

# Distributed Training

02457 Machine Learning Operations

Nicki Skafte Detlefsen,

Postdoc

DTU Compute

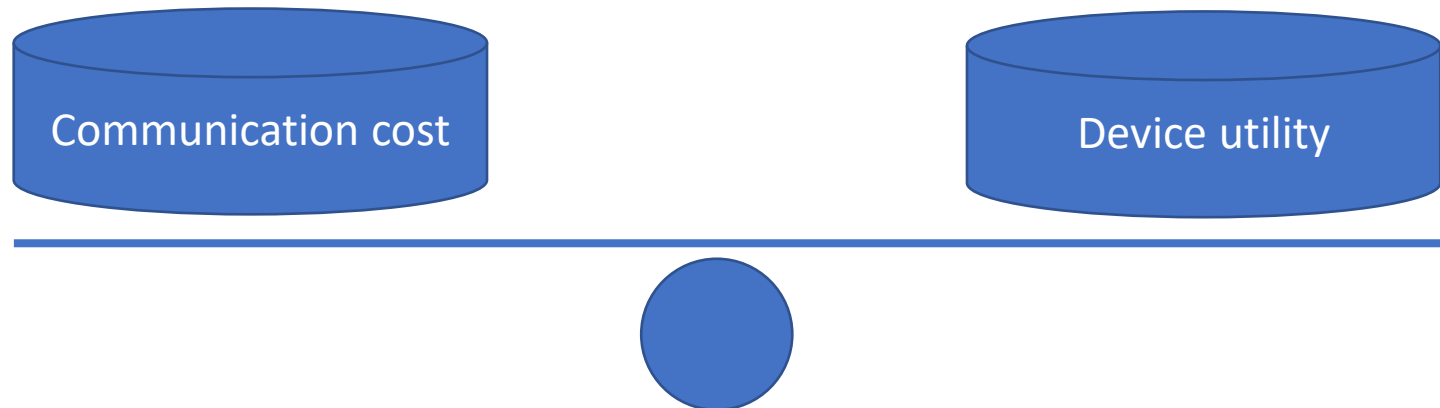
# What is distributed computations?



*Computing on multiple threads/devices/nodes in parallel*

We focus on training as it is the most computationally expensive part but doing testing or inference can also be done in distributed manner

Distributed computing is not always beneficial:



# Devices



- Three types of devices

- CPU

- General compute unit
    - 2-128 threads

- GPU

- Rendering unit
    - 1000-10000 threads

- TPU

- Specialized unit
    - 8-2048 threads



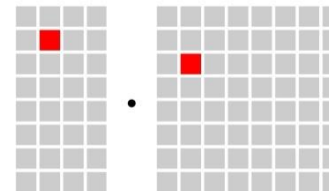
CPU



GPU

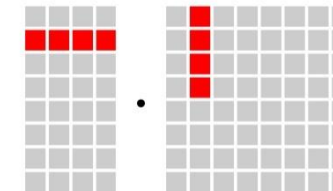


TPU

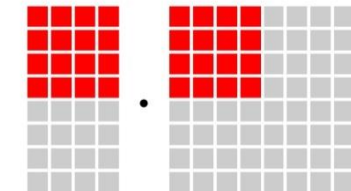


scalar

Compute Primitive



vector



tensor

Note that we are comparing apples to bananas!

# Memory

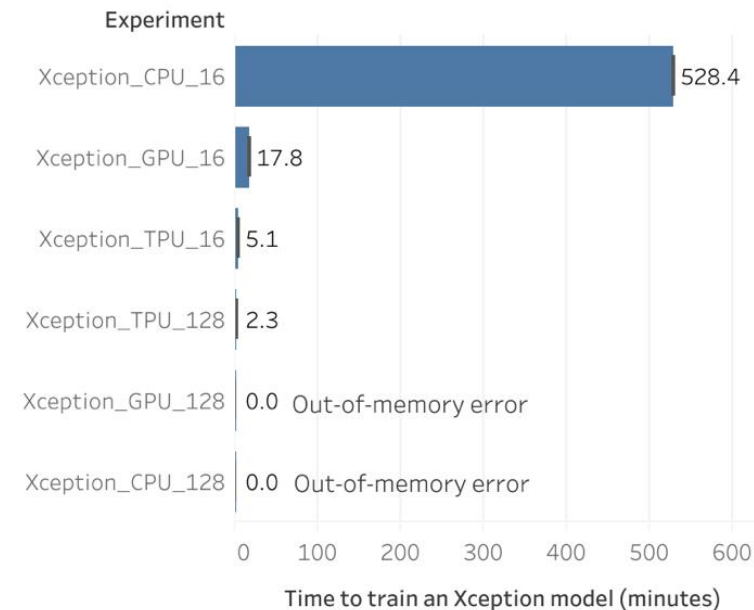


Equally important to what device you are using, is the amount of memory that you have available

With more memory

- Faster data transfer
- Higher data modality
- Larger models

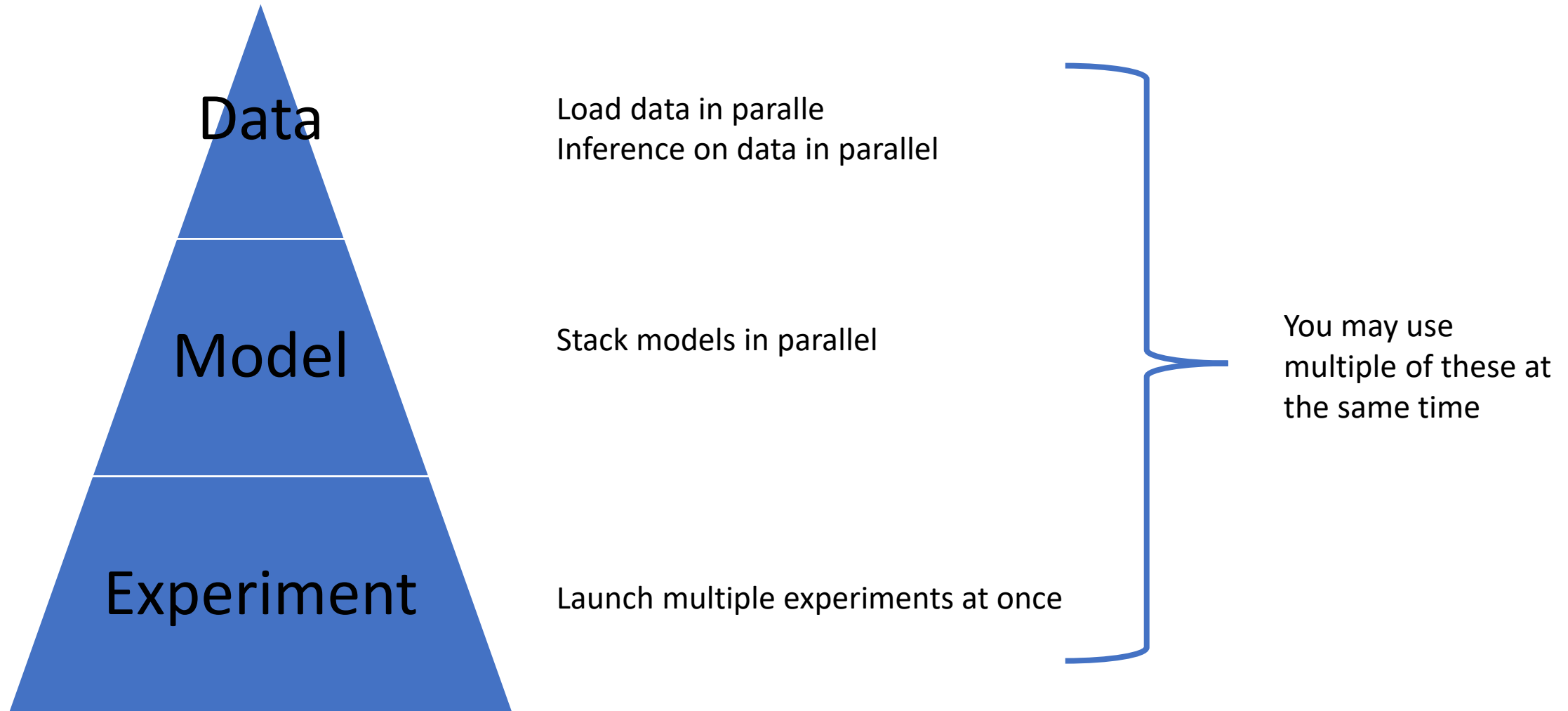
	CPU	GPU	TPU
Standard	32-64 GiB	12 GiB	64 GiB
Maximum	256 GiB	24 GiB	32 TiB



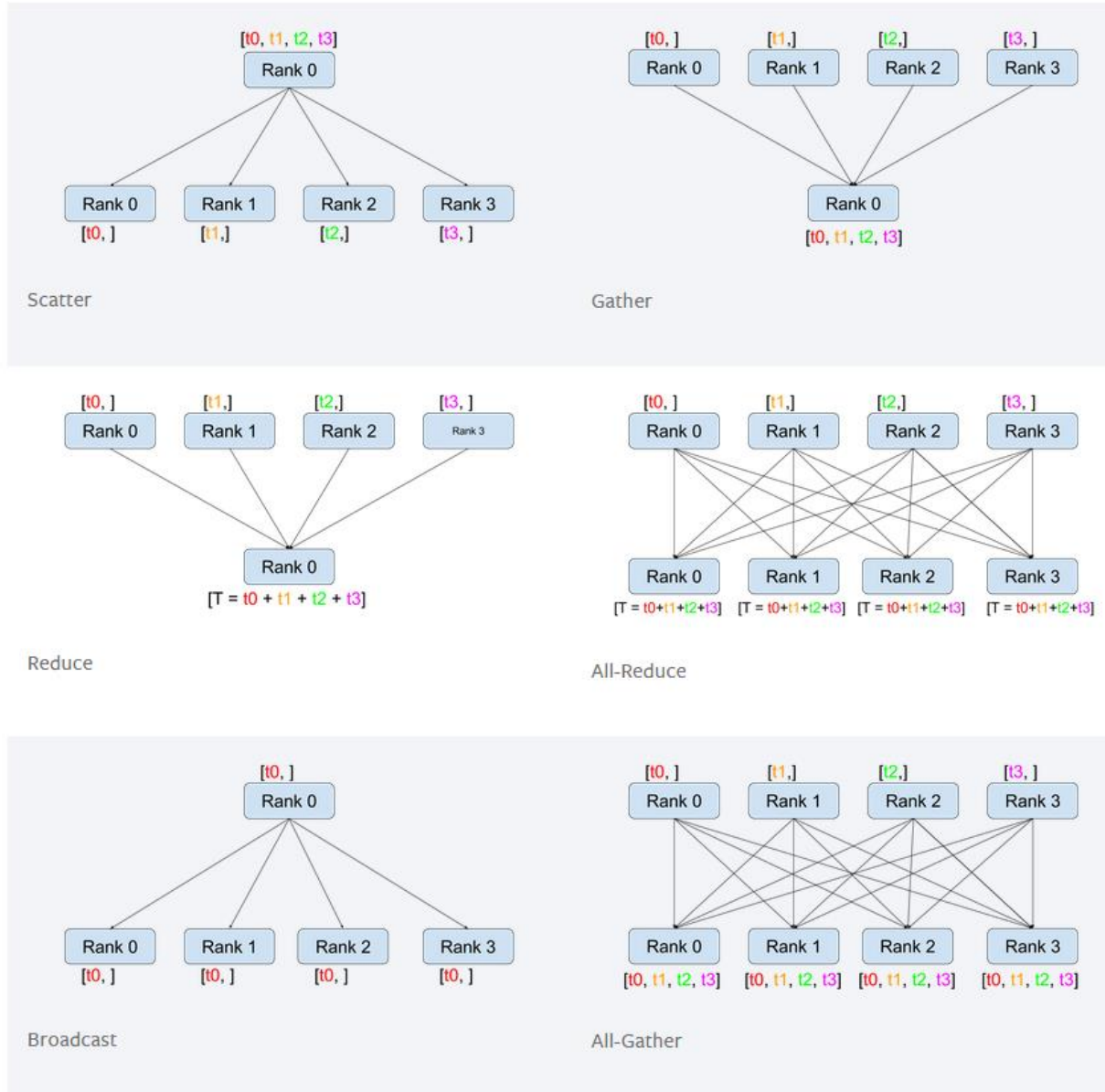
**Figure 3: CPUs vs GPUs vs TPUs for training an Xception model for 12 epochs.** Y-Axis labels indicate the choice of model, hardware, and batch size for each experiment. Increasing the batch size to 128 for TPUs resulted in an additional ~2x speedup.

<https://towardsdatascience.com/when-to-use-cpus-vs-gpus-vs-tpus-in-a-kaggle-competition-9af708a8c3eb>

# Many layers of distributed computations



# The six important communication types

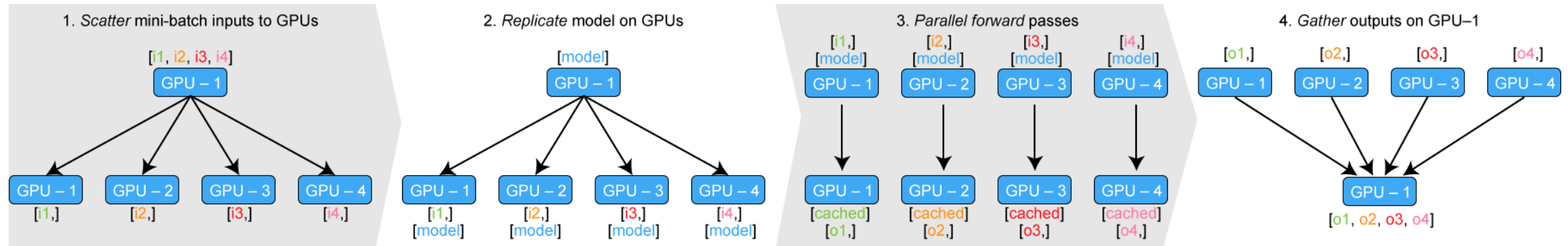


# Data Parallel: one process controls all

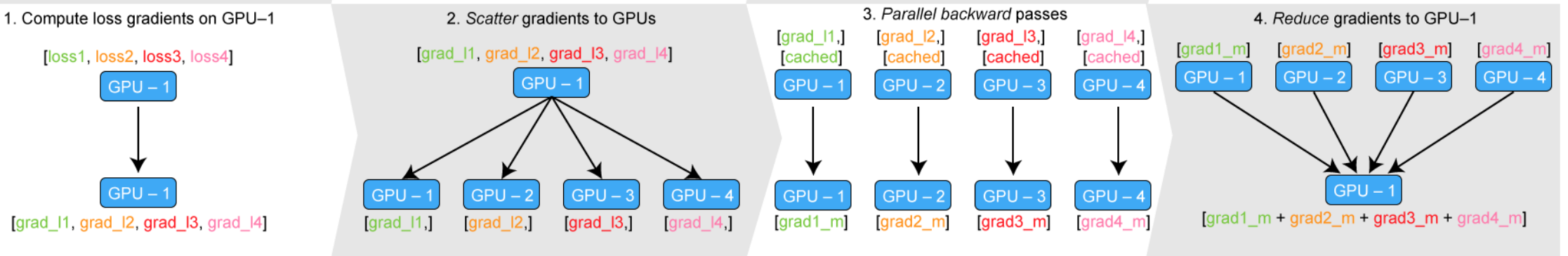


Simple as `parallel_model = torch.nn.DataParallel(model)`

Forward



Backward



Note that only GPU-1 parameters are updated, the replicas are destroyed after backward

# Distributed Data Parallel

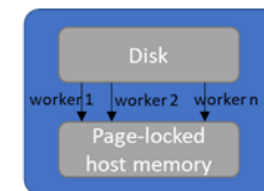
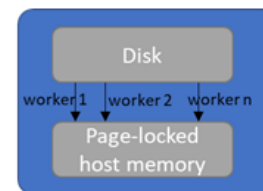
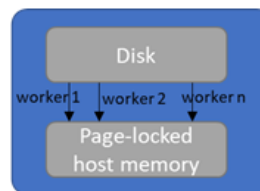
- In Distributed Data Parallel (DDP) all processes gets equal workload

## Distributed Data Parallel

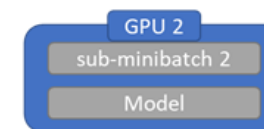
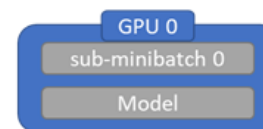
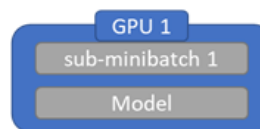
No master GPUs

Implemented in PyTorch  
`DistributedDataParallel`  
module

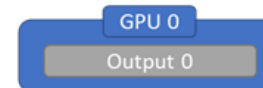
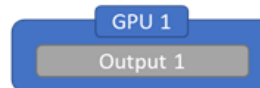
1. Load data from disk into page-locked memory on the host. Use multiple worker processes to parallelize data load. Distributed minibatch sampler ensures that each process loads non-overlapping data



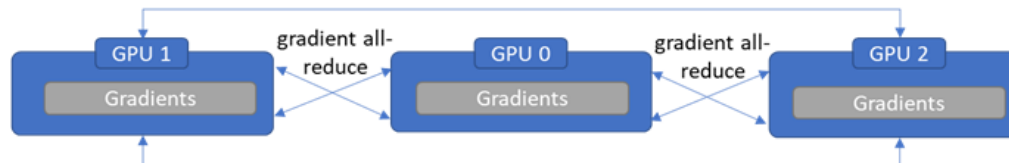
2. Transfer minibatch data from page-locked memory to each GPU concurrently. No data broadcast is needed. Each GPU has an identical copy of the model and no model broadcast is needed either



3. Run forward pass on each GPU, compute output



4. Compute loss, run backward pass to compute gradients. Perform gradient all-reduce in parallel with gradient computation



5. Update Model parameters. Because each GPU started with an identical copy of the model and gradients were all-reduced, weights updates on all GPUs are identical. Thus no model sync is required

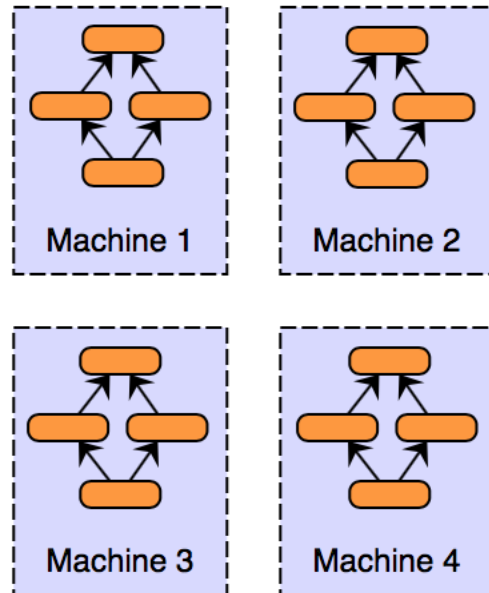




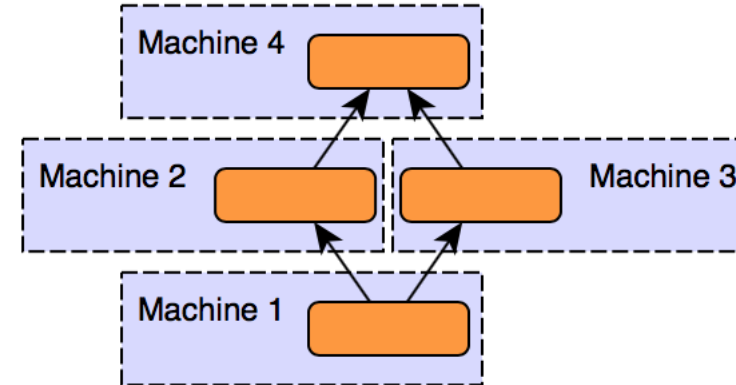
# Model parallelisme

- When your model is too big for one device

Data Parallelism



Model Parallelism



# How to do this in practise



- **DataParallel**
  - `parallel_model = torch.nn.DataParallel(model)`
- **Distributed Data Parallel**
  - Set a enviroment `MASTER_ADDR` and `MASTER_PORT`
  - Init a process group
  - `parallel_model = nn.parallel.DistributedDataParallel(model, device_ids=[gpu])`
  - Use `mp.spawn` to spawn multiple processes
  - ...
- **Model parallizeme**
  - A shit ton of `tensor.to('cuda:x')` calls

# Seperating engineering and research code



Getting code to run in parallel has somewhat become a research task!

*However maybe it should not be like that?*

## Research Code

```
l1 = nn.Linear(...)
l2 = nn.Linear(...)
decoder = Decoder()

x1 = l1(x)
x2 = l2(x2)
out = decoder(features, x)

loss = perceptual_loss(x1, x2, x) + CE(out, x)
```

## Engineering code

```
model.cuda(0)
x = x.cuda(0)

distributed = DistributedParallel(model)

with gpu_zero:
    download_data()

dist.barrier()
```

Spend time on research code and not engineering code!

# Why using a training framework



Spend time on research code and not engineering code

-> Why training frameworks exist!

- Reduce boilerplate = increase turn-around time
- Focus on what is important
- Reproduceability
- Shareability
- Consistency
- Scalability

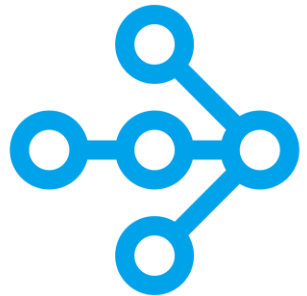
# Training Frameworks



Many frameworks exist for reducing boilerplate



Many frameworks for accelerating training



RAY



# Pytorch Lightning



Its just reorganised pytorch code!

Two core objects

- Lightning Module
  - Training, validation, test logic
  - Optimizer
- Trainer
  - The "rest"

## PyTorch

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        self.layer_1 = torch.nn.Linear(28 * 28, 128)  
        self.layer_2 = torch.nn.Linear(128, 10)  
  
    def forward(self, x):  
        x = x.view(x.size(0), -1)  
        x = self.layer_1(x)  
        x = F.relu(x)  
        x = self.layer_2(x)  
        return x  
  
# download data  
if global_rank == 0:  
    mnist_train = MNIST(os.getcwd(), train=True, download=True)  
    mnist_test = MNIST(os.getcwd(), train=False, download=True)  
  
dist.barrier()  
  
# transforms  
transform=transforms.Compose([transforms.ToTensor(),  
                               transforms.Normalize((0.1307,), (0.3081,))])  
mnist_train = MNIST(os.getcwd(), train=True, transform=transform)  
mnist_test = MNIST(os.getcwd(), train=False, transform=transform)  
  
# split dataset  
mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])  
mnist_test = MNIST(os.getcwd(), train=False, download=True)  
  
# build dataloaders  
mnist_train = DataLoader(mnist_train, batch_size=64)  
mnist_val = DataLoader(mnist_val, batch_size=64)  
mnist_test = DataLoader(mnist_test, batch_size=64)  
  
optimizer = torch.optim.Adam(pytorch_model.parameters(), lr=1e-3)  
  
def cross_entropy_loss(logits, labels):  
    return F.nll_loss(logits, labels)  
  
num_epochs = 1  
for epoch in range(num_epochs):  
    for train_batch in mnist_train:  
        x, y = train_batch  
        logits = pytorch_model(x)  
        loss = cross_entropy_loss(logits, y)  
        print('train loss: ', loss.item())  
        loss.backward()  
        optimizer.step()  
        optimizer.zero_grad()
```

# What you get for free



## Multi-GPU, multi-node

```
# 8 GPUs
# no code changes needed
trainer = Trainer(max_epochs=1, gpus=8)

# 256 GPUs
trainer = Trainer(max_epochs=1, gpus=8, num_nodes=32)
```

## TPU training

```
# no code changes needed
trainer = Trainer(tpu_cores=8)
```

## 16 bit precision

```
# no code changes needed
trainer = Trainer(precision=16)
```

## Experiment managers

```
from pytorch_lightning import loggers

logger = loggers.TensorBoardLogger('logs/')
logger = loggers.WandbLogger()
logger = loggers.CometLogger()
logger = loggers.MLFlowLogger()
logger = loggers.NeptuneLogger()
# ... and many more

trainer = Trainer(logger=logger)
```

# What you get for free



## Early stopping

```
es = EarlyStopping(monitor='val_loss')
trainer = Trainer(callbacks=[es])
```

## Model Checkpoint

```
checkpointing = ModelCheckpoint(monitor='val_loss')
trainer = Trainer(callbacks=[checkpointing])
```

## 40+ tricks and extensions

```
trainer = Trainer(
    max_epochs=10,
    auto_lr_find=True,
    gradient_clip_val=1.0,
    accumulate_grad_batches=10,
    max_steps=1000
    #... 40+ tricks and extensions
)
```

## Arbitrary functionality

```
# add arbitrary functionality
class MyNotifier(pl.Callback):

    def on_train_epoch_start(trainer, pl_module):
        slack.post('training started!')

notifier = MyNotifier()
trainer = Trainer(callbacks=[notifier])
```



