Introduction to Deep Learning – Some Practical Guidelines

Paul Rodriguez, PhD
(SDSC)

June 2025

Outline

- **Part II - Practical Guidelines for Running a Project:**

Choosing Hyperparameters – a bit of exploration and exploitation

Job workflow - make it efficient and easy to organize

CPUs vs GPUs

Parallelizing Models and Multinode/Multidevice Execution

Exercise/Demo, Multinode MNIST

Choosing Hyperparameters

- Hyperparameters are found by searching, not by the network algorithm

- Generally, hyperparameters related to:
  - architecture (layers, units, activation, filters, …)
  - algorithm (learning rate, optimizer, epochs, …)
  - efficient learning (batch size, normalization, initialization, …)

- Some options are determined by task:
  - loss function, CNN vs MLP, …

- Use what works, from related work or the latest recommendations,

Hyperparameters Search

- Can take a long time, hard to find global optimal
- Start with small data, short runs to get sense of range of good parameter values
- Easy but possibly time-consuming method:
  grid search over uniformly spaced values
- Do “exploration” then “exploitation”, ie search wide then search deep
  Keras Tuner functions can help with the wide search
  Raytune is similar tool for Pytorch

Hyperparameter Search Tool

- Several search strategies, such as:

Hyperband is like a tournament of hyperparameter configurations

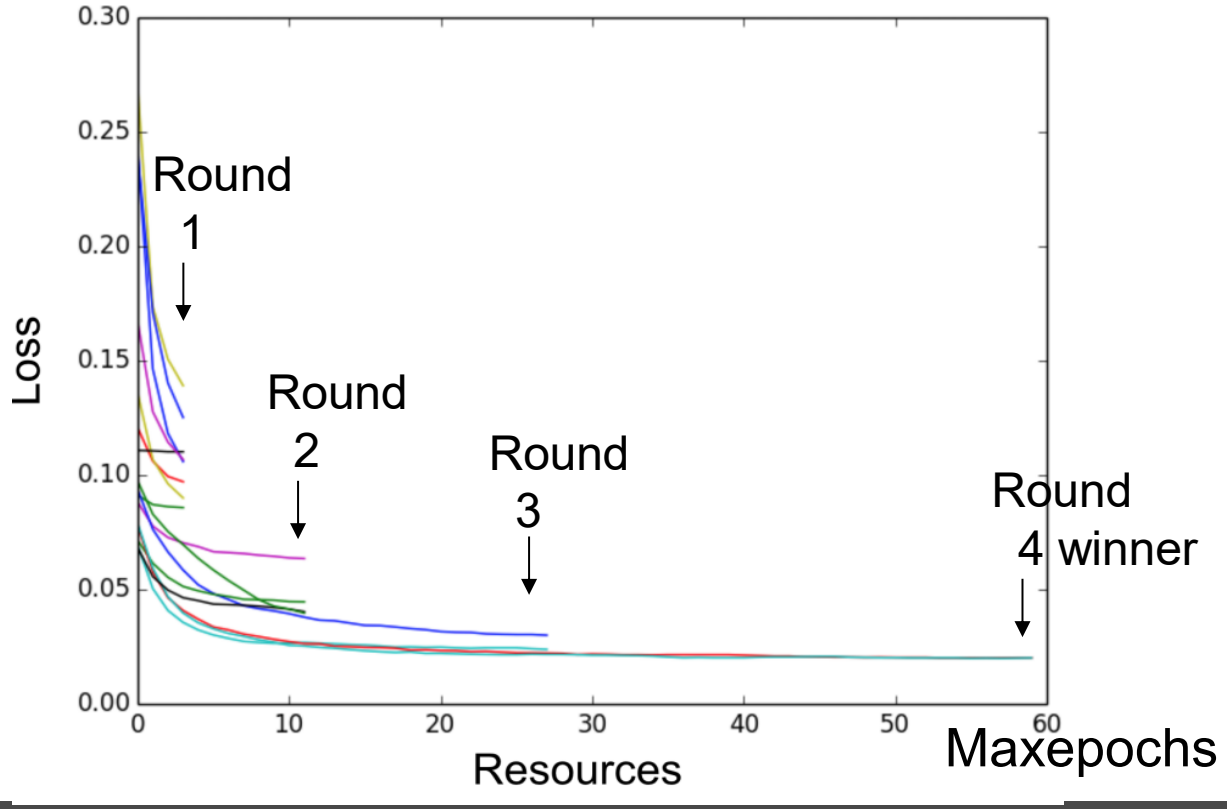
RandomSearch will search randomly through the space of configurations

Bayesian optimization is like a function approximation to pick out next configuration

- Typically, you usually wrap the model build function with arguments for hyperparameters

Hyperband Bracket

Each round runs several network configurations for small number of epochs
Several rounds with increasing epochs make up a bracket
Several brackets are run to end up with several possible overall winners.



Note, you could run a small grid search around hyperband winners to confirm performance

Keras Tuner code snippet

Set up function to make the model

```
def build_model(numfilters,activation_choice):   #<<------add code: if yo
                                                 #            list and change code t
    mymodel = keras.models.Sequential()
    mymodel.add(keras.layers.Convolution2D(numfilters,
                                           (3, 3),
                                           strides=1
```

Keras Tuner code snippet –

Set up function to make the model

Set up hyperparameter choices

```
def build_model(numfilters,activation_choice):   #<<------add code: if yo
                                                 #            list and change code t
    mymodel = keras.models.Sequential()
    mymodel.add(keras.layers.Convolution2D(numfilters,
                                           (3, 3),
                                           strides=1
```

Keras Tuner code snippet – put build-model inside a ‘wrapper’ function

Set up function to make the model

```
def build_model_hp(hp):
  hp_numfilters     = hp.Int('hpnumfilters',min_value=8,max_value=32,step=4)
  #your variable name          ^^^ the parameter name in the hp object
```

```
  return build_model(hp_numfilters,hp_Activation)   #<<---- dont forget to pass the new
```

Set up hyperparameter choices

```
def build_model(numfilters,activation_choice):   #<<------add code: if yo
                                                 #            list and change code t
    mymodel = keras.models.Sequential()
    mymodel.add(keras.layers.Convolution2D(numfilters,
                                           (3, 3),
                                           strides=1
```

Keras Tuner code snippet – put build-model inside a ‘wrapper’ function

Set up function to make the model

```
def build_model_hp(hp):
  hp_numfilters     = hp.Int('hpnumfilters',min_value=8,max_value=32,step=4)
  #your variable name          ^^^ the parameter name in the hp object
```

```
  return build_model(hp_numfilters,hp_Activation)   #<<---- dont forget to pass the new
```

Set up hyperparameter choices

```
def build_model(numfilters,activation_choice):   #<<------add code: if yo
                                                 #            list and change code t
    mymodel = keras.models.Sequential()
    mymodel.add(keras.layers.Convolution2D(numfilters,
                                           (3, 3),
                                           strides=1
```

Define ‘tuner’ object: uses the wrapper and model fit to search configurations

```
tuner = kt.Hyperband(build_model_hp,
                     objective  = 'val_accuracy',
                     max_epochs = num_max_epochs,
                     factor     = 3,
                     hyperband_iterations=10,
```

Workflow and Organizing Jobs

Job Level: What makes sense to include in each job?

Model Level: run & test model for each parameter configuration

Data Level: loop through cross validation datasets (if applicable)

- **Consider how long each a model runs for 1 configuration of hyperparameters for 1 dataset**
- **Organize jobs into reasonable chunks of work**
- **For large models consider model-checkpoints**
- **Tensorboard is available but needs to be secure (ask for details)**

Organizing Configurations – one way

Code snippet: using 'YAML' file to set up hyperparameter configuration

Create text file with "Parameter: Value" pairs

/home/p4rodrig/WKSHOPS...

```
filter_size: 16
num_filters: 16
```

Line: 1/2 Encoding: 1252 (ANSI - La

Read file as python dictionary

```python
import yaml

with open("./modelrun_args.yaml", "r") as f:
    my_yaml=yaml.safe_load(f)  #this returns a python dictionary

filter_size=my_yaml.get("filter_size")
num_filters=my_yaml.get("num_filters")
print('arguments, filter size:',filter_size,' num filters',num_filters)
```

note on using GPU

- **GPU node has multiple GPU devices**
- **By default tensforflow will run on 0th gpu device if GPU is available, otherwise it will use all CPU cores**

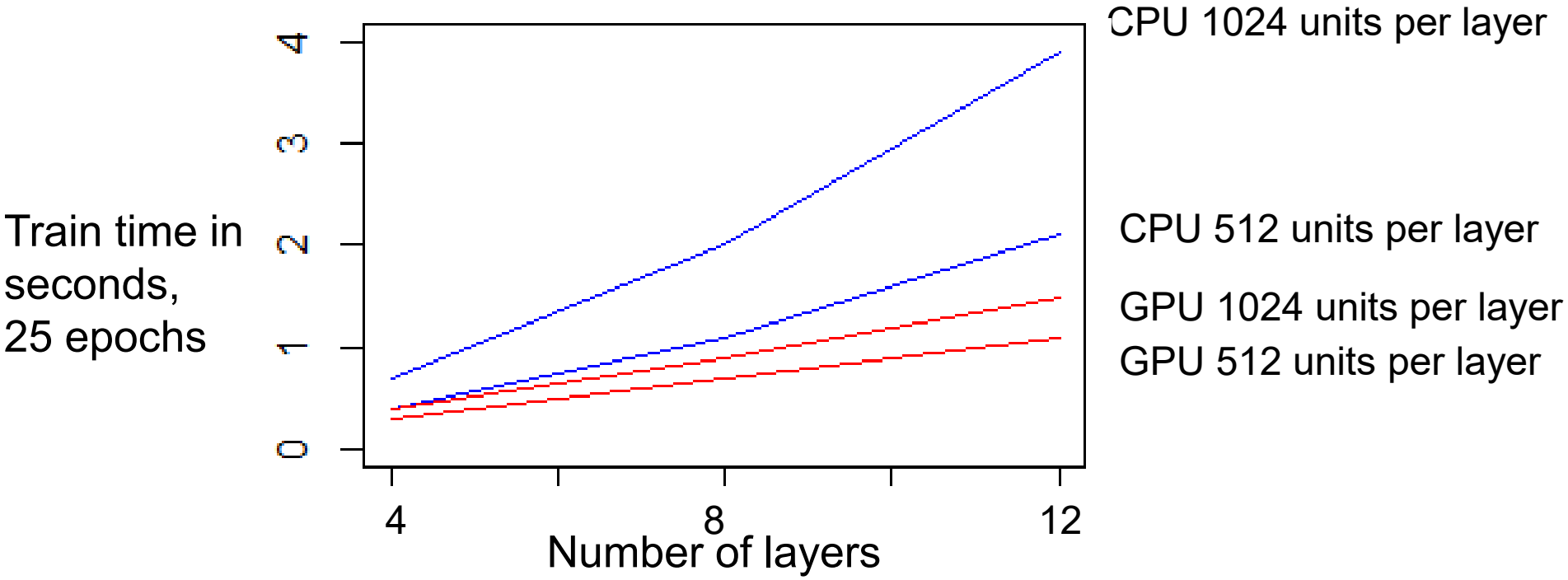
Code snippet to check for GPU devices

/home/p4rodrig/WKSHOPS/EXP-05192022/Intro_mnist_cnn2_forbatch.py - p4rodrig@login.expanse.sdsc.edu - Editor - W

Encoding Color

```
physical_devices = tf.config.list_physical_devices('GPU')
n_gpus           = len(physical_devices)
print("Info,config,Num GPUs:", n_gpus)
if tf.test.gpu_device_name():
  print('Info, test,Default GPU Device:{}'.format(tf.test.gpu_device_name()))
```

GPU shared (V100) vs CPU (128 cores)
For MLP with Dense Layers, 80000x200 data matrix



GPUs faster, but you might have to wait more in job queue; also some memory limits compared to CPU, may need to use smaller batch size

Scaling in Deep Learning

- Two Goals

1 Speed Up Learning - parallelizing data and/or model

2 Optimize Memory - as models scale up they take up too much memory
e.g. V100s have 32Gb limit and 8B float32 parameters would fill that

Parallelism strategies

- Data Parallelism: partition data and copy the model across devices,
  (this is probably easiest thing to do, least programming)

- Pipeline Parallelism: split up the model so that layers are on different devices, ie inter-layer partitions (you organize layers)

- Tensor Parallelism: intra-layer partitions (model has to support it)

Parallel DL models with multiple nodes/devices

- **Data Parallel:**

1. Launch your script on each device on each node

2. Split up data

3. Each device trains a copy of the model with a part of the data

4. Aggregate parameter updates across devices/nodes

Parallel DL models with multiple nodes/devices

- **Data Parallel:**

1. Launch your script on each device on each node
2. Split up data
3. Each device trains a copy of the model with a part of the data
4. Aggregate parameter updates across devices/nodes

- **Main functions:**

1. *mpirun* command (also *torchrun* exists but I haven't tested it much)
2. *Pytorch data loader will parallelize sampling*
3. *Pytorch DDP (Distributed Data Parallel) will wrap a model*
4. *DDP wil use the Pytorch 'init_backend' function*

For example, single node, single device execution

In slurm batch script:

singularity → python

Your python script

Load data

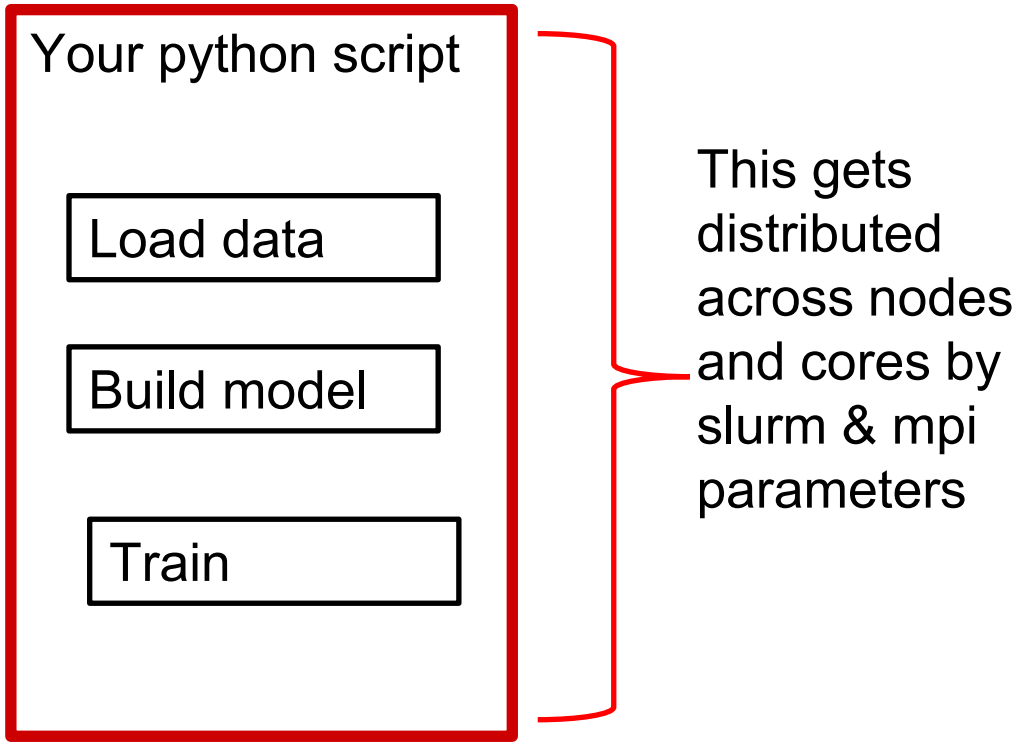
Build model

Train

Multinode, mpi launches instances

In slurm batch script:

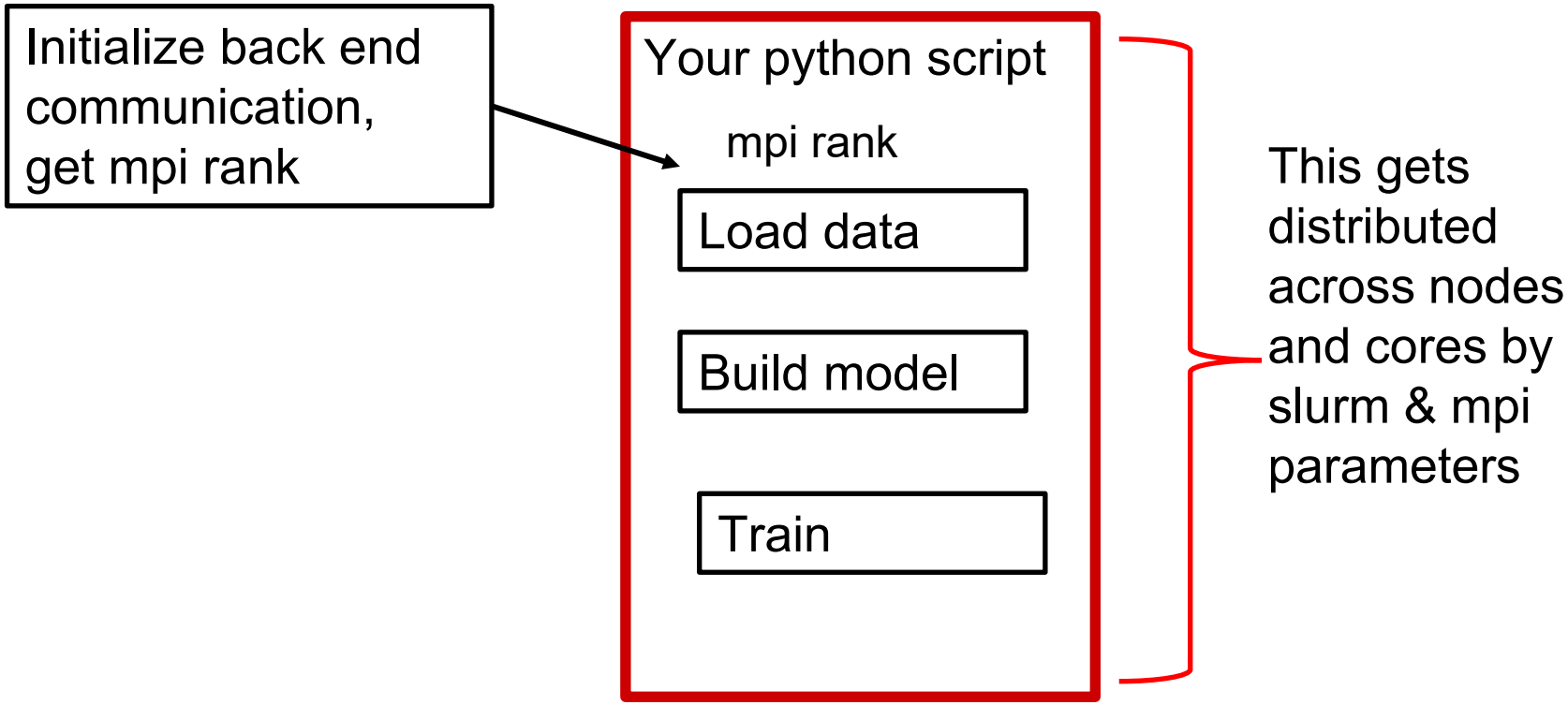
mpirun –n **number of tasks** singularity → python



Multinode, mpi launches instances

In slurm batch script:

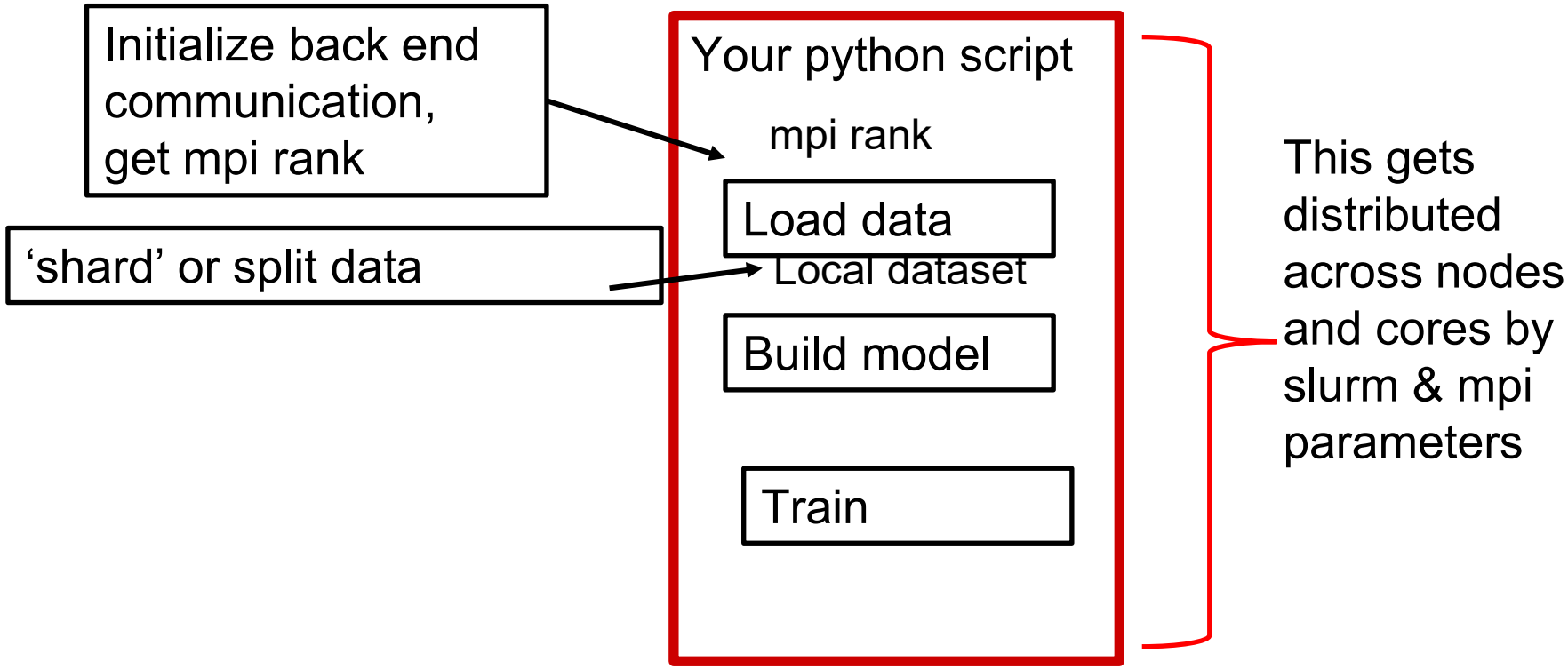
mpirun –n **number of tasks** singularity → python



Multinode, mpi launches instances

In slurm batch script:

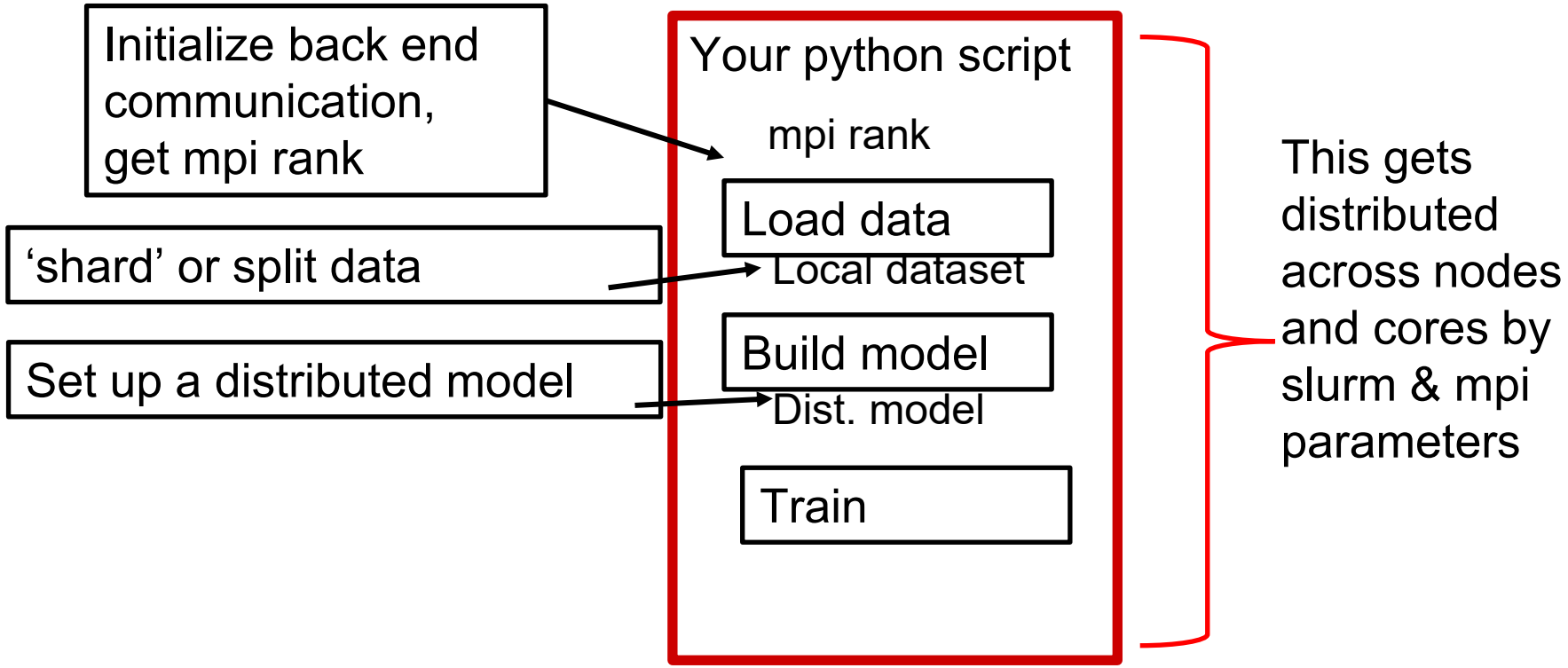
mpirun –n **number of tasks** singularity → python



Multinode, mpi launches instances

In slurm batch script:

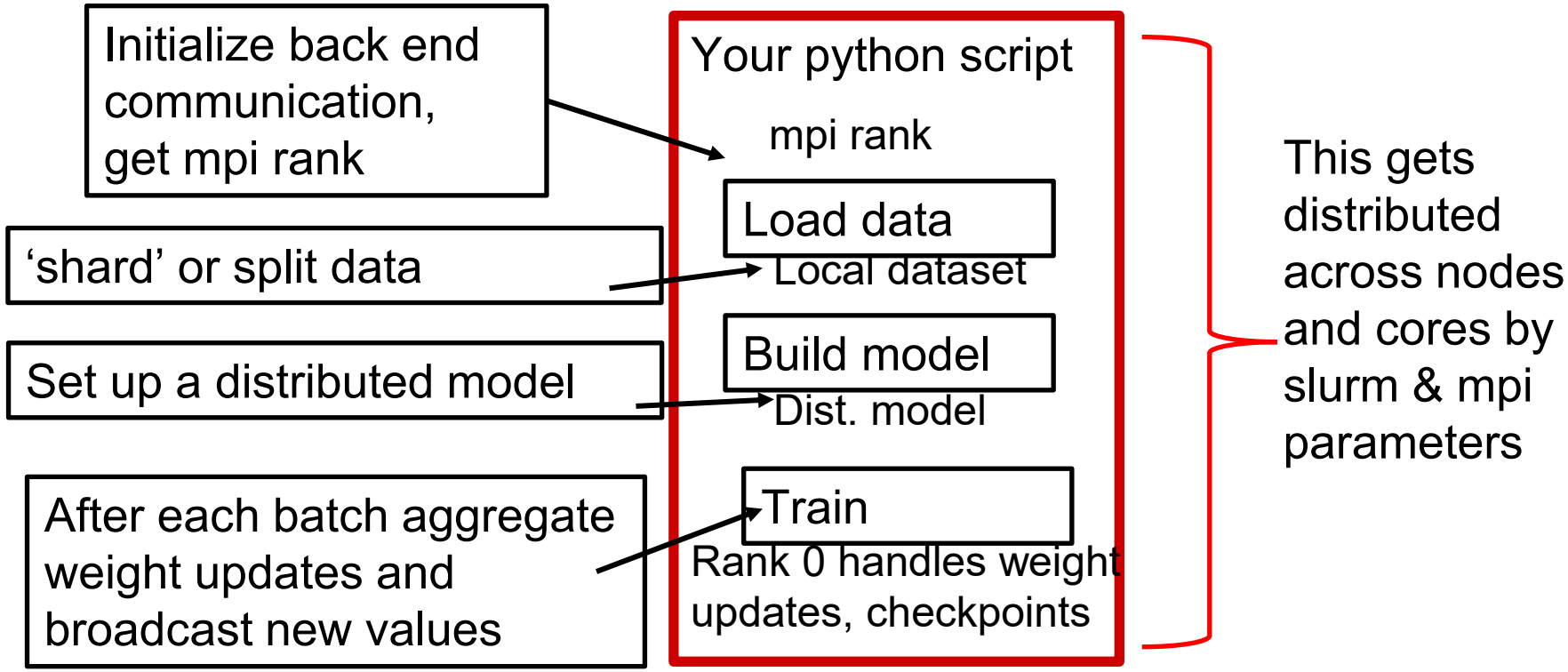
mpirun –n **number of tasks** singularity → python



Multinode, mpi launches instances

In slurm batch script:

mpirun –n **number of tasks** singularity → python



mpi launches one instance per processor

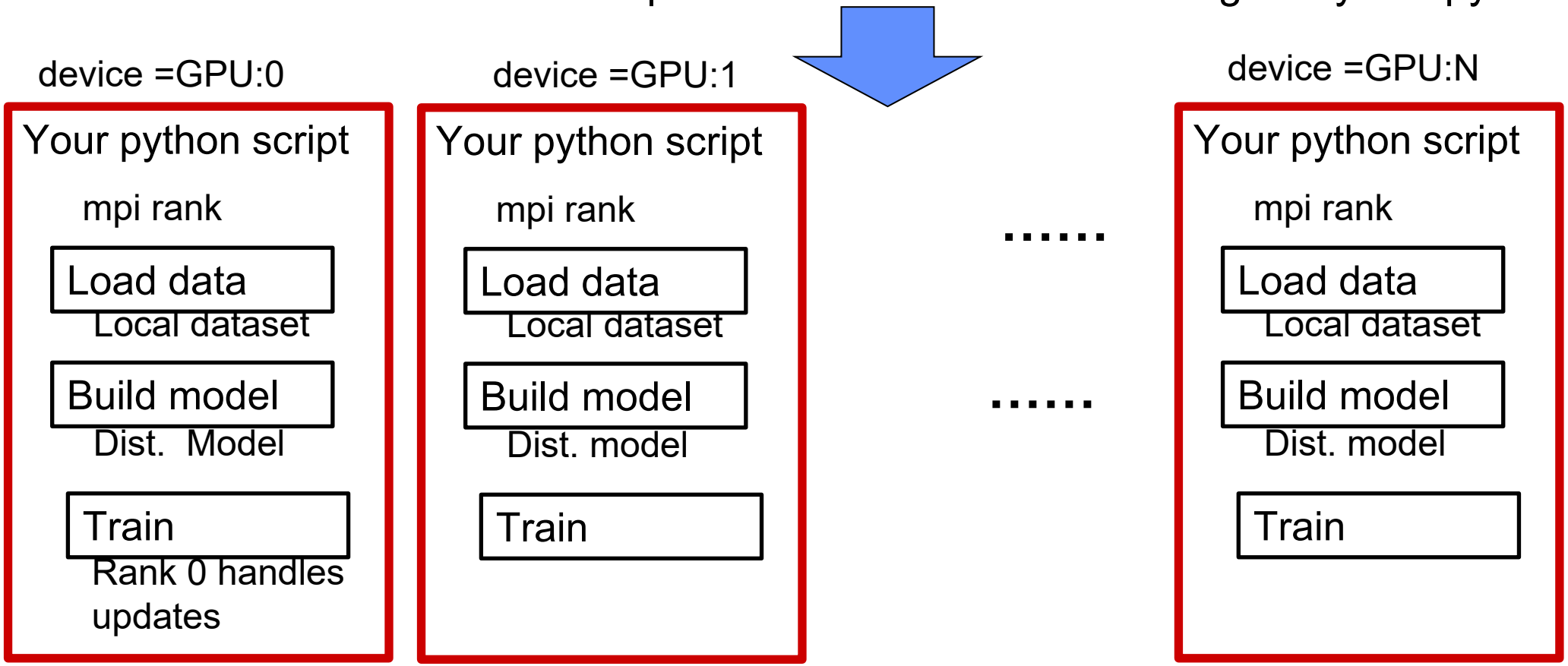
In slurm batch script:

mpirun –n **number of tasks** singularity → python

mpi launches one instance per processor

In slurm batch script:

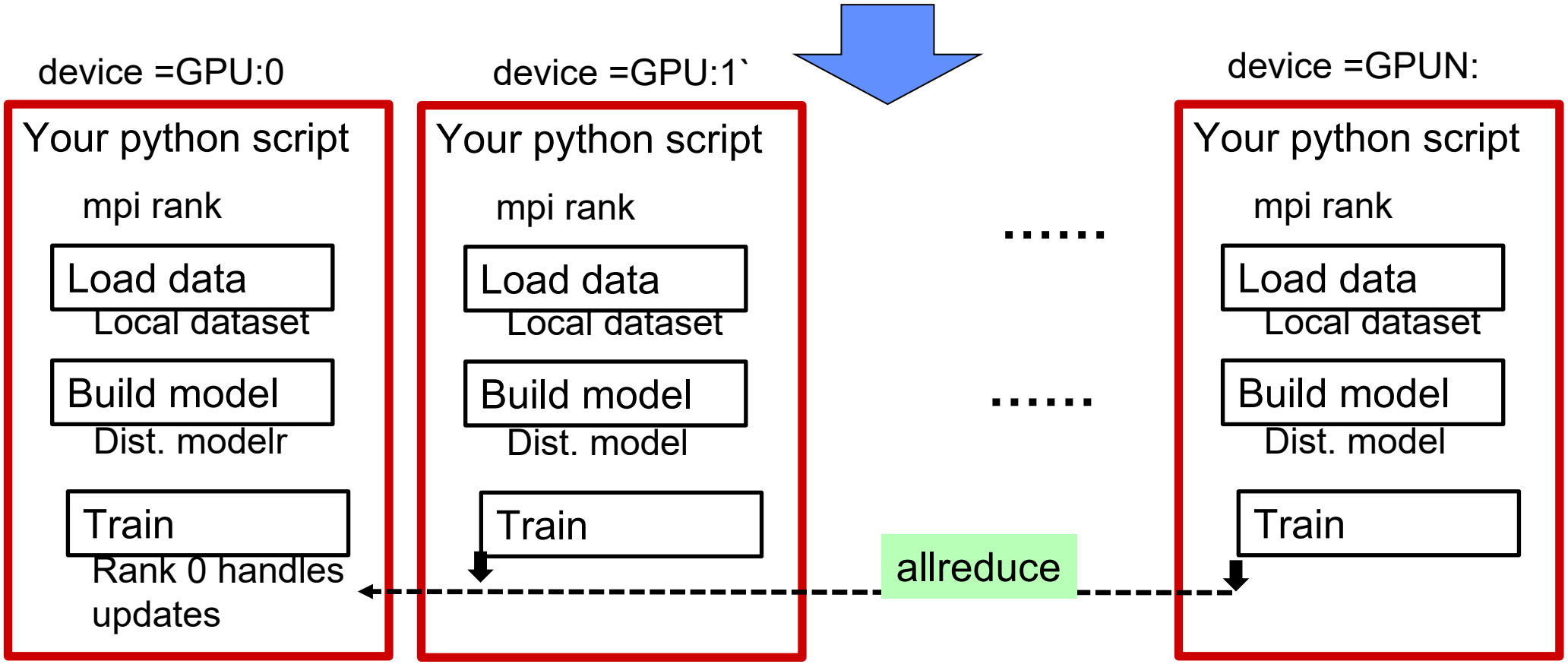
mpirun –n **number of tasks** singularity → python



For each batch: DDP will aggregate & share weights updates

In slurm batch script:

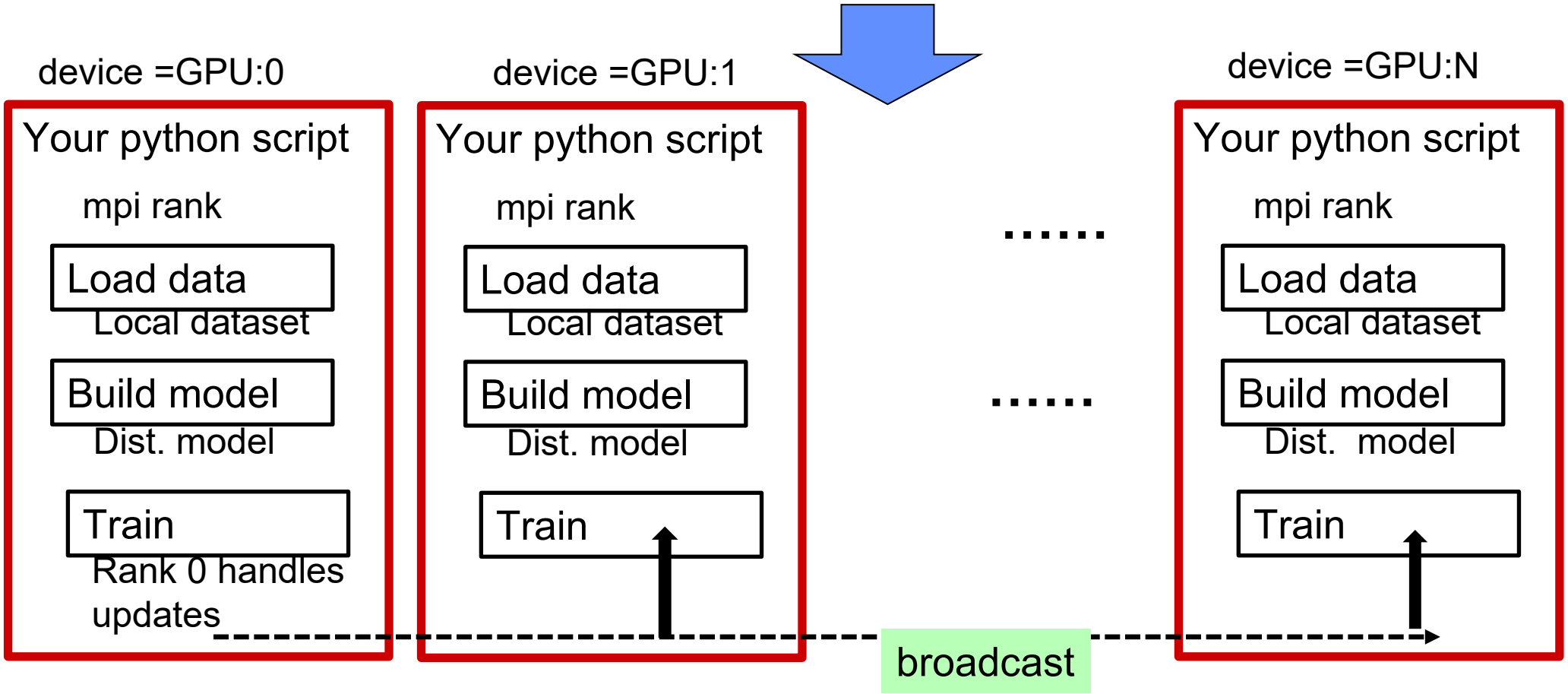
mpirun –n **number of tasks** singularity → python



For each batch: DDP will aggregate & share weights updates

In slurm batch script:

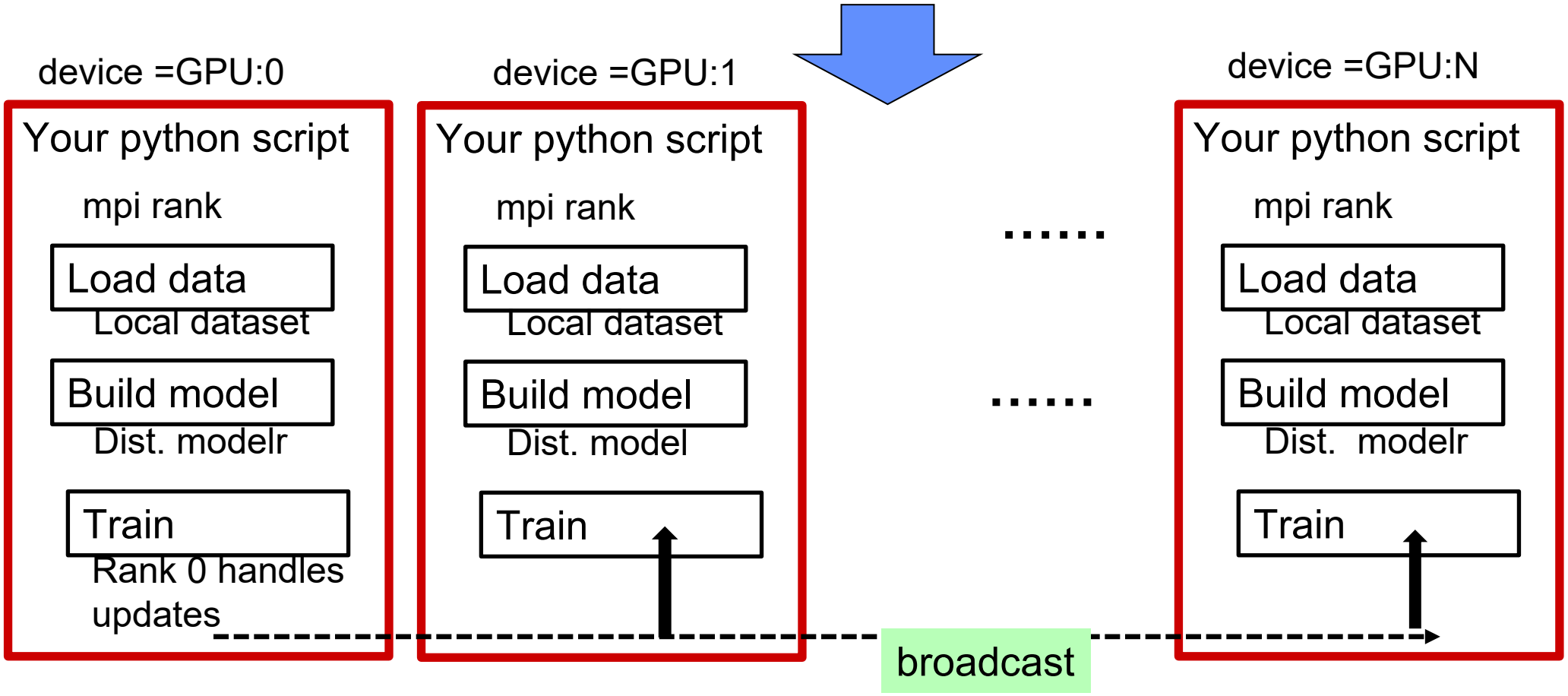
mpirun –n **number of tasks** singularity → python



For each batch: DDP will aggregate & share weights updates

In slurm batch script:

mpirun –n **number of tasks** singularity → python



Bigger batch size helps, but it uses more memory

Code snippets – Pytorch functions

Initialize back end communication, get mpi rank

```
world_size = int(os.environ['OMPI_COMM_WORLD_SIZE'])
rank       = int(os.environ['OMPI_COMM_WORLD_RANK'])
local_rank = int(os.environ['OMPI_COMM_WORLD_LOCAL_RANK’])
device = torch.cuda.set_device(local_rank)
torch.distributed.init_process_group('nccl',rank=rank,world_size=world_size
```

‘shard’ or split data

```
train_sampler = torch.utils.data.distributed.DistributedSampler(dataset1)
train_loader =torch.utils.data.DataLoader(dataset1,
                    batch_size =batch_size,
                    sampler   =train_sampler, …)
```

Set up a distributed model

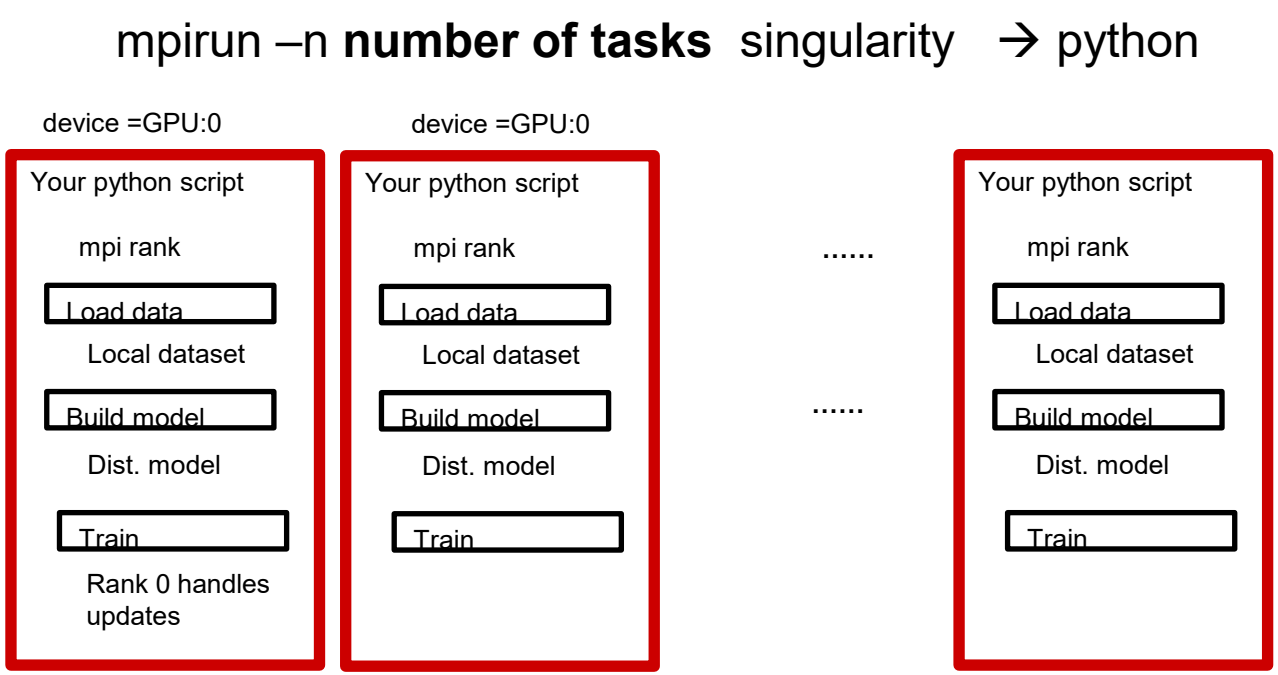
```
model = Net().to(device)
model = torch.nn.parallel.DistributedDataParallel()   #with default args
```

After each batch aggregate weight updates and broadcast new values

Exercise, multinode MNIST execution

- **Goal: Get familiar with Pytorch coding for multinode execution on Expanse**
- **Goal: Get familiar with slurm batch script multinode parameters**
- **Let's login to a terminal window**

(see next page for tasks to try)



SLURM batch script highlights

- Set up job resources

```
#!/usr/bin/env bash
#SBATCH --job-name=pyt-cpu
#SBATCH --account=gue998
#SBATCH --partition=compute
#SBATCH --nodes=2                #try 1 or 2
#SBATCH --ntasks-per-node=8   #try 8 or 16 or 32 etc..
….
```

- Set up 'master' ip address

```
#set up ip addresses for communication
declare -xr MASTER_ADDR=$(mpirun --allow-run-as-root -n 1 hostname -i
declare -xr MASTER_PORT=${MASTER_PORT:-15566};
```

- Use mpirun command

2*8=16

```
#use -n num-of-nodes * num-per-node
mpirun -n 16 -npernode 8  singularity exec --bind /expanse,/scratch
/cm/shared/apps/containers/singularity/pytorch/pytorch-latest.sif pyth
stdout_mnist_multinode_cpu.txt
```

Your task:

- run sbatch command for the slurm script

$ sbatch run-pyt-main-cpu2.sb

- ssh into node to see execution

$ squeue –u $USER (this will show your nodes)

$ ssh exp-##-## (this will connect to a node)

[exp-##-##] $ top –u $USER (this will show what's running)

```
[etrain107@login02 MultiNode_v2]$ squeue -u etrain107
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(RE
ASON)
          40567303   compute  pyt-cpu etrain10  R       0:13      2 exp-5-[34-3
5]
```

```
etrain107@exp-5-34:~
top - 22:44:29 up 21 days,  5:35,  1 user,  load average: 419.73, 116.12, 55.52
Tasks: 1811 total,   9 running, 1801 sleeping,   0 stopped,   1 zombie
%Cpu(s): 99.6 us,  0.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.3 hi,  0.0 si,  0.0 st
MiB Mem : 257485.8 total, 239778.3 free,  11741.2 used,   5966.2 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used. 243939.4 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
1923907 etrain1+  20   0   13.4g 671832 216164 R  1597   0.3  10:03.77 python3
1923910 etrain1+  20   0   13.3g 631752 216164 R  1597   0.2   9:47.36 python3
1923911 etrain1+  20   0   13.4g 671496 216340 R  1597   0.3  10:01.92 python3
1923908 etrain1+  20   0   13.3g 634908 216164 R  1597   0.2   9:24.16 python3
1923912 etrain1+  20   0   13.4g 683004 218276 R  1597   0.3  10:24.23 python3
1923909 etrain1+  20   0   13.4g 734788 216164 R  1596   0.3   9:59.80 python3
1923913 etrain1+  20   0   13.4g 688748 216164 R  1596   0.3   9:51.26 python3
1923914 etrain1+  20   0   13.4g 707712 216164 R  1595   0.3  10:12.39 python3
1929064 etrain1+  20   0   60420   6336   3528 R   1.0   0.0   0:00.13 top
1923604 etrain1+  20   0   89740   9844   8236 S   0.0   0.0   0:00.06 systemd
1923609 etrain1+  20   0  158092  10728      0 S   0.0   0.0   0:00.00 (sd-pa+
1923654 etrain1+  20   0   12984   3360   2832 S   0.0   0.0   0:00.01 bash
1923734 etrain1+  20   0  221900  12800   6316 S   0.0   0.0   0:00.09 mpirun
1923738 etrain1+  20   0  253768   7484   3716 S   0.0   0.0   0:00.00 srun
1923740 etrain1+  20   0   44548    888      0 S   0.0   0.0   0:00.00 srun
```

- review stdout output file

$ grep 'training time' stdout_mnist_...txt

- homework: find trade offs in speed up vs communication

```
etrain107@login02 MultiNode_v2]$ grep 'training time' stdout_
_8.txt
NFO rank: 1  training time: 27.54175
NFO rank: 14  training time: 27.54182
NFO rank: 2  training time: 27.54186
```

- **pause**

Deepspeed Memory Optimization

- Optimizers like Adam use a lot of memory **b/c it tracks momentum and variances of gradients for *each* weight parameter update:**

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon}$$

Deepspeed Memory Optimization

- Optimizers like Adam use a lot of memory **b/c it tracks momentum and variances of gradients for each weight parameter update:**

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon}$$

- **"Zero Redundancy" (ZeRO) optimizes memory storage by partitioning these terms to different devices and then gathering them only when needed**

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models 2020, Rajbhandari et al, Microsoft

Deepspeed: 3 stages of incrementally more partitioning

1. Optimizer state partitioning (ZeRO stage 1)
2. Gradient partitioning (ZeRO stage 2)
3. Parameter (weights) partitioning (ZeRO stage 3)

Deepspeed: 3 stages of incrementally more partitioning

1. Optimizer state partitioning (ZeRO stage 1)
2. Gradient partitioning (ZeRO stage 2)
3. Parameter (weights) partitioning (ZeRO stage 3)

All options go in a json file and passed as argument

```
--deepspeed_config
        ds_config.json
```

```
1-20:mnist_trialspipe$ more ds_config.json

   "train_batch_size":16,
   "bf16":  { "enabled": true },
   "fp16":  { "enabled": false},
   "gradient_clipping": 1.0,
   "zero_optimization":  { "stage": 0 },
   "zero_allow_untested_optimizer": true
```

Deepspeed code snippets

Deepspeed initialization creates a "**model_engine**" to wrap the model

```
model_engine, opt, _, _ = deepspeed.initialize(model=model,
model_parameters=model_params, args=args)
```

Deepspeed code snippets

Deepspeed initialization creates a "**model_engine**" to wrap the model

```
model_engine, opt, _, _ = deepspeed.initialize(model=model,
model_parameters=model_params, args=args)
```

The training loop now uses **model_engine** for forward,backward processing

```
output = model_engine(data)
    loss = loss_function(output, target)
    model_engine.backward()
    model_engine.step()
   htcore.mark_step()
```

Launch program instances with mpirun, or deepspeed launcher

- **End**