[COMSE6998-015] Fall 2024
Introduction to Deep Learning and LLM based Generative AI Systems

Parijat Dube and Chen Wang

Lecture 2 09/10/24

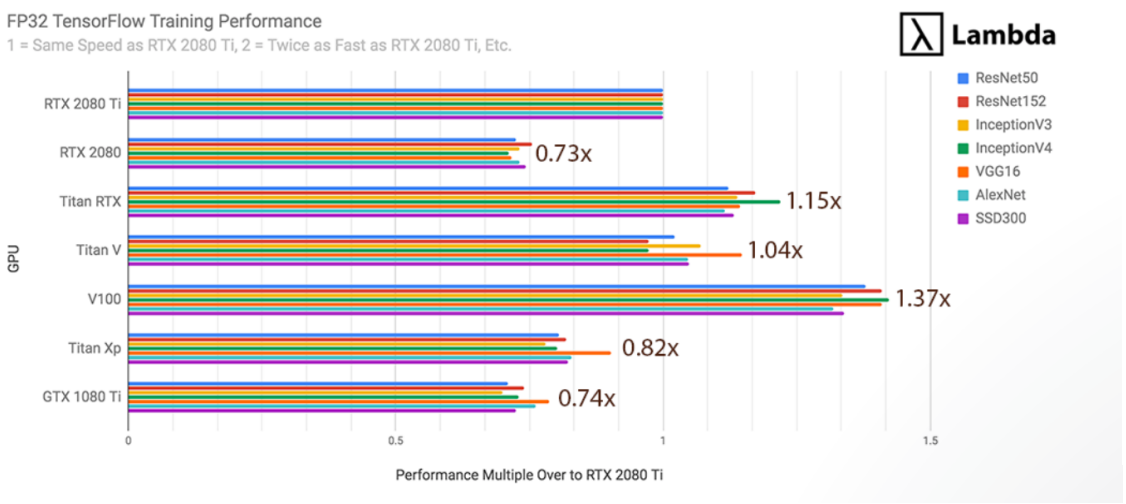
Recall from Last lecture

- ML concepts: Bias-variance tradeoff, generalization, regularization
- Algorithmic and system
- Steps in training a deep neural network; Stochastic gradient descent
- Hyperparameters in deep learning
- Vanishing gradient problem and Weight initialization
- Learning rate and batch size
- Momentum
- Batch normalization
- Regularization techniques: early stopping, dropout, dataset augmentation

Single Node, Single GPU Training

- Training throughput depends on:
  - Neural network model (activations, parameters, compute operations)
  - Batch size
  - Compute hardware: GPU type (e.g., Nvidia M60, K80, P100, V100)
  - Floating point precision (FP32 vs FP16)
    - Using FP16 can reduce training times and enable larger batch sizes/models without significantly impacting the accuracy of the trained model
- Increasing batch size increases throughput
  - Batch size is restricted by GPU memory
- Training time with single GPU very large: 6 days with Places dataset (2.5M images) using Alexnet on a single K40.
- Small batch size => noisier approximation of the gradient => lower learning rate => slower convergence

Single GPU Training



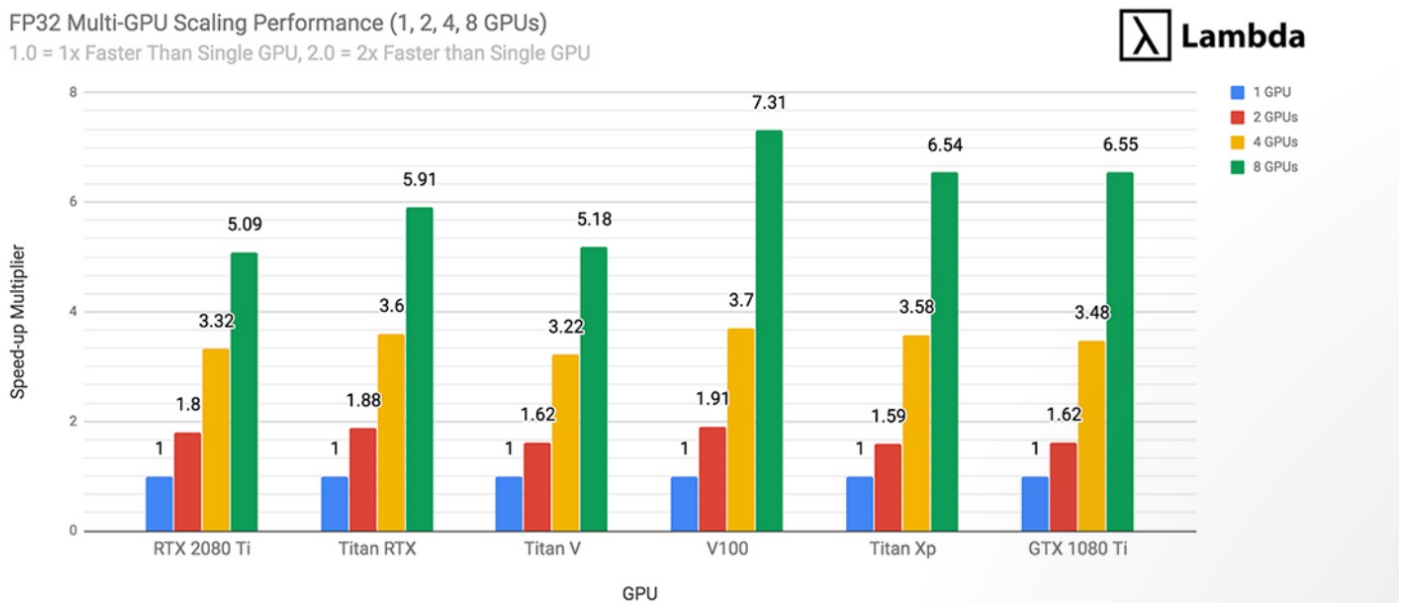
GPU Prices

- RTX 2080 Ti: $1,199.00
- RTX 2080: $799.00
- Titan RTX: $2,499.00
- Titan V: $2,999.00
- Tesla V100 (32 GB): ~$8,200.00
- GTX 1080 Ti: $699.00
- Titan Xp: $1,200.00

The scaling with GPU type is dependent on neural network architecture.

https://lambdalabs.com/blog/2080-ti-deep-learning-benchmarks/

Single Node, Multi-GPU Training



https://lambdalabs.com/blog/2080-ti-deep-learning-benchmarks/

Multi-GPU Execution Scaling

- Vertical scaling-up in a single node
  - NVIDIA DGX-1 (8 P100 GPUs) and DGX-2 (16 V100 GPUs) servers

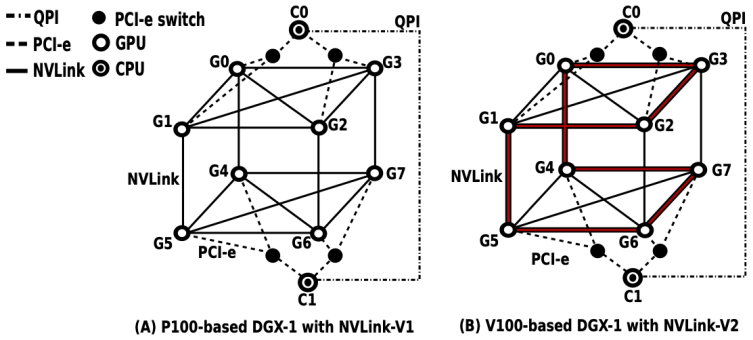


Fig. 1: PCIe and NVLink-V1/V2 topology for P100-DGX-1 and V100-DGX-1.

- Horizontal scaling-out across multiple nodes
  - Example: GPU accelerated supercomputers like Summit and Sierra from US Department of Energy

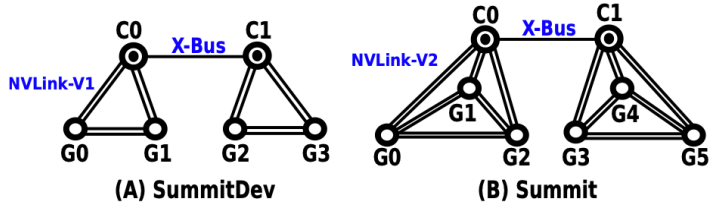


Fig. 2: NVLink interconnect topology for SummitDev and Summit.

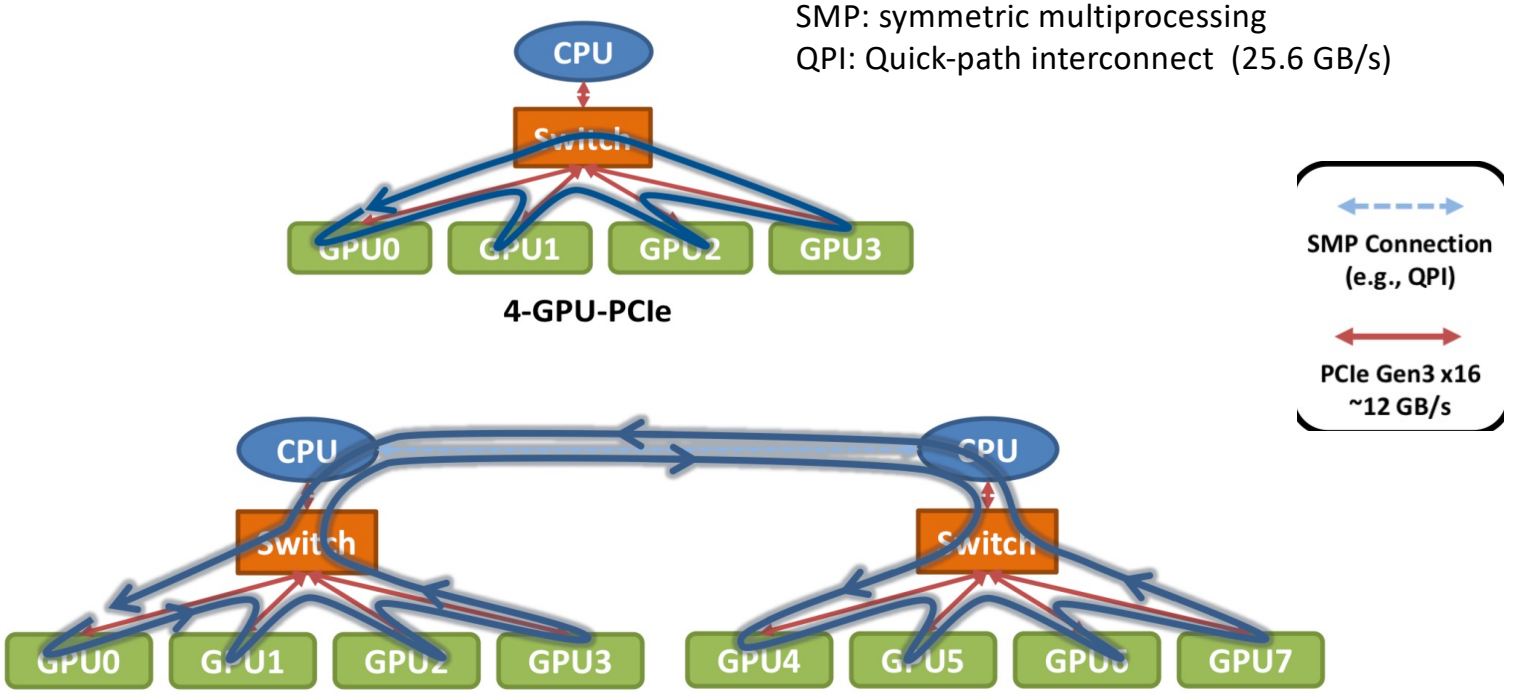
[Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect]

Single Node, Multi GPU Training

- Communication libraries (e.g., NCCL) and supported communication algorithms/collectives (broadcast, all-reduce, gather)
  - NCCL (“Nickel”) is library of accelerated collectives that is easily integrated and topology-aware so as to improve the scalability of multi-GPU applications
- Communication link bandwidth: PCIe/QPI or NVlink
- Communication algorithms depend on the communication topology (ring, hub-spoke, fully connected) between the GPUs.

Ring based collectives

PCIe: Peripheral Component Interconnect express
SMP: symmetric multiprocessing
QPI: Quick-path interconnect (25.6 GB/s)



PCIe and NVLink

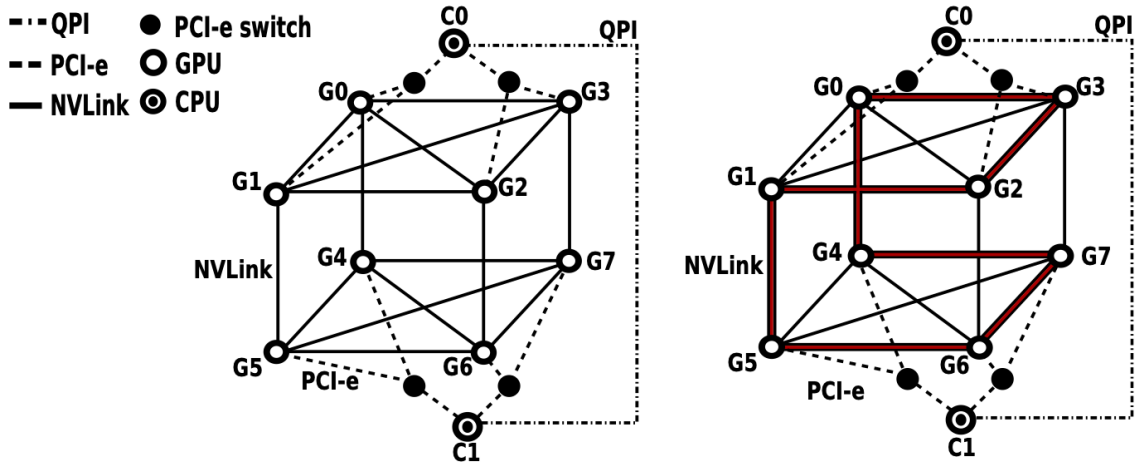


Fig. 1: PCIe and NVLink-V1/V2 topology for P100-DGX-1 and V100-DGX-1.

NVSwitch and PCIe in DGX-2

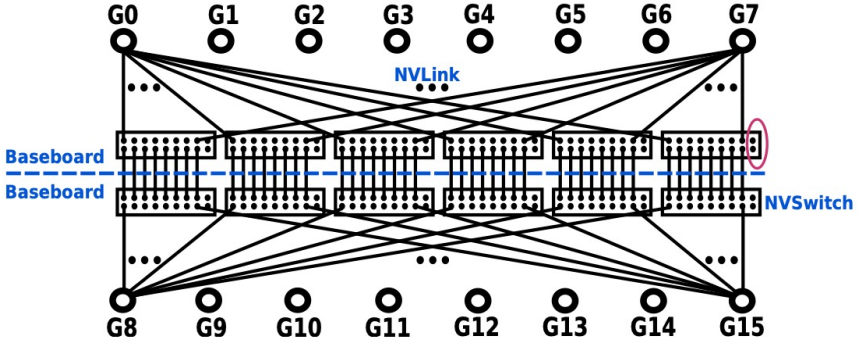


Fig. 3: NVSwitch interconnect topology in DGX-2.

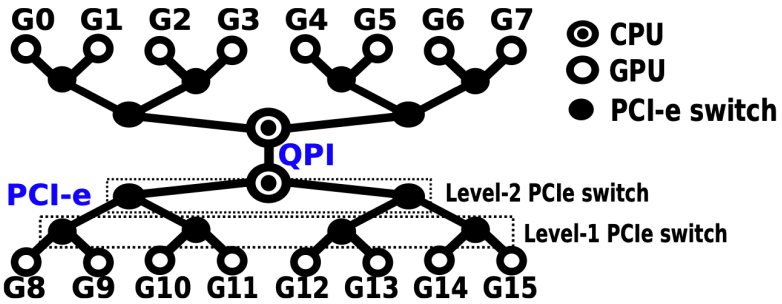
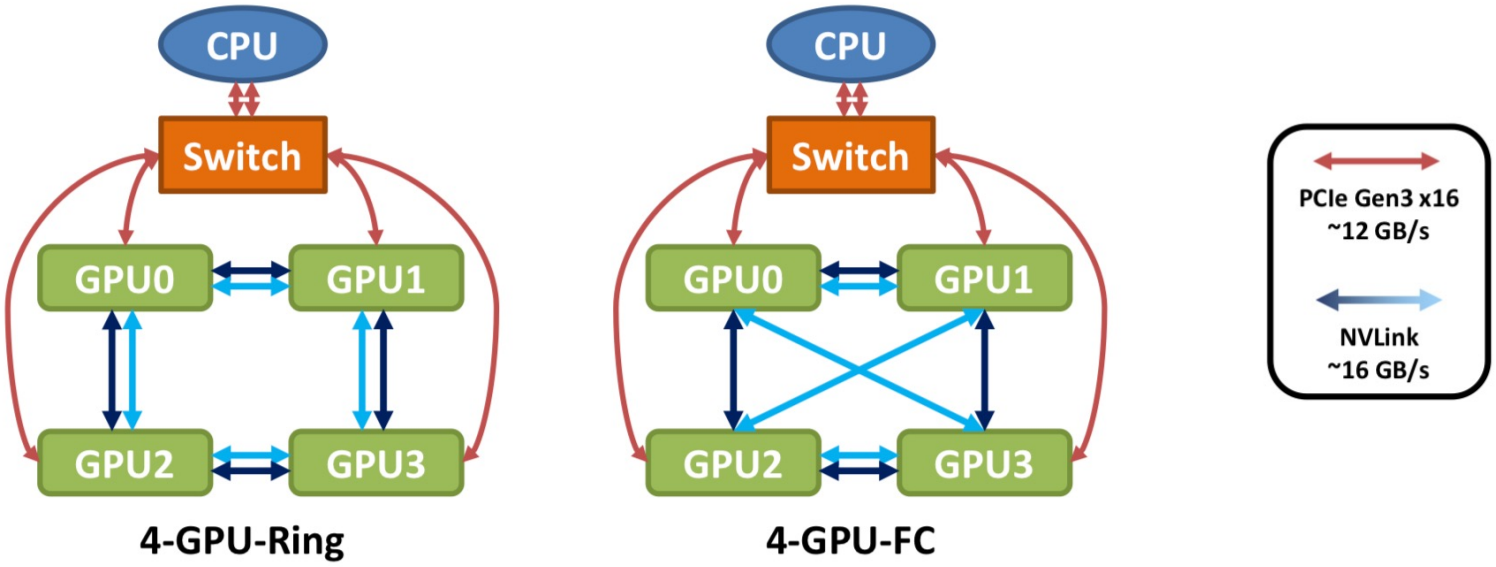


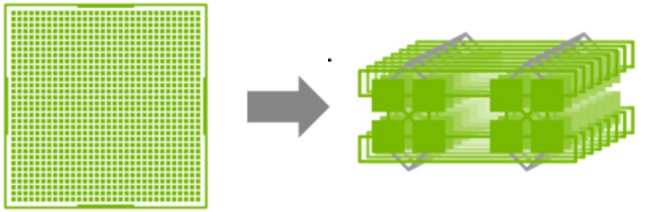
Fig. 4: PCIe interconnect topology in DGX-2.

Ring based collectives with NVLink



Distributed Training

- Type of Parallelism: Model, Data, Hybrid
- Type of Aggregation: Centralized, decentralizec
- Centralized aggregation: parameter server
- Decentralized aggregation: P2P, all reduce
- Performance metric: Scaling efficiency
  - Defined as the ratio between the run time of one iteration on a single GPU and the run time of one iteration when distributed over n GPUs.



1 GPU

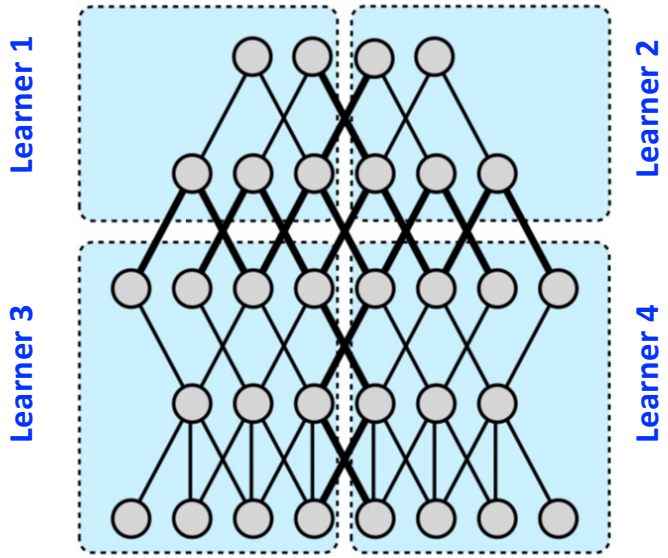
multi-GPU, multi-node

Parallelism

- Parallel execution of a training job on different compute units through scale-up (single node, multiple and faster GPUs) or scale-out (multiple nodes distributed training)
- Enables working with large models by partitioning model across learners
- Enables efficient training with large datasets using large “effective” batch sizes (batch split across learners)
  - Speeds up computation
- Model, Data, Hybrid Parallelism

Model Parallelism

- Splitting the model across multiple learners



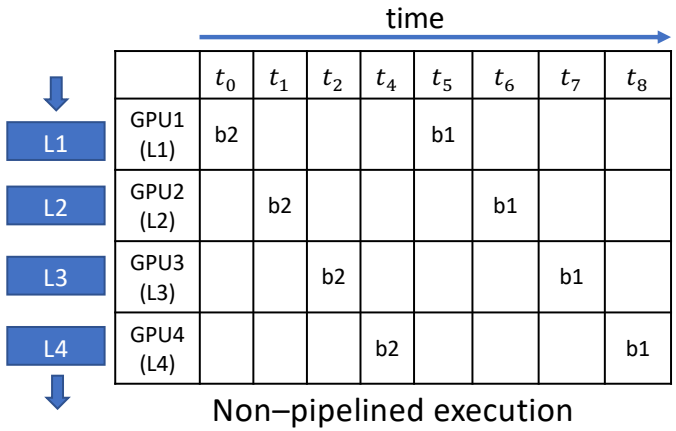
- 5 layered neural network
- Partitioned across 4 learners
- Bold edges cross learn boundaries and involve inter-learner communication
- Performance benefits depend on
    - Connectivity structure
    - Compute demand of operations
- Heavy compute and local connectivity –benefit most
    - Each machine handles a subset of computation
    - Low network traffic

DL Pipelining

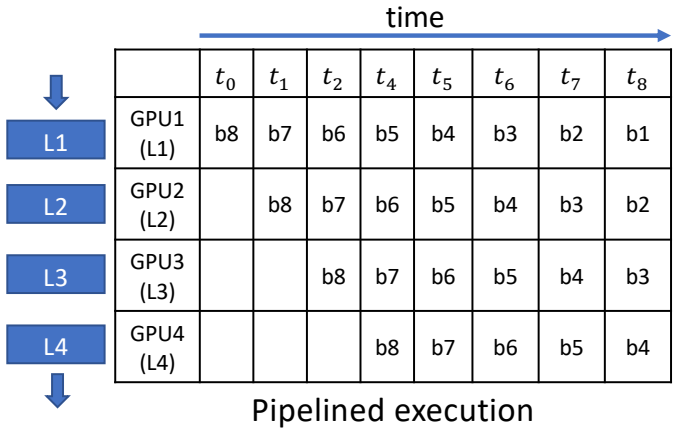
- Pipelining approach:
  - Split layers among compute engines
  - Each minibatch *b (or sample s)* goes from one compute engine to the next one: *no need to wait for next one to exit the pipeline*
- Is a form of **Model Parallelism**
- Pipelining performance
  - Ideal pipelining speedup *(number of pipeline stages)*

$$S\_time(stage\ time) = \frac{time\ without\ pipeline}{number\ of\ pipeline\ stages}$$

  - Speedup is higher for deeper networks
  - Ideal pipelining never reached because of "bubbles" that cause idle CPUs
  - SGD pipeline bubble:
    - Before weights update, all batches need to have completed forward (otherwise accept **staleness**)

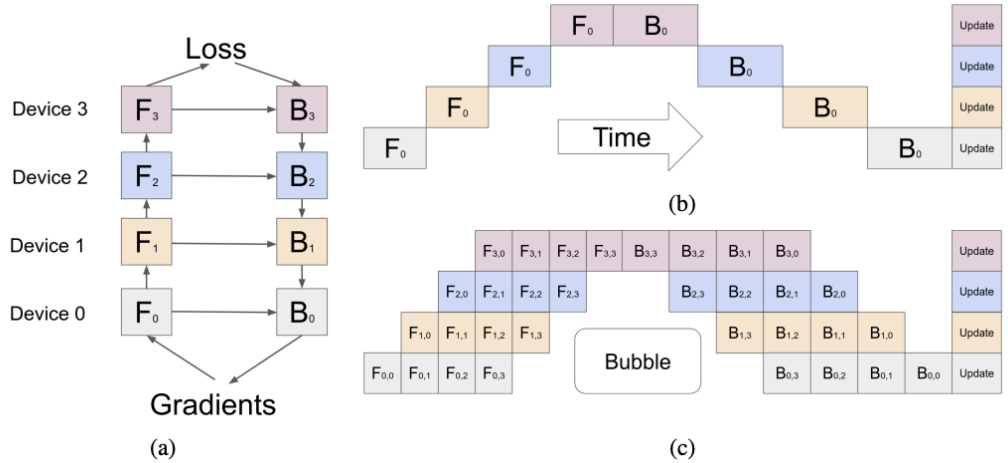


Non–pipelined execution



Pipelined execution

GPipe Pipelining



- Gpipe: a pipeline parallelism open-source library that allows scaling any network that can be expressed as a sequence of layers.
- Split global batch into multiple micro-batches and injects them into the pipeline concurrently
- Not memory-friendly and will not scale well with large batch. The activations produced by forward tasks have to be kept for all micro-batches until corresponding backward tasks start, thus leads to the memory demand to be proportional (O(M)) to the number of concurrently scheduled micro-batches (M).

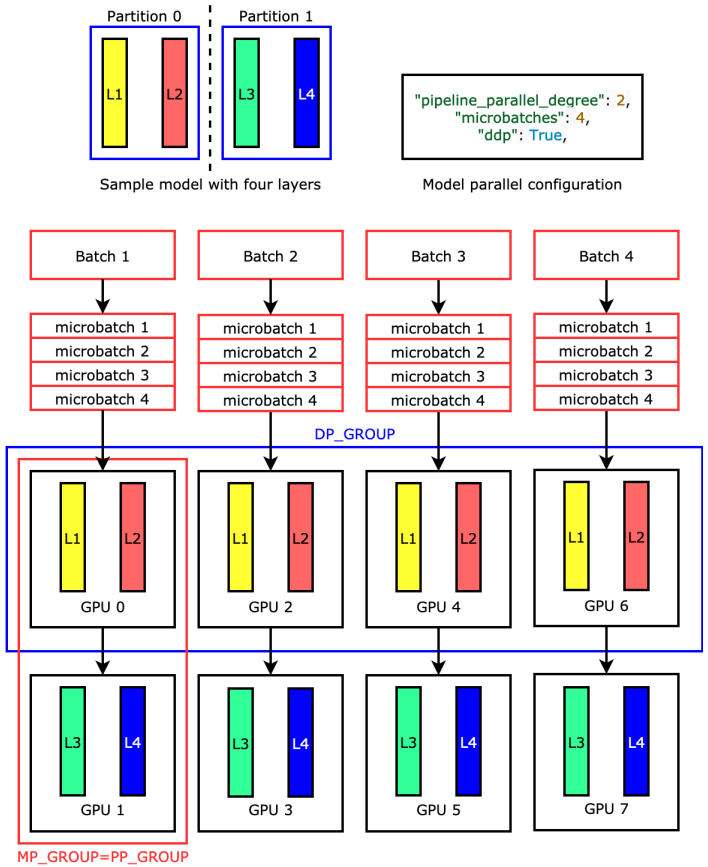
GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism

Model Parallelism and model saving technique in Amazon Sagemaker model parallel library

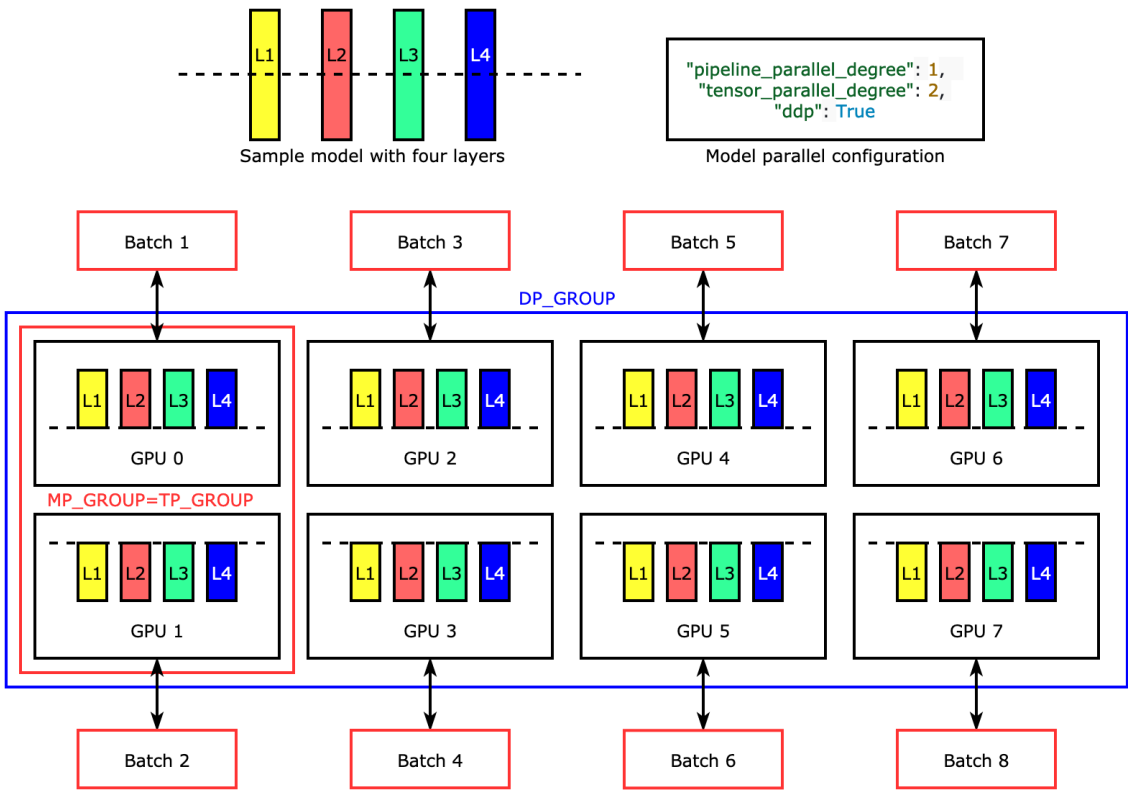
- Pipeline Parallelism
- Tensor Parallelism
- Optimizer state sharding
- Activation offloading and checkpointing

https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-intro.html

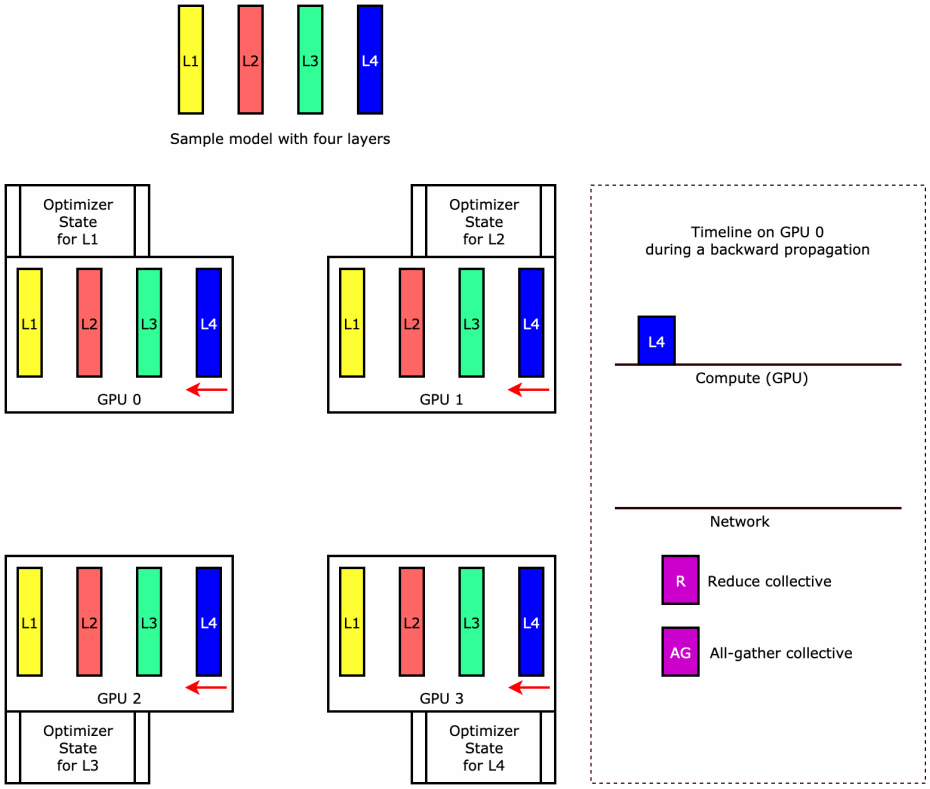
Pipeline Parallelism



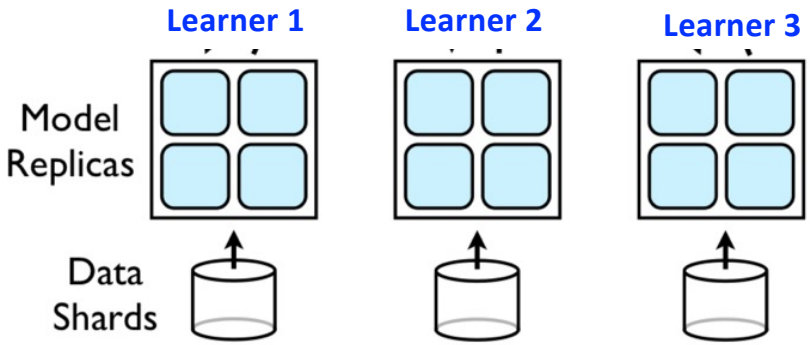
Tensor Parallelism



Optimizer state sharding

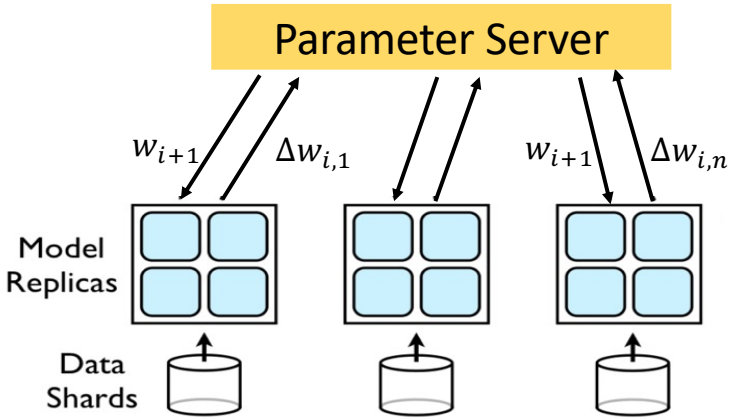


Data Parallelism



- Model is replicated on different learners
- Data is sharded and each learner work on a different partition
- Helps in efficient training with large amount of data
- Parameters (weights, biases, gradients) from different replicas need to be synchronized

Parameter server (PS) based Synchronization



- Each learner executes the entire model
- After each mini-batch processing a learner calculates the gradient and sends it to the parameter server
- The parameter server calculates new value of weights and sends them to the model replicas

Synchronous SGD and the Straggler problem

- PS needs to wait for updated gradients from all the learners before calculating the model parameters
- Even though size of mini-batch processed by each learner is same, updates from different learners may be available at different times at the PS
  - Randomness in compute time at learners
  - Randomness in communication time between learners and PS
- Waiting for slow and straggling learners diminishes the speed-up offered by parallelizing the training

Synchronous SGD Variants

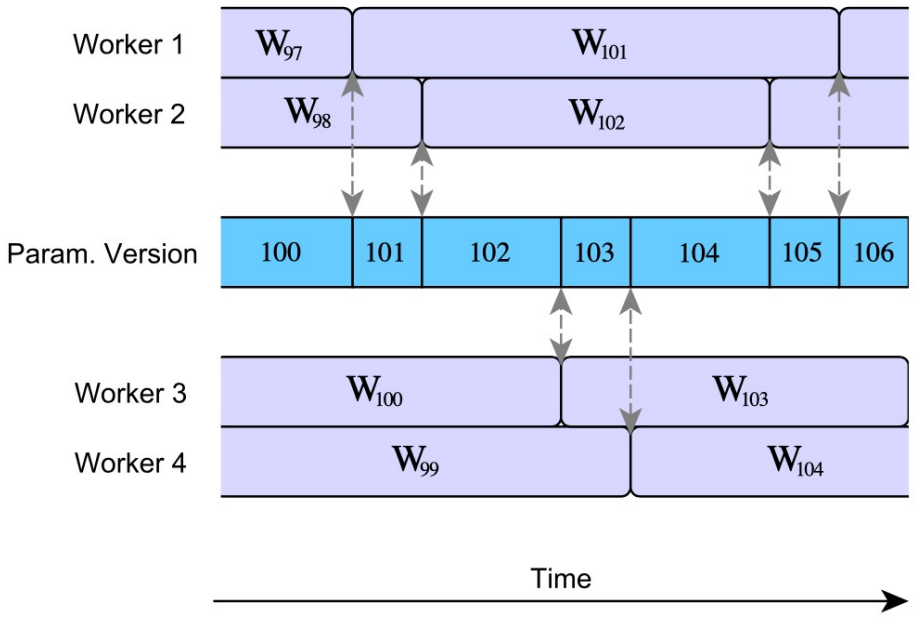
- P: total number of learners
- K: number of learners/mini-batches the PS waits for before updating parameters
- Lightly shaded arrows indicate straggling gradient computations that are canceled.

- K-sync SGD: PS waits for gradients from K *learners* before updating parameters; the remaining learners are canceled
- When K = P , K-sync SGD is same as Fully Sync-SGD
- K-batch sync: PS waits for gradients from K *mini-batches* before updating parameters; **the remaining (unfinished) learners are canceled**
  - Irrespective of which learner the gradients come from
  - Wherever any learner finishes, it pushes its gradient to the PS, fetches current parameter at PS and starts computing gradient on the next mini-batch based on the same local value of the parameters
- Runtime per iteration reduces with K-batch sync; error convergence is same as K-sync

Asynchronous SGD and Stale Gradients

- PS updates happen without waiting for all learners
- Weights that a learner uses to evaluate gradients may be old values of the parameters at PS
  - Parameter server asynchronously updates weights
  - By the time learner gets back to the PS to submit gradient, the weights may have already been updated at the PS (by other learners)
  - Gradients returned by this learner are stale (i.e., were evaluated at an older version of the model)
- Stale gradients can make SGD unstable, slowdown convergence, cause sub-optimal convergence (compared to Sync-SGD)

Stale Gradient Problem in Async-SGD



$$\vdots$$

$$W_{101} = W_{100} - \lambda \Delta W_{97}$$

$$W_{102} = W_{101} - \lambda \Delta W_{98}$$

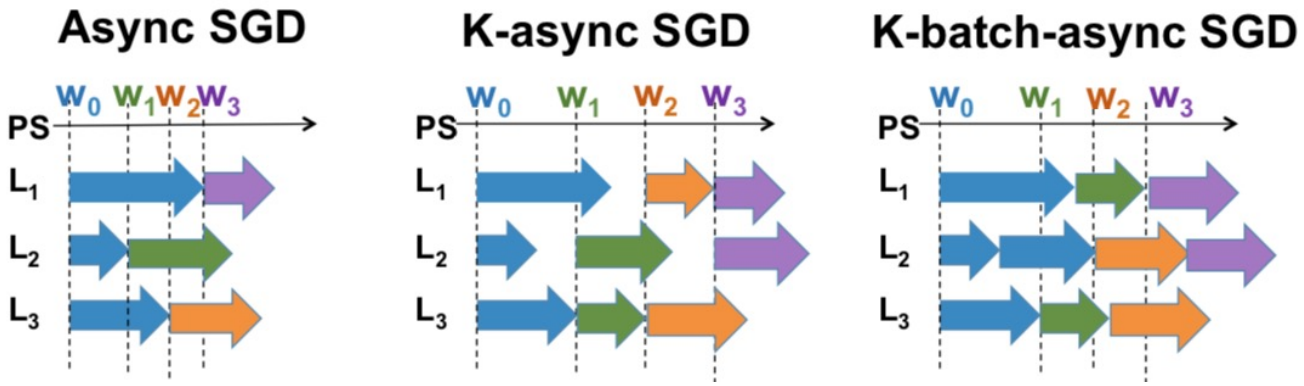
$$W_{103} = W_{102} - \lambda \Delta W_{100}$$

$$W_{104} = W_{103} - \lambda \Delta W_{99}$$

$$\vdots$$

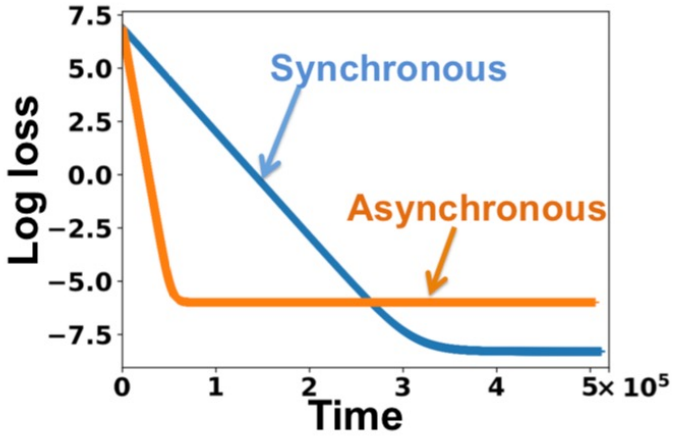
blog.skymind.ai Distrbuted Deep Learning, Part 1: An Introduction to Distributed Training of Neural Networks

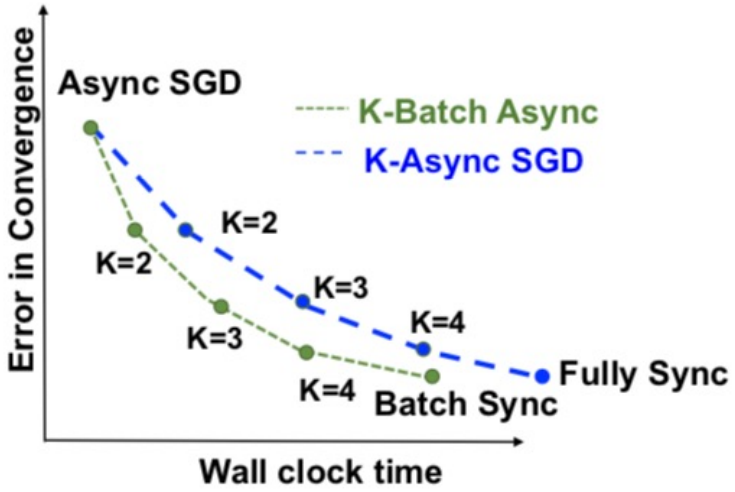
Asynchronous SGD and Variants



- K-async SGD: PS waits for gradients from K *learners* before updating parameters but the remaining learners are not canceled; each learner may be giving a gradient calculated at stale version of the parameters
- When K = 1 , K-async SGD is same as Async-SGD
- K-batch async: PS waits for gradients from K *mini-batches* before updating parameters; **the remaining learners are not canceled**
  - Wherever any learner finishes, it pushes its gradient to the PS, fetches current parameter at PS and starts computing gradient on the next mini-batch based on the current value of the PS
- Runtime per iteration reduces with K-batch async; error convergence is same as K-async

Error-Runtime Tradeoff in SGD Variants

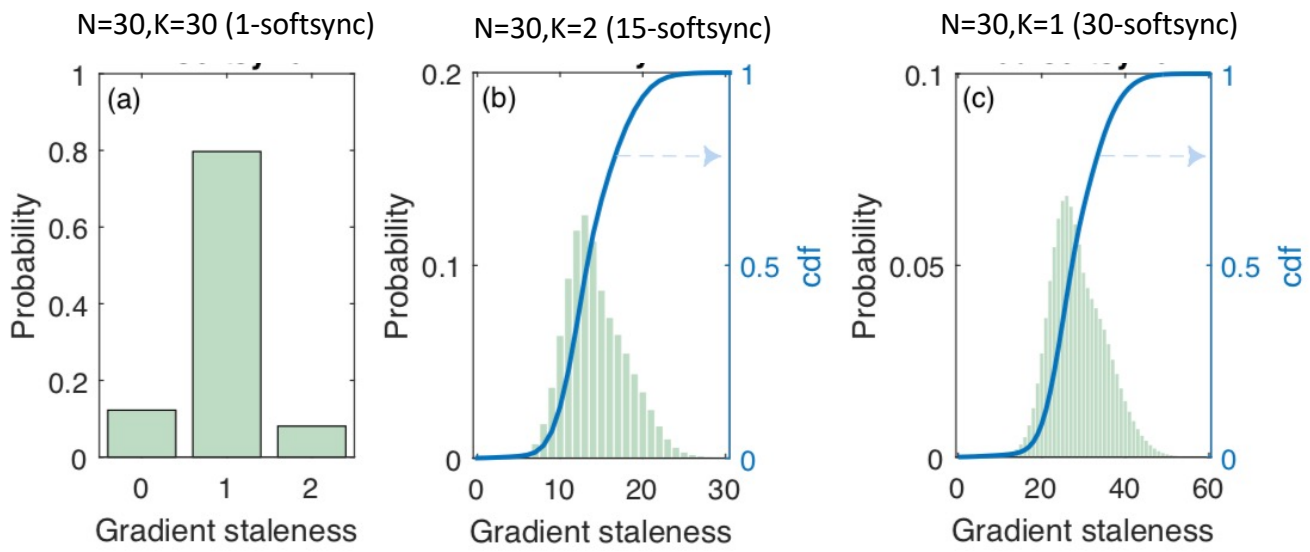




- Error-runtime trade-off for Sync and Async-SGD with same learning rate.
- Async-SGD has faster decay with time but a higher error floor.

Dutta et al. Slow and stale gradients can win the race: error-runtime tradeoffs in distributed SGD. 2018

Distribution of Staleness for K-batch Async



Called n-softsync protocol
K = floor (N/n)

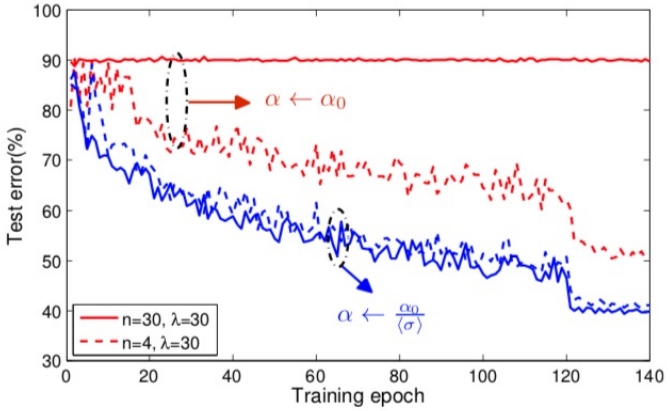
The gradient is on average N/K (N: number of learners) steps out of date by the time they are applied to the global parameter vector.

Gupta el al. Staleness-aware Async-SGD for Distributed Deep Learning. 2016

Staleness Dependent Learning Rate

- With staleness-dependent learning rate setting Async-SGD can achieve accuracy comparable to Sync-SGD while achieving linear speedup of ASGD
- Decrease the learning rate by mean staleness

$$learning\ rate\ (\alpha) = \frac{base\ learning\ rate\ (\alpha_0)}{average\ staleness\ (\langle\sigma\rangle)}$$



Gupta el al. Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study 2016

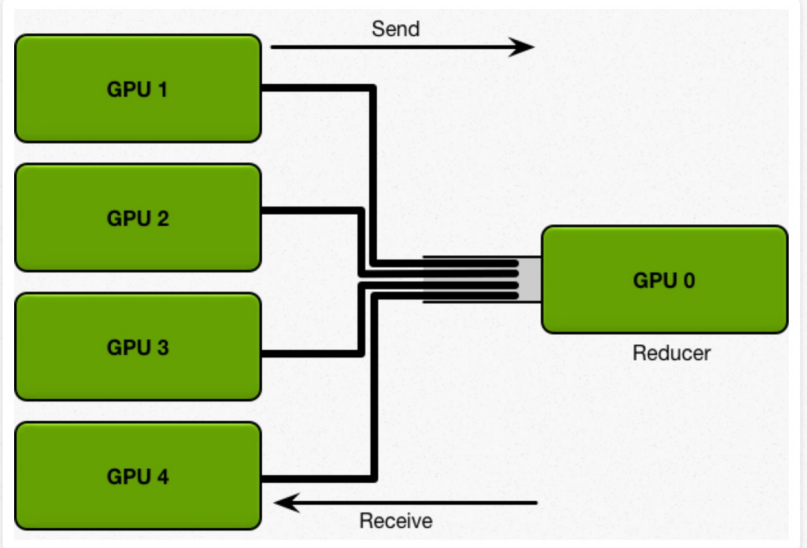
Reduction over Gradients

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

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

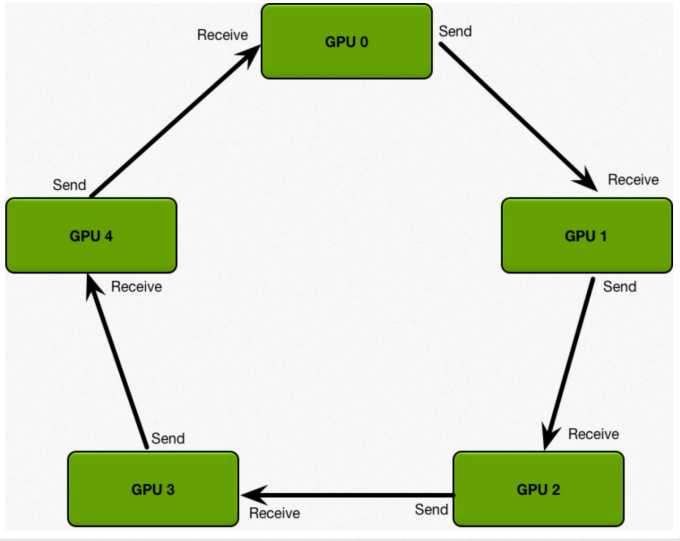
Reduction Topologies

Parameter server: single reducer



SUM (Reduce operation) performed at PS

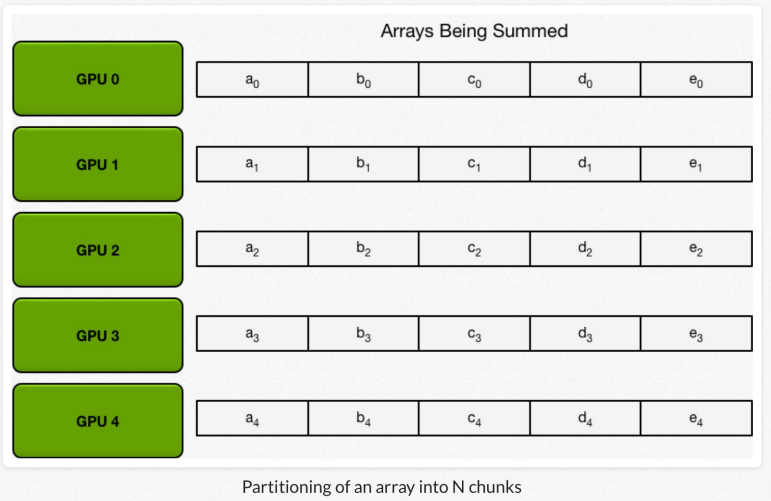
GPUs arranged in a logical Ring (aka bucket) :
all are reducers



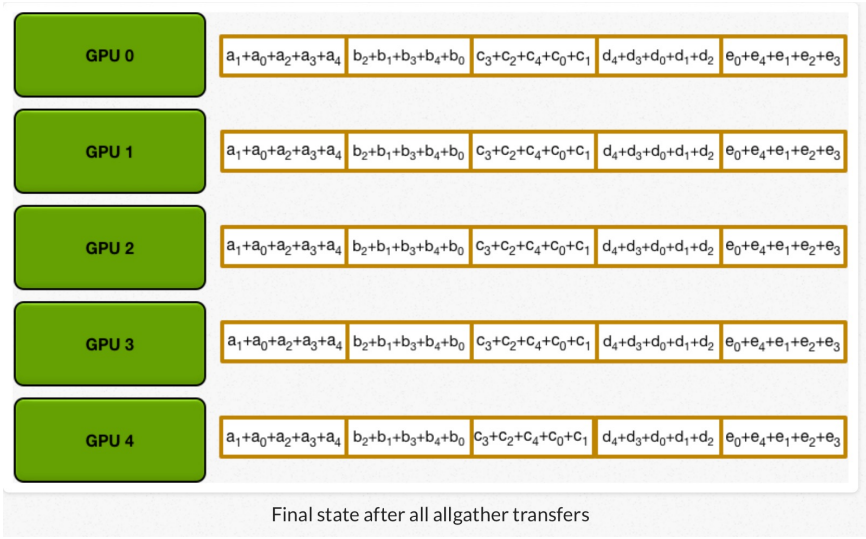
SUM (Reduce operation) performed at all nodes

- 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

All-Reduce



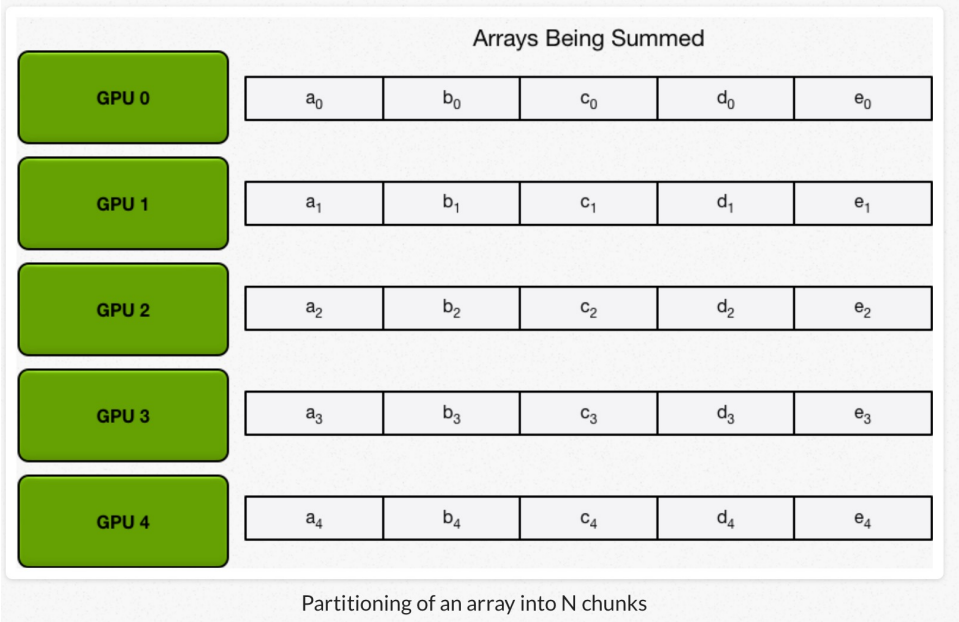
Partitioning of an array into N chunks



Final state after all allgather transfers

Ring All-Reduce

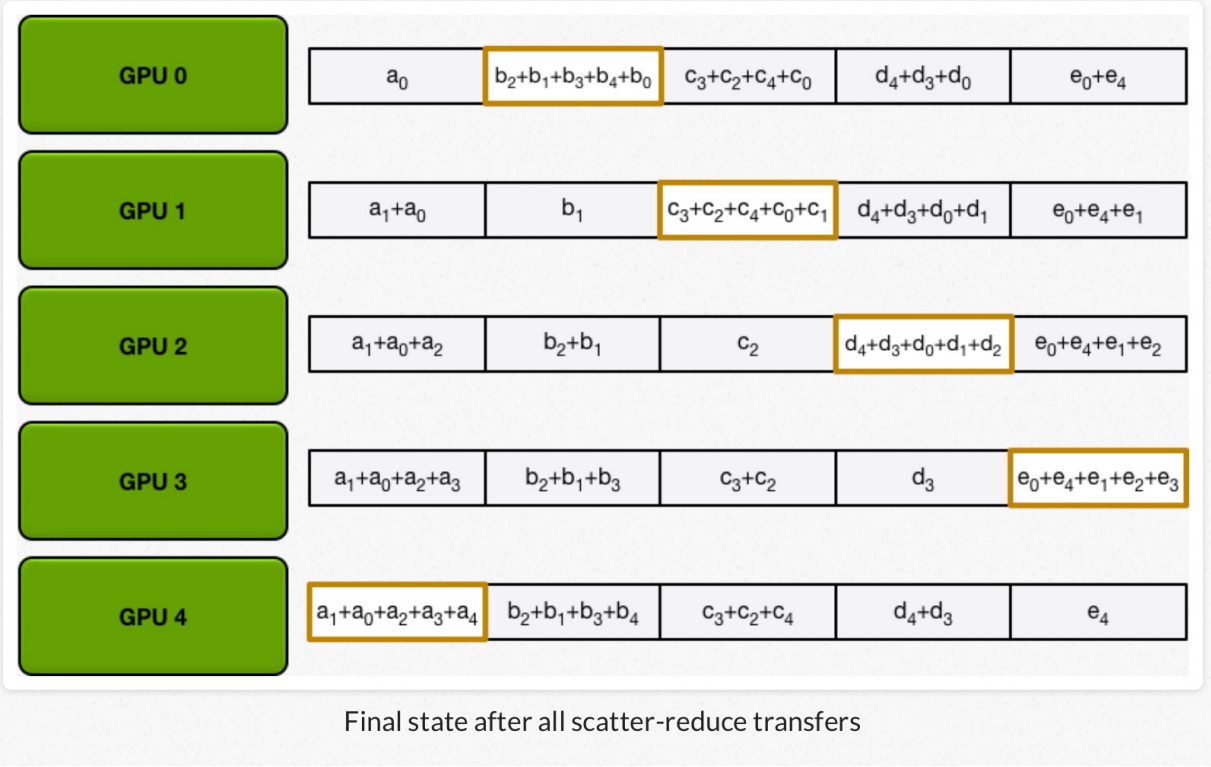
- Two step algorithm:
  - *Scatter-reduce*
    - GPUs exchange data such that every GPU ends up **with a chunk** of the final result
  - *Allgather*
    - GPUs exchange chunks from scatter-reduce such that all GPUs end up with the complete final result.



Partitioning of an array into N chunks

Ring All-Reduce: Scatter-Reduce Step

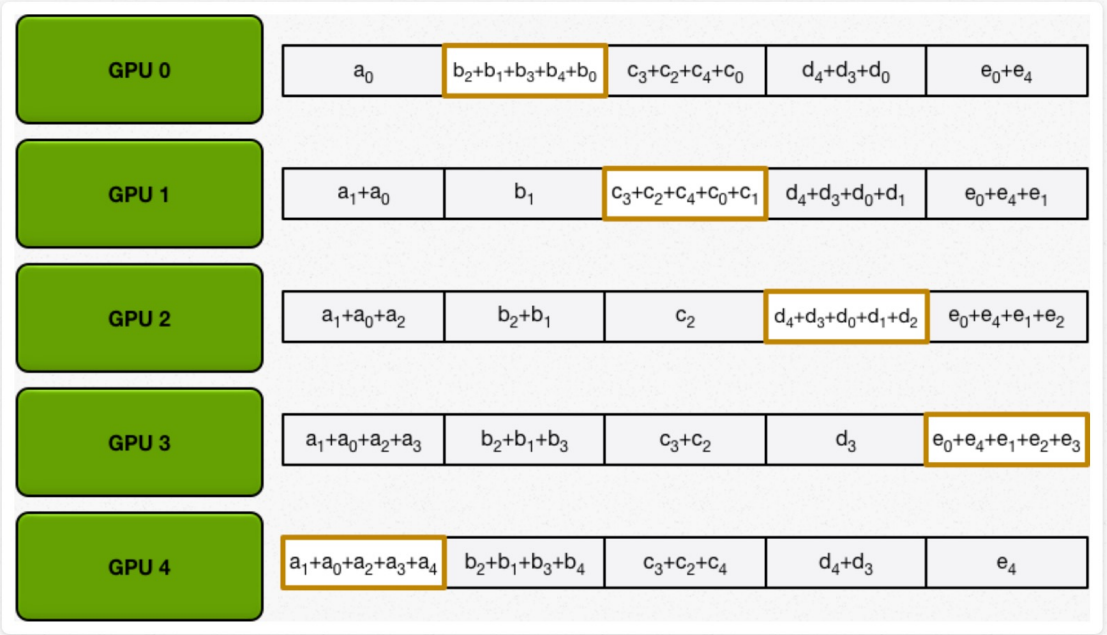
Ring All-Reduce: End of Scatter-Reduce Step



Final state after all scatter-reduce transfers

How many iterations in scatter-reduce step with N GPUs ?

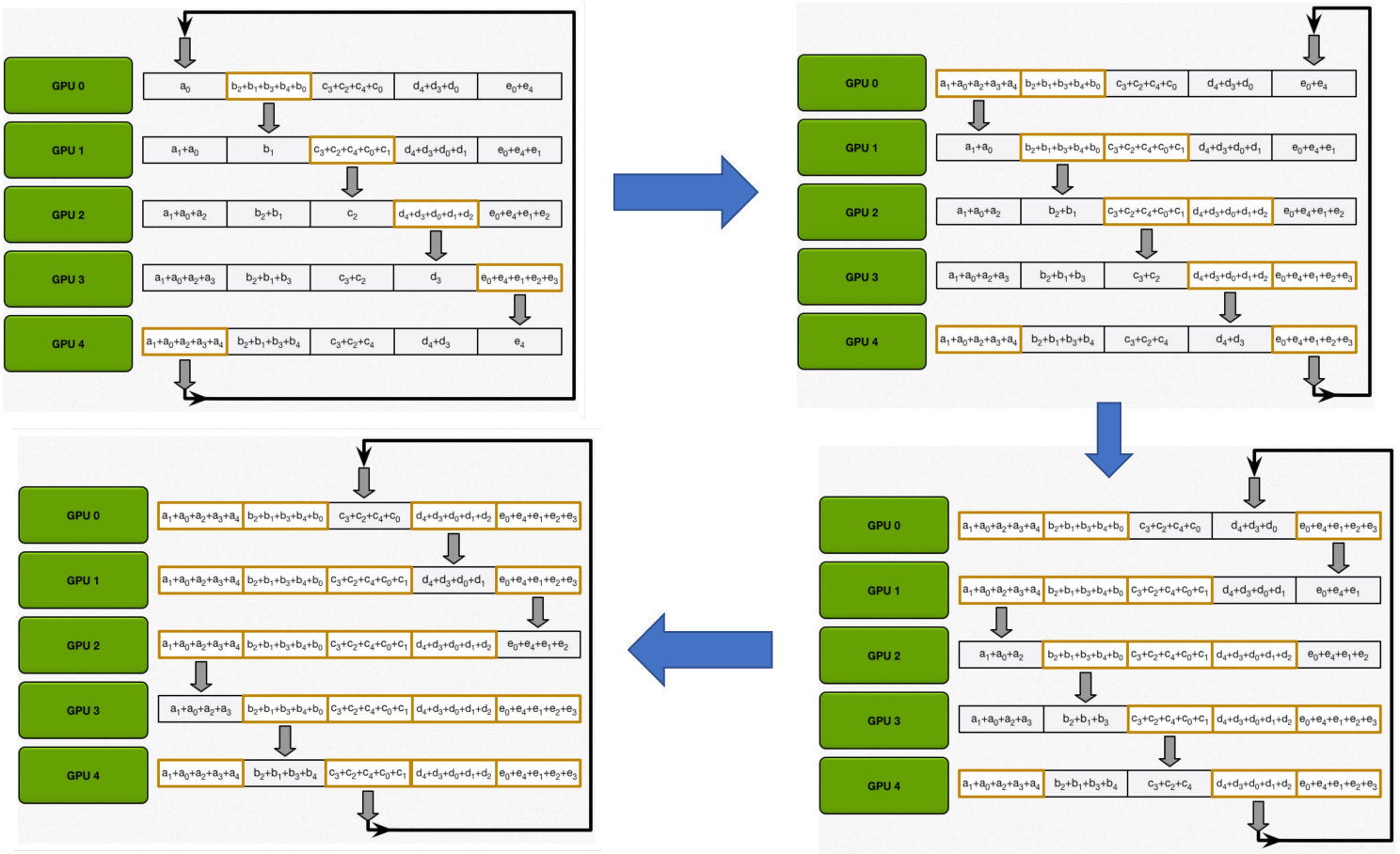
Ring All-Reduce: What to do next ?



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

How many iterations in allgather step with N GPUs ?

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)
- Which scheme is more bandwidth efficient ?
- 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

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

Learning Rates for Large Minibatches

Linear Scaling Rule: *When the minibatch size is multiplied by k, multiply the learning rate by k.*

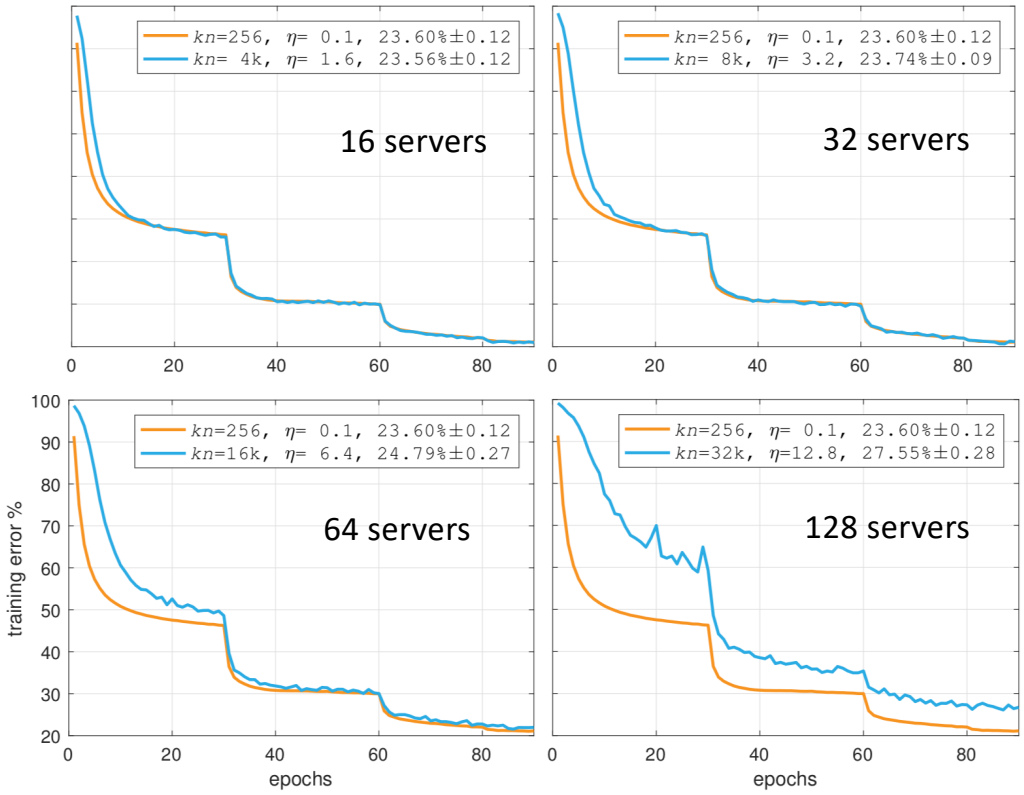
$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in \mathcal{B}} \nabla l(x, w_t)$$

$$w_{t+k} = w_t - \eta \frac{1}{n} \sum_{j<k} \sum_{x \in \mathcal{B}_j} \nabla l(x, w_{t+j})$$

$$\hat{w}_{t+1} = w_t - \hat{\eta} \frac{1}{kn} \sum_{j<k} \sum_{x \in \mathcal{B}_j} \nabla l(x, w_t)$$

Suppose $\nabla l(x, w_t) \approx \nabla l(x, w_{t+j})$ for $j < k$, when will $\hat{w}_{t+1} \approx w_{t+k}$?

Training Error Vs Batch Size: Distributed Training



Beyond 8k batch size, the training error deteriorates

Runtime Scaling

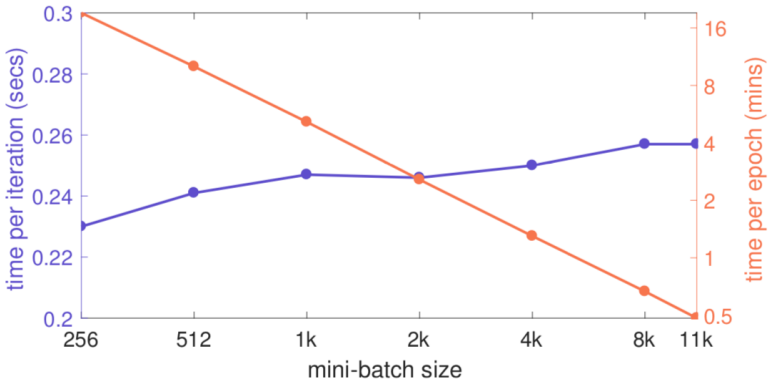


Figure 7. **Distributed synchronous SGD timing.** Time per iteration (seconds) and time per ImageNet epoch (minutes) for training with different minibatch sizes. The baseline ($kn = 256$) uses 8 GPUs in a single server , while all other training runs distribute training over ($kn/256$) server. With 352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

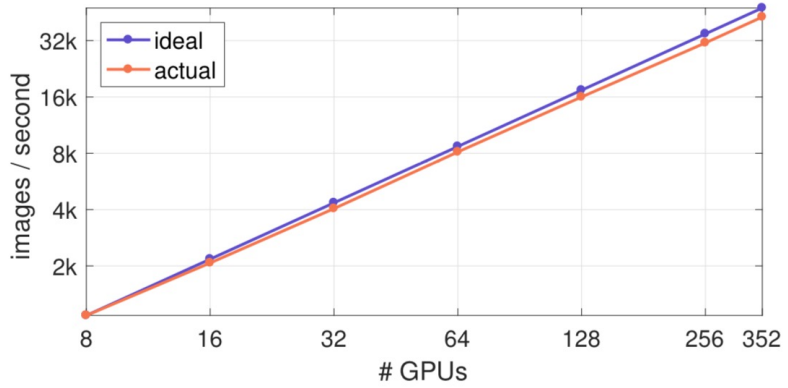


Figure 8. **Distributed synchronous SGD throughput.** The small overhead when moving from a single server with 8 GPUs to multi-server distributed training (Figure 7, blue curve) results in linear throughput scaling that is marginally below ideal scaling (~90% efficiency). Most of the allreduce communication 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 ? What about the final training error ?

Distributed Deep Learning Benchmarking Methodology

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

Speedup (throughput) with n machines = n x Scaling efficiency with n machines

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. **Why is this ratio a measure of scaling efficiency ?**
- One can satisfy any given scaling efficiency for any neural network by increasing the batch size and reducing communication overhead
- 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

Best practices when benchmarking distributed deep learning systems

- 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.

Best practices when benchmarking distributed deep learning systems

- 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.

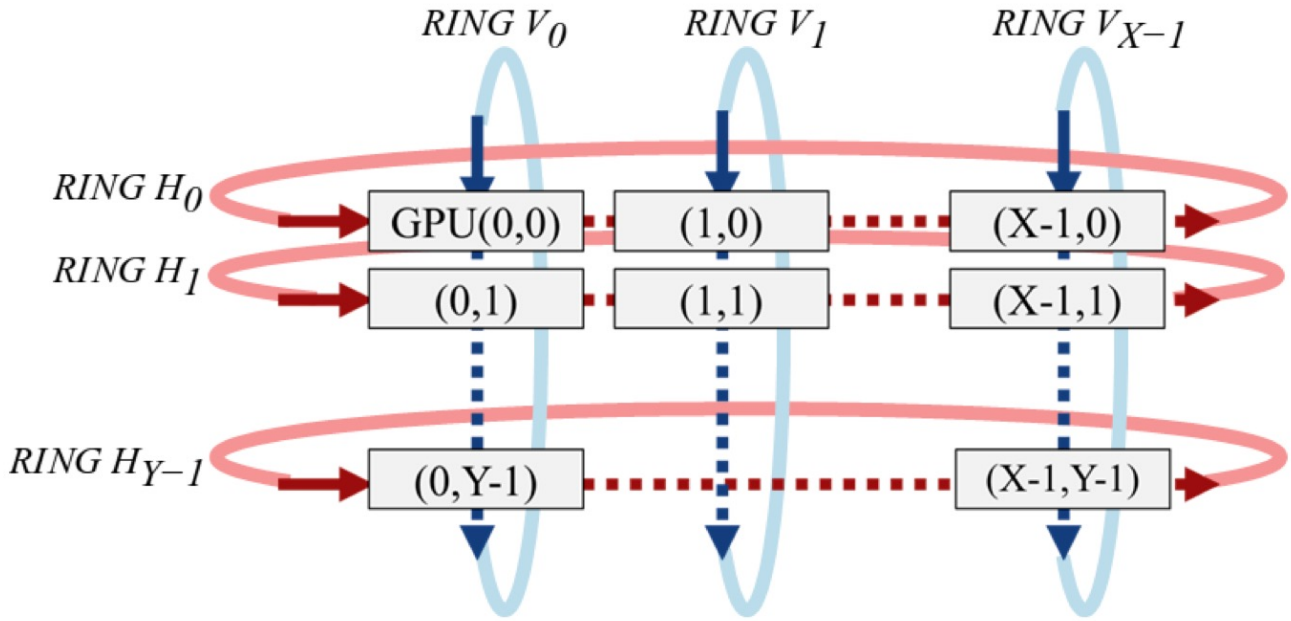
Imagenet1K/ResNet50 Training at Scale

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

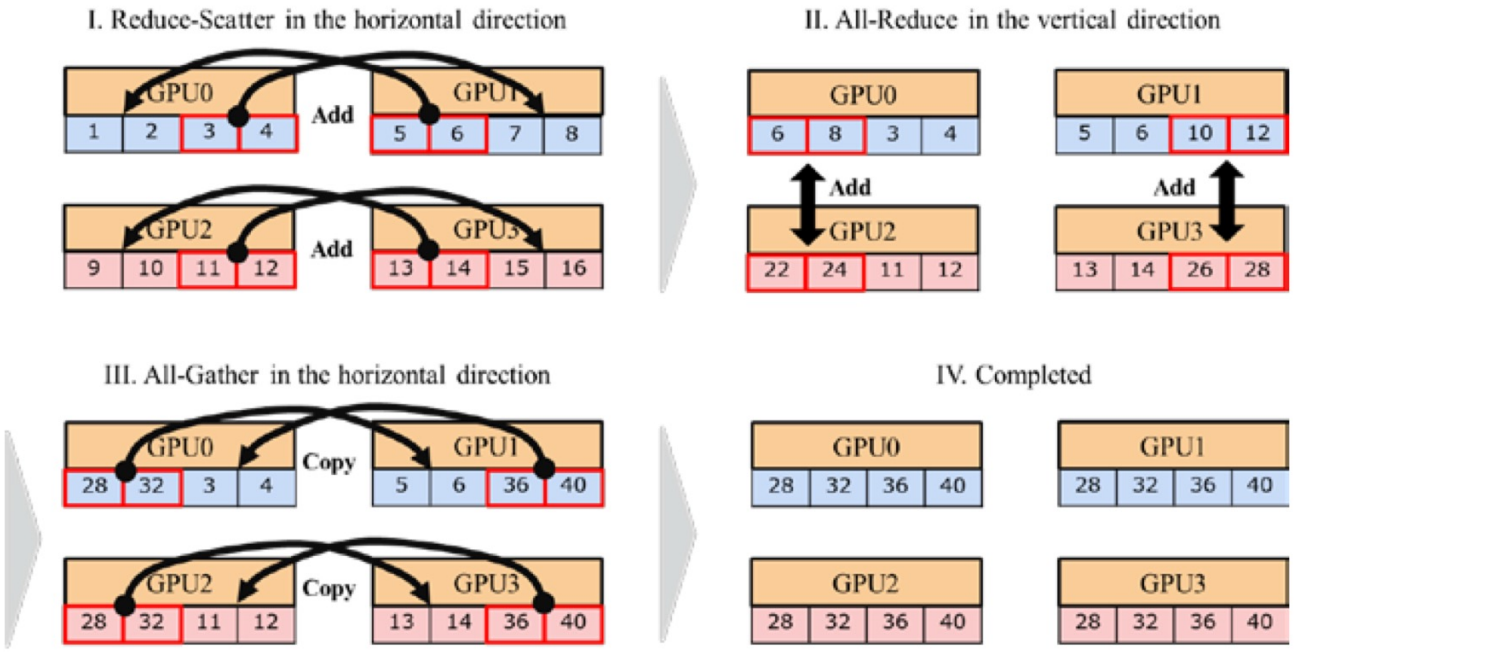
2-D Torus Topology for inter-gpu communication

multiple rings in horizontal and vertical orientations.



2-D Torus all-reduce

2D-Torus all-reduce steps of a 4-GPU cluster, arranged in 2x2 grid

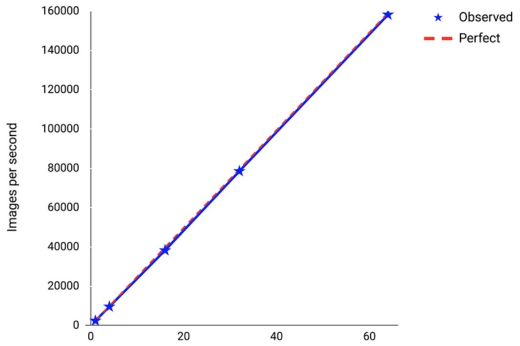


https://arxiv.org/pdf/1811.05233.pdf

Tensor Processing Units (TPUs)

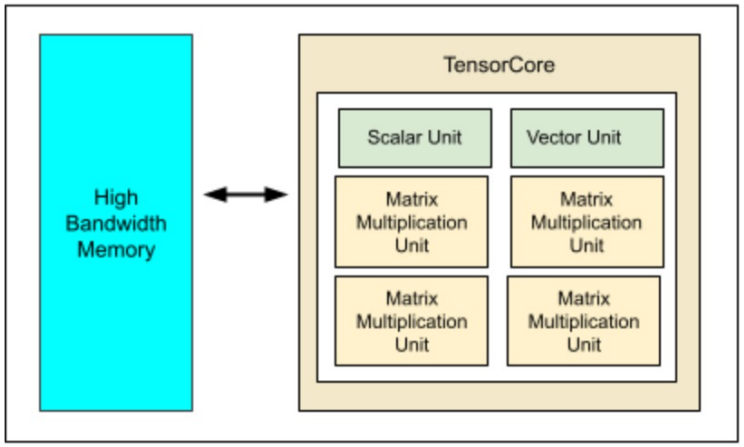
- TPU: application specific integrated circuits to accelerate machine learning workload
- TPU pod: multiple TPU chips connected to each other over a dedicated high-speed network connection
- https://cloud.google.com/tpu/docs/system-architecture

TPU v2 pod for ResNet-50: linearly scalable



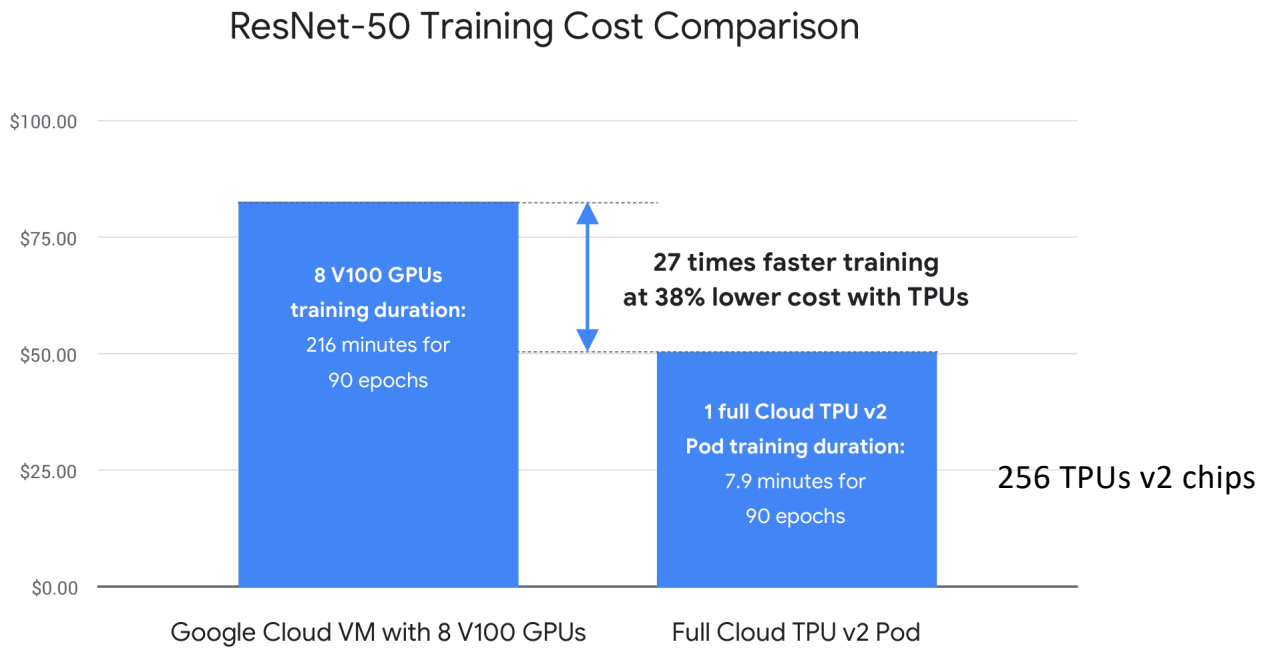
Number of TPU v2 chips

TPU v5e chip



A TensorCore has four matrix-multiply units (MXUs), a vector unit, and a scalar unit

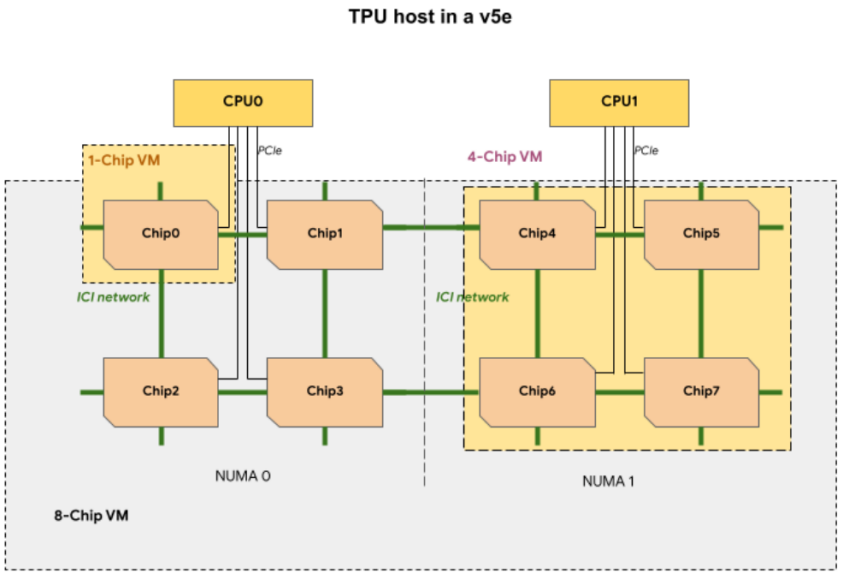
TPUs vs GPUs Performance



256 TPUs v2 chips

TPU VM on Google Cloud

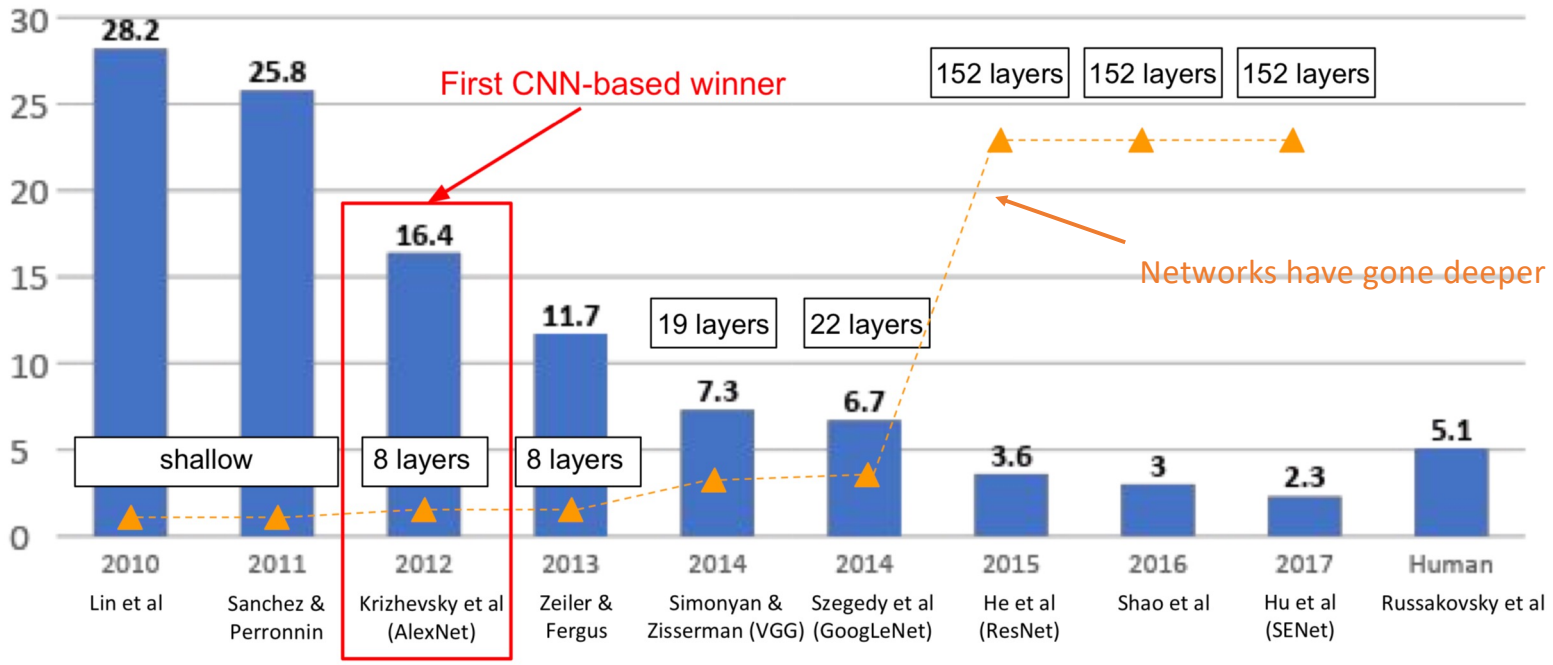
- A TPU host is a VM that runs on a physical computer connected to TPU hardware. TPU workloads can use one or more host.
- Tutorial on using Google Cloud TPU VM: Google Cloud Quickstart



NCCL

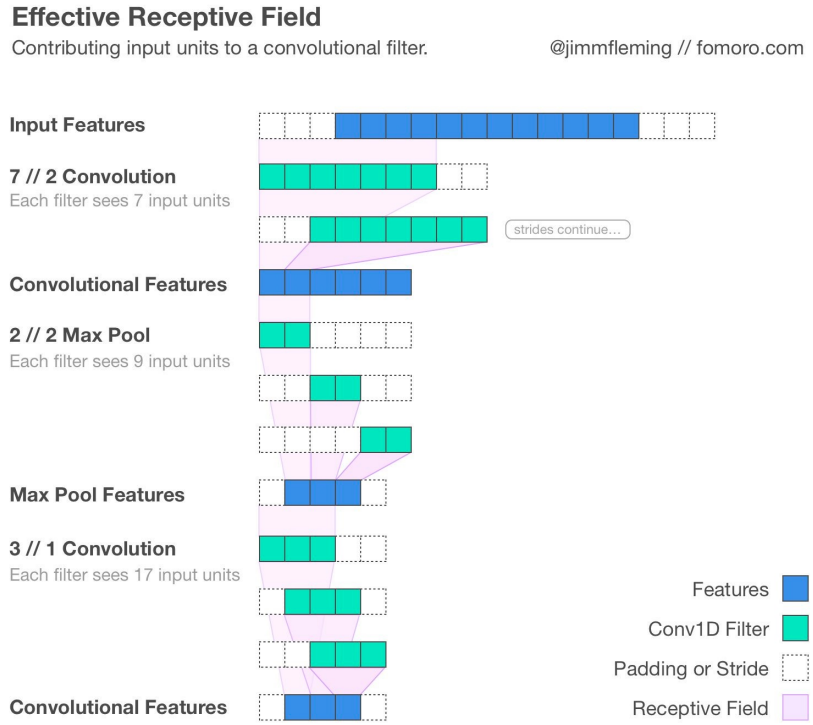
- Nvidia Common Communications Library
- NCCL implements optimized multi-GPU and multi-node communication primitives for NVIDIA gpus and networking
- NCCL provides routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter as well as point-to-point send and receive
- Communication primitive are optimized to achieve high bandwidth and low latency over PCIe and NVLink high-speed interconnects within a node and over NVIDIA Mellanox Network across nodes

Imagenet Large Scale Visual Recognition (ILSVRC) Challenge Winners



Receptive field

- A convolutional layer operates over a local region of the input to that layer
- The effective **receptive field** of a convolutional layer is the size of the input region to the network that contributes to a layers' activations
- For example:
  - if the first convolutional layer has a receptive field of 3x3 then it's effective receptive field is also 3x3
  - However if the second layer also has a 3x3 filter, then it's (local) receptive field is 3x3, but it's effective receptive field is 5x5

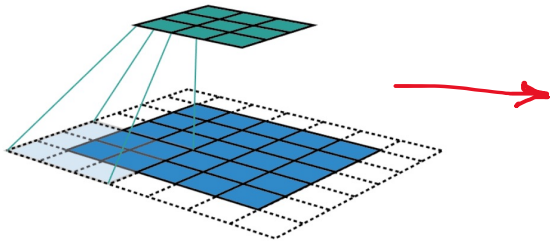


Receptive Field in a 2-d CNN

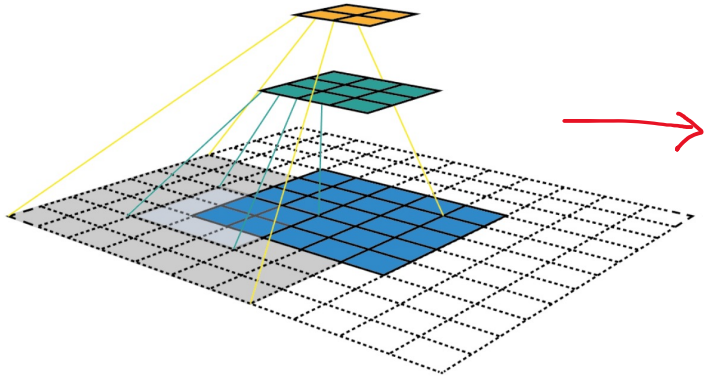
*The **receptive field** is defined as the region in the input space that a particular CNN's feature is looking at (i.e. be affected by)*

The common way to visualize a CNN feature map. Only looking at the feature map, we do not know where a feature is looking at (the center location of its receptive field) and how big is that region (its receptive field size). It will be impossible to keep track of the receptive field information in a deep CNN.

Convolution with kernel size k = 3x3, padding size p = 1x1, stride s = 2x2



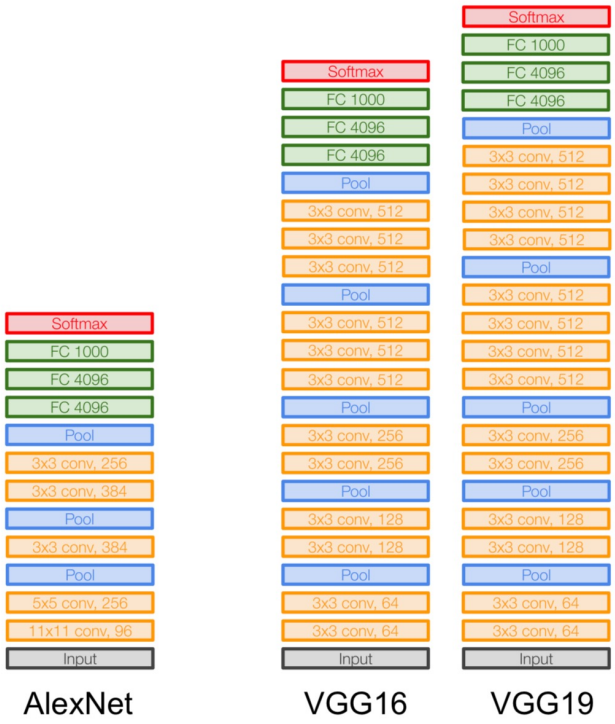
Applying the convolution on a 5x5 input map to produce the 3x3 green feature map



Applying the same convolution on top of the green feature map to produce the 2x2 orange feature map.

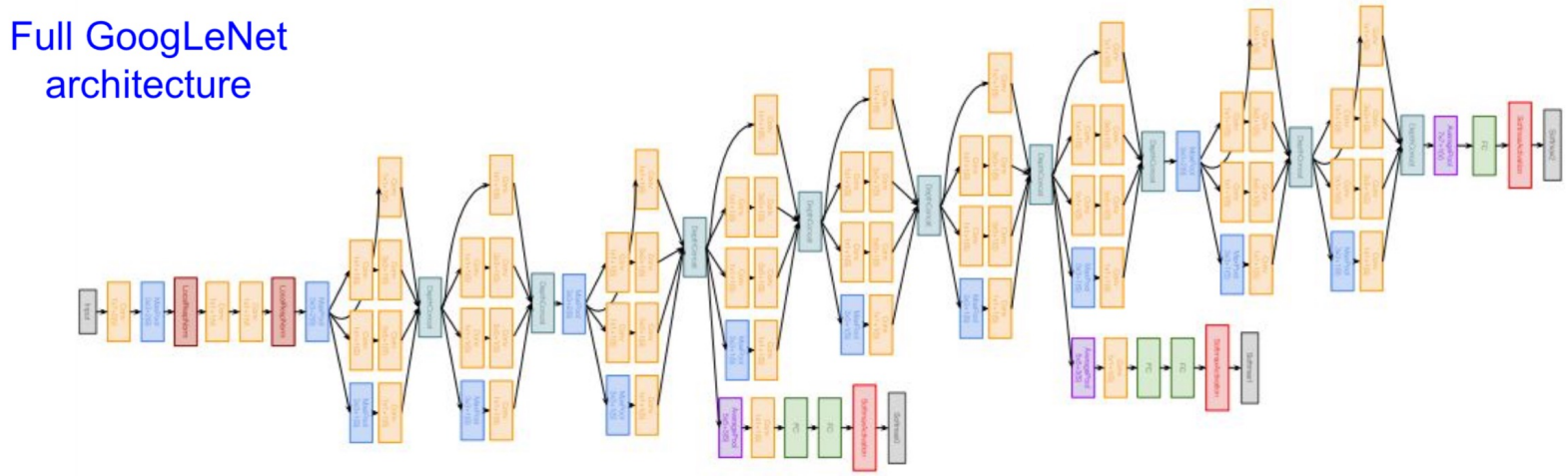
https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807

VGG Architecture



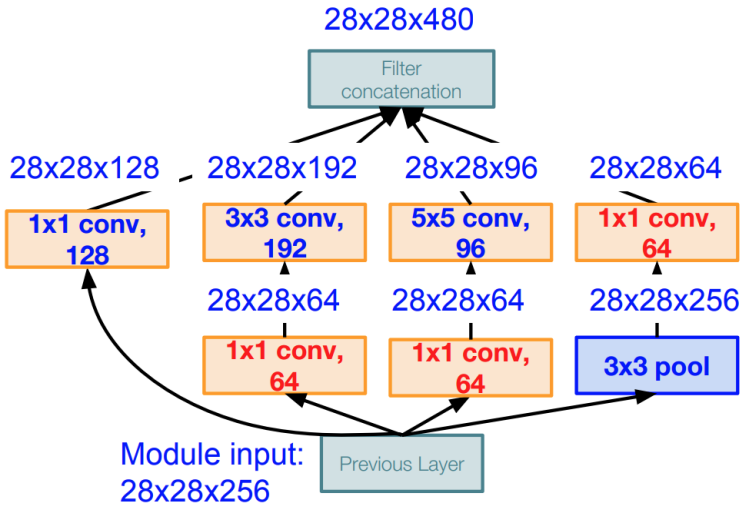
- Smaller filters (3x3, stride 1) compared to 11x11 and 5x5 in Alexnet
- Deeper nets: more layers compared to Alexnet (8 vs 16 or 19)
- Why smaller filters ?
  - Receptive field of 3 3x3 stacked conv. layers is same as a single 7x7 conv layer
  - More non-linearities with stack of smaller conv layers => makes decision function more discriminative
  - Lesser number of parameters: $3 * (3^2C^2)$ vs. 7^2C^2 for C channels per layer (55% less)

GoogleNet(2014)



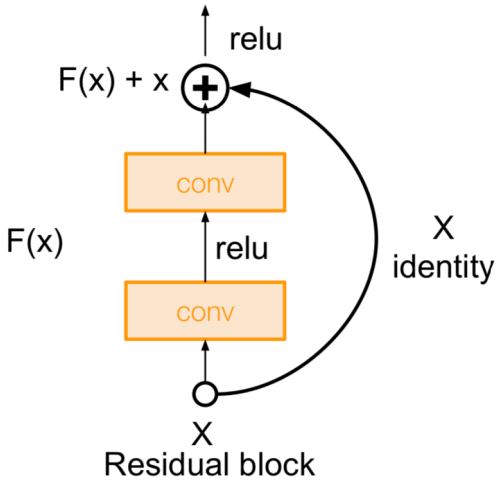
- Deeper network (22 layers)
- Computationally efficient “Inception” module
- Avoids expensive FC layers → uses average pooling
- 12x less params than AlexNet

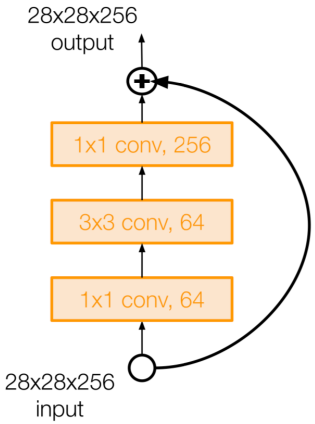
Inception module

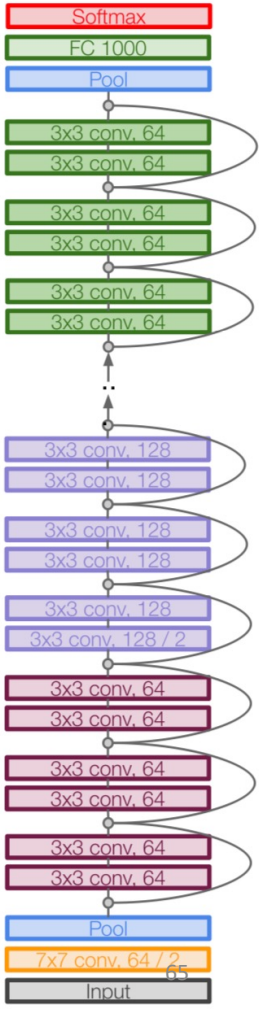


- Using same parallel layers as naive example, and adding 1x1 conv with 64 filter bottlenecks:
- Conv Ops:
  - [1x1 conv, 64] 28x28x64x1x1x256
  - [1x1 conv, 64] 28x28x64x1x1x256
  - [1x1 conv, 128] 28x28x128x1x1x256
  - [3x3 conv, 192] 28x28x192x3x3x64
  - [5x5 conv, 96] 28x28x96x5x5x64
  - [1x1 conv, 64] 28x28x64x1x1x256
- Total: 358M ops
  - Compared to 854M ops for naive version
- Bottleneck can also reduce depth after pooling layer (480 compared to 672)

ResNet

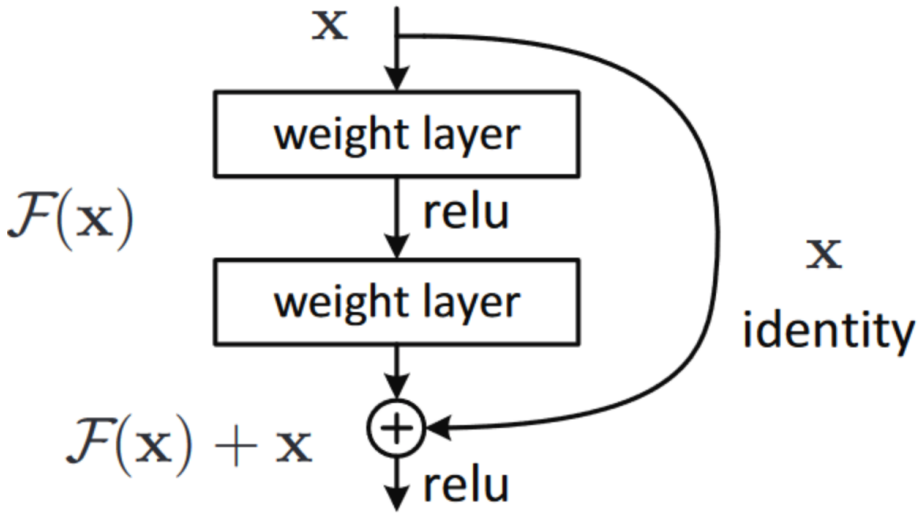






- Stack of residual blocks: each residual block has 2 3x3 conv layers; number of filters is doubled periodically
- Global average pooling layer after last conv layer
- Different depths: 34, 50, 101, 152
- Deeper network (ResNet50+) add 1x1 conv for computational efficiency

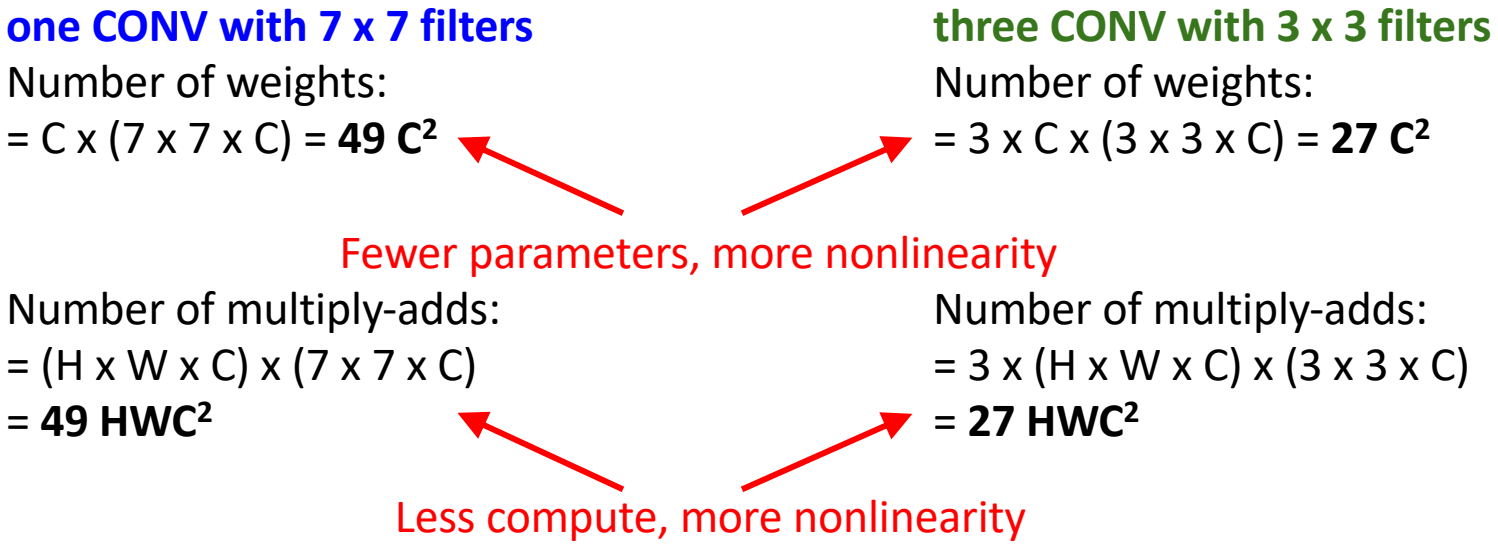
Residual connections prevents vanishing gradient



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H}\frac{\partial H}{\partial x} = \frac{\partial L}{\partial H}(\frac{\partial F}{\partial x} + 1) = \frac{\partial L}{\partial H}\frac{\partial F}{\partial x} + \frac{\partial L}{\partial H}$$

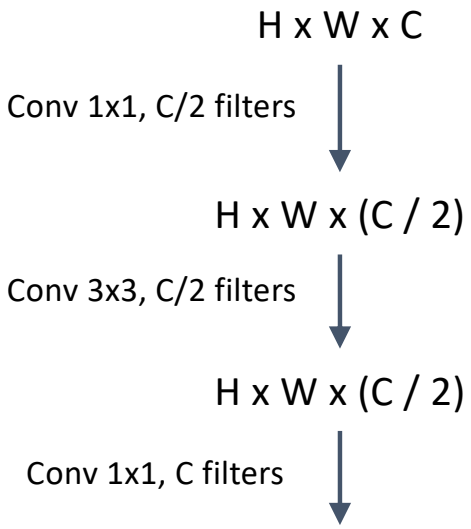
The power of small filters

- Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)



The power of small filters

- Why stop at 3 x 3 filters? → use 1 x 1!

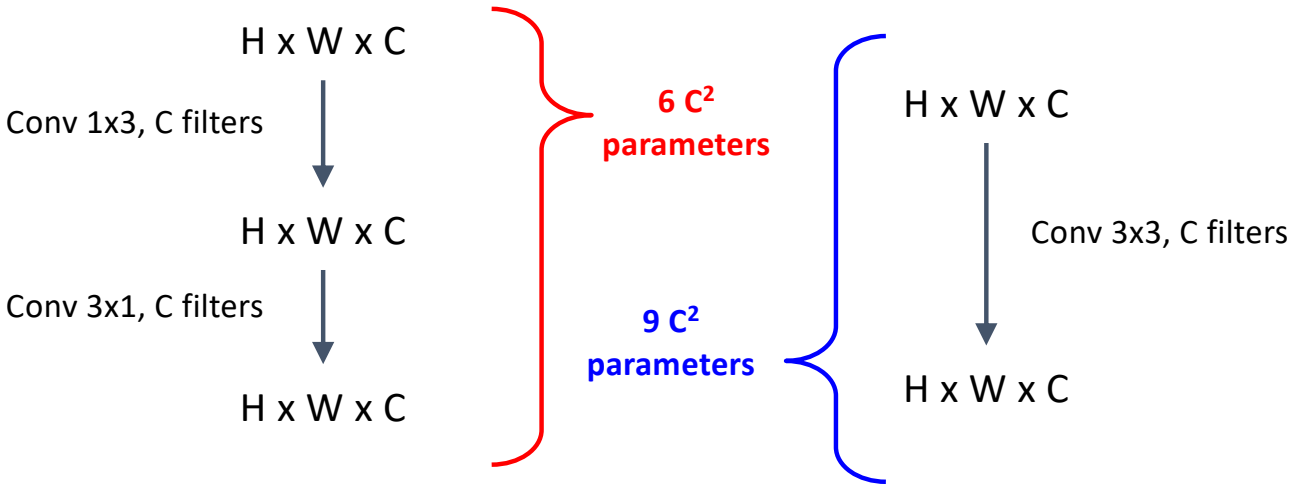


1. "bottleneck" 1 x 1 conv to reduce dimension
2. 3 x 3 conv at reduced dimension
3. Restore dimension with another 1 x 1 conv

[Seen in Lin et al, "Network in Network", GoogLeNet, ResNet]

The power of small filters

- Still using 3 x 3 filters … can we break it up?

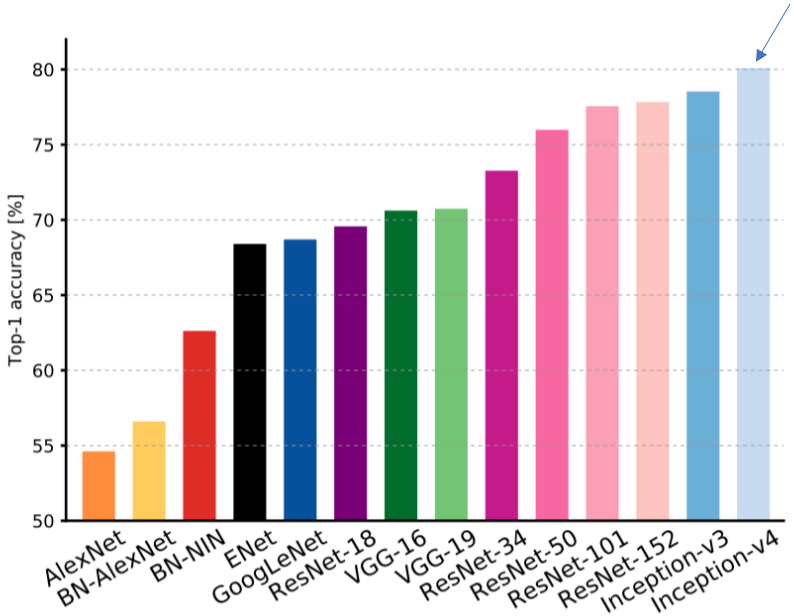


How to stack convolutions - Recap

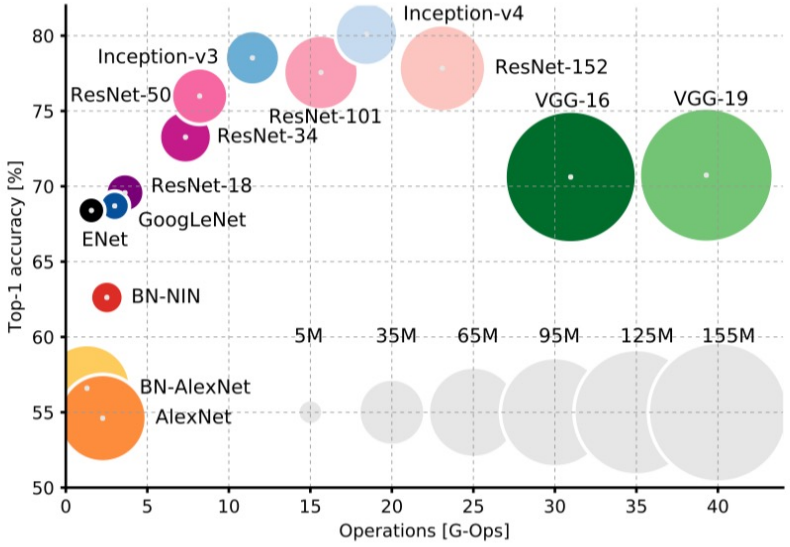
- Replace large convolutions (5 x 5, 7 x 7) with stacks of 3 x 3 convolutions
- 1 x 1 “bottleneck” convolutions are very efficient
- Replace N x N convolutions into 1 x N and N x 1
- All of the above give fewer parameters, less compute, more nonlinearity
- Szegedy et al, “Rethinking the Inception Architecture for Computer Vision”

Top1 Accuracy Comparison

Inception-v4: Resnet + Inception!

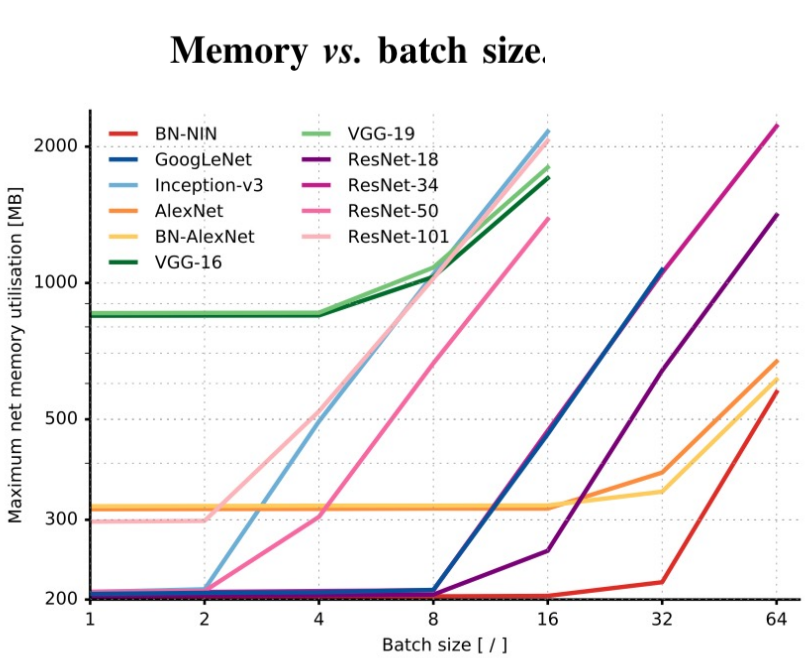


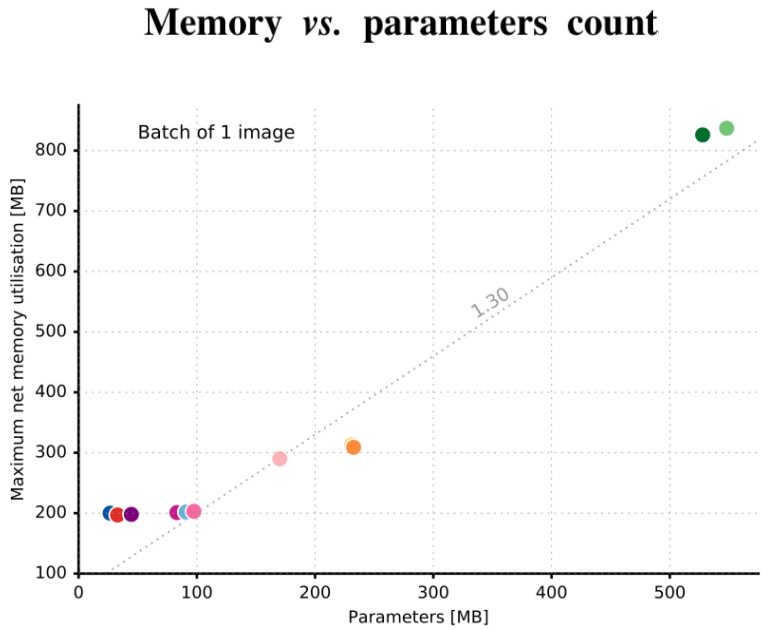
Top1 *vs.* network.



Top1 *vs.* operations, size $\propto$ parameters

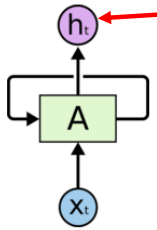
Memory Consumption



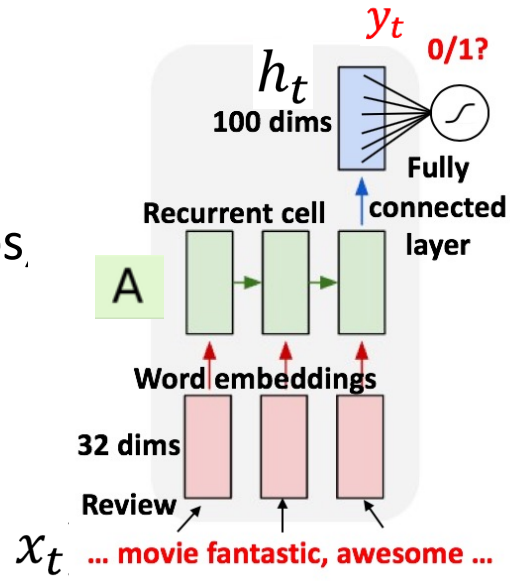


Recurrent Neural Networks

- Recurrent Neural Networks are networks with loops allowing information to persist.



Output is to predict a vector h_t, where $output\ y_t = \varphi(h_t)$ at some time steps (t)



Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, $A = f_W$, looks at some input x_t and outputs a value h_t. A loop allows information to be passed from one step of the network to the next.

new state — old state

$$h_t = f_W(h_{t-1}, x_t)$$

function with parameter W — Input vector at some time step

Recurrent Neural Networks Features

- The recurrent structure of RNNs enables the following characteristics:
  - Specialized for processing a sequence of values $x^{(1)}, \ldots, x^{(\tau)}$
    - Each value $x^{(i)}$ is processed with the **same network *A* that preserves past information**
  - Can scale to much **longer sequences** than would be practical for networks without a recurrent structure
    - Reusing network ***A*** reduces the required amount of parameters in the network
  - Can process **variable-length sequences**
    - The network complexity does not vary when the input length change
  - However, vanilla RNNs suffer from the training difficulty due to **exploding and vanishing gradients**.

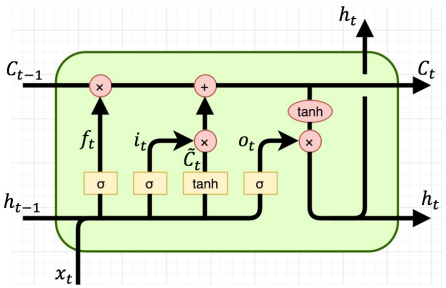
Pros and Cons of using RNNs

- Pros:
  - Possibility of processing input of any length
  - Model size not increasing with size of input
  - Computation takes into account historical information
  - Weights are shared across time
- Cons:
  - Computation being slow
  - Difficulty of accessing information from a long time ago
  - Cannot consider any future input for the current state
  - Suffers from vanishing and exploding gradient problem and hence training is unstable

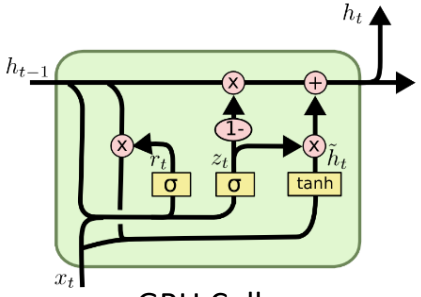
Networks with Memory

- Vanilla RNN operates in a “multiplicative” way (repeated tanh).
- Two recurrent cell designs were proposed and widely adopted:
  - **Long Short-Term Memory (LSTM)** (Hochreiter and Schmidhuber, 1997)
  - Gated Recurrent Unit (GRU) (Cho et al. 2014)
- Both designs process information in an “additive” way with gates to control information flow.
  - **Sigmoid gate outputs numbers between 0 and 1, describing how much of each component should be let through.**

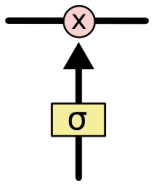
E.g. $f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right) = Sigmoid\left(W_f x_t + U_t h_{t\text{-}1} + b_f\right)$



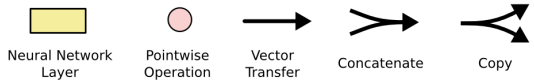
Standard LSTM Cell



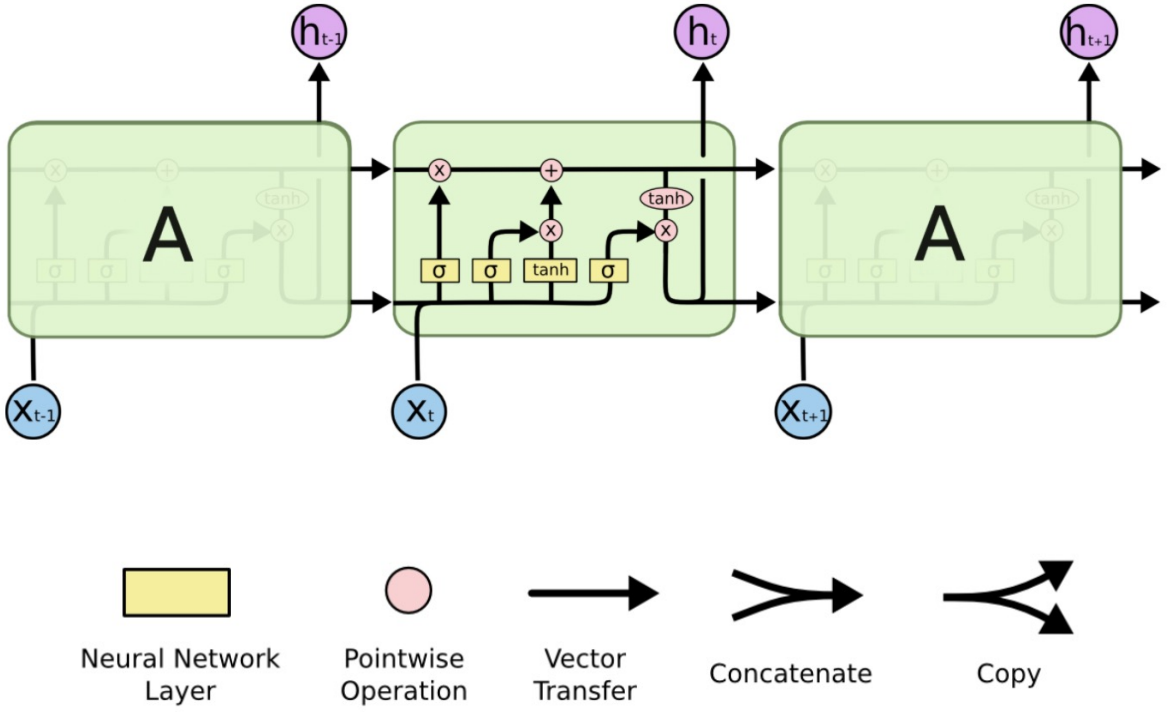
GRU Cell



A Sigmoid Gate



Long Short-Term Memory Networks (LSTM)



4 gates
4 fully connected layers
4 activation functions
4 math operations

Parameter calculation

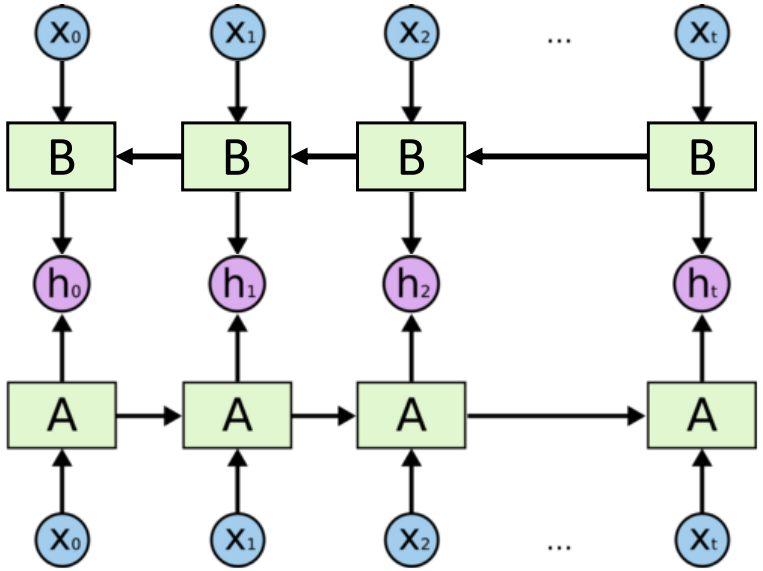
- Input to all FC networks is a concatenated vector of current input and hidden state vector

$$[h_{t-1}, x_t]$$

To calculate number of parameters. Each FC network has a parameter **weight** matrix of **[(m+n),n]** and a **bias** values of '**n**'. So total parameters at each FC network (m*n + sqr(n) + n) and for four FC networks it will be **4*(m*n + sqr(n) + n).**

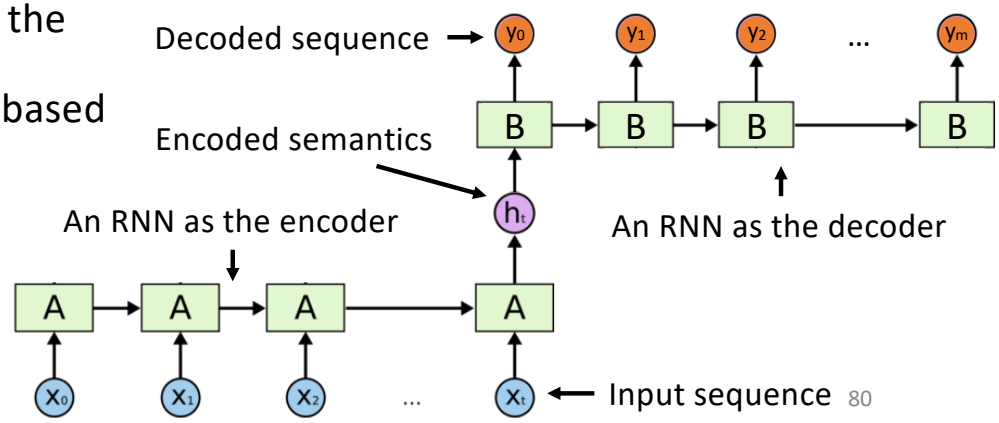
Sequence Learning with Multiple RNN Layers

- Bidirectional RNN
  - Connects two recurrent units (synced many-to-many model) of opposite directions to the same output.
  - Captures forward and backward information from the input sequence
  - Apply to data whose current state (e.g., h_0) can be better determined when given future information (e.g., $x_1, x_2, \ldots, x_t$)
    - E.g., in the sentence "the bank is robbed," the semantics of "bank" can be determined given the verb "robbed."



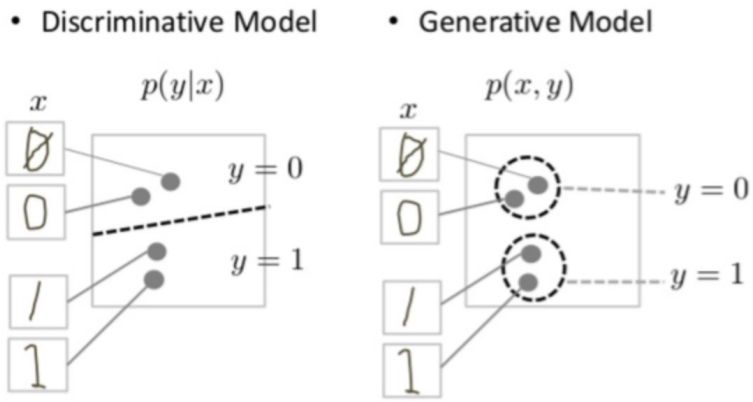
Sequence Learning with Multiple RNN Layers

- **Sequence-to-Sequence (Seq2Seq) model**
  - Developed by Google in 2018 for use in machine translation.
  - Seq2seq turns one sequence into another sequence. It does so by use of a recurrent neural network (RNN) or more often LSTM or GRU to avoid the problem of vanishing gradient.
  - The primary components are one Encoder and one Decoder network. The encoder turns each item into a corresponding hidden vector containing the item and its context. The decoder reverses the process, turning the vector into an output item, using the previous output as the input context.
  - **Encoder RNN**: extract and compress the semantics from the input sequence
  - **Decoder RNN**: generate a sequence based on the input semantics
  - Apply to tasks such as machine translation
    - Similar underlying semantics
    - E.g., "I love you." to "Je t'aime."



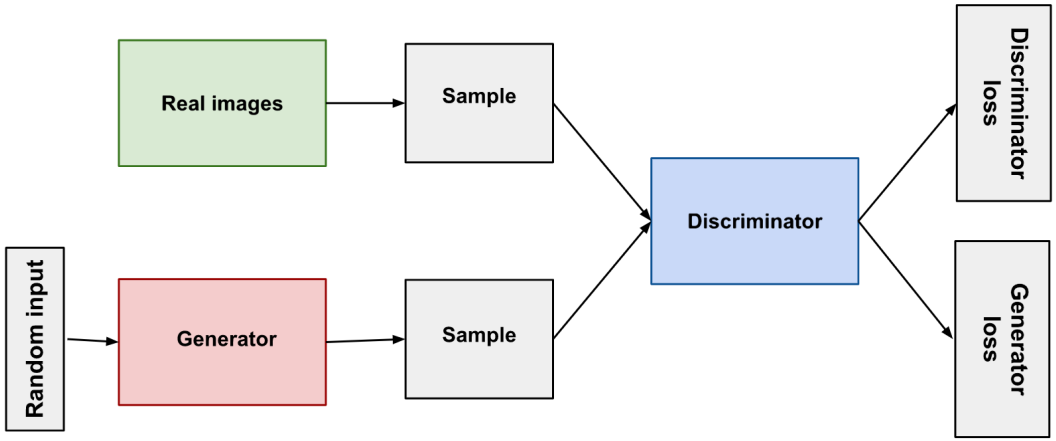
Generative vs Discriminative Models

- Generative models can generate new data instances.
- Discriminative models discriminate between different kinds of data instances.
- Generative models capture the joint probability p(X, Y), or just p(X) if there are no labels.
- Discriminative models capture the conditional probability p(Y | X).
- Discriminative models try to draw boundaries in the data space, while generative models try to model how data is placed throughout the space.



Generative Adversarial Networks (GANs)

- **Generator** learns to generate plausible data
- Generated instances are negative training examples for the discriminator
- Discriminator learns to distinguish the generator's fake data from real data.



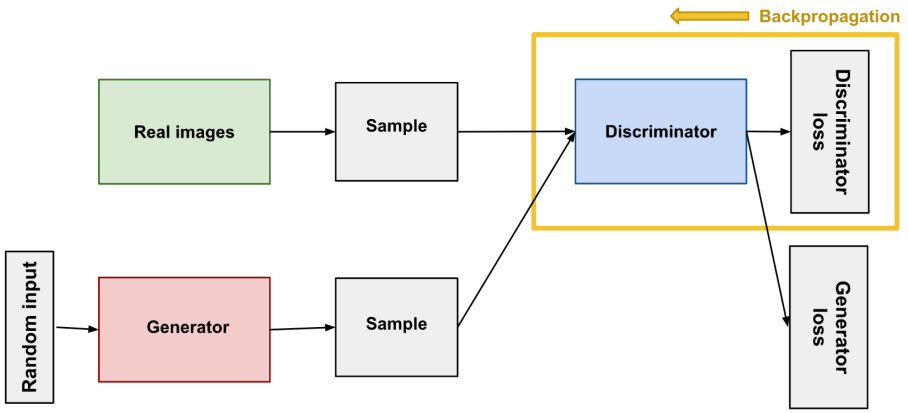
Discriminator Training

Discriminator is a classifier

- Distinguishes real data from data created by the generator

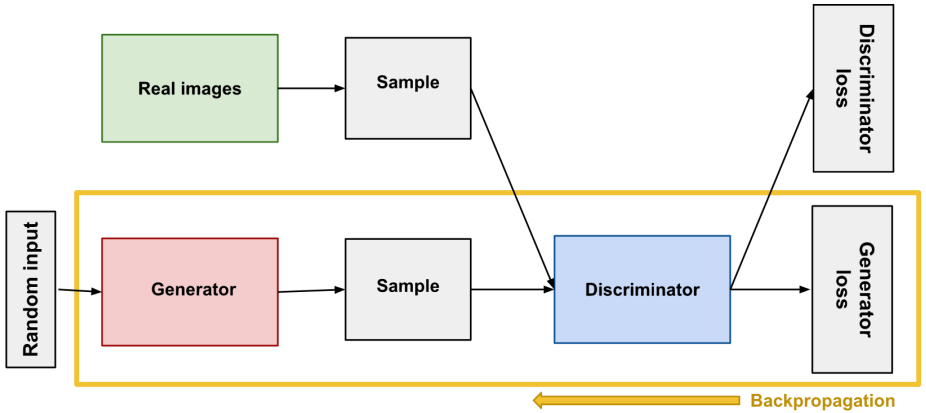
Discriminator training:

- Classifies both real data and fake data from the generator
- Discriminator loss penalizes the discriminator for misclassifying a real instance as fake or a fake instance as real.
- Discriminator updates its weights through backpropagation from the discriminator loss through the discriminator network.



Generator Training

- Generator is not directly connected to the loss
- Generator feeds into the discriminator which then produces the output
- Generator loss penalizes the generator for producing a sample that the discriminator network classifies as fake
- We don't want the discriminator to change during generator training (moving target)



GAN Training

- Alternating training of discriminator and generator
  1. The discriminator trains for one or more epochs.
  2. The generator trains for one or more epochs.
  3. Repeat steps 1 and 2 to continue to train the generator and discriminator networks.
- Convergence
  - As training progresses discriminator performance worsens
  - Eventually discriminator starts making random guesses
  - Generator get junk feedback

GAN Loss functions

- **Minimax loss**

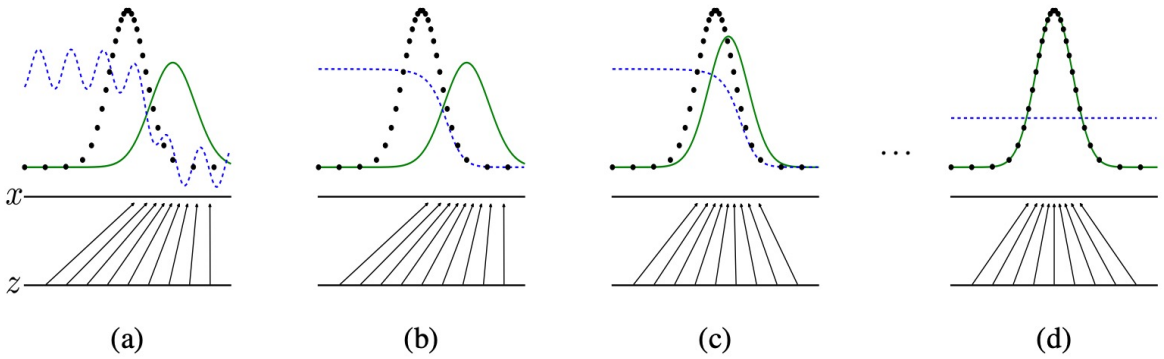
$$E_x[log(D(x))] + E_z[log(1 - D(G(z)))]$$

- `D(x)` is the discriminator's estimate of the probability that real data instance x is real.
- E_x is the expected value over all real data instances.
- `G(z)` is the generator's output when given noise z.
- `D(G(z))` is the discriminator's estimate of the probability that a fake instance is real.
- E_z is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).
- The formula derives from the cross-entropy between the real and generated distributions.

Min-max game

D and G play the following two-player minimax game with value function $V(G, D)$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$



GAN Training as Stochastic Gradient Descent

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Prepare for Lecture 3

- Get ready with your cloud setup
- HW1 (based on material in Lecture 1 and 2) will be released soon
- Start working on your project proposal
- Lecture 3 will cover ML on cloud platforms, TorchX, Ray, and scheduling on DL clusters