COMS E6998 010

Practical Deep Learning Systems Performance

Lecture 5 10/15/20

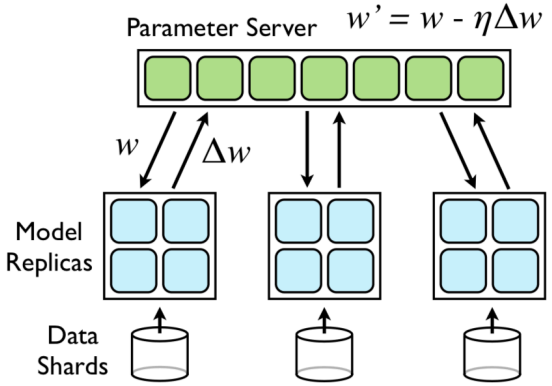
Logistics

- Homework 2 due Thursday Oct 15 by 11:59 PM
- Quiz 2 coming this weekend. Open Sat Oct 17 till Mon Oct 19.
  - Include material from last three lectures (including today)
- Seminar details due 10/16/2020: list of papers and team members
- Project proposals due 10/29/2020
  - Template posted
  - Project rubric posted
- Set-up your project meetings with me: schedule sign-up sheet on Collaborations

Recall from last lecture

- Gradient descent with Momentum and Nesterov momentum
- Single node, single and multi GPU training and its scaling
- Distributed training, model and data parallelism
- Centralized aggregation: Parameter server
- Synchronous SGD variants and straggler problem
- Asynchronous SGD variants and stale gradients problem
- Staleness dependent learning rate
- Decentralized aggregation: P2P

Downpour SGD



Downpour SGD (from Google, 2012)
Each model replica is a DistBelief model, Google's proprietory ML framework

- Model parallelism
- Automatically parallelizes computation in each machine using all available cores

Model and Data parallelism
Parameter server is sharded

- Different PS shards update their parameters asynchronously, they do not need to communicate among themselves
- Different model replicas (Distbelief models) run independently
- Model replicas are permitted to fetch parameters and push gradients in separate threads
- Tolerates variances in the processing speed of different model replicas
- Tolerates failure of model replicas

Downpour SGD: Sources of Stochasticity

- Model replicas will most likely be using “stale” parameters when calculating gradients locally
- No guarantee that at any given moment the parameters on each shard of the parameter server have undergone the same number of updates
  - Order of updates may be different at different PS shards
- Adagrad learning rate improves robustness of Downpour SGD

$$\eta_{i,k} = \frac{\gamma}{\sqrt{\sum_{j=1}^{k} \Delta w_{i,j}^2}}$$

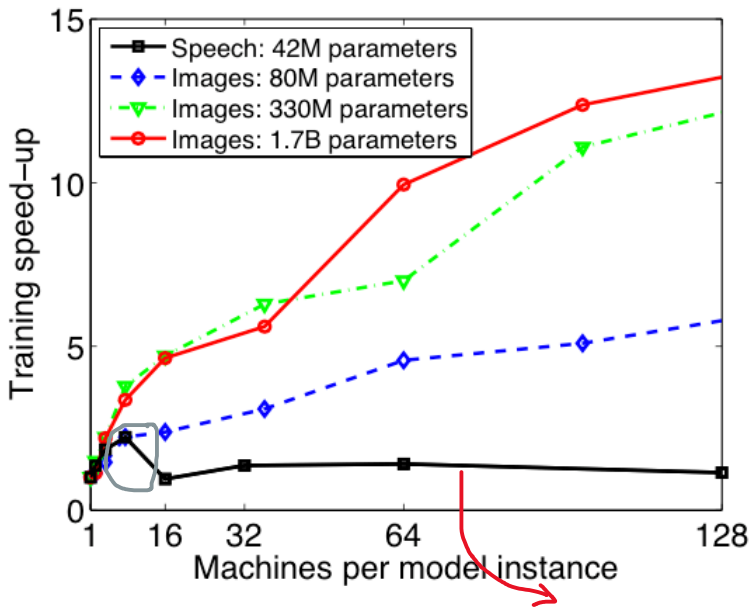
$\eta_{i,k}$: learning rate of the ith parameter at iteration k
$\Delta w_{i,k}$ is its gradient
γ is the constant scaling factor

- Adagrad implemented locally within each parameter server shard

Adagrad

- Parameter specific learning-rate
- Effective learning rate `decreases very fast because of gradient accumulation from the beginning of training
- Adagrad works well for sparse data

Scaling with Model Parallelism



Average training speed-up: the ratio of the time taken per iteration using only a single machine to the time taken using N partitions (machines)

Benefit diminishes when adding more machines:

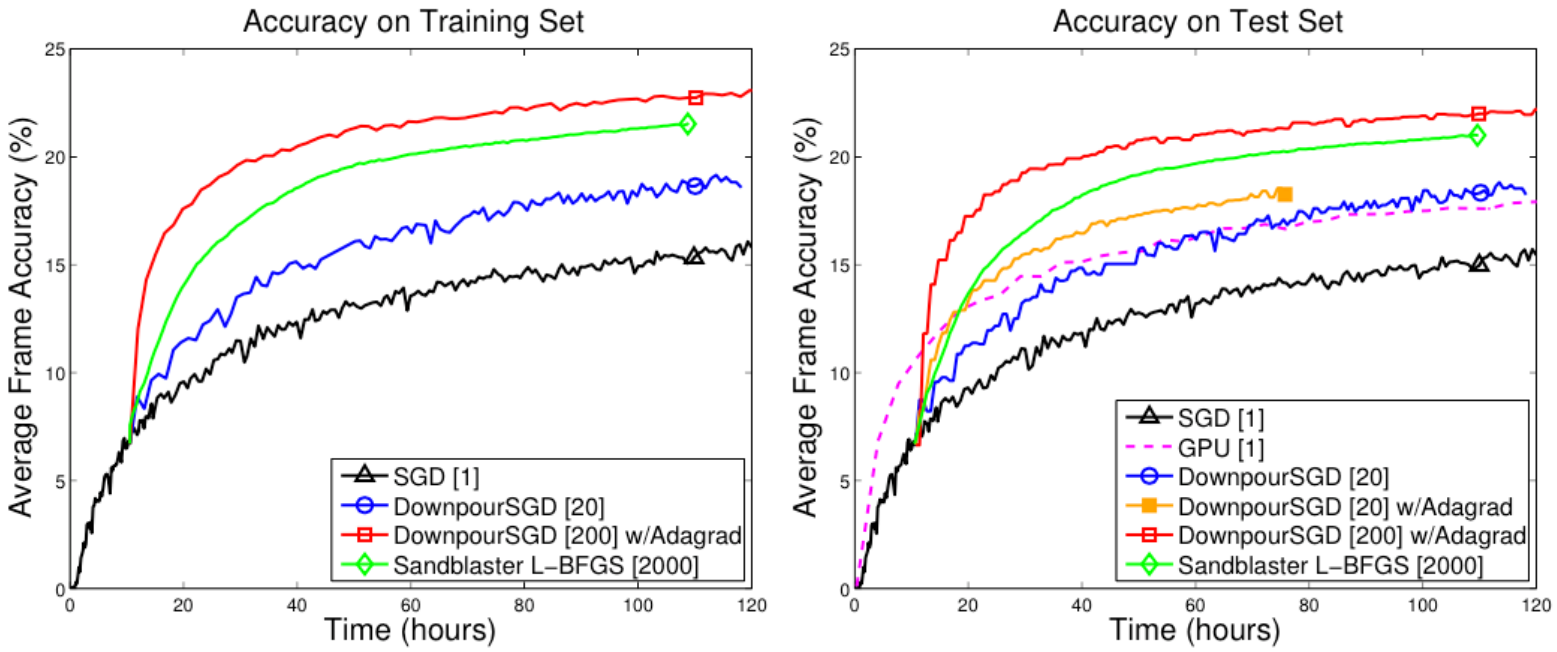
- Less work per machine
- Increased communication between machines

Speed-up with model parallelism

- Fully connected speech model with 42M parameters: 2.2x 8 machines
- Locally connected image model with 1.7B parameters: 12x with 81 machines

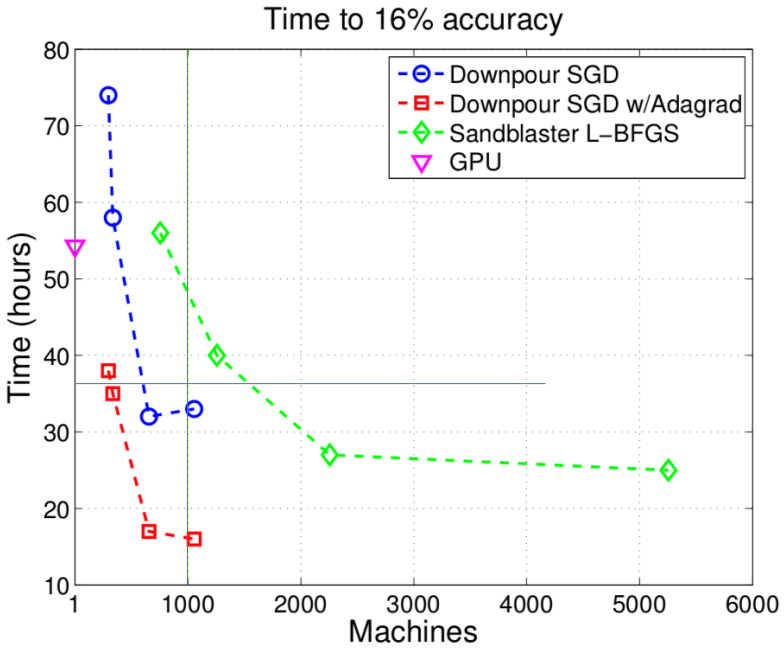
Less work per machine, network overhead dominates

Scaling with Downpour SGD



Each model replica is partitioned on 8 machines

Runtime-Resource Tradeoff in Downpour SGD



Points closer to the origin are preferable in that they take less time while using fewer resources.
Which gives the best tradeoff ?

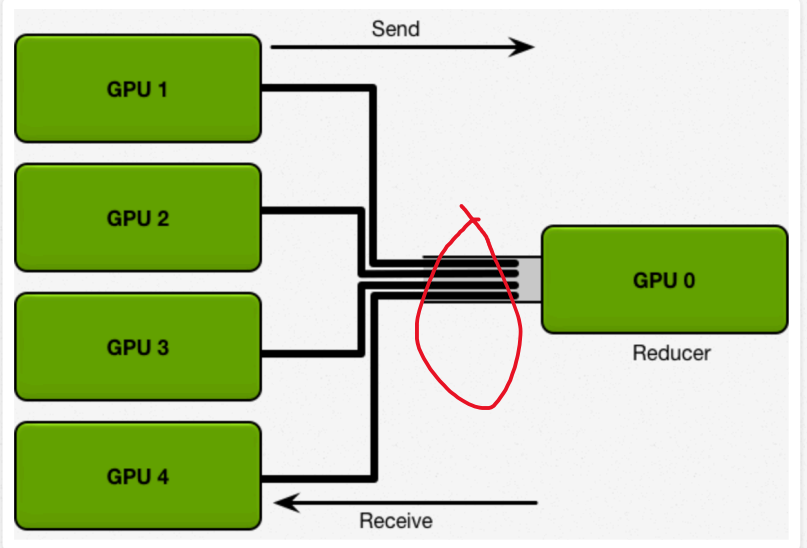
Reduction over Gradients

- To synchronize gradients of N learners, a reduction operation needs to be performed

$$\sum_{j=1}^{N} \Delta w_j$$

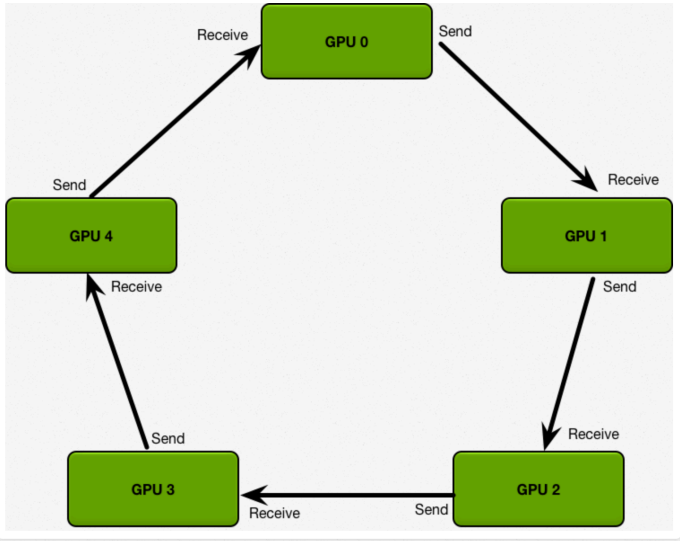
Reduction Topologies

Parameter server: single reducer



SUM (Reduce operation) performed at PS

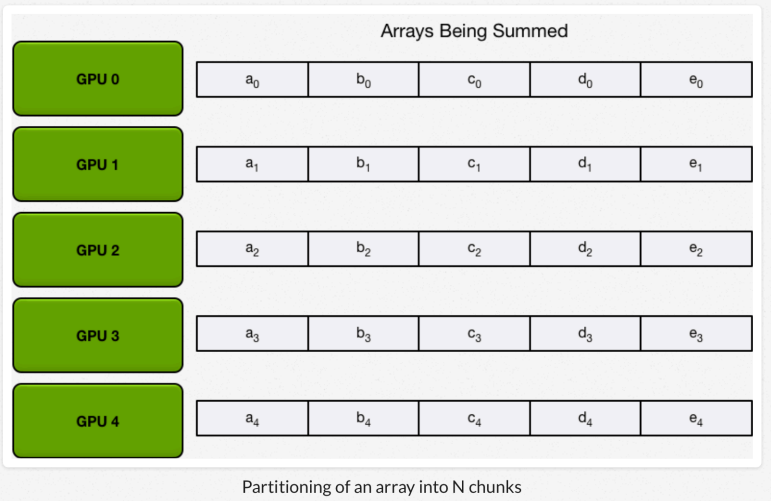
GPUs arranged in a logical Ring (aka bucket) :
all are reducers



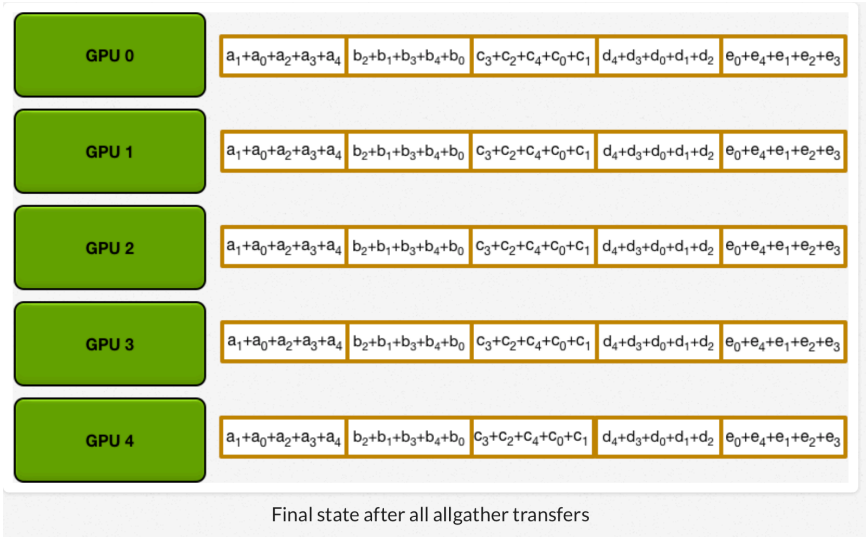
SUM (Reduce operation) performed at all nodes

- Each node has a left neighbor and a right neighbor
- Node only sends data to its right neighbor, and only receives data from its left neighbor

Example Problem



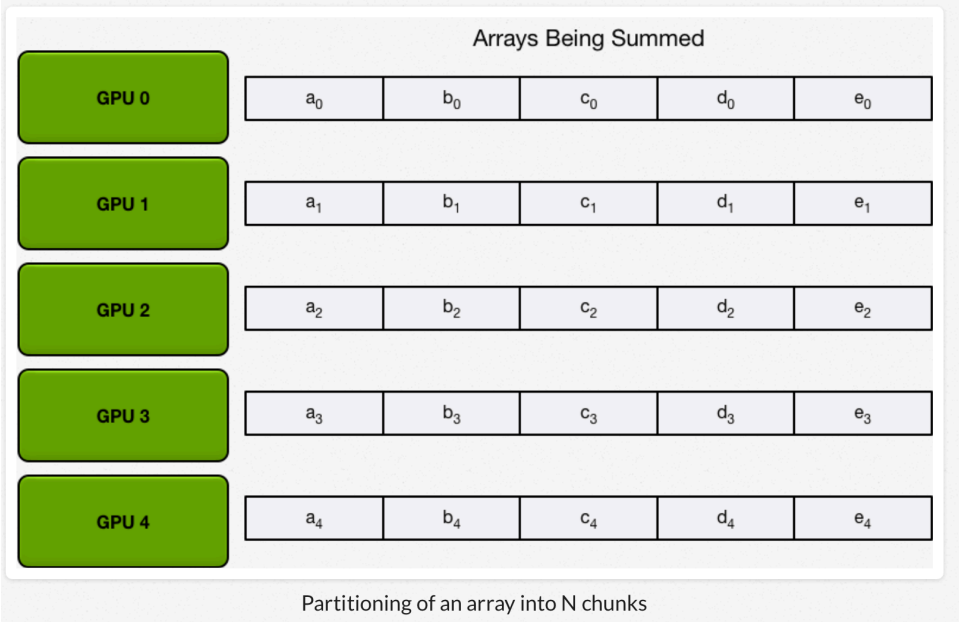
Partitioning of an array into N chunks



Final state after all allgather transfers

Ring All-Reduce

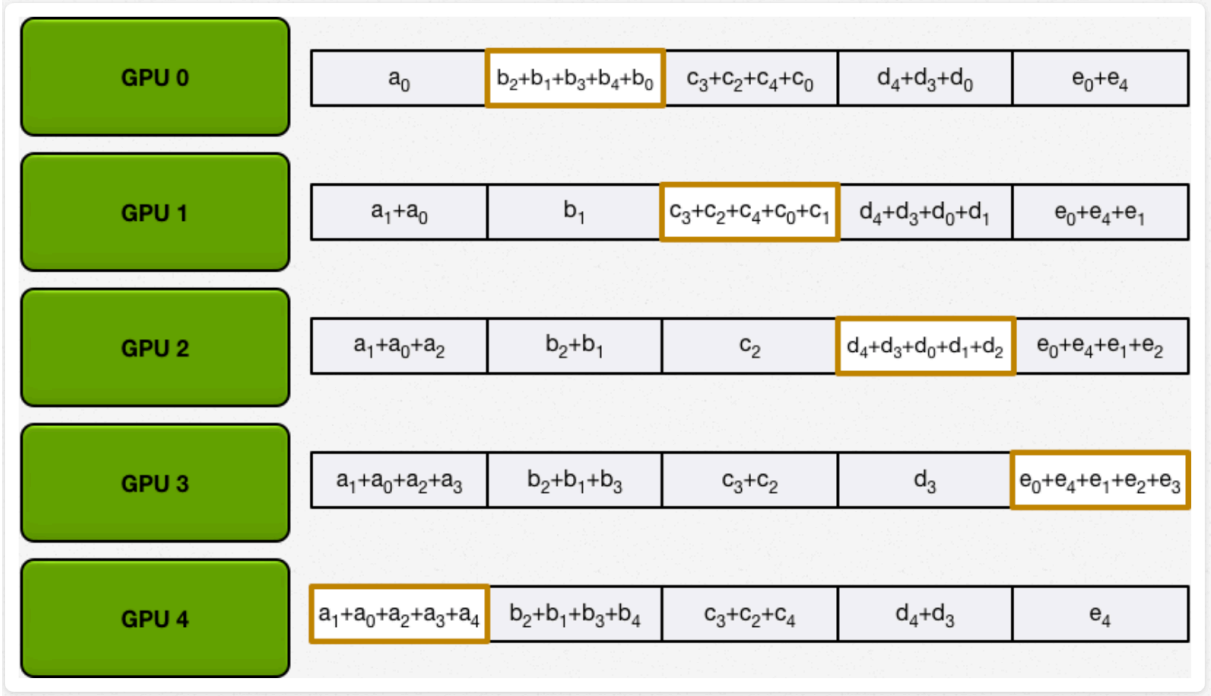
- Two step algorithm:
  - Scatter-reduce
    - GPUs exchange data such that every GPU ends up with a chunk of the final result
  - Allgather
    - GPUs exchange chunks from scatter-reduce such that all GPUs end up with the complete final result.



Partitioning of an array into N chunks

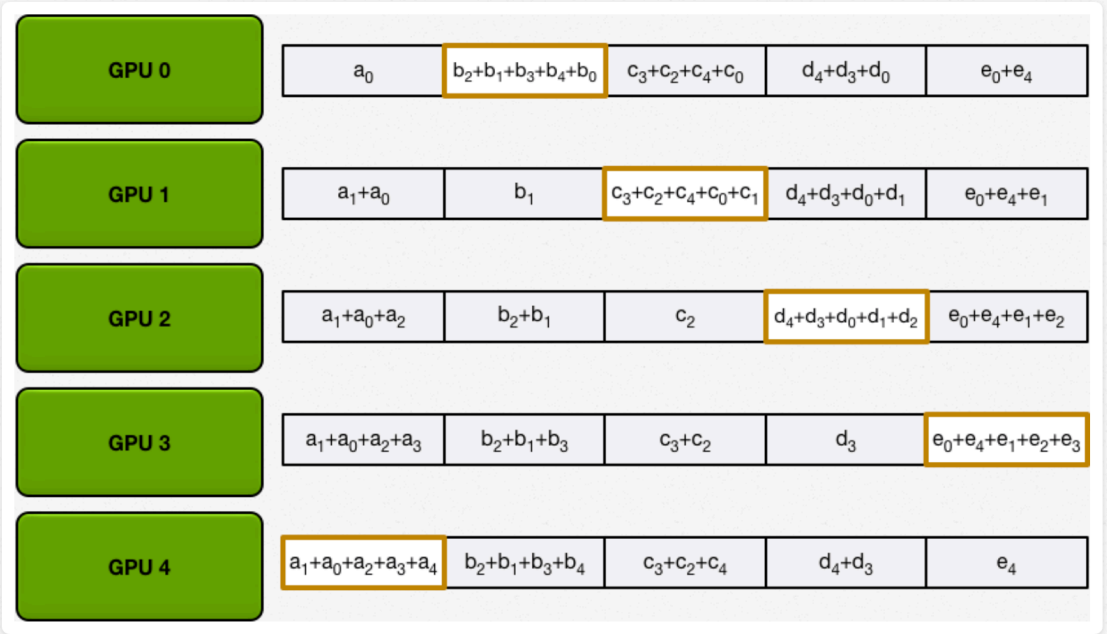
Ring All-Reduce: Scatter-Reduce Step

Ring All-Reduce: End of Scatter-Reduce Step



Final state after all scatter-reduce transfers

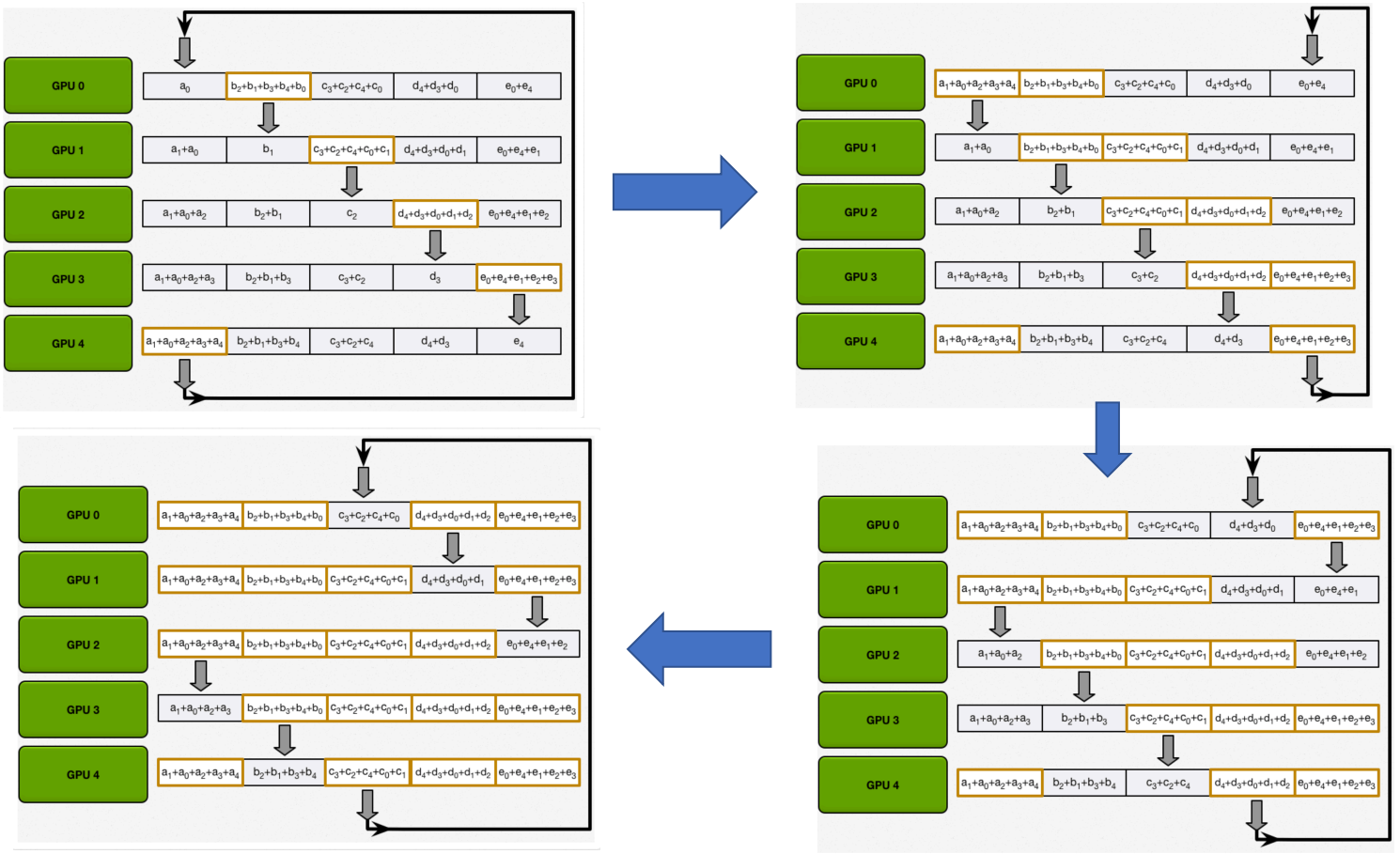
Ring All-Reduce: What to do next ?



Final state after all scatter-reduce transfers

GPU	Send	Receive
0	Chunk 1	Chunk 0
1	Chunk 2	Chunk 1
2	Chunk 3	Chunk 2
3	Chunk 4	Chunk 3
4	Chunk 0	Chunk 4

Ring All-Reduce: AllGather Step



Ring All-Reduce: End of AllGather Step

Final state after all allgather transfers

Parameter Server (PS) vs Ring All-Reduce: Communication Cost

- P: number of processes N: total number of model parameters
- PS (centralized reduce)
  - Amount of data sent to PS by (P-1) learner processes: N(P-1)
  - After reduce, PS sends back updated parameters to each learner
  - Amount of data sent by PS to learners: N(P-1)
  - Total communication cost at PS process is proportional to 2N(P-1)
- Ring All-Reduce (decentralized reduce)
  - Scatter-reduce: Each process sends N/P amount of data to (P-1) learners
    - Total amount sent (per process): N(P-1)/P
  - AllGather: Each process again sends N/P amount of data to (P-1) learners
  - Total communication cost per process is 2N(P-1)/P
- PS communication cost is proportional to P whereas ring all-reduce cost is practically independent of P for large P (ratio (P-1)/P tends to 1 for large P)
- Note that both PS and Ring all-reduce involve synchronous parameter updates

All-Reduce applied to Deep Learning

- Backpropagation computes gradients starting from the output layer and moving towards in the input layer
- Gradients for output layers are available earlier than inner layers
- Start all reduce on the output layer parameters while other gradients are being computed
- Overlay of communication and local compute

Scaling using compute-communication overlap in all-reduce

- http://andrew.gibiansky.com

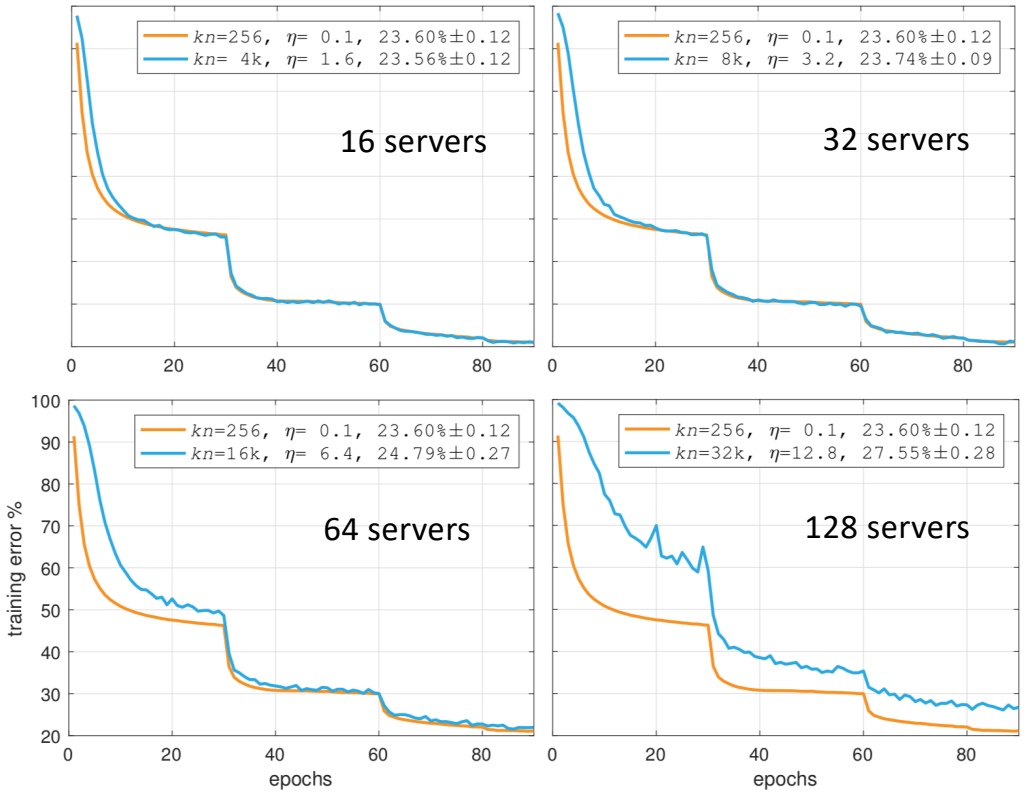
Distributed Scaling Speed up -- Facebook

- Demonstrated training of a ResNet-50 network in one hour on 256 GPUs
- Major techniques:
  - Data parallelism
  - Very large batch sizes
  - Learning rate adjustment technique to deal with large batches
  - Gradual warm up of learning rate from low to high value in 5 epochs ?
  - MPI_AllReduce for communication between machines
  - NCCL communication library between GPUs on a machine connected using NVLink
  - Gradient aggregation performed *in parallel* with backprop
    - No data dependency between gradients across layers ?
    - As soon as the gradient for a layer is computed, it is aggregated across workers, while gradient computation for the next layer continues

Imagenet Distributed Training

- Each server contains 8 NVIDIA Tesla P100 GPUs interconnected with NVIDIA NVLink
- 32 servers => 256 P100 GPUs
- 50 Gbit Ethernet ?
- Dataset: 1000-way ImageNet classification task; ~1.28 million training images; 50,000 validation images; top- 1 error
- ResNet-50
- Learning rate: $\eta = 0.1 \cdot \frac{kn}{256}$ **How was this chosen ? What is k and n ?**
  Is it batch size per GPU (k) and total number of GPUs (n) ?
- Mini-batch size per GPU: 32 (fixed, weak scaling across servers)
- Caffe2 **What is Caffe2 ?**

Training Error Vs Batch Size: Distributed Training



Beyond 8k batch size, the training error deteriorates

Runtime Scaling

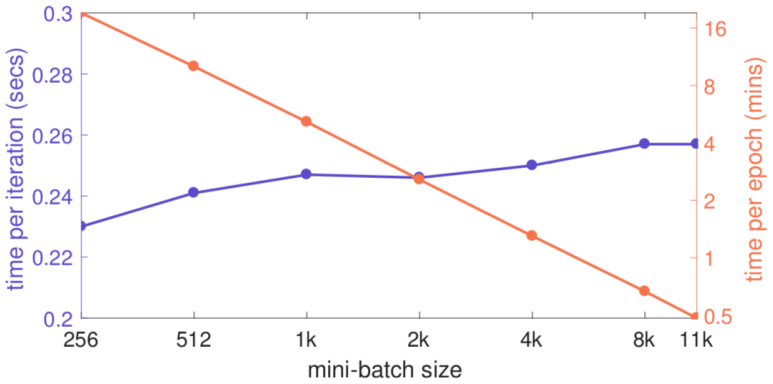


Figure 7. **Distributed synchronous SGD timing.** Time per iteration (seconds) and time per ImageNet epoch (minutes) for training with different minibatch sizes. The baseline ($kn = 256$) uses 8 GPUs in a single server , while all other training runs distribute training over ($kn/256$) server. With 352 GPUs (44 servers) our implementation completes one pass over all ~1.28 million ImageNet training images in about 30 seconds.

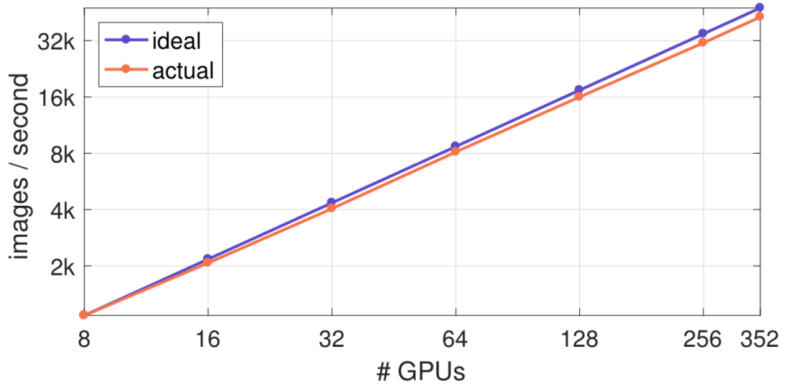


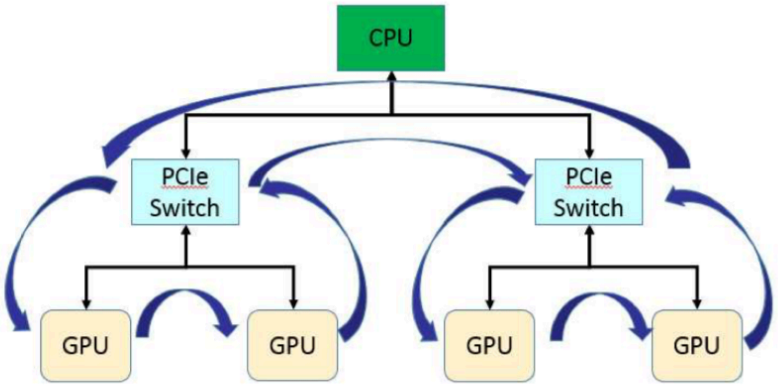
Figure 8. **Distributed synchronous SGD throughput.** The small overhead when moving from a single server with 8 GPUs to multi-server distributed training (Figure 7, blue curve) results in linear throughput scaling that is marginally below ideal scaling (~90% efficiency). Most of the allreduce communication time is hidden by pipelining allreduce operations with gradient computation. Moreover, this is achieved with commodity Ethernet hardware.

Questions

- Why time per iteration increases little with increasing minibatch size ?
- Why time per epoch decreases with increasing batch size ?
- With 44 servers how much time it takes to finish 1 epoch of training ?
- Can you get throughput (images/sec) from time per epoch ? Do you need to know batch size ?
- Can you get throughput (images/sec) from time to process a mini-batch (iteration) ? Do you need to know batch size ?
- If K-batch sync or K-sync (K < number of servers) was applied would the convergence been faster ?

PowerAI DDL --IBM

Communication ring over network tree topology



All nodes communicate concurrently with one other node

- 64 IBM Power8 servers
- Each server has 4 NVidia Tesla P100 GPUs connected through NVLink
- Batch size: 32 per GPU
- Effective batch size: 8192
- Infiniband
- IBM-Caffe and optimized Torch

PowerAI DDL Library

- Based on MPI
- Object is a tensor of gradients along with metadata (host, device etc.)

Distributed Deep Learning Benchmarking Methodology

- Scaling efficiency
- Accuracy and end-to-end training time
- Neural network
- Deep learning framework
- GPU type
- Communication overhead

Scaling efficiency

- Scaling efficiency: ratio between the run time of one iteration on a single GPU and the run time of one iteration when distributed over N GPUs.
- One can satisfy any given scaling efficiency for any neural network by increasing the batch size
- Too big a batch size will result in converging to an unacceptable accuracy or no convergence at all
- A high scaling efficiency without being backed up by convergence to a good accuracy and end to end training time is meaningless

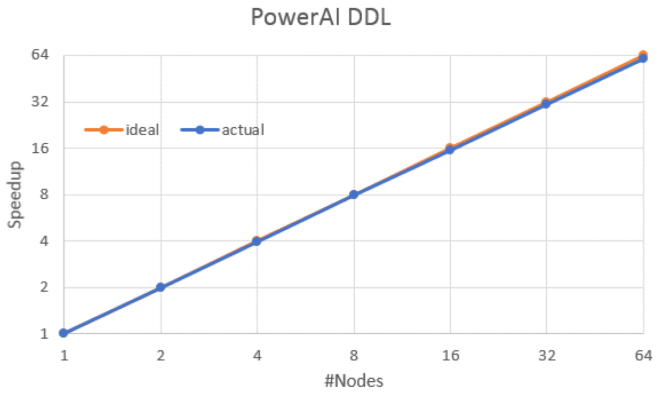
Other considerations

- Systems under comparison should train to same accuracy
- Accuracy should be reported on sufficiently large test set
- Compute to communication ratio can vary widely for different neural networks. Using a neural network with high compute to communication ratio can hide the ills of an inferior distributed Deep Learning system.
  - a sub-optimal communication algorithm or low bandwidth interconnect will not matter that much
- Computation time for one Deep Learning iteration can vary by up to 50% when different Deep Learning frameworks are being used. This increases the compute to communication ratio and gives the inferior distributed Deep Learning system an unfair uplift to the scaling efficiency.

Other considerations

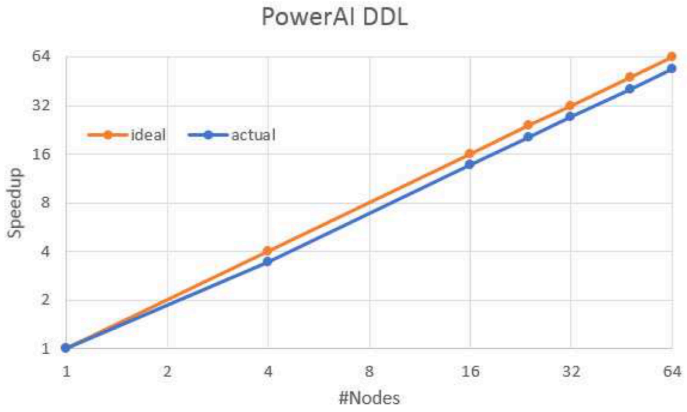
- A slower GPU increases the compute to communication ratio and again gives the inferior distributed Deep Learning system an unfair uplift to the scaling efficiency.
  - Nvidia P100 GPUs are approximately 3X faster than Nvidia K40 GPUs.
  - When evaluating the communication algorithm and the interconnect capability of a Deep Learning system, it is important to use a high performance GPU.
- Communication overhead is the run time of one iteration when distributed over N GPUs minus the run time of one iteration on a single GPU.
  - Includes the communication latency and the time it takes to send the message (gradients) among the GPUs.
  - Communication overhead gives an indication of the quality of the communication algorithm and the interconnect bandwidth.

PowerAI DDL --IBM



#GPUs	4	8	16	32	64	128	256
#Nodes	1	2	4	8	16	32	64
Speedup	1.0	2.0	3.9	7.9	15.5	30.5	60.6
Scaling efficiency	1.00	1.00	0.98	0.99	0.97	0.95	0.95

Figure 2: Resnet-50 for 1K classes using up to 256 GPUs with Caffe.



#GPUs	4	16	64	96	128	192	256
#Nodes	1	4	16	24	32	48	64
Speedup	1.0	3.5	13.7	20.5	27.2	40.3	53.7
Scaling efficiency	1.00	0.86	0.86	0.85	0.85	0.84	0.84

Figure 3: Resnet-50 for 1K classes using up to 256 GPUs with Torch.

- In Torch implementation, the reduction of gradients is not overlapped with compute; scaling efficiency is 84% vs 95% with Caffe

Imagenet1K/ResNet50 Training at Scale

Work	Batch size	Processor	DL Library	Interconnect	Training Time	Top-1 Accuracy	Scaling Efficiency
He et al	256	Tesla P100 x8	Caffe		29 hrs	75.3%	
Goyal et al (Facebook)	8K	Tesla P100 x256	Caffe2	50 Gbit Ethernet	60 mins	76.3%	~90%
Cho et al (IBM)	8K	Tesla P100 x256	Caffe	Infiniband	50 mins	75.01%	95%
Smith et al	8K → 16K	Full TPU Pod	Tensorflow		30 mins	76.1%	
Akiba et al	32K	Tesla P100 x1024	Chainer	Infiniband FDR	15 mins	74.9%	80%
Jia et al	64K	Tesla P40 x2048	Tensorflow	100 Gbit Ethernet	6.6 mins	75.8%	87.9%
Ying et al	32K	TPU v3 x1024	Tensorflow		2.2 mins	76.3%	
Ying et al	64K	TPU v3 x1024	Tensorflow		1.8 mins	75.2%	
Mikami et al	54K	Tesla V100 x3456	NNL	Infiniband EDR x2	2.0 mins	75.29%	84.75%

Cho et al achieved highest scaling efficiency; Goyal et al and Ying et al achieved highest accuracy

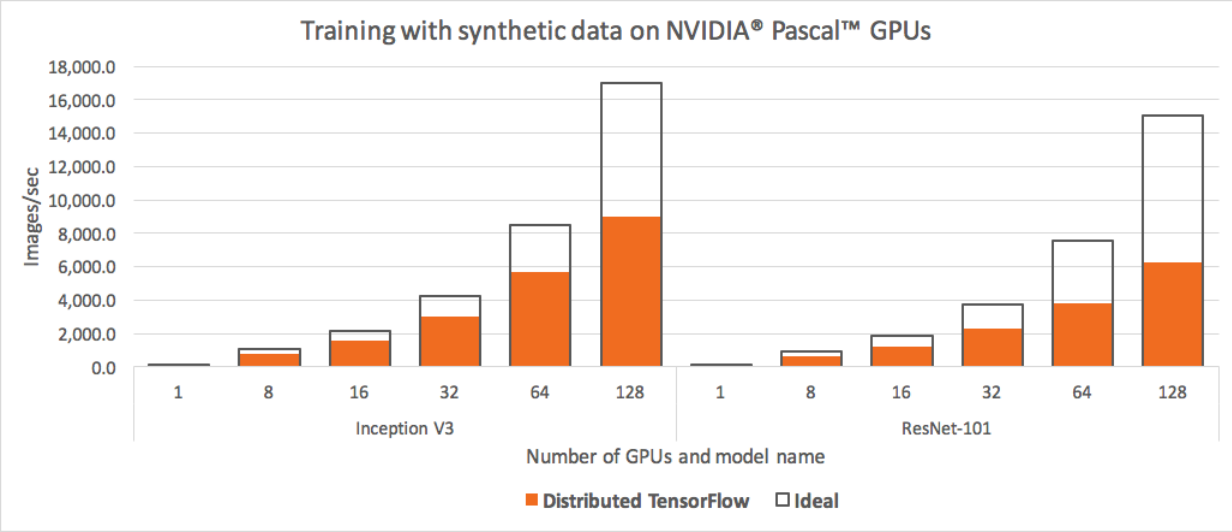
Meaningful Distributed DL System Comparison

- A popular neural network that has been widely trained and tuned
- Use a Deep Learning framework that is computationally-efficient
- Train to best accuracy on high performance GPUs
- Report:
  - Accuracy achieved
  - Training time to attain the accuracy
  - Scaling efficiency
  - Communication overhead
- Metric may also include power and cost.

Horovod – LF AI

- *Distributed training framework for TensorFlow, Keras, PyTorch, and MXNet. Goal is to make distributed Deep Learning fast and easy to use.*
- *https://github.com/horovod/horovod#why-not-traditional-distributed-tensorflow*
- Horovod Vs Distributed Tensorflow:
  - Requires much less code changes to run a Single-GPU Tensorflow program in a distributed manner across many GPUs https://eng.uber.com/horovod/
  - Horovod is faster than PS based distributed Tensorflow
    - In benchmarking Horovod against standard distributed TensorFlow, Uber has observed large improvements in its ability to scale, with Horovod coming in roughly twice as fast.

Distributed Tensorflow Benchmarking -- Uber



- Uses PS for aggregation
- About half of GPU resources are idle due to scaling inefficiencies when training on 128 GPUs.

Horovod Benchmarking – Uber

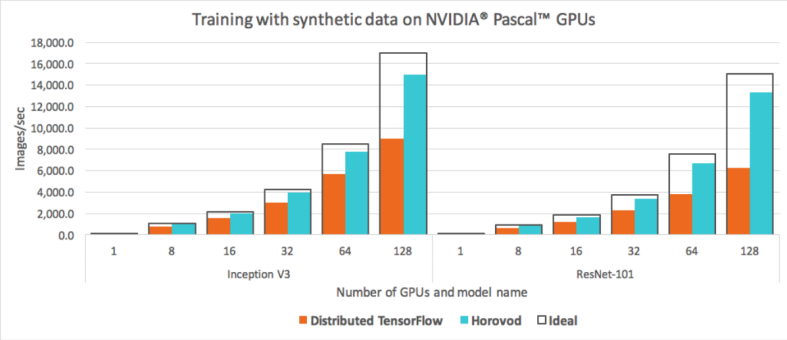


Figure 6: A comparison of images processed per second with standard distributed TensorFlow and Horovod when running a distributed training job over different numbers of NVIDIA Pascal GPUs for Inception V3 and ResNet-101 TensorFlow models over 25GbE TCP.

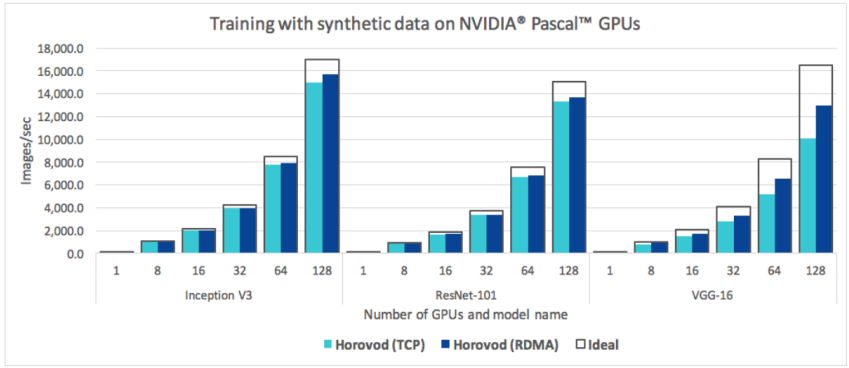


Figure 7: A comparison of the images processed per second of the Horovod over plain 25GbE TCP and the Horovod with 25GbE RDMA-capable networking when running a distributed training job over different numbers of NVIDIA Pascal GPUs for Inception V3, ResNet-101 and VGG-16.

- ~88% scaling efficiency using both Inception V3 and ResNet-101
- Training was ~2x faster than standard distributed TensorFlow

- Only 2-3% increase over TCP networking with RDMA for Inception V3 and ResNet-101
- VGG-16 model experienced a significant 30 percent speedup
  - VGG-16 has higher number of model parameters

Key Factors in Scalable DL Training

- Efficient compute and communication libraries
  - Intra GPU and inter machine communication topology
  - Overlap of compute and communication
- Framework support: Tensorflow, Pytorch, Keras, MxNet
- Ease of enablement: Minimum code changes to enable distribution of a single node training job
- Specific framework level optimizations for distribution
- Data storage
- Network bandwidth

Parameter and Gradient Compression

- Look at paper by N. Storm Scalable distributed dnn training using commodity gpu cloud computing 2015
- Look at Sec. 7.3 of paper by Ben-Nun et al. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis 2018
  - Quantization
  - Sparsification
  - Other techniques

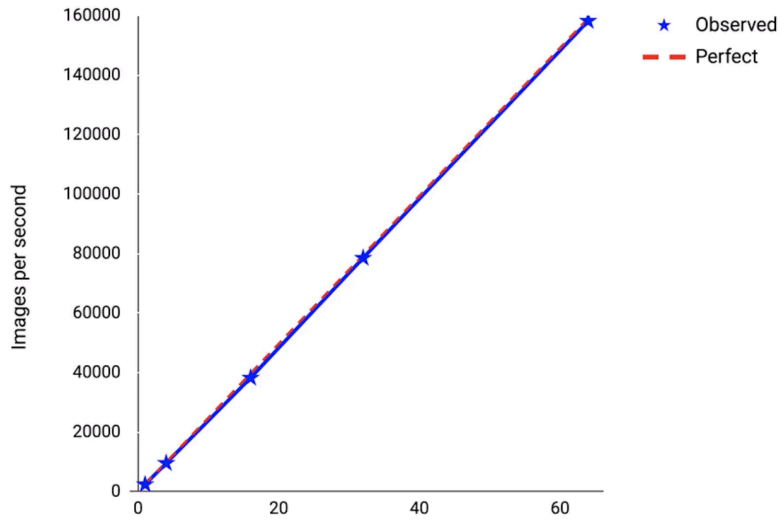
Tensorflow

- Go over Eager execution colab tutorial for Tf 2.0
  - https://www.tensorflow.org/tutorials/quickstart/advanced
- Distribution support in Tensorflow
  - tf.distribute.Strategy: TensorFlow API to distribute training across multiple GPUs, multiple machines or TPUs
  - https://www.tensorflow.org/guide/distributed_training
- Distributed training example using keras
  - https://www.tensorflow.org/tutorials/distribute/keras
- Tensorboard: TensorFlow's visualization toolkit

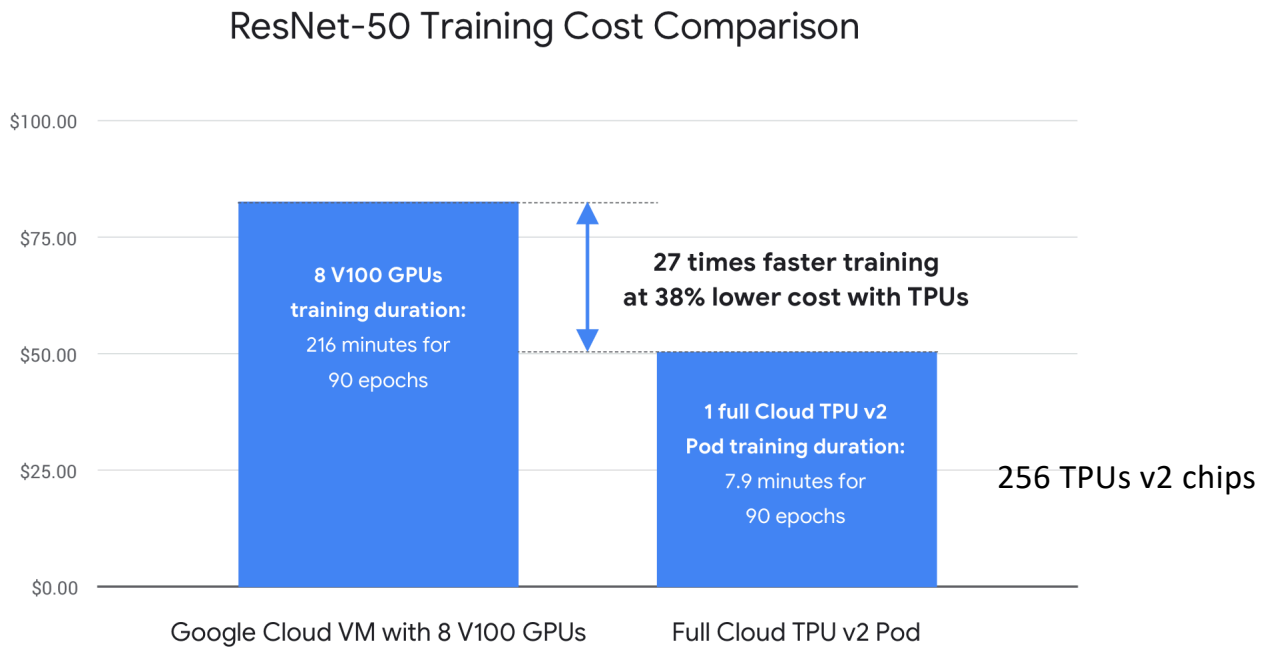
Tensor Processing Units (TPUs)

- TPU: domain specific architecture for Deep Learning
- TPU pod: multiple **TPU** devices connected to each other over a dedicated high-speed network connection
- Google Cloud Demo: Tensor Processing Unit
- Tutorial on using Google Cloud TPU to run MNIST: Google Cloud Quickstart

TPU v2 pod for ResNet-50: linearly scalable



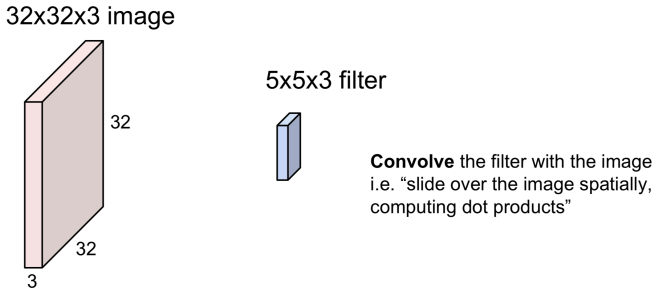
TPUs vs GPUs Performance

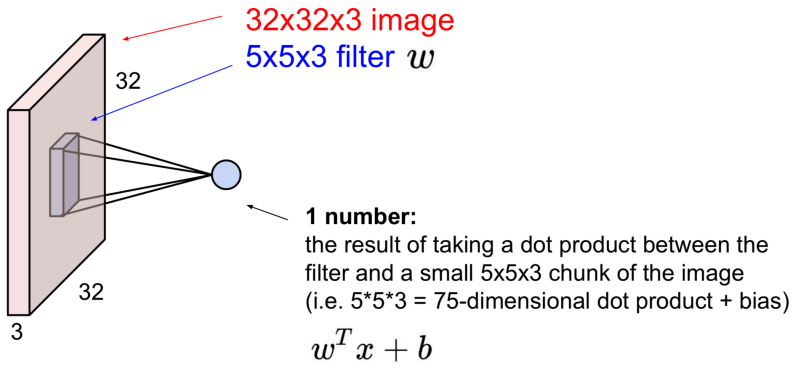


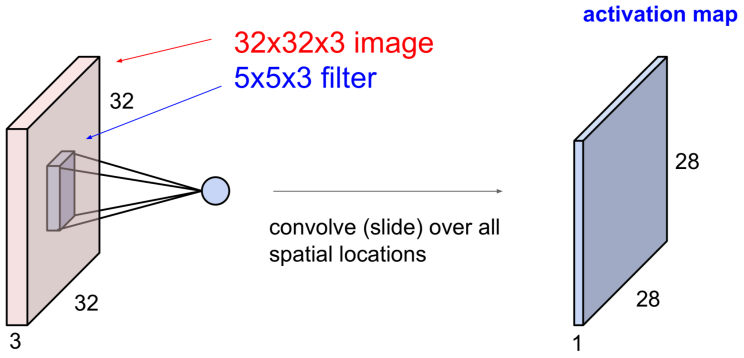
Convolutional Neural Networks (CNNs)

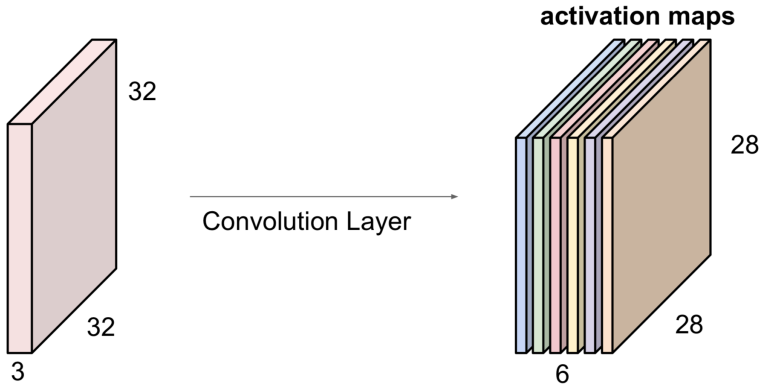
- Widely used in computer vision
- Specialized structure: inspired by Hubel and Wiesel experiments on visual cortexes of cat and monkey
- Success in ImageNet (ILSVRC) competitions brought them into limelight since 2012
- Layers have *volume*: length, width, depth
- The depth is 3 for color images, 1 for grayscale, and an arbitrary value for hidden layers.
- Three basic operations in CNNs are convolution, pooling, and ReLU.
  - The convolution operation is analogous to the matrix multiplication in a conventional network

Convolution Layer



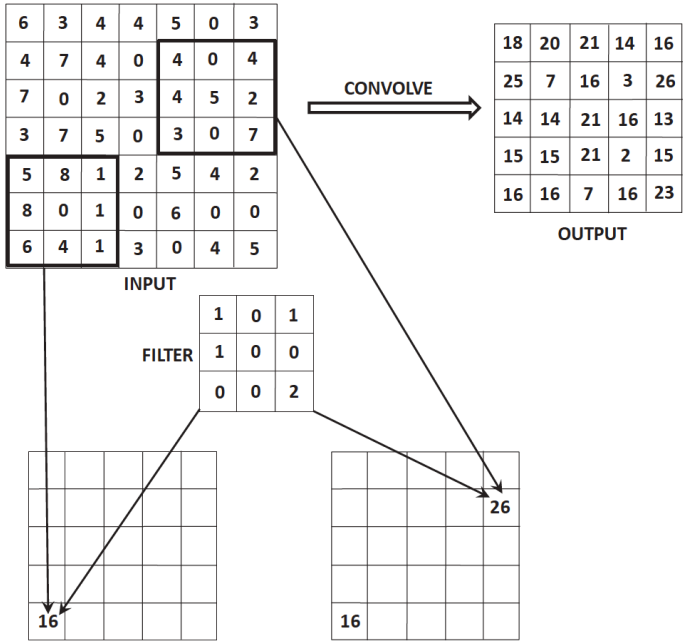






Multiple filters create depth in output volume

Example of Convolution



- Add up over multiple activations maps from the depth

Convolution Layer: Size of output

$L_q \times B_q \times d_q$

$F_q \times F_q \times d_q$

$(L_q - F_q + 1) \times (B_q - F_q + 1)$

with stride S_q

$L_q \times B_q \times d_q$

$F_q \times F_q \times d_q$

$(L_q - F_q)/S_q + 1 \times (B_q - F_q)/S_q + 1$

When a stride of S_q is used in the qth layer, the convolution is performed at the locations 1, $S_q + 1$, $2\,S_q + 1$, and so on along both spatial dimensions of the layer.

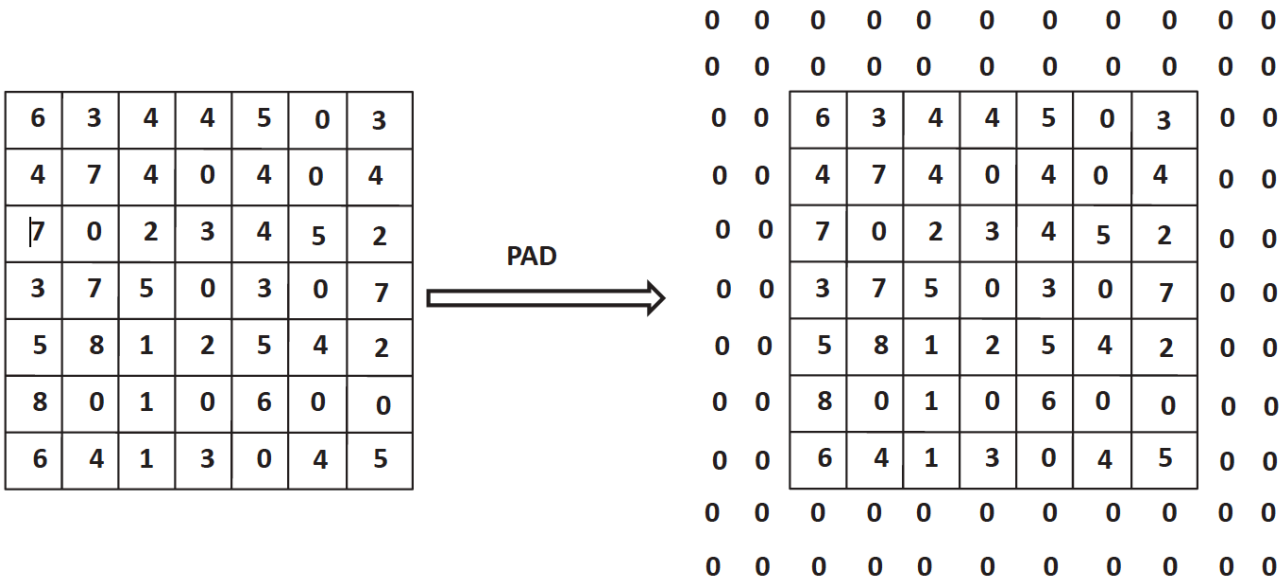
Convolution Neural Networks Properties

- Sparse connectivity because we are creating a feature from a region in the input volume of the size of the filter (usually much smaller than the input size)
  - Each neuron in the output layer has connectivity to a small subset of neurons in the input layer which are spatially close
- Shared weights because we use the same filter across entire spatial volume
  - Interpret a shape in various parts of the image in the same way
  - Reduces the number of parameters to learn
- A feature in a hidden layer captures some properties of a region of the input image.

Padding

- Convolution results in loss of information
  - Size of the output layer is smaller than the size of the input layer
  - Loss of information along the borders of the image (or of the feature map, in the case of hidden layers).
- By adding (Fq − 1)/2 “pixels” all around the borders of the feature map, one can maintain the size of the spatial image by taking stride =1
- Padding can be of any size up to Fq-1
- What happens if you add a padding of size Fq or greater ?

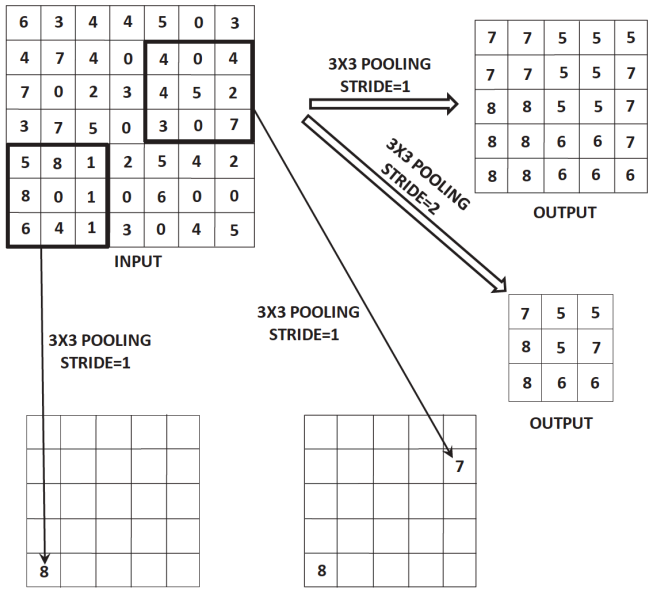
Padding example



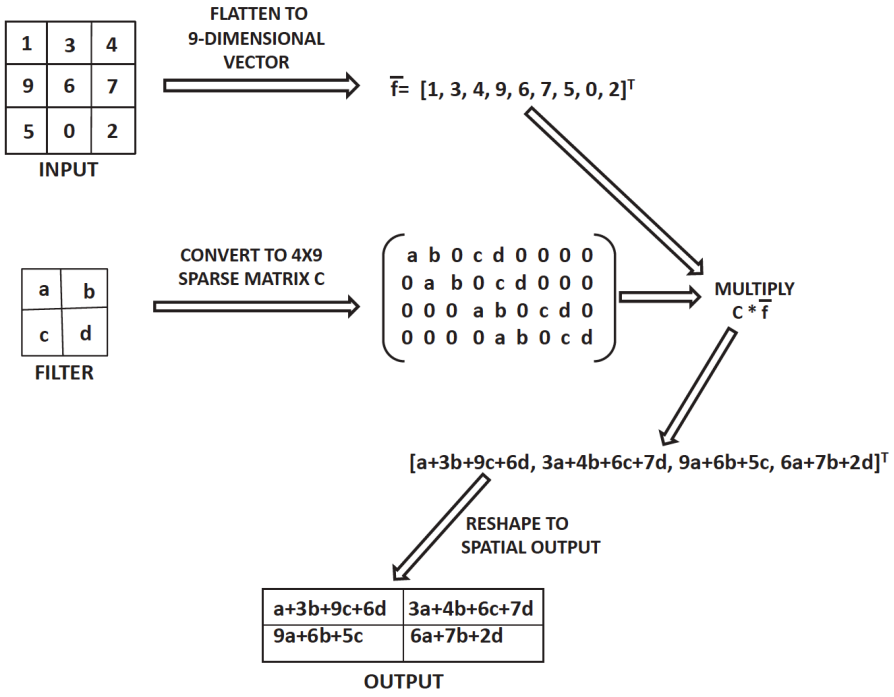
- A padding of size P increases both the length and width of input by 2P

Max Pooling

- Pooling operates on square regions of size P x P in each of the activation maps of the input and returns maximum of the values in those regions
- Pooling preserves the depth; depth of output is same as input
- Pooling is a subsampling technique; Greatly reduces the size of output activation maps and hence the number of parameters to learn in subsequent layers



Convolution as a Matrix Operation



Convolution Summary

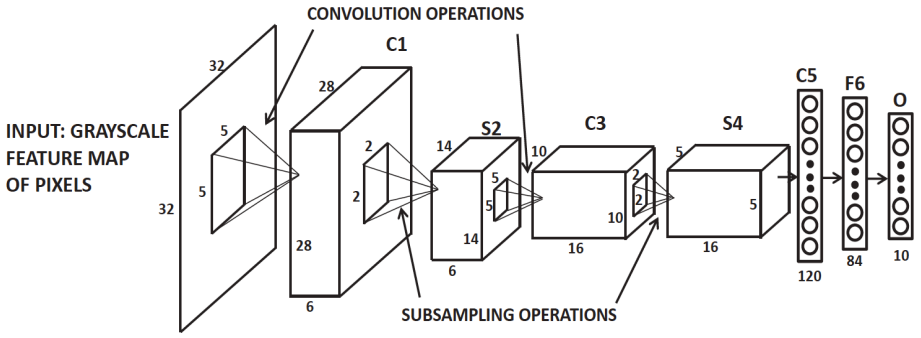
Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

Creating Convolutional Neural Networks

- Interleaving of convolution, pooling, and ReLU layers
- ReLU often follows the convolutional layers
- Max-pooling is often applied after convolution-ReLU combinations
- LeNet-5 (convolutional network for hand-written digits classification)
  - 7 layers, 3 convolutional layers (C1, C3, C5), 2 sub-sampling (pooling) layers (S2 and S4), 1 fully connected layer (F6), and an output (O) layer
  - Did not use ReLU activation functions
  - Used (scaled) hyperbolic tangent as activation function in hidden layers
  - C5 is basically a fully connected layer as each unit is mapped to all the 16 input activation maps and filter size is same as input maps
  - Pattern: Conv→Pool→Conv→ Pool→FullyConn→ FullConn→ Output

LeNet-5



Seminar Reading List

- **Distributed Training**
  - Alexander Sergeev, Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow 2018
  - Minsik Cho et al. PowerAI DDL 2017
  - TAL BEN-NUN, TORSTEN HOEFLER. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis
  - Andrew Gibiansky. Bringing HPC techniques to deep learning 2017
  - Goyal et al. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour 2017
  - Gupta et al. Staleness-aware Async-SGD for Distributed Deep Learning 2016
  - Gupta el al. Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study 2016
  - Nikko Storm. Scalable distributed dnn training using commodity gpu cloud computing 2015
  - Jeff Dean et al. Large Scale Distributed Deep Networks 2012

Suggested Reading

- Video: Inside Tensorflow tf.distribute.Strategy
- Google Cloud blog: What makes TPUs fine-tuned for deep learning?
- Andrew Gibiansky blog: Bringing HPC Techniques to Deep Learning 2017
- He et al. Deep Residual Learning for Image Recognition 2016
- Smith et al. Don't Decay the Learning Rate, Increase the Batch Size 2017
- Akiba et al. Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes 2017
- Jia et al. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes 2018
- Ying et al. Image Classification at Supercomputer Scale 2018
- H. Mikami et al. Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash 2018