CS336 Assignment #3 (scaling): : Scaling Laws

Yin Xiaogang

Spring 2025

If you are using LaTeX, you can use `\ifans{}` to type your solutions.
Please tag the questions correctly on Gradescope, otherwise the TAs will take points off if you don't tag questions.

1. Scaling Laws Review (5 points)

(a) (5 points) Problem (chinchilla_isoflops)
Write a script to reproduce the IsoFLOPs method describe above for fitting scaling laws using the final training loss from a set of training runs. For this problem, use the (synthetic) data from training runs given in the file data/isoflops_curves.json. This file contains a JSON array, where each element is an object describing a training run. Here are the first two runs for illustrating the format:

```
[
    {
    "parameters": 49999999,
    "compute_budget": 6e+18,
    "final_loss": 7.192784500319437
    },
    {
    "parameters": 78730505,
    "compute_budget": 6e+18,
    "final_loss": 6.750171320661809
    },
    ...
]
```

For fitting the scaling laws, the scipy package (and scipy.optimize.curve_fit in particular) might be useful, but youre welcome to use any curve fitting method youd like. While Hoffmann et al. [2022] fits a quadratic function to each IsoFLOP profile to find its minimum, we instead recommend you simply take the run with the lowest training loss for each compute budget as the minimum.

i. Show your extrapolated compute-optimal model size, together with the $\langle C_i, N_{opt}(C_i)\rangle$ points you obtained. What is your predicted optimal model size for a budget of 10^{23} FLOPs? What about for 10^{24} FLOPs?
Deliverable: A plot showing your scaling law for model size by compute budget, showing the data points used to fit the scaling law and extrapolating up to at least 10^{24} FLOPs. Then, a one-sentence response with your predicted optimal model size.
Solution: My predicted optimal model size for a budget of 10^{23} FLOPs is 39,792,556,129, and 85,491,178,611 for 10^{24} FLOPs.
I have obtained the following formula using scipy.optimize.curve_fit.

$$\begin{aligned} N &= a \cdot C^b + c && \text{a=1.350e+03, b=3.253e-01, c=-1.192e+09} \\ &= 1350 \cdot C^{0.3253} - 1.192 \cdot 10^9 \end{aligned}$$

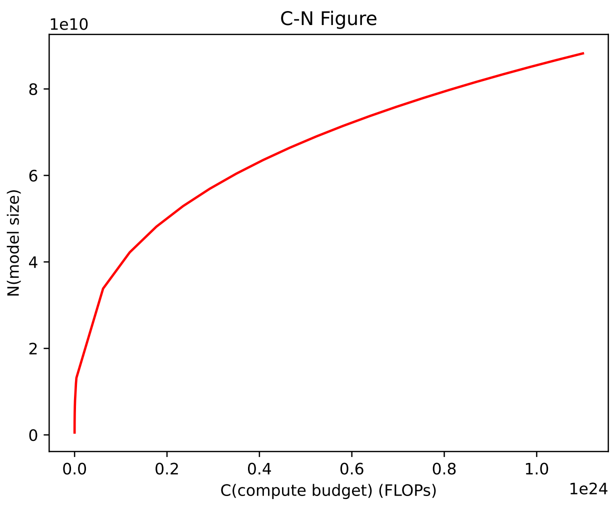

Figure 1: C-N figure

ii. Show your extrapolated compute-optimal dataset size, together with the $\langle C_i, D_{opt}(C_i)\rangle$ data points from the training runs. What is your predicted optimal dataset size for budgets of 10^{23} and 10^{24} FLOPs? **Deliverable**: A plot showing your scaling law for dataset size by compute budget, showing the data points used to fit the scaling law and extrapolating up to at least 10^{24} FLOPs. Then, a one-sentence response with your predicted optimal dataset size.

Solution: My predicted optimal model size for a budget of 10^{23} FLOPs is 364,796,370,806, and 1,525,768,382,027 for 10^{24} FLOPs.

I have obtained the following formula using scipy.optimize.curve_fit.

$$D = a \cdot C^b + c \qquad \text{a=1.802e-03, b=6.220e-01, c=5.948e+08}$$
$$= 0.001802 \cdot C^{0.6220} + 5.948 \cdot 10^8$$

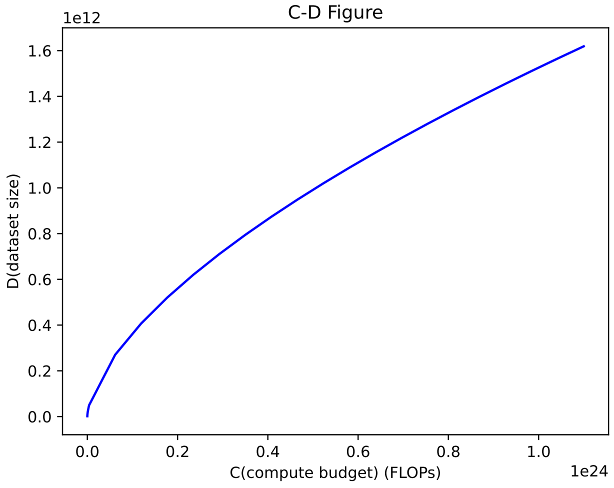

Figure 2: C-D figure

2. Distributed Data Parallel Training (42 points)

(a) (5 points) **Problem (distributed_communication_single_node)**

Write a script to benchmark the runtime of the all-reduce operation in the single-node multi-process setup. The example code above may provide a reasonable starting point. Experiment with varying the following settings:

Backend + device type: Gloo + CPU, NCCL + GPU.
all-reduce data size: float32 data tensors ranging over 1MB, 10MB, 100MB, 1GB.
Number of processes: 2, 4, or 6 processes.
Resource requirements: Up to 6 GPUs. Each benchmarking run should take less than 5 minutes.

Deliverable: Plot(s) and/or table(s) comparing the various settings, with 2-3 sentences of commentary about your results and thoughts about how the various factors interact.

Solution: On MacBook pro 2020, use Gloo + CPU, Use

data size	world size	run time
1M	2	0.0033
1M	4	0.0019
1M	6	0.0048
10M	2	0.0112
10M	4	0.0105
10M	6	0.0113
100M	2	0.0795
100M	4	0.1029
100M	6	0.1302
1G	2	0.7987
1G	4	1.0348
1G	6	1.2857

Increasing the number of processes does not necessarily reduce the runtime, especially when dealing with large amounts of data such as 1GB. Adding more processes may even increase the allreduce runtime, possibly due to the significant time consumed by data transmission between processes. About NCCL + GPU, I have not enough gpus nows.

(b) (5 points) **Problem (naive_ddp)**

Deliverable: Write a script to naively perform distributed data parallel training by all-reducing individual parameter gradients after the backward pass. To verify the correctness of your DDP implementation, use it to train a small toy model on randomly-generated data and verify that its weights match the results from single-process training.

Solution: cs336_systems/naive_ddp.py

(c) (3 points) **Problem (naive_ddp_benchmarking)**

In this nave DDP implementation, parameters are individually all-reduced across ranks after each backward pass. To better understand the overhead of data parallel training, create a script to benchmark your previously-implemented language model when trained with this nave implementation of DDP. Measure the total time per training step and the proportion of time spent on communicating gradients. Collect measurements in the single-node setting (1 node x 2 GPUs) for the XL model size described in §1.1.2.

Deliverable: A description of your benchmarking setup, along with the measured time per training iteration and time spent communicating gradients for each setting.

Solution: cs336_systems/naive_ddp_benchmarking.py. Using Gloo + CPU and small model size, in my Macbook pro 2020, The the measured time per training iteration is 14.94s, the time spent communicating gradients for each setting is 0.6997, about 4.68%

```
3, total time: 14.948427969999988, reduce time: 0.6959458242000209
1, total time: 14.94459973039999, reduce time: 0.7210226140000031
2, total time: 14.9476707444, reduce time: 0.6712903046000065
0, total time: 14.927452500999982, reduce time: 0.7106147133999798
```

(d) (2 points) **Problem (minimal_ddp_flat_benchmarking)** Modify your minimal DDP implementation to communicate a tensor with flattened gradients from all parameters. Compare its performance with the minimal DDP implementation that issues an allreduce for each parameter tensor under the previously-used conditions (1 node x 2 GPUs, XL model size as described in §1.1.2).

Deliverable: The measured time per training iteration and time spent communicating gradients under distributed data parallel training with a single batched all-reduce call. 1-2 sentences comparing the results when batching vs. individually communicating gradients.

Solution: cs336_systems/minimal_ddp_flat_benchmarking.py. Using Gloo + CPU and small model size, in my Macbook pro 2020, The the measured time per training iteration is 23.3792s, the time spent communicating gradients for each setting is 1.51746, about 6.49%. Because small model size and cpu, Using flat increase the overhead.

```
2, total time: 23.38606446599988, reduce time: 1.5045228166001834
1, total time: 23.372208501200113, reduce time: 1.5014608585999667
3, total time: 23.36522756079994, reduce time: 1.580048452200208
0, total time: 23.39337175940018, reduce time: 1.483804502199746
```

(e) (5 points) **Problem (ddp_overlap_individual_parameters)** Implement a Python class to handle distributed data parallel training. The class should wrap an arbitrary PyTorch nn.Module and take care of broadcasting the weights before training (so all ranks have the same initial parameters) and issuing communication calls for gradient averaging. We recommend the following public interface:

```
def __init__(self, module: torch.nn.Module):
"""
Given an instantiated PyTorch nn.Module to be
parallelized, construct a DDP container that will handle gradient synchronization across ranks.
"""
def forward(self, *inputs, **kwargs):
"""
Calls the wrapped moduleâĂŹs forward() method with the provided positional and keyword arguments.
"""
def finish_gradient_synchronization(self):
"""
When called, wait for asynchronous communication calls to be queued on GPU.
"""
```

To use this class to perform distributed training, well pass it a module to wrap, and then add a call to finish_gradient_synchronization() before we run optimizer.step() to ensure that the optimizer step, an operation that depends on the gradients, may be queued:

```
model = ToyModel().to(device)
ddp_model = DDP(model)
for _ in range(train_steps):
    x, y = get_batch()
    logits = ddp_model(x)
    loss = loss_fn(logits, y)
    loss.backward()
    ddp_model.finish_gradient_synchronization()
    optimizer.step()
```

Deliverable: Implement a container class to handle distributed data parallel training. This class should overlap gradient communication and the computation of the backward pass. To test your DDP class, first implement the adapters [adapters.get_ddp_individual_parameters] and [adapters.ddp_individual_parameters_on_after_backward] (the latter is optional, depending on your implementation you may not need it).

Then, to execute the tests, run uv run pytest tests/test_ddp_individual_parameters.py. We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

Solution: cs336_systems/ddp.py

(f) (1 point) **Problem (ddp_overlap_individual_parameters_benchmarking)**

i. Benchmark the performance of your DDP implementation when overlapping backward pass computation with communication of individual parameter gradients. Compare its performance with our previously-studied settings (the minimal DDP implementation that either issues an all-reduce for each parameter tensor, or a single all-reduce on the concatenation of all parameter tensors) with the same setup: 1 node, 2 GPUs, and the XL model size described in §1.1.2.

Deliverable: The measured time per training iteration when overlapping the backward pass with communication of individual parameter gradients, with 1-2 sentences comparing the results.

Solution: cs336_systems/ddp_overlap_individual_parameters_benchmarking.py. Using Gloo + CPU and small model size, in my Macbook pro 2020, The the measured time per training iteration is 15.4635s, the time spent communicating gradients for each setting is 0.736289 s, about 4.76147 % . The time spent communicating gradients ...

```
3, total time: 15.46724196474861, reduce time: 0.7726576314998965
2, total time: 15.461637291249644, reduce time: 0.7224488822494095
1, total time: 15.46450111025024, reduce time: 0.7475921859995651
0, total time: 15.460556565501065, reduce time: 0.7024589765005658
```

ii. Instrument your benchmarking code (using the 1 node, 2 GPUs, XL model size setup) with the Nsight profiler, comparing between the initial DDP implementation and this DDP implementation that overlaps backward computation and communication. Visually compare the two traces, and provide a profiler screenshot demonstrating that one implementation overlaps compute with communication while the other doesnt.

Deliverable: 2 screenshots (one from the initial DDP implementation, and another from this DDP implementation that overlaps compute with communication) that visually show that communication is or isnt overlapped with the backward pass.

Solution: to be done. I do not have 2 gpus now.

(g) (8 points) **Problem (ddp_overlap_bucketed)** Implement a Python class to handle distributed data parallel training, using gradient bucketing to improve communication efficiency. The class should wrap an arbitrary input PyTorch nn.Module and take care of broadcasting the weights before training (so all ranks have the same initial parameters) and issuing bucketed communication calls for gradient averaging. We recommend the following interface:

```
def __init__(self, module: torch.nn.Module, bucket_size_mb: float):
"""
Given an instantiated
PyTorch nn.Module to be parallelized, construct a DDP container that will handle gradient synchro
across ranks. Gradient synchronization should be bucketed, with each bucket holding
at most bucket_size_mb of parameters.
"""
def forward(self, *inputs, **kwargs):
"""
Calls the wrapped moduleâĂŹs forward() method with the
provided positional and keyword arguments.
```

```
        """
        def finish_gradient_synchronization(self):
        """
        When called, wait for asynchronous communication
        calls to be queued on GPU.
        """
```

Beyond the addition of a bucket_size_mb initialization parameter, this public interface matches the interface of our previous DDP implementation that individually communicated each parameter. We suggest allocating parameters to buckets using the reverse order of model.parameters(), since the gradients will become ready in approximately that order during the backward pass.

Deliverable: Implement a container class to handle distributed data parallel training. This class should overlap gradient communication and the computation of the backward pass. Gradient communication should be bucketed, to reduce the total number of communication calls. To test your implementation, complete [adapters.get_ddp_bucketed], [adapters.ddp_bucketed_on_after_backward], and [adapters.ddp_bucketed_on_train_batch_start] (the latter two are optional, depending on your implementation you may not need them).

Then, to execute the tests, run pytest tests/test_ddp.py. We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

Solution: cs336_systems/ddp.py

(h) (3 points) **Problem (ddp_bucketed_benchmarking)**

i. Benchmark your bucketed DDP implementation using the same config as the previous experiments (1 node, 2 GPUs, XL model size), varying the maximum bucket size (1, 10, 100, 1000 MB). Compare your results to the previous experiments without bucketingdo the results align with your expectations? If they dont align, why not? You may have to use the PyTorch profiler as necessary to better understand how communication calls are ordered and/or executed. What changes in the experimental setup would you expect to yield results that are aligned with your expectations?

Deliverable: Measured time per training iteration for various bucket sizes. 3-4 sentence commentary about the results, your expectations, and potential reasons for any mismatch.

Solution: cs336_systems/ddp_bucketed_benchmarking.py. Using Gloo + CPU and small model size, in my Macbook pro 2020, The the measured time per training iteration is 15.3479s, the time spent communicating gradients for each setting is 0.738696s, about 4.81301% . Because of cpu, no comment.

```
        3, total time: 15.350618762498925, reduce time: 0.7534022687495963
        2, total time: 15.347716027498791, reduce time: 0.7388711377498112
        1, total time: 15.342720182749872, reduce time: 0.7461342754995712
        0, total time: 15.350642050000715, reduce time: 0.716377919748993
```

ii. Assume that the time it takes to compute the gradients for a bucket is identical to the time it takes to communicate the gradient buckets. Write an equation that models the communication overhead of DDP (i.e., the amount of additional time spent after the backward pass) as a function 31 of the total size (bytes) of the model parameters (s), the all-reduce algorithm bandwidth (w, computed as the size of each ranks data divided by the time it takes to finish the all-reduce), the overhead (seconds) associated with each communication call (o), and the number of buckets (n_b). From this equation, write an equation for the optimal bucket size that minimizes DDP overhead.

Deliverable: Equation that models DDP overhead, and an equation for the optimal bucket size.

Solution:

1 DDP Communication Overhead Equation (Equal Compute/Comm Time per Bucket)

1.1 Core Assumptions

1. Time to compute gradients for one bucket = Time to communicate one bucket: $t_{\text{comp}} = t_{\text{comm}} = \frac{s_b}{w}$
2. Total model parameter size: s (bytes); number of buckets: n_b; bucket size: $s_b = \frac{s}{n_b}$
3. All-reduce bandwidth: $w = \frac{\text{per-rank data size}}{\text{communication time}}$ (bytes/sec)
4. Fixed overhead per communication call: o (seconds/call, e.g., network setup/handshake)

1.2 Compute-Communication Overlap Logic

- **1st bucket**: No prior overlap, sequential execution of compute + communication + fixed overhead: $t_{\text{comp}} + t_{\text{comm}} + o$
- **Remaining $n_b - 1$ buckets**: Full overlap (compute next bucket while communicating current one). Only the maximum of compute/comm time (equal here) plus fixed overhead is added: $t_{\text{comm}} + o$

1.3 Derivation of Total Overhead Equation

Substitute $s_b = \frac{s}{n_b}$ into the overhead components and simplify:

$$\begin{aligned} T(n_b) &= \left(\frac{s_b}{w} + \frac{s_b}{w} + o\right) + (n_b - 1)\left(\frac{s_b}{w} + o\right) \\ &= \frac{2s}{n_b \cdot w} + o + \frac{(n_b-1)s}{n_b \cdot w} + (n_b - 1)o \\ &= \frac{s(n_b+1)}{n_b \cdot w} + n_b \cdot o \end{aligned}$$

Exact DDP Communication Overhead Equation:

$$\boxed{T(n_b) = \frac{s(n_b+1)}{n_b \cdot w} + n_b \cdot o}$$

For large n_b ($n_b \gg 1$), $n_b + 1 \approx n_b$, and the equation simplifies to:

$$T(n_b) \approx \frac{s}{w} + n_b \cdot o$$

2 Optimal Bucket Size Derivation (Minimize Overhead)

2.1 Rewrite Overhead as a Function of Bucket Size s_b

Using $n_b = \frac{s}{s_b}$, substitute into the exact overhead equation:

$$\begin{aligned} T(s_b) &= \frac{s\left(\frac{s}{s_b} + 1\right)}{\frac{s}{s_b} \cdot w} + \frac{s}{s_b} \cdot o \\ &= \frac{s + s_b}{w} + \frac{s \cdot o}{s_b} \end{aligned}$$

2.2 Find Minimum via Calculus

Take the first derivative of $T(s_b)$ with respect to s_b and set it to 0:

$$\frac{dT}{ds_b} = \frac{1}{w} - \frac{s \cdot o}{s_b^2} = 0$$

Solve for s_b to get the optimal bucket size:

$$\boxed{s_b^* = \sqrt{s \cdot o \cdot w}}$$

2.3 Verify Global Minimum

Take the second derivative of $T(s_b)$:

$$\frac{d^2T}{ds_b^2} = \frac{2s \cdot o}{s_b^3} > 0$$

The positive second derivative confirms that s_b^* is the **global minimum**.

3 Key Conclusions

1. The exact DDP communication overhead under equal compute/comm time per bucket is $T(n_b) = \frac{s(n_b+1)}{n_b \cdot w} + n_b \cdot o$. 2. The theoretically optimal bucket size is $s_b^* = \sqrt{s \cdot o \cdot w}$, which balances fixed communication overhead and bucket-level parallel efficiency.

(i) (10 points) **Problem (communication_accounting)**

Consider a new model config, XXL, with d_model=16384, d_ff=53248, and num_blocks=126. Because for very large models, the vast majority of FLOPs are in the feedforward networks, we make some simplifying assumptions. First, we omit attention, input embeddings, and output linear layers. Then, we assume that each FFN is simply two linear layers (ignoring the activation function), where the first has input size d_model and output size d_ff, and the second has input size d_ff and output size d_model. Your model consists of num_blocks blocks of these two linear layers. Don't do any activation checkpointing, and keep your activations and gradient communications in BF16, while your accumulated gradients, master weights and optimizer state should be in FP32.

i. How much memory would it take to store the master model weights, accumulated gradients and optimizer states in FP32 on a single device? How much memory is saved for backward (these will be in BF16)? How many H100 80GB GPUs worth of memory is this?
Deliverable: Your calculations and a one-sentence response.
Solution:
- 126 * 2 * 53248 * 16384 * 4 * 5 /1024/1024/1024 = 4095G.
- 4095 * 6 /20 = 1228.5G is saved for backward (these will be in BF16).
- 1228.5/80 ≈ 16 H100 80GB GPUs worth of memory.

ii. Now assume your master weights, optimizer state, gradients and half of your activations (in practice every second layer) are sharded across N_{FSDP} devices. Write an expression for how much memory this would take per device. What value does N_{FSDP} need to be for the total memory cost to be less than 1 v5p TPU (95GB per device)? **Deliverable**: Your calculations and a one-sentence response.
Solution: The parameter number $\psi = 126 * 2 * 53248 * 16384/1024/1024/1024 = 204.75G$, so $N_{\text{FSDP}} = ceil((\psi * K + seq\_len * batch\_size * 53248 * 126)/95)$

iii. Consider only the forward pass. Use the communication bandwidth of $W_{\text{ici}} = 2 \cdot 9 \cdot 10^{10}$ and FLOPS/s of $C = 4.6 \cdot 10^{14}$ for TPU v5p as given in the TPU Scaling Book. Following the notation of the Scaling Book, use $M_X = 2$, $M_Y = 1$ (a 3D mesh), with X = 16 being your FSDP dimension, and Y = 4 being your TP dimension. At what per-device batch size is this model compute bound? What is the overall batch size in this setting?
Deliverable: Your calculations and a one-sentence response.
Solution: As Part 5 of the TPU Scaling Book, let $\alpha \equiv C/W_{\text{ici}} = 4.6{\cdot}10^{14}/2{\cdot}9{\cdot}10^{10} = 2550$,the ICI arithmetic intensity.
$N = XY = 64$.We are compute bound, we require

$$\frac{B}{N} > \frac{\alpha^2}{M_X M_Y F}$$

so

$$B > N \cdot \frac{\alpha^2}{M_X M_Y F} = 64 \cdot \frac{2550^2}{2 \cdot 1 \cdot 53248} = 3907.75$$

so B/X = 3908/16 = 245 per-device batch size is this model compute bound. 3908 is the overall batch size in this setting.

iv. In practice, we want the overall batch size to be as small as possible, and we also always use our compute effectively (in other words we want to never be communication bound). What other tricks can we employ to reduce the batch size of our model but retain high throughput?
Deliverable: A one-paragraph response. Back up your claims with references and/or equations.
Solution: To reduce overall batch size while retaining high throughput and avoiding communication-bound bottlenecks (beyond DP, FSDP, TP, PP, and EP), a suite of hardware, algorithmic, and engineering optimizations can be integrated: first, optimize hardware mesh configuration by aligning mesh axes with model layer dimensions (e.g., binding Transformer Attention layers to individual GPUs/TPUs) to balance small-batch data distribution and minimize idle devices, while enabling BF16/FP16 mixed precision and Tensor Core utilization to maximize compute density for small batches; second, adopt gradient accumulation to simulate large effective batches (via accumulating gradients from multiple small-batch iterations before parameter updates) and configure distributed frameworks like FSDP with gradient_as_bucket_view=True and limit_all_gathers=True to reduce redundant gradient synchronization overhead, paired with efficient optimizers (e.g., AdamW) and minimal warmup schedules to accelerate convergence and cut total training iterations; third, implement asynchronous data prefetching with persistent_workers and prefetch_factor in dataloaders to eliminate small-batch data loading latency, overlap compute and communication (e.g., async AllReduce with NCCL/XLA) to hide communication costs, and fuse adjacent model layers (e.g., Linear+LayerNorm+GELU via TorchScript or TensorRT) to reduce kernel launch overheadall of which collectively ensure that small-batch training leverages hardware efficiently without sacrificing throughput or model performance.

3. Optimizer State Sharding (20 points)

(a) (15 points) **Problem (optimizer_state_sharding)**

Implement a Python class to handle optimizer state sharding. The class should wrap an arbitrary input PyTorch **optim.Optimizer** and take care of synchronizing updated parameters after each optimizer step. We recommend the following public interface:

```
        def __init__(self, params, optimizer_cls: Type[Optimizer], **kwargs: Any):
        r"""
        Initializes the
        sharded state optimizer. params is a collection of parameters to be optimized (or parameter
        groups, in case the user wants to use different hyperparameters, such as learning rates, for diff
        parts of the model); these parameters will be sharded across all the ranks. The optimizer_cls
        parameter specifies the type of optimizer to be wrapped (e.g., optim.AdamW). Finally, any remaini
        keyword arguments are forwarded to the constructor of the optimizer_cls. Make sure to
        call the torch.optim.Optimizer super-class constructor in this method.
        """

        def step(self, closure, **kwargs):
        r"""
        Calls the wrapped optimizer s step() method with the provided
        closure and keyword arguments. After updating the parameters, synchronize with the other
        ranks.
        """

        def add_param_group(self, param_group: dict[str, Any]):
        r"""
        This method should add a parameter
        group to the sharded optimizer. This is called during construction of the sharded optimizer by
        the super-class constructor and may also be called during training (e.g., for gradually unfreezin
        layers in a model). As a result, this method should handle assigning the model s parameters
        among the ranks.
        """
```

Deliverable: Implement a container class to handle optimizer state sharding. To test your sharded optimizer, first implement the adapter [adapters.get_sharded_optimizer]. Then, to execute the tests, run uv run pytest tests/test_sharded_optimizer.py. We recommend running the tests multiple times (e.g., 5) to ensure that it passes reliably.

Solution: cs336_systems/ddp.py, class ShardedOptimizer which maybe need to be repaired.

(b) (5 points) **Problem ((optimizer_state_sharding_accounting)**

i. Create a script to profile the peak memory usage when training language models with and without optimizer state sharding. Using the standard configuration (1 node, 2 GPUs, XL model size), report the peak memory usage after model initialization, directly before the optimizer step, and directly after the optimizer step. Do the results align with your expectations? Break down the memory usage in each setting (e.g., how much memory for parameters, how much for optimizer states, etc.).

   Deliverable: 2-3 sentence response with peak memory usage results and a breakdown of how the memory is divided between different model and optimizer components.

   Solution: to be done

ii. How does our implementation of optimizer state sharding affect training speed? Measure the time taken per iteration with and without optimizer state sharding for the standard configuration (1 node, 2 GPUs, XL model size).

   Deliverable: 2-3 sentence response with your timings.

   Solution: to be done

iii. How does our approach to optimizer state sharding differ from ZeRO stage 1 (described as ZeRODP Pos in Rajbhandari et al., 2020)?

Deliverable: 2-3 sentence summary of any differences, especially those related to memory and communication volume.

Solution: We ignore "ReduceScatter the gradients", we shard the optimizer state by the number, not by size.(to be done)

Submission Instructions

You shall submit this assignment on GradeScope as two submissions – one for “Assignment 2 [coding]” and another for ‘Assignment 2 [written]”:

1. Run the `collect_submission.sh` script to produce your `assignment2.zip` file.
2. Upload your `assignment2.zip` file to GradeScope to “Assignment 2 [coding]”.
3. Upload your written solutions to GradeScope to “Assignment 2 [written]”.