Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Distributed Machine Learning and LLMs

Corso di Sistemi e Architetture per Big Data

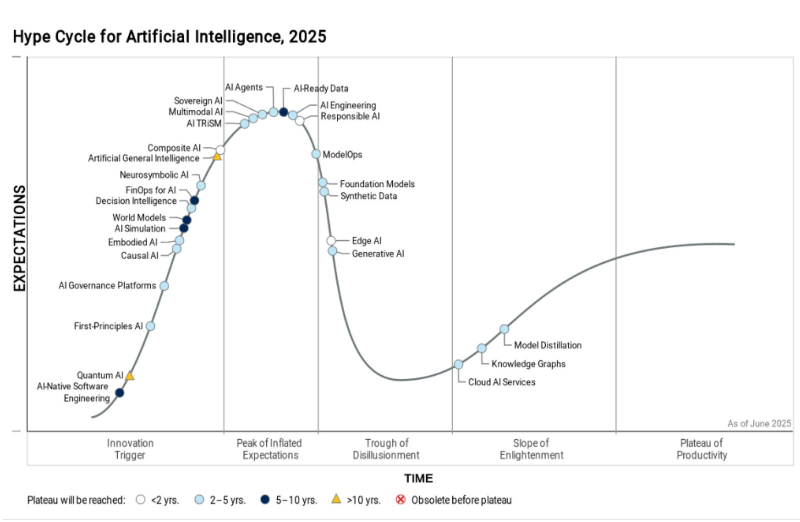
A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Artificial Intelligence and Machine Learning

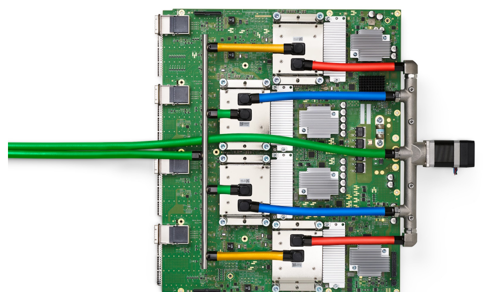
- AI and ML hype



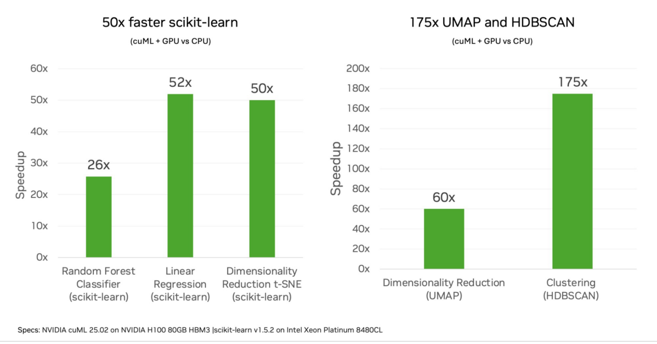
AI/ML accelerators and tools

- Huge leap in parallelization and innovation in AI/ML infrastructure and tools

- **Tensor Processing Unit (TPU)**: AI accelerator application-specific integrated circuit (ASIC) specialized in calculations with *tensors* (multi-dimensional matrices)
- Also as Cloud service https://cloud.google.com/tpu/

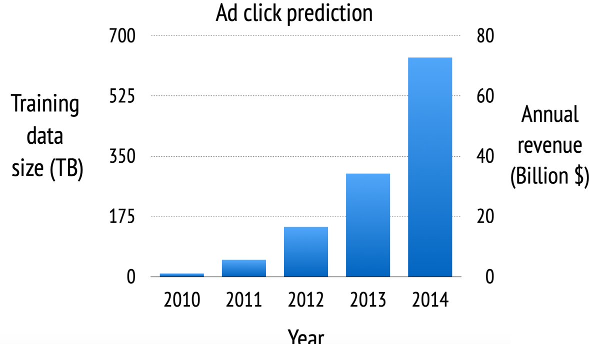


- Widely-used deep learning frameworks (e.g., TensorFlow, PyTorch, Scikit-learn) are **GPU-accelerated**
- Not only training, but also inference



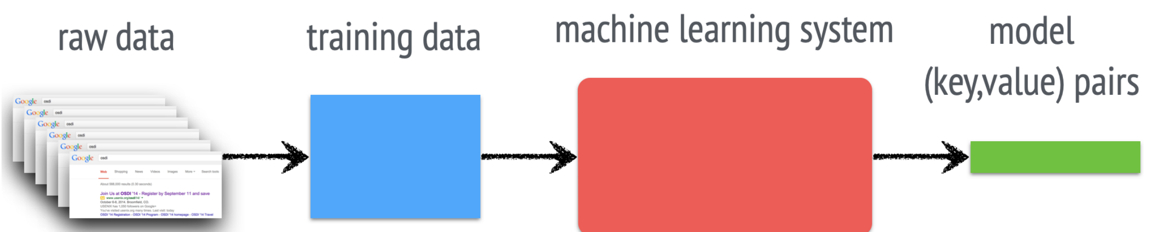
Is there a case for distributed ML?

- ML systems:
  - Drive significant revenue
  - Benefit from humongous amount of data
  - Outscale even powerful machines (GPUs, TPUs)
- Which systems? Example: ad click prediction



- How to face scalability needs? Let's **distribute ML**

What do ML algorithms look like?



- Common feature of ML algorithms?
  - Iterative in nature
- Key challenges:
  - Lots of data
  - Lots of parameters
  - Lots of iterations

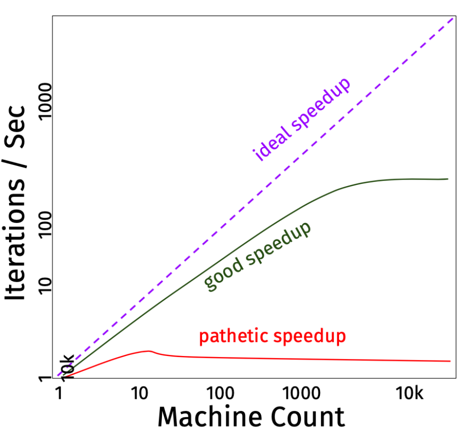
Scale of industry ML problems

- A taste of scale of ML industry problems
  - 100 billion examples
  - 10 billion features
  - 10TB - 10P training data
  - 100 - 1000 machines
- It's a problem of scale and scale changes everything!

> **Scale** has been the single most important force driving changes in system software over the last decade
>
> *– Technical perspective: Is scale your enemy, or is scale your friend?*
> John Ousterhout, CACM 54(7):110, July 2011.

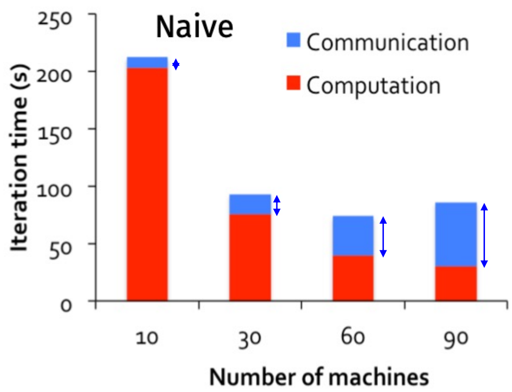
Scaling out distributed ML

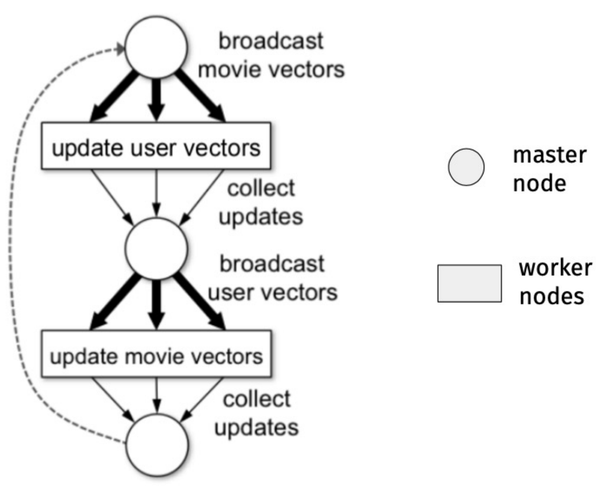
- 10-100s nodes enough for data/model
- Scale out for throughput
- Goal: more iterations/sec
  - Best case: 100x speedup from 1000 machines
  - Worst case: 50% slowdown from 1000 machines
- Can you think of reasons for performance degradation?



Challenge of communication overhead

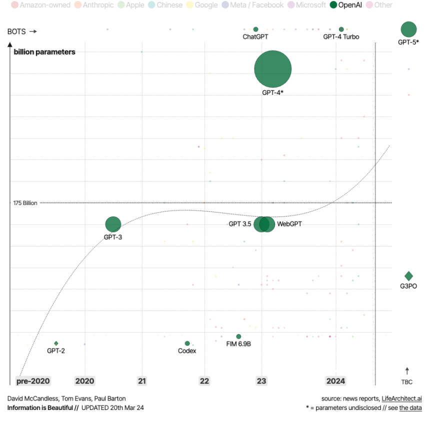
- Communication overhead scales badly with number of machines
  - E.g., Netflix-like recommender system based on matrix factorization





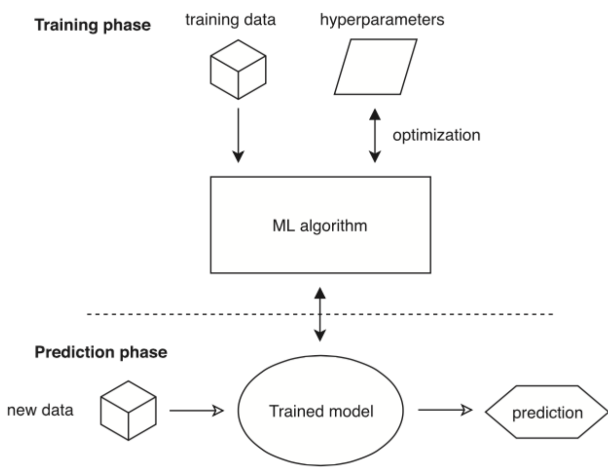
Requirements for distributed ML

- Scale to industry-size problems
  - Huge model size of large foundation models (LLMs), whose performance improves with model size and data volume
    - GPT-3 had 175 billion parameters (variables and inputs within model), GPT-4 is 10x larger
- Efficient communication
- Fault tolerance
- Easy to use



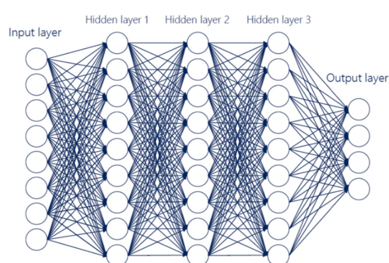
Distributed training and inference

- What can we perform in a distributed manner?
  - Training: build models
  - Inference: make predictions



Parallelization methods for distributed training

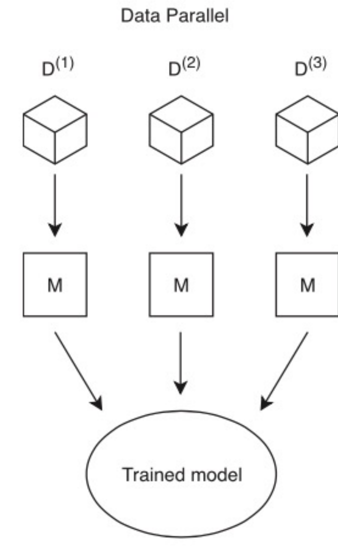
- Let's first focus on distributed training
  - We consider deep neural networks (DNNs), that is artificial neural networks that have an input layer, many hidden layers, an output layer

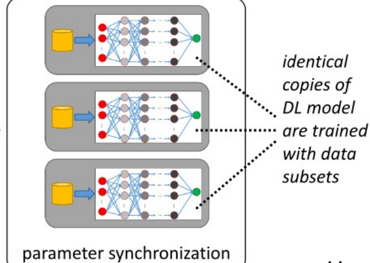


- Methods for distributed training
  1. Data parallelism: the usual SPMD approach
  2. Model parallelism
  3. Pipeline parallelism
  - Plus hybrid forms of parallelism not explored

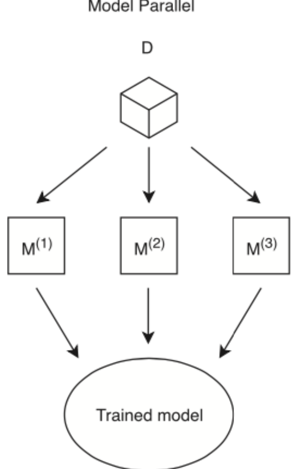
Method 1: Data parallelism

- Workers (machines or devices, e.g., GPUs) load an identical copy of model (M)
- Training data is split ($D^{(1)}$, $D^{(2)}$, …) into non-overlapping chunks or (slices) and fed into model replicas of workers for training
- Each worker performs training on its chunks of training data, which leads to updates of model parameters
  - Model parameters between workers need to be synchronized: how?



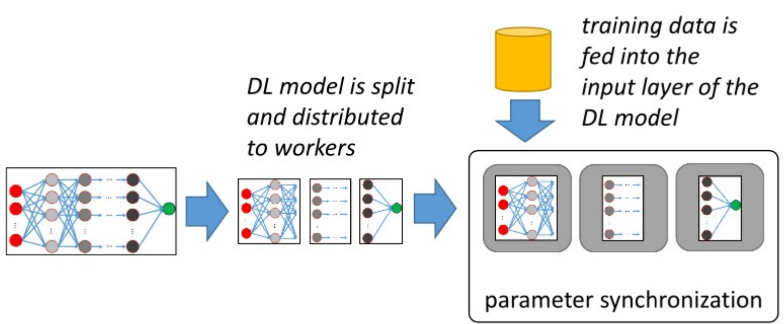


Method 2: Model parallelism



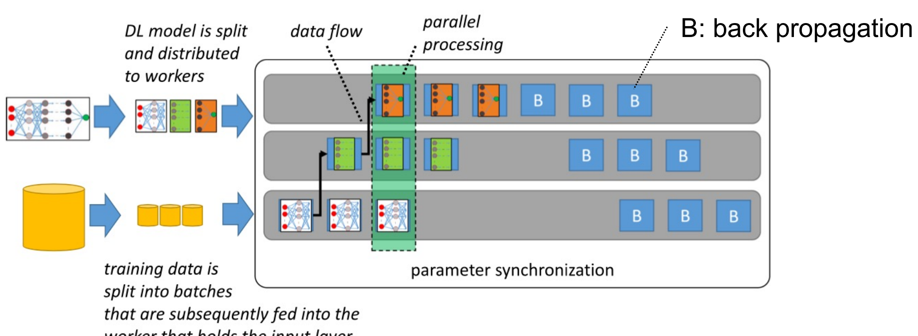
- Model is split ($M^{(1)}$, $M^{(2)}$, …) and each worker loads a different part of model for training
  - The model is the aggregate of all model parts
- Workers load an identical copy of data (D)

Method 2: Model parallelism



- Use case: Deep Learning (DL)
- Main idea: partition DNN layers among different workers
  - Worker(s) that hold input layer of DL model are fed with training data
  - In the forward pass, they compute their output signal which is propagated to workers that hold the next layer of DL model
  - In the backpropagation pass, gradients are computed starting at workers that hold the output layer of the DL model, propagating to workers that hold the input layers of the DL model

Method 3: Pipeline parallelism



- Combines model parallelism with data parallelism
- Use case: DL
  - Model is split and each worker loads a different part of model for training; training data is split into micro-batches
  - Every worker computes output signals for a set of micro-batches, propagating them to subsequent workers
  - In the backpropagation pass, workers compute gradients for their model partition for multiple micro-batches, immediately propagating them to preceding workers

Parallelization methods: Pros and cons

- Data parallelism
  - ✓ Can be used with every ML algorithm with an independent and identical distribution (i.i.d.) assumption over data samples (i.e., most ML algorithms)
  - ✓ Does not require domain knowledge of model
  - ✗ Parameter synchronization may become bottleneck
  - ✗ Does not help when model size is too large to fit on a single machine

Parallelization methods: Pros and cons

- Model parallelism
  - ✗ Challenge: how to split the model into partitions that are assigned to parallel workers
    - Cannot automatically be applied to every ML algorithm, because model parameters generally cannot be split up
  - ✓ Reduced model's memory footprint
    - As model is split, less memory is required for each worker
  - ✗ Heavy communication needed between workers

Optimizations for data parallelism

- Challenges of parameter synchronization in data parallelism

1. How to synchronize parameters
   – Centralized or decentralized manner?
2. When to synchronize parameters
   – Should workers be forced to synchronize after each batch, or do we allow them more freedom to work with potentially stale parameters?

- How to minimize communication overhead for parameter synchronization

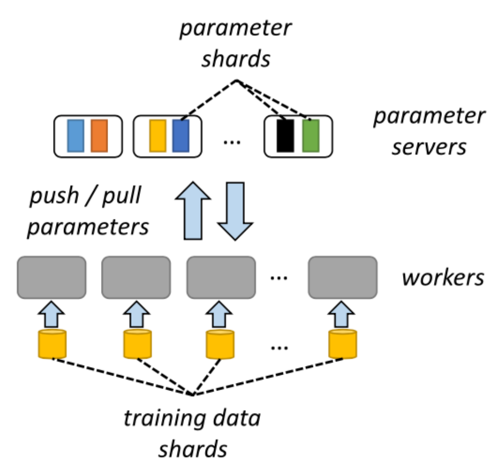
How to synchronize parameters: architecture

1. *How to synchronize parameters*
   – Centralized or decentralized architecture

- Main alternatives:
  – Centralized: parameter server
  – Decentralized: all-reduce

Centralized: Parameter server

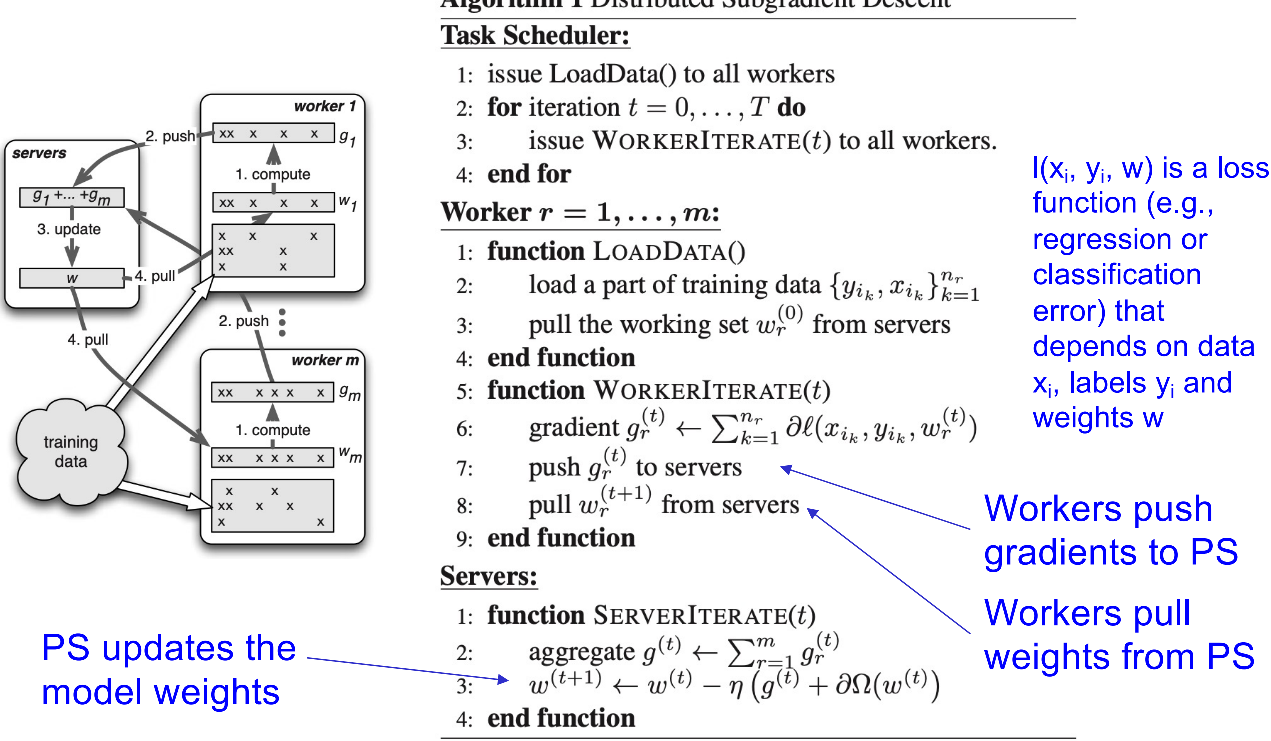
- Popular approach to distributed training introduced by Google with DistBelief
- Pull: workers pull the latest model weights (parameters) from Parameter Server (PS)



- Compute: workers compute gradients on their local shard of data
- Push: workers push the computed gradients back to PS
- Update: PS aggregates the gradients and updates the global model weights

Li et al., Scaling Distributed Machine Learning with the Parameter Server, OSDI'14

Parameter server: distributed gradient descent



Algorithm 1 Distributed Subgradient Descent

Task Scheduler:

1: issue LoadData() to all workers
2: **for** iteration $t = 0, \ldots, T$ **do**
3: issue WorkerIterate(t) to all workers.
4: **end for**

Worker $r = 1, \ldots, m$:

1: **function** LoadData()
2: load a part of training data $\{y_{i_k}, x_{i_k}\}_{k=1}^{n_r}$
3: pull the working set $w_r^{(0)}$ from servers
4: **end function**
5: **function** WorkerIterate(t)
6: gradient $g_r^{(t)} \leftarrow \sum_{k=1}^{n_r} \partial \ell(x_{i_k}, y_{i_k}, w_r^{(t)})$
7: push $g_r^{(t)}$ to servers
8: pull $w_r^{(t+1)}$ from servers
9: **end function**

Servers:

1: **function** ServerIterate(t)
2: aggregate $g^{(t)} \leftarrow \sum_{r=1}^{m} g_r^{(t)}$
3: $w^{(t+1)} \leftarrow w^{(t)} - \eta \left(g^{(t)} + \partial \Omega(w^{(t)}) \right)$
4: **end function**

$l(x_i, y_i, w)$ is a loss function (e.g., regression or classification error) that depends on data x_i, labels y_i and weights w

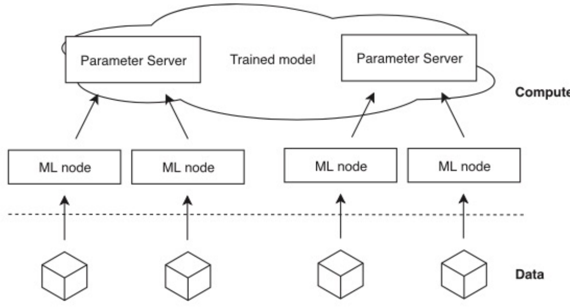
Workers push gradients to PS

Workers pull weights from PS

PS updates the model weights

Parameter server: multiple PSs

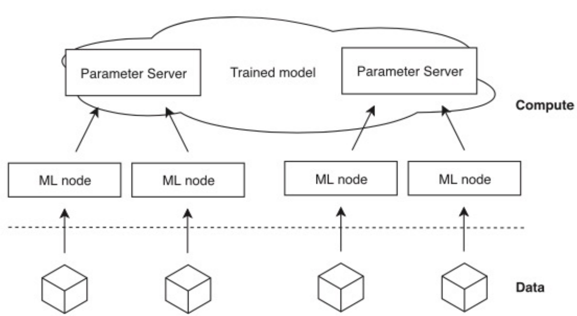
- To mitigate performance bottleneck and SPoF, use multiple parameter servers which manage the model's parameters



  - Parameters are partitioned among multiple PSs and each PS is responsible for maintaining the parameters in its partition
  - When a worker sends a gradient, it partitions that gradient vector and send each chunk to the corresponding PS; later, it will receive the corresponding chunk of the updated model from that PS

Parameter server: multiple PSs

- To mitigate performance bottleneck and SPoF, use multiple parameter servers which manage the model's parameters

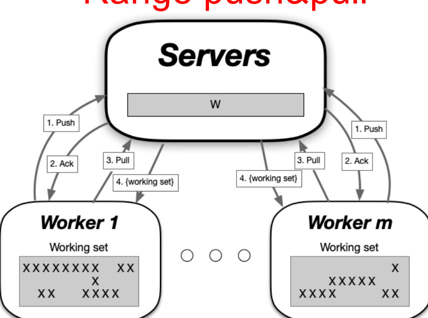


  – Parameters are partitioned among multiple PSs and each PS is responsible for maintaining the parameters in its partition
  – When a worker sends a gradient, it partitions that gradient vector and send each chunk to the corresponding PS; later, it will receive the corresponding chunk of the updated model from that PS

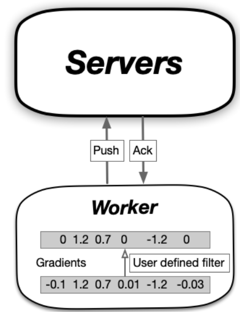
Parameter server: efficient communication

- To optimize PS communication efficiency, multiple techniques:
  – Store model parameters as key-value (KV) pairs
    - Weights and gradients are structured as a distributed KV-store: workers can request or update only specific parameters or layers
  – Range push&pull
    - Parameters are batched into contiguous blocks
  – User-defined filters
    - Filters are used to drop insignificant gradients (close to zero)

Range push&pull

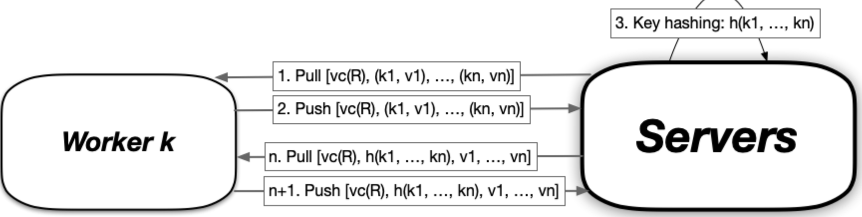


User-defined filters

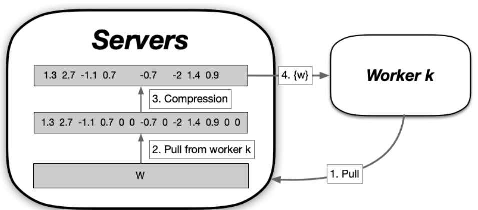


Parameter server: efficient communication

- To optimize PS communication efficiency, multiple techniques:
  – Key caching
    - In the first iteration, the workers cache the keys (do not change)
    - In all subsequent iterations, the workers transmit only the values to reduce network overhead

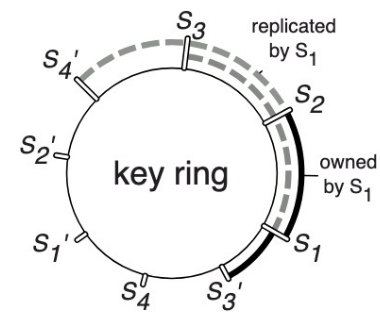


  – Compression



Parameter server: fault tolerance

- Consistent hashing
  – Servers and parameters are mapped on ring
  – Parameters are replicated on k servers



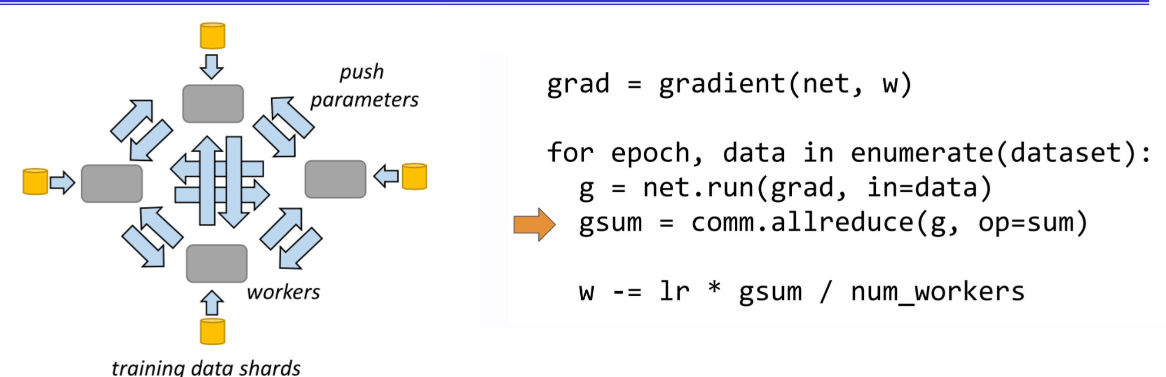
- Chain replication

- Vector clocks
  – To keep track of the iteration progress of workers
- Goal: failure recovery < 1 s without interrupting training

Parameter server: scalability

- Parameter server (stateful):
  - Critical operation: PS maintains the model state
  - Two-phase migration:
    - Parameter pre-copy
    - Final synchronization of updates
- Worker node (stateless):
  - Easier to manage
  - Scale-out: new workers receive assignment from Task Scheduler

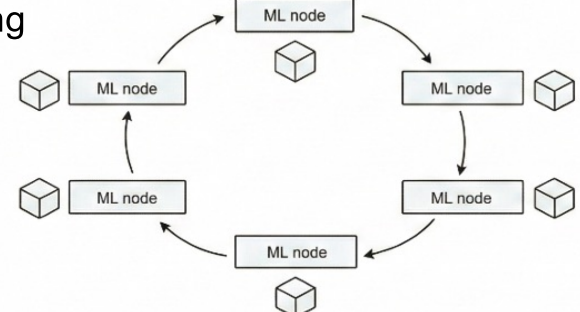
Decentralized: All-Reduce



- Every worker node has a copy of the full model weights
- No central server: workers communicate directly using collective communication such as All-Reduce
  - Computes some reduction (e.g., sum) of local data (e.g., gradients) on multiple workers to combine them and make the result (e.g., weights) available on all the workers
- Requires high-speed interconnects
  - E.g., NVLink (intra-node, 900 Gb/s), InfiniBand (inter-node, 400 Gb/s)

Decentralized: All-Reduce

- All-Reduce needs efficient implementation: naïve solution (all-to-all) is too costly
  - $O(n^2)$ messages with n nodes
- How? Use different topologies, such as ring or tree
- Ring All-Reduce
  - Nodes arranged in a logical ring
  - Each node sends/receives a chunk of data to/from its neighbors
  - Two phases: reduce-scatter (aggregation) + all-gather (broadcast)
  - Communication cost:
    - ✓ $O(m)$ per worker (m is data size), independent of n (as n grows large)
    - X But number of steps increases linearly with n



Decentralized: All-Reduce

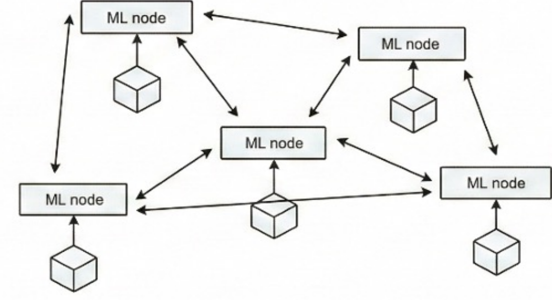
- Tree All-Reduce
  - Nodes arranged in a tree structure
  - Reduce phase: leaves send data to parents, parents aggregate
  - Broadcast phase: root sends reduced result down to leaves
  - Communication cost:
    - ✓ $O(\log n)$ steps
    - X But each step can involve large messages → can saturate network links at upper levels of tree
  - Efficient for latency-sensitive systems

Decentralized All-Reduce: pros and cons

- All-Reduce vs. Parameter Server
  - ✓ Simpler infrastructure: no need to deploy, manage, or scale centralized servers
  - ✓ No SPoF: robust against node dropouts; communication network dynamically updates without stopping the training
  - ✓ Lightweight checkpointing: avoids I/O bottleneck of checkpointing a centralized global model state
  - ✗ Communication cost increases with number of nodes (trees)
  - ✗ Sparse networks (e.g., rings) reduce network traffic but require more iterations to propagate weights and reach convergence
  - ✗ Algorithmic complexity: harder to debug and guarantee mathematical convergence
- All-Reduce: use for dense, massively parallel models (e.g., LLMs, transformers)
- Parameter Server: use for sparse, massive-scale models (e.g., recommendation systems)

Decentralized architectures: gossip-based

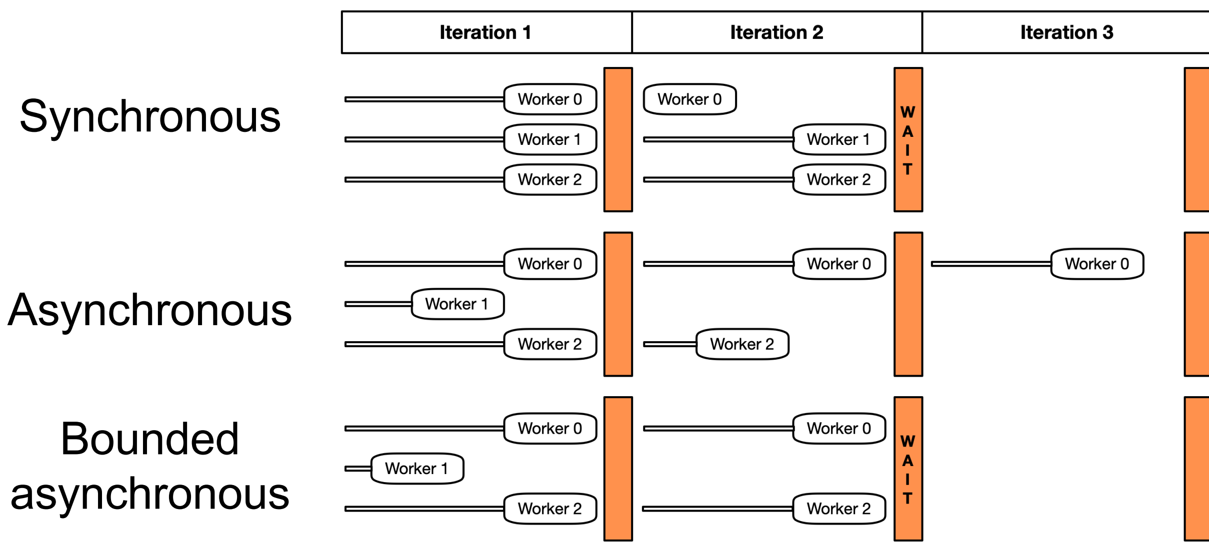
- Idea: exploit P2P collective intelligence using gossiping
- Eliminates the need for PS coordinator or global all-reduce communication
- Basic idea
  - Each node trains locally on its own data for one or more steps
  - Periodically, a node selects one or more neighbors
  - The nodes exchange model parameters (or gradients) and combine the received information, often by averaging
  - Repeating these local exchanges causes information to propagate through the network, eventually leading to approximate consensus



👉 Pros and cons of gossiping

When to synchronize parameters

2. *When to synchronize parameters* (PS focus)
   - Force workers to synchronize after each iteration
   - Or allow them more freedom to work with potentially stale parameters



When to synchronize: sync

- Synchronous (sync)
  - After each iteration (i.e., processing of a batch), workers synchronize their parameter updates, so that all workers use the same synchronized set of model parameters
  - Requires barriers between iterations
  - ✓ Model convergence is easier
  - ✗ *Straggler* problem, where the slowest worker slows down all others

Synchronization Barrier

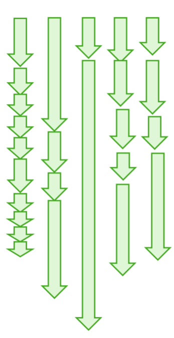
Synchronization Barrier

When to synchronize: alternatives

- How to address stragglers?
- Idea: relax synchronization
- How?
  – Asynchronous manner: a worker who finishes processing a batch can pull the current parameters from PS and start the next batch, even if other workers haven't finished processing the earlier batch
  – Suitable for geo-distributed training servers
- Be careful: usual trade-off between performance and model guarantees

When to synchronize: async

- Asynchronous (async)
  – No barriers at all: workers update their model completely independently from each other
  – Faster workers do not have to wait for stragglers to finish; they can keep pushing gradients and pulling (outdated) weights
  ✓ Completely avoids straggler problem
  ✗ No guarantees on a staleness bound, i.e., a worker may train on an arbitrarily stale model
  ✗ Hard to mathematically reason about model convergence



When to synchronize: bounded async

- Bounded asynchronous
  - Workers may train on stale parameters, but staleness is bounded
    - ML algorithms are robust, converge even with some stale state
  - ✓ Allows for mathematical analysis and proof of model convergence properties
  - ✓ Bound allows workers for more freedom in making training progress independently from each other, which mitigates the straggler problem and increases throughput

Architectures and synchronization: summing up

- Parameter server is flexible wrt. synchronization
- All-reduce is stricter, typically executed as a synchronous, blocking collective operation

Architecture	Communication pattern	Synchronization
Parameter Server	Point-to-point: push/pull with centralized server(s)	Can be synchronous or asynchronous
All-Reduce	Collective operation involving all nodes	Usually synchronous, but asynchronous variants exist
Gossip Learning	Neighbor-to-neighbor exchanges	Asynchronous / approximate by default

Example: TensorFlow

- TensorFlow: Python-friendly open-source library for ML and AI https://www.tensorflow.org/
  - Supports a wide range of ML and AI tasks, with focus on DL training and inference
  - Developed by Google Brain for internal, released in 2015
  - TensorFlow computations can be represented as a DAG of operations and data flow
  - Supports multiple devices
    - CPUs, GPUs and TPUs

TensorFlow graph representing a 2-layer NN

Example: TensorFlow

- tf.distribute.Strategy: TensorFlow API to distribute training across multiple devices https://www.tensorflow.org/api_docs/python/tf/distribute/Strategy
- Uses data parallelism to scale out model training
  - Supports both centralized (parameter server) and decentralized (all-reduce)
  - Supports both synchronous and asynchronous parameter update

Example: Pytorch

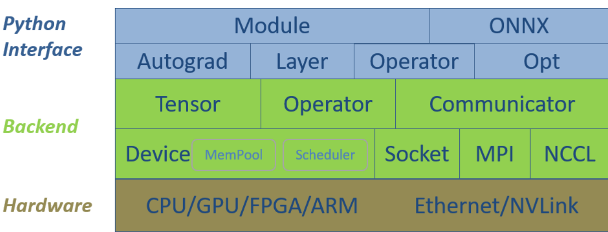
- Pytorch: open-source ML framework based on Torch library https://pytorch.org
- Scalable distributed training on multiple devices (CPUs, GPUs) by means of data parallelism and decentralized all-reduce https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html
  - All-reduce is built on top of efficient collective communication libraries: gloo (https://github.com/pytorch/gloo), MPI, and NVIDIA Collective Communications Library (NCCL https://developer.nvidia.com/nccl) https://docs.pytorch.org/docs/stable/distributed.html
- Also supports RPC-based distributed training for general distributed training scenarios
  - Can be used to implement parameter servers

Example: Apache Singa

- Apache Singa: open-source DL framework https://singa.apache.org/
- Distributed training on multiple devices (CPUs, GPUs) by means of different strategies (i.e., data parallelism, model parallelism and hybrid parallelism)

Layer					
Python Interface	Module			ONNX	
	Autograd	Layer	Operator	Opt	
Backend	Tensor	Operator	Communicator		
	Device (MemPool, Scheduler)	Socket	MPI	NCCL	
Hardware	CPU/GPU/FPGA/ARM		Ethernet/NVLink		

- DL models can be queried as stored procedures of RDBMS



Distributed LLM training

- Memory wall
  - Training a 70B model requires ~1.1TB of GPU RAM (VRAM) for weights, gradients, and optimizer states, excluding activations
    - VRAM footprint per parameter: 2B for weights, 2B for gradients, 12B for optimizer States (Adam)
  - Enterprise GPUs (e.g., NVIDIA H100) have 80GB or 141GB of VRAM → the model must be distributed
- High-speed hardware interconnects are required
  - Intra-node (inside server): NVIDIA NVLink or NVSwitch (1.8 Tb/s)
  - Inter-node (across racks): InfiniBand or RoCE (RDMA over Converged Ethernet); leverage GPUDirect RDMA
  - GPUDirect RDMA: bypasses CPU and host RAM, streaming data directly between GPU VRAM and NICs

Main approaches for Distributed LLM training

- Idea: exploit 3 distinct dimensions of parallelism, known as 3D Parallelism

1. Distributed Data Parallelism (DDP) and Sharding (FSDP)
   - DDP (we know it): replicates model, splits dataset, uses All-Reduce; each worker processes a different slice of the dataset (mini-batch), and gradients are synchronized via All-Reduce. Limitation: model must fit on smallest GPU memory
   - FSDP / ZeRO-3: when model is too large, Fully Sharded Data Parallelism shards weights, gradients, and optimizer states across workers; workers fetches layers just-in-time via All-Gather and discards them immediately to free VRAM

Main approaches for Distributed LLM training

- Idea: exploit 3 distinct dimensions of parallelism, known as 3D Parallelism

2. Tensor Parallelism (TP)
   - Slices individual layer matrices (Multi-Head Attention or MLP layers) within a single step (column-parallel or row-parallel); GPUs compute partial matrix multiplications concurrently, followed by intra-node All-Reduce or All-Gather to unify the results
   - Requires high-frequency communication → limited to intra-node (NVLink) to prevent network bottlenecks

Main approaches for Distributed LLM training

- Idea: exploit 3 distinct dimensions of parallelism, known as 3D Parallelism

3. Pipeline Parallelism (PP)
   - Slices the model sequentially by layer blocks across different nodes (e.g., node 1: layers 1-20, node 2: layers 21-40); node 1 computes its forward pass and forwards the boundary activations to node 2
   - Optimization: to minimize hardware idle time (pipeline bubble), the global batch is split into micro-batches (e.g., 1F1B scheduling)
     - 1F1B: each node alternates between executing one forward micro-batch and one backward micro-batch
   - Network fit: relatively low communication volume → ideal for inter-node (InfiniBand)

Distributed LLM training: challenges

- Stragglers: LLM training is synchronous at collective boundaries
- Activation checkpointing: to save VRAM, systems drop most intermediate activations during the forward pass and recalculate them on-the-fly during the backward pass
  - Trade-off ~33% increase in compute overhead
- Fault tolerance: asynchronous checkpointing to save training state to high-throughput distributed storage (e.g., HDFS) without stalling execution, allowing quick recovery from node failure

Distributed training challenges

- System and architectural challenges
  - Network overhead: communication scale grows with model size
  - Hardware heterogeneity: mixing different GPU generations or network links introduces stragglers
  - Fault tolerance: systems must survive hw failures without resetting
- Environmental and compute costs
  - Huge resource footprint: massively energy-hungry infrastructure
    - LLaMA-2-70B example: 1.7M GPU hours and emitted roughly 293 metric tons of CO_2 equivalent during training
- Data sovereignty and security
  - Privacy constraints: cross-border data laws (GDPR, CCPA) prevent pooling datasets into a single cloud
  - Paradigm shift: moving to Federated Learning to keep data local

Distributed ML inference

- Why also distributed inference?
  - Large models (e.g., GPT-4) are too big for a single accelerator/device
  - Low-latency requirements: real-time apps (chatbots, autonomous driving, robotics) require ms responses
  - Hardware constraints: memory, computation, energy/thermal budget
- How?
  - Model partitioning: split model graph across multiple devices, layers, or nodes to compute a single forward pass cooperatively
  - Training vs. inference distribution
    - Training optimizes for throughput
    - Inference optimizes for latency
- Use case: model splitting in the Edge-Cloud continuum

Recall: LLM inference

- LLM inference: generate outputs based on input prompts
- Differently from single parallel pass in DL models, LLMs generate text token-by-token
- Two-stage execution:
  - **Prefill** (*context enconding*): the model processes the input tokens (prompt) for the first time to produce key and value caches for the attention layers
  - **Decode** (*autoregressive generation*): generates new tokens sequentially, one by one using cached key/value pairs from prefill; each token depends on previous ones
- Prefill is compute-bound, decode is memory-bound
- Metrics:
  - Prefill: time to first token
  - Decode: time between tokens

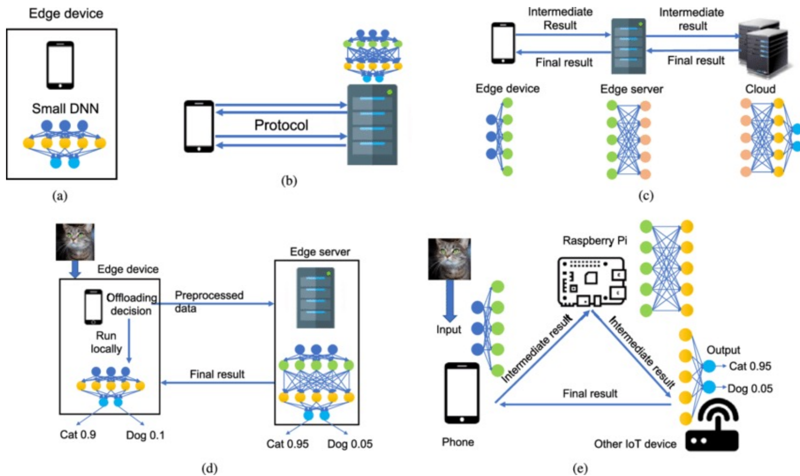
Distributed inference: execution paradigms

1. Synchronous inference (static batching)
   - Incoming requests are processed in fixed-size batches; all nodes process the same batch and advance through model layers in lockstep
   - Goal: maximize hardware utilization
   - X Increased latency while waiting for the batch to fill
   - X Performance limited by slowest request or device (straggler)
2. Asynchronous inference (dynamic batching)
   - Requests are processed immediately on arrival
   - Ideal for irregular, real-time traffic (e.g., live dashboards, mobile apps), maximize hardware saturation

- LLM inference: continuous batching, on a per token basis

Distributed inference: architectures

- Edge-Cloud inference architecture
  - How to setup distributed inference?
- Different alternatives



(a) On-device computation (b) Secure two-party communication (c) Computing across Edge-Cloud with DNN model partitioning (d) Offloading with model selection (e) Distributed computing across edge devices with DNN model partitioning

Distributed inference: optimization techniques

- Quantization and model compression
  - Reduce memory footprint, accelerate compute, and reduce network traffic by downscaling weights (e.g., from FP16 to INT8)
- Caching and reuse of results
  - E.g., KV-Cache for LLM: converts a compute-bound problem into a memory-lookup problem
- Adaptive computation
  - Early exits: allows simple inputs (e.g., an easy-to-classify image) to exit the neural network at intermediate layers
  - Dynamic routing (e.g., Mixture-of-Experts): activates only a small, specific subset of the network dynamically based on input request

Distributed inference: frameworks

- NVIDIA TensorRT https://developer.nvidia.com/tensorrt
  - High-performance DL inference
- ONNX Runtime https://onnxruntime.ai
- Ray Serve https://docs.ray.io/en/latest/serve

- LLM inference:
  - vLLMs https://vllm.ai/
  - Petals https://petals.dev/

References

- Mayer et al., Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques, and Tools, ACM Computing Surveys, 2020 https://dl.acm.org/doi/epdf/10.1145/3363554
- Verbraeken et al., A Survey on Distributed Machine Learning, ACM Computing Surveys, 2020 https://dl.acm.org/doi/epdf/10.1145/3377454
- Li et al., Scaling Distributed Machine Learning with the Parameter Server, OSDI'14, 2014 https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf
- Zeng et al., Distributed training of large language models: A survey, https://doi.org/10.1016/j.nlp.2025.100174