Towards Communication-efficient Vertical Federated Learning Training via Cache-enabled Local Updates

摘要：垂直联合学习（VFL）是一种新兴的范例，它允许不同的各方（例如，组织或企业）协作构建具有隐私保护的机器学习模型。在训练阶段，VFL仅在各方之间交换中间统计信息，即前向激活和后向导数，以计算模型梯度。**然而，由于其地理分布性质，VFL训练通常受到低WAN带宽的影响。**本文介绍了一种新颖有效的VFL训练框架CELU-VFL，该框架利用局部更新技术来减少跨方通信轮次。CELU-VFL缓存过时的统计信息，并重用它们来估计模型梯度，而不交换即时统计信息。提出了提高收敛性能的重要技术。首先，为了处理随机方差问题，我们提出了一种均匀抽样策略来公平地选择用于局部更新的陈旧统计量。其次，为了利用陈旧性带来的误差，我们设计了一种实例加权机制来度量估计梯度的可靠性。理论分析证明，CELU-VFL实现了类似于vanilla VFL训练的次线性收敛速度，但需要的通信轮次少得多。公共工作负载和真实工作负载的经验结果验证了CELU-VFL比现有工作负载快六倍。

WAN带宽的含义：

- **WAN（广域网）**：连接地理上分散的节点（如不同城市、国家的机构）的网络，通常通过公共互联网或专线实现。
- **低带宽**：指网络的最大数据传输速率较低（例如每秒仅能传输几MB数据），导致通信效率低下。

具体示例

假设一家银行（特征方）和一家电商（标签方）联合训练模型：若双方通过跨国WAN连接，网络带宽仅为10Mbps，传输1GB中间数据需约13分钟，而一次训练可能需要数百次迭代，总耗时将难以接受。

贡献

VFL中的本地更新有两个缺点：

（1）方差问题）对多个连续步骤**使用相同的陈旧小批次**将扩大模型梯度中的随机方差（在没有数据洗牌的SGD算法中可以观察到类似的现象）；

（2）近似误差）由于通过陈旧统计进行近似不可避免地将误差引入模型梯度，因此如果直接用近似梯度更新模型，将减慢收敛速度，甚至导致发散。根据上面的分析，VFL的状态与本地更新存在差异——执行更多的本地步骤可以更好地提高系统效率，但很容易损害统计效率。

CELU-VFL的贡献：

- 利用了支持缓存的本地更新技术。引入了工作集表，这个表缓存了前向和反向传播，以支持本地更新
- 为了提高统计效率，CELU-VFL解决了VFL中基于工作集表抽象的本地更新的两个缺点。（1）首先，为了解决方差问题，我们引入了统一的局部采样策略——不是重复使用相同的小批量，而是从工作集表中公平地采样缓存的统计信息以执行局部更新。（2）其次，为了减轻近似误差的影响，我们设计了一种状态感知的实例加权机制。其基本思想是使用余弦相似度来度量过时性，并根据每个实例的过时性来控制其贡献。

方案

符号：

- B：mini-batch的大小
- R： 每个小批次的最大使用次数；
- W : 工作集表的大小（缓存的小批次数）；
- ξ：实例权重机制中的阈值；
- $Z_A^{(i)}, \nabla Z_A^{(i)}$：第$i$个batch，用户$a$的正向推理结果和反向传播梯度，$\nabla Z_A^{(i)} = \frac{\partial L}{\partial Z_A^i}$
- $Z_A^{(i,j)}, \nabla Z_A^{(i,j)}$：第$i$个batch在第$j$次更新时的正向推理结果和反向传播梯度

CELU-VFL的三个主要组成部分

工作集表（workset table）

工作集表负责维护缓存的过时统计信息。它为每个缓存的小批量记录两个“时钟”：插入该批时的时间戳，以及该批完成更新时的iteration次数。

通信线程（communication worker）

通信线程进行正向推理和反向传播，交换$Z_A^{(i)}$，$\nabla Z_A^{(i)}$，把$Z_A^{(i)}$，$\nabla Z_A^{(i)}$存到工作集表中以便进行本地更新

本地线程（local worker）

本地线程通过缓存的统计信息执行本地更新，**本地线程和通信线程可以并行执行。**

与FedBCD不同，CELU-VFL利用循环局部抽样策略来选择不同的mini-batch进行本地更新以降低随机方差，并且引入权因子，以减轻陈旧性带来的影响

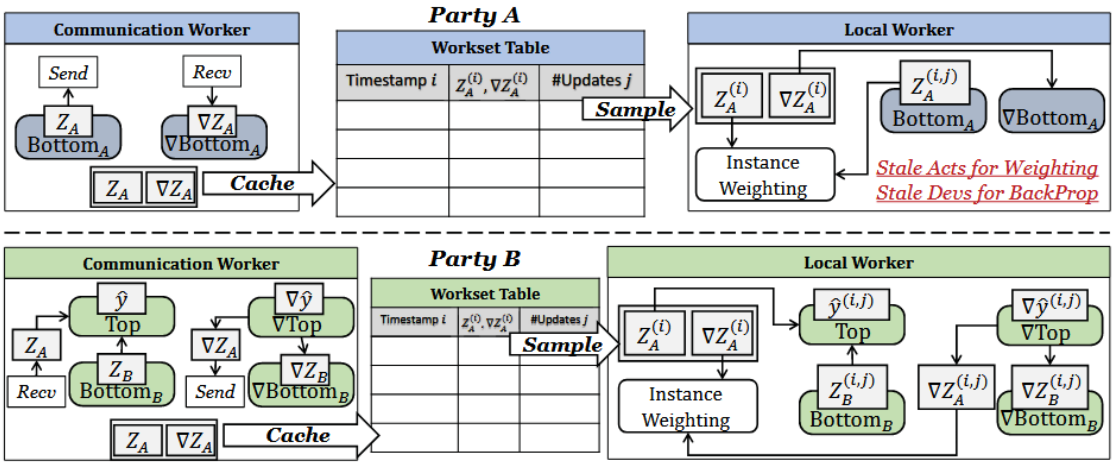
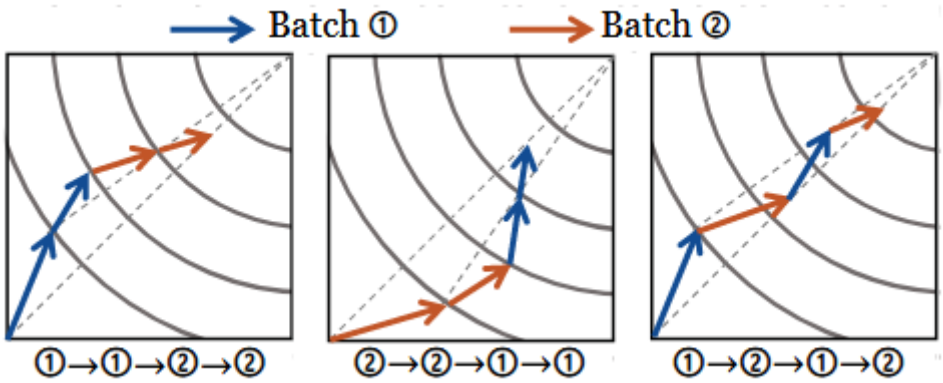


图1 CELU-VFL方案图

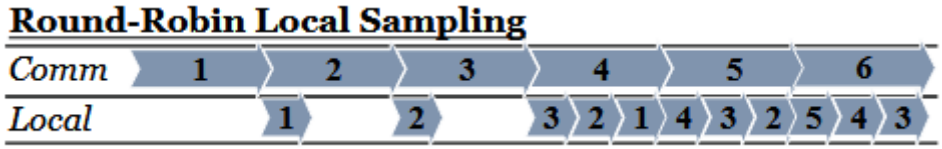
在上图Party B中的本地线程中，首先从工作集表中随机抽取一个batch，然后本地线程进行正向推理，然后利用得到的$\nabla Z_A^{(i)}$和$\nabla Z_A^{(i,j)}$计算权重，得到权重后再进行加权的反向传播

轮询局部采样

如下图所示，如果每个小批次用于连续步骤，则随机方差将累积并阻碍收敛。如果我们均匀地应用小批量，例如通过交替使用小批量，则可以减轻这种随机方差。



该方案设计了一个简单但有效的循环局部采样策略，该策略强调工作集的每个小批次不呢个在接下来的W-1个步骤中再次采样。如下图所示的并行线程。



这种策略相比于FedBCD，在每个小批次的最大使用次数R相同时，CELU-VFL收敛的更快。

权重机制

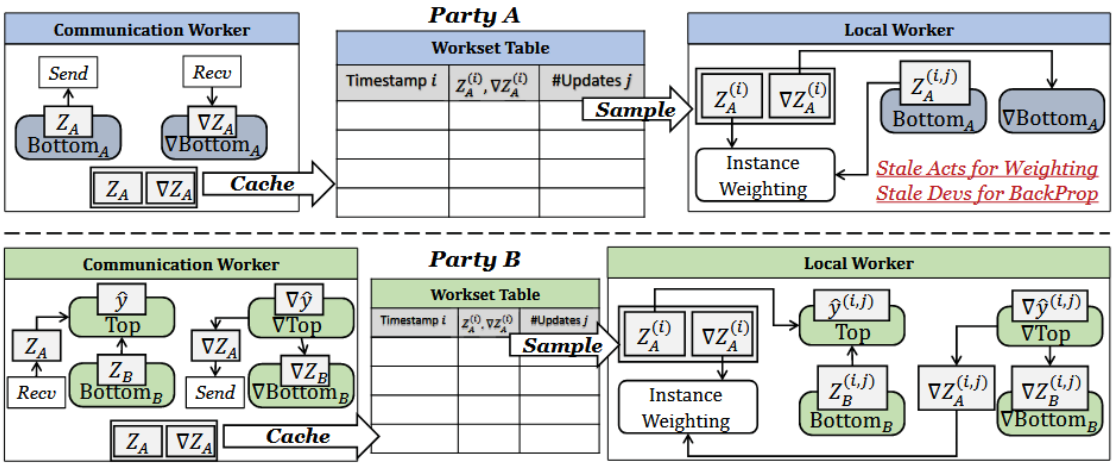


图1 CELU-VFL方案图

尽管已经有各种各样的工作讨论梯度优化算法对梯度中的某些噪声具有鲁棒性，但当陈旧性较大时，模型性能将不可避免地受到损害。

这篇文章使用前向推理和后向导数的余弦相似度来衡量批次的陈旧性。

如图一所示，party A没有标签，因此只能进行正向推理，在本地迭代时，它只能使用抽样的批次进行正向传播，正向传播得到本地输出后$Z_A^{(i,j)}$，$\nabla Z_A^{(i,j)}$与$Z_A^{(i)}$进行余弦相似度（权重）计算。

party B有标签，因此它可以使用工作集表中存储的$Z_A^{(i)}$和本地算出的$Z_B^{(i,j)}$合起来计算梯度$\nabla \hat{y}^{(i,j)}$，然后使用$\nabla Z_A^{(i,j)}$和工作集表的$\nabla Z_A^{(i)}$计算权重。

文中给出的解释：

对于全连接网络

$\theta \in R^{d_{in} \times d_{outs}}$

$$\nabla \theta = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial \theta} = Z_{in}^T \cdot \nabla Z_{out}$$

$$\nabla\tilde{\theta} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial \tilde{\theta}} = Z_{in}^{T} \cdot \nabla\tilde{Z}_{out}$$

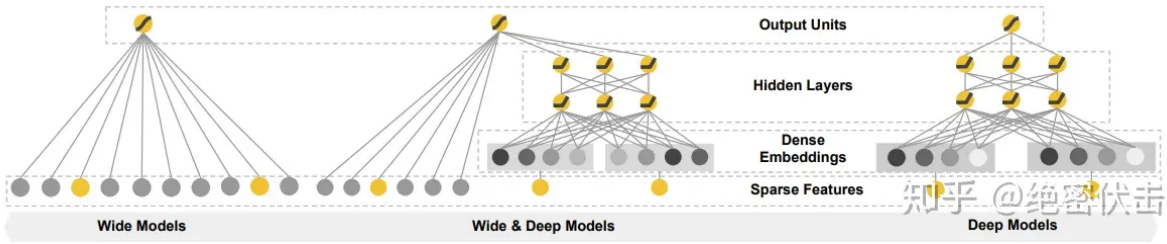
假设$\tilde{\nabla} z_{out}$是旧梯度，当二维矩阵$(\nabla\theta, \tilde{\nabla}\theta)$是平坦的。

$$\cos(\nabla\theta, \tilde{\nabla}\theta) = \cos(\nabla z_{out}, \tilde{\nabla} z_{out})$$

实验设置

使用wdl和dssm推荐模型，这里介绍wdl模型

这个模型时Google在2016年提出的推荐系统点击率预测模型



训练集90000个样本，验证集100000个样本，batch_size设为128，训练集一共被分为7032个batch

工作集表存5个batch，每个batch都会参与5次本地更新

参与者分别是guest和host，**guest有标签**

guest和host输出都是256维的特征向量，双方进行特征提取后，将2者的特征拼接为一个512维的向量，然后通过一层全连接神经网络转为1维，再经过sigmoid函数得到点击率概率。损失函数是交叉熵损失函数，最后除batch_size得到平均损失

可以改进的地方

那个权重计算看看能不能用Shapley值代替

代码

在pytorch中，反向传播之后只会保存模型参数的梯度，其他梯度都会被删除以节省存储空间。若想查看其他节点的梯度，对该节点的tensor使用.retain_grad()保存梯度。

```
####测试backward

def model_test(x,w1,w2,b):                #模型有3个参数，
w1,w2,b
    y = w1*x**2+w2*x+b
    return y

#初始化3个参数，并设置他们的requires_grad属性，若是使用
pytorch自带的linear，Conv2d等函数则不用设置该属性，pytorch会
自动设置这些参数需要梯度
test_params = torch.tensor([1.0, 0.0, 1.0],
requires_grad=True)
```

```
def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()


outputs = model_test(2, *test_params)

outputs.retain_grad()        #使用该函数保存loss对output的求导结果
loss = loss_fn(outputs, 9)   #计算loss

loss.backward()              #求导
print(outputs.grad)          #打印loss对output的求导结果


======输出========
tensor(-8.)                  #loss对output的求导结果就是-8
```

```
import torch
import torch.nn as nn

# 定义两层全连接网络
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(13, 128)  # 输入层到隐藏层
        self.fc2 = nn.Linear(128, 3)    # 隐藏层到输出层
        self.relu = nn.ReLU()

    def forward(self, x):
```

```
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# 初始化模型和输入
model = Net()
x = torch.randn(1, 13)  # 生成一个样本 (batch_size=1, input_dim=13)

# 前向传播
output = model(x)

# 已知损失函数对输出的导数为0.789
output_grad = torch.tensor([[0.789,1.22,0.456]], dtype=torch.float)  # 关键修复: 使用二维张量[1,1]

# 清除现有梯度（重要！）
model.zero_grad()

# 反向传播：将已知梯度传入backward()
output.backward(gradient=output_grad)

# 验证梯度传播结果（可选）
print("梯度检查:")
print(f"输出层权重梯度: {model.fc2.weight.grad}")
print(f"输出层偏置梯度: {model.fc2.bias.grad}")
print(f"隐藏层权重梯度: {model.fc1.weight.grad}")
print(f"隐藏层偏置梯度: {model.fc1.bias.grad}")
```