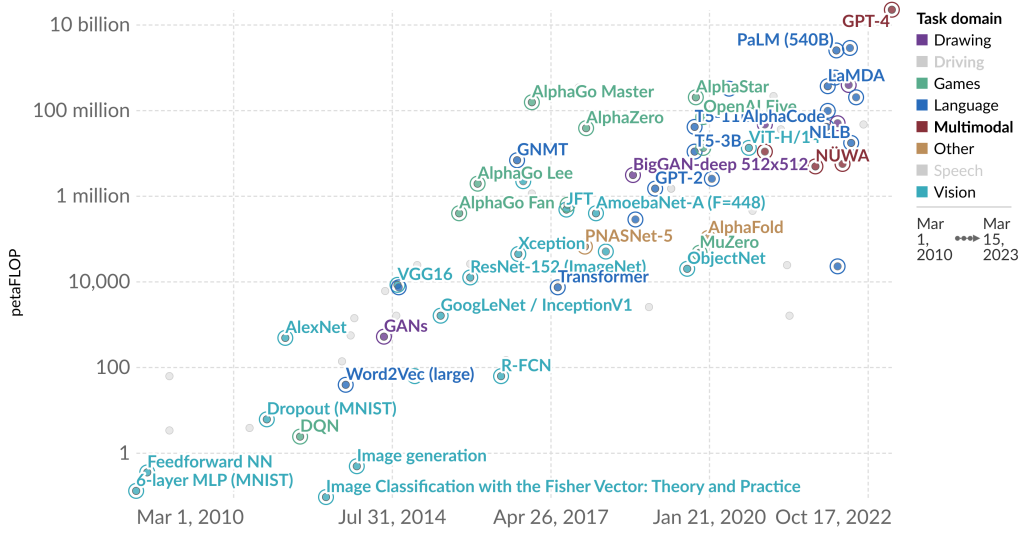
DAY 2: TOWARDS SCALABLE DEEP LEARNING
Distributed Training and Data Parallelism

2023-05-09 | Jenia Jitsev | Scalable Learning & Multi-Purpose AI Lab, Helmholtz AI, LAION @ JSC

JÜLICH
Forschungszentrum

DEEP LEARNING & DISTRIBUTED TRAINING

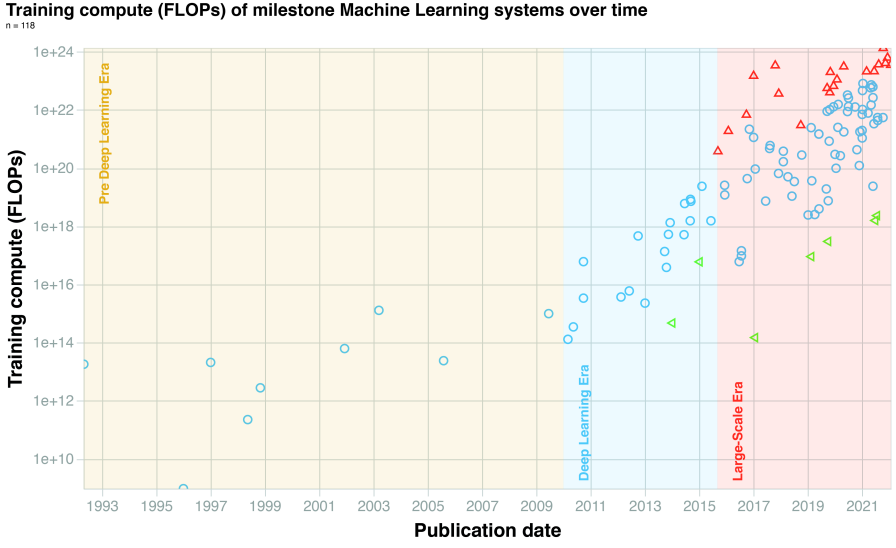
- Training models that solve complex, real world tasks requires large model and data scale



Sevilla et al., 2023; OurWorldInData.org/artificial-intelligence; CC BY Source

DEEP LEARNING & DISTRIBUTED TRAINING

- Compute demand increases exponentially, 3.4 months doubling time since 2012

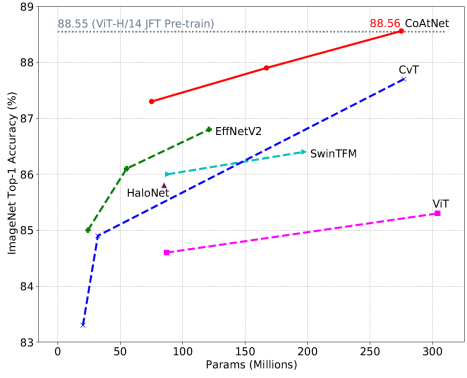


Sevilla et al., arXiv:2202.05924, 2023

DEEP LEARNING & DISTRIBUTED TRAINING

- Networks: large models, many layers, large number of parameters (weights)
  - Vision: Convolutional, Transformer and Hybrid networks
  - hundreds of layers, hundred millions of parameters (currently up to 20B)

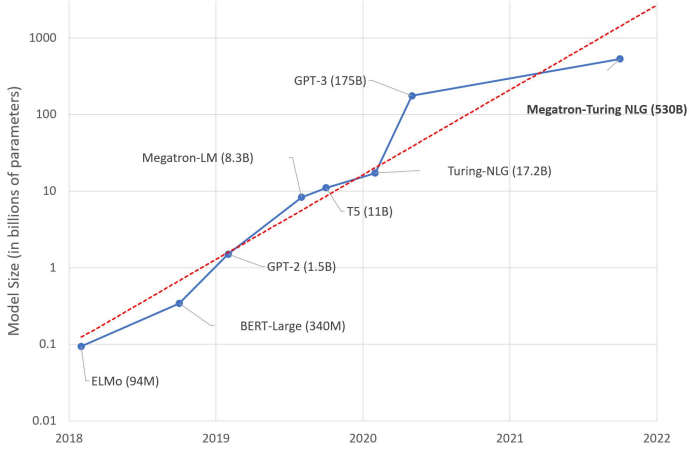
Models	Eval Size	#Params	#FLOPs	TPUv3-core-days	Top-1 Accuracy
ResNet + ViT-L/16	384^2	330M	-	-	87.12
ViT-L/16	512^2	307M	364B	0.68K	87.76
ViT-H/14	518^2	632M	1021B	2.5K	88.55
NFNet-F4+	512^2	527M	367B	1.86K	89.2
CoAtNet-3†	384^2	168M	114B	0.58K	88.52
CoAtNet-3†	512^2	168M	214B	0.58K	88.81
CoAtNet-4	512^2	275M	361B	0.95K	89.11
CoAtNet-5	512^2	688M	812B	1.82K	89.77
ViT-G/14	518^2	1.84B	5160B	>30K°	90.45
CoAtNet-6	512^2	1.47B	1521B	6.6K	90.45
CoAtNet-7	512^2	2.44B	2586B	20.1K	**90.88**



Dai et al, NeurIPS, 2021

DEEP LEARNING & DISTRIBUTED TRAINING

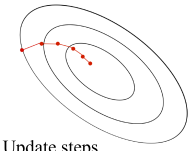
- Networks: large models, many layers, large number of parameters (weights)
  - Language: Transformer networks
  - hundreds of layers, billions of parameters (GPT-3: 175 Billion)



Narayanan et al, 2021

DEEP LEARNING & DISTRIBUTED TRAINING

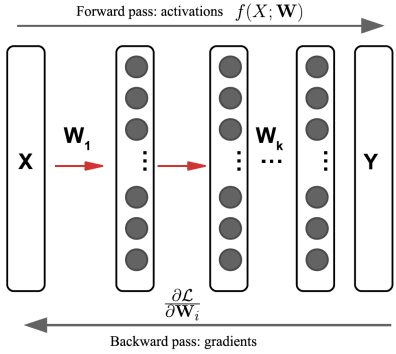
- GPT-3: 175 billion weights, $\approx$ 350 GB, does not fit on single GPU (A100: 40/80 GB)
- ResNets, ViT $<$ 1B weights, $\lesssim$ 40 GB, fit on single GPU
  - depending on chosen resolution of input X and batch size $|B|$!

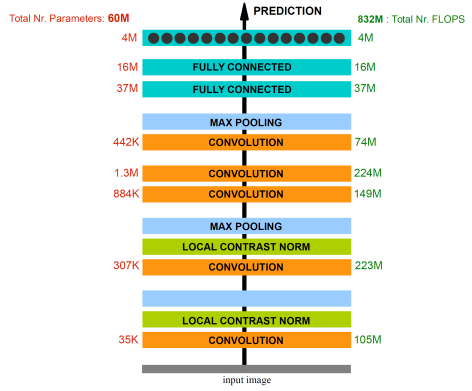


Update steps

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \nabla_{\mathbf{W}} \mathcal{L}_B$$

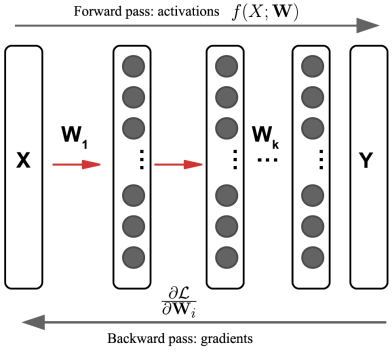
$$\mathcal{L}_{\mathbf{W_{t+1}}} \leq \mathcal{L}_{\mathbf{W_t}} \leq \cdots$$

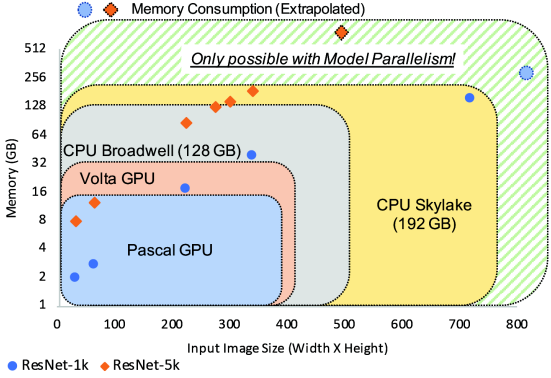




DEEP LEARNING & DISTRIBUTED TRAINING

- GPT-3: 175 billion weights, $\approx$ 350 GB, does not fit on single GPU
- ResNets, ViT $<$ 1B weights, $\lesssim$ 40 GB, fit on single GPU
  - depending on chosen resolution of input X and batch size $|B|$!

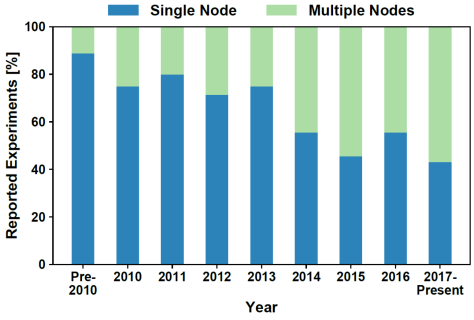


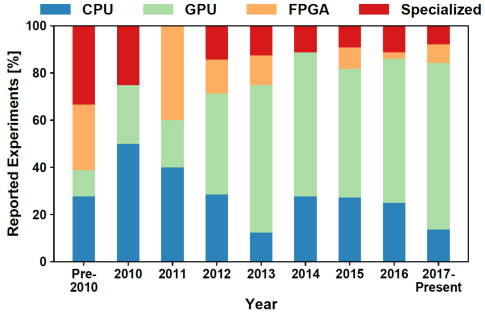


Awan et al., 2020

DISTRIBUTED TRAINING

- Use the computational power and memory capacity of multiple nodes of a large machine
- Requires taking care of internode communication

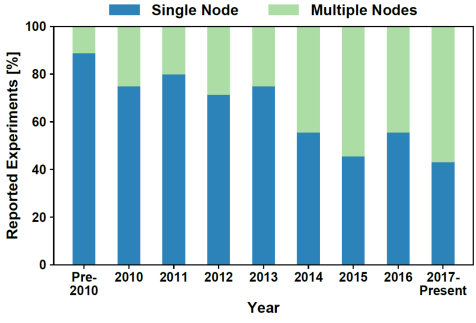


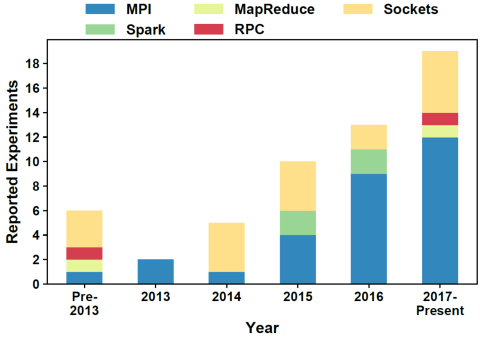


Ben-Nun & Hoefler, 2018

DISTRIBUTED TRAINING

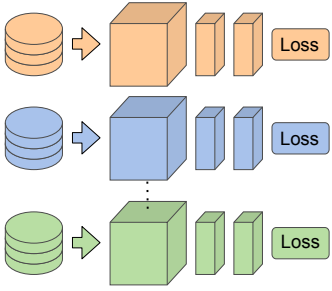
- Use the computational power and memory capacity of multiple nodes of a large machine
- Requires taking care of internode communication
- Requires high bandwidth interconnect between the compute nodes
  - HPC: InfiniBand (4× Mellanox 200 Gb/s cards on JUWELS Booster per node)
  - Not available on conventional clusters!

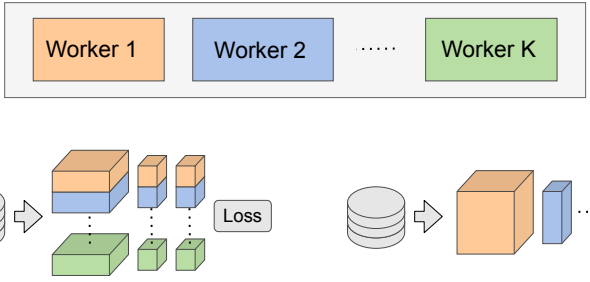




DISTRIBUTED TRAINING

- Depending on whether full model fits on a single GPU, different schemes
  - data parallelism: split only data across GPUs, model cloned on each GPU
  - model parallelism: split within layers across GPUs
  - pipeline parallelism: split layer groups across GPUs

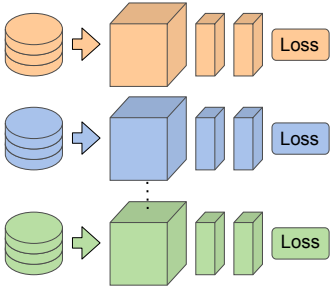


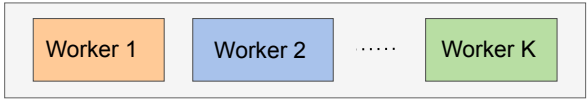


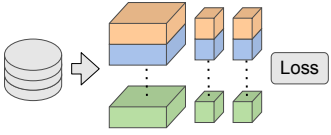
Laskin et al., 2020

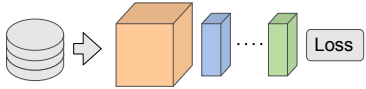
DISTRIBUTED TRAINING

- Model does not fit on single GPU: no training without parallelization possible at all
  - AlexNet in 2012; GPT-3; CLIP ViT G/14, . . .
- Model fits on single GPU: why distributed training?
  - multiple GPUs can drastically speed up training phase → **data parallelism**
    - e.g. ImageNet training: from days to hours or minutes







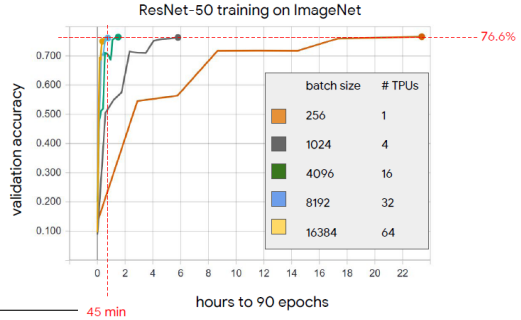


Laskin et al., 2020

DISTRIBUTED TRAINING

- ImageNet distributed training: from days, to hours, to minutes

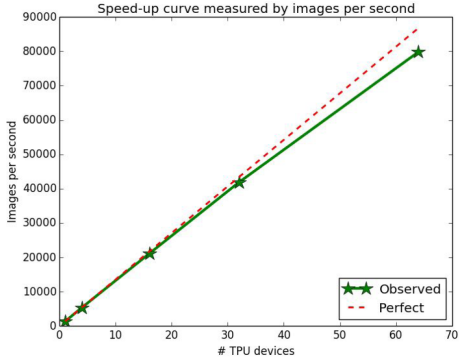
	Batch Size	Processor	DL Library	Time	Accuracy
He et al. [1]	256	Tesla P100 × 8	Caffe	29 hours	75.3 %
Goyal et al. [2]	8,192	Tesla P100 × 256	Caffe2	1 hour	76.3 %
Smith et al. [3]	8,192 → 16,384	full TPU Pod	TensorFlow	30 mins	76.1 %
Akiba et al. [4]	32,768	Tesla P100 × 1,024	Chainer	15 mins	74.9 %
Jia et al. [5]	65,536	Tesla P40 × 2,048	TensorFlow	6.6 mins	75.8 %
Ying et al. [6]	65,536	TPU v3 × 1,024	TensorFlow	1.8 mins	75.2 %
Mikami et al. [7]	55,296	Tesla V100 × 3,456	NNL	2.0 mins	75.29 %
This work	**81,920**	**Tesla V100 × 2,048**	**MXNet**	**1.2 mins**	**75.08%**



Yamazoto et al., 2019; Ying, 2018

DEEP LEARNING WITH DATA PARALLELISM

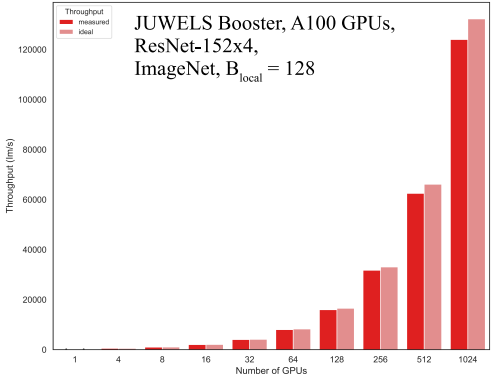
- Data parallelism: simple approach for efficient distributed training
  - whole network model **has to** fit on one GPU: depends on batch size!
  - split whole dataset across multiple workers
  - speeds up model training – given good scaling and well-tuned learning
- Faster training, shorter experiment cycle – more opportunities to test new ideas and models

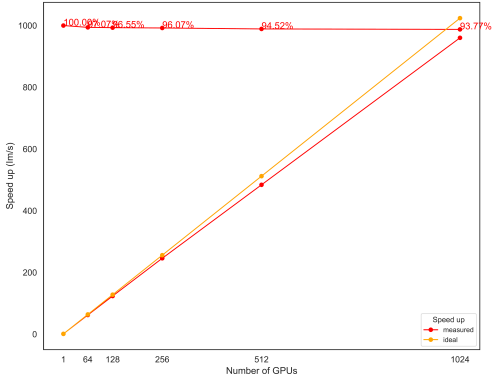


Ying, 2018

DEEP LEARNING WITH DATA PARALLELISM

- Data parallelism: simple approach for efficient distributed training
  - whole network model **has to** fit on one GPU: depends on batch size!
  - split whole dataset across multiple workers
  - good scaling: necessary (but not sufficient!) for model training speed up
- Faster training, shorter experiment cycle – more opportunities to test new ideas and models

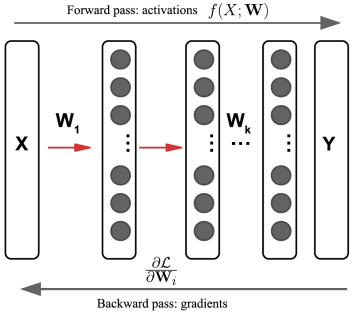


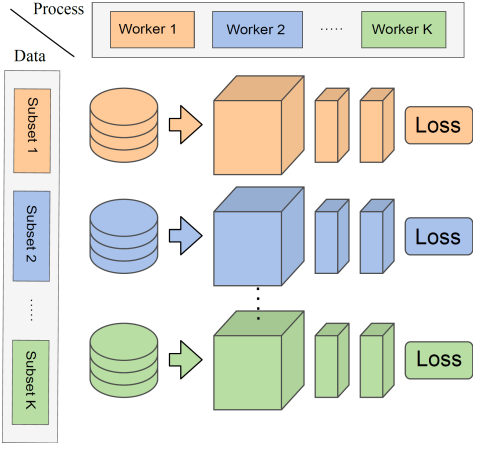


Cherti and Jitsev, arXiv:2106.00116, MedNeurIPS Workshop, 2021

DEEP LEARNING WITH DATA PARALLELISM

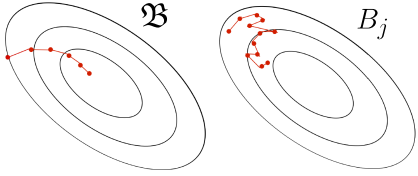
- Data parallelism: simple approach for efficient distributed model training
  - same model is cloned across *K* workers
  - each model clone trains on its dedicated subset of total available data
  - synchronous (S-SGD) or asynchronous (A-SGD) optimization; central (parameter server) or decentralized (ring-AllReduce) communication for weight updates

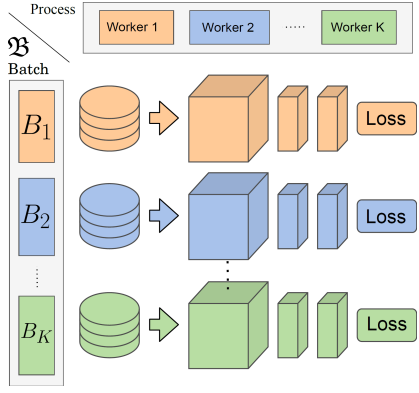




DEEP LEARNING WITH DATA PARALLELISM

- Data parallelism: simple approach for efficient distributed model training
  - can be understood as training a model using a larger mini-batch size $|\mathfrak{B}|$
    - $\mathfrak{B} = B_1 \cup \ldots \cup B_K$, $B_i \cap B_j = \emptyset$, $\forall i, j \in K$ workers
    - $|\mathfrak{B}| = K \cdot |B_{\text{ref}}|$, where $|B_{\text{ref}}| = n$ is original, reference batch size for a single worker



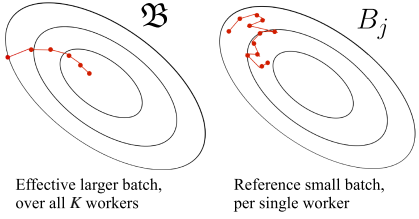


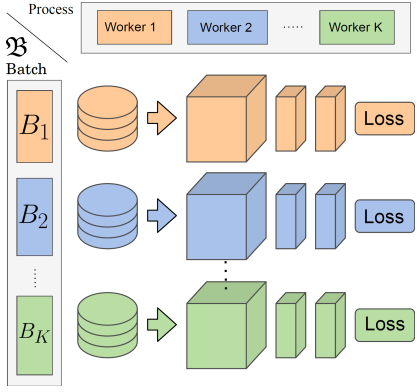
REMINDER: MINI-BATCH SGD

- Mini-batch SGD
  - perform an update step using loss gradient $\nabla_{\mathbf{W}}\mathcal{L}_B$ over a **mini-batch** of size $|B| = n \ll N$

$$\nabla_{\mathbf{W}}\mathcal{L}_B = \nabla_{\mathbf{W}}\frac{1}{n}\sum_{X_i \in B}\mathcal{L}_i$$

  - update step: $\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta\nabla_{\mathbf{W}}\mathcal{L}_B$



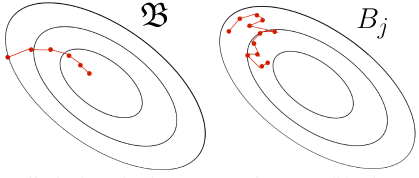


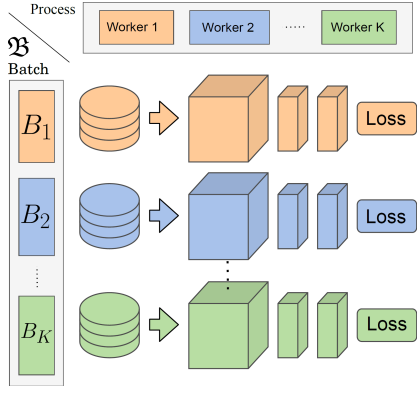
DEEP LEARNING WITH DATA PARALLELISM

- **Effective larger mini-batch** $\mathfrak{B}$ over K workers
  - perform an update step using loss gradient $\nabla_{\mathbf{W}}\mathcal{L}_{\mathfrak{B}}$ over a larger **effective** mini-batch $|\mathfrak{B}| = K \cdot |B_{\text{ref}}|,\ |B| = n \ll N$

$$\nabla_{\mathbf{W}}\mathcal{L}_{\mathfrak{B}} = \nabla_{\mathbf{W}}\frac{1}{K}\sum_{j=1}^{K}\frac{1}{n}\sum_{X_i \in B_j}\mathcal{L}_i = \nabla_{\mathbf{W}}\frac{1}{nK}\sum_{X_i \in \mathfrak{B}}\mathcal{L}_i$$

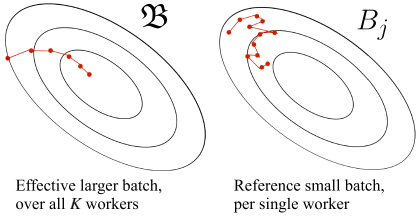
  - update step: $\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta\nabla_{\mathbf{W}}\mathcal{L}_{\mathfrak{B}}$

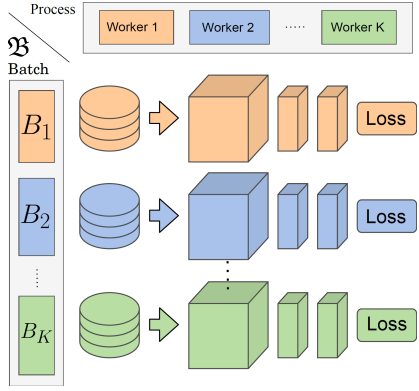




DEEP LEARNING WITH DATA PARALLELISM

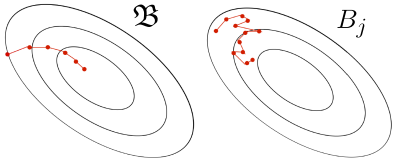
- Training a model using a larger mini-batch size $|\mathfrak{B}|$
  - $|\mathfrak{B}| = K \cdot |B_{\text{ref}}|$, where $|B_{\text{ref}}|$ is original, reference batch size for a single worker
  - Update step: $\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \nabla_{\mathbf{W}} \mathcal{L}_{\mathfrak{B}}$
  - **Reminder:** Changes optimization trajectory and weight dynamics compared to smaller mini-batch training

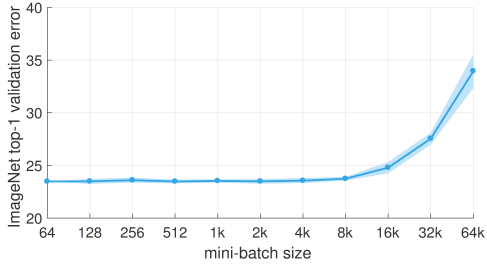




DEEP LEARNING WITH DATA PARALLELISM

- Training a model using a larger mini-batch size $|\mathfrak{B}|$
  - $|\mathfrak{B}| = K \cdot |B_{\text{ref}}|$, where $|B_{\text{ref}}|$ is original, reference batch size for a single worker
  - Update step: $\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \nabla_{\mathbf{W}} \mathcal{L}_{\mathfrak{B}}$
  - **Reminder:** Changes optimization trajectory and weight dynamics compared to smaller mini-batch training



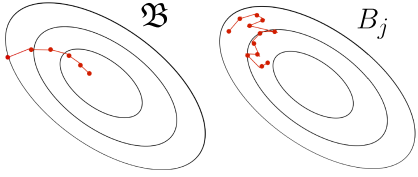


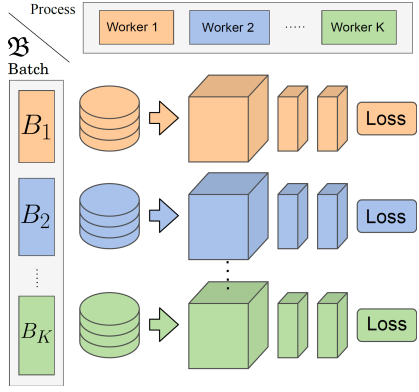
Goyal et al., 2017

DEEP LEARNING WITH DATA PARALLELISM

- Data parallel distributed training requires:
  - **proper data feeding** for each worker
  - setting up workers, one per each GPU (model clones)
  - **sync of model clone parameters** (weights) across workers: update step – **communication load**
    - after each forward/backward pass on workers' mini-batches

$$- \nabla_{\mathbf{w}} \mathcal{L}_{\mathfrak{B}} = \underbrace{\frac{1}{K} \sum_{j=1}^{K}}_{\text{across } K \text{ workers}} \underbrace{\nabla_{\mathbf{w}} \frac{1}{n} \sum_{X_i \in B_j} \mathcal{L}_i}_{\text{on worker } j}$$





DEEP LEARNING WITH DATA PARALLELISM

- Data parallelism: proper data feeding for each worker
  - important not to let GPUs “starve” while training

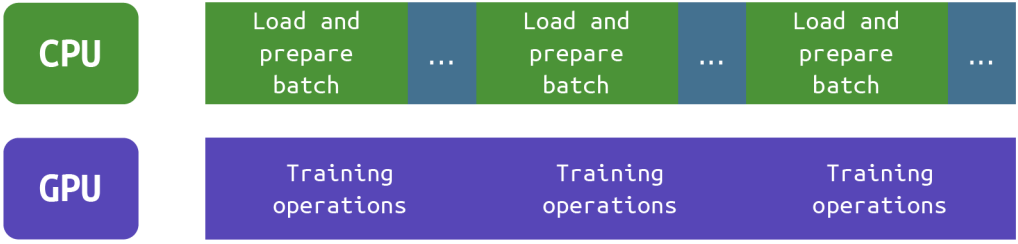


Figure: Mendonça, Sarmiento, ETH CSCS, 2020

DEEP LEARNING WITH DATA PARALLELISM

- Data parallelism: proper data feeding for each worker
  - important not to let GPUs “starve” while training

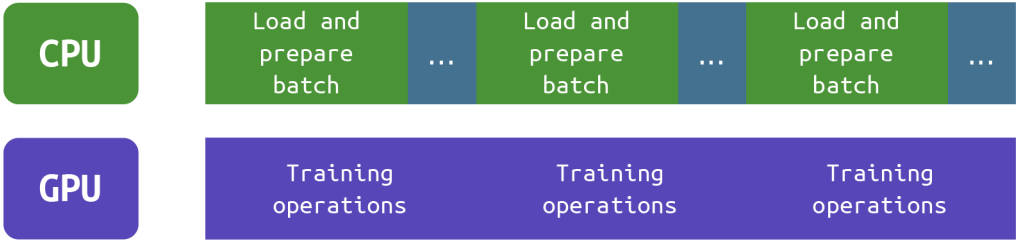
DEEP LEARNING WITH DATA PARALLELISM

- Data parallelism: proper data feeding for each worker
  - data pipelines: handled either by
    - internal TensorFlow (see tutorial) or PyTorch routines
    - specialized libraries, e.g. NVIDIA DALI, WebDataset

```
# Example for TensorFlow dataset API
import tensorflow as tf

[...]

# Instantiate a dataset object
dataset = tf.data.Dataset.from_tensor_slices(files)

[...]

# Apply input preprocessing when required
dataset = dataset.map(decode, num_parallel_calls=tf.data.AUTOTUNE)
dataset = dataset.map(preprocess, num_parallel_calls=tf.data.AUTOTUNE)

# Randomize
dataset = dataset.shuffle(buffer_size)

# Create a batch and prepare next ones
dataset = dataset.batch(batch_size)
dataset = dataset.prefetch(tf.data.AUTOTUNE)

[...]
```

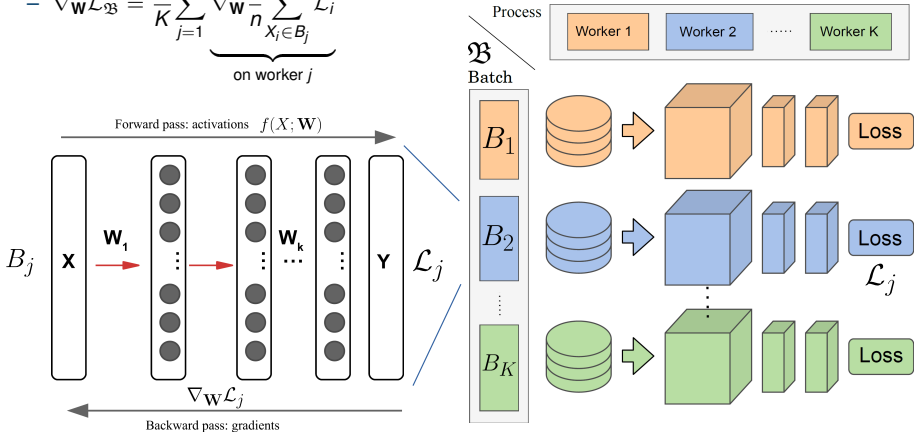
DEEP LEARNING WITH DATA PARALLELISM

- Data parallelism: proper data feeding for each worker
  - important not to let GPUs “starve” while training
  - data handling via **data pipeline** routines
  - use efficient **data containers**: HDF5, LMDB, TFRecords, WebDataset, . . .



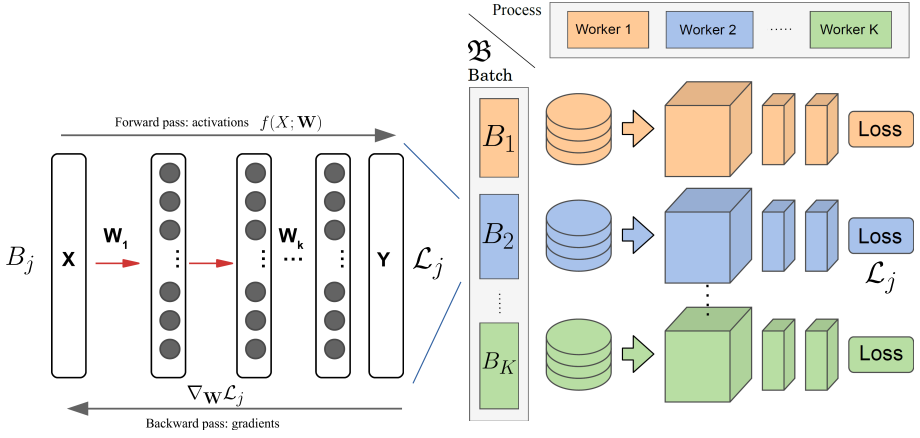
DEEP LEARNING WITH DATA PARALLELISM

- Data parallel distributed training requires:
  - proper data feeding for each worker: data pipelines, containers
  - setting up workers, one per each GPU (model clones)
  - **sync of model clone weights** across workers: update step $\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \eta \nabla_{\mathbf{W}} \mathcal{L}_{\mathfrak{B}}$
    - after each forward/backward pass on workers' mini-batches
    - $\nabla_{\mathbf{W}} \mathcal{L}_{\mathfrak{B}} = \frac{1}{K} \sum_{j=1}^{K} \underbrace{\nabla_{\mathbf{W}} \frac{1}{n} \sum_{X_i \in B_j} \mathcal{L}_i}_{\text{on worker } j}$



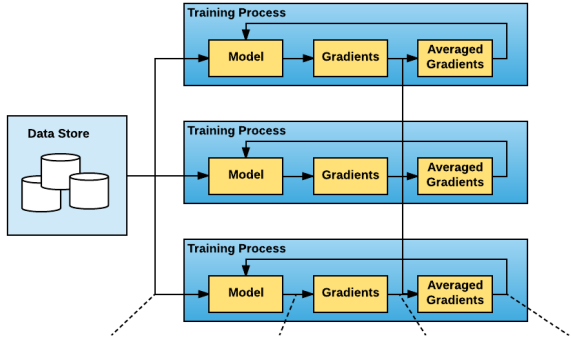
DEEP LEARNING WITH DATA PARALLELISM

- Data parallel distributed training requires:
  - proper data feeding for each worker: data pipelines, containers
  - **sync of model clone weights** across workers: handle communication between nodes
    - for large *K* and large model size – high bandwidth required! Enter stage **InfiniBand** – HPC
    - efficient internode communication while training on GPUs! Enter stage **Horovod**, **PyTorch DDP**, . . .



DEEP LEARNING WITH DATA PARALLELISM: HOROVOD

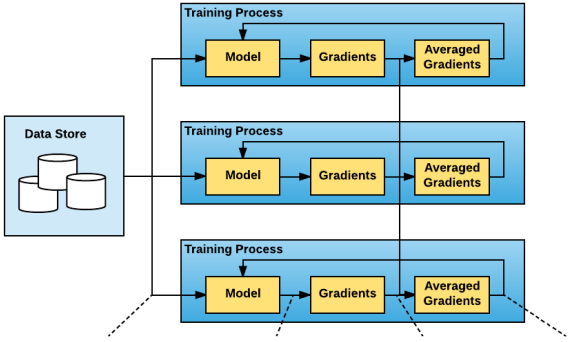
- Horovod: making data parallel distributed training easy
  - efficient worker communication during distributed training
    - synchronous, decentralized (**no** parameter servers, ring-AllReduce)
    - additional mechanisms like Tensor Fusion
  - works seamlessly with job managers (Slurm)
  - very easy code migration from a working single-node version



Sergeev, A., Del Balso, M., 2017

DEEP LEARNING WITH DATA PARALLELISM: HOROVOD

- Supports major libraries: TensorFlow, PyTorch, Apache MXNet
- Worker communication during distributed training
  - NCCL: highly optimized GPU-GPU communication collective routines
    - same as in MPI: `Allreduce`, `Allgather`, `Broadcast`
  - MPI: for CPU-CPU communication
  - Simple scheme: 1 worker – 1 MPI process
  - Process nomenclature as in MPI: `rank`, `world_size`
  - for local GPU assignment: `local_rank`

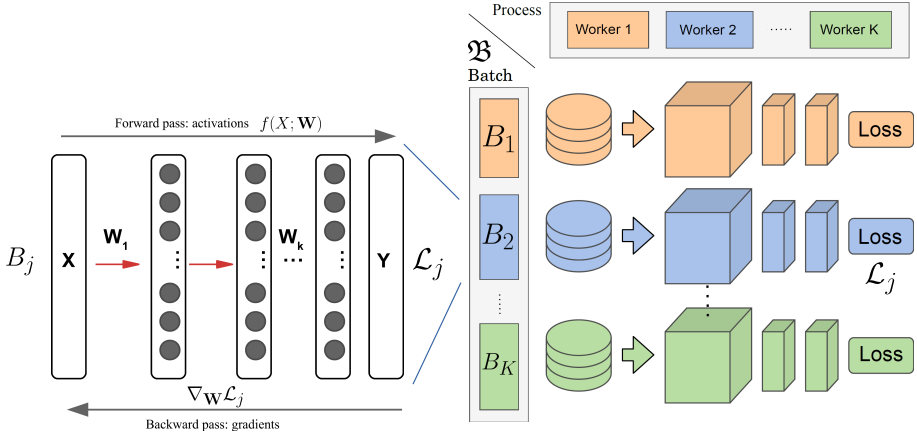


DEEP LEARNING WITH DATA PARALLELISM: HOROVOD

- Horovod: highly optimized library for data parallel distributed training
  - Name origin: “horovod” stands for east slavic (Ukraine, Russia, Bulgaria, Belarus, Poland, . . .) circle dance form

DISTRIBUTED TRAINING WITH HOROVOD

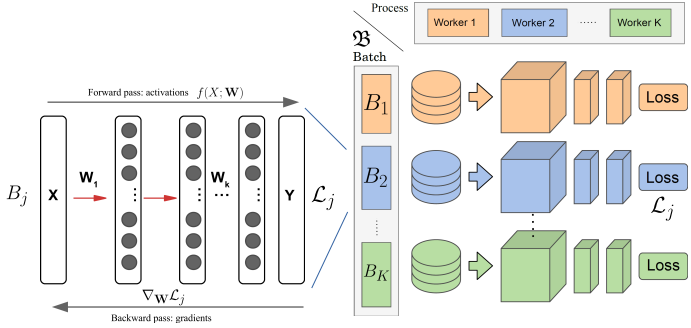
- Training model with a large effective mini-batch size:
  - $\mathfrak{B} = \bigcup_{i \le K} B_i,\ B_i \cap B_j = \emptyset,\ \forall i, j \in K;\ |\mathfrak{B}| = K \cdot |B_{\text{ref}}|$
  - B_{ref} is reference batch size for single worker



DISTRIBUTED TRAINING WITH HOROVOD

- Training loop: *K* workers, one per each GPU

```
init: sync weights of all K workers
for e in epochs:
  shard data subsets D_j to workers j
  for B in batches:
    each worker j gets its own B_j (local compute)
    each worker j computes its own dL_j (local compute)
    Allreduce: compute dL_B, average gradients (communication)
    Update using dL_B for all K workers (local compute)
```



DISTRIBUTED TRAINING WITH HOROVOD

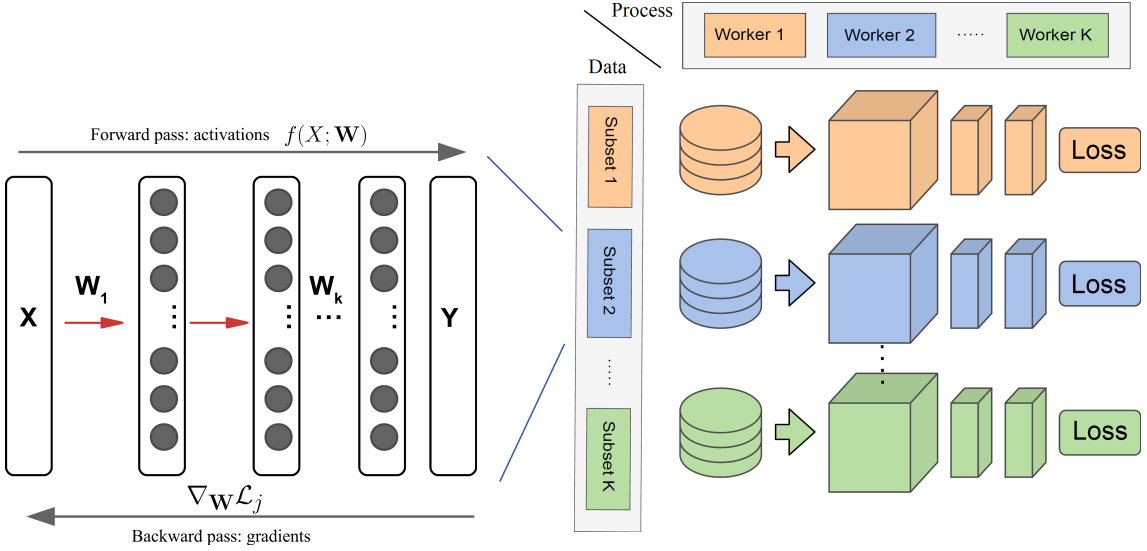
- User friendly code migration, simple wrapping of existing code
  - major libraries supported: TensorFlow, PyTorch, MXNet, . . .

```
import tensorflow as tf
import horovod.tensorflow.keras as hvd

# Initialize Horovod
hvd.init()

[...]

# Wrap optimizer in Horovod's
    DistributedOptimizer
opt = hvd.DistributedOptimizer(opt)

[...]
```

```
import torch
import horovod.torch as hvd

# Initialize Horovod
hvd.init()

[...]

# Wrap optimizer in Horovod's
    DistributedOptimizer
opt = hvd.DistributedOptimizer(opt)

[...]
```

DISTRIBUTED TRAINING WITH HOROVOD

- Handled by dataset pipeline (Horovod independent): data sharding



DISTRIBUTED TRAINING WITH HOROVOD

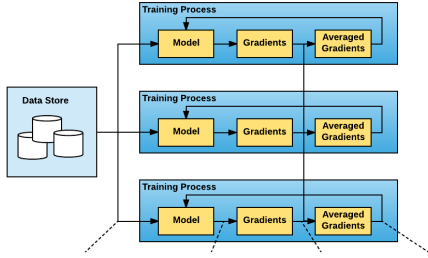
- Handled by dataset pipeline (Horovod independent): data sharding

```
# Example for TensorFlow dataset API
import tensorflow as tf
import horovod.tensorflow.keras as hvd

[...]

hvd.init()

# Instantiate a dataset object
dataset = tf.data.Dataset.from_tensor_slices(files)

[...]

# Get a disjoint data subset for the worker
dataset = dataset.shard(hvd.size(), hvd.rank())

[...]

# Randomize
dataset = dataset.shuffle(buffer_size)

# Create worker's mini-batch and prepare next ones
dataset = dataset.batch(batch_size)
dataset = dataset.prefetch(tf.data.AUTOTUNE)

[...]
```

DISTRIBUTED TRAINING WITH HOROVOD

- Create a Slurm job script for the code wrapped with Horovod
  - K Horovod workers correspond to K tasks in total, 1 MPI process each
  - K = nodes · tasks-per-node = nodes · gpus-per-node

```
#!/bin/bash

#SBATCH --account=training2306
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=20
#SBATCH --time=00:20:00
#SBATCH --gres=gpu:4
#SBATCH --partition=dc-gpu

srun python train_model.py
```



DISTRIBUTED TRAINING WITH HOROVOD

Basics to parallelize your model

- Use Horovod to wrap existing model code
- Use data containers and pipelines to provide data to workers efficiently
- Create a Slurm job script to submit the wrapped code

DATA PARALLEL DISTRIBUTED TRAINING

Summary

- Opportunity to efficiently speed up training on large data
- Requires K GPUs, the larger K, the better
- Training with a larger effective batch size $|\mathfrak{B}| = K|B_{\text{ref}}|$
- Data pipelines, high bandwidth network (InfiniBand) pave the way
- Horovod, PyTorch DDP, TensorFlow DistributedStrategies
- Additional measures to stabilize training – upcoming lectures