专业综合设计与训练总结报告

班　级：_____

学　号：_____

姓　名：_____

教　师：_____

同组成员：_____

2020.08.01

目 录

一、CNN 网络学习总结

（1）卷积层

1、卷积层与全连接层的比较

在通常的神经网络中，全连接层（相邻层的神经元全部连接在一起）忽略了数据的形状（维度）。比如输入数据是图像时，图像通常是高、长、通道方向上的3维形状。图像的3维形状中应该含有重要的空间信息。比如，空间上邻近的像素为相似的值、RBG的各个通道之间分别有密切的关联性、相距较远的像素之间没有什么关联等，3 维形状中可能隐藏有值得提取的本质模式。然而，因为全连接层会忽视形状，将全部的输入数据作为相同的神经元（同一维度的神经元）处理，所以无法利用与形状相关的信息。

而卷积层可以保持形状不变。当输入数据是图像时，卷积层会以3维数据的形式接收输入数据，并同样以3维数据的形式输出至下一层。因此，在CNN中，可以（有可能）正确理解图像等具有形状的数据。因此全连接层和卷积层的根本区别在于，全连接层从输入特征空间学到的是全局模式（涉及所有像素），而卷积层学到的是局部模式（小窗口中的像素，见图1）。

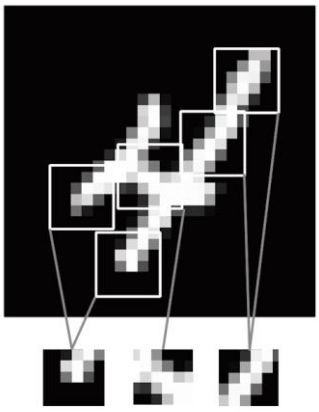


图 1 图像的局部模式，如边缘、纹理等

这个重要特性使卷积神经网络具有以下两个重要的性质。

- 卷积神经网络学到的模式具有**平移不变性**（translation invariant）。卷积神经网络在图像右下角学到某个模式之后，它可以在任何地方识别这个模式，比如左上角。对于密集连接网络来说，如果模式出现在新的位置，它只能重新学习这个模式。这使得卷积神经网络在处理图像时可以高效利用数据（因为视觉世界从根本上具有平移不变性），它只需要更少的训练样本就可以学到具有泛化能力的数据表示。
- 卷积神经网络可以学到**模式的空间层次结构**（spatial hierarchies of patterns）。第一个卷积层将学习较小的局部模式（比如边缘），第二个卷积层将学习由第一层特征组成的更大的模式，以此类推。这使得卷积神经网络可以有效地学习越来越复杂、越来越抽象的视觉概念（因为视觉世界从根本上具有空间层次结构）。

2、卷积运算

卷积层对输入矩阵进行填充并按照一定步幅滑动“窗口”（也可以叫卷积核、滤波器）从而计算（乘积累加运算）出输出矩阵。

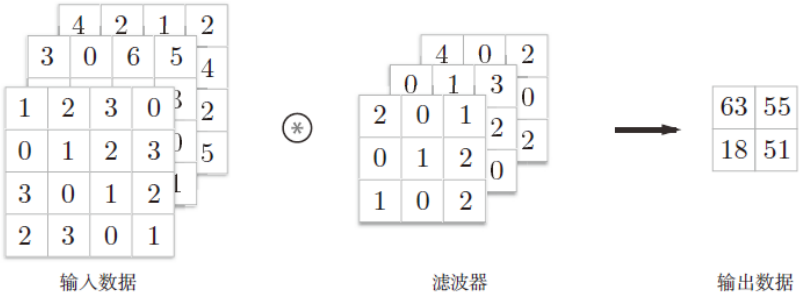


图 2 对 3 维数据进行卷积运算

假设输入大小为(H,W)，滤波器大小为(FH,FW)输出大小为(OH,OW)，填充为P，步幅为S。此时，可以计算出输出大小为

$$OH = \frac{H + 2P - FH}{S} + 1$$
$$OW = \frac{W + 2P - FW}{S} + 1$$

在实际的卷积层处理流中，输出数据往往并不是一维的，同时也可以选择加入偏置。通过应用 FN 个滤波器，输出特征图也生成了 FN 个。如果将这 FN 个特征图汇集在一起，就得到了形状为(FN, OH, OW) 的方块。

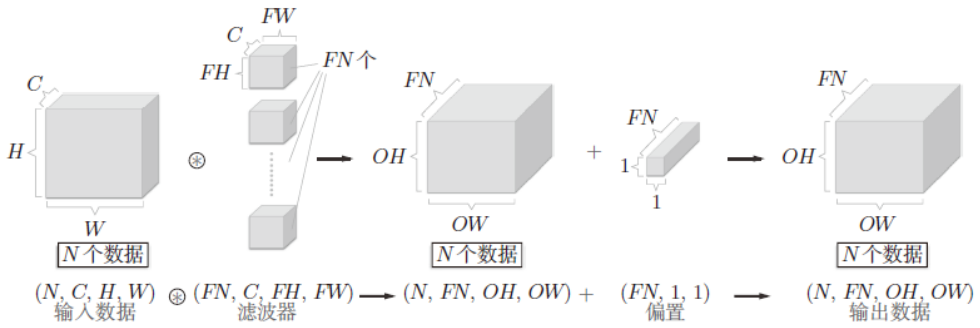


图 3 卷积运算处理流

（2）池化层

卷积层中用到的填充(Padding)操作是为了扩大输入数据的长和高，使得卷积运算可以在保持空间大小不变的情况下将数据传给下一层。而池化(Pool)则是缩小高、长方向上的空间的运算。如下图所示，将2×2的区域集约成 1 个元素的处理，缩小空间大小。

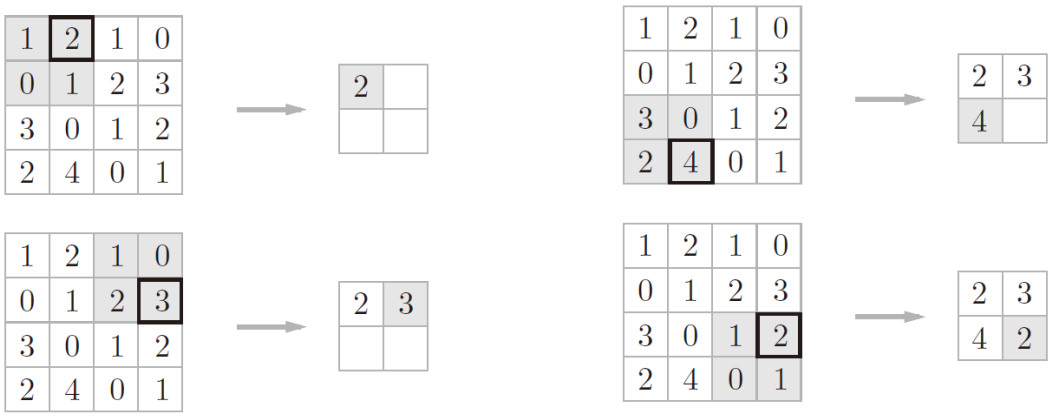


图 4 Max 池化处理示意图

一般来说，池化的窗口大小会和步幅设定成相同的值。除了 Max 池化之外，还有 Average 池化等。相对于 Max 池化是从目标区域中取出最大值，Average 池化则是计算目标区域的平均值。在图像识别领域，主要使用 Max 池化。

池化层具有以下特征。

- **没有要学习的参数**

  池化层和卷积层不同，没有要学习的参数。池化只是从目标区域中取最大值（或者平均值），所以不存在要学习的参数。

- **通道数不发生变化**

  经过池化运算，输入数据和输出数据的通道数不会发生变化。

- **对微小的位置变化具有鲁棒性**

  池化是根据某一区域内的数据来确定输出，所以当输入数据发生微小偏差时，池化（可能）仍会返回相同的结果。因此，池化对输入数据的微小偏差具有鲁棒性。

关于卷积层和池化层的实现可以使用 im2col 函数和 Numpy 完成，在现有深度学习框架中已经在内部实现，使用时只需通过函数调用即可。

（3）数据处理，基于 Pytorch

在学习任何深度学习时，准备合适正确的数据都至关重要，我们往往需要对数据进行一定的增强操作，分割数据集等。PyTorch 提供了许多工具，使数据处理和加载变得简单。

1、数据加载

```
# 加载公共数据集
# 以 MNIST 为例，Pytorch 还有 CIFAR-10/100，Imagenet，COCO 等
torchvision.datasets.MNIST(root, train = True, transform = None, target
_transform = None, download = False)

# 一个通用的数据加载器，用于加载自己的数据集，数据集中数据每个类别为一个文件夹
torchvision.datasets.ImageFolder(root="root folder path", [transform, t
arget_transform])
```

参数：

- root(string) - 数据集的根目录（如果下载数据集，也就是下载目录）。
- train(bool，optional) - 如果为 `True`，则返回训练数据集，否则返回测试数据集。
- download(bool，optional) - 如果为 `True`，则从官方下载数据集并将其放在根目录中。如果已下载数据集，不会再次下载。
- transform(callable ，optional) - 接收图像并返回转换后的数据。可以快速便捷地完成图像的预处理。
- target transform(callable ，optional) - 接收目标（标签）并对其进行转换的函数。

```
# 数据加载器。组合数据集和采样器，并在数据集上提供单进程或多进程迭代器。
torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampl
er=None, num_workers=0, collate_fn=<function default_collate>, pin_memo
ry=False, drop_last=False)
```

参数：

- dataset (*Dataset*) – 加载数据的数据集。
- batch_size (*int*, optional) – 每个 batch 加载多少个样本(默认: 1)。
- **shuffle** (*bool*, optional) – 设置为 `True` 时会在每个 epoch 重新打乱数据(默认: False).
- **sampler** (*Sampler*, optional) – 定义从数据集中提取样本的策略。如果指定，则忽略 `shuffle` 参数。

- **num_workers** (*int*, optional) – 用多少个子进程加载数据。0 表示数据将在主进程中加载(默认: 0)
- **collate_fn** (*callable*, optional) –
- **pin_memory** (*bool*, optional) –
- **drop_last** (*bool*, optional) – 如果数据集大小不能被 batch size 整除，则设置为 True 后可删除最后一个不完整的 batch。如果设为 False 并且数据集的大小不能被 batch size 整除，则最后一个 batch 将更小。(默认: False)

2、图像变换（数据预处理）torchvision.transforms

```
# Compose 将多个 transform 组合起来使用，注意前后顺序
torchvision.transforms.Compose([
     transforms.CenterCrop(10),
     transforms.ToTensor(),
 ])
```

可以将 Transform 划分为以下四大类

- **裁剪** -- Crop

中心裁剪：transforms.CenterCrop

随机裁剪：transforms.RandomCrop

随机长宽比裁剪：transforms.RandomResizedCrop

上下左右中心裁剪：transforms.FiveCrop

上下左右中心裁剪后翻转，transforms.TenCrop

- **翻转和旋转** -- FlipandRotation

依概率 p 水平翻转：transforms.RandomHorizontalFlip(p=0.5)

依概率 p 垂直翻转：transforms.RandomVerticalFlip(p=0.5)

随机旋转：transforms.RandomRotation

- **图像变换** -- transforms.Resize

标准化：transforms.Normalize

转为 tensor，并归一化至[0-1]：transforms.ToTensor

填充：transforms.Pad

修改亮度、对比度和饱和度：transforms.ColorJitter

转灰度图：transforms.Grayscale

线性变换：transforms.LinearTransformation()

仿射变换：transforms.RandomAffine

依概率 p 转为灰度图：transforms.RandomGrayscale

将数据转换为 PILImage：transforms.ToPILImage

通用变换，使用 lambd 作为转换器：transforms.Lambda

- **针对 transforms 的操作，使数据增强更灵活**

从给定的一系列 transforms 中选一个进行操作：transforms.Random Choice(transforms)

给一个 transform 加上概率，依概率进行操作：transforms. RandomApply (transforms, p=0.5)

将 transforms 中的操作随机打乱：transforms.RandomOrder

3、数据集的操作（Dateset）:

- **classtorch.utils.data.Dataset**:一个抽象类，所有其他类的数据集类都应该是它的子类。而且其子类必须重载两个重要的函数：len(提供数据集的大小）、getitem(支持整数索引)。
- **classtorch.utils.data.TensorDataset**:封装成 tensor 的数据集，每一个样本都通过索引张量来获得。

- **classtorch.utils.data.ConcatDataset**:连接不同的数据集以构成更大的新数据集。
- **classtorch.utils.data.Subset(dataset,indices)**:获取指定一个索引序列对应的子数据集。
- **torch.utils.data.random_split(dataset,lengths)**:按照给定的长度将数据集划分成没有重叠的新数据集组合。
- **classtorch.utils.data.Sampler(data_source)**:所有采样的器的基类。每个采样器子类都需要提供 iter 方-法以方便迭代器进行索引和一个 len 方法以方便返回迭代器的长度。
- **classtorch.utils.data.SequentialSampler(data_source)**:顺序采样样本，始终按照同一个顺序。
- **classtorch.utils.data.RandomSampler(data_source)**:无放回地随机采样样本元素。
- **classtorch.utils.data.SubsetRandomSampler(indices)**：无放回地按照给定的索引列表采样样本元素。
- **classtorch.utils.data.WeightedRandomSampler(weights,num_samples,replacement=True)**:按照给定的概率来采样样本。
- **classtorch.utils.data.BatchSampler(sampler,batch_size,drop_last)**:在一个 batch 中封装一个其他的采样器。
- **classtorch.utils.data.distributed.DistributedSampler(dataset,num_replicas=None,rank=None)**:采样器可以约束数据加载进数据集的子集。

示例：利用 SubsetRandomSampler / random_split 实现数据的划分

```
# SubsetRandomSampler 无放回地按照给定的索引列表采样样本元素
# 生成索引集合 train_indices 和 val_indices，
train_sampler=SubsetRandomSampler(train_indices)
valid_sampler=SubsetRandomSampler(val_indices)
```

```
train_loader=torch.utils.data.DataLoader(dataset, batch_size=batch_size
, sampler=train_sampler)
validation_loader=torch.utils.data.DataLoader(dataset, batch_size=batch
_size, sampler=valid_sampler)
```

```
# random_split 按照给定的长度将数据集划分成没有重叠的新数据集组合
# 计算划分数据集长度 train_size 和 test_size
train_size=int(0.8*len(full_dataset))
test_size=len(full_dataset)-train_size
train_dataset,test_dataset=torch.utils.data.random_split(full_dataset,[
train_size,test_size])
```

（4）基于 Pytorch 的迁移学习

关于数据和数据集的处理加载，前面都有涉及，此处主要是关于预训练模型的调整和使用。

参考链接:https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

在实践中，因为缺乏足够大的数据集，几乎没有人从 0 开始训练完整的卷积神经网络。通常会使用已经在大型数据集训练后的网络（如 ImageNet 等）,使用预训练的卷积网络可以作为初始网络，或者作为特征提取器。

这两种迁移学习的场景如下：

微调网络：使用预训练的网络进行初始化，而不是随机初始化，如采用在 imagenet 数据集训练过的网络，剩下的训练和通常的训练一样。（对各层参数进行微调和更新，继续在新任务中训练）。

卷积网络特征提取器：除全连接层外，冻结所有权重参数。将最终层替换为一个具有随机权重的全连接层，仅仅对这一层进行训练。（仅裁剪重新训练全连接层，全面的所有层都作为特征提取器）。

Pytorch 在 torchvision.models 模块中提供了 AlexNet、VGG、ResNet、

SqueezeNet 和 DenseNet 模型结构。

```
# pretrained=True 使用预训练的模型（包含参数）
# pretrained=False 仅返回模型网络结构（不含参数）
resnet18 = torchvision.models.resnet18(pretrained=True)
```

1、微调预训练模型，重置了全连接层（其权重参数随机初始化），微调整个网络

```
model_ft = models.resnet18(pretrained=True)  # 加载预训练网络
num_ftrs = model_ft.fc.in_features  # 获取全连接层输入特征
model_ft.fc = nn.Linear(num_ftrs, 2) # 重置全连接层
model_ft = model_ft.to(device)   # 设置运行设备 CPU/GPU
criterion = nn.CrossEntropyLoss()  # 选取损失函数
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)
 # 梯度优化算法
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma
=0.1)  # 学习率更新
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_schedu
ler, num_epochs=25)  #训练及更新
```

2、作为特征提取器，冻结卷积层原有参数，对初始化的全连接层权重参数进行训练

```
model_conv = torchvision.models.resnet18(pretrained=True) #加载预训练模型
for param in model_conv.parameters():
    param.requires_grad = False     #冻结参数
num_ftrs = model_conv.fc.in_features  #获取全连接层输入特征数
model_conv.fc = nn.Linear(num_ftrs, 2)  #重置全连接层
model_conv = model_conv.to(device)  # 设置运行设备 CPU/GPU
criterion = nn.CrossEntropyLoss()  # 选取损失函数
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, moment
um=0.9)  # 梯度优化算法
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gam
ma=0.1)   # 学习率更新
model_conv = train_model(model_conv, criterion, optimizer_conv, exp_lr_
scheduler, num_epochs=25)  #训练模型
```

（5）常用神经网络可视化工具

参考 Github 项目：https://github.com/ashishpatel26/Tools-to-Design-or-

Visualize-Architecture-of-Neural-Network

推荐以下工具：

1、Netron（网页版简洁好用，只需导入模型文件即可）

Github 地址：https://github.com/lutzroeder/Netron

网页地址：https://lutzroeder.github.io/netron/

在使用中发现其对 Pytorch 的.pt 和.pth 格式支持并不是很好，推荐使用以下代码转换为.onnx 格式。

```
# 将 Pytorch 搭建的网络模型保存为 ONNX 格式
import torch.onnx
import torchvision

model = torchvision.models.alexnet(pretrained=True).to(device)
input = torch.randn(10, 3, 224, 224).to(device)  # 输入大小应根据实际模型调整
torch.onnx.export(model, input, "alexnet.onnx", verbose=True)  # model 可以替换为自己的模型
```

2、NN-SVG（网页版，需手动输入每层参数，支持三种绘图模式）

Github 地址：https://github.com/alexlenail/NN-SVG

网页地址：http://alexlenail.me/NN-SVG/

3、ConvNetDraw（网页版，根据伪代码直接生成，绘图可调参数较少）

Github 地址：https://github.com/cbovar/ConvNetDraw

网页地址：https://cbovar.github.io/ConvNetDraw/

4、Tensorwatch

```
# tensorwatch 是一个可视化工具的集合，还能实现迭代中的可视化等
import tensorwatch as tw
import torchvision.models
```

```
alexnet_model = torchvision.models.alexnet()
img = tw.draw_model(alexnet_model, [1, 3, 224, 224])# 输入大小应根据实际模型调整
img.save(r'alexnet.jpg')
```

二、采用网络结构介绍

（1）DenseNet (Huang G，Liu Z，Laurens V D M，et al. Densely Connected Convolutional Networks[J]. 2016.)

研究表明，如果卷积网络在接近输入和接近输出地层之间包含较短地连接，那么，该网络可以显著地加深，变得更精确并且能够更有效地训练。DenseNet 就是基于这个观察提出的，其通过前馈的方式将每个层与其它层连接成密集卷积网络。如作者在论文中所述，DenseNet 缓解了消失梯度问题，加强了特征传播，鼓励特征重用，并大大减少了参数的数量。

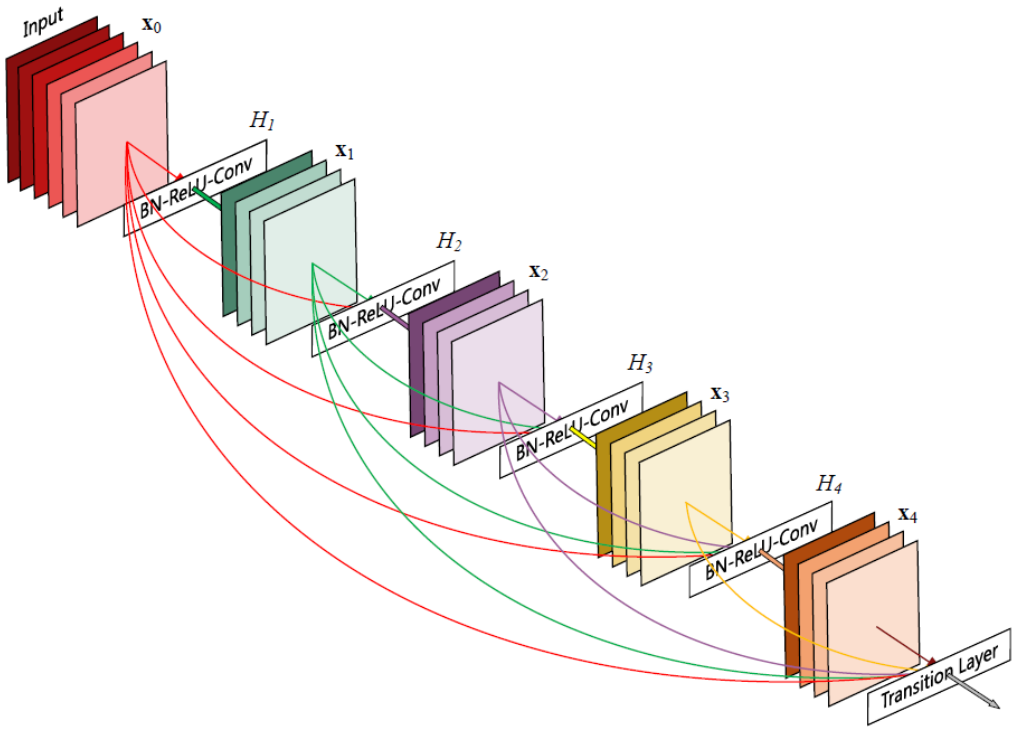


图 5 一个 5 层密集块，每层都将前面所有层的输出作为输入（来自论文）

该架构与 ResNet 相比，在将特性传递到层之前，没有通过求和来组合特性，而是通过连接它们的方式来组合特性。因此第 x 层（输入层不算在内）将有 x 个输入，这些输入是之前所有层提取出的特征信息。

因为不需要重新学习冗余特征图，这种密集连接模式相对于传统的卷积网络只需要更少的参数。传统的前馈体系结构可以看作是具有一种状态的算法，这种状态从一个层传递到另一个层。每个层从其前一层读取状态并将其写入后续层。它改变状态，但也传递需要保留的信息。研究提出的密集网络体系结构明确区分了添加到网络的信息和保留的信息。密集网层非常窄(例如，每层 12 个过滤器)，仅向网络的“集体知识”添加一小组特征映射，并且保持其余特征映射不变，并且最终分类器基于网络中的所有特征映射做出决策。

除了参数更少，另一个 DenseNet 的优点是改进了整个网络的信息流和梯度，这使得它们易于训练。每个层直接访问来自损失函数和原始输入信号的梯度，带来了隐式深度监控。这使得训练深层网络变得更简单。此外，研究人员观察到密集连接具有规则化效果，这减少了对训练集较小的任务的过拟合。

论文中仅有的两个公式非常精辟地表现了 ResNet 和 DenseNet 的本质区别。

$$\mathbf{x}_\ell = H_\ell(\mathbf{x}_{\ell-1}) + \mathbf{x}_{\ell-1}.$$

第一个公式描述 ResNet。这里的 l 表示层，x_l 表示 l 层的输出，H_l 表示一个非线性变换。所以对于 ResNet 而言，第 l 层的输出是 $l-1$ 层的输出加上对 $l-1$ 层输出的非线性变换。

$$\mathbf{x}_\ell = H_\ell([\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{\ell-1}]),$$

第二个公式描述 DenseNet。$[x_0, x_1, \ldots, x_{l-1}]$表示将 0 到 $l-1$ 层的输出特

征图串联，即通道的合并。而前面的 ResNet 是做值的相加，通道数不变。

实验结果及对比:

这里给出 Densenet 和 ResNet 在 CIFAR-100 和 ImageNet 上的对比结果。从图 6 中可以看出，只有 0.8M 的 DenseNet-100 性能已经超越 ResNet-1001，并且后者参数大小为 10.2M。而从图 7 中可以看出，同等参数大小时，DenseNet 也优于 ResNet 网络。

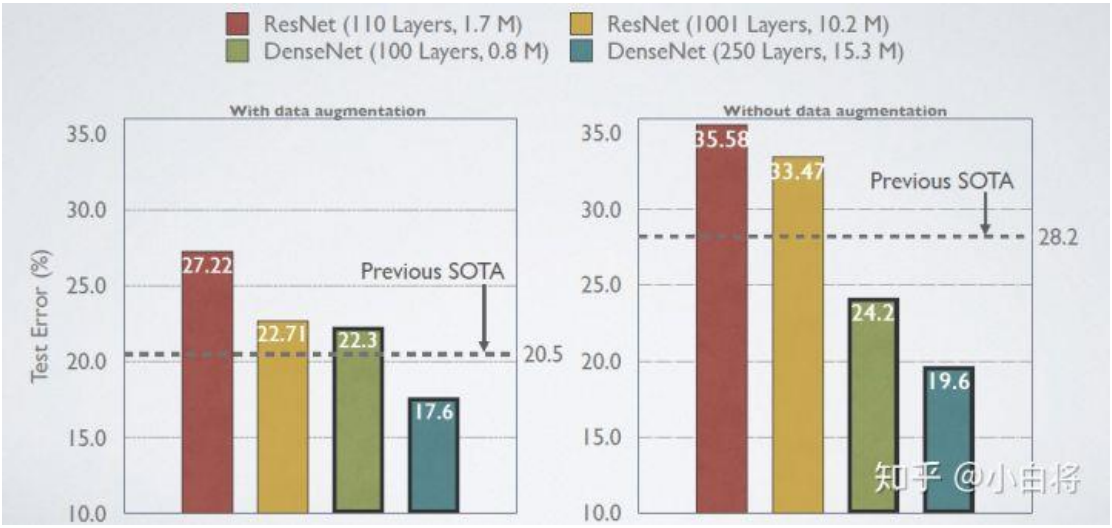


图 6 在 CIFAR-100 数据集上 ResNet vs DenseNet（图源水印）

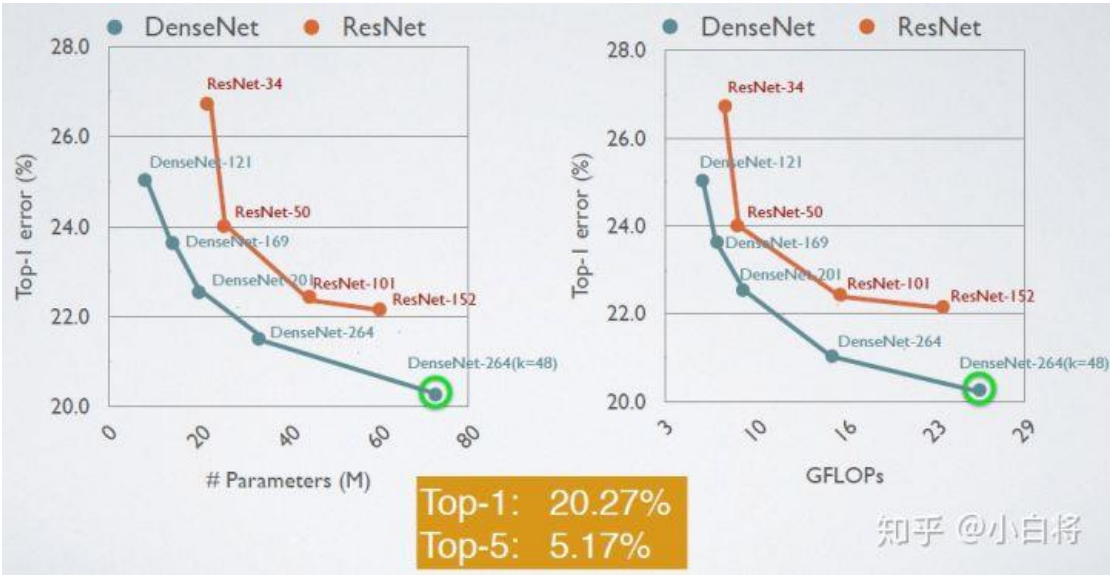
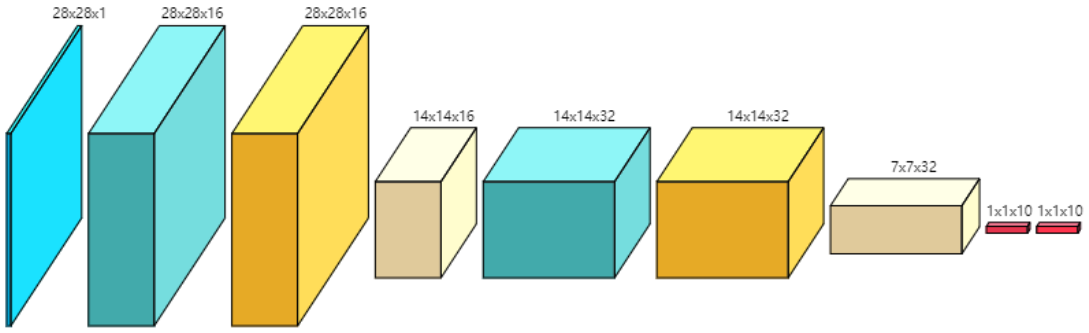


图 7 在 ImageNet 数据集上 ResNet vs DenseNet（图源水印）

（2）Net1（MNIST 训练）

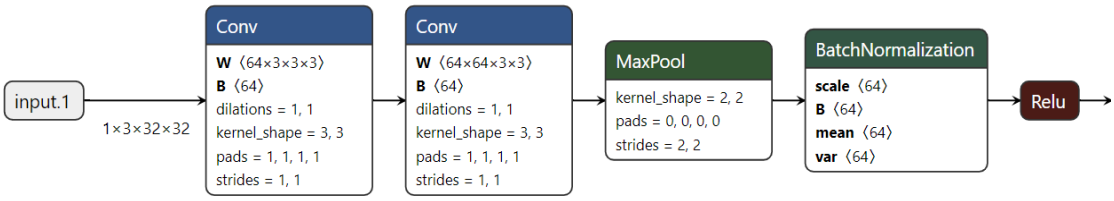
该网络使用两层卷积层（包括激活函数和最大池化）和一层全连接层作为输出。

```
# 伪代码
input(28, 28, 1)
conv(28, 28, 16)
relu(28, 28, 16)
pool(14, 14, 16)
conv(14, 14, 32)
relu(14, 14, 32)
pool(7, 7, 32)
fullyconn(1, 1, 10)
softmax(1, 1, 10)
```



（3）Net2（CIFAR-10 训练）

使用 Net1 在 CIFAR-10 上效果并不好，所以 Net2 增加了卷积层层数，并在全连接层中加入了 Dropout 来抑制过拟合。



完整模型结构（点击 https://temp.zwysun.tk/）

三、任务实现及结果

（1）任务描述及分析

光纤弯曲程度和其截面图像之间存在一定的关系，现已拍摄一组光纤 0-90 度弯曲所对应的图像，利用卷积神经网络探究其中的关系，并尝试完成分类。

1、学习卷积神经网络相关知识，了解卷积层、池化层的概念和处理过程；

2、使用预训练模型测试数据，学习卷积神经网络训练中超参数的调整；

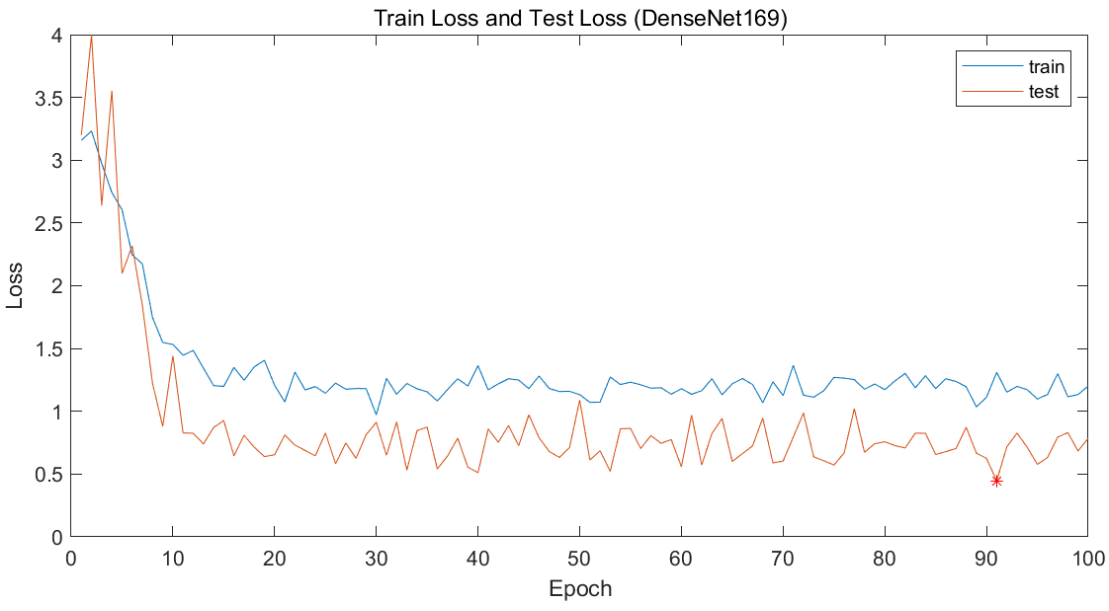
3、学习神经网络可视化方法。

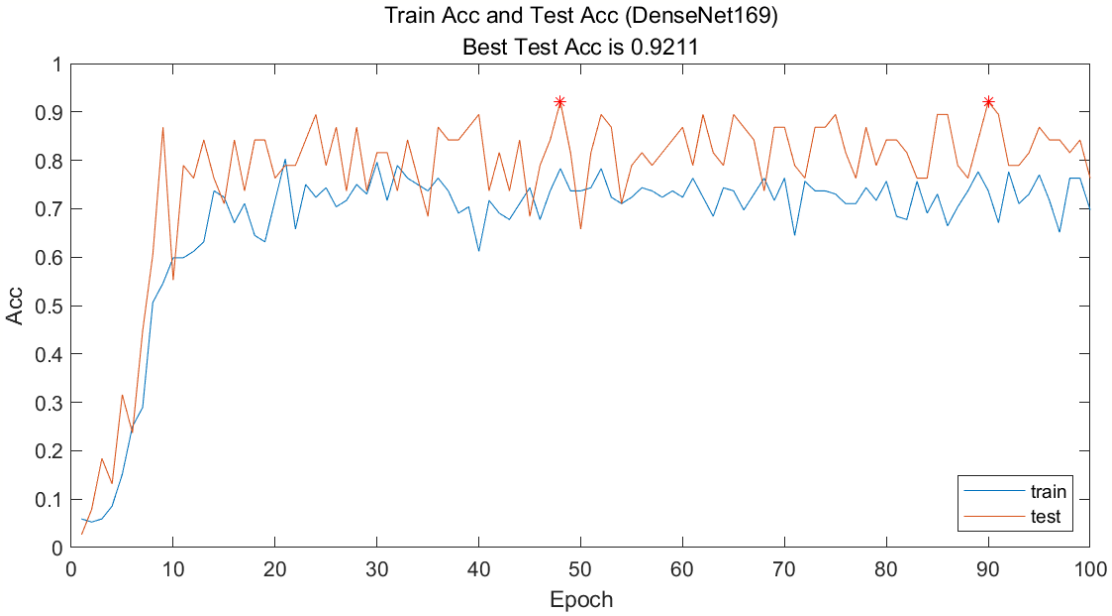
（2）利用 DenseNet 分类光纤图片

实验结果：

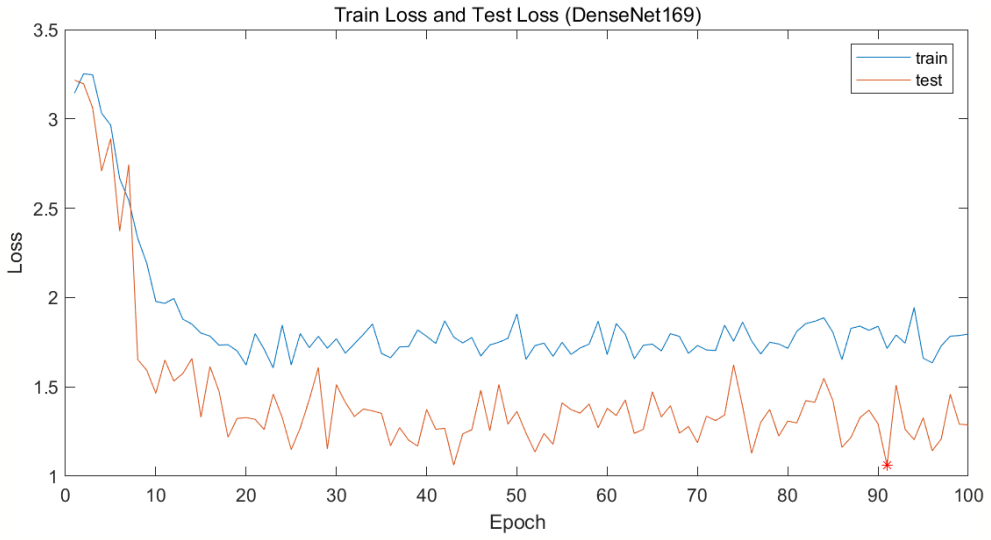
1. 利用**晚上**数据训练，最佳 acc=0.8947，平均 acc=0.236
2. 利用**上午**数据训练，最佳 acc=1，　　平均 acc=0.957
3. 利用晚上的最佳模型 测试 上午数据， 14 个图正确分类，acc=0.073
4. 利用上午的最佳模型 测试 晚上数据， 20 个图正确分类，acc=0.104
5. 可以初步推测上午和晚上的数据在该模型中存在较大的差别

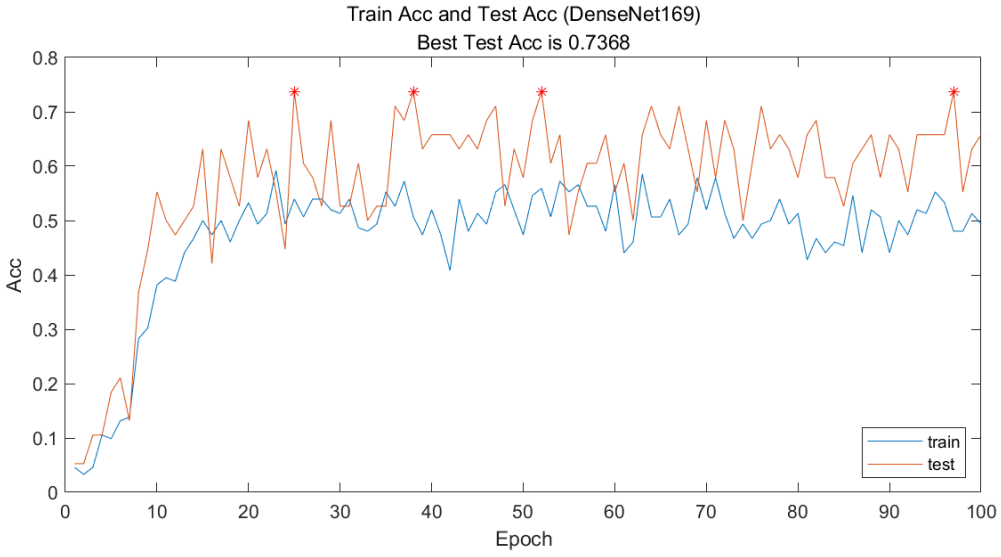
上午数据：




Train Acc and Test Acc (DenseNet169)
Best Test Acc is 0.9211
Acc
Epoch
train
test

晚上数据：


Train Loss and Test Loss (DenseNet169)
Loss
Epoch
train
test


Train Acc and Test Acc (DenseNet169)
Best Test Acc is 0.7368
Acc
Epoch
train
test

关于 Validation Accuracy 大于 Train Accuracy

参考文档：https://www.zhihu.com/question/270731692?sort=created 和 https://www.cnblogs.com/carlber/p/10892042.html

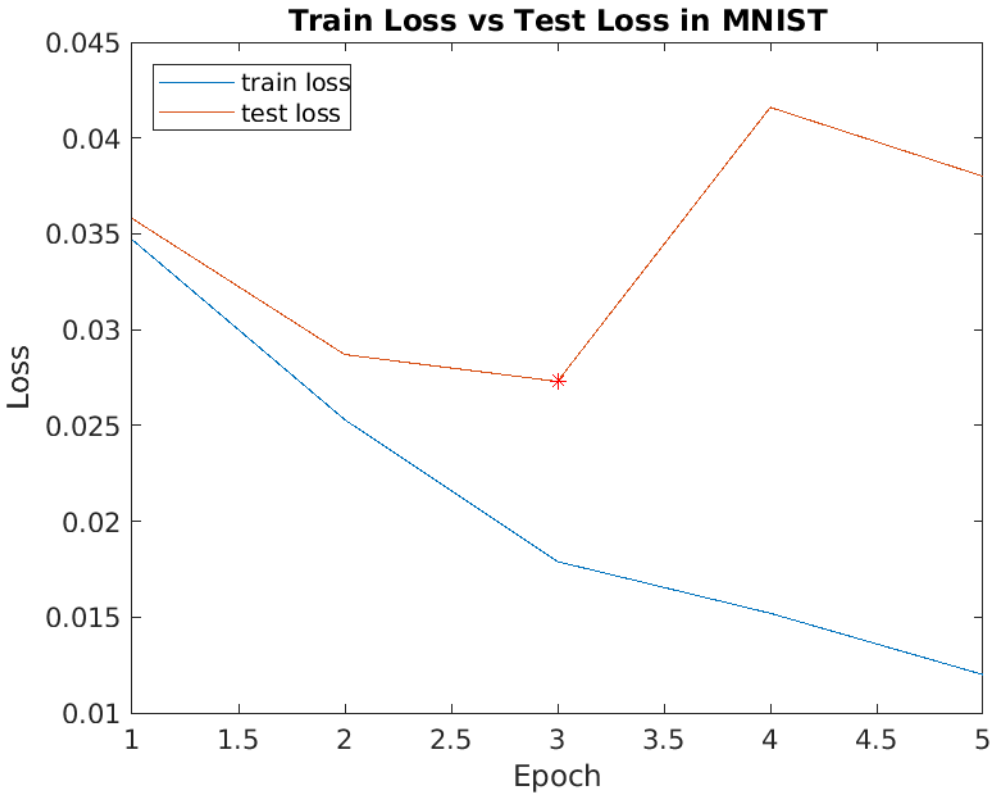
总结：

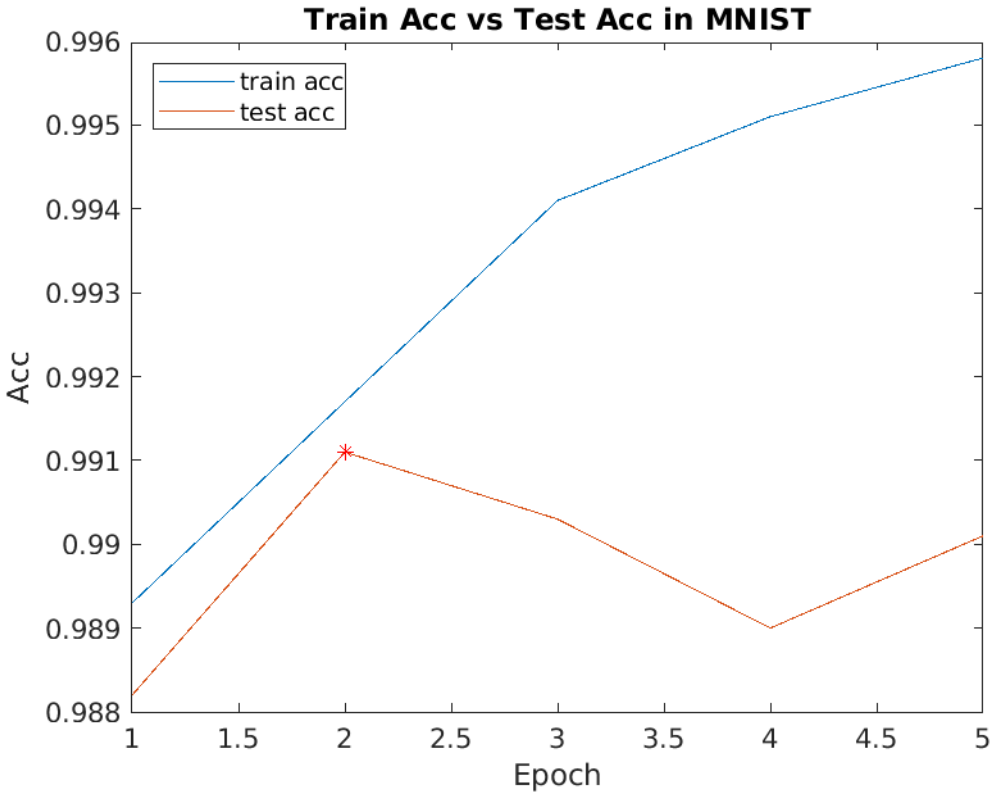
1、Dropout 层的影响，在训练中会随机屏蔽部分神经元而在测试中却是用上所有的神经元，所以可能出现测试时效果更好的情况。

2、数据增强（data augmentation）的影响，训练时往往会对训练数据进行一定的增强，但测试数据则没有，使得测试数据分布更佳。

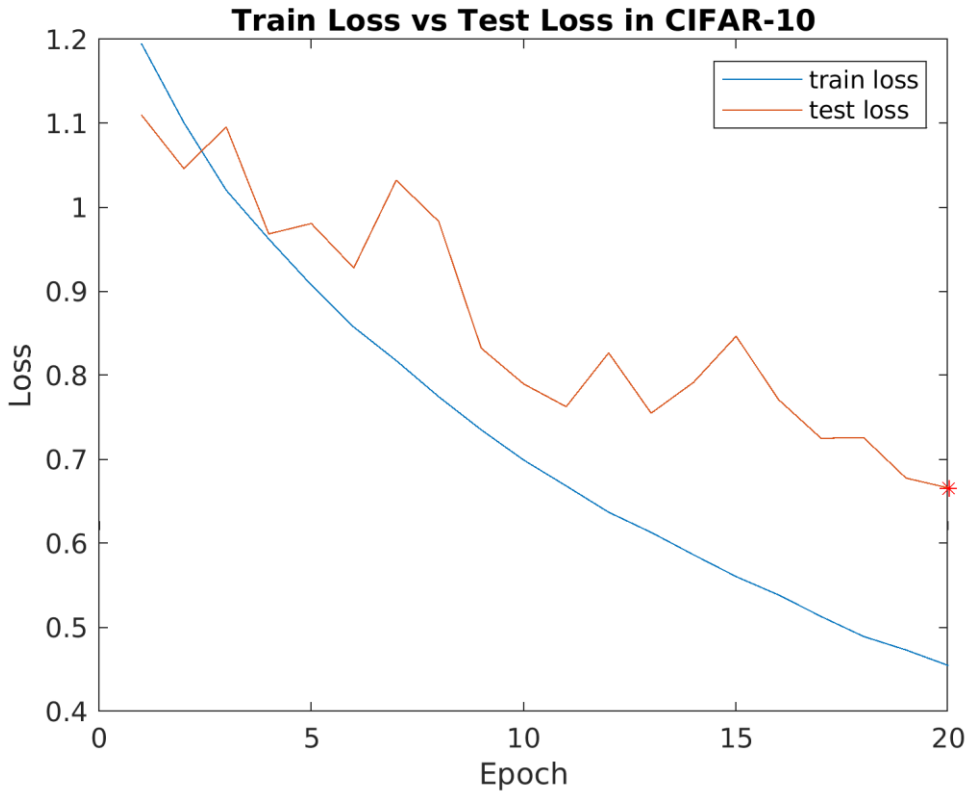
3、验证数据集较小时，在预训练模型上也容易出现该部分测试更佳的情况。

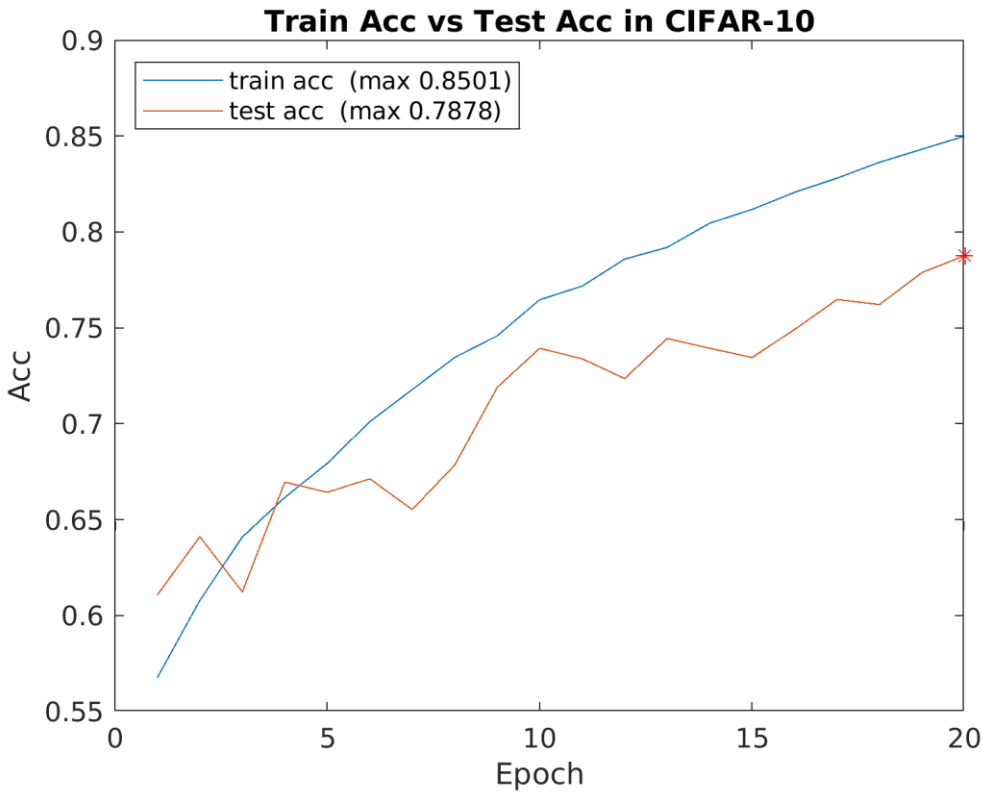
（3）Net1 在 MINST 上的效果





（4）Net2 在 CIFAR-10 上的效果





四、日志及总结

1、日志记录

小学期时间点：

- 6 月 29 日完成实习导师、题目与学生三方匹配；
- 7 月 13 日-25 日，开展实习内容；
- 8 月 31 日之前提交实习报告。

截止 7.19

- 学习 CNN 的基础知识，主要参考了 Deep Learning with Python(Author: Francois Chollet)，深度学习入门（作者：斋藤康毅）和 lamda 实验室魏秀参的解析卷积神经网络。同时也在网络上参考了一些相关的官方文档（尤其是 pytorch 的示例）和博客。
- 利用预训练模型实现迁移学习
- 问题：

1. 训练集准确率小于测试集
2. 没有实现随机划分

- 解决方案：

1. 交叉验证，看下是否还是训练精度低于测试精度；
2. 随机选择数据作为测试集；
3. 去掉 0-30 度的数据看下实验效果；
4. 查找一下是否有解决 dropout 局限性的方法；
5. 整理算法的框架图及伪代码

7.20

- 尝试实现训练集和测试集的随机划分，以及 K 折交叉验证
- 问题：

1. 一开始采用 pytoch 的包完成数据读取，预处理和分块；
2. 在利用 ImageFolder 生成 dataset 后 没有现有函数对其进行 transform（因为训练集和测试集采用
不同的方法 transform）
3. pytorch 没有现成 API 实现交叉验证

- 解决方案：

1. 考虑采用 sklearn 完成数据的处理，但还没找到合适的 transform 方法
2. 随机划分可以使用 pytorch 的
torch.utils.data.random_split 或者
sklearn.model_selection.train_test_split
3. 使用 KFold 进行 K 折划分
4. 重写训练函数，K 折交叉验证参考：
https://blog.csdn.net/foneone/article/details/104445320 ；
https://blog.csdn.net/Pl_Sun/article/details/106975414

7.21

- 添加随机划分和 K 折交叉验证（都已完成）

- 关于交叉验证的一些思考 https://www.jianshu.com/p/f14826061612
- 还没测试去掉 30 度以下数据的效果
- 问题：

  1. 标准化参数的选择，transforms.Normalize，目前灰度图采用的是 Mnist 的参数
  2. batch_size 的选择，一般可能偏大时效果变好（不一定），此数据集较小，目前认为 4 比较好
  3. 依然存在 val_acc > train_acc (根据目前训练过程，猜测和数据集划分方式有关，随机划分时
  验证集每一类都是平均的，K 折交叉验证时出现概率更小。数据集较小，容易出现巧合)
  4. 以下是关于准确率的（7.21 使用的都是 night_data，7.20 使用的是 morning_data）
  K 折（10 折）交叉验证很多个分组的效果明显较差（大多在 40%以下）
  未训练完，未测试 morning data
  对于随机划分数据集的准确率在 9:1 划分是 55%，8:2 划分是 76%
  利用 night_data 训练的模型去测试所有 morning data，准确率很低（百分之几）
  morning_data 和 night_data 存在较大偏差?

7.22

- ~~打牌（误）~~
- 跑交叉验证的模型，因为一些小错误导致最后的结果都没能保存下来
- 问题：

  1. 交叉验证综合起来看准确率下降明显（目前看大概 30%）， 5 折和 10 折划分跑出来结果相差较大
  2. 今天采用的 batch_size 是 16，感觉没有明显的差别
  3. 交叉验证中大多情况下，其中会有一两折的的准确率非常高，大概在 80%-90%。

7.23

- 添加了分层的交叉验证划分方式，目前来看效果改善很明显
- 参数如下，分层随机抽取，10 折，batch_size=4

    1. 晚上数据训练最佳 acc=0.8947，平均 acc=0.236
    2. 上午数据训练最佳 acc=1, 平均 acc=0.957
    3. 利用晚上的最佳模型测试上午数据，对了 14 个图，acc=0.073
    4. 利用上午的最佳模型测试晚上数据，对了 20 个图，acc=0.104

7.24 - 7.25

- 针对之前遇到的问题进行了一定分析
- 将所有代码重新梳理了一遍，检查是否存在不明显的逻辑错误，同时与官方文档（pytorch）进行比对
- 针对图像进行了一定增强处理，使得黑色背景（无信息部分）更少
- 测试了 Adagrad 的效果，因为失误清除了测试结果没有记录，效果与 SGD 差不多

    1. 改进不明显
    2. 可能选用的 DenseNet 不太适合?

7.28-7.29

- 在开源数据集中，选择了 MNIST 和 CIFAR10
- 在已有代码的基础上，重新编写了网络搭建和模型训练部分 其他进行微调
- MNIST 是灰度图，数据集较为简单。CIFAR-10 是 RGB 图，数据集更偏向人眼图片 更为复杂。
- DenseNet 测试以上两个数据集和论文中所述基本一致。使用自己的网络结构测试时，结果也在预期内。

完成小学期报告

- 主要分成四个部分

    1. CNN 基础知识的学习，以及在代码编写中遇到的一些问题和解决
    2. 所采用的网络结构（DenseNet 以及两个简单的 CNN 网络）及介绍

3. 模型的训练和结果
4. 学习日志和总结

2、总结

上学期在马老师的课程上学习过一些基础神经网络知识，但那时候只停留在简单的全连接层和梯度求导。但是课堂上学习的内容却是自己第一次真正的接触及了解到这一领域，受益良多。随着课程的推进，加之好朋友正是计院学生且刚好在研究这一方向，在日常的交流中收获了很多启发性的建议。

这一次小学期我们主要侧重于卷积神经网络的学习，在开始阶段我们各自阅读了几本相关的入门书籍作为知识基础，后续则是直接选用了预训练模型入手。选用预训练模型对初次入手更为友好，同时在实践中大多也是采用预训练模型使用。得益于 Pytorch 详实的文档和良好的学习生态，第一版迁移学习代码的完成基本没有遇到什么问题。反倒在随机数据集划分和 K 折交叉验证的实现上遇到较多问题，一是 Pytorch 并没有原生支持相关操作，二是对深度学习领域的不熟悉，一些相关工具包并不了解。之后在了解 sklearn 的用法后就成功实现了 K 折交叉验证。在应用 K 折交叉验证后，所得效果并不好，也根据相关资料去尝试优化了。

之后采用公共数据集后，考虑到 DenseNet 在公共数据集上已经有许多权威的测试，我们再做一遍意义并不大。所以选择了自己搭建的简单神经网络进行测试，并且重新编写了训练过程的代码。在两数据集上也都能达到预期效果。

五、附录：主要代码

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
from sklearn.model_selection import KFold, StratifiedShuffleSplit

import random
import shutil

plt.ion()   # 交互模式可以动态显示图像

"""### 硬件选择 GPU"""

# GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

"""## 数据加载及处理

### 下载数据集
可以从 Google Drive / Github / 其他数据地址下载数据
"""

!rm -rf CNN
!git clone https://github.com/Zwysun/CNN.git
!ls

!rm -rf CNN/dataset/morning/val
!rm -rf CNN/dataset/morning/train

"""### 数据预处理
利用 transforms 对图片进行预处理，可以分别针对训练集和验证集采取不同的处理方法

#### 比例随机划分 数据集和验证集
```

```
"""

data_dir = 'CNN/dataset/night'     # 数据目录
ratio_train = 0.7                  # 训练集比例

def data_random_split(current_dir,ratio_train):
    '''
    将当前文件夹中文件按一定比例分成 train 和 test 连个列表，列表存放文件名
    '''
    data_listdir = os.listdir(current_dir)
    random.shuffle(data_listdir)
    train_len = int(len(data_listdir)*ratio_train)
    train_listdir = data_listdir[:train_len]
    train_listdir = data_listdir[:train_len]
    test_listdir  = data_listdir[train_len:]
    return train_listdir,test_listdir

def data_generator(root,data_dir,ratio_train=0.8):
    '''
    root：输入你的 data 目录
    datadir：你要创建好的的文件夹，将会生成 train、和 val
    ratio_train:train_data 占总数据的比例
    '''
    listdir = os.listdir(root)
    train_data_dir = os.path.join(data_dir, "train")
    test_data_dir = os.path.join(data_dir, "val")
    os.makedirs(train_data_dir)
    os.makedirs(test_data_dir)
    for name in listdir:
        print(name)
        current_dir = os.path.join(root, name)
        print(current_dir)
        train_dir_a,test_dir_a = data_random_split(current_dir,ratio_tr
ain)
        train_listdir_c = os.path.join(train_data_dir,name)
        test_listdir_c  = os.path.join(test_data_dir,name)
        a=1 if os.path.exists(train_listdir_c) else os.makedirs(train_l
istdir_c)
        a=1 if os.path.exists(test_listdir_c) else os.makedirs(test_lis
tdir_c)
        for img in train_dir_a:
            train_listdir_b = os.path.join(current_dir, img)
            train_listdir_d = os.path.join(train_listdir_c, img)
            train_dir_b = shutil.copy(train_listdir_b,train_listdir_d)
```

```
        for img in test_dir_a:
            test_listdir_b  = os.path.join(current_dir, img)
            test_listdir_d  = os.path.join(test_listdir_c, img)
            test_dir_b = shutil.copy(test_listdir_b,test_listdir_d)

    print('ok')

# 随机划分为训练集和验证集
data_generator(data_dir, data_dir, ratio_train)

# !cd train_data
# !ls -lR |grep -v ^d|awk '{print $9}'

# 训练集 扩充及正则化
# 验证集 仅正则化
data_transforms = {
    'train': transforms.Compose([
        transforms.Grayscale(1),      # 转换为灰度图
        transforms.CenterCrop(1080),
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),    # 增广
        transforms.ToTensor(),
        # transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.
225])    # RGB 标准化参数(Imagenet Dateset)
        transforms.Normalize([0.1307], [0.3081])      # 灰度图标准化参数
(Mnist)
    ]),
    'val': transforms.Compose([
        transforms.Grayscale(1),
        transforms.CenterCrop(1080),
        transforms.Resize(224),
        transforms.ToTensor(),
        # transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.
225])
        transforms.Normalize([0.1307], [0.3081])      # 灰度图标准化参数
    ]),
}

image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_
size=4,
```

```
                                                shuffle=True, num_workers=
4)
              for x in ['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes
print (dataset_sizes)
print (image_datasets)
print (class_names)

"""#### K 折交叉验证(的数据处理)"""

data_dir = 'CNN/dataset/morning'    # 数据目录

data_transforms = transforms.Compose([
        transforms.Grayscale(1),      # 转换为灰度图
        transforms.CenterCrop(1080),
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),    # 增广
        transforms.ToTensor(),
        # transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.
225])    # RGB 标准化参数(Imagenet Dateset)
        transforms.Normalize([0.1307], [0.3081])      # 灰度图标准化参数
(Mnist)
    ])

image_datasets = datasets.ImageFolder(data_dir, data_transforms)

class_names = image_datasets.classes
dataset_sizes = len(image_datasets)
print ('Total:', dataset_sizes)
print (class_names)

# 不要运行，测试划分
X=np.arange(380).reshape(190,2)

for train_index,test_index in kf.split(X):
    print('train_index %s, test_index %s'%(train_index, test_index))

train_sampler = torch.utils.data.SubsetRandomSampler(train_index)
valid_sampler = torch.utils.data.SubsetRandomSampler(test_index)

"""### 显示部分图片"""

# 交叉验证这 用不了
```

```
def imshow(inp, title=None):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    # mean = np.array([0.485, 0.456, 0.406])
    # std = np.array([0.229, 0.224, 0.225])   # RGB
    mean = np.array([0.1307])
    std = np.array([0.3081])   # 灰度图

    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # pause a bit so that plots are updated

# Get a batch of training data
inputs, classes = next(iter(dataloaders))

# Make a grid from batch
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])

"""## 模型训练"""

def train_model(model, dataloaders, criterion, optimizer, scheduler, da
tasize, num_epochs=25):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
```

```
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / datasize[phase]
            epoch_acc = running_corrects.double() / datasize[phase]

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
            if phase == 'train':
              scheduler.step()

        print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
```

```
    print('Best val Acc: {:4f}'.format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model, best_acc

# K 折交叉验证

def k_fold_train(k, dataset, model, batch_size, criterion, optimizer, s
cheduler, num_epochs=25):
    # kf = KFold(k, shuffle=True)

    # 分层划分，避免出现某一层全在训练集/验证集的情况
    kf = StratifiedShuffleSplit(n_splits=k,train_size=1-
1/k,test_size=1/k,random_state=True)

    i = 1
    valid_acc_sum = 0
    Kfold_best_acc = 0.0
    kfold_model_wts = copy.deepcopy(model.state_dict())
    best_model_dict = kfold_model_wts

    X=np.arange(len(dataset)*2).reshape(len(dataset),2)
    y=np.arange(190)
    for j in range(19):
      y[j*10:j*10+10]=j

    for train_index,val_index in kf.split(X, y):
      print('*'*25,'第',i,'折','*'*25)
      datasize = {'train':len(train_index), 'val':len(val_index)}
      print('train:',datasize['train'],'val:',datasize['val'])
      print('train_index %s, test_index %s'%(train_index, val_index))

      i += 1

      train_sampler = torch.utils.data.SubsetRandomSampler(train_index)
      valid_sampler = torch.utils.data.SubsetRandomSampler(val_index)
      dataloaders = {'train': torch.utils.data.DataLoader(image_dataset
s,
                     batch_size=batch_size, num_workers=4, sampler=train_s
ampler),
                'val': torch.utils.data.DataLoader(image_datasets,
                     batch_size=batch_size, num_workers=4, sampler=valid_s
ampler)
```

```
            }
      model.load_state_dict(kfold_model_wts) #每一折训练前 都重置为初始状
态

      model, k_acc = train_model(model, dataloaders, criterion, optimiz
er, scheduler, datasize, num_epochs)
      valid_acc_sum += float(k_acc)
      if Kfold_best_acc < k_acc:
        Kfold_best_acc = k_acc
        best_model_dict = copy.deepcopy(model.state_dict()) # 保存最佳的
训练参数

    valid_acc = valid_acc_sum/(i-1)
    print('*'*25,'END','*'*25)
    print('the best acc in kfold', float(Kfold_best_acc))
    print('teh average val acc', valid_acc)
    model.load_state_dict(best_model_dict)
    return model

"""## 搭建网络结构"""

model_ft = models.densenet169(pretrained=True)
num_ftrs = model_ft.classifier.in_features
model_ft.classifier = nn.Linear(num_ftrs, 19, bias=True)  # 输出参数修改

num_ochannels = model_ft.features.conv0.out_channels
model_ft.features.conv0 = nn.Conv2d(1, num_ochannels, kernel_size=(7, 7
), stride=(2, 2), padding=(3, 3), bias=False) # 输入参数修改

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma
=0.1)
# print (model_ft)

"""## 训练

K 折交叉验证
```

```
"""

model_ft = k_fold_train(5, image_datasets, model_ft, 4, criterion, opti
mizer_ft, exp_lr_scheduler, num_epochs=100)

# 针对最优模型再次训练，可选

#show acc
model = torch.load('light.pth')
eval_loss = 0.
eval_acc = 0.
s= 0.
with torch.no_grad():
  for i, (inputs, labels) in enumerate(test_dataloaders):
    inputs = inputs.to(device)
    labels = labels.to(device)

    outputs = model(inputs)
    _, preds = torch.max(outputs, 1)
    for j in range(inputs.size()[0]):
      #s =s + int(class_names[preds[j]])
      #print(class_names[preds[j]])
      #if int(class_names[preds[j]]) == int(labels[j]):
      if class_names[preds[j]] == class_names[int(labels[j])]:
        s = s+1
print (s)
print (s/(len(test_dataloaders) * 4))
```