CASE SCHOOL
OF ENGINEERING
CASE WESTERN RESERVE
UNIVERSITY

CSDS 451: Designing High Performant Systems for AI

Lecture 14

10/14/2025

Sanmukh Kuppannagari

sanmukh.kuppannagari@case.edu

https://sanmukh.research.st/

Case Western Reserve University

CASE SCHOOL
OF ENGINEERING
CASE WESTERN RESERVE
UNIVERSITY

Outline

- Distributed Training Basics
- Pytorch Lightning

CASE SCHOOL
OF ENGINEERING
CASE WESTERN RESERVE
UNIVERSITY

Outline

- Distributed Training Basics
- Pytorch Lightning

Training

$$E(W): \min_{W} \sum (y_i - F(x_i : W))\text{^}2$$

$$W_{t+1} \leftarrow W_t - \alpha \frac{\delta E(W)}{\delta W}$$

Iteratively

$$\frac{\delta E(W)}{\delta W} = 2 \sum_i \left(y_i - F(x_i : W)\right) \times \frac{\delta F(x_i : W)}{\delta W}$$

For all samples, in each iteration

Error

Training

L layers in CNN

$$W_1, W_2, W_3, \dots, W_L$$

Weights (Filters/Kernels)

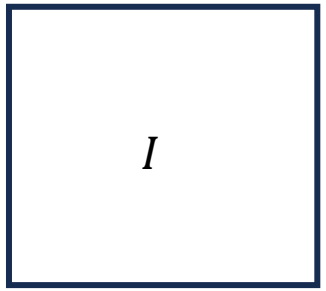
$$O_1, O_2, O_3, \dots, O_L$$

Output of layer

$$2 * Error * \frac{\delta F(x_i : W_{1:L})}{\delta W_1}, \frac{\delta F(x_i : W_{1:L})}{\delta W_2}, \dots, \frac{\delta F(x_i : W_{1:L})}{\delta W_L}$$

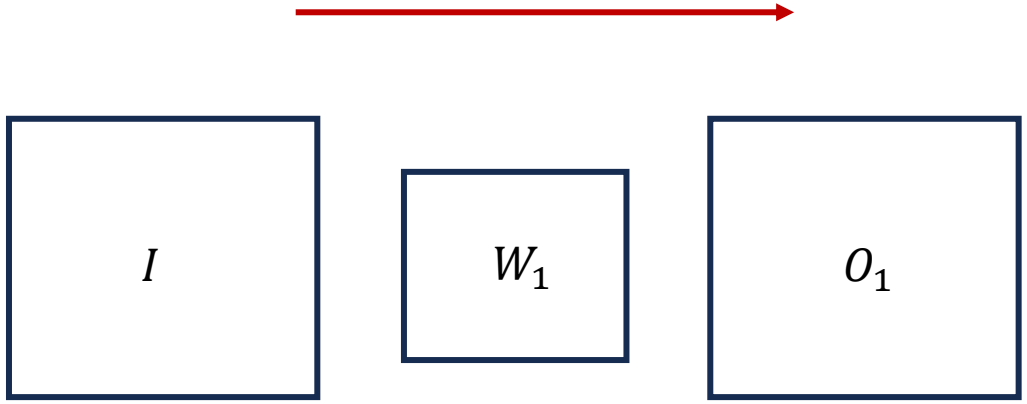
Compute these for all the samples

CNN Training: Forward Propagation



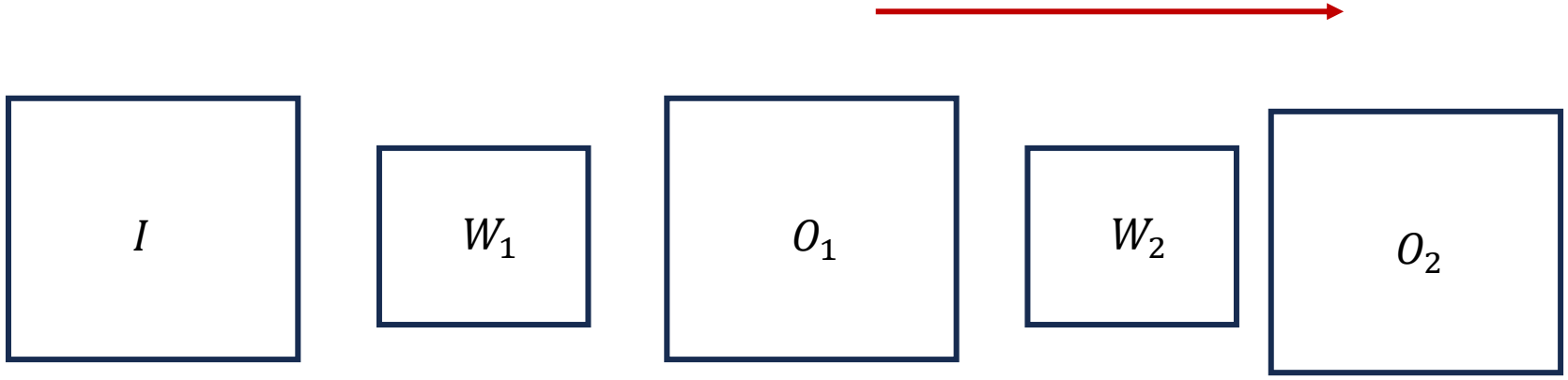
For a single image

CNN Training: Forward Propagation



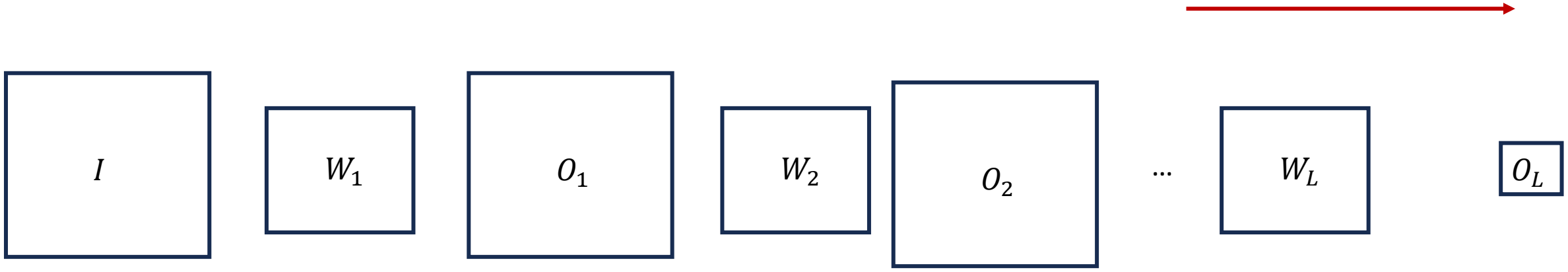
For a single image

CNN Training: Forward Propagation



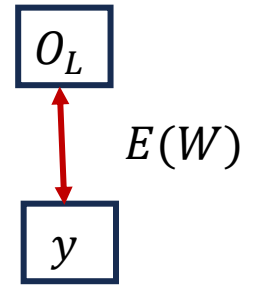
For a single image

CNN Training: Forward Propagation



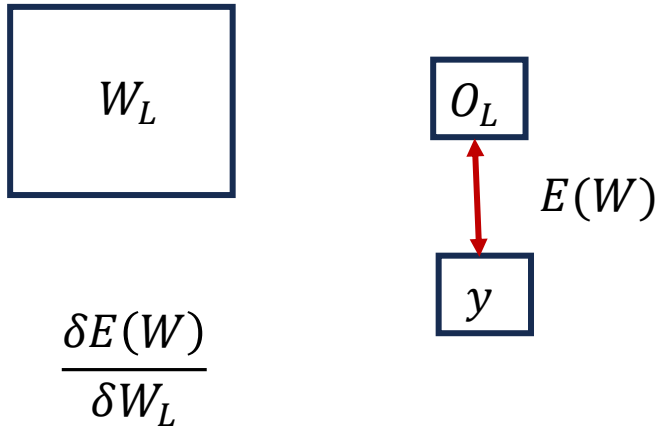
For a single image

CNN Training: Error Calculation



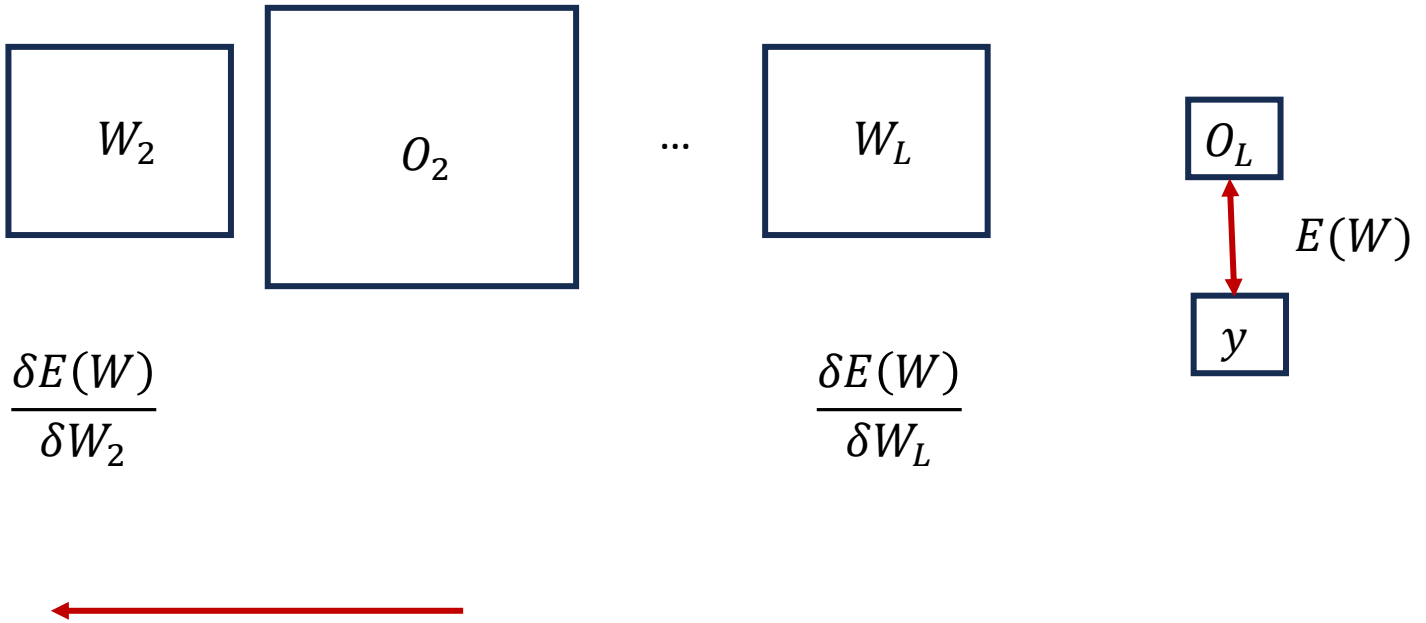
For a single image

CNN Training: Backward Propagation



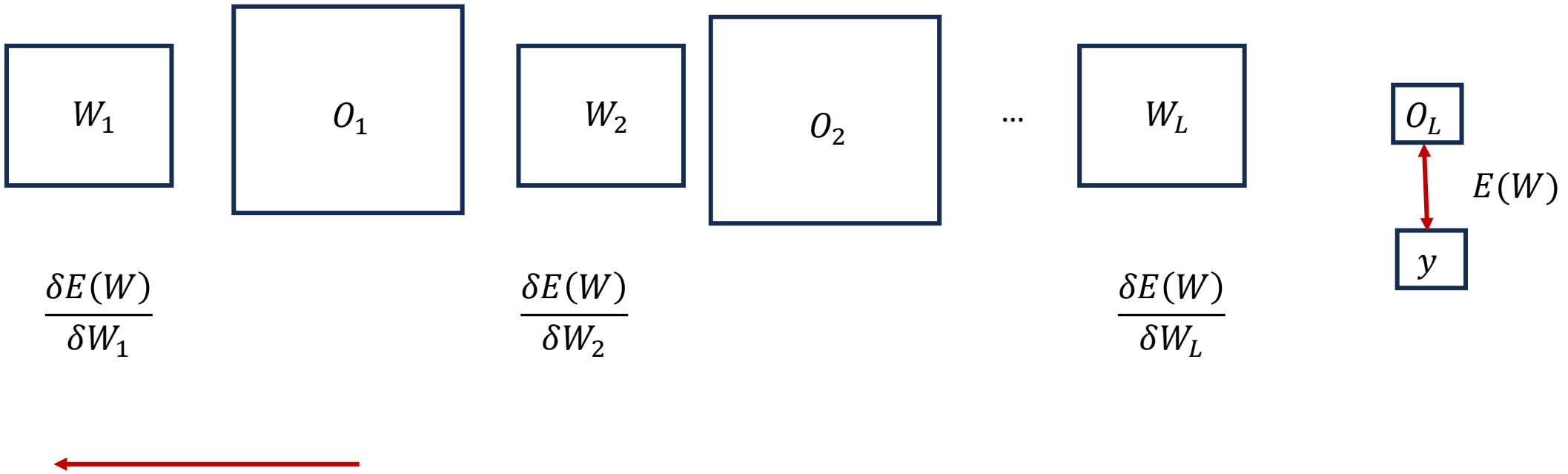
For a single image

CNN Training: Backward Propagation



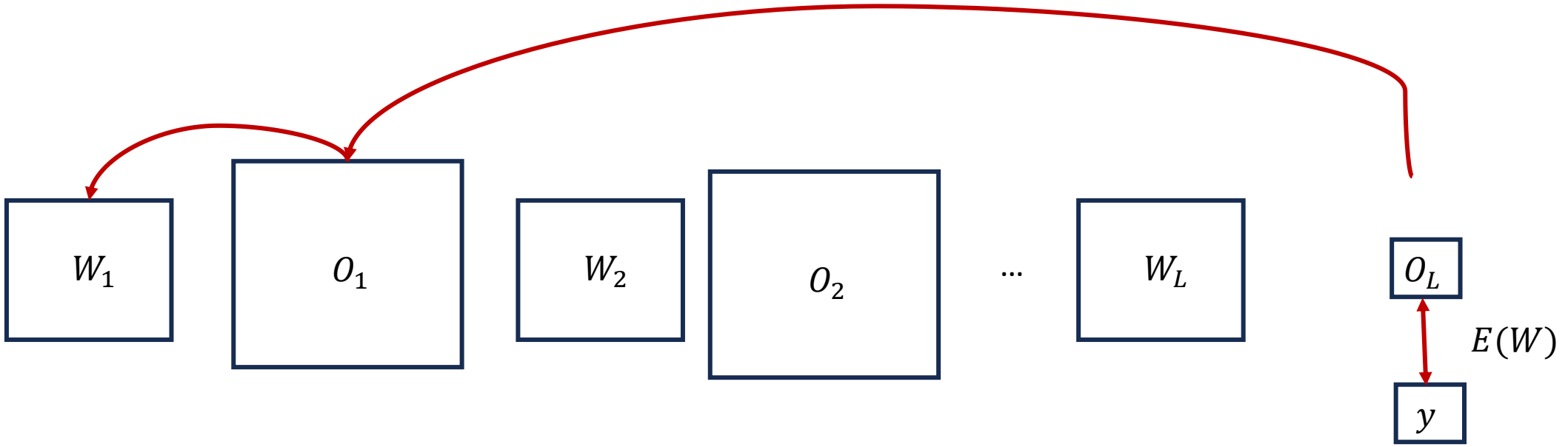
For a single image

CNN Training: Backward Propagation



For a single image

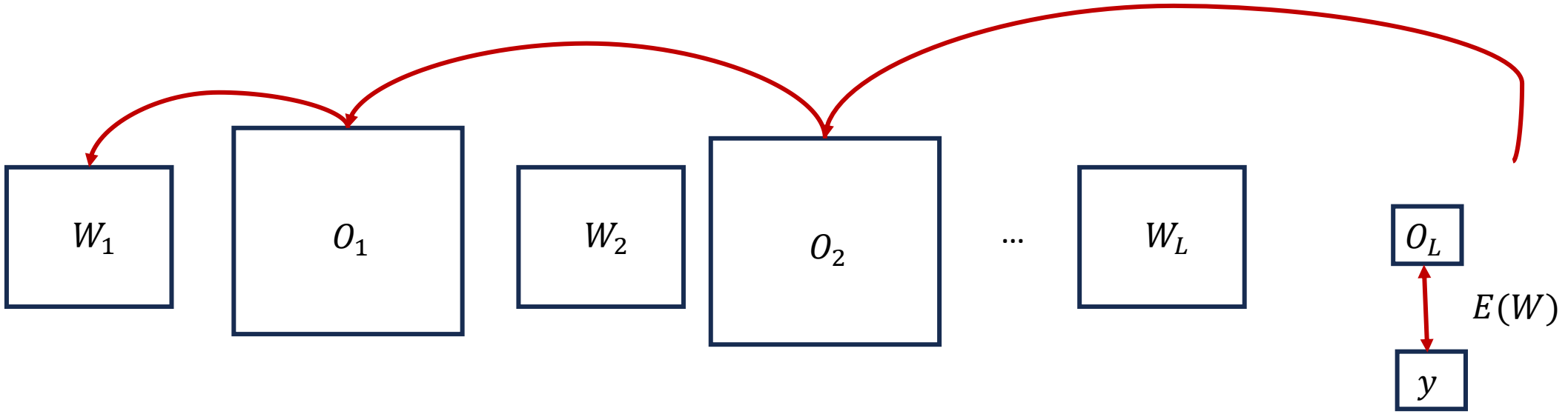
CNN Training: Backward Propagation



$$\frac{\delta E(W)}{\delta W_1} = \frac{\delta O_1}{\delta W_1} \times \frac{\delta E(W)}{\delta O_1}$$

For a single image

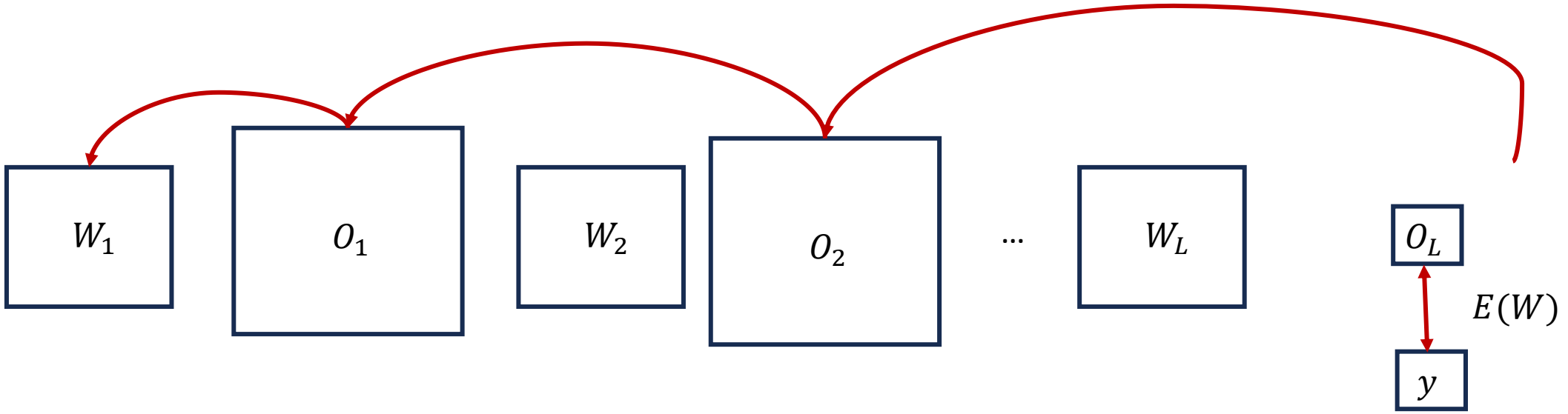
CNN Training: Backward Propagation



$$\frac{\delta E(W)}{\delta W_1} = \frac{\delta O_1}{\delta W_1} \times \frac{\delta O_2}{\delta O_1} \times \frac{\delta E(W)}{\delta O_2}$$

For a single image

CNN Training: Backward Propagation



$$\frac{\delta E(W)}{\delta W_1} = \frac{\delta O_1}{\delta W_1} \times \frac{\delta O_2}{\delta O_1} \times \frac{\delta O_3}{\delta O_2} \times \cdots \times \frac{\delta O_L}{\delta O_{L-1}} \times \frac{\delta E(W)}{\delta O_L}$$

For a single image

Training

$$E(W): \min_{W} \sum (y_i - F(x_i: W))\text{^}2$$

$$W_{t+1} \leftarrow W_t - \alpha \frac{\delta E(W)}{\delta W}$$

Iteratively

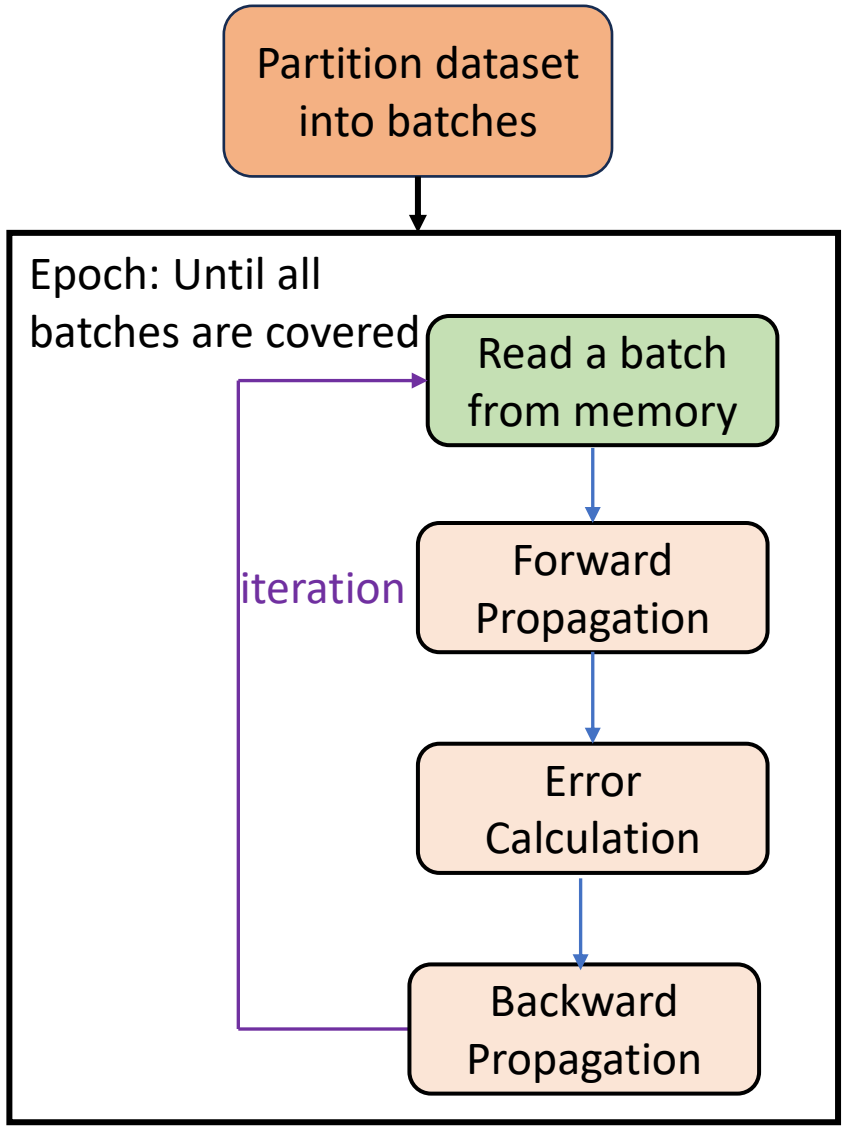
$$\frac{\delta E(W)}{\delta W} = 2\sum \underbrace{\left(y_i - F(x_i: W)\right)}_{\text{Error}} \times \frac{\delta F(x_i: W)}{\delta W}$$

For all samples, in each iteration

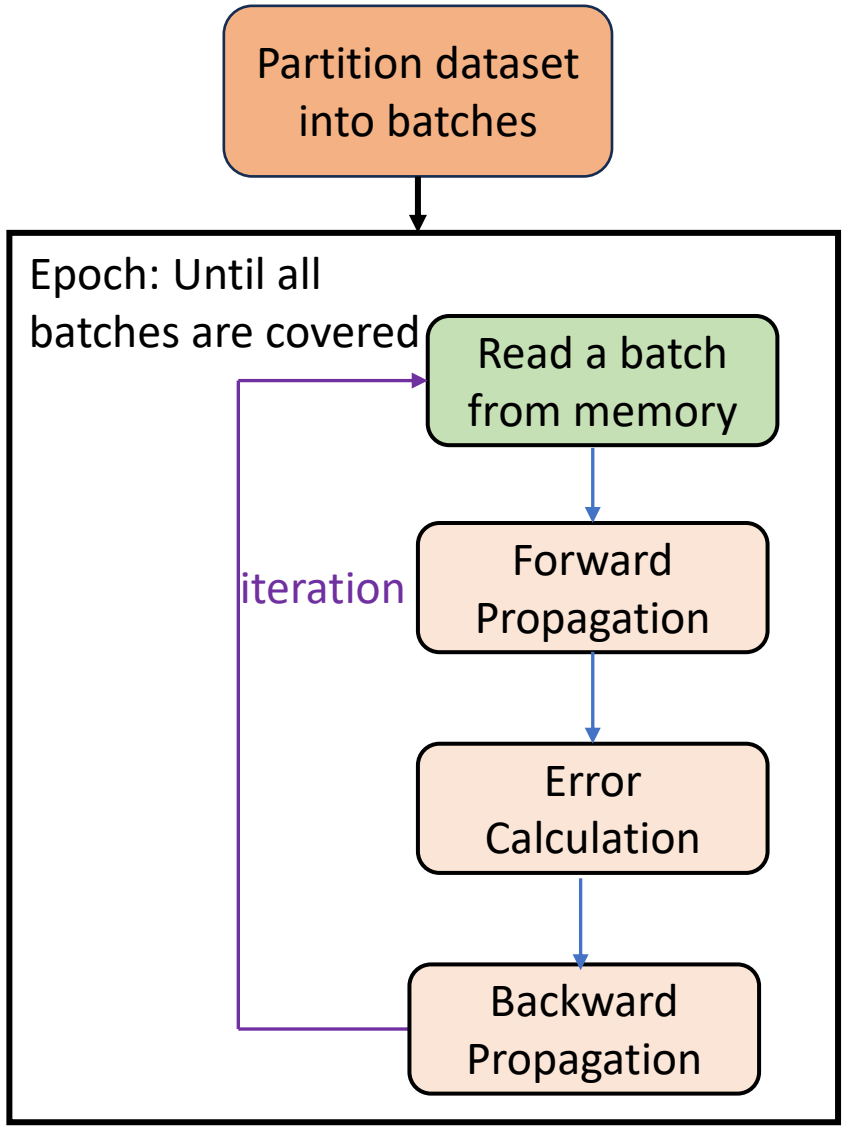
$$\frac{\delta E(W)}{\delta W_l} = \boxed{\frac{\delta O_l}{\delta W_l} \times \frac{\delta O_{l+1}}{\delta O_l} \times \frac{\delta O_{l+2}}{\delta O_{l+1}} \times \cdots \times \frac{\delta O_L}{\delta O_{L-1}}} \times \boxed{\frac{\delta E(W)}{\delta O_L}}$$

For multiple images in a batch: Simply sum up the gradients

Training a Machine Learning Model



Data Parallel Distributed Training



- Mini batches further partitioned in independent subsets.
- Each subset performs the iteration in parallel
- Need synchronization???

Gradient Update

$$W \leftarrow W \ - \alpha \frac{\delta E(W)}{\delta W}$$

- Lets dig deeper into Gradient Update

Gradient Update

- For a mini-batch with M samples and mean squared error
- $E(W) = \sum_{i=1}^{M} E_i(W)$
- $E_i(w) = (\mathrm{O_{Li}} - \mathrm{y_i})^2$
- $\frac{\delta E(W)}{\delta W} = \sum_{i=1}^{M} \frac{\delta E_i(W)}{\delta W}$

Gradient Update

$$\frac{\delta E(W)}{\delta W} = \frac{\delta E_1(W)}{\delta W} + \frac{\delta E_2(W)}{\delta W} + \ \ldots + \frac{\delta E_M(W)}{\delta W}$$

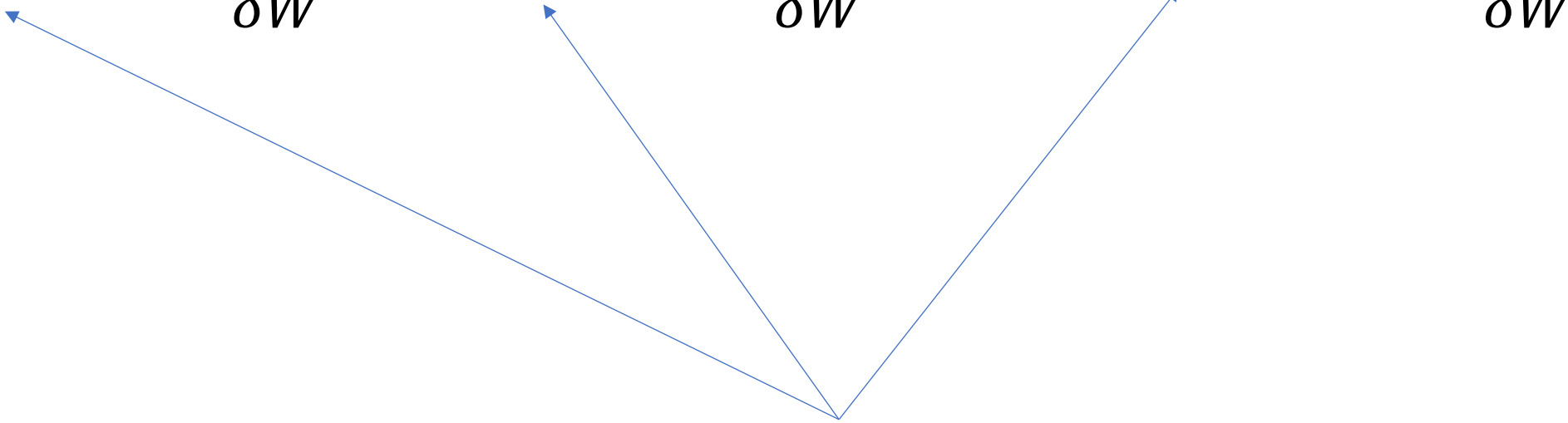
$$= \underbrace{2(\mathrm{O_{L1}} - \mathrm{y_1})\frac{\delta O_{L1}}{\delta W}} + \underbrace{2(\mathrm{O_{L2}} - \mathrm{y_2})\frac{\delta O_{L2}}{\delta W}} + \ \ldots + \underbrace{2(\mathrm{O_{LM}} - \mathrm{y_M})\frac{\delta O_{LM}}{\delta W}}$$

Can be calculated in parallel

Gradient Update

$$\frac{\delta E(W)}{\delta W} = \frac{\delta E_1(W)}{\delta W} + \frac{\delta E_2(W)}{\delta W} + \ \ldots + \frac{\delta E_M(W)}{\delta W}$$

$$= 2(\mathrm{O_{L1}} - \mathrm{y_1})\frac{\delta O_{L1}}{\delta W} + 2(\mathrm{O_{L2}} - \mathrm{y_2})\frac{\delta O_{L2}}{\delta W} + \ \ldots + 2(\mathrm{O_{LM}} - \mathrm{y_M})\frac{\delta O_{LM}}{\delta W}$$

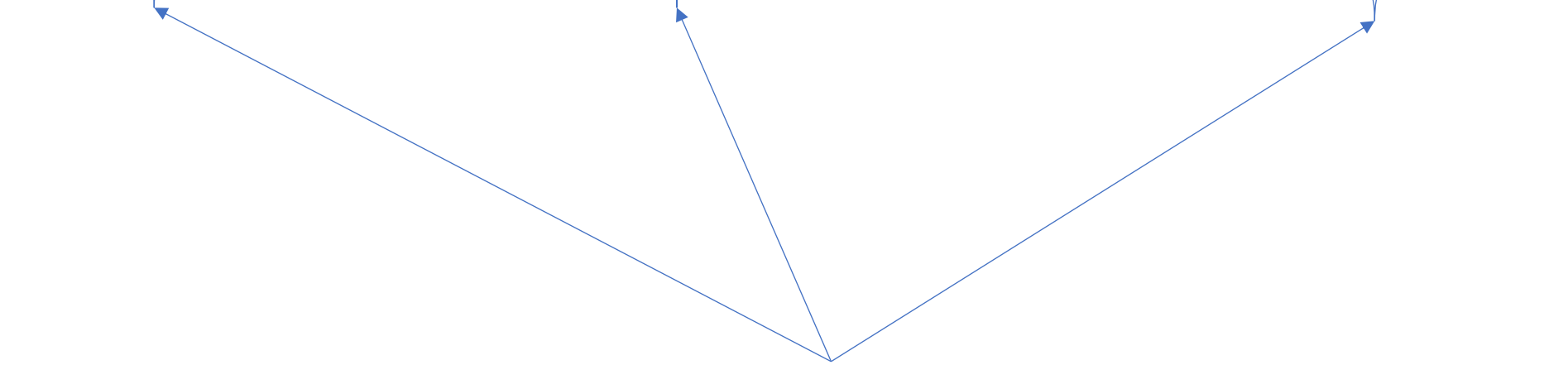


Forward propagation in parallel – Need multiple copies of model

Gradient Update

$$\frac{\delta E(W)}{\delta W} = \frac{\delta E_1(W)}{\delta W} + \frac{\delta E_2(W)}{\delta W} + \ \ldots + \frac{\delta E_M(W)}{\delta W}$$

$$= 2(\mathrm{O_{L1}} - \mathrm{y_1})\frac{\delta O_{L1}}{\delta W} + 2(\mathrm{O_{L2}} - \mathrm{y_2})\frac{\delta O_{L2}}{\delta W} + \ \ldots + 2(\mathrm{O_{LM}} - \mathrm{y_M})\frac{\delta O_{LM}}{\delta W}$$

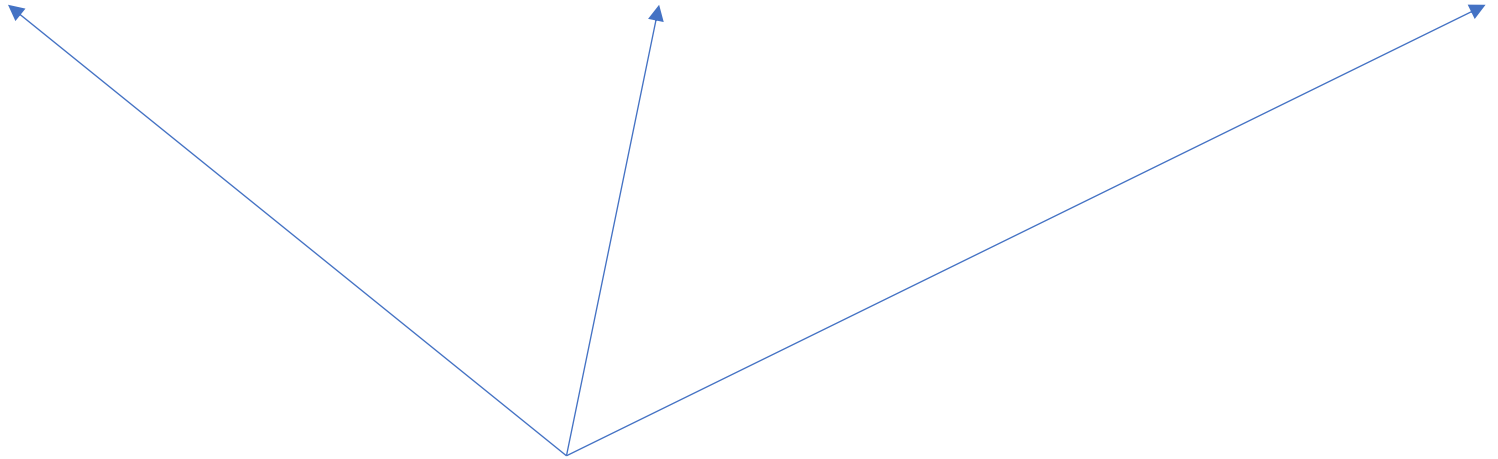


Error Calculation in parallel

Gradient Update

$$\frac{\delta E(W)}{\delta W} = \frac{\delta E_1(W)}{\delta W} + \frac{\delta E_2(W)}{\delta W} + \ \ldots + \frac{\delta E_M(W)}{\delta W}$$

$$= 2(\mathrm{O_{L1}} - \mathrm{y_1})\frac{\delta O_{L1}}{\delta W} + 2(\mathrm{O_{L2}} - \mathrm{y_2})\frac{\delta O_{L2}}{\delta W} + \ \ldots + 2(\mathrm{O_{LM}} - \mathrm{y_M})\frac{\delta O_{LM}}{\delta W}$$



Backward Propagation in parallel

Gradient Update

- If using p processes (or GPUs) for a mini-batch of size M
- Each process (or GPU) keeps a copy of the model W
- Each process performs the computations of M/p samples independently using its own model
- Note: No synchronization was needed till now

Gradient Update

- Weight Update Step:

$$W \leftarrow W - \alpha \frac{\delta E(W)}{\delta W}$$

- No synchronization needed till now.

Gradient Update

- Weight Update Step:

$$W \leftarrow W - \alpha \frac{\delta E(W)}{\delta W}$$

- Do we need synchronization now???

Gradient Update

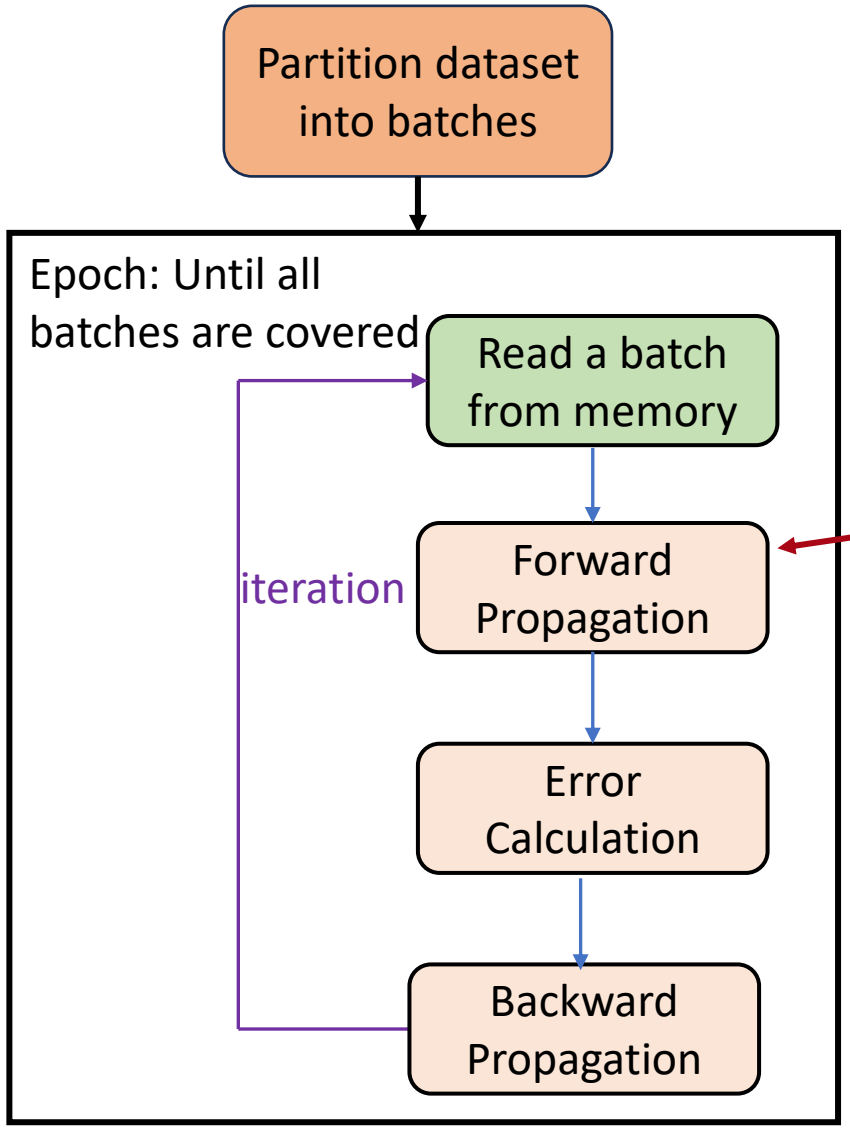
- Weight Update Step:

$$W \leftarrow W \ - \alpha \frac{\delta E(W)}{\delta W}$$

- Do we need synchronization now??? Yes.

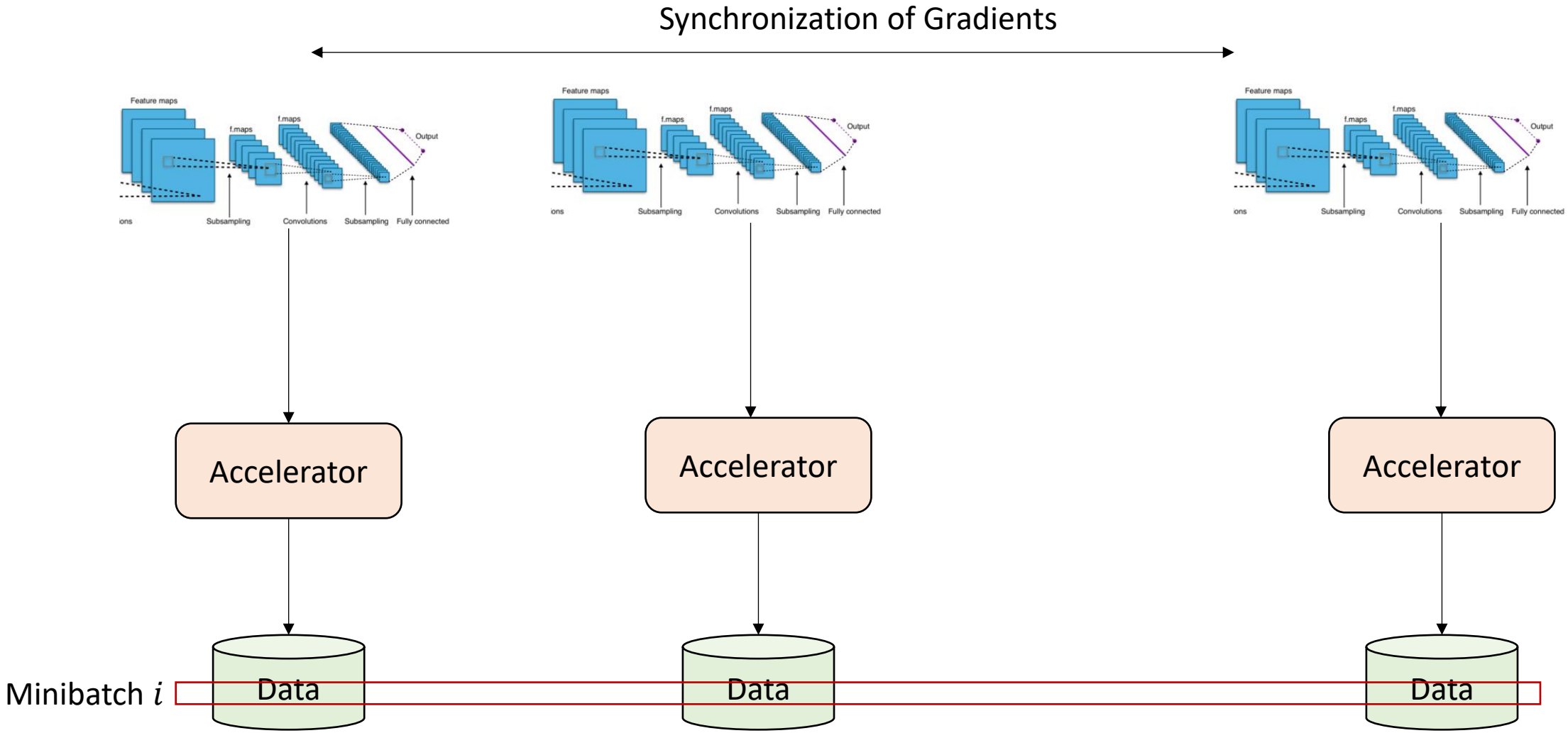
- Need to make sure that all GPUs have the same weights before they start processing the next mini-batch

Data Parallel Distributed Training



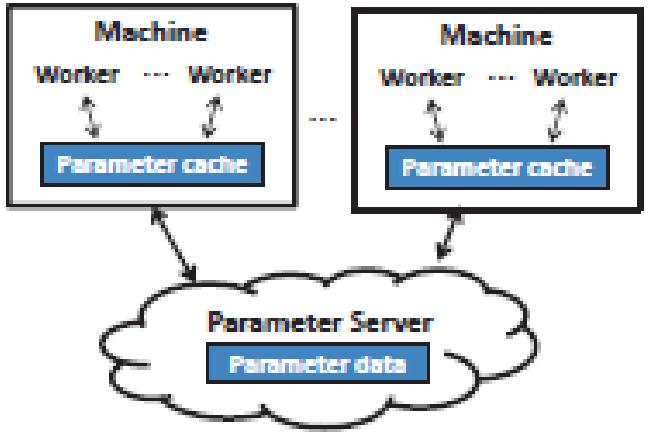
After each iteration, all parallel processors should use the same model weights to do the forward propagation on their portion of mini-batch

Data Parallel Training of Neural Networks



Data Parallel Training of Neural Networks

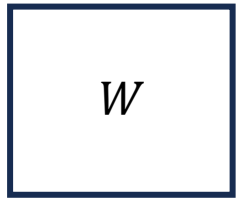
- How do we synchronize?
- Model #1: Parameter Server Model
- A dedicated server performs the weight updates and distributes the models to all the workers
- Resource: Li, Mu, et al. "Scaling distributed machine learning with the parameter server." *11th USENIX Symposium on operating systems design and implementation (OSDI 14)*. 2014.



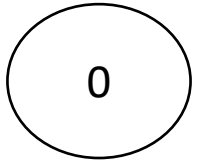
Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., & Xing, E. P. (2016, April). Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the eleventh european conference on computer systems* (pp. 1-16).

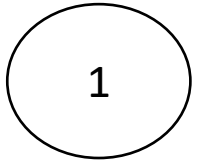
Parameter Server Model

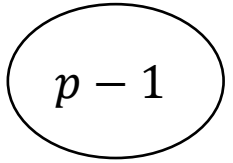
Parameter Server



Data Parallel GPUs



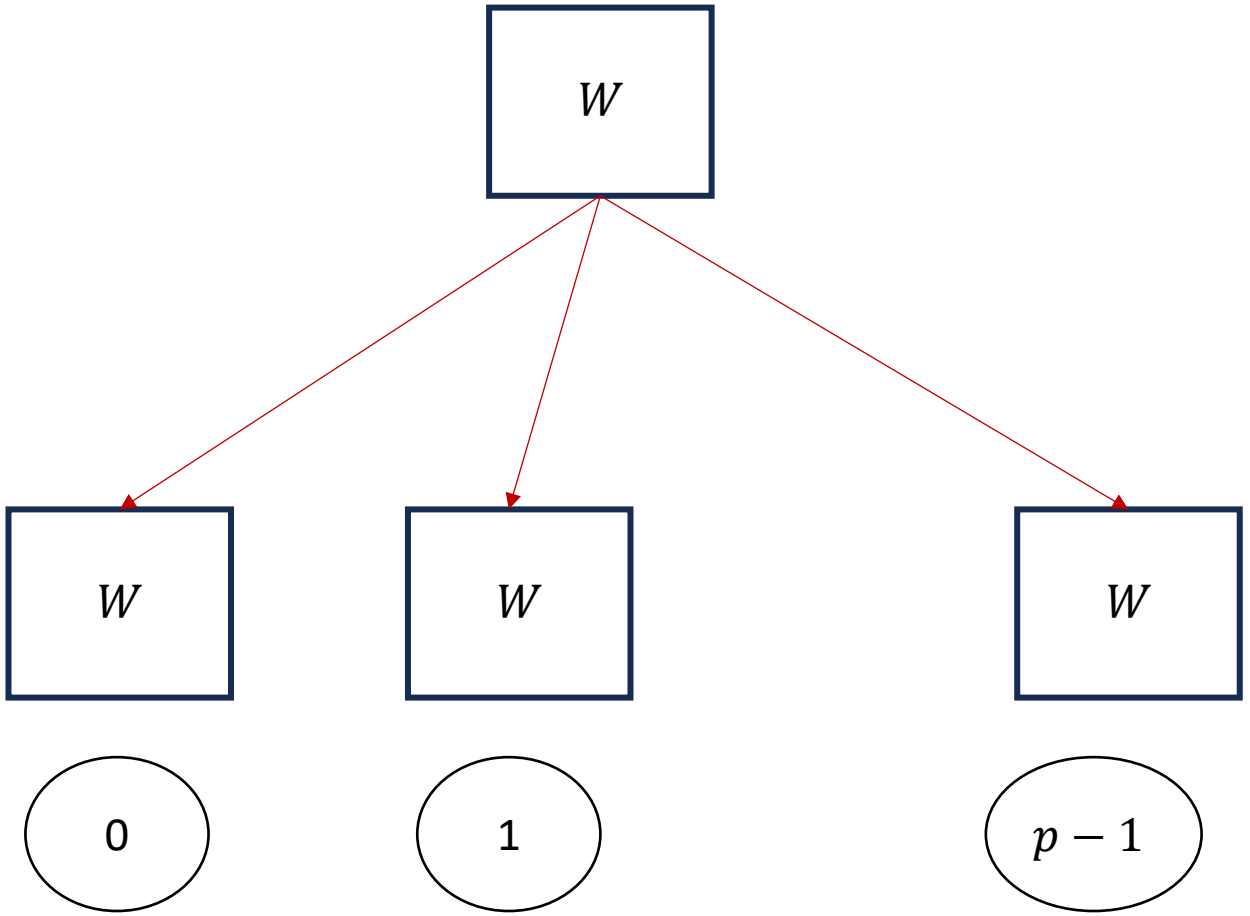




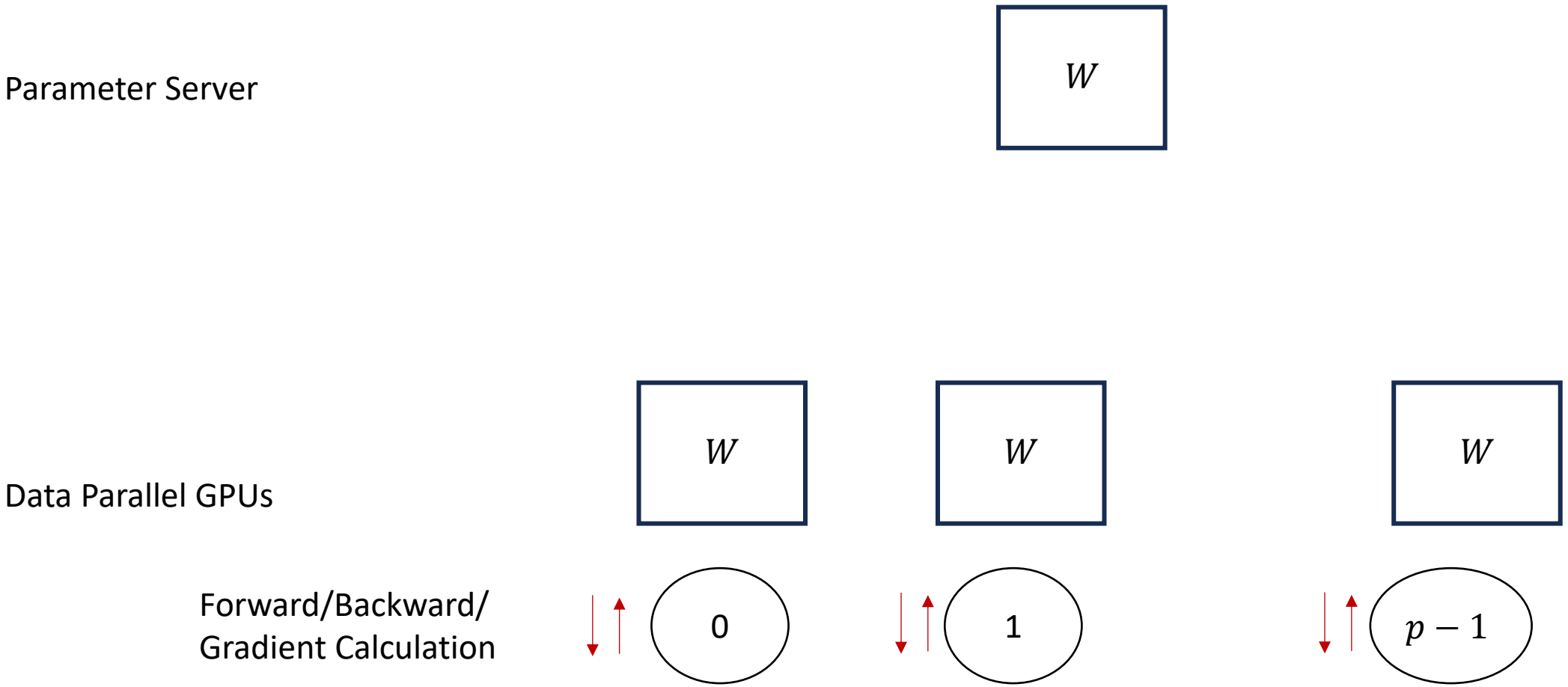
Parameter Server Model

Parameter Server

Data Parallel GPUs

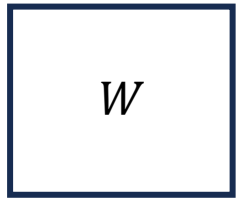


Parameter Server Model

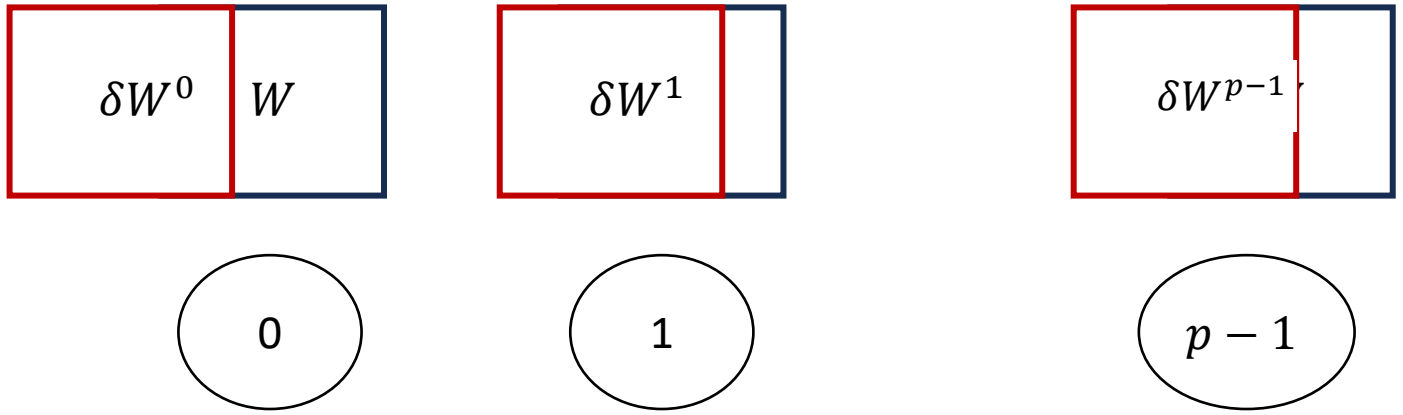


Parameter Server Model

Parameter Server



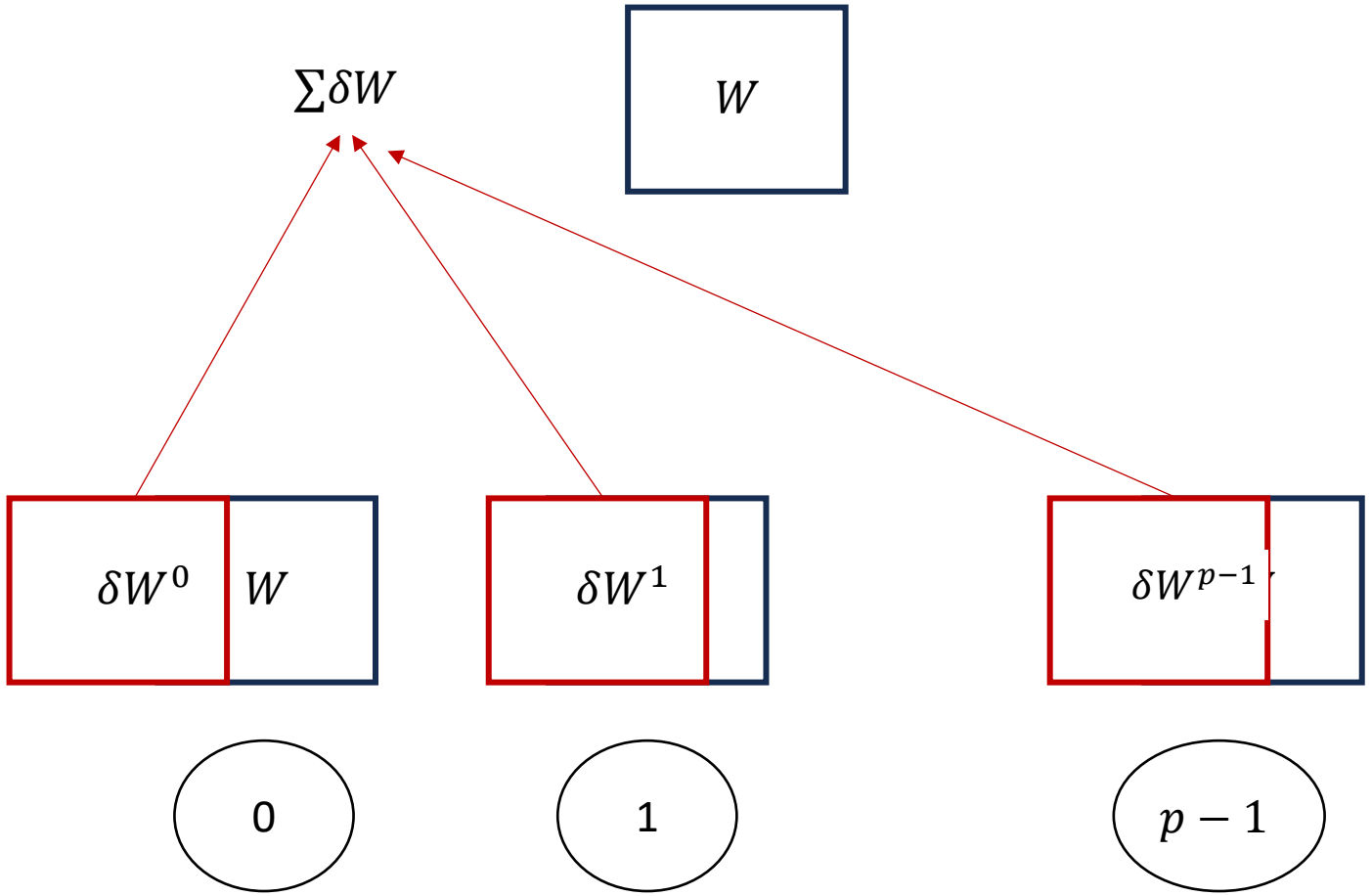
Data Parallel GPUs



Parameter Server Model

Parameter Server

Data Parallel GPUs

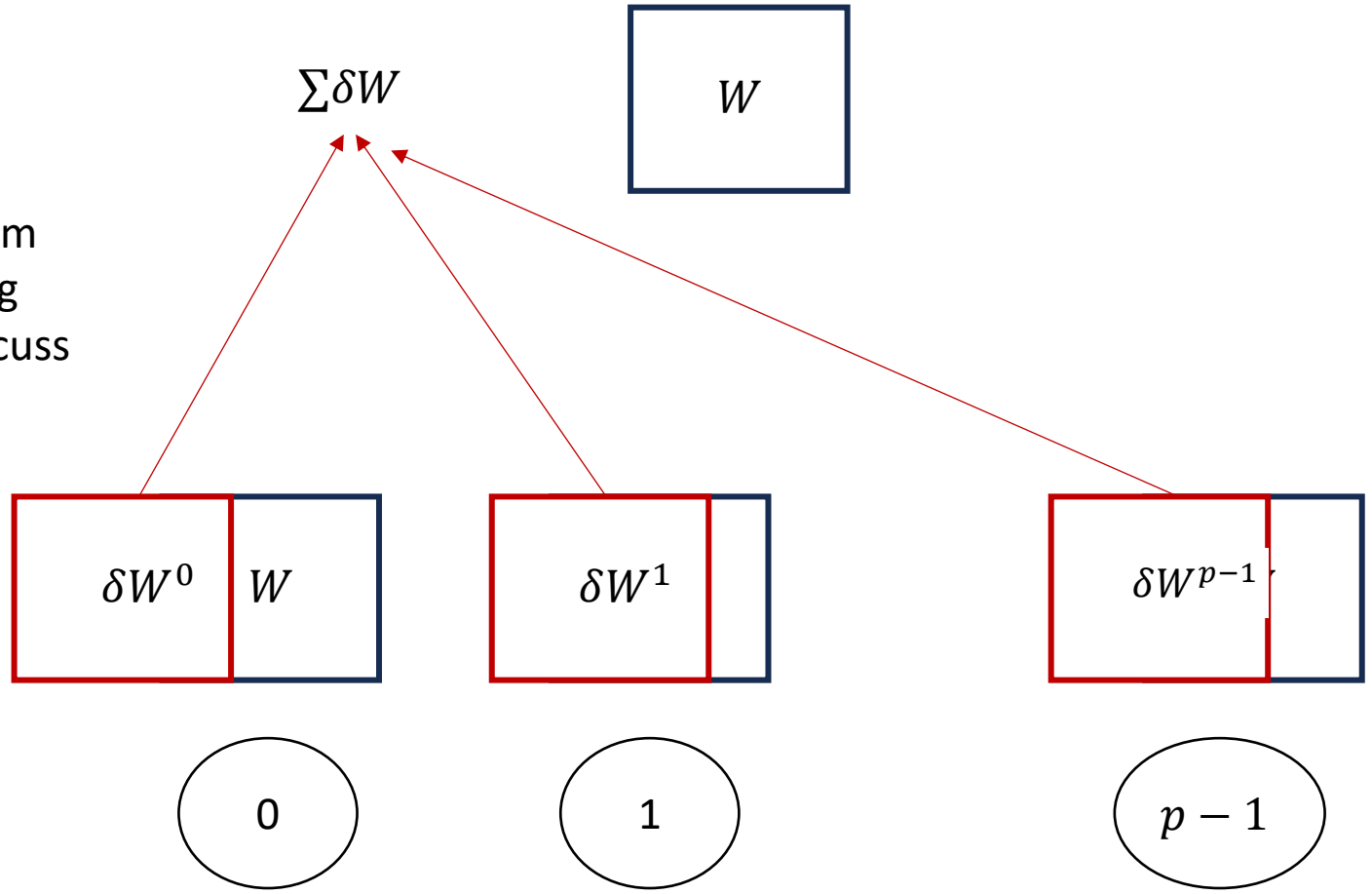


Parameter Server Model

Parameter Server

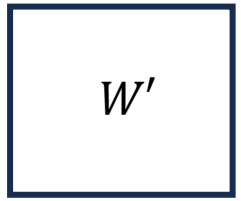
This operation of collecting data from different processors and aggregating them is called **Reduce** – We will discuss more when we discuss cluster of accelerators

Data Parallel GPUs

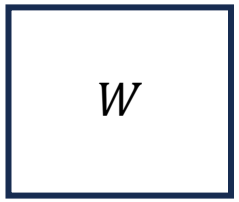


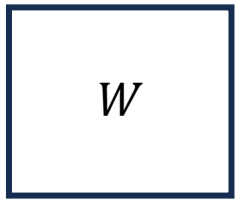
Parameter Server Model

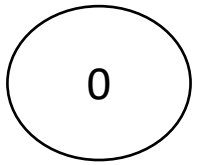
Parameter Server

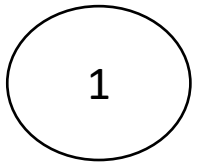


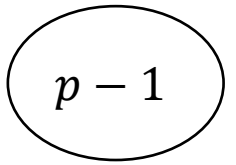
Data Parallel GPUs







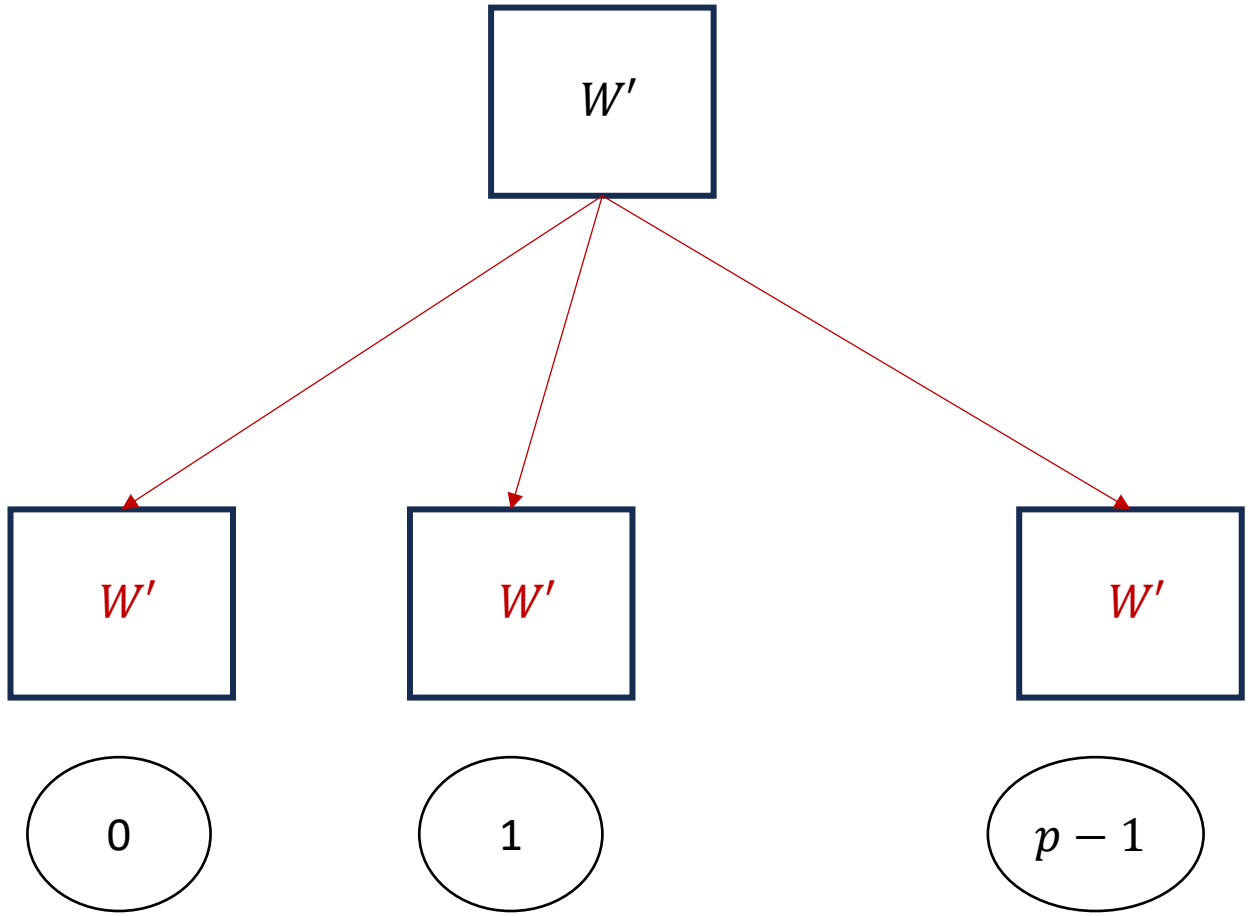




Parameter Server Model

Parameter Server

Data Parallel GPUs

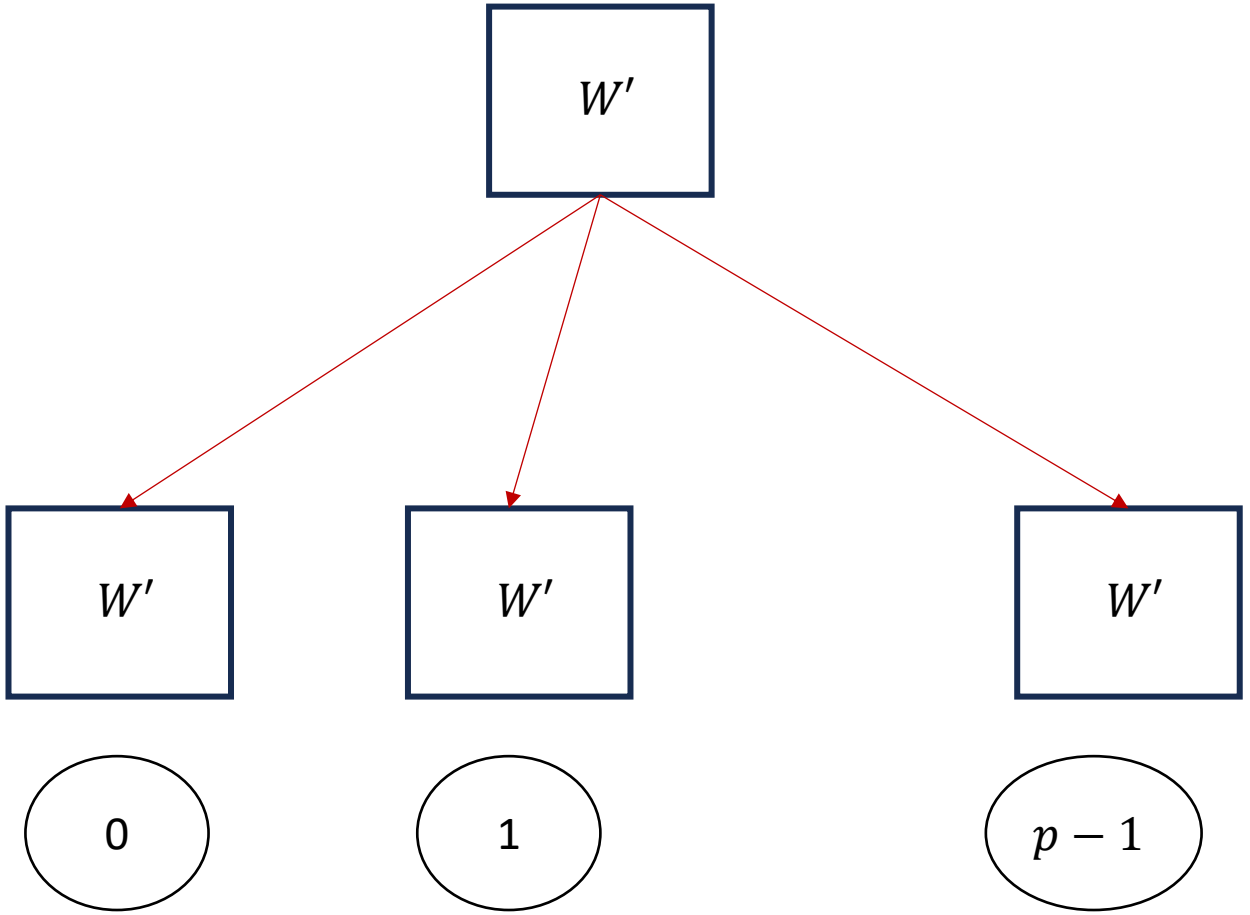


Parameter Server Model

Parameter Server

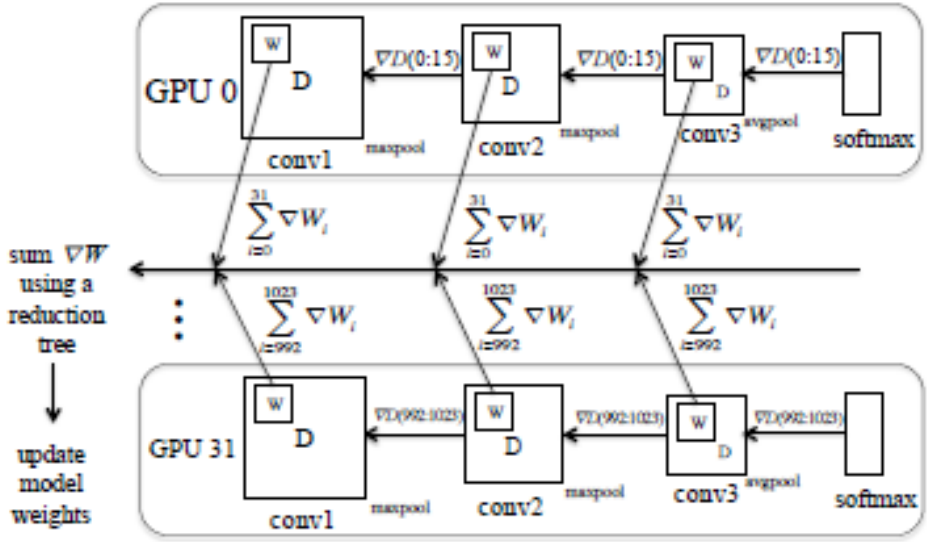
This operation of sending the same data from a single source to different processors is called **Broadcast** – We will discuss more when we discuss cluster of accelerators

Data Parallel GPUs



Data Parallel Training of Neural Networks

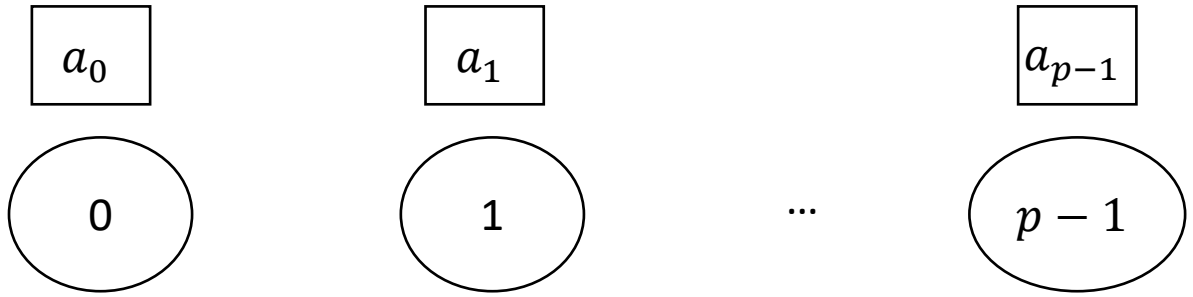
- How do we synchronize?
- Model #2: All-reduce based Synchronization
- No dedicated parameter server. Each worker calculates the gradients and broadcasts it to all other workers
- Each worker aggregates the gradients collected by all other workers and performs the update step



Iandola, F. N., Moskewicz, M. W., Ashraf, K., & Keutzer, K. (2016). Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2592-2600).

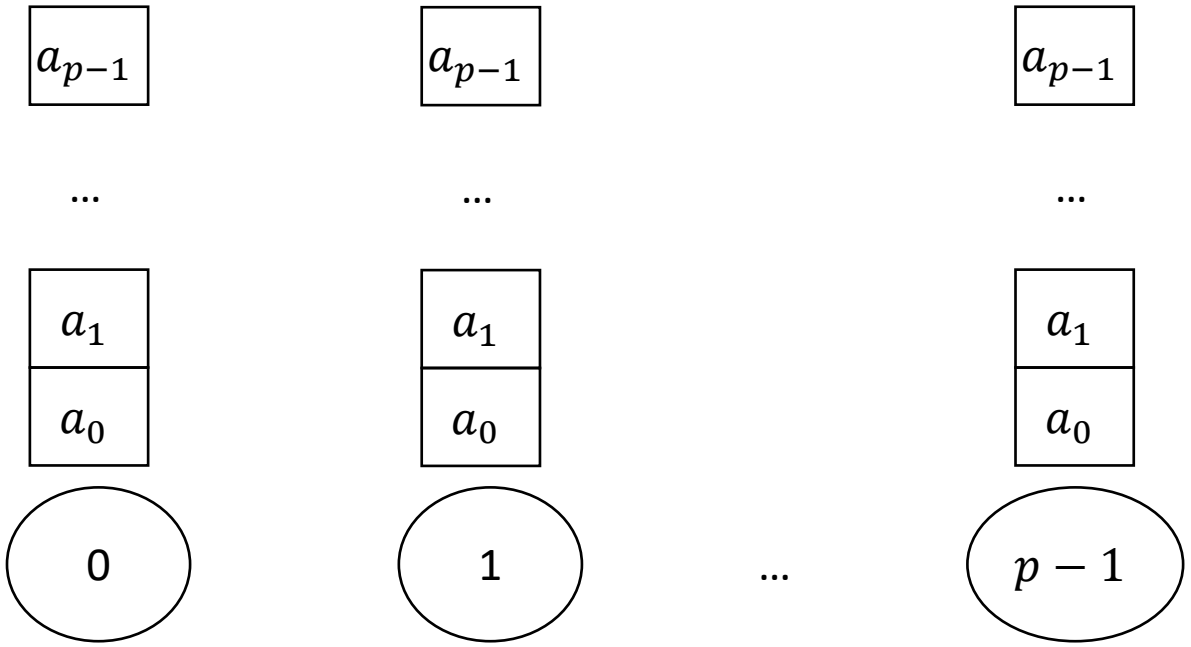
All Reduce Operation

- Each processor collects data from all the other processors and performs a reduce operation



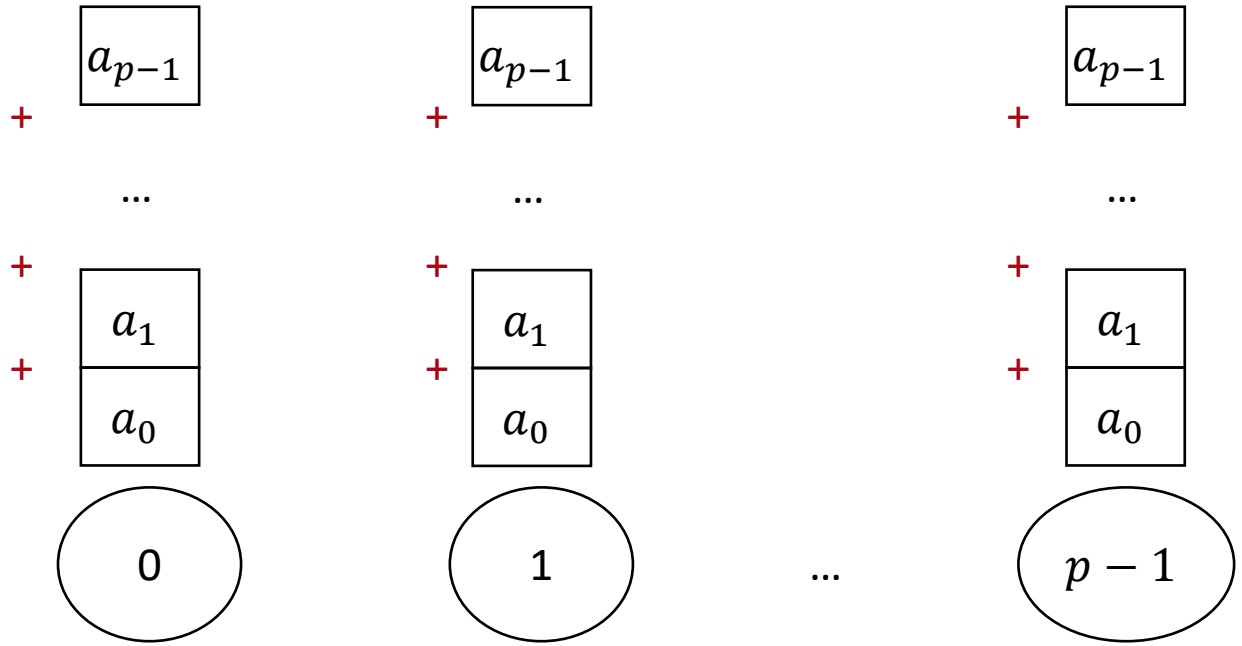
All Reduce Operation

- Each processor collects data from all the other processors and performs a reduce operation



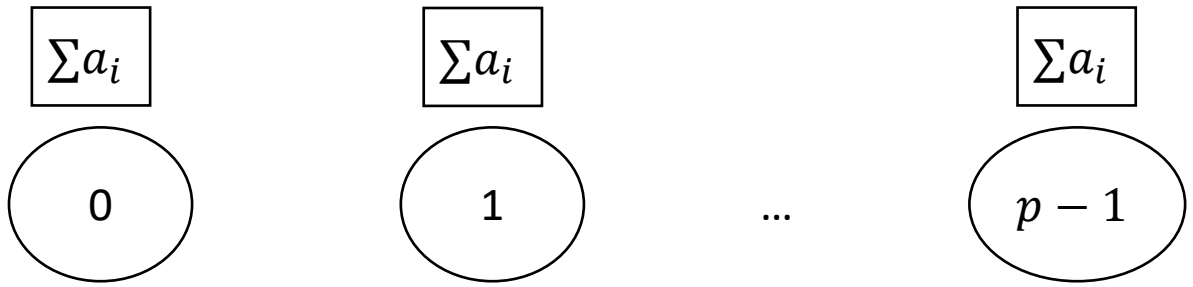
All Reduce Operation

- Each processor collects data from all the other processors and performs a reduce operation



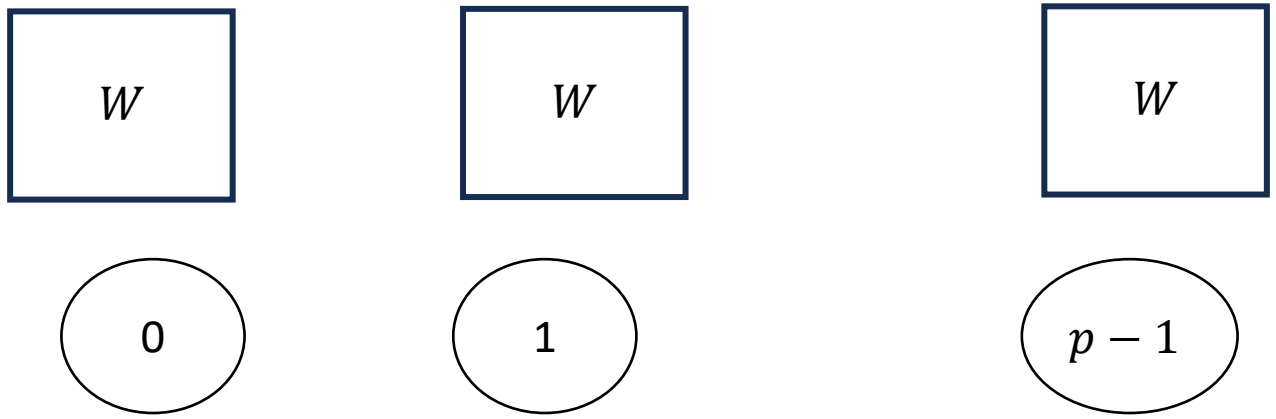
All Reduce Operation

- Each processor collects data from all the other processors and performs a reduce operation
- We will discuss how to efficiently implement this on a cluster of accelerators later in the class



All Reduce Model

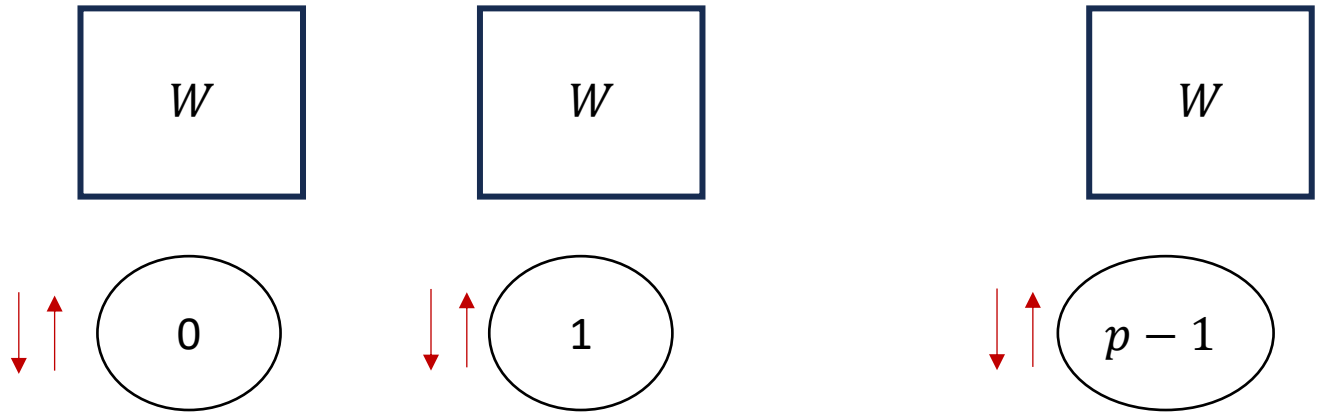
Data Parallel GPUs



All Reduce Model

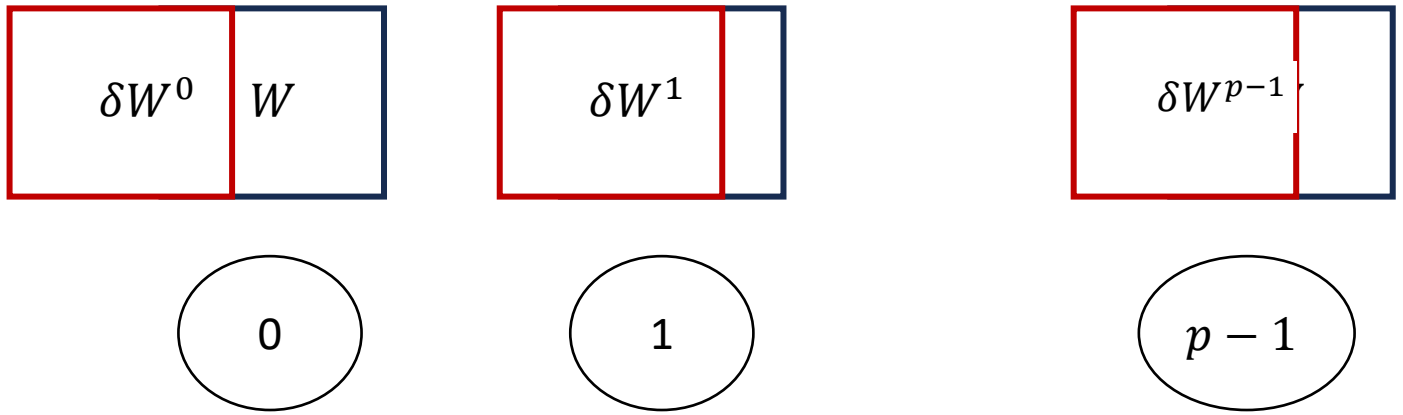
Data Parallel GPUs

Forward/Backward/
Gradient Calculation



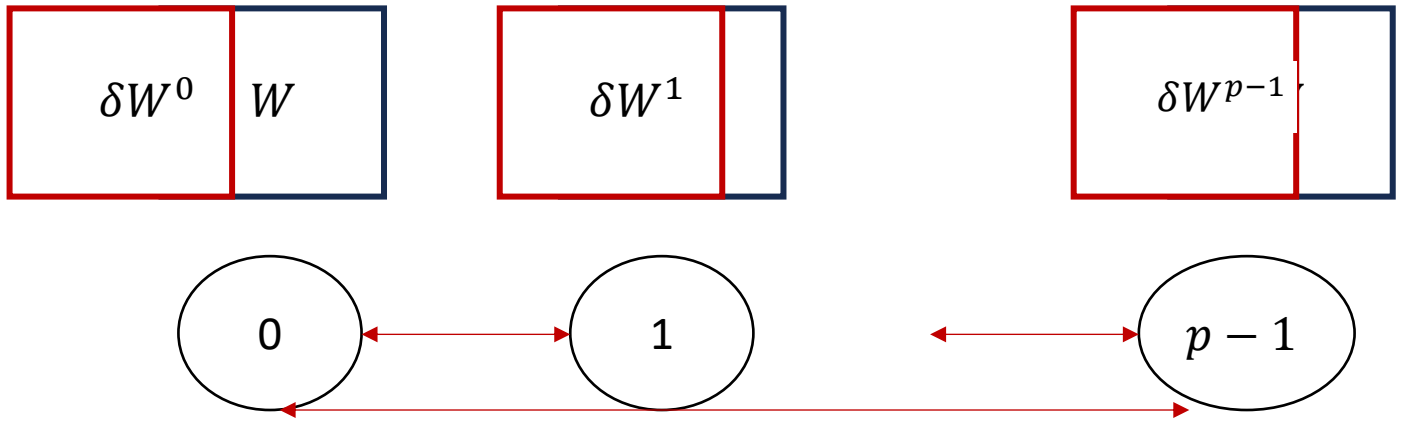
All Reduce Model

Data Parallel GPUs



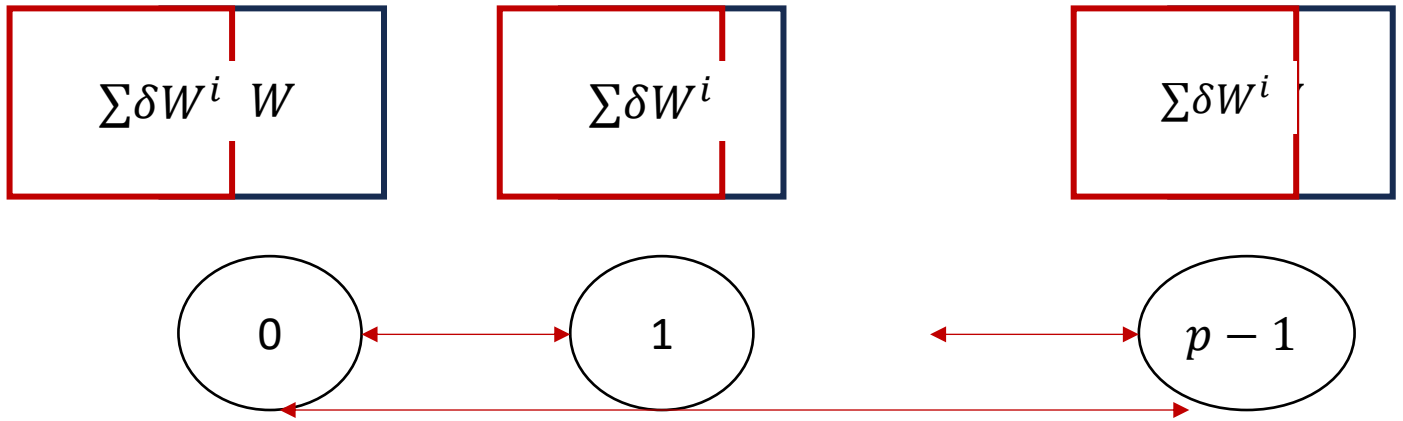
All Reduce Model

Data Parallel GPUs



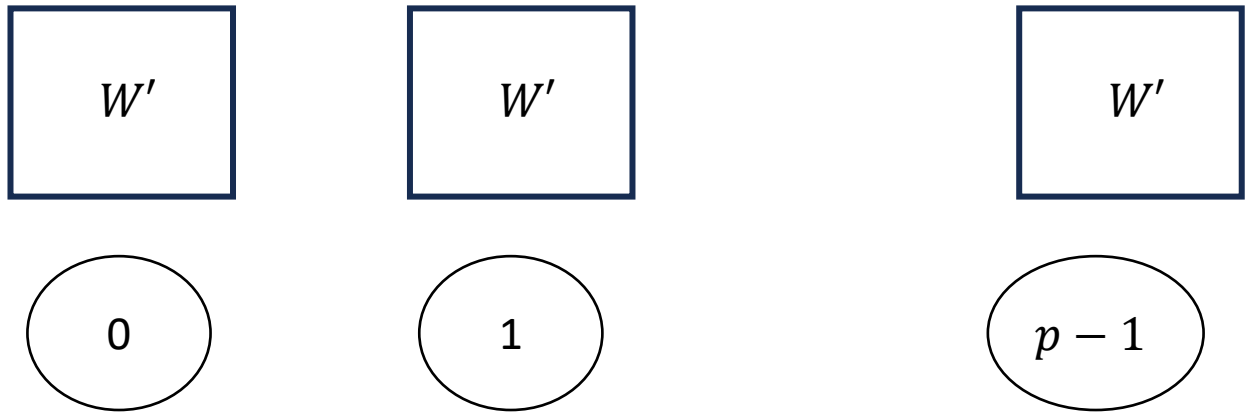
All Reduce Model

Data Parallel GPUs



All Reduce Model

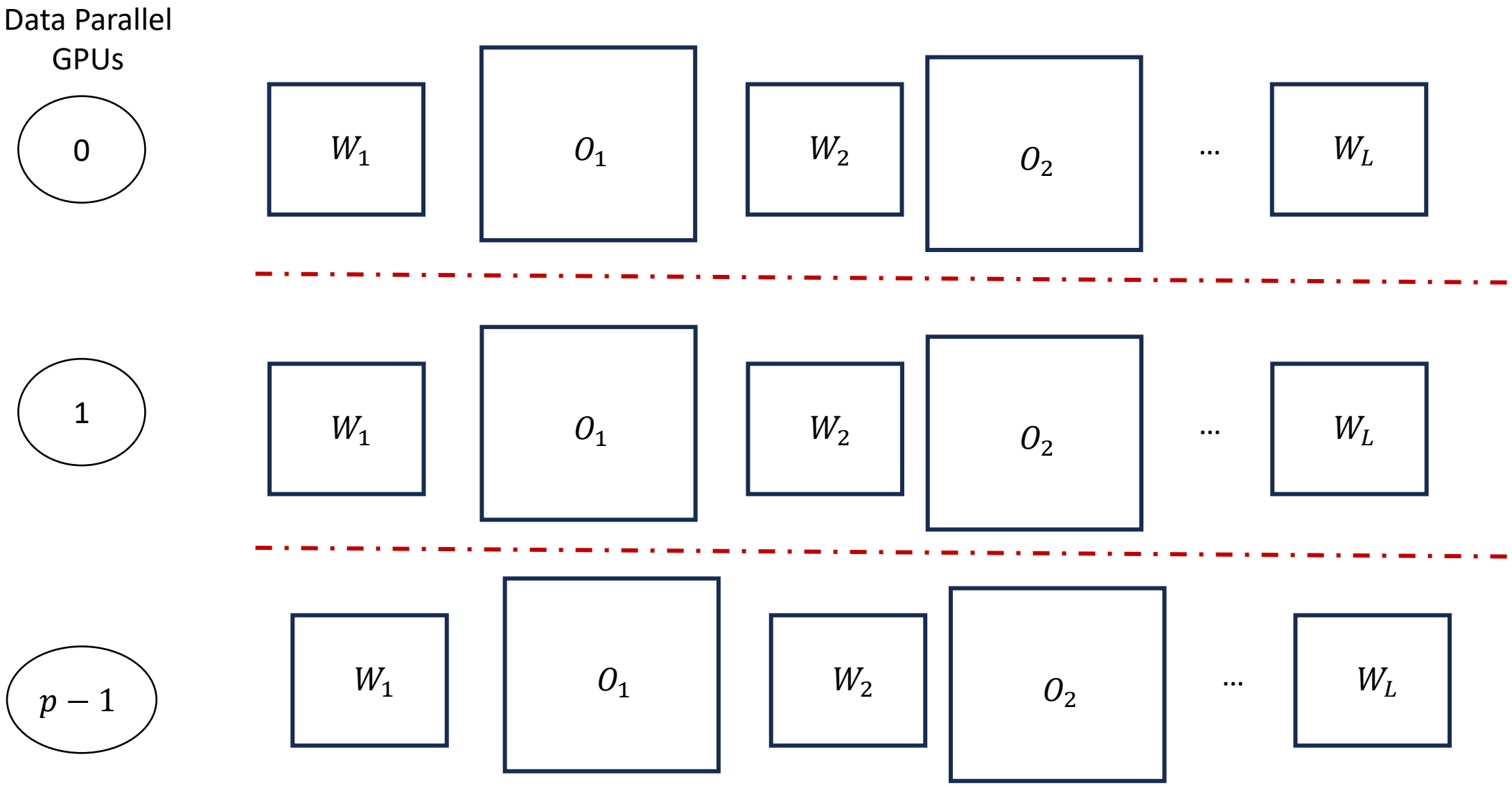
Data Parallel GPUs



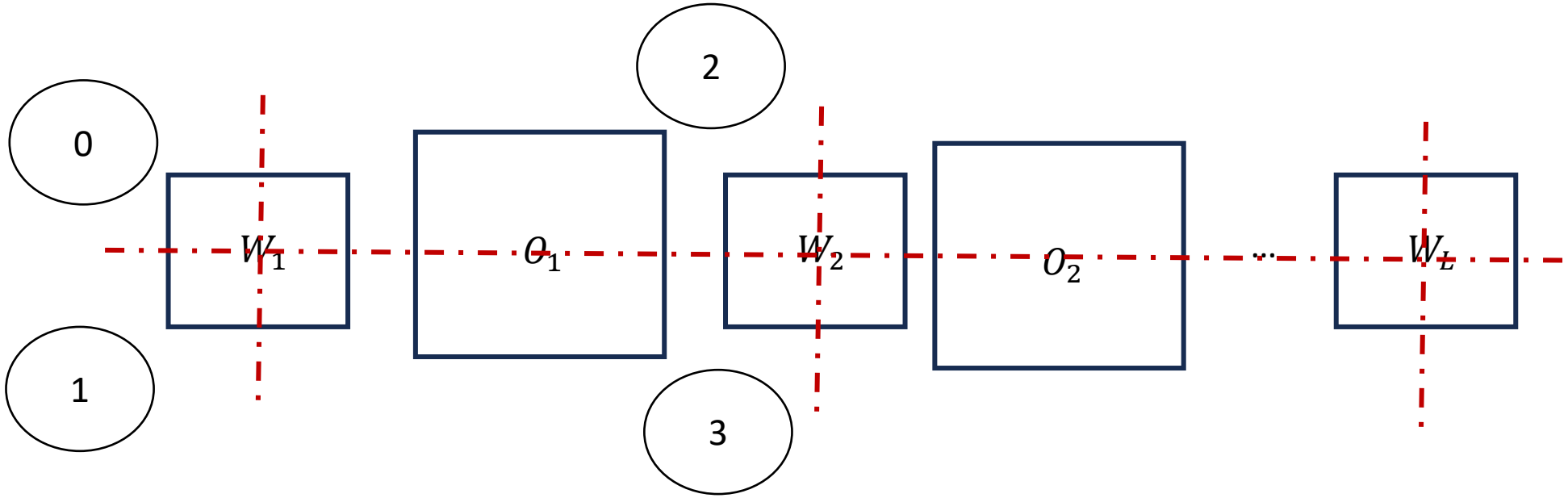
Model Parallel Distributed Training

- Data Parallelism
  - Weights are replicated on each GPU
- What if the model is too big for a single GPU?
- Model Parallelism: Split the model across GPUs

Model Parallel Distributed Training



Model Parallel Distributed Training



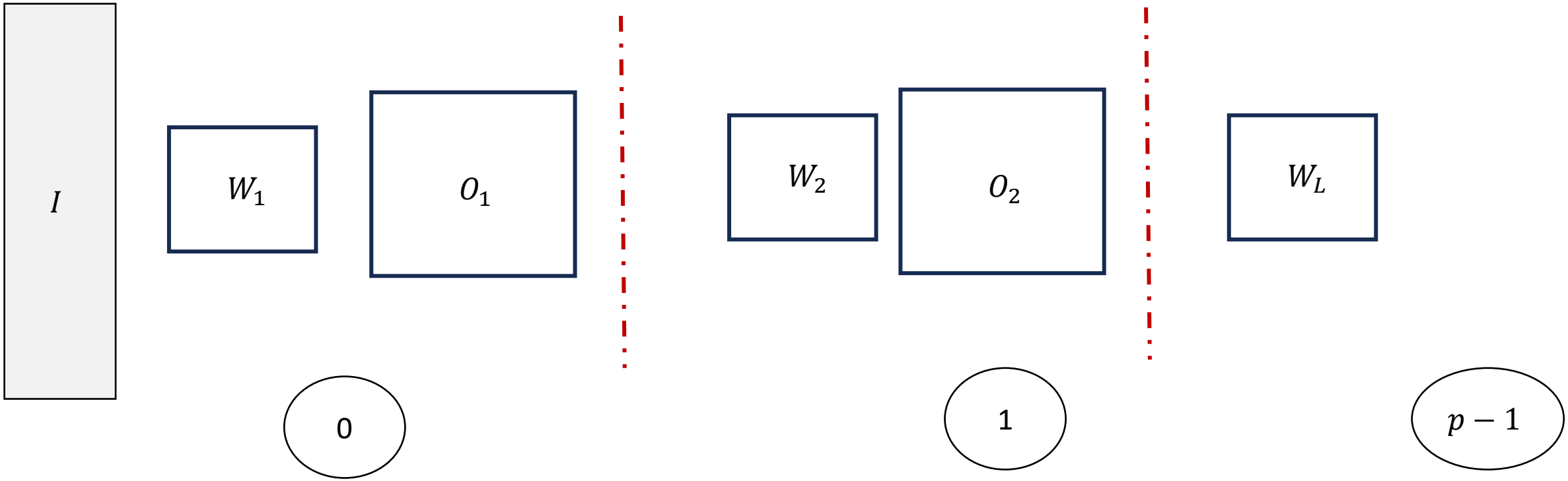
Partition the model across GPUs

Model Parallel Distributed Training

- Different ways to partition the model
- Layerwise Partitioning – Pipeline Parallelism
- Intra-Layer Partitioning – Tensor Parallelism

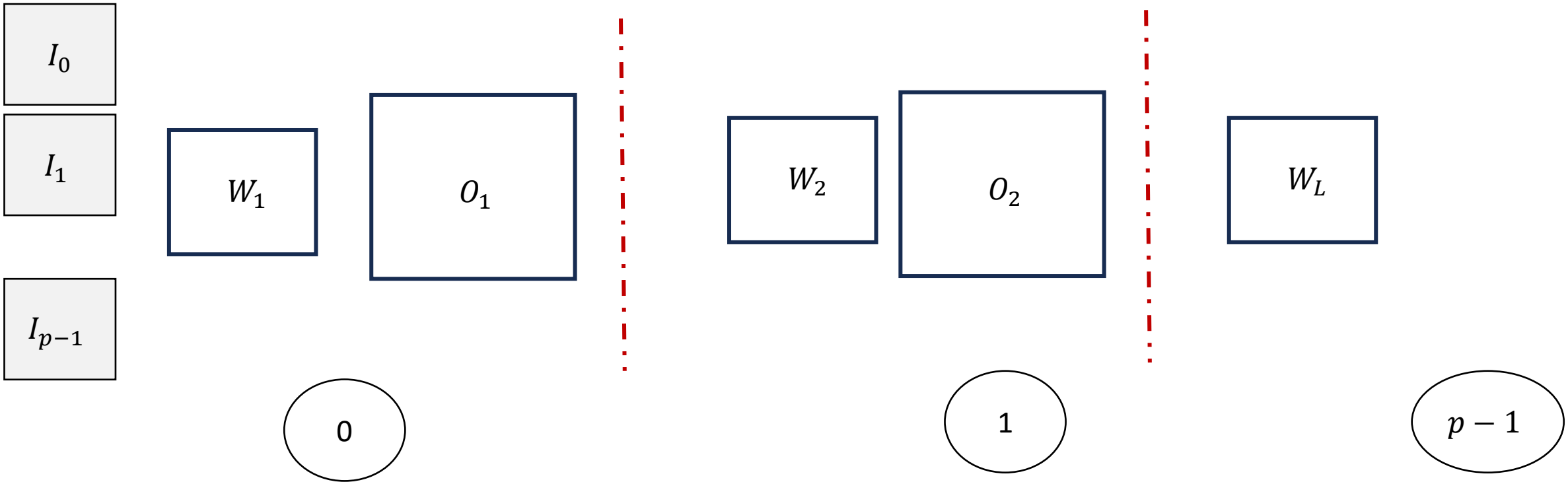
Pipeline Parallelism

Batch of inputs:
e.g., 64 images

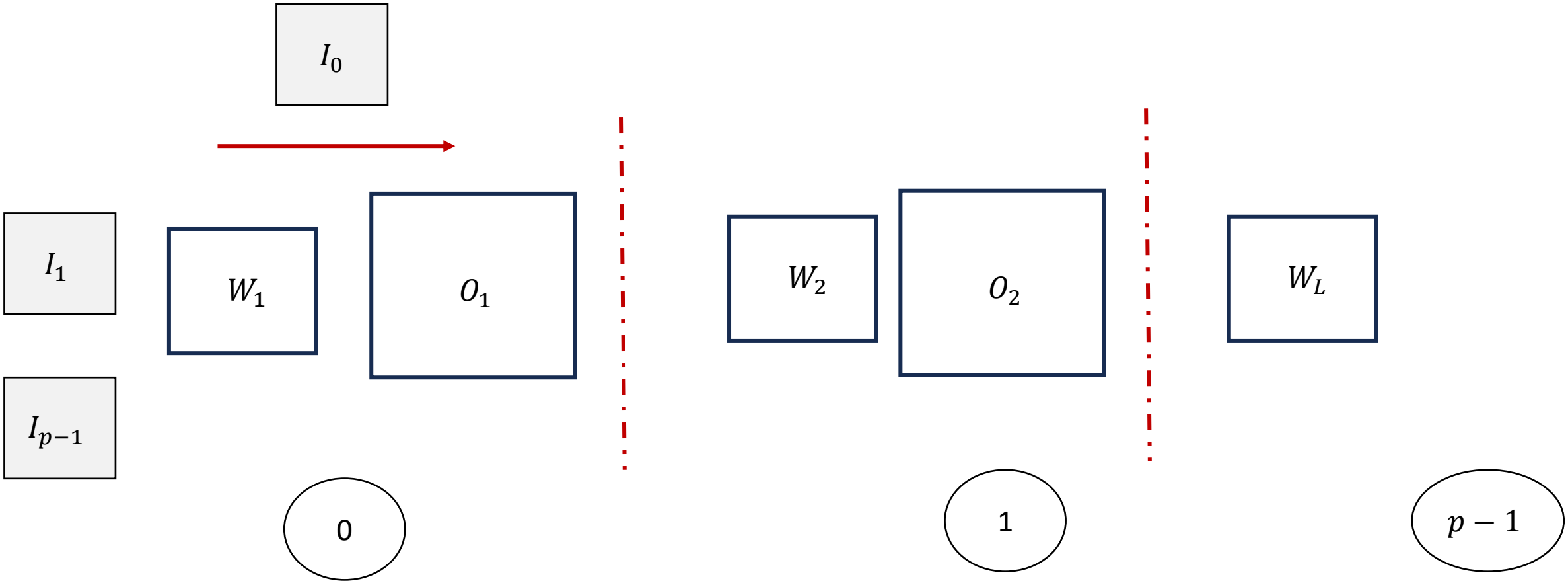


Pipeline Parallelism

Split input batch into p smaller mini-batches

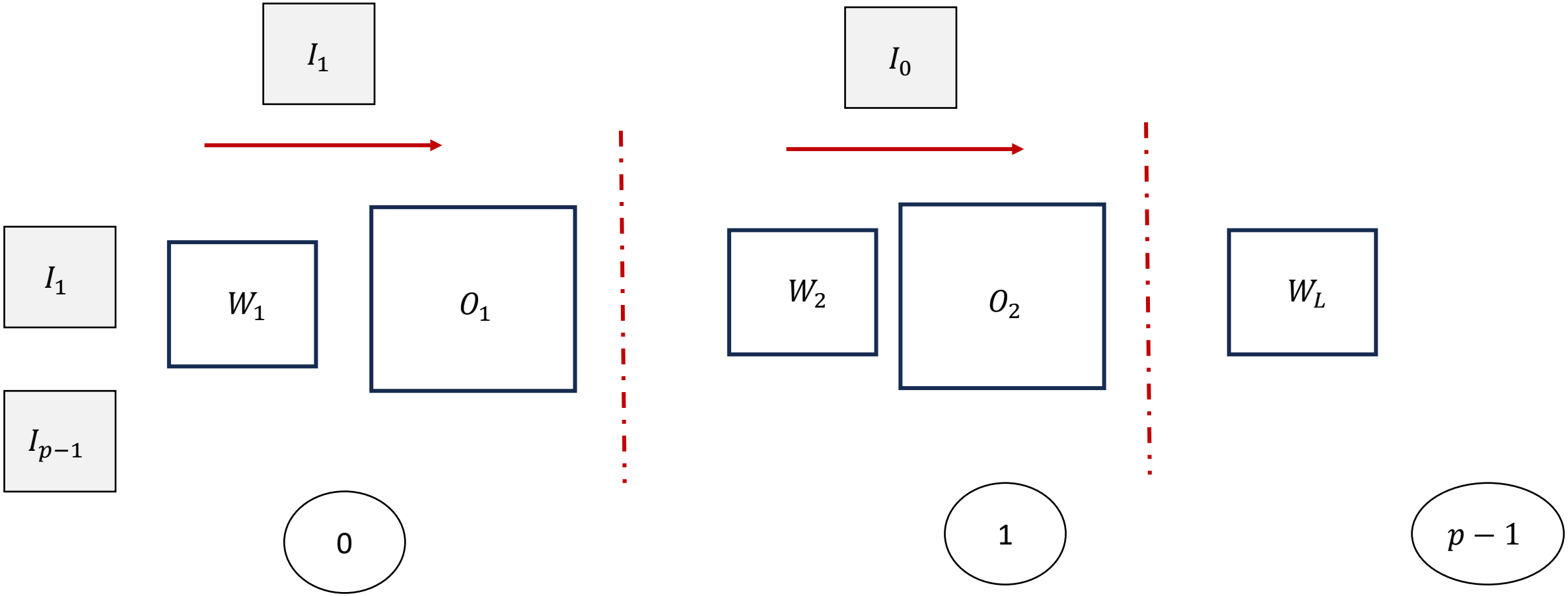


Pipeline Parallelism



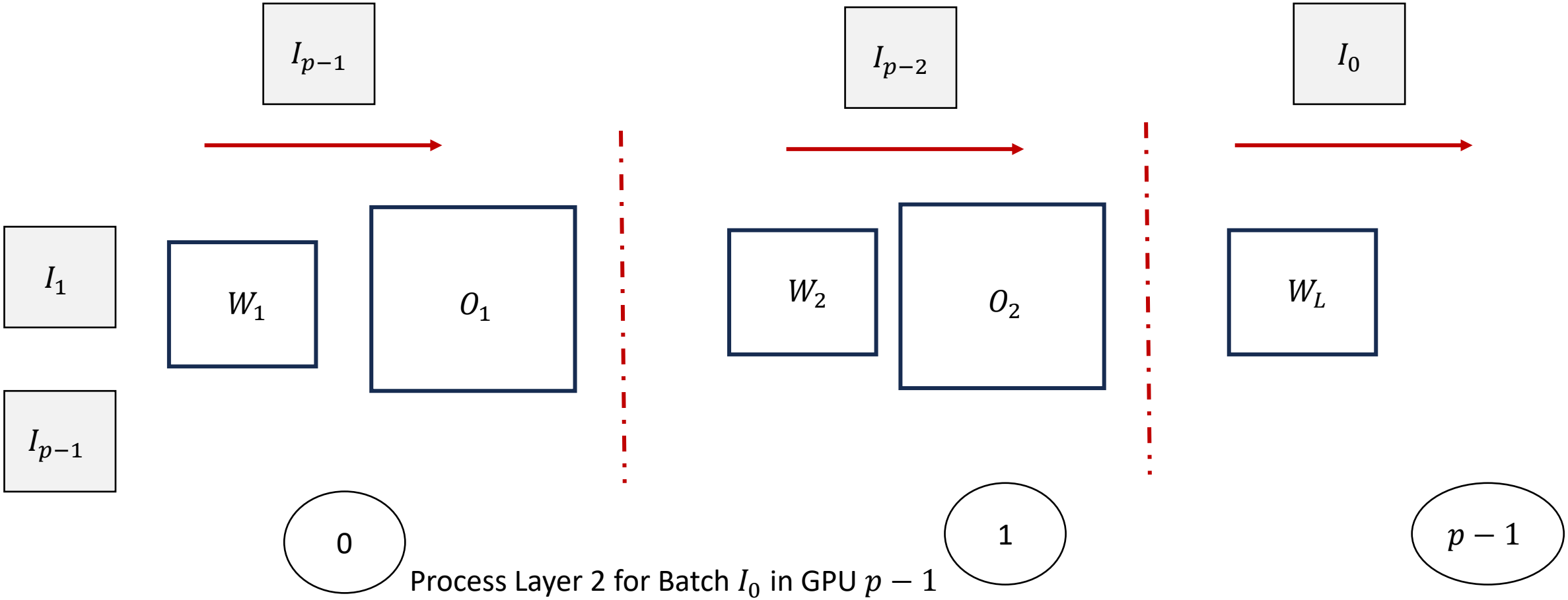
Process Layer 1 for Batch I_0 in GPU 0

Pipeline Parallelism



Process Layer 2 for Batch I_0 in GPU 1
Process Layer 1 for Batch I_1 in GPU 0

Pipeline Parallelism



Process Layer 2 for Batch I_0 in GPU $p-1$

…

Process Layer 2 for Batch I_{p-2} in GPU 1

Process Layer 1 for Batch I_{p-1} in GPU 0

Pipeline Parallelism

- When and what communications are needed?

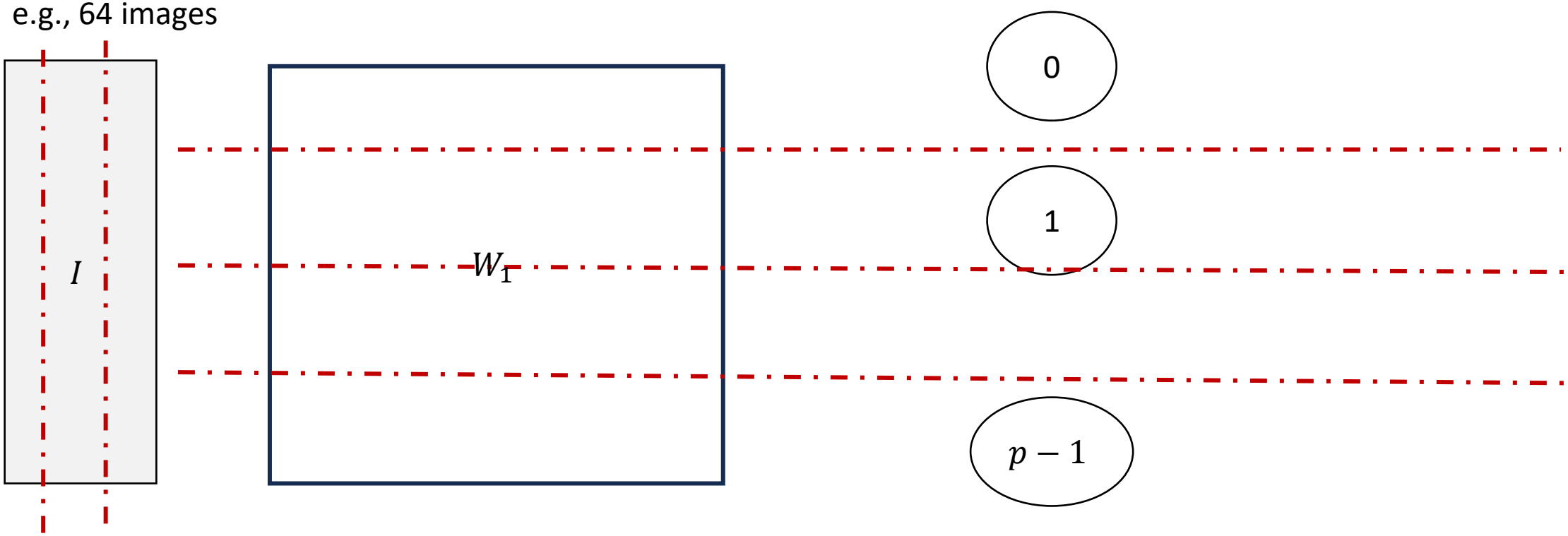
Pipeline Parallelism

- When and what communications are needed?

- Need to communicate the activations after each layer
  - In backward propagation, need to communicate gradients.

Tensor Parallelism

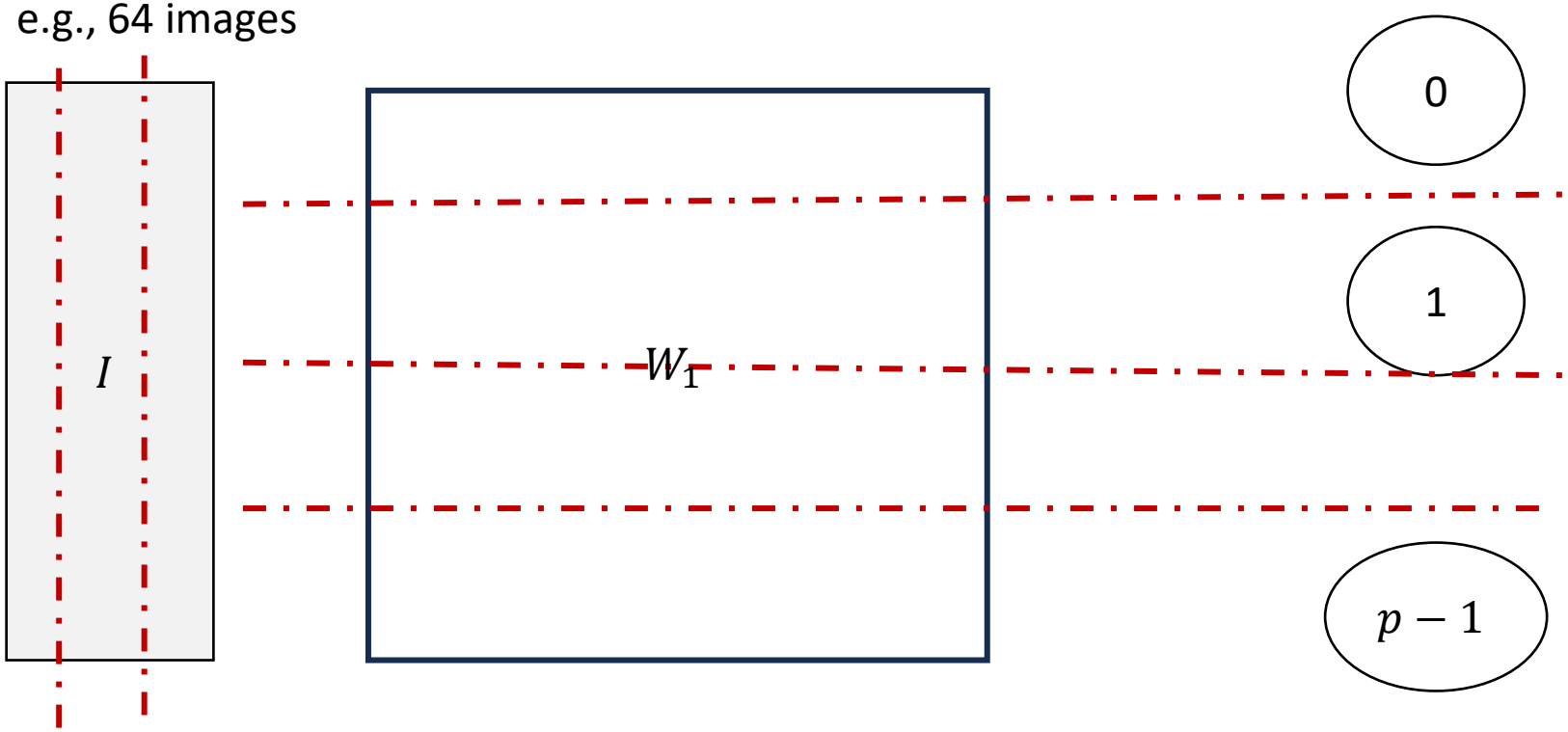
Batch of inputs:
e.g., 64 images



Partition across the embedding dimension of the inputs

Tensor Parallelism

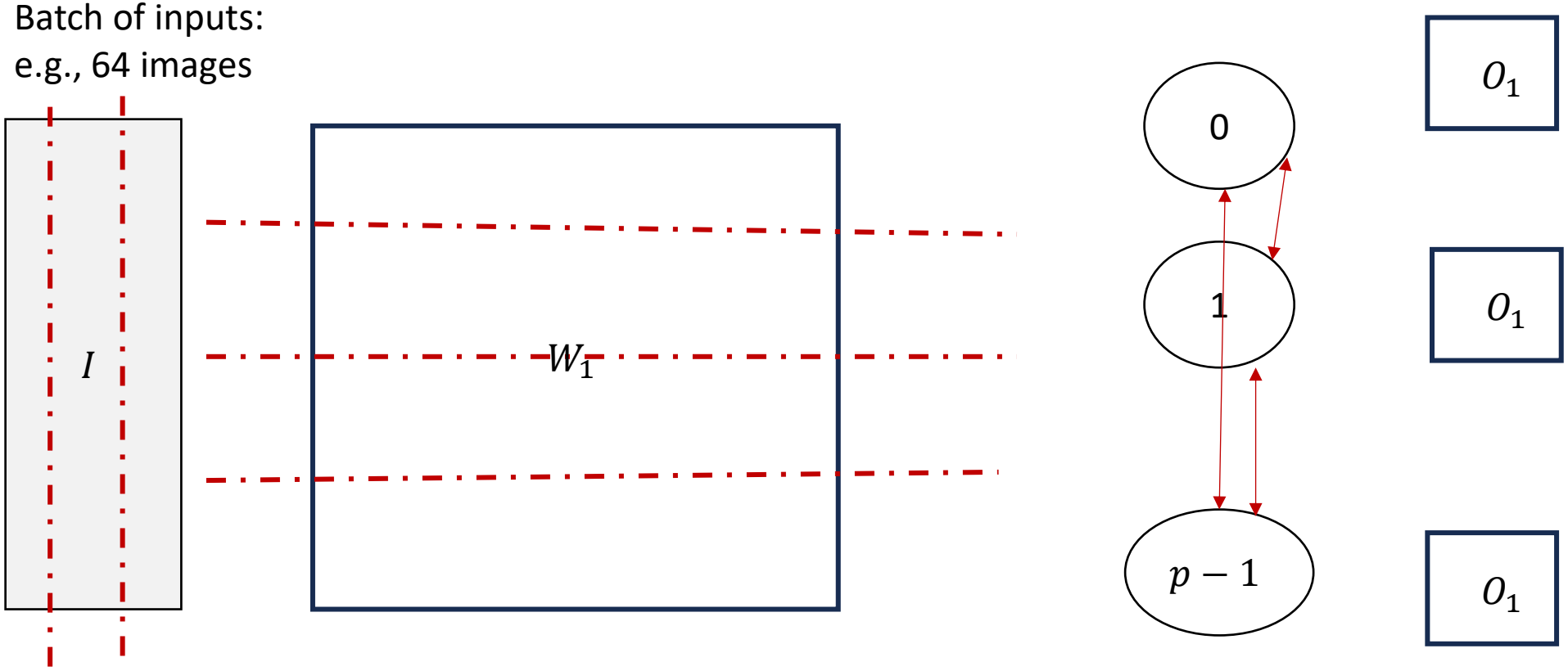
Batch of inputs:
e.g., 64 images



In each layer, each GPU applies a portion of the weights to its portion of the input

Tensor Parallelism

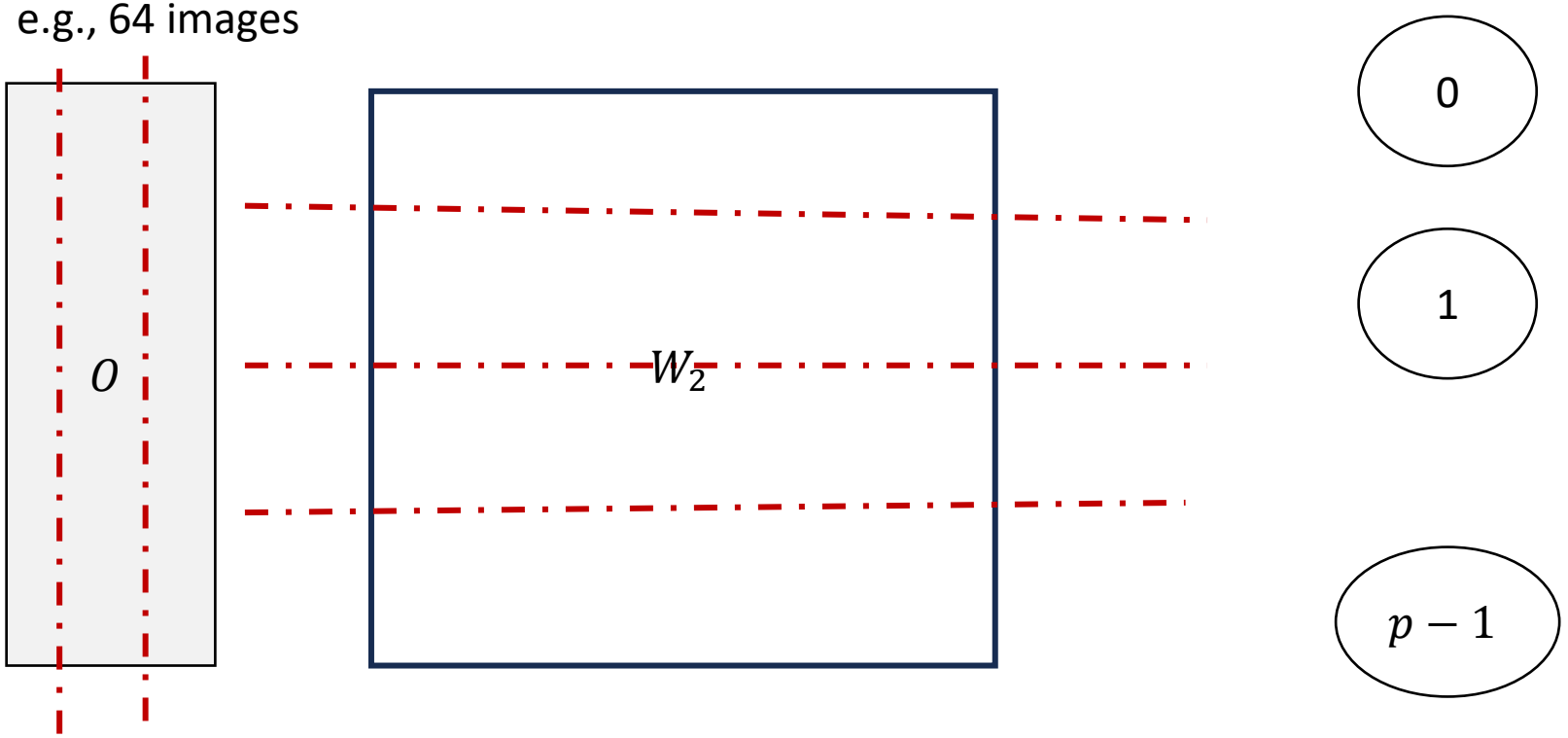
Batch of inputs:
e.g., 64 images



GPU communicate to collective produce output of the layer – Each GPU will get a copy of the output

Tensor Parallelism

Batch of inputs:
e.g., 64 images



Output is again partitioned across the embedding dimension to serve as input to the next layer

3D Parallelism

- In practice, all three parallelisms are combined
- Data + Pipeline + Tensor = 3D parallelism
- DeepSpeed: https://github.com/microsoft/DeepSpeed/tree/master?tab=readme-ov-file

3D Parallelism

- We will discuss the communication and computation optimizations for distributed training in detail when we are looking into cluster of accelerators

Outline

- Distributed Training Basics
- Pytorch Lightning

PyTorch Lightning

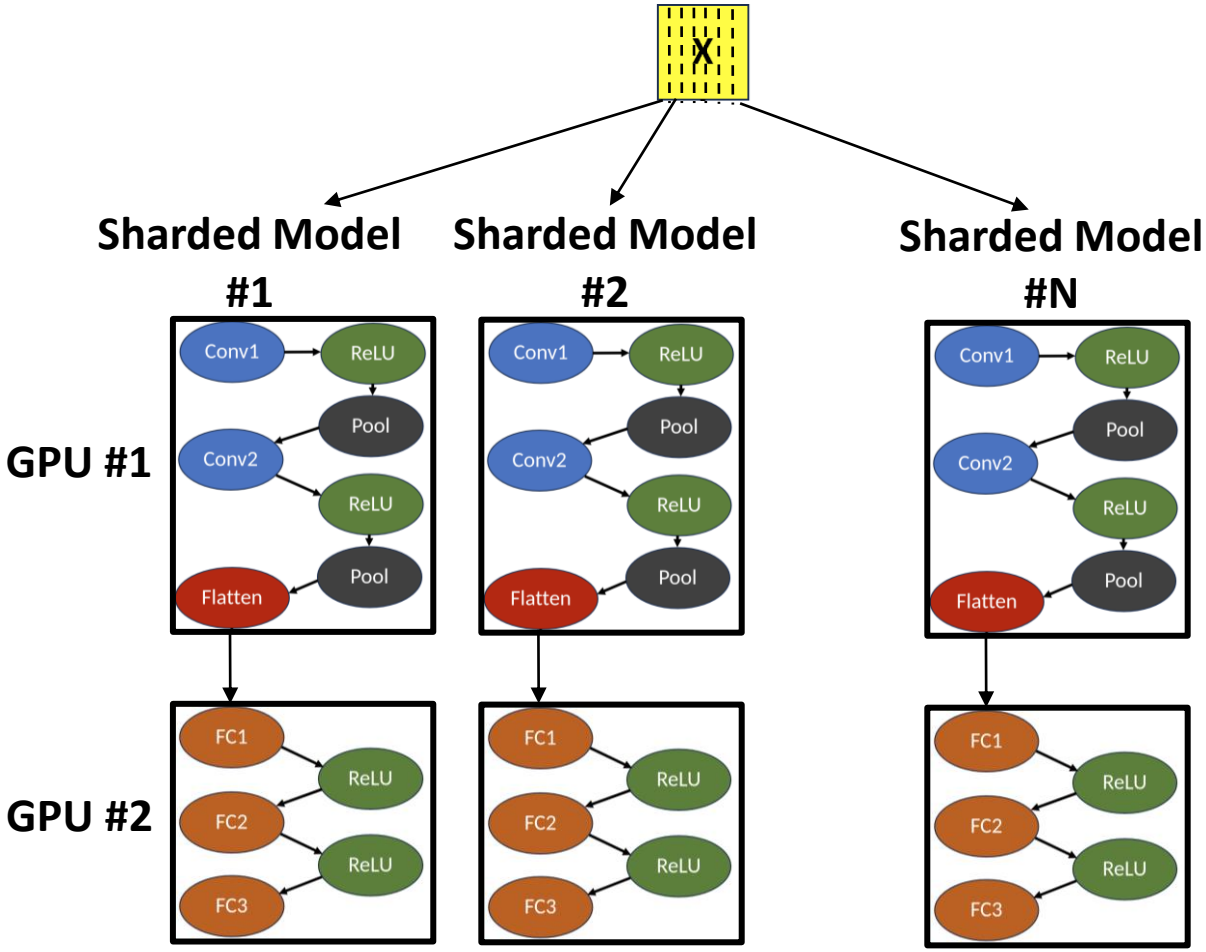
Facilitating Deep Learning

- An open source Python library created in 2019.
  - Offers high-level wrappers for PyTorch.
  - Aims to **decouple research from engineering**.

Pytorch Lightning – Distributed Training

- Supports Distributed Data Parallel and variants https://lightning.ai/docs/pytorch/stable/accelerators/gpu_intermediate.html

- Supports Data+Model Parallel – Fully Sharded Data Parallel
  - Model weights are partitioned
  - https://lightning.ai/docs/pytorch/stable/advanced/model_parallel/fsdp.html



Pytorch Lightning – Distributed Training

- Most of the programming remains similar to PyTorch

- Some additions/modifications include:
  - Define a sub-class of LightningModule to define interactions between the various layers of your models – PyTorch Lightning uses this information to perform FSDP
  - Define training_step within this module – enables PyTorch Lightning to optimizing training step
  - Configure optimizers within this module - enables PyTorch Lightning to apply optimizer specific optimizations

Pytorch Lightning – Distributed Training

```python
# define any number of nn.Modules (or use your current ones)
encoder = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))
decoder = nn.Sequential(nn.Linear(3, 64), nn.ReLU(), nn.Linear(64, 28 * 28))


# define the LightningModule
class LitAutoEncoder(pl.LightningModule):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        # it is independent of forward
        x, y = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = nn.functional.mse_loss(x_hat, x)
        # Logging to TensorBoard (if installed) by default
        self.log("train_loss", loss)
        return loss

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=1e-3)
        return optimizer


# init the autoencoder
autoencoder = LitAutoEncoder(encoder, decoder)
```

Instead of Training Loop, we have the following

```python
# model
autoencoder = LitAutoEncoder(Encoder(), Decoder())

# train model
trainer = L.Trainer()
trainer.fit(model=autoencoder, train_dataloaders=train_loader)
```

Pytorch Lightning – Acceleration and Parallelization

- You don't need to explicitly specify the device

```
# before lightning
def forward(self, x):
    x = x.cuda(0)
    layer_1.cuda(0)
    x_hat = layer_1(x)

# after lightning
def forward(self, x):
    x_hat = layer_1(x)
```

- You can achieve parallel training by simply setting the number of gpus and policy
- ddp – distributed data parallel
- Fsdp – Fully Sharded Data Parallel
  - Requires additional information on which layers to shard

```
# train on 8 GPUs (same machine (ie: node))
trainer = Trainer(gpus=8, accelerator='ddp')

# train on 32 GPUs (4 nodes)
trainer = Trainer(gpus=8, accelerator='ddp', num_nodes=4)
```

Pytorch Lightning – For you to Review

- Pytorch Lightning Training Basics - https://lightning.ai/docs/pytorch/stable/model/train_model_basic.html
- Training on GPU - https://lightning.ai/docs/pytorch/stable/accelerators/gpu_basic.html
- Distributed Training - https://lightning.ai/docs/pytorch/stable/levels/advanced_level_21.html

Next Class

- 10/16 Lecture 15
  - Acceleration of Transformer Models: Basics

CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

Thank You

- Questions?

- Email: sanmukh.kuppannagari@case.edu