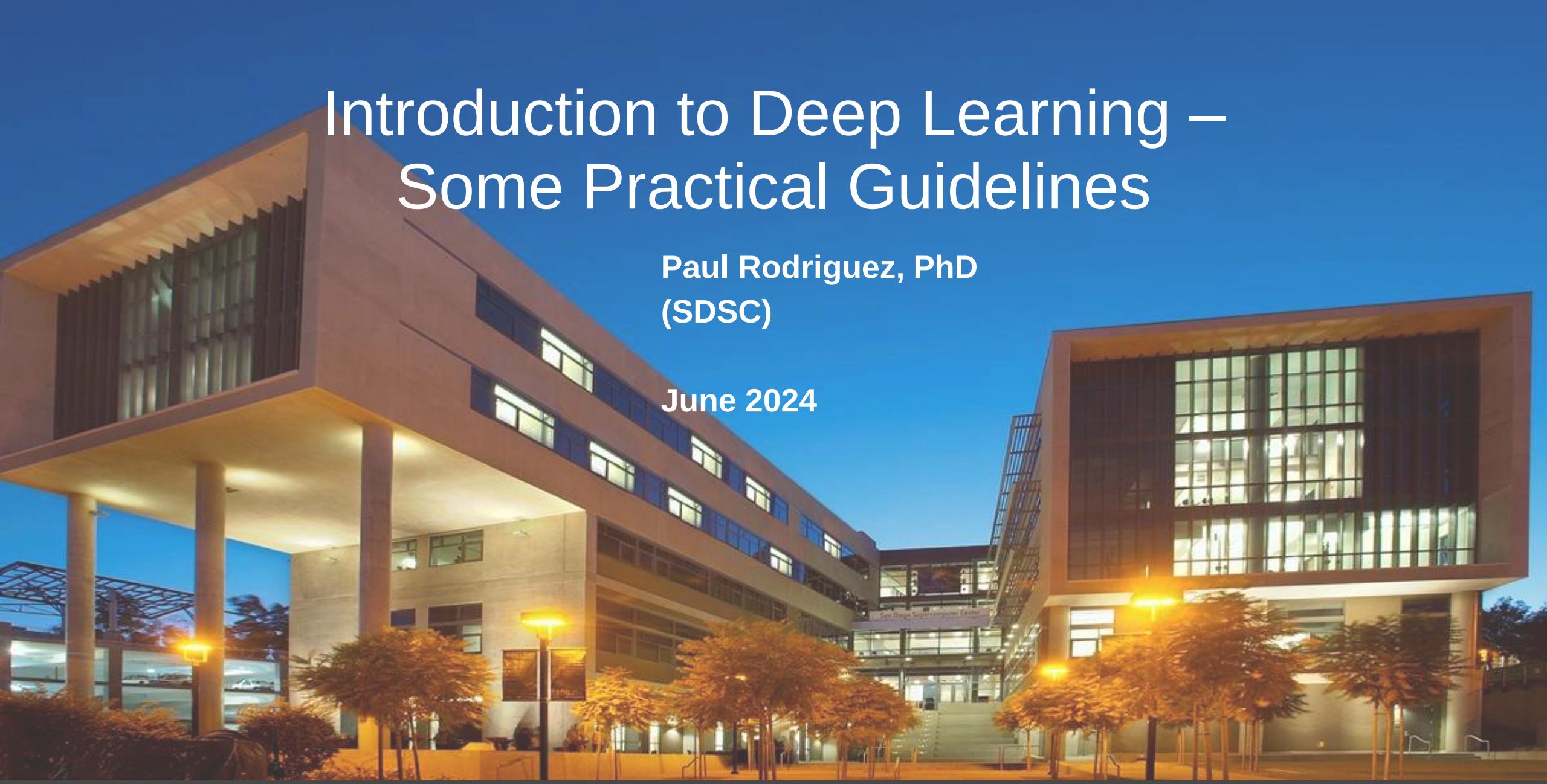
Introduction to Deep Learning – Some Practical Guidelines

Paul Rodriguez, PhD
(SDSC)

June 2024



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 Execution, with an exercise

Exercise, 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)

Keras Hyperparameter Search Tool

- Keras Hypertuner class implements several search strategies:

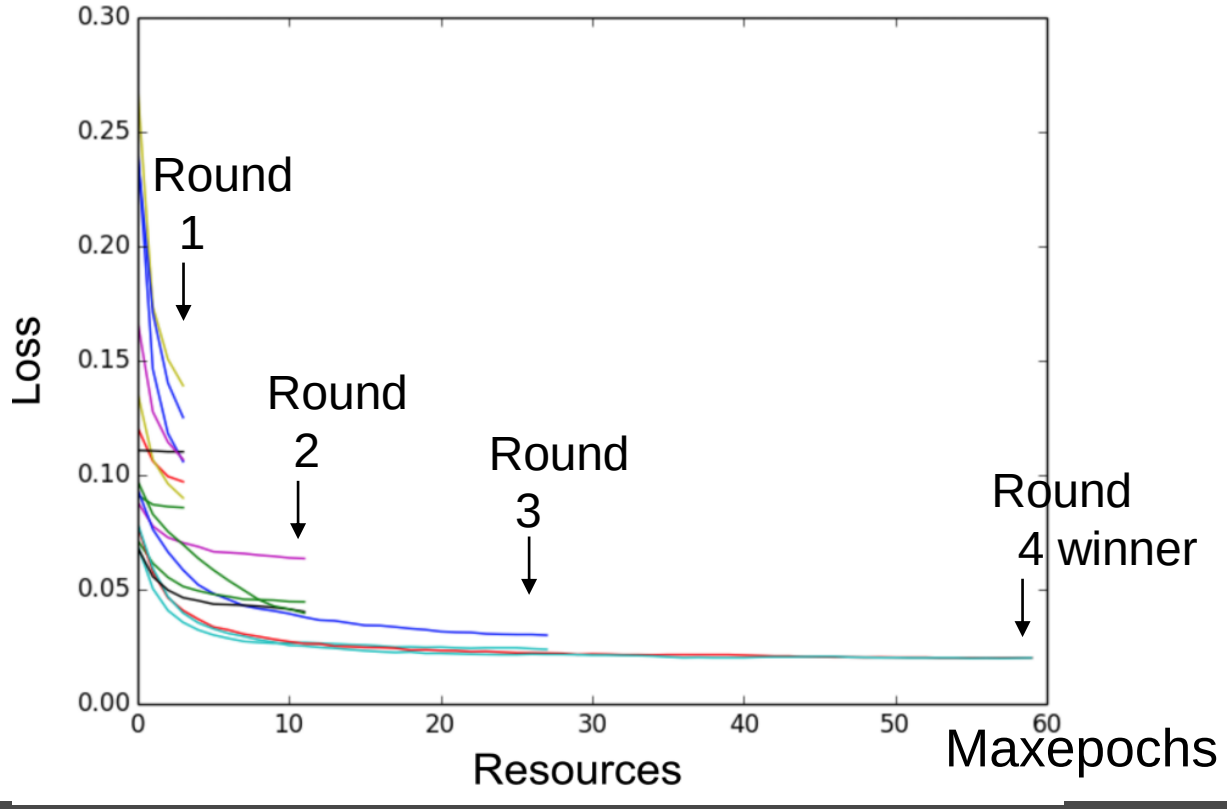
Hyperband is like a tournament of hyperparameter configurations

RandomSearch will search randomly through the space of configurations

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

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**

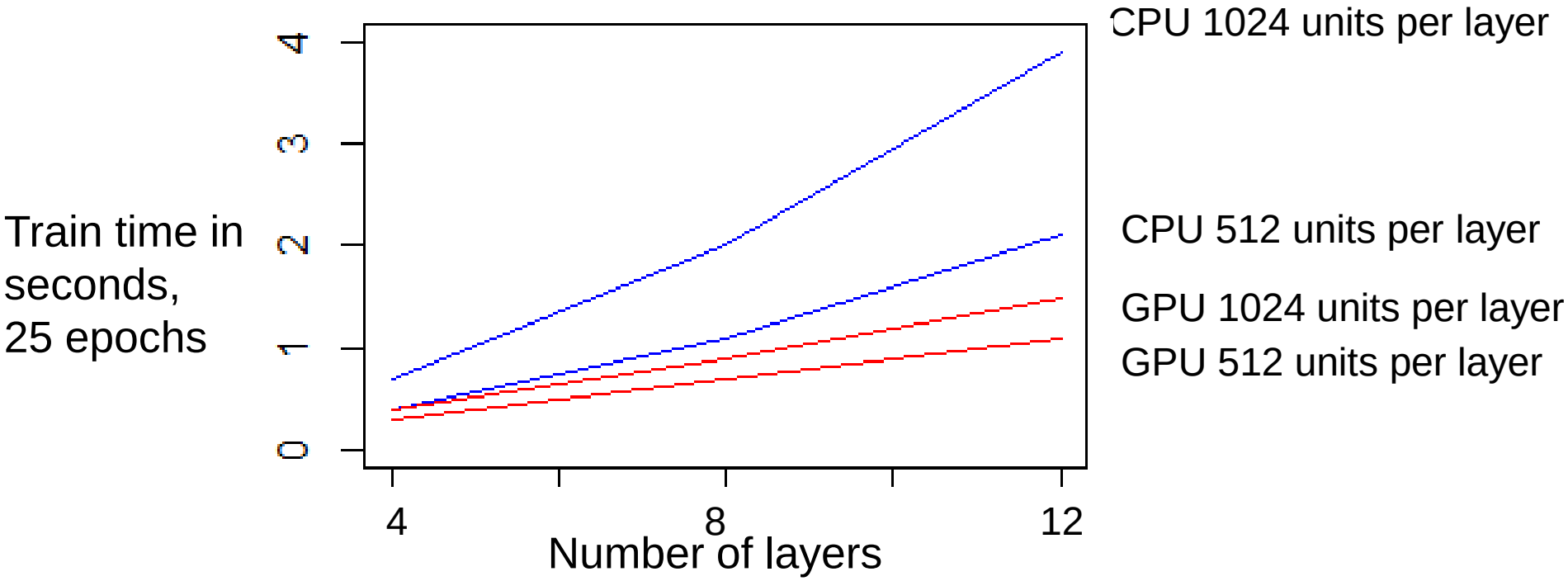
Code snippet to check for GPU devices

/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

Parallelism in Deep Learning

- Two Goals

1 Speed Up Learning - as data scales up training takes longer

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: split up weight matrix so that columns are on different devices, ie intra-layer partitions (model has to support it)

- Using mixed precision can reduce memory footprint at a cost of accuracy
- 'DEEPSPEED' can optimize memory usage at a cost of communication

Parallel DL models with multiple nodes/devices

- **Data Parallel:**
    1. Split up data
    2. Launch your script on each device
    3. Each device trains a copy of the model with a part of the data
    4. Aggregate parameter updates

Parallel DL models with multiple nodes/devices

- **Data Parallel:**
    1. Split up data
    2. Launch your script on each device
    3. Each device trains a copy of the model with a part of the data
    4. Aggregate parameter updates

- **Main tools: Keras/Tensorflow 'strategy' or use Horovod MPI wrappers**

Keras/Tensorflow strategy single GPU node

Set up a 'mirror' strategy

```
mirrored_strategy = tf.distribute.MirroredStrategy(["GPU:0", "GPU:1",  "GPU:2",  "GPU:3"])
```

You also need the strategy scope around the model definition so that it can make copies

```
if (n_gpus>0):
    with mirrored_strategy.scope():
      multi_dev_model=build_model()
```

Then train as normal (good to use batch size multiple of 32)

Keras/Tensorflow strategy multiple GPU node

Keras also has a ‘multiworker’ strategy but it requires setting up config files with IP addresses

But, on HPC systems resources are shared so IP addresses are dynamic

Thus, it is better to use Horovod with MPI and slurm batch job

For example, single node, single device execution

In slurm batch script:

singularity → python

Your python script

Load data

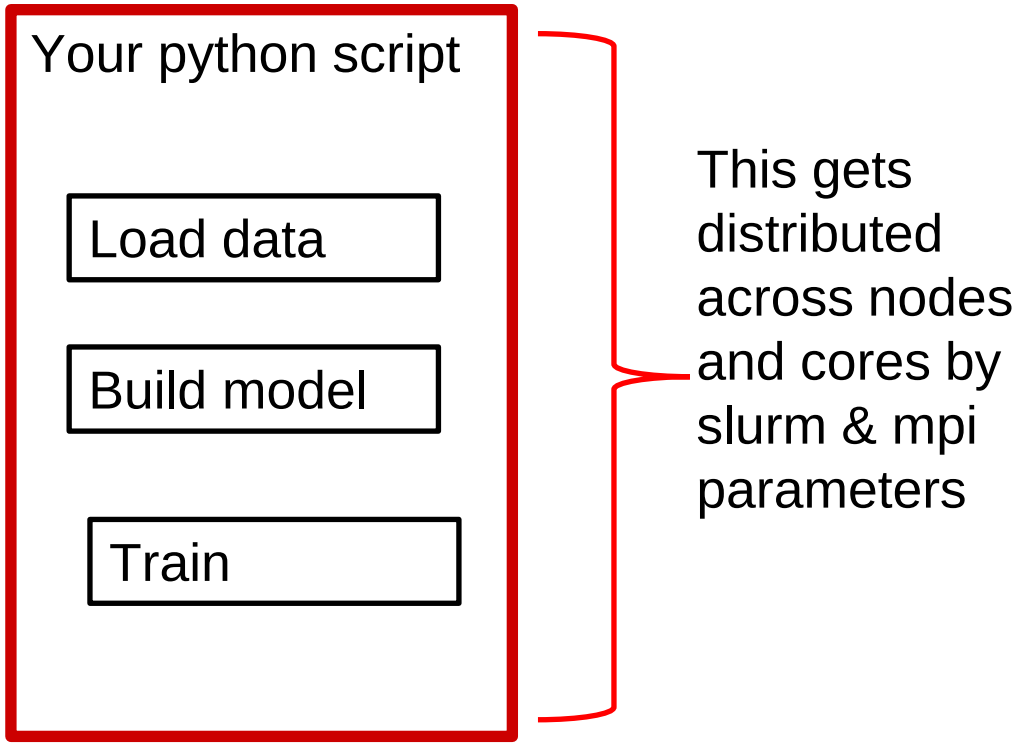
Build model

Train

Multinode, mpi launches instances

In slurm batch script:

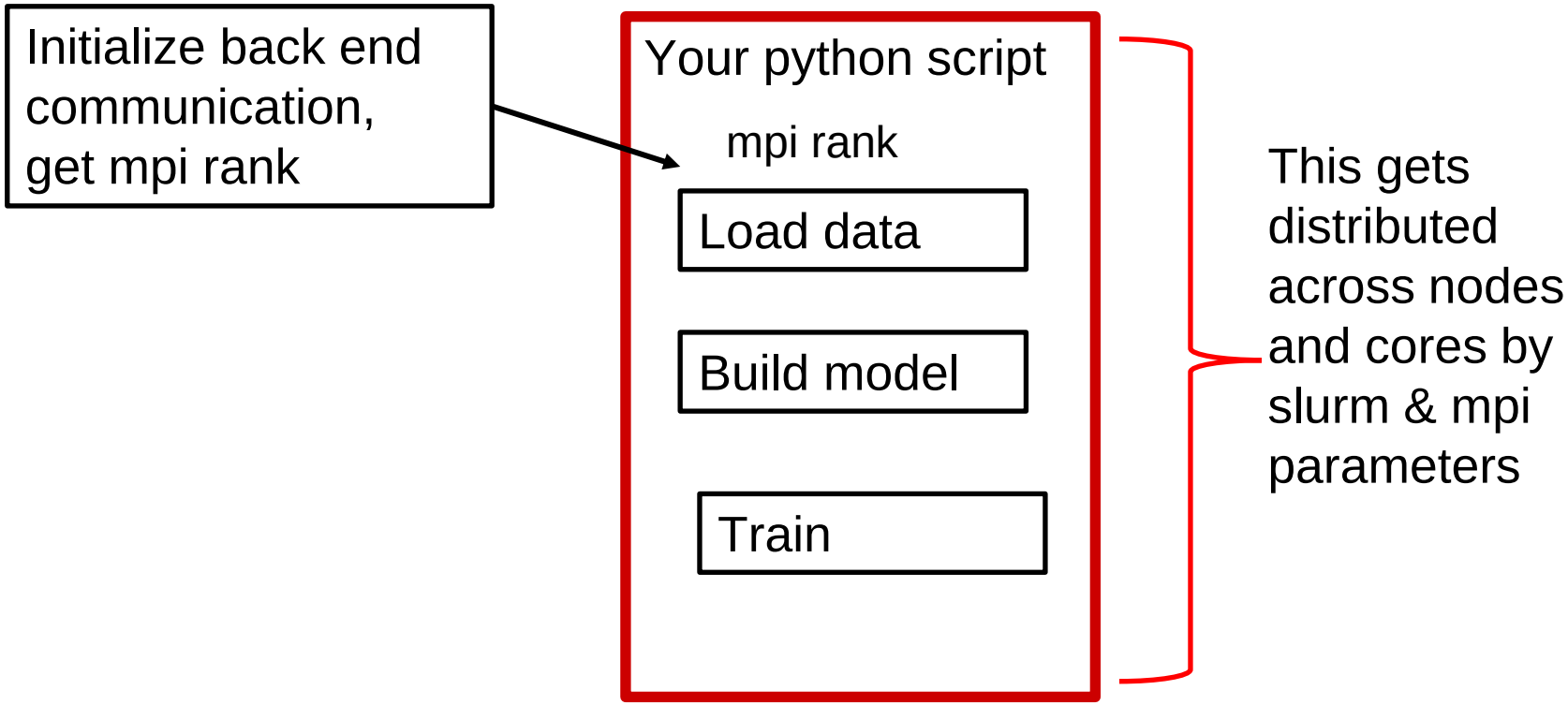
mpirun –n **number of tasks** singularity → python



Multinode, mpi launches instances

In slurm batch script:

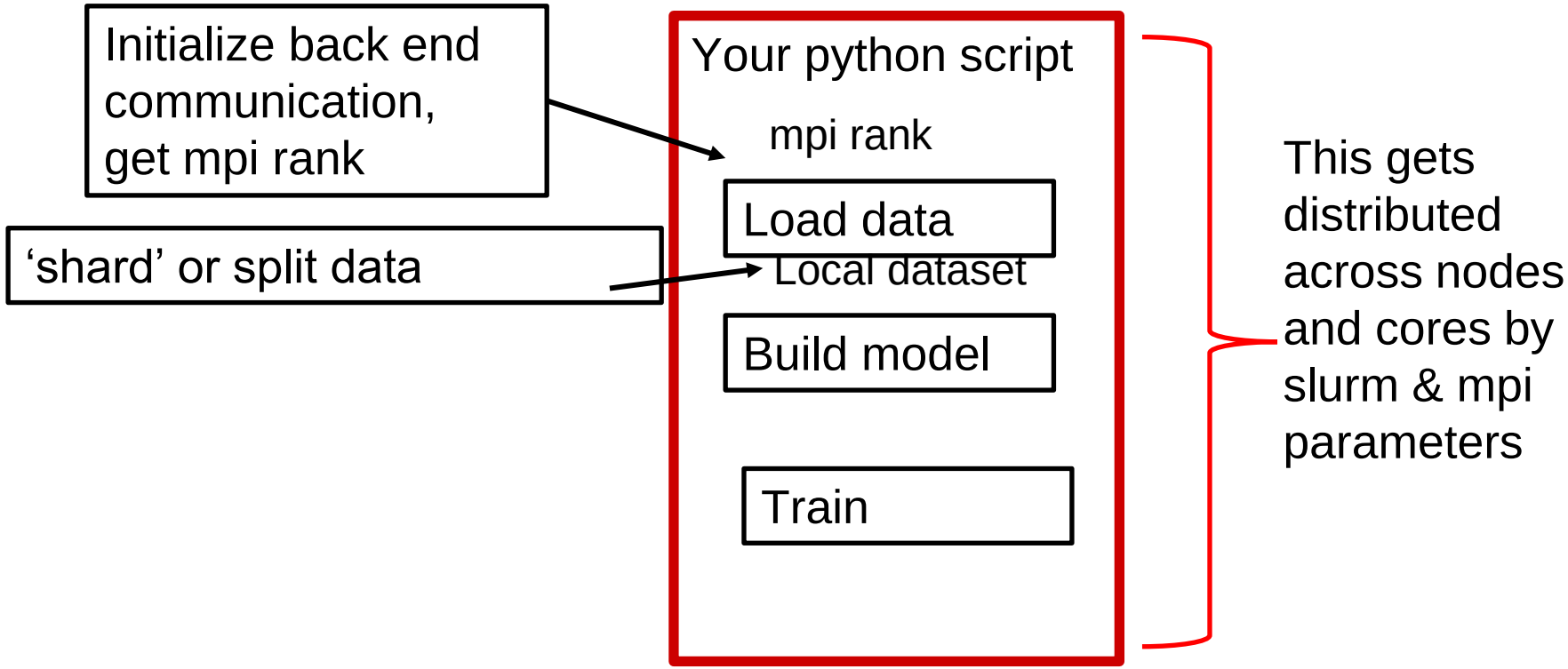
mpirun –n **number of tasks** singularity → python



Multinode, mpi launches instances

In slurm batch script:

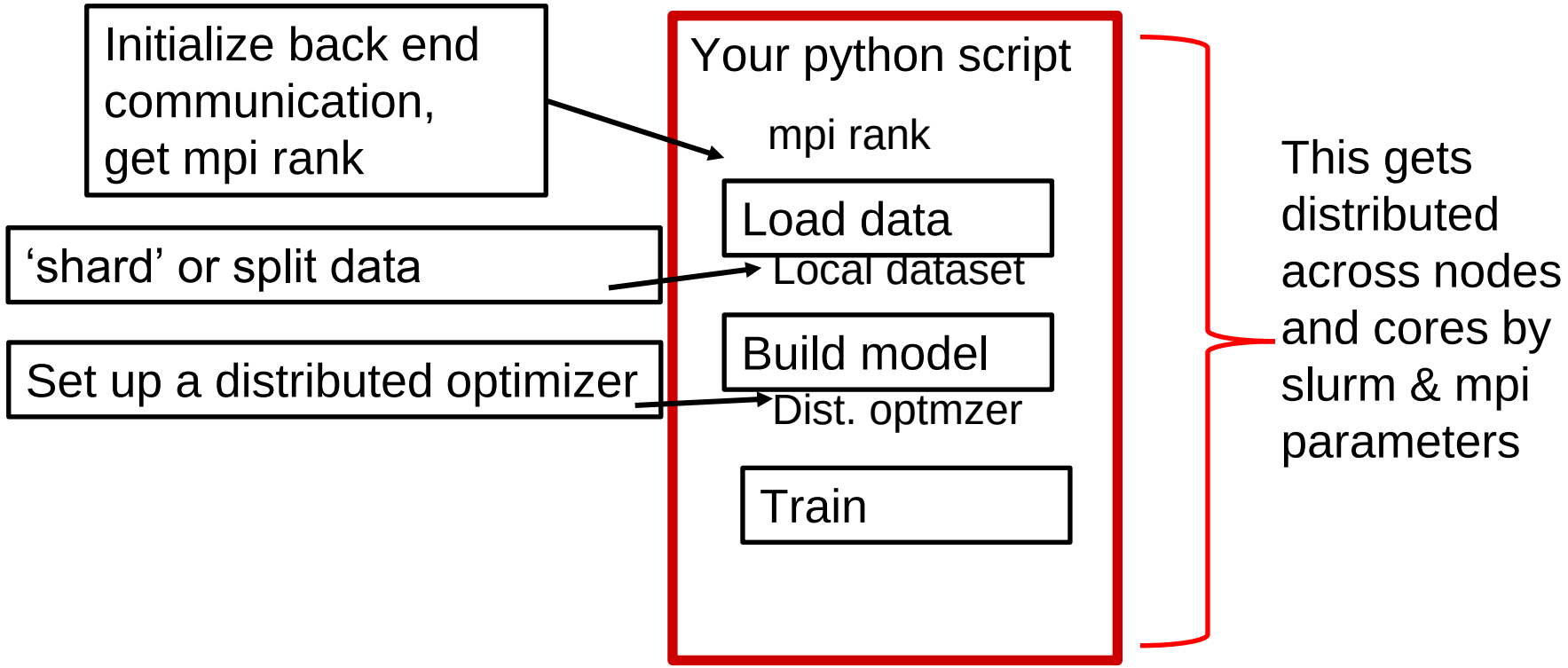
mpirun –n **number of tasks** singularity → python



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

Initialize back end communication, get mpi rank

‘shard’ or split data

Set up a distributed optimizer

After each batch aggregate weight updates and broadcast new values

Your python script

mpi rank

Load data

Local dataset

Build model

Dist. optmzer

Train

Rank 0 handles weight updates, checkpoints

This gets distributed across nodes and cores by slurm & mpi parameters

mpi launches one instance per processor

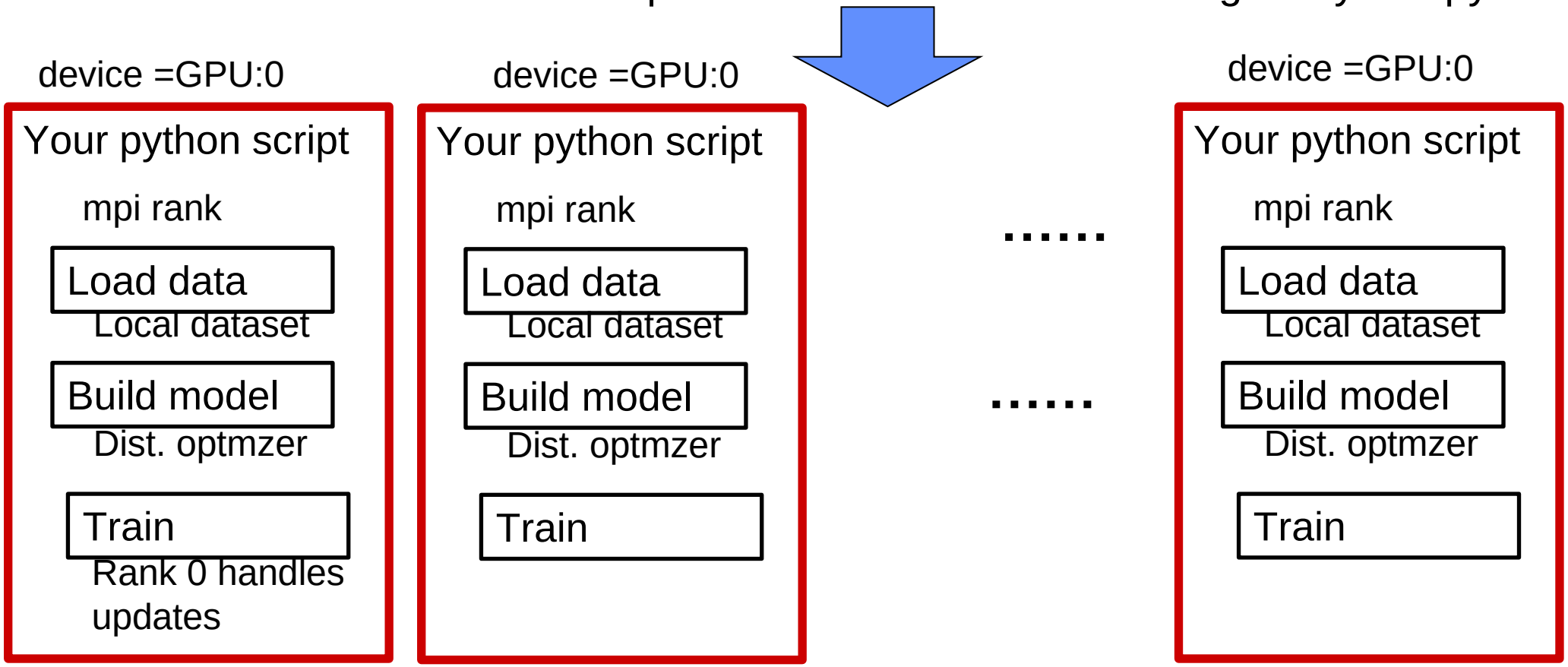
In slurm batch script:

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

mpi launches one instance per processor

In slurm batch script:

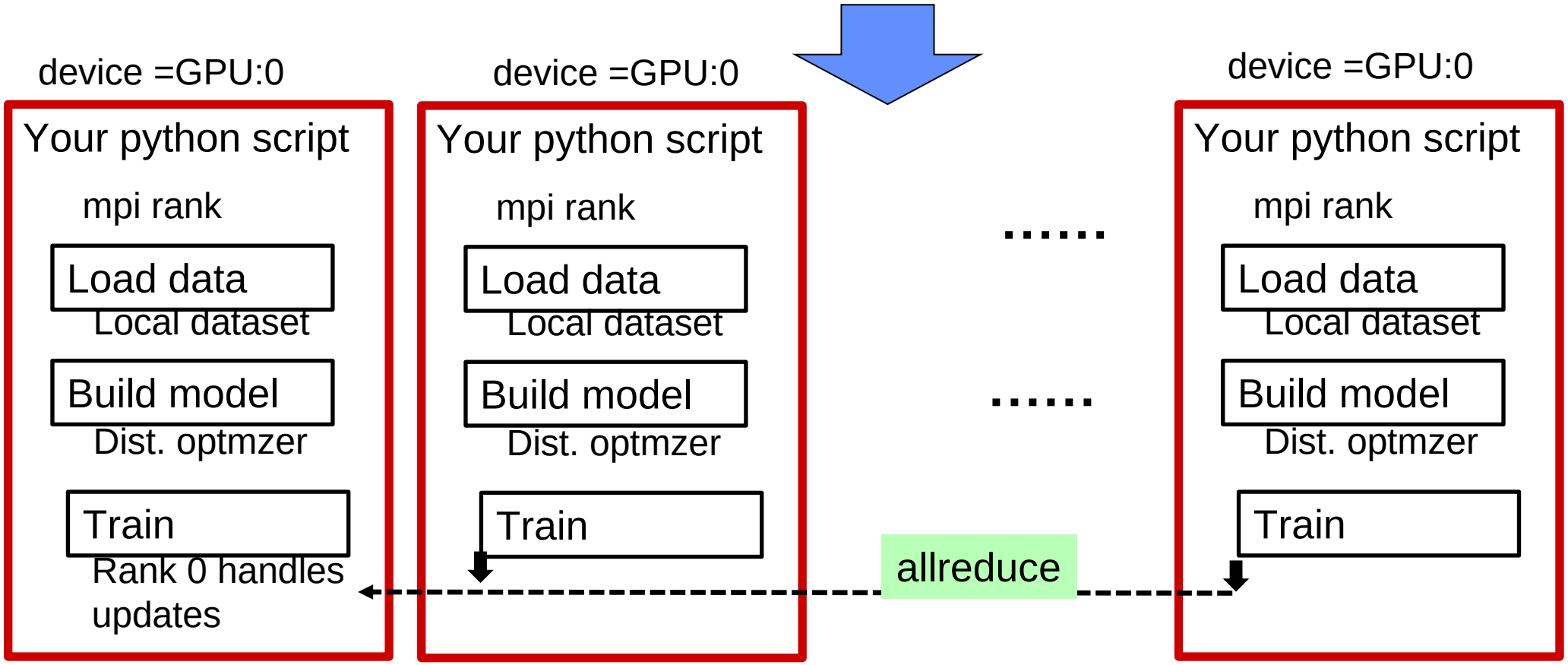
mpirun –n **number of tasks** singularity → python



For each batch: Horovod will aggregate & share weights updates

In slurm batch script:

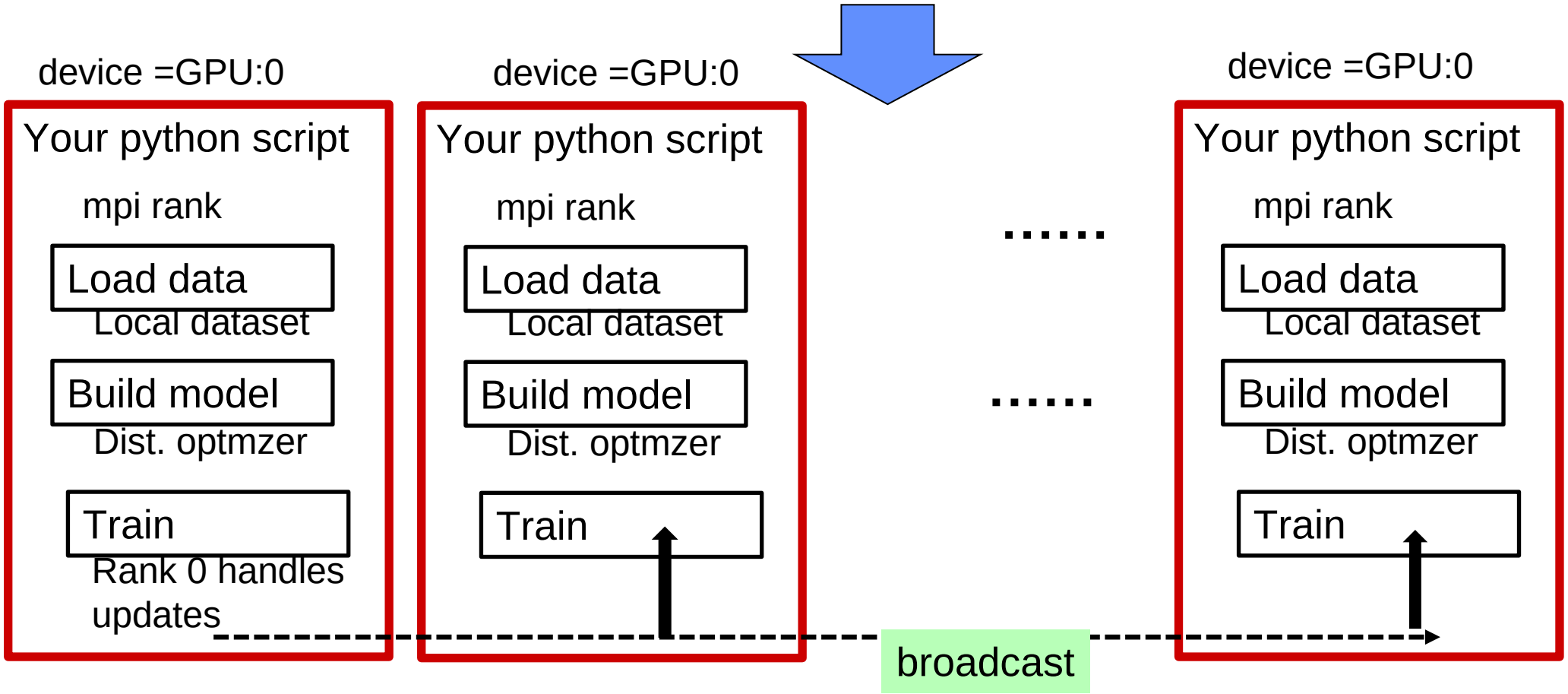
mpirun –n **number of tasks** singularity → python



For each batch: Horovod will aggregate & share weights updates

In slurm batch script:

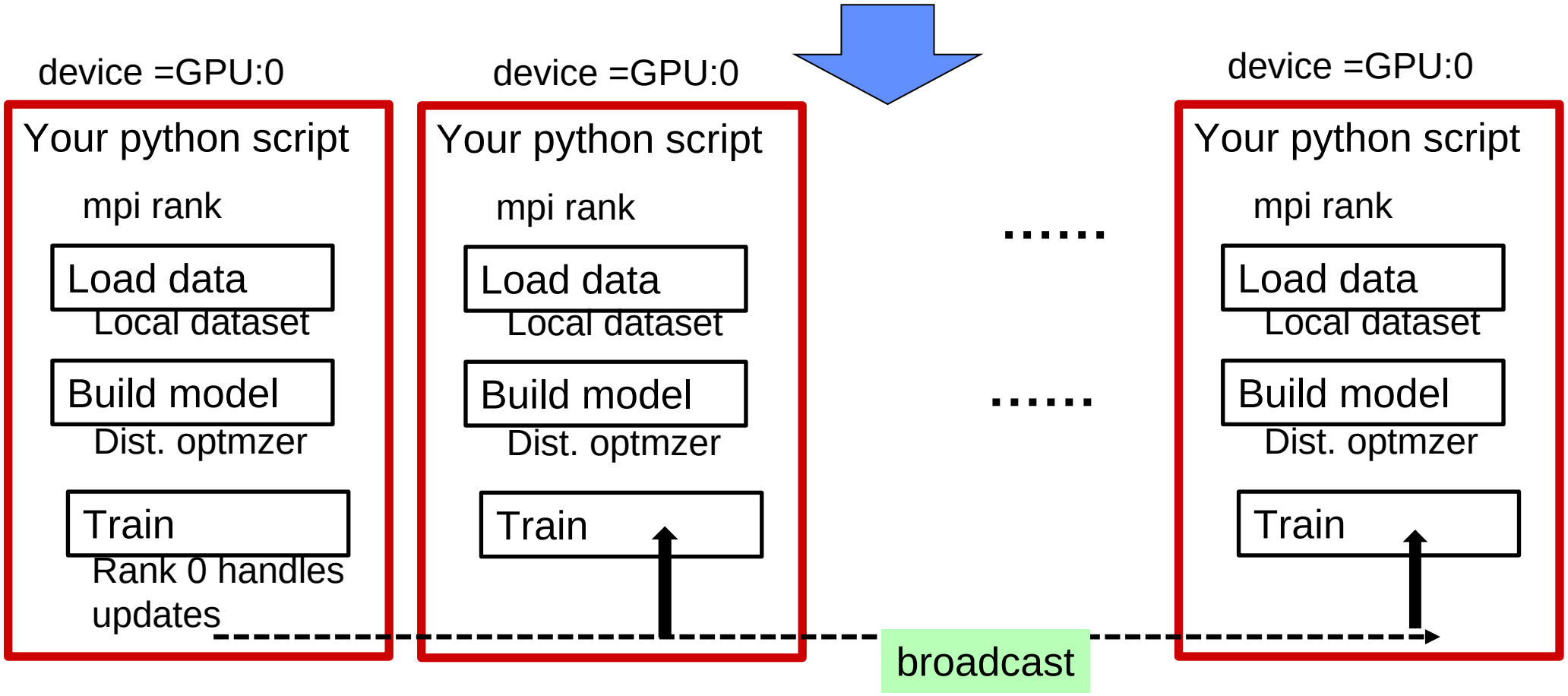
mpirun –n **number of tasks** singularity → python



For each batch: Horovod 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 – Horovod functions

Not many lines of code, but becareful with sharding, batch size,
See https://horovod.readthedocs.io/en/latest/keras.html

Initialize back end communication, get mpi rank

```
import horovod.tensorflow.keras as hvd
hvd.init()
```

'shard' or split data

```
Note: many ways to do this, either directly splitting numpy
arrays and/or using TF datasets
```

Set up a distributed optimizer

```
optimizer2use  = hvd.DistributedOptimizer(……)
```

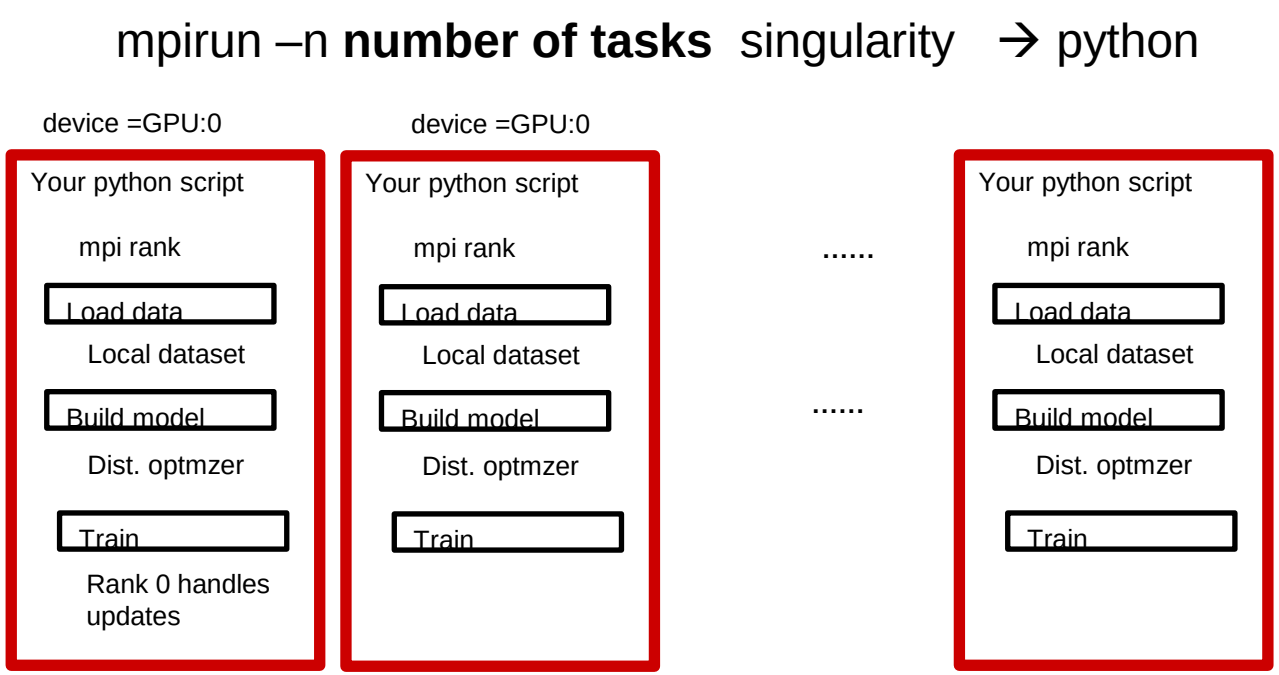
After each batch aggregate weight updates and broadcast new values

```
model.compile(optimizer    =optimizer2use,
 …
experimental_run_tf_function=False)
```

Exercise, multinode MNIST programming and execution

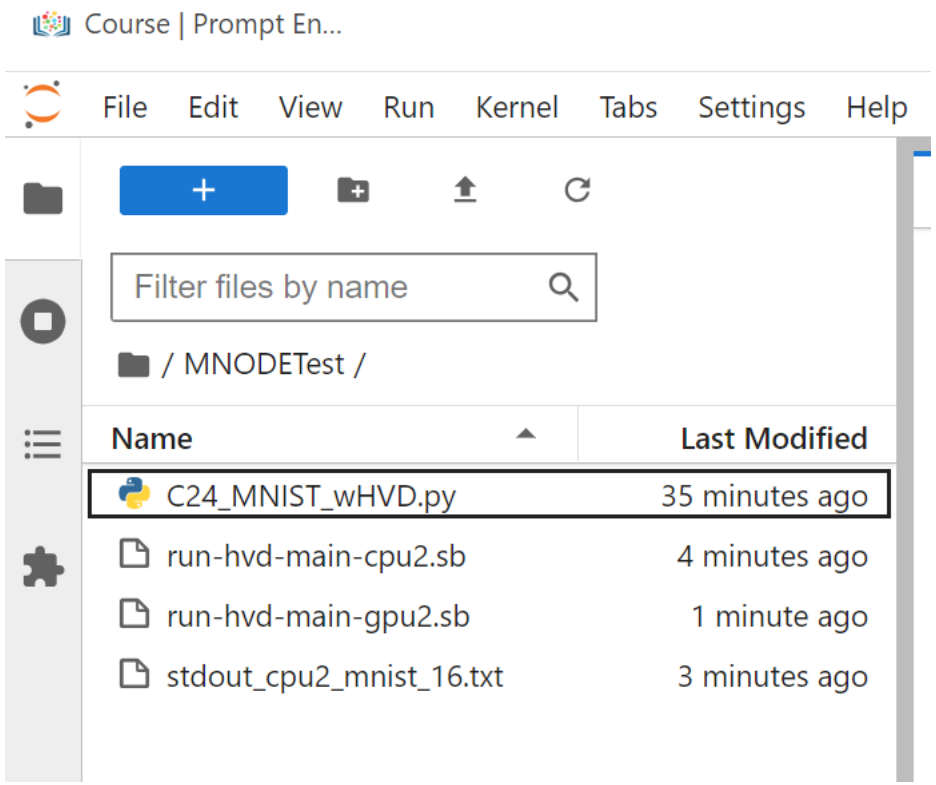
- **Goal: Get familiar with Keras and Horovod coding for multinode execution**
- **Goal: Get familiar with slurm batch script multinode parameters**
- **Let's login and start a notebook**

(see next pages for quick overview)



In terminal]$ **jupyter-compute-tensorflow**

In jupyter notebook session open the *MNIST_wHVD* notebook



Open the *run-hvd-main-cpu2.sb* slurm batch script

Follow instructions in the notebook and batch script

Initialize back end communication, get mpi rank

```
#-----------------------------------------------
import horovod.keras as hvd
hvd.init()
print('INFO, global rank:',hvd.rank(), ' localrank ',hvd.local_rank())
#-------------------------------------------------------------
```

split data using numpy arrays

```
# ------------------ Get Dataset ---------------------------
per_worker_batch_size = 32        #Pick factors of 32 (especially for GPU)
num_workers           = hvd.size()
```

Set up a distributed optimizer to manage weight updates

```
# -------------------------------------------------------

#------- Enter the num of processes to scale the learning rate here -------
optimizer2use    = tf.keras.optimizers.Adam(learning_rate=0.001*hvd.size())  #<<<<<<---------
optimizer2use    = hvd.DistributedOptimizer(optimizer2use)

# ------------ for HVD ---------------------------------------
#Specify `experimental_run_tf_function=False` to ensure TensorFlow
```

Your Task:

- run sbatch command for the slurm script (for cpu)

- change number of cpus to use, rerun, and review stdout output file

Launcher | MNIST_wHVD_solution.py

```
# 2
# Do a File->open of the run-hvd-main-cpu2.sb slurm batch script
#       optionally edit the number of cpus to use, try for example 4,8,16, and/or 32
# 3
# In a terminal window, submit the script and review the job status
#          ]$  sbatch run-hvd-main-cpu2.sb
#          ]$  squeue -u your-userid
#
# Optionally, ssh into the running nodes and run top command (top -u userid)
#
# 4
```

jupyter run-hvd-main-cpu2.sb 15 minutes ago Logout

File Edit View Language Plain Text

```
#SBATCH --job-name=tfhvd-cpu
#SBATCH --account=use300
#SBATCH --partition=compute
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=16  #<<<<<------ change this to 16 and observe changes in training time
#SBATCH --cpus-per-task=1
#SBATCH --mem=243G
#SBATCH --time=00:15:00
#SBATCH --output=slurm.cpu2.%x.o%j.out
```

```
[p4rodrig@login01 MNODE_wHVD]$
[p4rodrig@login01 MNODE_wHVD]$ grep 'done, rk: 15' stdout_*
stdout_cpu2_mnist_32.txt:INFO,done, rk: 15  train time: 2.48225  secs
stdout_mainhvd_cpu2.txt:INFO,done, rk: 15  train time: 2.31222  secs
[p4rodrig@login01 MNODE_wHVD]$
[p4rodrig@login01 MNODE_wHVD]$
```

- **pause**

```
in113@login01 3.3.Practical-Training]$ grep 'done, rank:  0  train' stdout_cpu2_mnist_*
ut_cpu2_mnist_16.txt:INFO,done, rank:  0  train time: 19.10057  secs
ut_cpu2_mnist_32.txt:INFO,done, rank:  0  train time: 15.46317  secs
ut_cpu2_mnist_8.txt:INFO,done, rank:  0  train time: 33.32454  secs
in113@login01 3.3.Practical-Training]$
```

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**