《人工智能安全》实验报告

Lab 3：模拟联邦学习

实验总览

本地模拟**联邦学习**过程（数据集实际上是放在了本地，而不是真正的跨域传输）

并且，这所有使用的两个数据集以及用于训练的模型属于经典的 $Computer \quad Vision(CV)$ 识别任务的范畴

文件组织

```
└── 联邦学习
    ├── data
    |   ├── cifar-10-batches-py # cifar数据集
    |   |   └── ...
    |   └── MNIST # mnist数据集
    |       └── ...
    ├── utils
    |   └── conf.json # 配置文件： 提供参数配置，如数据集选择、模型选择、客户端数量、抽样数量等
    └── catkin_ws_backup
    ├── client.py   # 客户端，用于本地训练
    ├── datasets.py
    ├── main.py
    ├── models.py   # 若干模型
    └── server.py   # 服务端
```

在改文件夹目录下的命令行输入

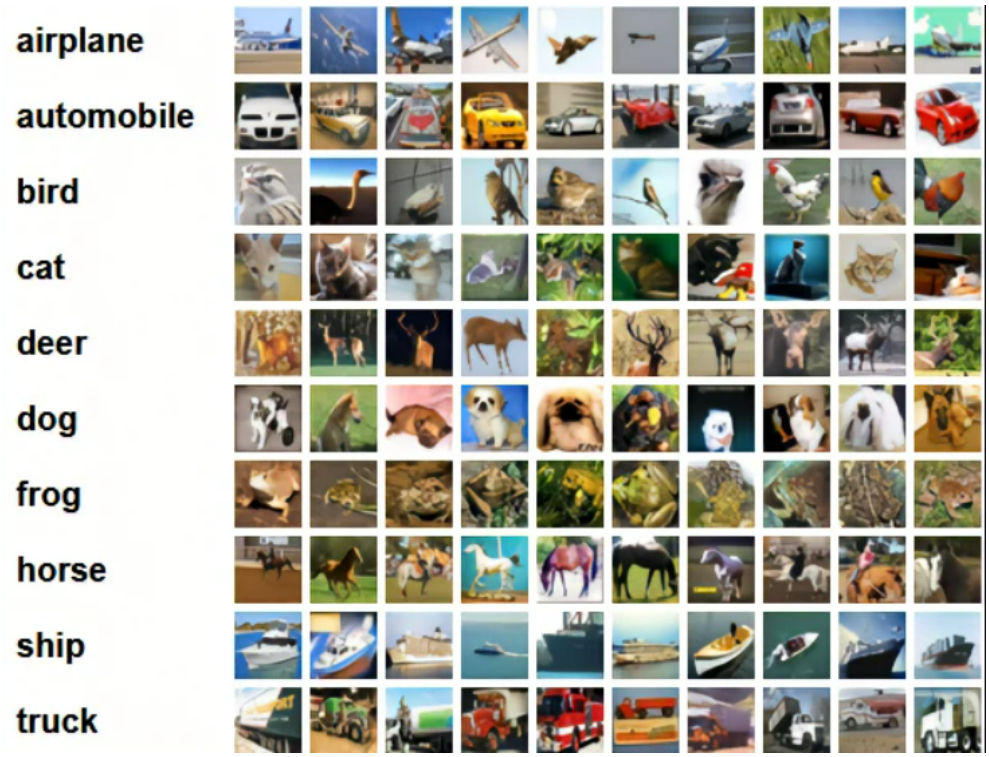
```
python main.py -c utils/conf.json
```

即可调用配置文件运行程序

数据集

CIFAR-10

其中包含10类32x32彩色图像，每类含有6000个图像。其中取50000个训练图像和10000个测试图像。这10类都是各自独立的，不会出现重叠。将数据集分为五个训练batch和一个测试batch，每个batch含有10000个图像。测试batch从各个类中随机抽取1000个图像。训练batch以随机顺序抽取剩余图像

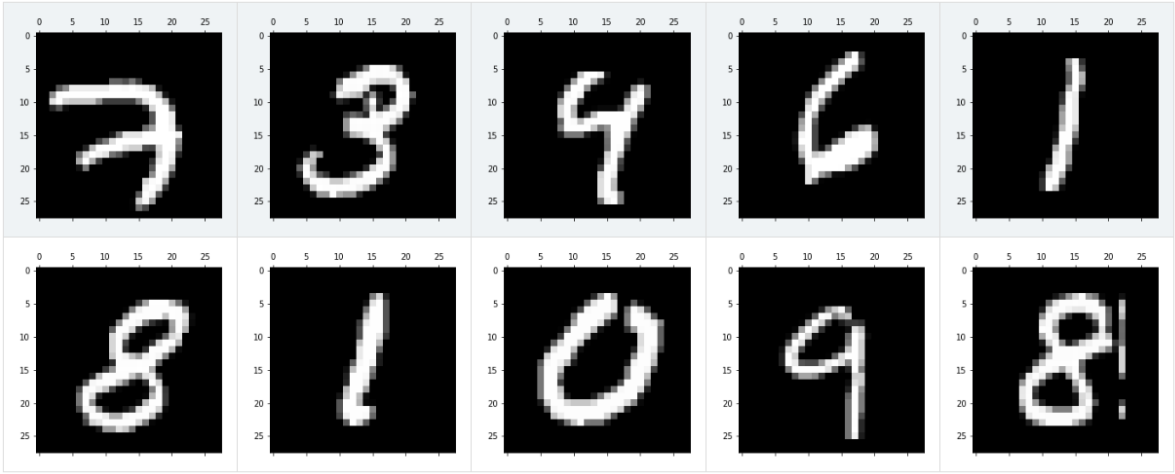


以下给出数据集文件内部结构，如上所说，其中办包含五个训练batch和一个测试batch:

文件名称	大小	说明
`test_batch`	31.0M	10000个测试图像
`data_batch_1`	31.0M	10000个测试图像
`data_batch_2`	31.0M	10000个测试图像
`data_batch_3`	31.0M	10000个测试图像
`data_batch_4`	31.0M	10000个测试图像
`data_batch_5`	31.0M	10000个测试图像

MNIST

上一个lab中也用到了该数据集，MNIST数据集一共有7万张图片，其中6万张是训练集，1万张是测试集。每张图片是28×28的$0-9$的手写数字图片组成。每个图片是黑底白字的形式，黑底用0表示，白字用0-1之间的浮点数表示，越接近1，颜色越白



在`conf.json`文件中

```
    "type" : "cifar",
```

这意味着实际上只使用了`cifar`数据，而没有用到`MNIST`

训练模型

在`model.py`中，我们可以直接使用`torchvision`封装好的几种目标识别的经典CV模型,具体如下：

```
import torch
from torchvision import models

def get_model(name="`VGG`16", pretrained=True):
    if name == "resnet18":
        model = models.resnet18(pretrained=pretrained)
    elif name == "resnet50":
        model = models.resnet50(pretrained=pretrained)
    elif name == "densenet121":
        model = models.densenet121(pretrained=pretrained)
    elif name == "alexnet":
        model = models.alexnet(pretrained=pretrained)
    elif name == "`VGG`16":
        model = models.`VGG`16(pretrained=pretrained)
    elif name == "`VGG`19":
        model = models.`VGG`19(pretrained=pretrained)
    elif name == "inception_v3":
        model = models.inception_v3(pretrained=pretrained)
    elif name == "googlenet":
        model = models.googlenet(pretrained=pretrained)

    # 是否配置cuda
    if torch.cuda.is_available():
        return model.cuda()
    else:
        return model
```

在`conf.json`中

```
"model_name" : "resnet18"
```

这意味着本实验只调用了`ResNet18`模型，而实际上没有调用其他的模型

实验原理

Federated Learning（联邦学习）

> 参考论文:
>
> Vertical Federated Learning
>
> Federated learning: Challenges, methods, and future directions
>
> A secure federated transfer learning framework
>
> Federated machine learning: Concept and applications

联邦学习是近年来兴起的一种**分布式机器学习技术**，其提出背景是现实生活中**数据难以集中管理**、**隐私安全问题**突出以及机器学习算法本身的局限性

1. 数据难以集中管理
2. 隐私安全问题（可以和我们正课的人工智能隐私性相联系）

在传统的集中式机器学习中，数据通常是集中存储在一个中心化服务器上的，而在联邦学习中，各个参与方（如设备、机构、用户）可以**共同训练模型**，而不需要把数据集中起来。这种**分散式的训练模式**，既能避免中央服务器集中存储数据带来的隐私泄露风险，也可以更好地利用分散的数据资源

横向联邦学习

根据我们的`main.py`，可以判断本次实验的联邦学习属于纵向联邦学习

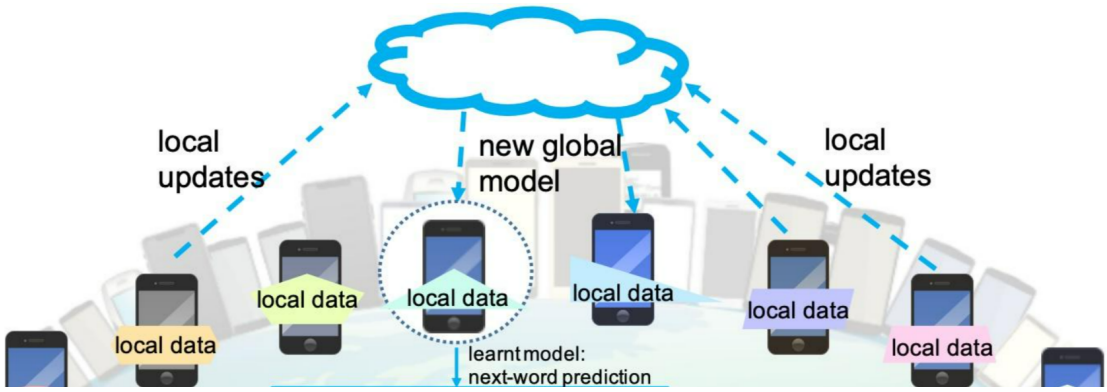
每个参与者之间的数据源之间的**特征相同**，但数据**分布不同**

横向联邦学习可以看作是一种**数据并行**的分布式机器学习框架。即每个用户设备上都有一个用户自己的数据集，用户设备上持有的**用户数据是不同的**，而每个用户设备都有训练**一个相同的模型**。这个模型与服务器上的全局模型是一致的

每个客户端在**本地**用自己的数据去**训练**这个模型，得到的模型参数通过网络信道**发送给服务器**。服务器对所有（或部分）用户设备的模型进行**聚合**，得到精度较高的、符合多个用户利益的**全局模型**

训练步骤为：

1. 各个用户设备从服务器**下载**最新的模型
2. 每个用户设备用自己的数据在本地进行模型**训练**
3. 各个用户设备将本地迭代好的模型**上传**给服务器
4. 服务器收集发送上来的所有模型，进行**模型聚合**。
5. 继续**重复**上述步骤，直至训练停止



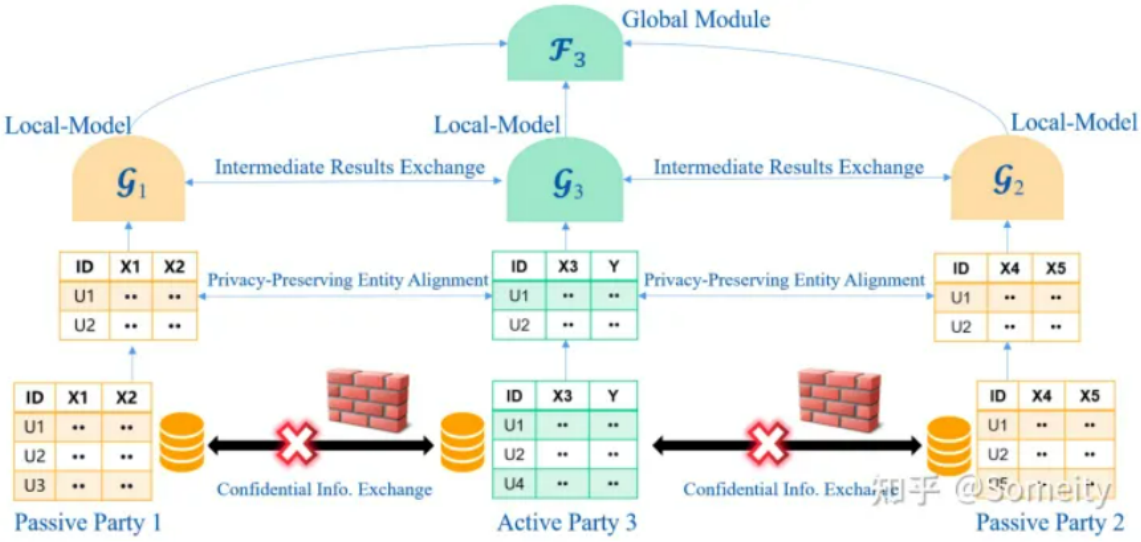
迁移联邦学习

迁移联邦学习：当每个参与者之间的数据源之间的**特征部分重叠**，数据**分布也部分重叠**时，以提高模型的泛化能力。例如，将在一个数据分布上训练的模型应用于另一个数据分布上，以提高模型在新数据上的表现

迁移联邦学习，实际上涉及到了机器学习模型的**泛化性**和**迁移性**问题

纵向联邦学习

纵向联邦学习本质上是多个参与方共同完成一件事情，把这件事情拆成多个部分去逐一分配给每个参与方。可以看作是一种**模型并行**的分布式机器学习框架。即每个用户设备上都有模型的一个部分，训练时需要用同一批量数据去让每个用户逐一得出自己那一部分的结果，然后按顺序将最终结果推导出来



训练步骤一般如下：

1. 所有用户设备进行**数据对齐**，选择同一个批量的数据。
2. 将数据按特征分成不同的切片，每个切片**下放**给对应的用户设备。
3. 用户设备拿到自己那部分的数据特征，用本地的模型分片进行**训练。**
4. 全局模型用每一个模块**聚合**完整的模型
5. 全局模型将每个模块的更新方式**下放**给每个参与者，参与者们完成本地更新
6. 重复以上步骤，直至训练结束

main.py

整个联邦学习的过程由一个服务器和多个客户端组成，这些客户端通过串联协作来训练一个全局模型

每一轮的迭代，服务端会从当前的客户端集合中随机挑选一部分参与本轮迭代训练，被选中的客户端调用本地训练接口 `local_train` 进行本地训练．最后服务端调用模型聚合函数 `model_aggregate` 来更新全局模型

具体代码的解释见下：

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Federated Learning')
    parser.add_argument('-c', '--conf', dest='conf')
    args = parser.parse_args()
    with open(args.conf, 'r') as f:
        conf = json.load(f)
```

```
    train_datasets, eval_datasets = datasets.get_dataset("./data/",
conf["type"])
    server = Server(conf, eval_datasets)
    clients = []
    for c in range(conf["no_models"]):
        clients.append(Client(conf, server.global_model, train_datasets, c))
    print("\n\n")
    # 在每个循环的周期中，随机选择一个客户端子集。这些被选中的客户参与到训练过程中
    for e in range(conf["global_epochs"]):
        candidates = random.sample(clients, conf["k"])
        # 服务器初始化一个权重累加器字典，以汇总来自客户端的模型更新
        # 它对全局模型的参数进行迭代，并初始化权重累积器字典中的相应条目
        weight_accumulator = {}
        # 对于每个选定的客户，它使用全局模型进行局部训练。客户端计算其本地模型和全局模型之间的
差异，并返回更新的模型参数
        for name, params in server.global_model.state_dict().items():
            # 服务器将从客户端收到的模型差异累积到权重累积器字典中
            weight_accumulator[name] = torch.zeros_like(params)
        for c in candidates:
            diff = c.local_train(server.global_model)
            for name, params in server.global_model.state_dict().items():
                weight_accumulator[name].add_(diff[name])
        # 在所有选定的客户端都贡献了他们的更新后，服务器通过将累积的差异加入全局模型的参数来执
行模型聚合
        server.model_aggregate(weight_accumulator)
        # 评估全局模型的性能，获得准确性和损失指标
        acc, loss = server.model_eval()
        print("Epoch %d, acc: %f, loss: %f\n" % (e, acc, loss))
```

server.py

联邦学习系统的服务器端。由一个类`class Server(object)`组成，作为中心服务器，该类的方法主要有3个

- 初始化： 服务器类
- 用配置文件和评估数据集进行初始化，配置文件`conf`包含联邦学习过程的各种设置

```
def __init__(self, conf, eval_dataset):
    self.conf = conf
    self.global_model = models.get_model(self.conf["model_name"])
    self.eval_loader = torch.utils.data.DataLoader(eval_dataset,
batch_size=self.conf["batch_size"], shuffle=True)
```

- 模型聚合： `model_aggregate`方法聚合了从客户节点收到的模型更新。它在全局模型的参数上进行迭代，并根据配置中指定的比例系数来组合更新

```
def model_aggregate(self, weight_accumulator):
    for name, data in self.global_model.state_dict().items():
        update_per_layer = weight_accumulator[name] * self.conf["lambda"]
        if data.type() != update_per_layer.type():
            data.add_(update_per_layer.to(torch.int64))
        else:
            data.add_(update_per_layer)
```

- 模型评估： `model_eval`方法评估全局模型在评估数据集上的性能。它通过迭代评估数据集，计算预测值，并将其与目标标签进行比较，来计算准确度和损失值

```
def model_eval(self):
    self.global_model.eval()
    total_loss = 0.0
    correct = 0
    dataset_size = 0
    for batch_id, batch in enumerate(self.eval_loader):
        data, target = batch
        dataset_size += data.size()[0]
        if torch.cuda.is_available():
            data = data.cuda()
            target = target.cuda()
        output = self.global_model(data)
        total_loss += torch.nn.functional.cross_entropy(output, target,
reduction='sum').item() # sum up batch loss
        pred = output.data.max(1)[1]  # get the index of the max log-
probability
        correct += pred.eq(target.data.view_as(pred)).cpu().sum().item()
    acc = 100.0 * (float(correct) / float(dataset_size))
    total_l = total_loss / dataset_size
    return acc, total_l
```

参数表如下：

Parameter	Description
eval_dataset	用于评估全局模型性能的评估数据集
batch_size	数据加载器的batch大小
model_name	使用的模型（可选模型具体可见于`model.py`）
lambda	在模型聚合过程中，控制客户端更新的贡献的缩放系数

client.py

定义了`class Client(object)`，它封装了与本地训练和模型更新相关的功能

- 初始化

客户端在训练数据集中的部分是通过根据客户端的ID选择一个索引子集而获得的。这确保了每个客户都能在训练数据集的**特定部分上操作**

```
def __init__(self, conf, model, train_dataset, id = -1):
    self.conf = conf
    self.local_model = models.get_model(self.conf["model_name"])
    self.client_id = id
    self.train_dataset = train_dataset
    all_range = list(range(len(self.train_dataset)))
    data_len = int(len(self.train_dataset) / self.conf['no_models'])
    train_indices = all_range[id * data_len: (id + 1) * data_len]
    self.train_loader = torch.utils.data.DataLoader(self.train_dataset,
batch_size=conf["batch_size"],

sampler=torch.utils.data.sampler.SubsetRandomSampler(train_indices))
```

- 使用提供的全局模型对客户的局部模型进行局部训练

使用`clone`和`copy_`方法将全局模型的参数复制到客户的本地模型。这确保了客户以与全局模型相同的初始模型参数开始。使用随机梯度下降（SGD）作为优化算法训练模型，在完成局部训练历时后，客户的局部模型参数和全局模型参数之间的差异被计算出来并存储在一个字典中，然后将差异作为`local_train`方法的输出被返回

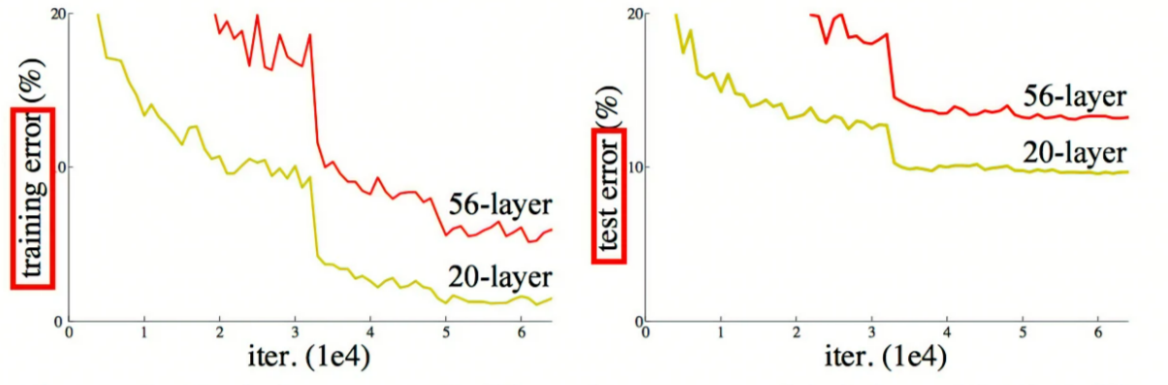
```
def local_train(self, model):
    for name, param in model.state_dict().items():
        self.local_model.state_dict()[name].copy_(param.clone())
    optimizer = torch.optim.SGD(self.local_model.parameters(),
lr=self.conf['lr'],
                                momentum=self.conf['momentum'])
    self.local_model.train()
    for e in range(self.conf["local_epochs"]):
    for batch_id, batch in enumerate(self.train_loader):
        data, target = batch
        if torch.cuda.is_available():
            data = data.cuda()
            target = target.cuda()
        optimizer.zero_grad()
        output = self.local_model(data)
        loss = torch.nn.functional.cross_entropy(output, target)
        loss.backward()

        optimizer.step()
        print("Epoch %d done." % e)
    diff = dict()
    for name, data in self.local_model.state_dict().items():
        diff[name] = (data - model.state_dict()[name])
    return diff
```

dataset.py

主要负责数据的选择、切片、归一化等操作，不再赘述

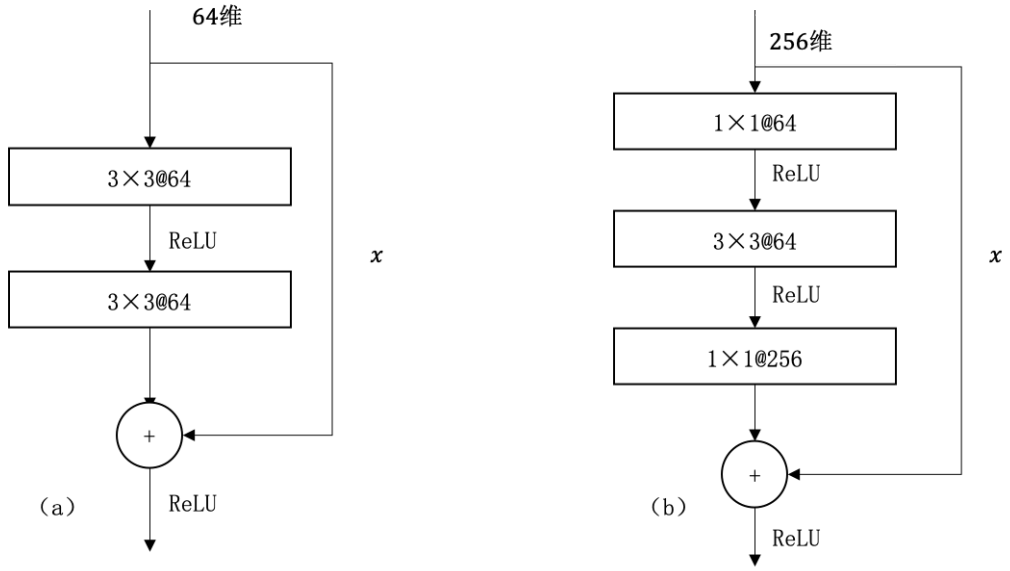
ResNet(残差神经网络)

> 参考论文：Deep Residual Learning for Image Recognition

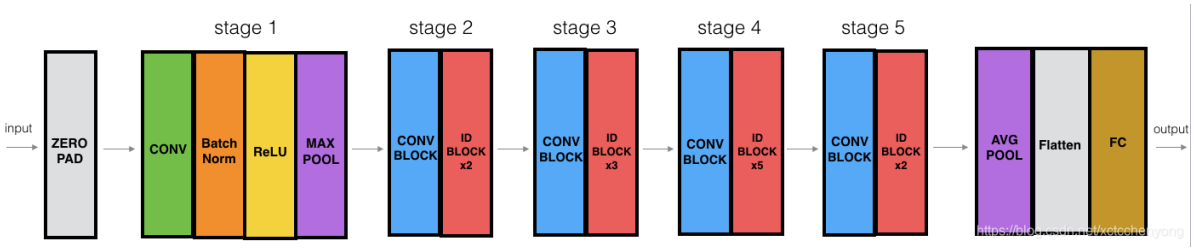


从图中可以看出网络深度达到一定程度时，深层网络的总体表现不如浅层网络，这种现象称为**退化问题(degradation problem)**

针对退化问题，`ResNet`分别构建了带有“快捷连接（Shortcut Connection）”的`ResNet`构建块、以及降采样的`ResNet`构建块，区降采样构建块的主杆分支上增加了一个1×1的卷积操作，见下图：



ResNet模型的架构图，仿照AlexNet的8层网络结构，我们也将ResNet划分成8个构建层（Building Layer）。一个构建层可以包含一个或多个网络层、以及一个或多个构建块（如ResNet构建块）



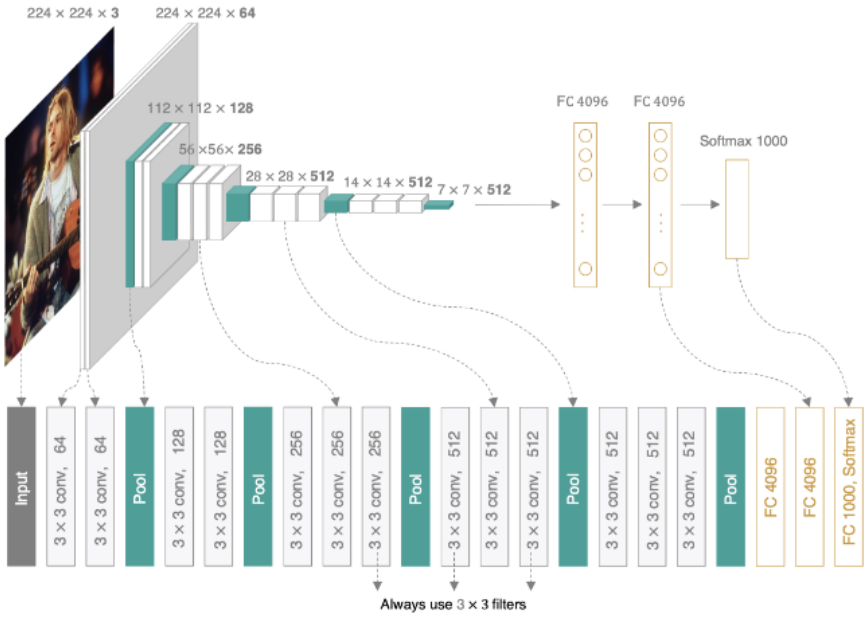
- 第一个构建层，由1个普通卷积层和最大池化层构建
- 第二个构建层，由3个残差模块构成
- 第三、第四、第五构建层，都是由降采样残差模块开始，紧接着3个、5个、2个残差模块

VGG（深卷积残差网络）

> 参考论文: Very Deep Convolutional Networks for Large-Scale Image Recognition

`VGG`的出现时间早于`ResNet`

`VGG`可以看成是加深版的`AlexNet`，整个网络由卷积层和全连接层叠加而成，和AlexNet不同的是，VGG中使用的都是小尺寸的卷积核(3×3)，其网络架构如下图所示：



- 每经过一个pooling层, 通道数目翻倍, 因为pooling会丢失某些信息, 所以通过增加通道数来弥补, 但是增加到512后, 便不再增加了, 可能是考虑到算力, 内存的问题
- FC全连接层, 最后输出的是1000个类的分类结果看D, E, 加卷积层都是在后面加, 因为前面的图片太大, 运算起来太耗时, maxpooling后, 神经图变小
- 纵向地看, 可以看到神经网络是由浅到深的. VGGNet采用递进式训练, 可以先训练浅层网络A, 然后用A初始化好的参数去训练后面深层的网络

`Resnet`在精度、速度上都碾压`VGG`， 但是`VGG`网络还存在以下无可替代的优点

1. 结构简洁，整个网络都是用了同样大小的卷积核（$3*3$）和最大池化尺寸（$2*2$）；
2. 几个小滤波器（$3*3$）卷积层的组合要比一个大滤波器（$5*5$或$7*7$）卷积层好；
3. 验证了通过不断加深网络结构可以提升性能；

实验结果分析

ResNet18 和 VGG16、VGG19 模型效果对比

`VGG19`的使用较为简单，直接`conf.json`中的`model选项`从`resnet18`变为`vgg19`即可

我直接采用了CPU进行训练（我的CPU是`Intel Core i7-11800H`），`ResNet18`训练时长为$1h47m$, 而`VGG19`达到了$17h12m$

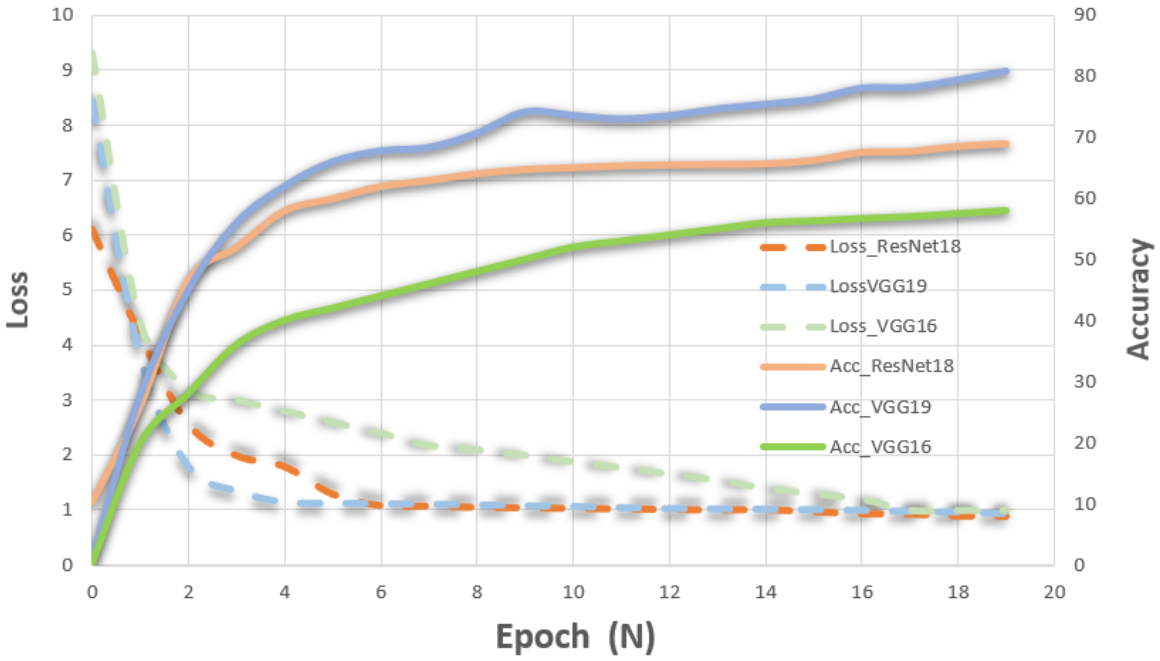
`VGG19`输出

```
Epoch 0, acc: 1.050000, loss: 8.403108
Epoch 1, acc: 27.740000, loss: 3.787284
...
```

`VGG16`输出

```
Epoch 0, acc: 0.150000, loss: 9.103875
Epoch 1, acc: 19.840000, loss: 4.413907
...
```

将训练过程作图可视化：



观察上图可以看出：

`VGG19` 比 `ResNet18` 的准确率高15%左右，但是在 `CIFAR` 数据集上，基于两个模型的特点：

1. 模型深度：`VGG19` 比 `ResNet18` 更深，它有19个卷积层和全连接层，而 `ResNet18` 只有18个层。在 `CIFAR` 这样相对较小的数据集上，更深的模型往往会导致过拟合问题。`ResNet18` 相对较浅，更适合 `CIFAR` 这样的小型数据集，因为它有更少的参数，可以更好地泛化
2. 参数效率：`ResNet18` 使用了残差连接，这种连接方式使得信息在层之间更容易传递，减轻了梯度消失的问题
3. 计算效率：`VGG19` 具有更多的层和参数，相比之下，`ResNet18` 具有更少的层和参数。在训练和推理过程中，`VGG19` 需要更多的计算资源和时间。对于 `CIFAR` 这样的小型数据集，`ResNet18` 由于其较少的参数量，可以更快地训练和推理

一般会认为 `ResNet18` 的表现通常比 `VGG19` 更优，但是这里并没有与预期相符合，推断可能是由于数据切分以及数据量过小的原因等等

并且我还尝试了 `VGG16`，对比可以发现 `ResNet18` 在精度、速度上都碾压 `VGG16`

`VGG16` 远低于 `VGG19` (`VGG19` 是 `VGG16` 的扩展版本，在 `VGG16` 的基础上增加了三个额外的卷积层，从而形成一个具有19个卷积层和全连接层的模型。 `VGG19` 的结构与 `VGG16` 非常相似，只是在最后一个卷积块中添加了两个额外的卷积层), 这说明**加深卷积网络的深度，确实可以起到提高准确率的作用**

CIFAR-10 和 MNIST 数据集效果对比

首先需要注意，由于 `MNIST` 只有"黑白", 他的颜色通道只有一个，而不是像 `CIRAR-10` 是RGB三色有三个颜色通道，需要在 `server.py` 和 `client.py` 启用

```python
# expand the channel of MNIST data from 1 to 3
data = data.expand(-1,3,-1,-1)
```

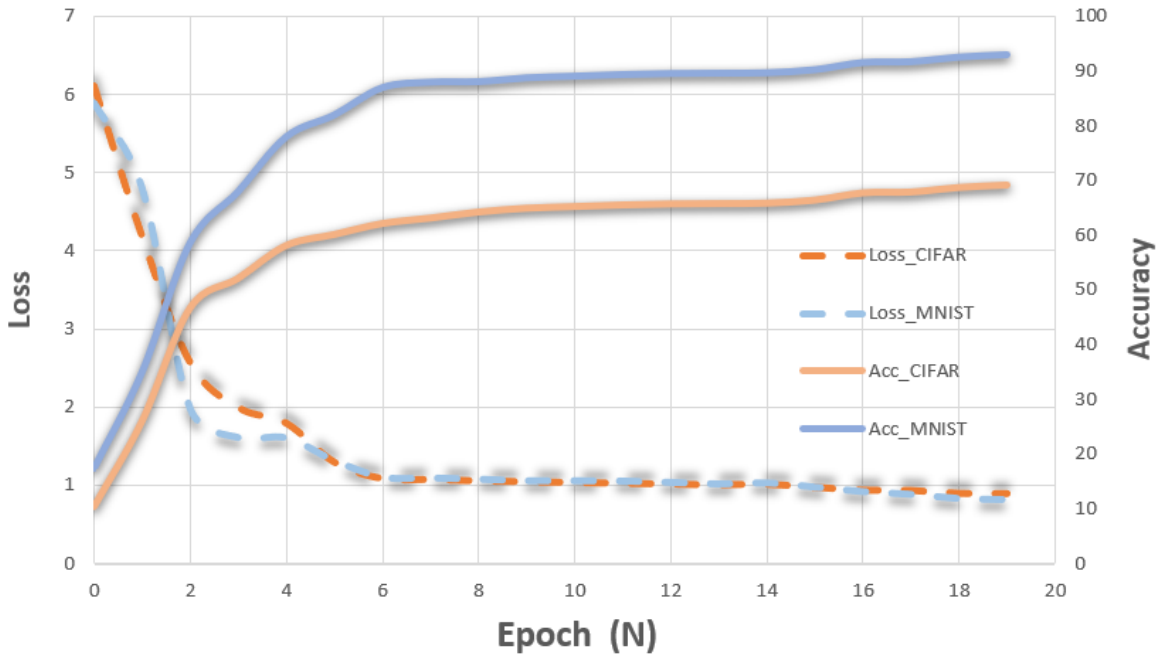
并且，还会存在`batch_size`不匹配的问题，需要在`client.py`中做如下修改：

```
# 注释掉两处下列这个句子
# loss = torch.nn.functional.cross_entropy(output, target)
# 添加下列句子
actual_batch_size = data.size()[0]  # Get the actual batch size
loss = torch.nn.functional.cross_entropy(output[:actual_batch_size],
target[:actual_batch_size])
```

记录下训练的输出后，如：

```
Epoch 0, acc: 3.330000, loss: 6.011744
Epoch 1, acc: 25.950000, loss: 4.109299
Epoch 2, acc: 46.600000, loss: 2.575171
...
Epoch 18, acc: 68.590000, loss: 0.900947
Epoch 19, acc: 69.030000, loss: 0.898190
```

然后绘制出训练过程：



可以发现`ResNe`t在`CIFAR-10`上的表现远远比`MNIST`要好，这也符合我们的直观判断

- 首先，与`CIFAR-10`相比，`MNIST`在图像复杂性和多样性方面是一个更简单的数据集。`MNIST`由手写数字的灰度图像组成，而CIFAR-10包含来自十个不同类别的各种物体的彩色图像。`MNIST`数据集的简单性和统一性使模型更容易学习和归纳模式
- 其次，数据集的规模也起到了一定的作用。`MNIST`有6万张训练图像和1万张测试图像，而`CIFAR-10`有5万张训练图像和1万张测试图像。有了更大的训练集，模型有更多的例子可以学习，使它们能够捕捉到更复杂的关系并提高它们的性能
- 此外，`MNIST`和`CIFAR-10`之间的图像分辨率差异会影响模型的性能。`MNIST`图像是低分辨率的（28x28像素），而CIFAR-10图像是高分辨率的（32x32像素）。`CIFAR-10`中增加的图像分辨率引入了更多的细节和变化，使得模型在提取有意义的特征方面更具挑战性

思考 & 建议

建议 1

把`CIFAR-10 dataset`文件放在本地，固然有利于方便使用，而不可能出现`http error`、`Connection timed out`、`Access failed`等一些列远程学习可能会出现的问题, 但是这导致文件特别大的问题

我推荐可以改为服务端`server.py`调用 🤗 Hugging Face而不是直接调用本地文件

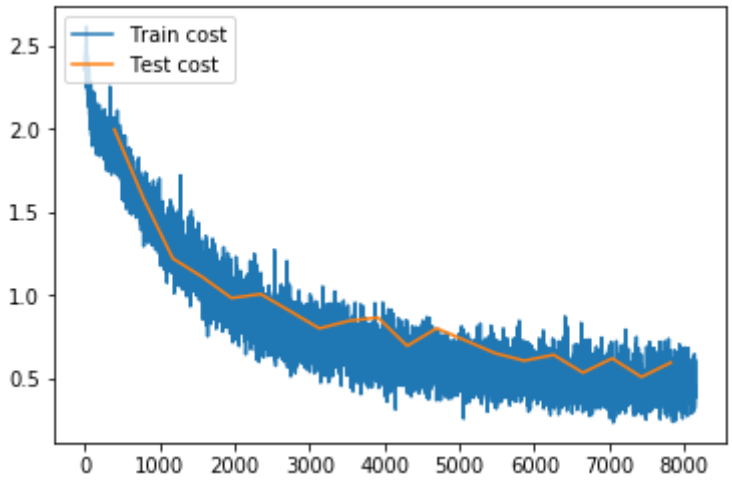
或者使用`paddle.dataset.cifar`来直接调用

再或者，和模型的调用一样用`torchvison`

```
from torchvision.datasets import CIFAR10
from torchvision.transforms import Compose, ToTensor, Normalize
RAW_DATA_PATH = './rawdata'
transform = Compose([
    ToTensor(),
    Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])
dataset = CIFAR10(
        root=RAW_DATA_PATH,
        download=True,
        train=True, #True加载训练集，False加载测试集
        transform=transform
    )
```

建议 2

可以增加一些可视化的图表来记录训练过程，比如直接使用`PaddlePaddle`（不过这样做的前提是数据和模型也使用`PaddlePaddle`）提供的可视化日志输出接口`paddle.v2.plot`，以折线图的方式显示`Train cost`和`Test cost`，效果如下（而不是和我一样用`matlibplot`来画）：



建议3

可以介绍一些免费的计算资源，比如google的colab，kaggle等可以免费使用的GPU算力资源