《机器学习编程实践》课程——DAY 8

课程内容

- 构建简单的计算图
- 多卡并行

一、 构建简单的计算图

1 动态图和静态图的区别

1.1 动态图

- 在前向传播时，已经把图构建出来——前向传播到哪里，图跟随生成至哪里
- 反向传播时，跟着图去进行逐步往前回传
- 优势：更方便，可以随时对计算图进行修改

1.2 静态图

- 正向传播时只是正向传播
- 反向传播时把整个图构建出来

2 具体操作——以动态图为例

2.1 变量命名

命名元组和列表，用于记录计算图中的每个操作

```
class TapeEntry(NamedTuple):
  inputs: List[str] # 操作的输入变量名列表
  outputs: List[str] # 操作的输出变量名列表
  # propagate函数：用于反向传播时应用链式法则计算梯度
  propagate: Callable[[List[Variable]], List[Variable]]

gradient_tape: List[TapeEntry] = [] # 全局列表，用于记录所有计算操作(计算图)
```

2.2 计算图重置

```
def reset_tape():
    gradient_tape.clear()
    global _name
    _name = 0 # 重置变量命名计数器
```

2.3 前向传播（存储计算图）

```
# 以构建乘法算子为例
def operator_mul(self: Variable, rhs: Variable) -> Variable:
    # 如果乘1.0则直接返回原变量
    if isinstance(rhs, float) and rhs == 1.0:
        return self
    # 否则计算相乘值
    r = Variable(self.value * rhs.value)
    print(f'{r.name} = {self.name} * {rhs.name}')

    # 记录操作的输入输出变量名
    inputs = [self.name, rhs.name]
    outputs = [r.name]

    # 反向传播函数，计算乘法操作的梯度（输入：output的梯度；输出：input的梯度）
    def propagate(dL_doutputs: List[Variable]):
        dL_dr, = dL_doutputs

        # 计算导数
        dr_dself = rhs
        dr_drhs = self

        # 使用链式法则
        dL_dself = dL_dr * dr_dself
        dL_drhs = dL_dr * dr_drhs
        dL_dinputs = [dL_dself, dL_drhs]
        return dL_dinputs

    # 将操作记录到计算图中，并返回结果
    gradient_tape.append(TapeEntry(inputs=inputs, outputs=outputs,
        propagate=propagate))
```

```
return r # 返回计算结果
```

计算图在存什么：

- input
- output
- 反向传播的规则——依靠这个可以计算出所有变量的梯度

2.4 反向传播（计算梯度）

```
# 初始化梯度字典，设置损失函数L对自己的梯度为1
def grad(L, desired_results: List[Variable]) -> List[Variable]:
    dL_d: Dict[str, Variable] = {}
    dL_d[L.name] = Variable(torch.ones(()))
    print(f'd{L.name} ------------------------')

    # 辅助函数，收集变量的梯度
    def gather_grad(entries: List[str]):
        return [dL_d[entry] if entry in dL_d else None for entry in entries]

    for entry in reversed(gradient_tape):
        dL_doutputs = gather_grad(entry.outputs)
        # 反向遍历计算图，跳过不需要梯度的路径
        if all(dL_doutput is None for dL_doutput in dL_doutputs):
            continue

        # 执行反向传播并累加梯度
        dL_dinputs = entry.propagate(dL_doutputs)
        for input, dL_dinput in zip(entry.inputs, dL_dinputs):
            if input not in dL_d:
                dL_d[input] = dL_dinput
            else:
                dL_d[input] += dL_dinput

    for name, value in dL_d.items():
        # 打印梯度信息
        print(f'd{L.name}_d{name} = {value.name}')
    print(f'------------------------')
    # 返回所需变量的梯度
```

```
    return gather_grad(desired.name for desired in desired_results)
```

2.5 使用示例

```
a_global, b_global = torch.rand(4), torch.rand(4)

def simple(a, b):
    t = a + b
    return t * b

reset_tape()
a = Variable.constant(a_global, name='a')
b = Variable.constant(b_global, name='b')
c = simple(a, b)
loss = c.sum()
# 得到对应梯度
da, db = grad(loss, [a, b])
print("da", da)
print("db", db)
```

2.6 一些其它基本操作

```
# 加法操作符重载
def operator_add(self: Variable, rhs: Variable) -> Variable:
    r = Variable(self.value + rhs.value)
    print(f'{r.name} = {self.name} + {rhs.name}')

    def propagate(dL_doutputs: List[Variable]):
        dL_dr, = dL_doutputs
        dr_dself = 1.0
        dr_drhs = 1.0
        dL_dself = dL_dr * dr_dself
        dL_drhs = dL_dr * dr_drhs
        return [dL_dself, dL_drhs]

    gradient_tape.append(TapeEntry(inputs=[self.name, rhs.name],
        outputs=[r.name], propagate=propagate))
    return r
```

```
# 求和操作重载
def operator_sum(self: Variable, name: Optional[str]) -> 'Variable':
   r = Variable(torch.sum(self.value), name=name)
   print(f'{r.name} = {self.name}.sum()')

   def propagate(dL_doutputs: List[Variable]):
      dL_dr, = dL_doutputs
      size = self.value.size()
      return [dL_dr.expand(*size)]

   gradient_tape.append(TapeEntry(inputs=[self.name], outputs=[r.name],
       propagate=propagate))
   return r

# 扩展操作重载
def operator_expand(self: Variable, sizes: List[int]) -> 'Variable':
   assert (self.value.dim() == 0) # only works for scalars
   r = Variable(self.value.expand(sizes))
   print(f'{r.name} = {self.name}.expand({sizes})')

   def propagate(dL_doutputs: List[Variable]):
      dL_dr, = dL_doutputs
      return [dL_dr.sum()]

   gradient_tape.append(TapeEntry(inputs=[self.name], outputs=[r.name],
       propagate=propagate))
   return r
```

二、 多卡并行

什么时候用到多卡分布：希望加快速度，如希望处理 512 个数据，但显卡只能处理 128 个数据时——开多个线程

1 DP

1.1 基本原理

- 多张卡共享权重，分别计算出结果

- 将计算结果汇总在 rank:0 卡上
- 在 rank:0 卡上进行正常的反向传播，获得新的模型权重
- 将权重广播到其他卡上，覆盖原本权重

1.2 示例

假设模型输入为 (32, input_dim)(32 表示 batch_size)；输出为 (32, output_dim)；使用 4 个 GPU 训练：

nn.DataParallel 的作用：

- 将这 32 个样本拆成 4 份，发送给 4 个 GPU 分别做 forward，生成 4 个大小为 (8, output_dim) 的输出
- 将这 4 个输出都收集到 cuda:0 上并合并成 (32, output_dim)

1.3 使用代码

```
model = nn.DataParallel(model)
```

1.4 特点

- 其他卡相当于用来帮忙，真正计算的还是主程序
- 不适用于超大模型，因为每个设备上都要复制整个模型

2 DDP

采用梯度同步机制，让每张卡都进行梯度更新（与 DP 不同）

2.1 关键组件

- **reducer(梯度规约器)**:负责在反向传播过程中聚合所有参数的梯度,并通过 AllReduce 操作在多卡间同步梯度，确保所有进程的模型参数保持一致
- **bucket(梯度桶)**：为了减少通信次数，DDP 将多个梯度分组成 Bucket，按桶（Bucket）为单位进行 AllReduce，而非逐参数同步

2.2 基本原理

- **构造阶段**
  - 从 rank 0 进程广播模型的 state_dict() 给所有其他进程，确保所有模型副本参数一致
  - 每个进程创建一个 Reducer，并将所有模型参数的梯度划分到多个 bucket 中
  - 每个 bucket 会在其中所有参数的梯度就绪后立即触发 allreduce 操作
- **前向传播**

  将输入传入本地模型

  需要提前设置 **find_unused_parameters=True**，以遍历计算图标记未参与反向传播的参数，若设置为 **False**，DDP 会假设每个参数都产生梯度，若有参数未参与反向传播，DDP 会永远等待这些梯度，导致程序卡住。
- **反向传播**
  - 用户调用 loss.backward() 后，DDP 注册的 autograd hook 会在每个参数梯度准备好时被触发
  - Reducer 检测 bucket 中的所有梯度是否准备就绪，若是，则异步启动 allreduce 操作进行梯度平均
  - 所有 bucket 同步完成后，模型的所有副本将具有一致的.grad 值
- **优化阶段**

  每个进程**独立执行** optimizer.step()，由于所有模型初始状态一致，且每轮参数梯度都已全局平均，最终各模型参数仍保持一致

2.3 代码示例

单卡模型修改为多卡训练的示例：

```
import os
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import DataLoader
from torch.utils.data.distributed import DistributedSampler
import torchvision
```

```
import torchvision.transforms as transforms

# 初始化进程组，确保所有GPU进程能互相通信
def setup(rank, world_size):
   dist.init_process_group(
                  backend="nccl", # 使用NVIDIA的NCCL后端（GPU最佳选择）
                  # 通过环境变量初始化（需提前设置MASTER_ADDR和MASTER_PORT）
                  init_method="env://",
                  rank=rank,  # 当前进程的全局排名（0~world_size-1）
                  world_size=world_size , # 总进程数（通常等于GPU数量）
                  )
   torch.cuda.set_device(rank) # 绑定当前进程到对应GPU

# 销毁进程组，释放资源
def cleanup():
   dist.destroy_process_group()

# 获取分布式数据加载器
def get_dataloader(rank, world_size , batch_size):
   transform = transforms.Compose([
                       # 数据增强：随机水平翻转
                       transforms.RandomHorizontalFlip(),
                       transforms.ToTensor(),
                       ])
   train_dataset = torchvision.datasets.CIFAR10(
                  root="./data",
                  train=True,
                  download=True,
                  transform=transform
                  )
   sampler = DistributedSampler(
             train_dataset ,
             num_replicas=world_size , # 总进程数
             rank=rank,
             shuffle=True # 打乱数据
             )
   train_loader = DataLoader(
                  dataset=train_dataset ,
                  batch_size=batch_size ,
                  sampler=sampler ,
```

```
                    num_workers=4, # 数据加载的线程数
                    pin_memory=True, # 锁页内存，加速GPU传输
                    )
    return train_loader

# 训练主函数
def train(rank, world_size , epochs=5, batch_size=64):
    setup(rank, world_size)

    train_loader = get_dataloader(rank, world_size , batch_size)

    # 初始化模型
    model = torchvision.models.resnet18(num_classes=10).to(rank)
    ddp_model = DDP(model, device_ids=[rank])

    # 定义损失函数和优化器
    criterion = nn.CrossEntropyLoss().to(rank)
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.1, momentum=0.9)

    for epoch in range(epochs):
        train_loader.sampler.set_epoch(epoch)
        ddp_model.train()
        total_loss = 0.0

        for images , labels in train_loader:
            images = images.to(rank, non_blocking=True)
            labels = labels.to(rank, non_blocking=True)

            outputs = ddp_model(images)
            loss = criterion(outputs , labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

    # 仅rank 0打印日志（避免重复输出）
    if rank == 0:
        print(f"Epoch␣[{epoch+1}/{epochs}],␣Loss:␣{total_loss:.4f}")
```

```
cleanup() # 清理分布式资源
```

将上述文件保存为 DDP_train.py 后，启动命令：

```
torchrun --nproc-per-node=4 ddp_train.py
```

参数说明：

- –nproc-per-node：每台机器的进程数，通常等于 GPU 数
- –nnodes：总机器数量（默认 1）；
- –node-rank：当前节点编号；
- –rdzv-backend：用于节点发现的方式，默认是 c10d；
- –rdzv-endpoint：rendezvous 的 IP:port 地址，用于多机；

完整多机命令：

```
torchrun --nnodes=2 --node-rank=0 --nproc-per-node=4 \
--rdzv-endpoint=host0:29500 ddp_train.py
```

2.4 特点

- 每个设备独立进行前向传播、反向传播和梯度计算
- 每个设备更新自己的模型参数，使用 AllReduce 来同步每个设备上的梯度
- 在大规模分布式环境下，仍然可能会有一定的通信瓶颈

3 DeepSpeed

DP 和 DDP 每张显卡上都存储了一份完整的参数，产生冗余；DeepSpeed 将优化器参数切分到不同的显卡上来**消除冗余**

3.1 执行过程

执行具体计算时，显存中只有部分权重，此时：

- 用 all_gather 从所有 GPU 上拉取该层的所有权重片段
- 拼成一整块参数，放到当前显卡用于计算
- 计算完之后，立即释放，避免显存占用
- 梯度更新之后将自己负责更新的参数广播到全部 GPU

3.2 使用方法

一般使用 Huggingface 集成方法。