TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

**Macroarea di Ingegneria**
**Dipartimento di Ingegneria Civile e Ingegneria Informatica**

# Introduction to Distributed Machine Leaning and Federated Machine Leaning

## Corso di Sistemi e Architetture per Big Data

A.A. 2024/25

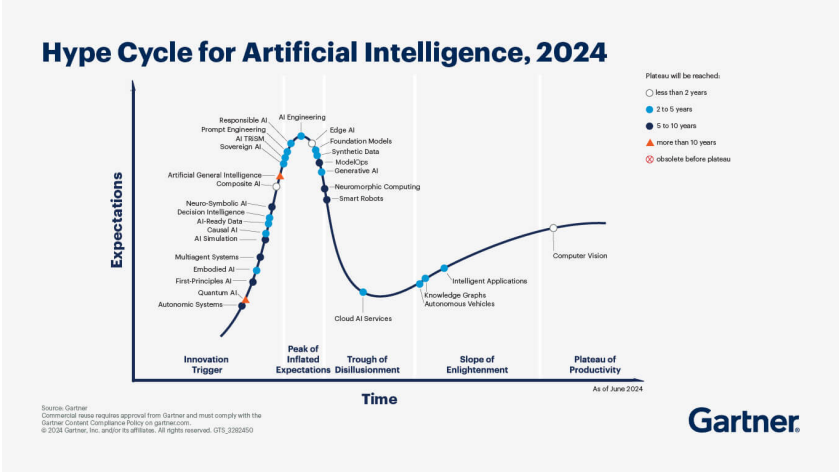
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

## Artificial Intelligence and Machine Learning

- AI and ML hype

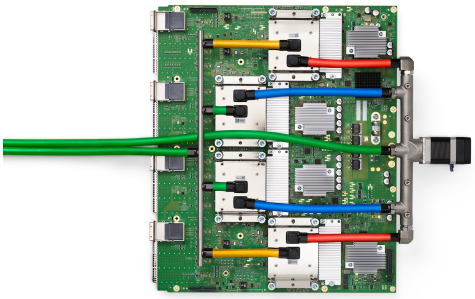




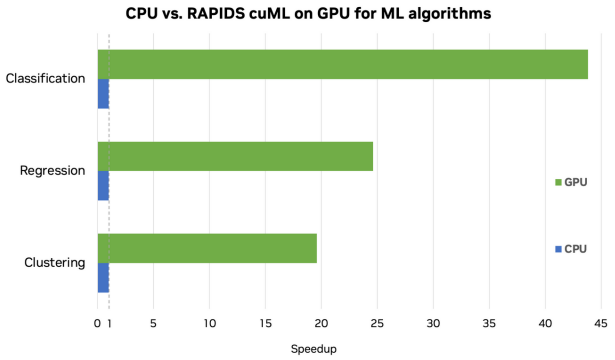
# Machine Learning

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

- **Tensor Processing Unit (TPU)**: AI accelerator application-specific integrated circuit (ASIC) specialized in calculations with *tensors* (multi-dimensional matrices)
- Also available as Cloud service: Google Cloud TPU



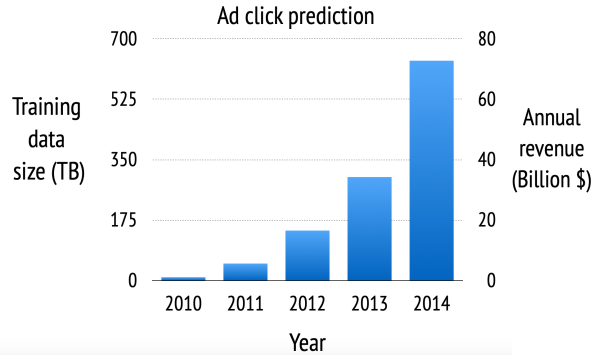
- Widely-used deep learning frameworks (e.g., TensorFlow, PyTorch, Scikit-learn) are GPU-accelerated
- Not only training, but also inference






# Is there a case for distributed ML?

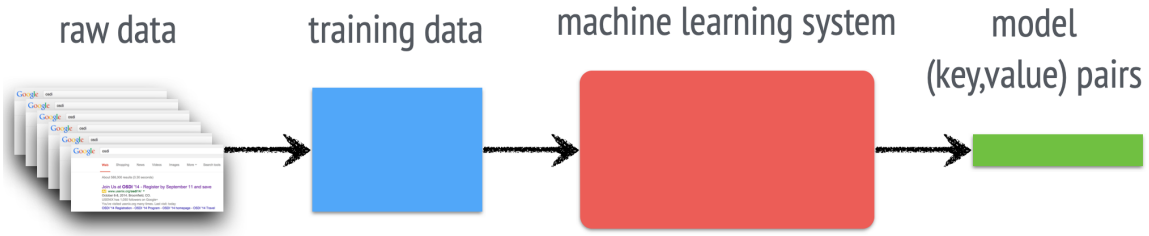
- ML systems:
  - Drive significant revenue
  - Benefit from humongous amount of data
  - Outscale even powerful machines (GPUs, TPUs)
- Which systems? Example: ad click prediction




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

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

# 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

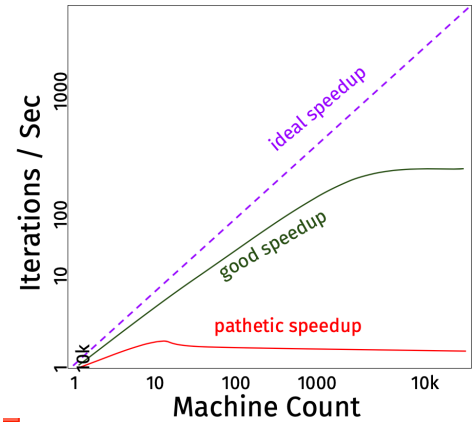
- A taste of scale of ML industry problems
  - 100 billion examples
  - 10 billion features
  - 10TB - 10P training data
  - 100 - 1000 machines
- It's a problem of scale and scale changes everything!

**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

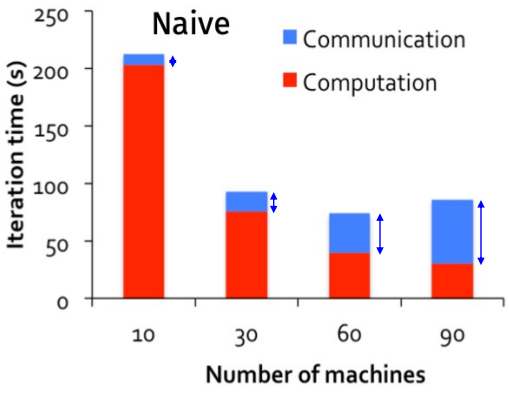
- 10-100s nodes enough for data/model
- Scale out for throughput
- Goal: more iterations/sec
  - Best case: 100x speedup from 1000 machines
  - Worst case: 50% slowdown from 1000 machines
- Can you think of reasons for performance degradation?

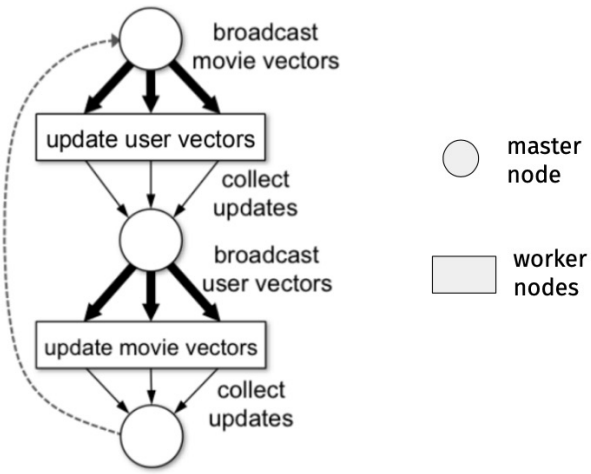





# Challenge of communication overhead

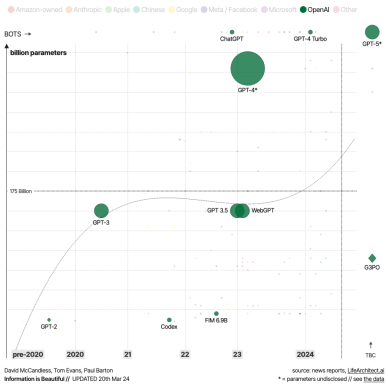
- Communication overhead scales badly with number of machines
  - E.g., Netflix-like recommender system based on matrix factorization



# Requirements for distributed ML

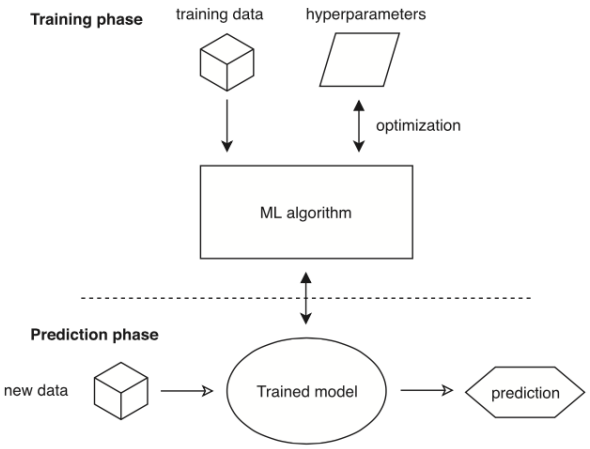
- Scale to industry-size problems
  - Immense model size of large foundation models (LLMs), whose performance improves with model size and data volume
    - GPT-3 had 175 billion parameters (variables and inputs within model), GPT-4 is 10x larger
- Efficient communication
- Fault tolerance
- Easy to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





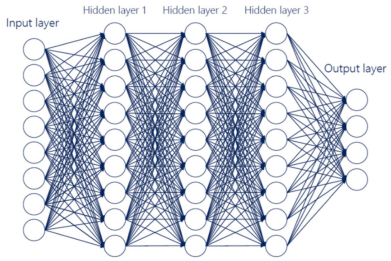

# Distributed training and inference

- What can we perform in a distributed manner?
  - Training: process of using a ML algorithm to build a model
  - Inference: process of using a trained ML algorithm to make a prediction



# Parallelization methods for distributed training

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

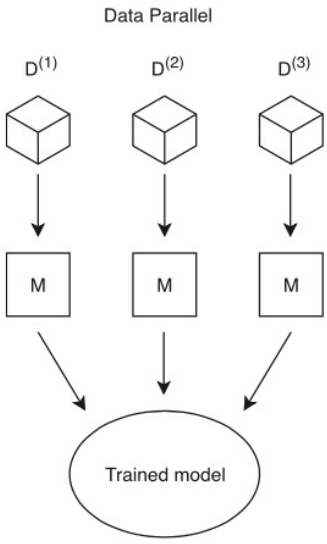


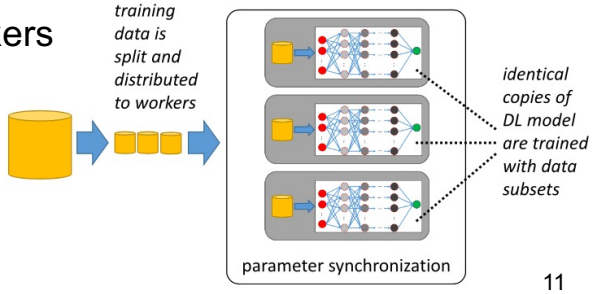

- Methods for distributed training
  1. Data parallelism: the usual SPMD approach
  2. Model parallelism
  3. Pipeline parallelism
  - Plus hybrid forms of parallelism that we do not explore



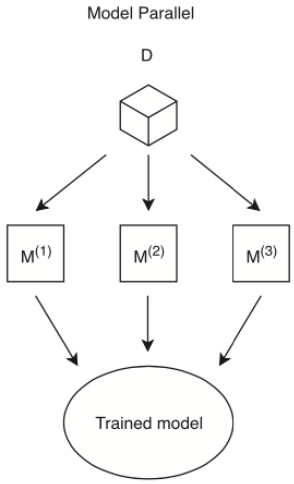
# Method 1: Data parallelism

- 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?



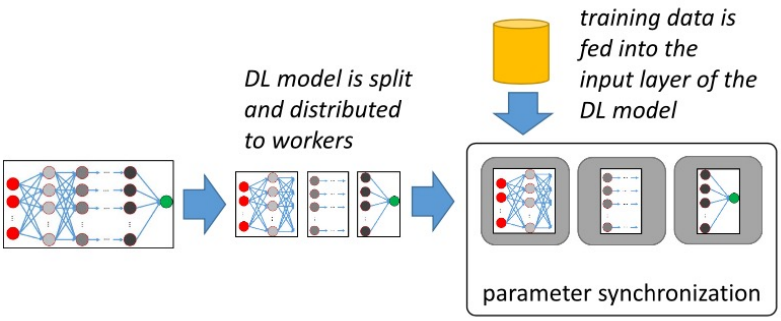
# Method 2: Model parallelism




- 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)

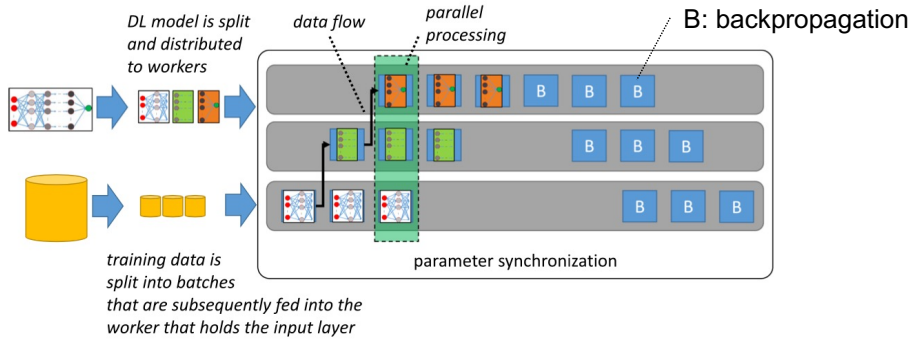


# Method 2: Model parallelism




- 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

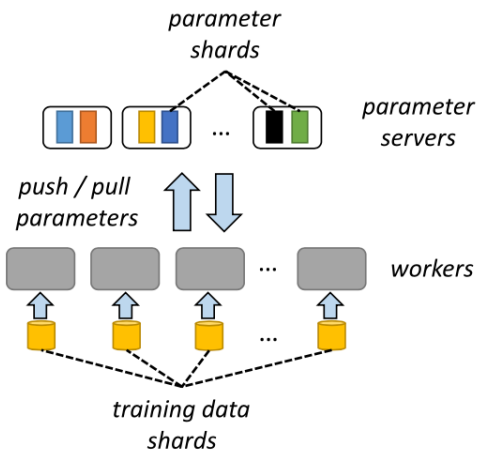
# How to synchronize parameters: architecture

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

- We consider the two main approaches
  - Centralized: parameter server
  - Decentralized: all-reduce

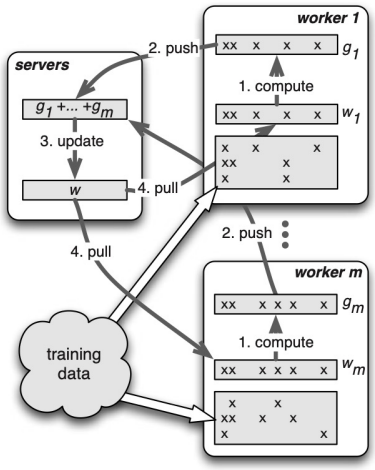


# Centralized: Parameter server




- The most prominent architecture of data parallel ML systems
- Workers periodically push their computed parameters (or parameter updates) to a parameter server (PS), which keeps the shared model, and pull the updated model parameters from PS

# 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 parameters w

Workers send gradients to PS

Workers pull weights from PS

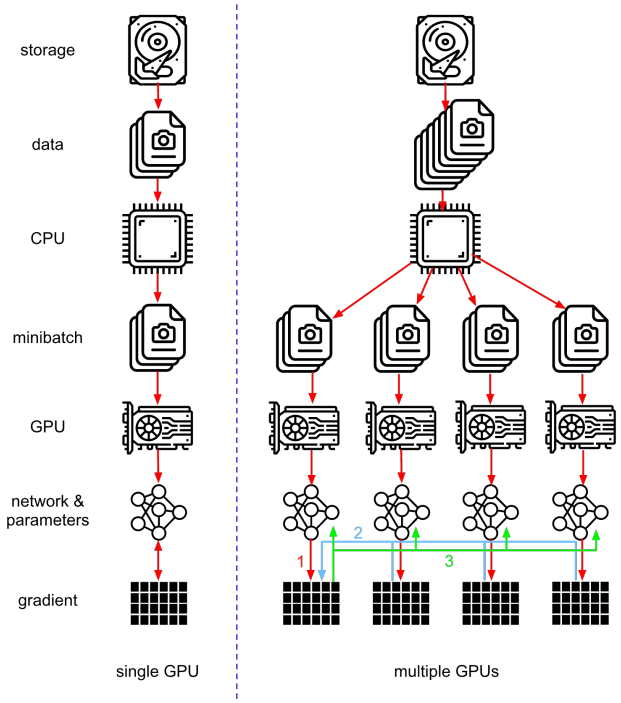
PS updates the model weights

https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf



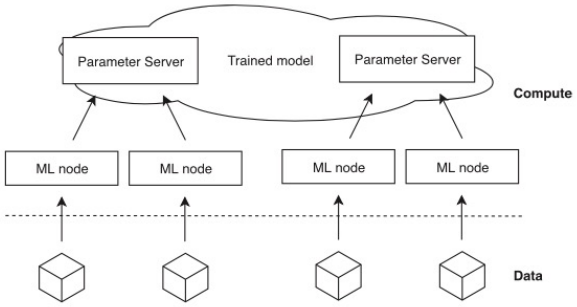
# Parameter server: multiple GPUs

1. Compute loss and gradient on each GPU
2. All gradients are aggregated on one GPU acting as parameter server
3. Parameter update happens and parameters are re-distributed to all GPUs



# Centralized: Multiple parameter servers

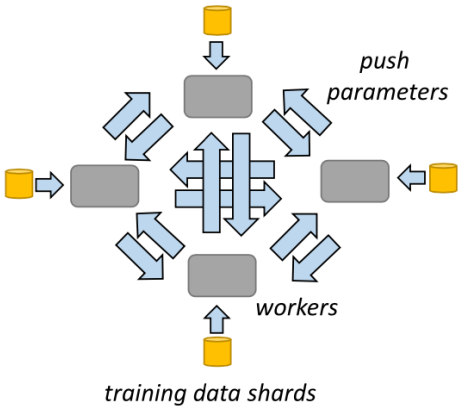
- To mitigate performance bottleneck and SPoF, there can be multiple parameter servers which manage the model's parameters




  - Parameters are partitioned among multiple PSs and each PS is only responsible for maintaining the parameters in its partition
  - When a worker wants to send 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 parameter server



# Decentralized: All-reduce




```
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: collective communication which computes some reduction (e.g., sum) of data (e.g., gradients) on multiple workers and make the result (e.g., weights) available on all the workers
- All-reduce should be implemented efficiently because naïve solution (all-to-all) is too costly: communication cost of fully connected network is $O(n^2)$ with $n$ workers
- How? Use different topologies, such as ring or tree

# Decentralized: Pros and cons

- Decentralized architecture pros
  - ✓ No need of implementing parameter server(s), which also eases deployment
  - ✓ Easier to achieve fault tolerance: no SPoF (if single PS)
    - 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 cost increases (at most quadratically) with number of workers
  - ✗ Changing the communication topology or partitioning the gradients induces new complexities and trade-offs



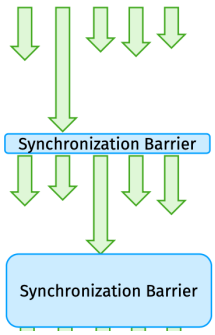
# When to synchronize

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?

- Synchronous
- Bounded asynchronous
- Asynchronous

# 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
  - ✓ Reasoning about model convergence is easier
  - ✗ *Straggler* problem, where the slowest worker slows down all others



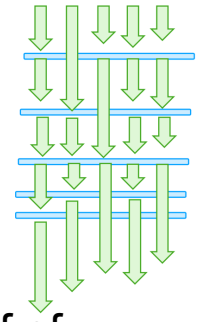



# When to synchronize

- How to address straggler problem?
- Let's relax the synchronization requirement
- 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
  - Asynchronous manner is suitable to geo-distributed training servers
- But be careful: this is the usual trade-off between performance and model guarantees

# When to synchronize

- Bounded asynchronous
  - Workers may train on stale parameters, but staleness is bounded
    - ML algorithms are robust, converge even with some stale state

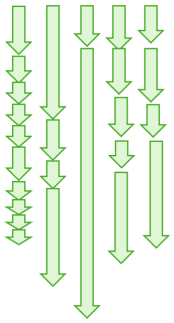


  - ✓ Allows for mathematical analysis and proof of model convergence properties
  - ✓ Bound allows workers for more freedom in making training progress independently from each other, which mitigates the straggler problem and increases throughput



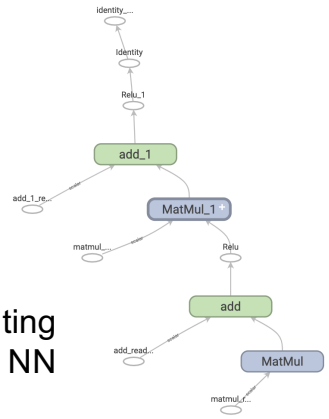
# When to synchronize: async

- Asynchronous (async)
  - No barriers at all: 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



# Example: TensorFlow

- TensorFlow: Python-friendly open-source software library for ML and AI https://www.tensorflow.org/
  - Can be used across a range of ML and AI tasks, but focus on training and inference of DNNs
  - Initially developed by Google Brain team for internal Google use in research and production, then released in 2015
  - A TensorFlow computation is described by a DAG (again!) with operations and units of data that flow between operations
  - Can run also on multiple devices
    - CPUs and accelerators (TPUs and GPUs)




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 (based on parameter server) and decentralized (based on all-reduce)
  - Supports both synchronous and asynchronous parameter update

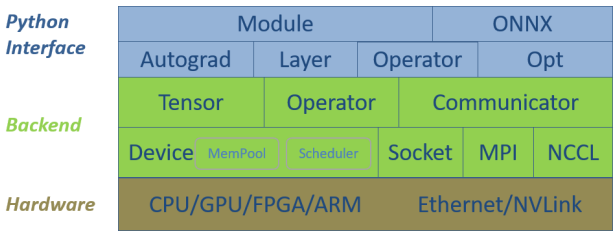
# Example: Pytorch

- 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

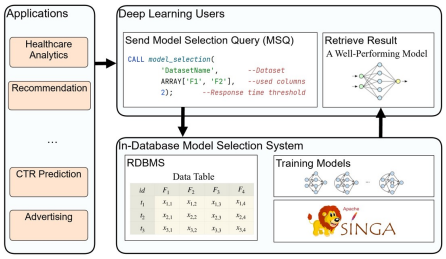


# Example: Apache Singa

- 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)




- DL models can be queried as stored procedures of RDBMS



# Issues to address in distributed ML training

- Efficient communication among computing nodes
- Heterogeneity of computing nodes
- Resource and energy-hungry
  - E.g., pre-training of LLaMa-2-70B takes $1.7 \times 10^6$ of GPU hours and consumes $2.5 \times 10^{12}$ J of energy
- Fault tolerance
- Privacy protection
  - We briefly consider federated learning



# Distributed ML inference

- Why also distributed inference?
  - Large models (e.g., GPT) are too big for a single device
  - Real-time response needs in low-latency applications
  - Hardware limitations (memory, computation, energy)
- How?
  - Partitioning model inference across multiple devices or nodes
  - Difference between training and inference distribution
  - Main use case: model splitting in the Edge-Cloud continuum

# Distributed inference: architectures

- Synchronous vs. asynchronous methods
  - Synchronous: all nodes/devices wait for each other to complete a step
  - Asynchronous: nodes operate independently without waiting

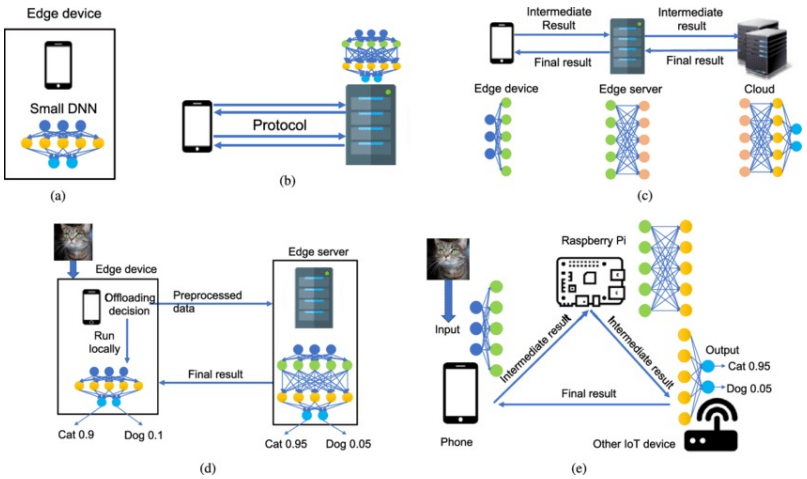
    Scales better in large deployments or when input data arrives irregularly (e.g., mobile apps, real-time dashboards)

    Needs more careful system design to handle concurrency and consistency



# 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
- Caching and reuse of results
- Early exits in model execution
- Adaptive computation (e.g., dynamic routing)



# Distributed inference: frameworks

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

# What is federated learning?

- Scenario: training setting is distributed, collaborative, with multiple devices/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

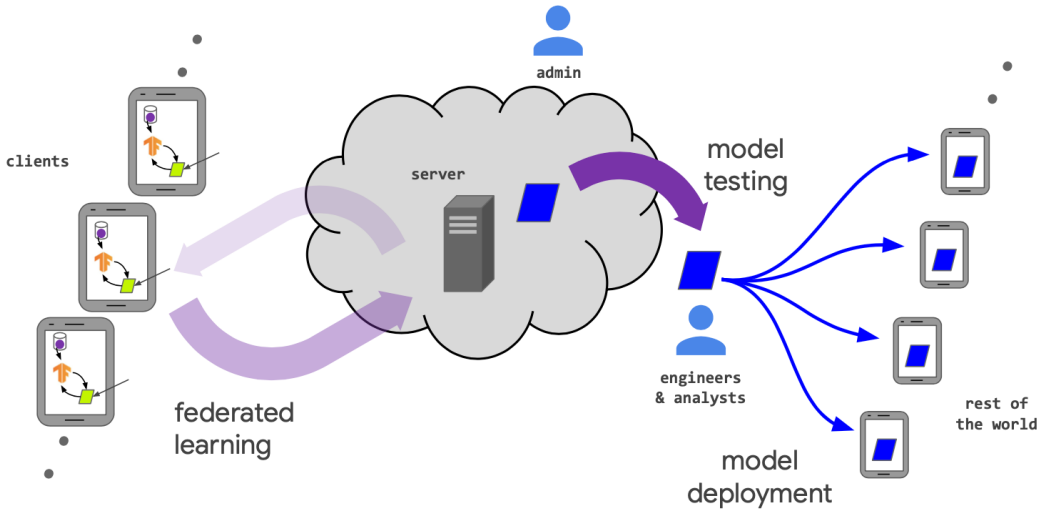



# What is federated learning?

- Federated leaning (FL): distributed ML setting where multiple clients collaborate in training a ML model, 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
- Constraint: given an evaluation metric (e.g, accuracy), performance of model learned by FL should be better than that of model learned by local training with the same model architecture

# FL system

- Major components: parties (e.g., clients), manager (e.g., server), and communication-computation framework to train the machine learning model
  - A central orchestration server organizes the training, but never sees raw data






# FL training process

- A server orchestrates the training process, by repeating the following steps until training is stopped:

1. **Client selection**: server samples from a set of clients meeting eligibility requirements (e.g., in order to avoid impacting user device)
2. **Broadcast**: selected clients download the current model weights and a training program (e.g., a TensorFlow graph) from server
3. **Client computation**: each selected client locally computes a model update by executing the training (e.g., running stochastic gradient descent, SGD) on local data and sends the model weight updates to server

# FL training process

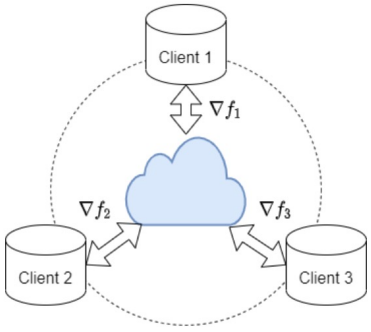
4. **Aggregation**: server combines the model updates received by the selected clients
   - For efficiency, stragglers might be dropped
   - This stage is also the integration point for many other techniques, including: secure aggregation for added privacy, lossy compression of aggregates for communication efficiency, and noise addition and update clipping for differential privacy
5. **Model update**: server locally updates the shared model based on the aggregated update computed from clients that participated in the current round

- These steps are repeated until model converges or maximum number of iterations is reached



# Approaches for FL training

- Federated averaging (*FedAvg*): the first and most widely used approach
  - It runs a number of steps of SGD in parallel on a small sampled subset of devices and then averages the resulting model updates via a central server once in a while

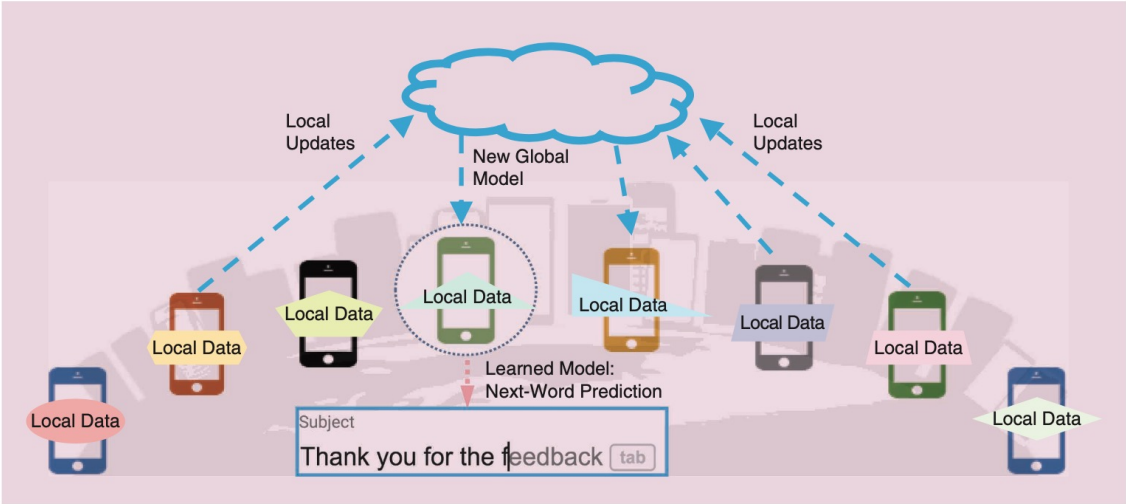



$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta \cdot \sum_{k=1}^{K} \frac{n_k}{n} \nabla f_k$$

- Decentralized FL approaches also exist

# Example: FL application

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






# Example: FL application

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

McMahan et al., Communication-Efficient Learning of Deep Networks from Decentralized Data, ArXiV, 2016 https://arxiv.org/abs/1602.05629

# FL main challenges

- Communication overhead
  - Can be addressed with decentralized architecture (P2P, graph, blockchain)
- System heterogeneity
  - Constrained resources on edge devices, including limited network bandwidth and latency
- Statistical heterogeneity
- Privacy concerns
  - Although local data are not exposed in FL, exchanged model parameters may still leak sensitive information about data (e.g., model inversion attack and membership inference attack)
  - How to address? Using cryptographic methods (e.g., homomorphic encryption), secure multi-party computation, differential privacy



# FL frameworks

- Some open-source frameworks for federated learning
  - Flower https://flower.ai/
  - PySyft https://github.com/OpenMined/PySyft
  - TensorFlow Federated https://www.tensorflow.org/federated
  - FedN https://github.com/scaleoutsystems/fedn
  - FedML https://fedml.ai/

# References


- Mayer et al., Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques, and Tools, ACM Computing Surveys, 2020 https://arxiv.org/pdf/1903.11314
- Verbraeken et al., A Survey on Distributed Machine Learning, ACM Computing Surveys, 2020 https://arxiv.org/pdf/1912.09789

- McMahan and Ramage, Federated Learning: Collaborative Machine Learning without Centralized Training Data, Google AI blog, 2017 https://research.google/blog/federated-learning-collaborative-machine-learning-without-centralized-training-data/
- McMahan et al., Communication-Efficient Learning of Deep Networks from Decentralized Data, ArXiV, 2016 (revised 2023) https://arxiv.org/abs/1602.05629
- Kairouz et al., Advances and Open Problems in Federated Learning, Foundations and Trends in Machine Learning, 2021 https://www.nowpublishers.com/article/Details/MAL-083