TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Introduction to Distributed and Federated Machine Leaning

Corso di Sistemi e Architetture per Big Data

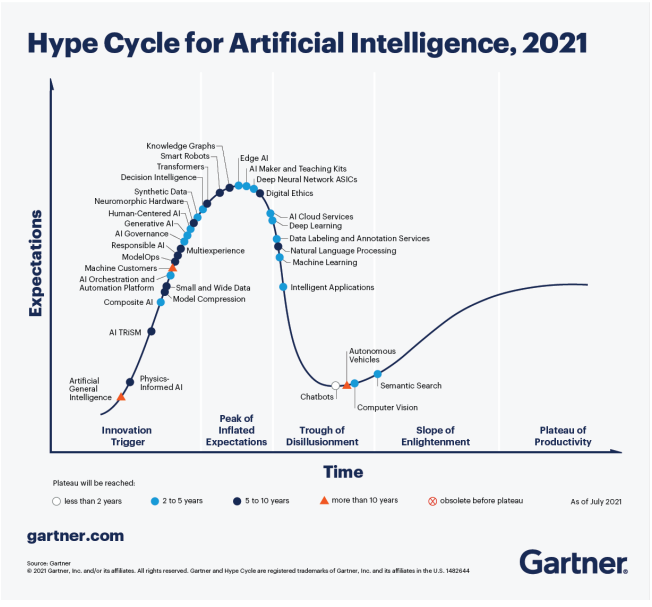
A.A. 2021/22

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Machine Learning

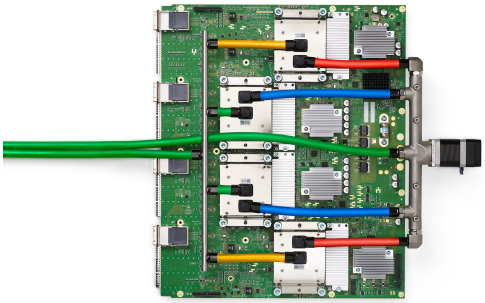
- The Machine Learning (ML) hype



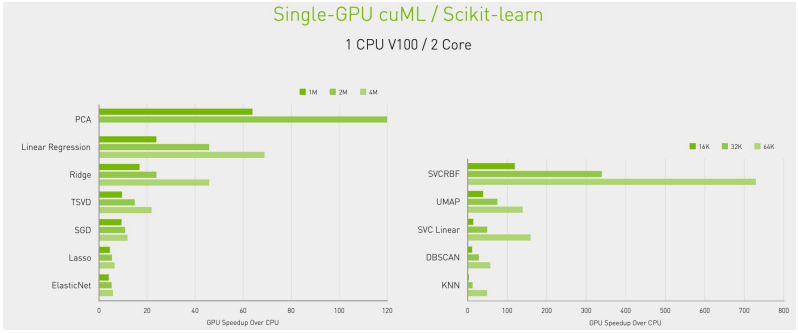
Machine Learning

- Enabled by huge leap in parallelization and innovation in ML infrastructure and tools

Tensor Processing Unit (TPU): AI accelerator application-specific integrated circuit (ASIC), also as Cloud service (Google Cloud TPU)

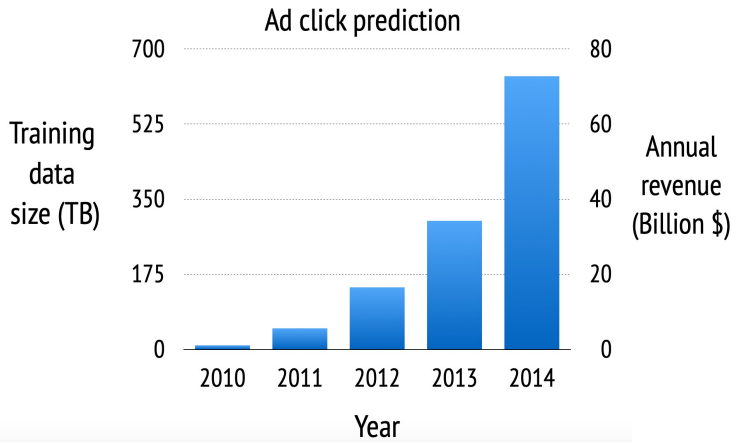


Every major deep learning framework (e.g., TensorFlow, PyTorch) is already GPU-accelerated



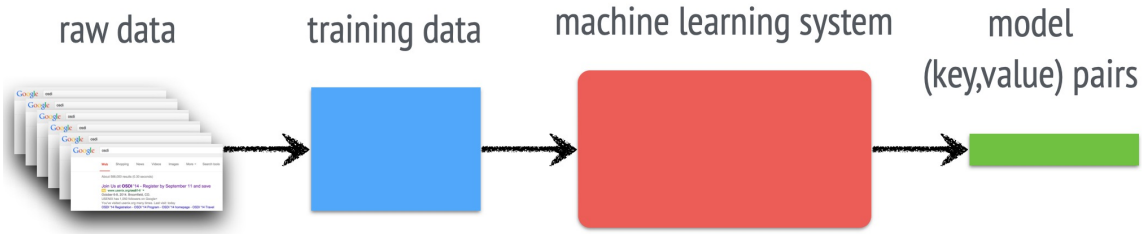
Is there a case for distributed ML?

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



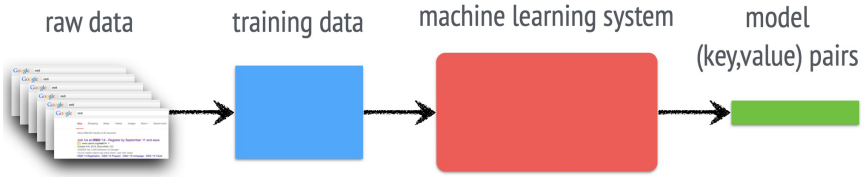
Li et al, Scaling Distributed Machine Learning with the Parameter Server, OSDI'14

What do ML algorithms look like?



- Common feature when computing ML algorithms?
  - ML algorithms are **iterative** in nature
- Three key challenges:
  - Lots of data
  - Lots of parameters
  - Lots of iterations

Scale of industry ML problems



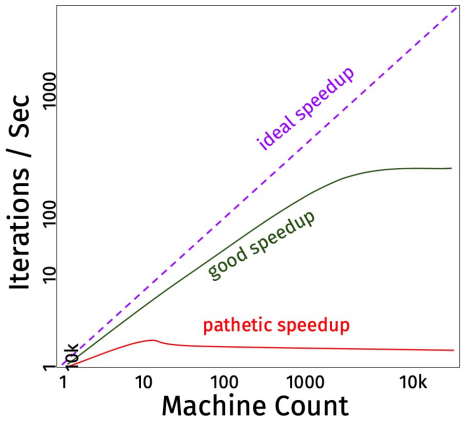
- Scale of ML industry problems
  - 100 billion examples
  - 10 billion features
  - 1T - 1P training data
  - 100 - 1000 machines
- It's a problem of scale and scale changes everything!

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.

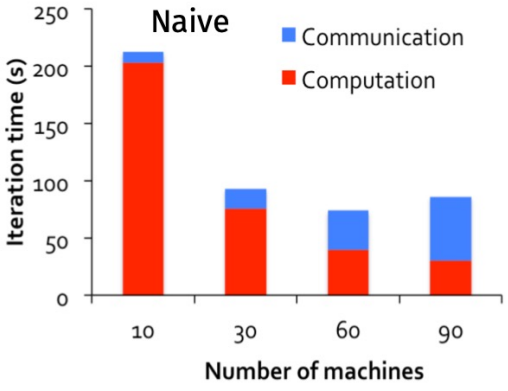
Scaling out distributed ML

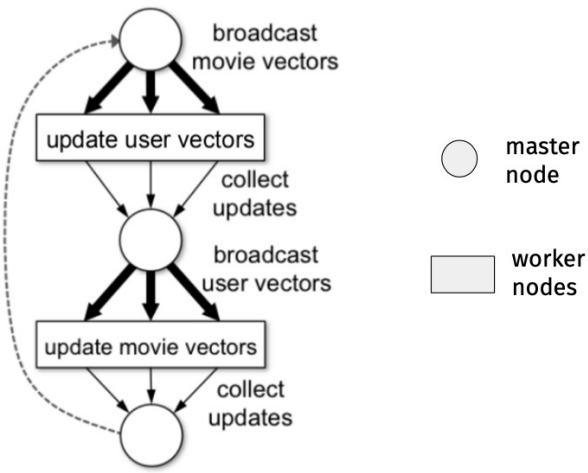
- 10-100s nodes enough for data/model
- Scale out for throughput
- Goal: more iterations/sec
  - Best case: 100x speedup from 1000 machines
  - Worst case: 50% slowdown from 1000 machines
- Can you think of reasons for performance degradation?



Challenge of communication overhead

- Communication overhead scales badly with number of machines
  - E.g., for Netflix-like recommender systems





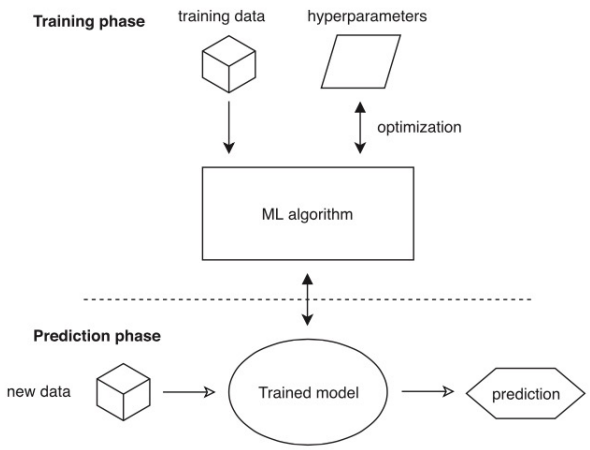
Requirements of distributed ML

- Scale to industry problems
- Efficient communication
- Fault tolerance
- Easy to use

Parallelization methods for distributed ML

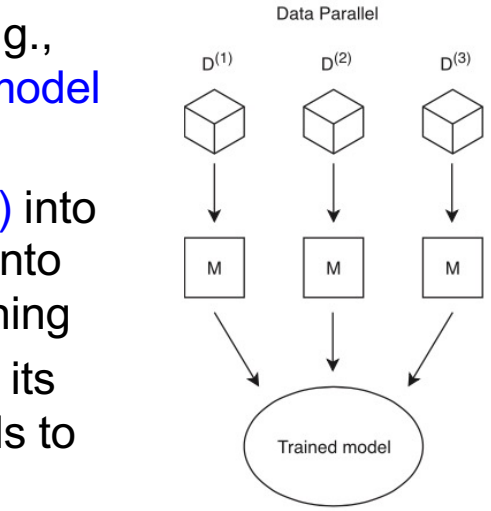
1. Data parallelism
2. Model parallelism
3. Pipeline parallelism

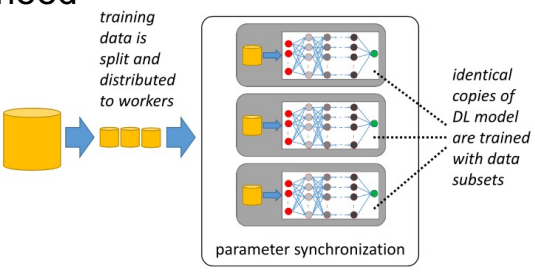
- Plus hybrid forms of parallelism



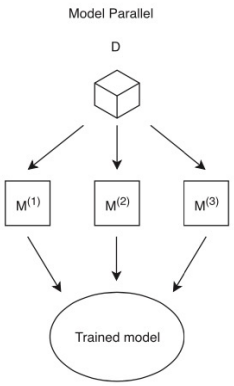
Method 1: Data parallelism

- 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 and fed into model replicas of workers for training
- Each worker performs training on its chunk of training data, which leads to updates of model parameters
  - Model parameters between workers need to be synchronized: how?



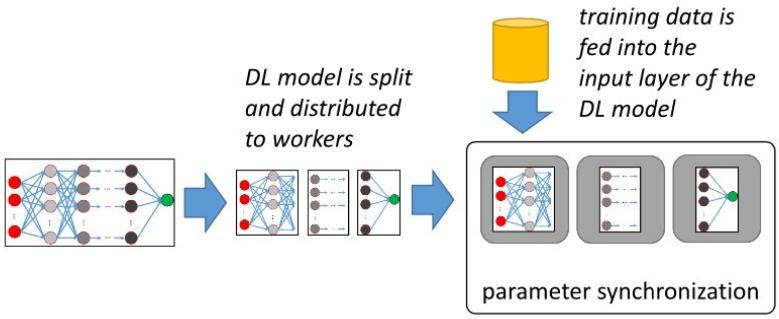


Method 2: Model parallelism



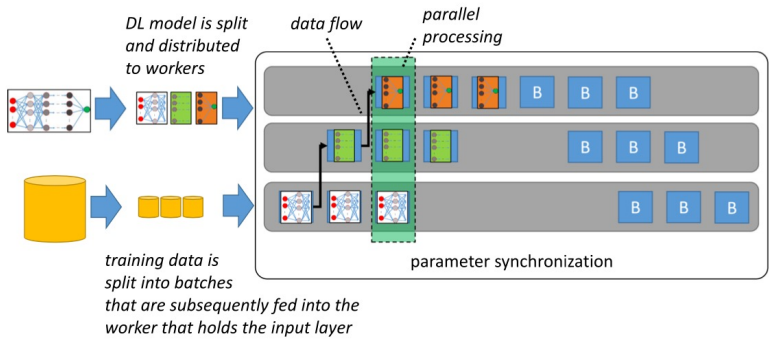
- Model is split ($M^{(1)}$, $M^{(2)}$, …) and each worker loads a different part of model for training
- Workers load an identical copy of data (D)

Method 2: Model parallelism



- Use case: DL
  - 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 microbatches
  - Every worker computes output signals for a set of microbatches, propagating them to subsequent workers
  - In the backpropagation pass, workers compute gradients for their model partition for multiple microbatches, 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, because model parameters generally cannot be split up
  - ✓ Reduced memory footprint
  - ✗ Heavy communication needed between workers

Optimizations for data parallelism

- Challenges of parameter synchronization in data-parallel ML systems

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

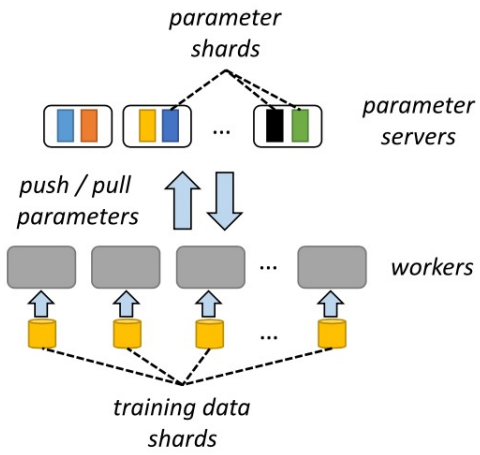
- How to minimize communication overhead for parameter synchronization

Communication topology

1. *How to synchronize parameters*
   - Centralized architecture or decentralized manner?

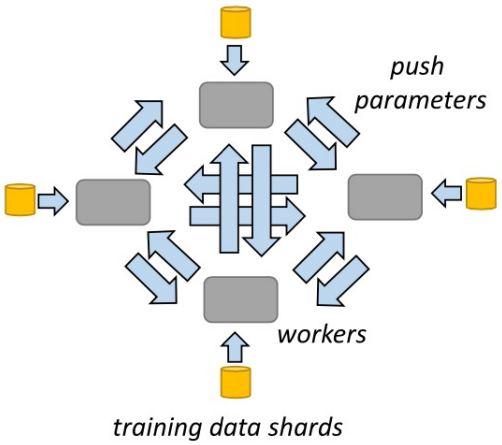
- Centralized: parameter server
- Decentralized: ring all-reduce

Parameter server architecture



- Workers periodically report their computed parameters or parameter updates to a (set of) parameter server(s) (PSs)
- The most prominent architecture of data parallel ML systems

Ring all-reduce architecture




```
grad = gradient(net, w)

for epoch, data in enumerate(dataset):
  g = net.run(grad, in=data)
  gsum = comm.allreduce(g, op=sum)

  w -= lr * gsum / num_workers
```

- All-reduce: compute some reduction (e.g., sum) of data on multiple machines and materialize the result on all those machines
- Communication cost of fully connected network is $O(n^2)$ with n workers: bottleneck
- Common alternative: employ a ring topology

Architecture: Pros and cons

- Decentralized architecture pros
  - No need of implementing and tuning a parameter server, which also eases the deployment
  - Fault tolerance can be achieved more easily, because there is no single point of failure (parameter server)
    - When a node in the decentralized architecture fails, other nodes can easily take over its workload and training proceeds without interruptions
    - Heavy-weight checkpointing of parameter server state is not necessary
- Decentralized architecture cons
  - Communication increases quadratically with number of workers
  - Changing the topology or partitioning the gradients induce new complexities and trade-offs

When to synchronize

2. *When to synchronize parameters*
   - Should the workers be forced to synchronize after each batch, or do we allow them more freedom to work with potentially stale parameters?

- Synchronous
- Bounded asynchronous
- Asynchronous

When to synchronize

- Synchronous
  - After each iteration (processing of a batch), workers synchronize their parameter updates
  - Requires barriers (recall BSP)
  - ✓ Reasoning about model convergence is easier
  - ✗ Straggler problem, where the slowest worker slows down all others

When to synchronize

- Bounded asynchronous
  - ✓ Workers may train on stale parameters, but staleness is bounded
  - ✓ 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 to some extent and increases throughput

When to synchronize

- Asynchronous
  - Workers update their model completely independently from each other
  - ✓ 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

What is federated ML?

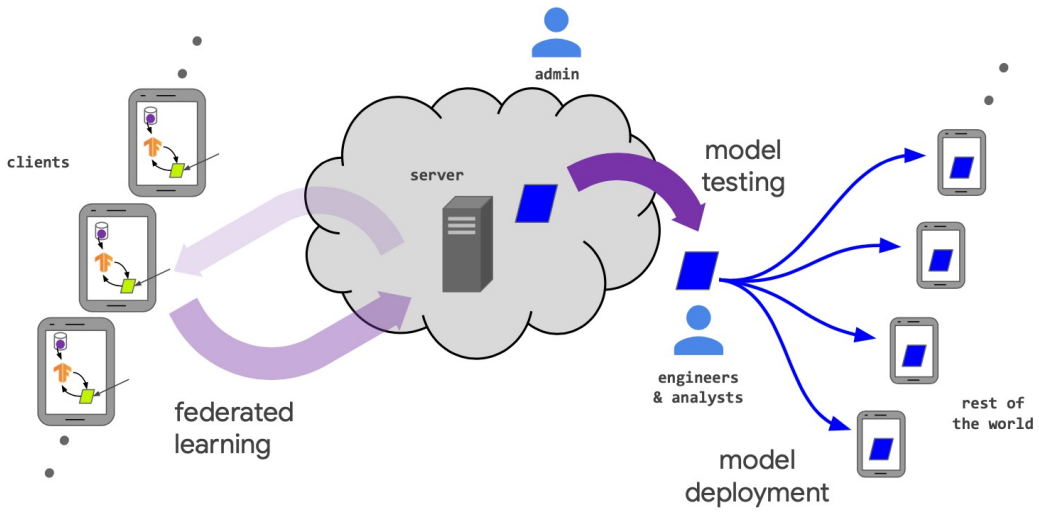
- Scenario: training settings are distributed, collaborative, and multiple parties/clients
- Goal: train collaboratively a ML model on multiple client devices located at the network edge, where data is generated locally and remains decentralized
  - No centralized training data: each client stores its own data and cannot read data of other clients
  - Data is not independently or identically distributed

What is federated ML?

- A broad definition: Federated learning (FL) is a ML setting where multiple entities (clients) collaborate in solving a ML problem, under the coordination of a central server or service provider. Each client's raw data is stored locally and not exchanged or transferred; instead, focused updates intended for immediate aggregation are used to achieve the learning objective

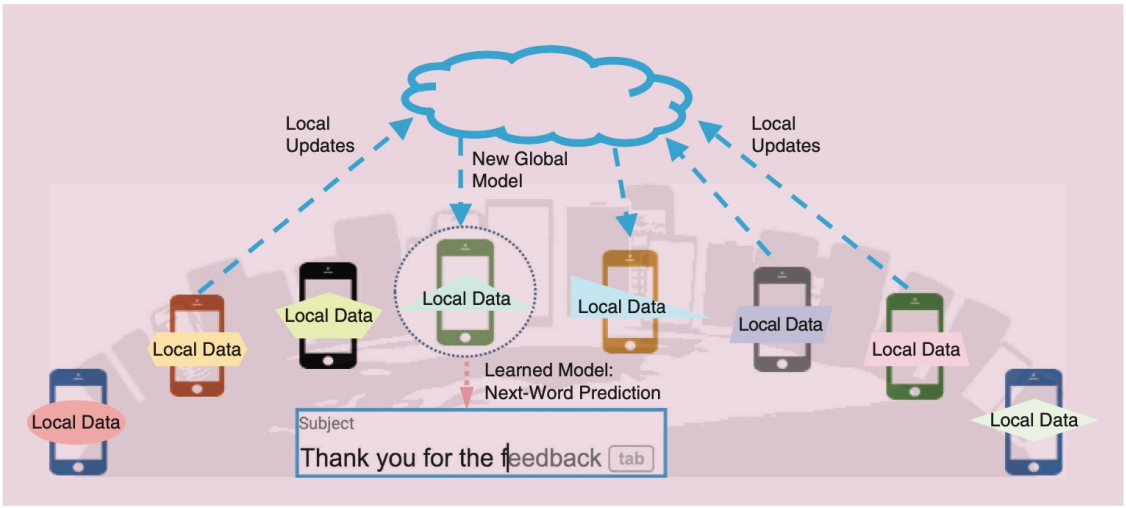
Federated learning system

- A central orchestration server organizes the training, but never sees raw data



Example application of FL

- Next-word prediction on mobile phones, while preserving privacy of data and reducing strain on network



Example application of FL

- Goal: train a predictor in a distributed fashion, rather than sending the raw data to a central server
- How it works
  - Remote devices communicate with a central server periodically to learn a global model
  - At each communication round, a subset of selected phones performs local training on their nonidentically distributed user data, and sends these local updates to the server
  - After incorporating updates, the server sends back the new global model to another subset of devices
  - Iterative training process continues across the network until convergence is reached or some stopping criterion is met

FL main challenges

- Communication overhead
- System heterogeneity
- Statistical heterogeneity
- Privacy concerns

References

- Mayer et al., Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques, and Tools, ACM Computing Surveys, 2020
- Verbraeken et al., A Survey on Distributed Machine Learning, ACM Computing Surveys, 2020

- McMahan and Ramage, Federated Learning: Collaborative Machine Learning without Centralized Training Data, Google AI blog, 2017
- Kairouz et al., Advances and Open Problems in Federated Learning, 2021