HIGH PERFORMANCE MACHINE LEARNING

Caspar van Leeuwen
High Performance ML consultant
SURF

Parallel computing for Deep Learning

Content

- benefits of parallellization
- parallellization strategies
- Hands-on: hyperparameter grid search on an HPC system
- parallel stochastic gradient descent (SGD)
- synchronous and asynchronous parallel SGD
- communication backends
- frameworks for distributed deep learning
- documentation of distributed DL frameworks

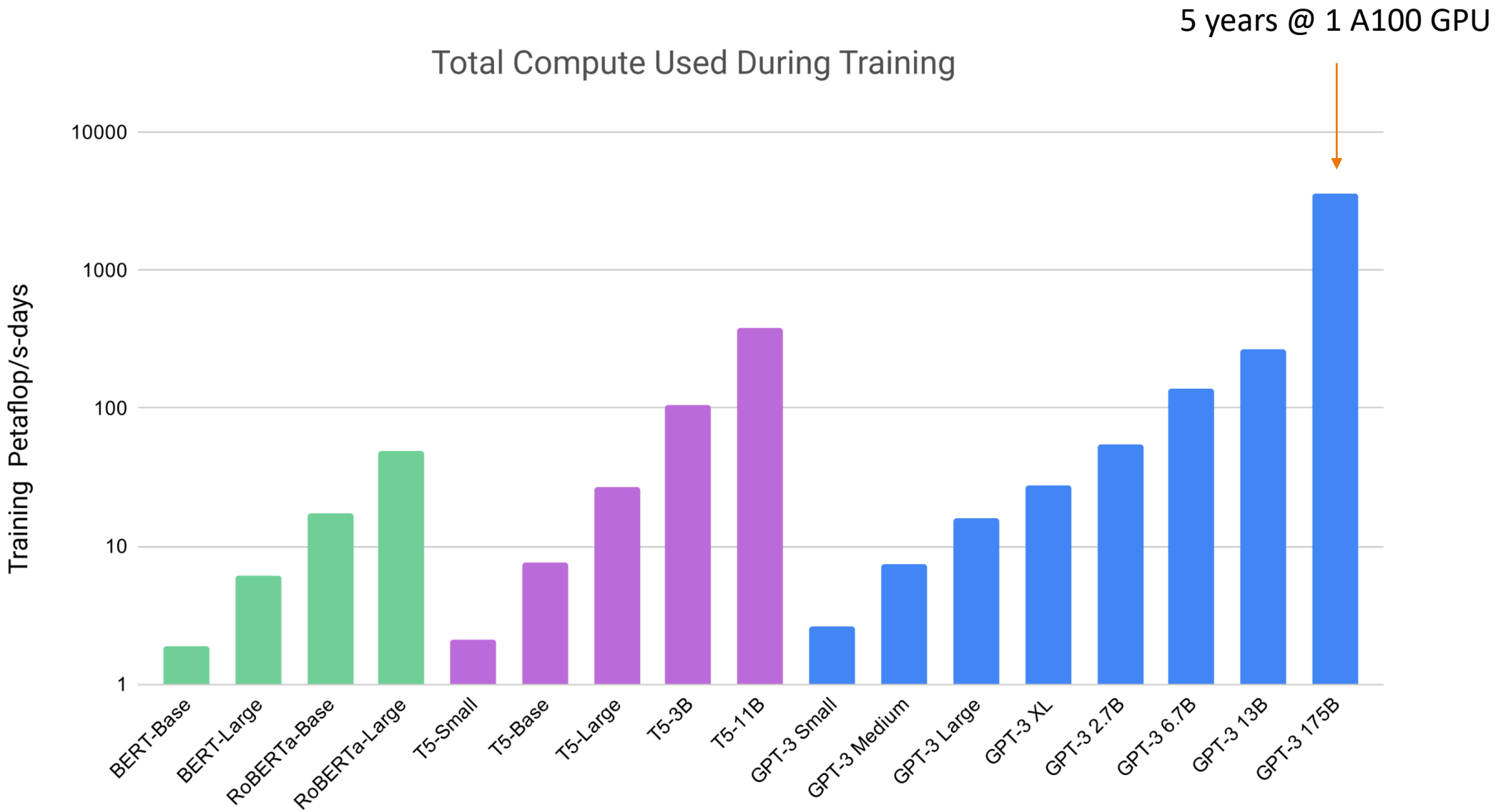
Parallel computing for Deep Learning

Goal: understand the documentation of distributed DL frameworks.

From TensorFlow docs on “distribution strategy”:

- “tf.distribute.Strategy intends to cover a number of use cases along different axes… Synchronous vs asynchronous training: These are two common ways of distributing training with data parallelism. In sync training, all workers train over different slices of input data in sync, and aggregating gradients at each step. In async training, all workers are independently training over the input data and updating variables asynchronously. Typically sync training is supported via all-reduce and async through parameter server architecture.”
- “MultiWorkerMirroredStrategy currently allows you to choose between two different implementations of collective ops. CollectiveCommunication.RING implements ring-based collectives using gRPC as the communication layer. CollectiveCommunication.NCCL uses Nvidia's NCCL to implement collectives.”

Parallelization: why?



Parallelization: why?

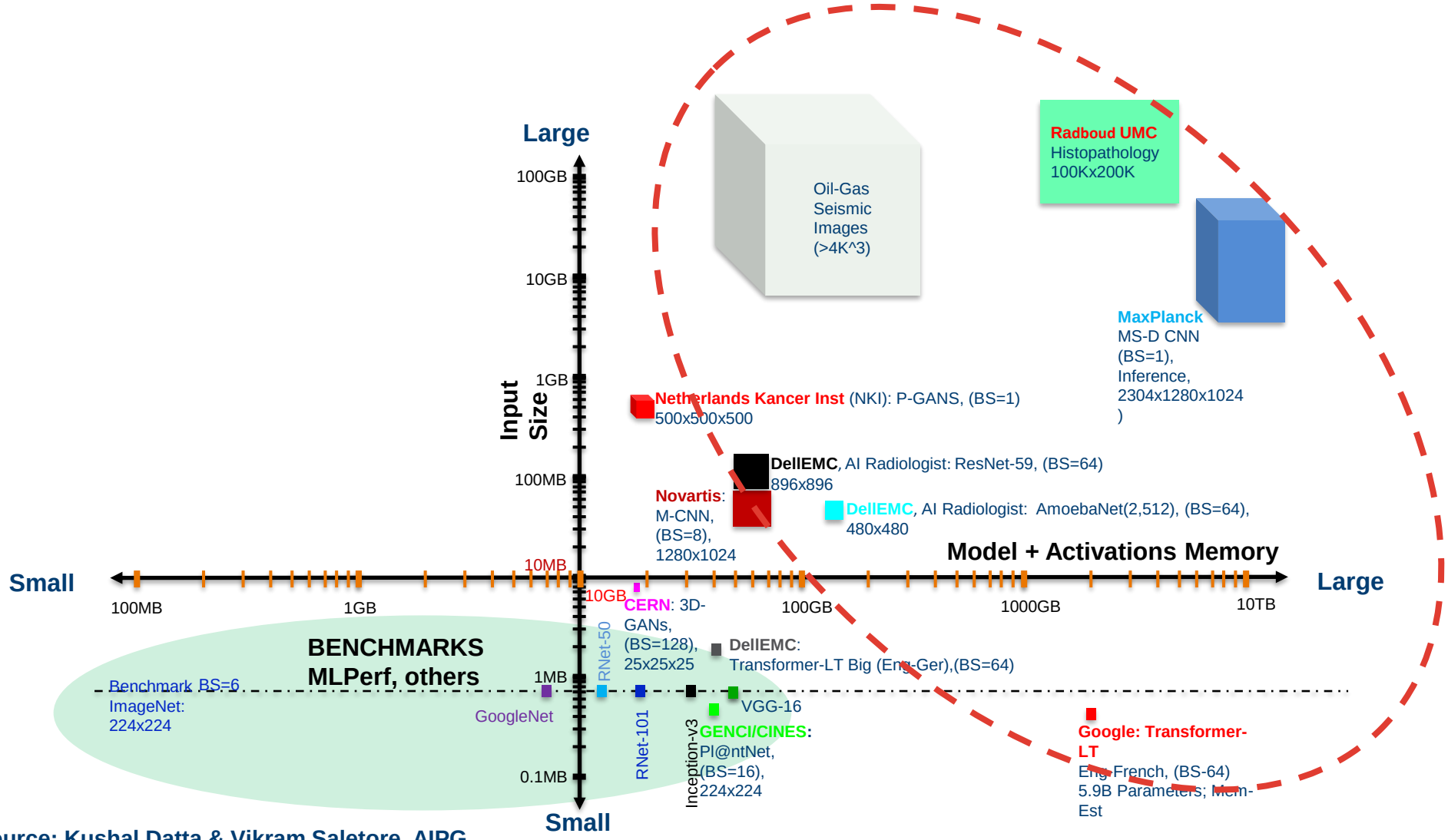
Faster trainings …

- Enables learning on larger datasets
- Enables improved accuracy through better hyperparemeter tuning
- Enables larger, more complex models
- …

Bigger models (high memory requirement) …

- Enables larger, more complex models

Parallelization: when?



Source: Kushal Datta & Vikram Saletore, AIPG, Intel

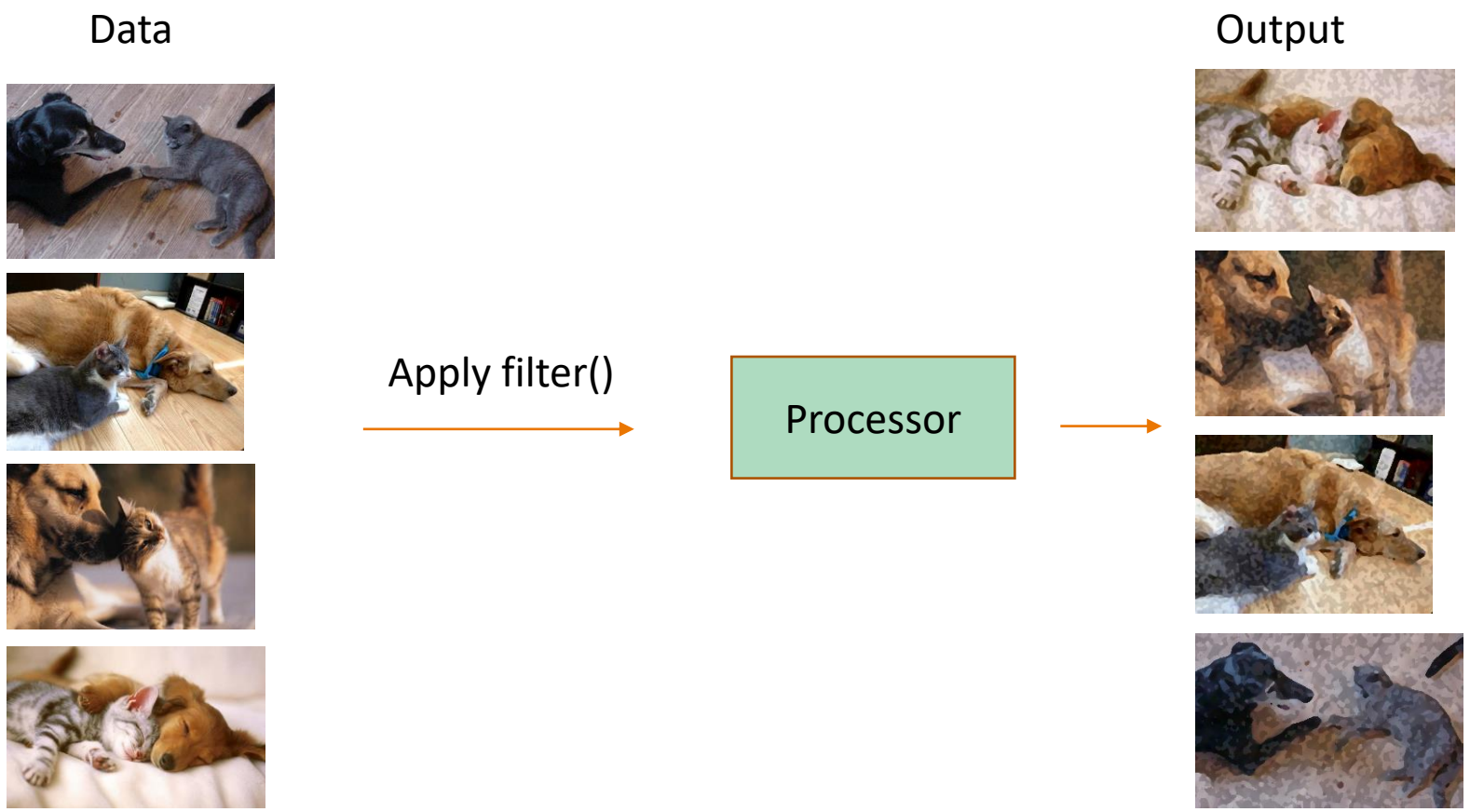
Parallelization: the basics

What is parallel computing?

- Multiple processors or computers working on a single computational problem

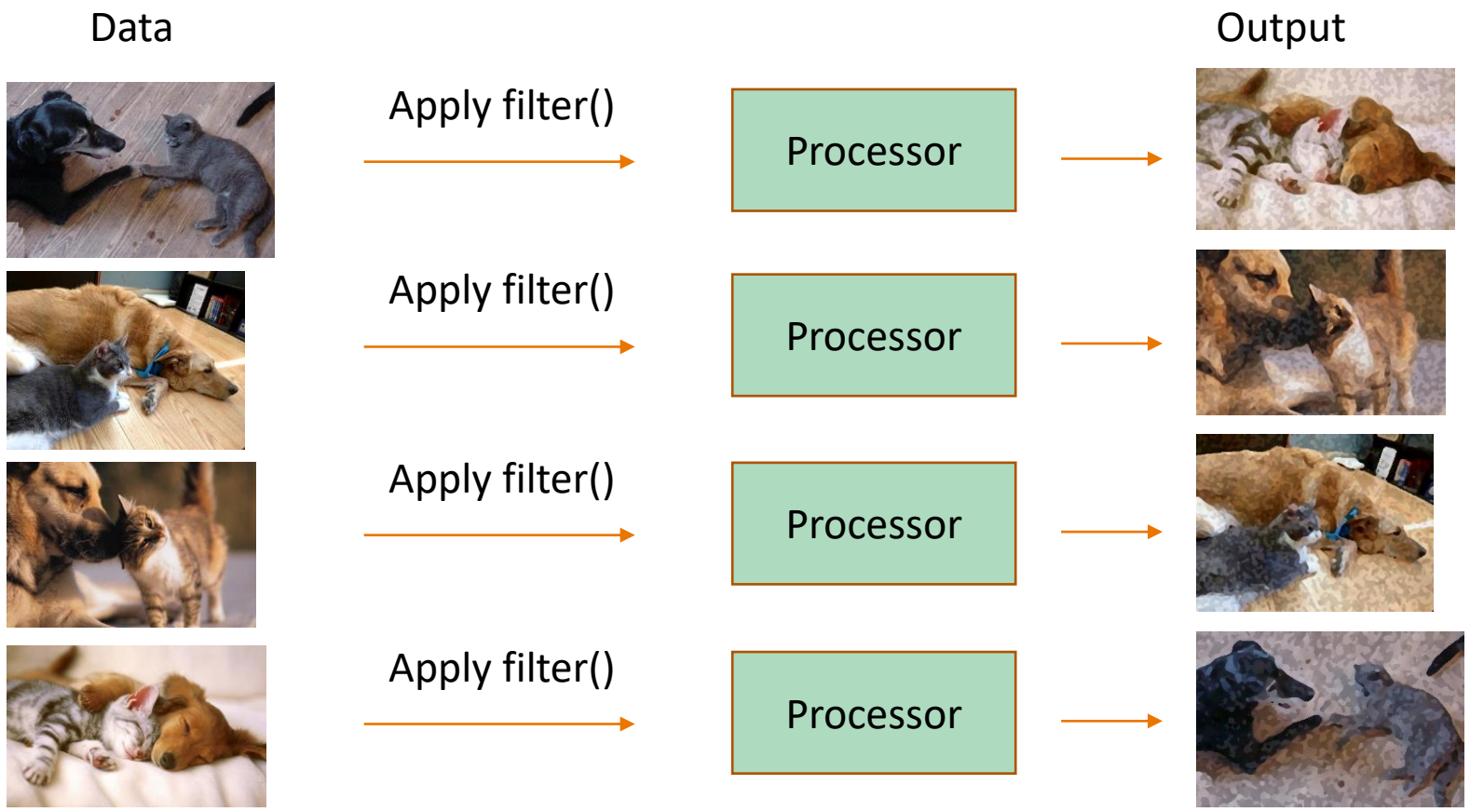
Parallelization: the basics

Serial computing



Parallelization: the basics

Parallel computing



Parallelization: the basics

Benefits:

- Solve computationally intensive problems (speedup)
- Solve problems that don’t fit a single memory (multiple computers)

Requirements:

- Problem should be divisible in smaller tasks

Types of parallelization

What types of parallelization exist?

- Instruction level parallelism
- Embarrassingly parallel
- Data parallel
- Tensor parallel
- Model parallelism
- Hybrid data/Tensor parallelism
- Pipeline parallelism

Increasing complexity

https://huggingface.co/transformers/v4.9.2/parallelism.html

Types of parallelization

- Instruction level parallelism
  - Executing multiple instructions (e.g. additions) in parallel
  - Examples: vector instructions (CPU), Tensor Cores (GPU)
  - https://en.wikipedia.org/wiki/Instruction-level_parallelism
  - More this afternoon!

X =	1.1	3.7	-1.6	2.3				
		+			=	-2.3	5.4	-1.8	7.5
Y =	-3.4	1.7	-0.2	5.2				

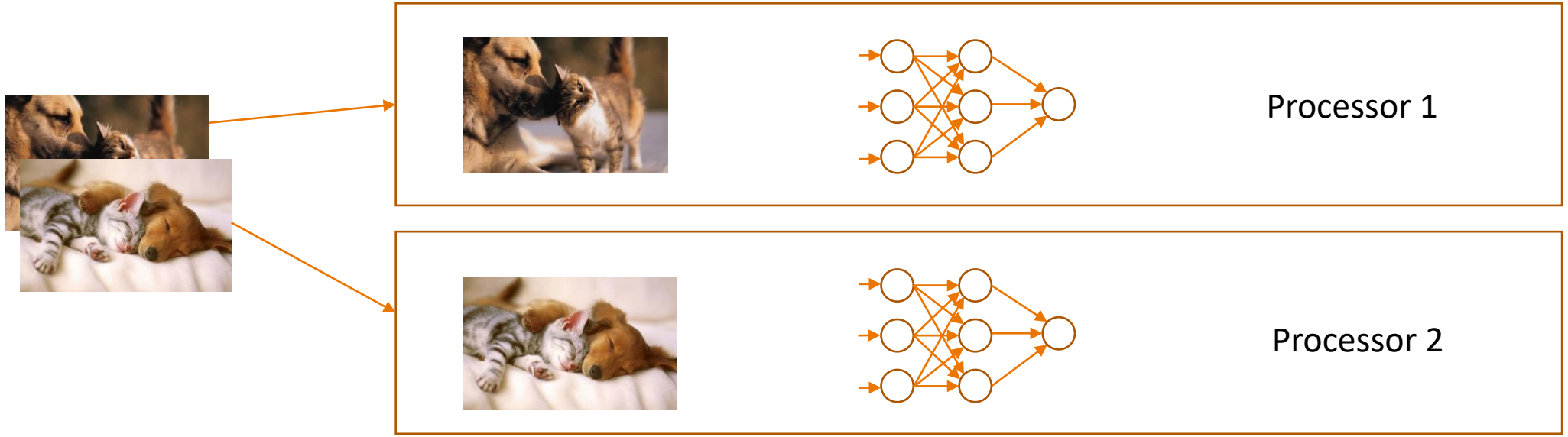
Types of parallelization

- Embarrassingly parallel
  - A workload or problem where little or no effort is needed to separate into number of parallel tasks
  - Examples: hyperparameter grid search, training multiple model architectures
  - https://en.wikipedia.org/wiki/Embarrassingly_parallel

Data Parallelism

Train a single model, single set of hyperparameters, but ***faster***

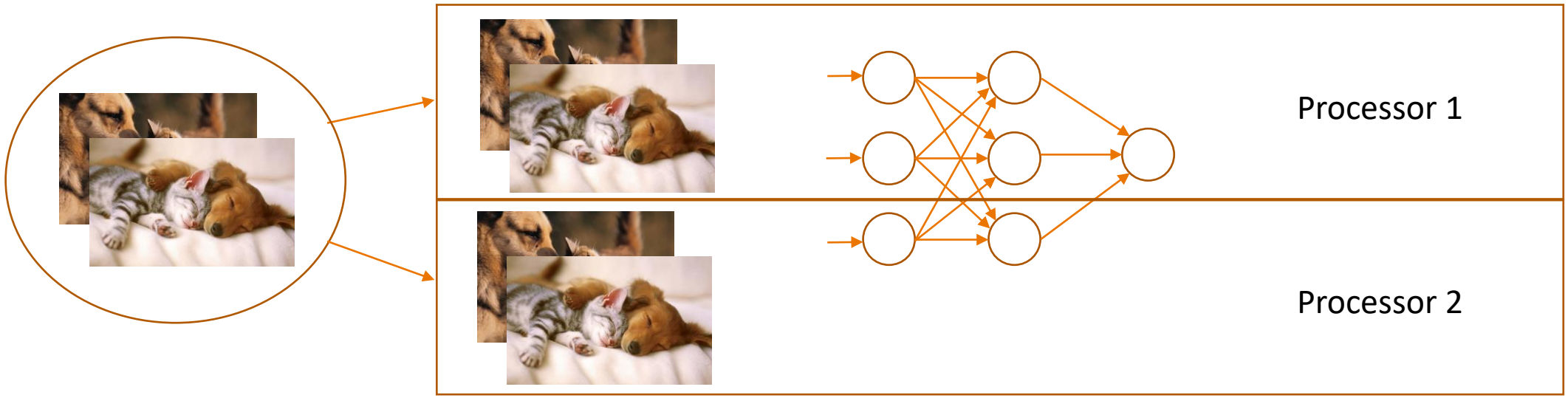
- Split the batch over multiple processors (CPUs/GPUs)
- Each processor holds a copy of the model
- Forward pass: calculated by each of the workers
- Backward pass: gradients computed (per worker), communicated and aggregated



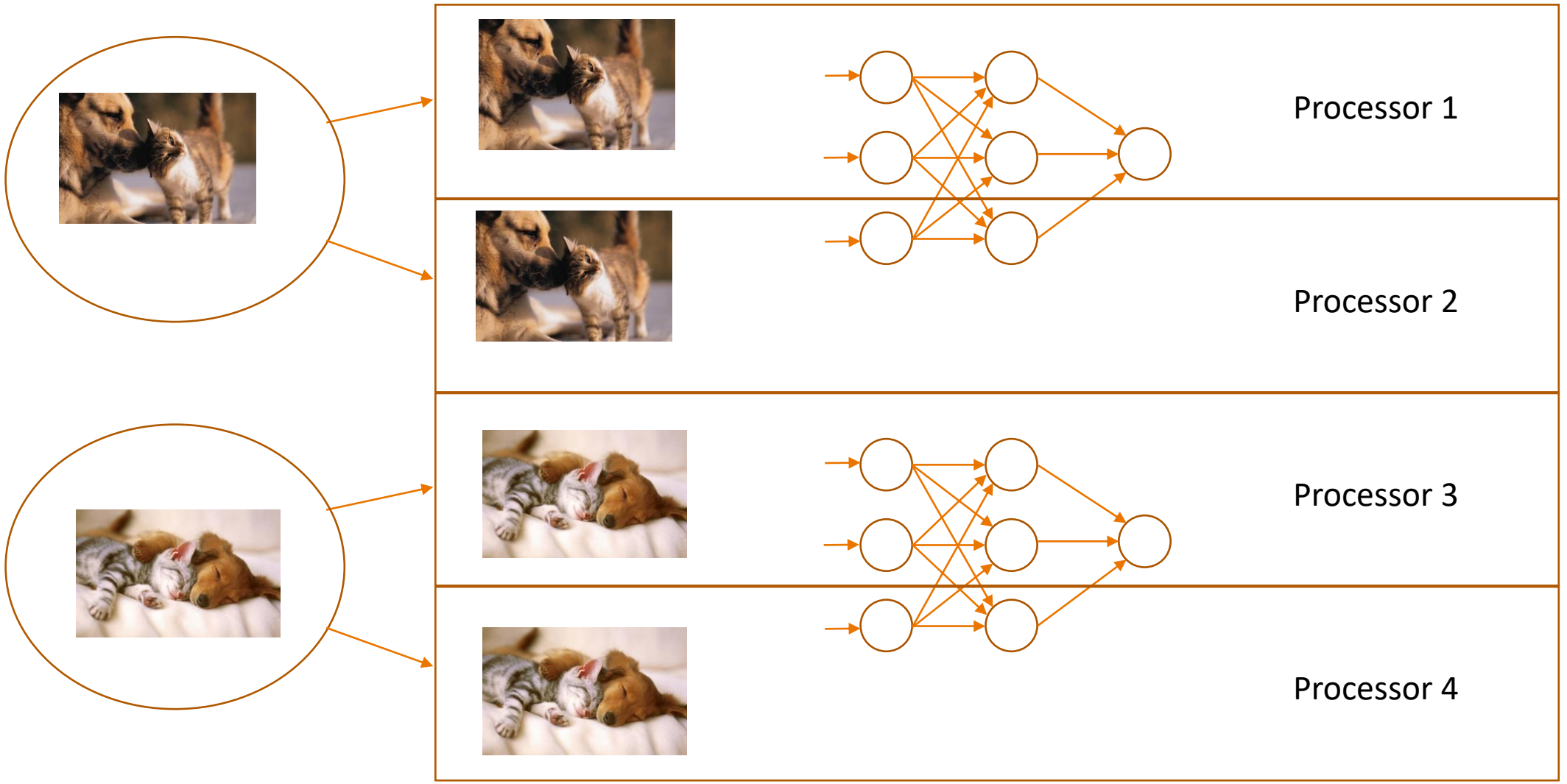
Tensor parallelism

Train a single ***very big*** model, single set of hyperparameters

- Split (a single layer, i.e. tensor of) the model over multiple processors (CPUs/GPUs)
- Each processor sees all the data
- Communication needed both during forward and backward pass!



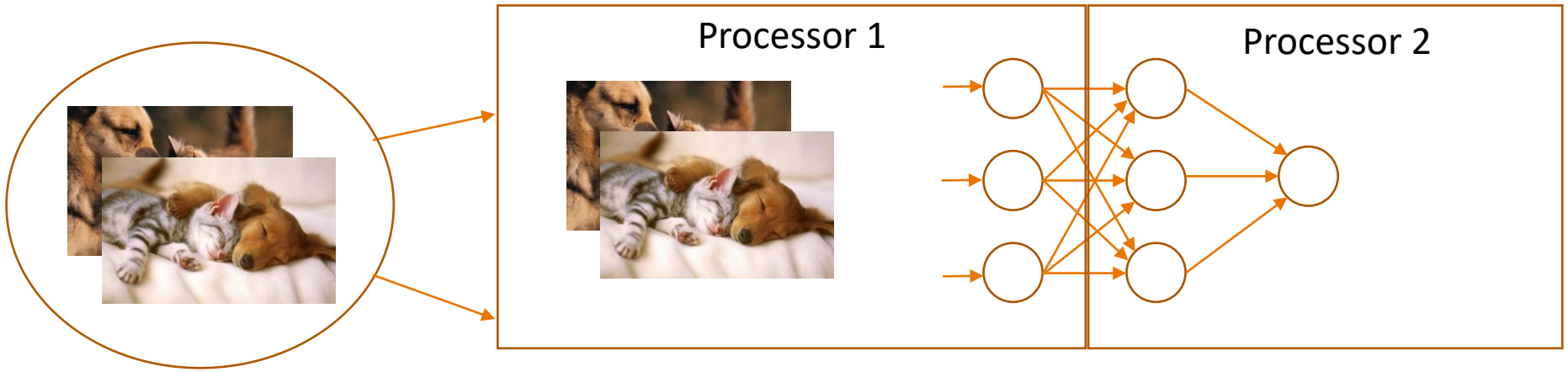
Hybrid Tensor/data parallelism



Model parallelism

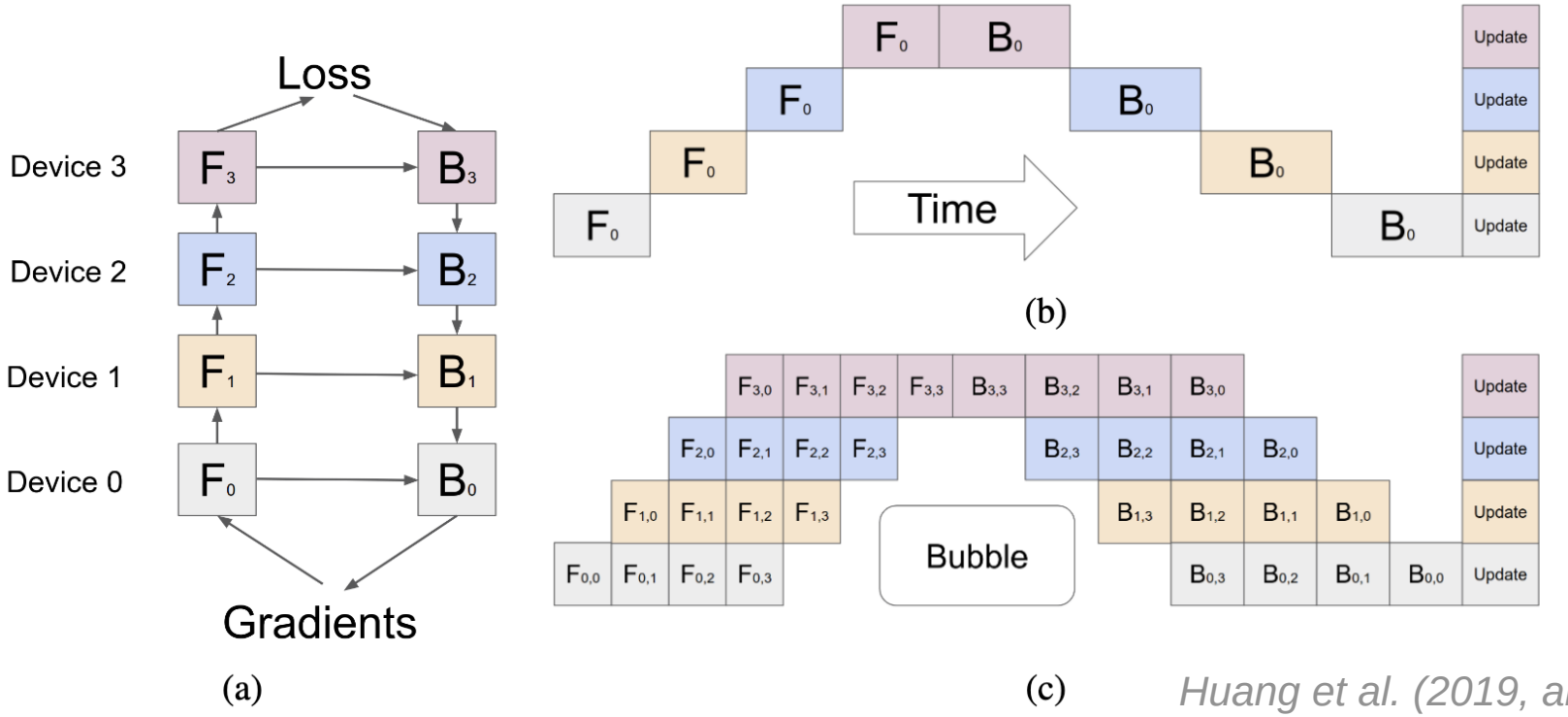
Train a single ***very big*** model, single set of hyperparameters

- Split the model (by layer) over multiple processors (CPUs/GPUs)
- Communication needed both during forward and backward pass!



Pipeline (model) parallelism

- Model parallelism, executed in a pipelined fashion over multiple micro-batches.
- More efficient than model parallelism: hides communication with computation
- E.g. Gpipe (for PyTorch / TensorFlow), PipeDream (PyTorch)
- See https://pytorch.org/docs/stable/pipeline.html



Huang et al. (2019, arXiv:1811.06965)

What type of parallelism is applicable?

- Multiple (independent) training runs => Embarrassingly parallel
- Single model takes too long too train => Data parallel
- Single model is too big for memory => Tensor / Model / Pipeline parallelism
- Single model is too big for memory *and* takes too long to train => Hybrid parallelism
  - All of the well-known, big models (GPT-X) are trained this way

What type of parallelism is applicable?

- Multiple (independent) training runs => Embarrassingly parallel → Any cluster will do, no fast network needed
- Single model takes too long too train => Data parallel
- Single model is too big for memory => Tensor / Model / Pipeline parallelism
- Single model is too big for memory *and* takes too long to train => Hybrid parallelism
  - All of the well-known, big models (GPT-X) are trained this way

What type of parallelism is applicable?

- Multiple (independent) training runs => Embarrassingly parallel
- Single model takes too long too train => Data parallel
- Single model is too big for memory => Tensor / Model / Pipeline parallelism
- Single model is too big for memory *and* takes too long to train => Hybrid parallelism
  - All of the well-known, big models (GPT-X) are trained this way

HPC cluster needed, i.e. with fast network and fast connections between e.g. GPUs in a single node

What type of parallelism is applicable?

- Multiple (independent) training runs => Embarrassingly parallel
- Single model takes too long too train => Data parallel
- Single model is too big for memory => Tensor / Model / Pipeline parallelism
- Single model is too big for memory *and* takes too long to train => Hybrid parallelism
  - All of the well-known, big models (GPT-X) are trained this way

Note: use data parallel whenever you can. Use model / Pipeline parallel if you *really* need to. Even then, consider alternatives:

- Model pruning
- Use different hardware architecture (e.g. data parallel @ CPU)
- Reduced precision datatypes (discussed later today)

Hands-on: hyperparameter grid search on an HPC system

In this hands-on, we will do a grid search on batch size & learning rate.

We will use a feature of the SLURM scheduler to submit an array job. This job runs the same job script multiple times, but with one essential difference: the `SLURM_ARRAY_TASK_ID` environment variable is different for each element of the array job. We use this as an index to our array in `array_config.txt` to make each task do something different.

Exercise

Submit the `array.batch` job. While it is running, inspect the `array_config.txt` and `array.batch` to see if you can understand what is going on. Once finished, inspect the output. How many output files do you have?

Data parallel stochastic gradient descent

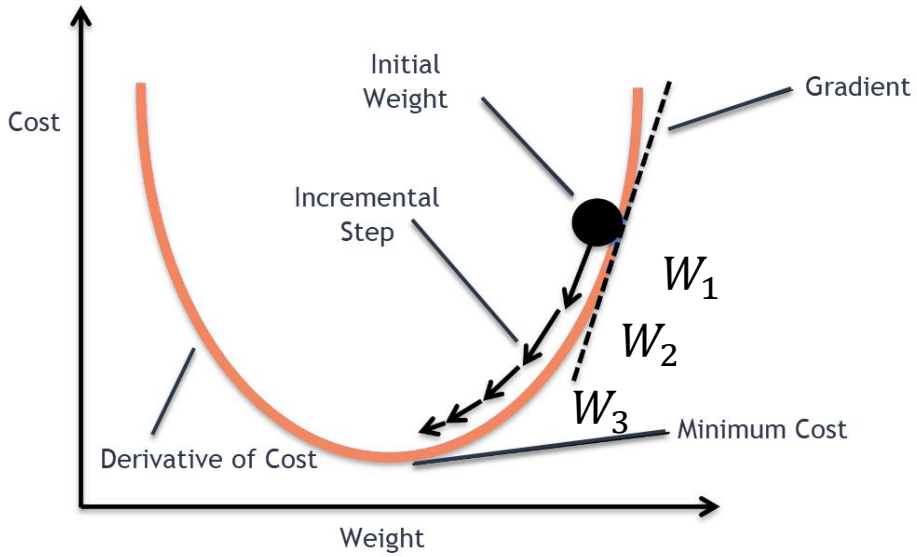
- Most networks are trained using stochastic gradient descent (SGD)
- Distributed stochastic gradient can be done in two ways
  - Synchronous SGD
  - Asynchronous SGD

Stochastic gradient descent (SGD)

SGD: find optimum by following the slope

$w = w - \eta \nabla \mathrm{Q}(\mathrm{w})$

w = weights, η = learning rate, $\nabla \mathrm{Q}(\mathrm{w})$ = gradient for current batch.



Data parallel synchronous SGD

- Each device (j) computes the gradients ($\nabla Q_j(w)$) based on its own batch!
- Needs to be aggregated before updating weights

Device 1	Device 2	Device 3	Device 4
$g_1 = \nabla Q_1(w)$	$g_2 = \nabla Q_2(w)$	$g_1 = \nabla Q_3(w)$	$g_1 = \nabla Q_4(w)$

$$\nabla Q(w) = \sum_j \nabla Q_j(w)$$

$$w = w - \eta \nabla Q(w)$$

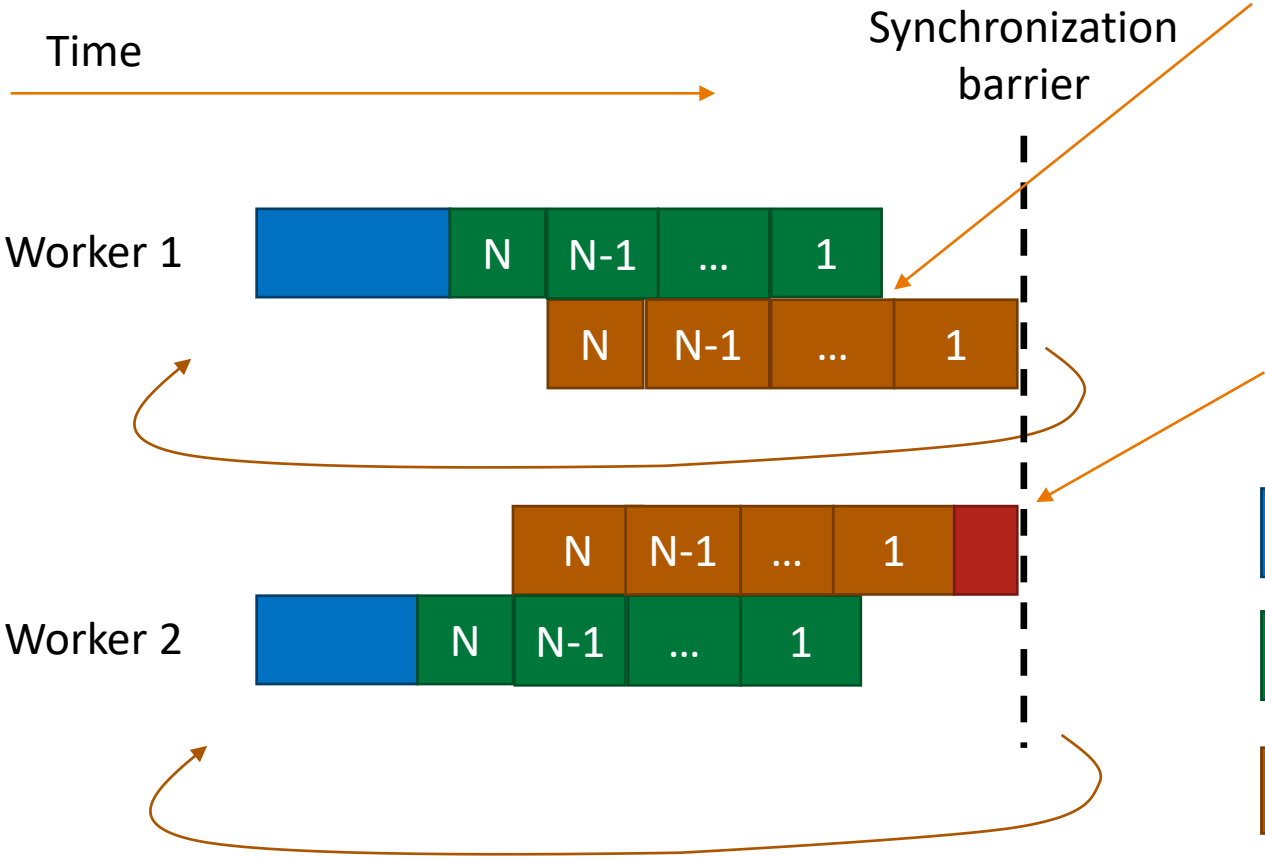
Data parallel synchronous SGD

Effect on batch size:

- For *N* workers that each see *n* examples: batch size effectively n × N.
- Larger batch => generally needs to be compensated by higher learning rate.
- No exact science!
  - Some use $\eta_{distributed} = \eta_{serial} \cdot N$
  - Some use $\eta_{distributed} = \eta_{serial} \cdot \sqrt{N}$
  - Experiment!

Data parallel synchronous SGD

A different view…



Pro tip:
- Overlap communication and computation (don't waste compute cycles waiting for communication!)
- *Most* (distributed) DL frameworks already take care of this for you ☺

Pro tip 2:
- Penalty for *synchronous* SGD: you have to wait for the slowest worker, before next iteration.
- Make sure all workers are equally fast!

Forward pass

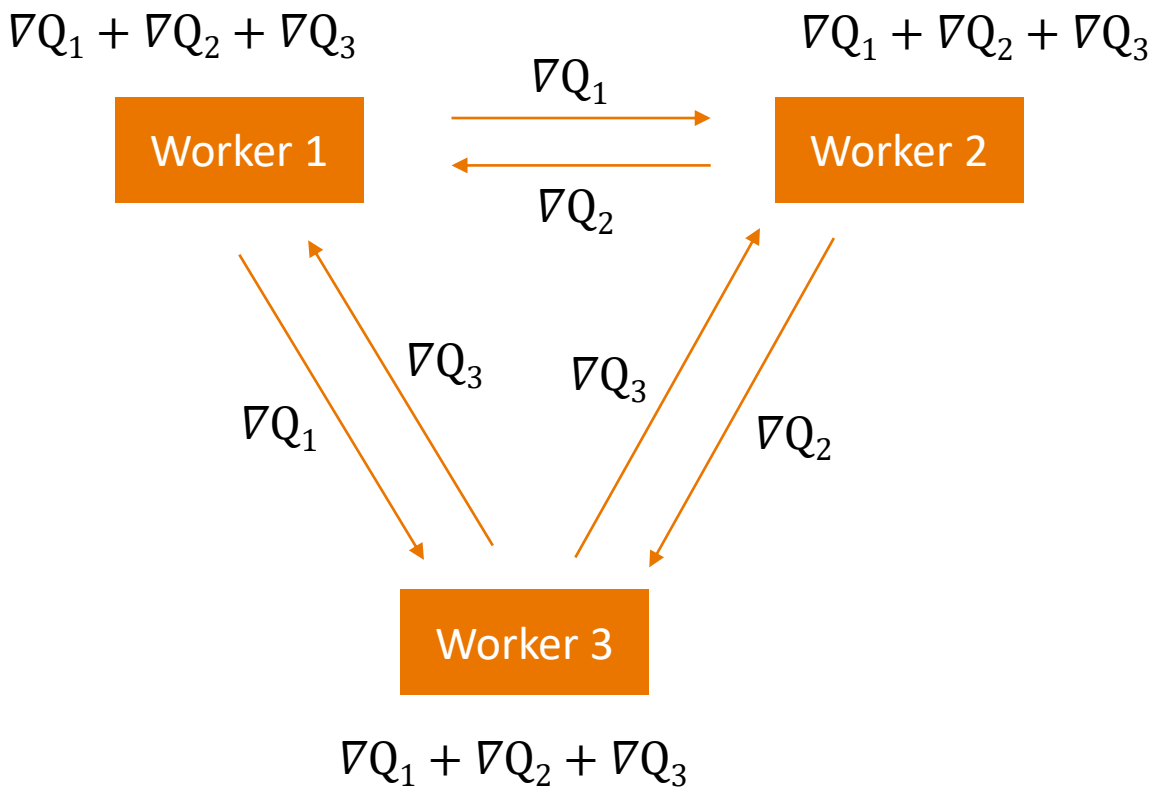
N — Backward pass, compute gradient of layer N

N — Backward pass, communicate gradient N

Wait

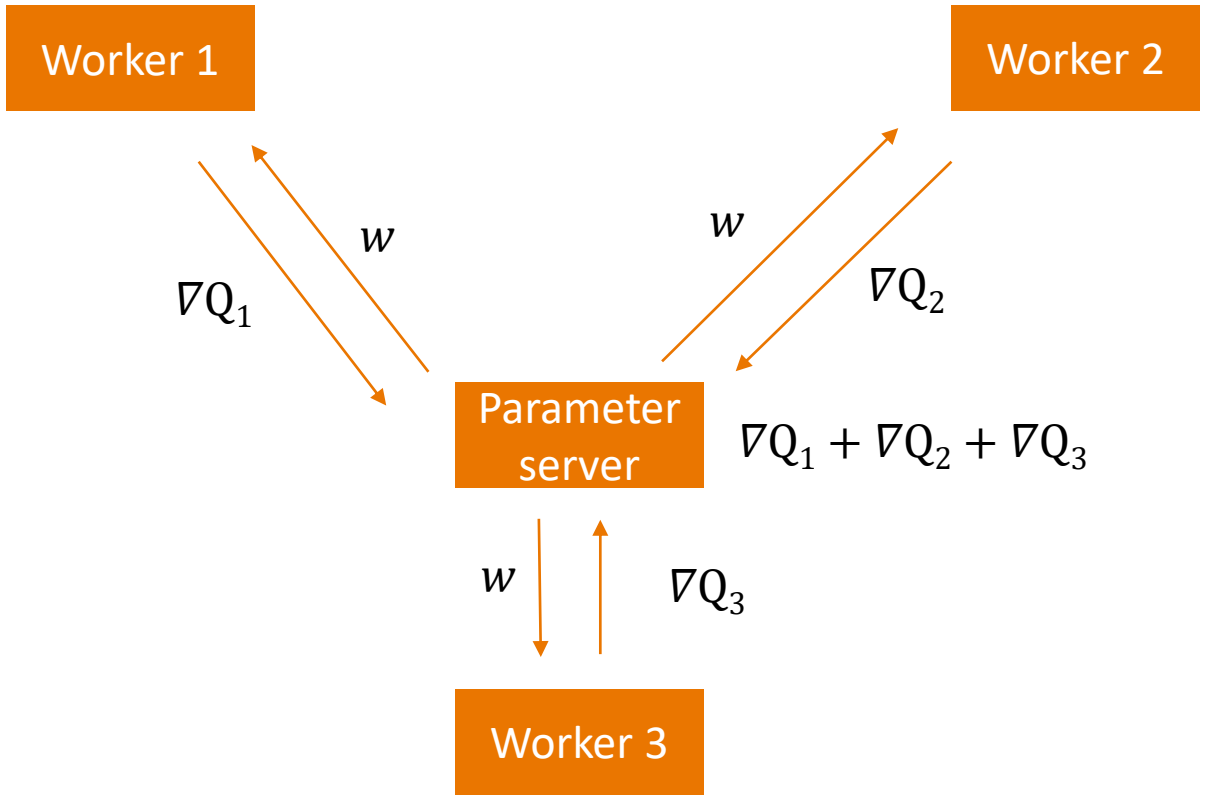
Decentralized data parallel synchronous SGD

Gradients are communicated and aggregated by *all* workers



Centralized data parallel synchronous SGD

There is also an alternative, where a parameter server is used to aggregate the gradients, and distribute the updated model:



Centralized vs decentralized

- Centralized approach does not scale well: parameter servers create a communication bottleneck
- More info, see e.g. https://arxiv.org/pdf/1705.09056.pdf

Communicating gradients…

Ok, so the most widely accepted approach is

distributed…

data parallel…

synchronous…

SGD…

… but how do distributed deep learning frameworks aggregate their gradients in such a setup?

A bit of history

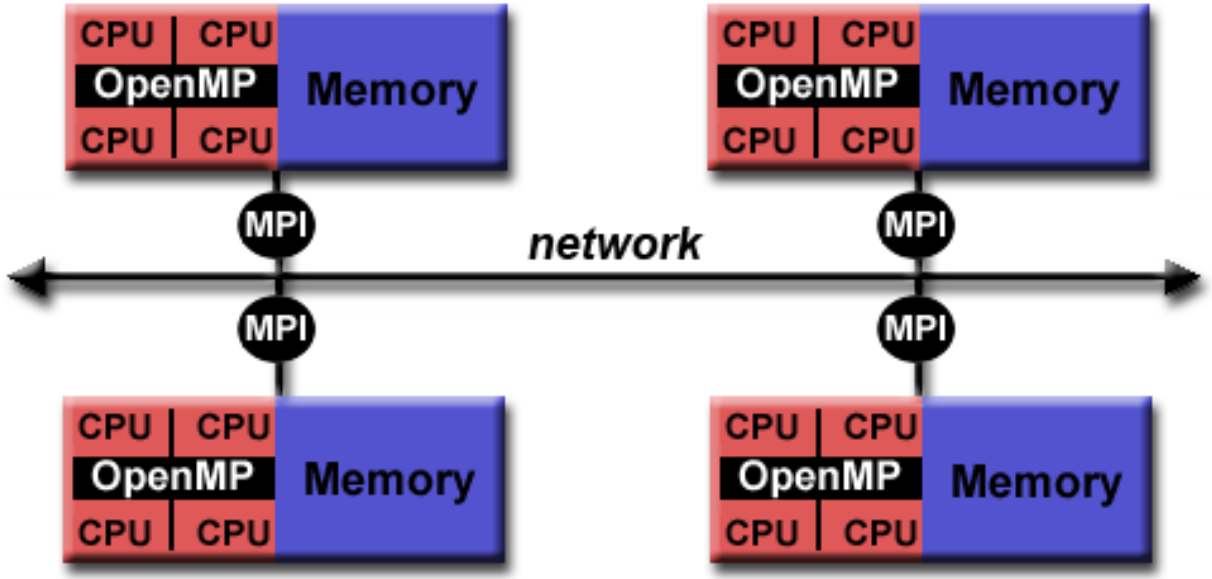
‘Traditionally’ a lot of machine learning was *not* done in an HPC context. As a result:

- Most frameworks had little focus on distributed learning
- Most frameworks that offered distributed learning were based on parameter servers
- Most AI experts probably never heard of MPI...

The Message Passing Interface (MPI)

MPI is a standard for *parallelization* on a *distributed memory system*

- Distributed memory system: processors can't access each other's memory
- Explicit communication (over a network) is required between one memory and another to work on the same task
- MPI is the 'language' of this communication
- MPI is the *de facto* standard for traditional HPC applications



The Message Passing Interface (MPI)

MPI has routines to send data between individual workers…

The Message Passing Interface (MPI)

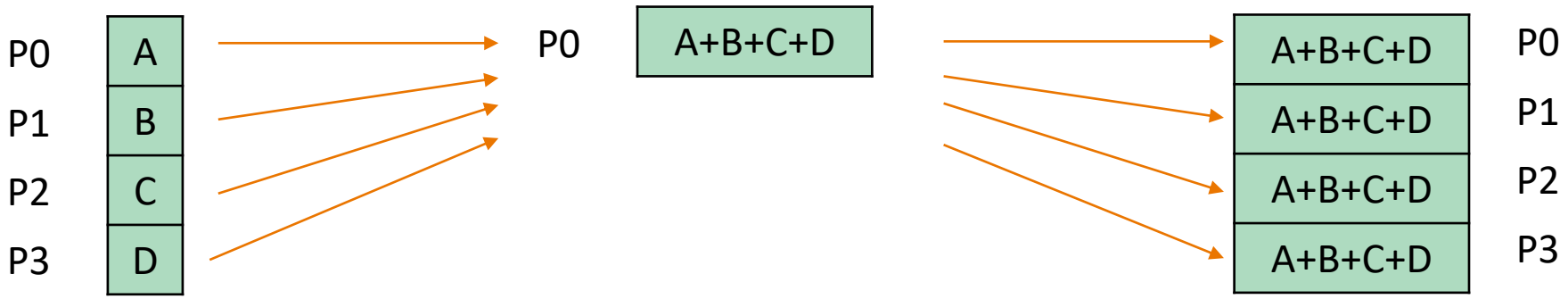
But also to broadcast data to other workers…

The Message Passing Interface (MPI)

And most importantly (for deep learning): apply *collective* operations, such as ‘allreduce’. This operation is widely used in distributed deep learning to aggregate gradients!

The Message Passing Interface (MPI)

- MPI is a standard: it defines what AllReduce should *do*, not *how it should be done*.
- MPI libraries implement MPI functions. These libraries decide *how it should be done*.
- Example: an inefficient allreduce operation could implemented like this:



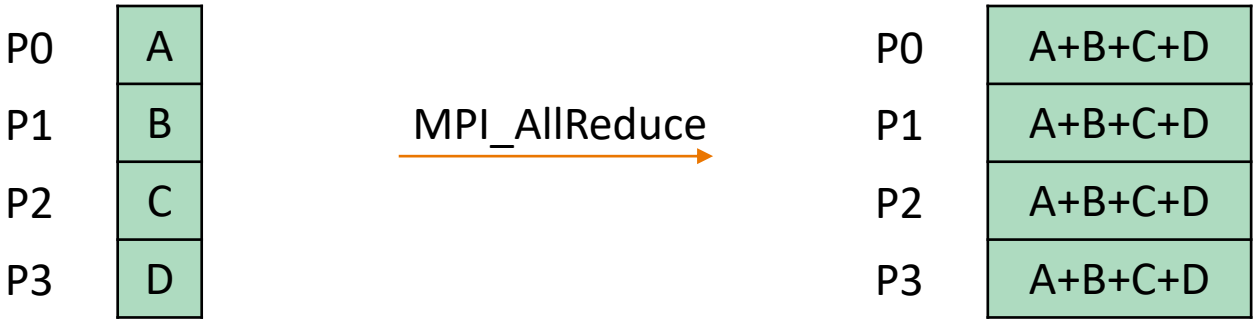
Relevance of MPI for distributed deep learning

Why is this relevant?

- Distributed DL frameworks often support multiple communication backends for their collective allreduce operations
- These backends often either implement (part of) the MPI API or something similar
- It is important to pick a communication backend with an efficient implementation.
- The most efficient implementation may vary per hardware.
- Example: Nvidia's NCCL library implements a subset of MPI collective operations. These implementations are highly optimized for Nvidia GPUs.

MPI / NCCL

- MPI often used to communicate gradients
- MPI_AllReduce aggregates and sums gradients (remember: $\nabla Q(w) = \sum_j \nabla Q_j(w)$)
- NVIDIA's NCCL library contains an implementation of MPI routines optimized for GPU ⇔ GPU communication



Frameworks for distributed learning

- TensorFlow's tf.distribute: quite tricky to program. Lot's of code changes needed from serial to distributed (https://www.tensorflow.org/guide/distributed_training)
- TensorFlow + Horovod: serial => distributed with minimal code changes (https://horovod.readthedocs.io/en/stable/tensorflow.html)
- PyTorch's torch.distributed (https://pytorch.org/tutorials/intermediate/dist_tuto.html)
- PyTorch + Horovod: serial => distributed with minimal code changes (https://horovod.readthedocs.io/en/stable/pytorch.html)
- PyTorch Lightning: hides a lot of boiler plate code (also nice for serial training). Very little changes needed between serial & parallel execution, especially on a SLURM cluster (https://pytorch-lightning.readthedocs.io/en/latest/clouds/cluster.html#slurm-managed-cluster)

Hands-on: data parallel with torch.distributed and PyTorch Lightning

- Submit the `ddp.batch` and `ddp_lightning.batch` jobs
- Inspect the code that they run: `mnist_classify_ddp.py` and `mnist_classify_ddp_lightning.py`. Can you see the advantage that PyTorch Lightning offers? Can you think of any disadvantages?

Recap of goal: understand docs of DL frameworks

From TensorFlow docs on “distribution strategy”:

- “tf.distribute.Strategy intends to cover a number of use cases along different axes… Synchronous vs asynchronous training: These are two common ways of distributing training with data parallelism. In sync training, all workers train over different slices of input data in sync, and aggregating gradients at each step. In async training, all workers are independently training over the input data and updating variables asynchronously. Typically sync training is supported via all-reduce and async through parameter server architecture.”
- “MultiWorkerMirroredStrategy currently allows you to choose between two different implementations of collective ops. CollectiveCommunication.RING implements ring-based collectives using gRPC as the communication layer. CollectiveCommunication.NCCL uses Nvidia's NCCL to implement collectives.”

Practical tips & take home messages

- If increased throughput is the goal, use data parallelism
- If a large model is the goal, use model (or hybrid or pipeline) parallelism, but consider the consequences (slower training) and alternatives (model pruning, CPU-based training, etc)
- Account for the difference in convergence behavior of data parallel SGD, e.g. by adjusting & experimenting with the learning rate.
- *Synchronous* parallel SGD is the most common approach for distributed learning, because it is well understood. *Asynchronous* parallel SGD can scale very well, but convergence behavior is less clear.
- Use an efficient backend for collective communications (e.g. NCCL)

Further reading

- Distributed TensorFlow using Horovod: https://towardsdatascience.com/distributed-tensorflow-using-horovod-6d572f8790c4
- Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis: https://arxiv.org/pdf/1802.09941.pdf
- Prace best practice guide for Deep Learning: http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-Deep-Learning.pdf
- Technologies behind Distributed Deep Learning: https://preferredresearch.jp/2018/07/10/technologies-behind-distributed-deep-learning-allreduce/
- PyTorch Distributed: https://pytorch.org/tutorials/beginner/dist_overview.html and https://pytorch.org/docs/stable/distributed.html