DISTRIBUTED TRAINING WITH PYTORCH

Umar Jamil

Downloaded from: https://github.com/hkproj/pytorch-transformer-distributed
License: Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0):
https://creativecommons.org/licenses/by-nc/4.0/legalcode

Not for commercial use

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

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?

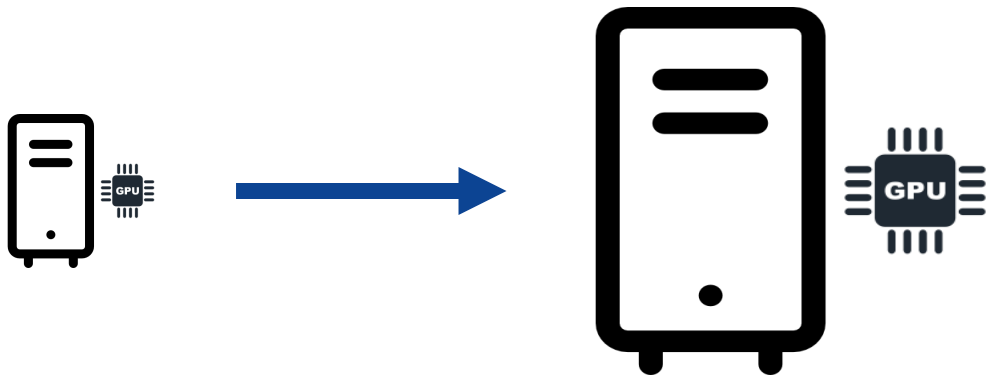
Imagine you want to train a Language Model on a very big dataset, for example the entire content of Wikipedia. The dataset is quite big, because it is made up of millions of articles, each of them with thousands of tokens. To train this model on a single GPU may be possible, but it poses some challenges:

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

No code change



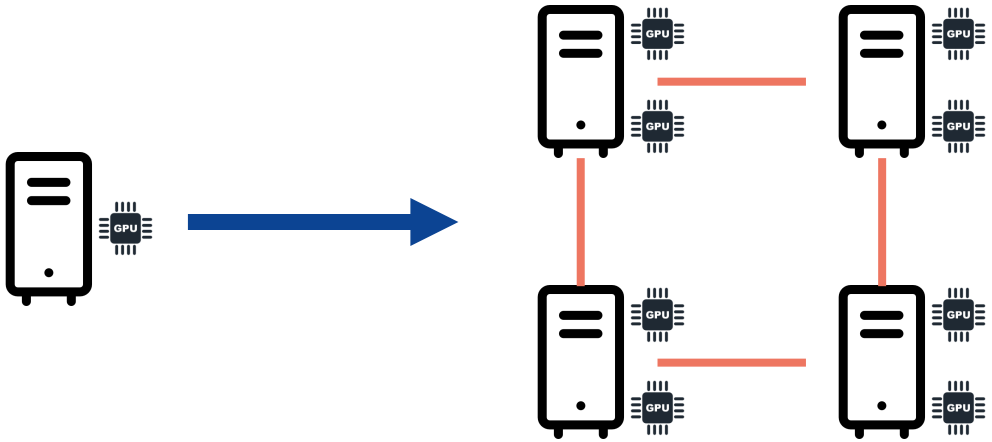
1x Server
8GB RAM
4GB GPU Memory

1x Server
64GB RAM
32GB GPU Memory

Horizontal scaling

Minimal code change (thanks to PyTorch)

In this video we will explore horizontal scaling



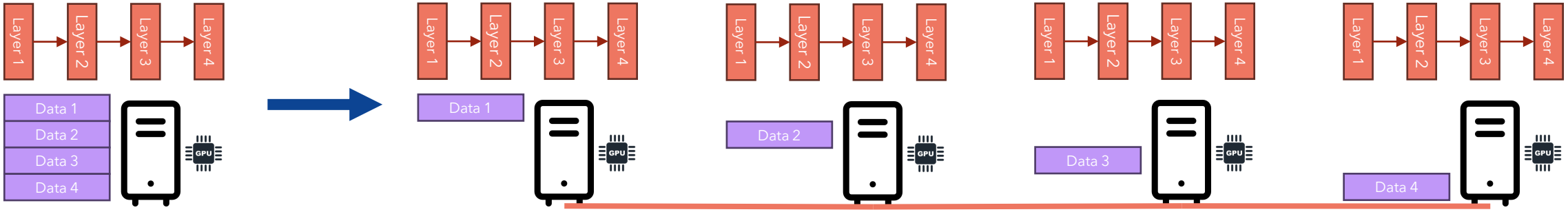
1x Server
8GB RAM
4GB GPU Memory

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

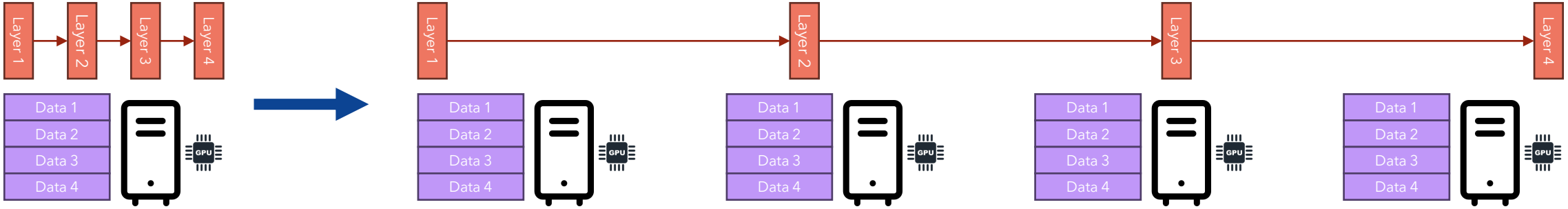
Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

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)

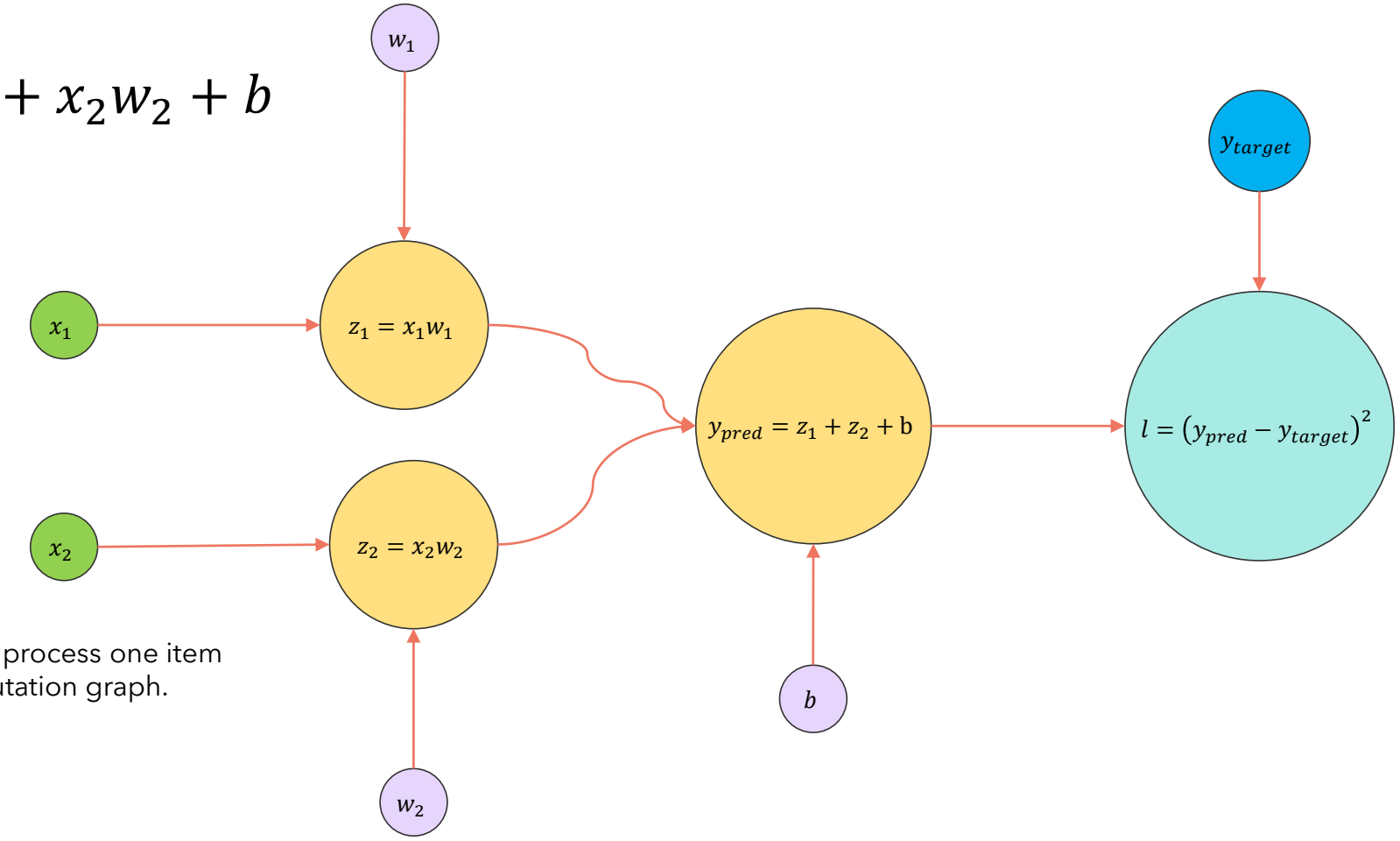
```python
def train_no_accumulate(params: ModelParameters, num_epochs: int = 10, learning_rate: float = 1e-3):
    for epoch in range(1, num_epochs+1):
        for (x1, x2), y_target in training_data:
            # Calculate the output of the model
            z1 = x1 * params.w1
            z2 = x2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

Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

Computation graph

PyTorch will convert our neural network into a computational graph.

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



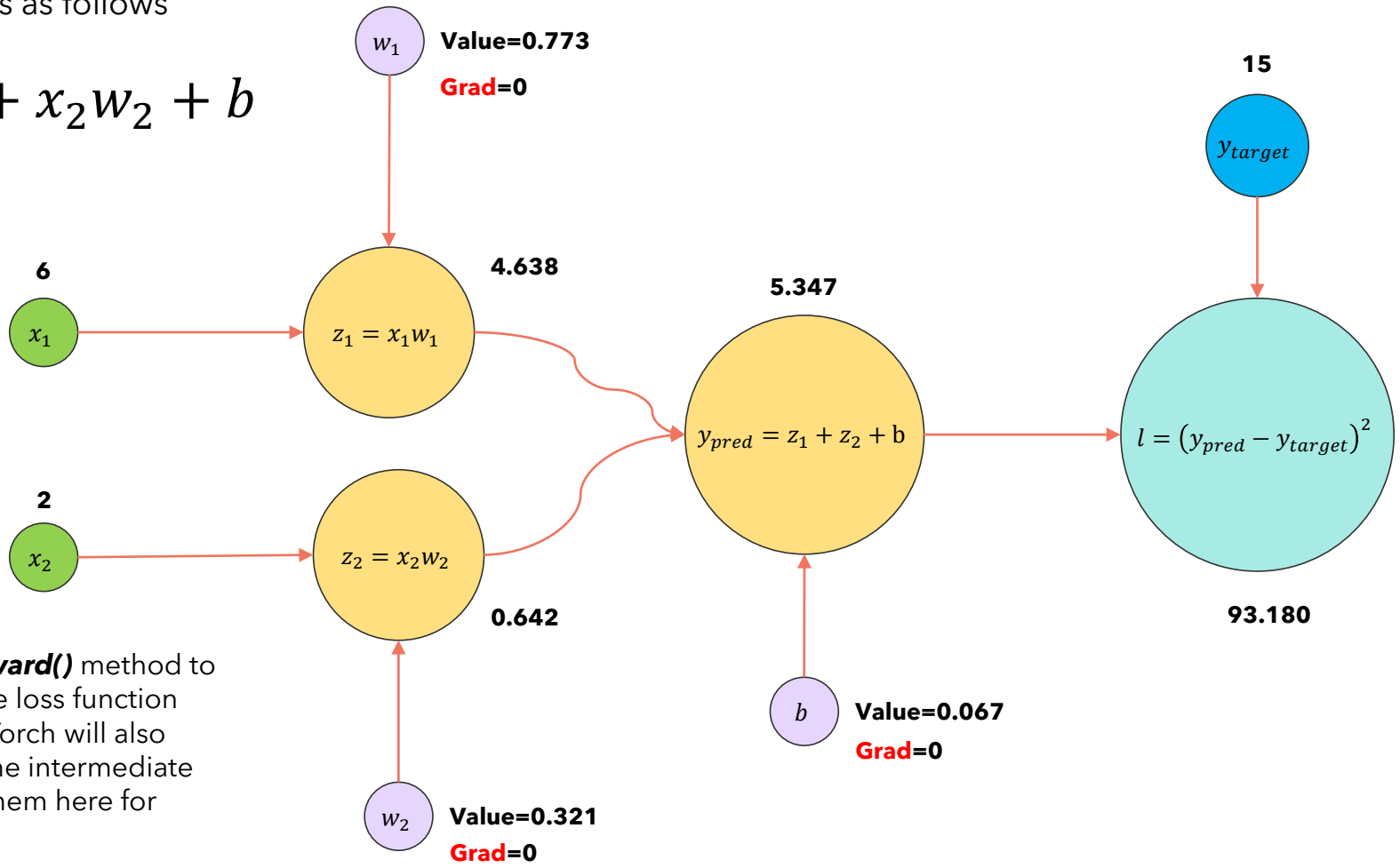
Let's visualize the training process one item at a time using our computation graph.

Computational graph: step 1 (*forward*)

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

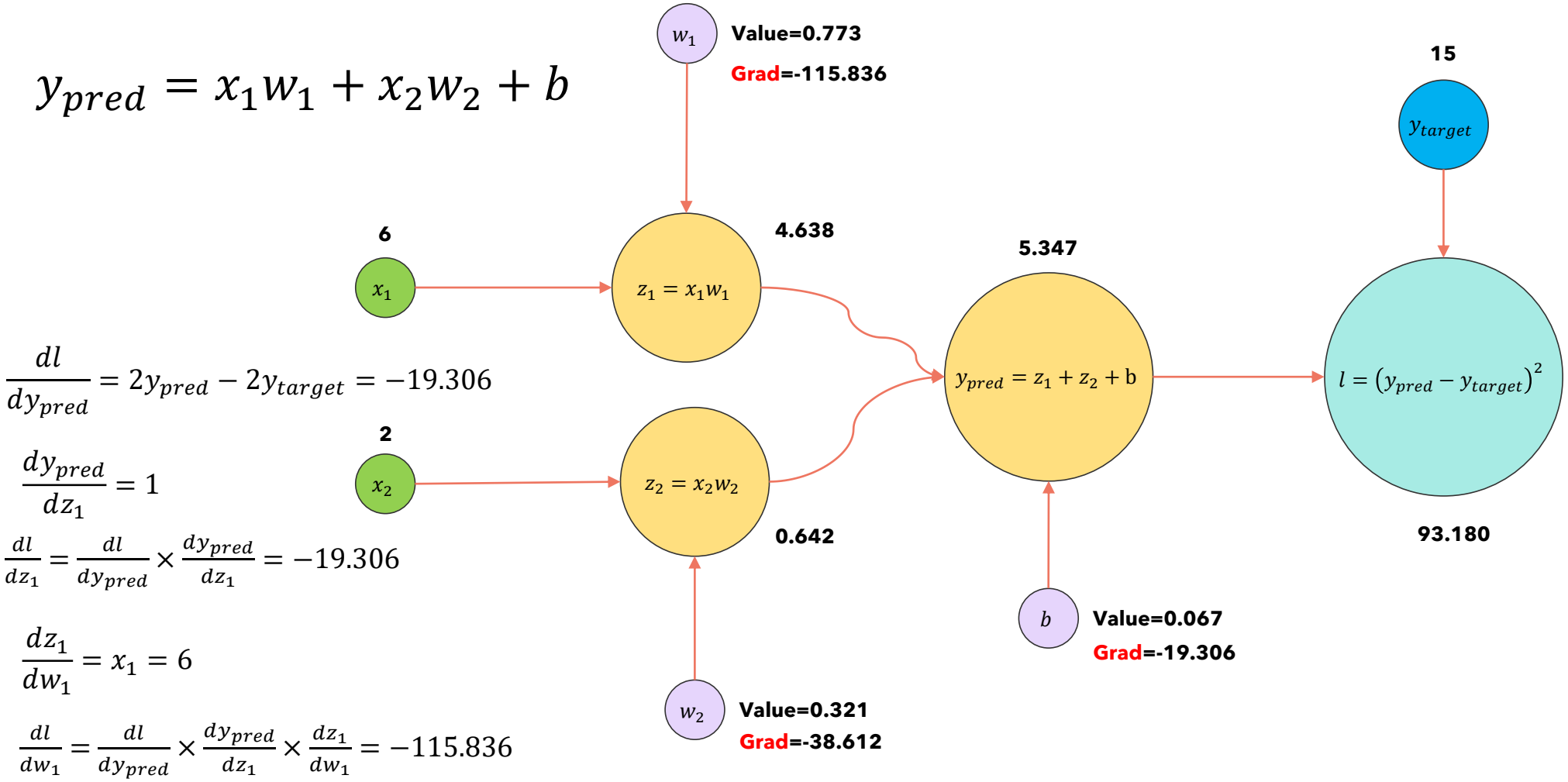
We initialize the weights as follows

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



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*)

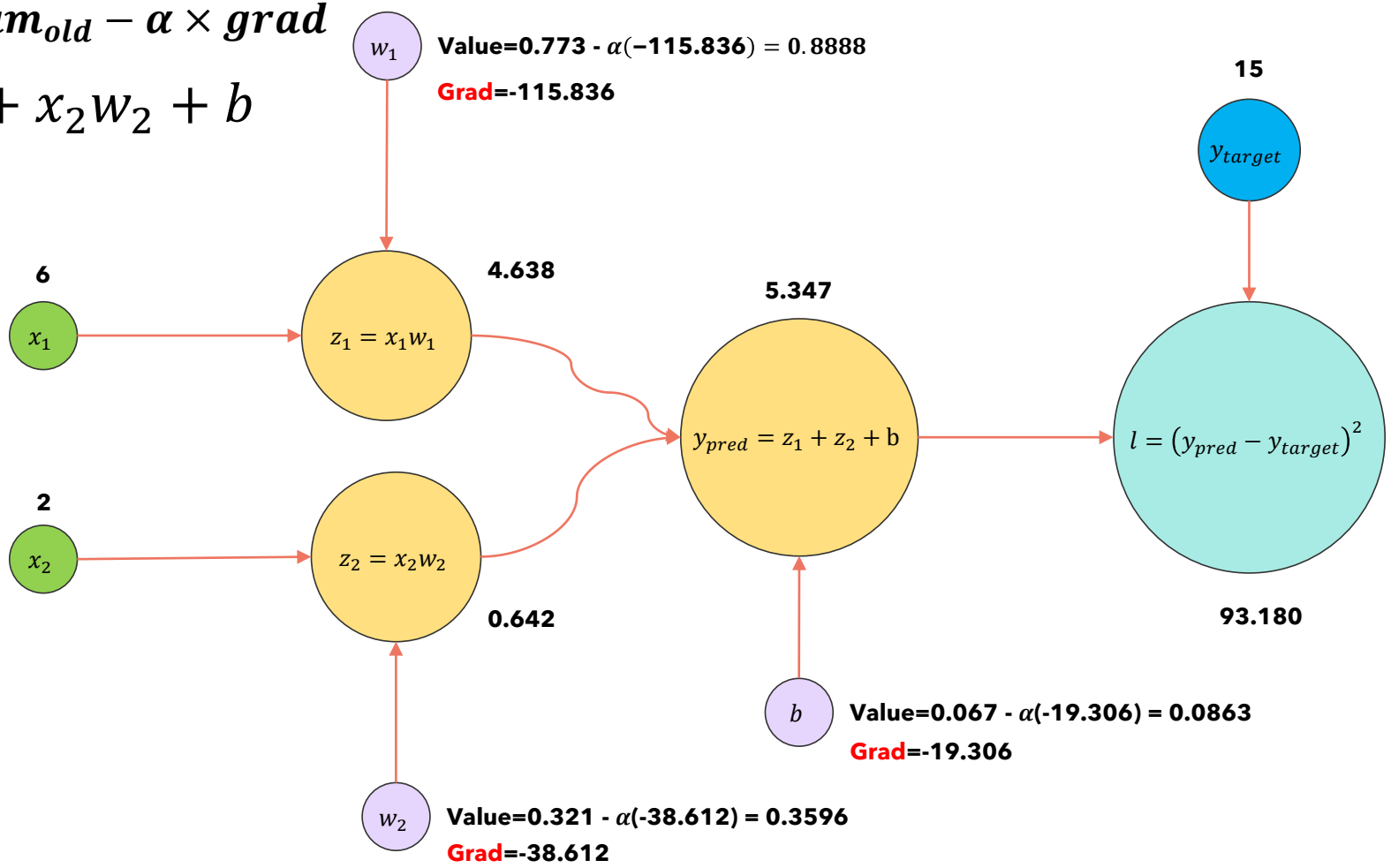


Computational graph: step 1 (*optimizer.step*)

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

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

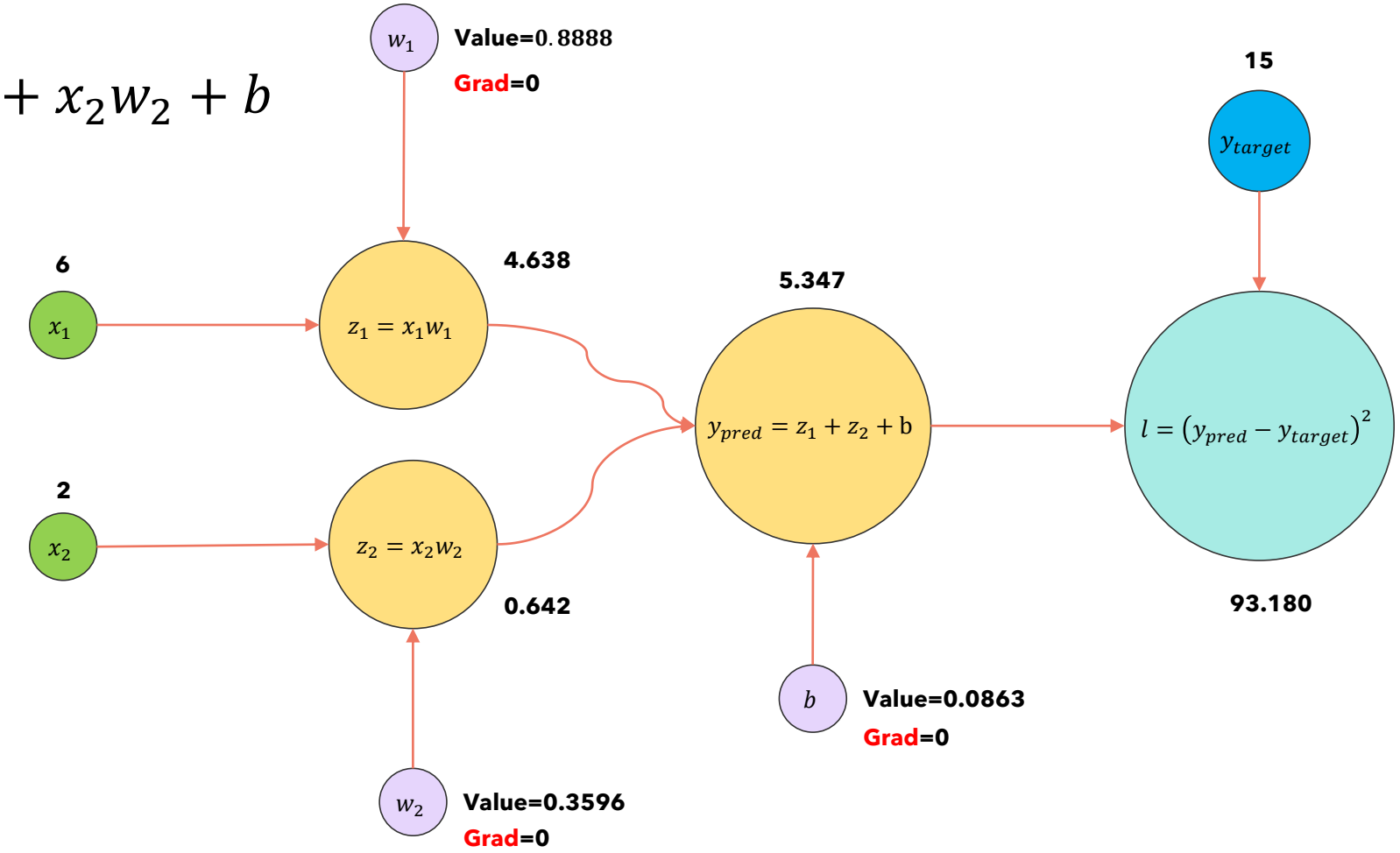
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



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

We reset the gradient of all the parameters to zero.

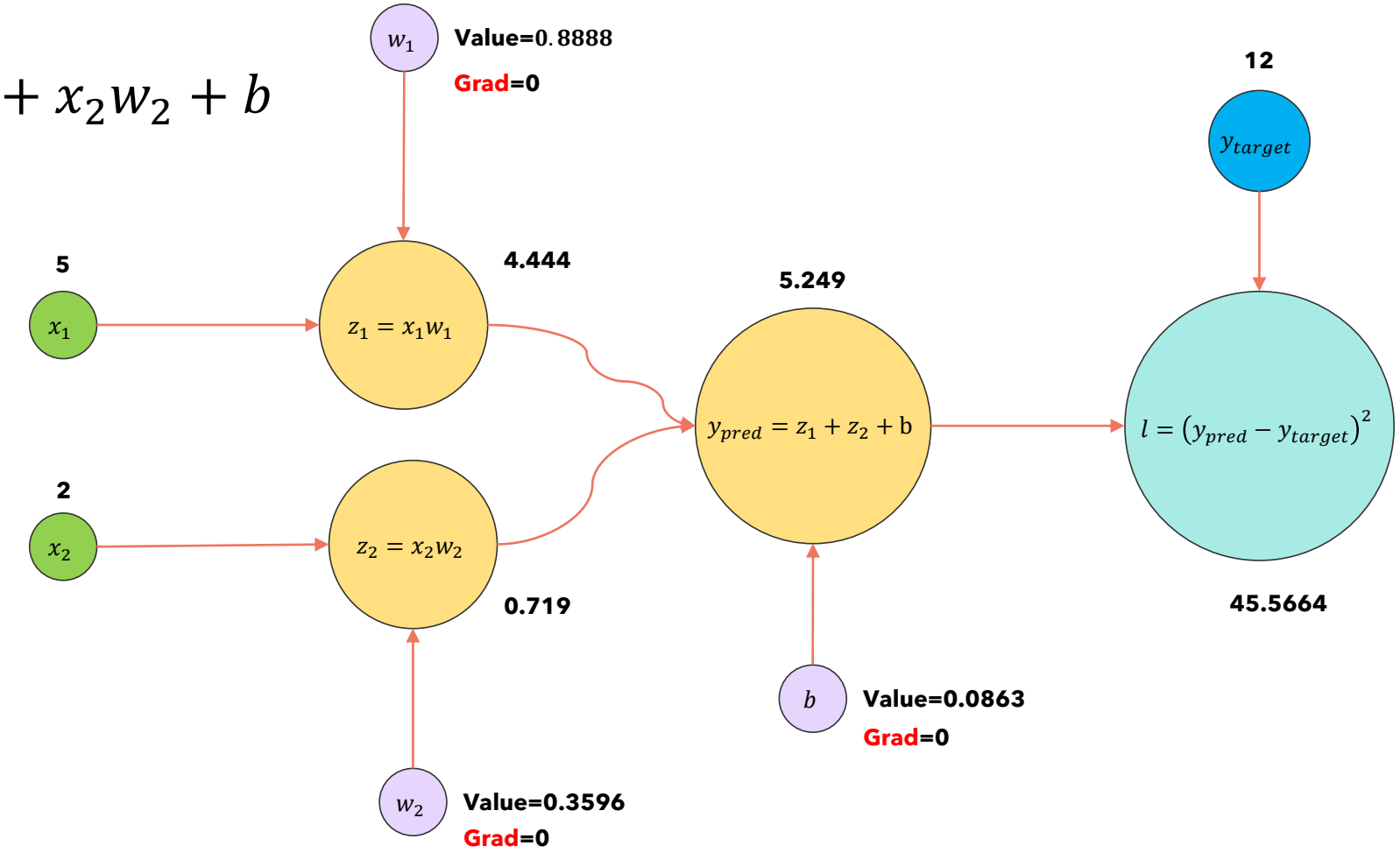
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



Computational graph: step 2 (*forward*)

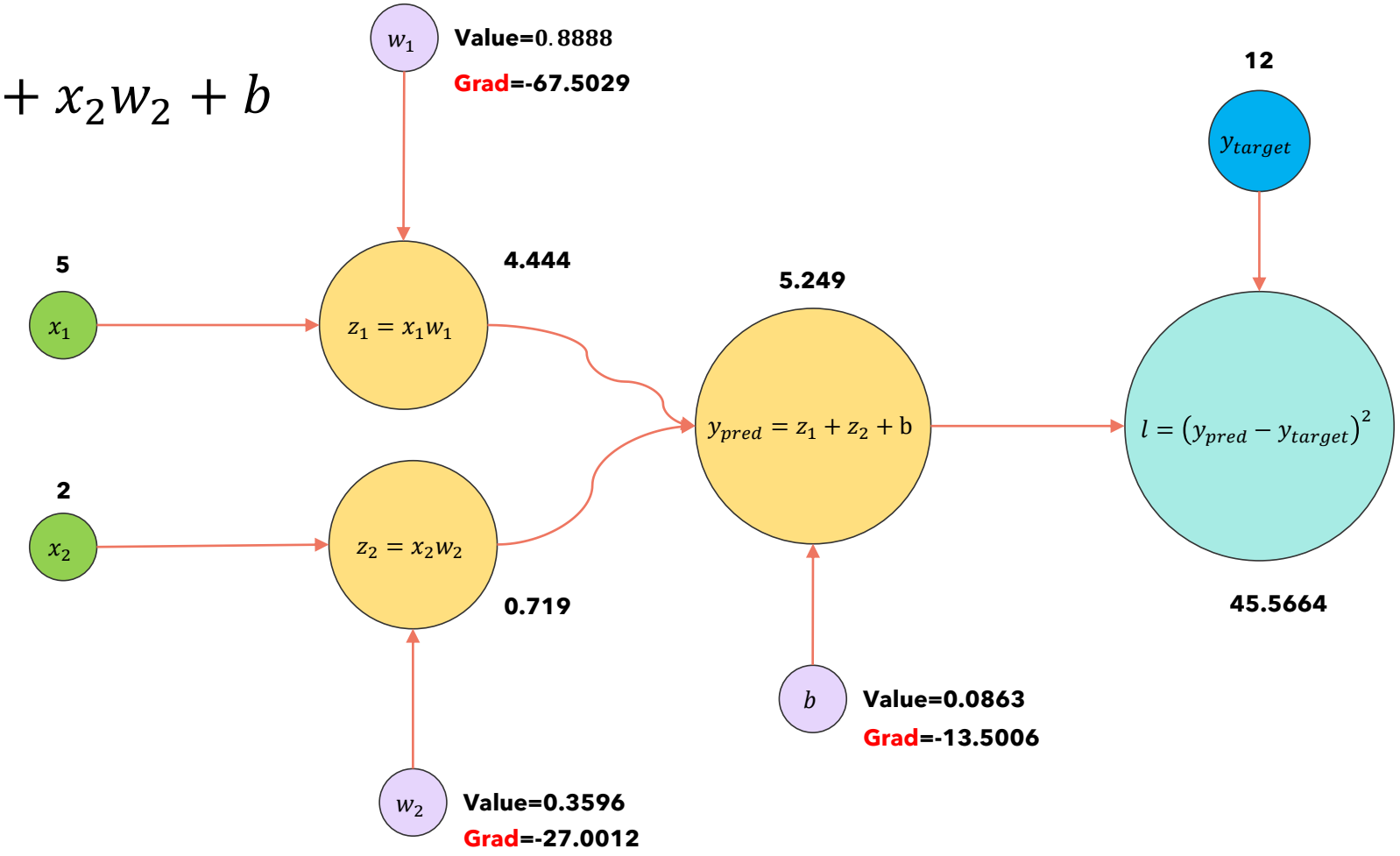
We run a forward step using the input $x_1 = 5$, $x_2 = 2$ and $y_{target} = 12$.

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



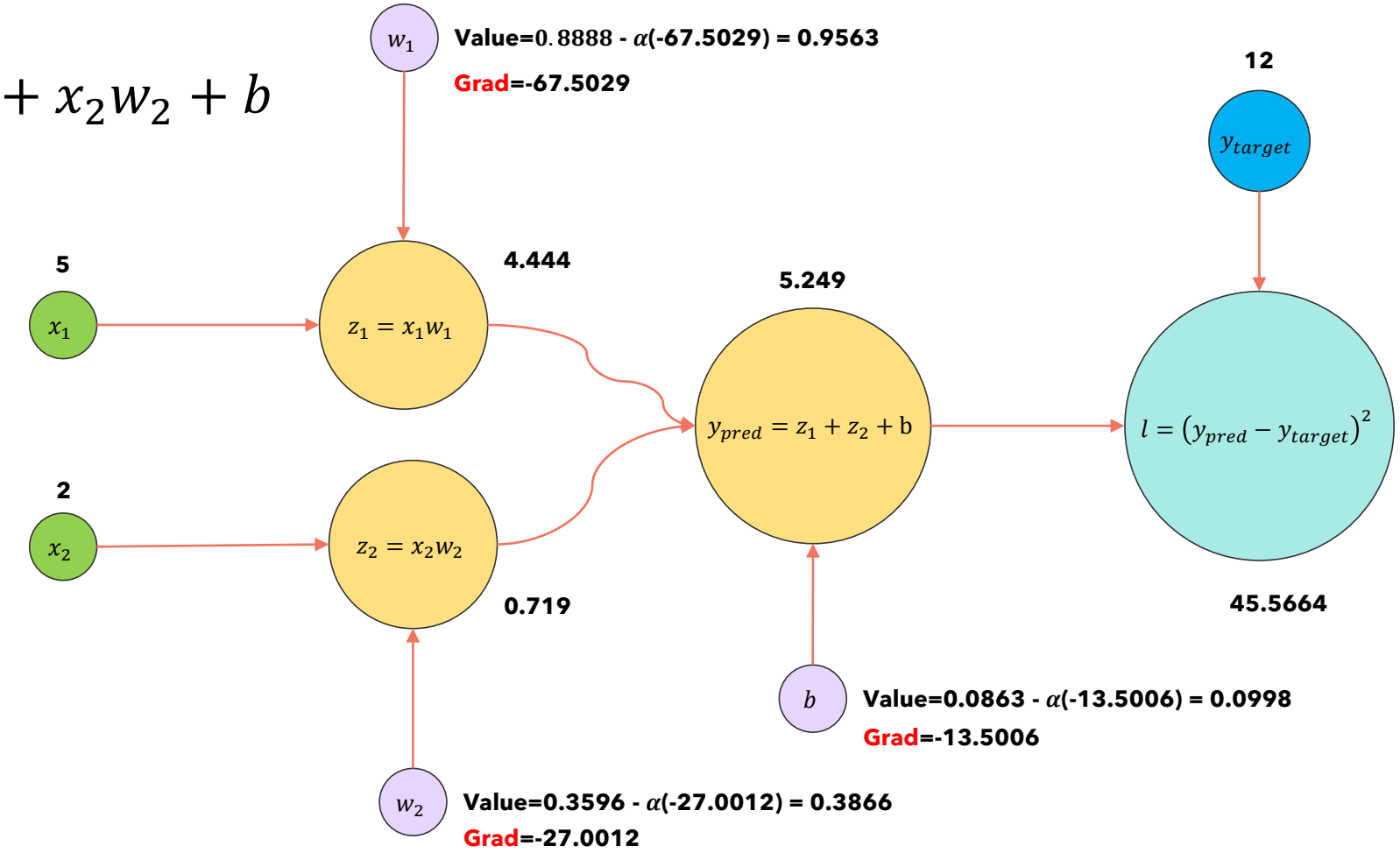
Computational graph: step 2 (*loss.backward*)

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



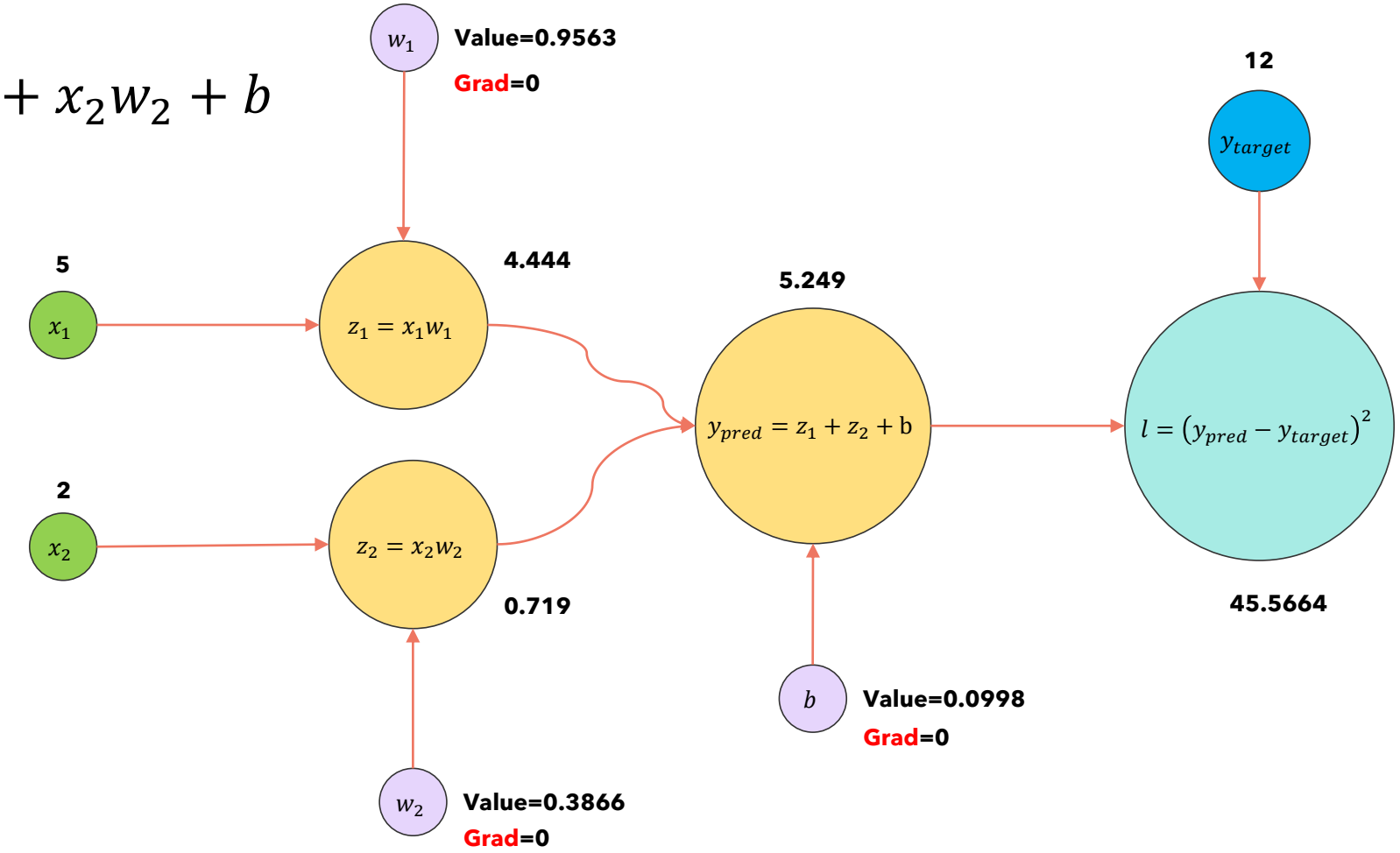
Computational graph: step 2 (*optimizer.step*)

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

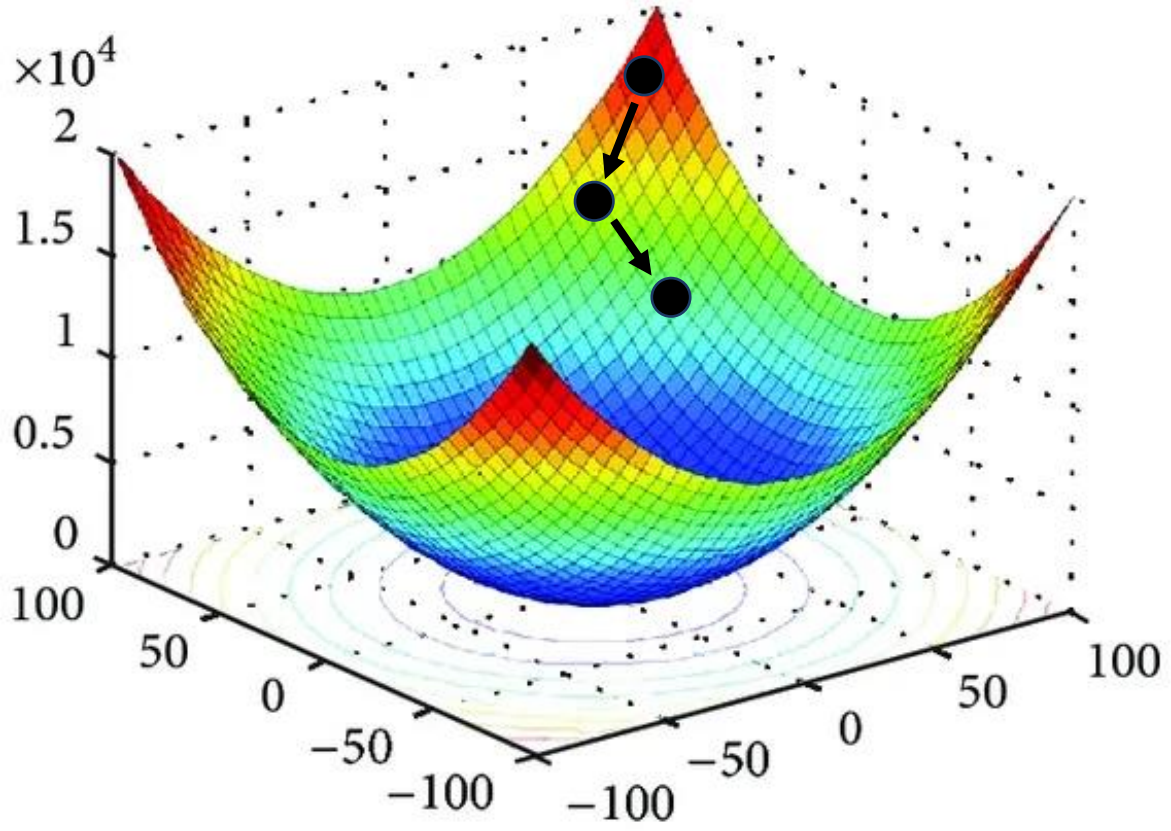


Computational graph: step 2 (*optimizer.zero*)

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



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)

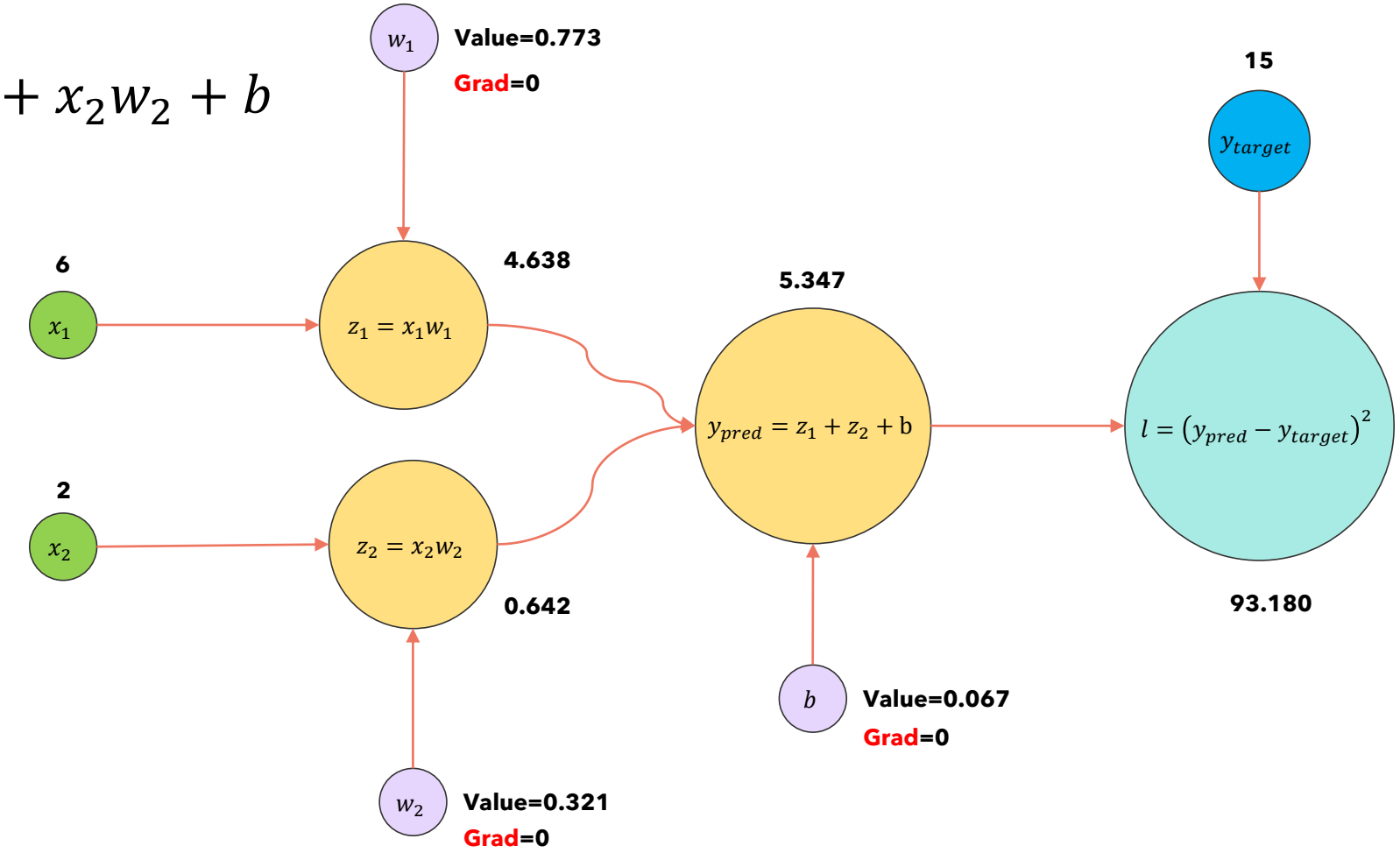
```python
def train_accumulate(params: ModelParameters, num_epochs: int = 10, learning_rate: float = 1e-3, batch_size: int = 2):
    for epoch in range(1, num_epochs+1):
        for index, ((x1, x2), y_target) in enumerate(training_data):
            # Calculate the output of the model
            z1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

Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

Computational graph: step 1 (*forward*)

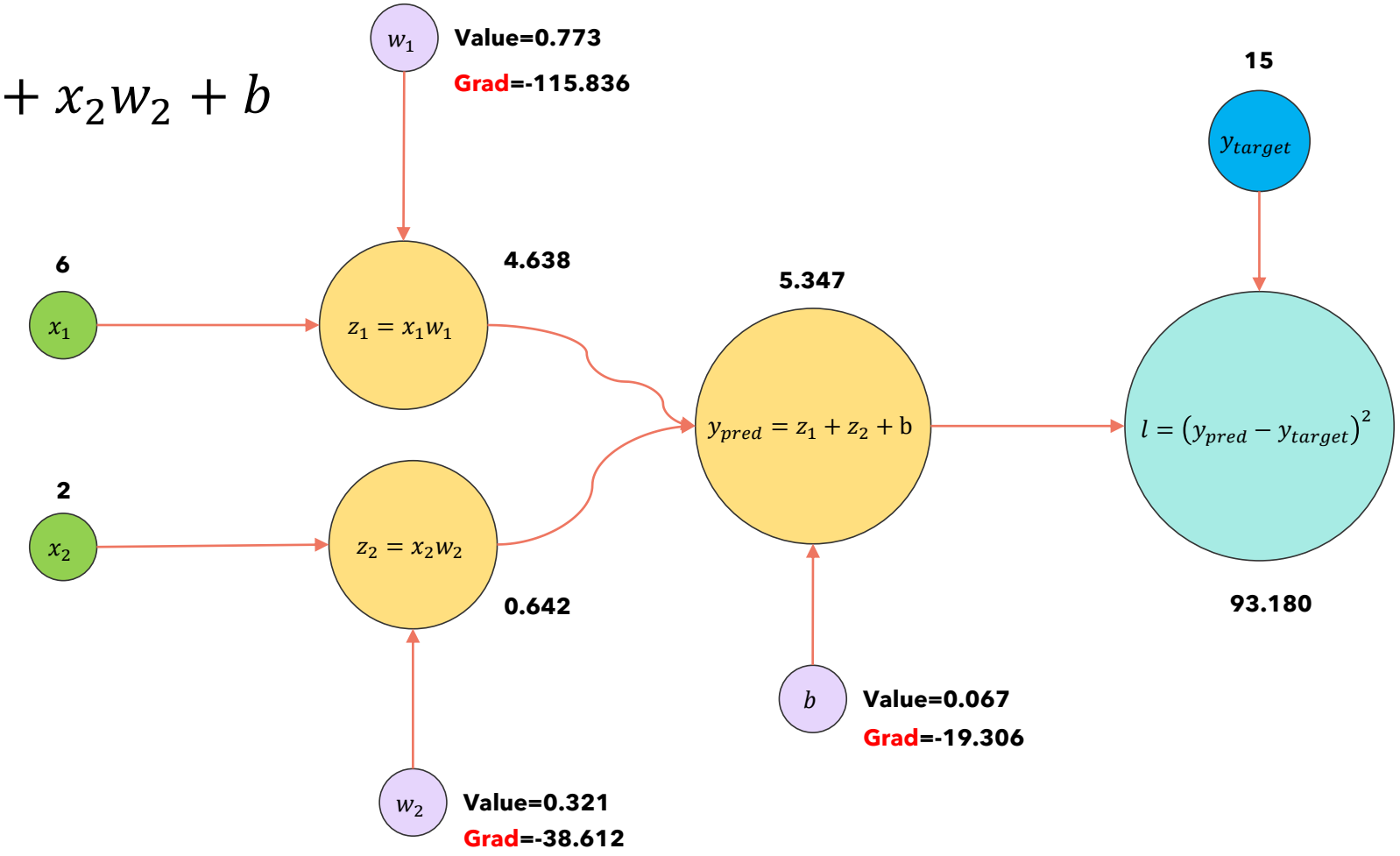
We run a forward step using the input $x_1 = 6$, $x_2 = 2$ and $y_{target} = 15$.

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



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

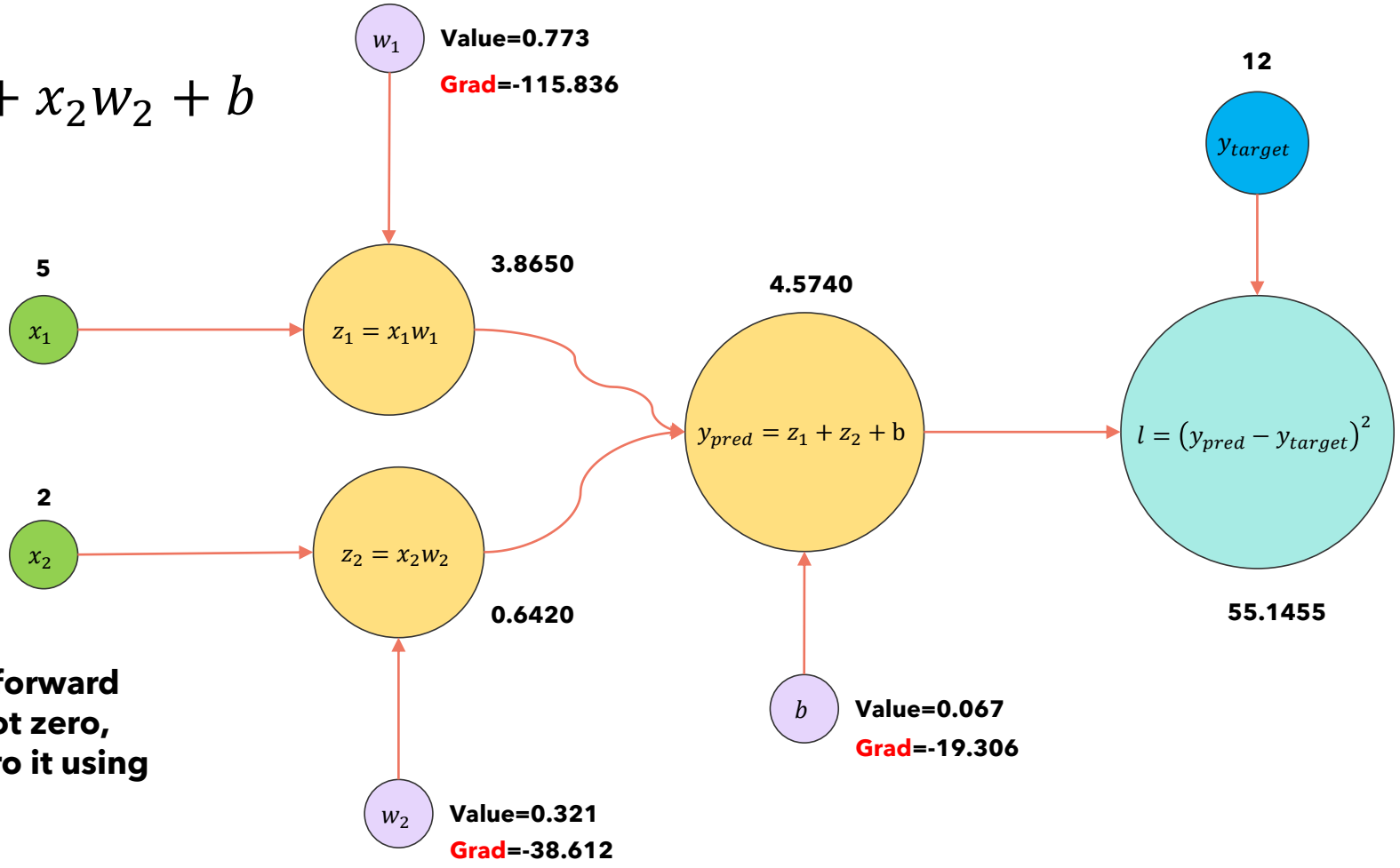
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



Computational graph: step 2 (*forward*)

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

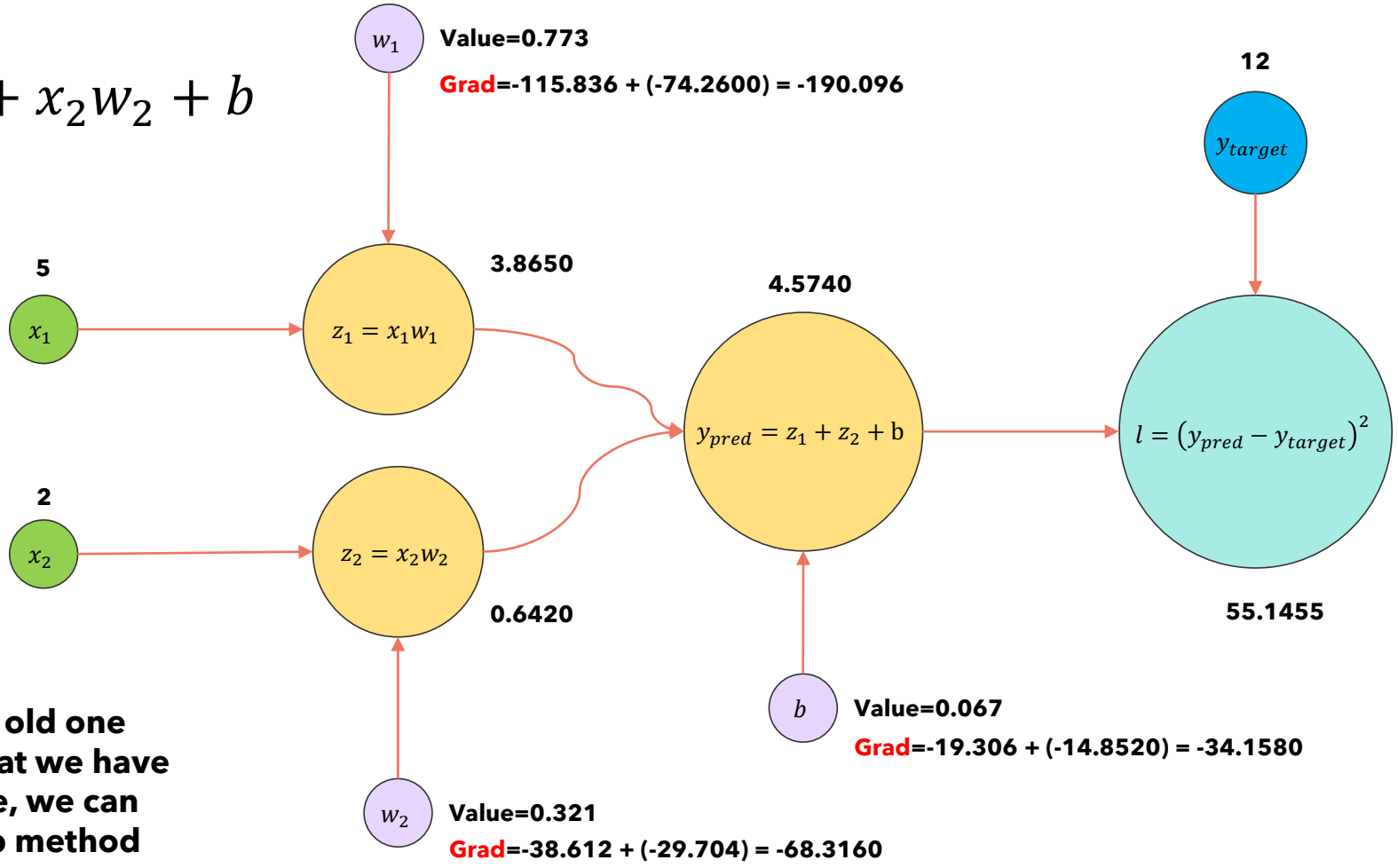
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



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)

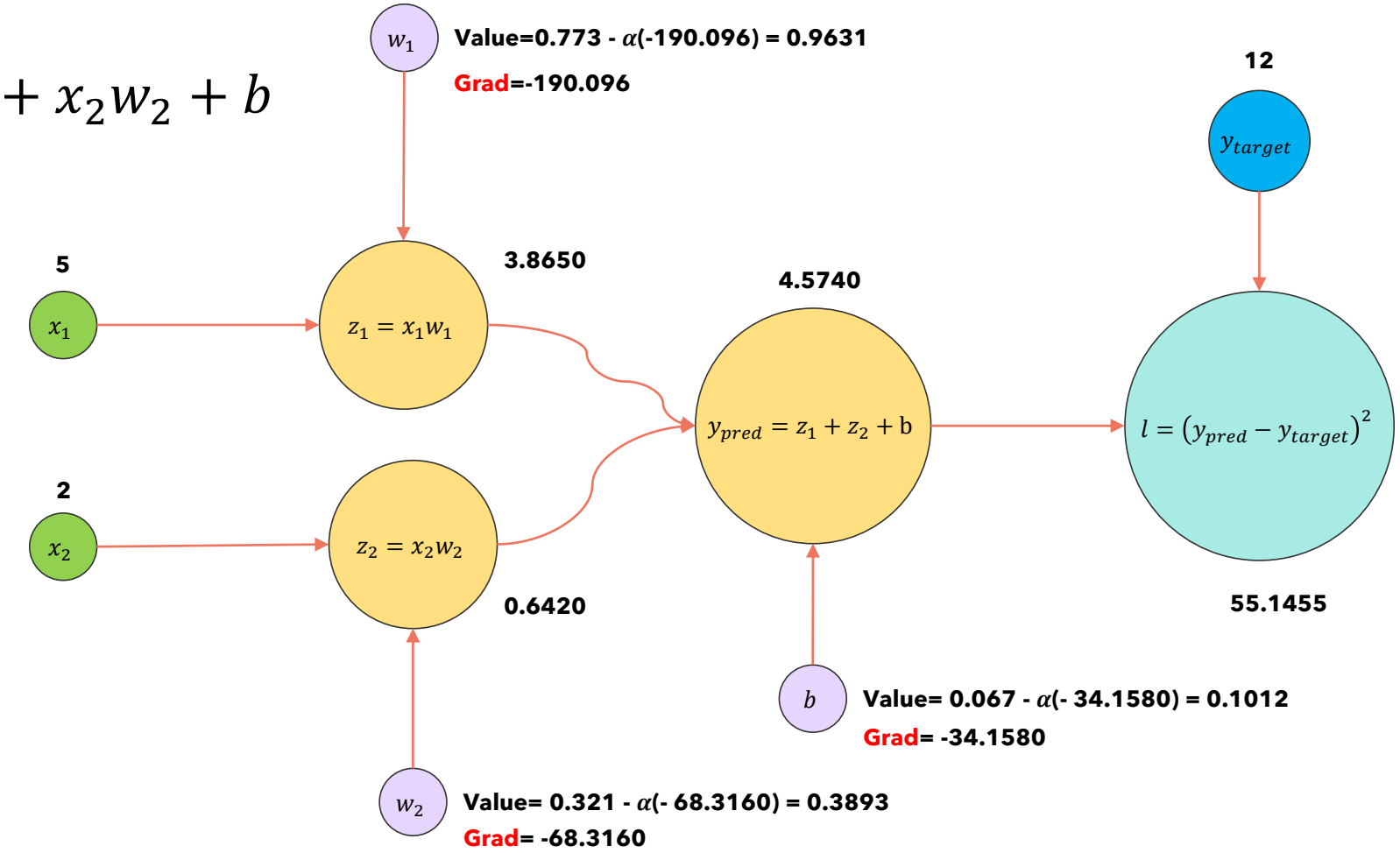
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



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)

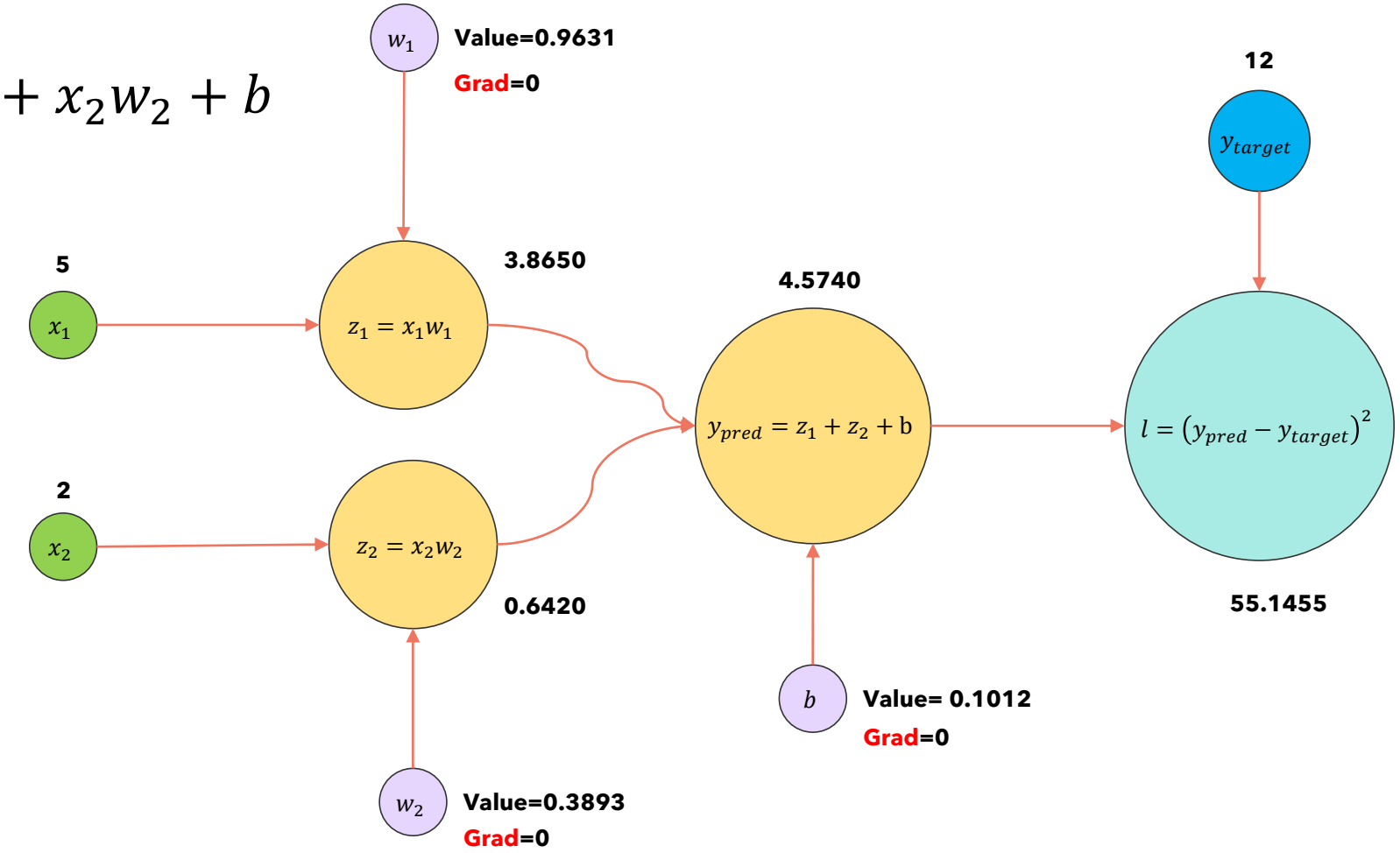
$$y_{pred} = x_1 w_1 + x_2 w_2 + b$$



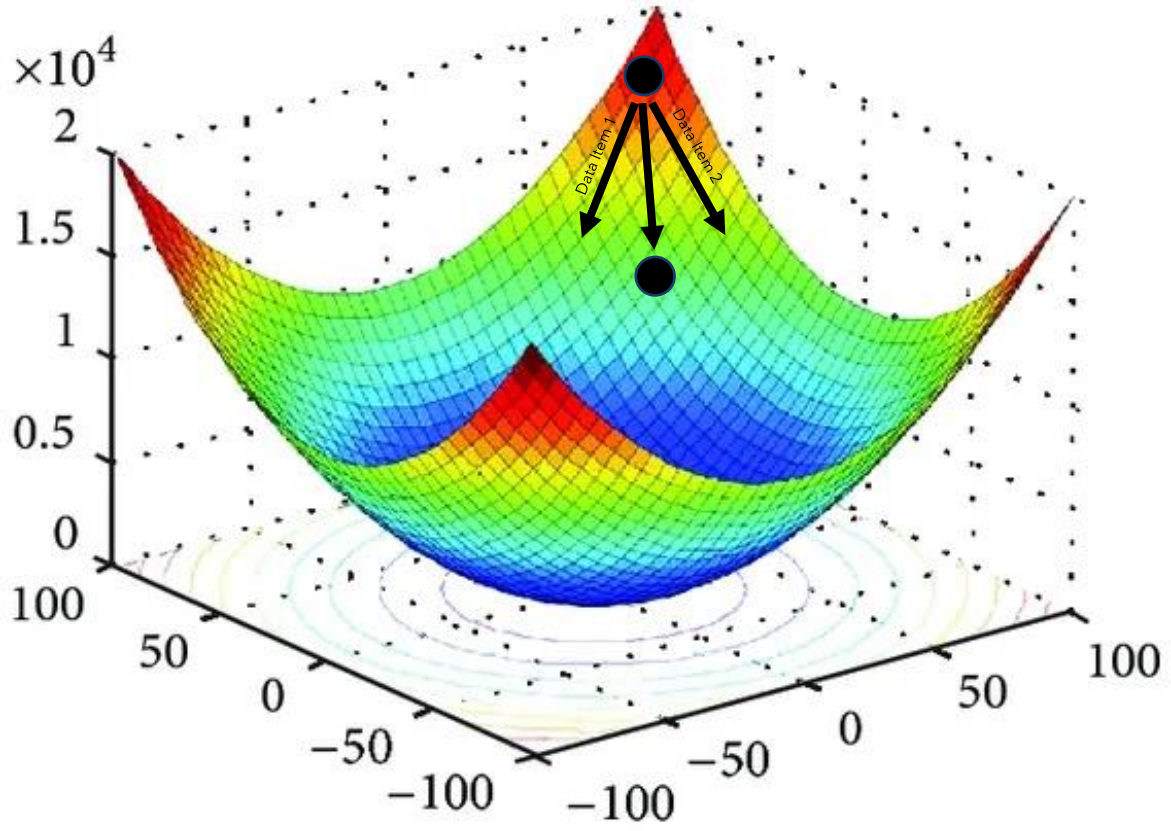
Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

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)

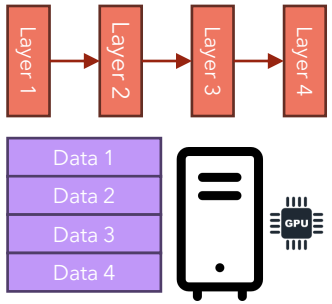
With gradient accumulation, we update the parameters of the model only after we accumulated the gradient of a batch

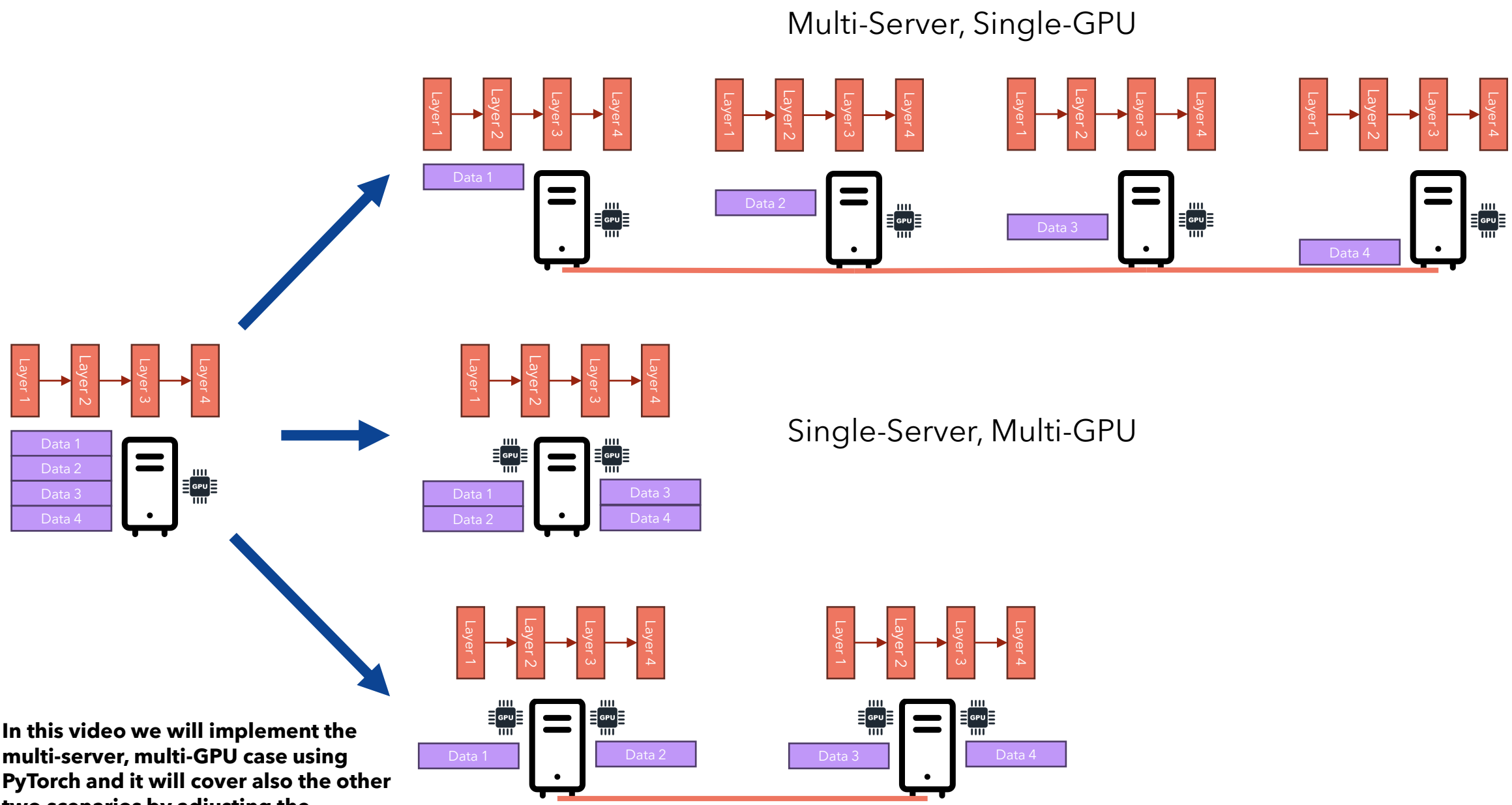
Outline

- 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

Distributed Data Parallel in detail

Imagine you have a training script that is running on a single computer/GPU, but it's very slow, because: the dataset is big and you can't use a big batch size as it will result in an Out Of Memory error on CUDA. Distributed Data Parallel is the solution in this case. It works in the following scenarios:





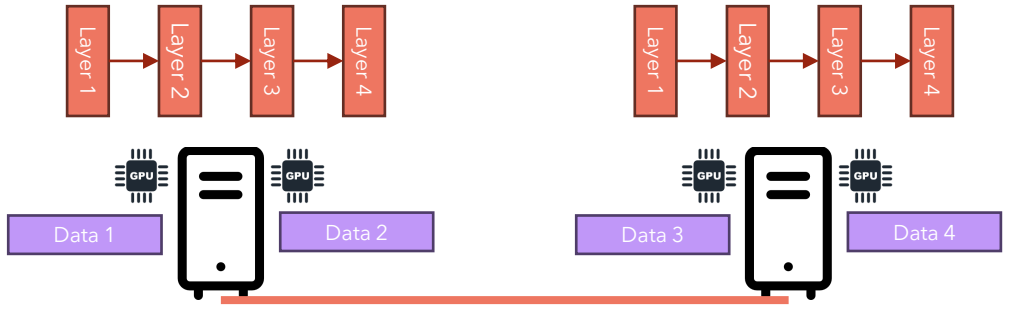
In this video we will implement the multi-server, multi-GPU case using PyTorch and it will cover also the other two scenarios by adjusting the parameters.

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

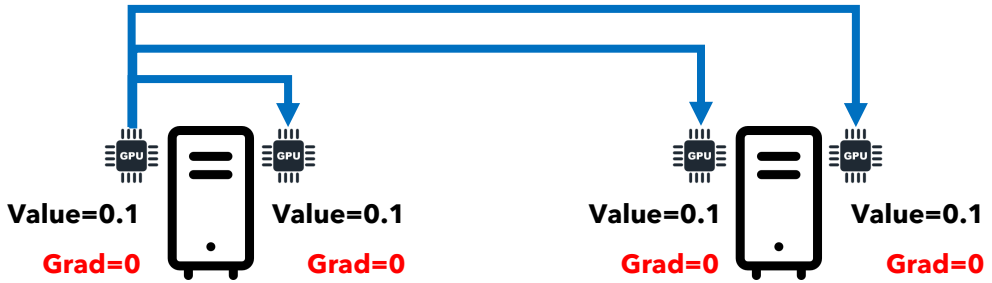


Distributed Data Parallel: step 1

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

Distributed Data Parallel: step 1

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

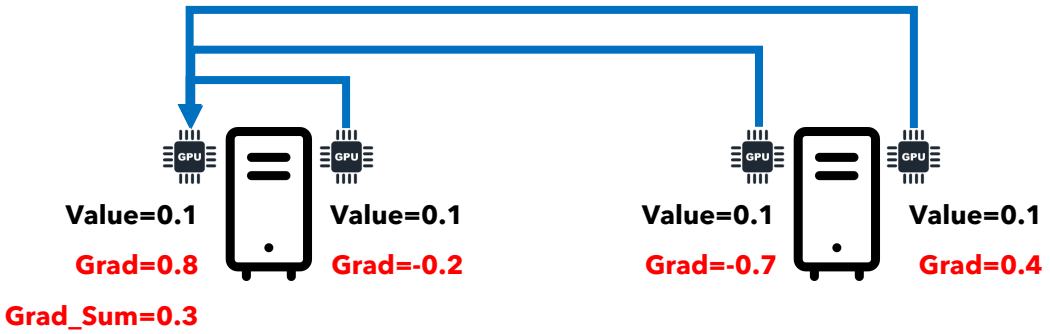


Distributed Data Parallel: step 2

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.

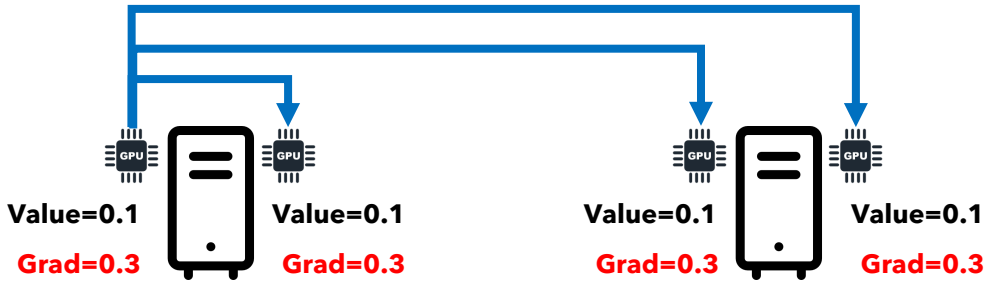
Distributed Data Parallel: step 3

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

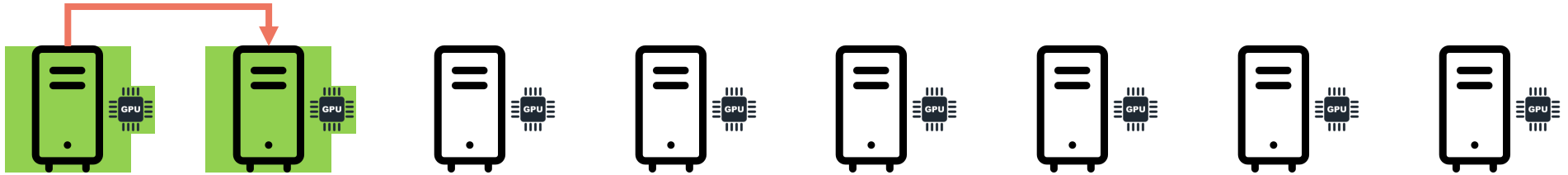
In distributed computing environments, a node may need to communicate with other nodes. If the communication pattern is similar to a client and a server, then we talk about point-to-point communication, because one client connects to one server in a request-response chain of events.

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

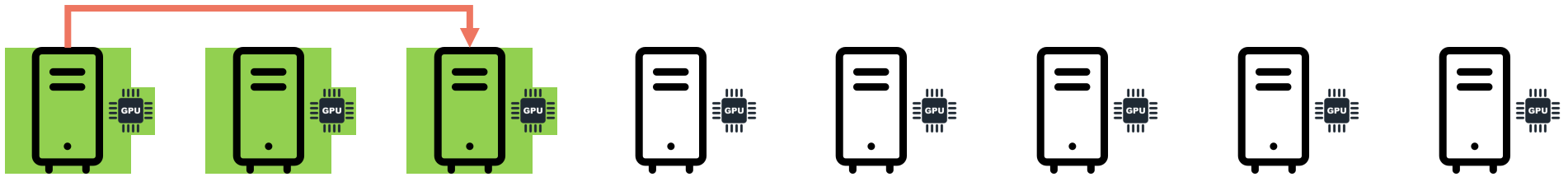
Imagine you need to send a file to 7 friends. With point-to-point communication, you'd send the file iteratively to each of the friend one by one. Suppose the internet speed is 1 MB/s and the file is 5 MB in size.



Total time: 5s

Point-To-Point

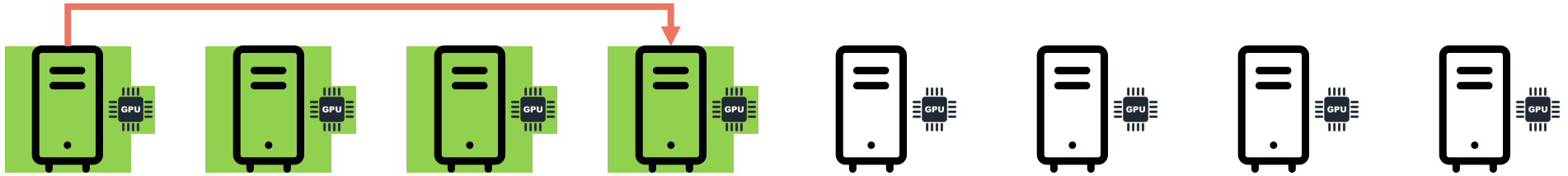
Imagine you need to send a file to 7 friends. With point-to-point communication, you'd send the file iteratively to each of the friend one by one. Suppose the internet communication is 1MB/s and the file is 5MB in size.



Total time: 10s

Point-To-Point

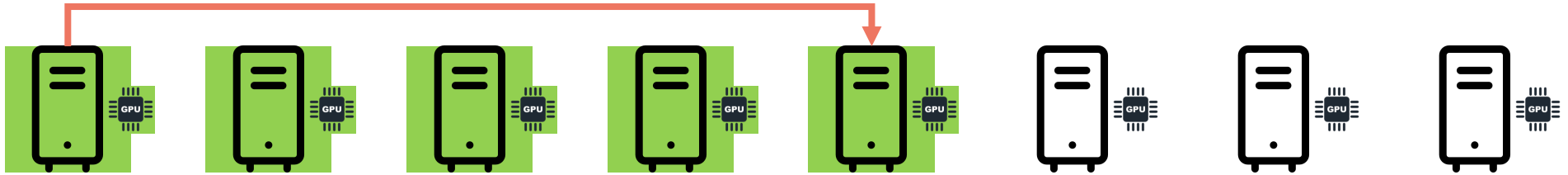
Imagine you need to send a file to 7 friends. With point-to-point communication, you'd send the file iteratively to each of the friend one by one. Suppose the internet communication is 1MB/s and the file is 5MB in size.



Total time: 15s

Point-To-Point

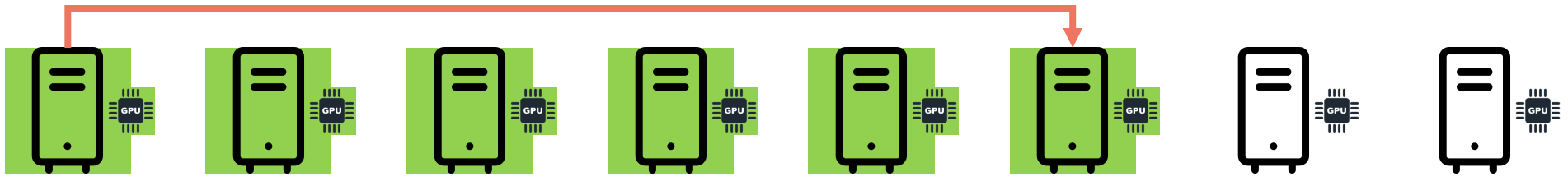
Imagine you need to send a file to 7 friends. With point-to-point communication, you'd send the file iteratively to each of the friend one by one. Suppose the internet communication is 1MB/s and the file is 5MB in size.



Total time: 20s

Point-To-Point

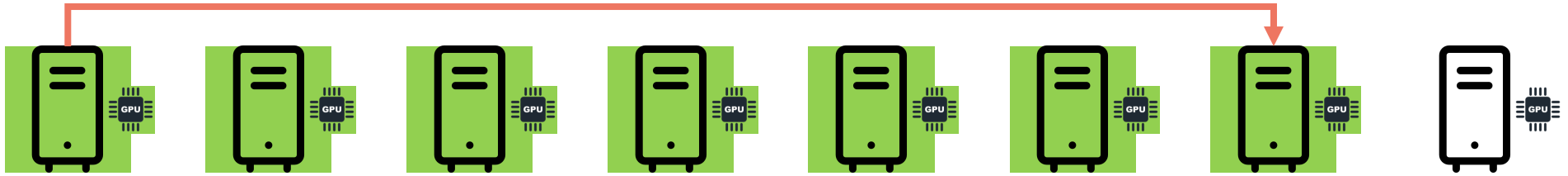
Imagine you need to send a file to 7 friends. With point-to-point communication, you'd send the file iteratively to each of the friend one by one. Suppose the internet communication is 1MB/s and the file is 5MB in size.



Total time: 25s

Point-To-Point

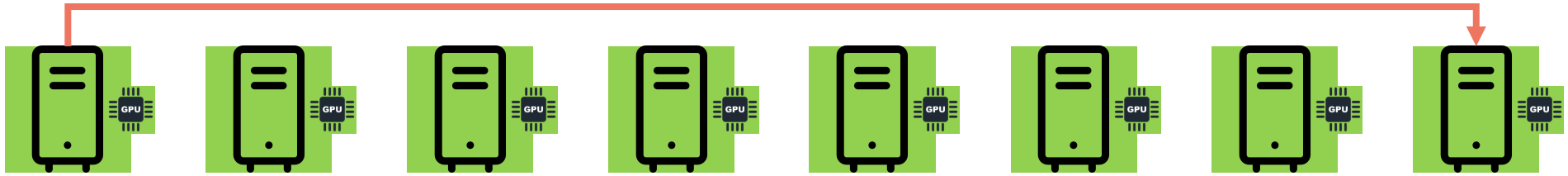
Imagine you need to send a file to 7 friends. With point-to-point communication, you'd send the file iteratively to each of the friend one by one. Suppose the internet communication is 1MB/s and the file is 5MB in size.



Total time: 30s

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 communication is 1MB/s and the file is 5MB in size.

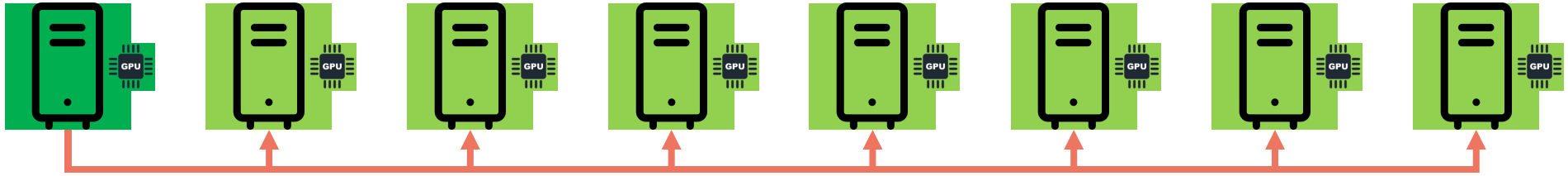


Total time: 35s

What if we sent the file simultaneously to all the 7 friends?

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.**

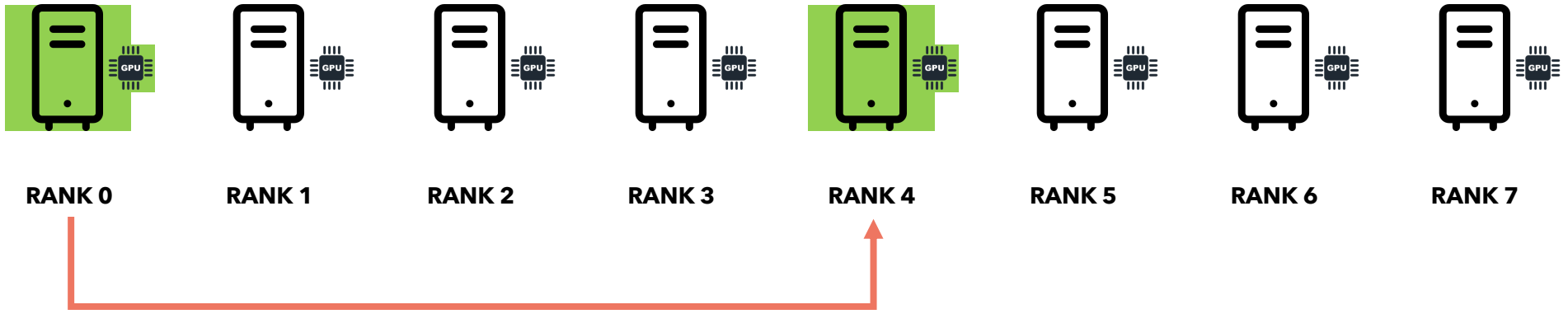


Total time: 35s

Let's see how collective communication would manage this!

Collective Communication: Broadcast

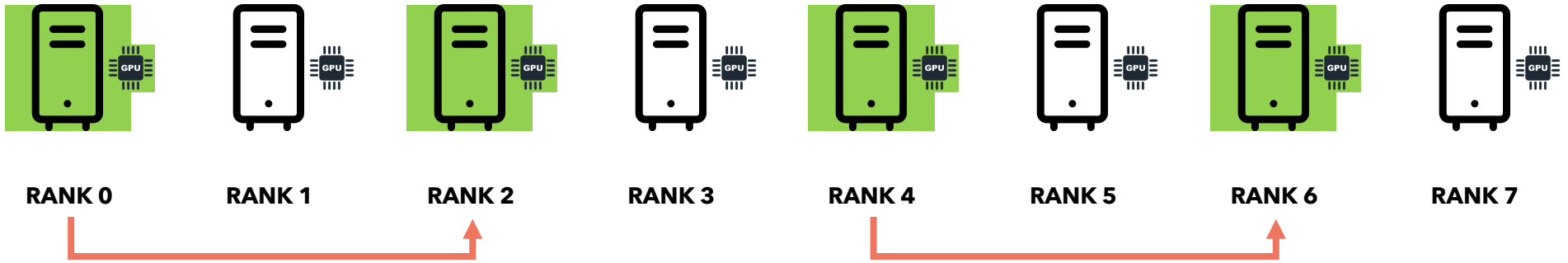
The operation of sending a data to all the other nodes is known as the **Broadcast** operator. Collective Communication libraries (e.g. NCCL) assign a unique ID to each node, known as **RANK**. Suppose we want to send 5 MB with an internet speed of 1 MB/s.



Total time: 5s

Collective Communication: Broadcast

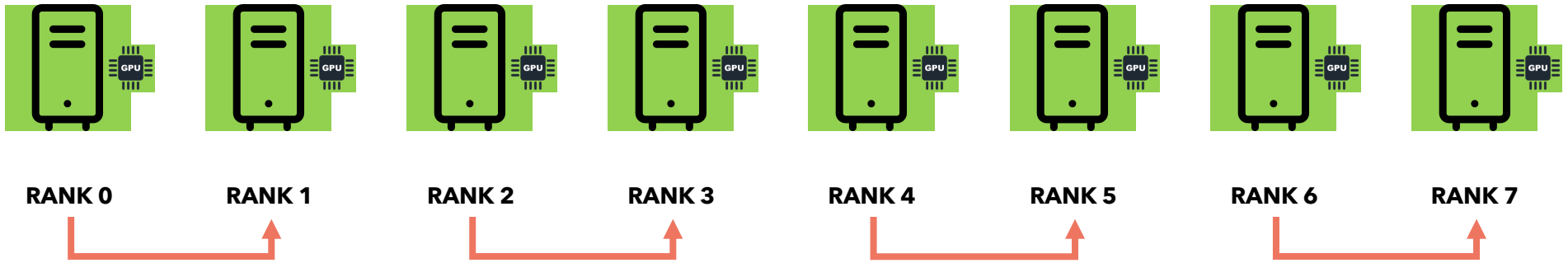
The operation of sending a data to all the other nodes is known as the Broadcast operator. Collective Communication libraries (e.g. NCCL) assign a unique ID to each node, known as **RANK**. Suppose we want to send 5 MB with an interned speed of 1 MB/s.



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

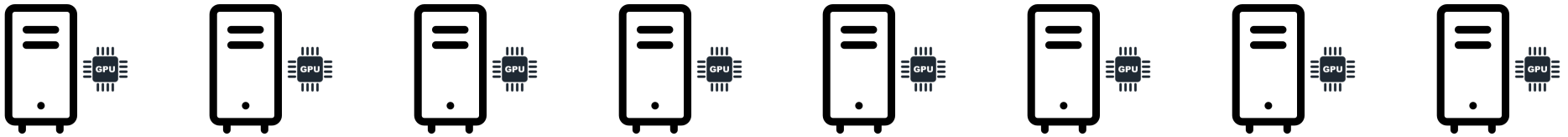
The **Broadcast** operator is used to send the initial weights to all the other nodes when we start the training loop.

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

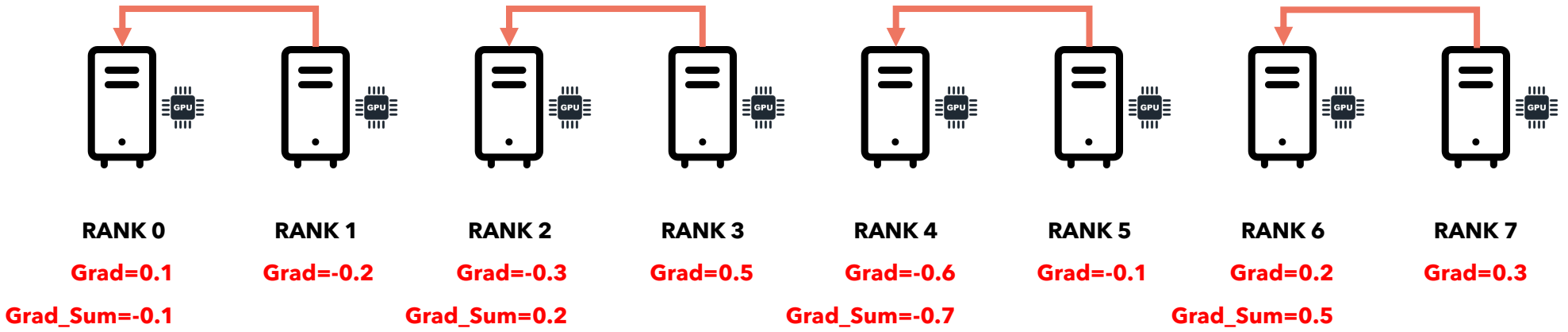
Initially, each node has its own gradient.



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

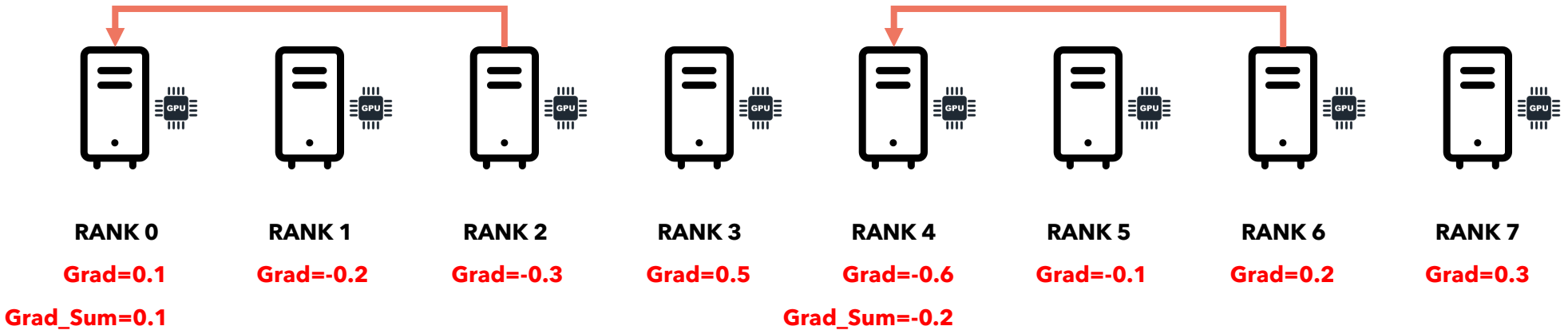
Collective Communication: Reduce

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



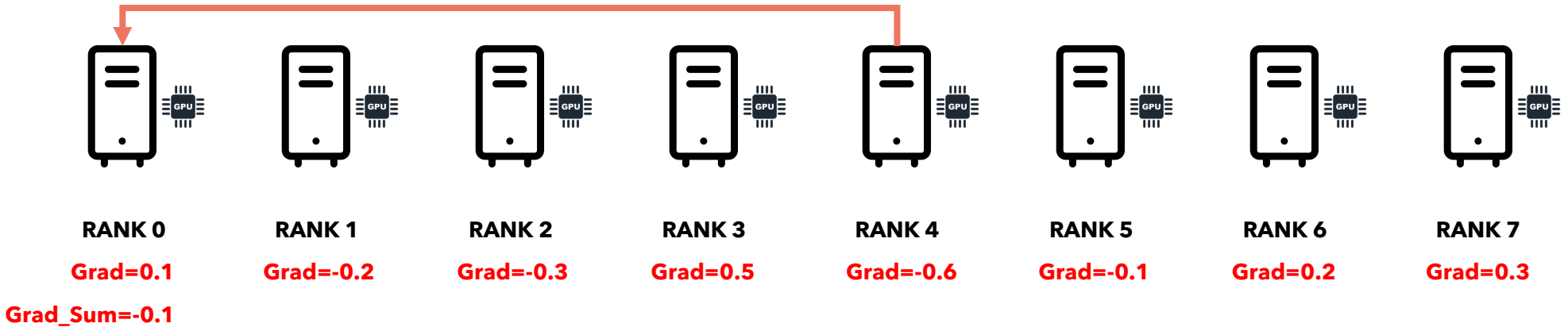
Step = 1

Collective Communication: Reduce



Step = 2

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

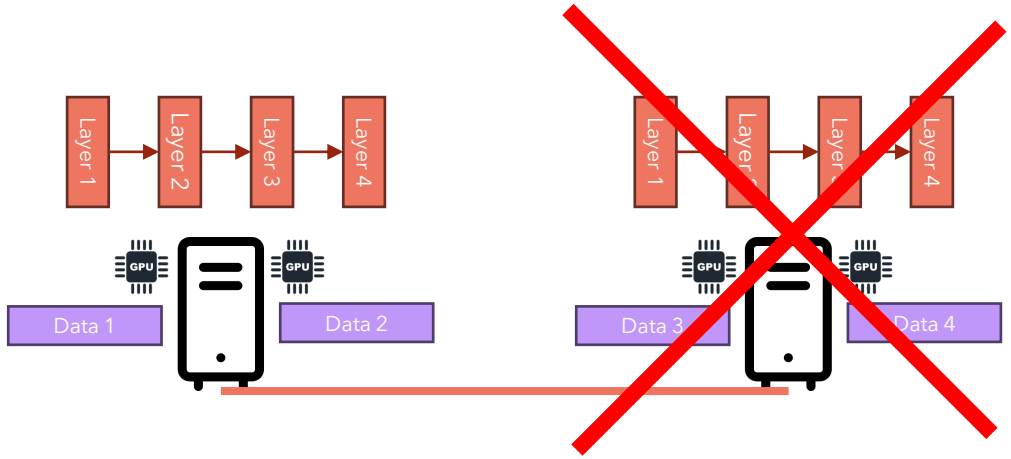
Having accumulated the gradients of all the nodes into a single node, we need to send the cumulative gradient to all the nodes. This operation can be done using a **Broadcast** operator.

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?

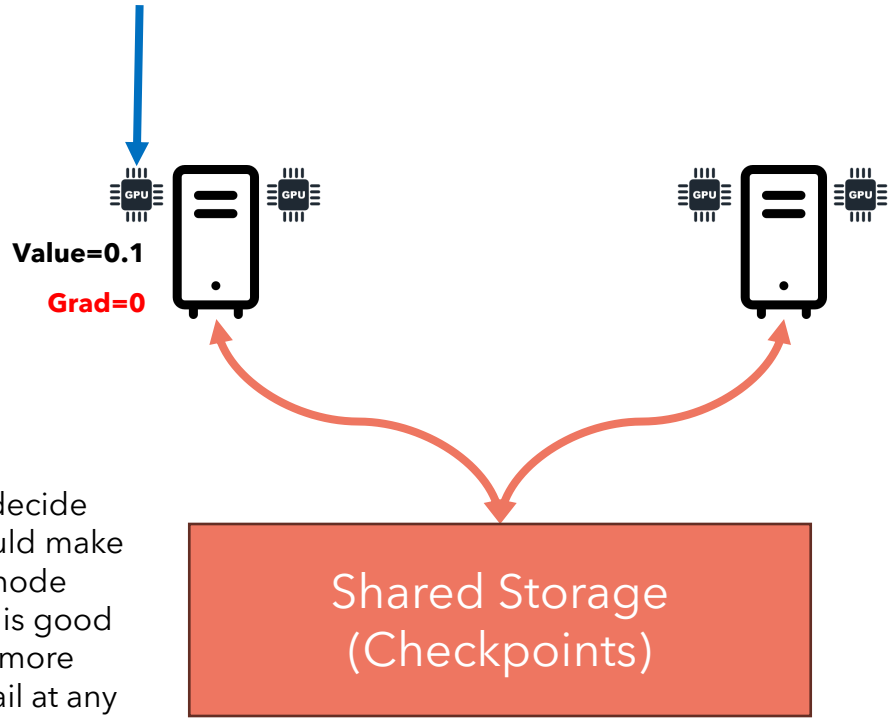


One way, would be to restart the entire cluster and that's easy. However, by restarting the cluster, the training would restart from zero, and we would lose all the parameters and computation done so far. **A better approach is to use checkpointing.**

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

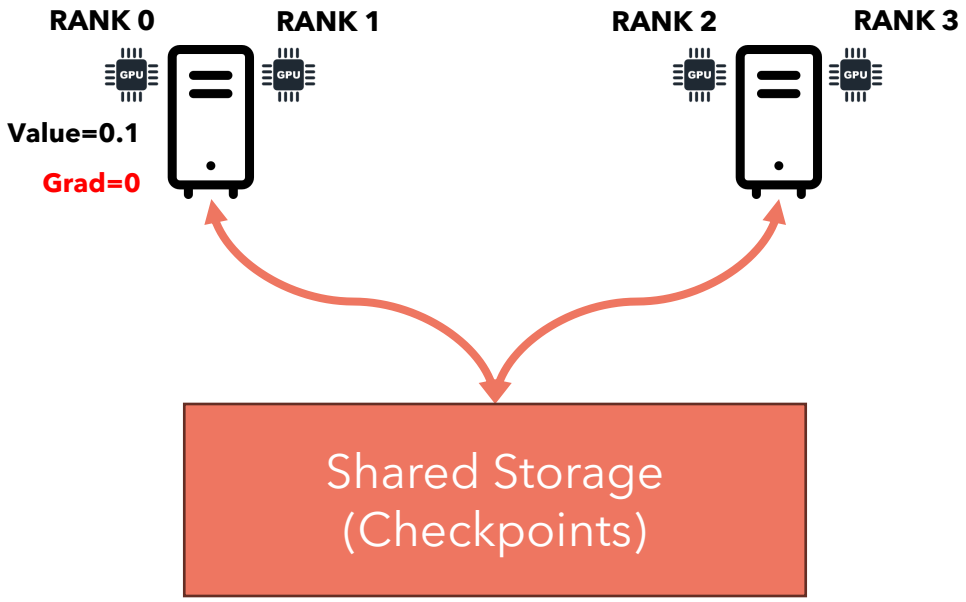
Model weights are initialized here (**using the latest checkpoint**)



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.



Talk is cheap. Show me the ~~code~~ **infrastructure**.

- Linus Torvalds (?)

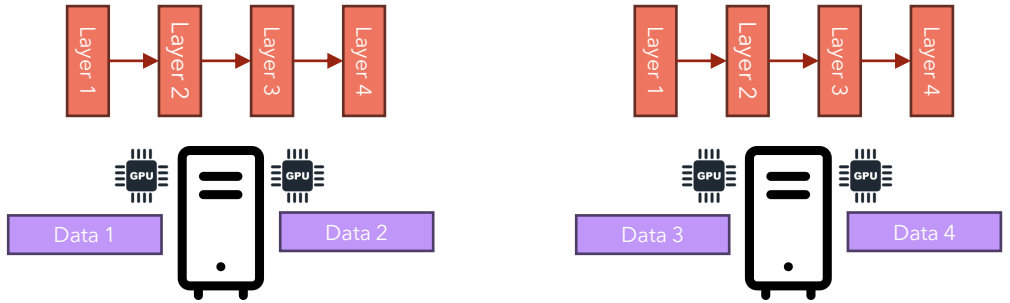
Talk is cheap. Show me the code.

- Linus Torvalds

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()
```

Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

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.

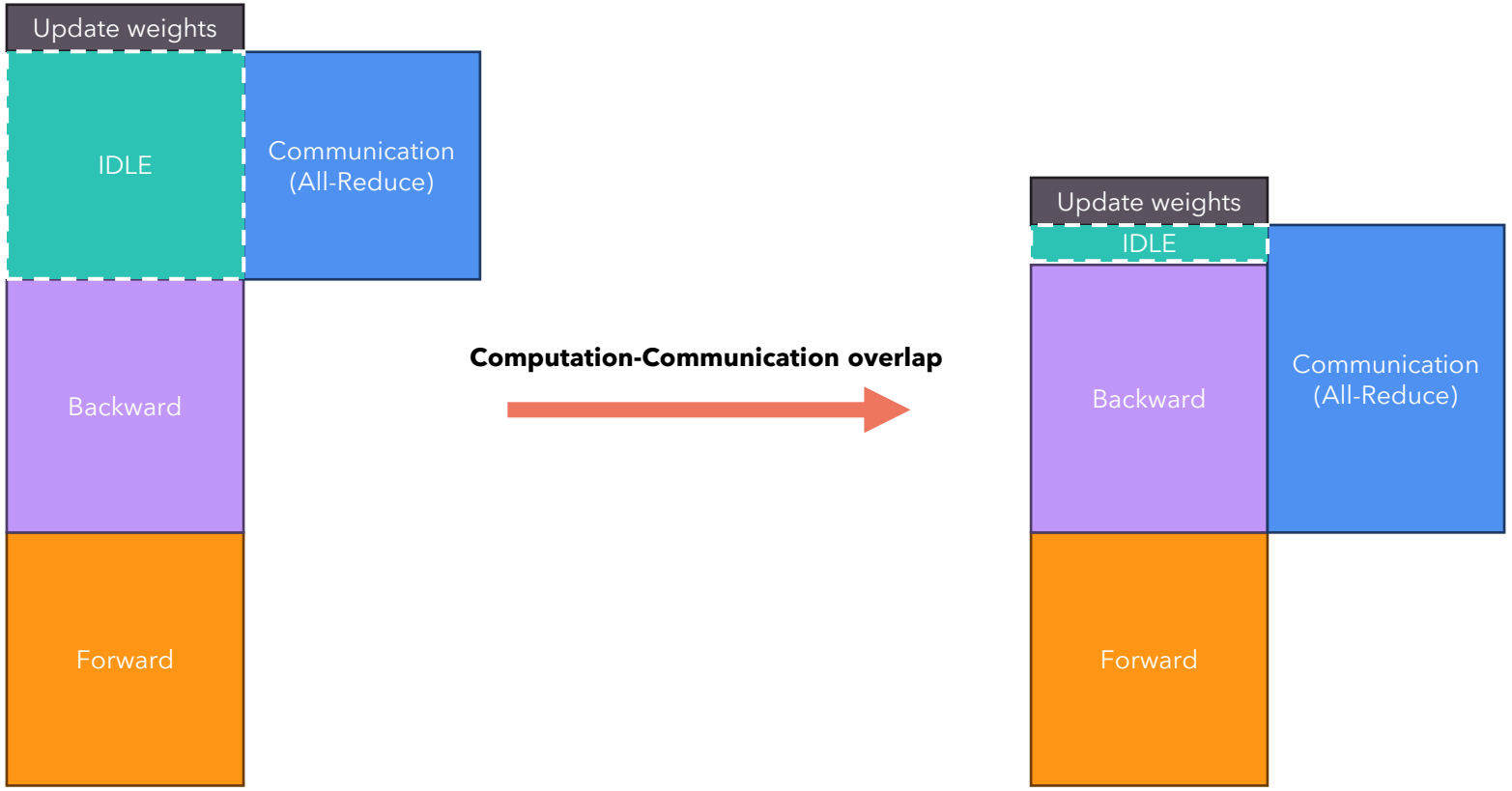
When does PyTorch synchronize gradients?

```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()
```

Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

PyTorch tricks: Computation-Communication overlap

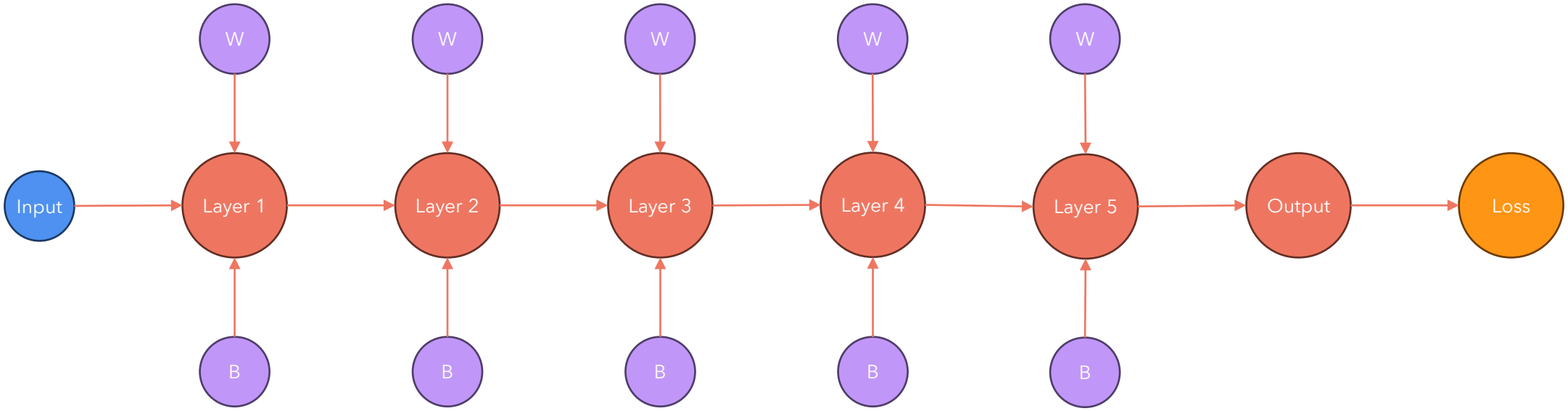
Since each GPU needs to send its gradient to a central node for accumulation, this can lead to an idle time in which the GPUs are not working, but only communicating with each other. PyTorch handles this communication delay in a smart way. Let's see how it works.



Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

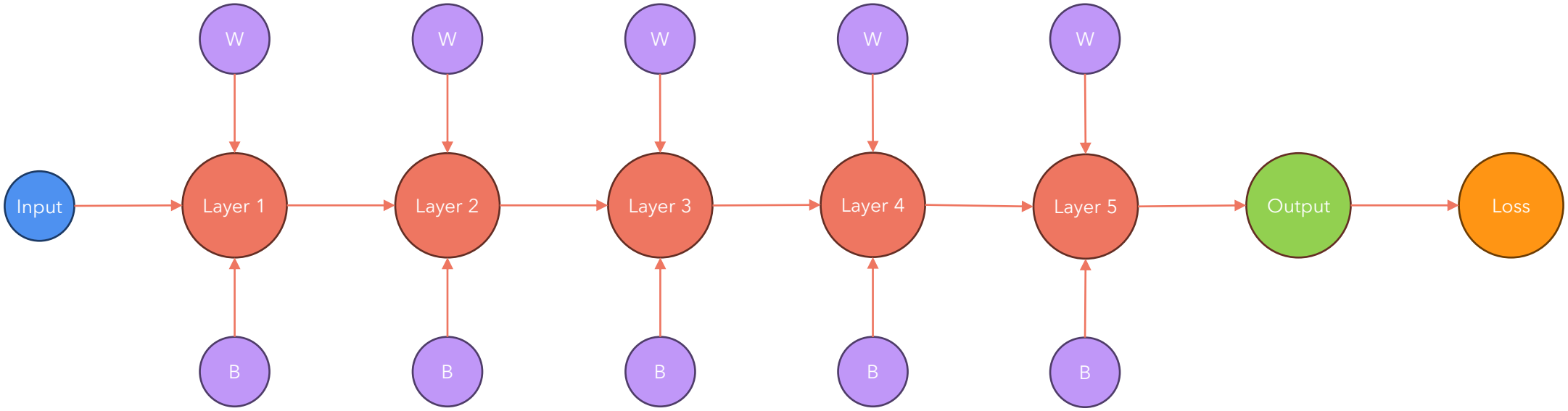
Computation-Communication overlap: details

PyTorch will communicate the gradient of a node as soon as it is available. This way, while the gradient is being computed (from right to left in this diagram), PyTorch is communicating it to the other nodes.



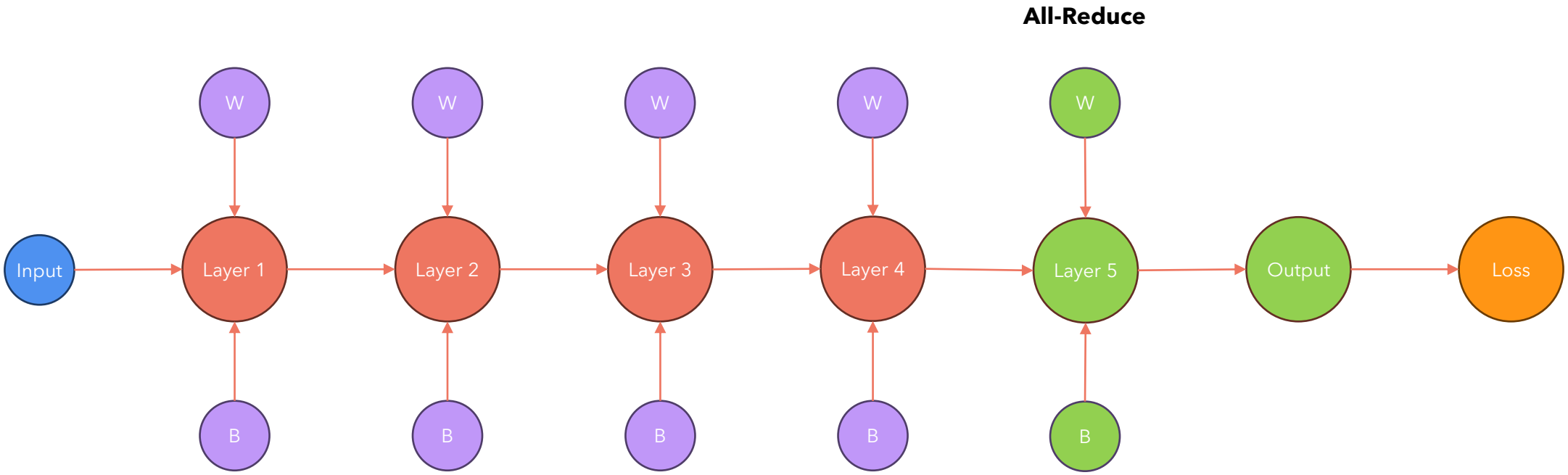
Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

Computation-Communication overlap: details



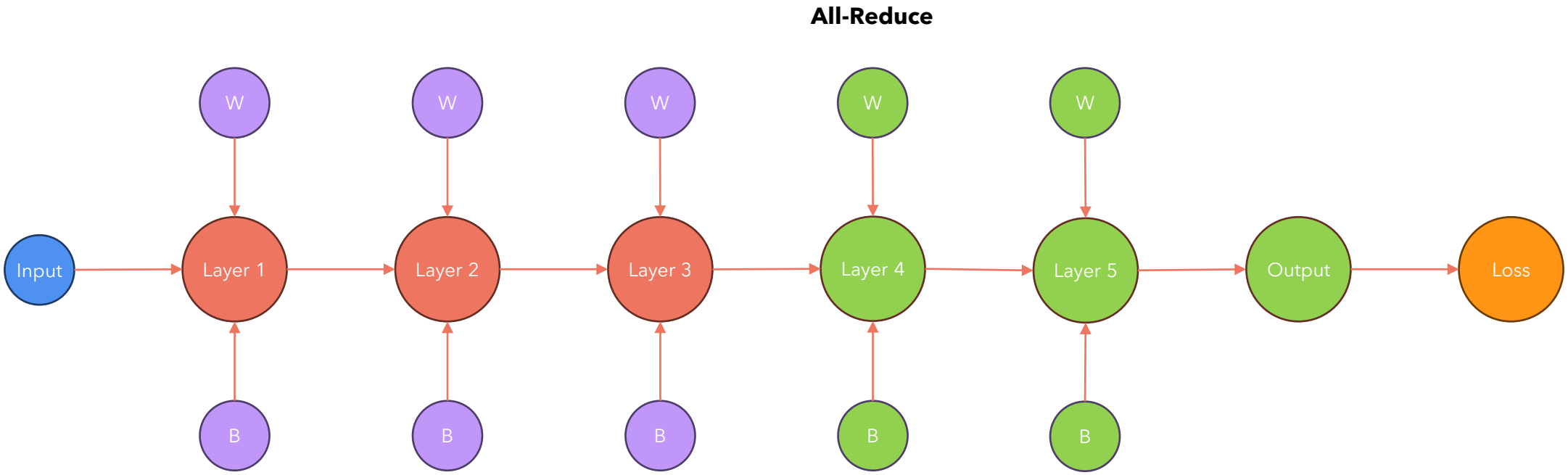
Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

Computation-Communication overlap: details



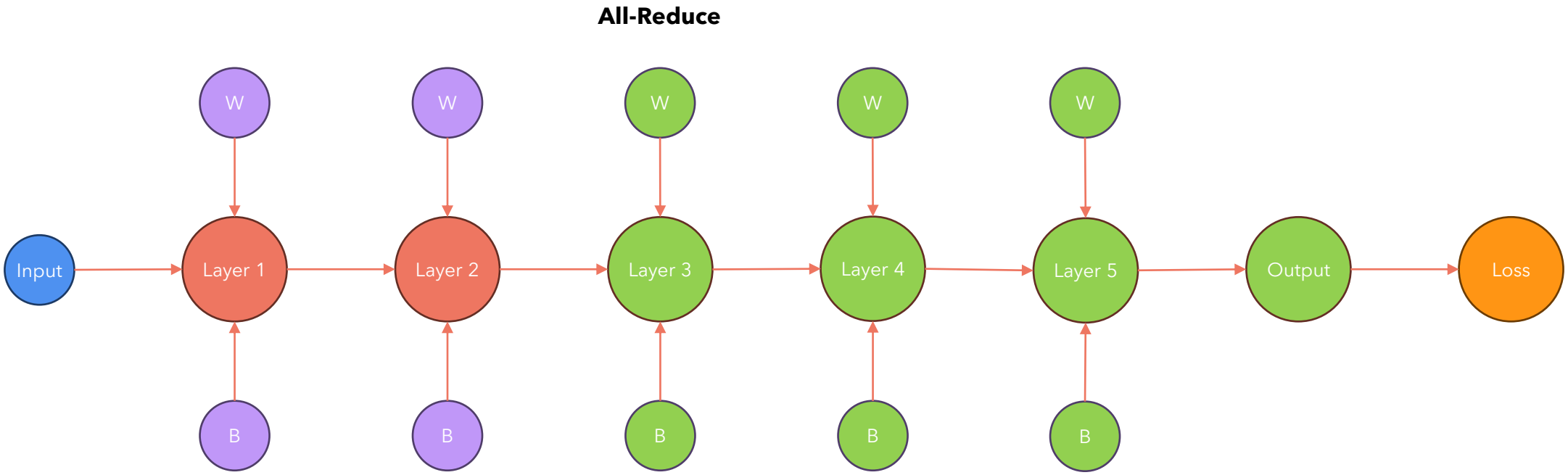
Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

Computation-Communication overlap: details



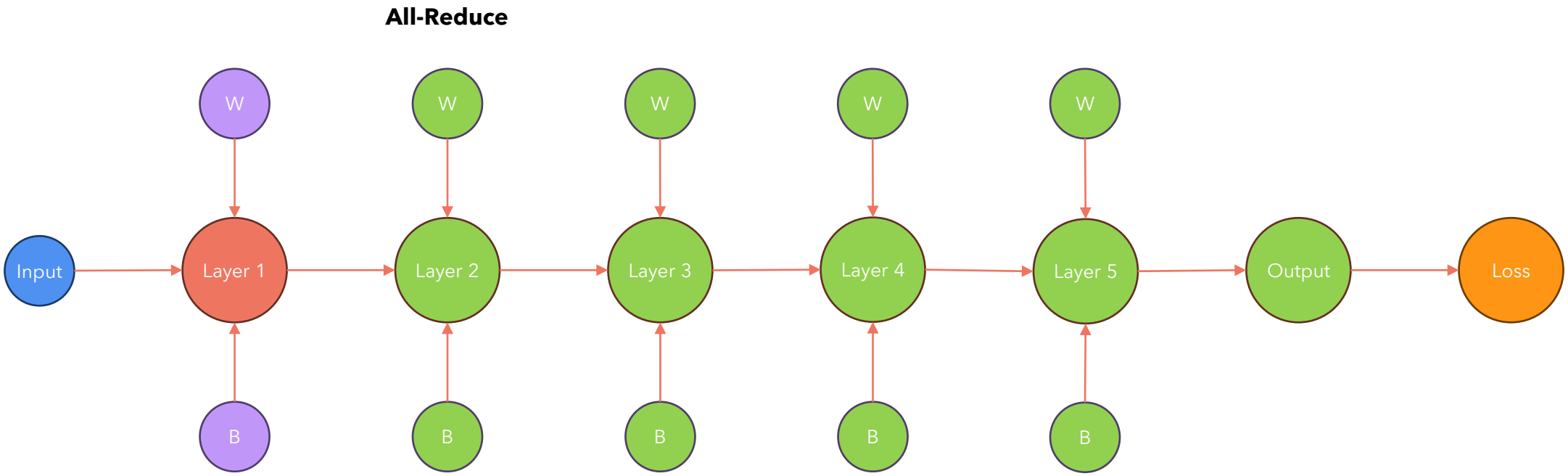
Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

Computation-Communication overlap: details



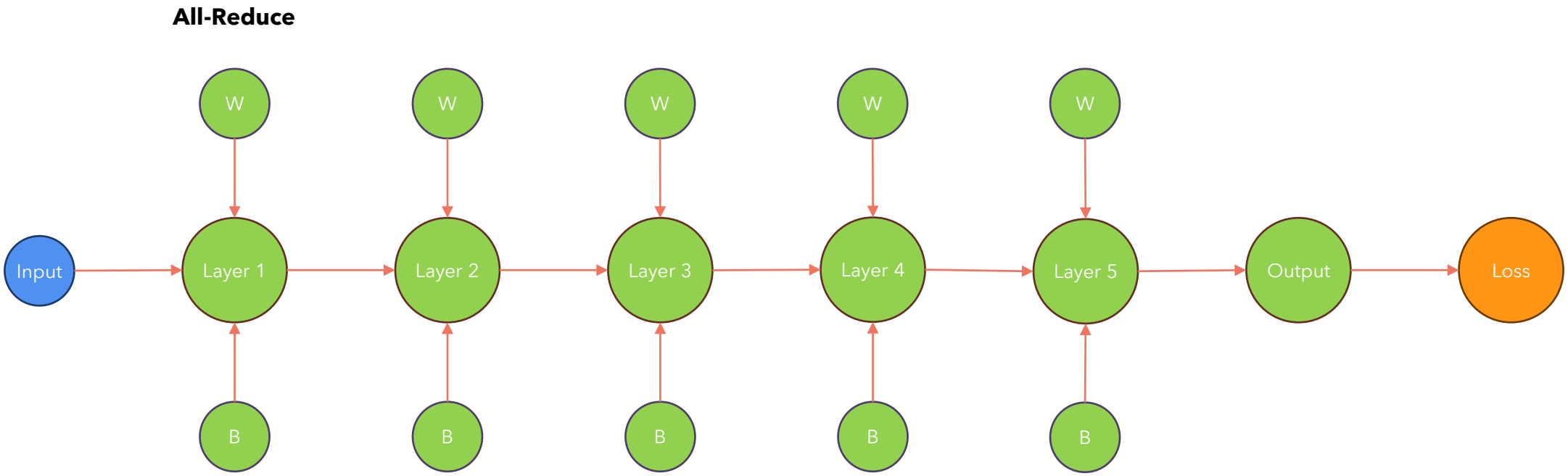
Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

Computation-Communication overlap: details



Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

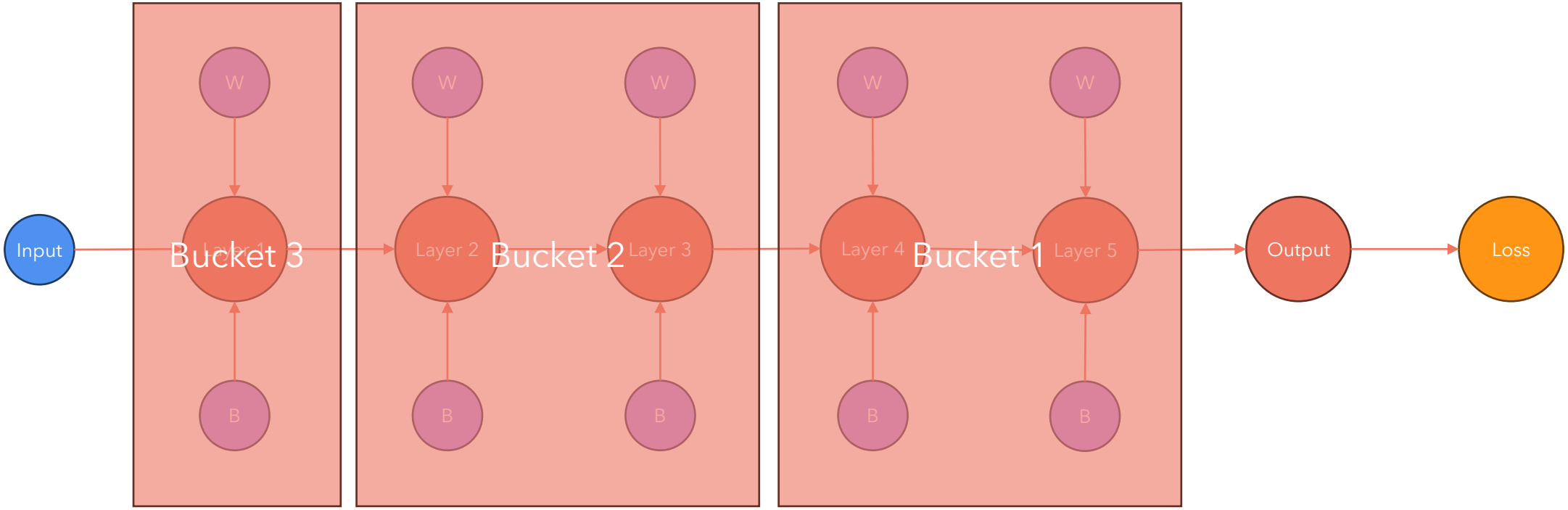
Computation-Communication overlap: details



Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

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.



Umar Jamil – https://github.com/hkproj/pytorch-transformer-distributed

Credits

- GPU icon from flaticon.com
- Computer icon from flaticon.com

Thanks for watching!
Don’t forget to subscribe for more amazing content on AI and Machine Learning!