STC稀疏三元压缩算法复现

1. 数据集介绍

MINIST数据集

MNIST是一个手写体数字的图片数据集，该数据集来由美国国家标准与技术研究所（National Institute of Standards and Technology (NIST)）发起整理，一共统计了来自250个不同的人手写数字图片，其中50%是高中生，50%来自人口普查局的工作人员。该数据集的收集目的是希望通过算法，实现对手写数字的识别。

2. logistic模型

```python
class logistic(nn.Module):
    """
    logistic模型，用于MINIST图片分类预测
    """

    def __init__(self, in_size=32 * 32 * 1, num_classes=10):
        super(logistic, self).__init__()
        self.linear = nn.Linear(in_size, num_classes)

    def forward(self, x):
        out = x.view(x.size(0), -1)
        out = self.linear(out)
        return out
```

3. 分布式培训设备模型

```python
class DistributedTrainingDevice(object):
    '''
    分布式培训设备类（客户端或服务器）
    dataloader: 由数据点（x，y）组成的pytorch数据集
    model: pytorch神经网络
    hyperparameters: 包含所有超参数的python dict
    experiment: 实验类型
    '''

    def __init__(self, dataloader, model, hyperparameters, experiment):
        self.hp = hyperparameters
        self.xp = experiment
        self.loader = dataloader
        self.model = model
        self.loss_fn = nn.CrossEntropyLoss()

    def copy(self, target, source):
        """拷贝超参数，结果保存在target中"""
        for name in target:
            target[name].data = source[name].data.clone()

    def add(self, target, source):
```

```
        """超参数做加法，结果保存在target中"""
        for name in target:
            target[name].data += source[name].data.clone()

    def subtract(self, target, source):
        """超参数做减法，结果保存在target中"""
        for name in target:
            target[name].data -= source[name].data.clone()

    def subtract_(self, target, minuend, subtrahend):
        """超参数做减法(minuend-subtrahend)，结果保存在target中"""
        for name in target:
            target[name].data = minuend[name].data.clone() - subtrahend[name].data.clone()

    def approx_v(self, T, p, frac):
        if frac < 1.0:
            n_elements = T.numel()
            n_sample = min(int(max(np.ceil(n_elements * frac), np.ceil(100 / p))), n_elements)
            n_top = int(np.ceil(n_sample * p))

            if n_elements == n_sample:
                i = 0
            else:
                i = np.random.randint(n_elements - n_sample)

            topk, _ = torch.topk(T.flatten()[i:i + n_sample], n_top)
            if topk[-1] == 0.0 or topk[-1] == T.max():
                return self.approx_v(T, p, 1.0)
        else:
            n_elements = T.numel()
            n_top = int(np.ceil(n_elements * p))
            topk, _ = torch.topk(T.flatten(), n_top)  # 返回列表中最大的n_top个值

        return topk[-1], topk

    def stc(self, T, hp):
        """稀疏三元组压缩算法"""
        hp_ = {'p': 0.001, 'approx': 1.0}
        hp_.update(hp)

        T_abs = torch.abs(T)

        v, topk = self.approx_v(T_abs, hp_["p"], hp_["approx"])
        mean = torch.mean(topk)  # 前n_top的均值

        out_ = torch.where(T >= v, mean, torch.Tensor([0.0]).to(device))  # 大于均值的重新赋值为均值，小于自己的赋值为0
        out = torch.where(T <= -v, -mean, out_)  # 小于副的均值的赋值为-v，大于的赋值为out_对应索引值

        return out

    def compress(self, target, source):
        '''
        分别对每一个超参数进行稀疏三元压缩
```

```
        '''
        for name in target:
            target[name].data = self.stc(source[name].data.clone(), self.hp)
```

4. 客户端模型

```
class Client(DistributedTrainingDevice):
    """
    客户端类，继承分布式培训设备类
    """

    def __init__(self, dataloader, model, hyperparameters, experiment, id_num=0):
        super().__init__(dataloader, model, hyperparameters, experiment)

        self.id = id_num

        # 超参数
        self.W = {name: value for name, value in self.model.named_parameters()}
        self.W_old = {name: torch.zeros(value.shape).to(device) for name, value in self.W.items()}
        self.dW = {name: torch.zeros(value.shape).to(device) for name, value in self.W.items()}
        self.dW_compressed = {name: torch.zeros(value.shape).to(device) for name, value in self.W.items()}
        self.A = {name: torch.zeros(value.shape).to(device) for name, value in self.W.items()}

        self.n_params = sum([T.numel() for T in self.W.values()])
        self.bits_sent = []

        optimizer_object = getattr(optim, self.hp['optimizer'])
        optimizer_parameters = {k: v for k, v in self.hp.items() if k in optimizer_object.__init__.__code__.co_varnames}

        self.optimizer = optimizer_object(self.model.parameters(), **optimizer_parameters)

        # 学习率动态变化
        self.scheduler = getattr(optim.lr_scheduler, self.hp['lr_decay'][0])(self.optimizer, **self.hp['lr_decay'][1])

        # 状态记录
        self.epoch = 0
        self.train_loss = 0.0

    def synchronize_with_server(self, server):
        # W_client = W_server
        self.copy(target=self.W, source=server.W)

    def train_cnn(self, iterations):

        running_loss = 0.0
        for i in range(iterations):

```

```python
            try:  # Load new batch of data
                x, y = next(self.epoch_loader)
            except:  # Next epoch
                self.epoch_loader = iter(self.loader)
                self.epoch += 1

                # 动态调整lr
                if isinstance(self.scheduler, optim.lr_scheduler.LambdaLR):
                    self.scheduler.step()
                if isinstance(self.scheduler, optim.lr_scheduler.ReduceLROnPlateau) and 'loss_test' in self.xp.results:
                    self.scheduler.step(self.xp.results['loss_test'][-1])

                x, y = next(self.epoch_loader)

            x, y = x.to(device), y.to(device)

            self.optimizer.zero_grad()

            y_ = self.model(x)

            loss = self.loss_fn(y_, y)
            loss.backward()
            self.optimizer.step()

            running_loss += loss.item()

        return running_loss / iterations

    def compute_weight_update(self, iterations=1):

        # 设置为训练模式
        self.model.train()

        # W_old = W
        self.copy(target=self.W_old, source=self.W)

        # W = SGD(W, D)
        self.train_loss = self.train_cnn(iterations)

        # dW = W - W_old
        self.subtract_(target=self.dW, minuend=self.W, subtrahend=self.W_old)

    def compress_weight_update_up(self, compression=None, accumulate=False, count_bits=False):

        if accumulate and compression[0] != "none":
            # 超参数压缩，联邦通信优化
            self.add(target=self.A, source=self.dW)
            self.compress(target=self.dW_compressed, source=self.A)
            self.subtract(target=self.A, source=self.dW_compressed)

        else:
            # 没有任何压缩措施
            self.compress(target=self.dW_compressed, source=self.dW, )
```

5. 服务端模型

```python
class Server(DistributedTrainingDevice):
    """
    服务端类，继承分布式培训设备类
    """

    def __init__(self, dataloader, model, hyperparameters, experiment, stats):
        super().__init__(dataloader, model, hyperparameters, experiment)

        # Parameters
        self.W = {name: value for name, value in self.model.named_parameters()}
        self.dW_compressed = {name: torch.zeros(value.shape).to(device) for name, value in self.W.items()}
        self.dW = {name: torch.zeros(value.shape).to(device) for name, value in self.W.items()}

        self.A = {name: torch.zeros(value.shape).to(device) for name, value in self.W.items()}

        self.n_params = sum([T.numel() for T in self.W.values()])
        self.bits_sent = []

        self.client_sizes = torch.Tensor(stats["split"])

    def average(self, target, sources):
        """求超参数平均函数，平均值赋值在target中"""
        for name in target:
            target[name].data = torch.mean(torch.stack([source[name].data for source in sources]), dim=0).clone()

    def aggregate_weight_updates(self, clients, aggregation="mean"):
        # dW = aggregate(dW_i, i=1,..,n)
        self.average(target=self.dW, sources=[client.dW_compressed for client in clients])

    def compress_weight_update_down(self, compression=None, accumulate=False, count_bits=False):
        if accumulate and compression[0] != "none":
            # 对超参数进行稀疏三元压缩
            self.add(target=self.A, source=self.dW)
            self.compress(target=self.dW_compressed, source=self.A)
            self.subtract(target=self.A, source=self.dW_compressed)

        else:
            self.compress(target=self.dW_compressed, source=self.dW)

        self.add(target=self.W, source=self.dW_compressed)

    def evaluate(self, loader=None, max_samples=50000, verbose=True):
        """评估服务端全局模型的训练效果"""
        self.model.eval()

        eval_loss, correct, samples, iters = 0.0, 0, 0, 0
```

```
        if not loader:
            loader = self.loader
        with torch.no_grad():
            for i, (x, y) in enumerate(loader):

                x, y = x.to(device), y.to(device)
                y_ = self.model(x)
                _, predicted = torch.max(y_.data, 1)
                eval_loss += self.loss_fn(y_, y).item()
                correct += (predicted == y).sum().item()
                samples += y_.shape[0]
                iters += 1

                if samples >= max_samples:
                    break
            if verbose:
                print("Evaluated on {} samples ({} batches)".format(samples, iters))

            results_dict = {'loss': eval_loss / iters, 'accuracy': correct / samples}

        return results_dict
```

6. 图片数据集DataLoader类

```
class CustomImageDataset(Dataset):
    '''
    图片数据集DataLoader类
    inputs : numpy array [n_data x shape]
    labels : numpy array [n_data (x 1)]
    '''

    def __init__(self, inputs, labels, transforms=None):
        assert inputs.shape[0] == labels.shape[0]
        self.inputs = torch.Tensor(inputs)
        self.labels = torch.Tensor(labels).long()
        self.transforms = transforms

    def __getitem__(self, index):
        img, label = self.inputs[index], self.labels[index]

        if self.transforms is not None:
            img = self.transforms(img)

        return (img, label)

    def __len__(self):
        return self.inputs.shape[0]
```

7. MNIST数据下载与标准化

```
def get_mnist():
    '''下载mnist数据集数据'''
```

```python
    data_train = torchvision.datasets.MNIST(root=os.path.join(DATA_PATH, "MNIST"), train=True, download=True)
    data_test = torchvision.datasets.MNIST(root=os.path.join(DATA_PATH, "MNIST"), train=False, download=True)

    x_train, y_train = data_train.train_data.numpy().reshape(-1, 1, 28, 28) / 255, np.array(data_train.train_labels)
    x_test, y_test = data_test.test_data.numpy().reshape(-1, 1, 28, 28) / 255, np.array(data_test.test_labels)

    return x_train, y_train, x_test, y_test

def get_default_data_transforms(name, train=True, verbose=True):
    """数据集标准化处理函数"""
    transforms_train = {
        'mnist': transforms.Compose([
            transforms.ToPILImage(),
            transforms.Resize((32, 32)),
            # transforms.RandomCrop(32, padding=4),
            transforms.ToTensor(),
            transforms.Normalize((0.06078,), (0.1957,))
        ]),
    }
    transforms_eval = {
        'mnist': transforms.Compose([
            transforms.ToPILImage(),
            transforms.Resize((32, 32)),
            transforms.ToTensor(),
            transforms.Normalize((0.06078,), (0.1957,))
        ]),
    }

    if verbose:
        print("\nData preprocessing: ")
        for transformation in transforms_train[name].transforms:
            print(' -', transformation)
        print()

    return (transforms_train[name], transforms_eval[name])
```

8. 数据集分配

```python
def split_image_data(data, labels, n_clients=10, classes_per_client=10, shuffle=True, verbose=True, balancedness=None):
    '''
    分割数据集
    data : [n_data x shape]
    labels : [n_data (x 1)] from 0 to n_labels
    '''
    # constants
    n_data = data.shape[0]
    n_labels = np.max(labels) + 1

    if balancedness >= 1.0:
        data_per_client = [n_data // n_clients] * n_clients
```

```
        data_per_client_per_class = [data_per_client[0] // classes_per_client] * n_clients
    else:
        fracs = balancedness ** np.linspace(0, n_clients - 1, n_clients)
        fracs /= np.sum(fracs)
        fracs = 0.1 / n_clients + (1 - 0.1) * fracs
        data_per_client = [np.floor(frac * n_data).astype('int') for frac in fracs]

        data_per_client = data_per_client[::-1]

        data_per_client_per_class = [np.maximum(1, nd // classes_per_client) for nd in data_per_client]

    if sum(data_per_client) > n_data:
        print("Impossible Split")
        exit()

    # sort for labels
    data_idcs = [[] for i in range(n_labels)]
    for j, label in enumerate(labels):
        data_idcs[label] += [j]
    if shuffle:
        for idcs in data_idcs:
            np.random.shuffle(idcs)

    # split data among clients
    clients_split = []
    c = 0
    for i in range(n_clients):
        client_idcs = []
        budget = data_per_client[i]
        c = np.random.randint(n_labels)
        while budget > 0:
            take = min(data_per_client_per_class[i], len(data_idcs[c]), budget)

            client_idcs += data_idcs[c][:take]
            data_idcs[c] = data_idcs[c][take:]

            budget -= take
            c = (c + 1) % n_labels

        clients_split += [(data[client_idcs], labels[client_idcs])]

    return clients_split
```

9. 读取数据集

```
def get_data_loaders(hp, verbose=True):
    """获取数据集的dataloader形式"""
    x_train, y_train, x_test, y_test = get_mnist()  # 获取数据集

    transforms_train, transforms_eval = get_default_data_transforms(hp['dataset'], verbose=False)  # 数据集标准化处理

```

```
    split = split_image_data(x_train, y_train, n_clients=hp['n_clients'],
                             classes_per_client=hp['classes_per_client'],
balancedness=hp['balancedness'],
                             verbose=verbose)  # 根据客户端分割数据集
    # 建立数据集的Dataloader
    client_loaders = [torch.utils.data.DataLoader(CustomImageDataset(x, y,
transforms_train),

 batch_size=hp['batch_size'], shuffle=True) for x, y in split]
    train_loader = torch.utils.data.DataLoader(CustomImageDataset(x_train,
y_train, transforms_eval), batch_size=100,
                                               shuffle=False)
    test_loader = torch.utils.data.DataLoader(CustomImageDataset(x_test,
y_test, transforms_eval), batch_size=100,
                                              shuffle=False)

    stats = {"split": [x.shape[0] for x, y in split]}

    return client_loaders, train_loader, test_loader, stats
```

10. 模型训练

```
def train():
    hp = {
        "communication_rounds": 20,
        "dataset": "mnist",
        "n_clients": 50,
        "classes_per_client": 10,
        "local_iterations": 1,
        "weight_decay": 0.0,
        "optimizer": "SGD",
        "log_frequency": -100,
        "count_bits": False,
        "participation_rate": 1.0,
        "balancedness": 1.0,
        "compression_up": ["stc", {"p": 0.001}],
        "compression_down": ["stc", {"p": 0.002}],
        "accumulation_up": True,
        "accumulation_down": True,
        "aggregation": "mean",
        'type': 'CNN', 'lr': 0.04,
        'batch_size': 100,
        'lr_decay': ['LambdaLR', {'lr_lambda': lambda epoch: 1.0}],
        'momentum': 0.0,
    }
    xp = {
        "iterations": 100,
        "participation_rate": 0.5,
        "momentum": 0.9,
        "compression": [
            "stc_updown",
            {
                "p_up": 0.001,
                "p_down": 0.002
            }
        ],
```

```python
        "log_frequency": 30,
        "log_path": "results/trash/"
    }
    # 加载数据集并根据客户端来进行划分
    client_loaders, train_loader, test_loader, stats = get_data_loaders(hp)
    # 初始化服务器与客户端的神经网络模型
    net = logistic()
    clients = [Client(loader, net, hp, xp, id_num=i) for i, loader in enumerate(client_loaders)]
    server = Server(test_loader, net, hp, xp, stats)
    # 开始训练
    print("Start Distributed Training..\n")
    t1 = time.time()
    for c_round in range(1, hp['communication_rounds'] + 1):
        # 随机选择一定的客户端来训练
        participating_clients = random.sample(clients, int(len(clients) * hp['participation_rate']))
        # 客户端
        for client in participating_clients:
            client.synchronize_with_server(server)  # 加载当前全局模型参数
            client.compute_weight_update(hp['local_iterations'])  # 权重更性
            client.compress_weight_update_up(compression=hp['compression_up'], accumulate=hp['accumulation_up'],
                                             count_bits=hp["count_bits"])  # 超参数压缩，联邦通信优化

        # 服务端
        server.aggregate_weight_updates(participating_clients, aggregation=hp['aggregation'])  # 聚集客户端的权重
        server.compress_weight_update_down(compression=hp['compression_down'], accumulate=hp['accumulation_down'],
                                           count_bits=hp["count_bits"])  # 超参数压缩，联邦通信优化
        # 全局模型评估
        print("Evaluate...")
        results_train = server.evaluate(max_samples=5000, loader=train_loader)
        results_test = server.evaluate(max_samples=10000)
        # 日志情况
        print({'communication_round': c_round, 'lr': clients[0].optimizer.__dict__['param_groups'][0]['lr'],
                'epoch': clients[0].epoch, 'iteration': c_round * hp['local_iterations']})
        print({'client{}_loss'.format(client.id): client.train_loss for client in clients})

        print({key + '_train': value for key, value in results_train.items()})
        print({key + '_test': value for key, value in results_test.items()})

        print({'time': time.time() - t1})
        total_time = time.time() - t1
        avrg_time_per_c_round = (total_time) / c_round
        e = int(avrg_time_per_c_round * (hp['communication_rounds'] - c_round))
```

```python
        print("Remaining Time (approx.):", '{:02d}:{:02d}:{:02d}'.format(e // 3600, (e % 3600 // 60), e % 60),
              "[{:.2f}%]\n".format(c_round / hp['communication_rounds'] * 100))
```

11. 运行结果

```
Evaluated on 10000 samples (100 batches)
{'communication_round': 1, 'lr': 0.04, 'epoch': 1, 'iteration': 1}
{'client0_loss': 0.6224172711372375, 'client1_loss': 0.8087294101715088, 'client2_loss': 0.5417265892028809, 'client3_loss': 1.158571720123291, 'client4_loss': 1.4353818893432617,
{'loss_train': 0.7287373006343841, 'accuracy_train': 0.7918}
{'loss_test': 0.7628428570926189, 'accuracy_test': 0.7903}
{'time': 9.199519634246826}
Remaining Time (approx.): 00:02:54 [5.00%]

Evaluate...
Evaluated on 5000 samples (50 batches)
Evaluated on 10000 samples (100 batches)
{'communication_round': 2, 'lr': 0.04, 'epoch': 1, 'iteration': 2}
{'client0_loss': 0.4170497953891754, 'client1_loss': 0.475925475358963, 'client2_loss': 0.5516355633735657, 'client3_loss': 0.8919769525527954, 'client4_loss': 0.3647277951240539б,
{'loss_train': 0.36639190673828126, 'accuracy_train': 0.8964}
{'loss_test': 0.37969540430232884, 'accuracy_test': 0.8915}
{'time': 18.243558406829834}
Remaining Time (approx.): 00:02:44 [10.00%]

Evaluate...
Evaluated on 5000 samples (50 batches)
Evaluated on 10000 samples (100 batches)
{'communication_round': 3, 'lr': 0.04, 'epoch': 1, 'iteration': 3}
{'client0_loss': 0.37829846143722534, 'client1_loss': 0.3302837610244751, 'client2_loss': 0.3939962387084961, 'client3_loss': 0.43883463740348816, 'client4_loss': 0.500298559665679
{'loss_train': 0.33650437742471695, 'accuracy_train': 0.9012}
{'loss_test': 0.34531043529510497, 'accuracy_test': 0.8984}
{'time': 29.232567310333252}
Remaining Time (approx.): 00:02:45 [15.00%]
```