



# 江蘇大學

**JIANGSU UNIVERSITY**

基于手写数据集的 CV 探讨

**MNIST**

姓名： 陆佳欢

学号： 3190105095

学院： 数科院

专业： 应用数学系

2022 年 4 月 6 日

### 摘要

本文通过在 MNist 数据集上通过分析各种网络结构正确认识 CV，具体认识与实现模型以求更加深刻的认识计算机视觉。

实验的第一部分：自制了 beamer 主题，目的用于报告的展示。以求达到能够清晰表达的作用，为了提高实验的效率，本文使用 Anaconda 也重新安装了 pytorch\_gpu 版。

实验的第二部分：从五部分来建模。首先通过写 MLP 模型，具体且清晰的实现了一般 DNN 建模的步骤，并且在初步微调模型得到了 test 集上 97% 的成绩。接着加大网络的深度，通过建立 LeNet、AlexNet 丰富了报告的内容，同时得到了更高的 acc，其中 LeNet 跑 10epoch 模型平均准确率有 97.6%，而 AlexNet 更深达到了 99.26%，也是这 5 个网络中最高的 acc。并且在 AlexNet 中，分别对比了手写以及模块化的不同，同时发现优化器在选择 adam 的时候再第一个 epoch 会出现很大的误差，而且这种情况不可逆，因而得选择 SGD。接着使用 GoogLeNet 去加大网络的深度，使用了 Inception V1 块，但是遗憾的是 GooLeNet 在跑 test 的时候并未得到预期的结果。在换成 SGD 的情况下依旧失效，未知其原因。最后，为了避免 ResNet 出现同样的情况。我使用目前正在使用的 fastai，使用它的轮子进行迁移学习。分别跑了 ResNet18 以及 ResNet34 得到 acc 分别是 99%、98%。模型的网络变深，test 的结果反倒变差了，尽管 ResNet 有 BN 层，但是有理由相信，模型已经出现了轻微的过拟合。

实验的第三部分：在具体认识模型的同时，提供了一些提高模型准确率的数据增强手段。

**关键词：** pytorch   MNIST   GoogNet   ResNet   fastai   迁移学习

# 目录

<b>1</b>	<b>安装环境</b>	<b>1</b>
1.1	beamer 主题的制作 . . . . .	1
1.2	pytorch 环境的配置 . . . . .	1
1.2.1	Anaconda 创建虚拟环境 . . . . .	1
1.2.2	安装 pytorch_gpu . . . . .	2
<b>2</b>	<b>实验部分</b>	<b>3</b>
2.1	MLP . . . . .	3
2.1.1	MLP 网络结构 . . . . .	3
2.1.2	train 可泛化 . . . . .	4
2.1.3	test 可泛化 . . . . .	4
2.1.4	准确率 . . . . .	4
2.2	LeNet . . . . .	5
2.2.1	LeNet 实验结果 . . . . .	7
2.3	AlexNet . . . . .	7
2.3.1	AlexNet 手写 . . . . .	7
2.3.2	AlexNet 实验结果 . . . . .	8
2.4	GoogLeNet . . . . .	9
2.4.1	inception . . . . .	9
2.4.2	Inception 块的解释 . . . . .	11
2.4.3	实验结果 . . . . .	11
2.5	ResNet . . . . .	11
2.5.1	基于 fastai 的代码框架 . . . . .	11
2.5.2	ResNet18 及 ResNet34 对比 . . . . .	11
<b>3</b>	<b>实验结果分析</b>	<b>12</b>
3.1	总结 . . . . .	12
3.2	改进模型 . . . . .	13
	<b>参考文献</b>	<b>13</b>

# 1 安装环境

## 1.1 beamer 主题的制作

在 31 日下午，我考虑用 PPT 来介绍我实验的主题，因为这样会让我的报告做的比较有条理感，于是我考虑用 beamer 来展示论文的结构。主要的原因是它能节省大量的时间。于是我开始寻找本校的 beamer 主题。遗憾的是并没有同仁做这件事。于是，兴致使然做了 beamer 江苏大学的非官方版。模板的制作参考了兰州大学和东南大学以及华东师范大学的制作；其中 UI 部分是艺术学院一位不愿留名的女生做的。最终在 31 晚上做完了这个主题，并且上传了 [latex](#) 搜索江苏大学便能寻到。或者打开我的 [github](#) 库也能找到本次实验的全部内容。



图 1: beamer 主题图

## 1.2 pytorch 环境的配置

由于我长时间使用 colab 以及 kaggle 或者云平台，我的 pytorch\_gpu 失效了。所以不得不考虑重新安装。

### 1.2.1 Anaconda 创建虚拟环境

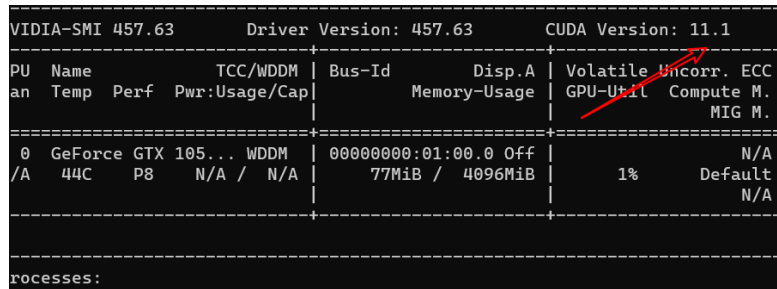
- 1、 打开 **cmd**.
- 2、 **conda init** 初始化.
- 3、 **conda create -n pytorch python=3.7** 创建一个 3.7 的环境.
- 4、 **conda env list** 查看环境.

5、 **conda activate pytorch** 转到 pyotrch 虚拟环境.

6、 **pip list** 查看包.

## 1.2.2 安装 pytorch\_gpu

1、 命令行中 **nvidia-smi** 查看显卡型号.



VIDIA-SMI 457.63				Driver Version: 457.63		CUDA Version: 11.1	
PU	Name	TCC/WDDM	Bus-Id	Disp.A	Volatile	Uncorr. ECC	
an	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.
0	GeForce GTX 105...	WDDM	00000000:01:00.0	Off		N/A	
/A	44C P8	N/A / N/A	77MiB / 4096MiB		1%	Default	N/A

rocesses:

图 2: 显卡型号 11.1

2、 打开pyotch找对应版本.

3、 复制对应链接到 cmd 里注意不要直接运行.

```
# CUDA 11.1
pip install torch==1.10.0+cu111 torchvision==0.11.0+cu111 torchaudio==0.10.0 -f https://download.pytorch.org
```

图 3: 我选择的版本

4、 我们需要分别安装这三个程序包，而不是网上的直接 enter(会失败).

```
pip install torch==1.10.0 -f https://download.pytorch.org/whl/torch_stable.html
```

图 4: 演示 1

```
pip install cu111 torchvision==0.11.0 -f https://download.pytorch.org/whl/torch_stable.html
```

图 5: 演示 2

5、 接着安装**cuda 与 cudnn**

注：红色可以直接访问

## 2 实验部分

### 2.1 MLP

MLP 是多层感知机模型，也是最基本的 DNN 之一，本次对 MNist 手写数据集的 MLP 部分采用 3 层结构，分别是输入层，隐藏层，以及输出层。

```
flag = torch.cuda.is_available() #是否可用返回 boolean
if flag:
    print("CUDA 可使用")
else:
    print("CUDA 不可用")
ngpu= 1 #有几张卡
# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
print("驱动为: ",device)
print("GPU 型号: ",torch.cuda.get_device_name(0))
```

上述代码能够输出 GPU 版本并且通用全部网络。

```
CUDA可使用
驱动为:  cuda:0
GPU型号:  GeForce GTX 1050 Ti
```

图 6: GPU

#### 2.1.1 MLP 网络结构

```
MLPNet(
  (h1): Linear(in_features=784, out_features=300, bias=True)
  (relu1): ReLU()
  (out): Linear(in_features=300, out_features=10, bias=True)
  (softmax): Softmax(dim=1)
)
```

图 7: MLP 网络结构

**解释说明:** 输入 784 的神经元，然后隐层的输出为 300 最后输出层为 10。相当于第一层有 784 个节点，然后隐藏层有 300 个，原因在于有些没连类似 Dropout，最后由于做的是 10 用 softmax 分类，所有输出层为 10 个节点。

### 2.1.2 train 可泛化

```
for epoch in range(num_epochs):
    for i, (x_images, y_labels) in enumerate(train_loader):
        # 因为全连接会把一行数据当做一条数据，因此我们需要将一张图片转换到一行上
        # 原始数据集的大小: [ 100, 1, 28, 28]
        # resize 后的向量大小: [-1, 784]
        images = x_images.reshape(-1, 28*28).to(device)
        labels = y_labels.to(device)
        # 正向传播以及损失
        y_pre = model(images) #前向传播
        loss = criterion(y_pre, labels) # 计算损失函数
        # 反向传播
        # 梯度清空，反向传播，权重更新
        optimizer.zero_grad() #梯度归零 因为训练的过程通常使用mini-batch方法，所以如果
                                # 不将梯度清零的话，梯度会与上一个batch的数据相关，因此该函数要写在反向传播
                                # 和梯度下降之前。
        loss.backward()
        optimizer.step() #执行一次优化步骤，通过梯度下降法来更新参数的值。因为梯度下降
                        # 是基于梯度的所以在执行optimizer.step()函数前应先执行loss.backward()函数来
                        # 计算梯度。

    if (i+1) % 64 == 0:
        print( f'epoch [{epoch+1}/{num_epochs}], step [{i+1}/{n_total_steps}], Loss: {
            loss.item():.4f}')
        print("模型训练完成")
```

### 2.1.3 test 可泛化

```
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()
    acc = 100.0 * n_correct / n_samples
    print(f'Accuracy of the network on the 10000 test images: {acc} %')
```

### 2.1.4 准确率

Accuracy of the network on the 10000 test images: 97.14 %

图 8: MLP 网络准确率

## 2.2 LeNet

LeNet-5 网络的默认的输入图片的尺寸是 32\*32, 而 Mnist 数据集的图片的尺寸是 28 \* 28, 因此, 采用 Mnist 数据集时, 每一层的输出的特征值 feature map 的尺寸与 LeNet-5 网络的默认默认的特征 map 的尺寸是不一样的, 需要适当的调整。

```
class LeNet(nn.Module):
    # 定义构造方法函数, 用来实例化
    def __init__(self):
        super(LeNet, self).__init__() # 5层, 2个卷积层+3个fc全连接层
        # 1*1*28*28
        # 第一个卷积层: 输入通道数=1, 输出通道数=6, 卷积核大小=5*5, 默认步长=1
        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 6, kernel_size = 5)
        # 6 * 24 * 24
        # 第二个卷积层: 输入通道数=6, 输出通道数=16, 卷积核大小=5*5, 默认步长=1
        self.conv2 = nn.Conv2d(in_channels = 6, out_channels = 16, kernel_size = 5)
        # 16 * 8 * 8

        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(in_features = 16 * 4 * 4, out_features= 120)
        # 16 * 4 * 4
        self.fc2 = nn.Linear(in_features = 120, out_features = 84)
        self.fc3 = nn.Linear(in_features = 84, out_features = 10)
        # 第一个全连接层: 输入特征数=256, 输出特征数=120
        # 也可以理解成: 将256个节点连接到120个节点上
        # 第三个全连接层: 输入特征数=84, 输出特征数=10 (这10个维度我们作为0-9的标识来确定识别出的是那个数字。)
        # 也可以理解成: 将84个节点连接到10个节点上
    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        # x = F.log_softmax(x, dim=1) # 这个问题前面FC建模已经说过了 参考官方LeNet
    return x
```

输出公式 = (M-K+2P) / S + 1

M: 输入神经元个数/大小

K: 卷积大小

P: 零填充

S: 步长

第一个卷积层: self.conv1=nn.Conv2d(1,6,5) :

其参数意义为:

输入通道为 1 (输入图像是灰度图)

输出通道为 6

卷积核 kernel\_size 为 5\*5

24输出维度 = 28输入维度 - 5卷积核size + 1



所以输出 shape 为:  $6 \times 24 \times 24$

第一个激活函数: `out = F.relu(out)`

输出维度不变 仍为  $6 \times 24 \times 24$

第一个最大池化层: `out = F.max_pool2d(out, 2, 2)`

该最大池化层在  $2 \times 2$  空间里向下采样。

12 输出维度 = 24 输入维度 / 2。

所以输出 shape 为:  $6 \times 12 \times 12$

第二个卷积层 `self.conv2=nn.Conv2d(6,16,5)`

其参数意义为:

输入通道为 6 (第一个最大池化层的输出通道数)

输出通道为 16 (需要用到的卷积核就有 16 种)

卷积核 `kernel_size` 为  $5 \times 5$

8 输出维度 = 12 输入维度 - 5 卷积核 size + 1

所以输出 shape 为:  $16 \times 8 \times 8$

第二个激活函数 `out = F.relu(out)`

特征提取结束

第二个最大池化层: `out = F.max_pool2d(out, 2, 2)`

该最大池化层在  $2 \times 2$  空间里向下采样。

4 输出维度 = 8 输入维度 / 2。

所以输出 shape 为:  $16 \times 4 \times 4$

输出前的数据预处理

因为全连接层 `Linear` 的输出为最后的输出,

而全连接层 `Linear` 要求的输入为展平后的多维的卷积成的特征图 (特征图为特征提取部分的结果)

输出前的数据预处理结束

输出即全连接层

第一个全连接层 `self.fc1=nn.Linear(16*4*4, 120)`

输入维度为  $16 \times 4 \times 4 = 256$

设定的输出维度为  $120 \times 1$

激活函数 `out = F.relu(out)`

输出维度不变, 仍为  $120 \times 1$

第二个全连接层 `self.fc2=nn.Linear(120, 84)`

输入维度为 84

设定的输出维度为  $84 \times 1$

第三个激活函数 `out = F.relu(x)`

输出维度不变, 仍为  $84 \times 1$

第三个全连接层 `self.fc3=nn.Linear(84, 10)`

输入维度为  $10 \times 1$

输出维度设定为  $10 \times 1$  (因为是一个 10 分类的问题, 所以最后要变成  $10 \times 1$ )

第三个激活函数 `out = F.log_softmax(out, dim=1)`

用 `F.log_softmax()` 将数据的范围改到  $[0, 1]$  之内, 表示概率。

输出维度仍为  $10 \times 1$ , 其值可以视为概率。

### 2.2.1 LeNet 实验结果

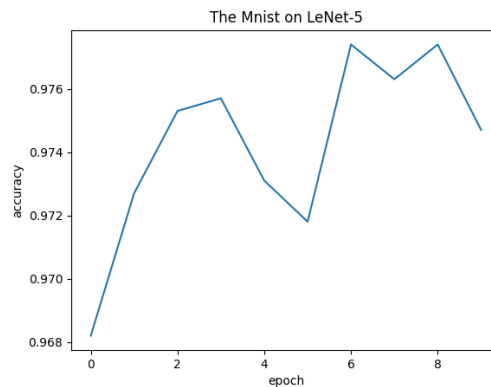


图 9: LeNet10\_epoch\_acc

### 10 epoch 平均 acc

```
Accuracy of the network on the 10000 test images: 0.9742599999999999 %
```

图 10: LeNet10\_sqr\_acc

## 2.3 AlexNet

### 2.3.1 AlexNet 手写

```
class AlexNet(nn.Module):
    # 定义构造方法函数，用来实例化
    def __init__(self):
        super(AlexNet, self).__init__() #
        self.conv1=nn.Conv2d(in_channels=1,out_channels=96,kernel_size=11,
                               stride=4,padding=1)
        self.conv2=nn.Conv2d(in_channels=96,out_channels=256,kernel_size=5,
                               padding=2)
        #接下来连续3分卷积层和较小的卷积窗口
        self.conv3=nn.Conv2d(256,384,kernel_size=3,padding=1)
        self.conv4=nn.Conv2d(384,384,kernel_size=3,padding=1)
        self.conv5=nn.Conv2d(384,256,kernel_size=3,padding=1)
        self.fc1=nn.Linear(6400,4096)
        self.fc2=nn.Linear(4096,4096)
        self.fc3=nn.Linear(4096,10)
        # AlexNet本来是做1000分类的，我们采用迁移学习的思想该为10fc3
    def forward(self, x): # 输入shape: torch.Size([1, 1, 224, 224])
        x=F.max_pool2d(F.relu(self.conv1(x)),kernel_size=3,stride=2)
        x=F.max_pool2d(F.relu(self.conv2(x)),kernel_size=3,stride=2)
        x=F.relu(self.conv3(x))
```

```
x=F.relu(self.conv4(x))
x=F.max_pool2d(F.relu(self.conv5(x)),kernel_size=3,stride=2)
x=torch.flatten(x,1)
x=F.relu(self.fc1(x))
x=F.dropout(x,p=0.5)
x=F.relu(self.fc2(x))
x=F.dropout(x,p=0.5)
x=self.fc3(x)

return x
```

神经网络手写其实是非常简单的，只要把每一层写出来然后连起来就行。AlexNet 是在 LeNet 的基础上增加了一些层，同时加入 pooling 这样可以突出特征，使用更多的层，一般来讲会提高模型的 acc，当层数过深的时候，也会出现模型能力差的情况，比如梯度爆炸，参数过多，导致难以训练，一般一个模型的提升从数据处理上做文章，然后是模型更深或者更宽。

### 2.3.2 AlexNet 实验结果

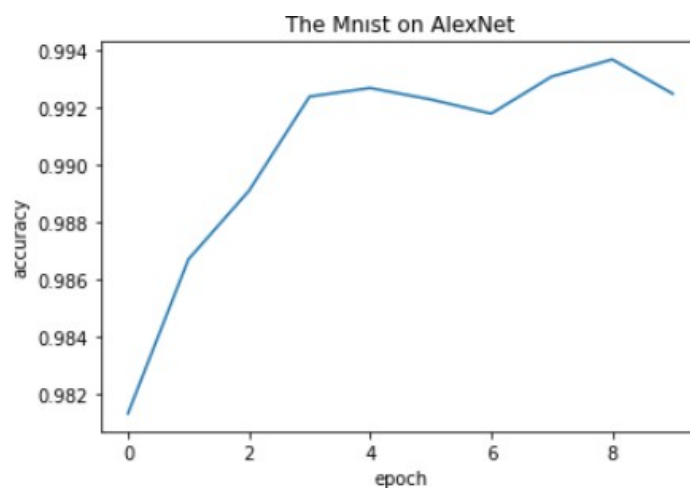


图 11: LeNet10\_epoch\_acc

### AlexNet\_10epoch

```
Accuracy of the network on the 10000 test images: 99.056 %
```

图 12: AlexNet\_sqr\_acc

## 2.4 GoogLeNet

inception（也称 GoogLeNet）是 2014 年 Christian Szegedy 提出的一种全新的深度学习结构，在这之前的 AlexNet、VGG 等结构都是通过增大网络的深度（层数）来获得更好的训练效果，但层数的增加会带来很多副作用，比如 overfit、梯度消失、梯度爆炸等。inception 的提出则从另一种角度来提升训练结果：能更高效的利用计算资源，在相同的计算量下能提取到更多的特征，从而提升训练结果。

一般来说，提升网络性能最直接的办法就是增加网络深度和宽度，但一味地增加，会带来诸多问题：

- 1、参数太多，如果训练数据集有限，很容易产生过拟合；
- 2、网络越大、参数越多，计算复杂度越大，难以应用；
- 3、网络越深，容易出现梯度弥散问题（梯度越往后穿越容易消失），难以优化模型。我们希望在增加网络深度和宽度的同时减少参数，为了减少参数，自然就想到将全连接变成稀疏连接。但是在实现上，全连接变成稀疏连接后实际计算量并不会质的提升，因为大部分硬件是针对密集矩阵计算优化的，稀疏矩阵虽然数据量少，但是计算所消耗的时间却很难减少。

### 2.4.1 inception

Inception 结构的主要思路是怎样用密集成分来近似最优的局部稀疏结构。

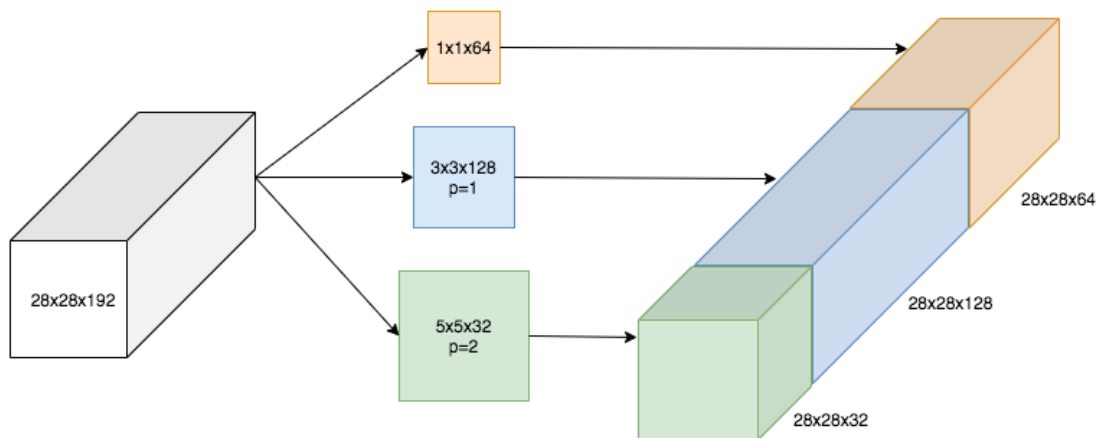
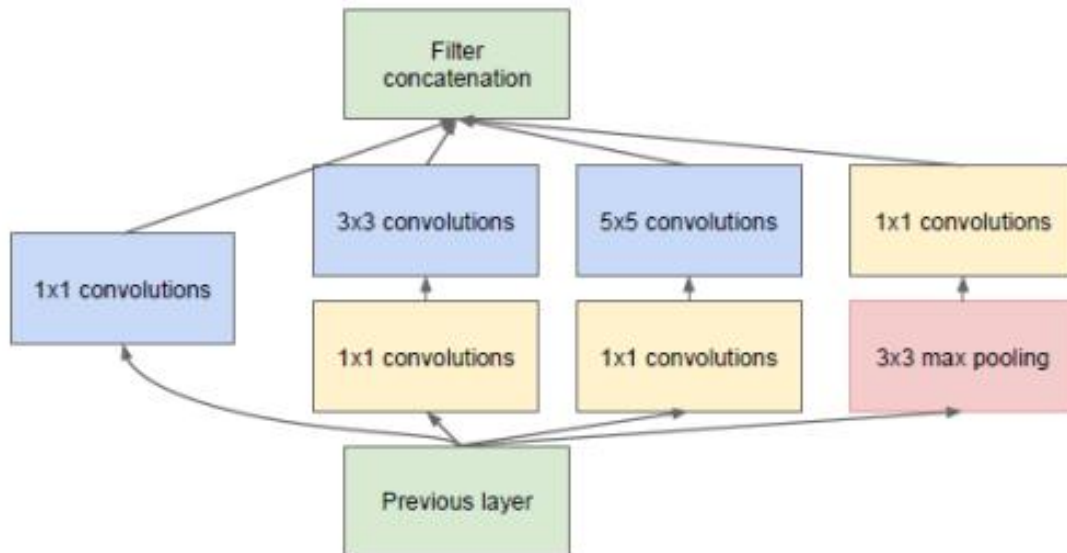


图 13: Inception 块

Naive Inception 单元的详细工作过程，假设在上图中 Naive Inception 单元的前一层输入的数据是一个 32x32x256 的特征图，该特征图先被复制成 4 份并分别

被传至接下来的 4 个部分。我们假设这 4 个部分对应的滑动窗口的步长均为 1，其中， $1 \times 1$  卷积层的 *Padding* 为 0，滑动窗口维度为  $1 \times 1 \times 256$ ，要求输出的特征图深度为 128； $3 \times 3$  卷积层的 *Padding* 为 1，滑动窗口维度为  $3 \times 3 \times 256$ ，要求输出的特征图深度为 192； $5 \times 5$  卷积层的 *Padding* 为 2，滑动窗口维度为  $5 \times 5 \times 256$ ，要求输出的特征图深度为 96； $3 \times 3$  最大池化层的 *Padding* 为 1，滑动窗口维度为  $3 \times 3 \times 256$ 。这里对每个卷积层要求输出的特征图深度没有特殊意义，仅仅举例用，之后通过计算，分别得到这 4 部分输出的特征图为  $32 \times 32 \times 128$ 、 $32 \times 32 \times 192$ 、 $32 \times 32 \times 96$  和  $32 \times 32 \times 256$ ，最后在合并层进行合并，得到  $32 \times 32 \times 672$  的特征图，合并的方法是将各个部分输出的特征图相加，最后这个 Naive Inception 单元输出的特征图维度是  $32 \times 32 \times 672$ ，总的参数量就是  $1 \times 1 \times 256 \times 128 + 3 \times 3 \times 256 \times 192 + 5 \times 5 \times 256 \times 96 = 1089536$ 。但是 Naive Inception 有两个非常严重的问题：首先，所有卷积层直接和前一层的输入数据对接，所以卷积层中的计算量会很大；其次，在这个单元中使用的最大池化层保留了输入数据的特征图的深度，所以在最后进行合并时，总的输出的特征图的深度只会增加，这样增加了该单元之后的网络结构的计算量。于是人们就要想办法减少参数量来减少计算量，在受到了模型“Network in Network”的启发，开发出了在 GoogleNet 模型中使用的 Inception 单元（Inception V1），这种方法可以看做是一个额外的  $1 \times 1$  卷积层再加上一个 ReLU 层。如下所示：



(b) Inception module with dimension reductions

图 14: Inception v1

### 2.4.2 Inception 块的解释

inception 模块的基本机构上图，整个 inception 结构就是由多个这样的 inception 模块串联起来的。inception 结构的主要贡献有两个：一是使用 1x1 的卷积来进行升降维（利用 NiN 的手段）；二是在多个尺寸上同时进行卷积再聚合。

### 2.4.3 实验结果

我用 GoogLeNet 建模失败了，原因不知。它不像 AlexNet 那样如果出错会出现一个较大的误差，GoogLeNet 是一个 epoch 的均误差，即每个 epoch 误差都很大，已知道不存在 opt 的问题。具体原因还未找到，模型的建立可以查我的 jupyter 实验报告。

## 2.5 ResNet

由于模型建模与 googLeNet 的相似性，我便未考虑去直接建模。而是尝试用 fastai 框架去建模，fastai 是继承 pytorch 的一个框架，吸引我使用它的原因是。一方面, 我正在读他们的书知道他们数据增强部分做的很好；另一方面建模的流程非常简单，并且采取迁移学习的策略很容易能够达到预期的结果。

### 2.5.1 基于 fastai 的代码框架

```
import fastbook
fastbook.setup_book()
from fastai.vision.all import *
from fastbook import *
matplotlib.rc('image', cmap='Greys')
path = untar_data(URLs.MNIST)#下载手写数据集
path.ls()#获取数据集的地址
# 使用路径创建一个图像数据转换器对象
dls = ImageDataLoaders.from_folder(path, train='training',
valid='testing')# 创建 DataLoader
learn = cnn_learner(dls, resnet18, pretrained=False,
loss_func=LabelSmoothingCrossEntropy(), #标签平滑或者换resnet18
metrics=accuracy)
learn.fit_one_cycle(1, 0.1)# 一个epoch, 学习率0.1
# 使用迁移学习训练
learn.unfreeze()
learn.lr_find()
learn.fit_one_cycle(3, slice(1e-6, 1e-4))
```

### 2.5.2 ResNet18 及 ResNet34 对比

**Table 1** ResNet18

<i>epoch</i>	<i>train_loss</i>	<i>valid_loss</i>	<i>accuracy</i>	<i>time</i>
0	0.561522	0.526307	0.991500	12:55
1	0.559923	0.525104	0.990900	10:18
2	0.548604	0.524869	0.991600	10:19
3	0.549611	0.524795	0.991000	10:22

**Table 2** ResNet34

<i>epoch</i>	<i>train_loss</i>	<i>valid_loss</i>	<i>accuracy</i>	<i>time</i>
0	0.583426	0.594473	0.986200	22:21
1	0.578651	0.543047	0.987100	16:14
2	0.581934	2.380104	0.983400	16:49

我们可以看到的是当 Mnist 选择 ResNet18 的时候 acc 反而比 ResNet34 的效果好，也表明了模型并不是越深越好的事实。通过 fastai 自带包可以画出学习率图如下：

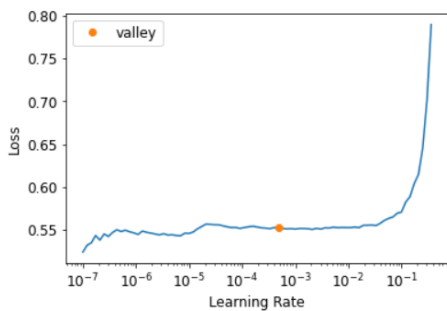


图 15: ResNet18 learning-loss

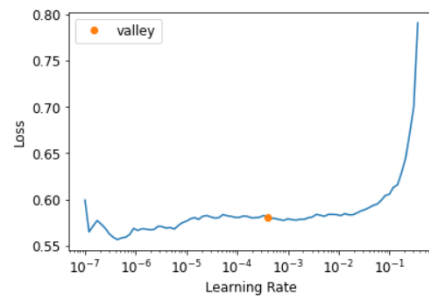


图 16: ResNet34 learning-loss

### 3 实验结果分析

#### 3.1 总结

Mnist 手写数据集本身不是一个复杂的数据集，而且它是单通道的图片，因而建模起来并不复杂。如果数据增强做得好点的话，train 部分是可以训练到 100%。但是过多的 epoch 也可能导致模型过拟合，从而在 test 上面 acc 下降，例如 ResNet18 和 ResNet34, ResNet18 的 4 个 epoch 平均的 acc 为 99.1%，而 ResNet34 却只有 98%。可能就是 train 过拟合了。在 5 个网络里，其中训练效果最好的是 AlexNet，它的模型效果达到 99.26%。可能也是因为并不复杂的原因。考察 5 个网络，发现均能超过 97%，可见效果还不错。

### 3.2 改进模型

可以进一步做数据增强, fastai 中提供了 mixup 混合形式, 这样使得图片直接更加独立。对于 googLeNet 可以将 5x5 的卷积核换成 3x3 的, 并且使用 1x1 降维。不要 Resize 它的尺寸, 直接建模。因为 Resize 可能会损失图像的一些特征。

## 参考文献

- [1] Yu H, Yang L T, Zhang Q, et al. Convolutional neural networks for medical image analysis: state-of-the-art, comparisons, improvement and perspectives[J]. Neuro-computing, 2021, 444: 92-110.
- [2] 机器学习及其应用 [M]. 清华大学出版社有限公司, 2006.
- [3] Targ S, Almeida D, Lyman K. Resnet in resnet: Generalizing residual architectures[J]. arXiv preprint arXiv:1603.08029, 2016.



## 附录 A：网络 LeNet

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torch.optim as optim
import matplotlib.pyplot as plt #数据可视化
import numpy as np
from PIL import Image
from torchvision import datasets, transforms

learning_rate = 0.01
batch_size = 64
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
# 将数据集下载到指定目录下,这里的transform表示,数据加载时所需要做的预处理操作
# 加载训练集合
train_dataset = torchvision.datasets.MNIST(
    root='.data',
    train=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ]), # 转化为tensor且数据标准化注意在FC中我们未做数据增强
    download=True)
# 加载测试集合
test_dataset = torchvision.datasets.MNIST(
    root='.data',
    train=False,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ]),
    download=True)

train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=batch_size, # 一个批次的大小为128张
    shuffle=True # 随机打乱
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    shuffle=True
)

class LeNet(nn.Module):
    # 定义构造方法函数,用来实例化
```

```
def __init__(self):
    super(LeNet, self).__init__() # 5层, 2个卷积层+3个fc全连接层
    # 1*1*28*28
    # 第一个卷积层: 输入通道数=1, 输出通道数=6, 卷积核大小=5*5, 默认步长=1
    self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5) # 6 * 24
        * 24
    # 第二个卷积层: 输入通道数=6, 输出通道数=16, 卷积核大小=5*5, 默认步长=1
    self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5) # 16 *
        8 * 8

    # an affine operation: y = Wx + b
    self.fc1 = nn.Linear(in_features=16 * 4 * 4, out_features=120) # 16 * 4 * 4
    self.fc2 = nn.Linear(in_features=120, out_features=84)
    self.fc3 = nn.Linear(in_features=84, out_features=10)
    # 第一个全连接层: 输入特征数=256, 输出特征数=120
    # 也可以理解成: 将256个节点连接到120个节点上
    # 第三个全连接层: 输入特征数=84, 输出特征数=10 (这10个维度我们作为0-9的标识来
        确定识别出的是那个数字。)
    # 也可以理解成: 将84个节点连接到10个节点上

def forward(self, x):
    # Max pooling over a (2, 2) window
    x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
    x = F.max_pool2d(F.relu(self.conv2(x)), 2)
    x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    # x = F.log_softmax(x, dim=1) # 这个问题前面FC建模已经说过了 参考官方LeNet
    return x

def train(epoch):
    run_loss = 0.0
    for batch_idx, data in enumerate(train_loader, 0): # 循环次数batch_idx的最大循环值
        +1 = (MNIST数据集样本总数60000 / BATCH_SIZE )
        inputs, target = data
        inputs, target = inputs.to(device), target.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, target)
        loss.backward() # 反向计算梯度
        optimizer.step() # 优化参数
        run_loss += loss.item()
        if batch_idx % 200 == 199:
            print('[%d, %5d] loss: %.3f' % (epoch + 1, batch_idx + 1, run_loss / 200))
            run_loss = 0.0

def test():
    correct = 0
    total = 0
    with torch.no_grad(): # 此时已经不需要计算梯度, 也不会进行反向传播
        for data in test_loader:
            images, labels = data
```

```
        images, labels = images.to(device), labels.to(device) #将数据转移到cuda上
        outputs = model(images)
        _, predicted = torch.max(outputs.data, dim=1) # 将输出结果概率最大的作为预
            测值, 找到概率最大的下标, 输出最大值的索引位置
        total += labels.size(0)
        correct += (predicted == labels).sum().item() #正确率累加
    print('accuracy on test set: %d %% ' % (100 * correct / total))
    return correct / total

if __name__ == '__main__':
    model = LeNet().to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    epoch_list = []
    acc_list = []
    for epoch in range(10):
        train(epoch)
        acc = test()
        epoch_list.append(epoch)
        acc_list.append(acc)
    plt.plot(epoch_list, acc_list)
    plt.title("The Mnist on LeNet-5")
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.show()
    acc = np.array(acc_list).mean()
    print(f'Accuracy of the network on the 10000 test images: {acc} %')
    torch.save(model.state_dict(), 'LeNet.pth')
```