Distributed Learning

CS6787 Lecture 9 — Fall 2020

First, a recap…
Asynchronous Parallelism

Limits on parallel performance

- Synchronization
  - Have to synchronize to keep the workers aware of each other's updates to the model — otherwise can introduce errors
- **Synchronization can be very expensive**
  - Have to stop all the workers and wait for the slowest one
  - Have to wait for several round-trip times through a high-latency channel
- **Is there something we can do about this?**

Idea: Just Don't Synchronize

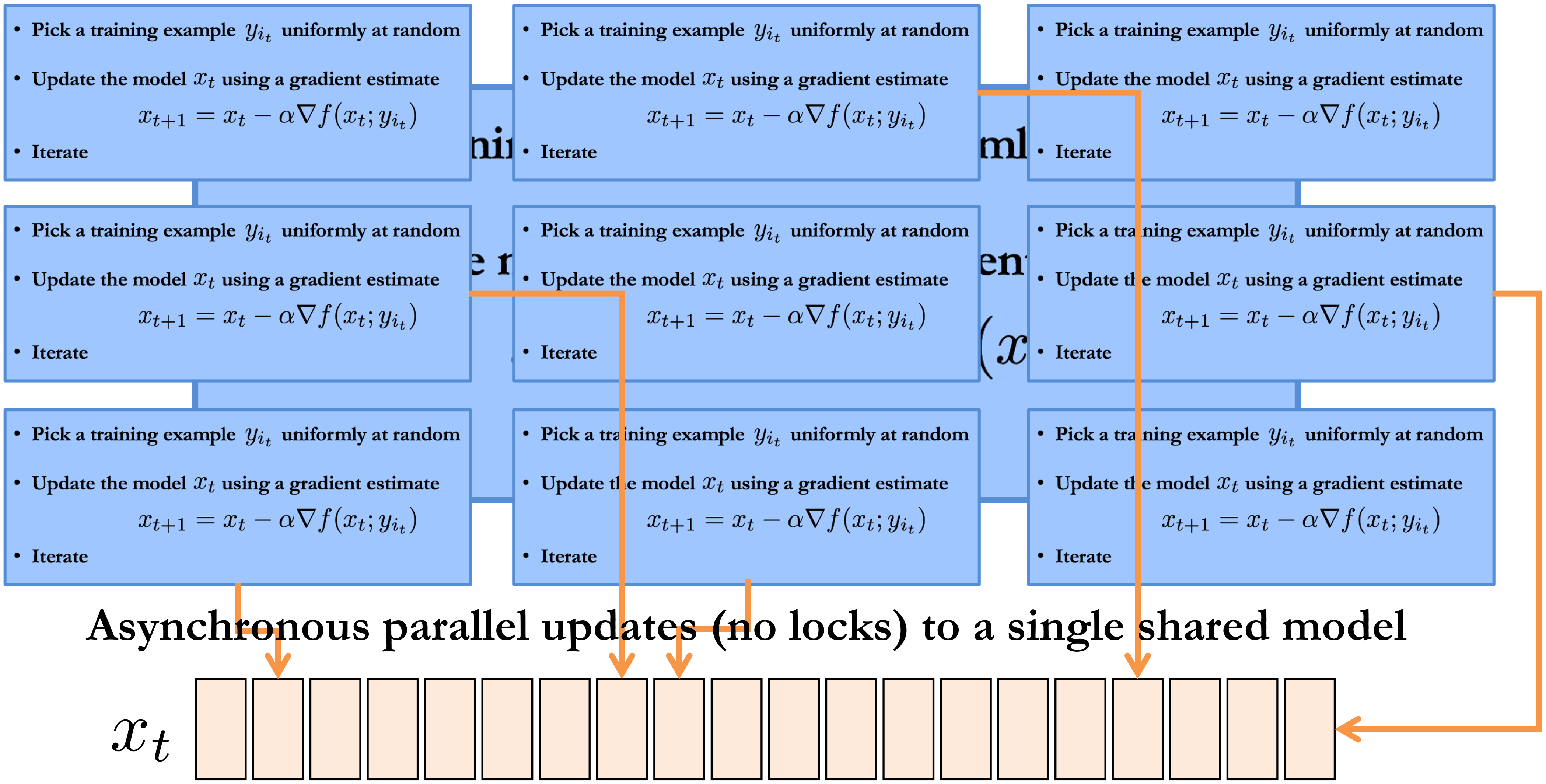
- Not synchronizing adds **errors due to race conditions**

- But our methods were already noisy — **maybe these errors are fine**

- If we don't synchronize, get **almost perfect parallel speedup**

Fast Parallel SGD: HOGWILD!

Multiple parallel workers



- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

- Iterate

- Pick a training example y_{i_t} uniformly at random
- Update the model x_t using a gradient estimate

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{i_t})$$

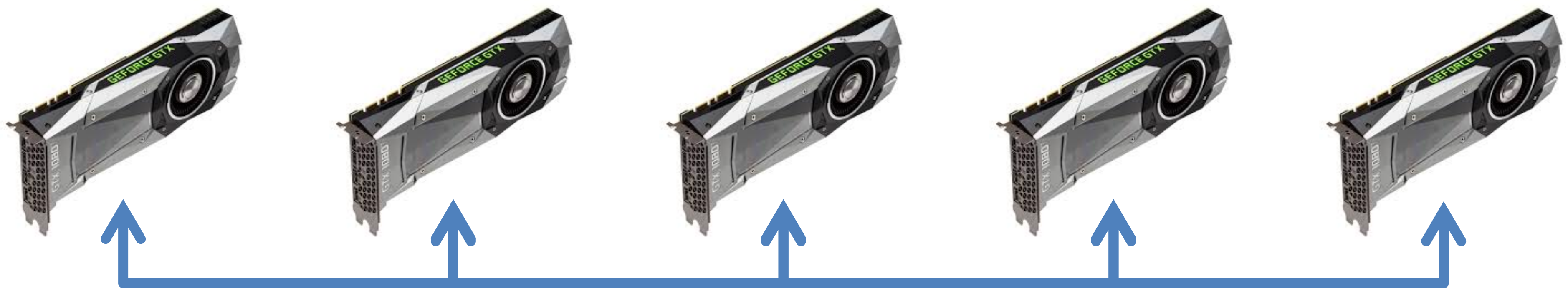
- Iterate

Asynchronous parallel updates (no locks) to a single shared model

DISCUSSION

Distributed Machine Learning

Main idea: **use multiple machines to do learning**.



Why distribute?

- Train **more quickly**
- Train **models too large** to fit on one machine
- Train when the **data are inherently distributed**

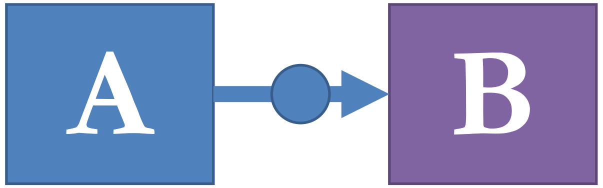
Distributed computing basics

- Distributed parallel computing **involves two or more machines collaborating on a single task by communicating over a network.**
  - Distributed computing requires explicit (i.e. written in software) communication among the workers.
  - **No shared memory abstraction!** (Unlike parallelism on 1 machine)

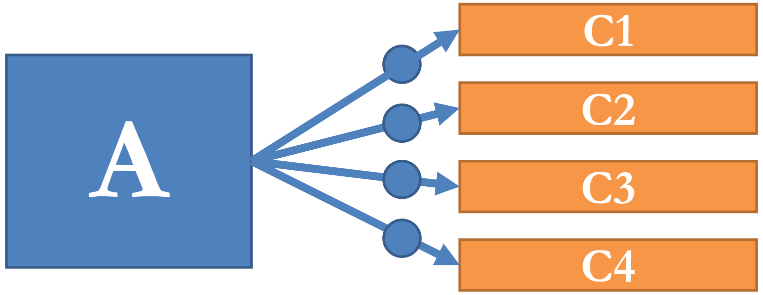
- There are a **few basic patterns of communication** that are used by distributed programs.

Basic patterns of distributed communication

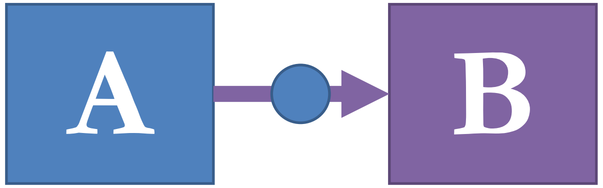
Push: Machine A sends some data to machine B.



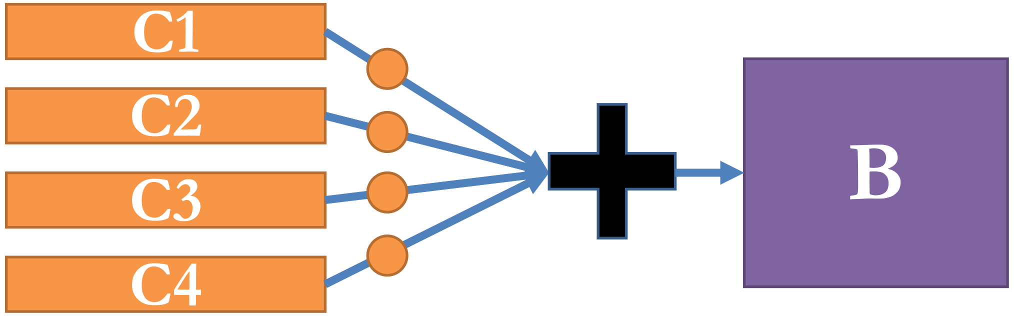
Broadcast: Machine A sends data to many machines.



Pull: Machine A requests some data from machine B.

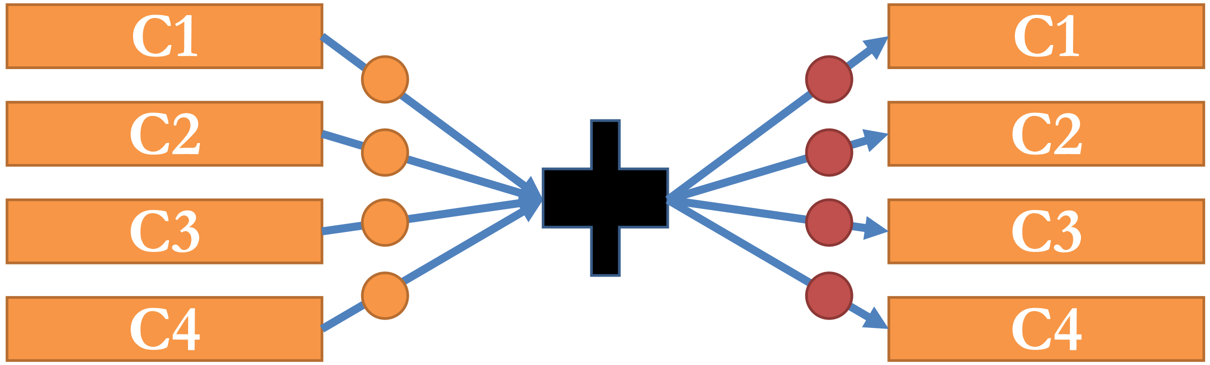


Reduce: Compute some reduction of data on multiple machines and materialize result on B.

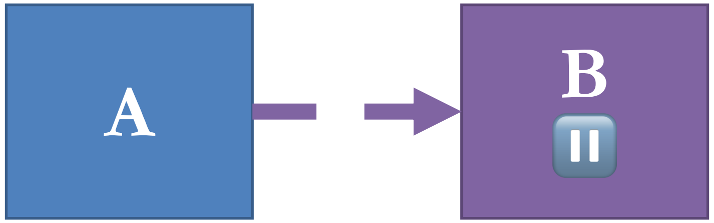


Basic patterns of distributed communication (cont'd)

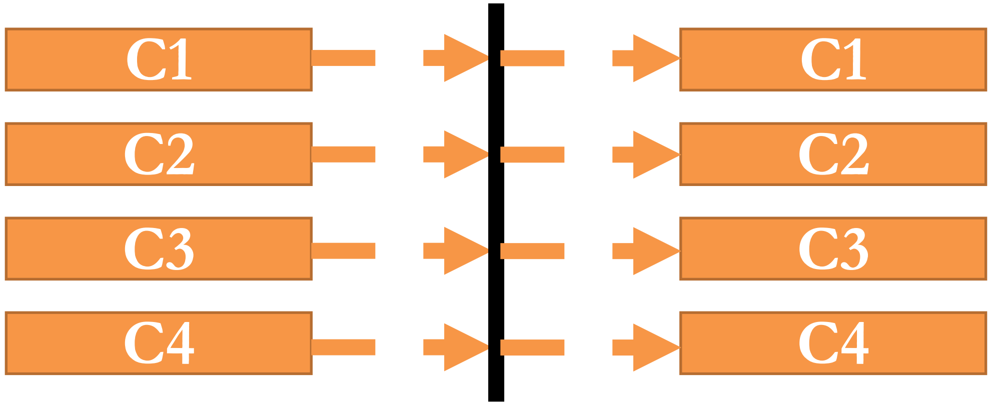
All-reduce: Compute some reduction of data on multiple machines and materialize result on all those machines.



Wait: Pause until another machine says to continue.



Barrier: Wait for all workers to reach some point in their code.



All these operations can be synchronous or asynchronous.

Overlapping computation and communication

- Communicating over the network can have high latency
  - we want to hide this latency

- An important principle of distributed computing is **overlapping computation and communication**

- For the best performance, we want our workers to **still be doing useful work while communication is going on**
  - rather than having to stop and wait for the communication to finish
  - sometimes called a **stall**
  - **asynchronous communication** can help a lot here

Running SGD with All-reduce

- All-reduce gives us a simple way of running learning algorithms such as SGD in a distributed fashion.

- Simply put, the idea is to just **parallelize the minibatch.** We start with an identical copy of the parameter on each worker.

- Recall that SGD update step looks like:

$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{B} \sum_{b=1}^{B} \nabla f_{i_{b,t}}(w_t),$$

Running SGD with All-reduce (continued)

- If there are M worker machines such that $B = M \cdot B'$, then

$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{M} \sum_{m=1}^{M} \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_t).$$

- Now, we assign the computation of the sum when m = 1 to worker 1, the computation of the sum when m = 2 to worker 2, et cetera.

- After all the gradients are computed, we can perform the outer sum with an **all-reduce operation**.

Running SGD with All-reduce (continued)

- After this all-reduce, the whole sum (which is essentially the minibatch gradient) will be present on all the machines
  - so each machine can now update its copy of the parameters

- Since sum is same on all machines, the parameters will update in lockstep

- **Statistically equivalent to sequential SGD!**

Algorithm 1 Distributed SGD with All-Reduce

```
input: loss function examples $f_1, f_2, \ldots$, number of machines $M$, per-machine minibatch size $B'$
input: learning rate schedule $\alpha_t$, initial parameters $w_0$, number of iterations $T$
for $m = 1$ to $M$ run in parallel on machine $m$
    load $w_0$ from algorithm inputs
    for $t = 1$ to $T$ do
        select a minibatch $i_{m,1,t}, i_{m,2,t}, \ldots, i_{m,B',t}$ of size $B'$
        compute $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_{t-1})$
        all-reduce across all workers to compute $G_t = \sum_{m=1}^{M} g_{m,t}$
        update model $w_t \leftarrow w_{t-1} - \frac{\alpha_t}{M} \cdot G_t$
    end for
end parallel for
return $w_T$ (from any machine)
```

Same approach can be used for momentum, Adam, etc.

Benefits of distributed SGD with All-reduce

- The algorithm is easy to reason about, since it's **statistically equivalent to minibatch SGD**.
  - And we can use the same hyperparameters for the most part.

- The algorithm is easy to implement
  - since all the worker machines have the same role and it runs on top of standard distributed computing primitives.

Drawbacks of distributed SGD with all-reduce

- We're **not overlapping computation and communication**.
  - While the communication for the all-reduce is happening, the workers are idle.

- The **effective minibatch size is growing with the number of machines**
  - If we *don't* want to run with a large minibatch size for statistical reasons, this can prevent us from scaling to large numbers of machines using this method.

- Potentially requires **lots of network bandwidth** to communicate to all workers.

Where do we get the training examples from?

- There are two general options for distributed learning.

- **Training data servers**
  - Have one or more non-worker servers dedicated to storing the training examples (e.g. a distributed in-memory filesystem)
  - The worker machines load training examples from those servers.

- **Partitioned dataset**
  - Partition the training examples among the workers themselves and store them locally in memory on the workers.

The Parameter Server Model

The Basic Idea

- Recall from the early lectures in this course that a lot of our theory talked about the convergence of optimization algorithms.
  - This convergence was measured by some function over the parameters at time t (e.g. the objective function or the norm of its gradient) that is decreasing with t, which shows that the algorithm is making progress.

- For this to even make sense, though, we need to be able to talk about the value of the parameters at time t as the algorithm runs.
  - E.g. in SGD, we had

$$w_{t+1} = w_t - \alpha_t \nabla f_{i_t}(w_t)$$

Parameter Server Basics Continued

- For a program running on time t is just the value of s DRAM) at that time.

- But in a distributed setting must be done explicitly.
    - Each machine will usually l time, some of which may have been u ently than others, especially if we want to do something more complicate reduce.

- This raises the question: **when reasoning about a distributed algorithm, what we should consider to be the value of the parameters a given time**?

For SGD with all-reduce, we can answer this question easily, since the value of the parameters is the same on all workers (it's guaranteed to be the same by the all-reduce operation). We just appoint this identical shared value to be the value of the parameters at any given time.

The Parameter Server Model

- The parameter server model answers this question differently by appointing a single machine, the **parameter server**, the explicit responsibility of maintaining the current value of the parameters.
  - The most up-to-date gold-standard parameters are the ones stored in memory on the parameter server.

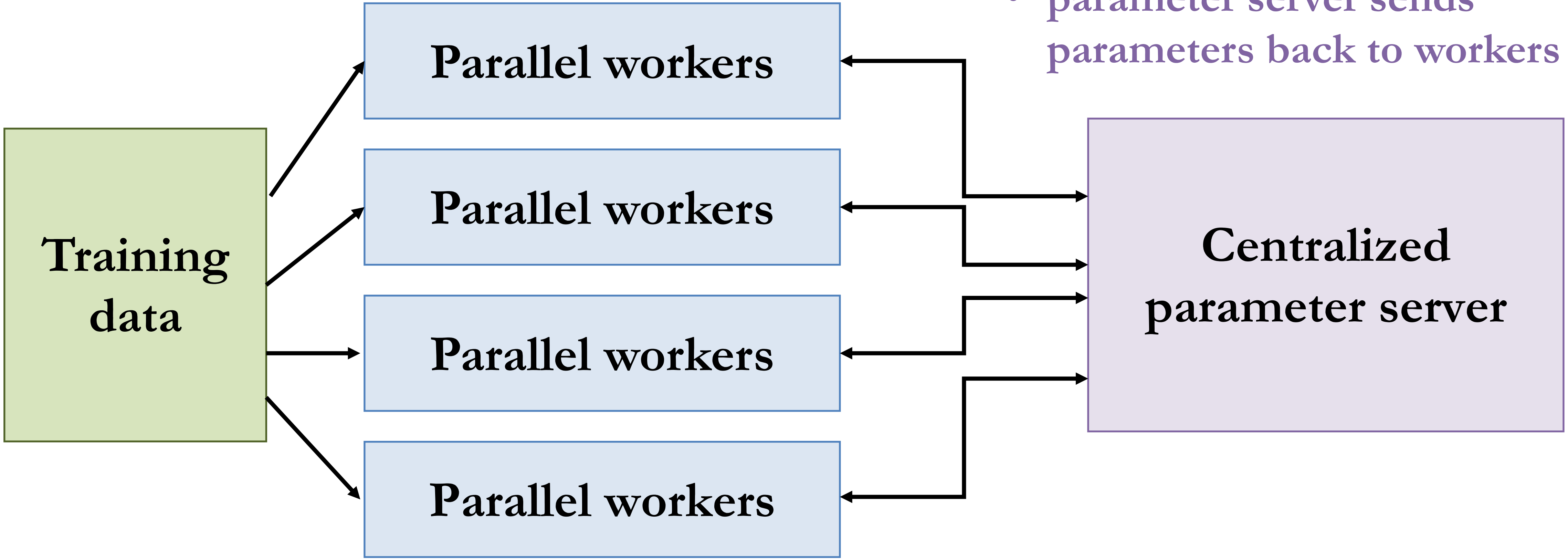
- The parameter server updates its parameters by using gradients that are computed by the other machines, known as **workers**, and pushed to the parameter server.

- Periodically, the parameter server **broadcasts its updated parameters** to all the other worker machines, so that they can use the updated parameters to compute gradients.

Parameter server model: visually

- A common model for distributed ML

- workers send gradients to parameter server
- parameter server sends parameters back to workers



Learning with the parameter server

- Two options when learning with a parameter server

- **Synchronous distributed training**
  - Similar to all-reduce, but with gradients summed on a central parameter server
  - Still **equivalent to sequential minibatch SGD**

- **Asynchronous distributed training**
  - Compute and send gradients and add them to the model as soon as possible
  - Broadcast updates whenever they are available

Parameter server summary

- The parameter server **holds the central copy of the weights**

- Each worker **computes gradients** on minibatches the data
  - Then sends those gradients back to the parameter server

- Periodically, the worker pulls an updated copy of the weights from the parameter server.

- All this can be done **asynchronously**.

Multiple parameter servers

- If the parameters are too numerous for a single parameter server to handle, we can use **multiple parameter server machines.**

- We partition the parameters among the multiple parameter servers
  - Each server is only responsible for maintaining the parameters in its partition.
  - When a worker wants to send a gradient, it will partition that gradient vector and send each chunk to the corresponding parameter server; later, it will receive the corresponding chunk of the updated model from that parameter server machine.

- This lets us **scale up to very large models!**

Other Ways To Distribute

The methods we discussed so far distributed across the minibatch (for all-reduce SGD) and across iterations of SGD (for asynchronous parameter-server SGD).

But there are other ways to distribute that are used in practice too.

“Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent.” NeurIPS 2017

Decentralized learning

- Idea: learn **without any central coordination**
  - No parameter server; each worker has its own copy of the model
- Workers update by doing the following:
  - Run an SGD update step using an example stored on that worker,
  - Average the worker’s current model with the models of some other workers, usually its neighbors in some sparse graph
    - This limits total communication
- This is sometimes called a **gossip algorithm**

e.g. “Local SGD Converges Fast and Communicates Little.” ICLR 2019

Local SGD

- Many parallel workers update their own copy of the model by running SGD steps using their own local data

- Periodically the workers all average by taking an **all-reduce**
  - Like all-reduce SGD, but the all-reduce happens less frequently than at every SGD iteration

- Can **generalize better** than large-batch SGD
  - “Don’t use large mini-batches, use local SGD.” ICLR 2020

So far: Data Parallelism

- The methods we’ve discussed are parallelizing over examples
  - Each worker is running the same computation to compute gradients, just on different examples.

- This is an instance of **data parallelism**

- But **data parallelism is not the only option…**

Model Parallelism

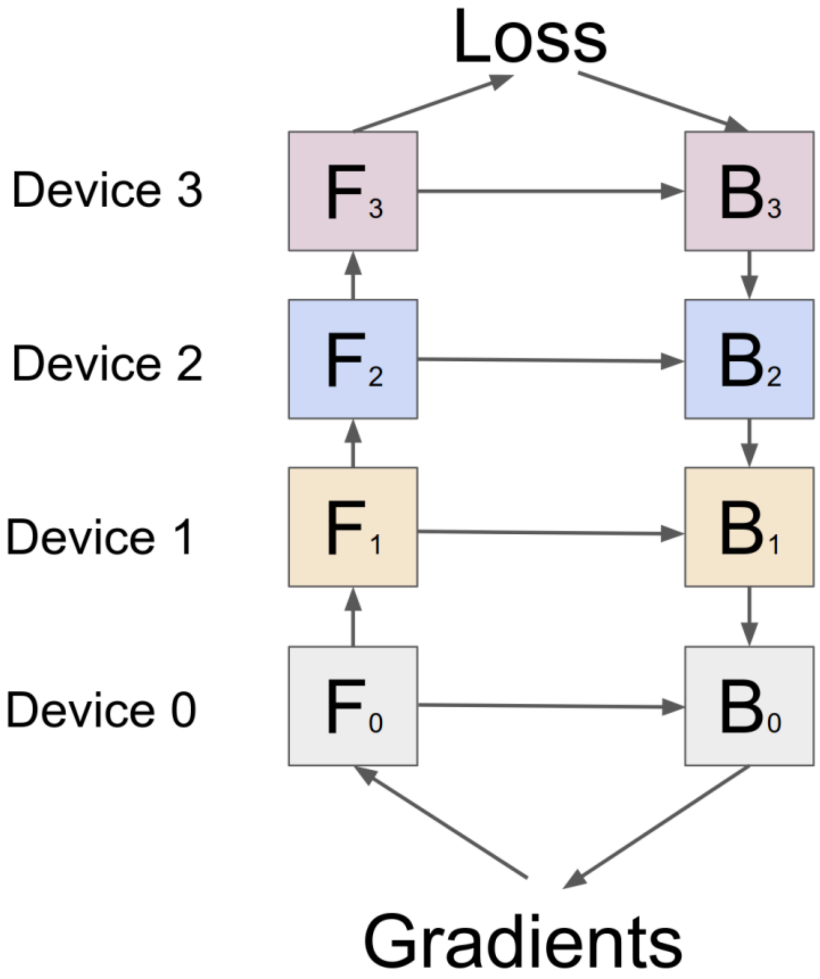
- Main idea: **partition the layers** of a neural network among different worker machines.

- This makes each worker responsible for a subset of the parameters.

- Forward and backward signals running through the neural network during backpropagation now also run across the computer network between the different parallel machines.
  - Particularly useful if the parameters won’t fit in memory on a single machine.
  - This is very important when we move to specialized machine learning accelerator hardware, where we’re running on chips that typically have limited memory and communication bandwidth.

Pipeline Parallelism

- Distribute a DNN over multiple workers by **assigning each layer to its own worker**.
  - Each worker manages and updates the parameters for its own layer.
  - Use **microbatching** to avoid stalls
- Advantage: **workers no longer need to store the entire model**
  - Can often keep parameters in memory



From "GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism"

Federated learning

- Sometimes, **your data is inherently distributed**
  - For example, data gathered on people's mobile phones
  - For example, data measured by internet-of-things devices

- Rather than centralizing the data, may want to learn on the distributed devices themselves
  - E.g. to preserve the privacy of users

- This is called **federated learning**
  - **Lots of interest from industry right now**

Distributed computing for hyperparameter optimization

- This is something we've already talked about.

- Many commonly used hyperparameter optimization algorithms, such as **grid search and random search**, are very simple to distribute.
  - They can easily be run on many parallel workers to get results faster.

Questions?

- Upcoming things
  - Final project proposal — **due today**
  - Programming Assignment 2 — **due Wednesday**