delete $w^{(i-x)}$. This coordination across stages is asynchronous.

Staleness. We can now formalize the degree of staleness of weight updates for each of these techniques. For this discussion, we assume a straight pipeline with the model split into n stages; the weights in each stage are represented as w_1, w_2, and so on. In addition, we denote $w_1^{(t)}$ as the weights w_1 after t minibatches.

Now, after every minibatch, we compute the gradient $\nabla f(w_1, w_2, \ldots, w_n)$ averaged over all samples in the minibatch. Vanilla minibatch SGD (f is the loss function we're trying to optimize and ν is the learning rate) has the following gradient update,

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t)}, w_2^{(t)}, \ldots, w_n^{(t)})$$

With weight stashing, gradients in stage 1 are computed with weights that are n steps delayed, gradients for stage 2 are computed with weights that are $n-1$ steps delayed, and so on. Mathematically, this means our weight update looks like,

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t-n+1)}, w_2^{(t-n+2)}, \ldots, w_n^{(t)})$$

Without weight stashing, the weight update is not a valid gradient of the loss function f for any weight vector $w_1, w_2, \ldots, w_n$.

Adding vertical sync alters the weight update to,

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t-n+1)}, w_2^{(t-n+1)}, \ldots, w_n^{(t-n+1)})$$

This is semantically the same as data parallelism with BSP synchronization on n machines (with the same original minibatch size on each machine).

Weight stashing is critical for meaningful learning.[4] PipeDream's default semantics (weight stashing but no vertical sync) are between regular minibatched SGD on a single machine, and data parallelism with BSP synchronization [8, 18]. Our evaluation demonstrates its effectiveness across several models, datasets, and hardware configurations.

3.5 GPU Memory Management

As minibatches enter and leave the pipeline, the system has to ensure that the inputs, weights, and other intermediate state required by the GPU for its computation are present in GPU memory. If not managed carefully, the overhead of dynamic memory allocation in the GPU, and data transfer between GPU and CPU memory can greatly reduce hardware efficiency.

PipeDream extracts the layer parameters from the DNN model and computes the size of activations, parameters, and intermediate state that needs to be stored at each stage, across the active minibatches present in the pipeline. The

[4] In our experiments, we find that the impact of *vertical sync* is negligible. PipeDream's default semantics exclude vertical sync as it requires more metadata to be stored at every stage in the pipeline.

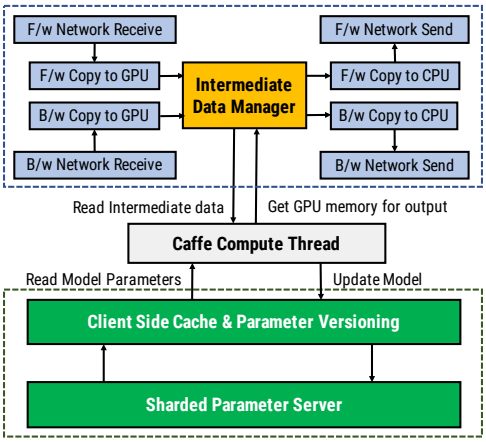


Figure 9: Architecture of the stage runtime and integration with Caffe. PipeDream provides the ML worker thread (Caffe) pointers to GPU memory containing layer input data, parameters, and buffers for recording layer outputs and parameter updates. PipeDream manages buffer pools, and handles all intra- and inter-machine communication.

number of minibatches for which each stage has to maintain intermediate state varies from stage to stage. While the output stage has to maintain intermediate state for only one active minibatch, the input stage needs to do so for NOAM minibatches. PipeDream allocates all required GPU memory at the beginning of training, and reuses the allocated memory as appropriate. This significantly reduces the overhead of dynamically managing GPU memory.

4 Implementation

Figure 7 shows PipeDream's high-level workflow. The input to our system is a model architecture, the training dataset, and the number of GPUs that will be used for training. PipeDream first profiles the model on a single machine with a subset of minibatches from the training dataset. It then runs the optimization algorithm described in Section 3.2 to partition the DNN model into k stages, with some stages replicated. The PipeDream runtime then assigns each stage to a single GPU.

Figure 9 shows the high-level architecture of the stage runtime in PipeDream. The interface to PipeDream is implemented as a C++ library that manages the parameter and intermediate data for the ML worker that runs on the GPU. In our current implementation, we use Caffe [20] as the ML worker. However, PipeDream is extensible and can work with other ML frameworks such as Tensorflow [1], MXNet [5], and CNTK [33] as well.

As an initialization step, the PipeDream library in each machine initializes the GPU data structures corresponding to the stage that is assigned to the machine. This involves 1) initializing the ML worker with the layers that it must execute as part of the stage, and 2) statically allocating memory in the GPU for the activations, weights, gradients, and the intermediate state (which includes the input activations and stashed weights for each active minibatch). Once a machine is initialized, the ML worker pulls its next work assignment from PipeDream; PipeDream's runtime provides the ML worker with pointers to input data. The input stage kick starts the pipeline by creating a minibatch of forward work for its ML worker. From then on, each machine follows the 1F1B scheduling algorithm, while limiting the total number of active minibatches to NOAM.

For the assigned minibatch (forward or backward), the ML worker iterates through each layer in the stage and performs the relevant work for the minibatch. The ML worker uses appropriate PipeDream API calls to get pointers to the inputs, parameters, outputs, gradients, and intermediate state for each layer. Once the minibatch is processed, the ML worker indicates the completion of work to PipeDream, and pulls its next work item.

Parameter State. For each stage, PipeDream maintains all parameters associated with the layers assigned to the stage directly in GPU memory. The parameters for each layer are stored separately, and each assigned a unique ID. If the stage is not replicated, PipeDream applies the updates to the most recent version of the parameter data stored in GPU memory when the weight update is available in the provided GPU buffer. If the stage is replicated, the weight update is copied to host memory and then sent to the parameter server. When a newer version of the parameters becomes available, the prior version is *not* immediately discarded, as part of the weight stashing scheme. Parameter data is only discarded once a backward pass that uses fresher parameters is performed.

Intermediate State. Each layer's intermediate data is also assigned a unique blob ID. Upon receiving intermediate data from the prior stage (or from disk in the case of the input stage), PipeDream copies the intermediate data to GPU memory and places a pointer to the associated buffer in a work queue. Intermediate data from the forward pass is not discarded until the associated minibatch completes that stage's backward pass. Intermediate data from the backward pass is released as soon as the ML worker finishes using it, and if necessary, after it is sent to the next stage. Due to the differing requirements for intermediate data in the forward and backward pass, stages in PipeDream commonly manage multiple versions of intermediate data from forward passes, and just a single version of intermediate data from the currently running backward pass.

Data Parallelism. PipeDream uses a distributed parameter server, similar to GeePS [9], to synchronize parameters for layers of data-parallel stages. Using wait-free back propagation, weight gradients are communicated to

servers as soon as they are as computed, rather than waiting for computation to finish for all layers. Each worker contains an instance of a parameter server shard that stores a unique subset of the parameters. The server shards push the newest version of their parameters to the other shards as soon as updates from all stage replicas are aggregated.

Since we support replication of individual stages, data-parallel training can be thought of as a special case in our framework – we represent this as a single stage that contains all the layers of the DNN model, and replicate the stage across all available machines.

All inter-machine communication between PipeDream's stages, both in data-parallel and pipeline-parallel settings, uses ZeroMQ [46] and an efficient communication stack with fast custom serialization.

Checkpointing. PipeDream supports periodic checkpointing of model parameters for fault-tolerance, with default checkpointing across stages at the end of every epoch. Checkpoints don't require expensive global coordination; each stage locally decides to dump its model parameters when it performs the backward pass for the last minibatch in an epoch. Restarting a failed training run due to a stage failure entails starting from the last epoch successfully checkpointed by all the stages.

5 Evaluation

This section compares the effectiveness of PipeDream with data-parallel training and model-parallel training across models on two clusters. The results of our experiments support a number of important findings: 1) combining pipelining, model parallelism, and data parallelism performs significantly better than using model parallelism or data parallelism alone, 2) PipeDream greatly reduces the overhead of communication compared to data-parallel training, and 3) PipeDream's improvements are higher for configurations that have a lower computation-to-communication ratio.

5.1 Experimental Setup

Datasets. We used two datasets in our experiments. The first is the dataset for the Large Scale Visual Recognition Challenge 2012 (ILSVRC12) [32], also called the ImageNet 1K dataset. This dataset has ∼1.3 million training images categorized into 1000 classes, and 50,000 validation images. The second is the Microsoft Video description corpus (MSVD) [3], which is a collection of YouTube clips depicting different activities, collected on Mechanical Turk. The dataset contains 1,970 videos and a vocabulary of 12,594 words to describe them.

Clusters. We used two different clusters in our experiments. *Cluster-A* is a private cluster of NVIDIA Titan X GPUs with 12 GB of GPU device memory. Each machine has a E5-2698Bv3 Xeon CPU with 64 GB of RAM. The machines are connected via a 25 Gbps Ethernet interface. *Cluster-B* is public cloud cluster (AWS p3.2xlarge instances) of NVIDIA V100 GPUs, with 16 GB of GPU device memory. Each machine has a E5-2690 Xeon CPU, 64 GB of RAM with a 10 Gbps Ethernet interface. Machines on both clusters run 64-bit Ubuntu 16.04 with CUDA toolkit 8.0 and cuDNN v6.

In comparison to Cluster-A, Cluster-B has faster GPUs, but a slower network. As a result, as we show later in this section, the performance benefits of PipeDream are higher in Cluster-B than in Cluster-A.

Models and Training Methodology We used three different DNN models in our experiments: 1) VGG16 [36] with a model size of 550 MB, 2) Inception-v3 [19] with a model size of 157 MB, and 3) S2VT [43], a sequence-to-sequence model for video transcription, with a model size of 349MB. We use the ILSVRC12 dataset to train VGG16 and Inception-v3, and the MSVD dataset to train S2VT model. In our experiments, we trained the VGG16 and S2VT models using SGD with a momentum of 0.9 and an initial learning rate of 0.01. For Inception-v3, we used RMSProp [40] with an initial learning rate of 0.045, decayed every two epochs using an exponential rate of 0.94. We used a mini-batch size of 32 per machine for VGG16 and Inception-v3 and a mini-batch size of 80 per machine for S2VT. For all the experiments, we measure the time taken to train the models until they reach their *advertised validation accuracy*: top-1 accuracy of 68% for VGG16, top-1 accuracy of 67% for Inception-v3, and METEOR [11] score of 0.294 for S2VT. Guided by prior work, we adjust the learning rate during training to converge to the desired result faster [31, 37, 38, 13, 24].

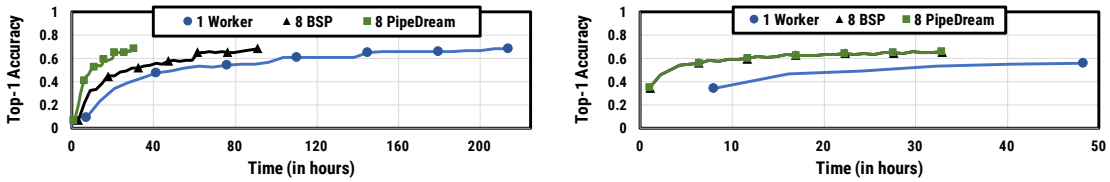
PipeDream's Data-Parallel Implementation. To measure the performance improvements introduced from using PipeDream, we compare to PipeDream in data-parallel and single machine configurations. This allows us to perform a fair comparison of the different schemes. To confirm its efficiency, we compared PipeDream's data-parallel implementation to the open source version of GeePS, an efficient data-parallel DNN training system that also uses Caffe at individual workers and performs favorably, if not better than, other state-of-the-art DNN frameworks [9]. In our experiments, PipeDream's data-parallel configuration ran at least as fast as GeePS for all models and datasets tested.

5.2 PipeDream vs. Data Parallelism

Table 1 summarizes results comparing PipeDream with data-parallel training (BSP). For the three models, the table shows PipeDream's auto-generated configuration and the corresponding speedup in training time over single machine and data-parallel training (BSP). It also shows the communication reduction achieved by PipeDream

DNN Model	# Machines (Cluster)	BSP speedup over 1 machine	PipeDream Config	PipeDream speedup over 1 machine	PipeDream speedup over BSP	PipeDream communication reduction over BSP
VGG16	4 (A)	1.47×	2-1-1	3.14×	2.13×	90%
	8 (A)	2.35×	7-1	7.04×	2.99×	95%
	16 (A)	3.28×	9-5-1-1	9.86×	3.00×	91%
	8 (B)	1.36×	7-1	6.98×	5.12×	95%
Inception-v3	8 (A)	7.66×	8	7.66×	1.00×	0%
	8 (B)	4.74×	7-1	6.88×	1.45×	47%
S2VT	4 (A)	1.10×	2-1-1	3.34×	3.01×	95%

Table 1: Summary of results comparing PipeDream with data-parallel configurations (BSP) when training models to their advertised final accuracy. "PipeDream config" represents the configuration generated by our partinioning algorithm—e.g., "2-1-1" is a configuration in which the model is split into three stages with the first stage replicated across 2 machines.



(a) VGG16 **(b)** Inception-v3

Figure 10: Accuracy vs. time for VGG16 and Inception-v3 with 8 machines on Cluster-A

compared to data-parallel training.

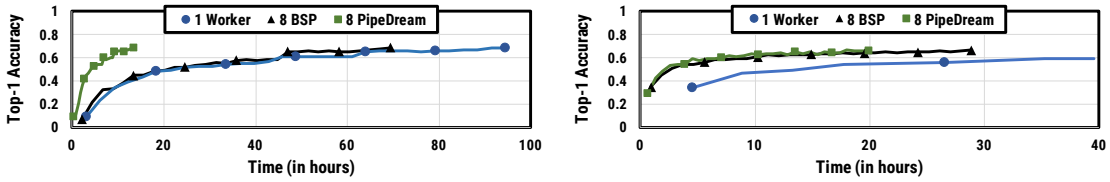
PipeDream Configurations. As described in Section 3.2, given a DNN model and a set of machines, PipeDream's optimizer selects the best configuration that partitions the layers of the model into multiple stages and assigns one or more machines to each stage. While most prior research has focused on improving data-parallel training, our results indicate that the best configurations are neither fully data-parallel nor fully model-parallel. In all but one of our experiments, the best PipeDream configuration combines model parallelism, pipelining, and data parallelism; each of these configurations significantly outperform data-parallel training, thus highlighting the importance of pipeline parallelism. PipeDream's optimizer recommends data-parallel as the best configuration for Inception-v3 with 8 machines in Cluster-A.

Base Results: 8 Machines in Cluster-A. Figure 10 shows accuracy vs. training time for VGG16 and Inception-v3, using 8 machines in Cluster-A, for both BSP and PipeDream [5]. The first conclusion we draw is that for VGG16, BSP with 8 machines reduces training time by only 2.35× compared to training with a single machine since with 8 machines, the communication overhead for VGG16 in Cluster-A is 72%. PipeDream eliminates 95% of this communication overhead thereby improving performance by 7.04× compared to training with single machine (2.99× compared to BSP). Second, for Inception-v3, the communication overhead with 8 machines on Cluster-A is just 5%. As a result, BSP achieves near perfect scaling with a 7.66× speedup over a single machine. PipeDream's partitioning algorithm (Section 3.2) selects a data-parallel configuration (no pipelining or model parallelism) to train Incpetion-v3 on Cluster-A, thus matching the data-parallel BSP performance (Figure 10b).

Effect of using faster compute (V100s). Figure 11 shows accuracy vs. training time for VGG16 and Inception-v3, using 8 machines in Cluster-B, for both BSP and PipeDream. Compared to Cluster-A, Cluster-B employs faster V100 GPUs with 10Gbps interconnect between the machines (as granted by the cloud provider). Thus, models running on Cluster-B have lower computation-to-communication ratios. We note that the faster GPUs result in faster end-to-end training time — e.g., training time for VGG16 reduces from 220 hours on Cluster-A to little less than 100 hours on Cluster-B (Figures 10 and 11). We also observe that the higher communication overhead causes both BSP and PipeDream to scale less effectively to 8 machines. However, this increased communication overhead affects BSP significantly more than PipeDream. Moving from Cluster-A to Cluster-B, the speedup of PipeDream over BSP increases from 2.99× to 5.12× for VGG16. Even for Inception-v3, PipeDream improves training time by 45% compared to BSP on Cluster-B.

Due to space constraints we do not present end-to-end training results for AlexNet [26] and ResNet-50 [16] models. Experiments with these models on Cluster-B showed that PipeDream provides a 1.21x and 6.78x throughput improvement for ResNet-50 and AlexNet respectively, compared to 8 machine data-parallel BSP.

[5] For Figure 10– 12 each displayed point represents 5 epochs



(a) VGG16 **(b)** Inception-v3

Figure 11: Accuracy vs. time for VGG16 and Inception-v3 with 8 machines on Cluster-B

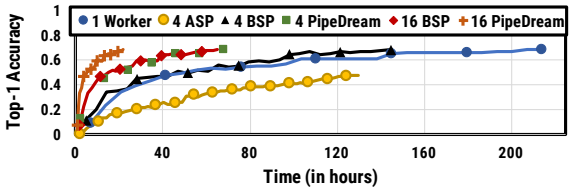


Figure 12: Accuracy vs. time for different configurations for VGG16 on Cluster-A with 4 and 16 workers.

Effect of varying number of machines. With an increase in the number of machines, the communication overhead also increases for all models (as shown in Figure 1). Figure 12 compares the accuracy vs. training time for VGG16, using 4 and 16 machines in Cluster-A, for BSP and PipeDream. As expected, BSP scales poorly with increasing number of machines. With 4, 8, and 16 machines, BSP offers speedups of only 1.47×, 2.35×, and 3.28×, respectively, compared to single machine training. In contrast, PipeDream offers speedups of 3.14×, 7.04×, and 9.86× with 4, 8, and 16 machines compared to training on a single machine. Notably, PipeDream with 4 machines is almost as good as BSP with 16 machines.

Comparison to asynchronous parallel (ASP). To reduce communication overhead in data-parallel training, we experimented with running four machine data-parallel with ASP synchronization. Unlike BSP, which synchronizes the parameter data after every mini-batch, ASP has no synchronization, and the workers use the most recent parameter data *available*. Figure 12 also shows the accuracy vs. time curve for ASP with 4 machines in Cluster-A. Due to ASP's poor statistical efficiency, PipeDream reaches a 48% accuracy 7.4x faster than ASP data-parallel, even though ASP has no communication overhead.

Training a Recurrent Neural Network. VGG16 and Inception-v3 are Convolutional Neural Networks (CNNs), used for tasks such as image classification. We also evaluate PipeDream's performance on the S2VT model, which is a sequence-to-sequence recurrent neural network that generates descriptive captions for videos. For many recurrent neural networks consisting of LSTM layers, BSP data-parallel training scales poorly. For S2VT, BSP with 4 machines reduces training time by only 1.1x compared to single machine training. This is because with 4 machines, the communication overhead for S2VT on Cluster-A is 70%. PipeDream reduces the communication overhead by 95% compared to BSP, in turn slashing training time by 3.34× compared to single machine training (3.01× compared to BSP).

5.3 Value of Data Parallelism in stages

Figure 13 plots the reduction in training time compared to single machine training for three parallelization approaches: 1) simple model parallelism (no pipelining or data parallelism), 2) pipeline parallelism without data parallelism (no stage replication), and 3) PipeDream (combined model parallelism, pipelining and data parallelism). The figure shows these results for training VGG16 using 4 and 8 machines on Cluster-A.

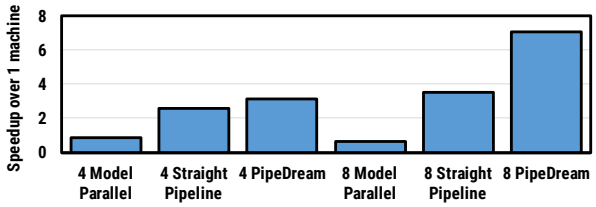


Figure 13: Model Parallelism vs. Pipeline Parallelism vs. PipeDream for VGG16 on Cluster-A

Model Parallelism. Simple model parallelism uses only one machine at any point in time, and hence is slower than the single machine configurations. We also implement model-parallel configurations in Tensorflow [1], which does not support pipelining but implements model parallelism as described in this paper; we observe similar performance drop-offs.

Straight Pipelines. Combining model parallelism with pipelining results in straight pipeline configurations (no data parallelism compared to PipeDream). Straight pipeline configurations greatly reduce training time compared to single machine training—2.56× and 3.49× with 4 and 8 machines, respectively. In fact, these improvements are better than data parallel training, which achieves corresponding speedups of 1.47× and 2.35× compared to single machine training.

Pipeline Parallelism. PipeDream's pipeline parallelism provides the biggest reductions in training time—3.14× and 7.04× with 4 and 8 machines compared to

single machine training. These results demonstrate that a combination of pipelining, model parallelism and data parallelism achieve faster training than either model parallelism, model parallelism with pipelining, or data parallelism.

6 Conclusion

Pipeline-parallel DNN training addresses the communication overheads that bottleneck data-parallel training of very large DNNs. PipeDream automatically partitions and aggressively pipelines DNN training across worker machines. Compared to state-of-the-art approaches, PipeDream is up to 5× faster in "time to target accuracy" for experiments with five different DNNs on two different clusters.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016. URL https://www.tensorflow.org/.

[2] Baidu Inc. Bringing HPC Techniques to Deep Learning, 2017. URL http://research.baidu.com/bringing-hpc-techniques-deep-learning/.

[3] D. L. Chen and W. B. Dolan. Collecting highly parallel data for paraphrase evaluation. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 190–200. Association for Computational Linguistics, 2011.

[4] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. In *International Conference on Learning Representations Workshop Track*, 2016. URL https://arxiv.org/abs/1604.00981.

[5] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL http://arxiv.org/abs/1512.01274.

[6] X. Chen, A. Eversole, G. Li, D. Yu, and F. Seide. Pipelined Back-propagation for Context-dependent Deep Neural Networks. In *Interspeech*, 2012.

[7] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pages 571–582, 2014.

[8] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *USENIX Annual Technical Conference*, pages 37–48, 2014.

[9] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.

[10] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[11] M. Denkowski and A. Lavie. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the ninth workshop on statistical machine translation*, pages 376–380, 2014.

[12] DGX-1. NVIDIA DGX-1. https://www.nvidia.com/en-us/data-center/dgx-1/.

[13] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011.

[14] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[15] S. Hadjis, C. Zhang, I. Mitliagkas, D. Iter, and C. Ré. Omnivore: An optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*, 2016.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

[17] 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. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012.

[18] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[19] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[21] N. Kalchbrenner, E. Grefenstette, and P. Blunsom. A convolutional neural network for modelling sentences. *CoRR*, abs/1404.2188, 2014. URL http://arxiv.org/abs/1404.2188.

[22] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *2014 IEEE Confer-*

ence on Computer Vision and Pattern Recognition, pages 1725–1732, June 2014.

[23] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing. STRADS: a distributed framework for scheduled model parallel machine learning. In *EuroSys*, pages 5:1–5:16, 2016.

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

[25] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[27] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing. On model parallelization and scheduling strategies for distributed machine learning. In *NIPS*, pages 2834–2842, 2014.

[28] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 1, page 3, 2014.

[29] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[30] A. Mirhoseini, H. Pham, Q. Le, M. Norouzi, S. Bengio, B. Steiner, Y. Zhou, N. Kumar, R. Larsen, and J. Dean. Device placement optimization with reinforcement learning. 2017. URL https://arxiv.org/abs/1706.04972.

[31] J. Močkus. On bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference*, pages 400–404. Springer, 1975.

[32] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[33] F. Seide and A. Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. URL https://github.com/Microsoft/CNTK.

[34] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. On parallelizability of stochastic gradient descent for speech dnns. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE SPS, May 2014.

[35] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[36] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[37] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[38] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 368–380. ACM, 2015.

[39] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[40] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2): 26–31, 2012.

[41] Uber Technologies Inc. Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow, 2017. URL https://eng.uber.com/horovod/.

[42] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), Aug. 1990.

[43] S. Venugopalan, M. Rohrbach, J. Donahue, R. Mooney, T. Darrell, and K. Saenko. Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542, 2015.

[44] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014. URL http://arxiv.org/abs/1411.4555.

[45] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T. Liu, and W. Ma. Lightlda: Big topic models on modest computer clusters. In *WWW*, pages 1351–1361. ACM, 2015.

[46] ZeroMQ. Distributed Messaging. http://zeromq.org/.

[47] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6.