当代人工智能实验三--图像分类及经典 CNN 实现

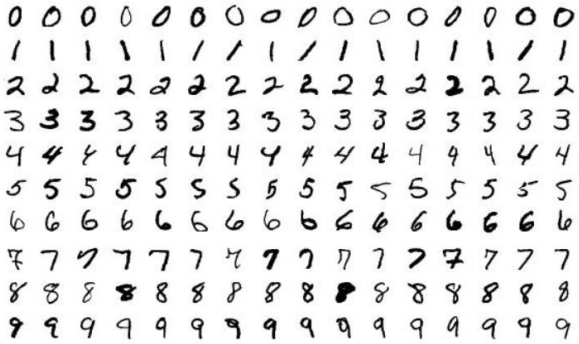
--10215501435 杨茜雅

实验任务

对于 MNIST 手写数字数据集 http://yann.lecun.com/exdb/mnist/，通过使用搭建多种神经网络模型，对于数据集训练一个模型，然后对于测试集的图像数据进行预测，并且和给定的标签进行对比，给出预测的准确率。**本次实验中，我除了实现必选的三种架构: LeNet，AlexNet，ResNet 之外，还实现了 VGGNet, MobileNet 和 GoogLeNet 一共六个架构。并运用权重衰减和提前停止对模型进行优化。**

数据集

该数据集一共包括 70000 张图像，其中训练集包含 60000 张图像，测试集包含 10000 张图像。MNIST 数据集中的图像为单通道黑白图像，图像尺寸为 28x28，其中手写数字被规范到了 20 x20 的范围内，是一个非常优质的适合模式识别的数据集。



由于数据集已经规范化，所以我们不需要对数据进行额外的预处理。只需要将数据进行加载，并且将训练集划分，验证之后再在测试集上进行预测。剩下的工作就是搭建神经网络了。

划分数据集

我们需要将原始的数据集划分为训练集和验证集。Pytorch 里面的 torchvision 已经帮我们实现了 MNIST 方法用于加载 MNIST 数据集，并且可以对其进行张量化和标准化等操作。

```
# Loading the dataset and preprocessing
train_dataset = torchvision.datasets.MNIST(root='./data',
                                           train=True,
                                           transform=transforms.Compose([
                                               # transforms.Resize((32, 32)),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=(0.1307,), std=(0.3081,))]),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          transform=transforms.Compose([
                                              # transforms.Resize((32, 32)),
                                              transforms.ToTensor(),
                                              transforms.Normalize(mean=(0.1325,), std=(0.3105,))]),
                                          download=True)
```

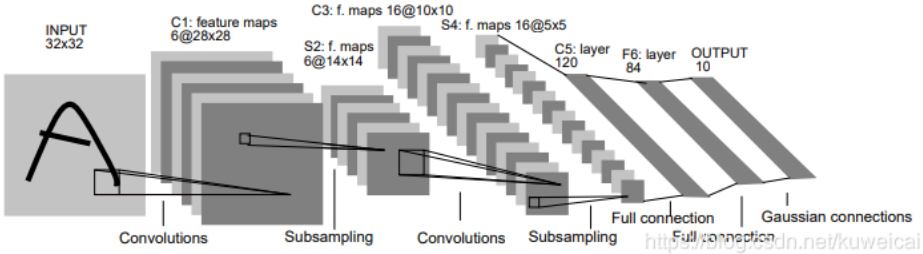
在加载完数据集之后，我们可以使用 torch.utils.data.random_split 方法将数据集按照 **80:20** 比例划分为训练集与验证集。

```
# divide the dataset
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(train_dataset, [train_size, val_size])
```

搭建网络

1、LeNet

LeNet 是卷积神经网络的开山之作，也是将深度学习推向繁荣的一座里程碑。
Yann LeCun 于上世纪 90 年代提出了 LeNet，他首次采用了卷积层、池化层这两个全新的神经网络组件；LeNet 在手写字符识别任务上取得了瞩目的准确率。
LeNet 网络有一系列的版本，其中以 LeNet-5 版本最为著名，也是 LeNet 系列中效果最佳的版本。
LeNet-5 使用 5 个卷积层来学习图像特征；卷积层的**权重共享**特点使得它相较于全连接层，节省了相当多的计算量与内存空间；同时卷积层的**局部连接**特点可以保证图像的**空间相关性**。
以下是 LeNet 的网络结构示意图：



由于我们的图片输入的原始尺寸是 28*28，所以我们需要先上采样到 32*32 的大小，通过

```
x = F.interpolate(x, size=32, mode='bilinear', align_corners=False)
```

方法。

搭建完成后，可以使用 torchinfo.summary 方法打印出网络的结构，以及参数的数量。发现 LeNet 的参数数量只有 61750，比全连接的 FNN 在图像识别上的参数规模优化了许多。

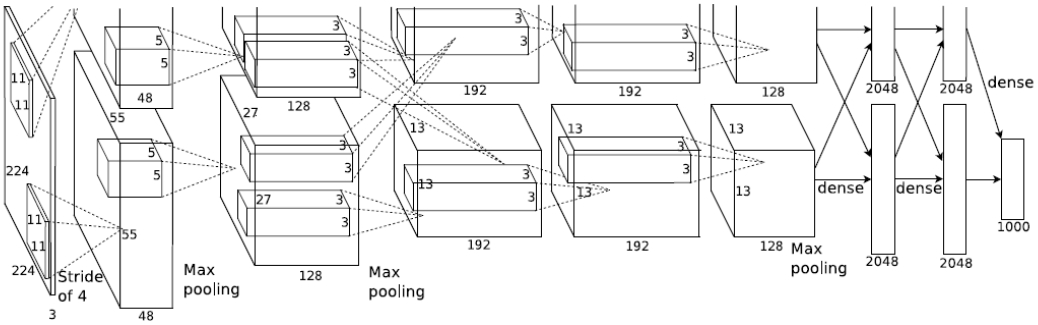
LeNet 模型的结构和参数

```
LeNet                                    [64, 10]                  --
├─Sequential: 1-1                        [64, 6, 14, 14]           --
│    └─Conv2d: 2-1                       [64, 6, 28, 28]           156
│    └─BatchNorm2d: 2-2                  [64, 6, 28, 28]           12
│    └─ReLU: 2-3                         [64, 6, 28, 28]           --
│    └─MaxPool2d: 2-4                    [64, 6, 14, 14]           --
├─Sequential: 1-2                        [64, 16, 5, 5]            --
│    └─Conv2d: 2-5                       [64, 16, 10, 10]          2,416
│    └─BatchNorm2d: 2-6                  [64, 16, 10, 10]          32
│    └─ReLU: 2-7                         [64, 16, 10, 10]          --
│    └─MaxPool2d: 2-8                    [64, 16, 5, 5]            --
├─Linear: 1-3                            [64, 120]                 48,120
├─ReLU: 1-4                              [64, 120]                 --
├─Dropout: 1-5                           [64, 120]                 --
├─Linear: 1-6                            [64, 84]                  10,164
├─ReLU: 1-7                              [64, 84]                  --
├─Dropout: 1-8                           [64, 84]                  --
├─Linear: 1-9                            [64, 10]                  850
==========================================================================================
Total params: 61,750
Trainable params: 61,750
Non-trainable params: 0
Total mult-adds (M): 27.08
```

2、AlexNet

第一个典型的 CNN 是 LeNet5 网络，而第一个大放异彩的 CNN 却是 AlexNet。2012 年在全球知名的图像识别竞赛 ILSVRC 中，AlexNet 横空出世，直接将错误率降低了近 10 个百分点。

以下是 AlexNet 的网络结构：



AlexNet 整体的网络结构包括：1 个输入层（input layer）、5 个卷积层（C1、C2、C3、C4、C5）、2 个全连接层（FC6、FC7）和 1 个输出层（output layer）。在将图片通过 8 倍上采样到 224*224 的尺寸以后，我们将图片输入到网络中。

搭建完网络后，我们发现它的参数是 4000 多万个，是 LeNet 的 700 倍左右。如此庞大的参数量，在当时的 GPU 水平下是一个不小的挑战。更深层次的网络，也意味着更加多的特征提取与更高的学习准确率。

AlexNet 模型的结构和参数

```
AlexNet                                  [64, 10]                  --
├─Sequential: 1-1                       [64, 10]                  --
│    └─Upsample: 2-1                    [64, 1, 224, 224]         --
│    └─Conv2d: 2-2                      [64, 96, 54, 54]          11,712
│    └─ReLU: 2-3                        [64, 96, 54, 54]          --
│    └─MaxPool2d: 2-4                   [64, 96, 26, 26]          --
│    └─Conv2d: 2-5                      [64, 256, 26, 26]         614,656
│    └─ReLU: 2-6                        [64, 256, 26, 26]         --
│    └─MaxPool2d: 2-7                   [64, 256, 12, 12]         --
│    └─Conv2d: 2-8                      [64, 384, 12, 12]         885,120
│    └─ReLU: 2-9                        [64, 384, 12, 12]         --
│    └─Conv2d: 2-10                     [64, 384, 12, 12]         1,327,488
│    └─ReLU: 2-11                       [64, 384, 12, 12]         --
│    └─Conv2d: 2-12                     [64, 256, 12, 12]         884,992
│    └─ReLU: 2-13                       [64, 256, 12, 12]         --
│    └─MaxPool2d: 2-14                  [64, 256, 5, 5]           --
│    └─Flatten: 2-15                    [64, 6400]                --
│    └─Linear: 2-16                     [64, 4096]                26,218,496
│    └─ReLU: 2-17                       [64, 4096]                --
│    └─Dropout: 2-18                    [64, 4096]                --
│    └─Linear: 2-19                     [64, 4096]                16,781,312
│    └─ReLU: 2-20                       [64, 4096]                --
│    └─Dropout: 2-21                    [64, 4096]                --
│    └─Linear: 2-22                     [64, 10]                  40,970
==========================================================================================
Total params: 46,764,746
Trainable params: 46,764,746
Non-trainable params: 0
Total mult-adds (G): 60.08
==========================================================================================
Input size (MB): 0.20
Forward/backward pass size (MB): 311.63
Params size (MB): 187.06
Estimated Total Size (MB): 498.89
```

3、VGGNet11

2014 年，牛津大学计算机视觉组 和 Google DeepMind 公司一起研发了新的卷积神经网络，并命名为 VGGNet。VGGNet 是比 AlexNet 更深的深度卷积神经网络，该模型获得了 2014 年 ILSVRC 竞赛的第二名。

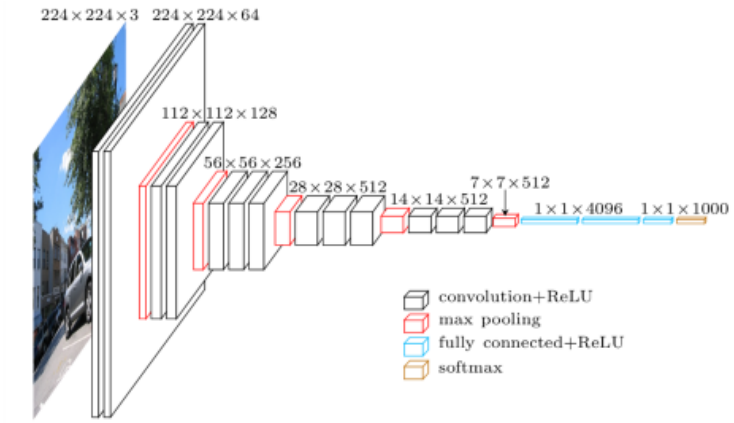
VGG 的结构与 AlexNet 类似，区别是深度更深，但形式上更加简单。它们的一个主要的区别是，VGG 没有采用 AlexNet 中比较大的卷积核尺寸（如 7x7），而是通过降低卷积核的大小（3x3），增加卷积子层数。这样做可以提取到更多的图片中的特征信息，提升分类效果。VGG 由 5 层卷积层、3 层全连接层、1 层 softmax 输出层构成，层与层之间使用 maxpool（最大化池）分开，所有隐藏层的激活单元都采用 ReLU 函数。作者在原论文中，根据卷积层不同的子层数量，设计了 A、A-LRN、B、C、D、E 这 6 种网络结构。

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

我准备先搭建 VGG 11 然后依样画葫芦搭建 VGG 19，最终比较这两个网络对于 MNIST 的预测准确度是否有区别。

两种网络结构相似，都是由 5 层卷积层、3 层全连接层组成，区别在于每个卷积层的子层数量不同。它们的最后都是四个全连接层，神经元数分别是 4096, 4096, 1000, 10。

下面是一个 VGGNet11 更加直观清晰的示意图：



VGGNet 中，核心的组件是一个 VGG 块，它的深度是可变的。由于 VGG 块的基本结构是固定的，因此我们只需要指定卷积层的数量以及输入输出通道数即可创建 VGG 块。对于每个 VGG 块的第一个卷积层，我们将输入通道数设置为与输出通道数相同；对于后面的卷积

层，我们只需要保持输出通道数与输入通道数相同即可。

```python
class VGGBlock(nn.Module):
    def __init__(self, conv_num, in_channels, out_channels):
        super(VGGBlock, self).__init__()
        self.net = nn.Sequential()
        for i in range(conv_num):
            self.net.add_module(
                "conv_{0}".format(i), nn.Sequential(
                    nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=3, padding=1),
                    nn.ReLU()
                )
            )
            in_channels = out_channels
        self.net.add_module("pool", nn.MaxPool2d(kernel_size=2, stride=2))
```

搭建完成之后，可以打印出网络的 summary。可以看到这个网络的参数是亿级别的，是非常重的一个网络。

VGGNet11 模型的结构和参数

```
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
VGGNet11                                 [64, 10]                  --
├─Sequential: 1-1                        [64, 10]                  --
│    └─Upsample: 2-1                     [64, 1, 224, 224]         --
│    └─VGGBlock: 2-2                     [64, 64, 112, 112]        --
│    │    └─Sequential: 3-1              [64, 64, 112, 112]        640
│    └─VGGBlock: 2-3                     [64, 128, 56, 56]         --
│    │    └─Sequential: 3-2              [64, 128, 56, 56]         73,856
│    └─VGGBlock: 2-4                     [64, 256, 28, 28]         --
│    │    └─Sequential: 3-3              [64, 256, 28, 28]         885,248
│    └─VGGBlock: 2-5                     [64, 512, 14, 14]         --
│    │    └─Sequential: 3-4              [64, 512, 14, 14]         3,539,968
│    └─VGGBlock: 2-6                     [64, 512, 7, 7]           --
│    │    └─Sequential: 3-5              [64, 512, 7, 7]           4,719,616
│    └─Flatten: 2-7                      [64, 25088]               --
│    └─Linear: 2-8                       [64, 4096]                102,764,544
│    └─ReLU: 2-9                         [64, 4096]                --
│    └─Dropout: 2-10                     [64, 4096]                --
│    └─Linear: 2-11                      [64, 4096]                16,781,312
│    └─ReLU: 2-12                        [64, 4096]                --
│    └─Dropout: 2-13                     [64, 4096]                --
│    └─Linear: 2-14                      [64, 10]                  40,970
==========================================================================================
Total params: 128,806,154
Trainable params: 128,806,154
Non-trainable params: 0
Total mult-adds (G): 483.50
==========================================================================================
Input size (MB): 0.20
Forward/backward pass size (MB): 3806.34
Params size (MB): 515.22
Estimated Total Size (MB): 4321.76
```

4、VGGNet19

只需要修改一下 conv_num 就可以搭建出来一个 VGGNet19 网络。可以看到这个网络的参数数量也是亿级别的，但是由于卷积层数非常多，增加了多个卷积核大小为 3， 通道数为 256 和 512 的卷积核，所以它的 total mult-add 数量也达到了 11 版本的 3 倍左右。

VGGNet19 模型的结构和参数

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
VGGNet19                                 [64, 10]                  --
├─Sequential: 1-1                        [64, 10]                  --
│    └─Upsample: 2-1                     [64, 1, 224, 224]         --
│    └─VGGBlock: 2-2                     [64, 64, 112, 112]        --
│    │    └─Sequential: 3-1              [64, 64, 112, 112]        37,568
│    └─VGGBlock: 2-3                     [64, 128, 56, 56]         --
│    │    └─Sequential: 3-2              [64, 128, 56, 56]         221,440
│    └─VGGBlock: 2-4                     [64, 256, 28, 28]         --
│    │    └─Sequential: 3-3              [64, 256, 28, 28]         2,065,408
│    └─VGGBlock: 2-5                     [64, 512, 14, 14]         --
│    │    └─Sequential: 3-4              [64, 512, 14, 14]         8,259,584
│    └─VGGBlock: 2-6                     [64, 512, 7, 7]           --
│    │    └─Sequential: 3-5              [64, 512, 7, 7]           9,439,232
│    └─Flatten: 2-7                      [64, 25088]               --
│    └─Linear: 2-8                       [64, 4096]                102,764,544
│    └─ReLU: 2-9                         [64, 4096]                --
│    └─Dropout: 2-10                     [64, 4096]                --
│    └─Linear: 2-11                      [64, 4096]                16,781,312
│    └─ReLU: 2-12                        [64, 4096]                --
│    └─Dropout: 2-13                     [64, 4096]                --
│    └─Linear: 2-14                      [64, 10]                  40,970
==========================================================================================
Total params: 139,610,058
Trainable params: 139,610,058
Non-trainable params: 0
Total mult-adds (T): 1.25
==========================================================================================
Input size (MB): 0.20
Forward/backward pass size (MB): 7608.47
Params size (MB): 558.44
Estimated Total Size (MB): 8167.11
==========================================================================================
```

VGG 网络的一个缺点是，由于需要使用小卷积核深卷积层来模拟大卷积核的效果，同时达到提取更好的特征的效果，就会容易产生过拟合。提前停止技术可以有效防止过拟合。在这里我们利用划分好的验证集，采用一下提前停止的技术，在验证集的训练损失不再显著降低的时候，提前停止训练，防止过拟合的发生。

```
Epoch [1/5], train_loss: 0.0034, train_acc: 0.9272, val_loss: 0.0008, val_acc: 0.9851
Epoch [2/5], train_loss: 0.0009, train_acc: 0.9833, val_loss: 0.0005, val_acc: 0.9890
Epoch [3/5], train_loss: 0.0007, train_acc: 0.9872, val_loss: 0.0012, val_acc: 0.9775
Epoch [4/5], train_loss: 0.0006, train_acc: 0.9882, val_loss: 0.0005, val_acc: 0.9914
Early stooping at epoch 5...
```

5、MobileNet

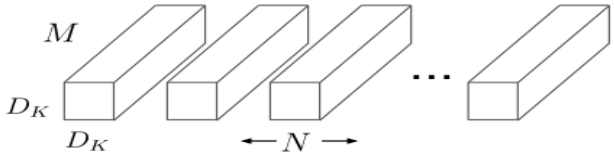
卷积神经网络（CNN）已经普遍应用在计算机视觉领域，并且已经取得了不错的效果。模型深度越来越深，模型复杂度也越来越高，卷积神经网络的运用遇到了一些问题。首先是模型过于庞大，面临着内存不足的问题，其次这些场景要求低延迟，或者说响应速度要快，想象一下自动驾驶汽车的行人检测系统如果速度很慢会发生什么可怕的事情。所以，研究小而高效的 CNN 模型在这些场景至关重要。

MobileNet 的目标是 目标在保持模型性能（accuracy）的前提下降低模型大小（parameters size），同时提升模型速度（speed, low latency）。

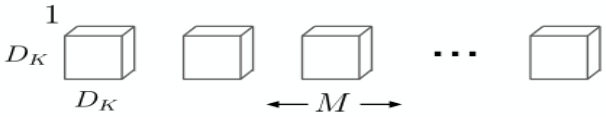
MobileNet 的基本单元是深度级可分离卷积，是一种可分解卷积操作，其可以分解为两个更小的操作：depthwise convolution 和 pointwise convolution。Depthwise convolution 和标准卷积不同，对于标准卷积其卷积核是用在所有的输入通道上（input channels），而 depthwise convolution 针对每个输入通道采用不同的卷积核，就是说一个卷积核对应一个

输入通道，所以说 depthwise convolution 是 depth 级别的操作。而 pointwise convolution 其实就是普通的卷积，只不过其采用 1x1 的卷积核。

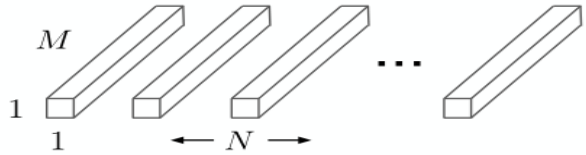
对于 depthwise separable convolution，其首先是采用 depthwise convolution 对不同输入通道分别进行卷积，然后采用 pointwise convolution 将上面的输出再进行结合，这样其实整体效果和一个标准卷积是差不多的，但是会大大减少计算量和模型参数量。



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

通过搭建网络，我们可以打印出以下网络结构，我们可以发现网络的参数只有 300 万，非常适用于敏捷快速的图像识别领域。

MobileNet 模型的结构和参数

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
MobileNet                                [64, 10]                  --
├─Sequential: 1-1                        [64, 10]                  --
│    └─Upsample: 2-1                     [64, 1, 224, 224]         --
│    └─Conv2d: 2-2                       [64, 32, 112, 112]        288
│    └─BatchNorm2d: 2-3                  [64, 32, 112, 112]        64
│    └─ReLU6: 2-4                        [64, 32, 112, 112]        --
│    └─MobileBlock: 2-5                  [64, 64, 112, 112]        --
│    │    └─Sequential: 3-1              [64, 64, 112, 112]        2,528
│    └─MobileBlock: 2-6                  [64, 128, 56, 56]         --
│    │    └─Sequential: 3-2              [64, 128, 56, 56]         9,152
│    └─MobileBlock: 2-7                  [64, 128, 56, 56]         --
│    │    └─Sequential: 3-3              [64, 128, 56, 56]         18,048
│    └─MobileBlock: 2-8                  [64, 256, 28, 28]         --
│    │    └─Sequential: 3-4              [64, 256, 28, 28]         34,688
│    └─MobileBlock: 2-9                  [64, 256, 28, 28]         --
│    │    └─Sequential: 3-5              [64, 256, 28, 28]         68,864
│    └─MobileBlock: 2-10                 [64, 512, 14, 14]         --
│    │    └─Sequential: 3-6              [64, 512, 14, 14]         134,912
│    └─MobileBlock: 2-11                 [64, 512, 14, 14]         --
│    │    └─Sequential: 3-7              [64, 512, 14, 14]         268,800
│    └─MobileBlock: 2-12                 [64, 512, 14, 14]         --
│    │    └─Sequential: 3-8              [64, 512, 14, 14]         268,800
│    └─MobileBlock: 2-13                 [64, 512, 14, 14]         --
│    │    └─Sequential: 3-9              [64, 512, 14, 14]         268,800
│    └─MobileBlock: 2-14                 [64, 512, 14, 14]         --
│    │    └─Sequential: 3-10             [64, 512, 14, 14]         268,800
│    └─MobileBlock: 2-15                 [64, 512, 14, 14]         --
│    │    └─Sequential: 3-11             [64, 512, 14, 14]         268,800
│    └─MobileBlock: 2-16                 [64, 1024, 7, 7]          --
│    │    └─Sequential: 3-12             [64, 1024, 7, 7]          531,968
│    └─MobileBlock: 2-17                 [64, 1024, 7, 7]          --
│    │    └─Sequential: 3-13             [64, 1024, 7, 7]          1,061,888
│    └─AvgPool2d: 2-18                   [64, 1024, 1, 1]          --
│    └─Flatten: 2-19                     [64, 1024]                --
│    └─Linear: 2-20                      [64, 10]                  10,250
```

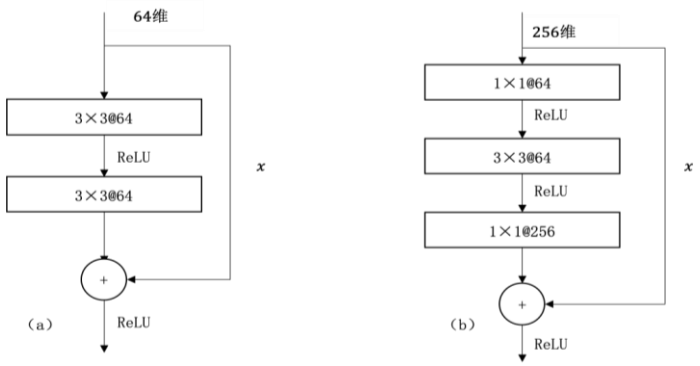
```
Total params: 3,216,650
Trainable params: 3,216,650
Non-trainable params: 0
Total mult-adds (G): 35.87
==========================================================================================
Input size (MB): 0.20
Forward/backward pass size (MB): 5163.72
Params size (MB): 12.87
Estimated Total Size (MB): 5176.78
```

6、ResNet

残差神经网络（ResNet）是由微软研究院的何恺明、张祥雨、任少卿、孙剑等人提出的。ResNet 在 2015 年的 ILSVRC（ImageNet Large Scale Visual Recognition Challenge）中取得了冠军。

残差神经网络的主要贡献是发现了“退化现象（Degradation）”，并针对退化现象发明了“快捷连接 Shortcut connection）”，极大的消除了深度过大的神经网络训练困难问题。神经网络的“深度”首次突破了 100 层、最大的神经网络甚至超过了 1000 层。非线性转换极大的提高了数据分类能力，但是，随着网络的深度不断的加大，我们在非线性转换方面已经走的太远，竟然无法实现线性转换。显然，在神经网络中增加线性转换分支成为很好的选择，于是，ResNet 团队在 ResNet 模块中增加了快捷连接分支，在线性转换和非线性转换之间寻求一个平衡。

按照这个思路，ResNet 团队分别构建了带有“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。



通过这个架构，我们可以构建 ResNet 模块。为了方便起见，我们直接将残差操作中的卷积层包装在公共残差块中，并在初始化时指定是否启用该残差块。

```python
class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride, res_conv=False):
        super(ResBlock, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=3, padding=1, stride=stride),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.Conv2d(in_channels=out_channels, out_channels=out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels)
        )
        if res_conv:
            self.res_conv = nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=1, stride=stride)
        else:
            self.res_conv = None
        self.relu = nn.ReLU()

    def forward(self, x):
        y = self.net(x)
        if self.res_conv:
            x = self.res_conv(x)
        y = y + x
        return self.relu(y)
```

搭建完成后打印模型结构，发现参数数量显著小于 VGGNet，但是分类结果却比 VGGNet 好，可以显示出残差连接的重要性。

ResNet 模型的结构和参数

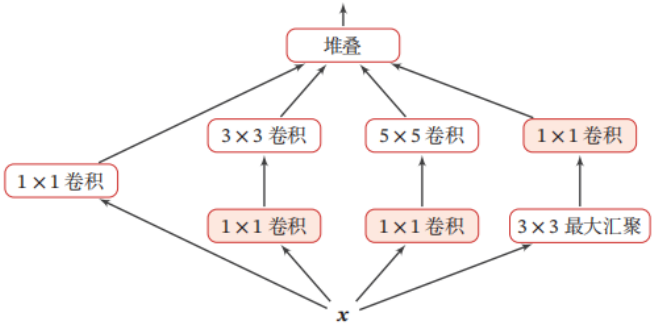
```
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
ResNet                                   [64, 10]                  --
├─Sequential: 1-1                        [64, 10]                  --
│    └─Upsample: 2-1                     [64, 1, 224, 224]         --
│    └─Conv2d: 2-2                       [64, 64, 112, 112]        3,200
│    └─BatchNorm2d: 2-3                  [64, 64, 112, 112]        128
│    └─ReLU: 2-4                         [64, 64, 112, 112]        --
│    └─MaxPool2d: 2-5                    [64, 64, 56, 56]          --
│    └─Sequential: 2-6                   [64, 64, 56, 56]          --
│    │    └─ResBlock: 3-1                [64, 64, 56, 56]          74,112
│    │    └─ResBlock: 3-2                [64, 64, 56, 56]          74,112
│    └─Sequential: 2-7                   [64, 128, 28, 28]         --
│    │    └─ResBlock: 3-3                [64, 128, 28, 28]         230,272
│    │    └─ResBlock: 3-4                [64, 128, 28, 28]         295,680
│    └─Sequential: 2-8                   [64, 256, 14, 14]         --
│    │    └─ResBlock: 3-5                [64, 256, 14, 14]         919,296
│    │    └─ResBlock: 3-6                [64, 256, 14, 14]         1,181,184
│    └─Sequential: 2-9                   [64, 512, 7, 7]           --
│    │    └─ResBlock: 3-7                [64, 512, 7, 7]           3,673,600
│    │    └─ResBlock: 3-8                [64, 512, 7, 7]           4,721,664
│    └─Sequential: 2-10                  [64, 10]                  --
│    │    └─AdaptiveAvgPool2d: 3-9       [64, 512, 1, 1]           --
│    │    └─Flatten: 3-10                [64, 512]                 --
│    │    └─Linear: 3-11                 [64, 10]                  5,130
==========================================================================================
Total params: 11,178,378
Trainable params: 11,178,378
Non-trainable params: 0
Total mult-adds (G): 111.19
==========================================================================================
Input size (MB): 0.20
Forward/backward pass size (MB): 2453.41
Params size (MB): 44.71
Estimated Total Size (MB): 2498.33
```

7、GoogLeNet

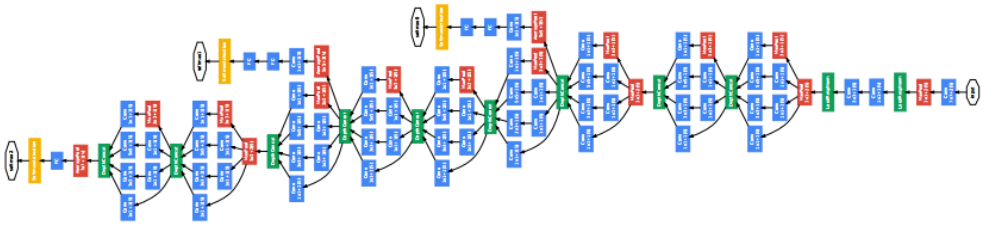
在卷积网络中，如何设置卷积层的卷积核大小是一个十分关键的问题。在 Inception 网络中，一个卷积层包含多个不同大小的卷积操作，称为 **Inception 模块** 。Inception 网络是由有多个 Inception 模块 和少量的汇聚层堆叠而成。

Inception 模块同时使用 1 × 1、3 × 3、5 × 5 等不同大小的卷积核，并将得到的特征映射在深度上拼接（堆叠）起来作为输出特征映射。

下面给出了 Inception 模块的基本结构，采用了 4 组平行的特征抽取方法，分别为 1 × 1、3 × 3、5 × 5 和 3 × 3 的最大汇聚。同时为了提高计算效率，减少参数数量， Inception 模块在进行 3 × 3、5 × 5 的卷积之前，3 × 3 的最大汇聚之后，会进行一次 1 × 1 的卷积减少特征映射的深度。如果特征映射之间存在冗余信息，那么这就相当于一次特征抽取。



GoogLeNet 由 9 个 Inception v1 模块和 5 个汇聚层以及其他一些卷积层和 全连接层构成，总共为 22 层网络。如下图所示：



我们可以使用以下的代码实现 Inception 结构:

```python
#inception结构
class Inception(nn.Module):
    def __init__(self, in_channels, ch1x1, ch3x3red, ch3x3, ch5x5red, ch5x5, pool_proj):
        super(Inception, self).__init__()

        self.branch1 = BasicConv2d(in_channels, ch1x1, kernel_size=1)

        self.branch2 = nn.Sequential(
            BasicConv2d(in_channels, ch3x3red, kernel_size=1),
            BasicConv2d(ch3x3red, ch3x3, kernel_size=3, padding=1)   # 保证输出大小等于输入大小
        )

        self.branch3 = nn.Sequential(
            BasicConv2d(in_channels, ch5x5red, kernel_size=1),
            BasicConv2d(ch5x5red, ch5x5, kernel_size=5, padding=2)   # 保证输出大小等于输入大小
        )

        self.branch4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1),
            BasicConv2d(in_channels, pool_proj, kernel_size=1)
        )

    def forward(self, x):
        branch1 = self.branch1(x)
        branch2 = self.branch2(x)
        branch3 = self.branch3(x)
        branch4 = self.branch4(x)

        outputs = [branch1, branch2, branch3, branch4]
        return torch.cat(outputs, 1)
```

为了解决梯度消失问题，GoogLeNet 在网络中间层引入两个辅助分类器加强监督信息。但是在这里由于测试的时候准确率遇到了一些问题，还没有解决，所以就暂时实现一个没有辅助分类器的版本。

网络的结构如下，可以看到通过 Inception 模块的堆叠，虽然网络的层数很深，但是总共的参数量却不大。

GoogLeNet 模型的结构和参数（截取部分）

```
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
GoogLeNet                                [64, 10]                  --
├─Upsample: 1-1                          [64, 1, 224, 224]         --
├─BasicConv2d: 1-2                       [64, 64, 112, 112]        --
│    └─Conv2d: 2-1                       [64, 64, 112, 112]        3,200
│    └─ReLU: 2-2                         [64, 64, 112, 112]        --
├─MaxPool2d: 1-3                         [64, 64, 56, 56]          --
├─BasicConv2d: 1-4                       [64, 64, 56, 56]          --
│    └─Conv2d: 2-3                       [64, 64, 56, 56]          4,160
│    └─ReLU: 2-4                         [64, 64, 56, 56]          --
├─BasicConv2d: 1-5                       [64, 192, 56, 56]         --
│    └─Conv2d: 2-5                       [64, 192, 56, 56]         110,784
│    └─ReLU: 2-6                         [64, 192, 56, 56]         --
├─MaxPool2d: 1-6                         [64, 192, 28, 28]         --
├─Inception: 1-7                         [64, 256, 28, 28]         --
│    └─BasicConv2d: 2-7                  [64, 64, 28, 28]          --
│    │    └─Conv2d: 3-1                  [64, 64, 28, 28]          12,352
│    │    └─ReLU: 3-2                    [64, 64, 28, 28]          --
│    └─Sequential: 2-8                   [64, 128, 28, 28]         --
│    │    └─BasicConv2d: 3-3             [64, 96, 28, 28]          18,528
│    │    └─BasicConv2d: 3-4             [64, 128, 28, 28]         110,720
```

```
│    └─Sequential: 2-42                  [64, 128, 7, 7]           --
│    │    └─MaxPool2d: 3-71              [64, 832, 7, 7]           --
│    │    └─BasicConv2d: 3-72            [64, 128, 7, 7]           106,624
├─AdaptiveAvgPool2d: 1-18                [64, 1024, 1, 1]          --
├─Dropout: 1-19                          [64, 1024]                --
├─Linear: 1-20                           [64, 10]                  10,250
==========================================================================================
Total params: 5,977,530
Trainable params: 5,977,530
Non-trainable params: 0
Total mult-adds (G): 96.40
==========================================================================================
Input size (MB): 0.20
Forward/backward pass size (MB): 1651.80
Params size (MB): 23.91
Estimated Total Size (MB): 1675.91
```

模型优化方案

我发现，随着网络层数的增加，过拟合等影响预测准确率的现象会不可避免地产生。为了避免这种情况，我们可以采用如下的优化方案，对于我们的模型进行一些优化。

加入权重衰减

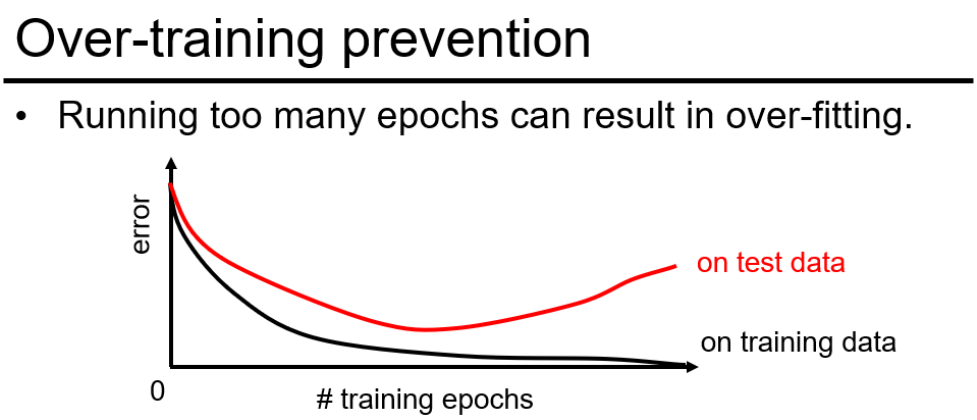
从模型的复杂度上解释：更小的权值 w，从某种意义上说，表示网络的复杂度更低，对数据的拟合更好。

从数学方面的解释：过拟合的时候，拟合函数的系数往往非常大，拟合函数需要顾忌每一个点，最终形成的拟合函数波动很大。在某些很小的区间里，函数值的变化很剧烈。这就意味着函数在某些小区间里的导数值（绝对值）非常大，由于自变量值可大可小，所以只有系数足够大，才能保证导数值很大。而正则化是通过约束参数的范数使其不要太大，所以可以在一定程度上减少过拟合情况。

在我们的模型中，我们定义优化器为 Adam，并且设置权值衰减值为 0.001

```python
# Setting the optimizer with the model parameters and learning rate
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr, weight_decay=0.001)
```

提前停止



如果运行过多的轮次，可能会造成模型的过拟合，从而降低模型在测试集上的表现。但是我们又不知道 epochs 设置为多少是合理的，因为对于每一个网络情况都不一样。在这里我们简单定义一个提前停止的技术：如果在验证集上的 loss 在连续 3 个 epoch 没有增加，就提前终止。

```
# 训练集和验证集的 loss 之差
best_loss = 1
epochs_without_improvement = 0
for epoch in range(num_epochs):
    model.train()
    # 记录每一个 epoch 的 loss 和 acc
    train_loss, train_acc, val_loss, val_acc = 0, 0, 0, 0
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = model(images)
        loss = cost(outputs, labels)
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # calculate the accuracy
        output = F.softmax(outputs, dim=1)
        pred = torch.argmax(output, dim=1)
        acc = torch.sum(pred == labels)
        train_loss += loss.item()
        train_acc += acc.item()

    model.eval()
    with torch.no_grad():
        for i, (images, labels) in enumerate(val_loader):
            images = images.to(device)
            labels = labels.to(device)
            # Forward pass
            outputs = model(images)
            loss = cost(outputs, labels)
            # calculate the accuracy
            output = F.softmax(outputs, dim=1)
            pred = torch.argmax(output, dim=1)
            acc = torch.sum(pred == labels)
            val_loss += loss.item()
            val_acc += acc.item()
    # 计算每一个 epoch 的训练损失和精度
    train_loss_epoch = train_loss / train_size
    train_acc_epoch = train_acc / train_size
    # 计算验证损失和精度
    val_loss_epoch = val_loss / val_size
    val_acc_epoch = val_acc / val_size
    # 计算验证集的平均损失看是否实现提前停止
    if val_loss_epoch < best_loss:
        best_loss = round(val_loss_epoch, 4)
        epochs_without_improvement = 0
    else:
        epochs_without_improvement += 1
    # 实现提前停止的轮数
    patience = 3
    if (epochs_without_improvement == patience):
        print('Early stooping at epoch {}...'.format(epoch+1))
        break
```

我们使用 python cnn.py -model LeNet --num_epoch 30 --batch_size 64 -lr 0.0001 运行程序，发现程序会在 21 回合时提前停止。

调整参数

我们以 LeNet 为例，对于学习率进行调整：

开始，把学习率设置为 0.01，发现在十个 epoch 中间 loss 一直很大，并且一直在上下波动，accuracy 不高。考虑是学习率过大导致梯度无法下降，减小梯度：

```
Epoch 10/10
188/188 [==============================] - 1s 8ms/step - loss: 2.3015 - accuracy: 0.1113 - val_loss: 2.3009 - val_accuracy: 0.1166
```

将学习率设置为 0.001 发现 loss 依然上下波动，accuracy 不高，再减小梯度：

```
Epoch 10/10
188/188 [==============================] - 1s 8ms/step - loss: 2.3023 - accuracy: 0.1106 - val_loss: 2.3017 - val_accuracy: 0.0975
```

将学习率设置为 0.0001 发现程序取得了较好的在训练集和测试集上面的表现，最终在验证集上面的准确率达到了 98.12%。
我们选择超参数 lr 为 0.0001。

预测结果

在这里我使用 A100-40GB 的 GPU 对于不同的模型，**采用参数： --num_epoch 5 --batch_size 64 -lr 0.0001 进行训练。得到训练时间与准确率：**

模型	LeNet	AlexNet	VGGNet-11	VGGNet-19	ResNet	MobileNet	GoogLeNet
时间（秒）	40.72	64.39	321.26	661.17	139.1	176.96	193.44
准确率（%）	97.48	99.22	99.23	99.14	99.37	98.81	99.07

可以发现以下几点：

1、 所有网络的预测准确率都超过了 95%，表现都非常不错。原因除了深度神经网络具有更好的拟合性以外，MNIST 数据集本身比较简单和优秀且完善的数据预处理也占了很大一部分。

2、 ResNet 的预测准确率是最高的，超过了专门用作图像识别的 VGGNet。说明残差的直连边对于防止过拟合、提升识别准确率是很有帮助的。

3、 LeNet 作为较早提出来的模型，因为层数不够的原因，在图像识别上还是不如后来出现的一些网络。

4、 VGGNet 的识别时间最长，其中 19 版本是 11 版本的两倍，这是因为它超量的加和数量让 GPU 产生了巨量的运算量。VGGNet 在 A100 上的表现也如此之慢，也表明此类网络在需要敏捷反应的应用场景中是较为受限的。

5、 MobileNet 的性能并没有很显著，可能是因为大家都识别得很好。但是它优秀的参数量会让它在实际应用中显得更有潜力，适合部署在移动端。

遇到的问题、总结与反思

在这次实验中，我实现了 **LeNet, AlexNet, VGGNet, MobileNet, ResNet, GoogLeNet** 等网络，并且实践了**权重衰减，提前停止**这样的优化算法，在 CNN 图像识别上有了许多探索。但是在过程中也遇到了一些问题。

问题 1: 在 LeNet 中将图片 resize 到 (32, 32) 的时候报错。
解决方法：使用 torch.nn.functional.interpolate 方法，指定需要转换成的尺寸或者变换的倍数。因为图片是使用 torch.utils.data.DataLoader 按照 batch 传入的，输入大小是 (batch_size, 1, 28, 28)，所以不能简单地去使用 resize 方法。

问题 2: 没有正确地划分训练集与验证集，导致无法进行提前停止的操作。后面已经按照文中提到的解决了。

问题 3: 使用 torchinfo.summary 的时候没有传入输入的尺寸，导致无法打印出参数的个数，不知道模型的性能如何。
解决方法：改为 summary(model, (64, 1, 28, 28), device='cuda') 即可