Data 188: Introduction to Deep Learning

Scaling up training

Speaker: Eric Kim
Lecture 20 (Week 13)
2026-04-14, Spring 2026. UC Berkeley.

Announcements

- HW4 released: "Transformers for NLP (machine-translation)"
    - Groups of 4!
    - Start early!

Today's lecture

Multi-GPU training

Distributed training

Scaling laws

Horizontal vs Vertical scaling

When working with large models + large datasets, we want to keep scaling up, ideally by throwing more compute (aka hardware, aka $) at the problem

- Horizontal scaling: add more compute (machines, GPUs)
- Vertical scaling: make each individual machine/GPU faster

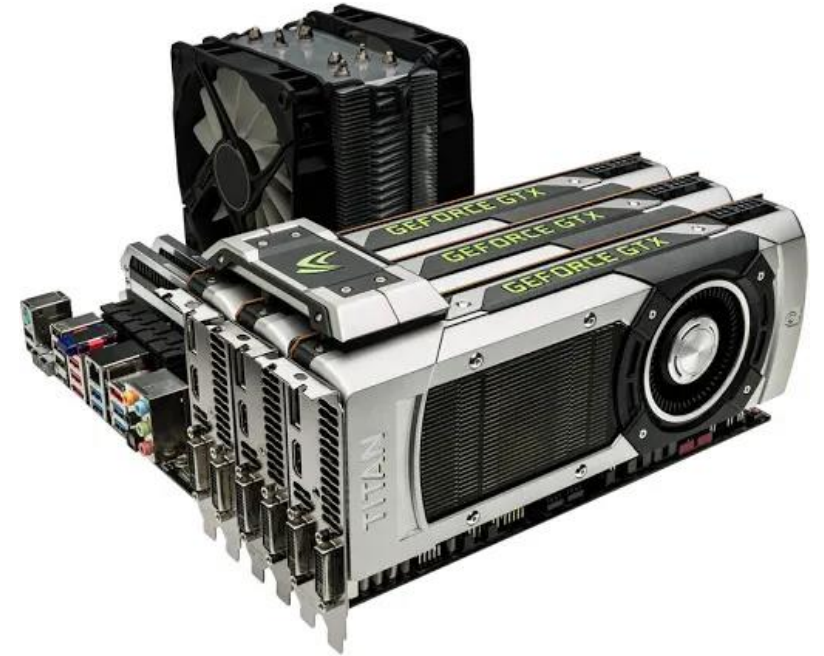
Both are important! But, horizontal scaling is often quicker and easier to do

Multi-GPU

Idea: let's attach multiple GPUs to a single machine at a time! Double the GPU's, double the compute!

Engineering Question: how to effectively utilize multiple GPUs on a single machine for training DNN models?

For simplicity: assume that all GPUs on a single machine are all the same type, ie same exact model (in practice this is true 99% of the time)



https://shivaraj-karki.medium.com/set-up-a-multi-gpu-server-from-scratch-ca36d97f934a

Multi-GPU + Cloud computing costs

As of 2026, it's common for practitioners/researchers to rent GPU's on the cloud from cloud providers such as AWS EC2.

This is often more cost-effective than buying + maintaining your own GPU clusters (that will inevitably become obsolete in 1-2 years!).

Instance type	Num GPUs	Total GPU memory (GB)	Cost per year ($) (on-demand)*
p4d.24xlarge (2020-11-02)	8 (A100)	320 GB	$192k $290k (2024-11)
g6e.24xlarge (2024-08-15)	4	192 GB	$132k
g6e.48xlarge (2024-08-15)	8	384 GB	$264k
p5en.48xlarge (2024-12)	8 (H200)	1128 GB	$554k

* Turns out that it's non-trivial to efficiently attach multiple GPUs on a single machine. See: Nvidia NVLink [link]

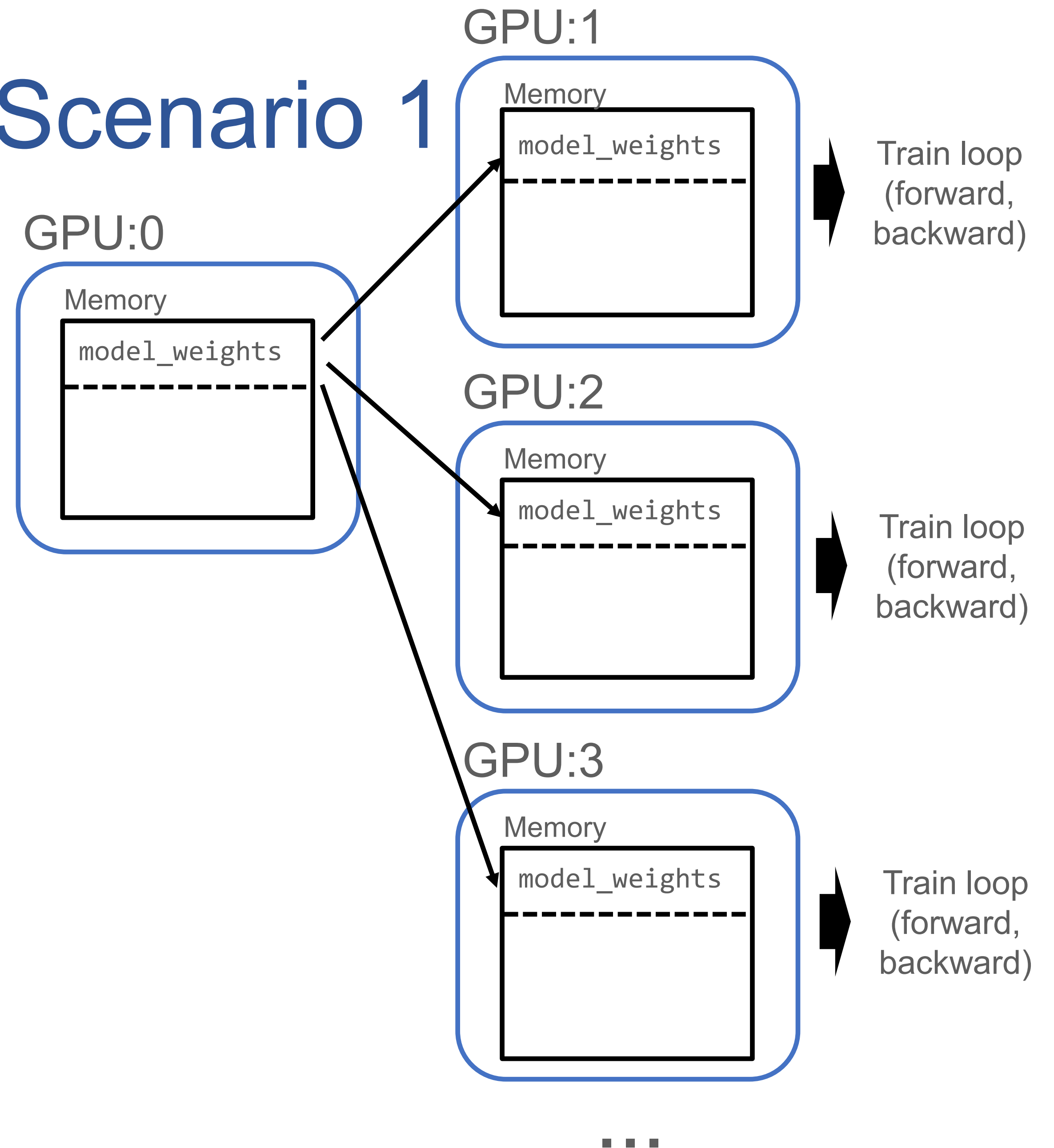
* Cost measured on 2026-04-13 via this AWS web tool.

Multi-GPU: Scenario 1

Suppose our machine has 8 GPUs, and our DNN model (and activations) can fit on a single GPU.

Question: if we wanted to maximize training throughput, what's one way we can utilize the 8 GPUs?

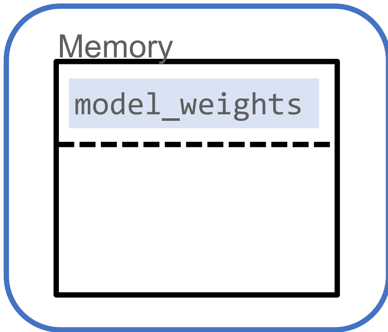
Answer: load the model onto all 8 GPUs separately, and have each GPU do their own forward/backward passes in parallel!



Implementation Note: each GPU trainer gets its own training Dataloader. Take care to ensure that each Dataloader splits up the training dataset appropriately (ex: don't want all N GPU workers to train on the same batches!)

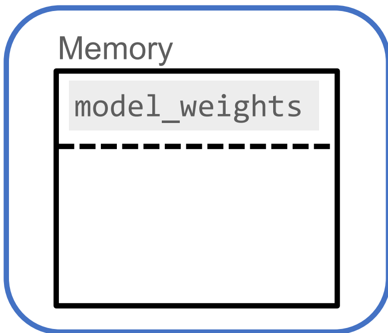
Multi-GPU: Scenario 1

GPU:0



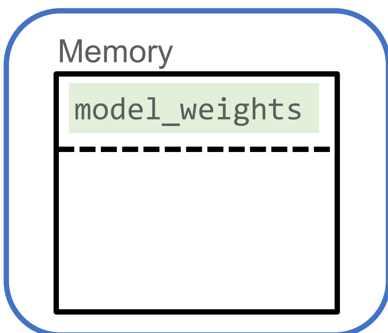
Train loop (forward, backward)

GPU:1



Train loop (forward, backward)

GPU:2



Train loop (forward, backward)

...

Question: suppose each GPU worker has different training dataset splits (aka each GPU worker operates on different train batches).
After N training batches, will the model parameters be the same across all GPU workers?

Answer: Nope! Each GPU worker's train loop will process different batches, thus the gradients for GPU:0 will be different than GPU:1, resulting in different model weights for each worker. This complicates things!

Question: how can we ensure that, after each training batch (for all N GPUs), the model_weights are the same ("in sync") for all N GPUs?

Answer: one popular way is to aggregate all of the GPU worker's gradients (average), then have each GPU worker apply the same aggregated ("global average") gradient.
Ensures that model_weights is the same on all GPU workers!

Multi-GPU: Aggregate gradients

Note: there exist other more efficient, cleverer ways of optimizing this. Ex: NCCL's ring, tree algorithms: [link]

GPU:0
Memory
model_weights
Train loop (forward, backward)
Gradient for GPU:0

GPU:1
Memory
model_weights
Train loop (forward, backward)
Gradient for GPU:1

GPU:2
Memory
model_weights
Train loop (forward, backward)
Gradient for GPU:2

Aggregate gradients (Average)

...

Step 1: Aggregate incoming gradients

GPU:0
Memory
model_weights
Apply gradient
Global averaged gradient

GPU:1
Memory
model_weights
Train loop (forward, backward)
Global averaged gradient

GPU:2
Memory
model_weights
Train loop (forward, backward)
Global averaged gradient

Aggregate gradients (Average)

...

Step 2: Apply global averaged gradient

Pytorch: DistributedDataParallel (DDP)

torch.nn.parallel.DistributedDataParallel (aka "DDP")

DDP does exactly what we proposed! Consists of two parts:

- **Model copy**. copy the initial model weights to each GPU, via `.to(device)`
- **Gradient synchronization**. adds a "hook" to the backwards() pass to aggregate the gradients across all GPUs (via averaging)
  - This ensures that all GPUs use the same gradients for each step, which ensures that each GPU always has the same model weights

Each GPU processes their own batches independently

For 8 GPUs, you've effectively increased your batchsize by 8x!

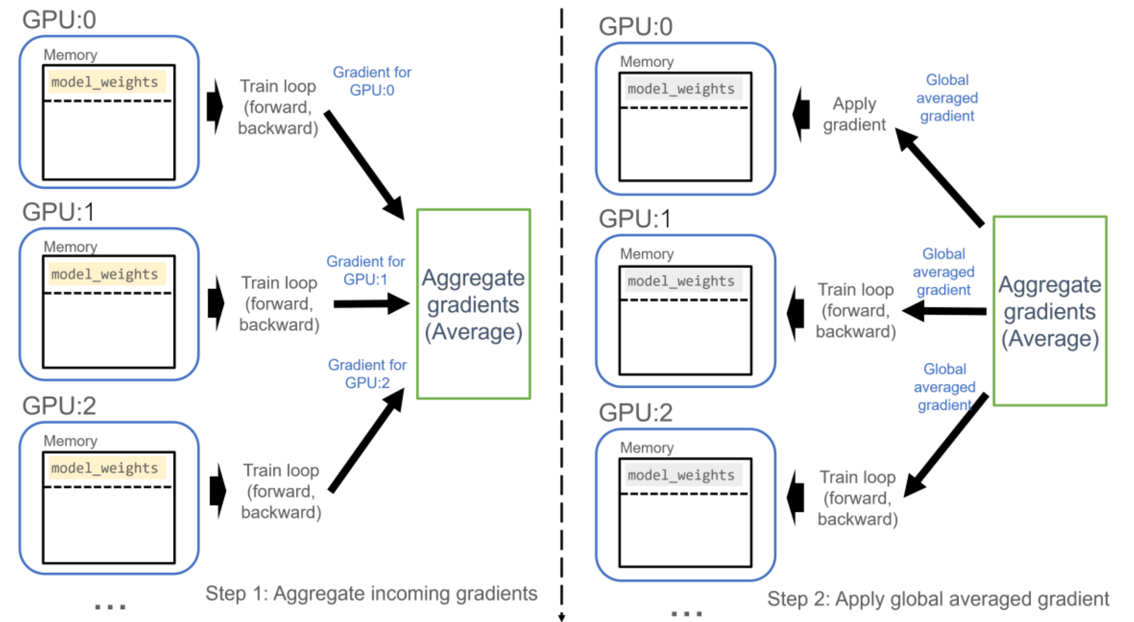
DDP: multi-GPU training

For N GPUs, each GPU first independently performs their own "local" forward/backwards pass to calculate N different gradient updates.

Then, DDP aggregates the N gradients and averages them, producing an aggregated gradient.

DDP then transmits this aggregated gradient to each GPU.

Each GPU then updates their (local) weights with this averaged gradient. At this point, all GPU's "local" model weights will be exactly the same ("in sync") due to this synchronized gradient calculation.



Terminology: world_size, rank

world_size: the total number of workers. Ex: total number of GPUs

rank: an integer between [0, world_size-1] (inclusive).

Setup: each GPU will be assigned a "rank" and the "world_size", and will use this metadata to perform its job

Ex: Suppose world_size is 8.

A GPU worker assigned rank=0 will use GPU0: `torch.device("cuda:0")`

- Rank=1 will use GPU1: `torch.device("cuda:1")`
- ...
- Rank=7 will use GPU7: `torch.device("cuda:7")`

DDP in pytorch

```python
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP

class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = nn.Linear(10, 10)
        self.relu = nn.ReLU()
        self.net2 = nn.Linear(10, 5)
    def forward(self, x):
        return self.net2(self.relu(self.net1(x)))
```

```
# use torch_elastic to launch script N times, one
# for each GPU worker
# --nnodes: number of nodes. For us, 1 machine
# --nproc_per_node: number of GPUs. For us: 8
torchrun --nnodes=1 --nproc_per_node=8 --rdzv_id=100 --
rdzv_backend=c10d --rdzv_endpoint=$MASTER_ADDR:29400
elastic_ddp.py
```

```python
def demo_basic():
    dist.init_process_group("nccl")
    rank = dist.get_rank()
    print(f"Start running basic DDP example on rank {rank}.")
    # create model and move it to GPU with id rank
    device_id = rank % torch.cuda.device_count()
    model = ToyModel().to(device_id)
    ddp_model = DDP(model, device_ids=[device_id])
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(device_id)
    loss_fn(outputs, labels).backward()
    optimizer.step()
    dist.destroy_process_group()
    print(f"Finished running basic DDP example on rank {rank}.")

if __name__ == "__main__":
    demo_basic()
```

Boilerplate code to initialize distributed backend (NCCL)

`device_id` is the assigned GPU device to use: f"cuda:{device_id}"

Wrap model on DDP()

Can use `ddp_model` just like regular `model`! Very convenient.

For more info, see: https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

all_reduce: Gradient aggregation

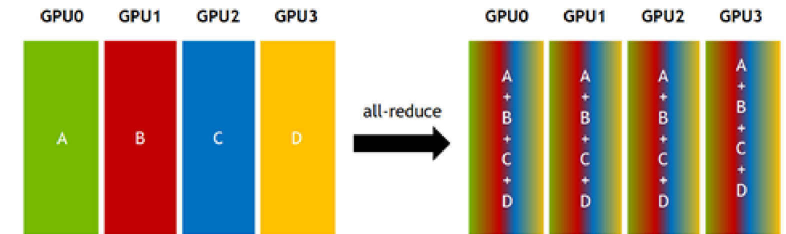
Recall: in DDP, for N GPUs, DDP must aggregate the N gradients, average them, and then transmit this aggregated gradient to each GPU.

This cross-GPU operation is known as: "**all_reduce**"

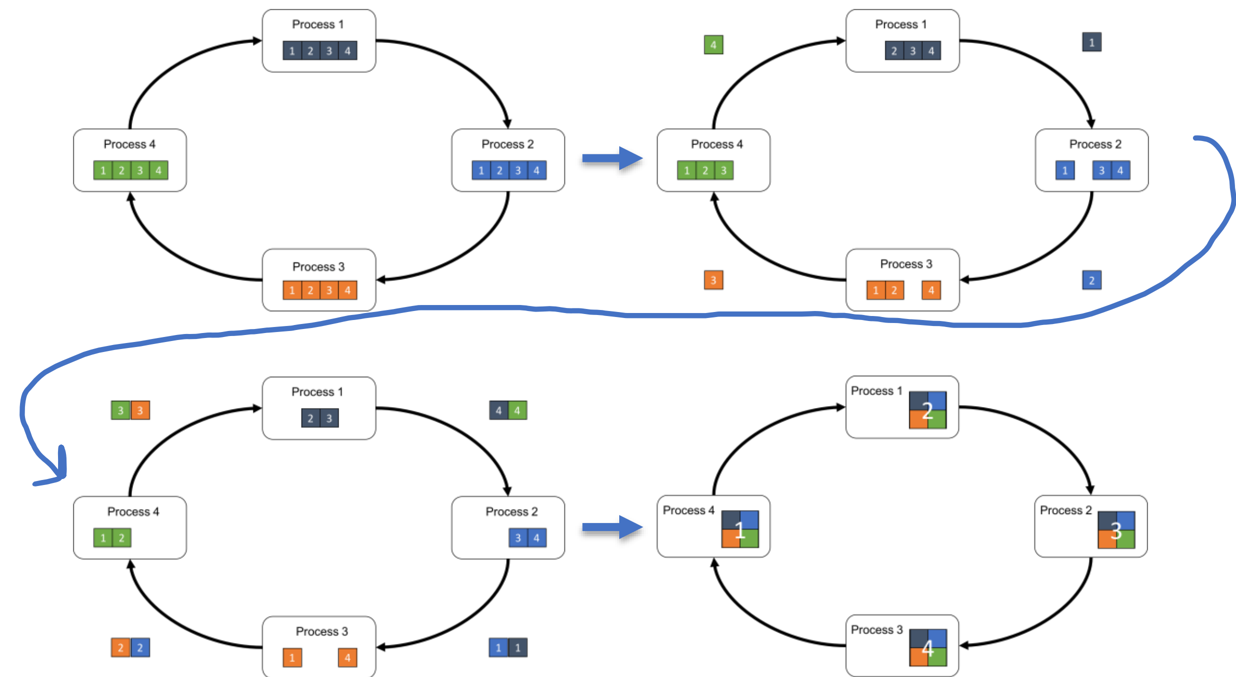
For multi-GPU setups, all_reduce can be a significant bottleneck for training throughput.

Optimized algorithms exist for this, such as "ring" all_reduce

Pytorch supports dispatching the all_reduce call to a variety of libraries [link]
For Nvidia GPUs: use NCCL [link]



https://developer.nvidia.com/blog/fast-multi-gpu-collectives-nccl/



https://tech.preferred.jp/en/blog/technologies-behind-distributed-deep-learning-allreduce/

Multi-GPU: Scenario 2

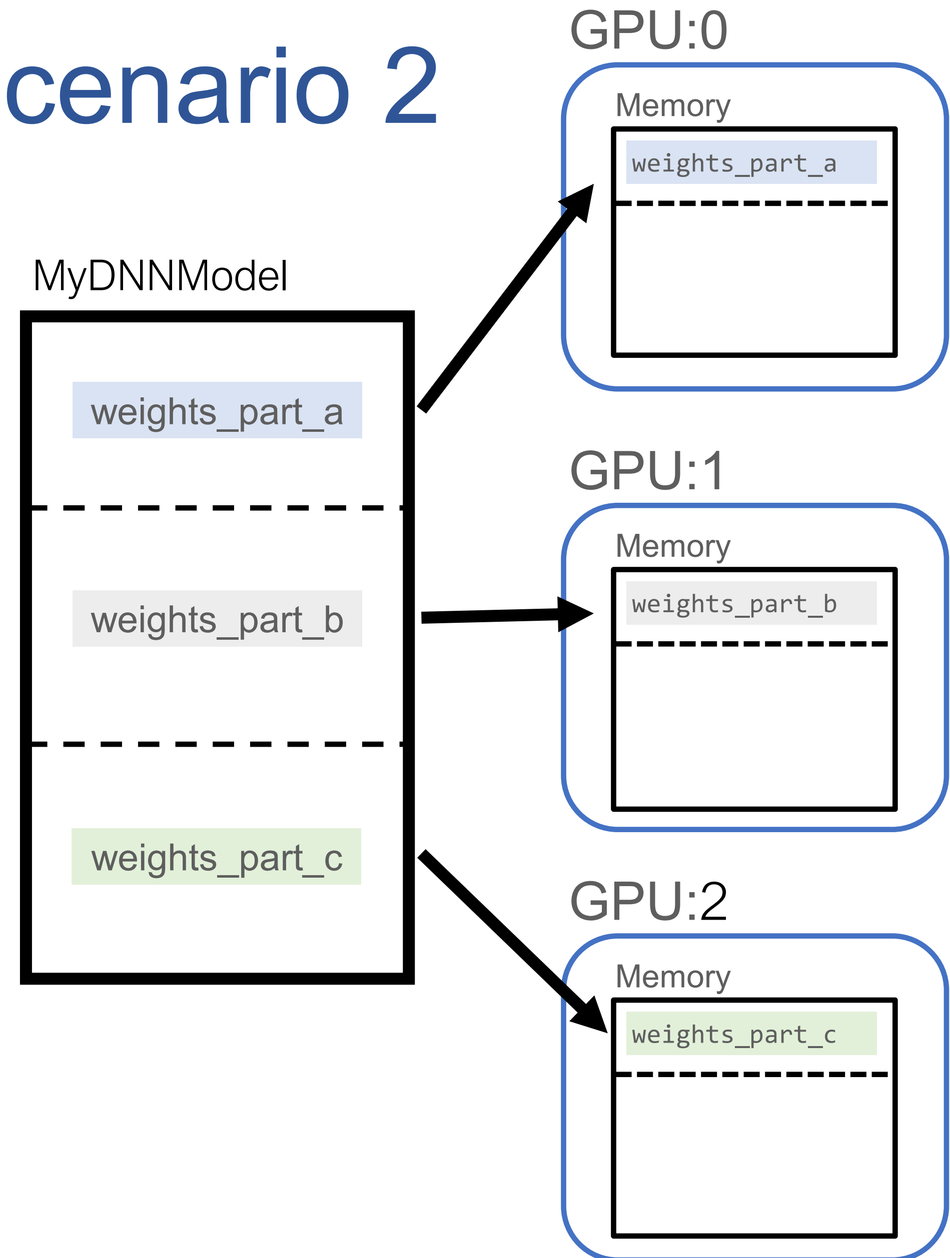
Suppose our machine has 8 GPUs, and our DNN model (and activations) can't fit on a single GPU.

Question: how can we utilize the 8 GPUs to train our DNN model?

Answer: split the model up across multiple GPUs!

Question: what downsides can you think of?

Answer: slower training, due to additional cross-GPU communication



Pytorch: FullyShardedDataParallelism (FSDP)

Pytorch's FSDP implements this "model sharding" idea.

Implementation challenge: how to distribute model parameters in a way that minimizes cross-gpu copies?

Heuristic: FSDP works well for very large models, ie when model weights are a substantial fraction of available GPU memory

FSDP is also handy to increase total batchsize at the expense of some efficiency

DDP vs FSDP

Name	Impact on: GPU memory	Impact on: Train throughput	When to use?
DistributedDataParallel (DDP)	For N GPUs, model weights are duplicated N times (waste). Con!	Each GPU does forward/backward independently: no cross-GPU communication required (except for all_reduce). Pro!	If your model+activations comfortably fits in GPU memory AND you are happy with your current batch_size
FullyShardedDataParallel (FSDP)	Model weights are instantiated only once (split across N GPUs). Pro!	Training throughput is worse due to cross-GPU communication. Con!	If your model can't fit on a single GPU: MUST use FSDP. Or: if you need a higher batchsize but it won't fit with DDP: try FSDP!

Multi-Node, Multi-GPU: DDP

Suppose we have M machines, each with N GPUs. How to effectively utilize this for training DNN models?

- Fortunately, the earlier principles generalize quite nicely!

DDP:

- M=1 machine, N GPUs: all_reduce across GPUs (within a single machine)
- M>1 machines, N GPUs each: all_reduce across ALL GPUs
  - Involves **cross-machine** communication during gradient sync!

FSDP

- M=1 machine, N GPUs: split model parameters across N GPUs
- M>1 machines, N GPUs each: split model parameters across M*N GPUs
  - Involves **cross-machine** communication during forward/backward

DDP in pytorch (multi-node)

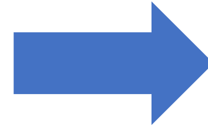
```python
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP

class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = nn.Linear(10, 10)
        self.relu = nn.ReLU()
        self.net2 = nn.Linear(10, 5)
    def forward(self, x):
        return self.net2(self.relu(self.net1(x)))

def demo_basic():
    dist.init_process_group("nccl")
    rank = dist.get_rank()
    print(f"Start running basic DDP example on rank {rank}.")
    # create model and move it to GPU with id rank
    device_id = rank % torch.cuda.device_count()
    model = ToyModel().to(device_id)
    ddp_model = DDP(model, device_ids=[device_id])
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(device_id)
    loss_fn(outputs, labels).backward()
    optimizer.step()
    dist.destroy_process_group()
    print(f"Finished running basic DDP example on rank {rank}.")

if __name__ == "__main__":
    demo_basic()
```



```
# use torch_elastic to launch script N times, one
# for each GPU worker
# --nnodes: number of nodes. For us, 4 machines
# --nproc_per_node: number of GPUs. For us: 8
torchrun --nnodes=4 --nproc_per_node=8 --rdzv_id=100 --rdzv_backend=c10d --rdzv_endpoint=$MASTER_ADDR:29400 elastic_ddp.py
```

Now, `rank` spans multiple machines.
Ex: if each machine has 8 GPUs, then:
Rank=[0, 1, ..., 7]: Machine 0
Rank=[8, 9, ..., 15]: Machine 1

Other than that, everything is nearly the same! Pytorch set things up nicely so that very little has to change when scaling from M=1 machines to M=4 machines.
The world_size/rank abstraction makes things very convenient :)

For more info, see: https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

Example: "Train ImageNet in 1 hour"

In "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour" (2018) [link] the authors, using a large distributed GPU cluster, trained a ResNet-50 model on ImageNet-1k "from scratch" in one hour. A neat accomplishment for that time!

- Hardware: 256 GPUs ("Big Basin" [link] GPU cluster internal to Facebook, 16GB GPU mem per card. Nvidia Tesla P100).

Distributed training setup: basically DDP (they used Caffe2, not pytorch, but same idea)

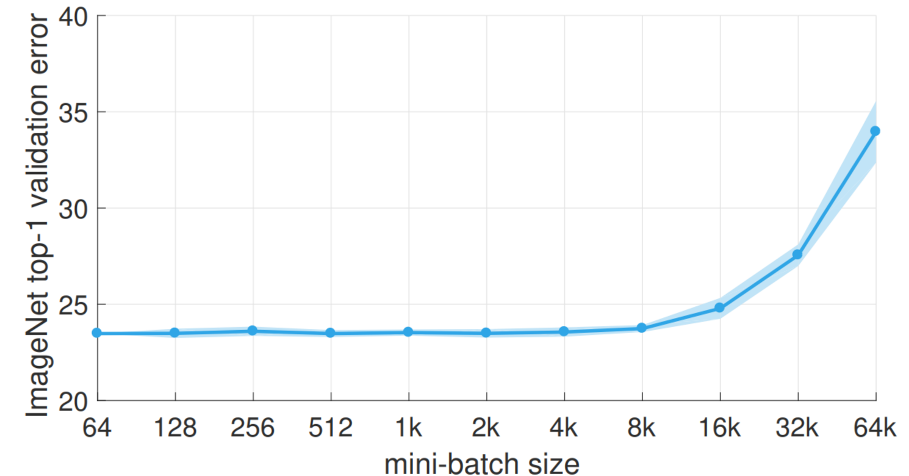


Figure 1. **ImageNet top-1 validation error** ***vs*****. minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. For all minibatch

Learning rate: Linear scaling rule

Learning: when scaling up the number of GPUs (aka increasing the effective batchsize), one must adjust the learning rate accordingly ("linear scaling rule").

Rule: double the batchsize -> double the learning rate.

Num GPUs	Effective batchsize	LR
8	32*8 = 256	0.1
16	512	0.2
32	1024	0.4
64	2048	0.8
128	4096	1.6
256	8192	3.2
...	...	...

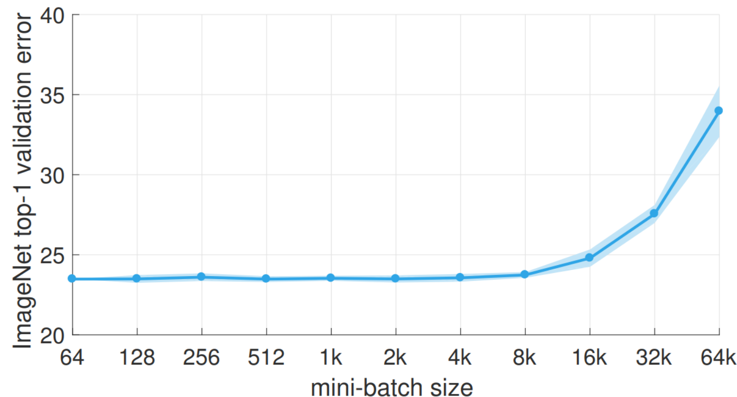


Figure 1. **ImageNet top-1 validation error *vs*. minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. For all minibatch

Gradient quality vs num steps?

Observation: if you keep the number training epochs fixed, then increasing the batchsize leads to fewer model updates.

- Higher batchsize -> higher quality gradient updates, but fewer parameter updates
- Lower batchsize -> noisier gradient updates, but more parameter updates.

What is best? Paper's answer: higher batchsize AND higher learning rate.

- ...to a point. Beyond batchsize=8k, classification error **starts increasing**.

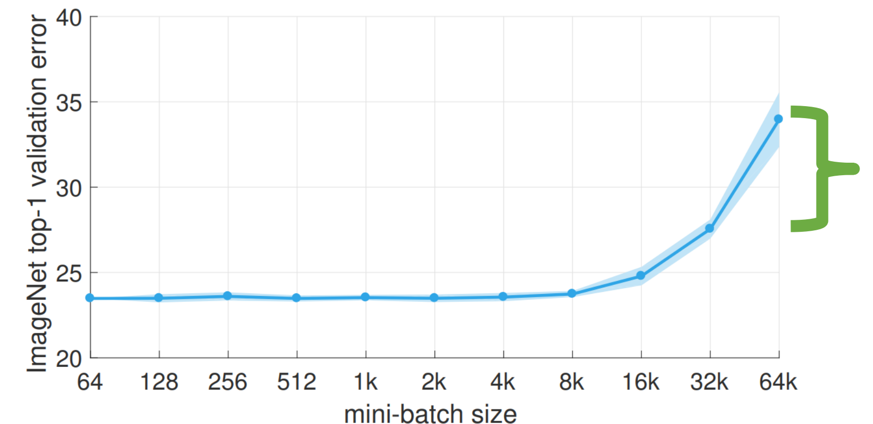


Figure 1. **ImageNet top-1 validation error *vs*. minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. For all minibatch

"Train ImageNet in 1 hour": Hardware advances

A sign that GPU hardware (and DNN libraries + distributed training frameworks) is advancing quickly

...And, a hint that ImageNet-1k is starting to feel small!

(later in Aug 2018, someone showed we can train ImageNet in 18 mins for $40 using AWS cloud! [link])

In 2026, I bet things are even faster + cheaper! Technology marches on...

Tangent: Learning rate schedules

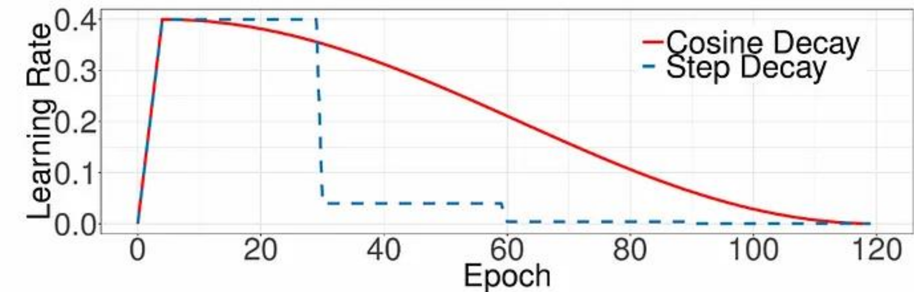
So far in this class, we've used a single learning rate. In practice, it's better to use learning rate schedules

Start learning rate small, then gradually ramp it up to a larger value (eg the first ~100 iterations)

- Intuition: starting learning rate too high often leads to training divergence (eg NaN losses). Thus, we start it low to get the model weights in a "healthy" region, then slowly increase the learning rate

Over the course of training, decay the learning rate

- Intuition: during early parts of training, model needs to make big steps (high LR). But, near the end of training, model is focusing on finer-grained details (small LR).



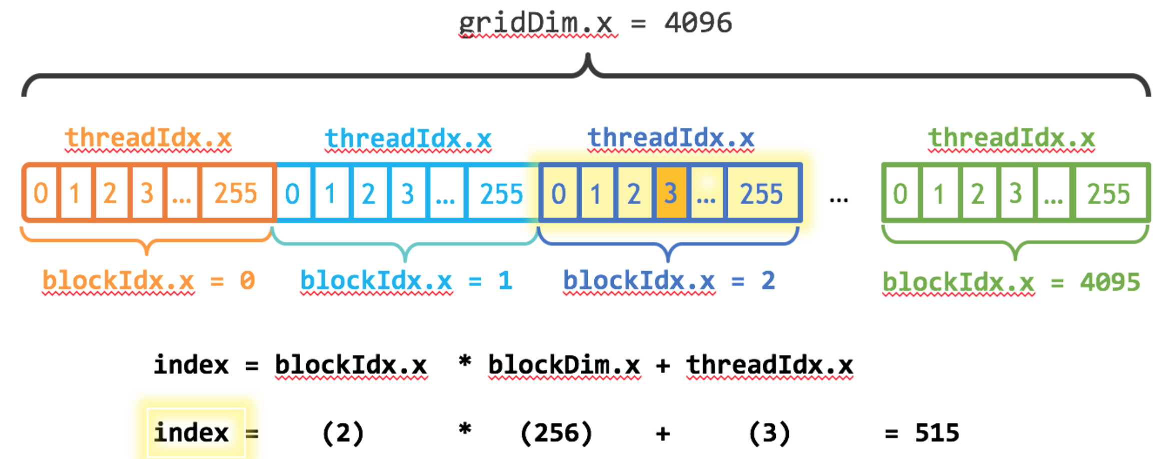
https://pub.towardsai.net/are-you-sure-that-you-can-implement-image-classification-networks-d5f0bffb242d

(unused) CUDA example: element-wise vector addition

Consider: elementwise-vector addition. Given two vectors x, y (with shape=[N]), output (x+y).
This is an "embarrassingly parallel" problem: chunk up the input vectors into K chunks, and process each chunk independently in parallel!

Here, we divide the input array into blocks. Within each block, we assign threads to each block element.

```
__global__
void add(int n, float * x, float * y) {
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```



Ex: for vector element-wise addition, one way to do it ("embarassingly parallel"):
- Break the input vector into numBlocks chunks
- within each block, have a separate thread perform a single `y[i] = a[i] + b[i]`

To learn more, see: https://developer.nvidia.com/blog/even-easier-introduction-cuda/