深度学习大作业

林宝诚 2019080030 无 98

1. 环境配置

本地: 用以训练较小的模型（模型 1，2，3）

PyTorch 版本：1.8.1

Torchvision 版本：0.9.1

Cuda 版本: 10.2

GPU 型号：Nvidia MX150

```
(pytorch) C:\Users\MY PC>python
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.__version__
'1.8.1'
>>> import torchvision
>>> torchvision.__version__
'0.9.1'
>>>
```

Google Colabolatory：用以训练较大的模型（模型 4，5）

PyTorch 版本：1.8.1

Torchvision 版本：0.9.1

Cuda 版本: 10.1

GPU 型号：Tesla T4

2. 基准模型运行

```
Train Epoch: 59 / 60 [1600/2100 (76%)] Loss: 0.052056 Accuracy: 1.000000
Train Epoch: 59 / 60 [1920/2100 (92%)] Loss: 0.061073 Accuracy: 1.000000
============================
val accuracy:46.28571701049805%
test accuracy:47.71428680419922%

Process finished with exit code 0
```

3. 深度学习实验

源代码	说明
main_final.py	性能最好的一次实验代码（实验 3.3.1 Momentum）
utility_final.py	训练 & 测试模型代码
model.py	模型网络结构代码（详情见 a）
my_resnet.py	残差网络组成部分代码
dataset.py	处理数据集代码（详情见 b）
lossfunction_final.py	损失函数代码（详情见 c）
visualization.py	结果可视化代码（详情见 d）
base_model_3_1_1.pth	实验 3.1.1 训练的 base_model 模型，main_final.py 会用到

**注：这里仅提供性能最好的一次实验代码（可执行）和每个实验用到的关键代码，若助教需要其他实验的可直接执行代码，我这里也可以整理给助教。

a. 定义网络结构

模型 1：基准模型 -- base_model

模型 2：残差模型（BasicBlock）-- MyBaseModel1

模型 3：残差模型（BottleNeck）-- MyBaseModel2

模型 4：torchvision.models 中的 ResNet18（预训练）-- MyResnet18

模型 5：torchvision.models 中的 VGG16（预训练）-- MyVgg16

**模型 4 和模型 5 由于只修改了最后输出层（从 1000 改成 35 个输出），因此报告中略过其构造

模型 1-3 构造一览（假设输入的 Tensor Shape 为[32, 3, 112, 112]）

模型 1：base_model

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
base_model                               --                        --
├─Conv2d: 1-1                            [32, 64, 56, 56]          9,472
├─BatchNorm2d: 1-2                       [32, 64, 56, 56]          128
├─ReLU: 1-3                              [32, 64, 56, 56]          --
├─MaxPool2d: 1-4                         [32, 64, 28, 28]          --
├─Conv2d: 1-5                            [32, 128, 28, 28]         73,856
├─BatchNorm2d: 1-6                       [32, 128, 28, 28]         256
├─ReLU: 1-7                              [32, 128, 28, 28]         --
├─MaxPool2d: 1-8                         [32, 128, 14, 14]         --
├─Conv2d: 1-9                            [32, 256, 14, 14]         295,168
├─AdaptiveMaxPool2d: 1-10                [32, 256, 1, 1]           --
├─BatchNorm1d: 1-11                      [32, 256]                 512
├─ReLU: 1-12                             [32, 256]                 --
├─Linear: 1-13                           [32, 64]                  16,448
├─BatchNorm1d: 1-14                      [32, 64]                  128
├─ReLU: 1-15                             [32, 64]                  --
├─Linear: 1-16                           [32, 35]                  2,275
==========================================================================================
Total params: 398,243
```

```
base_model(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3))
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu): ReLU(inplace=True)
  (max_pooling): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
  (GAP): AdaptiveMaxPool2d(output_size=(1, 1))
  (bn3): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1): Linear(in_features=256, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=35, bias=True)
)
```

模型 2：MyBaseModel1

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
MyBaseModel1                             --                        --
├─Conv2d: 1-1                            [32, 64, 56, 56]          9,408
├─BatchNorm2d: 1-2                       [32, 64, 56, 56]          128
├─ReLU: 1-3                              [32, 64, 56, 56]          --
├─MyBasicBlock: 1-4                      [32, 64, 56, 56]          --
│    └─Conv2d: 2-1                       [32, 64, 56, 56]          36,864
│    └─BatchNorm2d: 2-2                  [32, 64, 56, 56]          128
│    └─ReLU: 2-3                         [32, 64, 56, 56]          --
│    └─Conv2d: 2-4                       [32, 64, 56, 56]          36,864
│    └─BatchNorm2d: 2-5                  [32, 64, 56, 56]          128
│    └─ReLU: 2-6                         [32, 64, 56, 56]          --
├─MyBasicBlock: 1-5                      [32, 128, 28, 28]         --
│    └─Conv2d: 2-7                       [32, 128, 28, 28]         73,728
│    └─BatchNorm2d: 2-8                  [32, 128, 28, 28]         256
│    └─ReLU: 2-9                         [32, 128, 28, 28]         --
│    └─Conv2d: 2-10                      [32, 128, 28, 28]         147,456
│    └─BatchNorm2d: 2-11                 [32, 128, 28, 28]         256
│    └─Sequential: 2-12                  [32, 128, 28, 28]         --
│    │    └─Conv2d: 3-1                  [32, 128, 28, 28]         8,192
│    │    └─BatchNorm2d: 3-2             [32, 128, 28, 28]         256
│    └─ReLU: 2-13                        [32, 128, 28, 28]         --
├─AdaptiveMaxPool2d: 1-6                 [32, 128, 1, 1]           --
├─Linear: 1-7                            [32, 35]                  4,515
==========================================================================================
Total params: 318,179
```

```
MyBaseModel1(
  (conv): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (BasicBlock1): MyBasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (BasicBlock2): MyBasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (GAP): AdaptiveMaxPool2d(output_size=(1, 1))
  (fc): Linear(in_features=128, out_features=35, bias=True)
)
```

模型 3：MyBaseModel2

```
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
MyBaseModel2                             --                        --
├─Conv2d: 1-1                            [32, 64, 56, 56]          9,408
├─BatchNorm2d: 1-2                       [32, 64, 56, 56]          128
├─ReLU: 1-3                              [32, 64, 56, 56]          --
├─MyBottleneck: 1-4                      [32, 128, 56, 56]         --
│    └─Conv2d: 2-1                       [32, 32, 56, 56]          2,048
│    └─BatchNorm2d: 2-2                  [32, 32, 56, 56]          64
│    └─ReLU: 2-3                         [32, 32, 56, 56]          --
│    └─Conv2d: 2-4                       [32, 32, 56, 56]          9,216
│    └─BatchNorm2d: 2-5                  [32, 32, 56, 56]          64
│    └─ReLU: 2-6                         [32, 32, 56, 56]          --
│    └─Conv2d: 2-7                       [32, 128, 56, 56]         4,096
│    └─BatchNorm2d: 2-8                  [32, 128, 56, 56]         256
│    └─Sequential: 2-9                   [32, 128, 56, 56]         --
│    │    └─Conv2d: 3-1                  [32, 128, 56, 56]         8,192
│    │    └─BatchNorm2d: 3-2             [32, 128, 56, 56]         256
│    └─ReLU: 2-10                        [32, 128, 56, 56]         --
├─MyBottleneck: 1-5                      [32, 256, 28, 28]         --
│    └─Conv2d: 2-11                      [32, 64, 56, 56]          8,192
│    └─BatchNorm2d: 2-12                 [32, 64, 56, 56]          128
│    └─ReLU: 2-13                        [32, 64, 56, 56]          --
│    └─Conv2d: 2-14                      [32, 64, 28, 28]          36,864
│    └─BatchNorm2d: 2-15                 [32, 64, 28, 28]          128
│    └─ReLU: 2-16                        [32, 64, 28, 28]          --
│    └─Conv2d: 2-17                      [32, 256, 28, 28]         16,384
│    └─BatchNorm2d: 2-18                 [32, 256, 28, 28]         512
│    └─Sequential: 2-19                  [32, 256, 28, 28]         --
│    │    └─Conv2d: 3-3                  [32, 256, 28, 28]         32,768
│    │    └─BatchNorm2d: 3-4             [32, 256, 28, 28]         512
│    └─ReLU: 2-20                        [32, 256, 28, 28]         --
├─AdaptiveMaxPool2d: 1-6                 [32, 256, 1, 1]           --
├─Linear: 1-7                            [32, 35]                  8,995
==========================================================================================
Total params: 138,211
```

```
MyBaseModel2(
  (conv): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (BottleNeck1): MyBottleneck(
    (conv1): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (BottleNeck2): MyBottleneck(
    (conv1): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (GAP): AdaptiveMaxPool2d(output_size=(1, 1))
  (fc): Linear(in_features=256, out_features=35, bias=True)
```

卷积层参数计算回顾：

一个卷积层是由多个卷积滤波器组成：

卷积滤波器大小：（Cin x Kh x Kw）[输入通道，滤波器高度，滤波器宽度]

卷积层：（Cout x Cin x Kh x Kw）[输出通道（滤波器数量），输入通道，滤波器高度，滤波器宽度]

偏置：Cout（即每个卷积滤波器对应一个偏置）

所以，$total\ number\ of\ parameters = Cout \times Cin \times Kh \times Kw + Cout\ (if\ bias\ is\ enabled)$

卷积层输出大小回顾：

卷积层输入大小：BS x Cin x Hin x Win [Batch Size，输入通道，输入高度，输入宽度]

卷积层输出大小：BS x Cout x Hout x Wout [Batch Size，输出通道（滤波器数量），输出高度，输出宽度]

$$Wout = floor(\frac{Win - Kw + 2P}{S}) + 1$$

*输出高度的计算方法一致，只需将 W 改为 H 即可。

线性层参数回顾：

线性层：In x Out [输入神经元数量，输出神经元数量]

参数数量 $total\ number\ of\ parameters = In \times Out$

线性层输出大小：最后一个维度从 In 变为 Out，其他维度不变

例：输入大小：BS x In → 输出大小：BS x Out

Batch Normalization 层

参数：Cout x 2

残差模型科普

理论上，而言随着深度学习模型越来越深（隐藏层越来越多），其性能会比较浅的模型好或者一致（较浅模型性能为其下限）。举个例子，假定一个较简单的模型性能已经达到 x%，通过增加新的隐藏层，如果要达到较浅模型的性能下限，这些新加入的隐藏层只需要学习 Identity Function，即输入与输出一致的隐藏层。但是，根据 Paper 的实验结果，到一定深度时，增加模型的深度反而会降低性能，这就表明了 Identity Function 是较难学习的。因此，若要通过增加深度提升模型性能，就必须要解决此问题。Paper 中提出的想法即为残差模型。如图所示，通过将输入与输出相连，可以直接为模型提供一个很难学会的 Identity funtion，模型就能突破瓶颈，模型随着深度增加而性能下降的问题得以缓解。

***Paper：** K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition"

$$g(x) = x + f(x)$$

如果要学 Identity function，$f(x) = 0$ 即可；如果是一般的学习，$f(x) = g(x) - x$。

这部分的科普仅说明残差网络的中心思想，细节部分我还没完全摸透，如有错误或遗漏部分，还请老师助教指正，谢谢！

b. 数据集（Dataset.py）

数据集	类别数	每个类别样本数量	样本总数
训练集	35	60	2100
验证集	35	10	350
测试集	35	40	1400
补充数据集	35	20	700

dataset.py 中的一些新函数

- _samples_filtering & choose_class

  功能：在将数据录入 Dataloader 之前，用以选出特定组别的数据，在进行特征可视化的时候会用到。

- _create_noisy_data

  功能：用以将训练样本的初始标签按照一定比例随机设置，在进行抗噪声样本研究的时候会用到

---

c. 损失函数

损失函数 1：交叉熵 Cross Entropy Loss

计算方法：

根据 PyTorch 官网的定义，单个样本损失：

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

最后输出的损失为 Batch 内所有样本 Loss 的平均值。

损失函数 2：Triplet Loss（Cosine Similarity） + Cross Entropy Loss（分类训练）

计算方法：

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

首先，将 Batch 内所有特征向量归一化，使其长度为 1 。接着，计算 Batch 内任意两个特征向量的**内积**（即图中的方程）。对于一个特定向量，从属于同组别的向量内积中选取最小的值；从属于不同组别的向量内积中选取最大的数值并计算 Loss：

$$L(a, p, n) = \max\{s(a_i, n_i) - s(a_i, p_i) + margin, 0\}$$

针对 Batch 内每一个向量计算对应的 Loss 后，将所有 Loss 相加取平均作为最终输出的损失。

损失函数 3：Triplet Loss（欧氏距离）+ Cross Entropy Loss（分类训练）

计算方法：

首先，计算 Batch 内任意两个特征向量的**欧氏距离**。对于一个特定向量，从属于同组别的向量内积中选取最大的值；从属于不同组别的向量内积中选取最小的数值并计算 Loss：

$$L(a,p,n) = \max\{d(a_i,p_i) - d(a_i,n_i) + margin, 0\}$$

针对 Batch 内每一个向量计算对应的 Loss 后，将所有 Loss 相加取平均作为最终输出的损失。

损失函数 4：平方误差（Squared Loss），均方误差（Mean Squared Loss）

计算方法：

- 平方误差：
  计算每个样本输出与真实标签的**平方差**，接着把整个 batch 中所有的 Loss **相加**得到最终输出的损失。

- 均方误差：
  计算每个样本输出与真实标签的**均方差**，接着把整个 batch 中所有的 Loss **相加取平均**得到最终输出的损失

**注：对于两个不同形式的 Triplet Loss，具体想法在损失函数部分会说明。

---

d. 结果可视化

可视化分析 1：PCA（特征分布分析）

可视化分析 2：T-SNE（特征分布分析）

可视化分析 3：训练& 验证准确率 & Loss 变化 & 测试准确率

可视化分析 4：性能随超参数的变化（学习率），性能随噪声样本比例变化

可视化分析 5：混淆矩阵（Confusion Matrix）

3.1.1）实验内容：对每个模型利用如下参数进行训练

实验 3.1.1 参数设置：

Image_size: [112, 112]

Batch_size: 32

Learning_rate: 0.01

Milestones=[15]

Epochs: 20

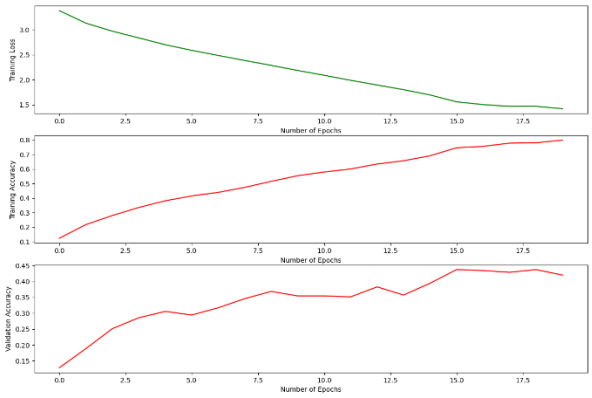
Loss_function: Cross Entropy Loss

Optimizer: Stochastic Gradient Descent (SGD)

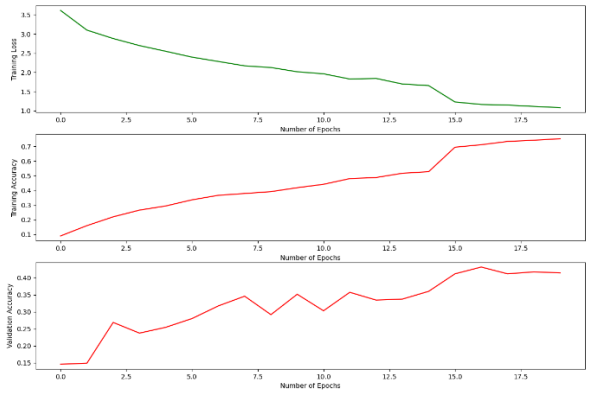
模型 1-3 训练实验

1. Base_model
2. MyBaseModel1
3. MyBaseModel2

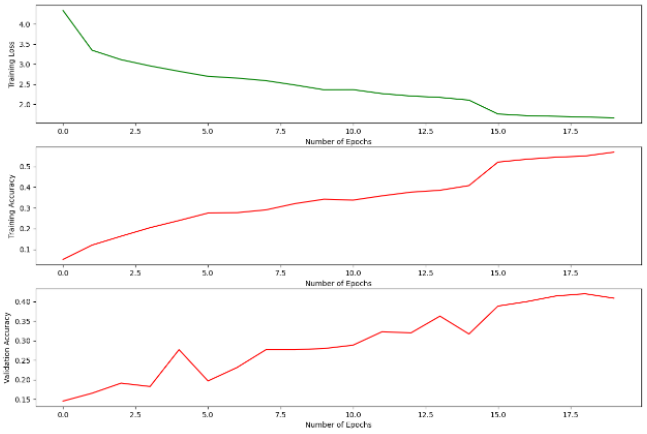
训练结果：



图【1】：base_model 训练数据



图【2】：MyBaseModel1 训练数据



图【3】：MyBaseModel2 训练数据

测试准确率结果：

base_model：约 38%

MyBaseModel1：39.79%

```
test accuracy:39.78571319580078%
```

MyBaseModel2：36.43%

```
test accuracy:36.42856979370117%
```

总体分析：

从结果看到每个模型的 Loss 和准确率分别呈下降和上升趋势，因此如果增加训练的周期可能会得到更好的模型。另外，若仔细看准确率的具体数值，可以发现到模型 1 和模型 2 中的训练准确率&验证和测试准确率的差距大，因此模型有过拟合的倾向。造成差距较大的可能原因有很多，包括训练数据集太少、损失函数不佳、模型太复杂等许多其他原因。再者，模型 3 的表现与其他两个模型有些许不同。虽然前者的总体准确率是最低，但是训练准确率与验证&测试准确率较为接近，因此其过拟合程度较低，有可能需要训练更多周期或添加数据才能够有准确率的突破。

最后，值得注意的是，在 Epoch=15 部分，准确率有个明显的跃升，根据设置（Milestones），恰好在这个周期会将学习率调整至 0.001，可见学习率对模型训练的重大影响。

模型 4、5 训练实验

1. Vgg16
2. Resnet18

实验说明：

这两个模型在训练时，其特征提取层利用了预训练的结果，训练中只我将特征提取层的训练冻结了，只训练分类器（Classifier）部分。另外，由于这两个预训练模型是使用大小为[224，224]的图片进行训练的，与本次实验使用的[112,112]大小不同，因此，我想在不更改特征提取层参数的情况下比较两者的性能差异。

测试准确率结果：

图片大小：[112,112]

Vgg16：85.29%

```
test accuracy:85.28571319580078%
```

Resnet18：77.43%

```
test accuracy:77.42857360839844%
```

图片大小：[224,224]

Vgg16：92.64%

```
test accuracy:92.64285278320312%
```

Resnet18：90.43%

```
test accuracy:90.42857360839844%
```

横向分析（同一个模型）：

从结果可以清楚看到使用[112,112]大小的图片进行训练的结果较差，这是因为像素较少的图片中会有较多的信息损失，因此利用同样的特征提取层会欠缺准确，如果想要进一步提升准确率，必须要调整特征提取层的参数。另一方面，此实验也证明了预处理数据的重要性，即在特征提取层参数不变的情况下，输入图像数据的大小必须与原训练的图片大小一致，否则其无法发挥最好的性能。

纵向分析（模型间）：

根据实验结果，可发现 Vgg16 在此次实验中的性能较好。通过比较两者的参数数量（Vgg16 分类器有，Resnet 分类器有），可以合理推测 Resnet18 的性能相对较差是因为其可以训练的参数远少于 Vgg16，自由度较低，进而导致分类器没办法很准确地对 35 个组别进行分类。

总体分析：

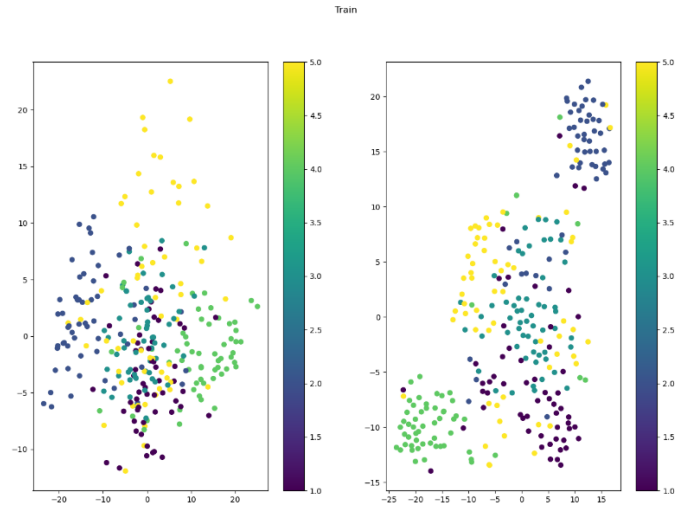
与模型 1-3 相比，这两个预训练模型的性能不是同一个级别上。这是由于预训练模型是由前沿的优秀研究员利用大量的数据进行训练，并花了一段时间进行详细的参数调整，最终得到的最佳成果。而此次深度学习实验中，因受限于数据量、计算资源、时间和能力，所以是没办法从“零”训练起一个可以与他们平起平坐的模型（即使分类任务简单许多）。最后，使用这两个预训练模型进行实验的目的是想要了解较前端的深度学习模型，并以此为榜样进行学习。

3.1.2）对模型选取 5 个组别进行特征可视化分析

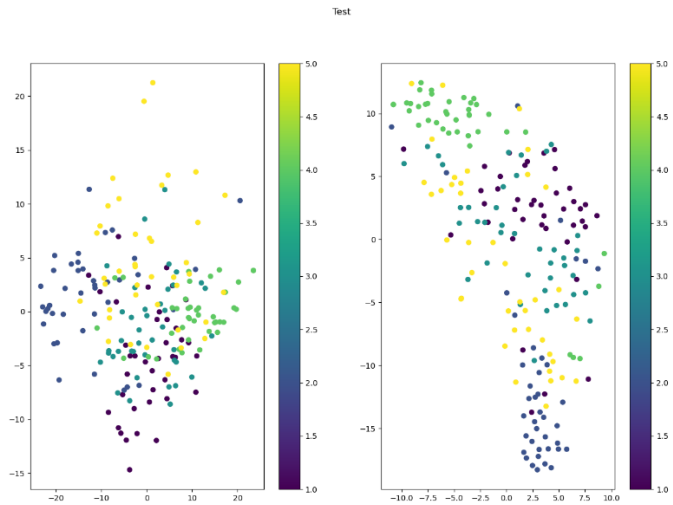
实验 3.1.2 统一选取组别：1，2，3，4，5

*注：在一个图中，左图为 PCA，右图为 TSNE

1. Base_model

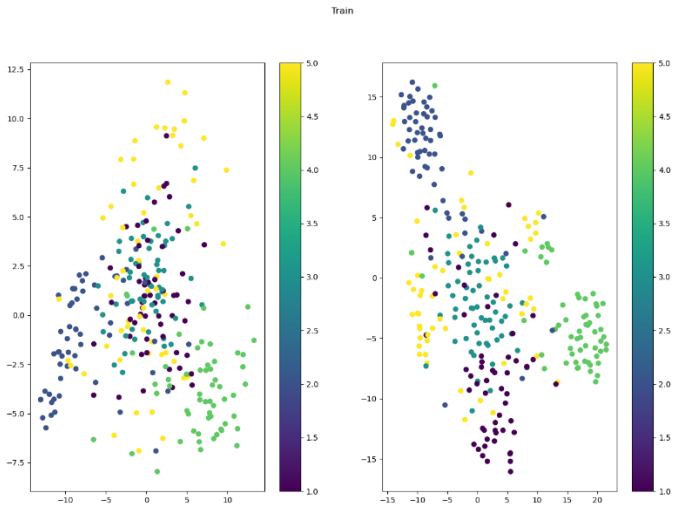


图【4】：base_model 训练集特征向量

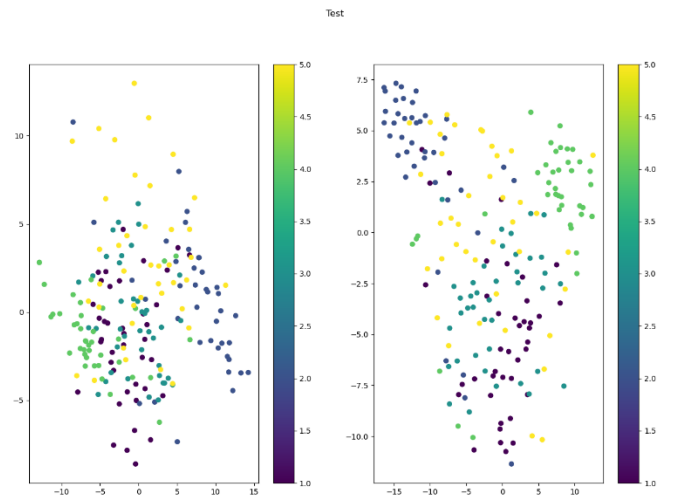


图【5】：base_model 测试集特征向量

2. MyBaseModel1

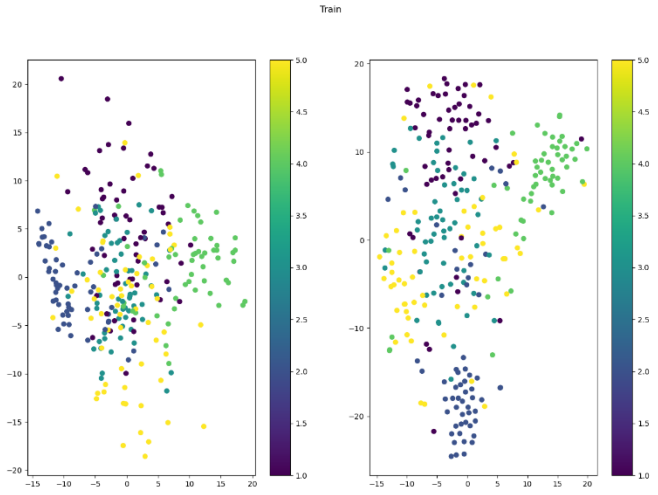


图【6】：MyBaseModel1 训练集特征向量

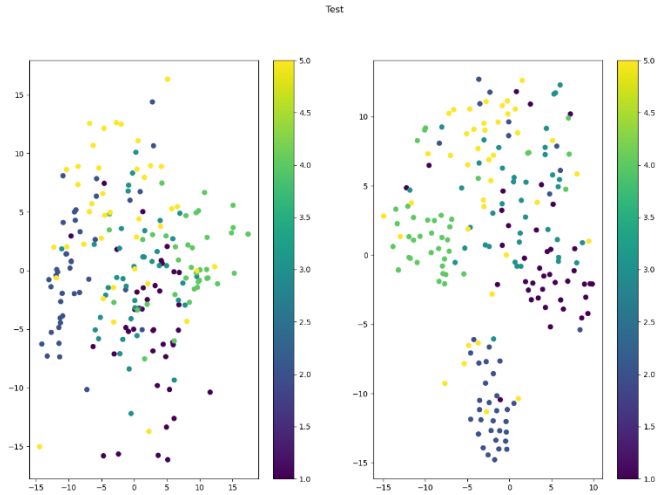


图【7】：MyBaseModel1 测试集特征向量

3. MyBaseModel2



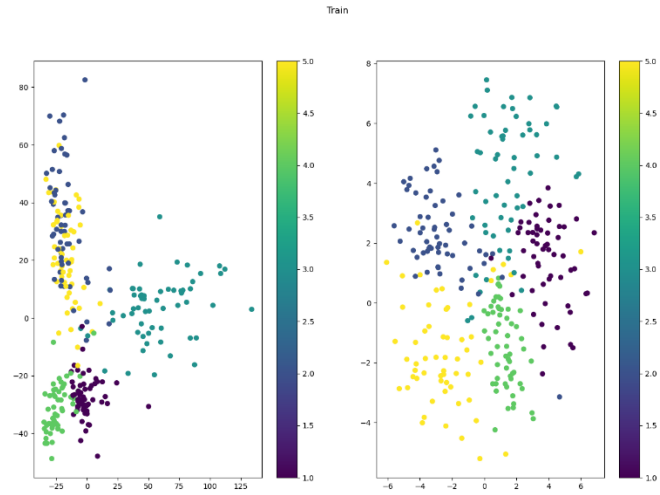
图【8】： MyBaseModel2 训练集特征向量



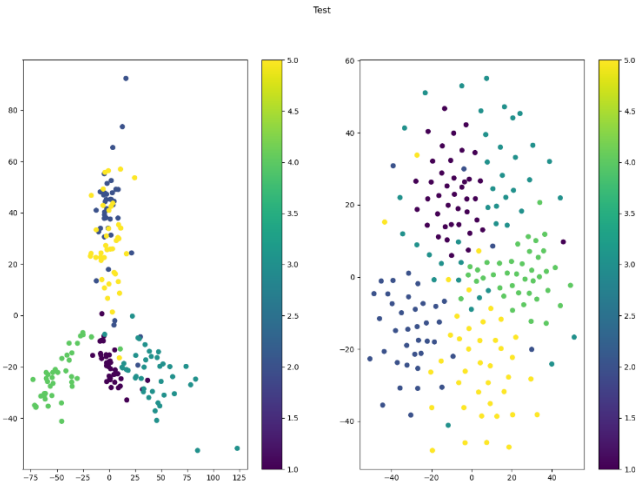
图【9】： MyBaseModel2 测试集特征向量

4. Vgg16

图片大小：[112,112]



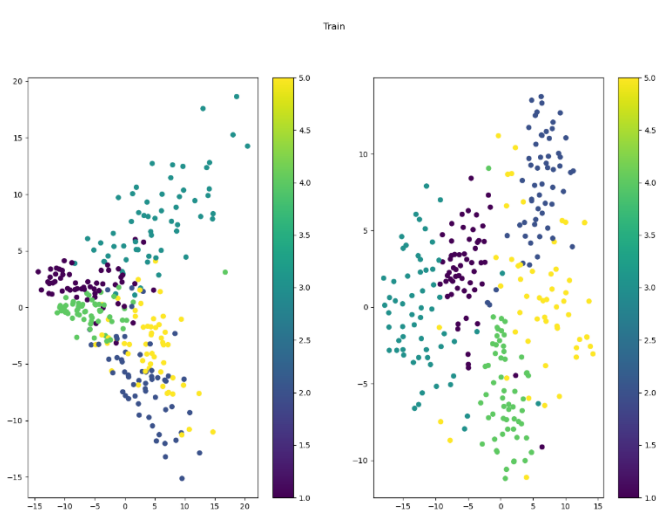
图【10】： Vgg16 [112,112] 训练集特征向量



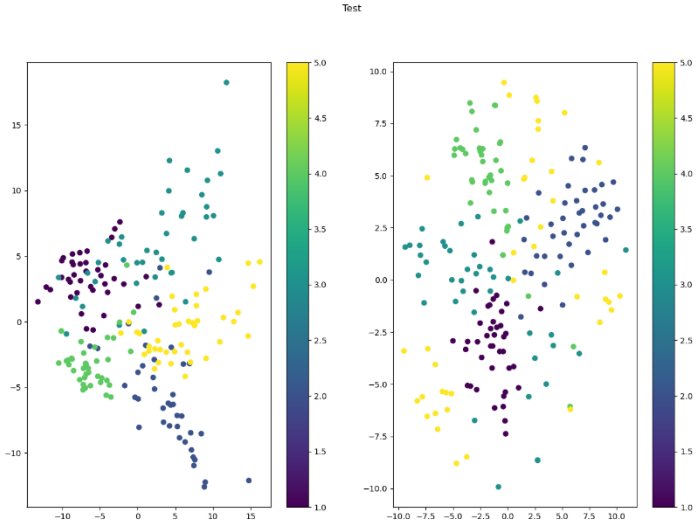
图【11】： Vgg16 [112,112] 测试集特征向量

5. Resnet18

图片大小：[112,112]

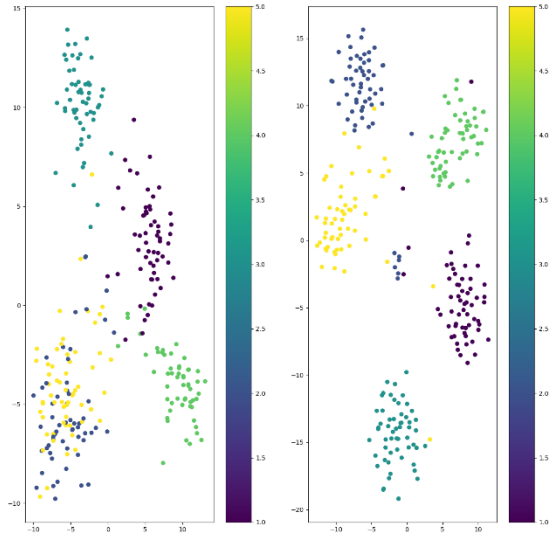


图【12】：Resnet18 [112,112] 训练集特征向量

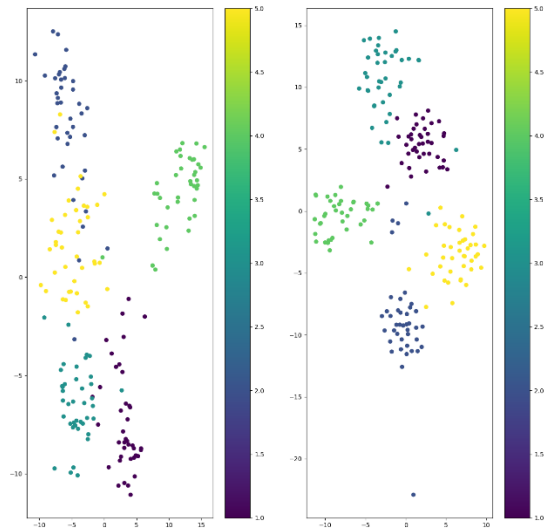


图【13】：Resnet18 [112,112] 测试集特征向量

图片大小：[224,224]



图【14】：Resnet18 [224,224] 训练集特征向量

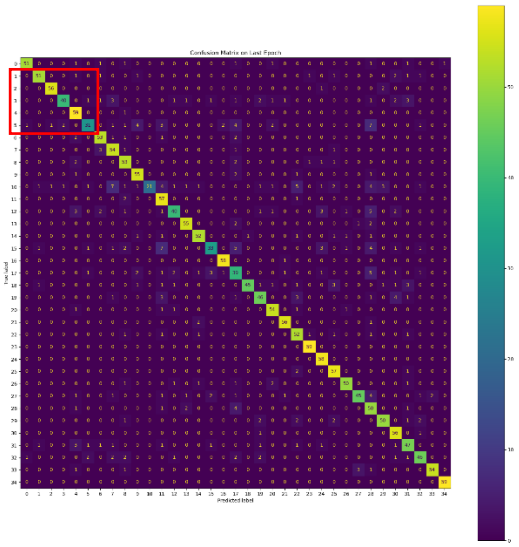


图【15】：Resnet18 [224,224] 测试集特征向量

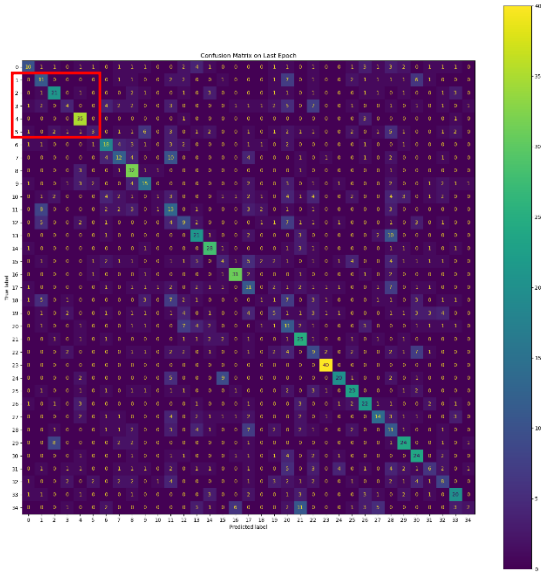
总体分析：

首先，模型 1-3 的测试集特征向量可视化总体效果欠佳，毕竟测试准确率较低。值得注意的是，模型 1 和模型 2 的训练准确率都不算低（达到 70%以上），可是训练集的特征向量并未分成 5 组，其中第 3 组和第 5 组（TSNE）基本参杂在一起，且在测试集特征向量 TSNE 图中，第第 3 组和第 5 组的效果也最差。为了辅助分析，这里使用了模型 1 和模型 2 的混淆矩阵。从混淆矩阵（注意红框部分），虽然两个模型对于组别 3 和 5 的训练正确率高，可是测试正确率却远远较低，同时联想起特征向量可视化的图，这种情况的确合乎常理。因此，可以合

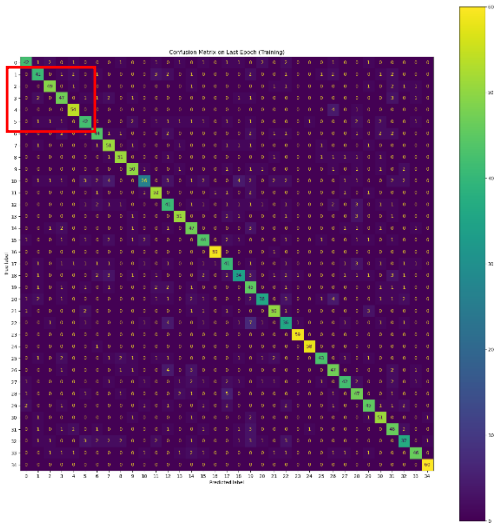
理推测模型对于这两个组别过拟合了，即模型在训练时并没有正确提取特征，而是硬生生‘背诵’了数据。至于模型 3，其情况与上述分析相似，只是其总体特征提取和分类的性能较差。



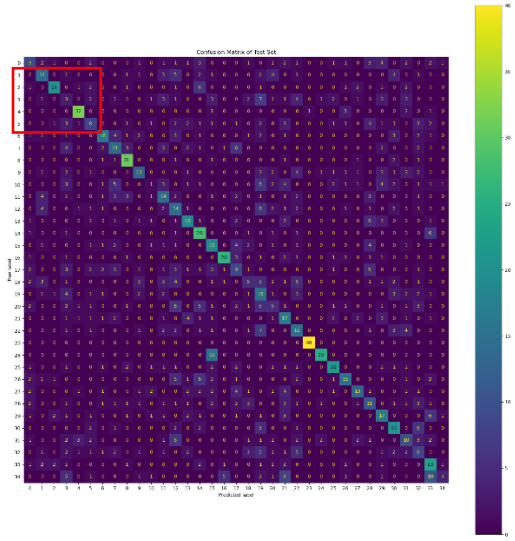
图【16】：base_model 训练集混淆矩阵



图【17】：base_model 测试集混淆矩阵



图【18】：MyBaseModel1 训练集混淆矩阵



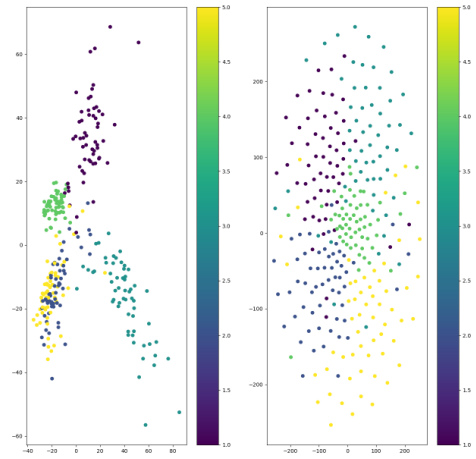
图【19】：MyBaseModel1 测试集混淆矩阵

其次，两个使用了[112, 112]大小图片的预训练模型的特征向量可视化效果明显优于前三个模型，可见可视化效果和模型的性能之间有很强的关联性。

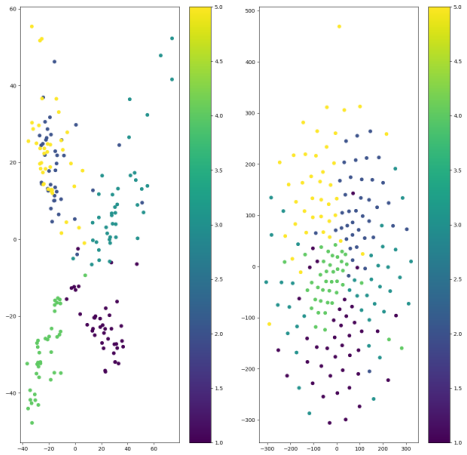
值得一提的是，作为模范，我对 Resnet18（使用图片大小[224，224]）的特征向量也进行可视化，其效果全面碾压了其他模型的可视化效果，因此其分类准确率达到 90%是十分合理的。

额外发现：

除了使用图片大小[224，224]的 Resnet18 之外，我也对图片大小[224，224]的 Vgg16 的特征向量进行了可视化，可是结果却不在我的预料之中。如图所示，其效果并没有与 Resnet18 平起平坐。对于此现象，我有想到两种解释（不知道对不对），第一种是 Vgg16 的特征向量的原本维度很高（25088 维，Resnet 有 512 维），因此进行降维后损失的信息较多导致效果较差；第二种解释是用来训练 Vgg16 特征提取层和 Resnet18 特征提取层的损失函数不同，前者的效果似乎比较像是使用 Softmax Loss (Cross Entropy)，而后者则较像 Center Loss 或 Triplet Loss 的效果（即不同类间的距离大）

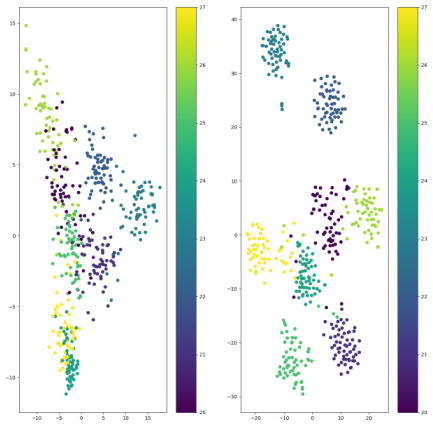


图【20】：Vgg16 [224,224] 训练集特征向量

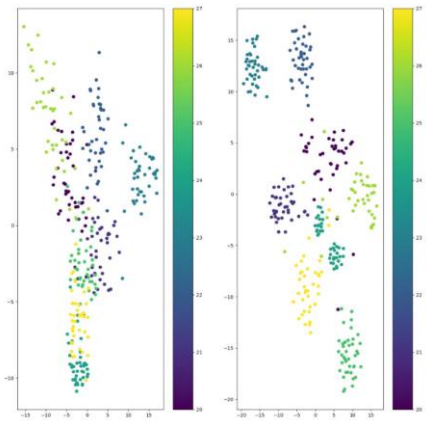


图【21】：Vgg16 [224,224] 测试集特征向量

最后，附上两个使用了附加数据训练后的 Resnet18（图片大小[224,224]）的可视化（更多组别）供欣赏用途。



图【22】：Resnet18 [224,224]8 组训练集特征向量



图【23】：Resnet18 [224,224] 8 组测试集特征向量

3.2.1）实验内容：对模型 1、2 调整损失函数，并重新训练。

实验说明：

由于实验 3.1.1 仅使用了 Cross Entropy 进行训练，此实验将会使用另外三个不同的损失函数训练模型，并观察其性能。

实验 3.2.1 参数设置：

Image_size: [112, 112]

Batch_size: 32

Learning_rate: 0.01

Milestones=[15]

Epochs: 20

Optimizer: Stochastic Gradient Descent (SGD)

损失函数 1：Cross Entropy，结果见实验 3.1.1

损失函数 2：Triplet Loss (Cosine Similarity) + Cross Entropy （T1），Margin=0.25

训练方法：

- 首先，将分类器部分（Linear 层和其中的 Batch Norm 层）的参数冻结，并利用上述参数训练模型的特征提取部分（卷积层）
- 完成特征提取部分的训练后，将所有除了分类器部分的参数冻结，再利用上述参数训练模型的分类器部分。

训练结果：

1. base_model[4,8,16,23,24]

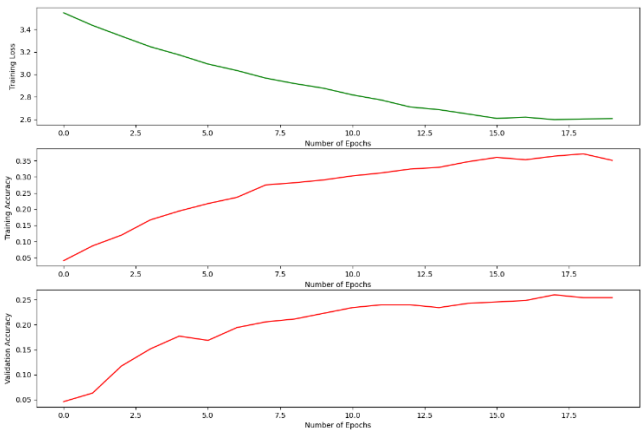
Train Epoch: 19 / 20 [0/2100 (0%)] Loss: 0.212398 Accuracy: 0.000000 Time Taken: 0.3369927406311035 seconds

Train Epoch: 19 / 20 [640/2100 (31%)] Loss: 0.188385 Accuracy: 0.062500 Time Taken: 8.186510562896729 seconds

Train Epoch: 19 / 20 [1280/2100 (62%)] Loss: 0.191547 Accuracy: 0.031250 Time Taken: 16.808339834213257 seconds

Train Epoch: 19 / 20 [1920/2100 (92%)] Loss: 0.234646 Accuracy: 0.031250 Time Taken: 24.91096520423889 seconds

图【24】：base_model 特征提取层部分训练数据（T1）



图【25】：base_model 分类层训练数据（T1）

测试准确率：25.57%

test accuracy:25.571430206298828%

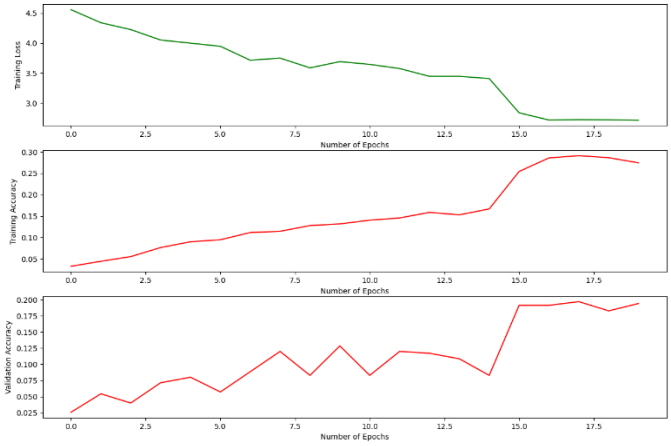
MyBaseModel1 [4,8,14,23,24]

Train Epoch: 19 / 20 [0/2100 (0%)] Loss: 0.247513 Accuracy: 0.000000 Time Taken: 0.5209920406341553 seconds

Train Epoch: 19 / 20 [640/2100 (31%)] Loss: 0.247348 Accuracy: 0.062500 Time Taken: 11.66211748123169 seconds

Train Epoch: 19 / 20 [1280/2100 (62%)] Loss: 0.245573 Accuracy: 0.156250 Time Taken: 22.23582410812378 seconds

Train Epoch: 19 / 20 [1920/2100 (92%)] Loss: 0.249786 Accuracy: 0.093750 Time Taken: 32.45624566078186 seconds



图【26】：MyBaseModel1 特征提取层部分训练数据（T1）　　图【27】：MyBaseModel1 分类器训练数据（T1）

测试准确率：19.93 %

test accuracy:19.928571701049805%

损失函数 3： Triplet Loss （欧氏距离）+ Cross Entropy （T2）， Margin=0.25

训练方法：同损失函数 2

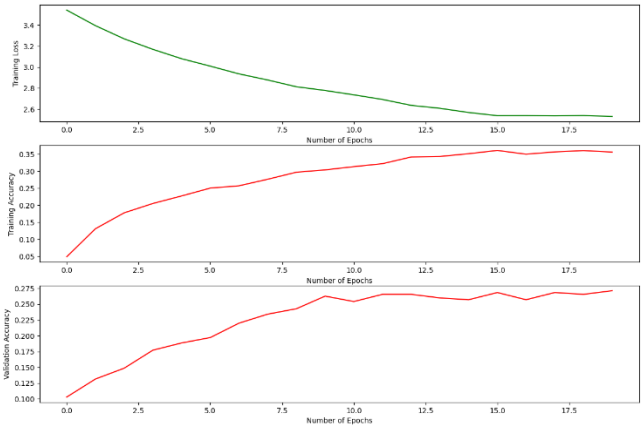
1. base_model

Train Epoch: 19 / 20 [0/2100 (0%)] Loss: 0.168964 Accuracy: 0.000000 Time Taken: 0.3750009536743164 seconds

Train Epoch: 19 / 20 [640/2100 (31%)] Loss: 0.320787 Accuracy: 0.093750 Time Taken: 8.58427095413208 seconds

Train Epoch: 19 / 20 [1280/2100 (62%)] Loss: 0.300530 Accuracy: 0.031250 Time Taken: 17.085932970046997 seconds

Train Epoch: 19 / 20 [1920/2100 (92%)] Loss: 0.242453 Accuracy: 0.031250 Time Taken: 25.073596477508545 seconds



图【28】：base_model 特征提取层部分训练数据（T2）　　图【29】：base_model 分类层训练数据（T2）

测试准确率：25.64%

test accuracy:25.64285659790039%

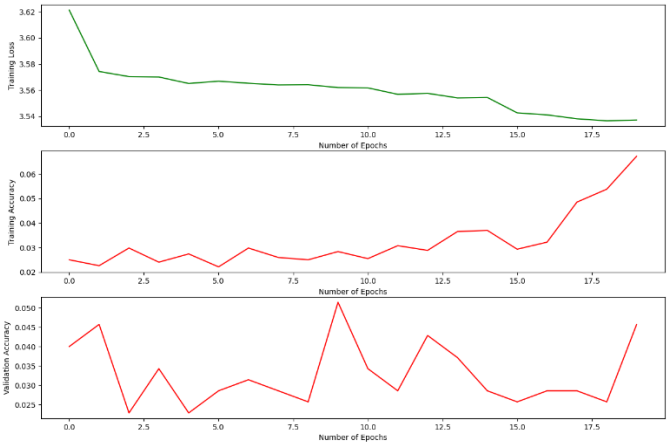
2. MyBaseModel1

Train Epoch: 19 / 20 [0/2100 (0%)] Loss: 0.162580 Accuracy: 0.062500 Time Taken: 0.47403430938720703 seconds

Train Epoch: 19 / 20 [640/2100 (31%)] Loss: 0.192145 Accuracy: 0.000000 Time Taken: 11.165436506271362 seconds

Train Epoch: 19 / 20 [1280/2100 (62%)] Loss: 0.278406 Accuracy: 0.000000 Time Taken: 21.6620032787323 seconds

Train Epoch: 19 / 20 [1920/2100 (92%)] Loss: 0.240548 Accuracy: 0.062500 Time Taken: 31.713863611221313 seconds



图【30】：MyBaseModel1 特征提取层部分训练数据（T2）　图【31】：MyBaseModel1 分类器训练数据（T2）

测试准确率：5.07%

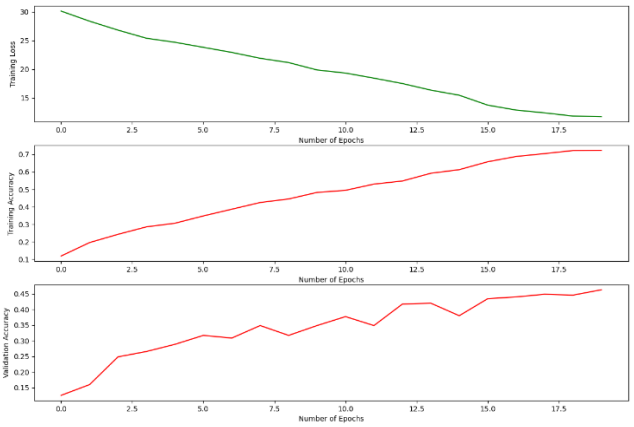
test accuracy:5.071428298950195%

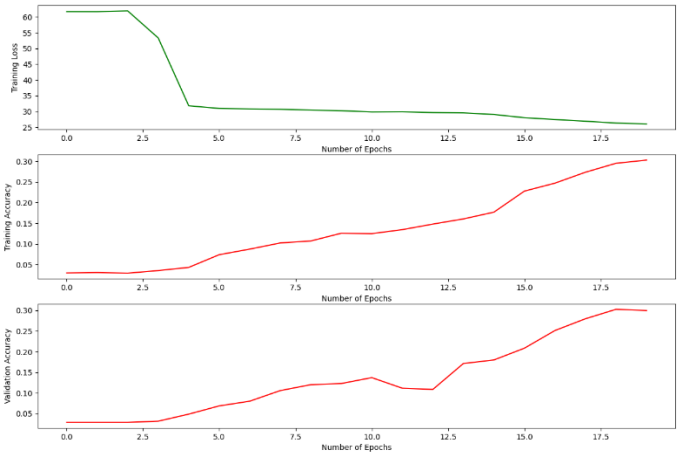
损失函数 4：平方差 Square Loss

训练方法：将真实标签变成 One Hot 形式，并在输出层添加 Softmax 函数，其他训练细节与 Cross Entropy 一致。

1.base_model

2.MyBaseModel1





图【32】：base_model 训练数据（方差）　图【33】：MyBaseModel1 训练数据（方差）

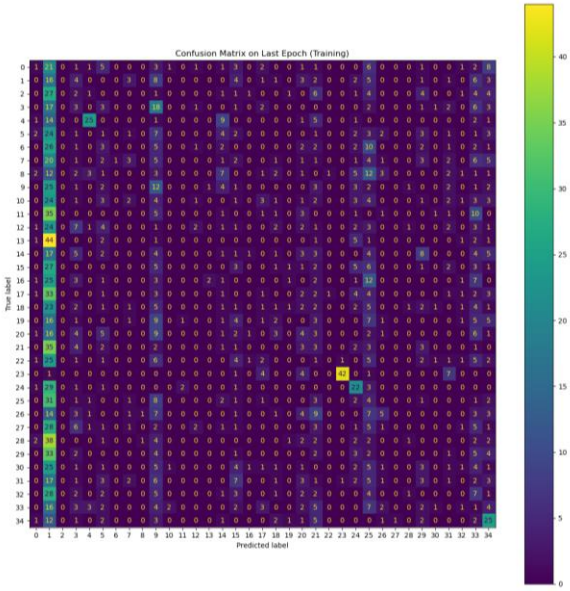
测试准确率

1. base_model：约 44%
2. MyBaseModel1：26.14%

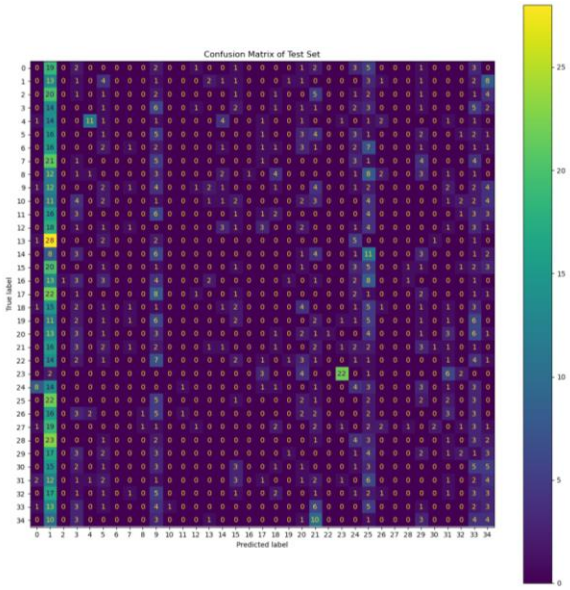
```
test accuracy:26.14285659790039%
```

结果分析：

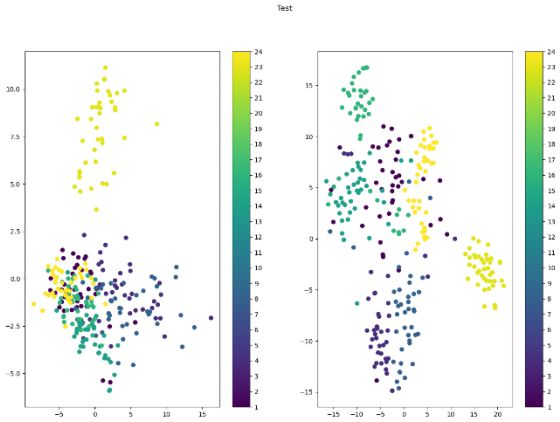
首先，通过观察个模型使用不同损失函数进行训练所获得的混淆矩阵，我发现到基本每个模型对六组训练数据的学习情况较好，这六组分别为【4，8，14，16，23，24】。另外针对 MyBaseModel1 在使用损失函数 3（T2）进行训练所展现的糟糕性能（准确率：5%），通过观察混淆函数，我发现到一个有趣的现象，如图【34】和图【35】所示。在使用损失函数 2 训练后，模型几乎把所有的样本判断为第一组！因此，为了辅助分析，我将组别 1 也纳入考量，并将其与六组分类效果较好的组别一同进行特征向量可视化。



图【34】：MyBaseModel1 训练集混淆矩阵（T2）



图【35】：MyBaseModel1 测试集混淆矩阵（T2）



图【36】：MyBaseModel1 测试集特征向量可视化（T2）

根据图【36】，可以清楚发现第 23 组别的分类效果较好，与混淆矩阵结果相符。可是，令我感到奇怪的是，若单从图【36】进行猜测，模型对剩下 6 个组别的分类性能应该不会太糟糕（与实际情况相反）。针对这个问题，我并不是很确定其中的原因。排除代码出错等因素（代码实现看起来没错），我认为最有可能的原因包括 MyBaseModel1 的分类器部分性能有限（与 base_model 相比）和这个损失函数并没有将特征提取层训练好（此特征向量可视化可能损失了某些重要信息）。

接下来的部分会对每个损失函数进行具体分析：

1. **Cross Entropy：**

在进行分类问题时，交叉熵损失函数的使用很广泛，模型输出为样本属于某一组别的概率，概率最大的组别则为模型判断的组别。交叉熵损失函数与最大似然估计有一定的联系，即减少损失等同于通过最大似然估计的方法决定最佳的参数。

1. **Triplet Loss：**

Triplet Loss 的主要想法是将同样组别样本的特征向量在高维空间上尽可能接近，而不同组别样本的特征向量尽可能远离，以达到分类的效果。实验中使用两种不同方法定义两个向量是否接近，分别为 Cosine Similarity 和欧氏距离。其中，前者是计算向量与向量间的夹角（不在乎向量长度），即相同组别的特征向量间应该要有较小的夹角，不同组别则反之。后者就是直接计算两者在空间上的距离，即同组的特征向量欧氏距离较小，反之较大。

- **问题 1：**计算向量与向量间夹角是否真的合适？

  分析：在此分类问题上，我的想法是 Cosine Similarity 可能不太适合，因为其忽略了特征向量长度的意义。即使向量间夹角较小，在空间中实际的特征向量可能距离很远（向量间长度差距很大，可是方向接近），而这时候两个向量可能分别属于不同类别，可是算法却认为他们是同组的。

- **问题 2：**使用欧式距离作为向量相似度的基准时，模型有可能将所有特征向量映射到同一点上。

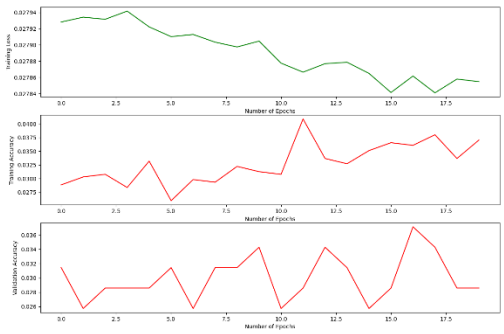
  分析：若所有特征向量都落在同一点（或者空间上十分接近），则损失函数会在 Margin 上徘徊，若 margin 设置得太小使得损失很低，则参数更新会十分缓慢，函数就会卡在此局部最小值上，也无法学习提取有用的特征。在 MyBaseModel1 的（T2）的情况中，我认为模型就是通过将几乎所有组别样本的特征向量映射至接近的位置，以达到降低损失函数的目的。（如方程所示，如果将 max(positive)尽可能降低而不管 min(negative)，同样能使损失减低）

$$Triplet\ Loss(Euclidean\ Distance) = \max(positive) - \min(negative) + margin$$

另外，通过找寻资料，我发现到 Triplet Loss 模型是较难训练的，稍有不慎模型就会将所有的特征向量映射在同一个点上，如此一来 Loss 就会停留在 margin 的值上，而模型也不会学习到任何有用的特征。可能的解决方法包括大幅降低学习率，避免函数快速收敛到 Margin，和使用更多的样本。

2. **平方差 & 均方差：**

由于平方差或者均方差通常使用在回归问题而较少使用在分类问题，因此我想通过实验观察他的具体效果。可是根据实验结果显示，对于 base_model，使用平方差损失函数训练的效果比使用 Cross Entropy 好（44%对比实验 3.1.1 中的 38%），这点我还未找到合理的解释。再者，我最后也尝试使用均方差作为损失函数，可是模型训练效果却很差，训练准确率不到 5%（见图【7】）。若与平方差相比，两者的差距只在均方差的计算中有取平均，而平方差没有，这造成了平方差的损失远大于均方差的损失。根据此思路，后者较小的损失可能使得模型更新参数的速度太慢，以至于无法有效地进行学习。



图【37】：base_model 训练数据（均方误差）

3.3.1）利用 Cross Entropy 损失函数，对 base_model 和 MyBaseModel1 添加新的数据进行训练（其他模型由于训练较耗时因此没有纳入实验部分），同时对两个超参数进行调整比较：学习率 lr， SGD 损失函数的 momentum。

1. 学习率调整

实验说明：

通过遍历的方法尝试三个不同的学习率，并将不同学习率下的模型训练情况绘制成一张图，以方便进行比较。另外，本次实验使用了实验 3.1.1 中已训练的模型，并添加新的数据（验证集 val 和附加集 addition）训练模型，而验证集和测试集则使用相同的数据集（test）。

实验设置：

Image_size: [112, 112]

Batch_size: 32

Learning_rate: [0.005，0.01，0.001]

Milestones=[15]

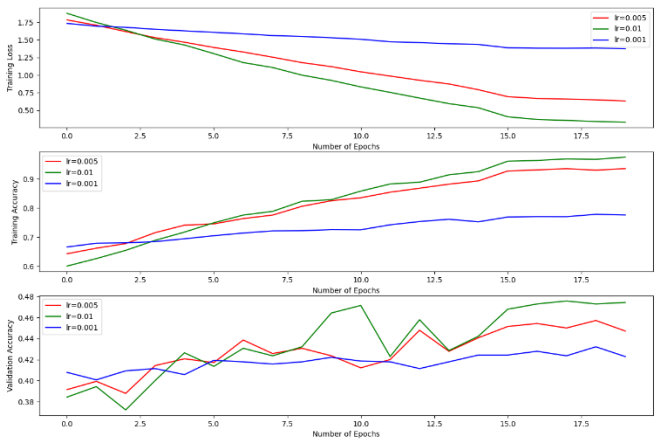
Epochs: 20

Loss_function: Cross Entropy Loss

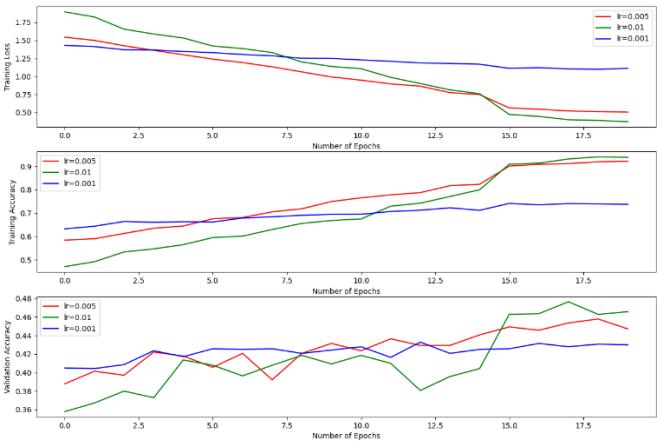
Optimizer: Stochastic Gradient Descent (SGD)

训练结果：

1. base_model
2. MyBasicBlock



图【38】： base_model 不同学习率训练数据



图【39】： MyBaseModel1 不同学习率训练数据

测试准确率：

模型	测试准确率	0.005	0.01	0.001
base_model		44.71%	47.43%	42.29%
MyBaseModel1		44.71%	46.57%	43%

- Base_model

```
test accuracy:44.71428298950195%, test accuracy:47.42857360839844%, test accuracy:42.28571319580078%,
```

- MyBaseModel1

```
test accuracy:44.71428680419922% test accuracy:46.57142639160156% test accuracy:43.0%
```

小结：

在 3 个不同的学习率中，0.01 的学习率使得两个模型性能均为最优。值得注意的是，从图中容易发现学习率为 0.001 的效果最差，收敛也是最缓慢的，很大可能是卡在了局部最小值上。因此，在添加新数据训练的过程中，不宜使用过小的学习率，通常情况下，小的学习率是用来微调模型的参数。另外，本次实验中使用的训练数据较多（增加了原本的验证集和附加集），因此可以看到模型的性能有较大的提升（从 38% 跃升至 47%），可见深度学习模型需要以较多的数据进行训练，才能够准确提取特征，并在未知的测试集上有较优的表现。

2. Momentum 调整

实验说明：

利用学习率调整实验表现最优的学习率（0.01），并选取性能较好的模型（base_model）进行训练（同样利用实验 3.1.1 中已训练的模型参数。）

实验设置：

Image_size: [112, 112]

Batch_size: 32

Learning_rate: 0.01

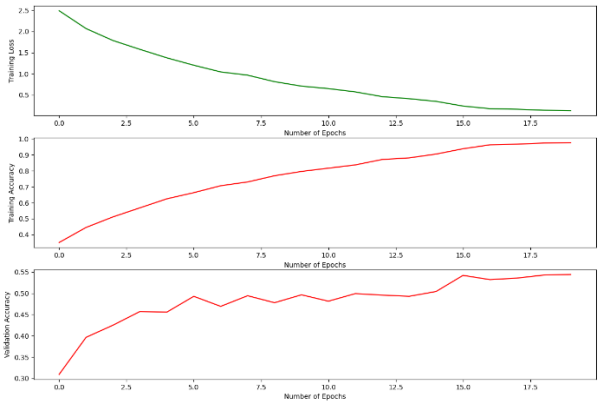
Milestones=[15]

Epochs: 20

Loss_function: Cross Entropy Loss

Optimizer: Stochastic Gradient Descent (SGD) with momentum=0.9

训练结果：



图【40】：base_model 训练数据（使用 Momentum）

测试准确率：54.43%

```
test accuracy:54.42856979370117%
```

小结：

从结果可以看到一个简单的调整（对优化算法进行改进），可以对模型最终的性能有显著的影响（47% - 54%）。在 Momentum 的加持下，模型的收敛速度更快，且能够避免参数梯度下降陷入局部最优，因此相应地能够提升模型的性能。

3.3.2）数据集的局限性分析

添加了更多数据并调整超参数进行训练后，模型的性能虽然有显著的提高，可是 3.3.1 的两个实验结果中可以看到训练准确率已经接近 100%，反而测试准确率却大约是前者的一半，这就表明了模型处于过拟合状态。当然，实验中可以根据许多方法避免过拟合，如增加正则化项以避免出现较大的权重值（过大的权重往往标识着过拟合）、降低模型的复杂性、增加 Dropout 层（随机将中间层输出变成 0）等。另一个可行的方法就是增加数据的数量，毕竟数据越多（假定错误数据的比例很低），模型的性能会越好。以 MNIST 数据集为例，其训练集中的数据数量为 60000（其测试集的为 10000），同时其对应的手写数字识别任务相对简单（只有 9 个不同的组别）。从任务难度和训练数据数量的角度相比，本次实验的数据量确实很少（即使添加了新的数据），因此训练一个性能较优的模型十分具有挑战性。

值得一提的是，在训练数据集较少的情况下，可以利用数据增广（Data Augmentation）来增加数据数量，以提升模型的性能和鲁棒性。在下一个实验中，我会用此方式来研究其对抗噪声样本的效果。

3.4.1）抗噪声样本分析

1. 添加噪声样本实验

实验说明：利用 base_model 模型对不同噪声比例的训练数据进行训练，并在同一张图中显示出性能。接下来采用下述对抗噪声的方法再次训练模型，并查看方法的有效性。

实验设置：

Image_size: [112, 112]

Batch_size: 32

Learning_rate: 0.01

Milestones=[15]

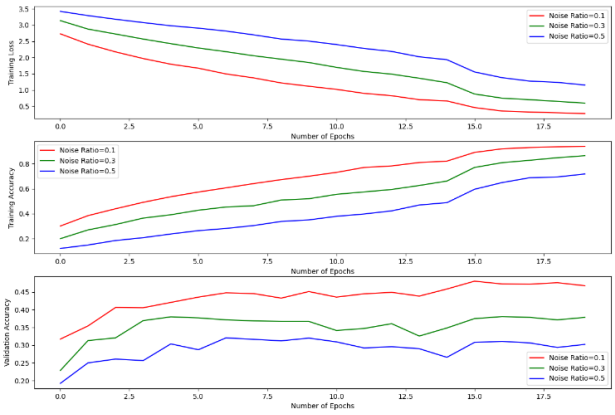
Epochs: 20

Loss_function: Cross Entropy Loss

Optimizer: Stochastic Gradient Descent (SGD) with momentum=0.9

3 种不同的噪声比例 Noise Ratio=[0.1，0.3，0.5]

训练结果：



图【41】：base_model 训练数据（不同噪声样本比例）

测试准确率：

噪声样本比例 0.1：46.79%

```
test accuracy:46.78571701049805%, Noise Ratio:0.1
```

噪声样本比例 0.3：37.86%

```
test accuracy:37.85714340209961%, Noise Ratio:0.3
```

噪声样本比例 0.5：30.21%

```
test accuracy:30.21428680419922%, Noise Ratio:0.5
```

小结：从结果易见，随着噪声样本的比例增加，模型的性能随之下降。

2. 抗噪声实验

实验说明：

本次实验的主要思路为采用一些避免过拟合的方法以对抗噪声样本。噪声样本能够使得模型性能下降是因为模型会尽可能去拟合错误的样本，因此通过一些避免过拟合的方法，可能可以将噪声样本对模型的影响降低，即模型不会过度去拟合任一数据，而是更加注重泛化程度。实验中将采取如下方法：

- 数据增强（Image Augmentation），即通过图片的各种变换如旋转，反转等改变图片以获取更多不同的数据。有了更多数据后，模型的鲁棒性也会提升。
- 正则化：在损失函数中增加 L2 正则化项，以从一定程度上缓解模型的过拟合

*注：在 Pytorch 中，利用了 transform 功能进行数据增强后， dataloader 不会直接利用变换将数据翻倍，而是根据特定概率在每一轮训练的 dataloader 中进行变换，因此在实验中我也提高了训练周期 Epoch，毕竟在此情况下训练周期的增加就等同于模型获得翻倍的数据了。

实验设置：

Image_size: [112, 112]

Batch_size: 32

Learning_rate: 0.01

Milestones=[15, 25]

Epochs: 30

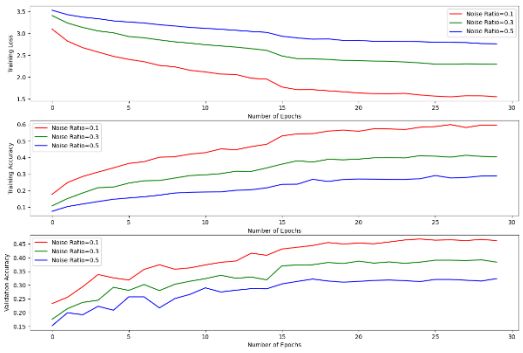
Loss_function: Cross Entropy Loss

Optimizer: Stochastic Gradient Descent (SGD) with momentum=0.9

Weight Decay: 0.001

猜想：此以避免过拟合为基础的抗噪声样本方法在噪声比例较低时会较有效，随着噪声比例的增加效果越差。这是因为若噪声比例过大，从模型的角度来看，正确标签的样本才是噪声，因此方法会失效。

训练结果



图【42】：base_model 训练数据（对抗噪声）

测试准确率：

噪声样本比例 0.1：46.21%

```
test accuracy:46.21428680419922%, Noise Ratio:0.1
```

噪声样本比例 0.3：38.36%

```
test accuracy:38.35714340209961%, Noise Ratio:0.3
```

噪声样本比例 0.5：32.26%

```
test accuracy:32.357139587402344%, Noise Ratio:0.5
```

小结：

从训练和测试结果来看，实验 3.4.1 中提出的对抗噪声方法不论在任何噪声比例都基本没有什么效果，因此实验猜想是不成立的。当然，造成实验不成功的原因除了方法原本就无效之外，两个实验中使用的训练样本不同（因为噪声样本的产生和 PyTorch 的 Transform 是随机的）也使实验结果并不那么可信和准确。但是，一个好的对抗噪声数据的手段在面对任意噪声时都应该展现优质的表现，因此我更倾向于上述提出的方法是没有效果或者效果低微的。

除了实验所提出的方法之外，习题课中提出的方法。另外，我有另一个想法就是使用性能优秀的预训练网络（如实验 3.1.1 中使用的 Resnet18）对噪声样本进行分类，并从中挑选对损失函数（Cross Entropy）“贡献”最大的样本（已知 Resnet 准确率很高，若损失突然有显著的提高，则可以合理怀疑有噪声样本。同时根据 Softmax 的性质，错误很有可能被放大，进一步提升此方案的可行性）。完成数据的清洗后，再将训练样本投入模型进行训练检查训练和测试结果即可。由于时间不足，在此就不展开测试上述方案。

最后，值得一提的是，利用性能好的网络进行数据清洗毫无疑问是可行的，但是假设一开始并没有高性能的模型，要如何有效进行数据清洗呢（除了人工清洗）？此问题值得深入思考。

实验总结和感想：

此次实验涉及许多不同任务，总体而言还是挺耗时的。深度学习领域的入门在现有的框架下（Tensorflow, PyTorch）门槛还是比较低的，进行基本的体验难度不会太大。如以 TensorFlow 为后台运作的 Keras，即使对深度学习理解有限，也较容易上手训练一个神经网络模型。另一方面，PyTorch 相对来说会较难，但伸缩性（Flexibility）也较好，值得一提的是，使用 PyTorch 进行实验能够加深我对深度学习的理解，毕竟许多部分需要有相应的知识才能进行实现。现在，经过了一连串的实验后，我也算是初步入门了深度学习，可是，接下来的学习曲线会较陡峭，需要掌握大量的数学知识才能够有显著的进步。就如损失函数部分的分析，在数学知识不充足的情况下想要给出合理的分析是十分困难的。毕竟若想要完全了解每个损失函数，必须研究其背后的数学依据（涉及许多概率论相关的知识）另外，也需要投入大量的时间阅读前沿的论文，以进一步扩大视野。综上所述，本次实验令我获益良多，除了提升编程技术之外，也让我了解研究深度学习的基本流程，并让我的未来职业方向多了一个具有吸引力的选择。遗憾的是，由于我是大二的学生（大三春季打算去交换，所以只能提早选了这门课），受限于时间（大二下专业课特别多）和能力，我并未能将实验完成得很好，若有任何疏漏和错误之处，还请老师和助教多多指正。希望来年有机会能再次选修到相关的课程进行更深入的学习。