Distributed Deep Learning with PyTorch

Zhao Zhang
zzhang@tacc.utexas.edu

Texas Advanced Computing Center
The University of Texas at Austin

Aug 6th, 2021

Outline

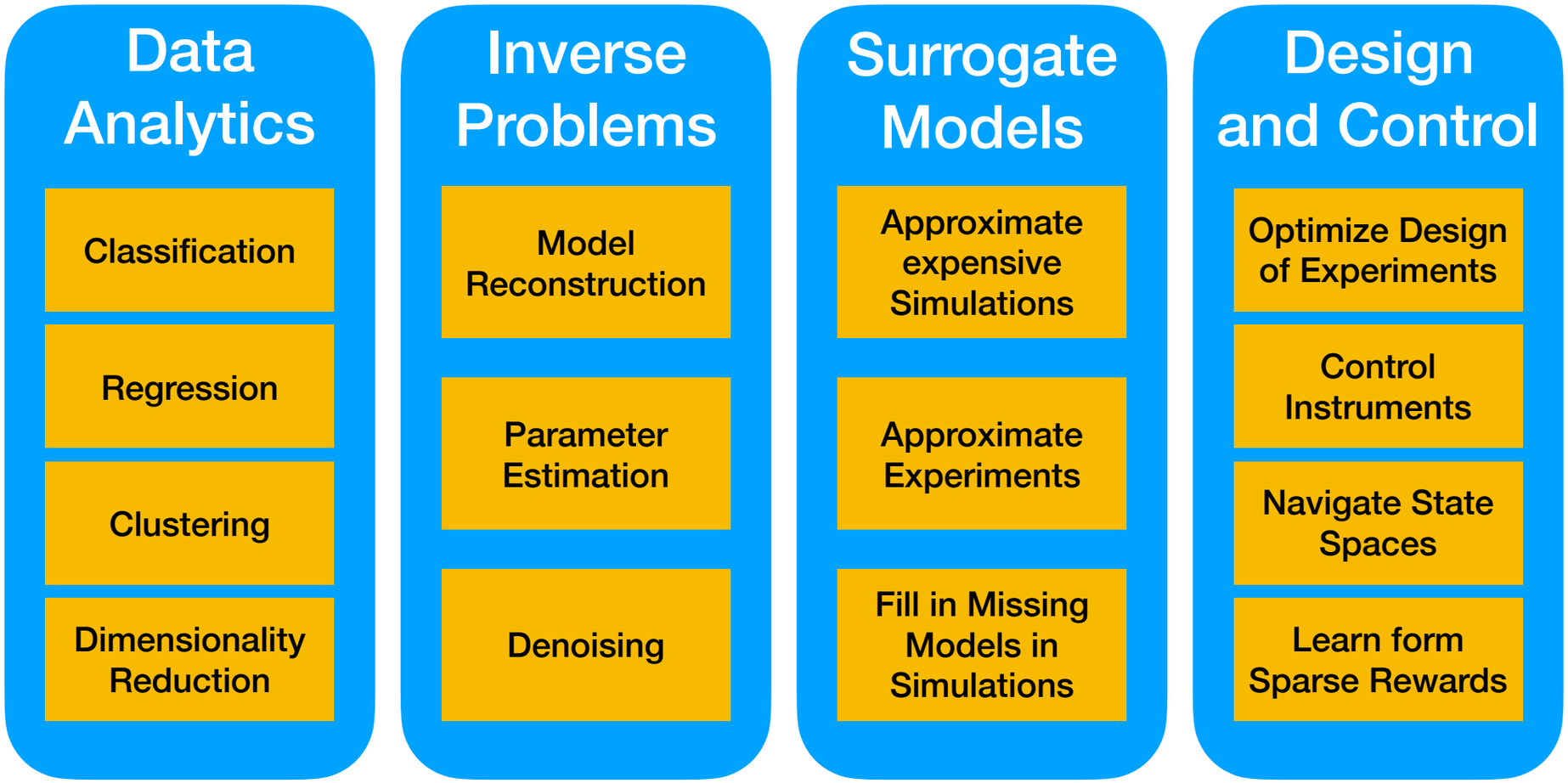
- AI in Science

- Distributed Deep Learning
  - PyTorch Introduction

- Distributed Training with K-FAC

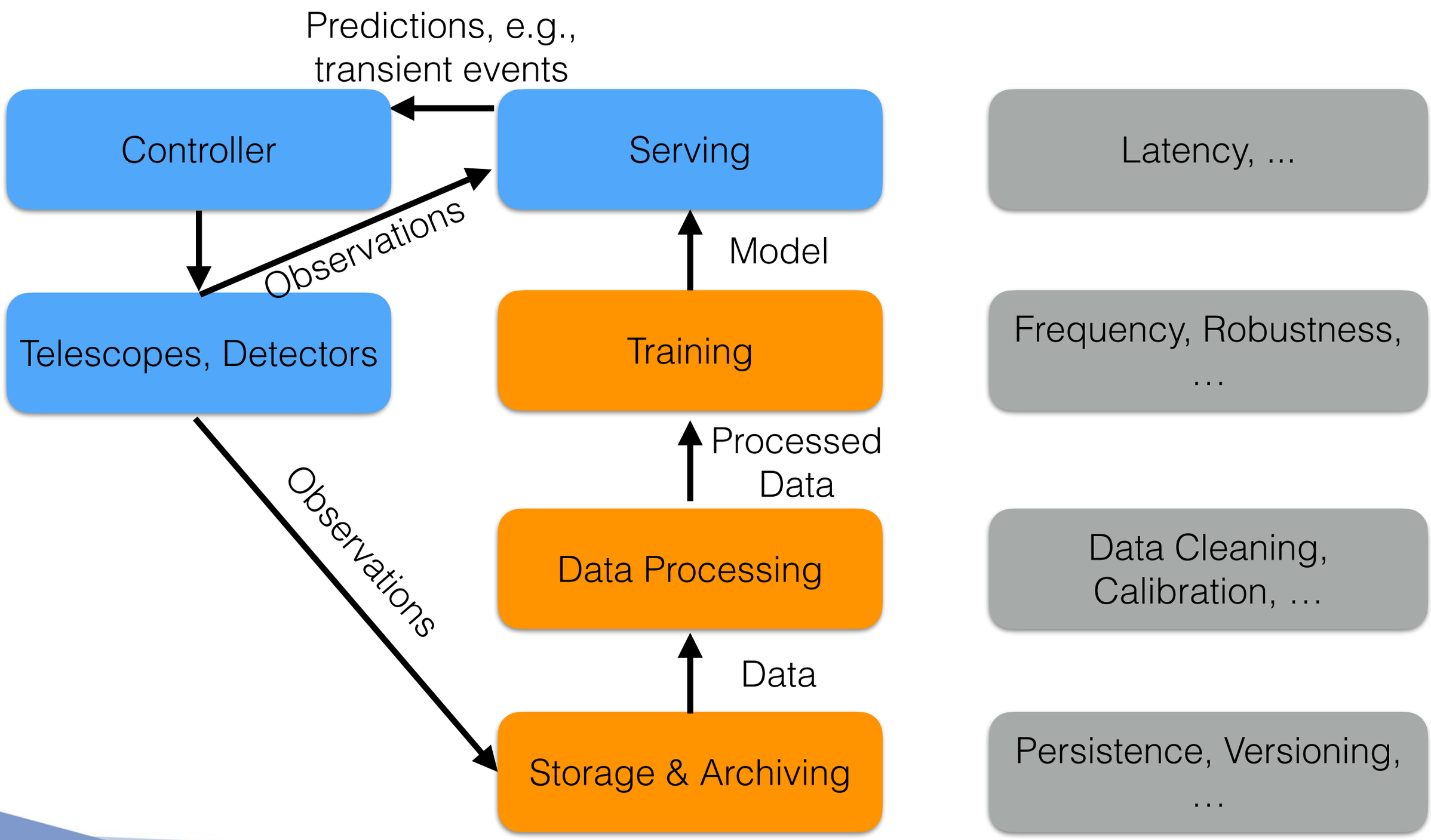
AI/ML/DL and HPC

- Artificial Intelligence (AI) is a driving application for exascale machines, along with simulation and big data

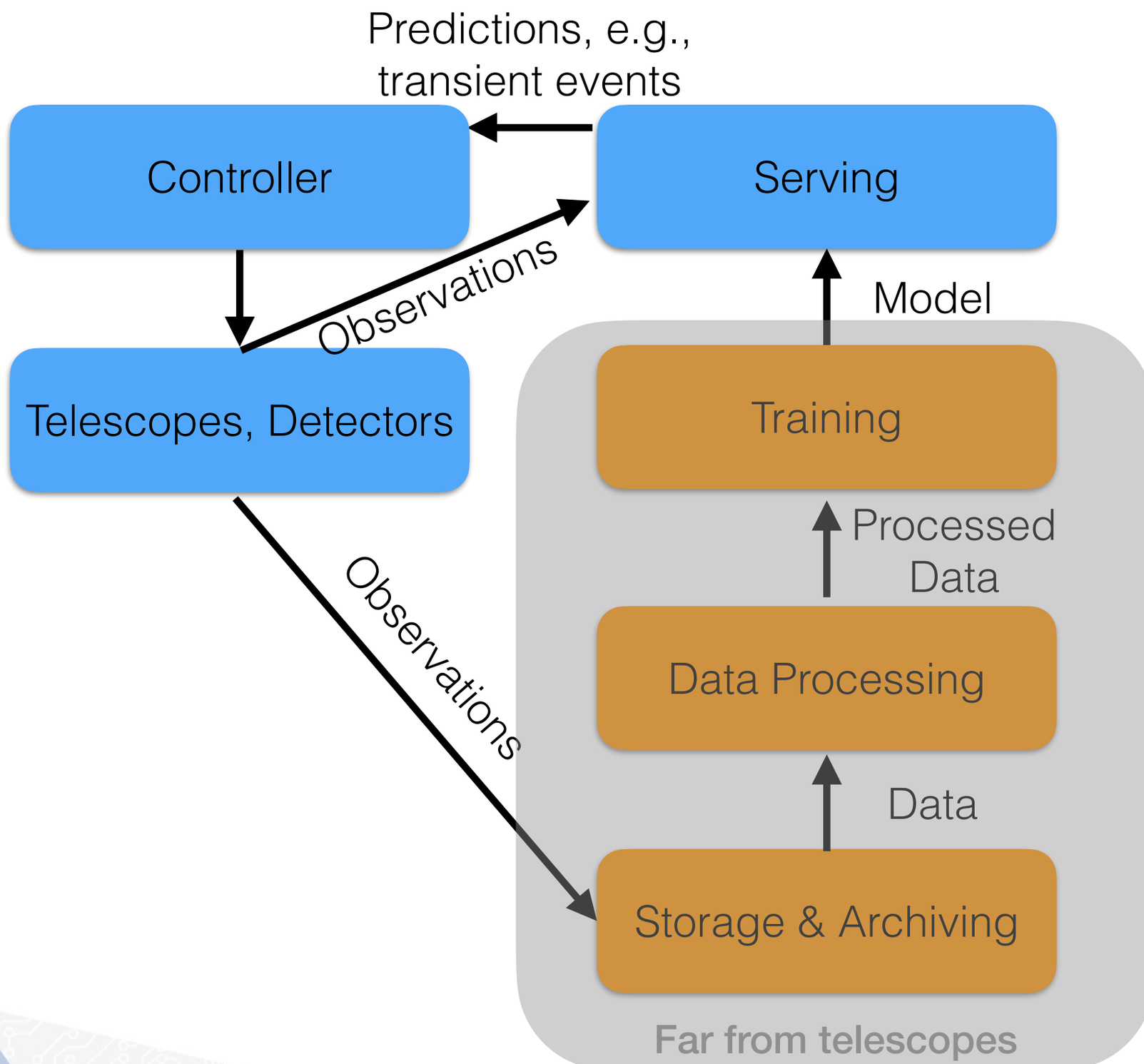


Credit: Kathy Yelick, in Monterey Data Conference, 2019

Data-driven Decision Making Pipeline



Data-driven Decision Making



Outline

- AI in Science

- Distributed Deep Learning
  - PyTorch Introduction

- Distributed Training with K-FAC

Scalable Deep Learning

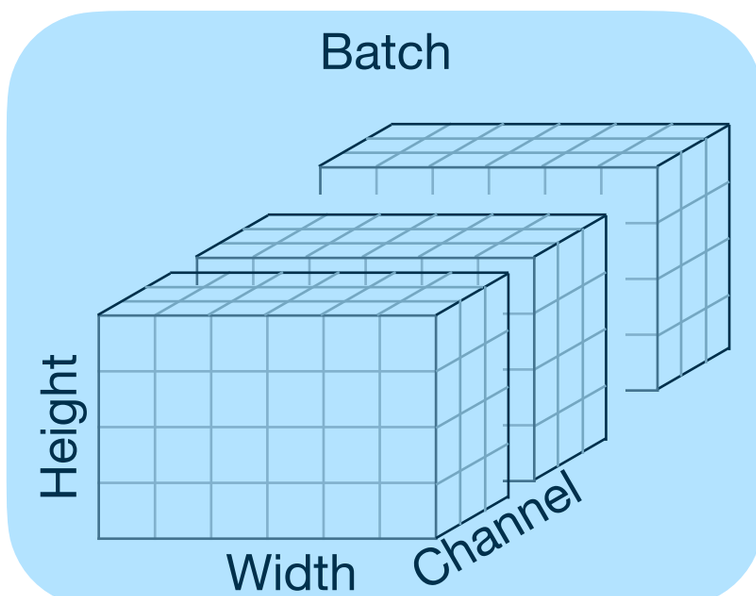
- Why would someone train on multiple nodes?
  - Shorter training time
  - Larger model

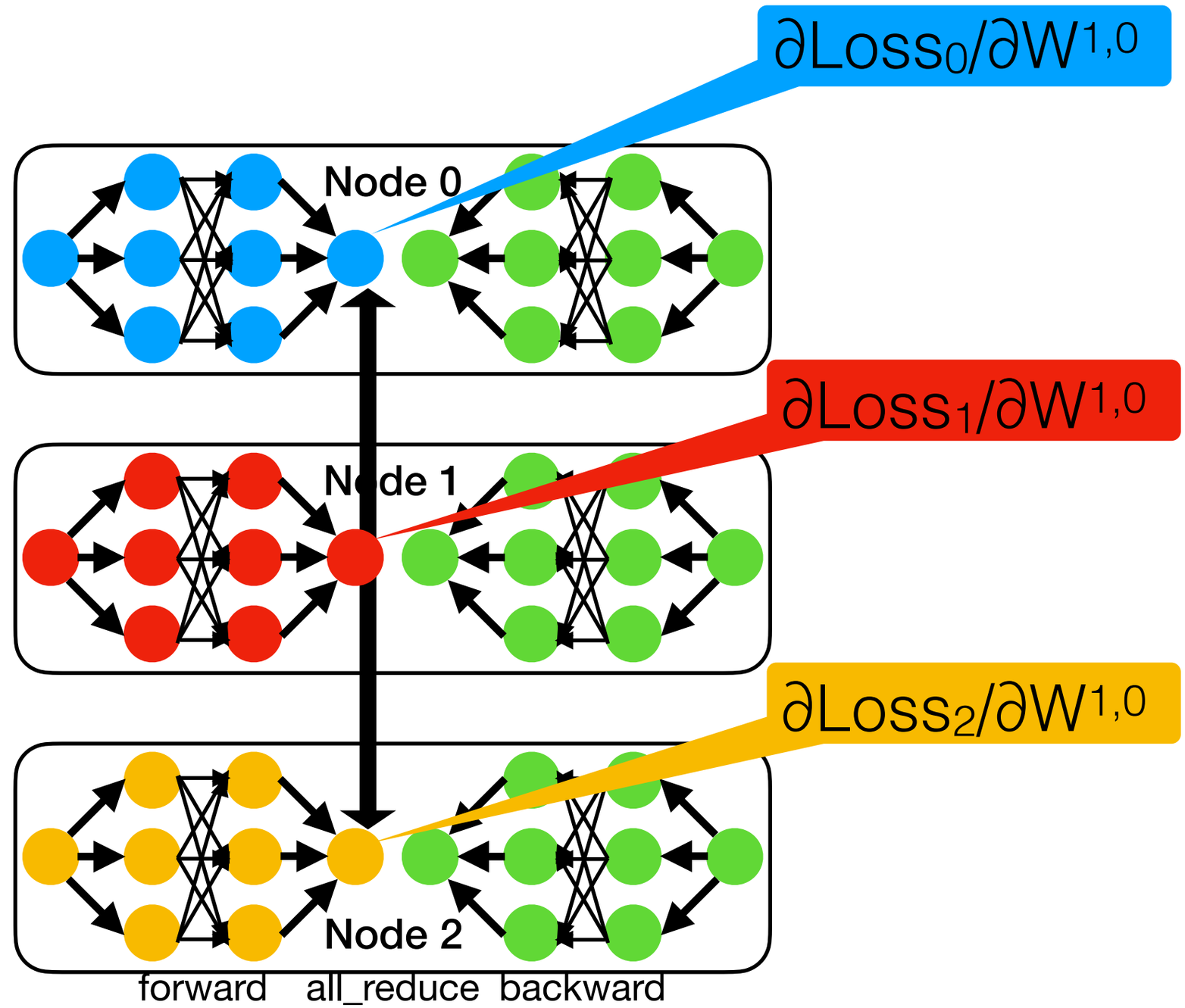
Distributed Training

- Data Parallel
- Model Parallel
- Pipeline Parallel
- Hybrid Parallel

Distributed Training

- Data Parallel

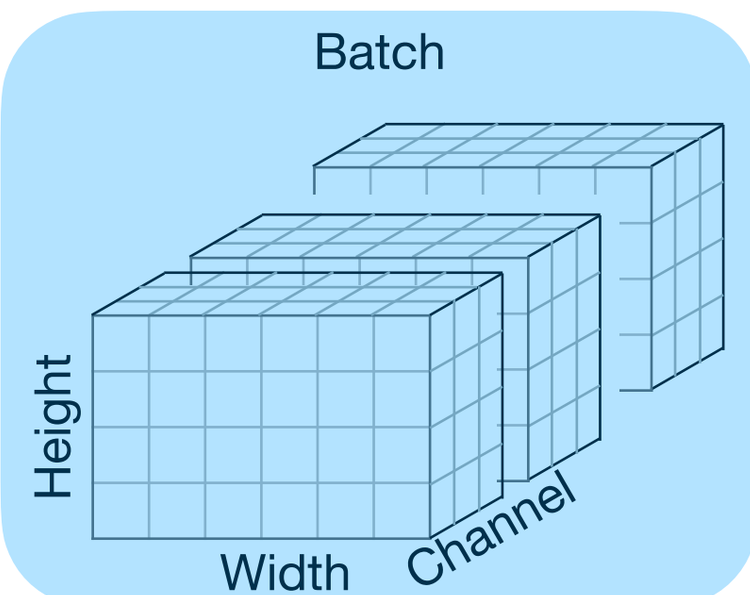


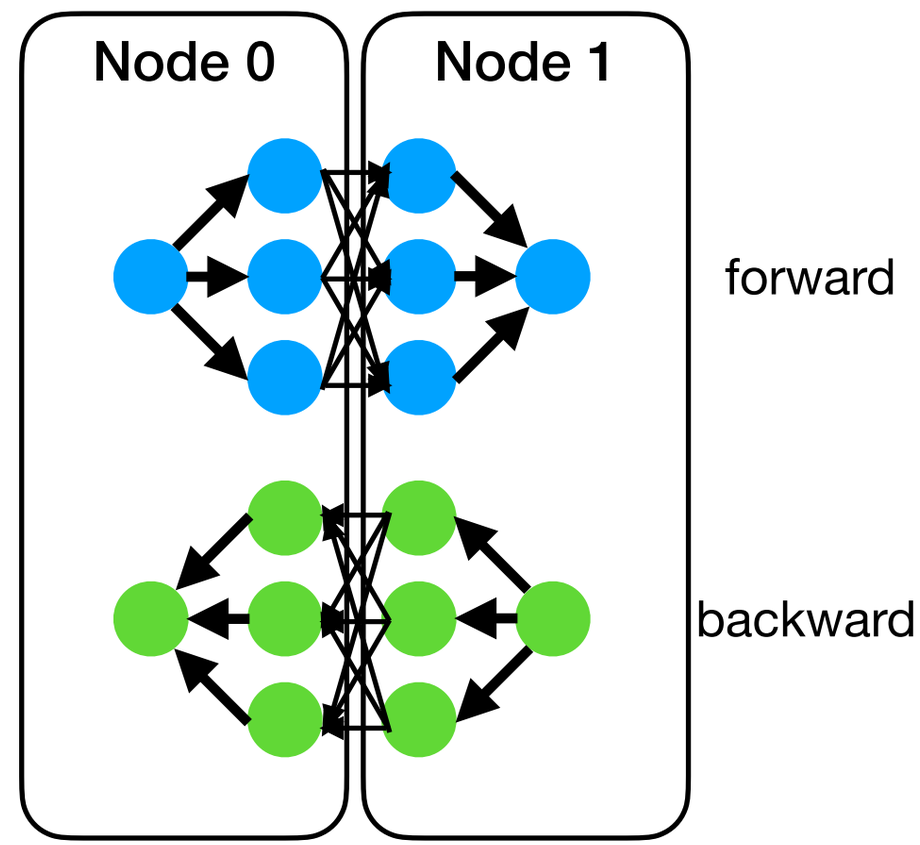


- **Parameters (Model) are duplicated on all nodes**

Distributed Training

- Model Parallel

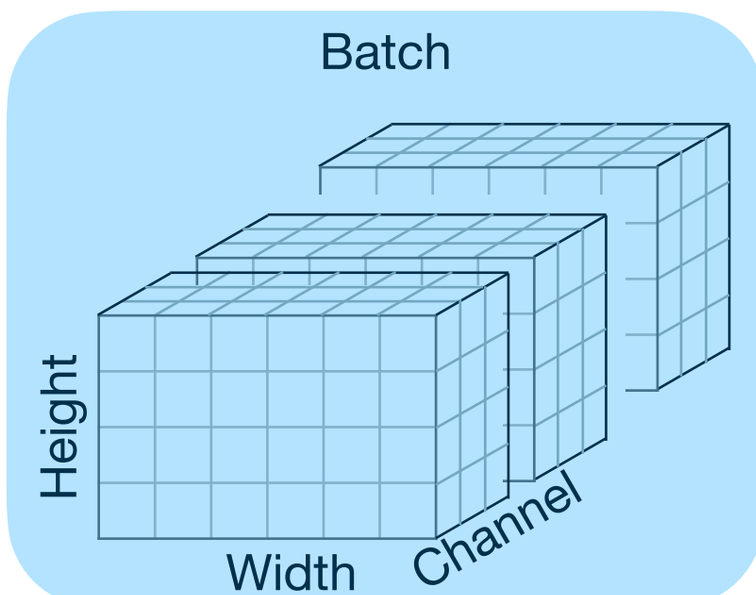


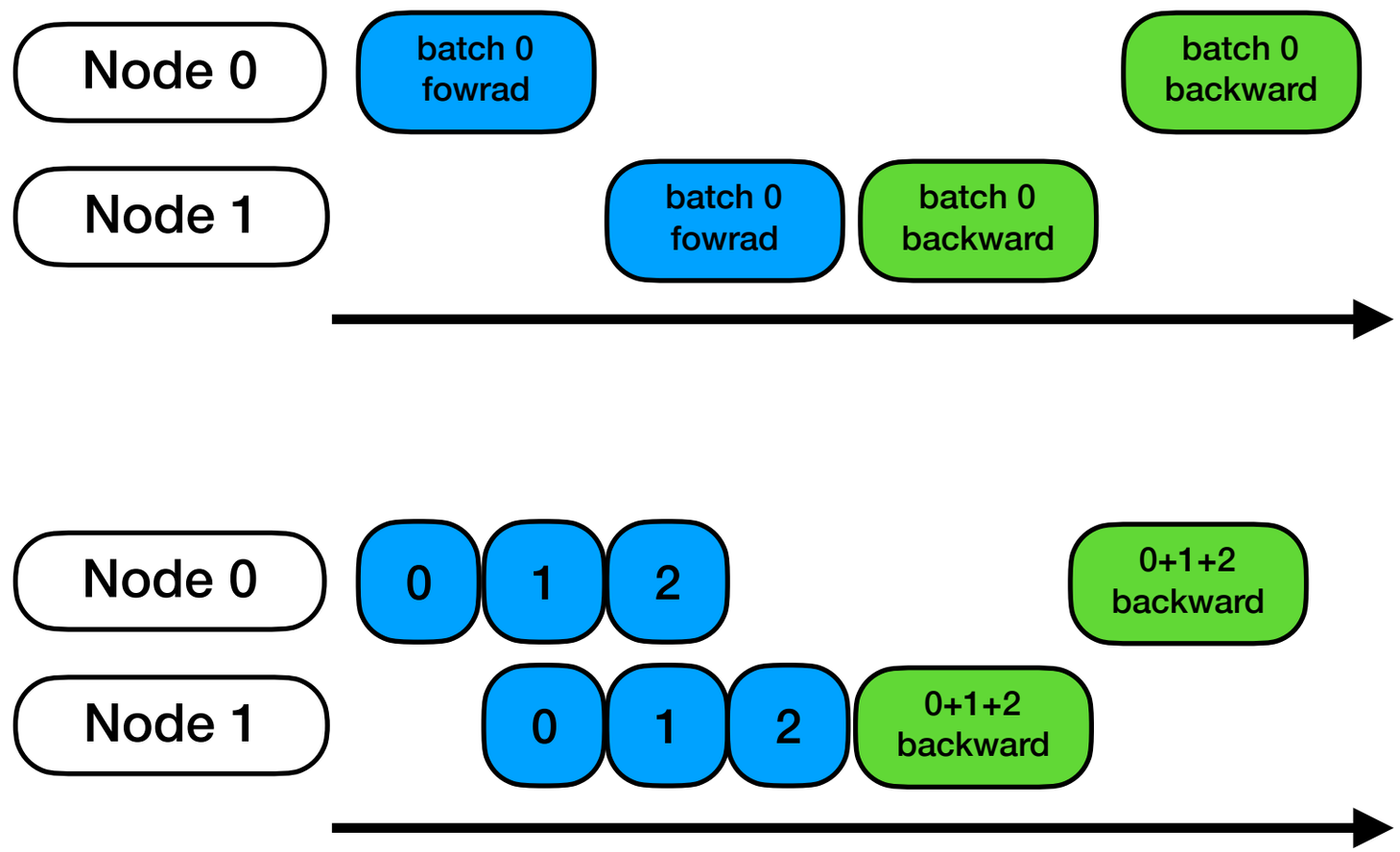


- **Weights are distributed to all nodes**

Distributed Training

- Pipeline Parallel





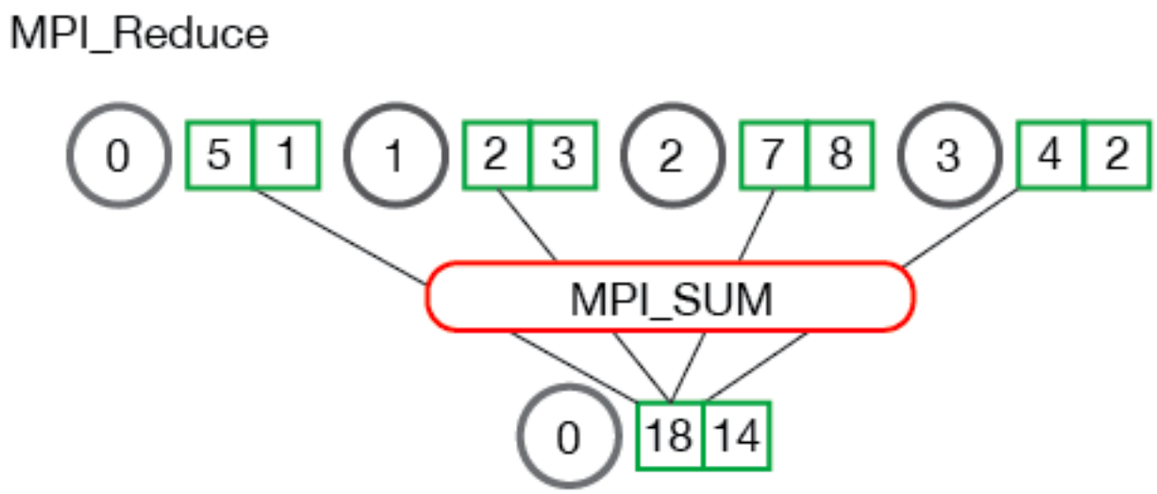
- **Weights are distributed to all nodes**

Distributed Training

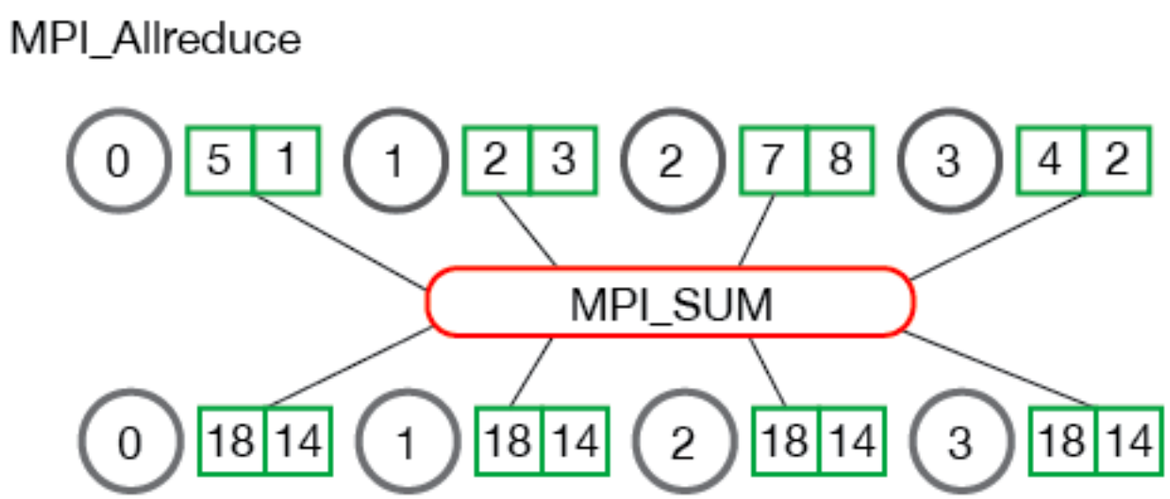
- Data Parallel is widely adopted by popular DL frameworks such as TensorFlow, PyTorch, and MXNet.
- Model Parallel is applied when a model exceeds memory limit.
- Pipeline Parallel assumes model being distributed across nodes, and each node holding multiple versions of the model. It is often used with asynchronous optimizers.

All-Reduce

- Reduce



- All-Reduce



All-Reduce

- Reduce-scatter algorithm

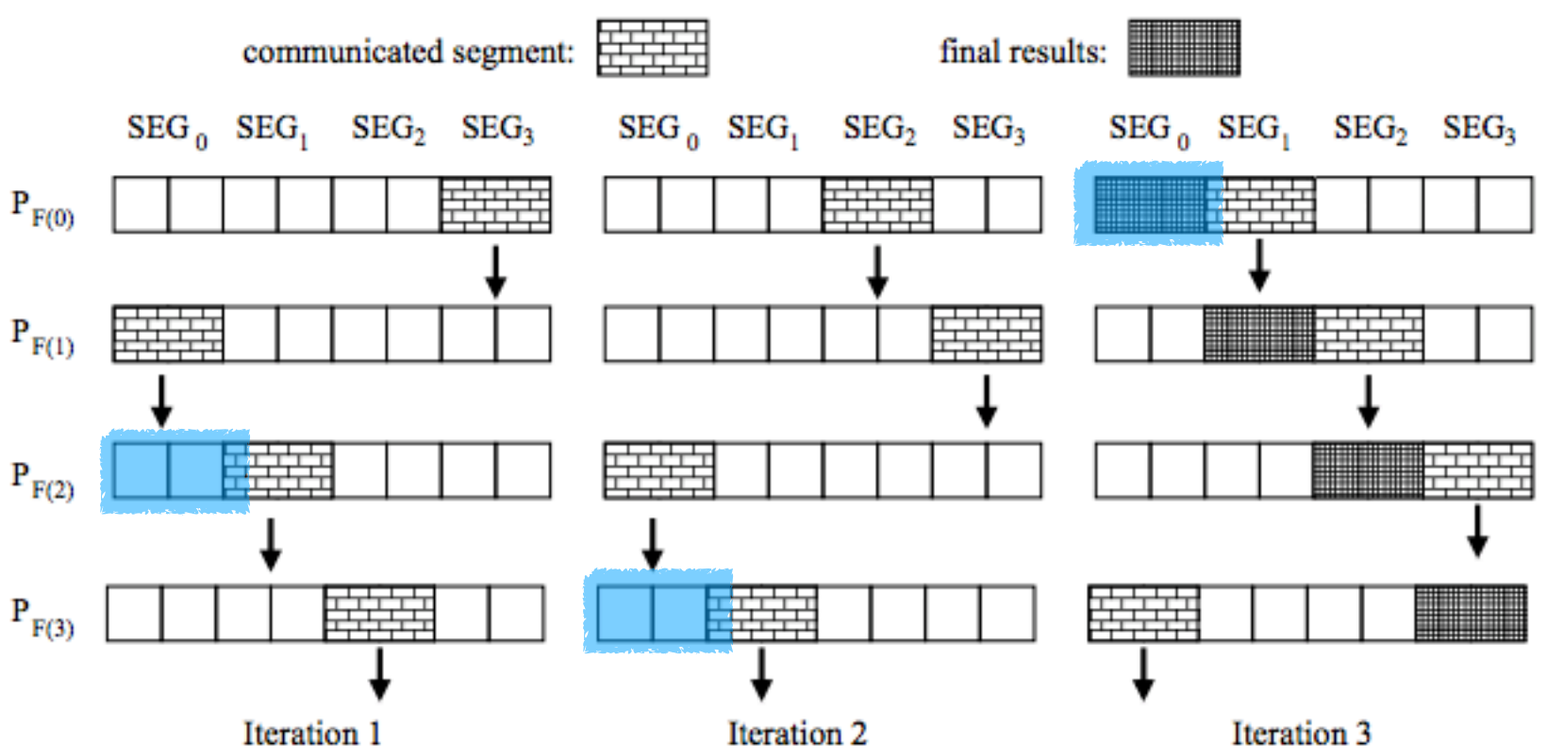


Figure 1: Logical ring reduce-scatter algorithm

All-Reduce

- Reduce-scatter algorithm

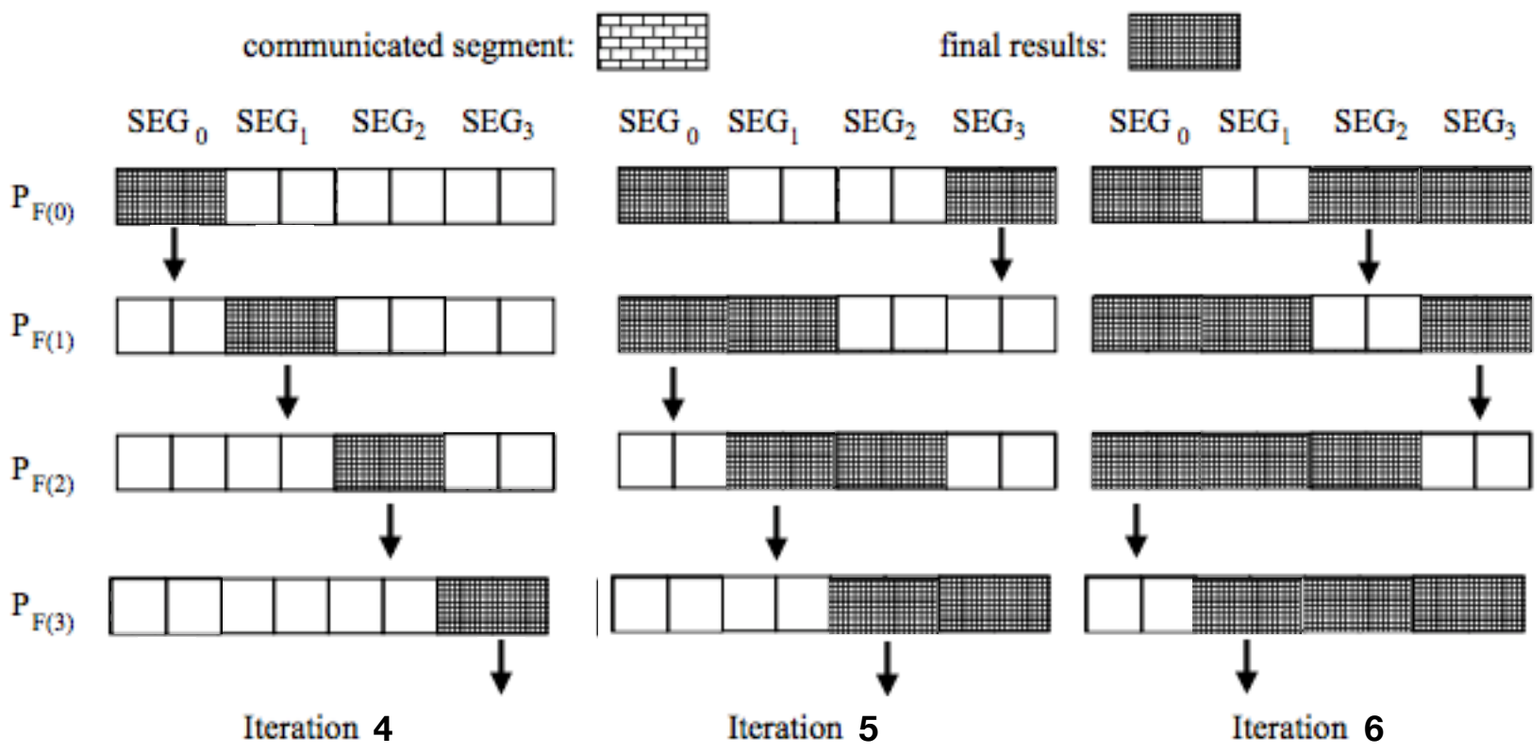


Figure 1: Logical ring reduce-scatter algorithm

Total_Comm_Time = 2 x (N-1) x $T_{transfer}$

torch.distributed

- The usage of torch.distributed is similar to that of MPI
- The notion of rank and world_size

PyTorch Programming

- Similar to Keras/TF, you can compose a PyTorch program in 4 steps:
  - Dataset Preparation
  - Model Definition
  - Optimizer Specification
  - Training Instrumentation

PyTorch Dataset

- Dataset —> Dataloader
- PyTorch has built-in datasets, e.g., CIFAR10

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose

train_data = datasets.CIFAR10(root="/tmp", train=True, download=True, transform=ToTensor())

test_data = datasets.CIFAR10(root="/tmp", train=False, download=True, transform=ToTensor())

batch_size = 128

# Create data loaders.
train_dataloader = DataLoader(train_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)
```

PyTorch Model — nn.Sequential()

```
import torch
from torch import nn

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = Net().to(device)
```

PyTorch Model — nn.Functional()

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  # 5*5 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square, you can specify with a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = Net().to(device)
```

PyTorch Optimizer

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

PyTorch Training

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

PyTorch Training

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) ==
                        y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
{test_loss:>8f} \n")
```

PyTorch Training

```
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print(“Done!”)
```

Distributed PyTorch Training

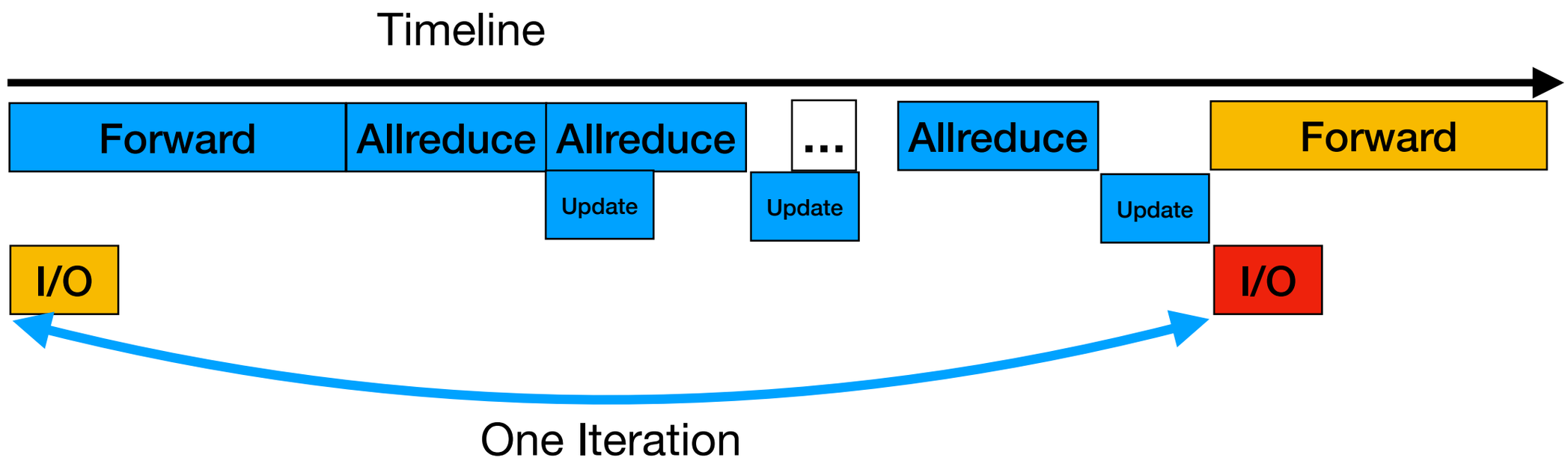
```
import torch.distributed as dist

def main():
    ...
    torch.distributed.init_process_group(backend='nccl', init_method='env://')
    ...
    torch.cuda.set_device(args.local_rank)
    ...
    model = torch.nn.parallel.DistributedDataParallel(model,
            device_ids=[args.local_rank])
    ...
    train_sampler = torch.utils.data.distributed.DistributedSampler(
            train_dataset, num_replicas=args.backend.size(), rank=args.backend.rank())
    train_loader = torch.utils.data.DataLoader(
            train_dataset, batch_size=args.batch_size * args.batches_per_allreduce,
            sampler=train_sampler, **kwargs)

    val_sampler = torch.utils.data.distributed.DistributedSampler(
            val_dataset, num_replicas=args.backend.size(), rank=args.backend.rank())
    val_loader = torch.utils.data.DataLoader(
            val_dataset, batch_size=args.val_batch_size,
            sampler=val_sampler, **kwargs)
    ...

    for epoch in epochs:
        train()
        test()
```

Distributed PyTorch Training

- On 1 node with 4 GPUs

```
python -m torch.distributed.launch --nproc_per_node=4
kfac_pytorch/examples/torch_cifar10_resnet.py \
     --data-dir /tmp/cifar-10-batches-py \
     --base-lr 0.1 \
     --batch-size 128 \
     --epochs 100 \
     --kfac-update-freq 0 \
     --model resnet32 \
     --lr-decay 35 75 90
```

Execution Model



- A simple understanding:

$$T = T_{forward} + \sum T_{allreduce}$$

Complex Model
Larger Batch

O(1/N)

O(N) or O(logN)

Faster Interconnect
Faster Algorithm
Less Data

DL and HPC

- Computation
  - DL training is computation intensive, while HPC has massive computing power
- Communication
  - The back-propagation algorithm is dominated by the all reduce operations (sum over gradients), while HPC has the low-latency high-bandwidth interconnect
- I/O
  - The massive I/O (1.3 million small image files for ImageNet x 90 epochs) can be problematic for conventional shared file system

Previous Work

- In Oct 2017, TACC collaborated with UC Berkeley and UC Davis and reduced ResNet-50 training to 20 mins with LARS using 2,048 Intel KNL processors.
- Follow-on Work
  - 6.6 minutes using 2,048 Tesla P40 GPUs from researchers in Tencent and Hong Kong Baptist University, 07/2018
    - Half-precision for forward computation and back propagation, single precision for LARS
  - 224 seconds using 2,176 Tesla V100 GPUs from Sony researchers, 11/2018
    - 2D-Torus optimization with LARS
  - 2.2 minutes with 1,024 chip Google TPUv3 cores, 11/2018
  - 67.1 seconds with 2,048 chip Google TPUv3 cores, 10/2019

Outline

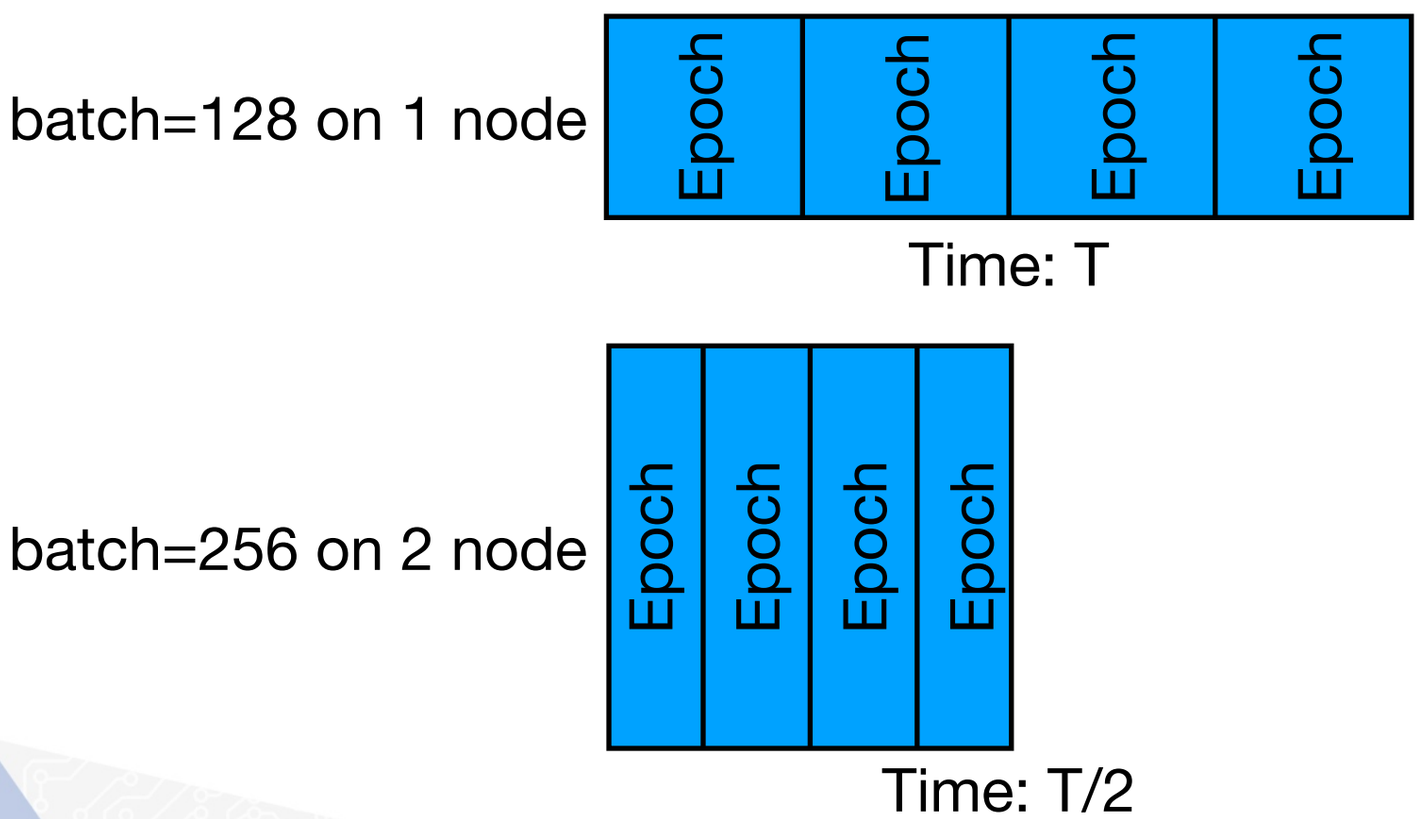
- AI in Science

- Distributed Deep Learning
  - PyTorch Introduction

- Distributed Training with K-FAC

Scaling SGD

- To efficiently leverage the computing power on HPC to reduce the training time, we need to use a large batch size
  - Each processor gets enough data to keep busy
  - Large batch size degrades test accuracy with the constraint of a fixed number of epochs



Scaling SGD

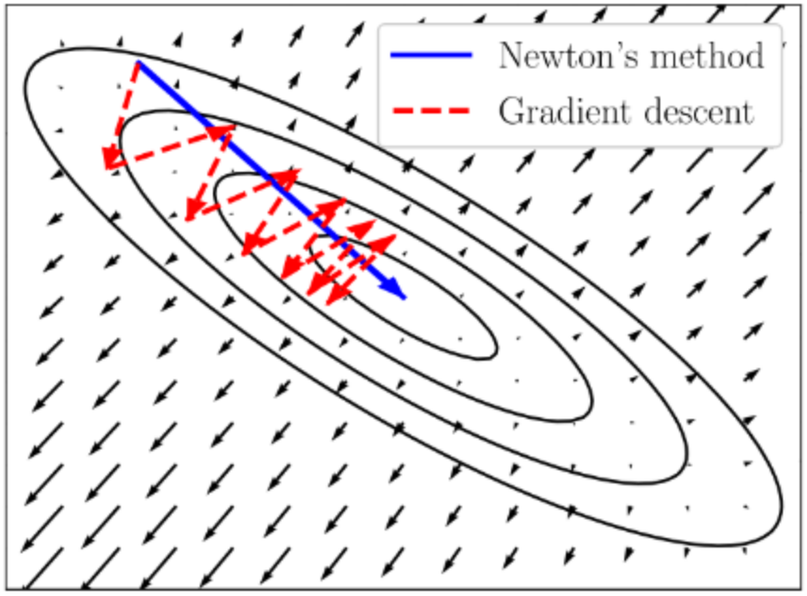
- To address the test-accuracy degradation challenge
  - Learning Rate Linear Scaling
  - Warmup

Observations

- Large batch training is a good candidate for second-order optimization
  - Large batches are more representative of the dataset's distribution
    - Enables second-order update decoupling
  - Gradient noise limits the maximum effective batch size and increases over the course of training (McCandlish, 2018)
    - Second-order methods optimize noise-independent terms better (Martens, 2014)
  - Second-order methods have higher computation-to-communication ratios
    - Enables greater benefits from layer-wise distribution schemes.

Second-order Optimization

- Second-order methods incorporate the curvature of the parameter space.
- Makes more per-iteration progress optimizing the objective function than first-order methods.
- Examples: Gauss-Newton, (L)BFGS, K-FAC
- **Expensive to compute!**



https://www.diva-portal.org/smash/get/diva2:1437676/FULLTEXT01.pdf

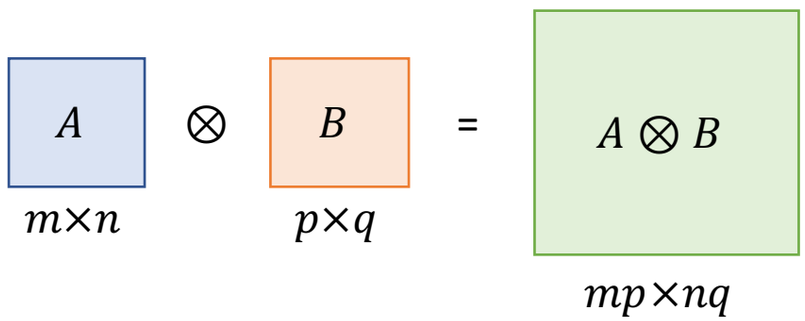
Kronecker-Factored Approximate Curvature

- K-FAC is an efficient approximation of the Fisher Information Matrix (FIM) (Martens+, 2015).
  - The FIM is equivalent to the Generalized Gauss-Newton (GGN) matrix, an approximation of the Hessian.
- Generalizes better with large batches and converges in fewer iterations than SGD (Ba+, 2017).
- Scales to extremely large batch sizes, e.g. 131k for ImageNet training (Osawa+, 2019).

Kronecker Product

Kronecker Product: $\otimes$

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$



$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 5 & 6 \\ 7 & 8 \\ 9 & 0 \end{bmatrix} = \begin{bmatrix} 1\times5 & 1\times6 & 2\times5 & 2\times6 \\ 1\times7 & 1\times8 & 2\times7 & 2\times8 \\ 1\times9 & 1\times0 & 2\times9 & 2\times0 \\ 3\times5 & 3\times6 & 4\times5 & 4\times6 \\ 3\times7 & 3\times8 & 4\times7 & 4\times8 \\ 3\times9 & 3\times0 & 4\times9 & 4\times0 \end{bmatrix}$$

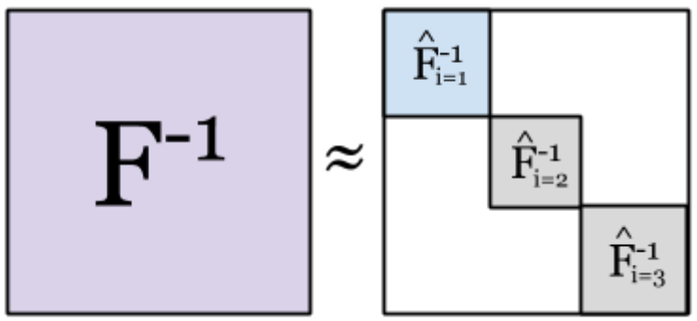
Kronecker-Factored Approximate Curvature

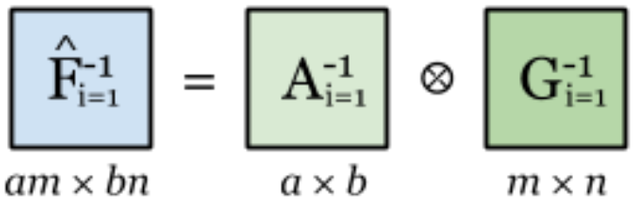
- SGD: $w^{(k+1)} = w^{(k)} - \frac{\alpha^{(k)}}{n} \sum_{i=1}^{n} \nabla L_i(w^{(k)})$
- K-FAC: $w^{(k+1)} = w^{(k)} - \frac{\alpha^{(k)} \boxed{F^{-1}(w^{(k)})}}{n} \sum_{i=1}^{n} \nabla L_i(w^{(k)})$

- $w^{(k)}$ = weight at iteration k
- $\alpha^{(k)}$ = learning rate at iteration k
- n = minibatch size
- $\nabla L_i(w^{(k)})$= gradient of the loss function L_i for the ith example with respect to $w^{(k)}$
- F^{-1}= Inverse Fisher Information Matrix approximation, acts as a gradient preconditioner

Kronecker-Factored Approximate Curvature

- K-FAC approximates the FIM as a block diagonal matrix
  - $F \approx \hat{F} = diag(\hat{F}_1, \ldots, \hat{F}_i, \ldots \hat{F}_L)$
  - $\hat{F}^{-1} = diag(\hat{F}_1^{-1}, \ldots, \hat{F}_i^{-1}, \ldots, \hat{F}_L^{-1})$
- $\hat{F}_i^{-1}$ is a Kronecker product of the activations of the (i-1)th-layer and the gradient w.r.t output of ith-layer.
  - $\hat{F}_i = a_{i-1} a_{i-1}^T \otimes g_i g_i^T = A_{i-1} \otimes G_i$





Kronecker-Factored Approximate Curvature

- Kronecker Product Properties
  - $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$
  - $(A \otimes B)\overrightarrow{c} = B^T \overrightarrow{c} A$

*__Note:__ In practice, $(\hat{F} + \gamma I)^{-1} \nabla L_i(w_i^{(k)})$ is computed to prevent ill-conditioned matrix inversion. γ is a damping constant.

- K-FAC Update Step for Layer i

$$w^{(k+1)} = w^{(k)} - \alpha^{(k)} \hat{F}^{-1} \nabla L_i(w^{(k)})$$

- where*:

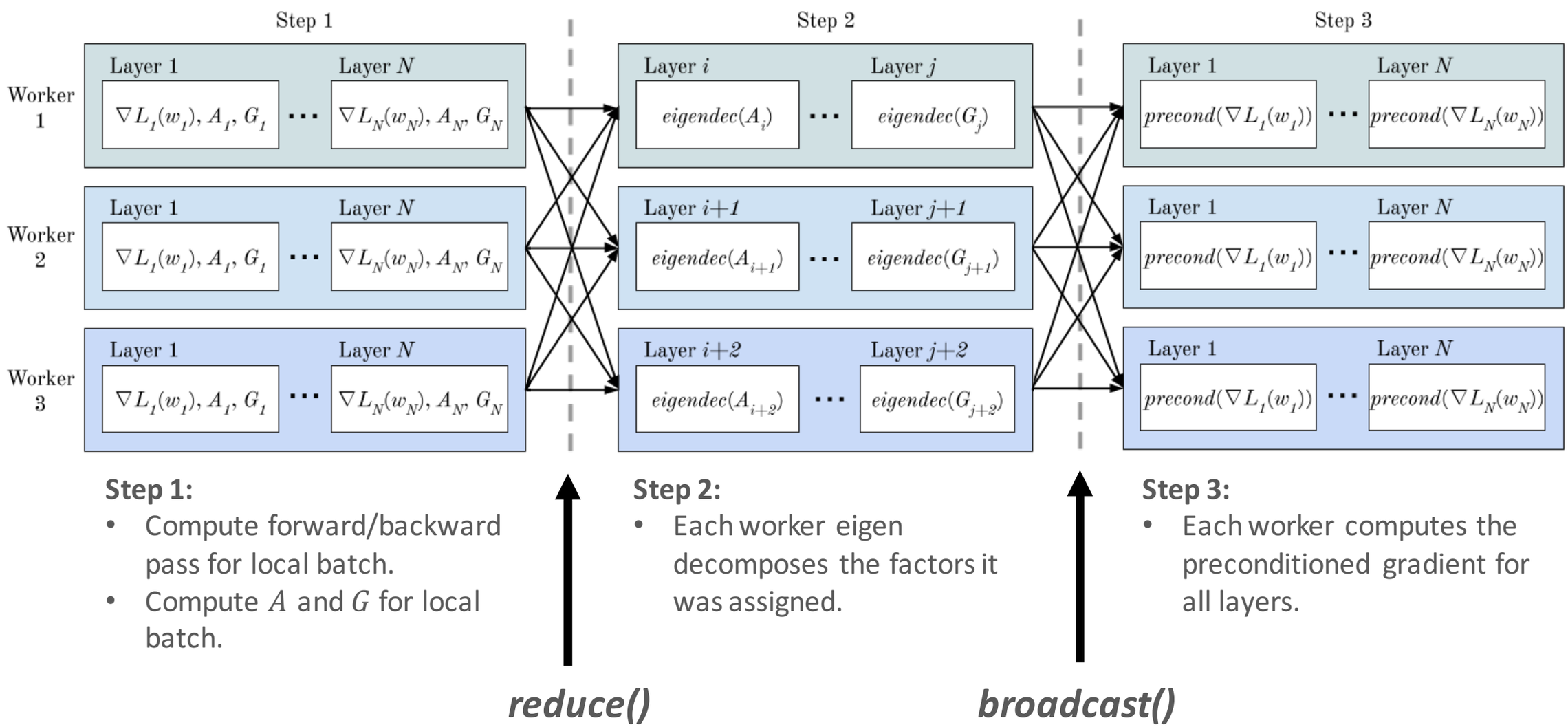
$$\hat{F}^{-1} \nabla L_i(w^{(k)}) = (A_{i-1} \otimes G_i)^{-1} \nabla L_i(w^{(k)})$$

$$\hat{F}^{-1} \nabla L_i(w^{(k)}) = (A_{i-1}^{-1} \otimes G_i^{-1}) \nabla L_i(w^{(k)})$$

$$\hat{F}^{-1} \nabla L_i(w^{(k)}) = \boxed{G_i^{-1} \nabla L_i(w^{(k)}) A_{i-1}^{-1}}$$

Preconditioned Gradient

Design: Parallelism



Step 1:

- Compute forward/backward pass for local batch.
- Compute A and G for local batch.

Step 2:

- Each worker eigen decomposes the factors it was assigned.

Step 3:

- Each worker computes the preconditioned gradient for all layers.

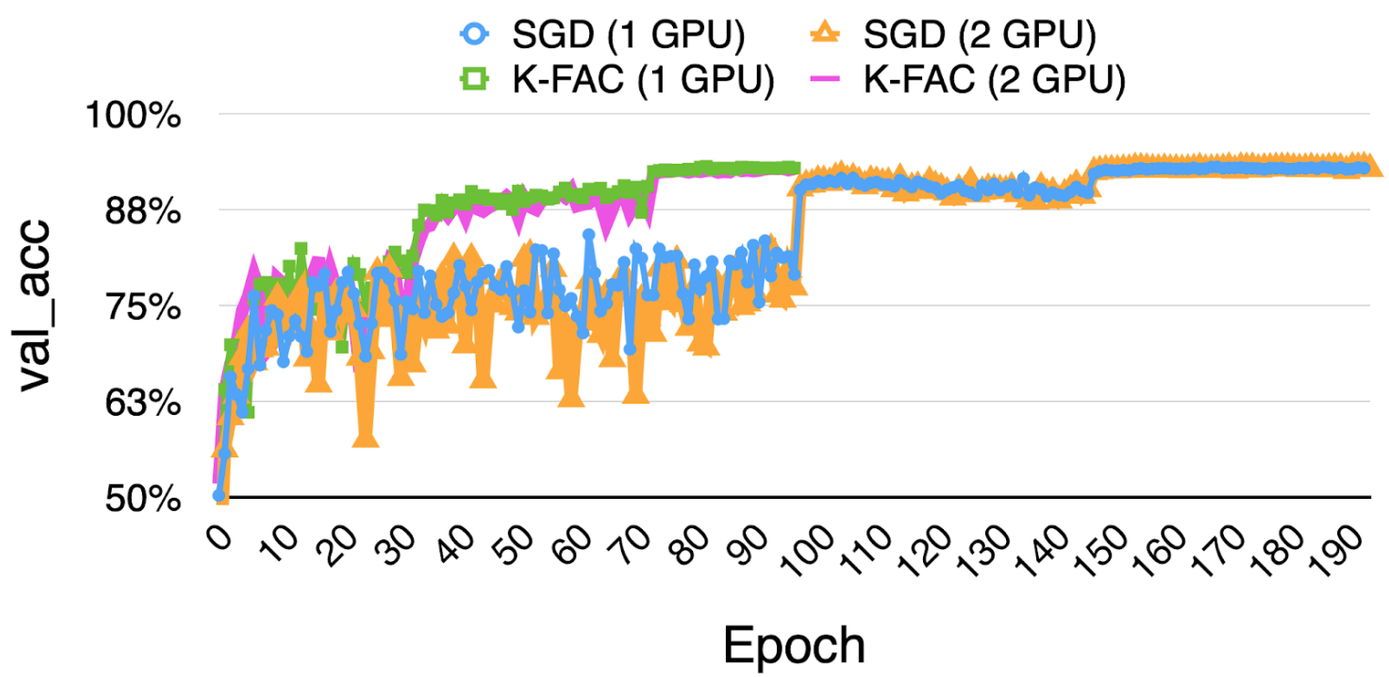
Design: Communication Optimizations

- Tradeoffs:
  - KFAC update steps require more communication (factors + eigendecompositions).
  - Non-KFAC update steps require **only** communicating the gradients, the **same amount of communication as SGD**.
  - As the KFAC update interval increases, the amount of communication required decreases compared to Osawa et al. where communication is constant with update interval size.

Implementation

- K-FAC as a gradient preconditioner to standard PyTorch optimizers.
  - Gradients are modified in-place after the backward pass and before the optimization step.
  - Allows for compatibility with any PyTorch optimizers/ preconditioner (e.g. SGD, Adam, LARS, LAMB, etc.).
  - Supports linear and Conv2D layers.
  - https://github.com/gpauloski/kfac_pytorch
- K-FAC supports Horovod and torch.distributed data-parallel training.
- Automatically registers model and determines communication backend.

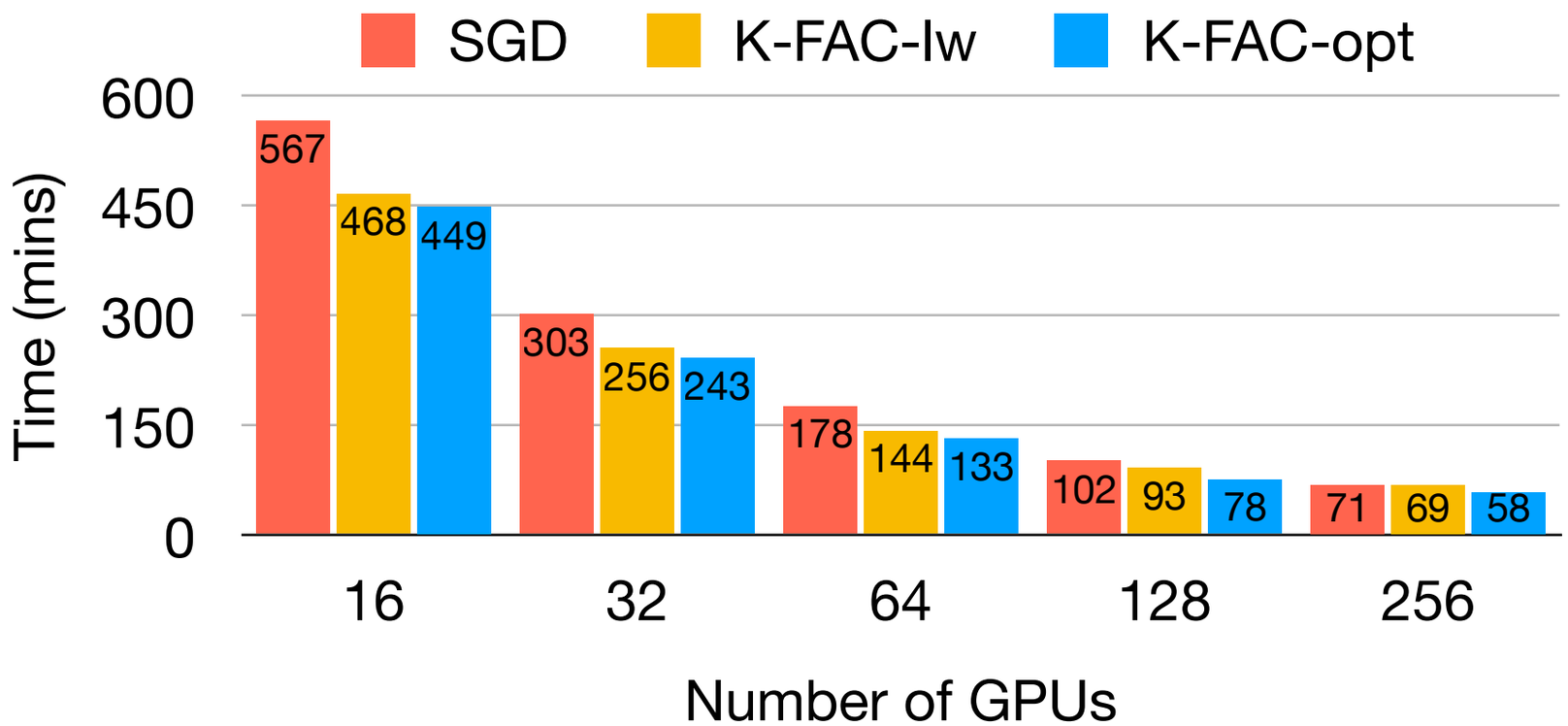
Correctness: Cifar10 + ResNet-32



Validation Accuracy				
GPUs	1	2	4	8
SGD	92.76%	92.77%	92.58%	92.69%
K-FAC	92.93%	92.76%	92.90%	92.92%

Performance: Scaling

- 17-25% improvements of ResNet-50 with ImageNet over SGD across scales



Conclusion

- We introduce an open source, distributed K-FAC preconditioner that is correct, efficient, and scalable.
- Converges to the 75.9% MLPerf ResNet-50 ImageNet baseline 18-25% faster than SGD.

Recommended Practice

- If utilization is low, try increasing the batch size while preserving the test accuracy till the processor is efficiently utilized (either RAM or Compute is saturated)

- While scaling up, try learning rate linear scaling or combine with warmup

- Scale out to multiple processors and multiple nodes with LARS/LAMB/K-FAC

Recommended Practice

- Figure out the default batch size (e.g., from literatures)
- On a single processor, run the training program with default batch size and observe the processor utilization (top for CPU, nvidia-smi for GPU)

top

```
top - 13:57:15 up 24 days, 17:01,  2 users,  load average: 17.23, 18.15, 10.77
Tasks: 697 total,   1 running, 696 sleeping,   0 stopped,   0 zombie
%Cpu(s): 53.7 us,  1.9 sy,  0.0 ni, 44.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 19592022+total, 18991632+free,  4150596 used,  1853308 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 18959068+avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
192773 zzhang    20   0   13.3g   2.1g  68472 S  2616  1.1 319:41.26 python3
```

nvidia-smi

```
Fri Jul 26 13:59:08 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.56       Driver Version: 418.56       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 108...  Off  | 00000000:02:00.0 Off |                  N/A |
|  0%   40C    P0    58W / 250W |      0MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 108...  Off  | 00000000:03:00.0 Off |                  N/A |
|  0%   41C    P0    59W / 250W |      0MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  GeForce GTX 108...  Off  | 00000000:82:00.0 Off |                  N/A |
|  0%   38C    P0    54W / 250W |      0MiB / 11178MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  GeForce GTX 108...  Off  | 00000000:83:00.0 Off |                  N/A |
```

Recent Research in Our Group

- [SC'21] Pauloski, J. Gregory, Qi Huang, Lei Huang, Shivaram Venkataraman, Kyle Chard, Ian Foster, and Zhao Zhang. "**KAISA: An Adaptive Second-order Optimizer Framework for Deep Neural Networks.**" To appear in SC21: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2021
- [SC'20] J. G. Pauloski, Z. Zhang, L. Huang, W. Xu, I. T. Foster. "**Convolutional Neural Network Training with Distributed K-FAC.**" In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-12. IEEE, 2020.
- [IPDPS'20] Z. Zhang, L. Huang, J. G. Pauloski, I. T. Foster. "**Efficient I/O for Neural Network Training with Compressed Data.**" In 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 409-418. IEEE, 2020.
- [CLUSTER'19] Z. Zhang, L. Huang, R. Huang, W. Xu, D. S. Katz. "**Quantifying the Impact of Memory Errors in Deep Learning.**" In 2019 IEEE International Conference on Cluster Computing (CLUSTER), p.1. IEEE, 2019.
- [TPDS'19] Y. You, Z. Zhang, J. Demmel, K. Keutzer, C. Hsieh. "**Fast Deep Neural Network Training on Distributed Systems and Cloud TPUs.**" in IEEE Transactions on Parallel and Distributed Systems (2019).
- [ICPP'18] Y. You, Z. Zhang, J. Demmel, K. Keutzer, C. Hsieh. "**ImageNet Training in Minutes.**" In Proceedings of the 47th International Conference on Parallel Processing, p. 1. ACM, 2018. Best Paper Award.
- [HPDC'17] Zhang, Zhao, Evan R. Sparks, and Michael J. Franklin. "**Diagnosing machine learning pipelines with fine-grained lineage.**" In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, pp. 143-153. 2017.

Summary

- AI in Science

- Distributed Deep Learning
  - PyTorch Introduction

- Distributed Training with K-FAC

Environment Setup

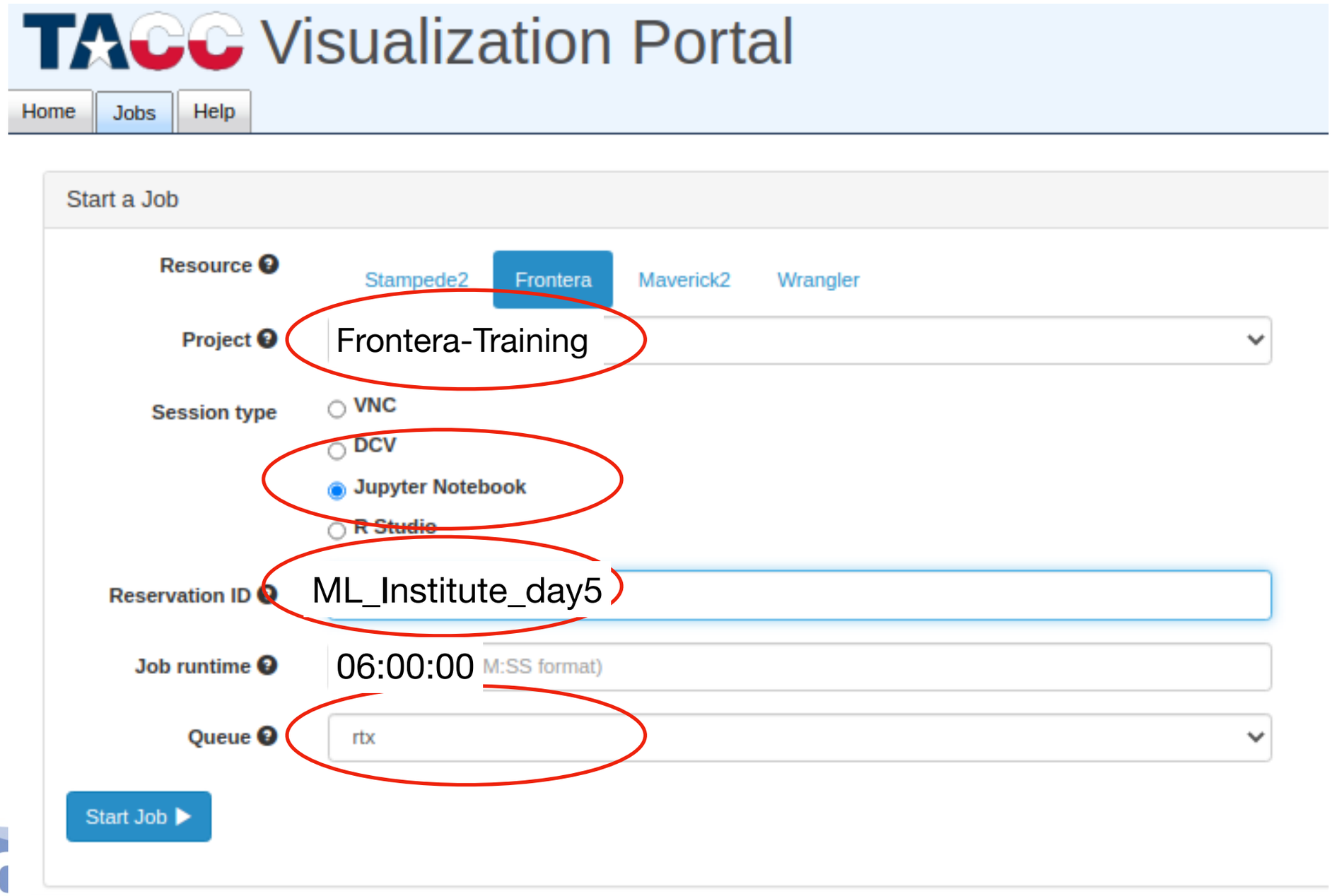
- idev -N 1 -n 1 -t 02:00:00 -p rtx -R
- bash /home1/00946/zzhang/ML-Institute-2021-PyTorch/env.sh
- exit

zzhang@staff.frontera:~ ⌥⌘1

```
staff.frontera(1024)$ idev -N 1 -n 1 -t 02:00:00 -p rtx -R

 NOTE: "->" are idev statements. "-->" are TACC/SLURM filter statements.

 -> Reservation      : Will try to find a reservation set for you.
 -> Found a single ACTIVE reservation, ML_Institute_day4, for you.


 -> Checking on the status of rtx queue. OK

 -> Defaults file    : ~/.idevrc
 -> System           : frontera
 -> Reservation name : ML_Institute_  (reservation ACTIVE  )
 -> Queue            : rtx            (reservation         )
 -> Nodes            : 1              (cmd line: -N        )
 -> Total tasks      : 1              (cmd line: -n        )
 -> Time (hh:mm:ss)  : 02:00:00       (cmd line: -t        )
 -> Project          : TACC-DIC       (~/.idevrc           )

---------------------------------------------------------------
          Welcome to the Frontera Supercomputer
---------------------------------------------------------------

--> Verifying valid submit host (staff)...OK
--> Verifying valid jobname...OK
--> Verifying valid ssh keys...OK
--> Verifying access to desired queue (rtx)...OK
--> Checking available allocation (TACC-DIC)...OK
--> Verifying that quota for filesystem /home1/00946/zzhang is at 80.73% allocated...OK
--> Verifying that quota for filesystem /work2/00946/zzhang/frontera is at 18.45% allocated...OK
Submitted batch job 3388237

 -> After your idev job begins to run, a command prompt will appear,
 -> and you can begin your interactive development session.
 -> We will report the job status every 4 seconds: (PD=pending, R=running).
```

Starting Jupyter

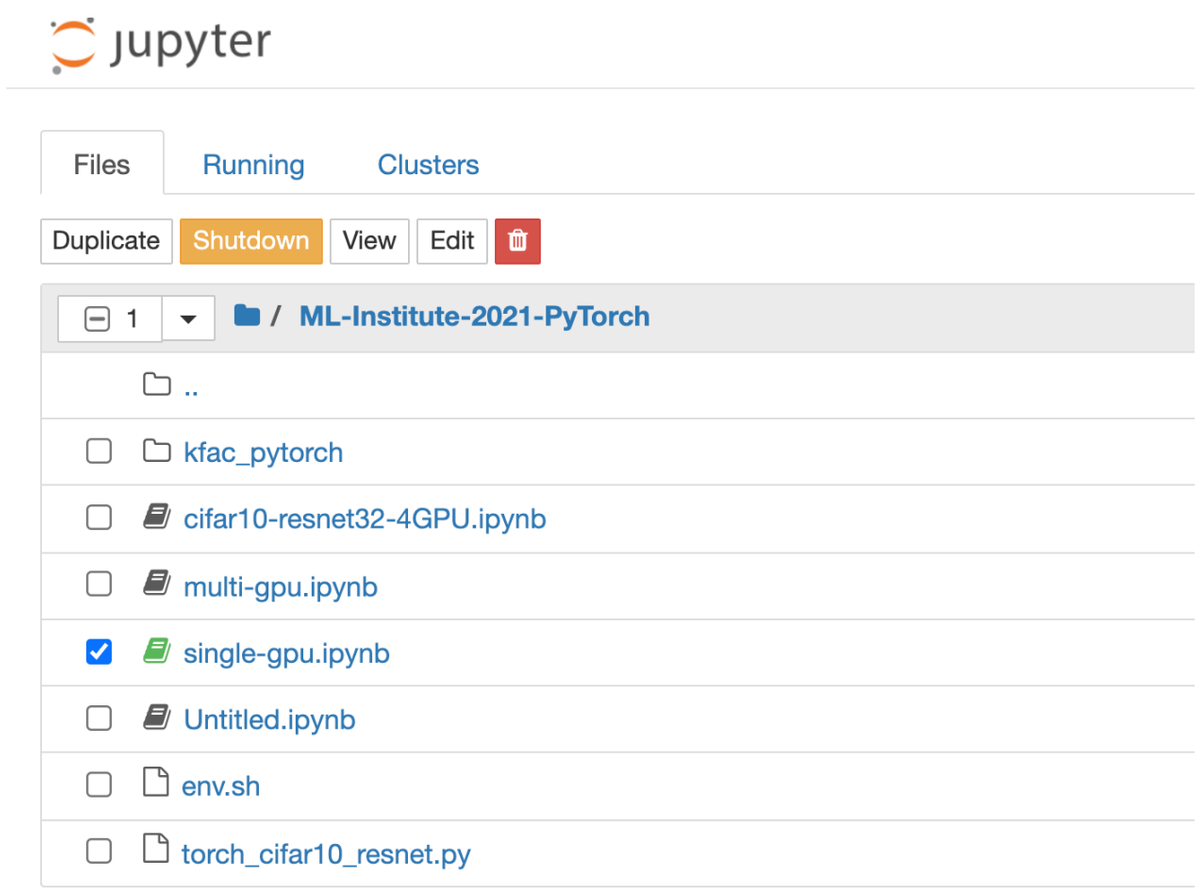
- Go to https://vis.tacc.utexas.edu
- Login with your training account credentials



Hands-on

- Go to ML-Institute-2021-PyTorch/

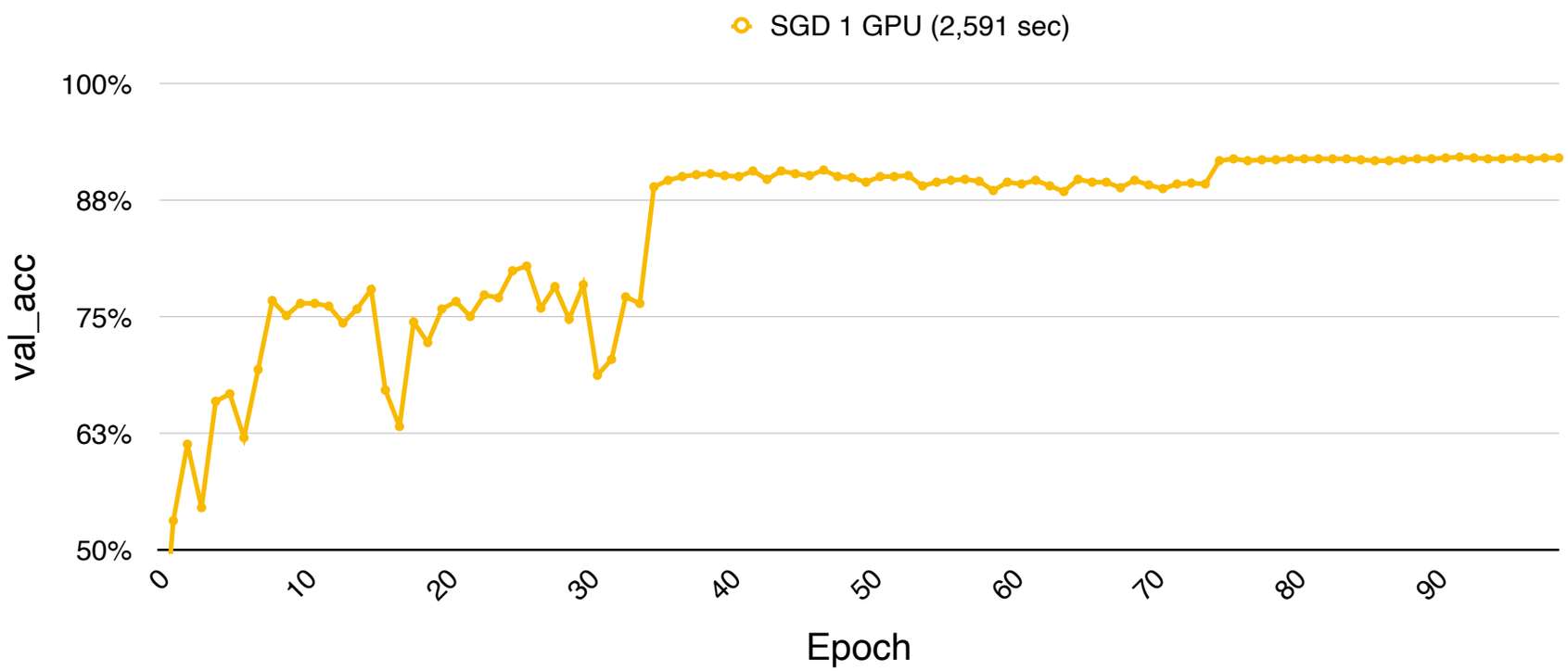
- Open single-gpu.ipynb



Hands-on

- Run single-gpu.ipynb

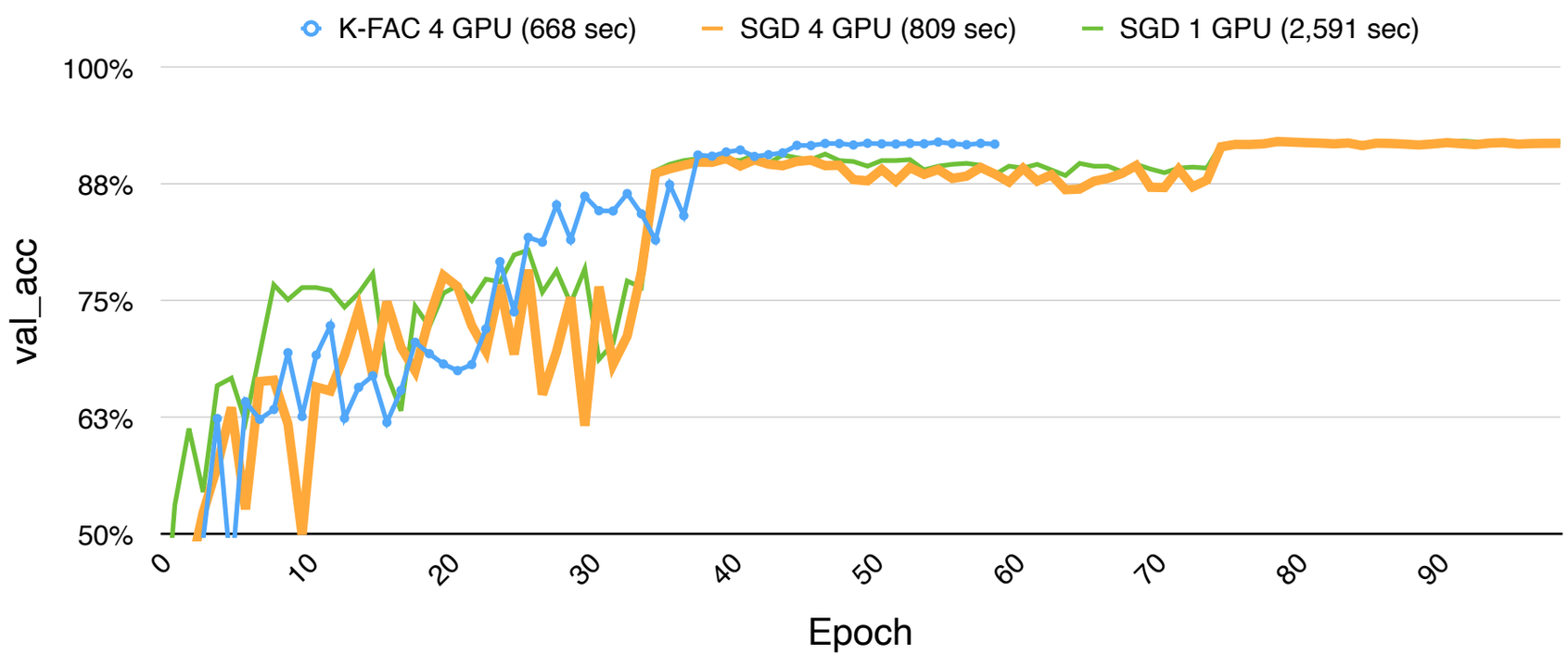
- Collect test_accuracy and plot it



Hands-on

- Shutdown single-gpu.ipynb
- Open cifar10-resnet32-4GPU.ipynb
- Run Cell [1] — SGD with 4 GPUs
- Run Cell [2]
- Run Cell [3] — K-FAC with 4 GPUs

Hands-on


K-FAC 4 GPU (668 sec)
SGD 4 GPU (809 sec)
SGD 1 GPU (2,591 sec)
100%
88%
75%
63%
50%
val_acc
0
10
20
30
40
50
60
70
80
90
Epoch