Distributed model pipelines

on just splitting it up

hi, I'm Helen!

- Machine Learning Engineer at Dessa
- Sidewalk Toronto Fellow
- Toronto Women's Data Group
- Background in mathematics + writing
- Recently a top-3% placement in the ACM RecSys Challenge
- Creator of @whalefakes, a Twitterbot which outputs whalefacts powered by GPT-2

Tweet

whalefakes
@whalefakes

just like all of your dads, whales never forget a dog's name

1:48 PM · Jul 8, 2019 · Twitter Web Client

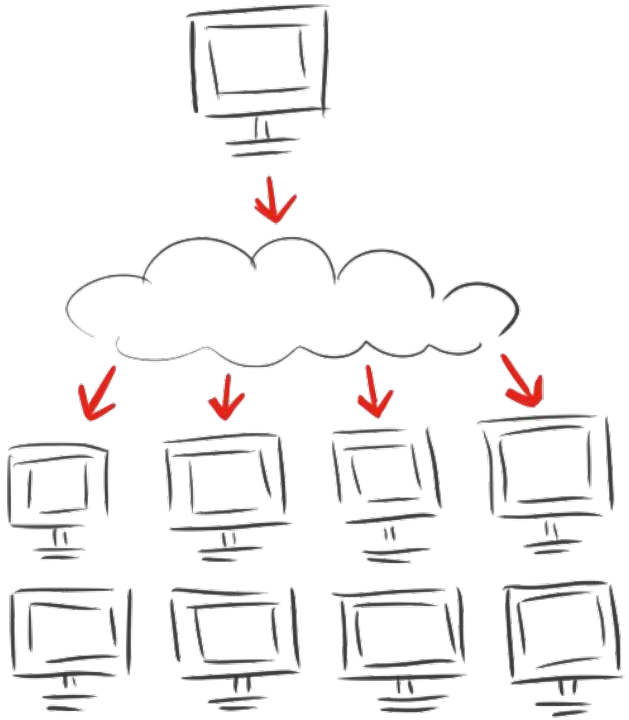
2 Retweets **24** Likes

Agenda

- What does it really mean to distribute something?
  - What is + isn’t possible and common misconceptions
- The role of the GPU
- Distributed data preprocessing
- Distributed model training
  - Data parallelism
  - Model parallelism
- Distributed model serving in production

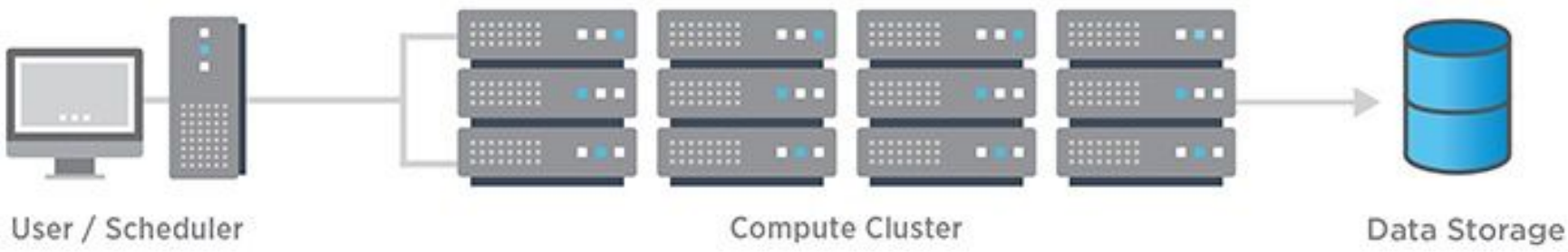
What is distributed computing???

- A group of computers (“a cluster”) working together as to appear as a single computer to the end-user
- These computers operate concurrently and can fail independently without affecting the whole system’s uptime
- You only need to interact with one point to send a job across the cluster
- No fear! Frameworks have been built for this



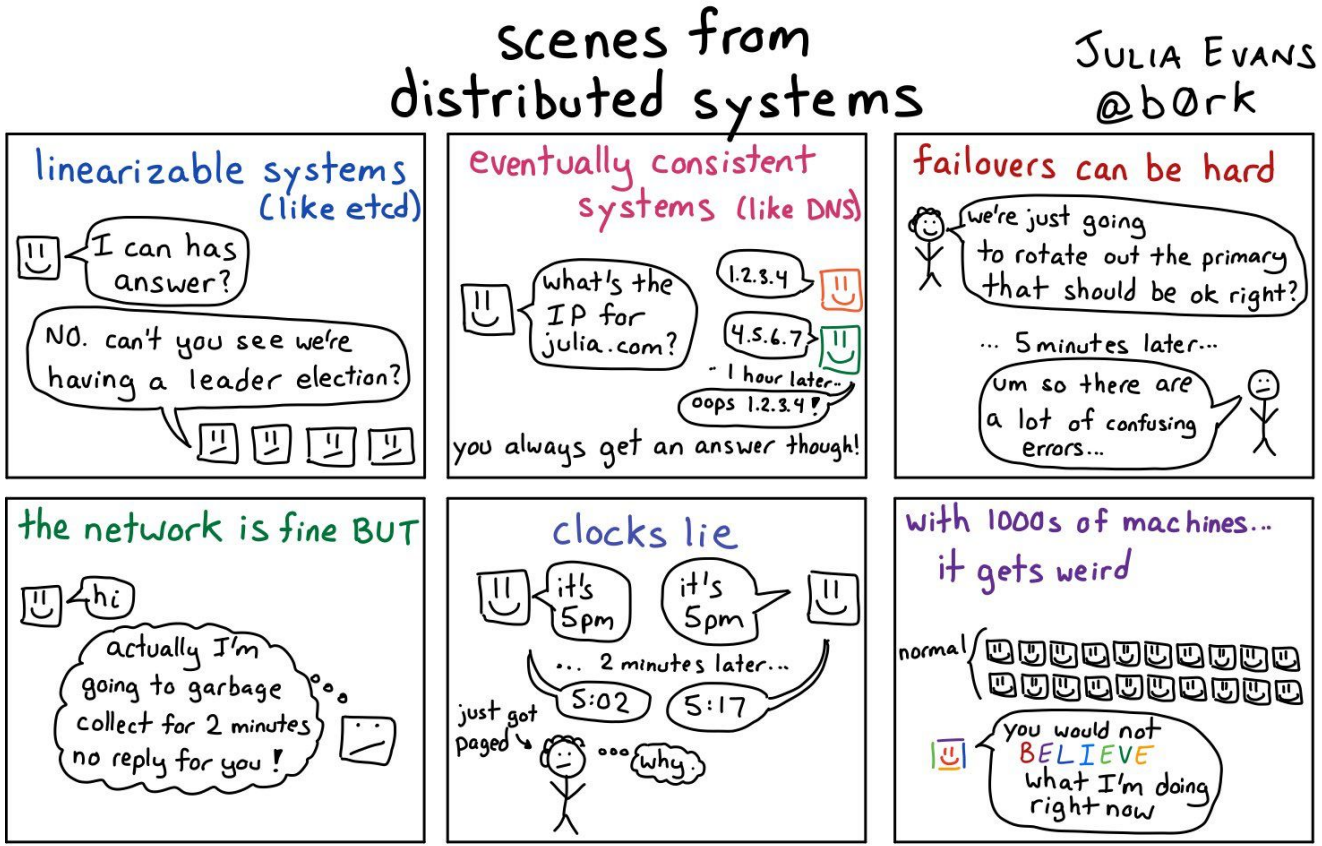
https://www.freecodecamp.org/news/a-thorough-introduction-to-distributed-systems-3b91562c9b3c/

Every single distributed computing paradigm ever is just some variation of this diagram



this looks like it would be simple enough, right?

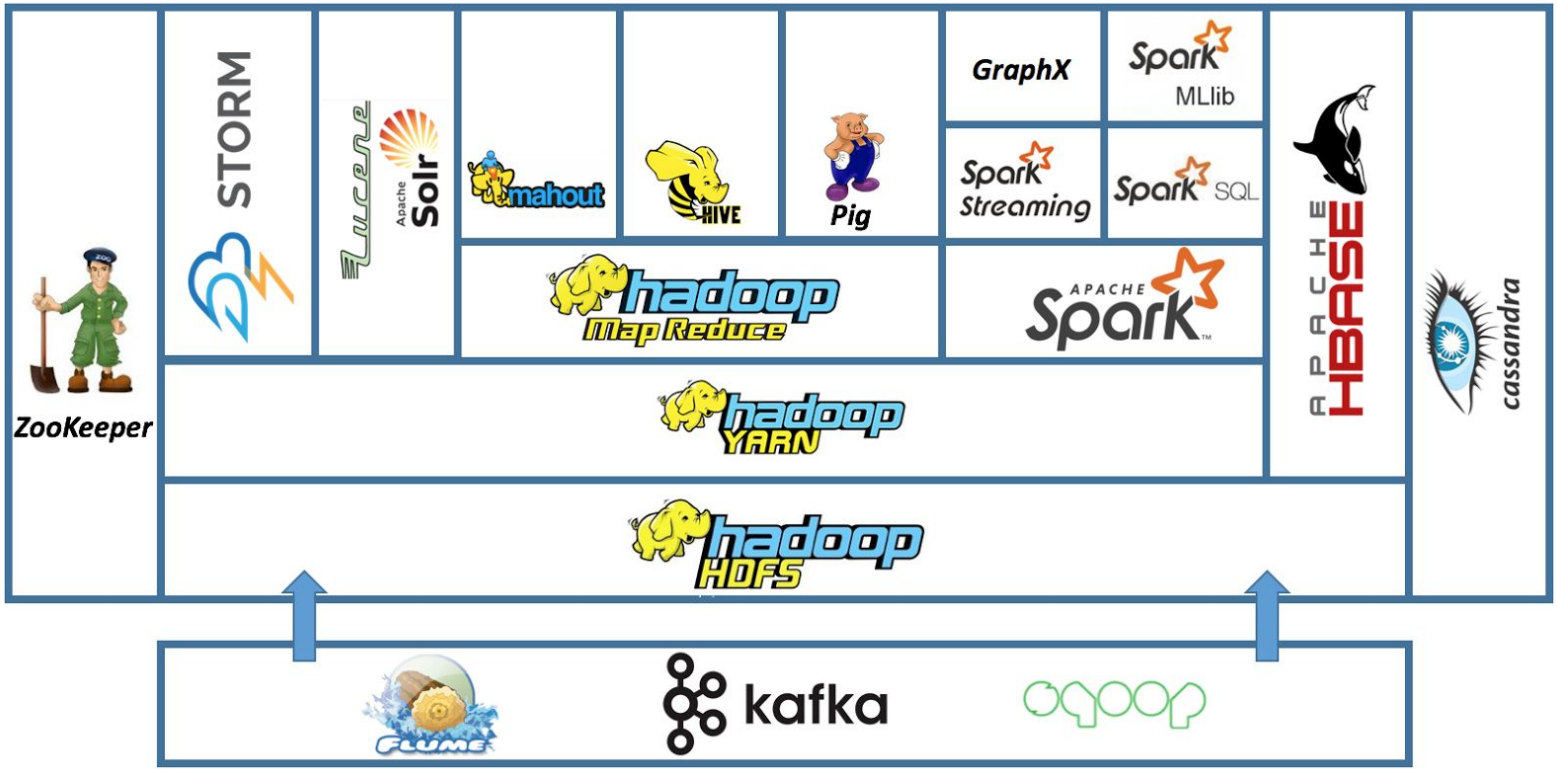
hint: no it's never that simple



the 2010s: the distributed computing hype train

- The rise and fall of “big data”
- There was a big data hype train in the early 2010s
  - Hadoop, MapReduce, big data lakes, YARN…
  - Lots of enterprises spent lots of money on this technology
  - This tech wasn’t built for machine learning originally
- It’s still used today, but for simple data preprocessing, reports, SQL queries, etc.

Distributed computing ecosystem in the early 2010s...



the CAP theorem

It is impossible for a distributed data store to simultaneously provide more than ⅔ of following.

- *Consistency*: Every read receives the most recent write or an error
- *Availability*: Every request receives a (non-error) response – without the guarantee that it contains the most recent write
- *Partition tolerance*: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

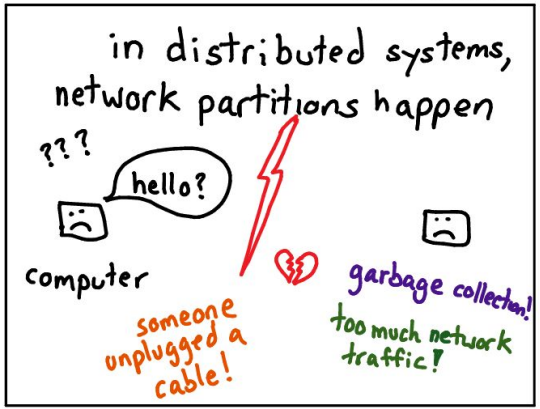
Source: Wikipedia

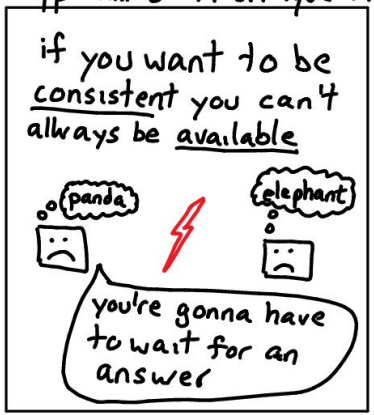
the CAP theorem

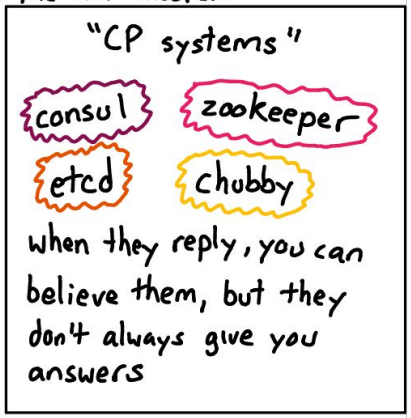
Julia Evans
@b0rk

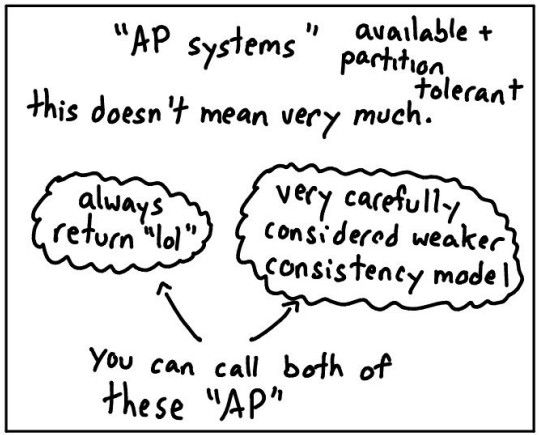
"C" stands for linearizable

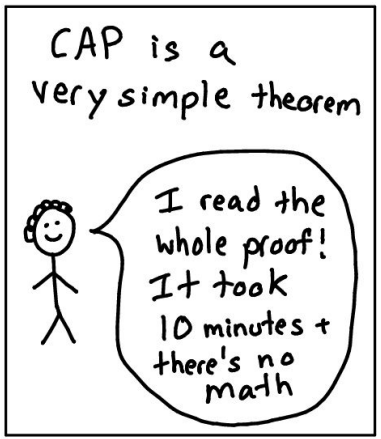
from Martin Kleppmann's "A critique of the CAP theorem"

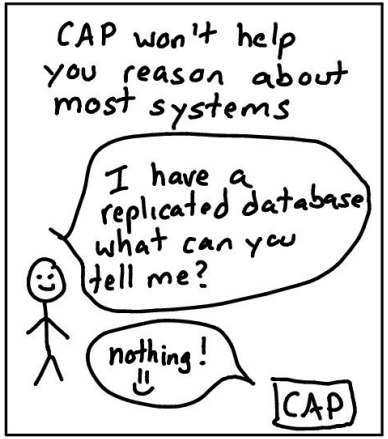












All this to say,

- Distributed computing is really hard and we're going to cover just the basics today
- There's an entire subfield of theoretical computer science dedicated to this
- It's worth taking an actual course on this topic if you're interested

When is distributed computing useful?

- When data doesn’t fit into memory
  - Ex. Your single server has 128GB of memory but your data is 250GB
- When your process can be chopped up into little bits which can run independently on many machines and have the resulted stitched back together
- When you have big models which don’t fit into the memory of a single GPU

Things we *can* do with distributed computing

- Distribute some parallelizable data preprocessing operations across many machines so they run faster (ie. with Spark)
- Distribute *data* across a cluster and run models on small parts of the data to train a model faster
- Distribute parts of a *model* across a cluster to train it faster

Things we *can't* do with distributed computing

- Distribute everything
  - Some things are not distributable
  - There is no generalizable "distribute this" algorithm
  - If you can't figure out how to split it into asynchronous pieces, it might not be a good fit for the distributed computing paradigm
- Sequential operations, sometimes (ie. sort all customer transactions by ID over all time)
  - Implementing a sort can get pretty tricky, as you can imagine
  - You can sort on a GPU
  - But sorting in certain out-of-memory CPU situations can be tough

Parallelizing data preprocessing

- If your data fits into server memory, you can distribute processing across a single machine with the Python multiprocessing module
  - This can be a lot of work to build from scratch
  - Slower than distributing across a cluster, of course
- Will not work well for processes that are inherently sequential

Parallelizing data preprocessing

- You can also use a framework like Spark or Dask
- **Spark**: Distribute across “executors” which take up a pre-allocated amount of memory on a machine (“worker”)
  - Requires a decent amount of knowledge on memory management
  - Runs as part of the Java / Scala ecosystem, not native Python
- **Dask**: A new framework built for scaling Python scientific libraries such as numpy, pandas, scikit-learn
  - Early stage, sometimes not stable, catching on in the community

A quick note on data residency

- In all of these frameworks, the location of your data matters
- If it is local on the machine you're running your code on, that's fine
- If it lives on the cloud (ie. in an S3 bucket), that's fine too
- If not either of the above, either the data or the code will have to be moved so they are accessible to the other
    - This can be an expensive operation if you're moving data

Data parallelism for model training

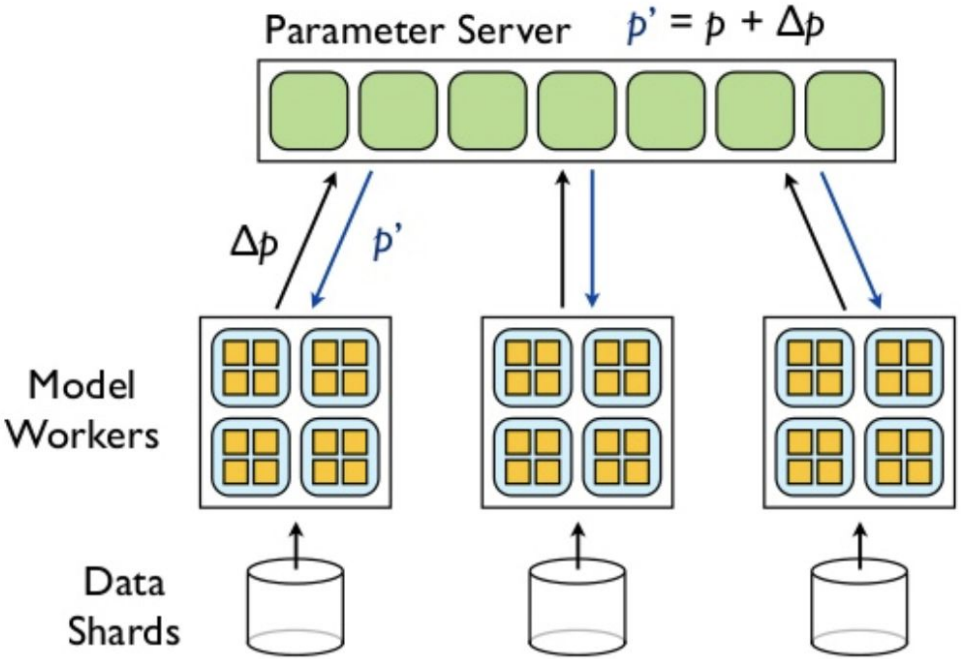
- Distributing parts of your data on several machines
- For each GPU/node, use the same model/parameters to do the forward propagation
- Send a small batch of different data to each node
- Compute the gradient normally
- Send the gradients back to the main node
    - This step is asynchronous because the speed of each GPU/node is slightly different.
- Once we get all the gradients, we calculate the (weighted) average of the gradients, and use the (weighted) average of the gradients to update the model/parameters.
- Then we move on to the next iteration.
- Source: https://leimao.github.io/blog/Data-Parallelism-vs-Model-Paralelism/

Data parallelism for model training

- This scales with the amount of data available; you optimize with the whole dataset quicker
- Major upside is less communication between the workers
- Downside: Models have to fit on a single node to do this

Data Parallelism:

Asynchronous Distributed Stochastic Gradient Descent



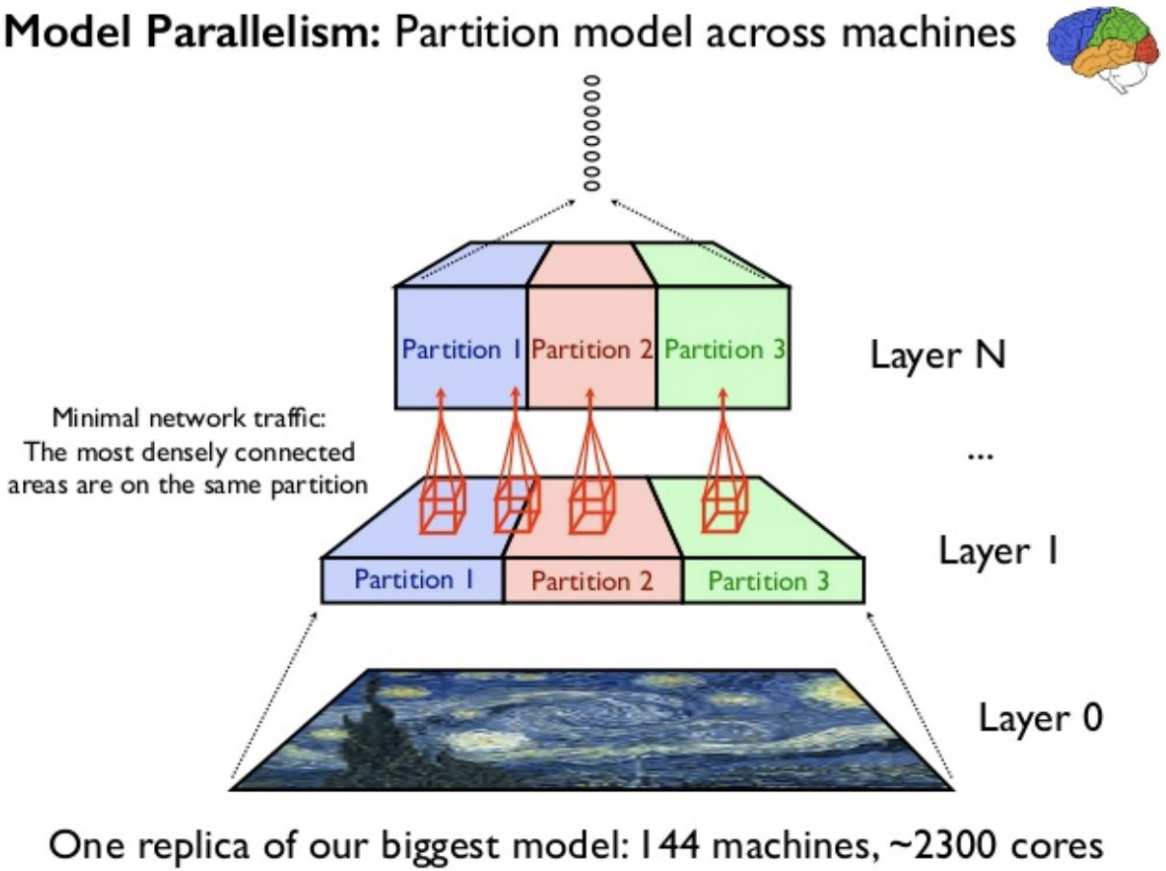
This one too. https://www.slideshare.net/hustwj/cikm-keynotenov2014

Model parallelism for model training

- Storing model parameters / weights / etc. is expensive because it requires GPU memory, which is often limited (16-32GB)
- So we split up the model layers across machines
- Here's how it goes:
- Divide the deep learning models to n pieces
- Put a few consecutive layers on a single node
- Calculate its gradients and send them onto the next GPU
- Source: https://leimao.github.io/blog/Data-Parallelism-vs-Model-Paralelism/

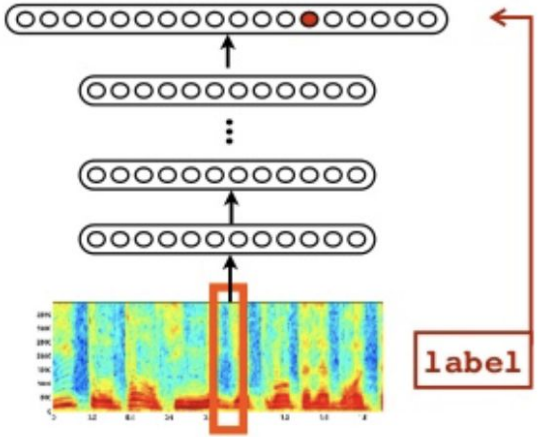
Model parallelism for model training

- **Example**: 10 GPUs and we want to train a simple ResNet50 model.
- Assign the first 5 layers to GPU #1, the second 5 layers to GPU #2, and so on, and the last 5 layers to GPU #10.
- During the training, in each iteration, the forward propagation has to be done in GPU #1 first.
- GPU #2 is waiting for the output from GPU #1, GPU #3 is waiting for the output from GPU #2, etc.
- Once the forward propagation is done, calculate the gradients for the last layers which resides in GPU #10
- Update the model parameters for those layers in GPU #10.
- Back propagate the gradients to the previous layers in GPU #9, etc.



I stole this great slide from Jeff Dean at Google: https://www.slideshare.net/hustwj/cikm-keynotenov2014

Acoustic Modeling for Speech Recognition



Close collaboration with Google Speech team

Trained in <5 days on cluster of 800 machines

30% reduction in Word Error Rate for English
("biggest single improvement in 20 years of speech research")

Launched in 2012 at time of Jellybean release of Android

Also this one. https://www.slideshare.net/hustwj/cikm-keynotenov2014

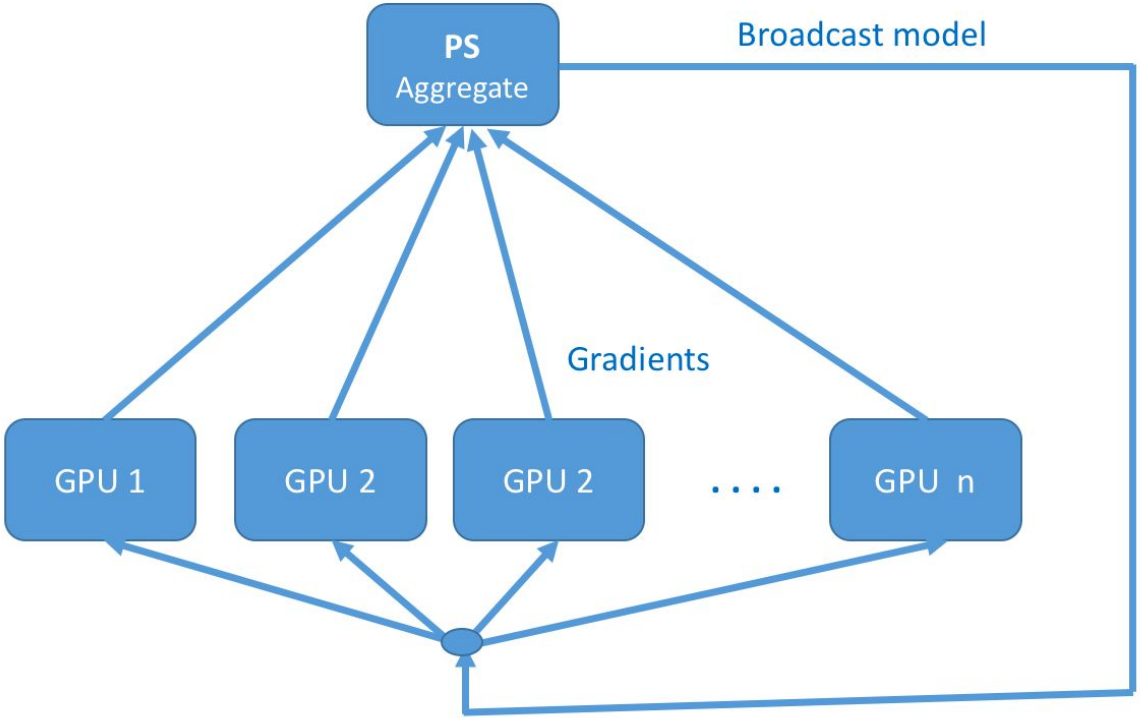
Model parallelism for model training

- GPUs can live on the same machine or across machines
- You can rent machines on the cloud with many GPUs in them
- If they live on the same machine, that's great
    - If they don't, they'll have to send parameters across a network to each other with some message passing interface
    - Make sure your machines are co-located!

Model parallelism for model training

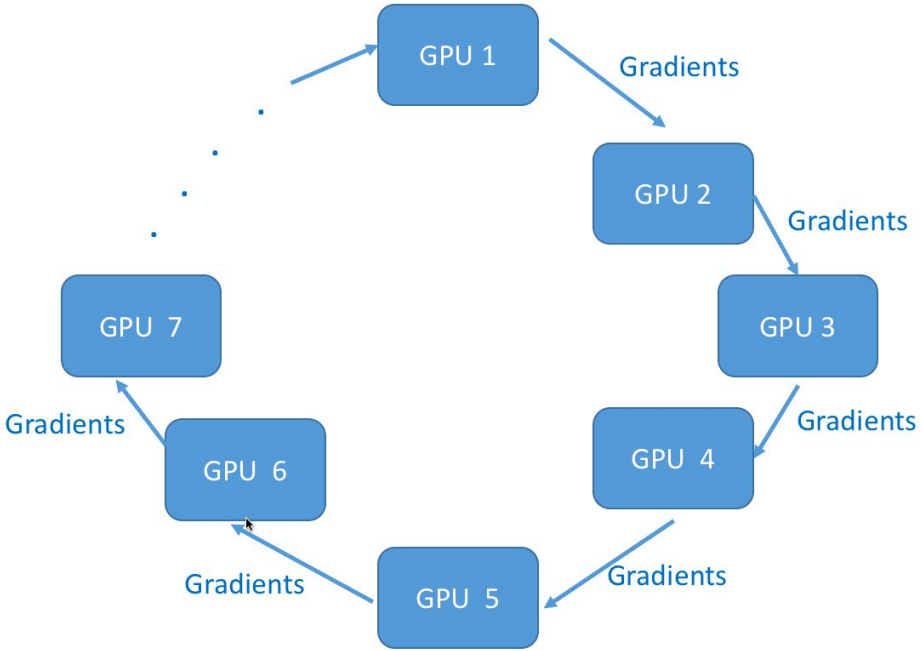
- This can be centralized or decentralized
- Depends on how quickly you can move lots of stuff over the network
- Centralized: uses a parameter server to orchestrate
- Decentralized: all the nodes talk to the next ones

Centralized: based on a parameter server



https://towardsdatascience.com/distributed-tensorflow-using-horovod-6d572f8790c4

Decentralized: all-reduce algorithm



https://towardsdatascience.com/distributed-tensorflow-using-horovod-6d572f8790c4

OK, but you said there were frameworks for this

- If that sounds like a lot of work, it is.
- Good thing that other people dealt with these problems already
- Distributed Tensorflow
  - This is built right into your regular Tensorflow + Keras distribution
- Horovod
  - A framework by Uber using the all-reduce algorithm

Within the Tensorflow ecosystem

- There's lots of built-in ways to distribute stuff already

Training API	MirroredStrategy	TPUStrategy	MultiWorkerMirroredStrategy	CentralStorageStrategy	ParameterServerStrategy
Keras API	Supported	Support planned in 2.0 RC	Experimental support	Experimental support	Supported planned post 2.0
Custom training loop	Experimental support	Experimental support	Support planned post 2.0 RC	Support planned in 2.0 RC	No support yet
Estimator API	Limited Support	Limited Support	Limited Support	Limited Support	Limited Support

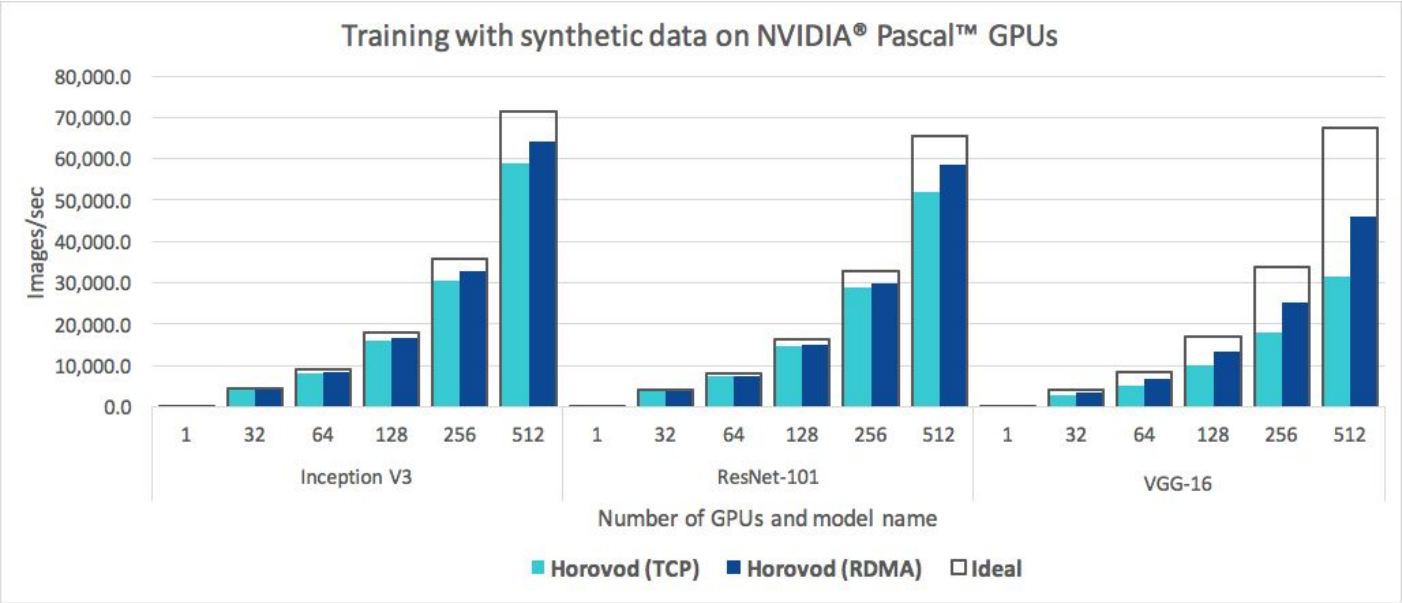
Docs are here: https://www.tensorflow.org/beta/guide/distribute_strategy

Try it for yourself in Keras

- Colaboratory link: https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/r2/tutorials/distribute/keras.ipynb
- This tutorial uses the tf.distribute.MirroredStrategy, which does in-graph replication with synchronous training on many GPUs on one machine.
  - It copies all of the model's variables to each processor.
  - Then, it uses all-reduce to combine the gradients from all processors and applies the combined value to all copies of the model.

Another way: Uber's Horovod framework

- A horovod is a traditional Russian dance where people hold hands in a circle, just in case you were wondering

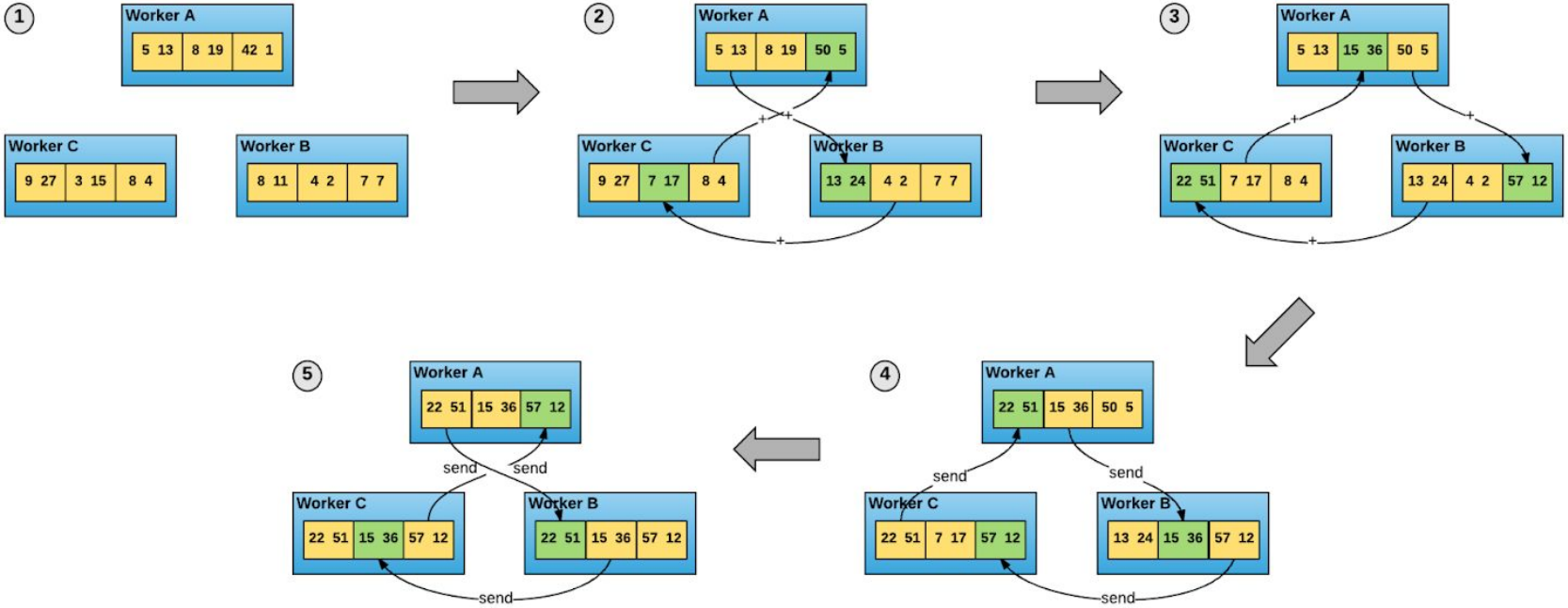


Horovod: how it works

- Each copy of the training script
  - reads a chunk of the data
  - runs it through the model
  - computes model updates (gradients)
- Gradients get averaged amongst the multiple copies
- Model gets updated; weights communicated to peers

Horovod applies Baidu's algorithm for averaging gradients and communicating those gradients to all nodes. It uses NCCL-2, NVIDIA's library that provides a highly optimized version of ring-allreduce across multiple machines. https://towardsdatascience.com/distributed-tensorflow-using-horovod-6d572f8790c4

Uber’s Horovod framework: all-reduce algorithm



Try Horovod yourself!

- https://github.com/horovod/horovod