分布式训练框架

需求分析

分布式训练存在两种范式：注重隐私的联邦学习和注重速度的分布式数据并行。

前者有中心服务器（controller）和实际执行训练的机器（client），由于数据具有non-IID特性，每次client进行完一个或多个epoch后（而不是step）向中心服务器发送得到的梯度，在中心服务器上聚合后再进行发布并开始下一轮训练； 而在分布式数据并行中，每台机器的地位都是相等的，数据是IID的，每训练一个step后通过allreduce或allgather这样的集合通信传播梯度。框架需要同时支持这两种范式并且对它们进行不同优化。

实验基于Mnist数据集和一个简单CNN神经网络模型进行

支持的特性

联邦学习

- 订阅发布barrier: Controller在双方匹配上后才开始发训练指令
- 容错机制：在每一轮的超时时间内，只要收到一个Client的结果就可以进行下一轮
- 断点重连：默认保存最新权重，若掉线重新开始时可通过配置init_path使用已有权重
- 高扩展性：通过配置文件来制定训练模型，数据、脚本
- 三个版本传输数据量优化: fp32_sparse(只传前10%大小的梯度)、int8量化(传输前把梯度量化成int8格式，传输后反量化)、sq8(结合前面两种优化方法)

分布式数据并行

- barrier: 在所有Worker匹配上后才开始进行训练
- 详细profile指标：统计数据传输量、dds通信时间、模型训练时间等详细指标
- 基于ZRDDS实现的分布式算子： allgather（稀疏梯度用）、allreduce
- DGC优化（deep gradient compression, ICLR2018）：热身后每个step仅更新0.1%的梯度进行训练，通过本地梯度累积和动量修正达到接近的精度

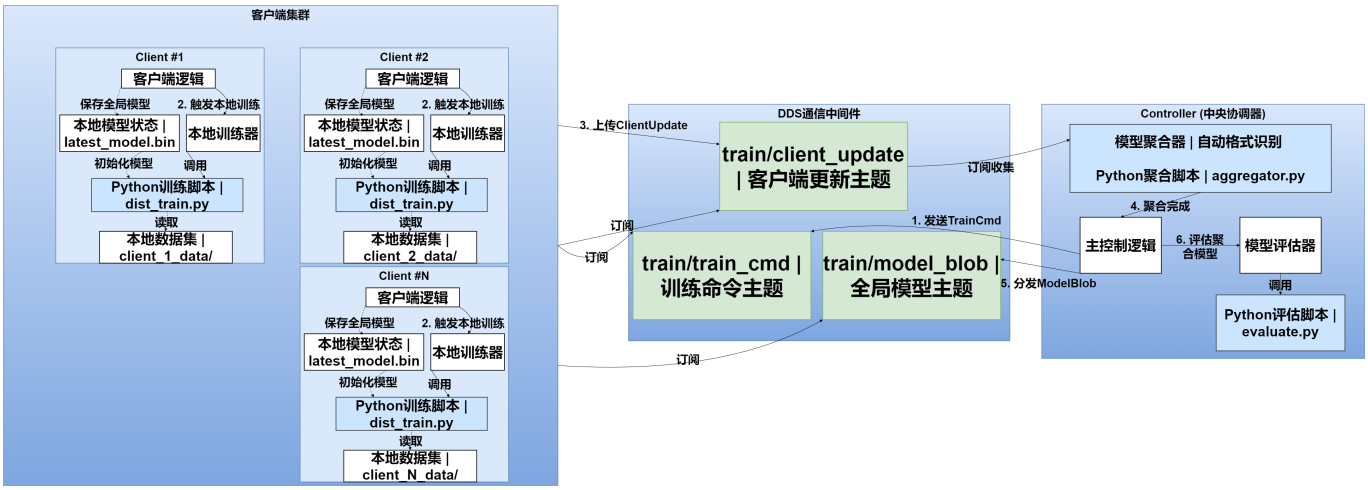
项目结构

```
distributed_training/
├── federated_learning/
│   ├── __init__.py
│   ├── Client.py
│   ├── Controller.py
│   ├── client.conf.json
│   └── controller.conf.json
│
├──normal_distributed/
│   ├── baseline/
│   │   ├── dds_barrier_verbose_baseline.py
│   │   ├── dgc_eval_baseline.py
│   │   ├── dgc_stepper_baseline.py
│   │   ├── train_base.py
```

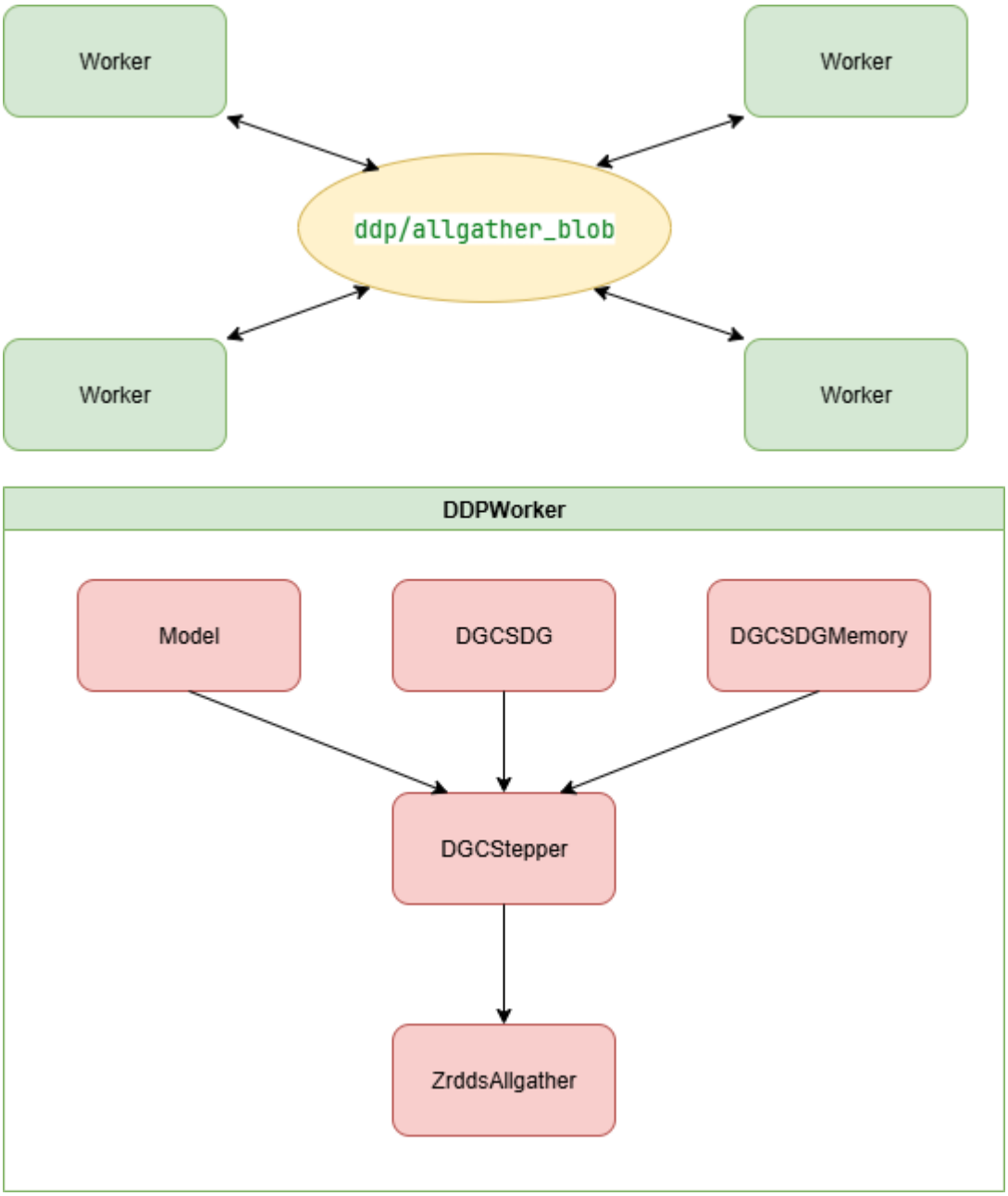
```
│   │   └── zrdds_dense_broadcast.py
│   └── dist_train_ddp_dgc/
│         ├── train.py
│         ├── compression.py
│         ├── dds_barrier_verbose.py
│         ├── dgc_eval.py
│         ├── dgc_stepper.py
│         ├── DGCSGD.py
│         ├── zrdds_allgather.py
│         └── memory.py
│         └── data/
│
│
├──federal_train_scripts/
    ├── aggregator.py
    ├── centric_train.py
    ├── dist_train_v2.py
    ├── dist_train_v3.py
    └── evaluate.py
```

架构图

联邦学习

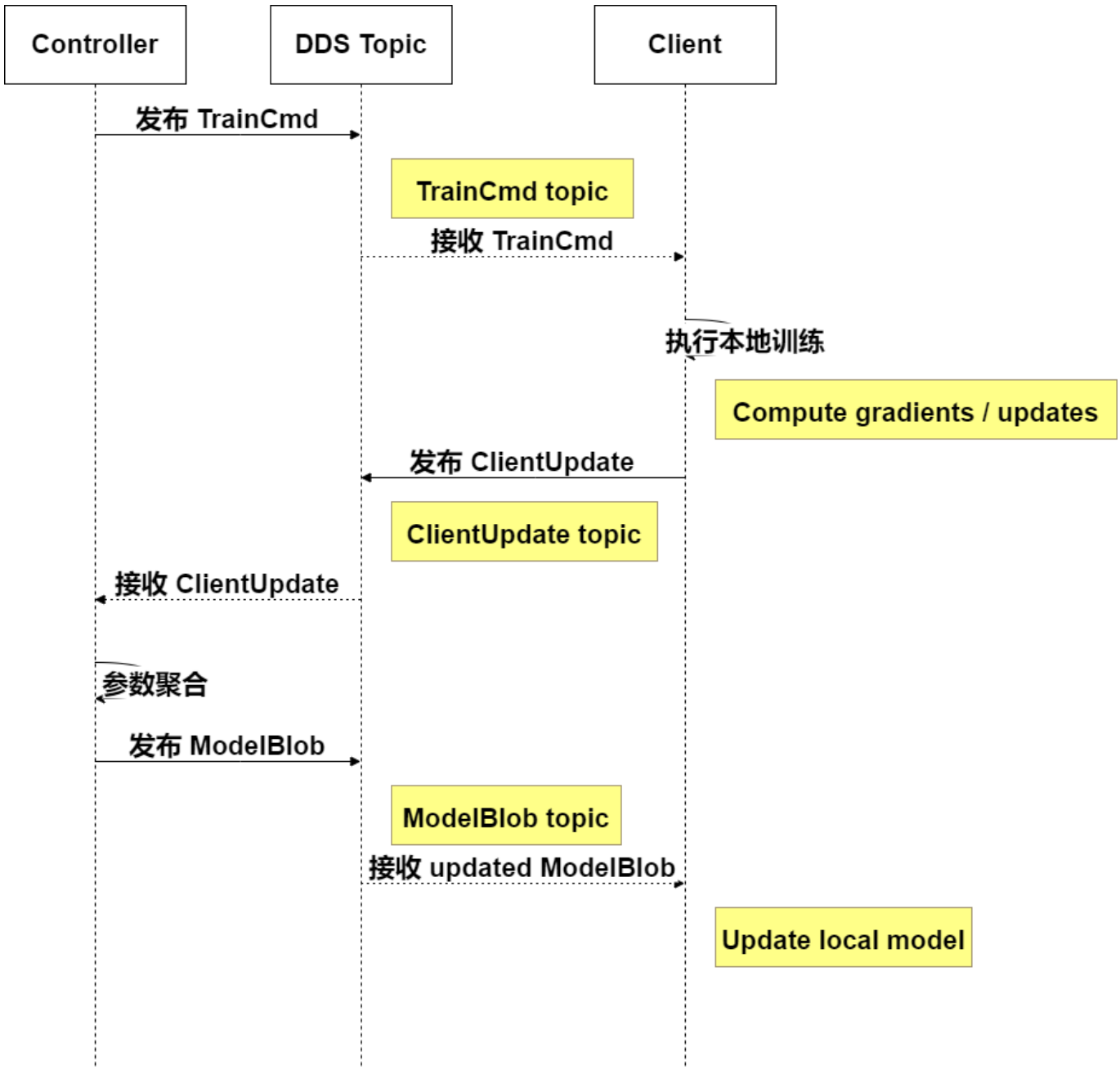


分布式数据并行

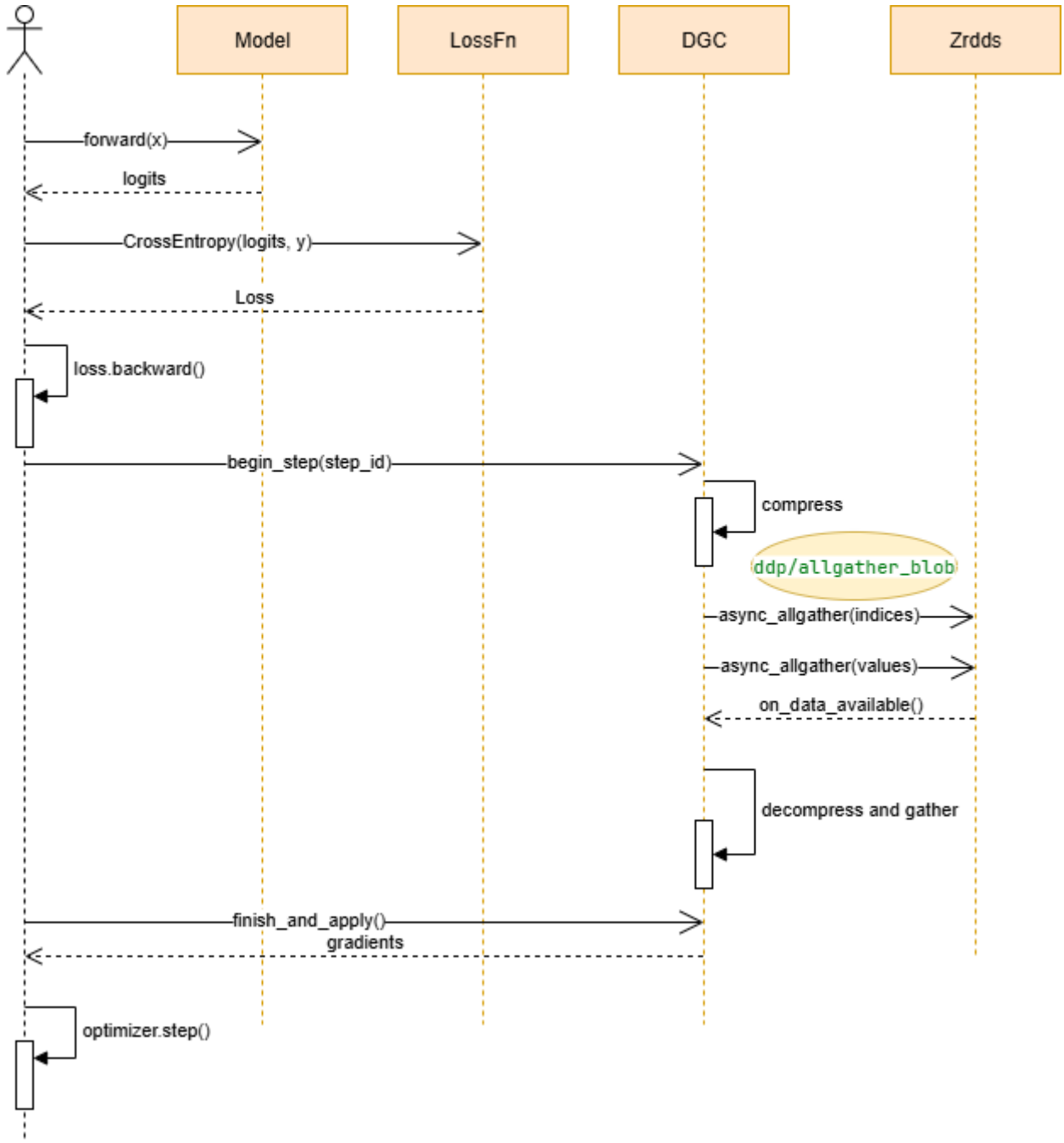

Worker
Worker
ddp/allgather_blob
Worker
Worker
DDPWorker
Model
DGCSDG
DGCSDGMemory
DGCStepper
ZrddsAllgather

流程图

联邦学习


Controller
DDS Topic
Client
发布 TrainCmd
TrainCmd topic
接收 TrainCmd
执行本地训练
Compute gradients / updates
发布 ClientUpdate
ClientUpdate topic
接收 ClientUpdate
参数聚合
发布 ModelBlob
ModelBlob topic
接收 updated ModelBlob
Update local model

分布式数据并行



原理介绍

联邦学习优化

1) INT8 量化

目的：把一整条 FP32 权重向量 压到 INT8 后再传，带宽约为 FP32 的 1/4

算法

- 按块长 chunk=C 把 w 切为 n=<<D/C>> 个块。
- 每块tensor大小：s_i = max_abs_in_block / 127
- 量化q_j = clip(round(w_j / s_i), -127, 127)
- 反量化 w_hat_j = q_j * s_i

带宽分析

- D = 向量长度，C = chunk
- bytes_Q8 ≈ D * 1 + (D / C) * 4
- bytes_FP32 = D * 4
- 当 C >= 1024 时，scale开销很小，bytes_Q8 / bytes_FP32 ≈ 1/4（我们实际跑用的8192）

2) FP32 稀疏

目的：只传本轮参数更新梯度的非零条目（Top-K），通过稀疏性省带宽。

算法

- 从本轮 中选择幅值最大的 K 个，稀疏度r=K/D（我们跑的时候取10%）。
- 仅发送这些 索引 + 浮点值；Controller 端按索引复原并聚合

带宽分析

- 每个非零梯度 = 索引 4B + 值 4B = 8B
- 若r=10%， 传输量2 * 10% = 20%

分布式数据并行优化

> 《Deep Gradient Compression: Reducing The Communication Bandwidth For Distributed Training》 聚焦分布式训练中的通信带宽瓶颈问题，提出深度梯度压缩（DGC）技术， 在不损失模型精度的前提下大幅降低梯度传输数据量，为大规模分布式训练 （尤其是移动端联邦学习）提供高效解决方案。 DGC方法由以下六个部分组成：
>
> 1）Gradient Sparsification
>
> 2）Local Gradient Accumulation
>
> 3）Momentum Correction
>
> 4）Local Gradient Clipping
>
> 5）Momentum Factor Masking
>
> 6）Warm-up Training

Algorithm 1: Deep Gradient Compression for vanilla momentum SGD on node k

Input: dataset χ
Input: minibatch size b per node
Input: momentum m
Input: the number of nodes N
Input: optimization function SGD
Input: initial parameters $w = \{w[0], \cdots, w[M]\}$

1: $U^k \leftarrow 0, V^k \leftarrow 0$
2: **for** $t = 0, 1, \cdots$ **do**
3: $G_t^k \leftarrow 0$
4: **for** $i = 1, \cdots, b$ **do**
5: Sample data x from χ
6: $G_t^k \leftarrow G_t^k + \frac{1}{Nb}\nabla f(x; \theta_t)$
7: **end for**
8: **if** Gradient Clipping **then**
9: $G_t^k \leftarrow Local\_Gradient\_Clipping(G_t^k)$
10: **end if**
11: $U_t^k \leftarrow m \cdot U_{t-1}^k + G_t^k$
12: $V_t^k \leftarrow V_{t-1}^k + U_t^k$
13: **for** $j = 0, \cdots, M$ **do**
14: $thr \leftarrow s\%\ of\ |V_t^k[j]|$
15: $Mask \leftarrow |V_t^k[j]| > thr$
16: $\widetilde{G_t^k}[j] \leftarrow V_t^k[j] \odot Mask$
17: $V_t^k[j] \leftarrow V_t^k[j] \odot \neg Mask$
18: $U_t^k[j] \leftarrow U_t^k[j] \odot \neg Mask$
19: **end for**
20: $All\text{-}gather:\ G_t \leftarrow \sum_{k=1}^{N} encode\left(\widetilde{G_t^k}\right)$
21: $\theta_{t+1} \leftarrow SGD(\theta_t, G_t)$
22: **end for**

1至7行描述了节点局部梯度更新的过程。

8至10行是局部梯度裁剪操作。

11、12两行涉及到动量修正。

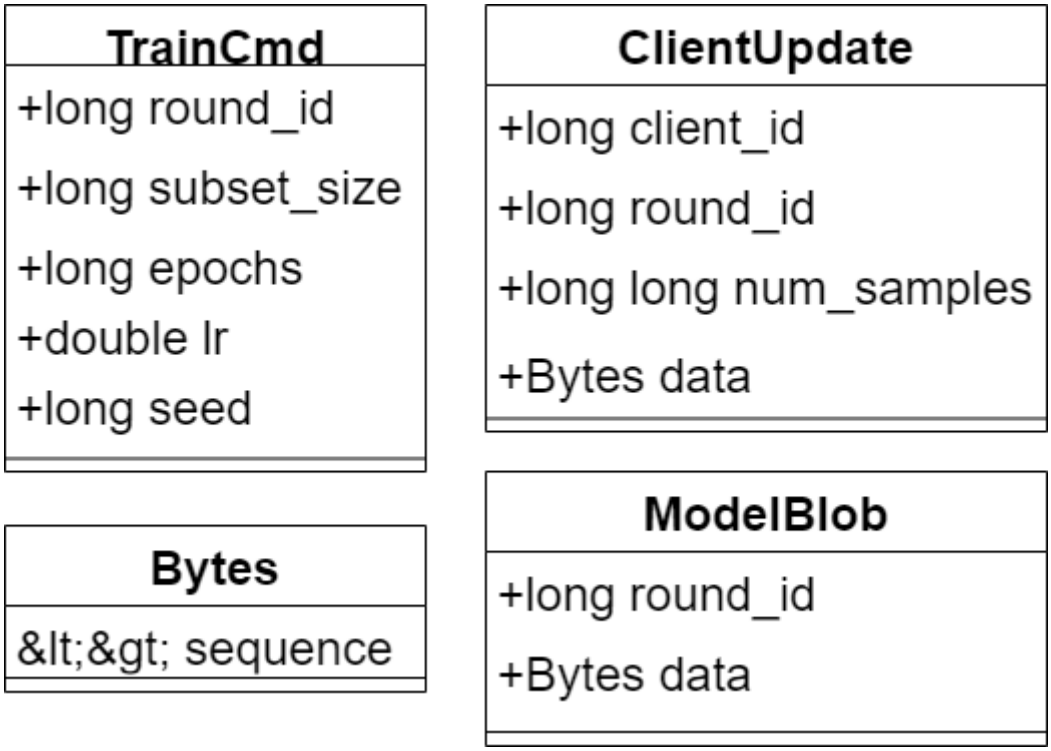
13至19行进行了梯度稀疏化操作，筛选出重要梯度，同时也保留次要梯度和其对应的动量项用于局部梯度累积。此外，这涉及到动量因子掩码操作，以应对延迟效应（Staleness Effect）。

20行是梯度聚合操作，得到全局梯度。

21行使用vanilla momentum SGD，通过全局梯度和历史模型参数更新模型参数。

详细设计

联邦学习



数据类型

数据结构名	DDS Topic 名称	功能说明
`TrainCmd`	`train/train_cmd`	Controller → Client：下发每轮训练指令（round, epoch, lr 等）
`ClientUpdate`	`train/client_update`	Client → Controller：上传本地训练后的梯度更新
`ModelBlob`	`train/model_blob`	Controller → Client：广播聚合后的全局模型

组件讲解

```
Client.start(self)
```

初始化 DDS 参与者与 Topic、创建读写端、监听器，进入接收 `TrainCmd`的状态

```
Client.run_training(self, round_id, subset, epochs, lr, seed)
```

按 `TrainCmd` 进行一轮本地训练

```
Client._s4_to_sq8_bytes(s4: bytes, chunk: int) -> bytes
```

把稀疏的FP32浮点数梯度压缩成int8类型，解析s4格式（用于描述稀疏增量的打包格式 S4：'S','4',0,1 | dim | k | idx[k] | vals[k]），计算 nChunks，求出scale后进行量化，然后重新打包

```
_TrainCmdListener.on_data_available() & _process(cmd)
```

接收并处理 `TrainCmd`；拉起训练并将结果写成 `ClientUpdate`,在`on_data_available`取样本

```
_ModelBlobListener.on_data_available()) & _process(mb)
```

收控制端发布的全局 FP32 模型，更新最近的模型。

```
Controller.init(self)
```

初始化 DDS Domain、Participant、根据配置文件选择barrier、设置监听。

```
Controller.run_round(self, subset_size, epochs, lr, seed)
```

写 `TrainCmd(round_id, subset_size, epochs, lr, seed)`,调用`_wait_for_streams(round_id, expected, min_clients, timeout_ms)`,等到至少 min_clients_to_aggregate 个客户端"完成,再调用`_collect_vectors(streams)`把各客户端包解码，最后调用`_apply_and_publish(cvs, round_id)`做梯度累加并在DDS中广播模型（发ModelBlob）

```
Controller._wait_for_streams(self, round_id, expected_clients, min_clients,
timeout_ms) -> Dict[int, ClientStream]
```

轮询统计完成数并打印 final-ready=X/expected (min=Y)；达标或超时返回（优先返回已完成者）。

```
Controller._collect_vectors(self, streams) -> List[ClientVec]
```

同一客户端多包：按前缀识别并解码（SQ8/S4/Q8/FP32），逐元素相加成单一向量；仅含 S4/SQ8则`is_delta=True`；是`num_samples`就 取Client最后一包。

分布式数据并行

DGCCompressor

`DGCCompressor`是该分布式训练框架中 **梯度压缩的核心组件**，通过稀疏化梯度和误差补偿机制显著减少通信开销，同时保持训练的精度。

核心功能

1. 对分布式训练中的梯度进行稀疏压缩，以降低通信开销
2. 支持 **误差补偿（Error Feedback）** 与 **动量累积**

3. 可配置压缩比例 (`compress_ratio`)、采样比例 (`sample_ratio`) 和采样方式（`strided_sample`）
4. 增加必要的压缩热身阶段 (`warmup_epochs`)，逐步增加压缩比例，提高训练稳定性

具体实现

1. **计算采样和选择 top-k 元素** 根据梯度绝对值重要性选择k个元素，并生成压缩索引

```
    def _sparsify(self, tensor, name):
     tensor = tensor.view(-1)
     numel, shape, num_selects, num_samples, top_k_samples, sample_stride =
self.attributes[name]
     importance = tensor.abs()
     if numel == num_samples:
         samples = importance
     else:
         if self.strided_sample:
             # 随机步长采样
             sample_start = random.randint(0, sample_stride - 1)
             samples = importance[sample_start::sample_stride]
         else:
             samples = importance[torch.randint(0, numel, (num_samples,),
device=tensor.device)]
     threshold = torch.min(torch.topk(samples, top_k_samples, 0,
largest=True, sorted=False)[0])
     mask = torch.ge(importance, threshold)
     indices = mask.nonzero().view(-1)
     num_indices = indices.numel()
     if numel > num_samples:
         for _ in range(self.max_adaptation_iters):
             if num_indices > num_selects:
                 if num_indices > num_selects * self.compress_upper_bound:
                     if self.resample:
                         indices = indices[torch.topk(importance[indices],
num_selects, 0, largest=True, sorted=False)[1]]
                         break
                     else:
                         threshold = threshold * self.compress_upper_bound
                 else:
                     break
             elif num_indices < self.compress_lower_bound * num_selects:
                 threshold = threshold * self.compress_lower_bound
             else:
                 break
             mask = torch.ge(importance, threshold)
             indices = mask.nonzero().view(-1)
             num_indices = indices.numel()
     indices = indices[:num_selects]
     values = tensor[indices]
     return values, indices, numel, shape, num_selects
```

2. **压缩接口** 使用误差反馈补偿梯度后进行稀疏压缩，返回压缩值和上下文信息

```
 def compress(self, tensor, name):
 if self.compress_ratio < 1.0 and name in self.attributes:
     # 误差反馈补偿
     tensor_compensated = self.memory.compensate(tensor, name,
accumulate=True)
     values, indices, numel, shape, num_selects =
self._sparsify(tensor_compensated, name)
     self.memory.update(name, (indices,))
     if numel < self.min_numel_to_compress:
         # 小张量直接走 dense
         ctx = (name, None, None, tensor.dtype, None, None)
         if self.fp16_values and tensor.dtype.is_floating_point:
             tensor = tensor.type(torch.float16)
         return tensor, ctx
     indices = indices.view(-1, 1)
     values = values.view(-1, 1)
     ctx = (name, numel, shape, values.dtype, indices.dtype,
tensor.data.view(numel))
     if self.fp16_values and values.dtype.is_floating_point:
         values = values.type(torch.float16)
     if self.int32_indices and not indices.dtype.is_floating_point:
         indices = indices.type(torch.int32)
     return (values, indices), ctx
 else:
     ctx = (name, None, None, tensor.dtype, None, None)
     if self.fp16_values and tensor.dtype.is_floating_point:
         tensor = tensor.type(torch.float16)
     return tensor, ctx
```

ddp_barrier_verbose

核心功能

实现分布式屏障，确保所有节点在通信前都已就绪，检查 DDS Writer/Reader 匹配状态，避免消息丢失，使用 allgather 同步各节点 rank，验证所有节点在线。

具体实现

Barrier 核心逻辑 在匹配到所有Reader与Writer后，每个节点发送自身 rank，收集所有节点 rank，确保全体节点到达屏障

```
# 1) 先确保 Writer/Reader 都有匹配（避免第一批包在对端未订阅时丢掉）
print("========== [barrier] matching ==========")
_ = _wait_reader_matched(zrdds_allgather.reader, min_reader_matches,
int(match_timeout_s * 1000))
_ = _wait_writer_matched(zrdds_allgather.writer, min_writer_matches,
int(match_timeout_s * 1000))

# 2) allgather barrier：每个 rank 发一个 4 字节 rank，收齐 world 份为止
```

```
print("========== [barrier] allgather ==========")
MAGIC_ROUND = 0x42615252  # 'BaRR', 与训练/评估的 round_id 空间隔离
payload = struct.pack("<I", int(rank))

# 发送并等待
h = zrdds_allgather.allgather_async(group_id=group_id, round_id=MAGIC_ROUND,
                                    name="barrier", part_id=0,
                                    rank=rank, world=world, payload=payload)
ok = True
try:
    frames = h[1](barrier_timeout_s)  # list[bytes] (按 rank 排序)
    seen = [struct.unpack("<I", b)[0] for b in frames]
    print(f"[barrier] seen ranks: {seen}")
    missing = [i for i in range(world) if i not in seen]
    if missing:
        ok = False
        print(f"[barrier] MISSING ranks: {missing}")
except TimeoutError as e:
    ok = False
    print(f"[barrier] TIMEOUT: {e}")

print(f"========== [barrier] result: {'OK' if ok else 'FAILED'} ==========")
return ok
```

dgc_eval

核心功能

- 在本地数据集上计算 **准确率**
- 使用 `ZrddsAllgather` 汇总各 rank 的正确预测数和样本总数
- 返回全局准确率 `(global_correct, global_total, acc)`，以支持分布式训练

具体实现

1. **打包/解包计数** 将每个 rank 的正确数和总数打包为二进制，便于在 DDS 中传输

```
_FMT = "<qq"  # 2 个 int64: correct, total
def _pack_counts(correct, total): return struct.pack(_FMT, correct, total)
def _unpack_counts(b): return struct.unpack(_FMT, b)
```

2. **全局汇总函数** 使用 `allgather_async` 收集各 rank 的计数，按 rank 排序并累加得到全局统计

```
def ddp_reduce_accuracy_counts(zrdds, group_id, round_id, name, rank, world,
correct, total, timeout_s=30.0):
    payload = _pack_counts(correct, total)
    h = zrdds.allgather_async(group_id, round_id, name=f"metric.{name}",
part_id=0,
                              rank=rank, world=world, payload=payload)
```

```
    frames = h[1](timeout_s)
    g_correct = sum(_unpack_counts(fb)[0] for fb in frames)
    g_total   = sum(_unpack_counts(fb)[1] for fb in frames)
    return g_correct, g_total, g_correct / g_total
```

3. **评估接口** 在本地数据上计算准确率，再调用全局汇总函数得到 DDP 全局准确率

```
@torch.no_grad()
def ddp_evaluate_top1(model, dataloader, device, zrdds, group_id, epoch_or_step,
name, rank, world, timeout_s=60.0):
    model.eval()
    correct = 0; total = 0
    for xb, yb in dataloader:
        logits = model(xb.to(device))
        pred = logits.argmax(dim=1)
        correct += (pred == yb.to(device)).sum().item()
        total   += yb.numel()
    return ddp_reduce_accuracy_counts(zrdds, group_id, epoch_or_step, name, rank,
world, correct, total, timeout_s)
```

DDPDGCStepper

核心功能

- 将 **DGC 压缩** 与 **ZrddsAllgather 异步通信**接入每个训练 step
- 仅统计 **纯网络等待时间**，不包含解压/聚合
- 提供 **窗口聚合统计与日志打印**（如每 N 个 steps 的平均等待时间、传输数据量等）

具体实现

1. **将梯度压缩并发起异步 allgather** 根据梯度类型选择压缩和发送方式，支持异步收集

```
for name, p in self.named_params:
    compressed, ctx = self.comp.compress(p.grad.data, name)
    self._ctx.append((name, ctx, p))
    if isinstance(compressed, (tuple, list)):  # 稀疏梯度
        values, indices = compressed
        h0 = self.comm.allgather_async(..., payload=self._to_bytes(values))
        h1 = self.comm.allgather_async(..., payload=self._to_bytes(indices))
        self._handles.append((name, True, (h0, h1)))
    else:  # 稠密梯度
        h = self.comm.allgather_async(..., payload=self._to_bytes(compressed))
        self._handles.append((name, False, (h,)))
```

2. **等待通信完成并解压/聚合梯度** 等待异步 allgather 完成，按 rank 汇总梯度，解压稀疏梯度或累加稠密梯度，并做全局平均

```
for (name, is_sparse, hs), (_, ctx, p) in zip(self._handles, self._ctx):
    if is_sparse:
        vals_lists = self._await_and_collect(hs[0], timeout_ms)
        idxs_lists = self._await_and_collect(hs[1], timeout_ms)
        dense = torch.zeros(ctx[1], dtype=torch.float32, device=device)
        dense.index_put_([i_cat], v_cat, accumulate=True)
        dense.mul_(1.0 / self.world)
        p.grad.data = dense.view(ctx[2])
    else:
        dense_lists = self._await_and_collect(hs[0], timeout_ms)
        acc = sum(self._from_bytes(vb, self.dtype_val, device) for vb in
dense_lists)
        acc.mul_(1.0 / self.world)
        p.grad.data.copy_(acc.view_as(p.grad.data))
```

DGCSGD

核心功能

- 基于 **PyTorch SGD** 实现的 **DGC兼容优化器**
- 支持 **动量（momentum）**、**权重衰减（weight_decay）** 和 **Nesterov 加速**
- 可以直接与 **DGC 压缩梯度机制** 配合使用，用于分布式训练

具体实现

单步更新梯度 对每个参数执行梯度更新：

- 先应用 **权重衰减**
- 再处理 **动量/缓冲**
- 如果使用 Nesterov，梯度会提前修正
- 最后按学习率更新参数

```
for group in self.param_groups:
    for p in group['params']:
        if p.grad is None:
            continue
        if weight_decay != 0:
            d_p = weight_decay * p.data
            if momentum != 0:
                buf = self.state[p].get('momentum_buffer', d_p)
                buf.mul_(momentum).add_(d_p, alpha=1 - dampening)
                d_p = buf if not nesterov else d_p.add(buf, alpha=momentum)
            d_p = d_p.add(p.grad)
        else:
            d_p = p.grad
        p.add_(d_p, alpha=-group['lr'])
```

DGCSGDMemory

核心功能

- 实现 **DGC 梯度压缩下的动量修正与误差反馈缓存**
- 支持 **Nesterov 动量、梯度裁剪** 和 **动量掩码**
- 记录每个参数的 **momentum** 与 **velocity**，用于梯度压缩后的误差补偿

具体实现

梯度补偿 使用 **momentum** 更新动量，累积到 velocity 中，实现误差反馈。

```
def compensate(self, grad, name, accumulate=True):
    mmt = self.momentums[name]
    if accumulate:
        vec = self.velocities[name]
        if self.nesterov:
            mmt.add_(grad).mul_(self.momentum)
            vec.add_(mmt).add_(grad)
        else:
            mmt.mul_(self.momentum).add_(grad)
            vec.add_(mmt)
        return vec
    else:
        if self.nesterov:
            mmt.add_(grad).mul_(self.momentum)
            return mmt.add(grad)
        else:
            mmt.mul_(self.momentum).add_(grad)
            return mmt.clone()
```

ZrddsAllgather

核心功能

- **实现分布式 allgather 操作**：每个 rank 将自己的数据分片广播，所有 rank 收集所有同名分片后返回
- **异步通信与句柄管理**：提供 `_AGHandle`，将 **通信等待** 与 **收集合并** 分离
- **数据分片管理与重组**：支持大 payload 的分片发送与按 rank 汇总

具体实现

- **打包/解包通信帧** 用固定头部格式和字节序列，实现分布式通信数据的统一编码和解码

```
def pack_frame(group_id, round_id, tensor_name, part_id, rank, world, seq,
seq_cnt, payload: bytes):
    g = group_id.encode('utf-8'); n = tensor_name.encode('utf-8')
    hdr = struct.pack(HDR_FMT, MAGIC, len(g), len(n), part_id, rank, round_id,
world, seq, seq_cnt, 0)
    return hdr + g + n + (payload or b"")

def unpack_frame(frame: bytes):
```

```
    magic, gl, nl, part_id, rank, round_id, world, seq, seq_cnt, _ =
struct.unpack(HDR_FMT, frame[:HDR_SIZE])
    off = HDR_SIZE
    group_id = frame[off:off+gl].decode(); off += gl
    name = frame[off:off+nl].decode(); off += nl
    payload = frame[off:]
    return group_id, round_id, name, part_id, rank, world, seq, seq_cnt, payload
```

- **异步 allgather 发送与句柄返回** 将大 payload 拆分成块发送，返回 `_AGHandle`，可单独 `wait()` 或 `collect()`

```
def allgather_async(self, *, group_id:str, round_id:int, name:str, part_id:int,
                    rank:int, world:int, payload:bytes, max_chunk=4<<20):
    chunks = [payload[i:i+max_chunk] for i in range(0, len(payload or b""),
max_chunk)] or [b""]
    key = self._make_key(group_id, round_id, name, part_id)
    ev = self._done.setdefault(key, threading.Event())
    for seq, ck in enumerate(chunks):
        mb = dds.ModelBlob()
        body = pack_frame(group_id, round_id, name, part_id, rank, world, seq,
len(chunks), ck)
        mb.data = body
        self.writer.write(mb)
    def _get_and_clear():
        mp = self._buckets.pop(key, {})
        self._done.pop(key, None)
        return [b"".join(mp[rk]) for rk in sorted(mp.keys())]
    return _AGHandle(key, ev, _get_and_clear)
```

- **接收数据与汇总** 按 rank 收集分片，所有分片到齐后触发完成事件

```
def _on_raw(self, raw: bytes):
    g, r, name, p, rank, world, seq, seq_cnt, payload = unpack_frame(raw)
    key = self._make_key(g, r, name, p)
    self._buckets[key][rank].append((seq, payload))
    if len(self._buckets[key][rank]) == seq_cnt:
        self._buckets[key][rank].sort(key=lambda x: x[0])
        self._buckets[key][rank] = [b for _, b in self._buckets[key][rank]]
    if len(self._buckets[key]) == world:
        self._done[key].set()
```

性能测试结果与分析

联邦学习实验条件

- batch_size：64
- epochs per round：1

- subset per round: 6000
- rounds：10

> 硬件配置
>
> controller:
>
> 12th Gen Intel(R) Core(TM) i5-1240P
>
> worker:
>
> 13th Gen Intel(R) Core(TM) i9-13900H
>
> 13th Gen Intel(R) Core(TM) i9-13980HX

分布式数据并行

以下四张图是我们的分布式数据并行baseline版本和DGC版本各方面的对比

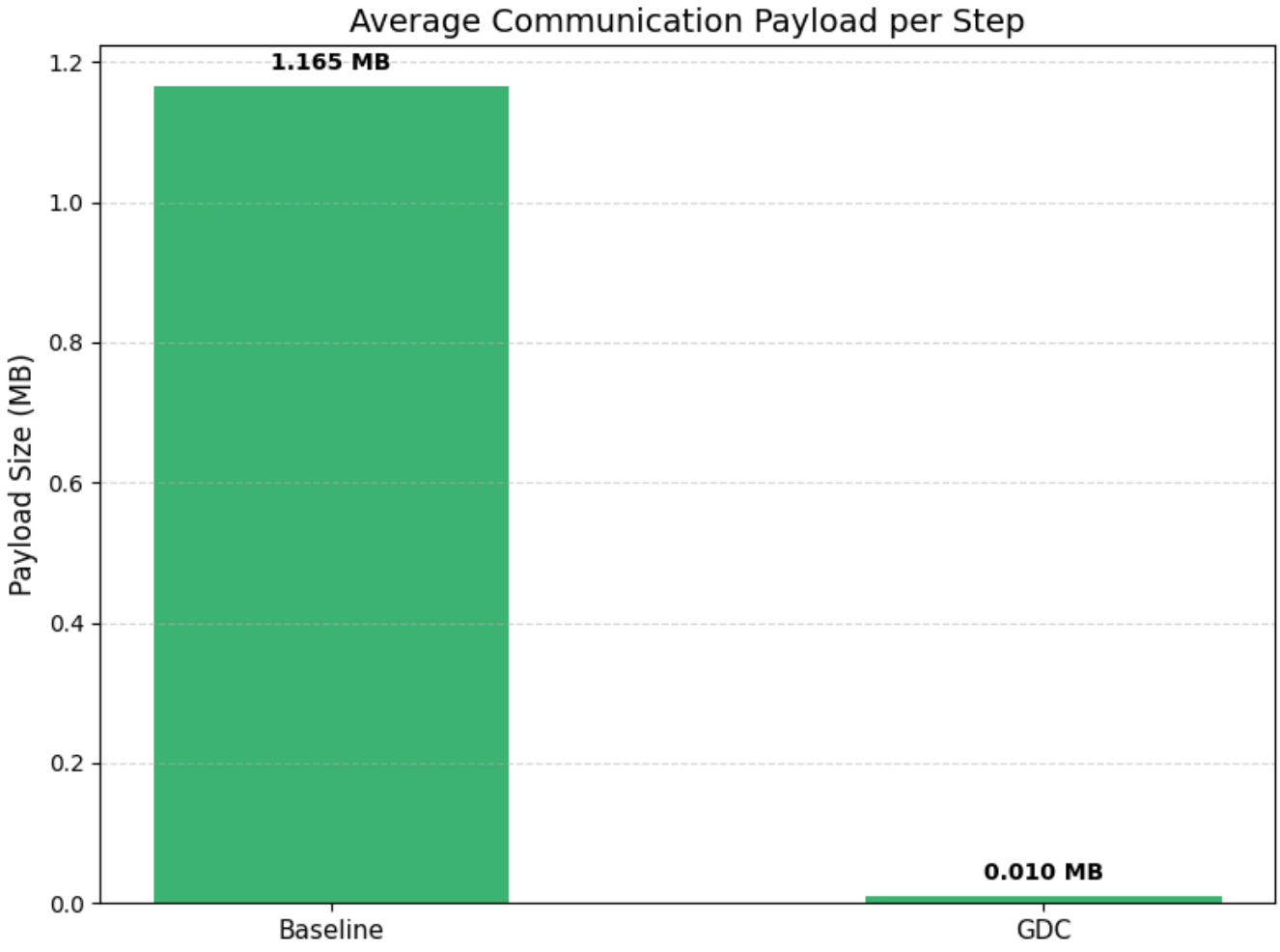


图1：数据传输量比较

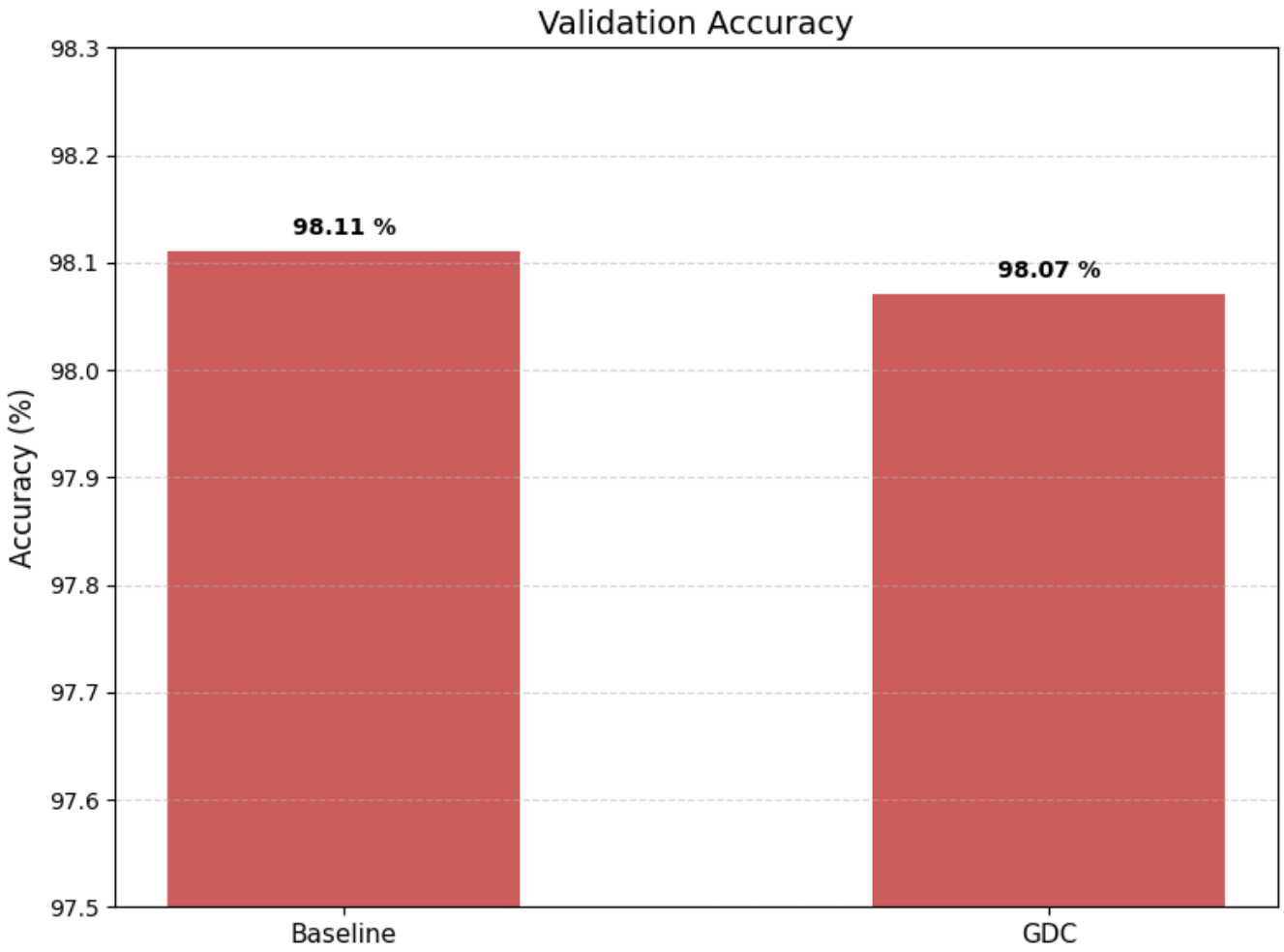

Validation Accuracy
Accuracy (%)
98.3
98.2
98.1
98.0
97.9
97.8
97.7
97.6
97.5
98.11 %
98.07 %
Baseline
GDC

图2：准确率比较

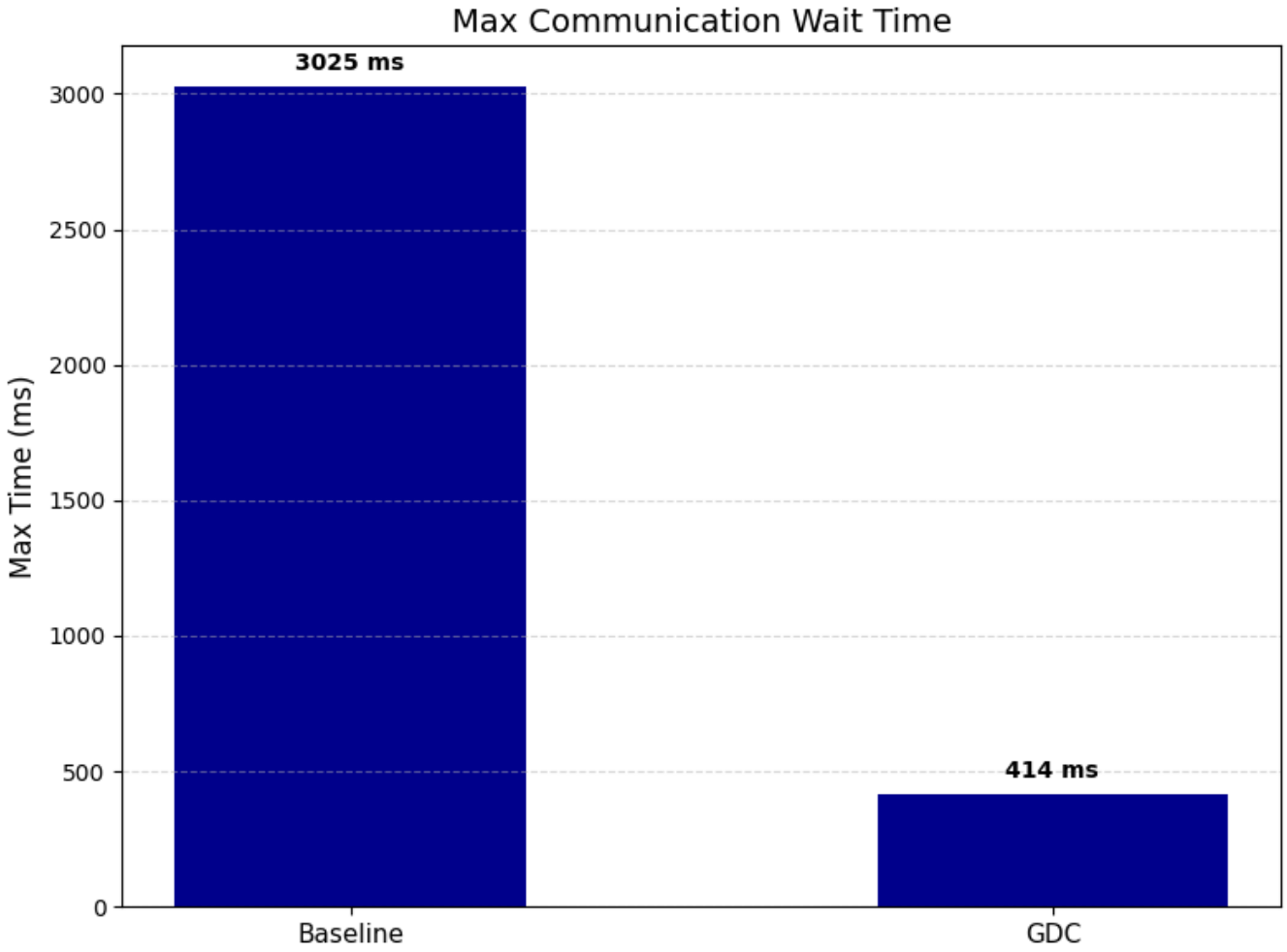


图3：最大通信时间比较（由于存在带宽没有打满的情况，最大值的差距更能体现性能之差）

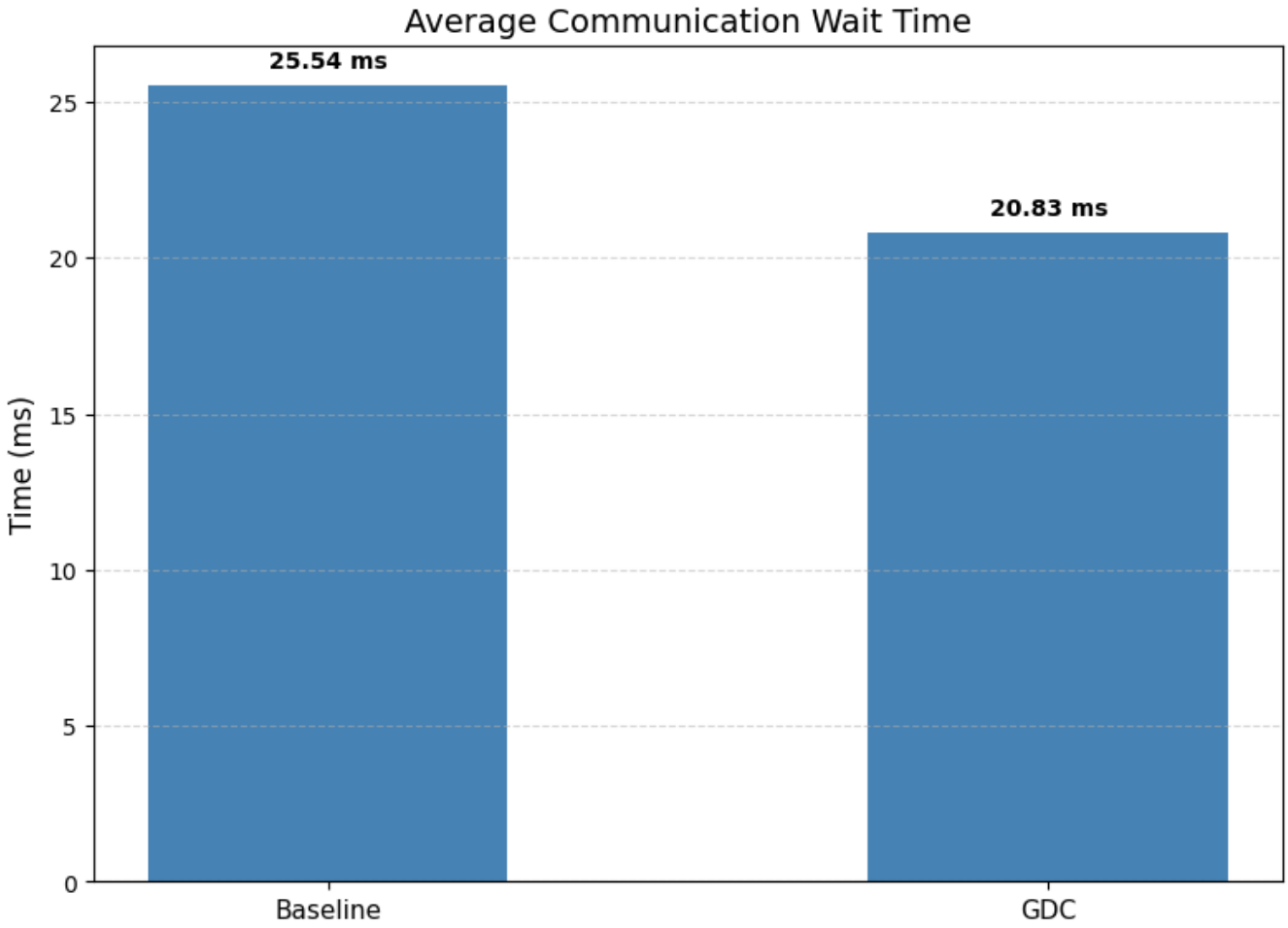


图4：平均通信时间比较（由于存在带宽没有打满的情况，平均值差距并不大）

联邦学习优化

以下四张图是我们的四种联邦学习优化在各方面的指标，fp32作为联邦学习baseline， int8_dense是量化版本， fp32_sparse只取前10%的梯度进行传输， int8_sparse结合了前面两种优化方法

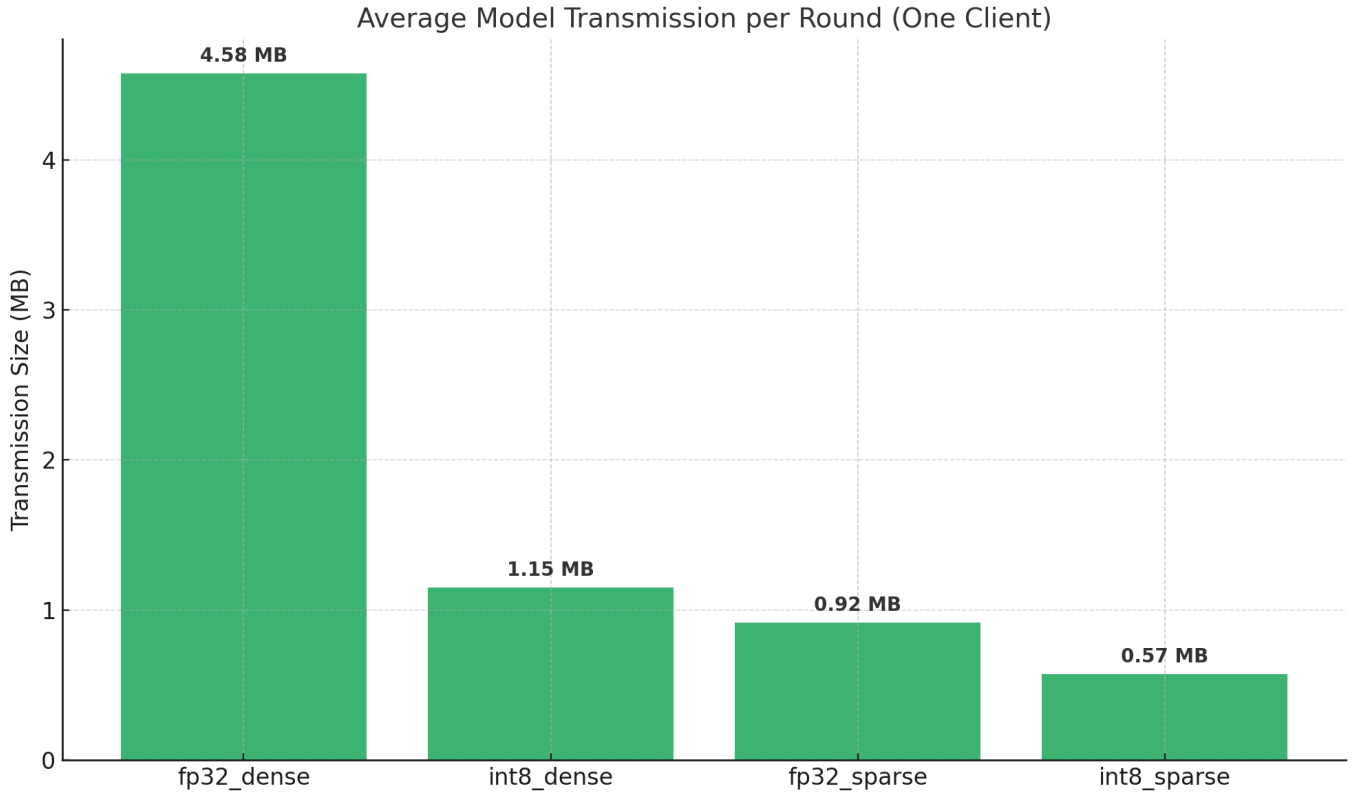


图1：四种方式数据传输量比较

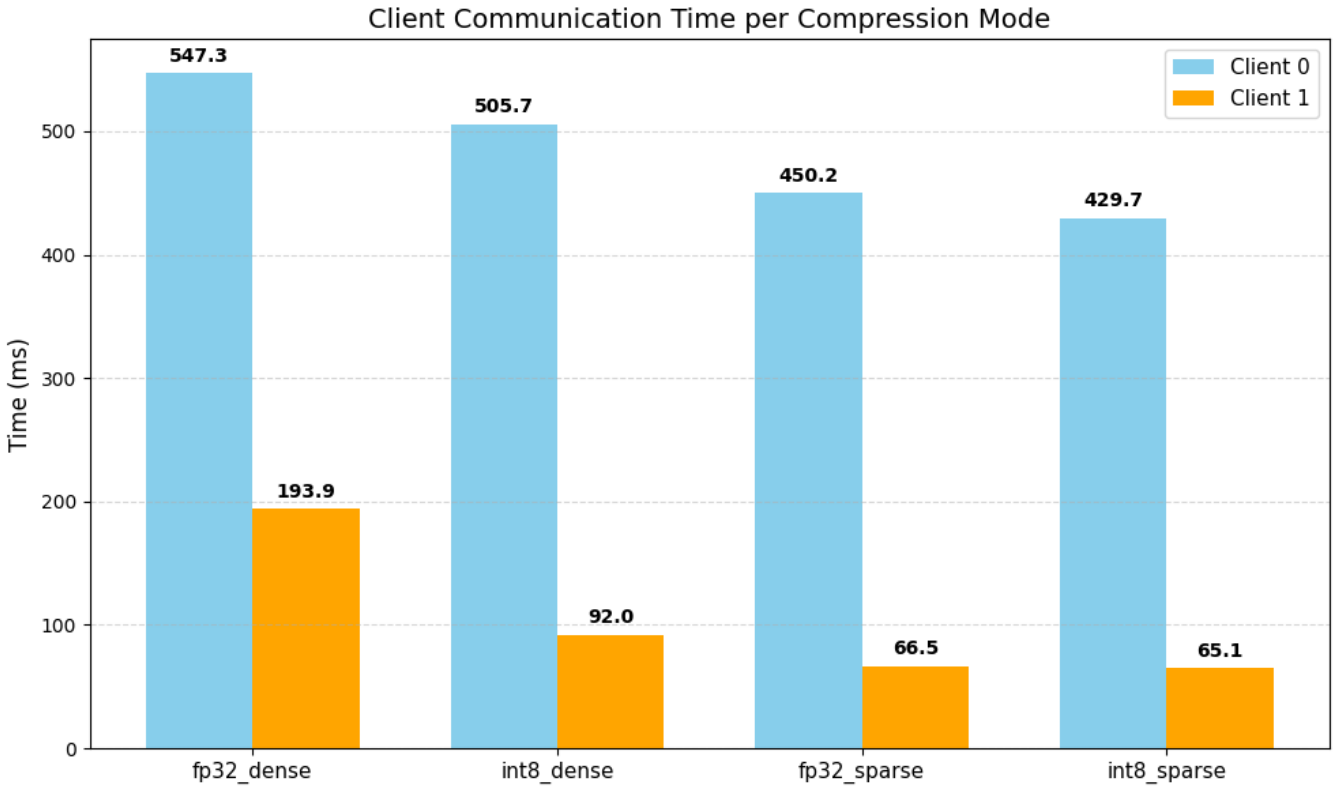


图2：四种方式DDS通信时间比较，Client1和Client0是与Controller通信的两台不同机器

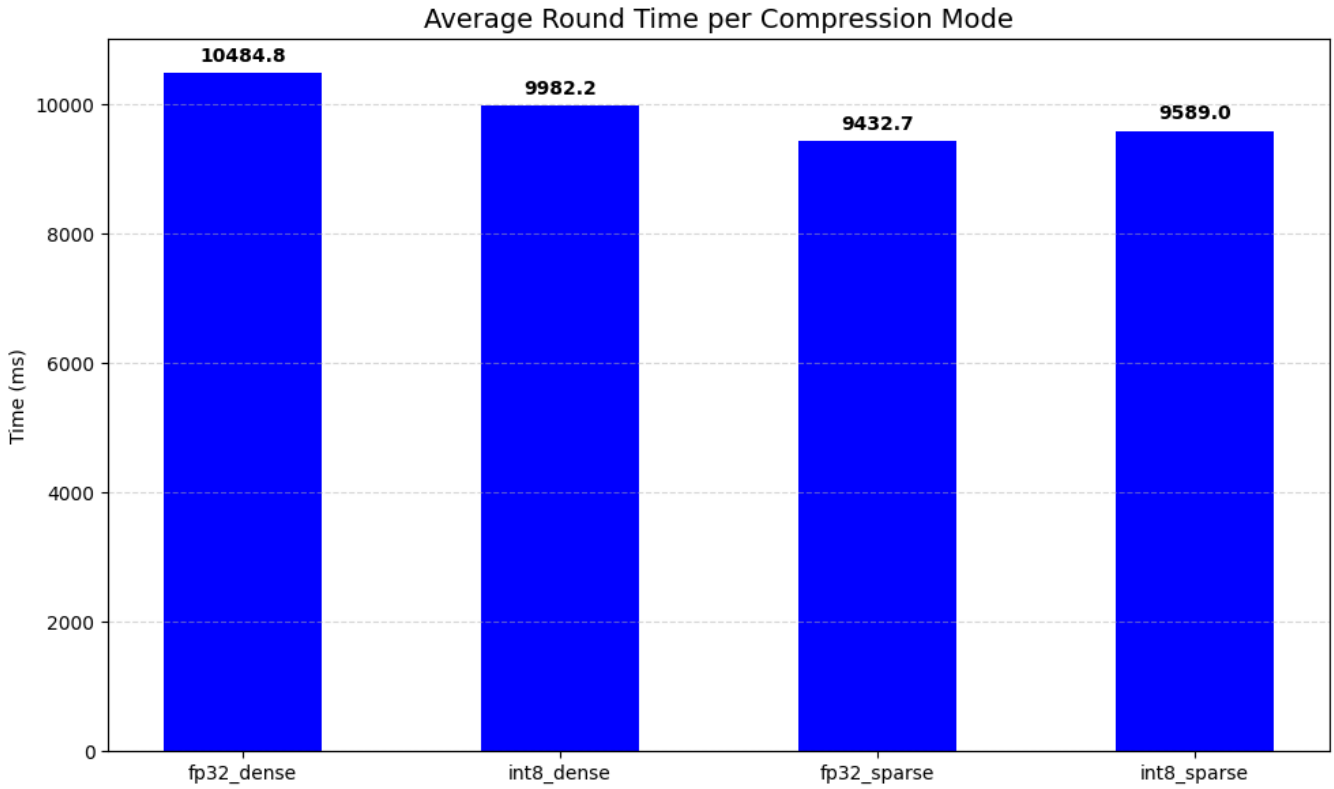


图3：四种方式每一轮时间比较，从发出训练命令开始到聚合并发布下一轮模型结束

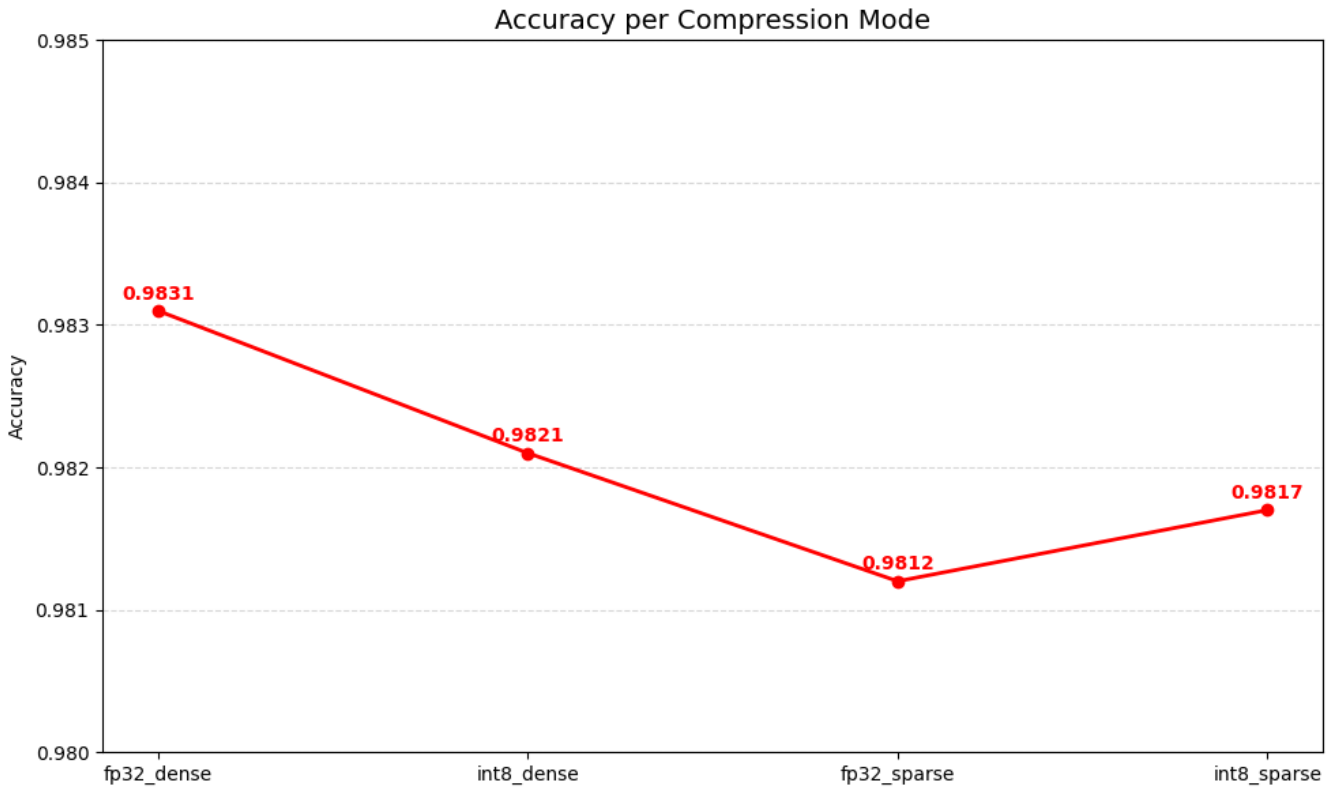


图4：四种方式模型精度比较，取10轮的最终结果

联邦学习

使用方法

联邦学习

> 编写对应的配置文件（如运行controller，就写一份controller.conf.json）,在命令行中以配置文件路径作为形参，然后启动训练的程序即可（Controller.py和Client.py） **分布式数据并行** 通过环境变量设置每台机器的WORLD和RANK，然后直接运行train.py或train_base.py