# PipeDream: Pipeline Parallelism for DNN Training
## Extended Abstract

Aaron Harlap[§], Deepak Narayanan[†], Amar Phanishayee[*],
Vivek Seshadri[*], Gregory R. Ganger[§], Phillip B. Gibbons[§]

[*]Microsoft Research      [§]Carnegie Mellon University      [†]Stanford University

## 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, such as image and video classification, speech recognition, and language translation [4, 5, 8, 9, 12]. As DNNs have become more widely developed and used, model sizes have continued to grow – models today have tens to hundreds of layers totalling 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 to training DNNs in parallel is data parallelism, where the DNN model is replicated on multiple machines, with each replica training independently on a different subset of the training dataset. The weight updates computed on individual workers are then aggregated to get a final weight update that reflects updates across all inputs; the amount of data communicated after each aggregation is proportional to the size of the model. While data parallel training works well with some popular models [1, 3], expected growth in DNN model sizes increases the overhead of communication. Indeed, some widely-used models are large enough that the communication overheads dominate (e.g., up to 85% of training time for VGG-16 [11]) and limit scaling.

Another approach to parallelization involves splitting a single model among the different machines; this approach is used when the full model is too large to fit on a single machine. In this "model parallel" regime, each machine is responsible for a subset of the model's parameters, and performs weight updates on this partition alone. In addition, only inter-layer values (activations and gradients) are communicated synchronously among the different machines, often leading to significantly less communication overhead (e.g., up to 95%). However, as shown in Figure 1, standard model parallelism leads to severe under-utilization of compute resources. Furthermore, it is often not clear how to split a model among the different machines for optimal performance. To solve this problem, we propose *pipeline parallelism*, an enhancement to model-parallelism, where multiple mini-batches are injected into the system at once to ensure efficient and concurrent use of compute resources; communication overheads are hidden as they overlap with computation.

Our system, PipeDream, supports pipelined training, and automatically determines how to systematically split a given model across the available compute nodes. Experiments confirm PipeDream's effectiveness for large models. For example, when using 4 machines to train the >100 million parameter VGG16 [11] on the ImageNet 1K [10]
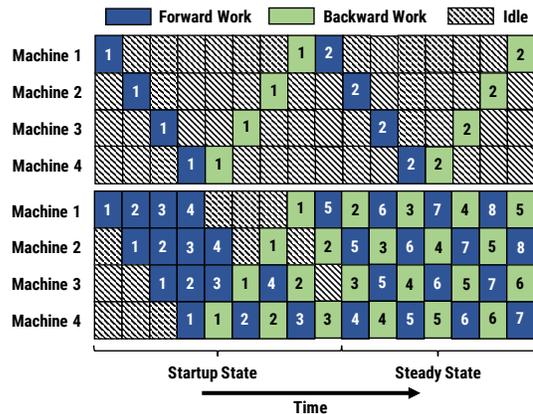


**Figure 1: Example pipelines with 4 machines, with (bottom) and without (top) pipelining enabled. The timeline with pipelining enabled shows startup and steady states. Numbers indicate ID of mini-batch being processed. This simplified example assumes that the forward and backward pass in every stage takes exactly one time unit (represented by a box), and that there are no communication delays. A single model's computation is split amongst the 4 machines.**

dataset, PipeDream converges 2.5× faster than using a single machine and 3× faster than data parallel training; the massive communication overheads make data parallel slower with 4 machines than using just a single machine. For the much smaller Inception-BN [7] network, PipeDream performs nearly identically to data parallelism, confirming that the pipeline parallel approach does not hurt even when communication is not the bottleneck.

## 2 OUR APPROACH: PIPELINING

Pipeline parallelism (PP) aims to avoid the synchronization bottleneck in data-parallel training (BSP). Our approach is inspired by pipelining in computer architecture, where execution units are organized into stages, and multiple instructions are in flight at a time.

*Overview.* PipeDream *automates* the process of partitioning the DNN layers among the different machines by profiling the per-layer compute times and inter-layer bandwidths for the entire model. With this information, PipeDream picks roughly even splits that minimize communication between compute nodes while balancing compute load across them, using a Dynamic Program to solve the optimization problem. Each split of the DNN is referred to as a stage. Each stage is mapped to a separate GPU, that performs the forward and backward pass for all the layers that are part of the 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*. In the forward phase, each stage performs the forward pass for each of the layers in that stage. In the backward phase, each stage performs the backward pass for each of the layers in that stage.
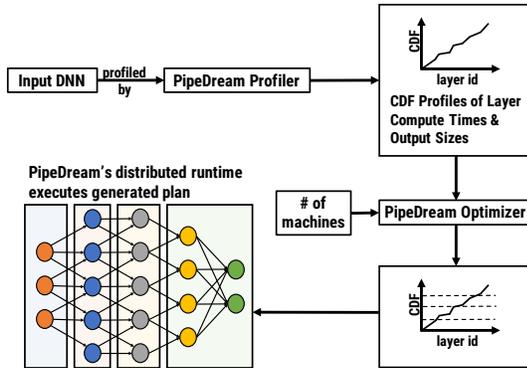
Figure 2: PipeDream's automated mechanism to partition DNN layers into stages. PipeDream first profiles the input DNN to get estimates for each layer's compute time and output size. Using these estimates, PipeDream's optimizer partitions the DNN's layers across the available machines.

To ensure that no GPU is idle at any point in time, we inject multiple mini-batches into the pipeline one after the other. After completing the forward pass for a mini-batch, each stage *asynchronously* sends the output activations to the next stage, while simultaneously starting to perform work for another mini-batch. Similarly, after completing the backward pass for a mini-batch, each stage *asynchronously* sends the output gradient to the previous stage, while starting computation for another mini-batch. This ensures that different GPUs are processing different mini-batches at the same time (Figure 1, bottom).

PipeDream's pipeline is bi-directional (Figure 1, bottom), since a layer's forward and backward pass needs to be performed on the same machine – this is different from instruction pipelines which are generally uni-directional. PipeDream's runtime needs to ensure that all machines in the pipeline are utilized, while also ensuring that both forward and backward passes for different inputs are performed. PipeDream also maintains multiple versions of the parameters at each machine (one for each mini-batch in progress), since we empirically observed that later layers seeing more updated parameters than earlier layers cause the model to converge slower.

*Advantages of PP over BSP.* There are two main reasons why PP is better than BSP for training huge DNN models. For DNNs with large models sizes, PP has far less communication overhead compared to BSP. 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 results in far less communication (e.g., >95% reduction for VGG16). In addition, PP overlaps the communication arising from each mini-batch with the computation for the next, avoiding network stalls, leading to better hardware efficiency.

## 3 EVALUATION

This section shows that for large models, PipeDream provides a 3–4× improvement over a standard data parallel approach, while significantly reducing communication (up to 95%).

*Setup.* We trained the VGG16 model on the ImageNet dataset, on 4 machines that have NVIDIA Titan X GPUs and 40 Gbps ethernet interfaces. We present results for four different configurations: (1) pipeline parallelism with four machines, (2) data parallelism with
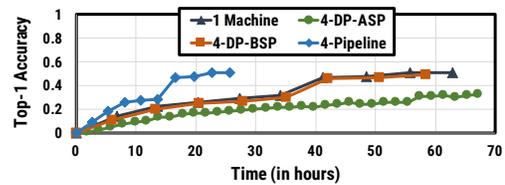


Figure 3: Accuracy versus time for a VGG16 model on the ImageNet dataset, using a mini-batch size of 32.

four machines using BSP synchronization, (3) data parallelism with four machines using ASP [2, 6] synchronization, and (4) a single machine. Each configuration, besides data-parallel ASP, was trained for nine epochs. After the fifth and seventh epoch, we reduced the learning rate by a magnitude of 10. We use a mini-batch size of 32 on each machine for each configuration, which is the largest mini-batch size that fits in GPU memory for VGG16.

*Results.* Figure 3 shows top-1 validation accuracy vs. training time for the four configurations. To evaluate performance, we compare the time taken to reach a certain validation accuracy. Compared to the single machine setup, pipeline-parallel achieves 50% validation accuracy 2.5x time faster. Compared to the 4-machine data-parallel with BSP setup, pipeline-parallel achieves 50% validation accuracy 3x faster. 4-machine data-parallel with BSP synchronization runs slower than a single machine, because of the large communication overheads (> 2× computation time) associated with VGG's large model size. In order to reduce communication overhead in data-parallel training, we experimented with running 4-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.* Due to ASP's poor statistical efficiency, pipeline parallel reaches a validation accuracy of 30% 4.3x faster than ASP data-parallel; ASP does not reach a validation accuracy of 50% after 35 epochs.

The single machine setup provides the best statistical efficiency, improving over both 4-machine data-parallel BSP and 4-machine pipeline-parallel. Both pipeline-parallel and data-parallel perform parameter updates on stale parameters, leading to a decrease in statistical efficiency. 4-machine data-parallel ASP has even worse statistical efficiency, since parameter updates can be performed on arbitrarily stale parameters. However, 4-machine data-parallel ASP provides the best hardware efficiency as it doesn't have any communication overhead, and also parallelizes processing across machines. Pipeline-parallel does have some overhead due to a slight imbalance in work distribution, but it is smaller than the communication overhead in data-parallel BSP, leading to a 2.47x faster time-per-epoch.

In addition, we observed that the amount of communication each machine performs in the pipelined setup is 0.7%–6.1% of the communication that each machine performs in a data-parallel setup.

## 4 FUTURE WORK

While PipeDream shows speedups for a small number of GPUs, it does increase the memory footprint of training as it needs to maintain previous versions of the parameters. As future work, we are exploring how to combine pipeline parallelism with data parallelism to support better scaling for a larger number of GPUs. We are also exploring how to apply pipeline parallelism to RNNs, which tend to scale poorly using existing data parallel approaches.

## REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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 Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, June 2014.

[10] 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.

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

[12] 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.