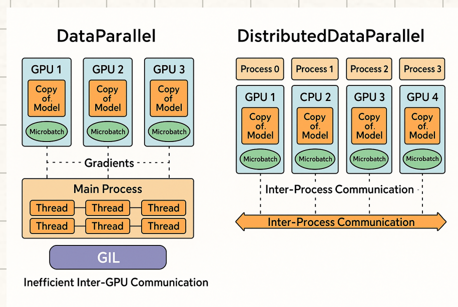
Training Optimization
— Data Parallelism

- Data Parallel: Single process, multi-threaded approach that works when the model fits on a single GPU. Each GPU keeps a copy of the model, processes different microbatches, then averages gradients across GPUs.

  Main Bottleneck: Rely on single-process, multi-thread communication → inefficient inter-GPU communication and potential slowdown due to GPU overhead

  The biggest problem of "Single process, multi-thread": GIL contention



- Synchronization Approaches: At the end of each minibatch, workers need to sync gradients / weights to avoid staleness.
  Two Synch approaches:
  1. Bulk Synch Parallel (BSP)
     Def: workers synch at the end of every minibatch.
     ✓: prevents model weights staleness, provides good learning efficiency
     ✗: Each machine need to halt and wait for others to send gradients
  2. Asynch Parallel (ASP)
     ✓: Each worker processes the data asych with no waiting or stalling.
     ✗: Easily stale weights being used thus lower the statistical learning efficiency ⇒ ↑ computation time but not speed up training time to converge.
     Fake hard work!

How to solve?

- DDP (Distributed Data Parallel): Each GPU worker has its own process and can work on multiple nodes/machines. Use Ring All-Reduce algo. to avoid central bottleneck.
  DDP has lower computation overhead than DataParallel

- ZeRO (Zero Redundancy Optimizer): Not only model parameters and gradients, but also the optimizer state (Adam momentum, Variance) also takes a lot of memory
  Split storage, not split computation
  parallel storage, not parallel computation.
  Its has 3 main optimization Stages:

Stage	Partitioned Component	Memory Reduction	Communication Cost	Notes
ZeRO-1	Optimizer States	4x	= DP	Recommend!
ZeRO-2	Gradients	8x	= DP	combined with ZeRO-1
ZeRO-3	Model Parameters	Linear (64 GPUs → 64x)	+50%	max memory but increase complexity

Summary: ZeRO partitions optimizer states, gradients, model parameters across different GPUs achieve near-linear memory savings while keeping the computation logic unchanged.

· Communication Primitives: Are the building blocks for distributed training, and used to synch parameters, gradients, optimizer states efficiently across multiple GPUs or nodes.

All-Reduce: Synch gradients

Every rank starts with its own tensor, ends up with the reduction (sum / mean / max / ...) across all ranks.

Implemention:

All-Reduce = Reduce-scatter (compute the global sum, shared) + All-Gather (reconstruct the full summed tensor on every rank)

eg. Rank 0 : g_0 — (all-reduce op SUM) → Sum($g_0 \cdots g_{N-1}$) at every rank

Rank 1 : g_1

⋮

Rank N-1 : g_{N-1}

Ring All-Reduce: Topology: ranks form a ring. each step every rank sends to its right neighbor and receives from its left.

min bandwidth bottleneck → fully use homogeneous lips (NVLinks), large X, but slow if too many steps

Two Stages:

1° Reduce-Scatter: reduce partial chunks while circulating

2° All-Gather: circulate reduced chunks so everyone gets the complete results.

Communication Overhead: with tensor size X bytes and N ranks:

Each rank sends / receives (N-1) messages per phase

Each message is $\frac{X}{N}$ bytes

→ per-rank traffic: $2 \times (N-1) \times \frac{X}{N}$ bytes.

Reduce-Scatter: reduced across ranks, only keep your $\frac{1}{N}$ shared.

used as the 1st step in optimized all-reduce.

eg. Each rank has: [chunk 0 | chunk 1 | ... | chunk N-1]

After reduce-scatter (SUM):

Rank 0 keeps reduced chunk 0

Rank 1 keeps reduced chunk 1

...

All-Gather: Each rank starts with its own $\frac{1}{N}$ shard; everyone ends with the full concatenated tensor.

used as the 2nd step in optimized all reduced

eg. Before:

Rank 0 : chunk 0

Rank 1 : chunk 1

After:

All ranks: [chunk 0 | chunk 1 | ... | chunk N-1]

Broadcast: One process sends data to all others

eg. distributing model weights at init

Reduce: Data from all process is reduced (eg. SUM) and the results is sent to single process

eg. All ranks → [SUM] → rank 0 has the result.

P.S not for synch gradients, if need results on all ranks → use all-reduced

Scatter: One process splits data and sends different chunks to each process eg. Rank 0 : [Chunk 0 | chunk 1 | . . . | chunk N-1]

↳ Rank 1 gets chunk 0

↳ Rank 2 gets chunk 1

It's suitable for distributing non-overlapping mini-batch or work assignments pre-shared by the rank 0 (root)

Gather : Each process sends data to a single process, which collets all the data

eg. All ranks → rank 0 collects [chunk 0, chunk 1, . . . chunk N-1]

Reference :

- Megatron-ML | ZeRO | Deepspeed | Mixed Precision
- Stanford CS224N : Lecture 12 - Efficient Training
- Stanford CS229S : Lecture 08 - Parallelism Lecture.
- Distributed Training with PyTorch : complete tutorial with cloud infra / code

DISTRIBUTED TRAINING WITH PYTORCH

Umar Jamil

Outline

- Introduction to distributed training
  - Why we need it
  - Data Parallel vs Model Parallel
- Review of neural networks
  - Loss function and gradient
  - Gradient accumulation
- Distributed Data Parallel training
  - How it works
  - Communication primitives
    - Broadcast operator
    - Reduce operator
    - All-Reduce operator
  - Managing failover
- Coding session
  - Infrastructure (Paperspace)
  - PyTorch code
- How PyTorch handles Distributed Data Parallel training
  - Bucketing
  - Computation-Communication overlap during backpropagation

What is distributed training?

Imagine you want to train a Language Model on a very big dataset, for example the entire content of Wikipedia. The dataset is quite big, because it is made up of millions of articles, each of them with thousands of tokens. To train this model on a single GPU may be possible, but it poses some challenges:

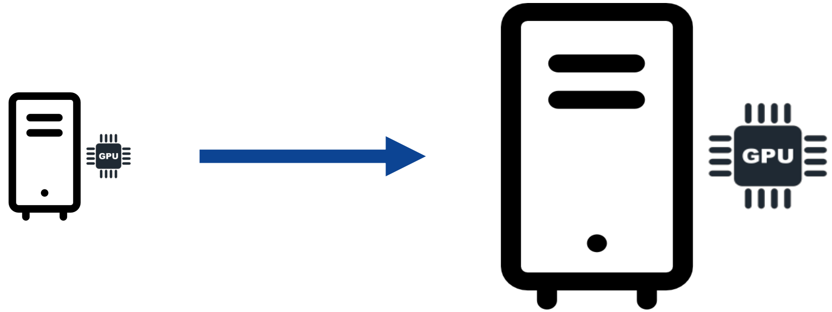
1. The model may not fit on a single GPU: this happens when the model has many parameters.
2. You are forced to use a small batch size because a bigger batch size leads to an *Out Of Memory* error on CUDA.
3. The model may take years to train because the dataset is huge.

If any of the above applies to you, then you need to scale your training setup. Scaling can be done vertically, or horizontally. Let's compare these two options.

Vertical scaling

money is all you need!

No code change



1x Server
8GB RAM
4GB GPU Memory

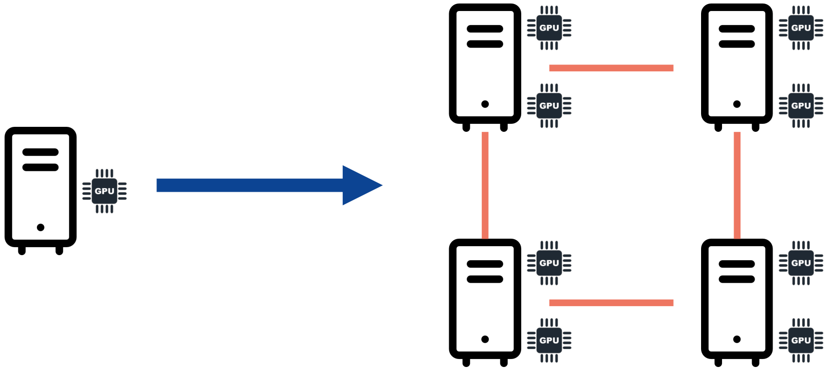
1x Server
64GB RAM
32GB GPU Memory

Horizontal scaling

Strategy is all you need

Minimal code change (thanks to PyTorch)

In this video we will explore horizontal scaling

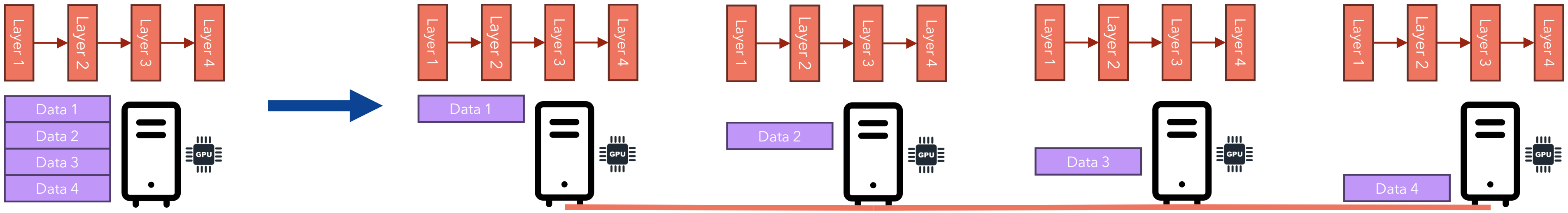


1x Server
8GB RAM
4GB GPU Memory

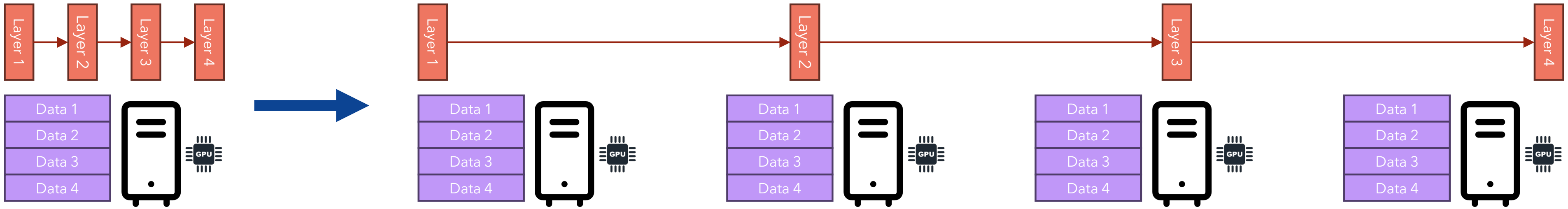
4x Servers
8GB RAM
4GB GPU Memory (x2)

Data Parallelism vs Model Parallelism

If the model **can** fit within a single GPU, then we can distribute the training on multiple servers (each containing one or multiple GPUs), with each GPU processing a subset of the entire dataset in parallel and synchronizing the gradients during backpropagation. This option is known as **Data Parallelism**.



If the model **cannot** fit within a single GPU, then we need to "break" the model into smaller layers and let each GPU process a part of the forward/backward step during gradient descent. This option is known as **Model Parallelism**.



In this video, we will focus on Data Parallelism.

Outline

- Introduction to distributed training
  - Why we need it
  - Data Parallel vs Model Parallel
- Review of neural networks
  - Loss function and gradient
  - Gradient accumulation
- Distributed Data Parallel training
  - How it works
  - Communication primitives
    - Broadcast operator
    - Reduce operator
    - All-Reduce operator
  - Managing failover
- Coding session
  - Infrastructure (Paperspace)
  - PyTorch code
- How PyTorch handles Distributed Data Parallel training
  - Bucketing
  - Computation-Communication overlap during backpropagation

A review of neural networks: a practical example

Imagine you want to train a neural network to predict the price (y_{pred}) of a house given two variables: the number of bedrooms in the house (x_1) and the number of bathrooms in the house (x_2). We think that the relationship between the output and the input variables is linear.

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$

Our goal is to use stochastic gradient descent to find the values of the parameters w_1, w_2 and b such that the MSE loss between the actual house price (y_{target}) and the predicted (y_{pred}) is minimized.

$$\underset{w1,w2,b}{\mathbf{argmin}} \left(y_{pred} - y_{target}\right)^2$$

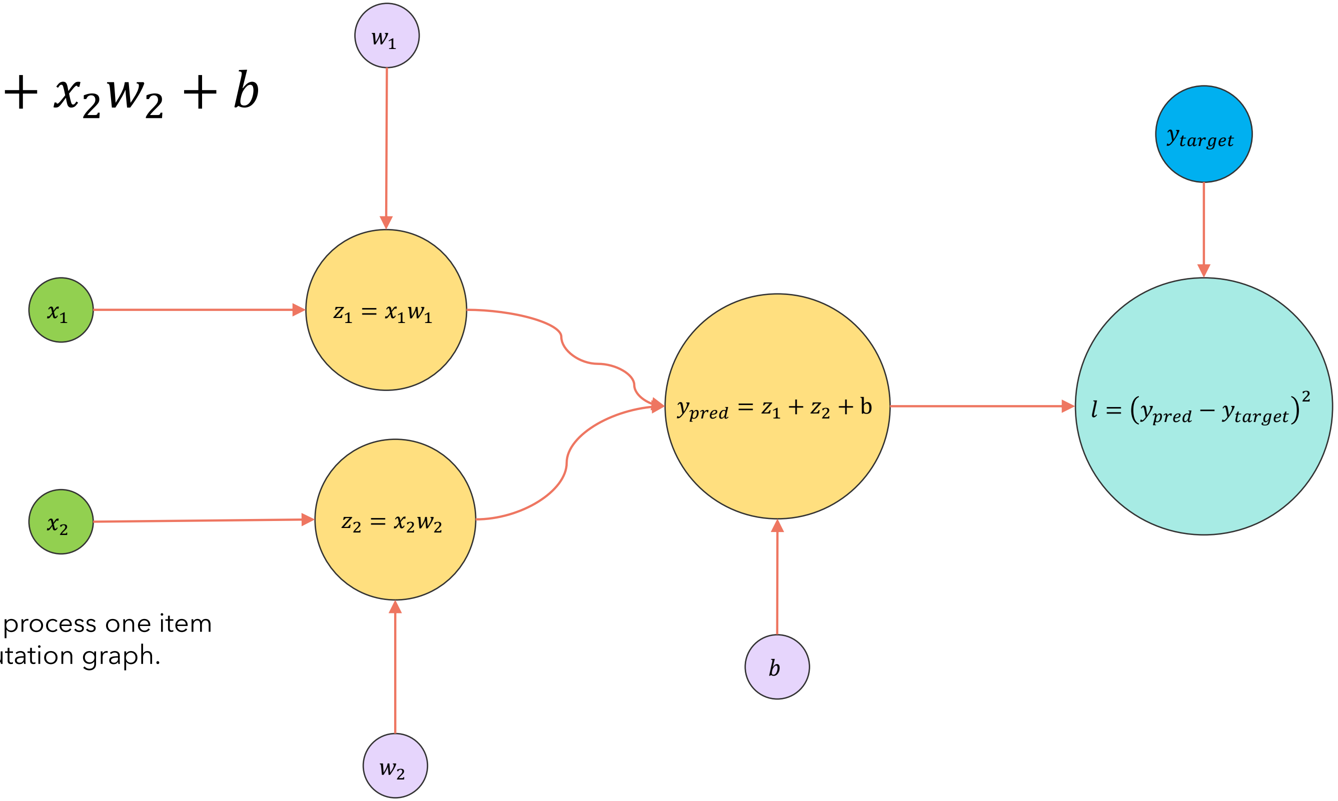
PyTorch's training loop (without accumulation)

```
def train_no_accumulate(params: ModelParameters, num_epochs: int = 10, learning_rate: float = 1e-3):
    for epoch in range(1, num_epochs+1):
        for (x1, x2), y_target in training_data:
            # Calculate the output of the model
            z1 = x1 * params.w1
            z2 = x2 * params.w2
            y_pred = z1 + z2 + params.b
            loss = (y_pred - y_target) ** 2

            # Calculate the gradients of the loss w.r.t. the parameters
            loss.backward()

            # Update the parameters (at each iteration)
            with torch.no_grad():
                # Equivalent to calling optimizer.step()
                params.w1 -= learning_rate * params.w1.grad
                params.w2 -= learning_rate * params.w2.grad
                params.b -= learning_rate * params.b.grad

                # Reset the gradients to zero
                # Equivalent to calling optimizer.zero_grad()
                params.w1.grad.zero_()
                params.w2.grad.zero_()
                params.b.grad.zero_()
```

Computation graph

PyTorch will convert our neural network into a computational graph.

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



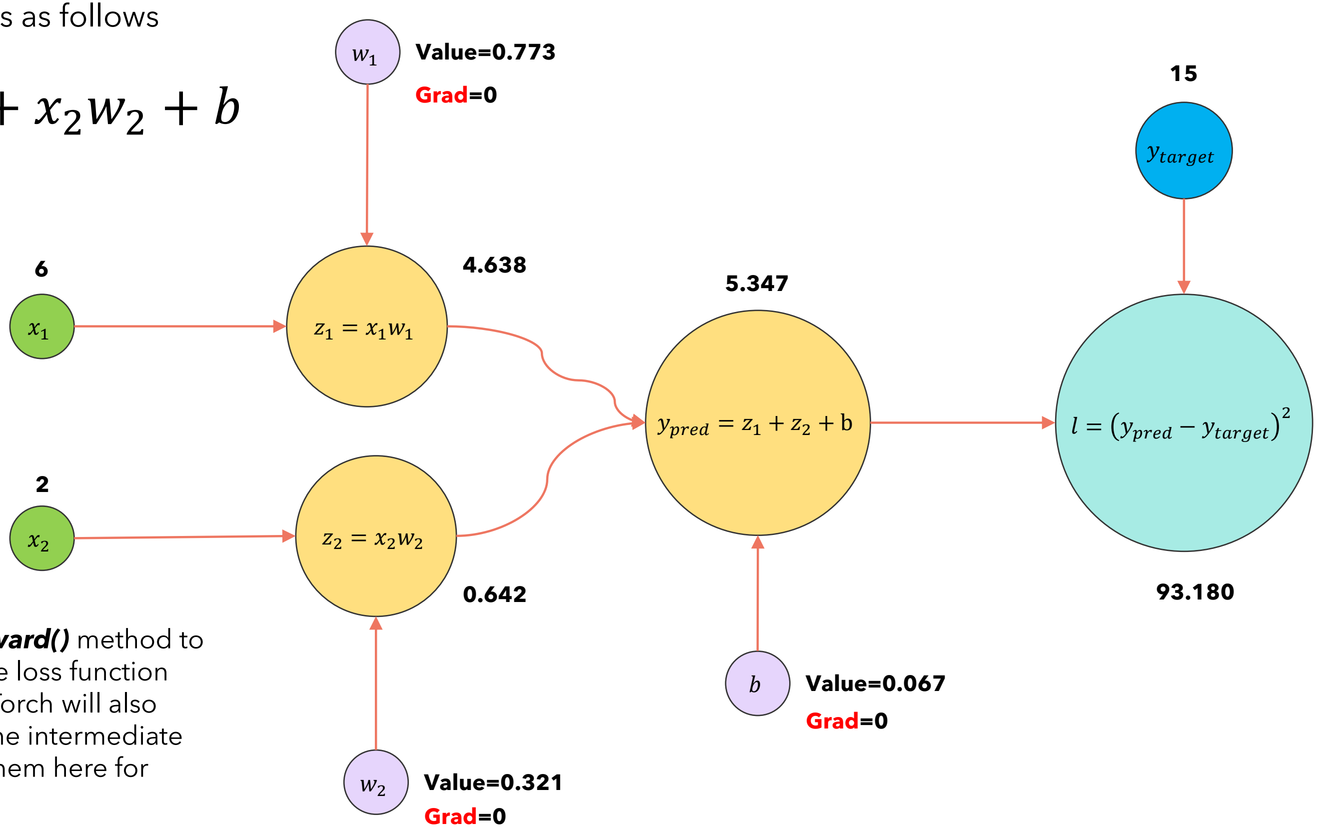
Let's visualize the training process one item at a time using our computation graph.

Computational graph: step 1 (*forward*)

We run a forward step using the input $x_1 = 6$, $x_2 = 2$ and $y_{target} = 15$.

We initialize the weights as follows

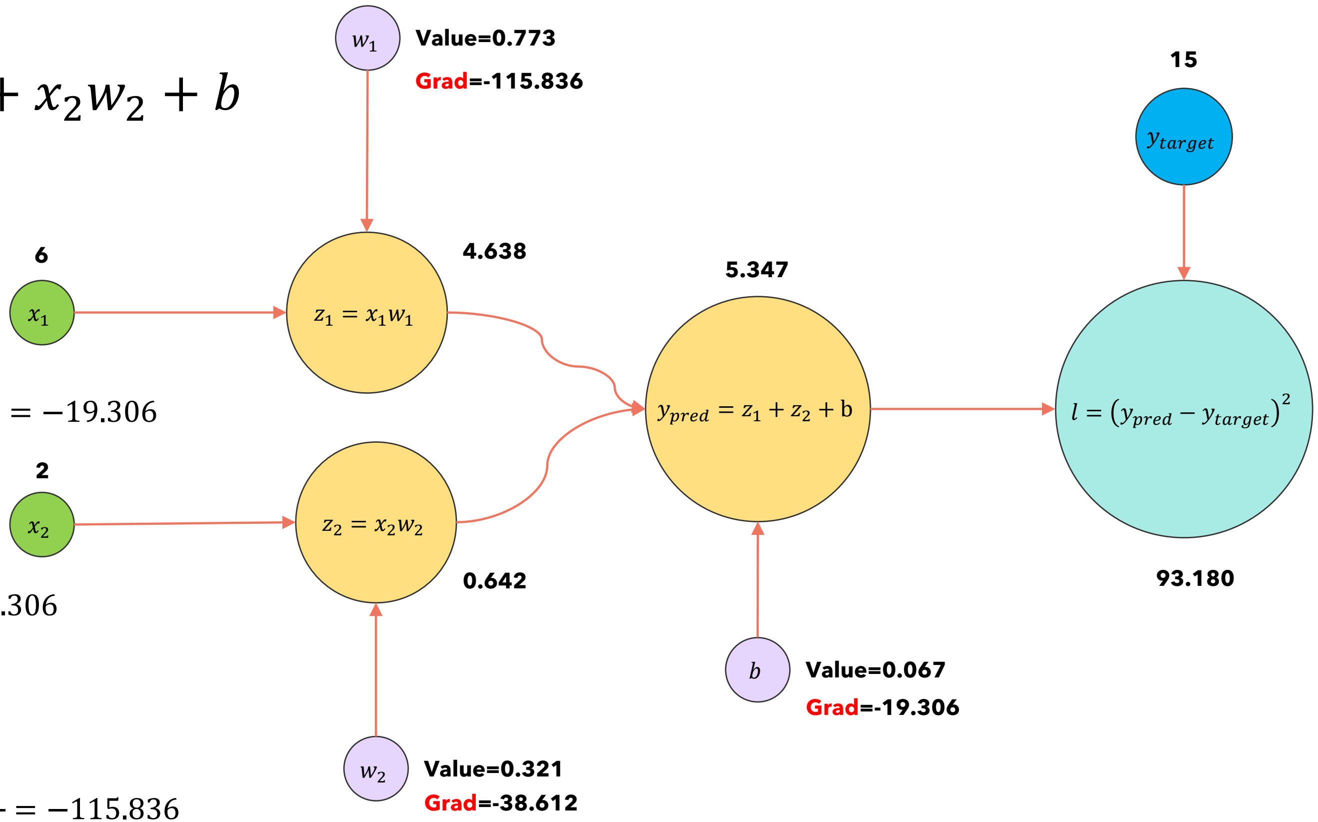
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



Now we call the ***loss.backward()*** method to calculate the gradient of the loss function w.r.t to each parameter. PyTorch will also compute the gradient for the intermediate nodes, but I will not show them here for simplicity.

Computational graph: step 1 (*loss.backward*)

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



$$\frac{dl}{dy_{pred}} = 2y_{pred} - 2y_{target} = -19.306$$

$$\frac{dy_{pred}}{dz_1} = 1$$

$$\frac{dl}{dz_1} = \frac{dl}{dy_{pred}} \times \frac{dy_{pred}}{dz_1} = -19.306$$

$$\frac{dz_1}{dw_1} = x_1 = 6$$

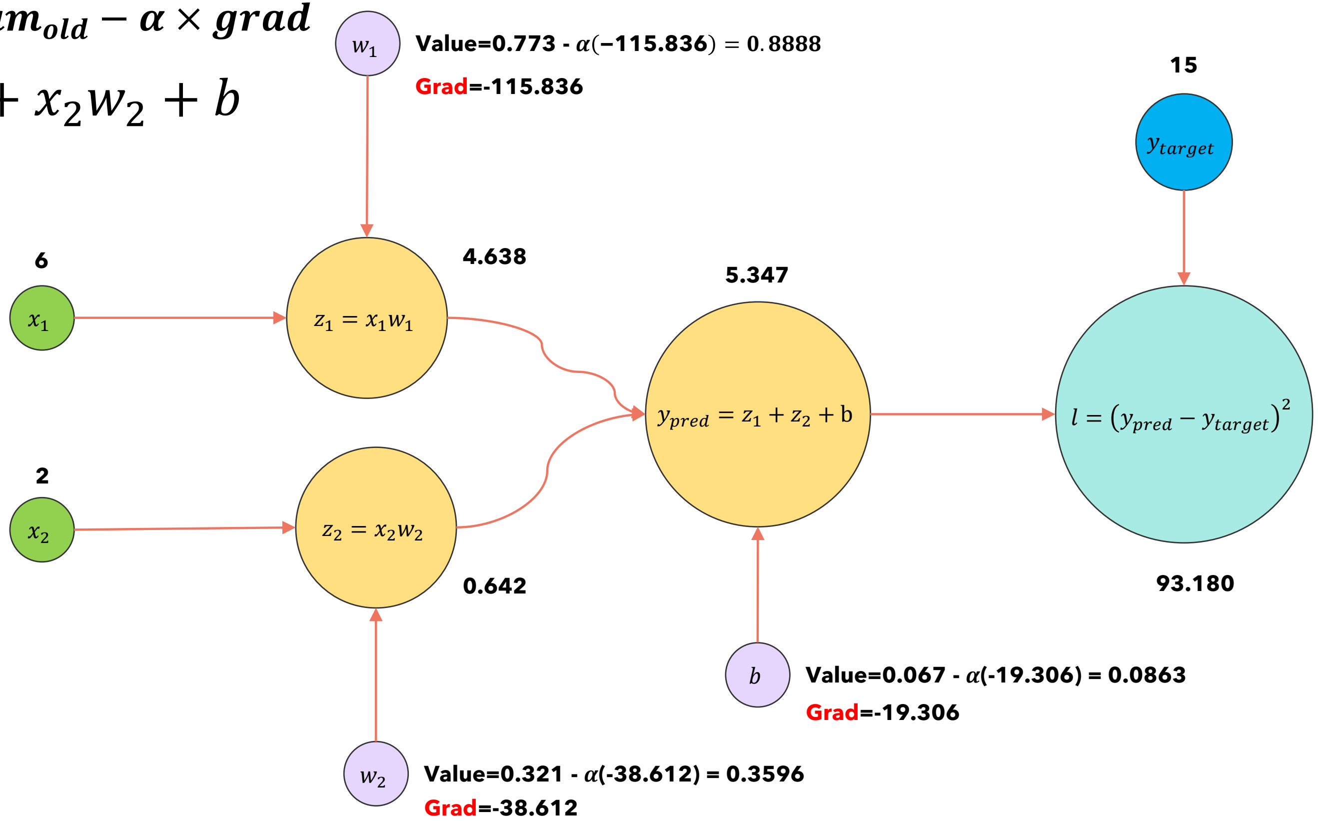
$$\frac{dl}{dw_1} = \frac{dl}{dy_{pred}} \times \frac{dy_{pred}}{dz_1} \times \frac{dz_1}{dw_1} = -115.836$$

Computational graph: step 1 (*optimizer.step*)

Suppose the **learning rate** is $\boldsymbol{\alpha} = \mathbf{10^{-3}}$. Each parameter is updated as follows:

$$\boldsymbol{param_{new} = param_{old} - \alpha \times grad}$$

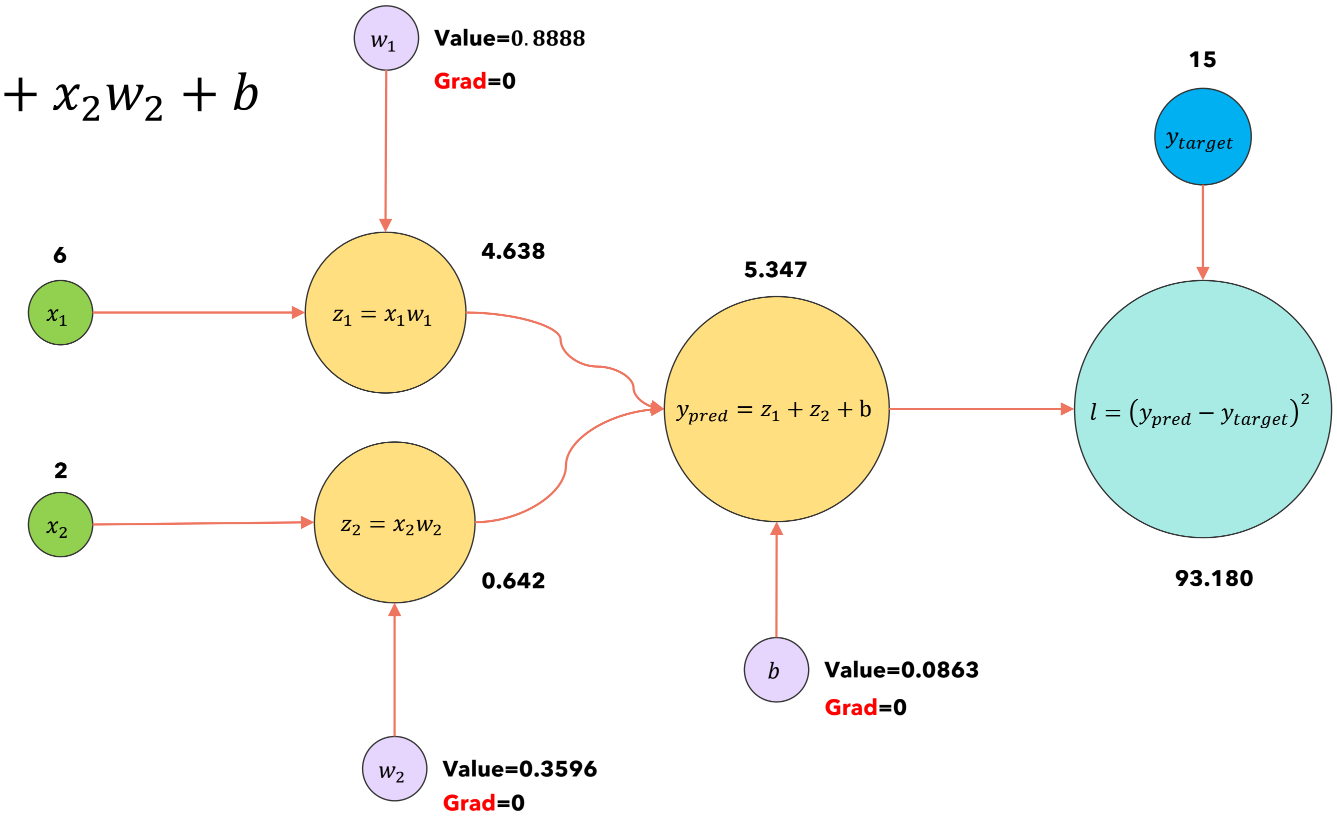
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



Computational graph: step 1 (*optimizer.zero*)

We reset the gradient of all the parameters to zero.

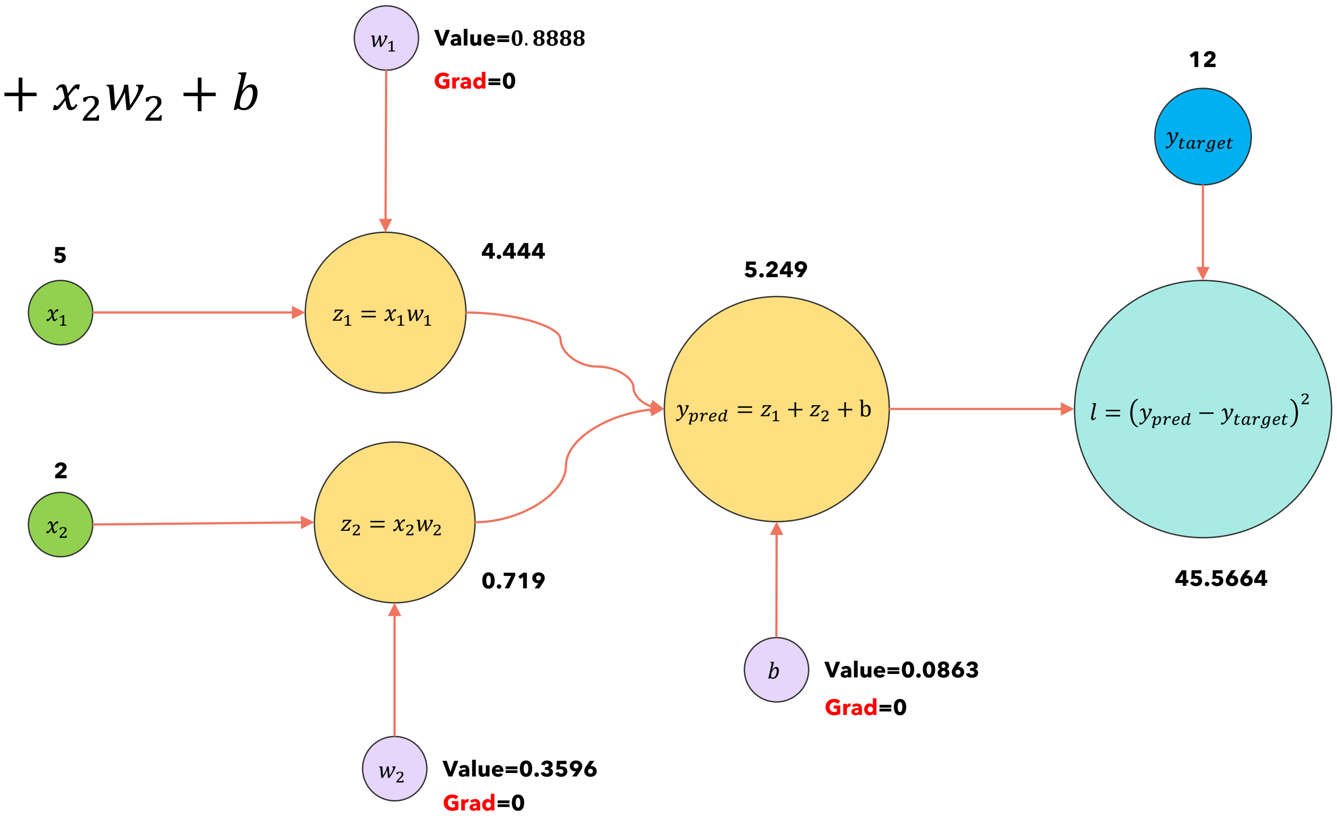
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



Computational graph: step 2 (*forward*)

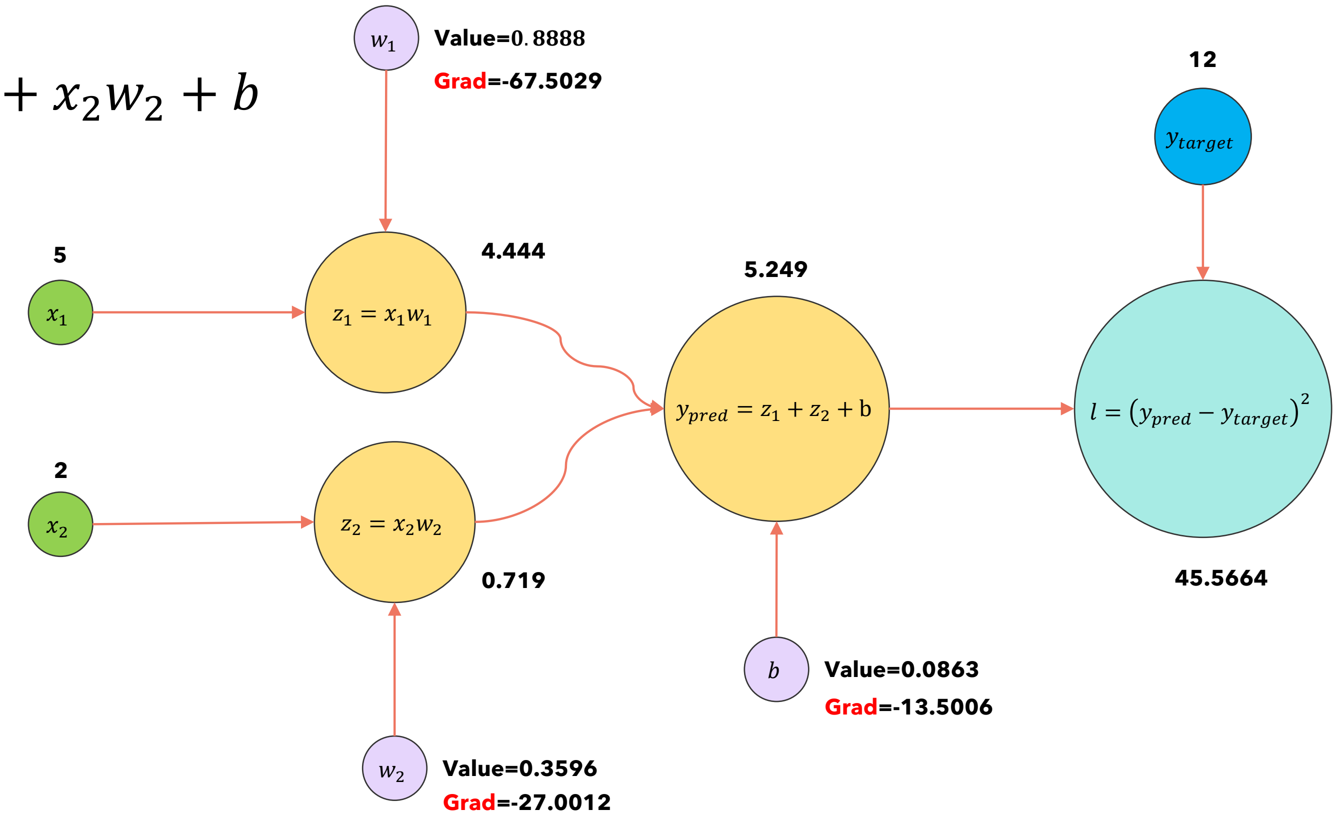
We run a forward step using the input $x_1 = 5$, $x_2 = 2$ and $y_{target} = 12$.

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



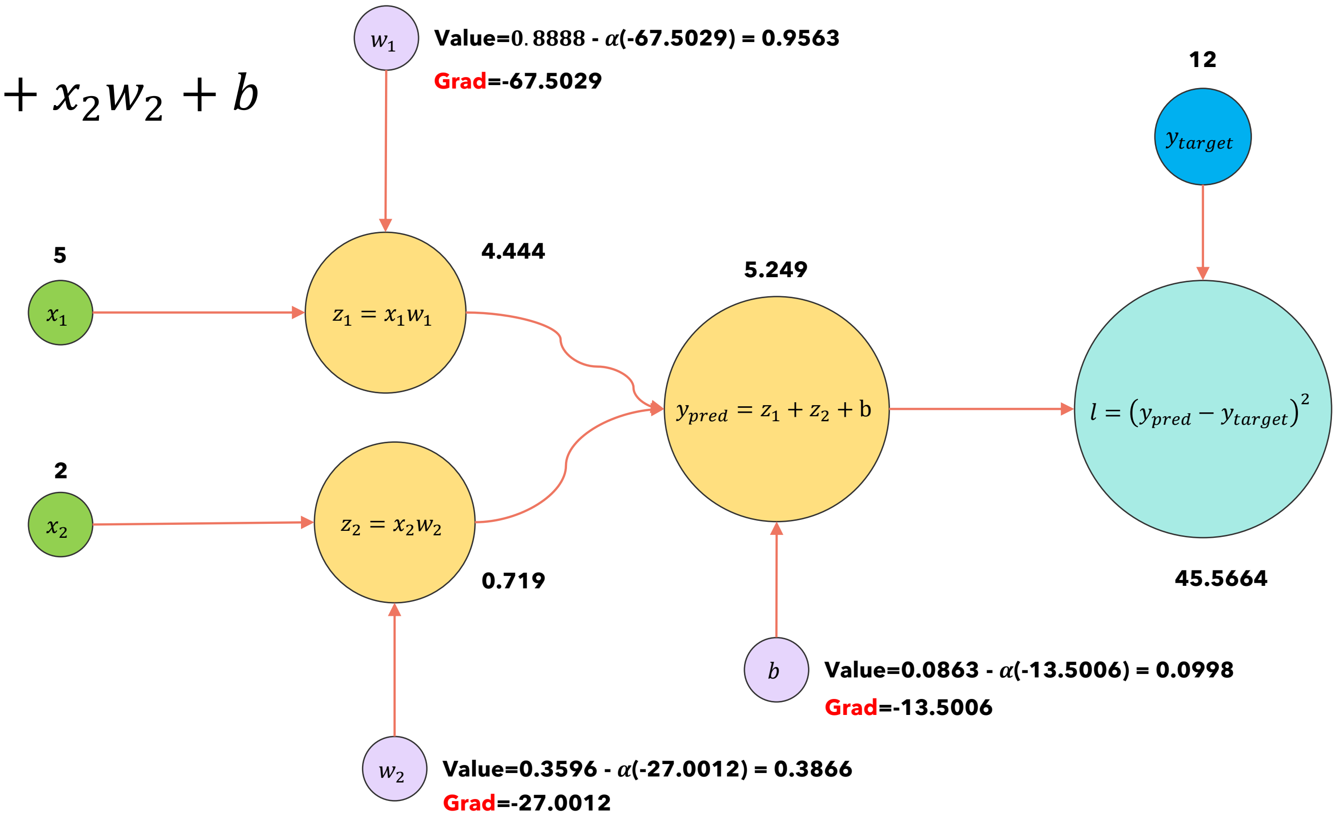
Computational graph: step 2 (*loss.backward*)

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



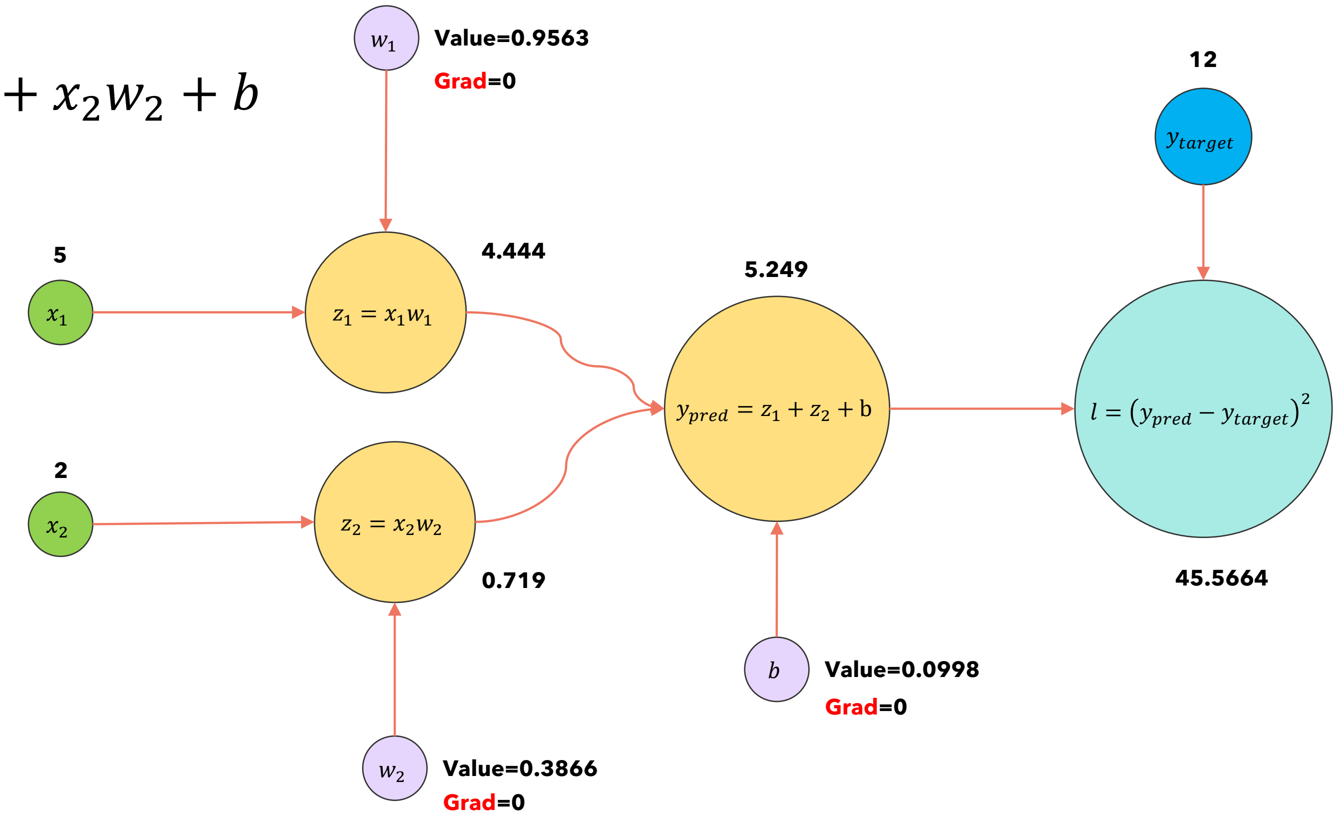
Computational graph: step 2 (*optimizer.step*)

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$

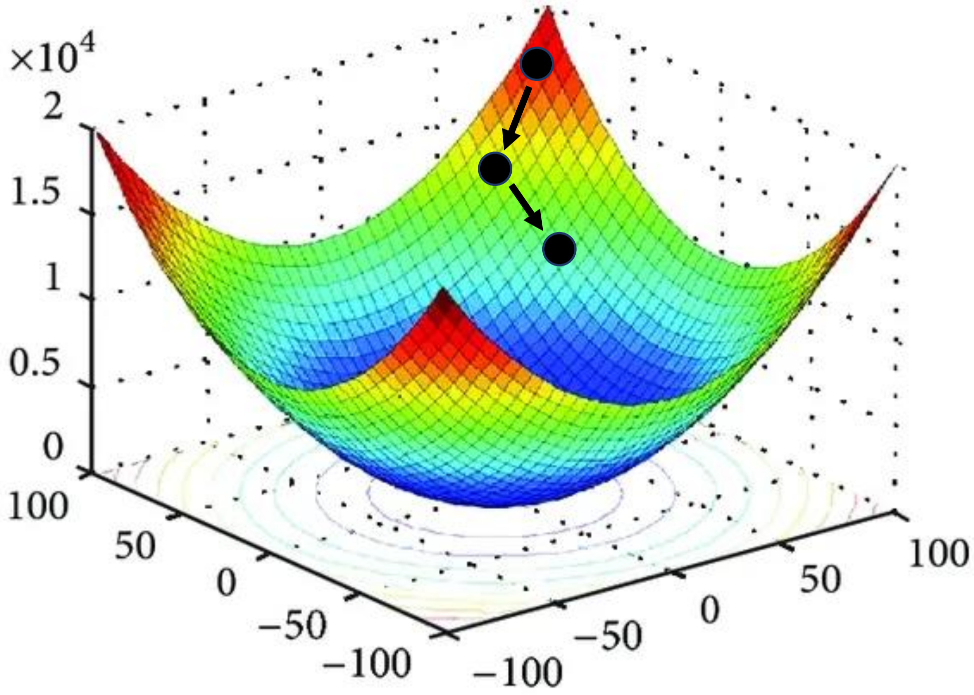


Computational graph: step 2 (*optimizer.zero*)

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



Gradient descent (without accumulation)



Initial weights

Data Item 1 (forward)

Data Item 1 (loss.backward)

Data Item 1 (optimizer.step)

Data Item 1 (optimizer.zero)

Data Item 2 (forward)

Data Item 2 (loss.backward)

Data Item 2 (optimizer.step)

Data Item 2 (optimizer.zero)

Without gradient accumulation, at every step (every data item), we update the parameters of the model.

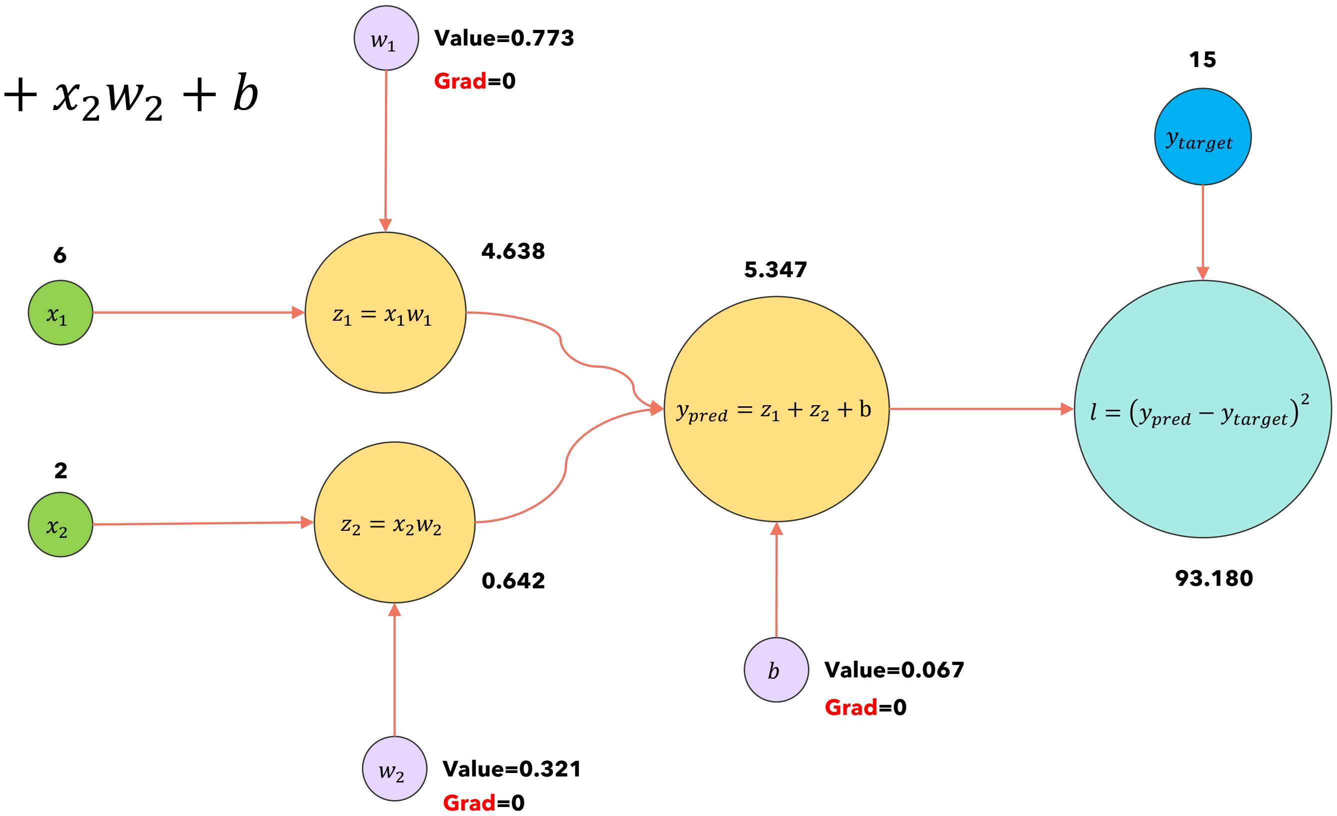
PyTorch's training loop (with accumulation)

```python
def train_accumulate(params: ModelParameters, num_epochs: int = 10, learning_rate: float = 1e-3, batch_size: int = 2):
    for epoch in range(1, num_epochs+1):
        for index, ((x1, x2), y_target) in enumerate(training_data):
            # Calculate the output of the model
            z1 = x1 * params.w1
            z2 = x2 * params.w2
            y_pred = z1 + z2 + params.b
            loss = (y_pred - y_target) ** 2

            # Calculate the gradients of the loss w.r.t. the parameters
            loss.backward()

            # Everytime we reach the batch size or the end of the dataset, update the parameters
            if (index + 1) % batch_size == 0 or index == len(training_data) - 1:
                with torch.no_grad():
                    # Equivalent to calling optimizer.step()
                    params.w1 -= learning_rate * params.w1.grad
                    params.w2 -= learning_rate * params.w2.grad
                    params.b -= learning_rate * params.b.grad

                    # Reset the gradients to zero
                    # Equivalent to calling optimizer.zero_grad()
                    params.w1.grad.zero_()
                    params.w2.grad.zero_()
                    params.b.grad.zero_()
```

Computational graph: step 1 (*forward*)

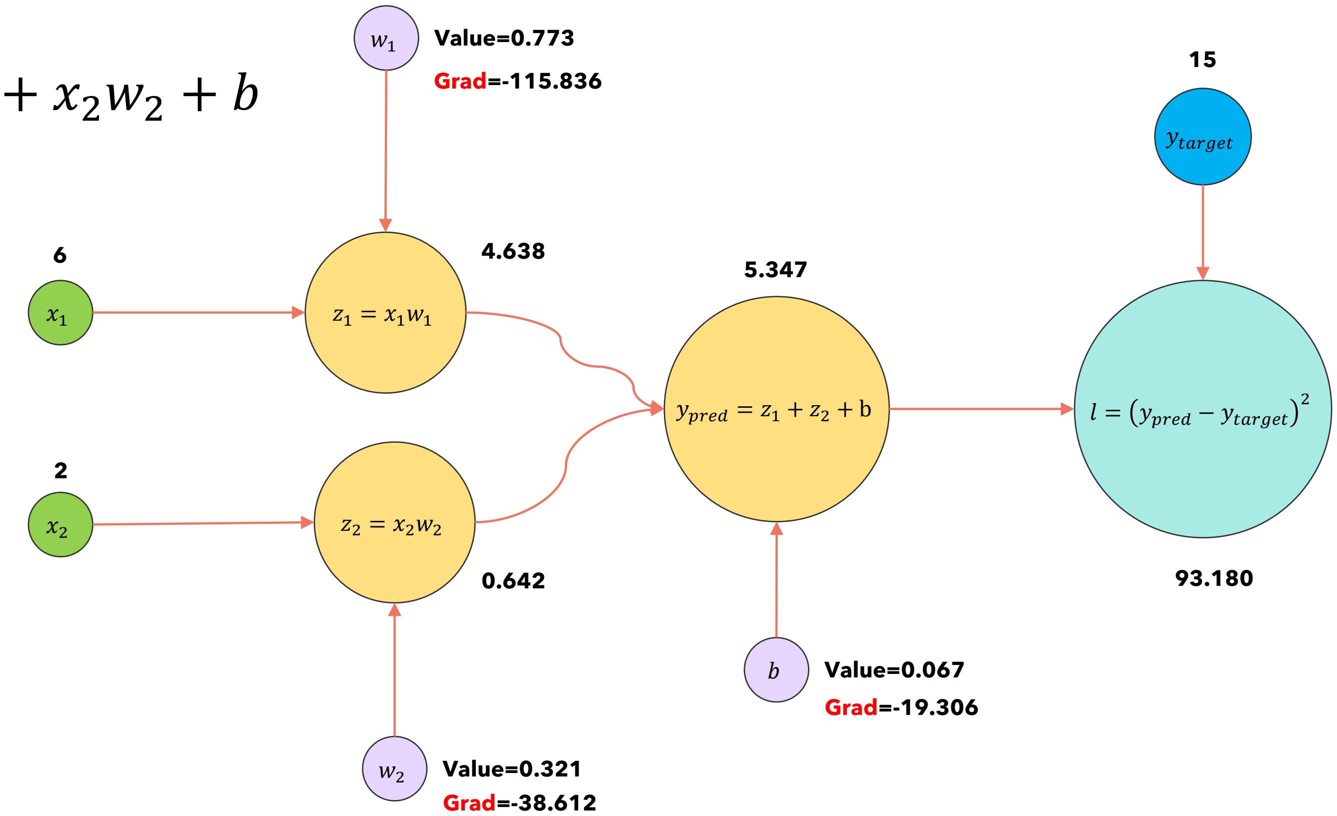
We run a forward step using the input $x_1 = 6$, $x_2 = 2$ and $y_{target} = 15$.

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



Computational graph: step 1 (*loss.backward*)

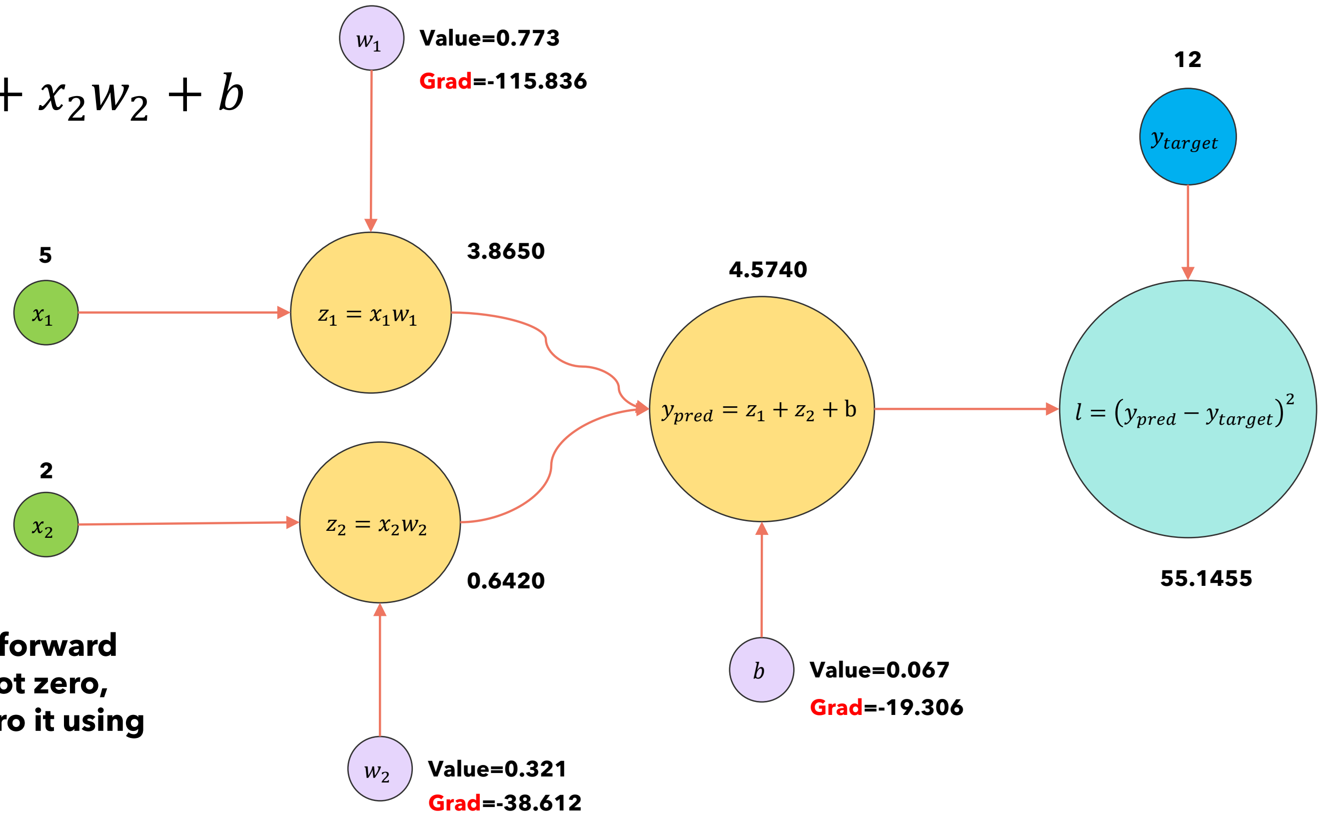
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



Computational graph: step 2 (*forward*)

We run a forward step using the input $x_1 = 5$, $x_2 = 2$ and $y_{target} = 12$.

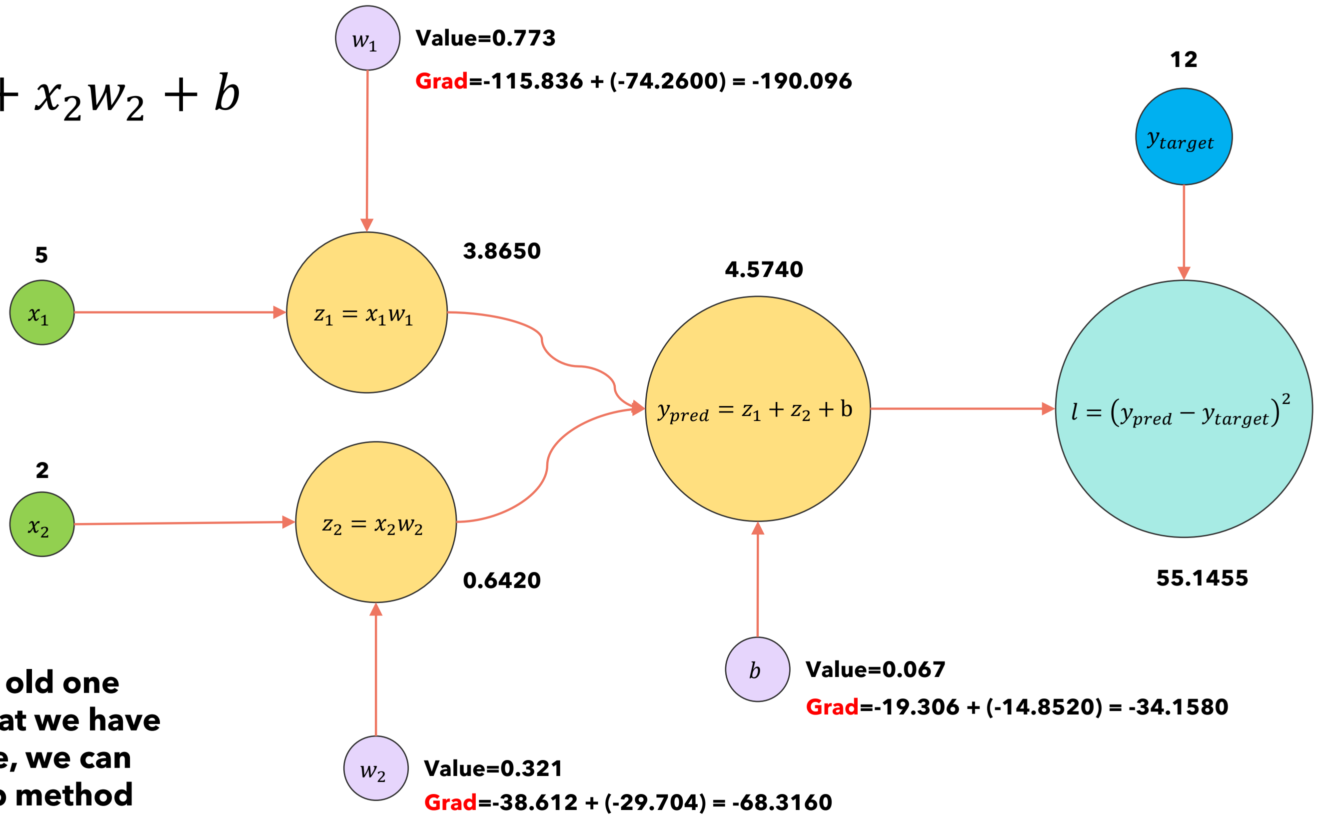
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



Note that during this forward step the gradient is not zero, because we didn't zero it using *optimizer.zero*

Computational graph: step 2 (loss.backward)

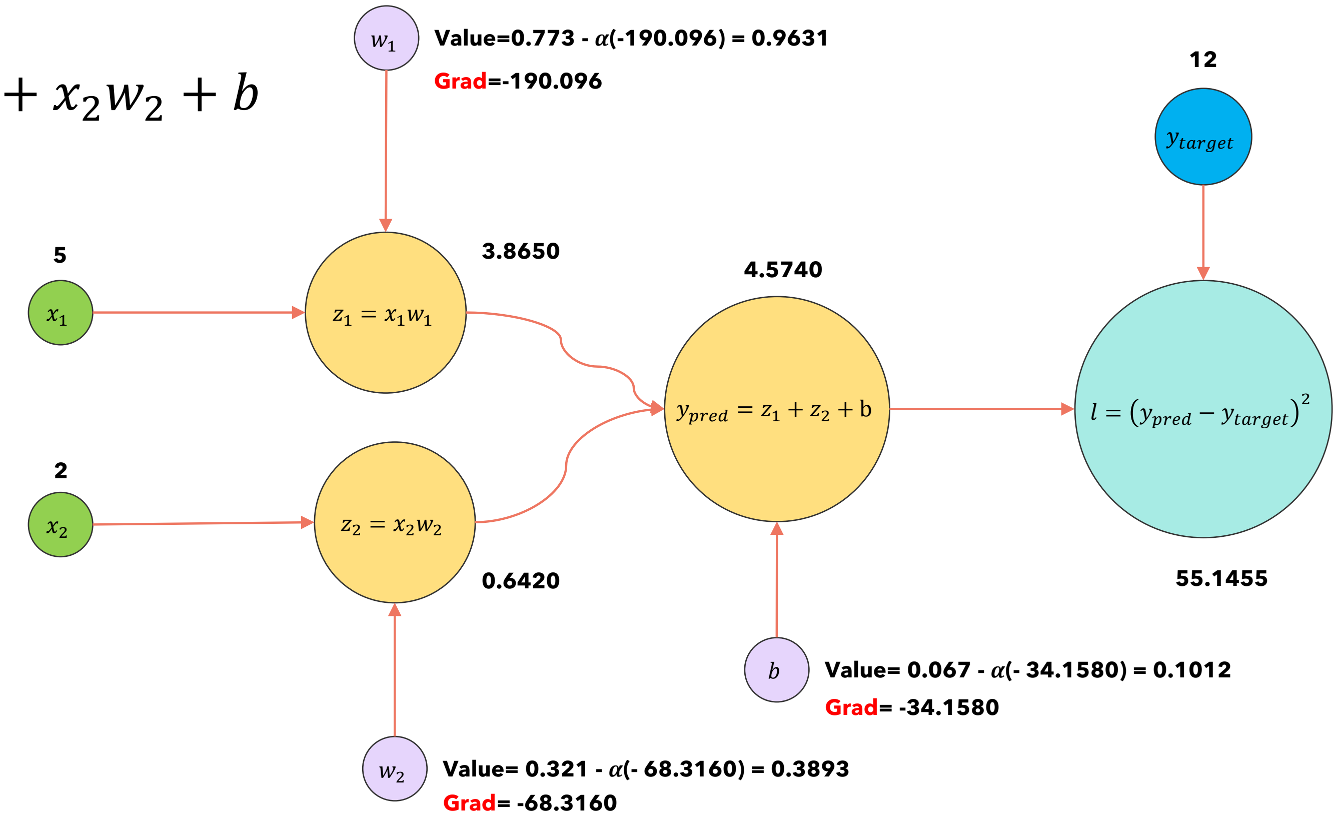
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



The new gradient is accumulated with the old one (summed up). Now that we have reached the batch size, we can run the optimizer.step method

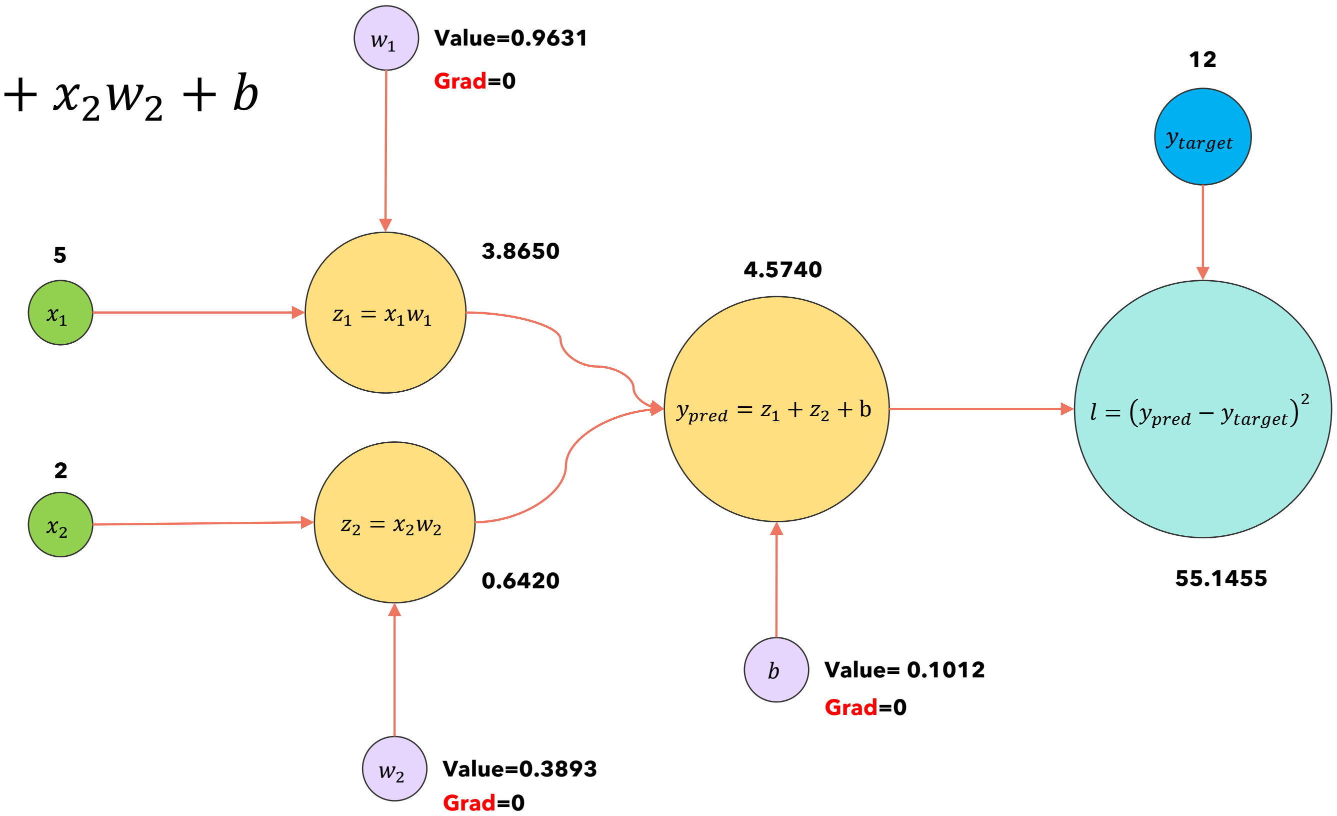
Computational graph: step 2 (optimizer.step)

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$

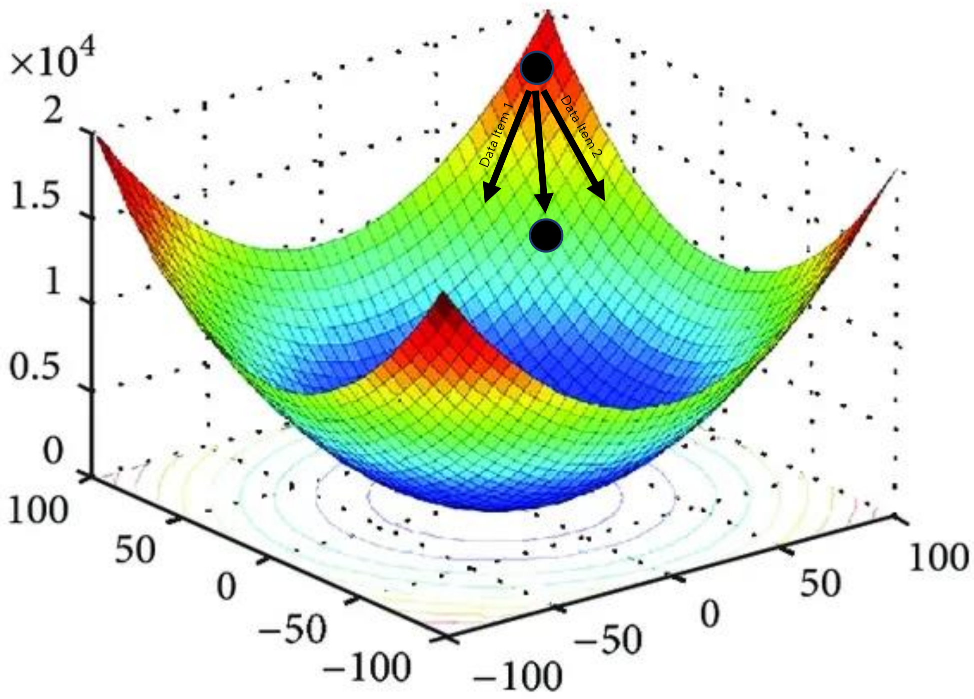


Computational graph: step 2 (optimizer.zero)

$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



Gradient descent (with accumulation)



Initial weights

Data Item 1 (forward)

Data Item 1 (loss.backward)

Data Item 2 (forward)

Data Item 2 (loss.backward)

The two gradients are summed up

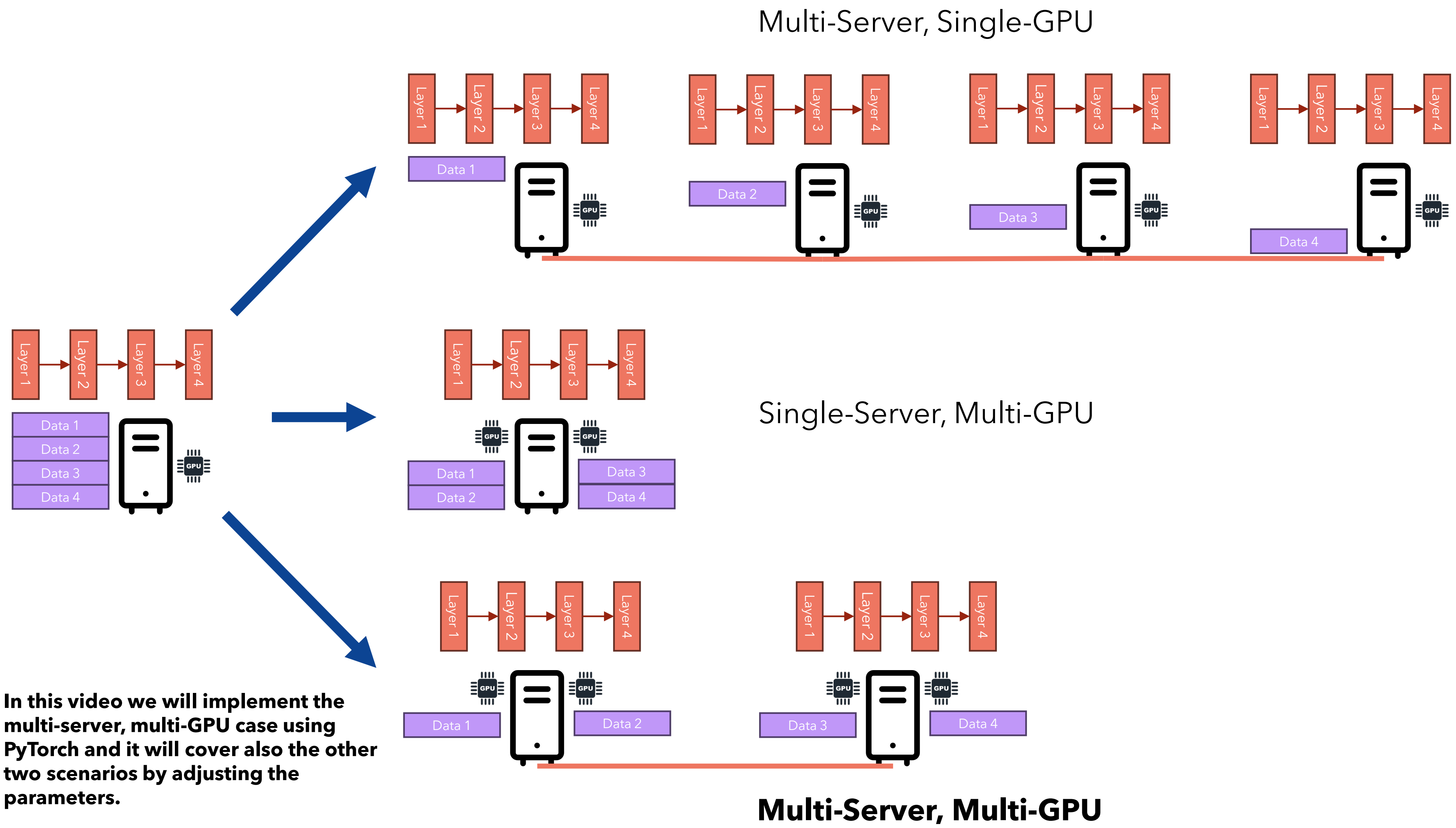
Data Item 2 (optimizer.step)

Data Item 2 (optimizer.zero)

With gradient accumulation, we update the parameters of the model only after we accumulated the gradient of a batch

Outline

- Introduction to distributed training
    - Why we need it
    - Data Parallel vs Model Parallel
- Review of neural networks
    - Loss function and gradient
    - Gradient accumulation
- Distributed Data Parallel training
    - How it works
    - Communication primitives
        - Broadcast operator
        - Reduce operator
        - All-Reduce operator
    - Managing failover
- Coding session
    - Infrastructure (Paperspace)
    - PyTorch code
- How PyTorch handles Distributed Data Parallel training
    - Bucketing
    - Computation-Communication overlap during backpropagation



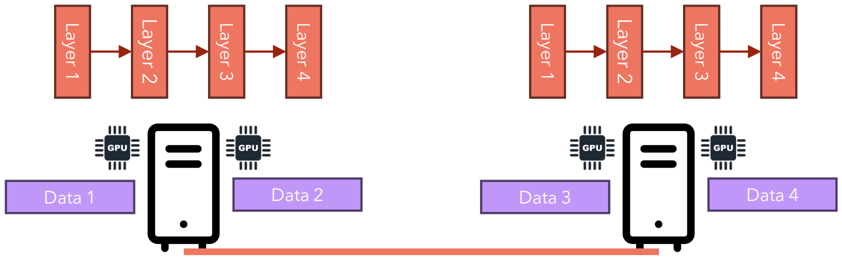
In this video we will implement the multi-server, multi-GPU case using PyTorch and it will cover also the other two scenarios by adjusting the parameters.

Distributed Data Parallel in detail

From now on, I will use the term "*node*" and "*GPU*" interchangeably. If a cluster is made up of 2 computers, each having 2 GPUs, then we have 4 nodes in total.

Distributed Data Parallel works in the following way:

1. At the beginning of the training, the model's weights are initialized on one node and sent to all the other nodes (**Broadcast**)
2. Each node trains the same model (with the same initial weights) on a subset of the dataset.
3. Every few batches, the gradients of each node are accumulated on one node (summed up), and then sent back to all the other nodes (**All-Reduce**).
4. Each node updates the parameters of its local model with the gradients received using its own optimizer.
5. Go back to step 2

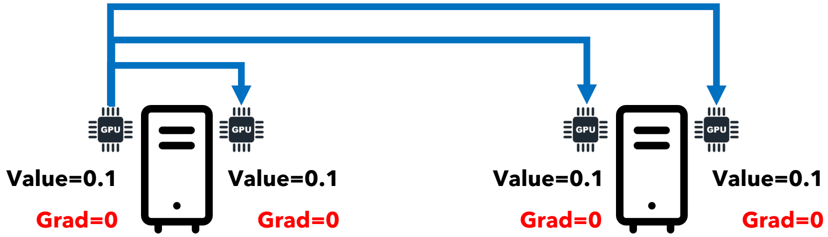


Distributed Data Parallel: step 1

Model weights are initialized here (e.g., randomly)

Distributed Data Parallel: step 1

Initial weights are sent to all the other nodes (Broadcast)

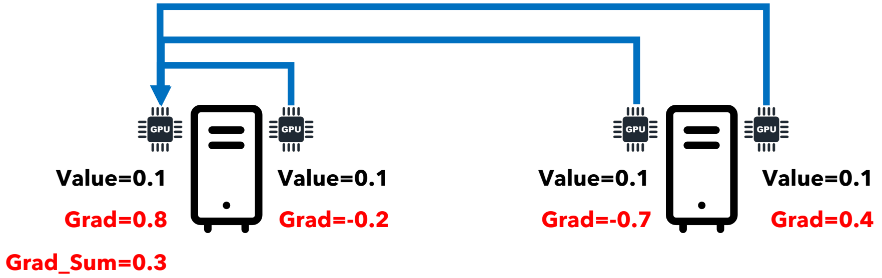


Distributed Data Parallel: step 2

Each node runs a forward and backward step on one or more batch of data.
This will result in a local gradient.
The local gradient may be the accumulation of one or more batches.

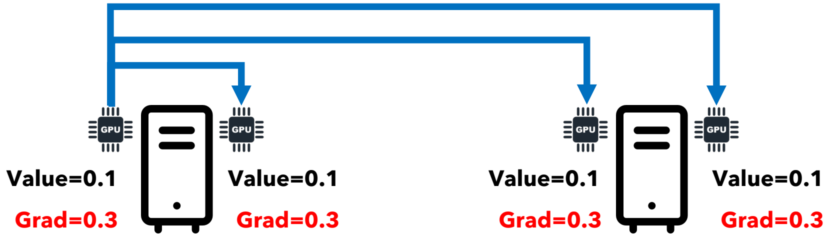
Distributed Data Parallel: step 3

The sum of all the gradients is cumulated on one node (Reduce)



Distributed Data Parallel: step 3

The cumulative gradient is sent to all the other nodes (Broadcast).
The sequence of Reduce and Broadcast are implemented as a single operation (All-Reduce).



Distributed Data Parallel: step 4

Each node updates the parameters of its local model using the gradient received.
After the update, the gradients are reset to zero and we can start another loop.

Collective Communication Primitives

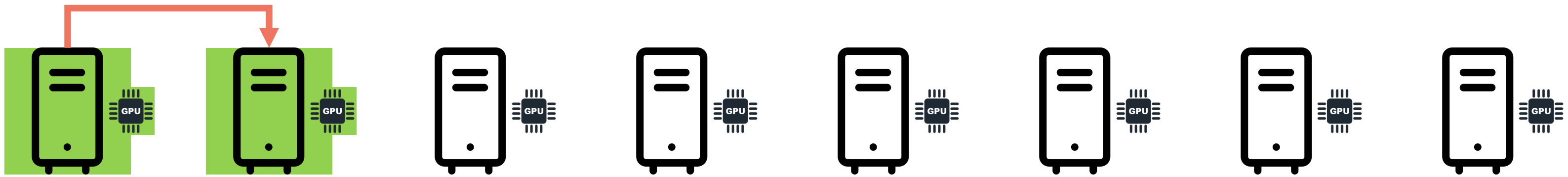
In distributed computing environments, a node may need to communicate with other nodes. If the communication pattern is similar to a client and a server, then we talk about point-to-point communication, because one client connects to one server in a request-response chain of events.

However, there are cases in which one node needs to communicate to multiple receivers *at once*: this is the typical case of data parallel training in deep learning: one node needs to send the initial weights to all the other nodes. Moreover, all the other nodes, need to send their gradients to one single node and receive back the cumulative gradient. **Collective communication allows to model the communication pattern between groups of nodes**.

Let’s visualize the difference between the two modes of communication.

Point-To-Point

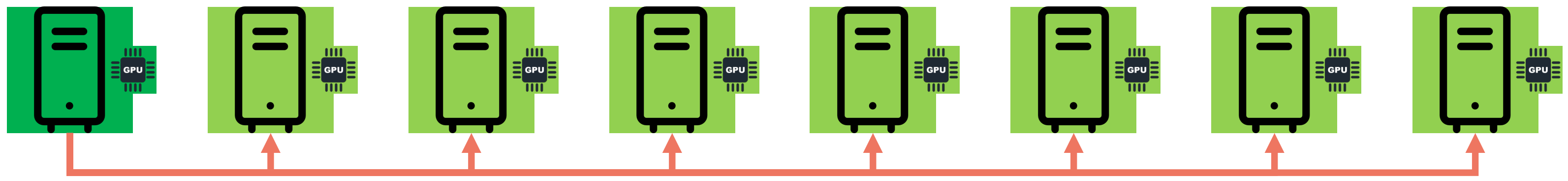
Imagine you need to send a file to 7 friends. With point-to-point communication, you'd send the file iteratively to each of the friend one by one. Suppose the internet speed is 1 MB/s and the file is 5 MB in size.



Total time: 5s

Point-To-Point

Since the internet communication is 1 MB/s and the file is 5 MB in size, your connection would be split among the 7 friends (each friend would be receiving the file at ~ 143 KB/s). **The total time is still 35s.**

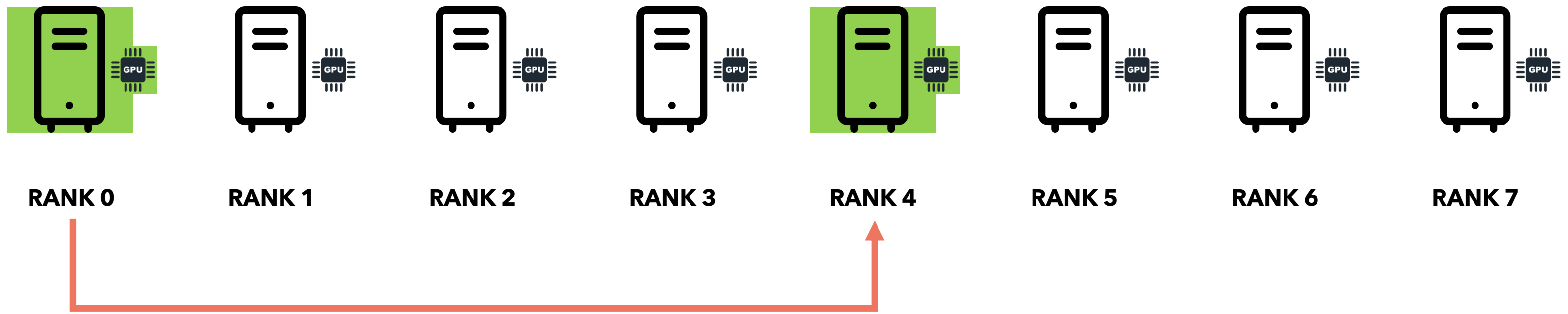


Total time: 35s

Let's see how collective communication would manage this!

Collective Communication: Broadcast

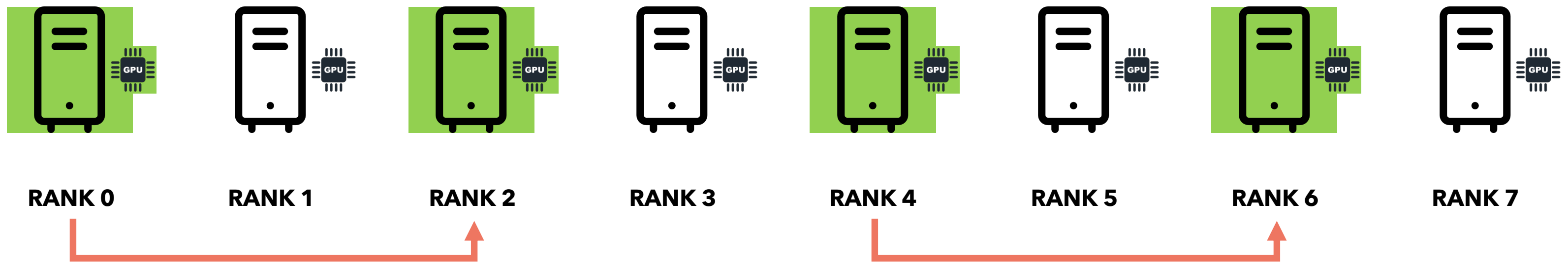
The operation of sending a data to all the other nodes is known as the **Broadcast** operator. Collective Communication libraries (e.g. NCCL) assign a unique ID to each node, known as **RANK**. Suppose we want to send 5 MB with an internet speed of 1 MB/s.



Total time: 5s

Collective Communication: Broadcast

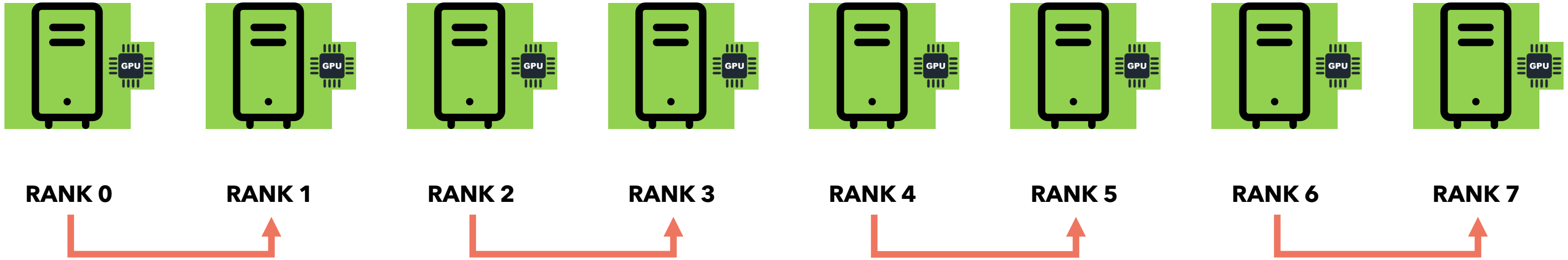
The operation of sending a data to all the other nodes is known as the Broadcast operator. Collective Communication libraries (e.g. NCCL) assign a unique ID to each node, known as **RANK**. Suppose we want to send 5 MB with an interned speed of 1 MB/s.



Total time: 10s

Collective Communication: Broadcast

The operation of sending a data to all the other nodes is known as the Broadcast operator. Collective Communication libraries (e.g. NCCL) assign a unique ID to each node, known as **RANK**. Suppose we want to send 5 MB with an interned speed of 1 MB/s.



Total time: 15s

This approach is known as Divide-and-Conquer. With collective communication, we exploit the interconnectivity between nodes to avoid idle times and reduce the total communication time.

Reduce operation

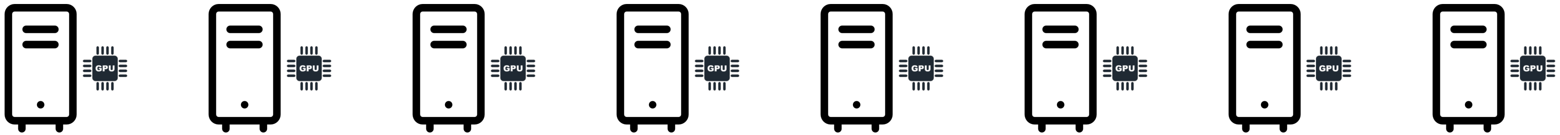
The **Broadcast** operator is used to send the initial weights to all the other nodes when we start the training loop.

At every few batches of data processed by each node, the gradients of all nodes need to be sent to one node and accumulated (summed up). This operation is known as **Reduce**.

Let’s visualize how it works.

Collective Communication: Reduce

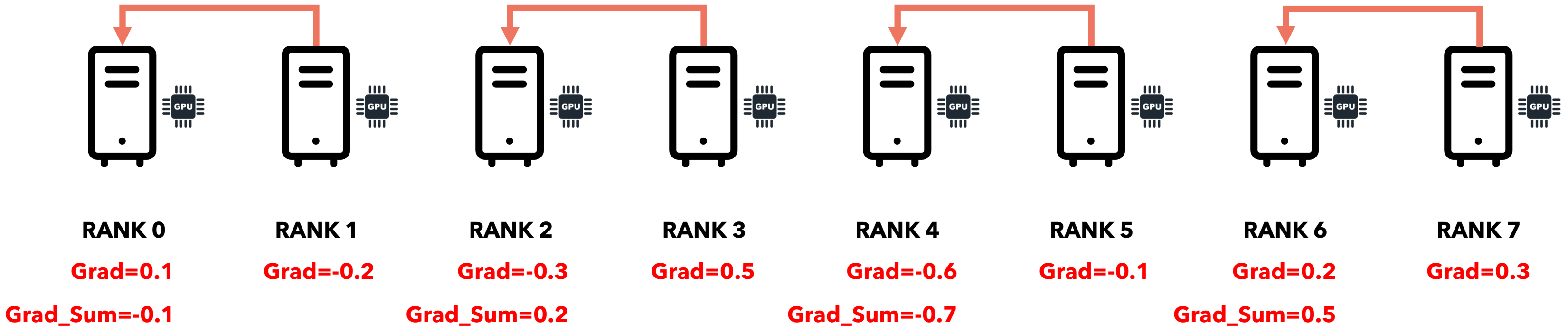
Initially, each node has its own gradient.



RANK 0	RANK 1	RANK 2	RANK 3	RANK 4	RANK 5	RANK 6	RANK 7
Grad=0.1	**Grad=-0.2**	**Grad=-0.3**	**Grad=0.5**	**Grad=-0.6**	**Grad=-0.1**	**Grad=0.2**	**Grad=0.3**

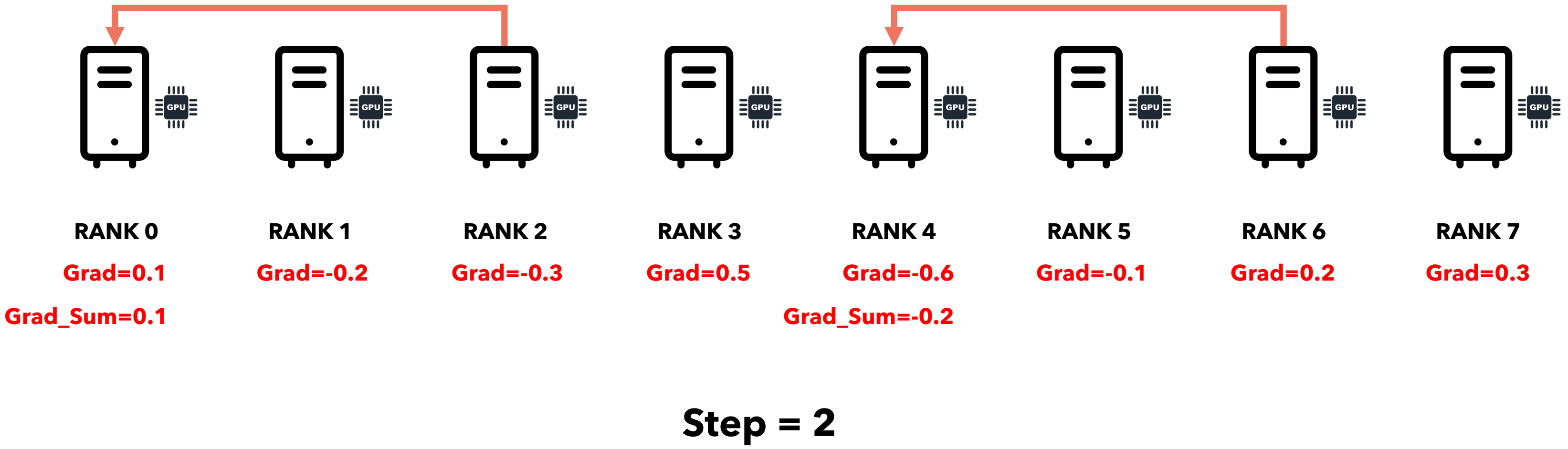
Collective Communication: Reduce

Each node sends the gradient to its adjacent node, who will sum it with its own gradient.

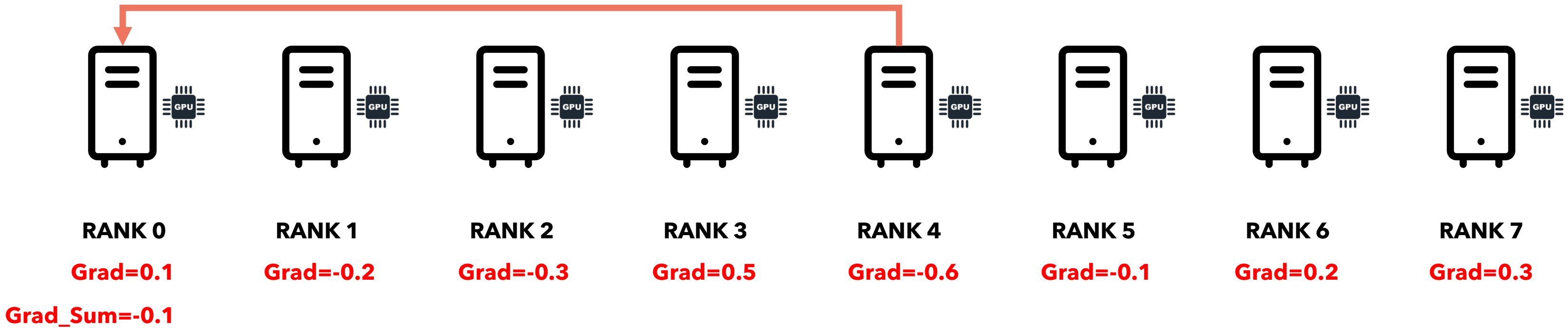


Step = 1

Collective Communication: Reduce



Collective Communication: Reduce



Step = 3

With only 3 steps we accumulated the gradient of all nodes into one node. It can be proven that the communication time is logarithmic w.r.t the number of nodes.

Collective Communication: All-Reduce

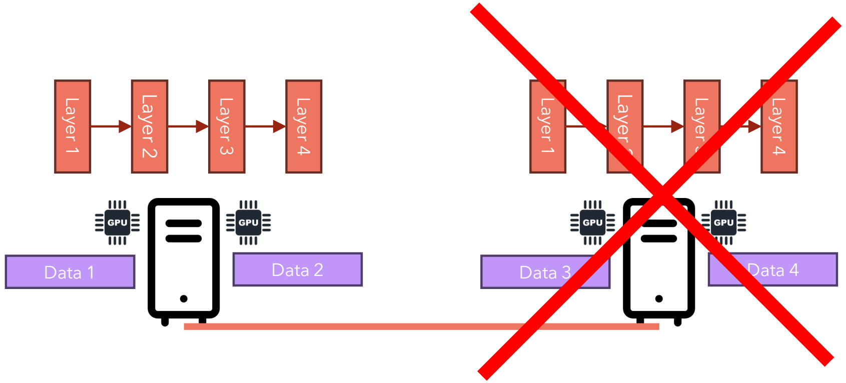
Having accumulated the gradients of all the nodes into a single node, we need to send the cumulative gradient to all the nodes. This operation can be done using a **Broadcast** operator.

The sequence of **Reduce-Broadcast** is implemented by another operator known as **All-Reduce**, whose runtime is generally lower than the sequence of **Reduce** followed by a **Broadcast**.

I will not show the algorithm behind **All-Reduce**, but you can think of it as a sequence of **Reduce** followed by a **Broadcast** operation.

Failover: what happens if one node crashes?

Imagine you're training in a distributed scenario like the one shown below and one of the nodes suddenly crashes. In these case, 2 GPUs out of 4 become unreachable. How should the system react?

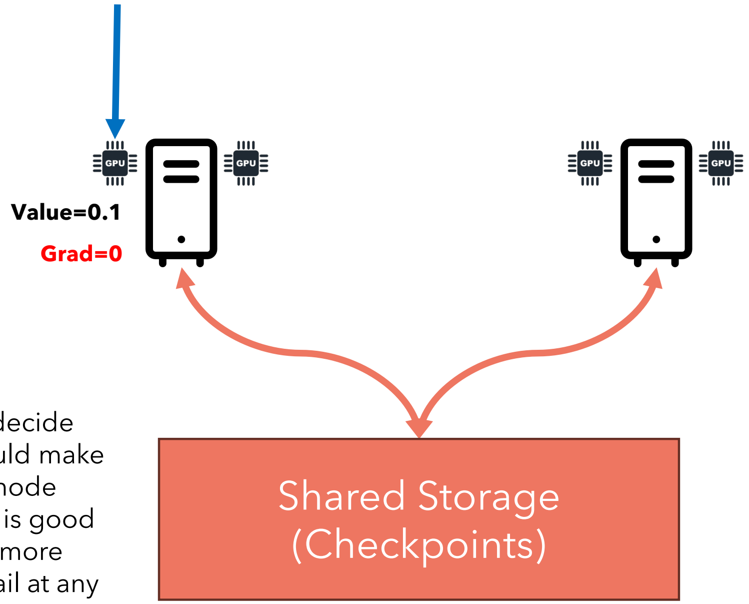


One way, would be to restart the entire cluster and that's easy. However, by restarting the cluster, the training would restart from zero, and we would lose all the parameters and computation done so far. **A better approach is to use checkpointing.**

Checkpointing means saving the weights of the model on a shared disk every few iterations (for example every epoch) and resume the training from the last checkpoint in case there's a crash.

Failover: using checkpointing

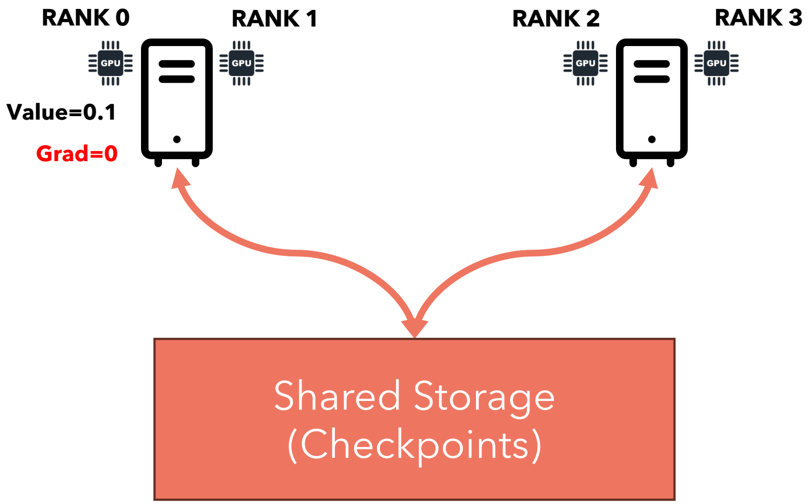
Model weights are initialized here (**using the latest checkpoint**)



We need a shared storage because PyTorch will decide which node will initialize the weights and we should make no assumption on which one will it be. So, every node should have access to the shared storage. Plus, it is good rule in distributed systems to not have one node more important than others, because every node can fail at any time.

Failover: who should save the checkpoint?

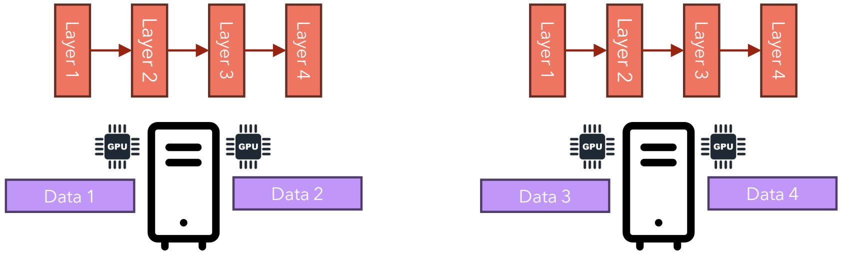
When we start the cluster, PyTorch will assign a unique ID (**RANK**) to each GPU. We will write our code in such a way that whichever node is assigned the **RANK 0** will be responsible for saving the checkpoint, so that the other nodes do not overwrite each other's files. So only one node will be responsible for writing the checkpoints and all the other files we need for training.



LOCAL_RANK vs RANK

The environment variable **LOCAL_RANK** indicates the ID of the GPU on the local computer, while the **RANK** variable indicates the a globally unique ID among all the nodes in the cluster.

Please note that ranks are not *stable*, meaning that if you restart the entire cluster, a different node may be assigned the rank number 0.



LOCAL_RANK	0	1	0	1
RANK	0	1	2	3

Used to save checkpoint and/or initialize services (like W&B)

How to integrate DistributedDataParallel into your project?

```python
def train():
    if global_rank == 0:
        initialize_services() # W&B, etc.

    data_loader = DataLoader(train_dataset, shuffle=False, sampler=DistributedSampler(train_dataset, shuffle=True))
    model = MyModel()
    if os.path.exists('latest_checkpoint.pth'): # Load latest checkpoint
        # Also load optimizer state and other variables needed to restore the training state
        model.load_state_dict(torch.load('latest_checkpoint.pth'))

    model = DistributedDataParallel(model, device_ids=[local_rank])
    optimizer = torch.optim.Adam(model.parameters(), lr=10e-4, eps=1e-9)
    loss_fn = torch.nn.CrossEntropyLoss()

    for epoch in range(num_epochs):
        for data, labels in data_loader:
            loss = loss_fn(model(data), labels) # Forward step
            loss.backward() # Backward step + gradient synchronization
            optimizer.step() # Update weights
            optimizer.zero_grad() # Reset gradients to zero

            if global_rank == 0:
                collect_statistics() # W&B, etc.

        if global_rank == 0: # Only save on rank 0
            # Also save the optimizer state and other variables needed to restore the training state
            torch.save(model.state_dict(), 'latest_checkpoint.pth')


if __name__ == '__main__':
    local_rank = int(os.environ['LOCAL_RANK'])
    global_rank = int(os.environ['RANK'])

    init_process_group(backend='nccl')
    torch.cuda.set_device(local_rank) # Set the device to local rank

    train()

    destroy_process_group()
```

Outline

- Introduction to distributed training
  - Why we need it
  - Data Parallel vs Model Parallel
- Review of neural networks
  - Loss function and gradient
  - Gradient accumulation
- Distributed Data Parallel training
  - How it works
  - Communication primitives
    - Broadcast operator
    - Reduce operator
    - All-Reduce operator
  - Managing failover
- Coding session
  - Infrastructure (Paperspace)
  - PyTorch code
- How PyTorch handles Distributed Data Parallel training
  - Bucketing
  - Computation-Communication overlap during backpropagation

Prerequisites

- Basic understanding of neural networks and PyTorch
- (Optional) watch my previous video on how to code a Transformer model from scratch

When does PyTorch synchronize gradients?

PyTorch will synchronize the gradients every time we call the method **loss.backward**. This will lead to:

1. Each node calculating its local gradients (derivative of the loss function w.r.t each node of the computational graph)
2. Each node will send its local gradient to one single node and receives back the cumulative gradient (**All-Reduce**)
3. Each node will update its weights using the cumulative gradient and its local optimizer.

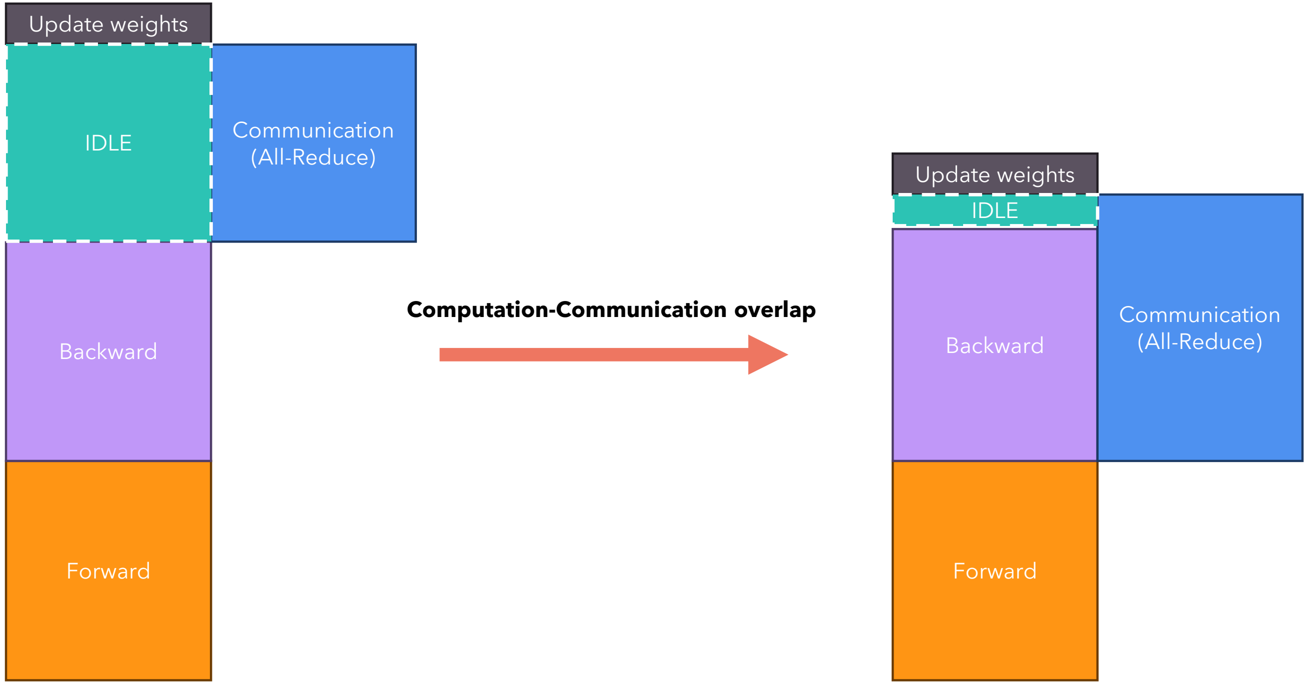
We can avoid PyTorch synchronizing the gradient at every backward step and instead, let it accumulate the gradient for a few steps by using the **no_sync()** context. Let's see how it works.

When does PyTorch synchronize gradients?

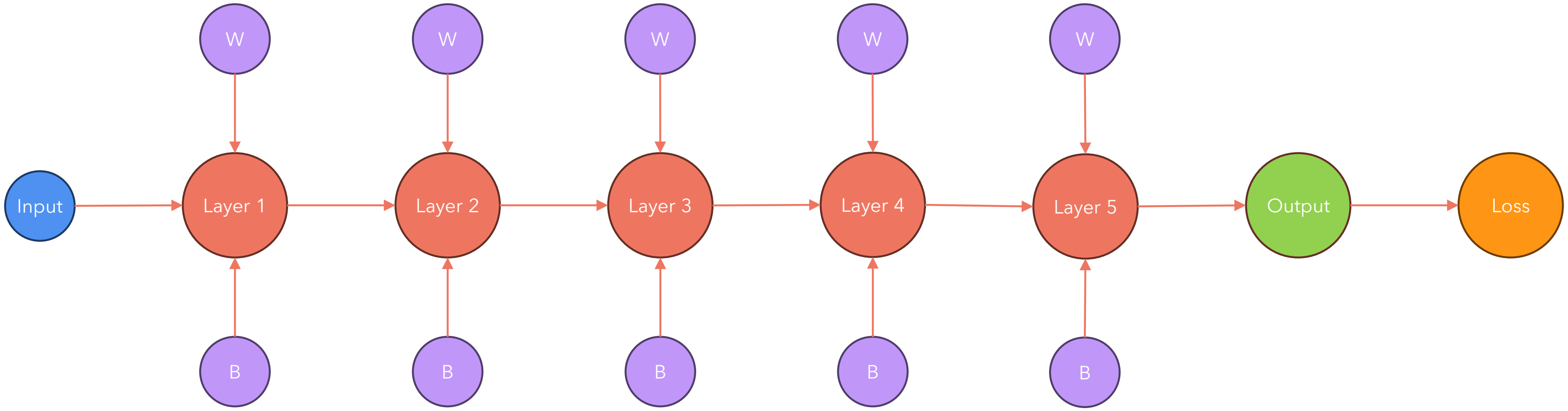
```python
def train():
    if global_rank == 0:
        initialize_services() # W&B, etc.

    data_loader = DataLoader(train_dataset, shuffle=False, sampler=DistributedSampler(train_dataset, shuffle=True))
    model = MyModel()
    if os.path.exists('latest_checkpoint.pth'): # Load latest checkpoint
        # Also load optimizer state and other variables needed to restore the training state
        model.load_state_dict(torch.load('latest_checkpoint.pth'))

    model = DistributedDataParallel(model, device_ids=[local_rank])
    optimizer = torch.optim.Adam(model.parameters(), lr=10e-4, eps=1e-9)
    loss_fn = torch.nn.CrossEntropyLoss()

    for epoch in range(num_epochs):
        for data, labels in data_loader:
            if (step_number + 1) % 100 != 0 and not last_step: # Accumulate gradients for 100 steps
                with model.no_sync(): # Disable gradient synchronization
                    loss = loss_fn(model(data), labels) # Forward step
                    loss.backward() # Backward step + gradient ACCUMULATION
            else:
                loss = loss_fn(model(data), labels) # Forward step
                loss.backward() # Backward step + gradient SYNCHRONIZATION
                optimizer.step() # Update weights
                optimizer.zero_grad() # Reset gradients to zero

            if global_rank == 0:
                collect_statistics() # W&B, etc.

        if global_rank == 0: # Only save on rank 0
            # Also save the optimizer state and other variables needed to restore the training state
            torch.save(model.state_dict(), 'latest_checkpoint.pth')


if __name__ == '__main__':
    local_rank = int(os.environ['LOCAL_RANK'])
    global_rank = int(os.environ['RANK'])

    init_process_group(backend='nccl')
    torch.cuda.set_device(local_rank) # Set the device to local rank

    train()

    destroy_process_group()
```

PyTorch tricks: Computation-Communication overlap

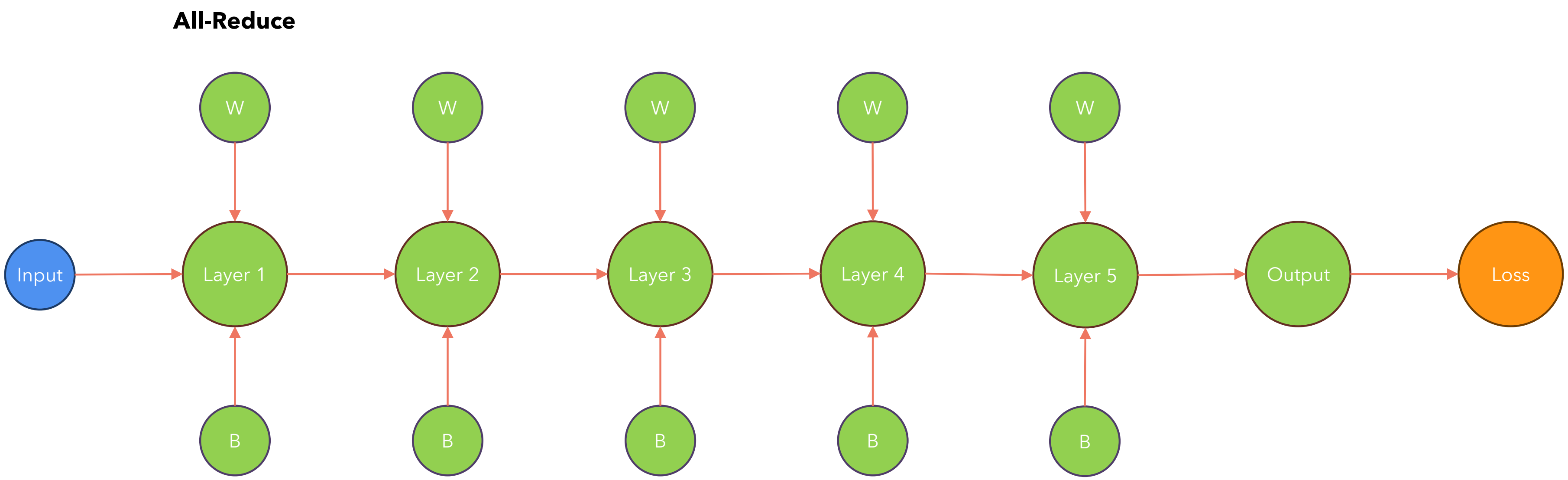
Since each GPU needs to send its gradient to a central node for accumulation, this can lead to an idle time in which the GPUs are not working, but only communicating with each other. PyTorch handles this communication delay in a smart way. Let's see how it works.



Computation-Communication overlap: details

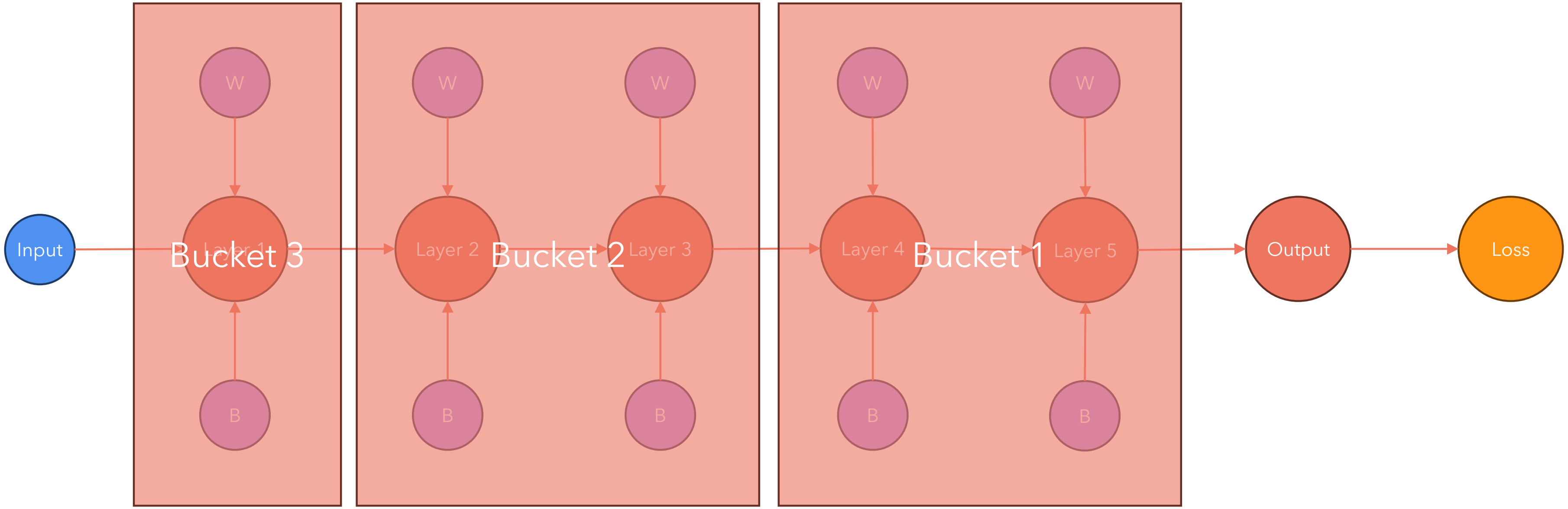


Computation-Communication overlap: details



Computation-Communication overlap: bucketing

Instead of sending each gradient one by one, which would result in a large communication overhead, gradients are packed together into buckets of equal size. PyTorch recommends 25MB as the size of the bucket.



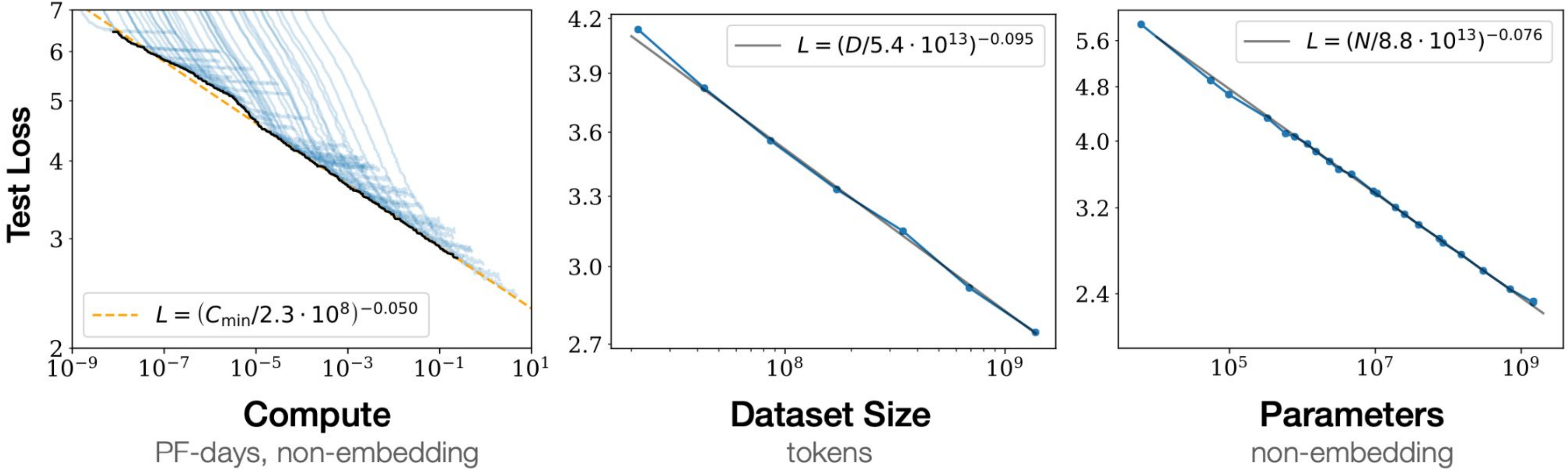
Parallelism Fundamentals

CS 229S Fall 2023

Today's Lecture

- Data parallelism
- Model Parallelism
- Pipeline Parallelism
- Practical challenges and solution for deploying parallelism in the real world

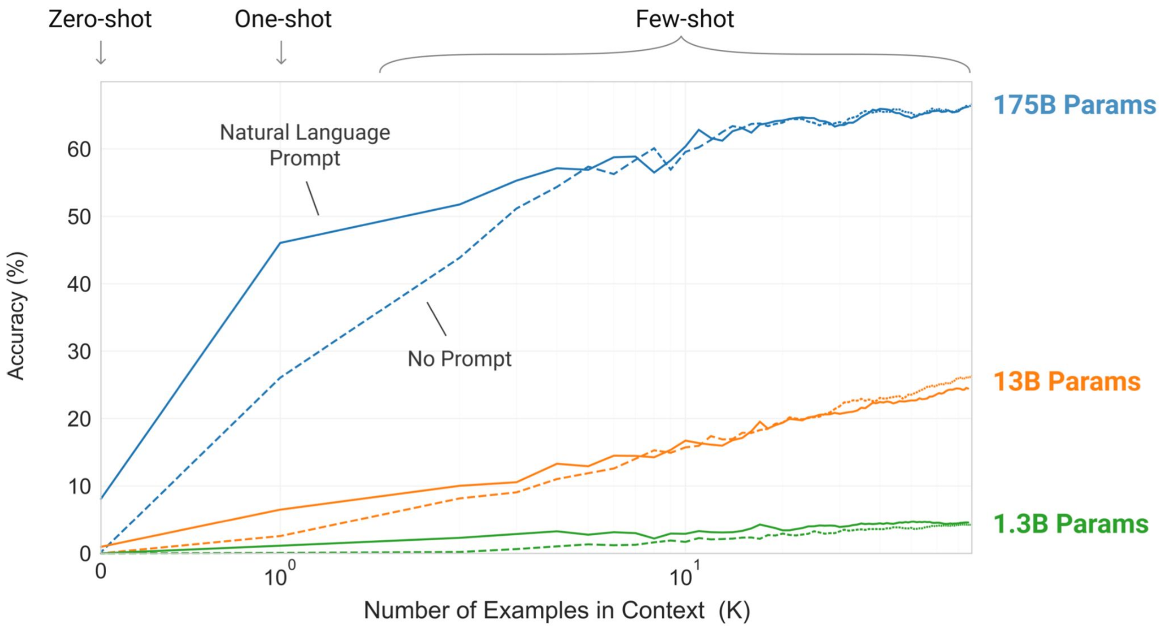
Model performance improves from all forms of scale



LLM performance improves from scaling up the total compute time, the dataset size, and the model size

Scaling Laws for Neural Language Models. Kaplan, et al. https://arxiv.org/pdf/2001.08361.pdf

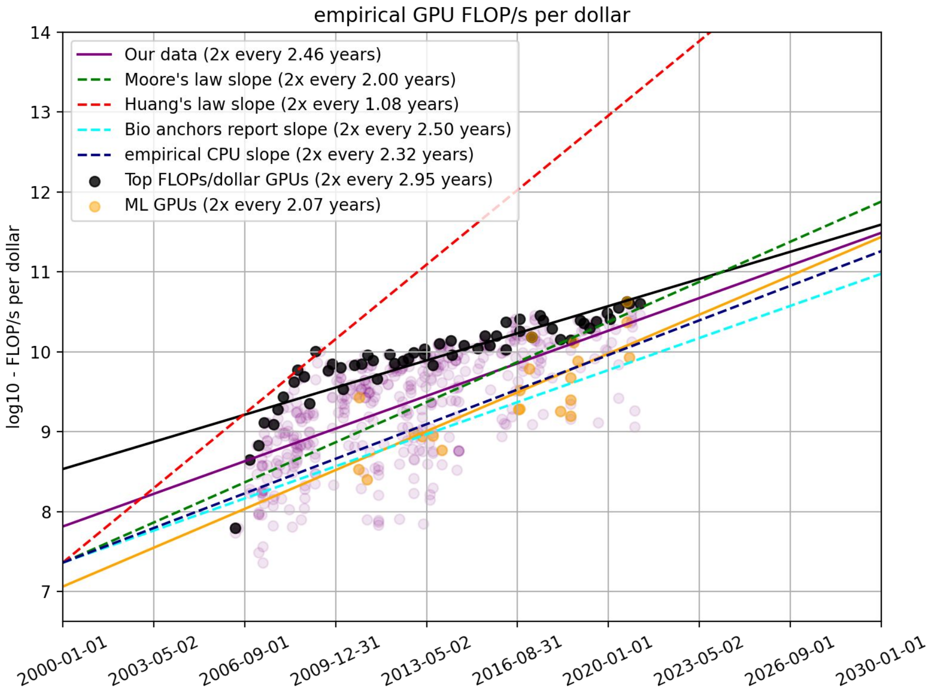
LLMs exhibit emergent properties *at scale*

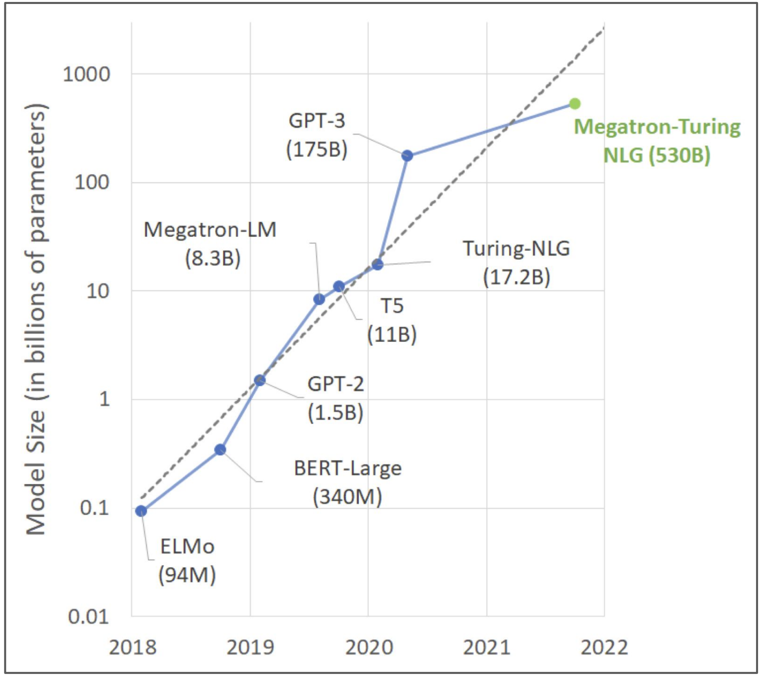


Scaling up to very large model and context sizes dramatically increases the capability of LLMs!

Language Models are Few-Shot Learners. Brown, et al. https://arxiv.org/pdf/2005.14165.pdf

Hardware improvements lag behind LLM scaling





GPU FLOPS performance doubles every ~2.5 years

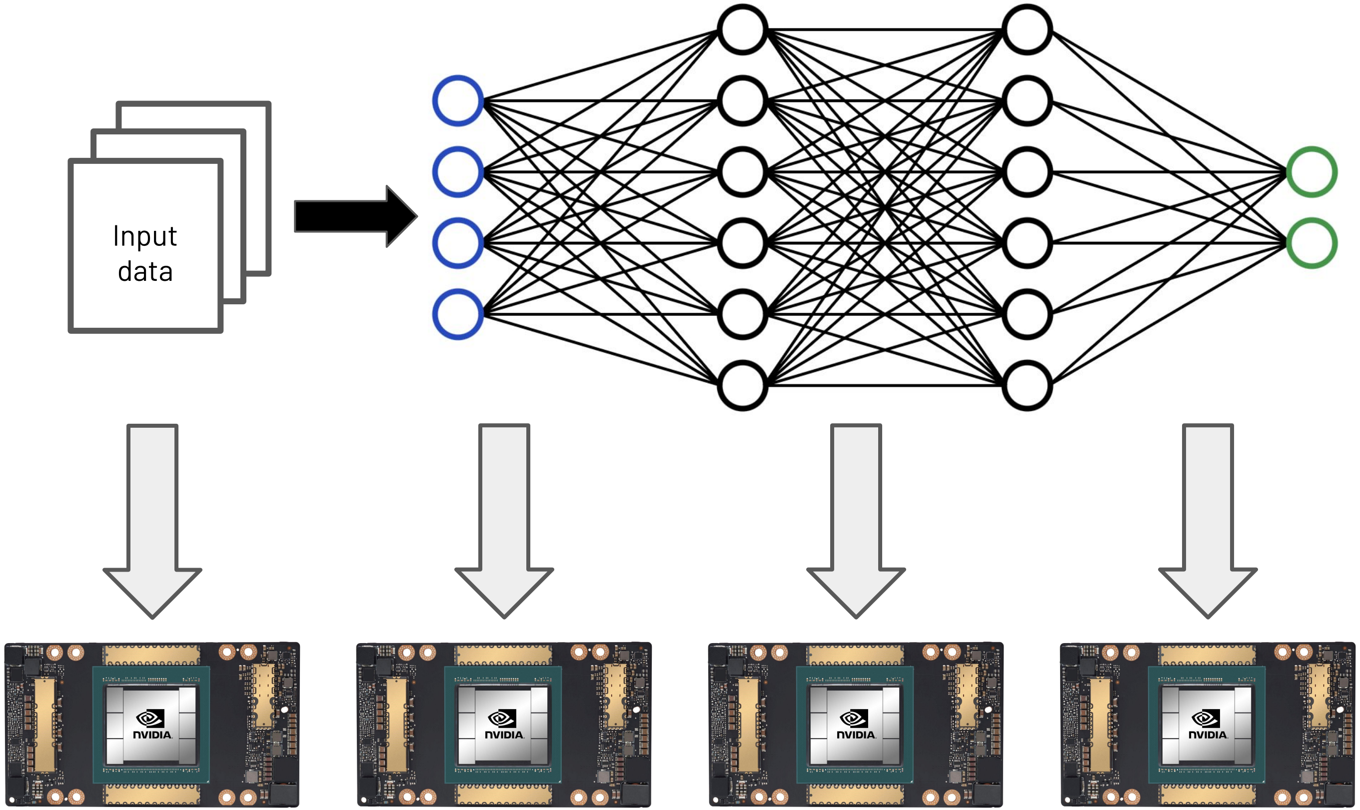
...but LLMs are getting ~**10x bigger every year**!

Trends in GPU Price-Performance. Hobbhahn and Besiroglu. https://epochai.org/blog/trends-in-gpu-price-performance
Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. Smith, et al. https://arxiv.org/pdf/2201.11990.pdf

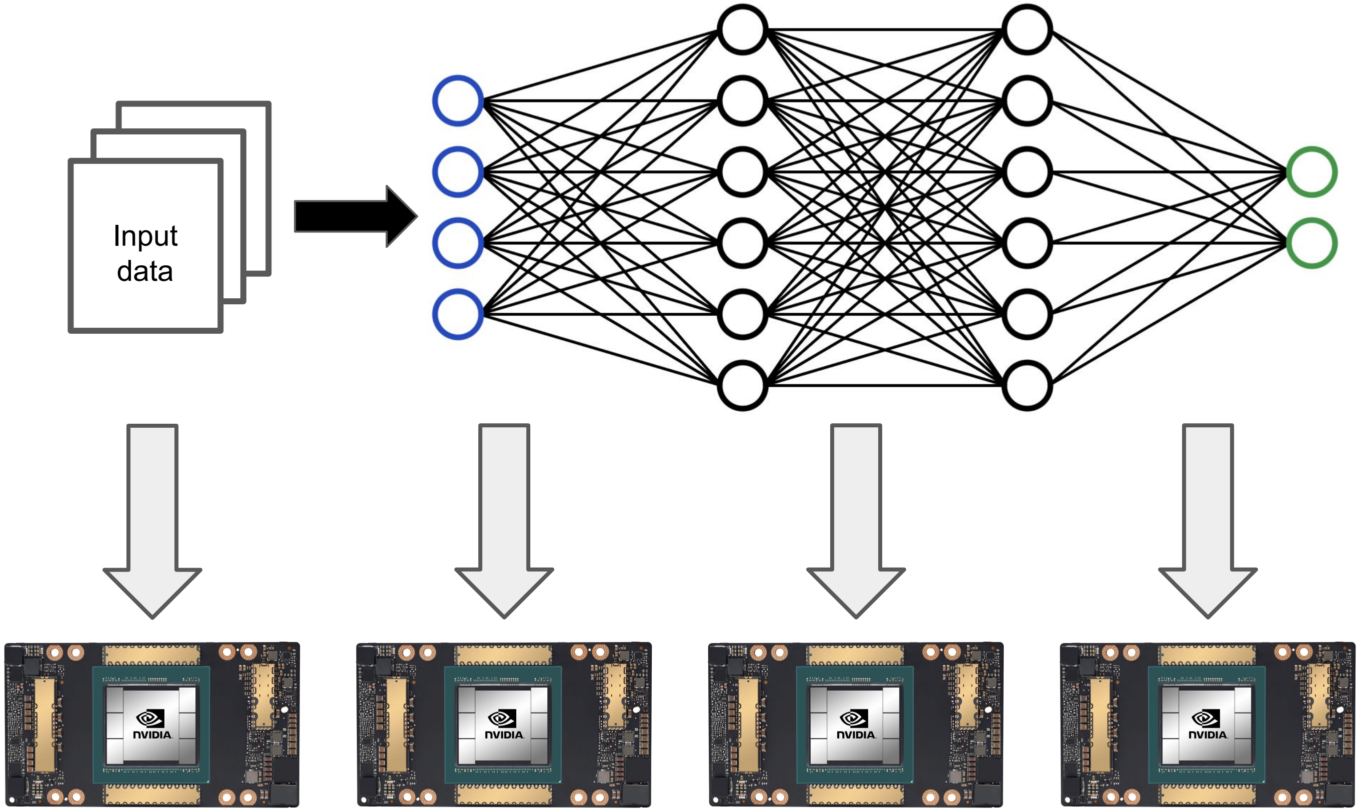
Parallelism is necessary to train and serve LLMs at scale

- We need to **parallelize** computation across multiple GPUs to handle the rapidly increasing scale of LLMs
- What are the key challenges when we start parallelizing LLMs?
  - **Minimizing communication overhead**: We want to minimize the time spent sending data between GPUs
  - **Minimizing synchronization overhead**: We want to minimize the dependencies between each GPU so that each GPU can proceed with its own computation independently
- Addressing these challenges are fundamental when it comes to designing parallel computing and distributed systems! (CS 149 and CS 244B for more detailed treatments)

How can we partition computation?

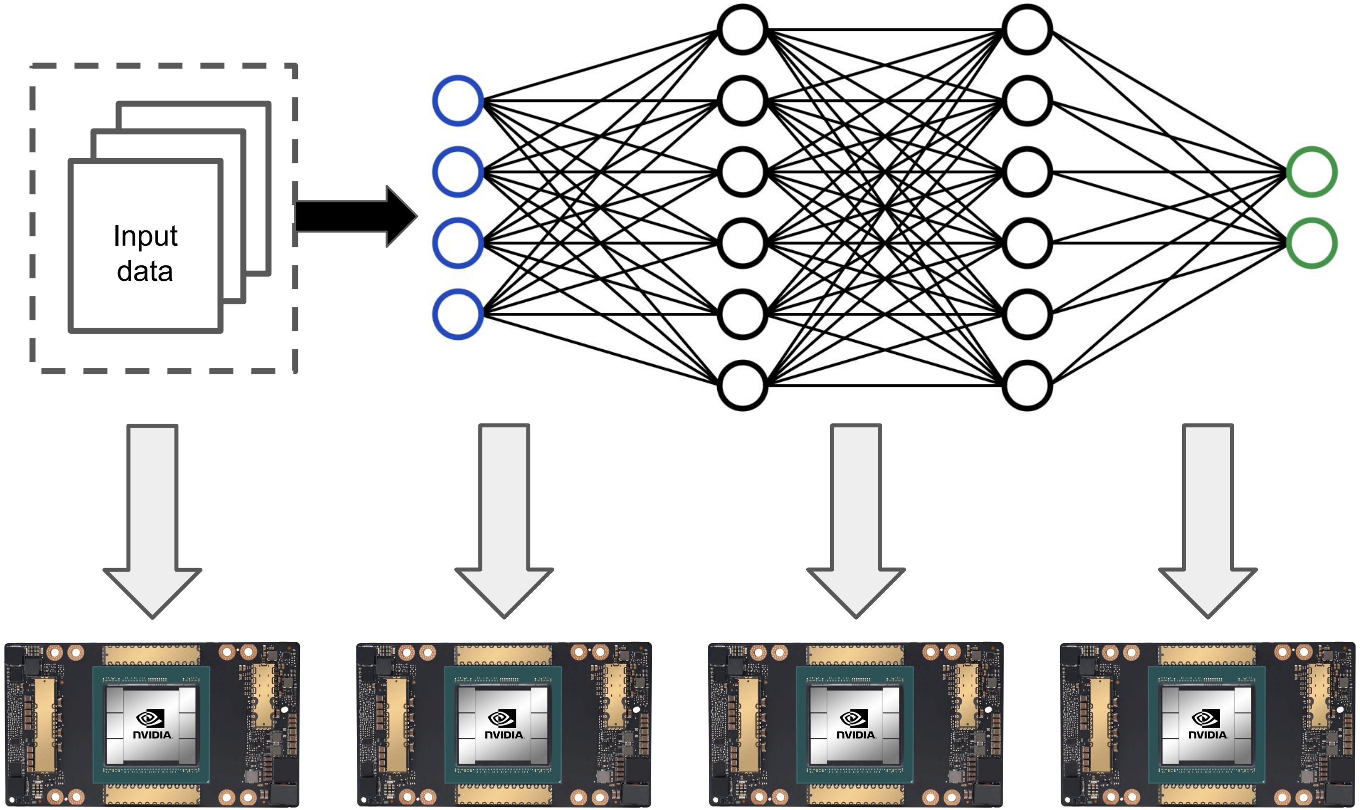


How can we partition computation?



We have two options for partitioning computation across devices

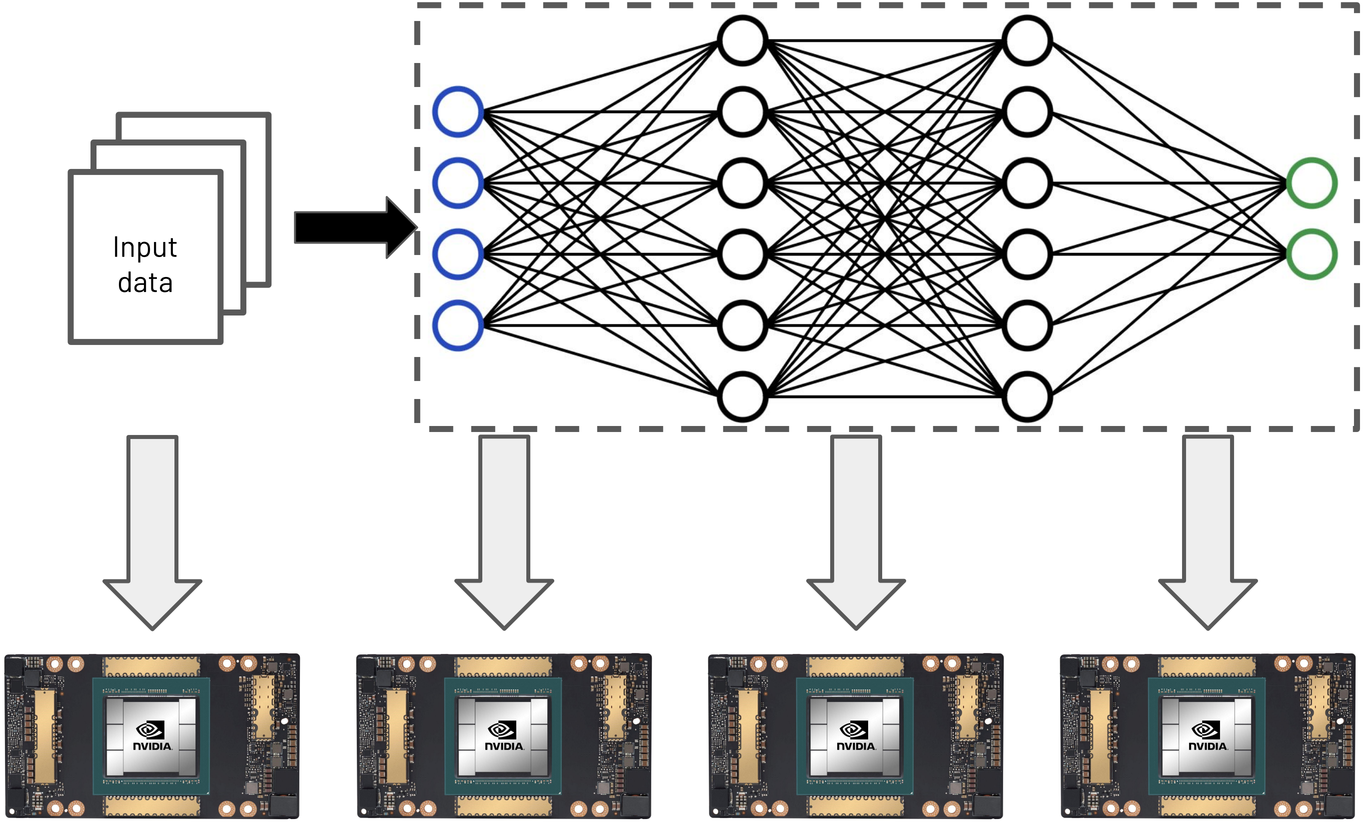
How can we partition computation?



We have two options for partitioning computation across devices

1. **Partition the input data**

How can we partition computation?



We have two options for partitioning computation across devices

1. Partition the input data
2. **Partition the model parameters**

What are the main techniques for parallelizing LLMs?

- There are three widely adopted techniques for parallelizing LLMs:

What are the main techniques for parallelizing LLMs?

- There are three widely adopted techniques for parallelizing LLMs:
  - **Data parallelism**: Partition the input data, and replicate the model weights

What are the main techniques for parallelizing LLMs?

- There are three widely adopted techniques for parallelizing LLMs:
    - **Data parallelism**: Partition the input data, and replicate the model weights
    - **Model parallelism**: Partition the model weights and replicate the input data

What are the main techniques for parallelizing LLMs?

- There are three widely adopted techniques for parallelizing LLMs:
    - **Data parallelism**: Partition the input data, and replicate the model weights
    - **Model parallelism**: Partition the model weights and replicate the input data
    - **Pipeline parallelism**: Partition the model weights across space and partition the input data across time

What are the main techniques for parallelizing LLMs?

- There are three widely adopted techniques for parallelizing LLMs:
  - **Data parallelism**: Partition the input data, and replicate the model weights
  - **Model parallelism**: Partition the model weights and replicate the input data
  - **Pipeline parallelism**: Partition the model weights across space and partition the input data across time
- Each of these techniques has tradeoffs which we will discuss

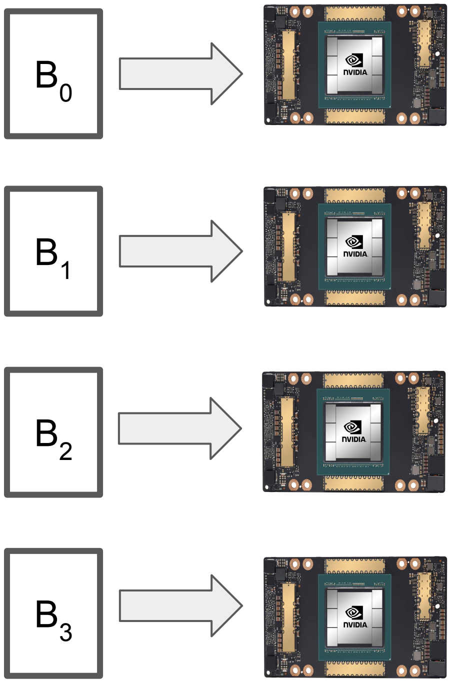
What are the main techniques for parallelizing LLMs?

- There are three widely adopted techniques for parallelizing LLMs:
  - **Data parallelism**: Partition the input data, and replicate the model weights
  - **Model parallelism**: Partition the model weights and replicate the input data
  - **Pipeline parallelism**: Partition the model weights across space and partition the input data across time
- Each of these techniques has tradeoffs which we will discuss
- These techniques are not mutually exclusive - they can be combined to build extremely efficient systems!

What are the main techniques for parallelizing LLMs?

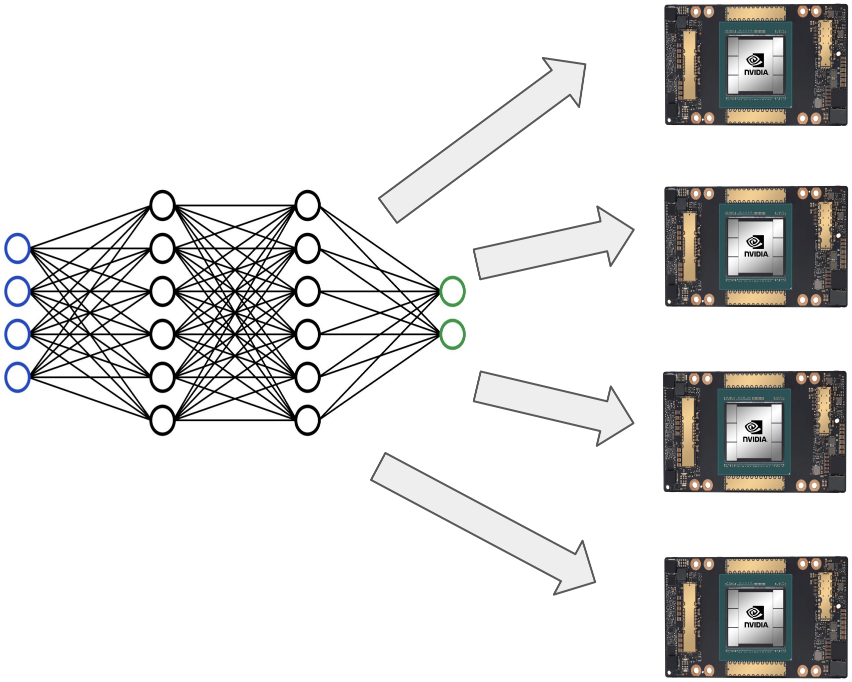
- There are three widely adopted techniques for parallelizing LLMs:
    - **Data parallelism**: Partition the input data, and replicate the model weights
    - **Model parallelism**: Partition the model weights and replicate the input data
    - **Pipeline parallelism**: Partition the model weights across space and partition the input data across time
- Each of these techniques has tradeoffs which we will discuss
- These techniques are not mutually exclusive - they can be combined to build extremely efficient systems!
- These techniques are not exhaustive - this is an active area of research!

Data parallelism



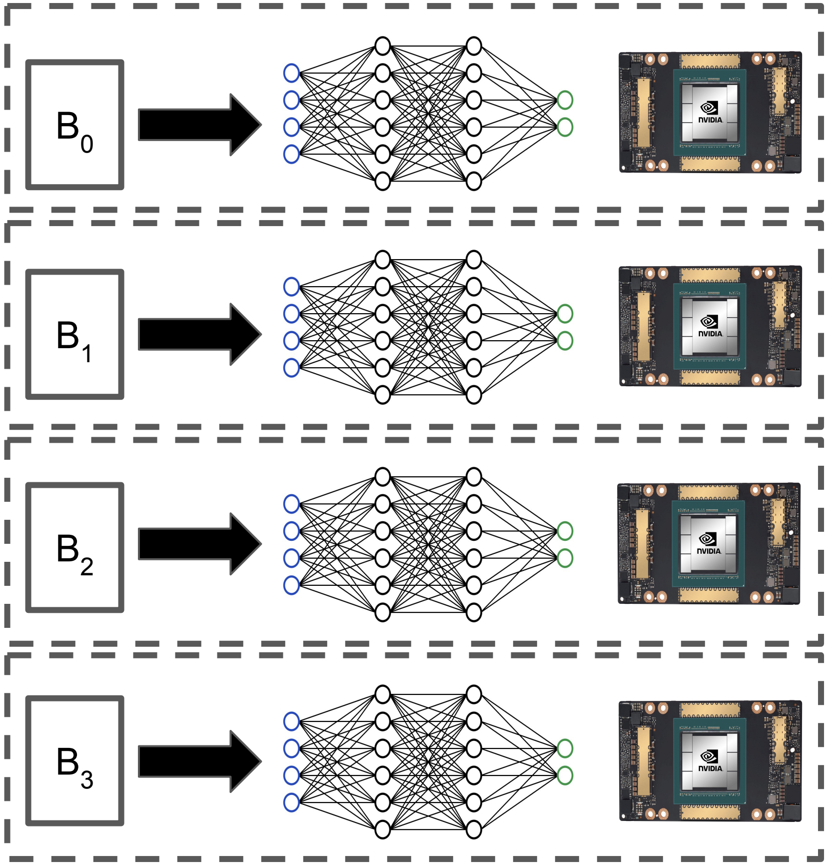
- We partition the input data on the batch dimension across the different GPUs
- For example, if we had 4 GPUs and a batch size of 64, then each GPU would receive an input sub-batch of size 16

Data parallelism



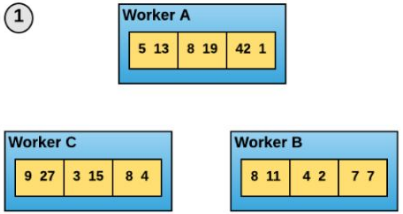
- We partition the input data on the batch dimension across the different GPUs
- For example, if we had 4 GPUs and a batch size of 64, then each GPU would receive an input sub-batch of size 16
- We replicate the model weights across the GPUs, meaning every GPU has a complete copy of all the weights

Data parallelism



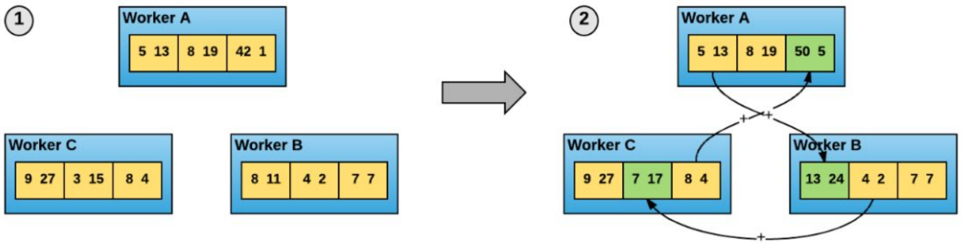
- Forward and backward pass computation on each GPU proceeds independently
- Each GPU produces its own set of gradients for its own sub-batch
- At the end of the backward pass, the gradients are aggregated across all GPUs and averaged
- The GPUs share their gradients via an *all-reduce* operation

All-Reduce Algorithm Iteratively Shares Data Across GPUs



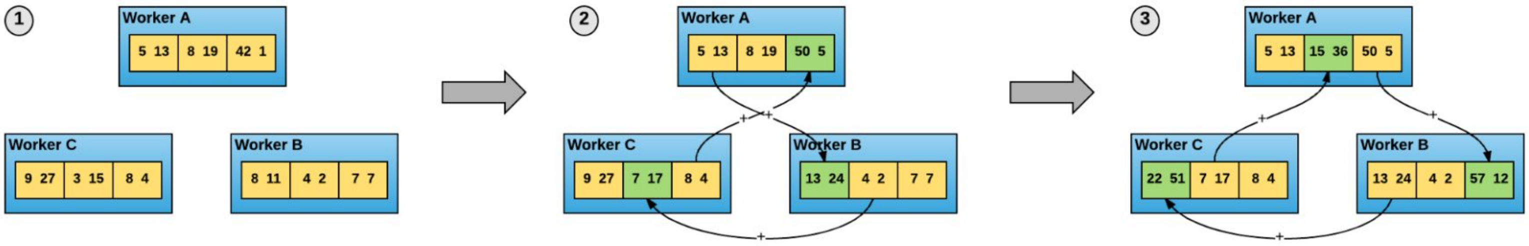
Horovod: fast and easy distributed deep learning in TensorFlow. Sergeev and Balso. https://arxiv.org/pdf/1802.05799.pdf

All-Reduce Algorithm Iteratively Shares Data Across GPUs



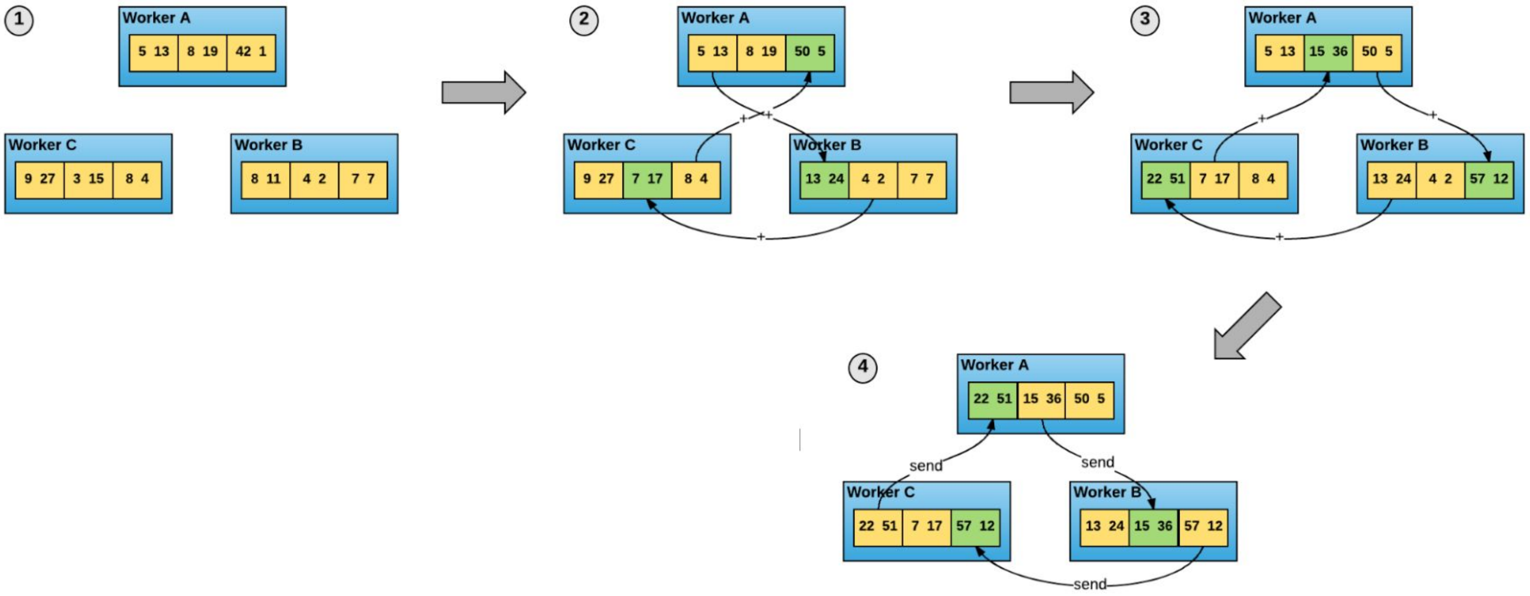
Horovod: fast and easy distributed deep learning in TensorFlow. Sergeev and Balso. https://arxiv.org/pdf/1802.05799.pdf

All-Reduce Algorithm Iteratively Shares Data Across GPUs



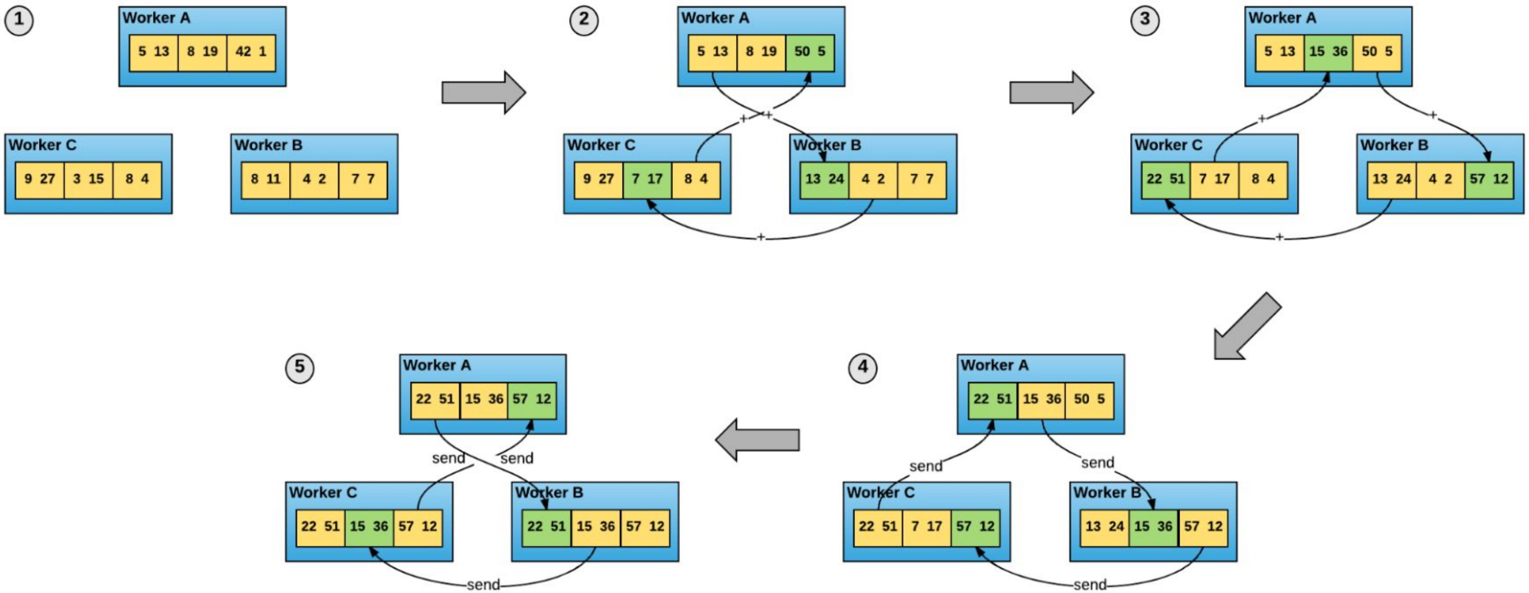
Horovod: fast and easy distributed deep learning in TensorFlow. Sergeev and Balso. https://arxiv.org/pdf/1802.05799.pdf

All-Reduce Algorithm Iteratively Shares Data Across GPUs



Horovod: fast and easy distributed deep learning in TensorFlow. Sergeev and Balso. https://arxiv.org/pdf/1802.05799.pdf

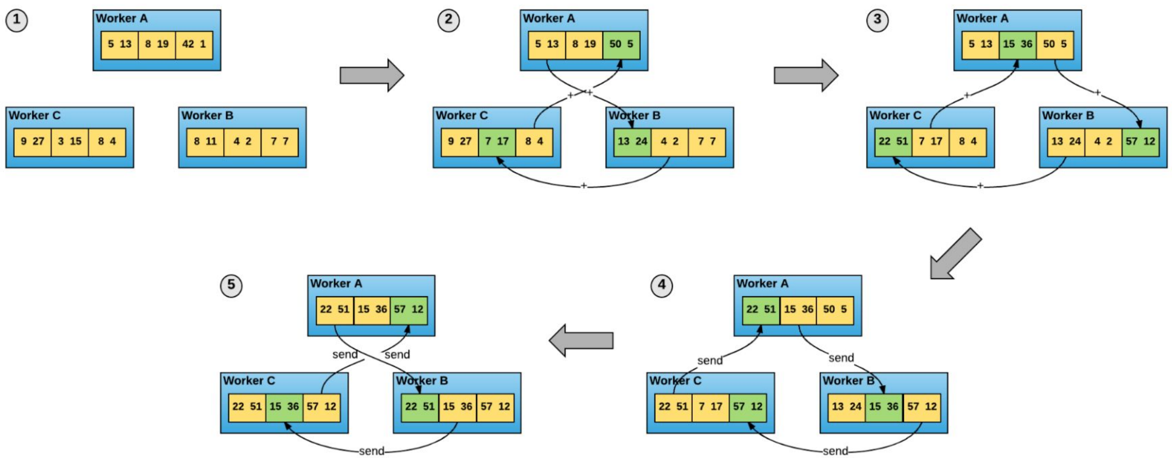
All-Reduce Algorithm Iteratively Shares Data Across GPUs



Horovod: fast and easy distributed deep learning in TensorFlow. Sergeev and Balso. https://arxiv.org/pdf/1802.05799.pdf

All-reduce Algorithm

- Ring all-reduce proceeds in 2 * (N-1) iterations for a group of N GPUs
- In each iteration, each node sends fragments of its data to 2 of its peers
  - First N-1 iterations: each GPU adds received values to its own data
  - Next N-1 iterations: each GPU replaces its own data with received values



Ring All-reduce Efficiency Analysis

- Total volume of data sent (N is the total number of GPUs):
    - Assume each GPU has data of size X bytes
    - Then each node sends (2 * (N-1)) * X/N bytes

Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. Patarasuk and Yuan. https://www.cs.fsu.edu/~xyuan/paper/09jpdc.pdf

Ring All-reduce Efficiency Analysis

- Total volume of data sent (N is the total number of GPUs):
  - Assume each GPU has data of size X bytes
  - Then each node sends 2 * (N-1) * X/N bytes
- Total time to complete all-reduce operation:
  - Assume network bandwidth of B bytes / second
  - Then the communication runtime is 2 * (N-1) * X /(N*B) seconds

Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. Patarasuk and Yuan. https://www.cs.fsu.edu/~xyuan/paper/09jpdc.pdf

Data Parallelism

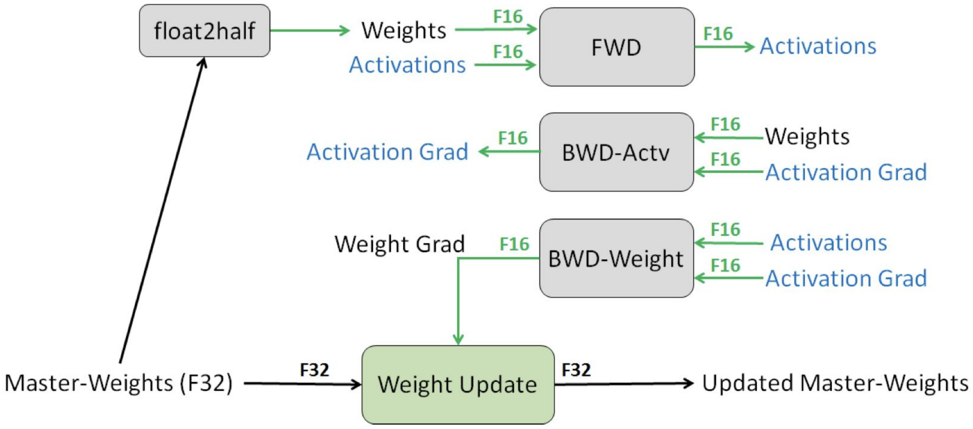
- What are the pros of data parallelism?
    - Most direct approach for increasing throughput since global batch size increases with each additional GPU
    - Relatively easy to implement (e.g. built-in support from PyTorch)

Data Parallelism

- What are the pros of data parallelism?
  - Most direct approach for increasing throughput since global batch size increases with each additional GPU
  - Relatively easy to implement (e.g. built-in support from PyTorch)
- What are the cons of data parallelism?
  - Requires the model weights and activations to fit in GPU memory (e.g. impossible to run GPT-3 with pure data parallelism on an 80 GB GPU)
  - Communication-intensive - every weight must be synchronized across all GPUs

Data Parallelism

- The main limitation of data parallelism is the high memory requirement
- If we train model with P parameters using mixed-precision training:
    - fp16 model parameters: 2P bytes
    - fp16 gradients: 2P bytes
    - Adam optimizer states
        - fp32 model parameters: 4P bytes
        - fp32 momentum: 4P bytes
        - fp32 variance: 4P bytes
    - Total: 2P + 2P + 4P + 4P + 4P = 16P bytes just for parameters and optimizer states

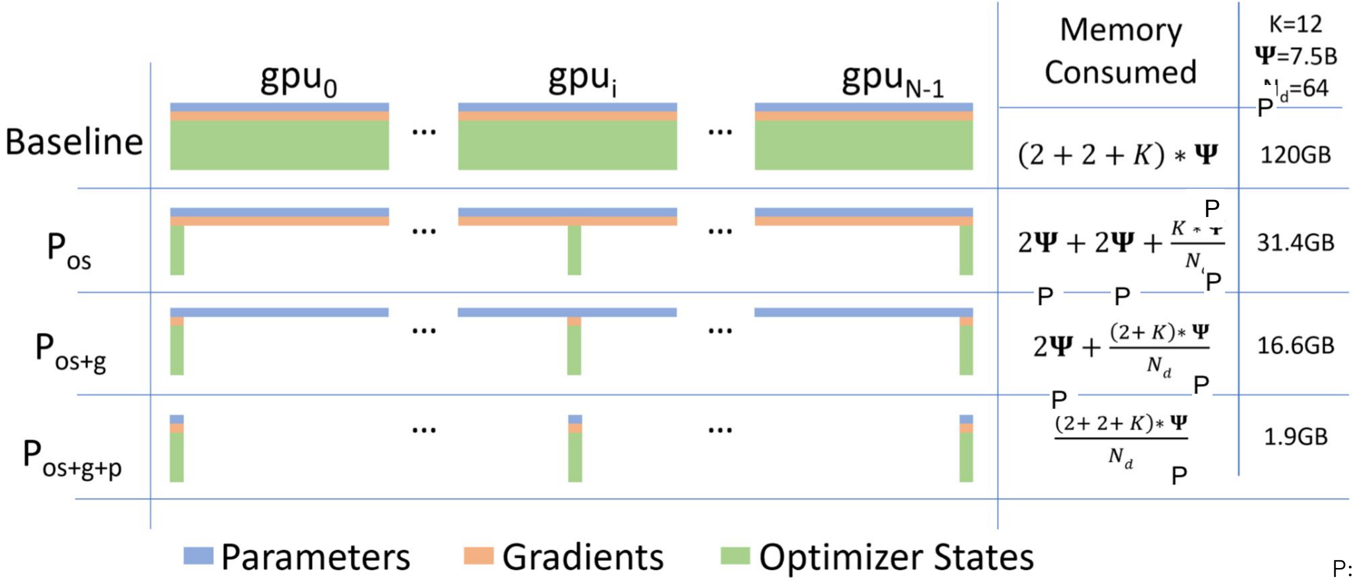


Mixed precision training

Mixed Precision Training. Narang, et al. https://arxiv.org/pdf/1710.03740.pdf
ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. Rajbhandari et al. https://arxiv.org/pdf/1910.02054.pdf

Data Parallelism

- We can significantly reduce memory usage by using the ZeRO techniques (Zero-Redundancy Optimizer)



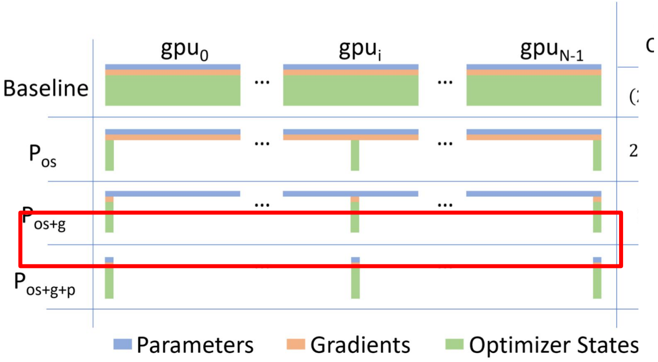
P: number of parameters

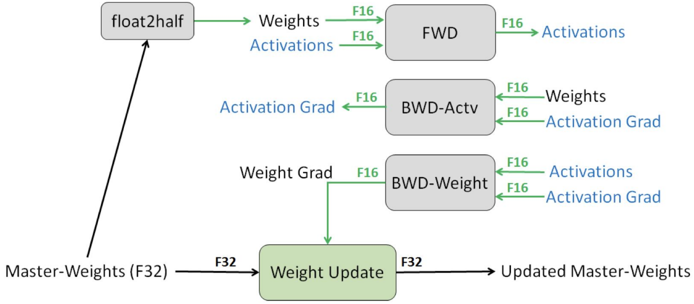
K: storage multiplier for the optimizer state

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. Rajbhandari et al. https://arxiv.org/pdf/1910.02054.pdf

ZeRO

- Stage 1: **Optimizer state partitioning**
    - Group the optimizer states into N_d equal partitions, such that the *i*-th (data parallel) process only updates the state for the *i*-th partition





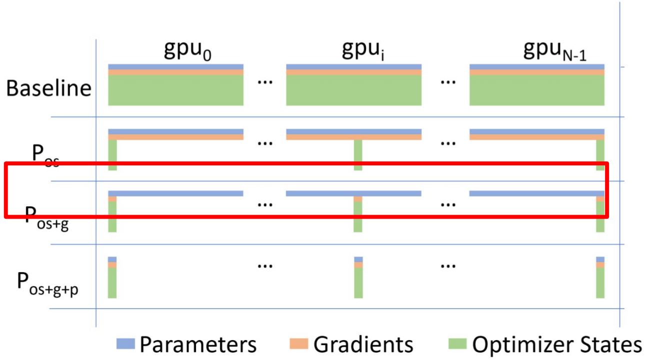
Mixed precision training

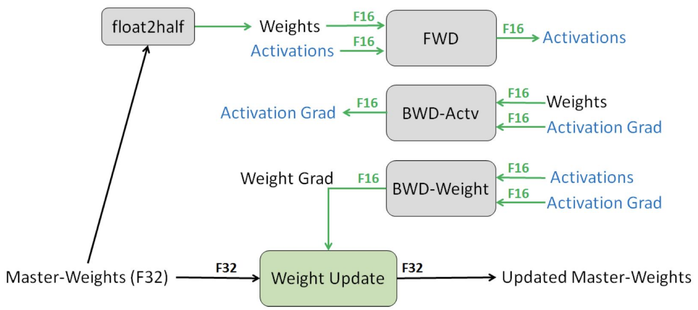
ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. Rajbhandari et al. https://arxiv.org/pdf/1910.02054.pdf

ZeRO

- Stage 1: **Optimizer state partitioning**
  - Group the optimizer states into N_d equal partitions, such that the *i*-th (data parallel) process only updates the state for the *i*-th partition
  - Each process performs an all-gather across the N_d processes at the end of each training step to get the fully updated parameters

ZeRO Stage c	**Memory Usage (bytes)**
Baseline	16P
Stage 1: P_{os}	$4P + 12P / N_d$





Mixed precision training

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. Rajbhandari et al. https://arxiv.org/pdf/1910.02054.pdf

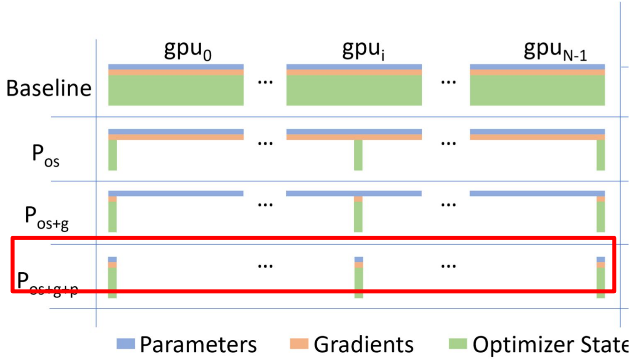
ZeRO

- Stage 2: **Optimizer state + gradient partitioning**
  - As each data parallel process only updates its own parameter partition, it only needs the reduced gradients for the corresponding parameters
  - This reduces the memory footprint required to hold the gradients on each data parallel process from 2P bytes to $2P / N_d$
  - Memory savings: By removing both gradient and optimizer state redundancy, we reduce the memory footprint further down to:

$$2P + 14P / N_d \approx 2P$$

Weights in fp16 — Gradients in fp16 and optimizer state

ZeRO Stage c	**Memory Usage (bytes)**
Baseline	16P
Stage 1: P_{os}	$4P + 12P / N_d$
Stage 2: P_{os+g}	$2P + 14 P / N_d$

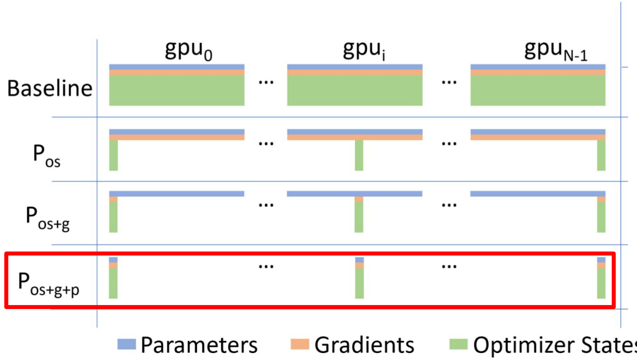


ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. Rajbhandari et al. https://arxiv.org/pdf/1910.02054.pdf

ZeRO

- Stage 3: **Optimizer state + gradient + parameter partitioning**
  - Instead of storing all parameters on all data parallel processes, we can just fetch (i.e. gather) the parameters we need to compute the forward and backward pass
  - This approach increases the total communication volume by 1.5x*, but reduces memory proportional to N_d
  - Memory savings: With parameter partitioning, we reduce the memory consumption 16P to 16P / N_d

ZeRO Stage	Memory Usage (bytes)
Baseline	16P
Stage 1: P_{os}	$4P + 12P / N_d$
Stage 2: P_{os+g}	$2P + 14P / N_d$
Stage 3: P_{os+g+p}	$16P / N_d$



*See paper for more details.

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. Rajbhandari et al. https://arxiv.org/pdf/1910.02054.pdf

PyTorch FSDP

- The PyTorch FSDP (Fully Sharded Data Parallel) API implements data parallelism with ZeRO optimizations
- Example usage:

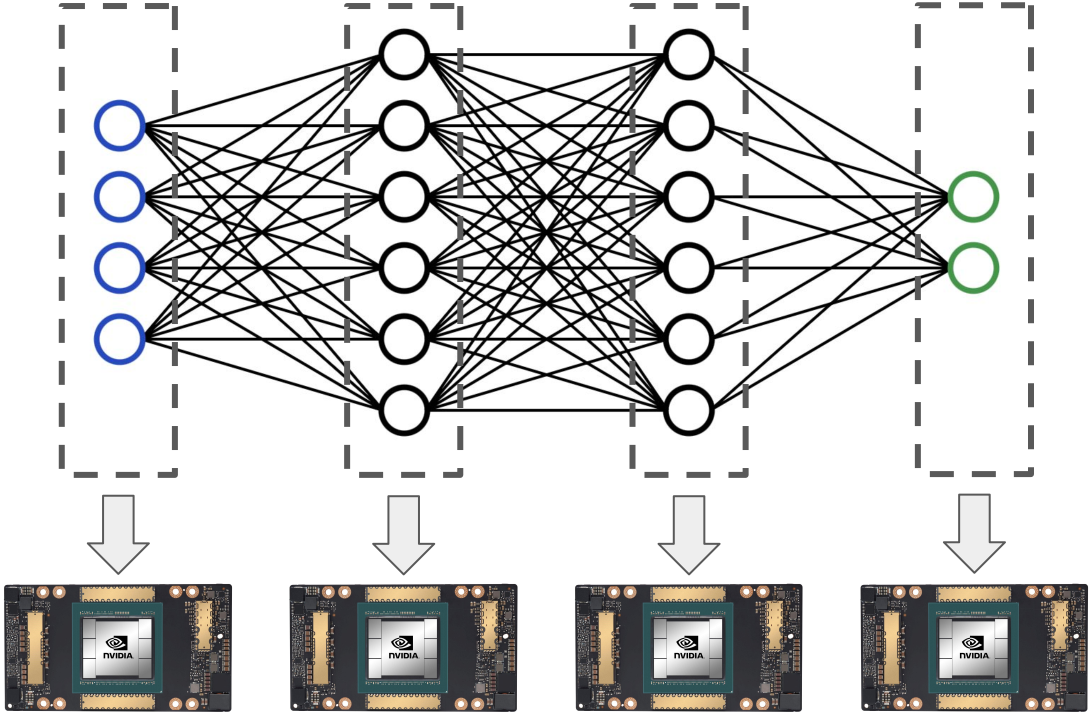
```
import torch
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP

torch.cuda.set_device(device_id)

module = ... # Module definition

sharded_module = FSDP(module)
optim = torch.optim.Adam(sharded_module.parameters(), lr=0.0001)

x = ... # Input data
y = sharded_module(x)

loss = y.sum()
loss.backward()

optim.step()
```

FullyShardedDataParallel. https://pytorch.org/docs/stable/fsdp.html

Model Parallelism

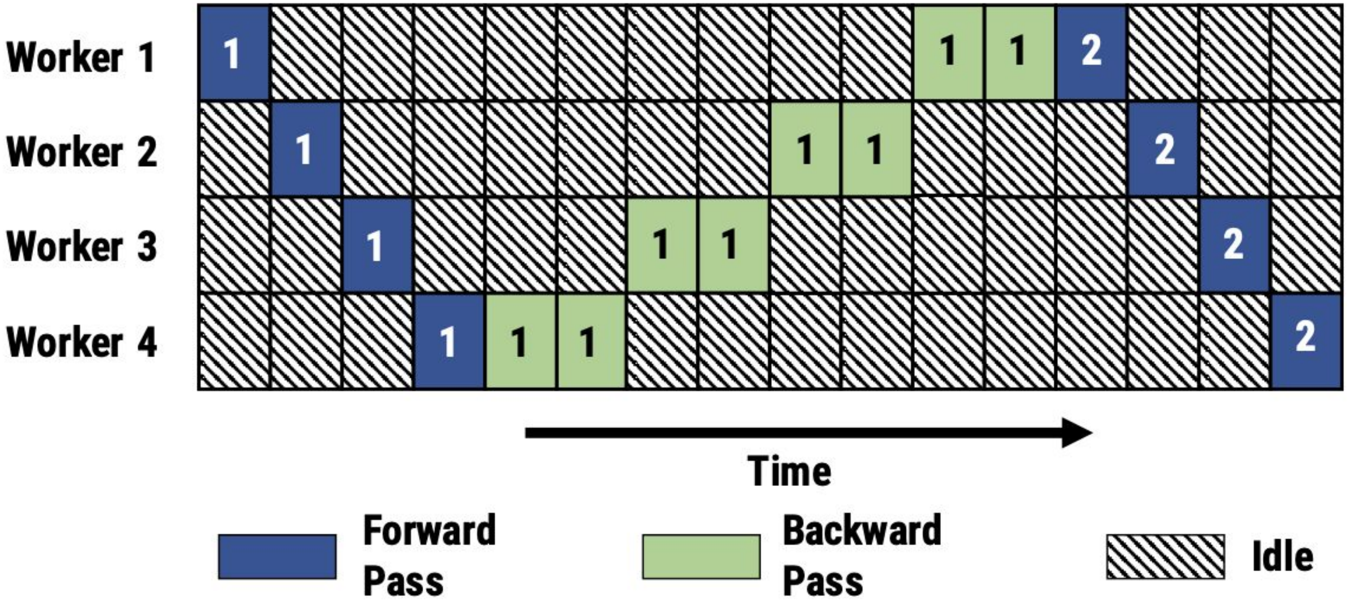
- Another way to reduce memory usage is *model parallelism*: replicate the input data, but partition the model weights
- Two general approaches for model parallelism:
    - Slice the model “vertically”: place subsets of layers on different GPUs
    - Slice the model “horizontally”: shard the model weights across different GPUs
- We will discuss the pros and cons of each these approaches

Model Parallelism



Vertically slicing the model gives each GPU its own subset of layers

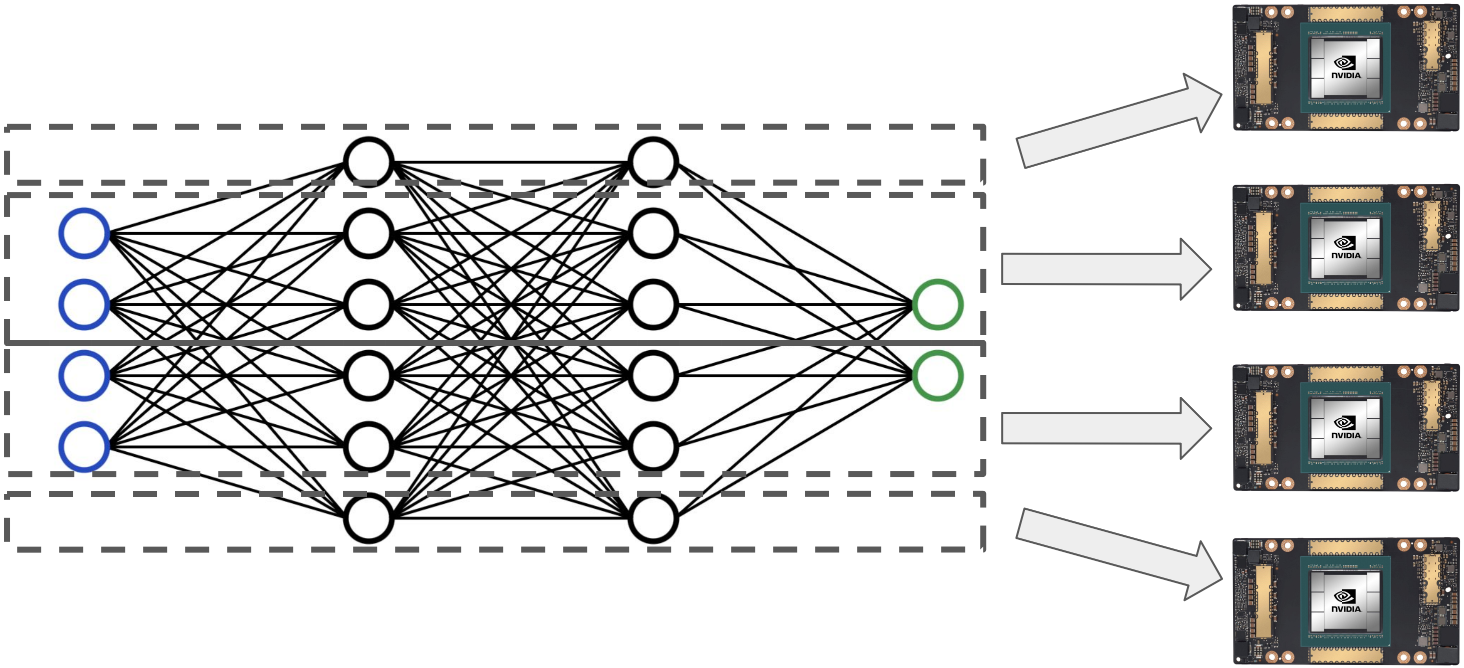
Model Parallelism



Vertical slicing severely lowers hardware utilization because the devices are frequently idle

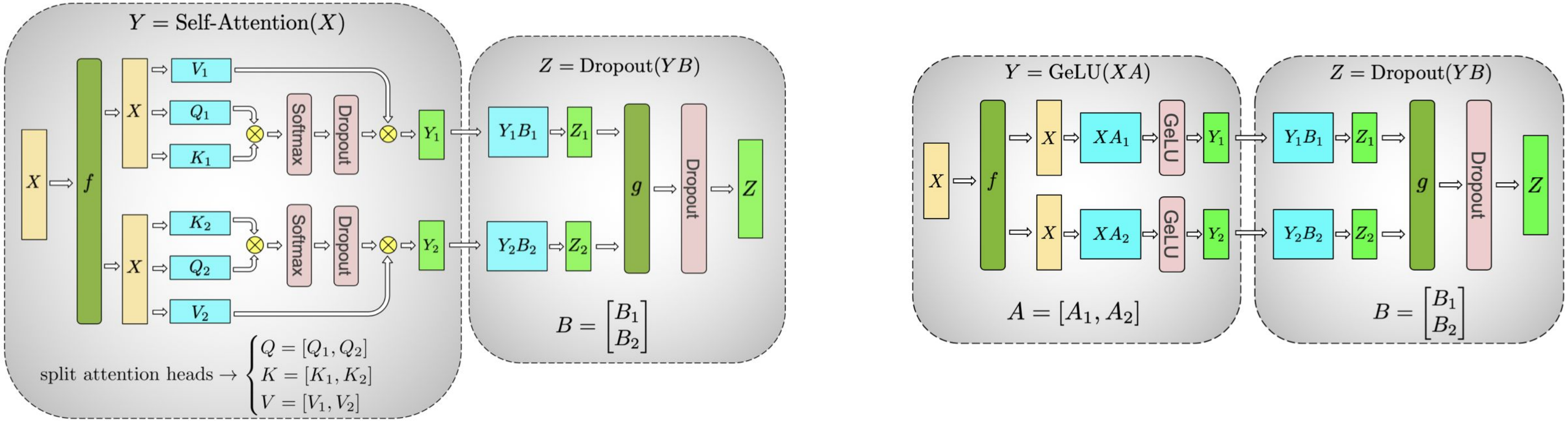
PipeDream: Generalized Pipeline Parallelism for DNN Training. Narayanan et al. https://deepakn94.github.io/assets/papers/pipedream-sosp19.pdf

Model Parallelism



Horizontally slicing the model shards the layers across the devices

Tensor Model Parallelism for LLMs



- For Transformer-based LLMs specifically, we can shard the self-attention and subsequent MLP weights (Megatron-style or tensor model parallelism)
- This requires adding an all-reduce after every attention and MLP computation to synchronize the weights

Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. Shoeybi et al. https://arxiv.org/pdf/1909.08053.pdf

Tensor Model Parallelism

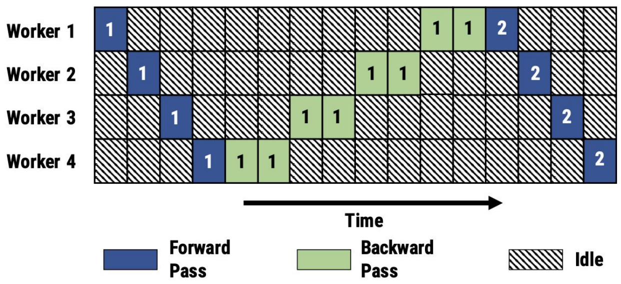
- What are the pros of tensor model parallelism (in particular Megatron)?
    - Reduces the amount of memory required per GPU
    - Keeps GPU utilization high compared to vertical slicing

Tensor Model Parallelism

- What are the pros of tensor model parallelism (in particular Megatron)?
  - Reduces the amount of memory required per GPU
  - Keeps GPU utilization high compared to vertical slicing
- What are the cons of tensor model parallelism (in particular Megatron)?
  - Very frequent synchronization (all-reduces) means we need extremely fast network connections to maintain high throughput
  - Not as easy to implement as data parallelism – need to add the synchronization ops manually to your Attention module

Tensor Model Parallelism

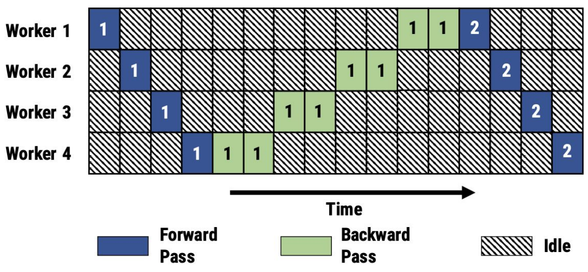
- Model parallelism solved our memory usage problem, but it requires very fast networking hardware due to the frequent all-reduces
- What do we do if we don't have high-memory GPUs or high-bandwidth network interconnect?
- Let's revisit the vertical slicing we saw earlier
  - This approach also reduces per-GPU memory usage
  - No all-reduces necessary - just send activations or gradients from one GPU to the next
  - Downside is poor GPU utilization - can we fix this?



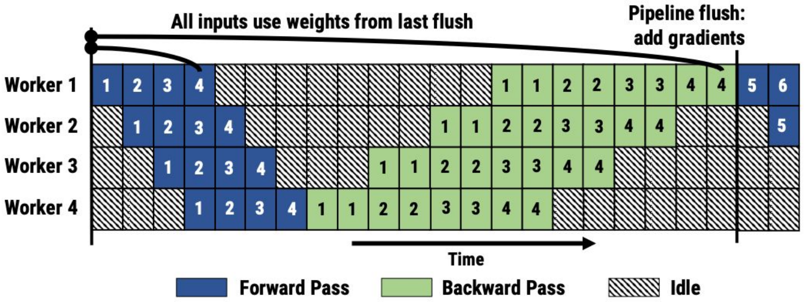
PipeDream: Generalized Pipeline Parallelism for DNN Training. Narayanan et al. https://deepakn94.github.io/assets/papers/pipedream-sosp19.pdf

Pipeline Parallelism

- We can apply the well-known technique of *pipelining* to improve GPU utilization
- Pipeline parallelism splits each batch into *microbatches* and injects multiple microbatches into the pipeline
- This can significantly reduce the amount of idle time on each GPU



Without pipelined computation

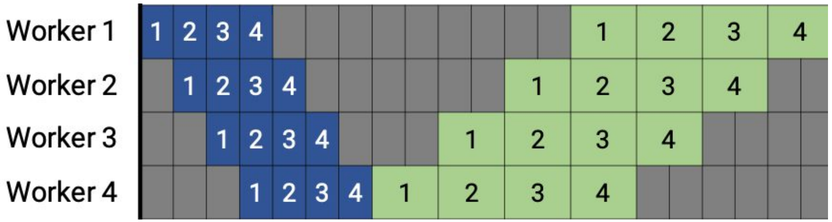


With pipelined computation

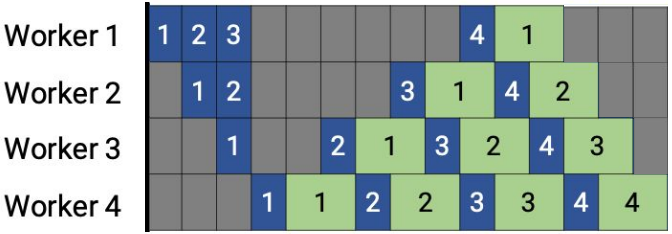
PipeDream: Generalized Pipeline Parallelism for DNN Training. Narayanan et al. https://deepakn94.github.io/assets/papers/pipedream-sosp19.pdf

Pipeline Parallelism

- Pipeline parallelism introduces two new considerations:
    - How to set the microbatch size
        - Larger microbatches = higher arithmetic intensity, smaller microbatches = smaller pipeline bubbles
    - How to decide the schedule of pipelined computation
        - We can dynamically choose which microbatches or layers to execute at each step; this will impact the pipeline bubble size



GPipe schedule

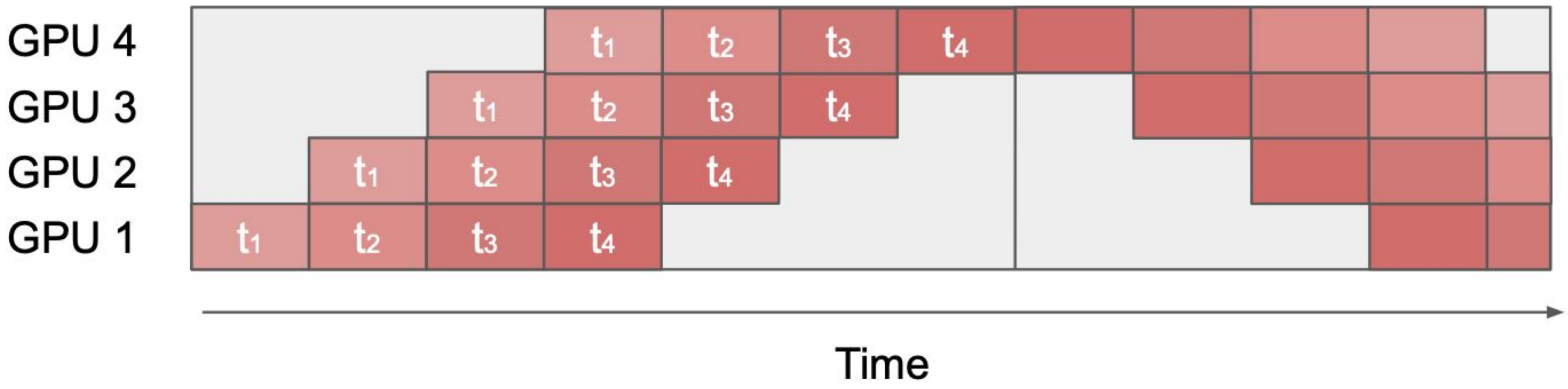


1F1B schedule

Memory-Efficient Pipeline Parallel DNN Training. Narayanan et al. https://arxiv.org/pdf/2006.09503.pdf

Pipeline Parallelism for LLMs

- We can extend pipeline parallelism to LLMs by pipelining specifically on the sequence dimension
- In particular, we can split the input sequence into subsequences and have each stage process the tokens for each subsequence incrementally



TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. Li et al. https://arxiv.org/pdf/2102.07988.pdf

Pipeline Parallelism

- What are the pros of pipeline parallelism?
  - Reduces the amount of memory required per GPU
  - Minimizes communication across GPUs (only point-to-point send operations instead of all-reduces)

Pipeline Parallelism

- What are the pros of pipeline parallelism?
  - Reduces the amount of memory required per GPU
  - Minimizes communication across GPUs (only point-to-point send operations instead of all-reduces)
- What are the cons of pipeline parallelism?
  - Still suffers from low GPU utilization due to pipeline bubbles
  - Difficult to implement because it requires scheduling the different microbatches to be executed concurrently

Summary of Different Parallelism Approaches

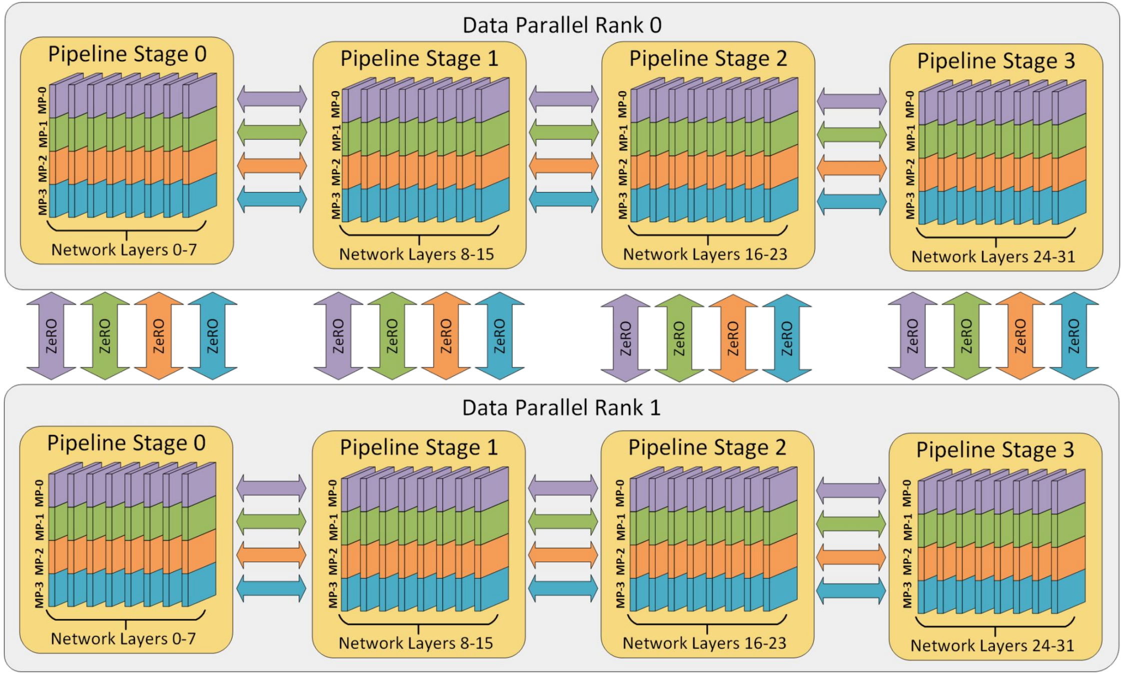
- In summary:
    - Data parallelism is effective if the model weights and activations fit into GPU memory
    - Tensor model parallelism is effective if the model weights and activations do not fit into GPU memory but we have a single server with very fast networking hardware
    - Pipeline parallelism is effective if the model weights and activations do not fit into GPU memory and we have multiple servers or a single server without fast networking hardware
- ...but we do not have to choose just one parallelism strategy!

End of Lecture 8

Appendix

3D / Pipeline-Tensor-Data (PTD) Parallelism

- We can use data parallelism, tensor model parallelism, and pipeline parallelism together (3D or PTD parallelism) to scale to thousands of GPUs
- This will be the focus of the Lecture 10



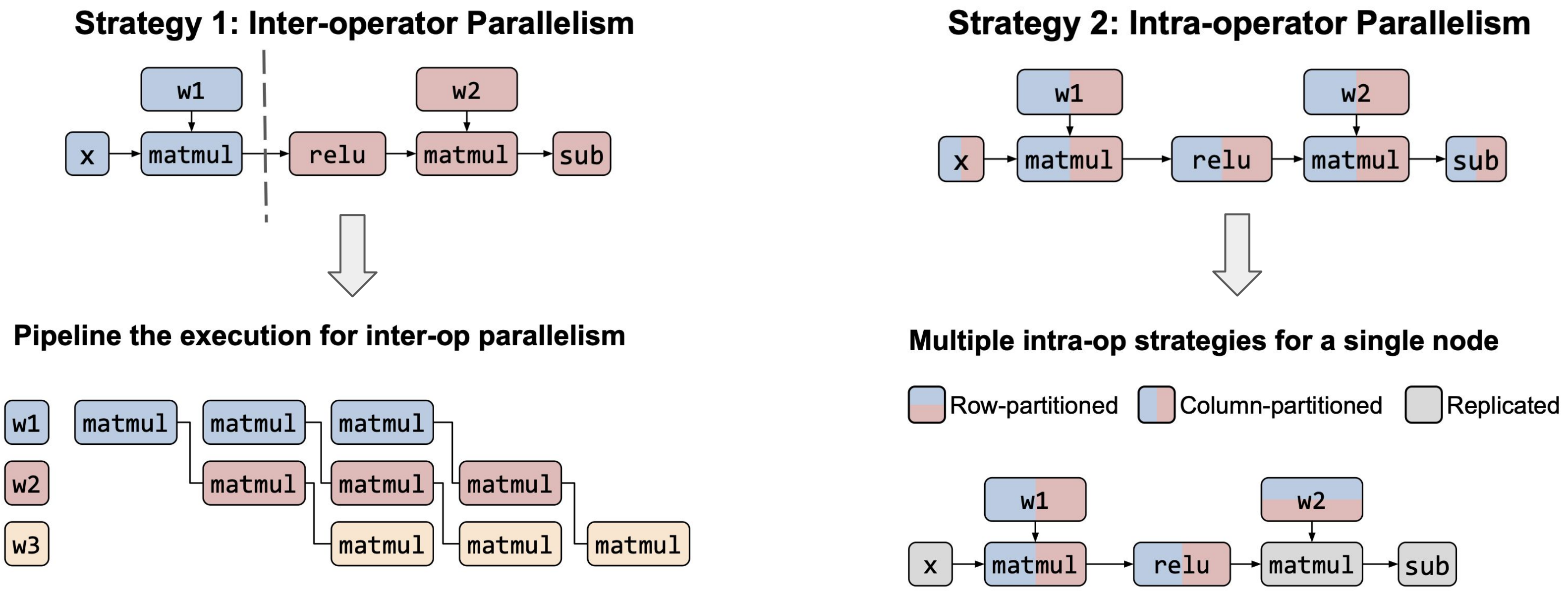
DeepSpeed: Extreme-scale model training for everyone. DeepSpeed Team, Majumder, and Wang. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/
Efficient Large-Scale Language Model Training on GPU Clusters using Megatron-LM. Narayanan, et al. https://dl.acm.org/doi/pdf/10.1145/3458817.3476209

Practical Challenges When Deploying Parallelism

- Practical challenges when trying to deploy parallelism:
  - The space of possible strategies grows combinatorially large
  - Experimenting with different parallelization strategies is slow / expensive at the scale of hundreds or thousands of GPUs
- Can we *automate* parallelization for a given model and cluster?

Automatic Parallelization with Alpa

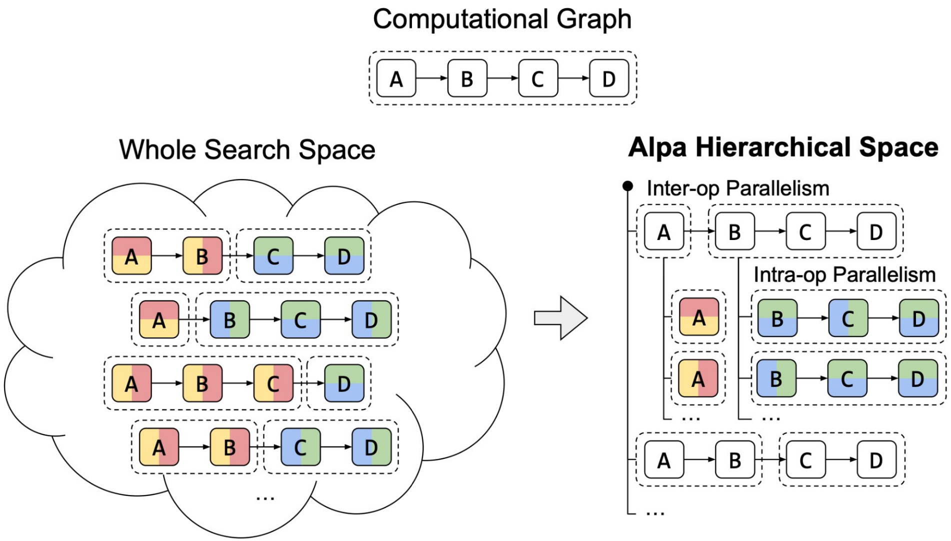
- Alpa is a system for automatically selecting and executing the optimal parallelism strategy for a given model and cluster
- Alpa separately considers *inter-operator* parallelism (i.e. pipeline parallelism) and *intra-operator* parallelism (i.e. data and tensor model parallelism)

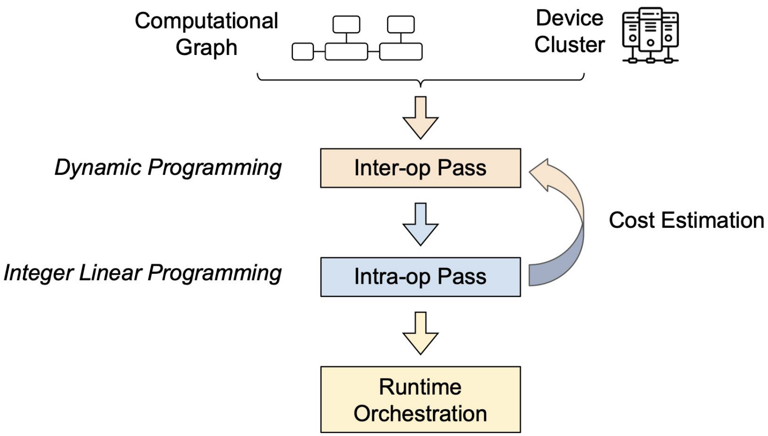


Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. Zheng et al. OSDI 2022 https://arxiv.org/pdf/2201.12023.pdf

Automatic Parallelization with Alpa

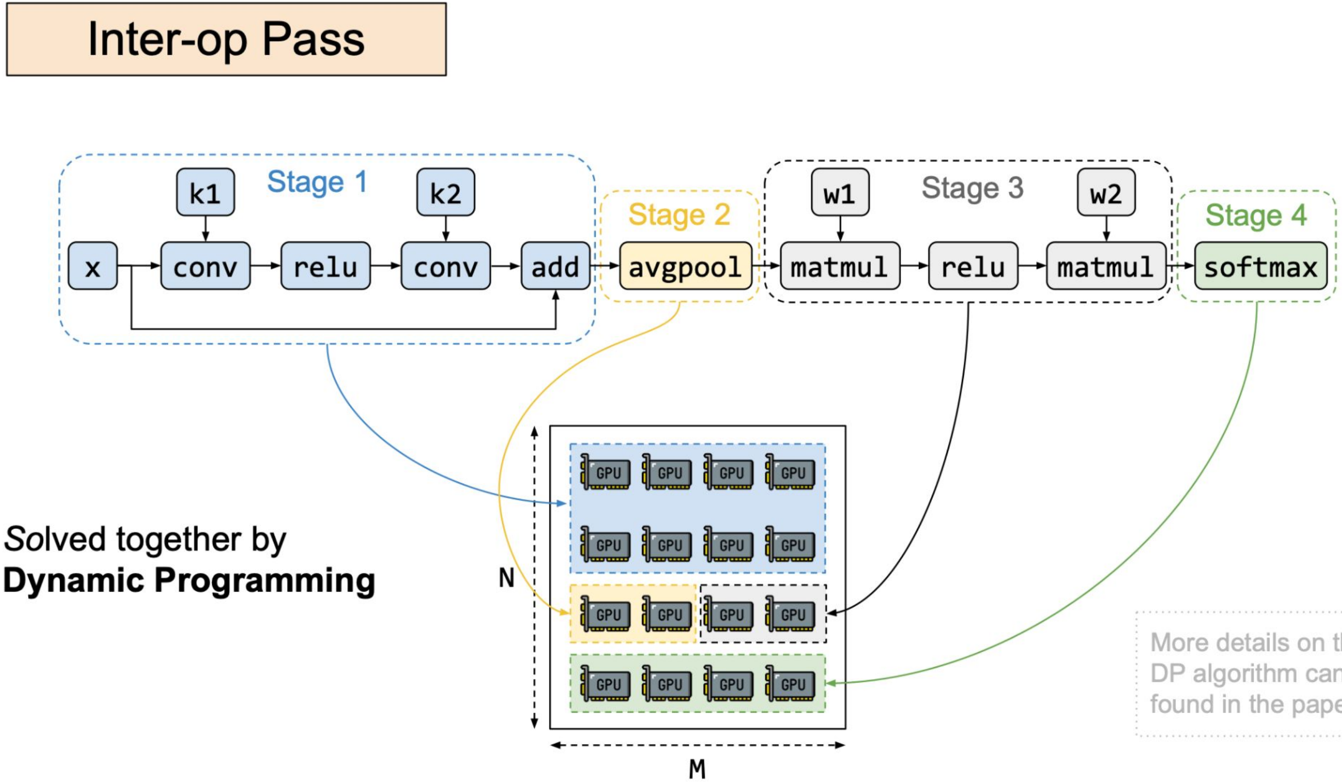
- Alpa will automatically search over the space of possible parallelism strategies for a given graph and choose the optimal combination of inter-op and intra-op parallelism





Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. Zheng et al. https://arxiv.org/pdf/2201.12023.pdf

Automatic Parallelization with Alpa



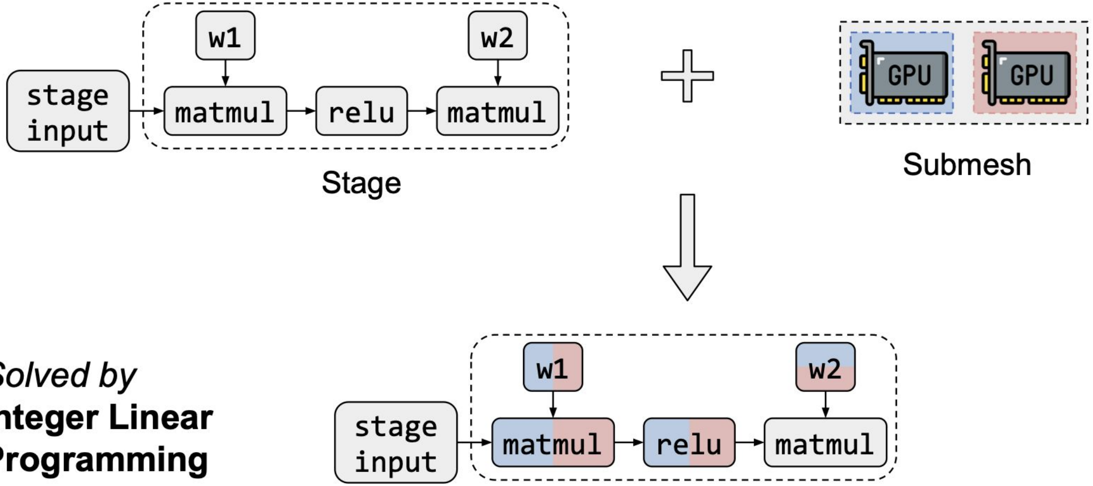
Solved together by
Dynamic Programming

More details on the DP algorithm can be found in the paper.

Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. Zheng et al. https://arxiv.org/pdf/2201.12023.pdf

Automatic Parallelization with Alpa

Intra-op Pass

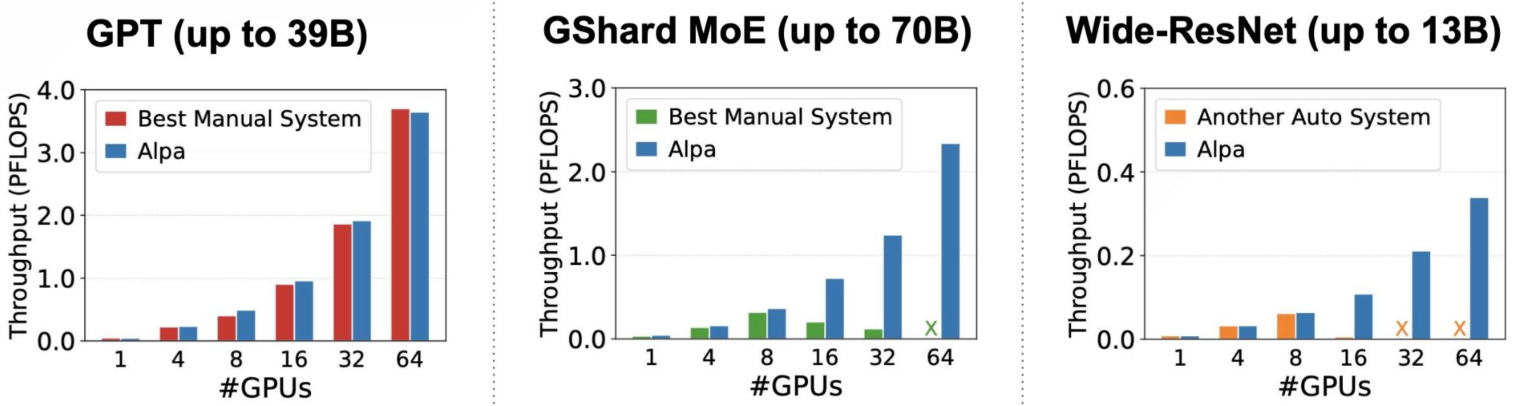


Solved by **Integer Linear Programming**

Stage with intra-operator parallelization

Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. Zheng et al. https://arxiv.org/pdf/2201.12023.pdf

Automatic Parallelization with Alpa



Match specialized manual systems.

Outperform the manual baseline by up to 8x.

Generalize to models without manual plans.

Weak scaling results where the model size grow with #GPUs.
Evaluated on 8 AWS EC2 p3.16xlarge nodes with 8 16GB V100s each (64 GPUs in total).

Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. Zheng et al. https://arxiv.org/pdf/2201.12023.pdf

Conclusion

- We need to parallelize data and models to train and serve LLMs at scale
- The most widely used forms of parallelism are data parallelism, tensor model parallelism, and pipeline parallelism
- Each of these parallelization strategies has trade-offs depending on the amount of memory usage and communication overhead
- Techniques such as Alpa can help automate the decision of which parallelization strategy or strategies to use