Homework 3

Task 1

Задание: Необходимо написать функцию, которую будут запускать N процессов и которая будет печать последовательно номера процессов num_iter раз.

Решение:

Во-первых, проходим в цикле по итерациям, в конце каждой итерации нулевой процесс печатает сепаратор.

Далее внутри внешнего цикла идет еще один цикл, который проходит по всем номерам процессов и в конце каждой итерации ставит барьер. Таким образом, все процессы ждут пока нужный процесс напечатает свой номер.

````
def run_sequential(rank, size, num_iter=10):
    """
    Prints the process rank sequentially according to its number over `num_iter` ite
    separating the output for each iteration by `---`
    Example (3 processes, num_iter=2):
    ```
    Process 0
    Process 1
    Process 2
    ---
    Process 0
    Process 1
````
````

````
    Process 2
    ```
    """

    for i in range(num_iter):
        for j in range(size):
            if rank == j:
                print(f"Process {j}")
            dist.barrier()
        if rank == 0 and i != num_iter - 1:
            print("---")
````

Вывод для команды `torchrun --nproc_per_node 10 sequential_print.py` и *`num_iter`* `=4` :

```
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
Process 8
Process 9
---
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
Process 8
Process 9
---
```
```

```
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
Process 8
Process 9
---
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
Process 8
Process 9
```

Task 2

Для начала я переписал код в файле ddp_cifar100.py:

1. Добавил валидационную выборку и прогон на ней
2. Добавил выбор бэкэнда и девайса
3. Добавил выбор реализации batch norm: кастомный или из torch
4. Добавил параметр accumulation steps

```
parser = argparse.ArgumentParser()
parser.add_argument("--backend", type=str, default="gloo",
                    choices=["gloo", "nccl"])
```

```
parser.add_argument("--device", type=str, default="cpu",
                    choices=["cpu", "cuda"])
parser.add_argument("--batch_norm", type=str, default="custom",
                    choices=["custom", "torch"])
parser.add_argument("--grad_accumulation", type=int, default=1)
args = parser.parse_args()
```

Команда для запуска:

Написание кастомного Batch Norm

Алгоритм получился интуитивным:

1. Во время forward
   a. считаем статистики,
   b. делаем all_reduce,
   c. усредняем статистики,
   d. насчитываем скользящие средние для теста
   e. запоминаем все вышеперечисленное
2. Во время backward:
   a. Опять-таки считаем статистики но уже по градиентам
   b. Если шаг кратен accumulation steps то делаем all reduce
   c. Вычисляем градиент по входу

Для gradient accumulation просто передаю флаг в модель, который показывает нужно ли делать all reduce во время backward.

```
class sync_batch_norm(Function):
    """
    A version of batch normalization that aggregates the activation statistics acros

    This needs to be a custom autograd.Function, because you also need to comm
```

```
on the backward pass (each activation affects all examples, so loss gradients f
the gradient for each activation).

For a quick tutorial on torch.autograd.function, see
https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custo
"""

@staticmethod
def forward(ctx, input, running_mean, running_std, eps: float, momentum: floa
    N, C = input.size(0), input.size(1)

    # Compute local sums along the batch dimension.
    local_sum = input.sum(dim=0)
    local_sum_sq = (input ** 2).sum(dim=0)

    # Pack the local statistics and count into a single tensor.
    # Note: count is stored as a 1-element tensor.
    count_tensor = torch.tensor([float(N)], device=input.device)
    stats = torch.cat([local_sum, local_sum_sq, count_tensor])

    # Aggregate statistics from all processes using a single all-reduce call.
    dist.all_reduce(stats)

    # Unpack the aggregated statistics.
    global_sum = stats[:C]
    global_sum_sq = stats[C:2 * C]
    global_count = stats[2 * C].item()  # Total number of examples across proce

    # Compute global mean and variance.
    global_mean = global_sum / global_count
    global_var = global_sum_sq / global_count - global_mean ** 2
    global_std = torch.sqrt(global_var + eps)

    # Normalize the input using the aggregated statistics.
    normalized = (input - global_mean) / global_std
```

```
        # Update running statistics.
        running_mean.data = running_mean.data * (1 - momentum) + global_mean *
        running_std.data = running_std.data * (1 - momentum) + global_std * momer

        # Save context for backward: input, global_mean, global_std, normalized.
        ctx.save_for_backward(input, global_mean, global_std, normalized)
        ctx.global_count = global_count
        ctx.eps = eps

        ctx.sync_grad = sync_grad

        return normalized

    @staticmethod
    def backward(ctx, grad_output):
        # Retrieve saved tensors and global count.
        input, global_mean, global_std, normalized = ctx.saved_tensors
        N = ctx.global_count
        C = input.size(1)

        # Compute the local sums for gradient statistics.
        grad_sum = grad_output.sum(dim=0)
        grad_mul = (grad_output * (input - global_mean)).sum(dim=0)

        if ctx.sync_grad:
            # Pack the gradient statistics into a single tensor.
            grad_stats = torch.cat([grad_sum, grad_mul])
            # Aggregate the gradients from all workers in one all-reduce call.
            dist.all_reduce(grad_stats)
            global_grad_sum = grad_stats[:C]
            global_grad_mul = grad_stats[C:2 * C]
        else:
            global_grad_sum = grad_sum
            global_grad_mul = grad_mul

        # Compute gradient with respect to the input using the batch norm backwar
```

```
        # Note that: normalized = (input - global_mean) / global_std.
        # Hence, the gradient dL/dx is given by:
        #   (1/global_std) * [grad_output - (global_grad_sum / N) - normalized * (glob
        grad_input = (grad_output - (global_grad_sum / N)
                      - normalized * (global_grad_mul / (global_std * N))) / global_std

        return grad_input, None, None, None, None, None


class SyncBatchNorm(_BatchNorm):
    """
    Applies Batch Normalization to the input (over the 0 axis), aggregating the acti
    across all processes. You can assume that there are no affine operations in thi
    """

    def __init__(self, num_features: int, eps: float = 1e-5, momentum: float = 0.1):
        super().__init__(
            num_features,
            eps,
            momentum,
            affine=False,
            track_running_stats=True,
            device=None,
            dtype=None,
        )
        # your code here
        self.register_buffer("running_mean", torch.zeros(num_features))
        self.register_buffer("running_std", torch.ones(num_features))
        self.eps = eps
        self.momentum = momentum

    def forward(self, input: torch.Tensor, sync_grad: bool = True) -> torch.Tensor:
        if not self.training:
            return (input - self.running_mean) / self.running_std

        return sync_batch_norm.apply(input, self.running_mean,
```

```
                  self.running_std, self.eps, self.momentum,
                  sync_grad)
```

Benchmarking the training pipeline

Для данного задания пришлось довольно сильно переписать код. Пришлось разделить его на функции:

1. **main** отвечает за создание датасетов, модели и запуск экспериментов
2. функция **run_experiment** делает прогон всех эпох, замеряет время каждой эпохи, максимальную память и качество на тесте. Для оценки времени и памяти я брал значения только между 10 и 90 перцентилями всех замеров и усреднял только по ним, чтобы исключить выборсы. Функция усреднения и замера памяти представлены далее:

```
def get_avg_between_percentiles(values, lower_percentile, upper_percentile):
    sorted_values = sorted(values)
    lower_idx = int(len(sorted_values) * lower_percentile)
    upper_idx = int(len(sorted_values) * upper_percentile)
    return sum(sorted_values[lower_idx:upper_idx]) / (upper_idx - lower_idx)


def measure_peak_memory(device):
    """
    Measure the peak GPU memory usage on the given device.
    Uses torch.cuda.max_memory_allocated and resets stats afterward.
    """
    torch.cuda.synchronize(device)
    peak_mem = torch.cuda.max_memory_allocated(device)
    torch.cuda.reset_peak_memory_stats(device)
    return peak_mem
```

3. Функции **train_epoch** и **test_epoch** говорят выполняют прогон модели на трейни и на тесте

Я взял также побольше эпох для более точных замеров.

Также для DDP модели из torch я использую контекстный менеджер model.no_sync() для уменьшения операций all_reduce.

Добавил также фиксацию одинакового сида на разных процессах, чтобы гарантировать, что модель инициализируется одинаково. После сид фиксируется разными числами на разных процессах.

Команда для запуска:

```
torchrun --nproc_per_node=2 ddp_cifar100.py \
    --backend=nccl \
    --device=cuda \
    --implementation=custom \
    --grad_accumulation=2 \
    --batch_size=32 \
    --num_epochs=20
```

Implementation	Final Test Accuracy	Avg Memory Peak	Avg Epoch Time
custom	0.397	56.00 MB	6.54s
torch	0.395	59.75 MB	6.85s

Tests for SyncBatchNorm

В данной секции я прикладываю код для тестирования SyncBatchNorm с помощью pytest. Я сравниваю выход после forward, а также градиент посчитанный по предложенному лоссу. Все тесты в предложенных конфигурациях проходят.

```
import os
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
import torch.nn as nn
import pytest
from syncbn import SyncBatchNorm
from functools import partial
```

```
import random


def init_process(rank, size, fn, master_port, backend='gloo'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = str(master_port)
    dist.init_process_group(backend, rank=rank, world_size=size)
    fn(rank, size)


def worker_process(rank, world_size, hid_dim, batch_size, queue):
    """Worker process function that runs SyncBN."""
    torch.manual_seed(42 + rank)
    inputs = torch.randn(batch_size, hid_dim)
    inputs.requires_grad = True

    sync_bn = SyncBatchNorm(hid_dim)
    outputs = sync_bn(inputs)
    loss = outputs[:batch_size//2].sum()
    loss.backward()

    queue.put({
        'rank': rank,
        'outputs': outputs.detach().numpy(),
        'grad_inputs': inputs.grad.detach().numpy(),
    })


@pytest.mark.parametrize("num_workers", [1, 4])
@pytest.mark.parametrize("hid_dim", [128, 256, 512, 1024])
@pytest.mark.parametrize("batch_size", [32, 64])
def test_batchnorm(num_workers, hid_dim, batch_size):
    # Set up multiprocessing context
    ctx = mp.get_context("spawn")
    queue = ctx.Queue()
```

```
# Launch worker processes
port = random.randint(25000, 30000)
processes = []
for rank in range(num_workers):
    p = ctx.Process(
        target=init_process,
        args=(rank, num_workers,
            partial(
                worker_process,
                hid_dim=hid_dim,
                batch_size=batch_size,
                queue=queue
                ),
            port)
    )
    p.start()
    processes.append(p)

# Create regular BatchNorm for comparison
inputs_full = torch.randn(batch_size * num_workers, hid_dim)
for i in range(num_workers):
    torch.manual_seed(42 + i)
    inputs_full[i * batch_size:(i + 1) * batch_size] = torch.randn(batch_size, hid_c
inputs_full.requires_grad = True

bn = nn.BatchNorm1d(hid_dim, affine=False)

# Forward pass with regular BatchNorm
outputs_bn = bn(inputs_full)

# Compute loss (sum over first B/2 samples for each worker)
loss_bn = torch.tensor(0.)
for i in range(num_workers):
    start_idx = i * batch_size
    mid_idx = start_idx + batch_size // 2
```

```
        loss_bn += outputs_bn[start_idx:mid_idx].sum()

    # Backward pass
    loss_bn.backward()

    worker_results = [queue.get() for _ in range(num_workers)]
    for p in processes:
        p.join()

    # Compare outputs and gradients
    atol = 1e-3
    rtol = 0.0

    # Compare each worker's outputs and gradients against the corresponding sli
    worker_results = sorted(worker_results, key=lambda x: x['rank'])
    for res in worker_results:
        r = res['rank']
        worker_out = torch.from_numpy(res['outputs'])
        worker_grad = torch.from_numpy(res['grad_inputs'])
        ref_out = outputs_bn[r * batch_size:(r + 1) * batch_size]
        ref_grad = inputs_full.grad[r * batch_size:(r + 1) * batch_size]
        assert torch.allclose(worker_out, ref_out, atol=atol, rtol=rtol), \
            f"Rank {r} outputs don't match: max diff = " \
            f"{(worker_out - ref_out).abs().max()}"
        assert torch.allclose(worker_grad, ref_grad, atol=atol, rtol=rtol), \
            f"Rank {r} gradients don't match: max diff = " \
            f"{(worker_grad - ref_grad).abs().max()}"
```

Команда для запуска:

```
pytest test_syncbn.py -v
```

Performance benchmarks

В данной секции я представляю замеры по времени в предложенных конфигурациях.

Для сравнение реализаций я написал отдельный файл performance.py. Для каждой конфигурации я делаю несколько разминочных запусков, чтобы прогреть гпу, после чего делаю 50 итераций замеров, считаю среднее время по этим 50-ти итерациям и выбираю максимальное значение среди гпу.

```
import os
import itertools
import torch
import torch.distributed as dist
import torch.nn as nn
import time
from syncbn import SyncBatchNorm as CustomSyncBatchNorm


def benchmark_syncbn(impl, hid_dim, batch_size, num_iters=50):
    local_rank = int(os.environ.get("LOCAL_RANK", 0))
    device = torch.device(f'cuda:{local_rank}')
    torch.cuda.set_device(device)
    # Reset peak memory stats for accurate measurement
    torch.cuda.reset_peak_memory_stats(device)

    # Set up the BN layer (without affine parameters to match our custom impleme
    if impl == "custom":
        bn_layer = CustomSyncBatchNorm(hid_dim).to(device)
    elif impl == "standard":
        bn_layer = nn.SyncBatchNorm(hid_dim, affine=False).to(device)
    else:
        raise ValueError("impl must be either 'custom' or 'standard'.")
    bn_layer.train()

    # Warmup few iterations to avoid one-time GPU overheads
    for _ in range(5):
        x = torch.randn(batch_size, hid_dim, device=device, requires_grad=True)
```

```
        out = bn_layer(x)
        loss = out.sum()
        loss.backward()

    # Synchronize before launching the timed runs.
    torch.cuda.synchronize(device)
    start_event = torch.cuda.Event(enable_timing=True)
    end_event = torch.cuda.Event(enable_timing=True)
    start_event.record()
    for _ in range(num_iters):
        x = torch.randn(batch_size, hid_dim, device=device, requires_grad=True)
        out = bn_layer(x)
        loss = out.sum()
        loss.backward()
    end_event.record()

    # Wait for all work on the GPU to finish.
    torch.cuda.synchronize(device)
    elapsed_time_ms = start_event.elapsed_time(end_event)
    avg_time_ms = elapsed_time_ms / num_iters

    peak_memory_bytes = torch.cuda.max_memory_allocated(device)
    peak_memory_mb = peak_memory_bytes / (1024 * 1024)
    return avg_time_ms, peak_memory_mb

def run_benchmarks():
    hid_dims = [128, 256, 512, 1024]
    batch_sizes = [32, 64]
    num_iters = 50
    results = {}  # structure: results[impl][(hid_dim, batch_size)] = (avg_time_ms, p

    for impl in ["custom", "standard"]:
        results[impl] = {}
        for hid_dim, batch_size in itertools.product(hid_dims, batch_sizes):
            avg_time, peak_mem = benchmark_syncbn(impl, hid_dim, batch_size, nun
            # Create tensors so we can reduce across processes.
```

```
        device = torch.device(f'cuda:{int(os.environ.get("LOCAL_RANK", 0))}')
        avg_time_tensor = torch.tensor(avg_time, device=device)
        peak_mem_tensor = torch.tensor(peak_mem, device=device)
        # Reduce max to capture worst-case performance across processes.
        dist.reduce(avg_time_tensor, dst=0, op=dist.ReduceOp.MAX)
        dist.reduce(peak_mem_tensor, dst=0, op=dist.ReduceOp.MAX)
        if dist.get_rank() == 0:
            results[impl][(hid_dim, batch_size)] = (avg_time_tensor.item(), peak_me
            print(f"[{impl}] hid_dim: {hid_dim}, batch_size: {batch_size} → "
                  f"Avg time: {avg_time_tensor.item():.3f} ms, "
                  f"Peak memory: {peak_mem_tensor.item():.2f} MB")
    return results

def main():
    dist.init_process_group(backend='nccl')
    run_benchmarks()
    dist.destroy_process_group()

if __name__ == "__main__":
    main()
```

Команда для запуска:

```
torchrun --nproc_per_node=2 performance.py
```

Результаты:

Implementation	hid_dim	batch_size	Avg time (ms)	Peak memory (MB)
custom	128	32	0.858	0.10
custom	128	64	0.880	0.19
custom	256	32	0.867	0.20
custom	256	64	0.900	0.39
custom	512	32	0.865	0.40
custom	512	64	0.969	0.77

Implementation	hid_dim	batch_size	Avg time (ms)	Peak memory (MB)
custom	1024	32	0.861	0.79
custom	1024	64	0.903	1.54
standard	128	32	0.874	0.09
standard	128	64	0.945	0.17
standard	256	32	0.857	0.17
standard	256	64	0.870	0.33
standard	512	32	0.870	0.34
standard	512	64	0.902	0.65
standard	1024	32	0.871	0.68
standard	1024	64	0.953	1.30

Task 3

В данном задании, необходимо реализовать многопроцессорную обработку валидацинной выборке. Изначально я и так ее обрабатывал параллельно на каждой gpu, используя distributed sampler. В задании необходимо было реализовать пересылку с помощью scatter, поэтому я начал отправлять индекс на каждую гпу и оставлять только необходимые для нее данные в датасете.

```
def scatter_dataset(dataset, size, rank, device):
    total_samples = len(dataset)
    chunk_size = total_samples // size
    if rank == 0:
        new_total = chunk_size * size
        full_indices = torch.arange(new_total, dtype=torch.long, device=device)
        scatter_list = list(full_indices.view(size, chunk_size))
    else:
        scatter_list = None
    recv_indices = torch.empty(chunk_size, dtype=torch.long, device=device)
    dist.scatter(recv_indices, scatter_list=scatter_list, src=0)
```

```
    # Create a Subset of the test dataset using the received indices.
    subset = Subset(dataset, recv_indices.tolist())
    return subset
```

После делаю агрегирование метрик, пересылая все результаты на нулевой процесс.

```
# Aggregate metrics from all workers (only rank 0 gets the sum).
dist.reduce(total_loss, dst=0, op=dist.ReduceOp.SUM)
dist.reduce(total_acc, dst=0, op=dist.ReduceOp.SUM)
dist.reduce(total_size, dst=0, op=dist.ReduceOp.SUM)
```