第二部分：卷积神经网络

代码基本结构

我们采用最基本的LeNet-5网络进行搭建，将其中的sigmoid激活函数替换为更加常用的ReLU激活函数；利用AdamW优化器进行优化。

LeNet本质上就是两个卷积层和池化层，最后用一个两层MLP做分类；网络很小，相当适合做手写汉字分类这种小型任务。

首先，导入train set和validation set。这里的Grayscale是必须的，因为数据集中有几张图片并不天然是灰度图，如果不加Grayscale，会报错expected input to have 1 channels, but got 3 channels instead

```
transform = transforms.Compose([
    transforms.Grayscale(),
    transforms.Resize((28, 28)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.8876], std=[0.3158]) # Data computed from the train set
])

train_data = ImageFolder(r'./train', transform=transform)
val_data = ImageFolder(r'./validation', transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
val_loader = DataLoader(val_data, batch_size=64, shuffle=False)
```

定义我们的网络，在这一阶段，我们就使用最原始的LeNet-5结构

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet,self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, padding=2)
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(16 * 5 * 5, 120)
        self.linear2 = nn.Linear(120, 84)
        self.linear3 = nn.Linear(84, 12)
```

```
12
13      def forward(self,input): #...
```

利用AdamW进行优化，使用余弦退火的lr decay策略

```
cross_entropy = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(net.parameters(), lr=3e-4)
scheduler =
torch.optim.lr_scheduler.CosineAnnealingLR(optimizer=optimizer,T_max=max_iter)
```

进行训练：

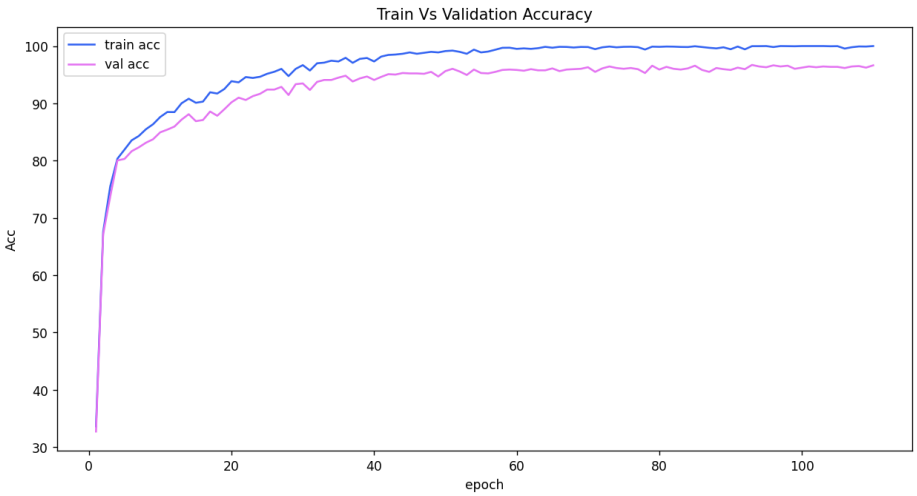
```
max_iter = 50
train_acc = []
val_acc = []
max_acc = 0
net = LeNet()

for epoch in range(max_iter):
    for i, (img, y) in enumerate(train_loader):
        predict_y = net(img)
        loss = cross_entropy(predict_y, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

    train_acc.append(train:=evaluate(net, train_loader))
    val_acc.append(val:=evaluate(net, val_loader))

    if val > max_acc:
        max_acc = val
    torch.save(net.state_dict(), f'val{max_acc}.pth')

    print(f'epoch {epoch}/{max_iter} loss: {loss.item()} train acc: {train},
val acc: {val}')
```

设计实验改进网络并论证

（以下所有实验均设置batchsize=64，仅考察网络结构的影响）

首先直接在LeNet-5上进行实验

```
epoch 90/110 loss: 0.0035358103923499584 train acc: 99.91599462365592, val acc: 96.23655913978494
epoch 91/110 loss: 0.002197315450757742 train acc: 99.42876344086021, val acc: 95.96774193548387
epoch 92/110 loss: 0.0038617094978690147 train acc: 99.98319892473118, val acc: 96.70698924731182
epoch 93/110 loss: 0.002358717378228903 train acc: 99.98319892473118, val acc: 96.43817204301075
epoch 94/110 loss: 0.0005150259821675718 train acc: 100.0, val acc: 96.30376344086021
epoch 95/110 loss: 0.013712611980736256 train acc: 99.83198924731182, val acc: 96.63978494623656
epoch 96/110 loss: 0.002446731086820364 train acc: 100.0, val acc: 96.43817204301075
epoch 97/110 loss: 0.00393671914935112 train acc: 99.98319892473118, val acc: 96.5725806451613
epoch 98/110 loss: 0.006184499710798263 5 train acc: 99.96639784946237, val acc: 96.03494623655914
epoch 99/110 loss: 0.0033404724672436714 train acc: 100.0, val acc: 96.23655913978494
epoch 100/110 loss: 0.001230711815878749 train acc: 100.0, val acc: 96.43817204301075
epoch 101/110 loss: 0.002911951392889023 train acc: 100.0, val acc: 96.30376344086021
epoch 102/110 loss: 0.0015712130116298795 train acc: 100.0, val acc: 96.43817204301075
epoch 103/110 loss: 0.0007702375878579915 train acc: 99.96639784946237, val acc: 96.37096774193549
epoch 104/110 loss: 0.001066322554834187 train acc: 99.98319892473118, val acc: 96.37096774193549
epoch 105/110 loss: 0.008579650893807411 train acc: 99.56317204301075, val acc: 96.16935483870968
epoch 106/110 loss: 0.0014970744960010052 train acc: 99.79838709677419, val acc: 96.43817204301075
epoch 107/110 loss: 0.010224247351288795 train acc: 99.93279569892474, val acc: 96.50537634408602
epoch 108/110 loss: 0.002285977825522423 train acc: 99.91599462365592, val acc: 96.23655913978494
epoch 109/110 loss: 0.0016723161097615957 train acc: 100.0, val acc: 96.63978494623656
96.70698924731182
```



LeNet-5

最高正确率只有**96.71**，我们将会看到，这个准确率并不令人满意。

接下来我们尝试不同的CNN架构。

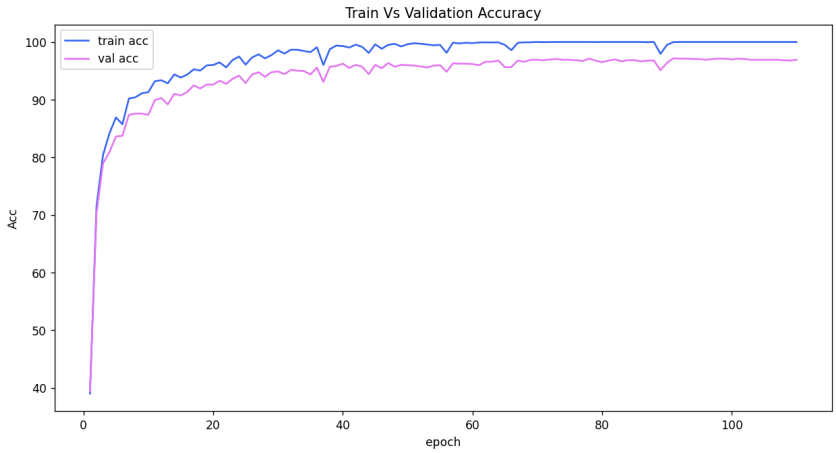
1. VGG作为CNN领域中相当重要的工作，规范了CNN的layer设计

我们接下来按照VGG的规则改造：

- 所有的卷积层 3*3, padding = 1，保证经过conv后图像大小不变
- 所有的池化层 2*2, stride = 2，保证经过pooling后图像大小减少一半
- 通道数目在池化层后翻倍

```
class VGG_Style_LeNet(nn.Module):
    def __init__(self):
        super(VGG_Style_LeNet,self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(6, 6, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(6, 12, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(12, 12, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(12 * 7 * 7, 120)
```

```
        self.linear2 = nn.Linear(120, 84)
        self.linear3 = nn.Linear(84, 12)

    def forward(self,input): #...
```




```
epoch 56/110 loss: 0.0030211019329726696 train acc: 99.88239247311827, val acc: 96.30376344086021
epoch 57/110 loss: 0.008482126519083977 train acc: 99.74798387096774, val acc: 96.23655913978494
epoch 58/110 loss: 0.0016168456058949232 train acc: 99.86559139784946, val acc: 96.23655913978494
epoch 59/110 loss: 0.010682533495128155 train acc: 99.79838709677419, val acc: 96.16935483870968
epoch 60/110 loss: 0.023122956976294518 train acc: 99.91599462365592, val acc: 95.96774193548387
epoch 61/110 loss: 0.0057548475451767445 train acc: 99.93279569892474, val acc: 96.5725806451613
epoch 62/110 loss: 0.005252412054687738 train acc: 99.91599462365592, val acc: 96.5725806451613
epoch 63/110 loss: 0.01819021627306938 train acc: 99.93279569892474, val acc: 96.7741935483871
epoch 64/110 loss: 0.006694206967949867 train acc: 99.49596774193549, val acc: 95.63172043010752
epoch 65/110 loss: 0.17818298935890198 train acc: 98.57190860215054, val acc: 95.63172043010752
epoch 66/110 loss: 0.011142244562506676 train acc: 99.86559139784946, val acc: 96.7741935483871
epoch 67/110 loss: 0.0037284570280462503 train acc: 99.93279569892474, val acc: 96.5725806451613
epoch 68/110 loss: 0.002492198720574379 train acc: 99.94959677419355, val acc: 96.90860215053763
epoch 69/110 loss: 0.0013656642986461152 train acc: 100.0, val acc: 96.90860215053763
epoch 70/110 loss: 0.0018331871833652258 train acc: 99.96639784946237, val acc: 96.84139784946237
epoch 71/110 loss: 0.0003939210146199912 train acc: 99.98319892473118, val acc: 96.9758064516129
epoch 72/110 loss: 0.0009100272436626256 train acc: 100.0, val acc: 97.04301075268818
```

VGG-style

在第72次epoch，第一次达到了准确率97。**最高的准确率为97.18**。单纯从epoch数目上来看，要快于LeNet-5，但是其每一次epoch所花费的时间要大于LeNet-5。准确率是高于LeNet的。

2. 接下来加入batchnormalization，dropout

加入batch normalization以加快训练，同时略微提高泛化性能。此外，在最后的MLP处，加入dropout策略（MLP权重数目多，不稀疏，从而dropout有比较好的效果；卷积层天然比较稀疏，dropout不需要）

```
class Modified_VGG_Style_LeNet(VGG_Style_LeNet):
    def __init__(self):
        super().__init__()
        self.norm1 = nn.BatchNorm2d(6)
        self.norm2 = nn.BatchNorm2d(6)
        self.norm3 = nn.BatchNorm2d(12)
        self.norm4 = nn.BatchNorm2d(12)
        self.norm5 = nn.BatchNorm1d(120)
        self.norm6 = nn.BatchNorm1d(84)
        self.dropout1 = nn.Dropout(p=0.5)
```

```
        self.dropout2 = nn.Dropout(p=0.5)

    def forward(self,input):# ...
```

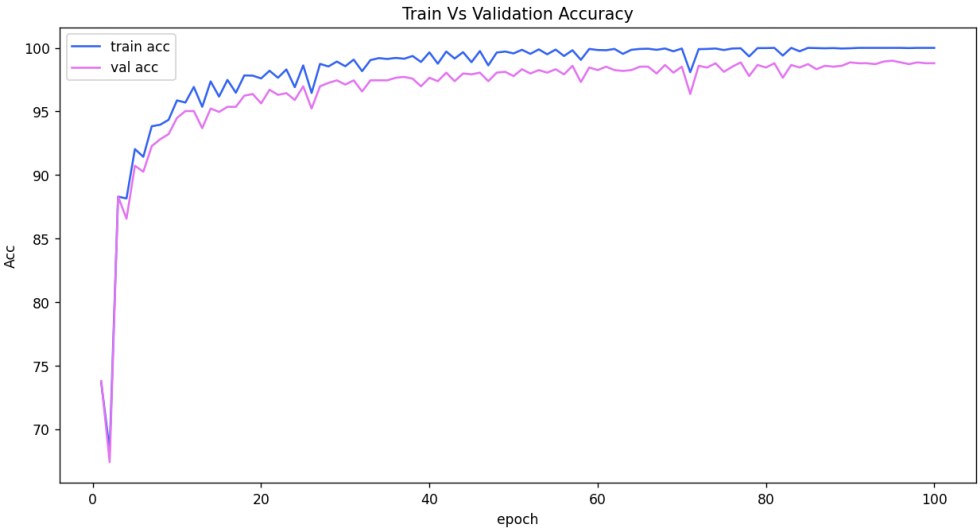
```
(base) PS C:\Users\lyk\Desktop\AIlab> python .\torchcnnbmp.py
epoch 0/50 loss: 2.0773863792419434 train acc: 67.55712365591398, val acc: 66.12903225806451
epoch 1/50 loss: 0.3305223286151886 train acc: 91.38104838709677, val acc: 89.04569892473118
epoch 2/50 loss: 0.11686895042657852 train acc: 94.69086021505376, val acc: 92.27150537634408
epoch 3/50 loss: 0.07739578932523727 train acc: 96.89180107526882, val acc: 94.01881720430107
epoch 4/50 loss: 0.10074853152036667 train acc: 97.86626344086021, val acc: 94.8252688172043
epoch 5/50 loss: 0.032630808651447296 train acc: 98.75672043010752, val acc: 95.56451612903226
epoch 6/50 loss: 0.11819347739219666 train acc: 99.1263440860215, val acc: 95.83333333333333
epoch 7/50 loss: 0.012670909985899925 train acc: 99.66397849462365, val acc: 96.37096774193549
epoch 8/50 loss: 0.0031628033611923456 train acc: 99.68077956989248, val acc: 96.63978494623656
epoch 9/50 loss: 0.029036719352006912 train acc: 99.14314516129032, val acc: 96.23655913978494
epoch 10/50 loss: 0.009284331463277344 train acc: 99.79838709677419, val acc: 96.50537634408602
epoch 11/50 loss: 0.07665139436721802 train acc: 99.59677419354838, val acc: 96.03494623655914
epoch 12/50 loss: 0.0031106453388929367 train acc: 99.56317204301075, val acc: 96.50537634408602
epoch 13/50 loss: 0.019691266119480133 train acc: 99.98319892473118, val acc: 97.1774193548387
epoch 14/50 loss: 0.013085154816508293 train acc: 100.0, val acc: 97.1774193548387
epoch 15/50 loss: 0.005856011062860489 train acc: 100.0, val acc: 97.31182795698925
epoch 16/50 loss: 0.0021927033085376024 train acc: 100.0, val acc: 97.37903225806451
epoch 17/50 loss: 0.00033682826324366033 train acc: 100.0, val acc: 97.44623655913979
epoch 18/50 loss: 0.0025729283224479129 train acc: 100.0, val acc: 97.11021505376344
epoch 19/50 loss: 0.0016181098762899637 train acc: 100.0, val acc: 97.44623655913979
epoch 20/50 loss: 0.0010859788162633777 train acc: 100.0, val acc: 97.51344086021506
epoch 21/50 loss: 0.0015344424173235893 train acc: 100.0, val acc: 97.1774193548387
epoch 22/50 loss: 0.0003992248675785959 train acc: 100.0, val acc: 97.44623655913979
epoch 23/50 loss: 0.0003176771861035377 train acc: 100.0, val acc: 97.44623655913979
epoch 24/50 loss: 0.0004031824937555939 train acc: 100.0, val acc: 97.58064516129032
epoch 25/50 loss: 0.00010940861102426425 train acc: 100.0, val acc: 97.44623655913979
epoch 26/50 loss: 0.0012462357990443707 train acc: 100.0, val acc: 97.51344086021506
epoch 27/50 loss: 0.0004784290213137865 train acc: 100.0, val acc: 97.37903225806451
```

在第13次epoch第一次达到了准确率97以上，**最高准确率为97.71**。可以看到训练相比之下十分迅速，而且准确率略微高于原始的VGG-style。

3. 在以GoogLeNet和ResNet为代表的后期CNN架构中，将最后的MLP更改为一个全的平均池化层。

我们也做这样的尝试，即增大channel的数目，最终通过一个大的average pooling来得到线性层，不需要再对MLP进行训练：

```
class AvgPool_Modified_VGG(Modified_VGG_Style_LeNet):
    def __init__(self):
        super().__init__()
        self.norm1 = nn.BatchNorm2d(32)
        self.norm2 = nn.BatchNorm2d(32)
        self.norm3 = nn.BatchNorm2d(64)
        self.norm4 = nn.BatchNorm2d(64)
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.avgpool = nn.AvgPool2d(kernel_size=7, stride=1)
        self.linear_last = nn.Linear(64,12)

```



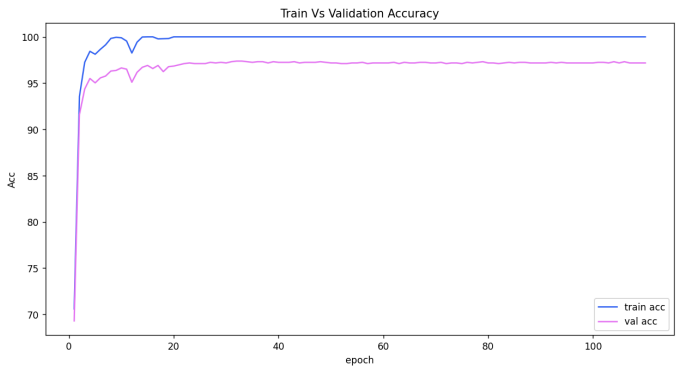
batchsize = 64，max_val_acc = 98.99

可以看到，这个模型相对更难训练，但是val和train之间的gap更小，即泛化性能更好，具有更大的潜能。其在**validation集上的准确率也更高，有98.99**。这是由于先前的MLP相比CNN更容易过拟合（其有多的多的可学习参数），将MLP替换为Avg Pooling当然可以显著地减缓过拟合现象。

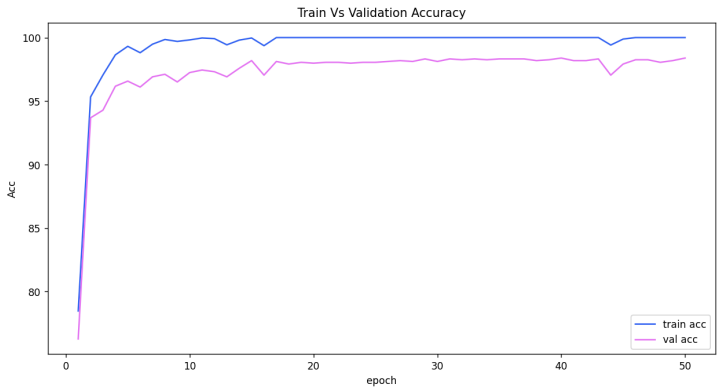
4. 最后考察batchsize的影响

在batchnorm & dropout & VGG-like的情况下考察batchsize的影响：

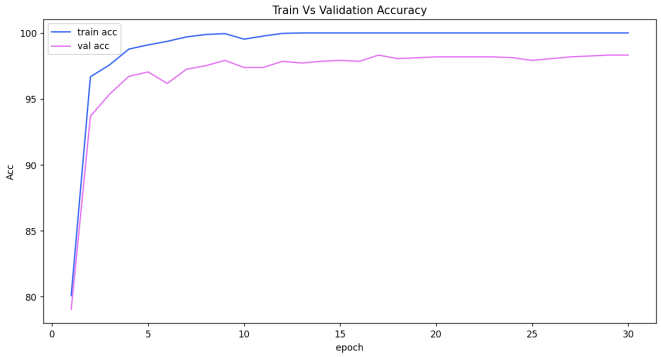
尝试batchsize = 32，16，8，4下最优的val_acc。



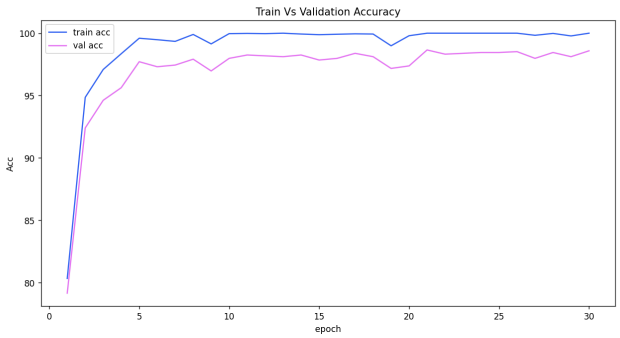
batchsize = 32 , max_val_acc = 97.38



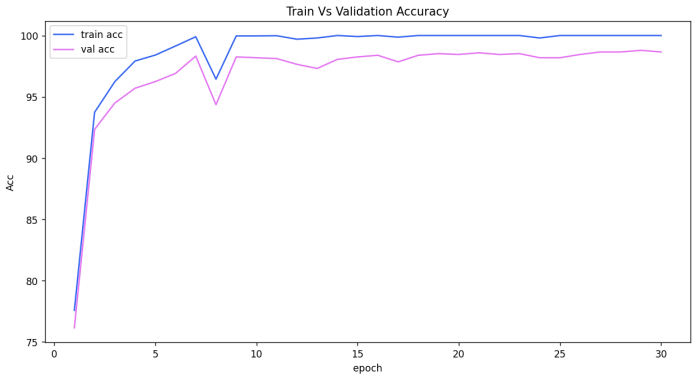
batchsize =16 , max_val_acc = 98.39



batchsize =16 , max_val_acc = 98.32



batchsize =8 , max_val_acc = 98.66



batchsize =4 , max_val_acc = 98.79

可以看到，随着batchsize的减小，尽量训练所花费的时间在增长，val_loss也在缓慢增长。这事实上具有理论依据：最小的batchsize训练得到的模型泛化性能往往更好，这是因为batchsize最小时，引入最多的随机性，使得模型得以收敛到一个更为平坦的极小值点（若该极小值点不那么平坦，则容易随机地游走出去）。

综合来看，我们来训练一个最好的模型，即batchnorm & dropout & VGG-like & batchsize = 4 & Full Avg Pooling，**此时得到的最好的val_acc为99.60**

5. 数据增强？

常见的CNN数据增强手段有翻折，旋转，噪点等；然而，对于28*28的黑白图像，大部分增强手段显得不那么有必要。此外，尽管在本任务下（即12分类），翻折和旋转应该能有不错的性能提升；但是更大范围手写汉字识别是肯定不能使用翻折和旋转作为数据增强手段的--汉字经过这种变化后往往会失去

其意义。此外，我们模型的准确率在测试集上也已经相当高了，在这里增加数据增强的手段来减少过拟合的现象似乎并不合适。

网络设计的理解

在实验中，本项目追溯了CNN发展的历史进程，从上个世纪的LeNet到VGG的模块化设计，再到GoogLeNet和ResNet加入的大型平均池化层（替代最后的MLP），一一测试了这些不同的结构。

可以看出，在我们这种较小的模型上，这些不同的模型在性能上并没有本质上的区别，在某些时候，甚至LeNet的表现要更好；但是后续的模型仍然有我们值得学习的点。

例如，VGG提出了三条规范化的规则，其能够保证图像的大小在经过conv层后不变，经过池化层后减少一半（减少的这一半“容积”通过对后续通道的翻倍得到补偿，即综合来看，图像的“体积”缓慢地倍减），这种大小一致的模块化设计，构成了后续更大更深的模型的基础。GoogLeNet添加了各种conv作为一个Inception Block，ResNet提供了被沿用至今的residual连接。这两个特定并没有在本项目中体现出来；与之相对，本项目考察了GoogLeNet和ResNet最后对MLP的替代。在大型模型中，MLP占了参数的主要部分，训练的开销也主要在MLP层，通过一个大型的平均池化层来替代最后的MLP可以有效地减少计算的开销。使得更大的模型得以训练。