# 《当代人工智能》实验报告

报告题目：当代人工智能——图像分类及经典 CNN 实现

姓　　名：

学　　号：

完成日期：2022 年 5 月 1 日星期日

# 华东师范大学数据科学与工程学院实验报告

**课程名称**：当代人工智能　　**年级**：2019　　**上机实践成绩**：

**指导教师**：　　**姓名**：　　**学号**：

**上机实践名称**：Project 3 图像分类及经典 CNN 实现　　**上机实践日期**：4 月

# 当代人工智能——图像分类及经典 CNN 实现

## 一、 项目目的

了解和认识经典的 CNN 架构，掌握使用 PyTorch 搭建并训练 CNN 神经网络的方法，并且使用 CNN 解决图像分类的实际问题。

## 二、 项目任务

复现多种经典的 CNN 网络架构，并完成 MNIST 手写数字识别任务。

## 三、 项目要求

1、 使用多种 CNN 模型（至少四种）：

a) 必选的三种架构：LeNet[1], AlexNet[2], ResNet[3]。
b) 其它：自己挑选感兴趣的一个或多个架构实现。

2、 代码可执行，结果可复现，且必须使用 argparse 包进行模型和超参数的选择。

3、 报告必须包含的三点：

a) 代码实现时遇到了哪些 bug？如何解决的？
b) 不同模型的实验结果及分析。
c) 你认为影响这几个模型性能的主要原因是什么？如何得到这个结论的？

## 四、 实验环境

使用 Python 语言在 PyCharm 开发环境下进行代码的编写，使用 torchvision 进行 MNIST 数据集的加载，进行 CNN 网络的复现，并将其运用于手写数字识别任务。

**执行代码依赖的库：**

```
import argparse
import torch
from torch import nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
```

## 五、 实验结果

**本次实验共实现了五种经典的 CNN 网络，分别是：**

Lenet、AlexNet、ResNet、VGGNet、GoogLeNet；

**main.py 运行命令示例如下：**

- python main.py --model lenet --lr 0.0001 --dropout 0.0 --epoch 10

```
python main.py --model lenet --lr 0.0001 --dropout 0.0 --epoch 10
```

- python main.py --model alexnet --lr 0.0001 --dropout 0.0 --epoch 10

```
python main.py --model alexnet --lr 0.0001 --dropout 0.0 --epoch 10
```

- python main.py --model resnet --lr 0.0001 --dropout 0.0 --epoch 10

```
python main.py --model resnet --lr 0.0001 --dropout 0.0 --epoch 10
```

- python main.py --model vggnet --lr 0.0001 --dropout 0.0 --epoch 10

```
python main.py --model vggnet --lr 0.0001 --dropout 0.0 --epoch 10
```

- python main.py --model googlenet --lr 0.0001 --dropout 0.0 --epoch 10

```
python main.py --model googlenet --lr 0.0001 --dropout 0.0 --epoch 10
```

同时提供调参脚本 run.sh,运行 run.sh 将生成 log.txt 日志文件，记录每个模型的不同超参数组合的训练、验证结果，从而选择最优的超参数，并报告模型在测试集上的表现。

（一）代码实现时需要注意的问题

1、在论文中，LeNet 没有使用 dropout 进行正则化，但 AlexNet 运用了 dropout，如下图所示。

We use dropout in the first two fully connected layers of Figure 2. Without dropout, our network exhibits substantial overfitting. Dropout roughly doubles the number of iterations required to converge.

由于在 argparse 的运行命令中向所有模型都提供了 dropout 超参数的选择，因此为了统一，在所有实现的网络的全连接层都运用了 dropout 进行正则化，同时在进行超参选择的组合中增加 dropout 为 0.0 的项，此时相当于没有使用 dropout。

2、在本次实验对 MNIST 数据集的预处理中，使用 torchvison.transform 将图片维度转换为 32×32 的大小，如下图所示，但 Resize 并不能增加图像的通道数，因此，需要将论文中的输入通道数由 3 更改为 1。

```
size = (32, 32)
# if args.model == "AlexNet" or args.model == "alexnet":
#     size = (227, 227)
data_tf = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Resize(size),
     transforms.Normalize([0.5], [0.5])]
)
```

```
raise ValueError("If size is a sequence, it should have 1 or 2 values")
```

3、在 AlexNet 论文中，网络的输入维度为 224 × 224 × 3，因此出现了一个问题。在进行 kernel_size 为 11，stride 为 4 的卷积操作时，(224 -

11)/4 出现无法整除的问题。通过查找相关资料，发现有的做法是将 AlexNet 的输入维度由 224 更改为 227，使之能够进行整除。

4、MNIST 数据集仅提供训练集和测试集，因此需要自己手动从训练集中划分验证集进行调超参，而不能直接在测试集上进行调超参，这里将验证集的比例设置为 0.2，如下图所示。

```
# 将训练集划分一部分为验证集
n_valid = int(0.2 * len(train_set))
train_set_valid, train_set_train = random_split(dataset=train_set,
                                                lengths=[n_valid, len(train_set) - n_valid],
                                                generator=torch.Generator().manual_seed(0))
train_dataloader = DataLoader(dataset=train_set_train, batch_size=batch_size, shuffle=True)
valid_dataloader = DataLoader(dataset=train_set_valid, batch_size=batch_size, shuffle=True)
```

5、在本次实验中实现了 GoogLeNet，但 GoogLeNet 网络的输出较为特殊，它的输出包含多个输出向量，而其它网络仅有一个输出向量，因此需要对 GoogLeNet 的输出作特别处理，并设置好辅助输出损失的权重值。同时应当注意，GoogLeNet 在训练时需要辅助输出，它对 GoogLeNet 的训练结果具有正则化效果，但是在推理时不需要辅助输出。

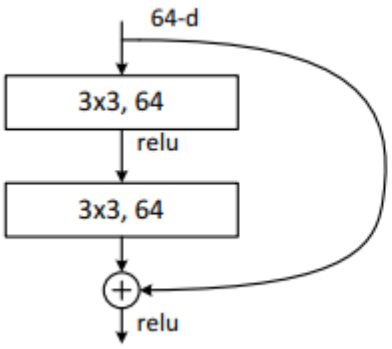
```
if IsGoogleNet:
    # 辅助分类器,训练时需要
    y_hat, y_hat1, y_hat2 = model(X)
    loss = loss_func(y_hat, y.long()).sum() + \
           0.3 * loss_func(y_hat1, y.long()).sum() + \
           0.3 * loss_func(y_hat2, y.long()).sum()
else:
    y_hat = model(X)
    loss = loss_func(y_hat, y.long()).sum()
```

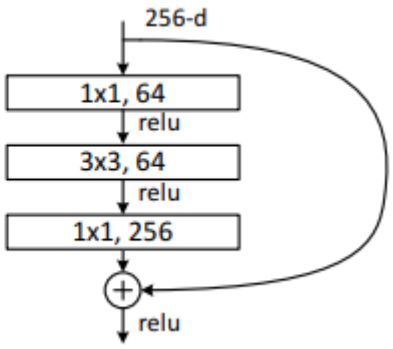
```
if IsGoogleNet:
    # 辅助分类器,推理时不时需要
    y_hat, y_hat1, y_hat2 = model(X)
else:
    y_hat = model(X)
loss = loss_func(y_hat, y.long()).sum()
```

6、CrossEntropyLoss()自带 softmax 函数，因此模型最后一层不需要

使用 softmax 函数，直接进行输出即可。

7、在 ResNet 残差网络实现中，当进行输入输出短接相加时，需要保证输入输出的通道数（feature map）相同，若不相同，需要进行处理使之相同，从而能够短接相加，如下图所示；




```
if in_channel != out_channel:
    self.shortcut_layer = nn.Sequential(
        nn.Conv2d(in_channel, out_channel, kernel_size=(1, 1), stride=(1, 1)),
    )
else:
    self.shortcut_layer = nn.Sequential()
```

（二）模型的实验结果及分析

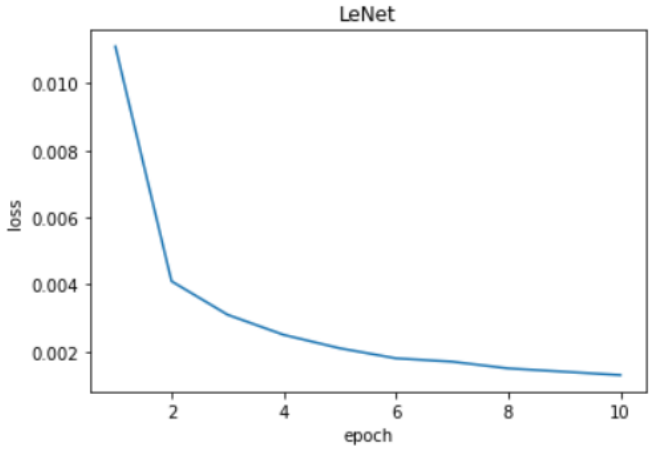
由于 MNIST 数据集只提供了训练集和测试集，因此将训练集划分一部分为验证集进行调超参，训练集与验证集的划分比例 4:1。同时编写调参脚本 run.sh，使用 argparse 包进行模型和超参数的选择。在 run.sh 脚本中测试了多种不同超参数的组合，然后把结果写到生成的 log.txt 日志文件中，记录每个模型的不同超参数组合的训练、验证结果，从而选择最合适的超参数，使模型在验证集上的效果最优，最后报告模型在测试集上的表现。日志示例如下方所示，在日志中报告了模型的训练超参数、训练过程中的损失和训练集准确率，用于调参的验证集准确率，以及最后报告的测试集准确率。

一次训练的日志示例如下：

```
beginning----------------------------2022-05-02 15:58:14----------------------------
model: lenet
epochs: 10
learning rate: 0.001
dropout: 0.2
training......
epoch 1, loss 0.0127, train acc 0.935
epoch 2, loss 0.0045, train acc 0.978
epoch 3, loss 0.0036, train acc 0.983
epoch 4, loss 0.0029, train acc 0.986
epoch 5, loss 0.0024, train acc 0.989
epoch 6, loss 0.0022, train acc 0.989
epoch 7, loss 0.0019, train acc 0.991
epoch 8, loss 0.0018, train acc 0.991
epoch 9, loss 0.0016, train acc 0.993
epoch 10, loss 0.0015, train acc 0.993
validating......
model lenet, epochs 10, lr 0.0010, dropout 0.20, valid loss 0.0039, valid acc 0.986
testing......
model lenet, epochs 10, lr 0.0010, dropout 0.20, test loss 0.0030, test acc 0.987
ending-----------------------------2022-05-02 16:01:43----------------------------
```

最终选择的最优超参数组合以及模型在测试集上的表现如下方所示：

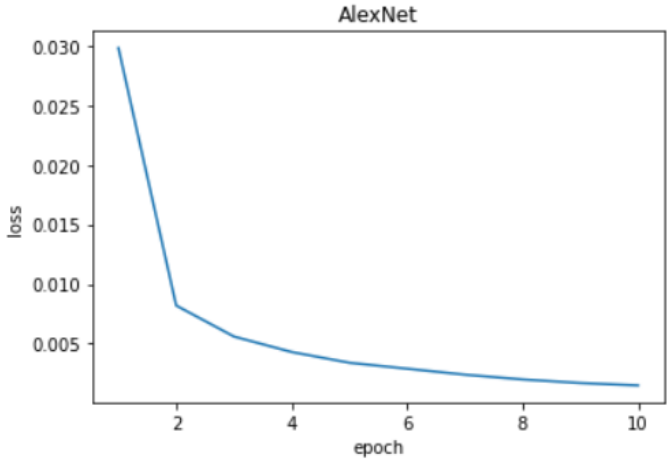
1、 LeNet：




超参数：--lr 0.001 --dropout 0.1 --epoch 10

模型在测试集上的表现，准确率：0.988

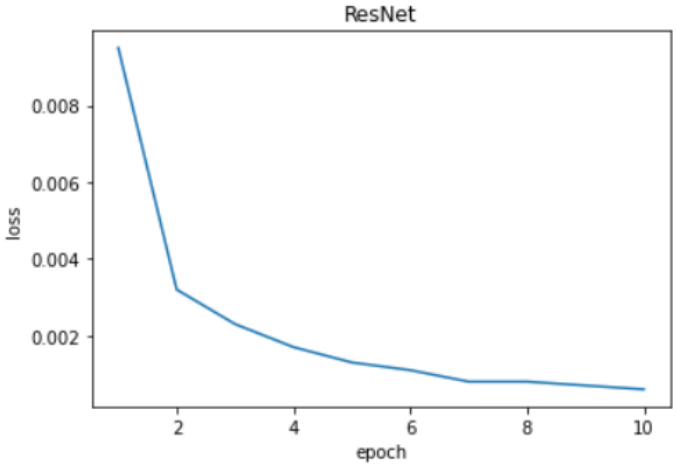
2、 AlexNet：




超参数：--lr 0.0001 --dropout 0.0 --epoch 10

模型在测试集上的表现，准确率：0.989

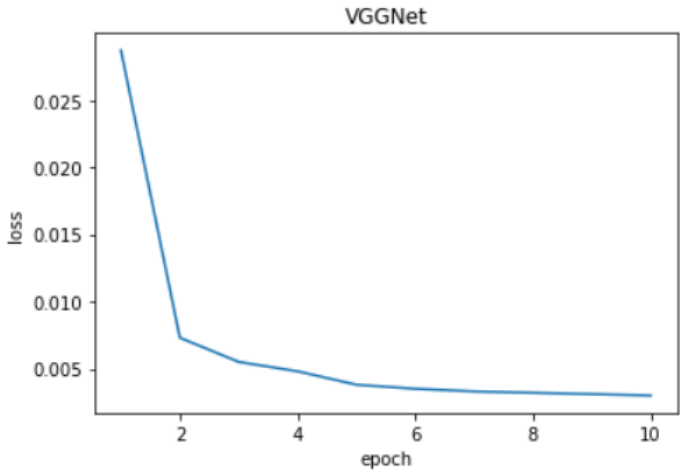
3、 ResNet：




超参数：--lr 0.0001 --dropout 0.2 --epoch 10

模型在测试集上的表现，准确率：0.992

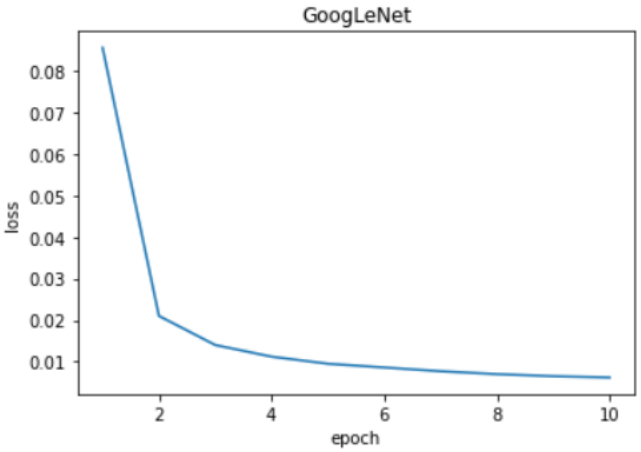
4、 VggNet：




超参数：--lr 0.001 --dropout 0.0 --epoch 10

模型在测试集上的表现，准确率：0.989

5、 GoogLeNet：




超参数：--lr 0.001 --dropout 0.0 --epoch 10

模型在测试集上的表现，准确率：0.986

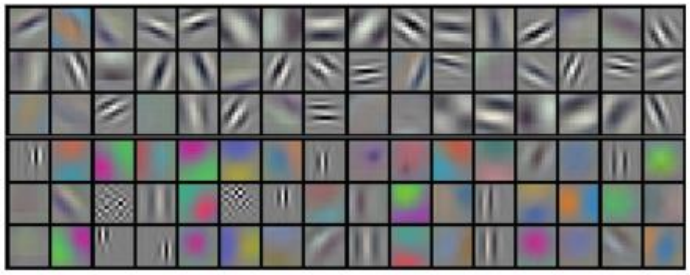
在以上实验结果中，我们可以看出，对于 MNIST 数据集的图像分类任务，Lenet、AlexNet、ResNet、VGGNet、GoogLeNet 网络都能达到比较理想的效果。由于 MNIST 数据集较为简单，分类类别也较少，因此拟合的难度不高，即使是较为浅层的 Lenet 也能有比较好的效果。实验中的模型在测试集上的准确率都能达到 98%以上，其中网络较深，拟合能力较强的 ResNet 甚至能达到 99%的水平。但同时我们也能看到，对于

ResNet 来说，需要考虑使用 dropout 增强模型的泛化能力，来使 ResNet 在验证集上的准确率达到最高。ResNet 若不考虑正则化，可能会存在一定程度的过拟合现象。

（三）影响模型性能的主要原因

第一，**网络宽度**。宽度，对于卷积神经网络来说，就是通道(channel)的数量。下图为 AlexNet 论文中，96 个卷积核所学习到的特征的 feature map，从图中我们可以看出，不同卷积核所提取的特征有所不同，有的 feature map 提取到纹理特征，有的 feature map 提取到颜色特征。因此，宽度将会影响模型的性能，足够的宽度将会支持模型学习到不同方面的特征，使得模型的学习能力更强，学习到的特征更丰富，从而提高模型的性能。

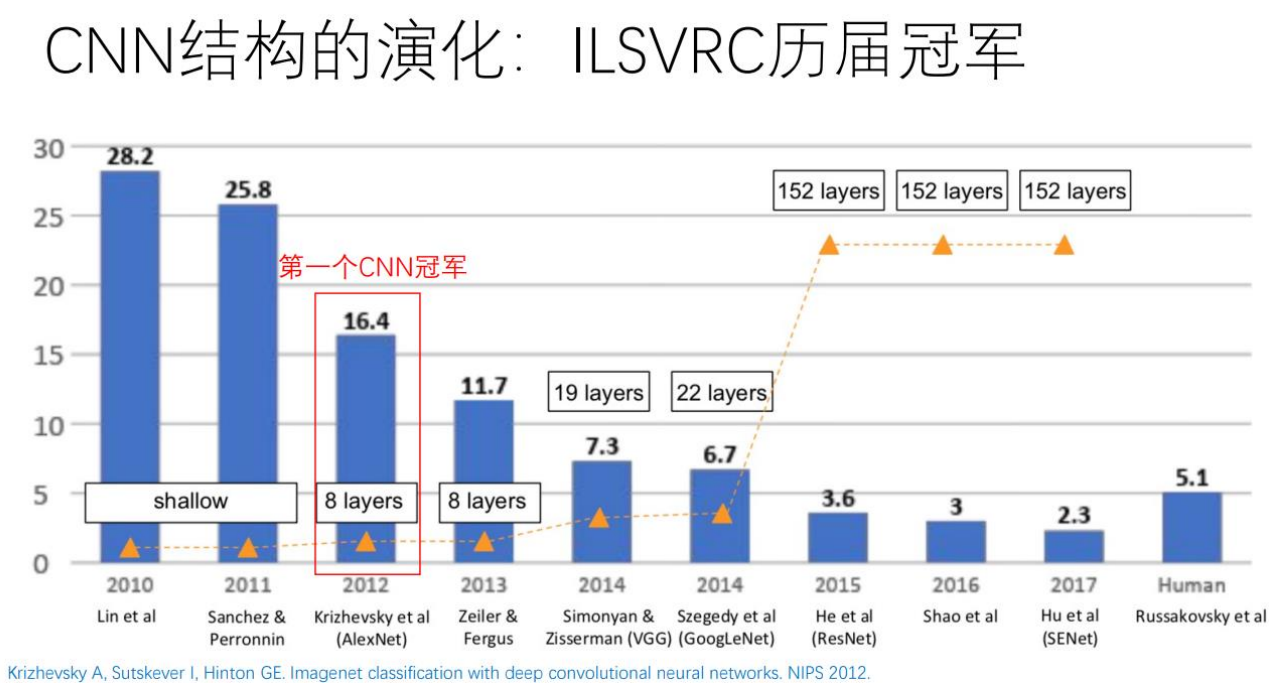
**Figure 3. Ninety-six convolutional kernels of size 11 × 11 × 3 learned by the first convolutional layer on the 224 × 224 × 3 input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2 (see Section 7.1 for details).**



第二，**网络深度**。一般来说，网络越深，性能越好。从理论上说，网络每一层的激活函数都将会产生一个非线性变化，随着网络深度的增加，模型的非线性表达能力，将会增强，有能力学习到更加复杂的变换，从而可以拟合更加复杂的特征输入。

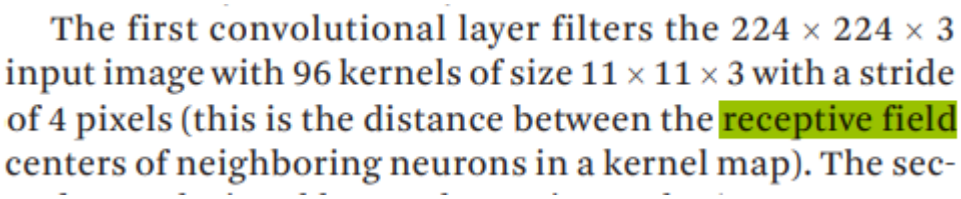
从下图 CNN 结构的演化过程我们可以明显看出，随着网络深度的不断增加，模型的错误率逐渐减小到一个比较小的值附近。从

AlexNet 到 VGG、GoogLeNet，再到 ResNet，模型深度不断增加，模型性能也相应地提升。本实验中，由于 mnist 数据集相对简单，即使是较为浅层的 LeNet 也能较好地学习到图像的特征，在 ImgeNet 等较为复杂的数据集上，深度对模型的性能影响将会更加明显。




Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. NIPS 2012.

第三，**卷积核大小**。从 AlexNet 论文中我们可以看出，AlexNet 第一层中用到了 11×11 卷积核。


The first convolutional layer filters the 224 × 224 × 3 input image with 96 kernels of size 11 × 11 × 3 with a stride of 4 pixels (this is the distance between the receptive field centers of neighboring neurons in a kernel map). The sec-

一般来说，卷积核越大，receptive field 越大，因此学习到的特征相应越多。但是大的卷积核会导致参数的增加，从而导致训练的计算量增加，不利于较深模型的训练。因此，在权衡利弊之后，一般多采用 3×3 这样较小的卷积核用来提取特征，使模型变得更加容易训练。

其它一些因素，例如**训练数据集、优化方法、正则化方法、卷积步长、padding** 等也会对模型性能产生影响。卷积核的步长代表了特征提取的精度，而 padding 将有助于学习到边缘的特征。优化方法和正则化方法将影响模型的训练效果，同时防止模型过拟合。数据集也是较为重要的，例如 AlexNet 论文中使用了数据增强的方法来减少模型的过拟合。

**5.1. Data augmentation**

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations (e.g., Refs.[4, 5, 30]). We employ two distinct forms of data augmentation, both of which allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. In our implementation, the transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free.

## 六、 模型实现

**注：**由于 MNIST 数据集图片维度为 28×28，因此在训练不同模型时，使用 torchvison.transform 将输入图片维度修改为为论文中的输入维度，或者为方便输入输出和减少训练时间，对卷积核或池化层大小有所更改，模型实现与论文中的描述部分不一致。

**声明：**kernel_ num：卷积核数量；

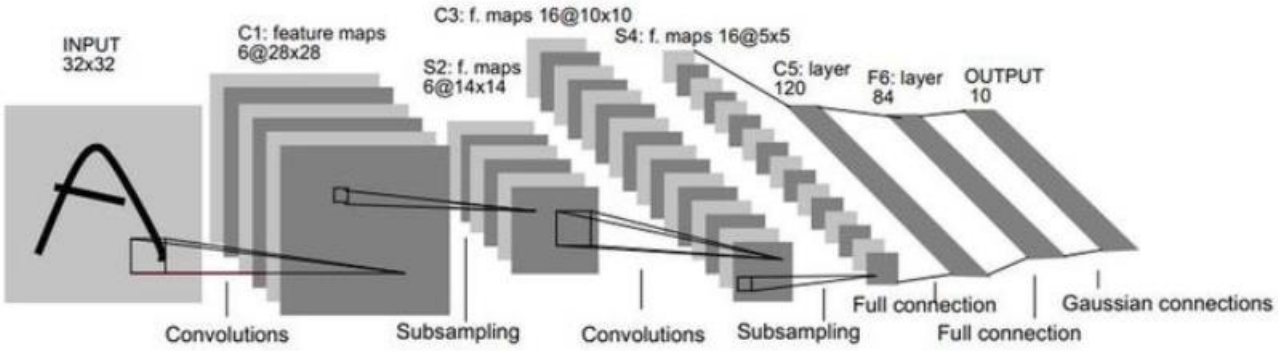
kernel_size：卷积核或池化大小；

stride：卷积或池化步长；

Padding：Padding 数；

### （一）LeNet

本次实验实现的 LeNet 模型如下图所示：




第 C1 层：卷积层。

kernel_ size ： 5， stride：1，kernel_ num ： 6；

输入形状：1@32×32，输出形状：6@28×28；

第 S2 层：池化层。

kernel_ size ： 2，stride：2；

输入形状：6@28×28, 输出形状：6@14×14；

第 C3 层：卷积层。

kernel_ size ： 5，stride：1，kernel_ num ： 16；

输入形状：6@14×14, 输出形状：16@10×10；

第 S4 层：池化层。

kernel_ size ： 2， stride：2；

输入形状：16@10×10, 输出形状：16@5×5；

第 C5 层：卷积层。

kernel_ size ： 5， stride：1，kernel_ num ： 120；

输入形状：16@5×5, 输出形状：120@1×1；

第 F6 层：全连接层。

输入维度：120, 输出维度：84；

第 7 层（OUTPUT）：全连接层。

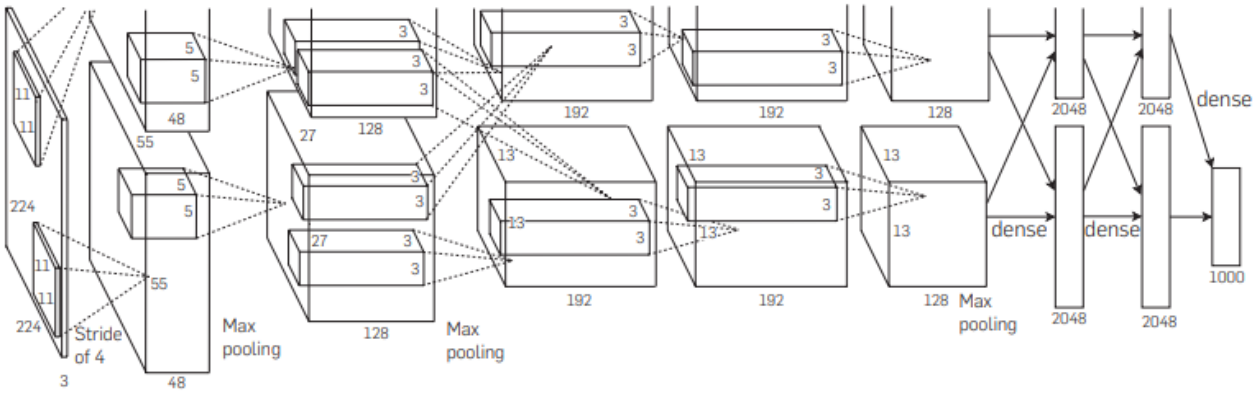
输入维度：84, 输出维度：10；

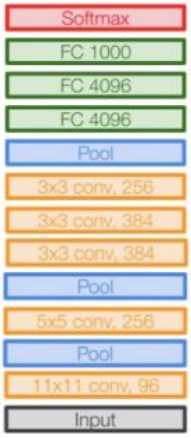
## （二）AlexNet

本次实验实现的 AlexNet 模型包含两个版本：

第一个版本与论文保存一致，如下图所示；

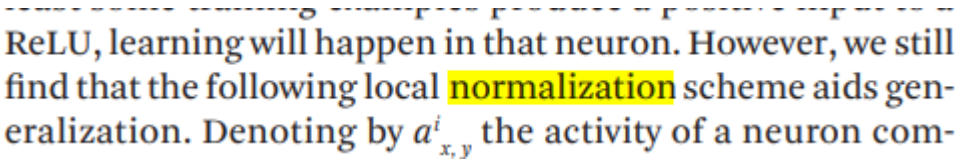
第二个版本更改了卷积核、池化层的大小，方便了输入输出，加快了训练速度。




AlexNet

同时，需注意 AlexNet 使用的正则化方法包括 local normalization；



ReLU, learning will happen in that neuron. However, we still find that the following local normalization scheme aids generalization. Denoting by $a^i_{x,y}$ the activity of a neuron com-

实现的 AlexNet 网络结构如下所示：

第 1 层：卷积池化层。

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 12；

输入形状：1@32×32，输出形状：12@32×32；

最大池化 kernel_ size ： 2， stride：2，padding：0；

输入形状：12@32×32，输出形状：12@16×16；

第 2 层：卷积池化层。

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 24；

输入形状：12@16×16，输出形状：24@16×16；

最大池化 kernel_ size ： 2， stride：2，padding：0；

输入形状：24@16×16，输出形状：24@8×8；

第 3 层：卷积层。

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 32；

输入形状：24@8×8，输出形状：32@8×8；

第 4 层：卷积层。kernel_ size ： ， stride：；

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 48；

输入形状：32@8×8，输出形状：48@8×8；

第 5 层：卷积池化层。

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 64；

输入形状：48@8×8，输出形状：64@8×8；

最大池化 kernel_ size ： 2， stride：2，padding：0；

输入形状：64@8×8，输出形状：64@4×4；

第 6 层：全连接层。

输入维度：1024，输出维度：512；

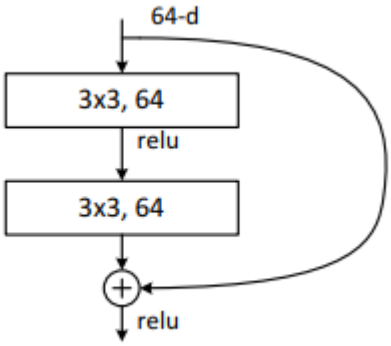
第 7 层：全连接层。

输入维度：512，输出维度：128；

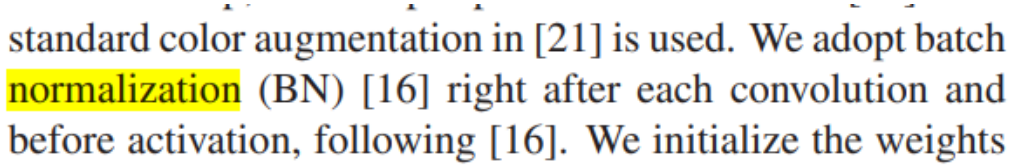
第 8 层：全连接层。

输入维度：128，输出维度：10；

## （四）ResNet



注：ResNet 模型由多个 Residual block 组成，本实验实现了一个 34-layer 的 ResNet 模型，包含 16 个 Residual block。为方便输入输出，将每个 Residual block 修改成为相同的结构。同时，需注意 ResNet 使用的正则化方法包括 batch normalization；


standard color augmentation in [21] is used. We adopt batch normalization (BN) [16] right after each convolution and before activation, following [16]. We initialize the weights

实现的 ResNet 网络结构如下所示：

在 Residual block 前包含一个卷积层：

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 8；

输入形状：1@32×32，输出形状：8@32×32；

每个 Residual block 包含 2 个卷积层，结构如下：

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 8；

输入形状：8@32×32，输出形状：8@32×32；

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 8；

输入形状：8@32×32，输出形状：8@32×32；

经过多个 Residual block 后输出：

平均池化 kernel_ size ： 2， stride：2，padding：0；

输入形状：8@32×32，输出形状：8@16×16；

全连接层；

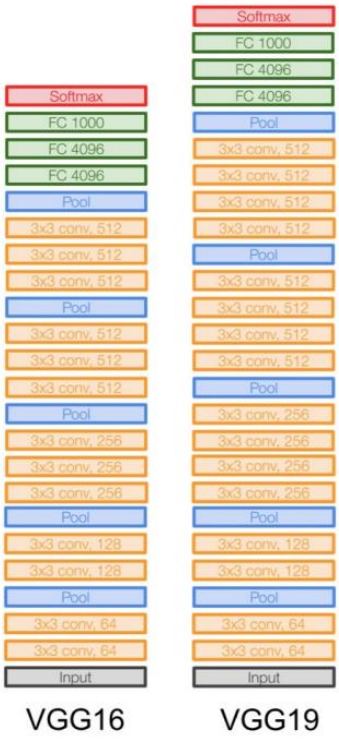
输入维度：8*16*16，输出维度：1000；

全连接层；

输入维度：1000，输出维度：10；

## （五）VGGNet

本次实验实现的 LeNet 模型如下图所示（VGG16）：




注：在模型的实现中将多个卷积层和一个池化层封装成一个 VGGNetBlock

第 1 个 Block：2 个卷积层和 1 个池化层。

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 32；

输入形状：1@32×32，输出形状：32@32×32；

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 32；

输入形状：32@32×32，输出形状：32@32×32；

最大池化 kernel_ size ： ， stride：，padding：；

输入形状：32@32×32，输出形状：32@16×16；

第 2 个 Block：与第 1 个 Block 结构相同。

第 3 个 Block：3 个卷积层和 1 个池化层。

输入形状：32@8×8，输出形状：32@8×8；

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 32；

输入形状：32@8×8，输出形状：32@8×8；

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 32；

输入形状：32@8×8，输出形状：32@8×8；

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 32；

输入形状：32@8×8，输出形状：32@8×8；

最大池化 kernel_ size ： ， stride：，padding：；

输入形状：32@8×8，输出形状：32@4×4；

第 4、5 个 Block：与第 3 个 Block 结构相同。

第 6 层：全连接层。

输入维度：32，输出维度：32；

第 7 层：全连接层。

输入维度：32，输出维度：32；

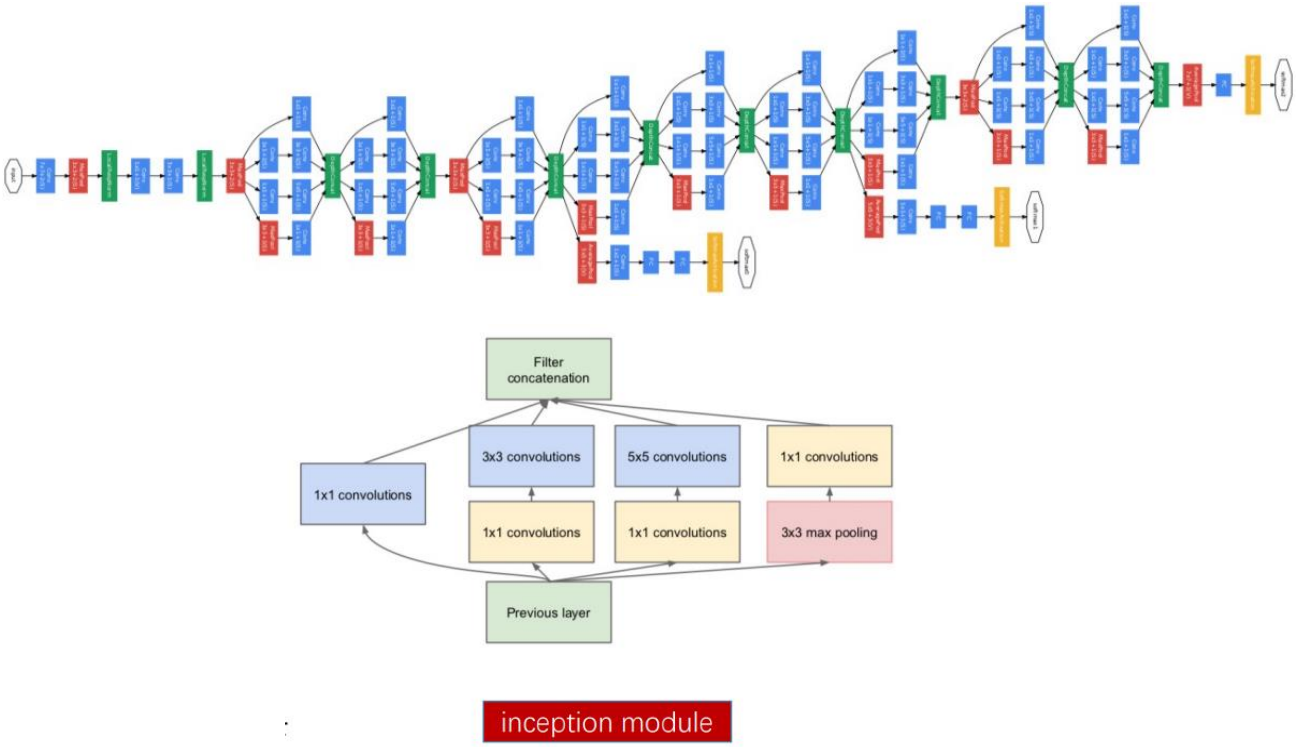
第 8 层：全连接层。

输入维度：32，输出维度：10；

## （六）GoogLeNet

本次实验实现的 GoogLeNet 模型如下图所示：

GoogLeNet 网络包含 9 个 module 模块；




Inception module 之前包含卷积和池化层：

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 16；

输入形状：1@32×32，输出形状：16@32×32；

最大池化 kernel_ size ： 2， stride：2，padding：0；

输入形状：16@32×32，输出形状：16@16×16；

卷积 kernel_ size ： 1， stride：1，padding：0，kernel_ num ： 16；

输入形状：16@16×16，输出形状：16@16×16；

卷积 kernel_ size ： 3， stride：1，padding：1，kernel_ num ： 16；

输入形状：16@16×16，输出形状：16@16×16；

池化 kernel_ size ： 2， stride：2，padding：0；

输入形状：16@16×16，输出形状：16@8×8；

每个 module 结构如下（每个 module 包含 4 个分支）：

```
def forward(self, x):
    out1 = self.channel1(x)
    out2 = self.channel2(x)
    out3 = self.channel3(x)
    out4 = self.channel4(x)
    out = [out1, out2, out3, out4]
    return torch.cat(out, 1)
```

```
self.channel1 = nn.Sequential(
    nn.Conv2d(in_channels, out_channel1, kernel_size=(1, 1), stride=(1, 1), padding=0),
    nn.ReLU(),
)
self.channel2 = nn.Sequential(
    nn.Conv2d(in_channels, out_channel21, kernel_size=(1, 1), padding=0),
    nn.ReLU(),
    nn.Conv2d(out_channel21, out_channel22, kernel_size=(3, 3), padding=1),
    nn.ReLU(),
)
self.channel3 = nn.Sequential(
    nn.Conv2d(in_channels, out_channel31, kernel_size=(1, 1), padding=0),
    nn.ReLU(),
    nn.Conv2d(out_channel31, out_channel32, kernel_size=(5, 5), padding=2),
    nn.ReLU(),
)
self.channel4 = nn.Sequential(
    nn.MaxPool2d(kernel_size=3, stride=1, padding=1),
    nn.Conv2d(in_channels, out_channel4, kernel_size=(1, 1)),
    nn.ReLU(),
)
```

为方便输入输出，第 1—9 个 module 结构相同。

在第 4、7、9 个 module 处皆包含一个输出模块：

第 1、2 个输出模块(辅助)：分别包含 2 个卷积层和 1 个池化层。

平均池化 kernel_ size ： 2， stride：2，padding：0；

输入形状：16@4×4；，输出形状：16@2×2；；

卷积 kernel_ size ： 3， stride：，padding：1，kernel_ num ： 16；

输入形状：16@2×2；，输出形状：16@2×2；

全连接；

输入维度：64, 输出维度：64；

全连接；

输入维度：64, 输出维度：10；

最终输出模块：包含 1 个池化层和 1 个全连接层。

平均池化 kernel_ size ： 2， stride：2，padding：0；

输入形状：16@2×2，输出形状：16@1×1；

全连接；

输入维度：16, 输出维度：10；

## 七、 实验总结

在本次实验中，实现了 Lenet、AlexNet、ResNet、VGGNet、GoogLeNet 共五种经典的 CNN 网络。在实现的过程中，我熟悉了使用 PyTorch 搭建并训练 CNN 神经网络的方法，对于卷积神经网络的基本结构已经有了基本的了解和理解，并且使用了多种 CNN 解决了图像分类的实际问题。同时，学会了使用编写脚本的方法进行调超参数。

## 八、 源代码

源代码附在文件内；

Main.py：实现模型的训练、验证、测试等；

MyLeNet.py：实现 LeNet 模型；

MyAlexNet.py：实现 AlexNet 模型；

MyResNet.py：实现 ResNet 模型；

MyVGGNet.py：实现 VGGNet 模型；

My:GoogLeNet.py：实现 GoogLeNet 模型；

Run.py:调参脚本;

Log.txt：调参脚本生成的调参日志;