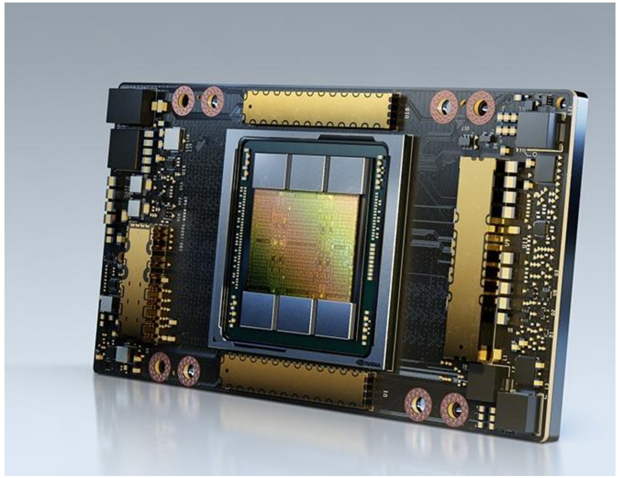
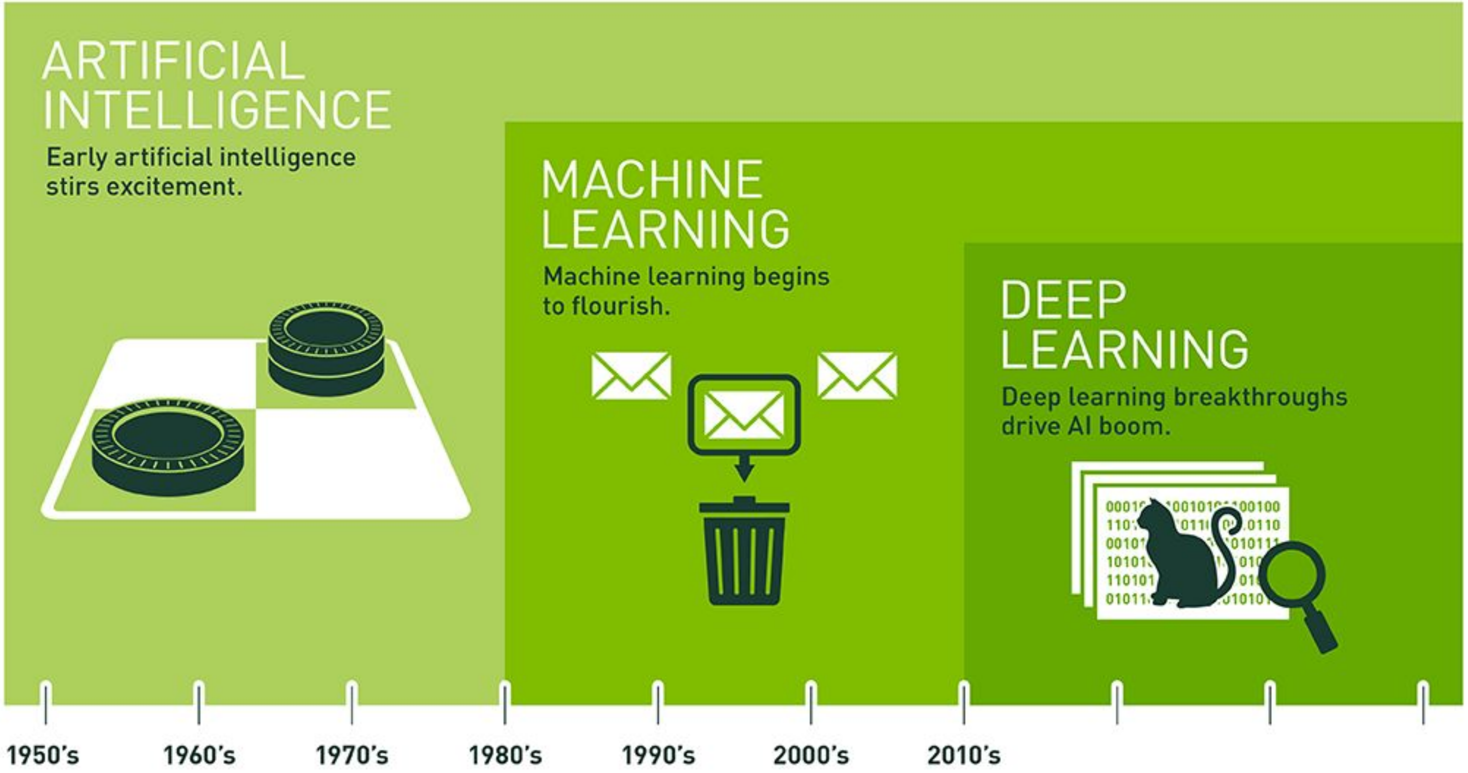
The Abdus Salam International Centre for Theoretical Physics



HPC for DL

Serafina Di Gioia
PostDoctoral Researcher @ ICTP

From LISP to the DL revolution…

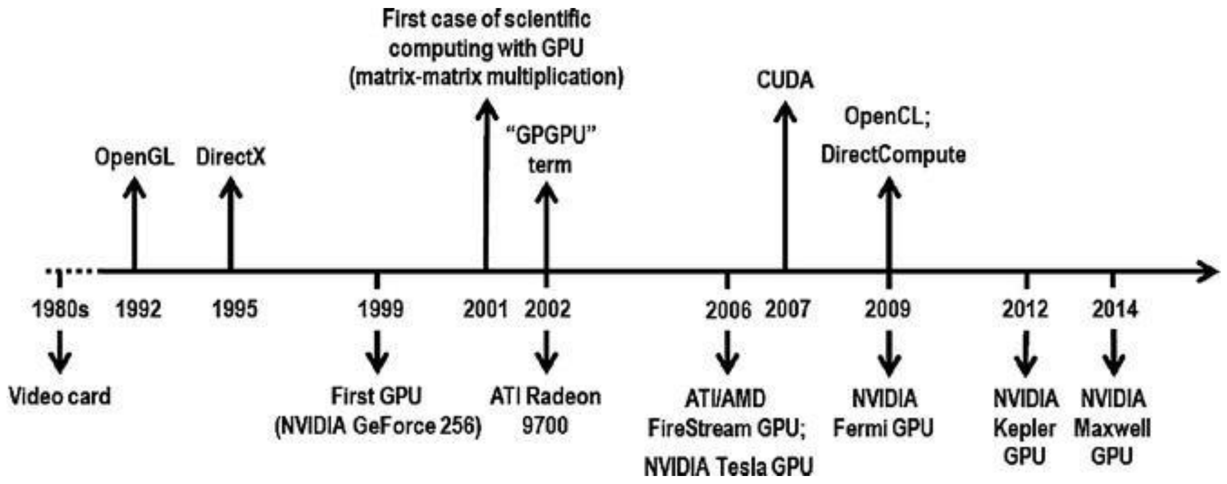

ARTIFICIAL INTELLIGENCE
Early artificial intelligence stirs excitement.
MACHINE LEARNING
Machine learning begins to flourish.
DEEP LEARNING
Deep learning breakthroughs drive AI boom.
1950's
1960's
1970's
1980's
1990's
2000's
2010's

Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

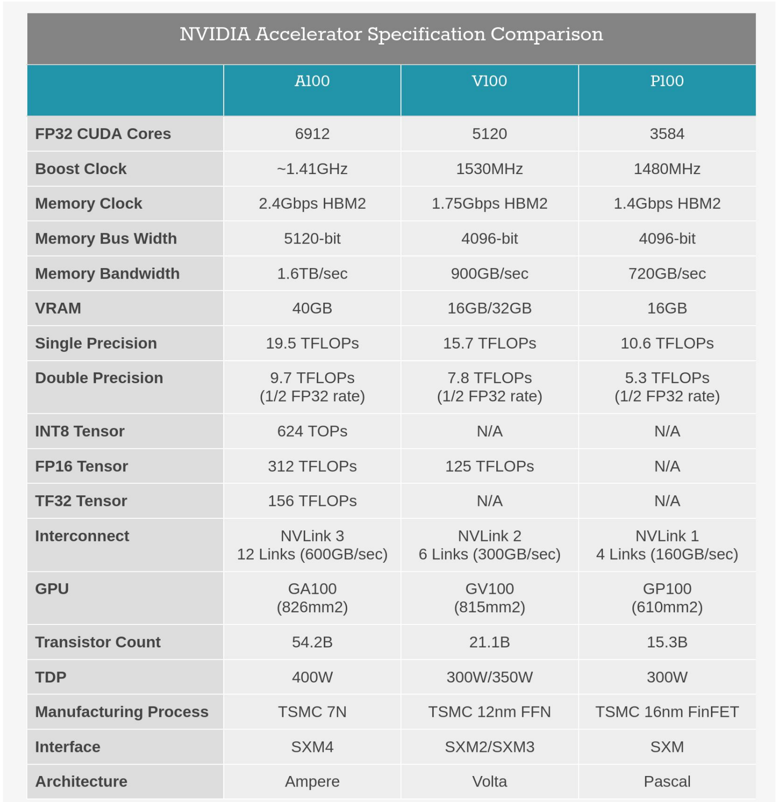
Main ingredients for DL breakthrough

- large datasets available (e.g IMAGENET)
- GPUs development (in particular, CUDA introduction)
- increased involvement of developers from CV and scientific communities

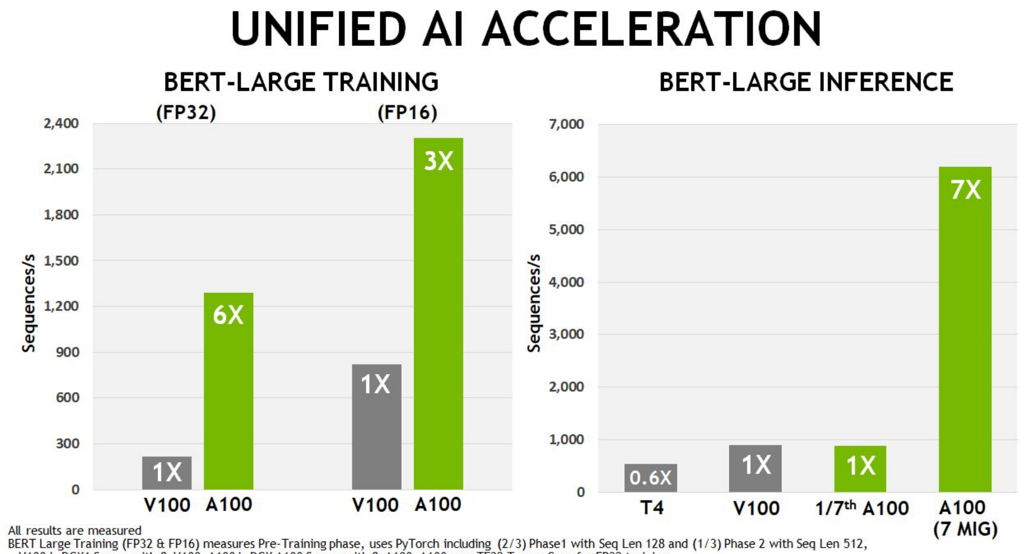
The DL era starts few years after that CUDA came to light



Incredible performance increase of NVIDIA



NVIDIA Accelerator Specification Comparison			
	A100	V100	P100
FP32 CUDA Cores	6912	5120	3584
Boost Clock	~1.41GHz	1530MHz	1480MHz
Memory Clock	2.4Gbps HBM2	1.75Gbps HBM2	1.4Gbps HBM2
Memory Bus Width	5120-bit	4096-bit	4096-bit
Memory Bandwidth	1.6TB/sec	900GB/sec	720GB/sec
VRAM	40GB	16GB/32GB	16GB
Single Precision	19.5 TFLOPs	15.7 TFLOPs	10.6 TFLOPs
Double Precision	9.7 TFLOPs (1/2 FP32 rate)	7.8 TFLOPs (1/2 FP32 rate)	5.3 TFLOPs (1/2 FP32 rate)
INT8 Tensor	624 TOPs	N/A	N/A
FP16 Tensor	312 TFLOPs	125 TFLOPs	N/A
TF32 Tensor	156 TFLOPs	N/A	N/A
Interconnect	NVLink 3 12 Links (600GB/sec)	NVLink 2 6 Links (300GB/sec)	NVLink 1 4 Links (160GB/sec)
GPU	GA100 (826mm2)	GV100 (815mm2)	GP100 (610mm2)
Transistor Count	54.2B	21.1B	15.3B
TDP	400W	300W/350W	300W
Manufacturing Process	TSMC 7N	TSMC 12nm FFN	TSMC 16nm FinFET
Interface	SXM4	SXM2/SXM3	SXM
Architecture	Ampere	Volta	Pascal



All results are measured
BERT Large Training (FP32 & FP16) measures Pre-Training phase, uses PyTorch including (2/3) Phase1 with Seq Len 128 and (1/3) Phase 2 with Seq Len 512, V100 is DGX1 Server with 8xV100, A100 is DGX A100 Server with 8xA100, A100 uses TF32 Tensor Core for FP32 training
BERT Large Inference uses TRT 7.1 for T4/V100, with INT8/FP16 at batch size 256. Pre-production TRT for A100, uses batch size 94 and INT8 with sparsity

Why GPUs for DL?

1) **Neural networks are embarrassingly parallel algorithm**
2) **most of the operations performed in DL models can be rewritten as matrix multiplications**
3) **big datasets require to perform big matrix computation (**extremely slow on CPU with respect to GPU**)**
4) **well established libraries, with specific classes for ML objects (e.g. cuDNN, more recently tensorRT)**

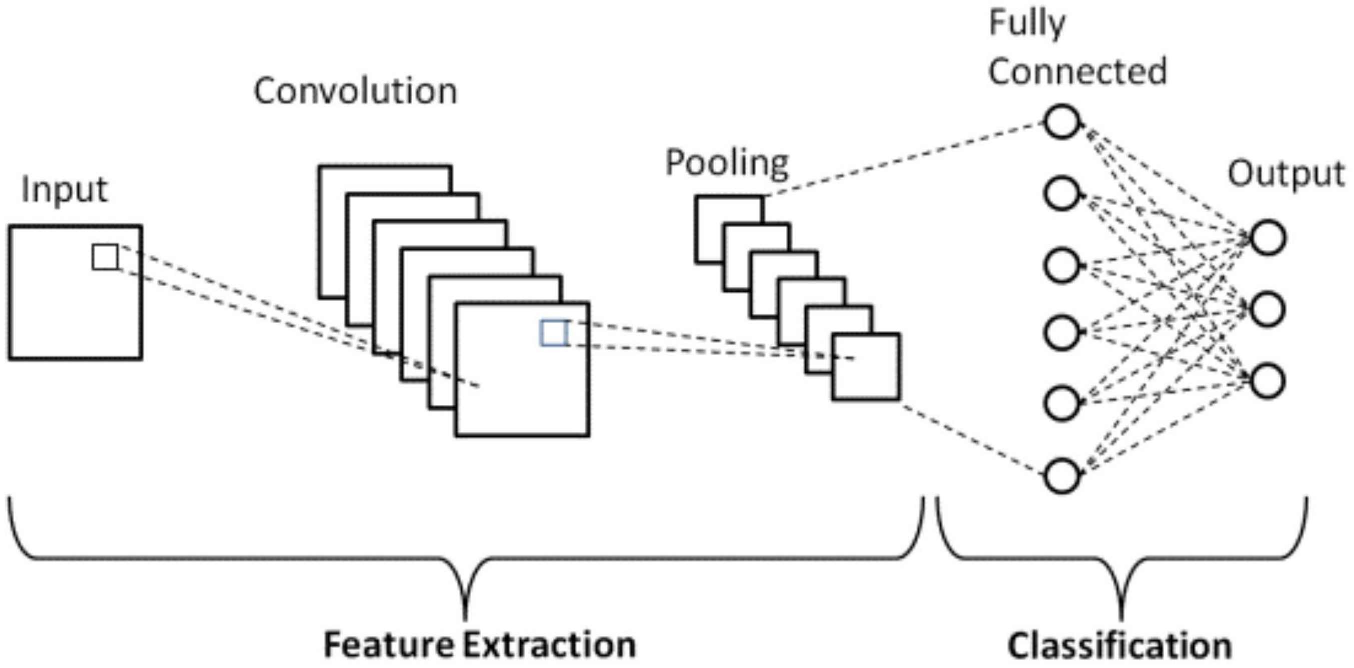
and we know that GPUs are very good in solving specific parallel tasks (e.g matrix multiplication) , thanks to

- +1000 cores (>100K threads)
- **SIMD / SIMT**
- **high memory bandwidth**
- **newer GPUs have also tensor cores (particularly suited to tensor ops typical of NNs), and mixed precision**

  However, also GPUs have limitations:

  - GPUs might not be as efficient for extreme sparse networks, due to the overhead of managing sparse data structures.
  - Some specialized sparse operations might not be as optimized as dense operations on GPUs.

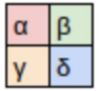
DL for image classification: Convolutional Neural Networks (CNN)



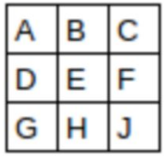
Everything in DL turns to be a matrix multiplication at the end…

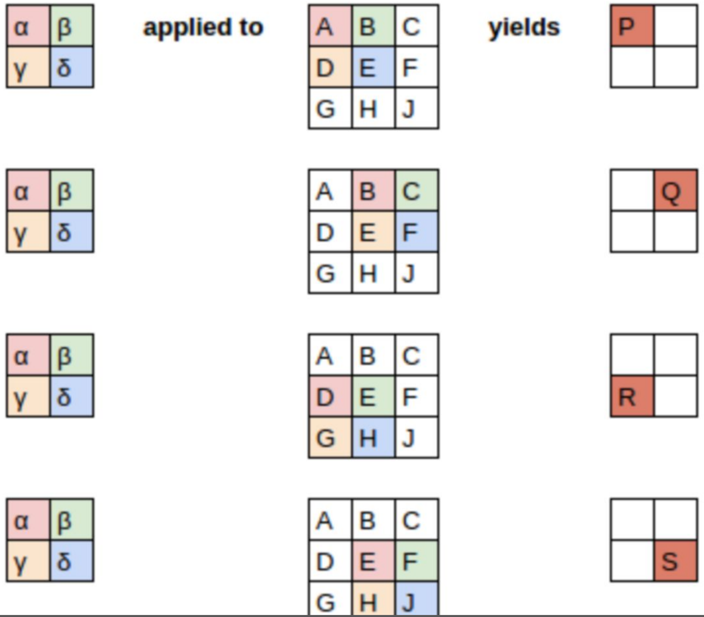
Let's take a look at basic element of CNN: convolution layer

Consider the case where we are applying (2,2) kernel



to a (3,3) matrix:

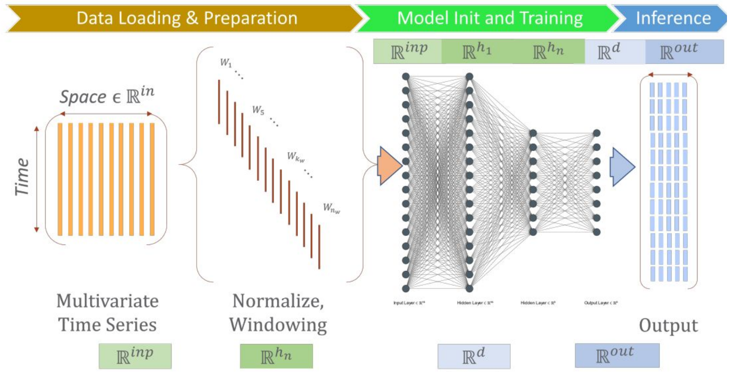


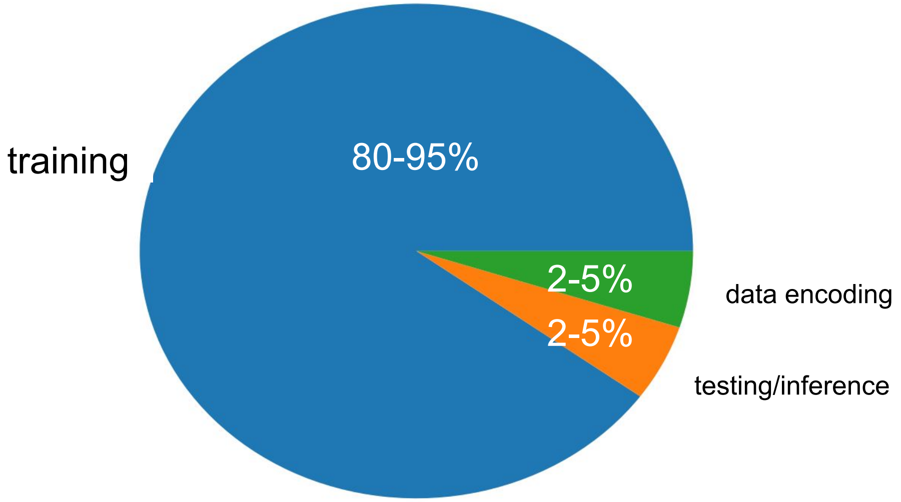


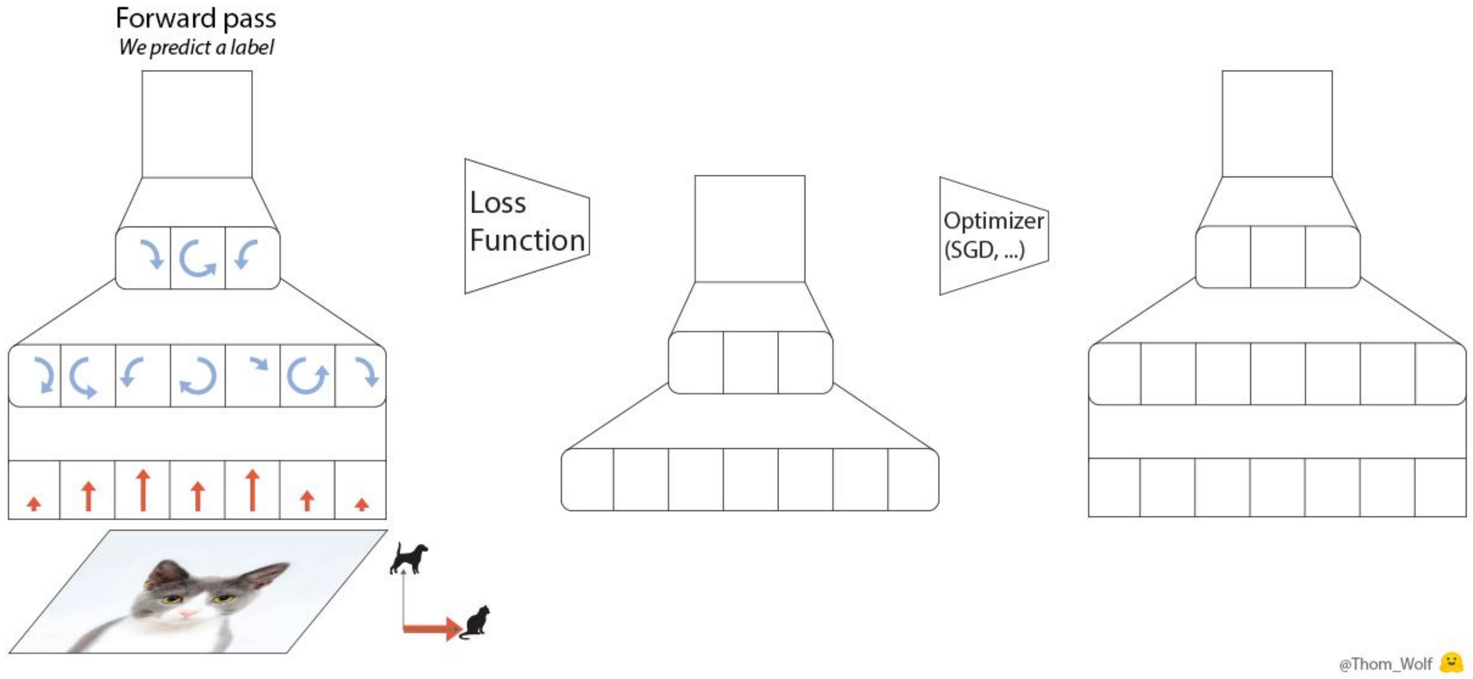
The convolution can be rewritten as

$$\begin{array}{ccccccccc} \alpha & \beta & 0 & \gamma & \delta & 0 & 0 & 0 & 0 \\ 0 & \alpha & \beta & 0 & \gamma & \delta & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha & \beta & 0 & \gamma & \delta & 0 \\ 0 & 0 & 0 & 0 & \alpha & \beta & 0 & \gamma & \delta \\ \hline A & B & C & D & E & F & G & H & J \end{array} * \begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ J \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \\ b \end{bmatrix} = \begin{bmatrix} \alpha A+\beta B+0C+\gamma D+\delta E+0F+0G+0H+0J+b \\ 0A+\alpha B+\beta C+0D+\gamma E+\delta F+0G+0H+0J+b \\ 0A+0B+0C+\alpha D+\beta E+0F+\gamma G+\delta H+0J+b \\ 0A+0B+0C+0D+\alpha E+\beta F+0G+\gamma H+\delta J+b \end{bmatrix} = \begin{bmatrix} \alpha A+\beta B+\gamma D+\delta E+b \\ \alpha B+\beta C+\gamma E+\delta F+b \\ \alpha D+\beta E+\gamma G+\delta H+b \\ \alpha E+\beta F+\gamma H+\delta J+b \end{bmatrix} = \begin{bmatrix} P \\ Q \\ R \\ S \end{bmatrix}$$

Analyzing the computational workload of DL models






Forward pass
We predict a label
Loss Function
Optimizer (SGD, ...)
@Thom_Wolf

Different strategies for Multi-GPUs training/evaluation

We can identify 5 different categories of parallelism

`model parallelism`

`tensor parallelism`

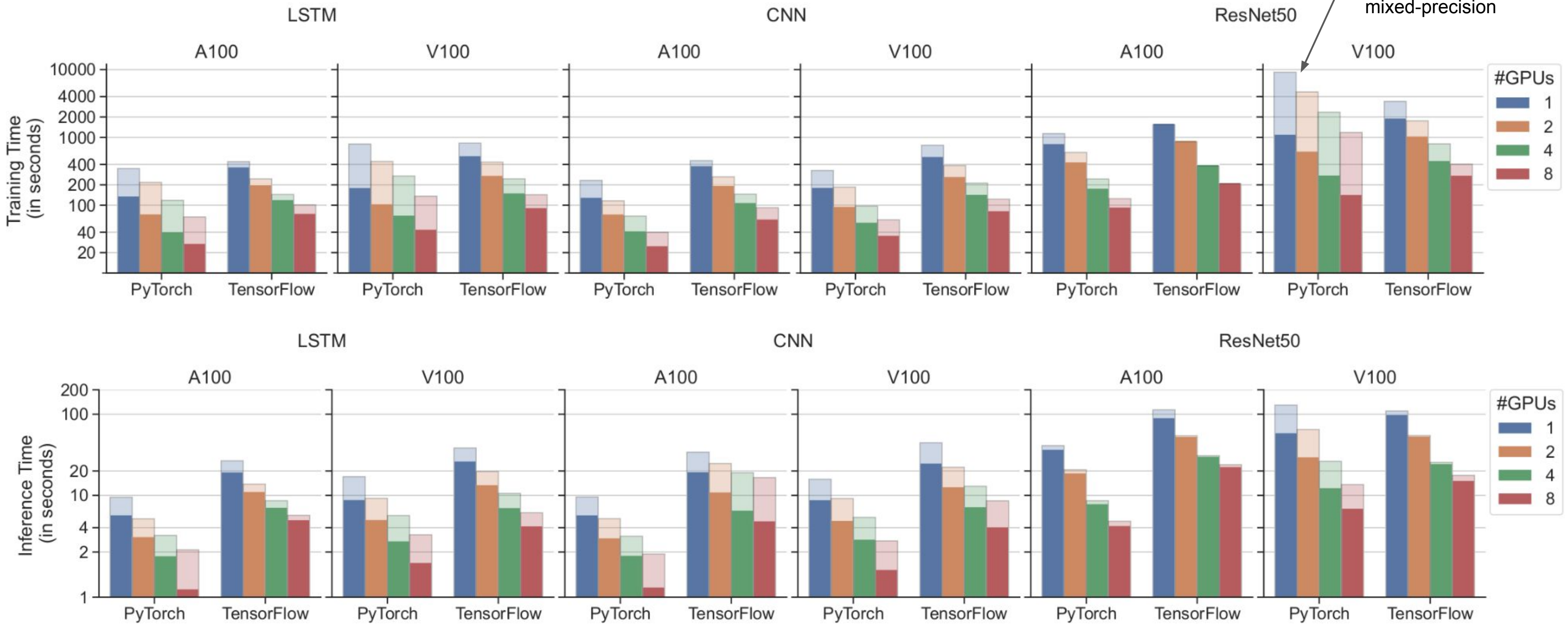
`data parallelism`

`sequence parallelism`

`pipeline parallelism`

Multi-GPUs benchmarks of DL models

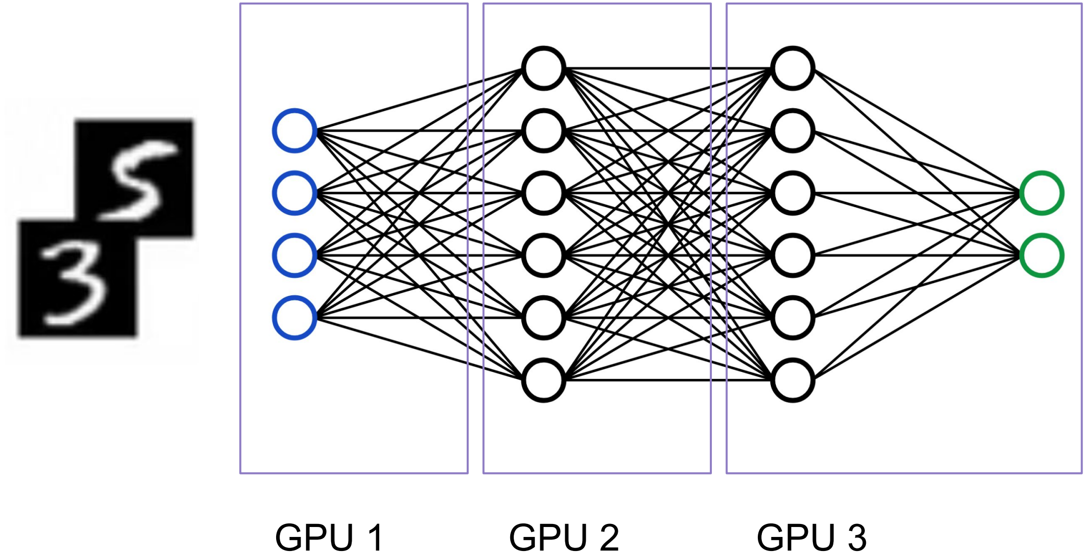
Shaded regions indicate the extra time without mixed-precision



Model parallelism

In this parallelism framework we choose to put different layers of the NN on different GPUs

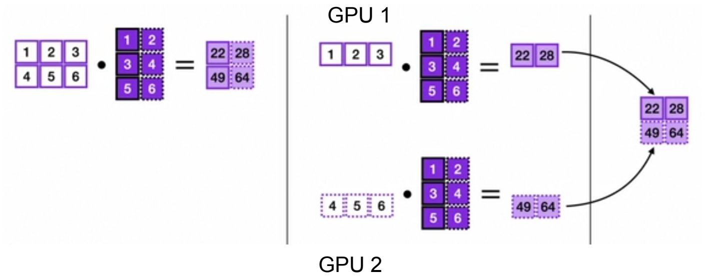
to work around GPU memory limits

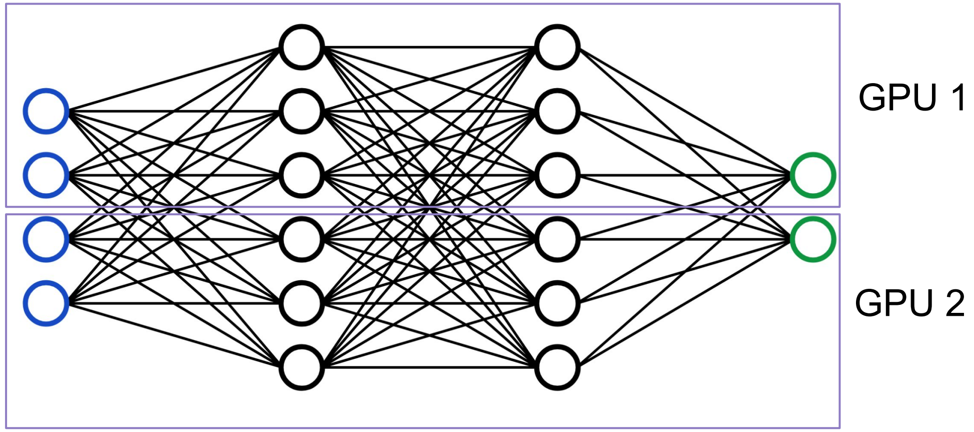


Tensor parallelism

In this framework we split the tensor operation done at each layer among different GPUs

similarly to what we would have done for matmul

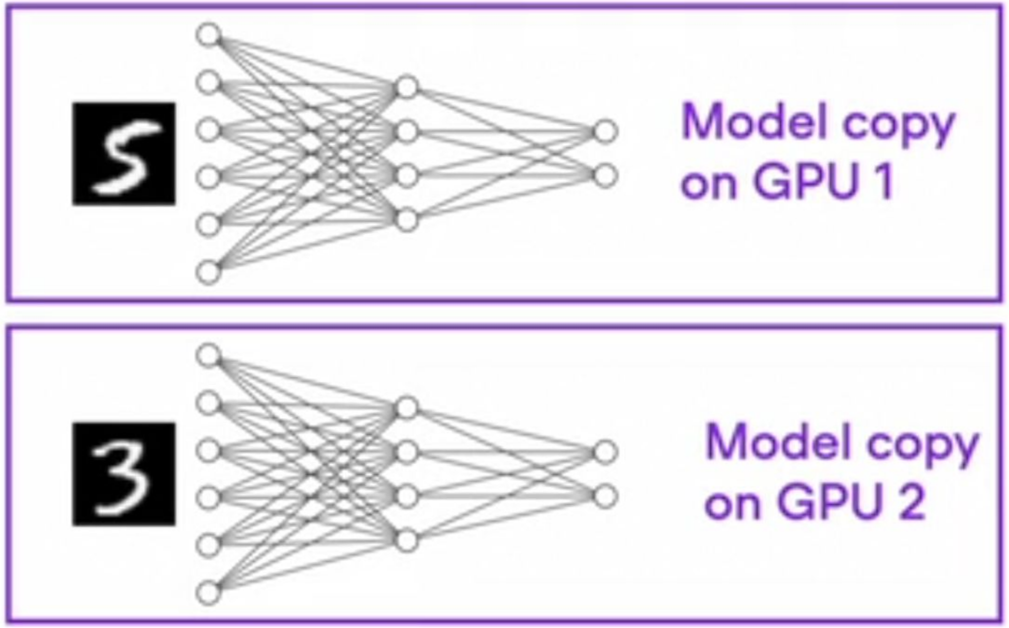




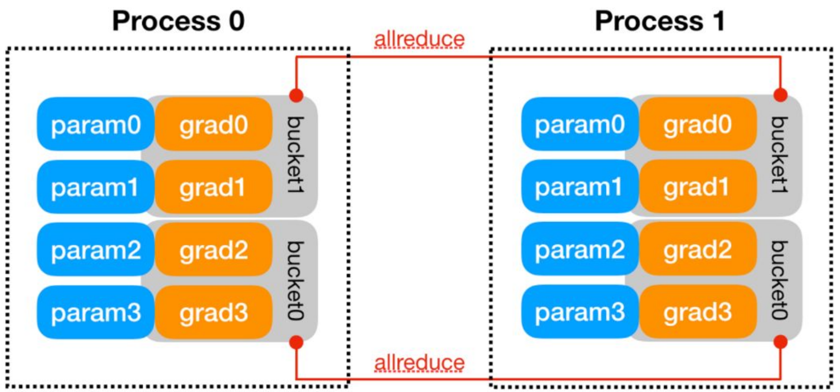
Data Parallelism

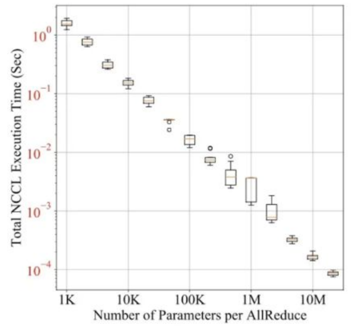
In this framework we split batches to train DL model

into different GPUs

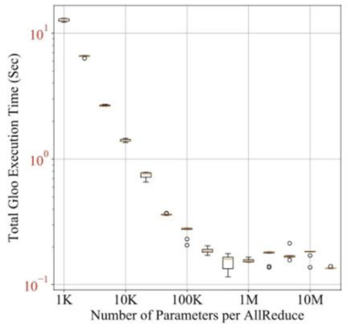


Distributed Data Parallel (DDP)

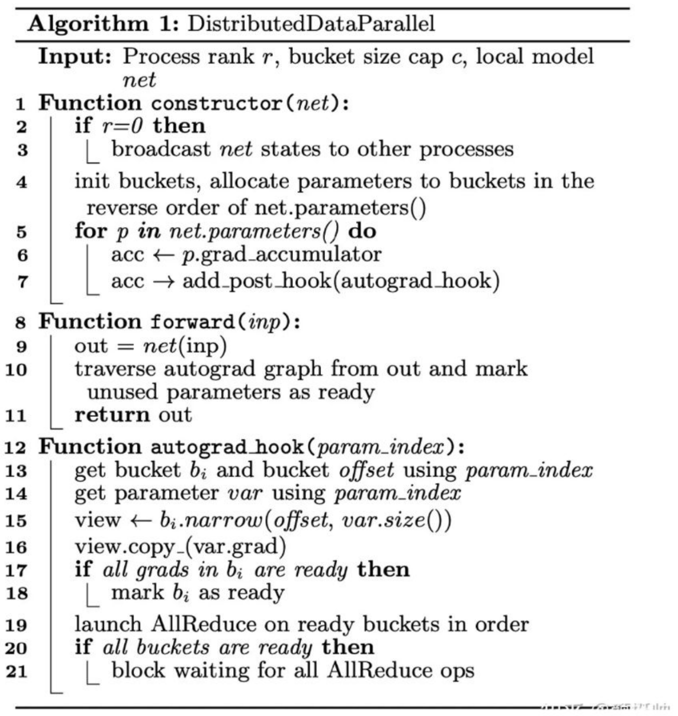




(a) NCCL



(b) GLOO



```
Algorithm 1: DistributedDataParallel
Input: Process rank $r$, bucket size cap $c$, local model $net$
Function constructor($net$):
    if $r=0$ then
        broadcast $net$ states to other processes
    init buckets, allocate parameters to buckets in the
      reverse order of net.parameters()
    for $p$ in $net.parameters()$ do
        acc ← $p$.grad_accumulator
        acc → add_post_hook(autograd_hook)
Function forward($inp$):
    out = $net$(inp)
    traverse autograd graph from out and mark
      unused parameters as ready
    return out
Function autograd_hook($param\_index$):
    get bucket $b_i$ and bucket $offset$ using $param\_index$
    get parameter $var$ using $param\_index$
    view ← $b_i.narrow(offset, var.size())$
    view.copy_(var.grad)
    if all grads in $b_i$ are ready then
        mark $b_i$ as ready
    launch AllReduce on ready buckets in order
    if all buckets are ready then
        block waiting for all AllReduce ops
```

Common guidelines by Pytorch documentation

1. Use DistributedDataParallel (DDP), if your model fits in a single GPU but you want to easily scale up training using multiple GPUs.
   - Use torchrun, to launch multiple pytorch processes if you are using more than one node.
   - See also: Getting Started with Distributed Data Parallel
2. Use FullyShardedDataParallel (FSDP2) when your model cannot fit on one GPU.
   - See also: Getting Started with FSDP2
3. Use Tensor Parallel (TP) and/or Pipeline Parallel (PP) if you reach scaling limitations with FSDP2.
   - Try our Tensor Parallelism Tutorial
   - See also: TorchTitan end to end example of 3D parallelism

More complex strategies for DL training

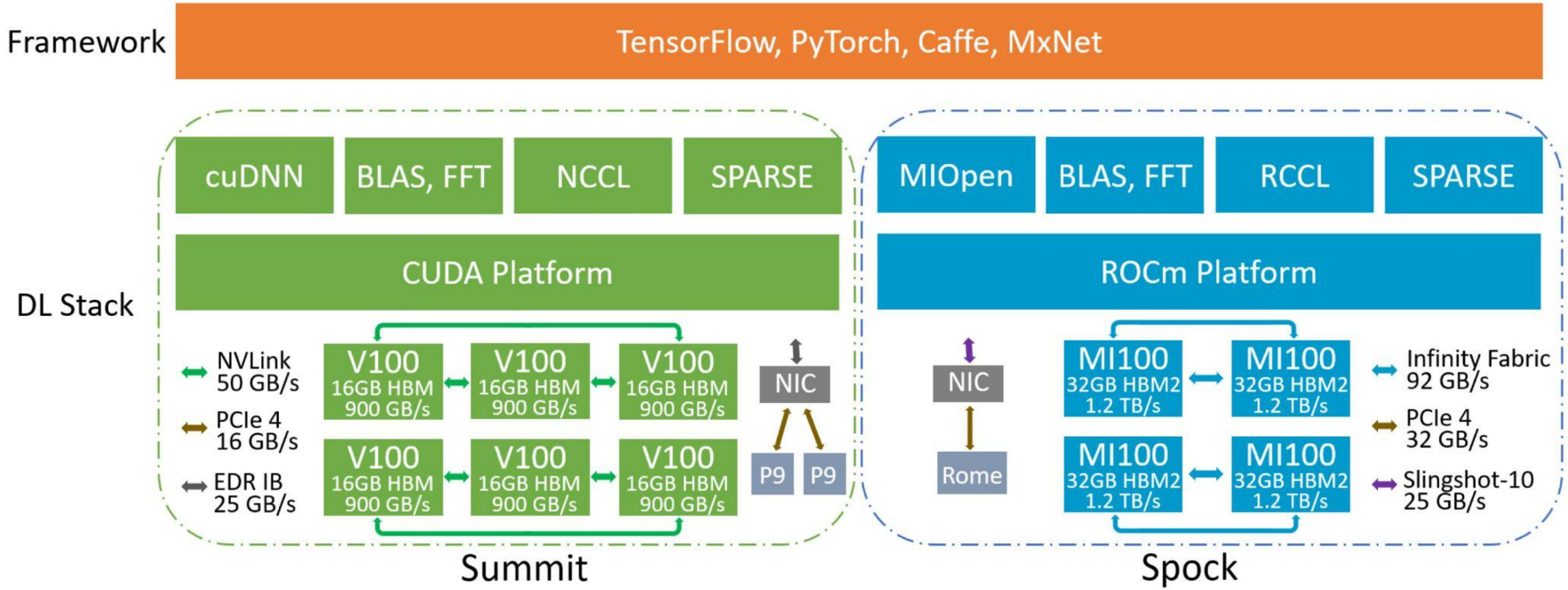
Sequence parallelism and pipeline parallelism frameworks

are obtained combining the previous approaches,

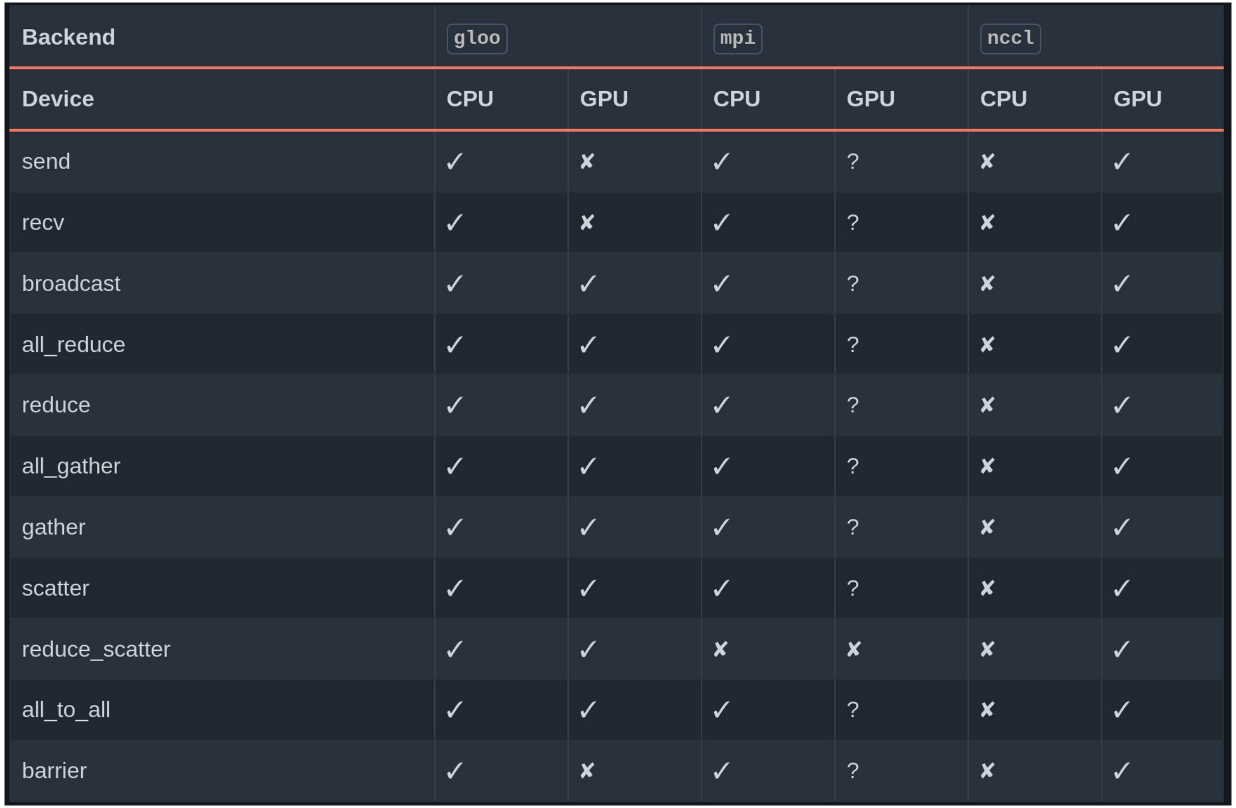
and are typically applied to DL models dealing with

spatio-temporal data.

What’s behind Pytorch/Tensorflow?

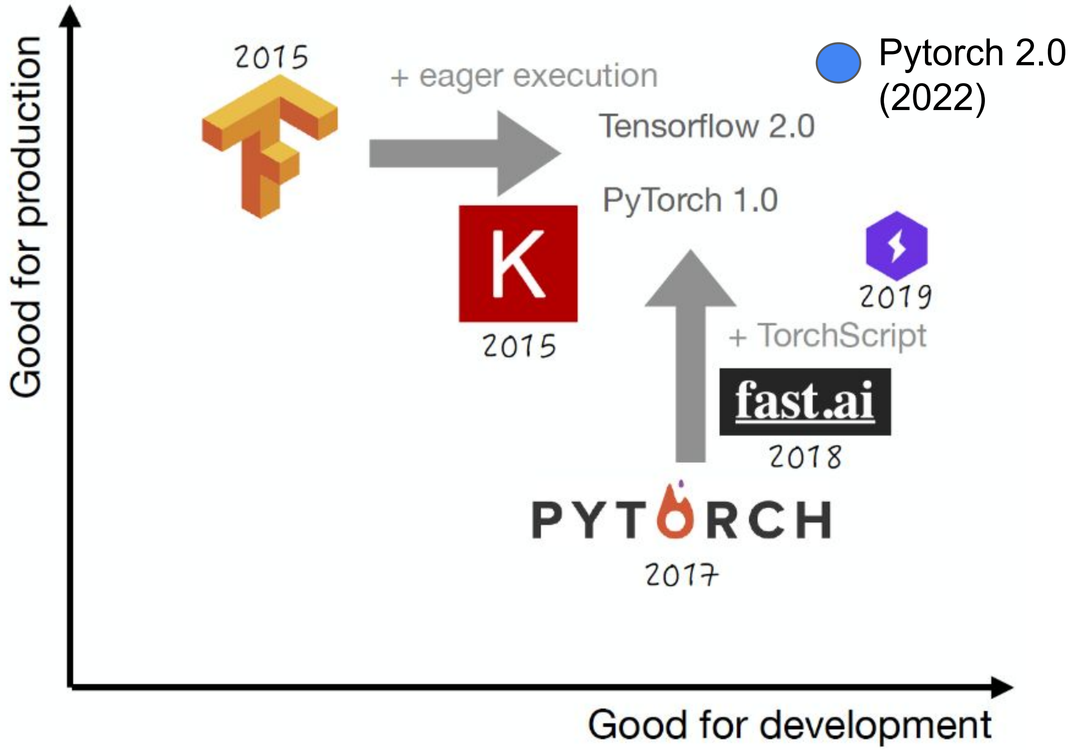


Backend of torch.distributed

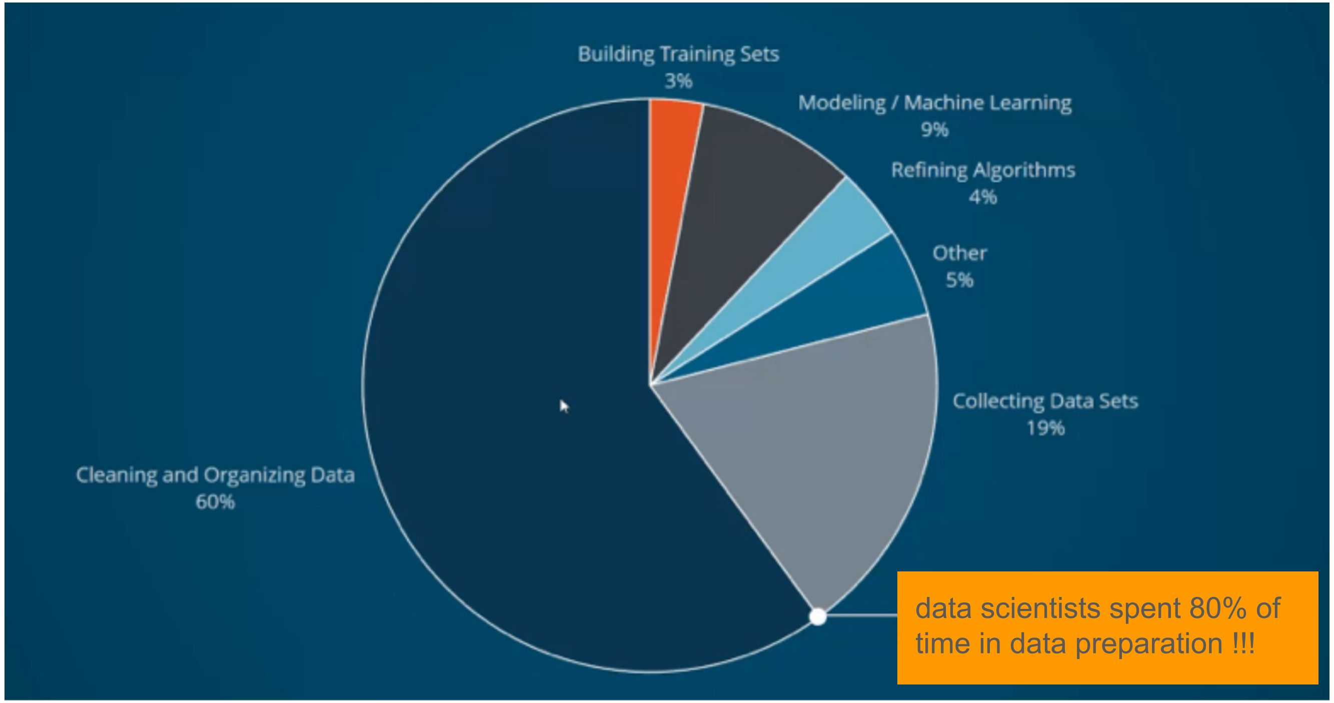


Backend	`gloo`		`mpi`		`nccl`	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✘	✓	?	✘	✓
recv	✓	✘	✓	?	✘	✓
broadcast	✓	✓	✓	?	✘	✓
all_reduce	✓	✓	✓	?	✘	✓
reduce	✓	✓	✓	?	✘	✓
all_gather	✓	✓	✓	?	✘	✓
gather	✓	✓	✓	?	✘	✓
scatter	✓	✓	✓	?	✘	✓
reduce_scatter	✓	✓	✘	✘	✘	✓
all_to_all	✓	✓	✓	?	✘	✓
barrier	✓	✘	✓	?	✘	✓

Python most used libraries/frameworks for DL

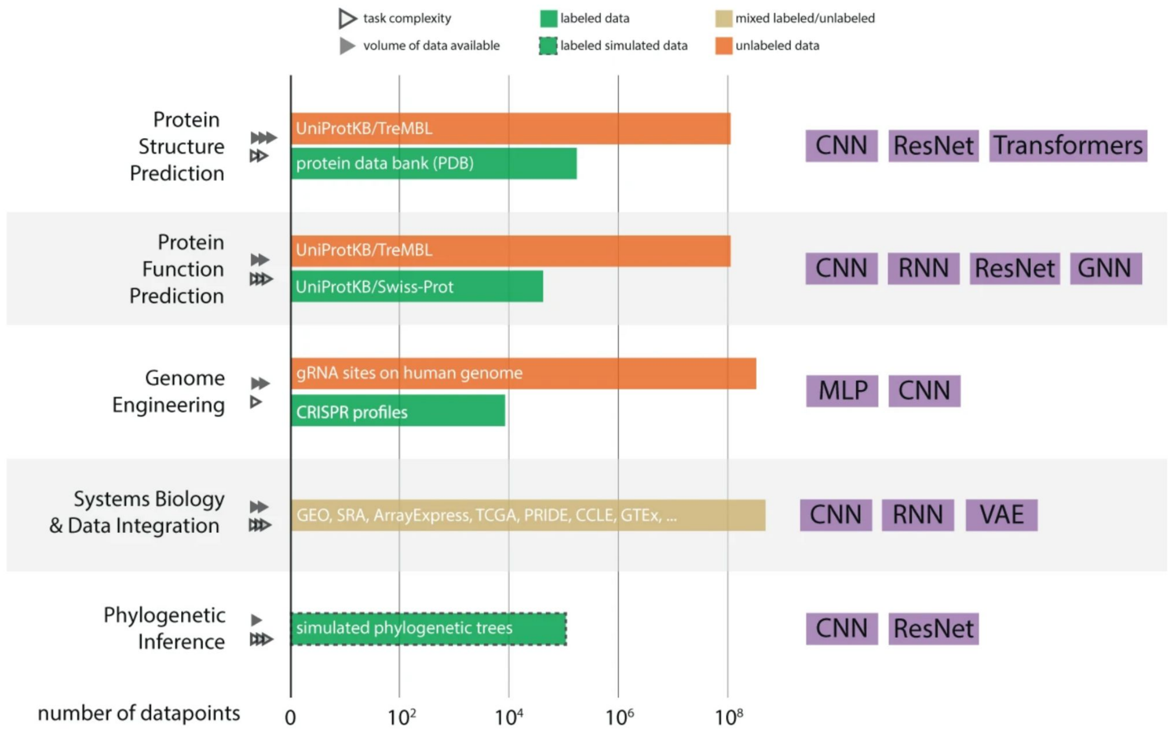


Do we need GPUs also for other ML tasks?



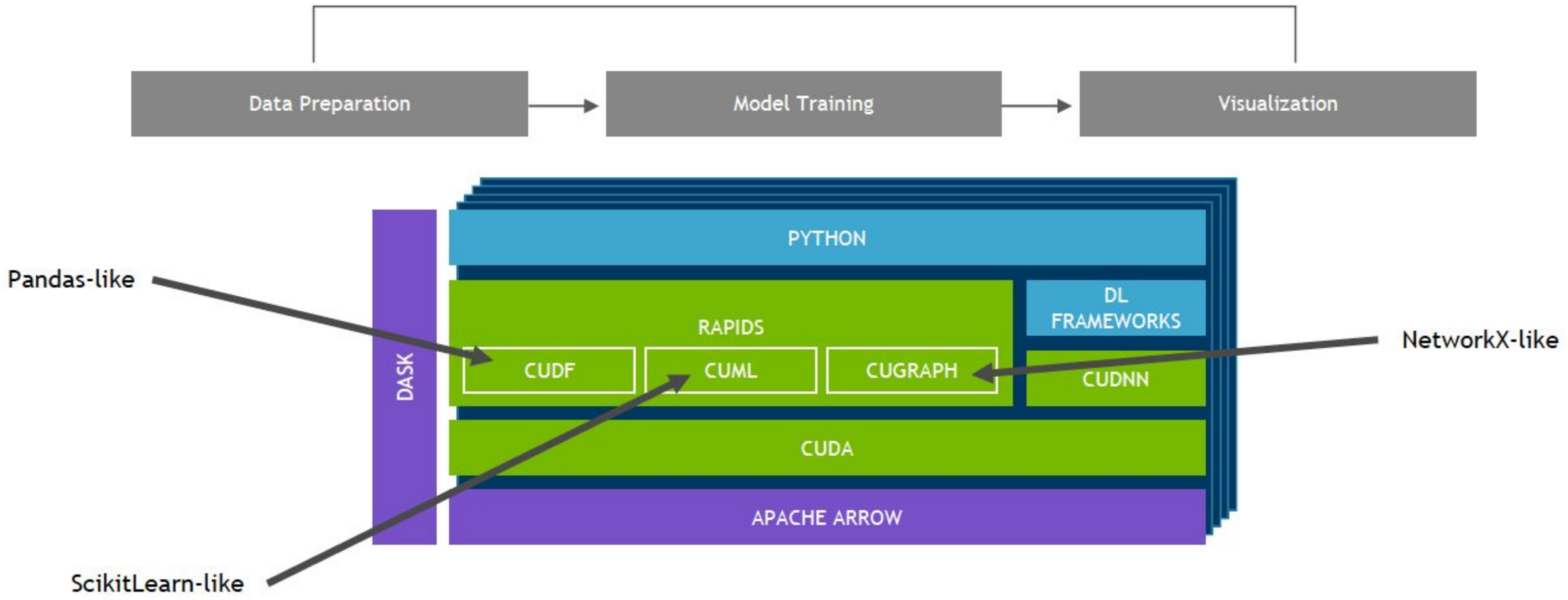
data scientists spent 80% of time in data preparation !!!

Typical sizes of DL data sets

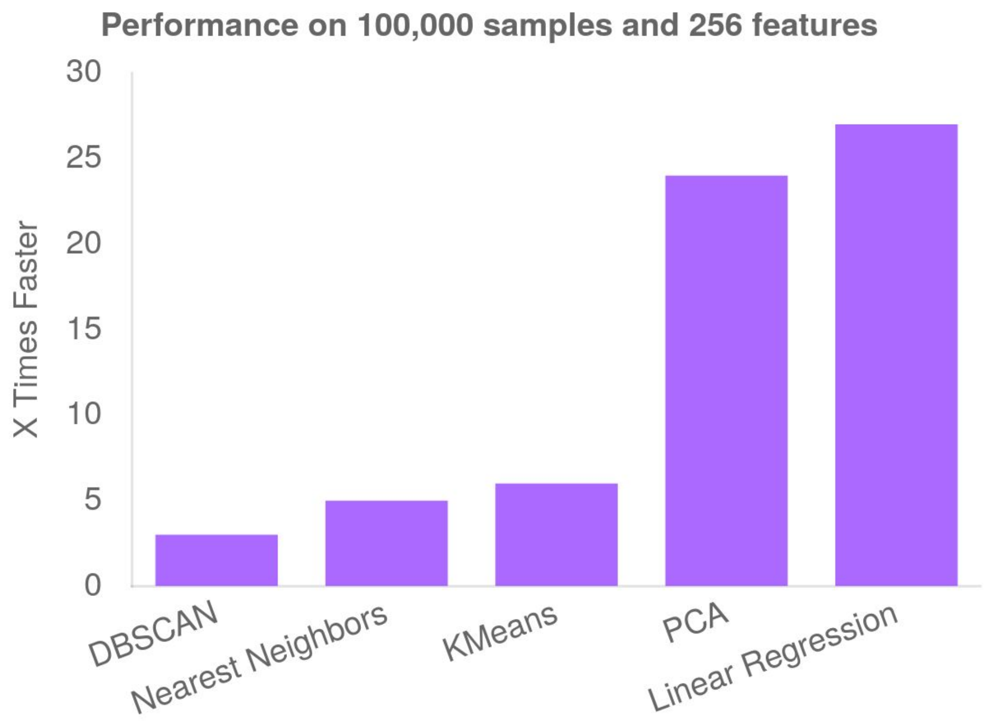


The increasing complexity of the new datasets, typical of big data epoch motivates the need for GPU-based libraries for feature analysis and data preprocessing

GPU-based libraries outside of Pytorch

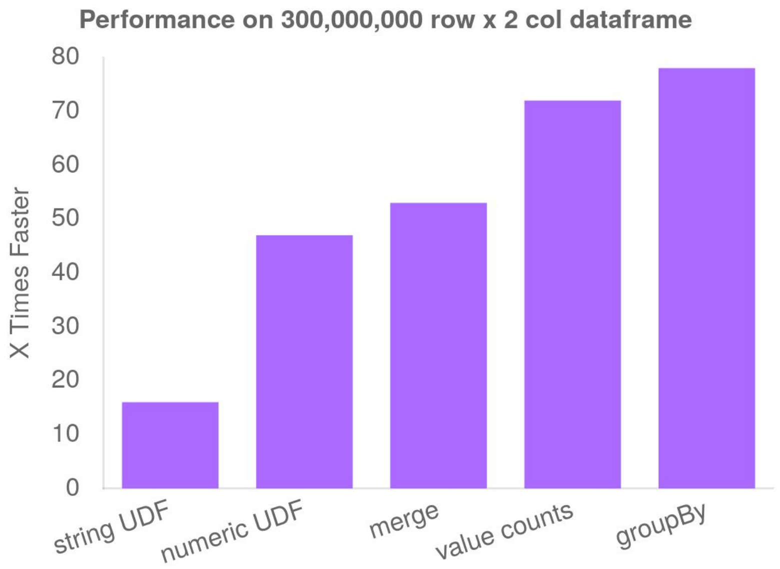


GPU for classical ML



* Benchmark on AMD EPYC 7642 (using 1x 2.3GHz CPU core) w/ 512GB and NVIDIA A100 80GB (1x GPU) w/ scikit-learn v1.2 and cuML v23.02

GPUs for data preprocessing



* Benchmark on AMD EPYC 7642 (using 1x 2.3GHz CPU core) w/ 512GB and NVIDIA A100 80GB (1x GPU) w/ pandas v1.5 and cuDF v23.02

References

Milan Jain, Sayan Ghosh, Sai Pushpak Nandanoori, 2022, *Workload Characterization of a Time-Series Prediction System for Spatio-Temporal Data*

Junqi Yin et. al, 2021, *Comparative evaluation of deep learning workloads for leadership-class systems*

NVIDIA Booklet on GPU development, 2021

Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions