深度学习导论 HW4

PB22151796 莫环欣

定义

数据增强与读取

由于助教已经提供了数据加载器的文件，本次实验中直接导入文件并使用其中方法，改动之处在于新增了类别名称并显式指定了导出名单（仅服务于类别名数组，其余方法本身就可导出）

```
__all__ = [
    "classes",
    "load_cifar10_subset",
    "SimCLRDatasetWrapper",
    "get_augmentations",
]

classes = [
    "plane",
    "car",
    "bird",
    "cat",
    "deer",
    "dog",
    "frog",
    "horse",
    "ship",
    "truck",
]
```

并且补全了剩下的两种数据增强措施

> 其实可以引入高斯噪声、随机裁剪和自动增强 AutoAugment(policy=AutoAugmentPolicy.CIFAR10)（在 torchvision.transforms 里），我之前在 CNN 中使用这两种增强结合其它措施取得了很不错的效果（0.9069）

```
transform_list = [
    # 1. 随机调整大小并裁剪到32x32
    transforms.RandomResizedCrop(size=32),
    # 2. 以0.5的概率水平翻转图像
    transforms.RandomHorizontalFlip(p=0.5),
    # 3. 以0.8的概率应用颜色抖动  亮度、对比度、饱和度和色调
    transforms.RandomApply(
        [
            transforms.ColorJitter(
                brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1
            )
        ],
        p=0.8,
    ),
    # 4. 以0.2的概率转换为灰度图
    transforms.RandomGrayscale(p=0.2),
    transforms.ToTensor(),
]
```

之后在需要时直接调用即可获得数据集

```
train_subset, test_subset = dataloader.load_cifar10_subset(
    "./data", subset_classes=10, train_percent=0.1
)
# 划分验证集
valid_size = int(0.2 * len(train_subset))
train_size = len(train_subset) - valid_size
train_subset, valid_subset = torch.utils.data.random_split(
    train_subset, [train_size, valid_size]
)
```

当然，针对于自监督训练阶段，还需要经过一层专门的包装

```
train_subset = dataloader.SimCLRDatasetWrapper(train_subset)
valid_subset = dataloader.SimCLRDatasetWrapper(valid_subset)
```

模型定义

按文档里的说法，对接入的基础编码器去除最后的分类层，并接上一个`MLP`作为投影头，这里默认是带`BatchNorm`的实现

```python
class SimCLRModel(nn.Module):
    def __init__(self, base_encoder, projection_dim=128):   # 128是推荐大小，太大可能引入噪声，太小会导致表达能力不足
        super(SimCLRModel, self).__init__()

        # 基础编码器（去掉分类层） <-- 提取输入图像的高位特征表示
        self.encoder = nn.Sequential(*list(base_encoder.children())[:-1])

        # 投影头（带BN的MLP） <-- 整合起来，后续分析中可以直接替换
        self.projection_head = nn.Sequential(
            nn.Linear(512, 512),    # 保持特征表达能力的同时作变换
            nn.BatchNorm1d(512),
            nn.ReLU(inplace=True),
            nn.Linear(512, projection_dim),
        )

    def forward(self, x):
        x = self.encoder(x)
        x = x.view(x.size(0), -1)
        x = self.projection_head(x)
        return x
```

后续在自监督学习阶段直接调用`resnet18`作为模型骨架

```python
base_encoder = models.resnet18(weights=models.ResNet18_Weights.IMAGENET1K_V1)
```

并且支持对投影头进行修改或拆卸

```python
if analysis_type == "projection_head":
    if param_value == "without_bn":
        # 不带 BatchNorm
        model.projection_head = nn.Sequential(
            nn.Linear(512, 512), nn.ReLU(inplace=True), nn.Linear(512, 128)
        )
    elif param_value == "none":
        model.projection_head = nn.Identity()   # 恒等映射，拿来占位
        model.forward = lambda x: model.encoder(x).view(x.size(0), -1)
```

到了微调阶段，在加载完模型结构之后需要将投影头更换为一层线性层作为分类头，并按文档要求冻结编码器部分的权重以求更好地观察自监督学习阶段权重产生的影响（否则上面也更新那前面的影响就看得不明显了）

```python
# 加载预训练模型
base_encoder = models.resnet18(weights=None)
model = SimCLRModel(base_encoder)

# 替换投影头为分类头  <-- 这里需要先替换结构再加载模型，不然加载权重时会报错
model.projection_head = nn.Linear(512, 10)  # 分类头

# 加载权重时只加载编码器部分
pretrained_weights = torch.load(model_name, map_location=device)
model.encoder.load_state_dict(
    {k: v for k, v in pretrained_weights.items() if k.startswith("encoder.")},
    strict=False,
)

# 冻结编码器
for param in model.encoder.parameters():
    param.requires_grad = False
```

对比损失函数

按照文档中的公式，将两个样本拼接后作出相似矩阵（并屏蔽对角元素，自己肯定跟自己完全相似），并除以温度系数τ，之后再计算交叉熵取平均即可

两个正样本的互相定位也好理解，生成一个`0 ~ N-1`的数组（这里定位到的就是`z_i`），再加上批次大小就可以定位到`z_j`，后面计算交叉熵的时候传入对立的正样本定位即可

```python
class SimCLRLoss(nn.Module):
    def __init__(self, temperature=0.5):
        super(SimCLRLoss, self).__init__()
        self.temperature = temperature

    def forward(self, z_i, z_j):
        batch_size = z_i.size(0)
        representations = torch.cat([z_i, z_j], dim=0)  # (2*batch_size, projection_dim)

        similarity_matrix = F.cosine_similarity(
            representations.unsqueeze(1), representations.unsqueeze(0), dim=2
        )  # (2N, 2N)
        # 得到 \tau
        similarity_matrix = similarity_matrix / self.temperature  # 除以温度系数

        # 让z_i与z_j相互找到
        labels_j = torch.arange(batch_size, dtype=torch.long, device=device)
        labels_i = (labels_j + batch_size) % (2 * batch_size)

        # 屏蔽掉对角线（样本自身与自身）
        mask = torch.eye(2 * batch_size, dtype=torch.bool, device=device)
        similarity_matrix = similarity_matrix.masked_fill(mask, -np.inf)

        # 计算平均对比损失
        loss_i = F.cross_entropy(similarity_matrix[:batch_size], labels_i)
        loss_j = F.cross_entropy(similarity_matrix[batch_size:], labels_j)
        loss = (loss_i + loss_j) / 2
        return loss
```

训练器定义

这里还是可以使用祖传训练逻辑，整体基本不需要改动，仅需针对自监督训练阶段加上特判：

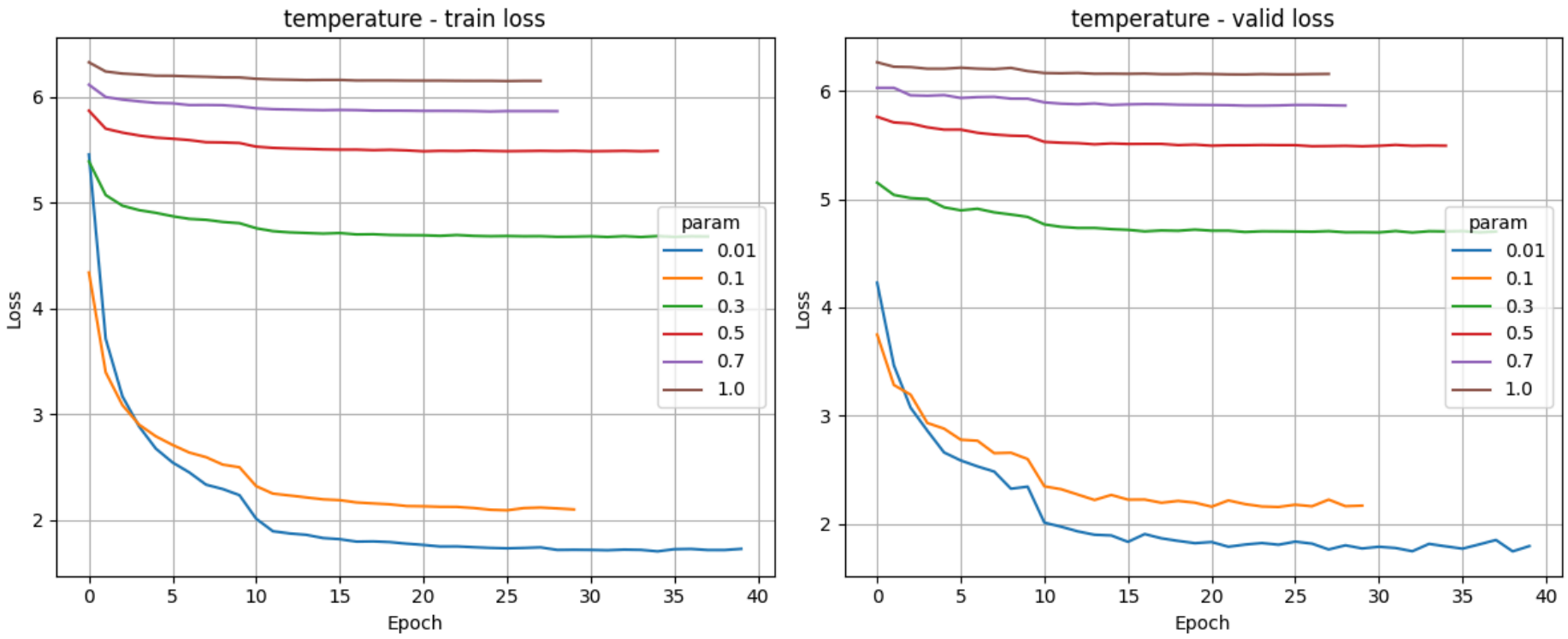
- 损失函数更换为对比损失函数
  - 前向传播时也分别传两个值（两个正样本）
- 观测指标由准确率改为损失
  - 保存逻辑与早停逻辑

训练与分析

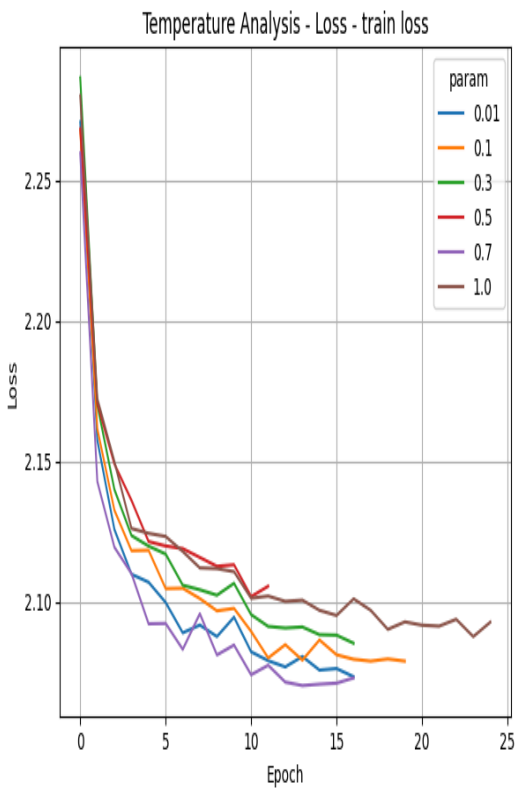
实验中超参数的影响

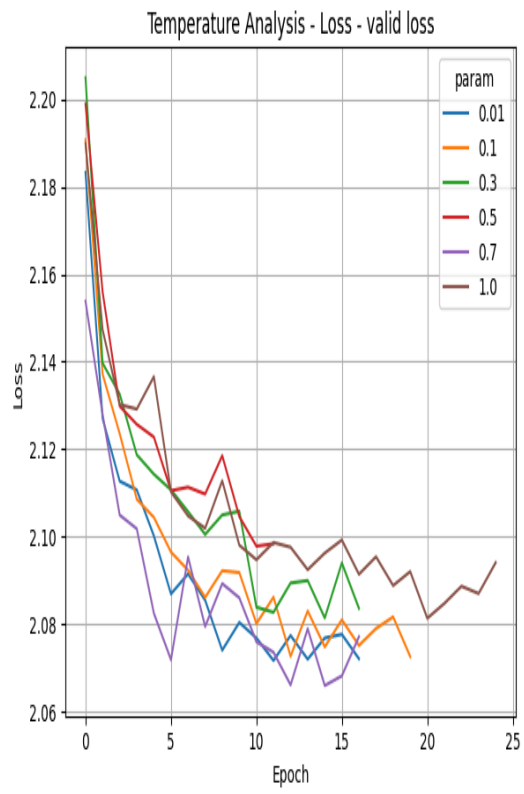
温度系数

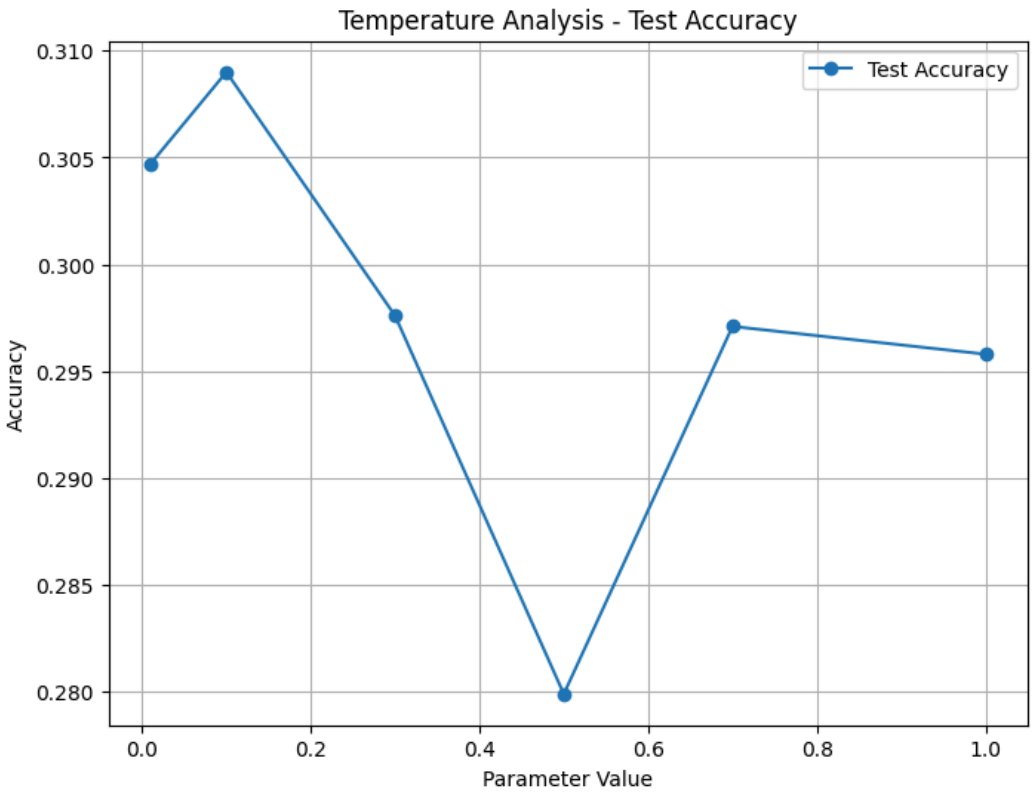
温度系数与自监督对比学习阶段正负样本分布情况有关（具体来说是`softmax`的分布平滑程度），温度系数越小（one-hot），模型对于正负样本的区分就越强，在相同`batch_size`情况下这种区分能力就体现在训练-验证损失的下降速度上，但可能太关注局部特征而忽略全局特征，泛化能力也会变差（模型对区分太严格了，导致对负样本的学习不够（负样本间的相似性会被放大），这些都会体现在微调情况上）；温度系数增大之后（更平滑）虽然区分能力减弱，但学习到的特征表示更为广泛，泛化能力会有提升



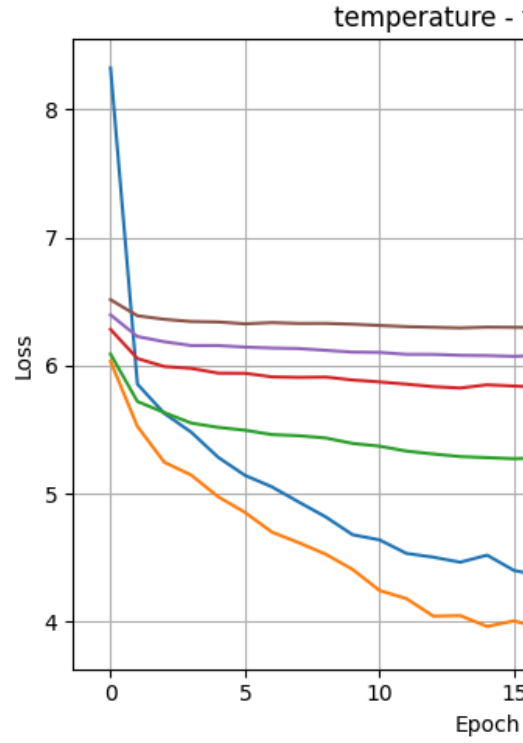
这里准确率较低是因为我们仅微调了一层线性层（编码器被冻结），并且由于资源限制取的点数相对还是比较少，对真实变化曲线的拟合效果还是不够好，偶尔会出现一些极端值

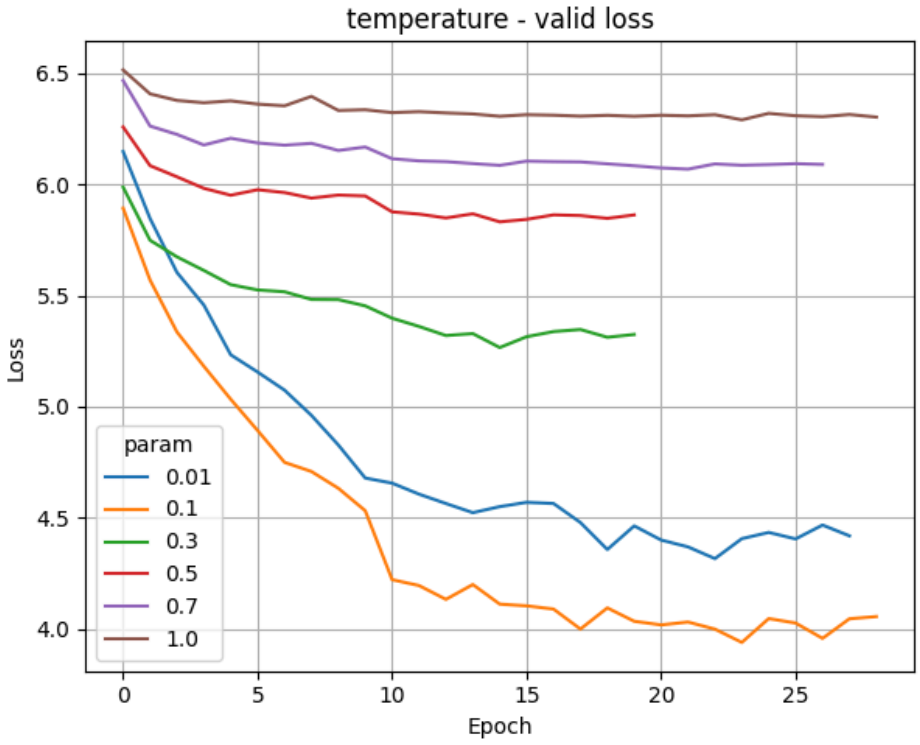


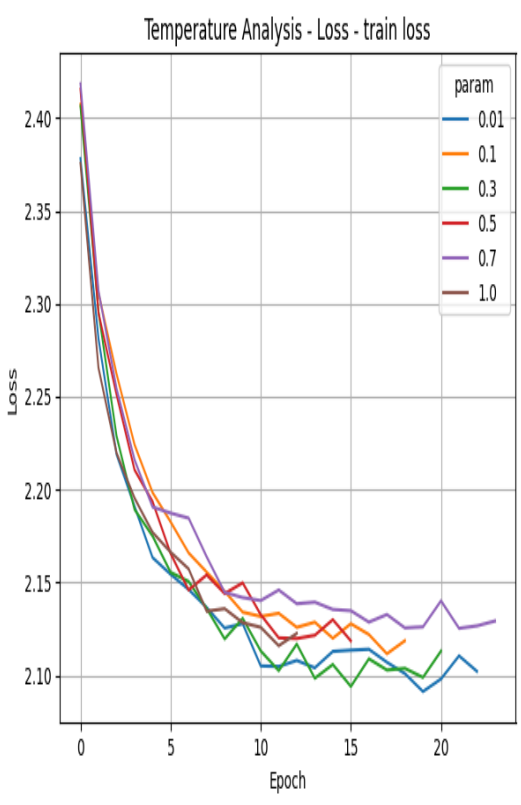


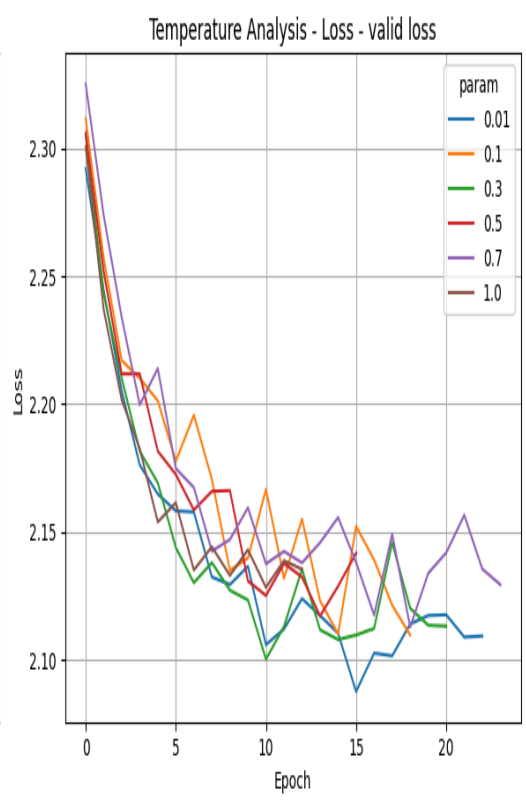


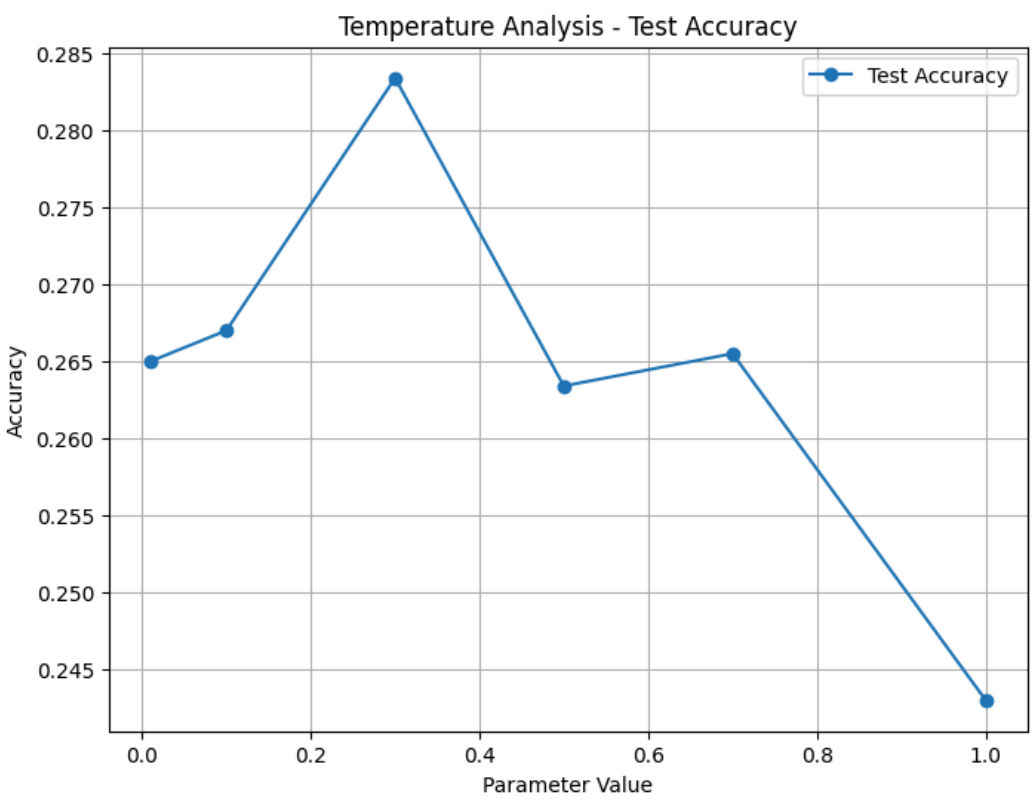
当然，如果移除预训练`ResNet18`的权重，上面的描述会看得更为准确，温度系数越小损失下降的速度也越快；并且在微调阶段测试集上的表现也更符合理论表述（没有受到预训练权重中倾向的影响），这里为快速测试使用了`0.2`倍数据集











批量大小

如果用全量数据集，小`batch_size`情况下实在太慢了（下图左侧，第一组四轮就跑了十个小时），因此还是改用小一点的数据集，只取`0.2`

```
batch_sizes = [32, 128, 256, 512]
bs_results = train_truly("batch_size", batch_sizes)
plot_comparison(bs_results, "batch_size", batch_sizes, title_prefix="batch_size")
```

[8] 656m 23.5s

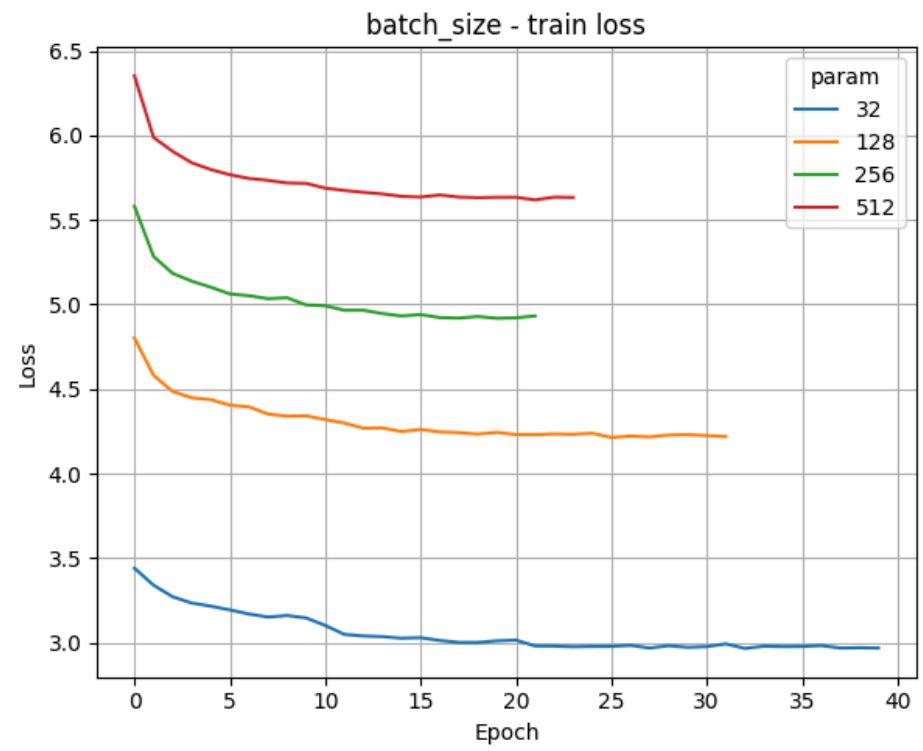
```
Testing: batch_size param: 32 ===
Epoch 1/40
Train Loss: 3.205950, Train Acc: 0.000000 | Val Loss: 3.136598, Val Acc: 0.000000
/home/ubuntu/.local/lib/python3.10/site-packages/torch/utils/data/dataloader.py:624: User
  warnings.warn(
Epoch 2/40
Train Loss: 3.055868, Train Acc: 0.000000 | Val Loss: 3.093686, Val Acc: 0.000000
Epoch 3/40
Train Loss: 3.016236, Train Acc: 0.000000 | Val Loss: 3.026747, Val Acc: 0.000000
Epoch 4/40
Train Loss: 2.979332, Train Acc: 0.000000 | Val Loss: 3.020425, Val Acc: 0.000000
```

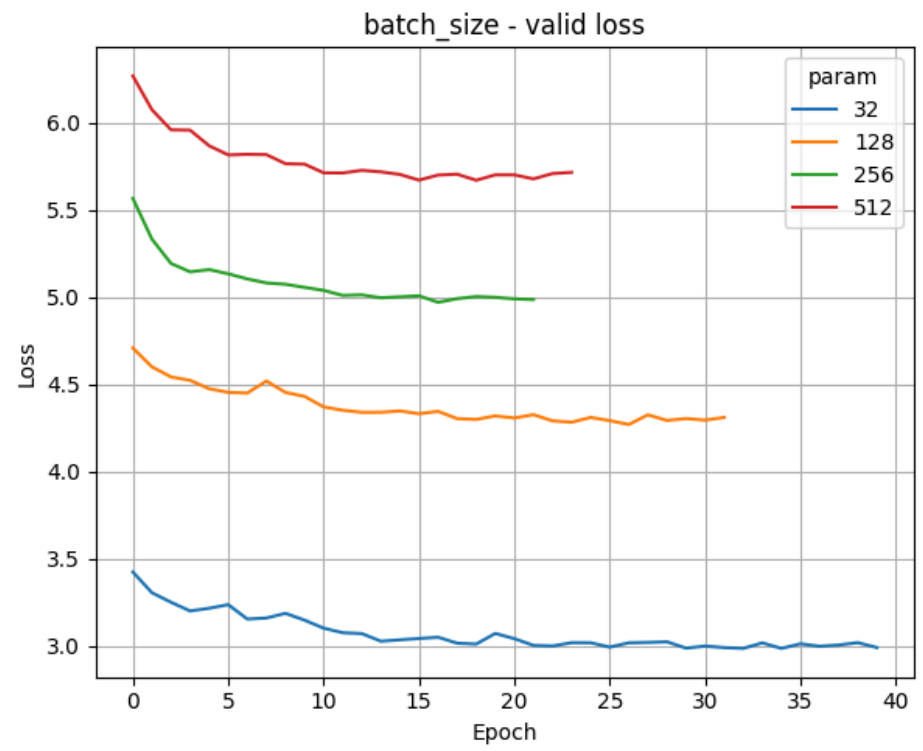
```
temperatures = [0.01, 0.1, 0.3, 0.5, 0.7, 1.0]
tp_results = train_truly("temperature", temperatures)
plot_comparison(tp_results, "temperature", temperatures, title_prefix="temperature")
```

[7] ✓ 1936m 28.5s

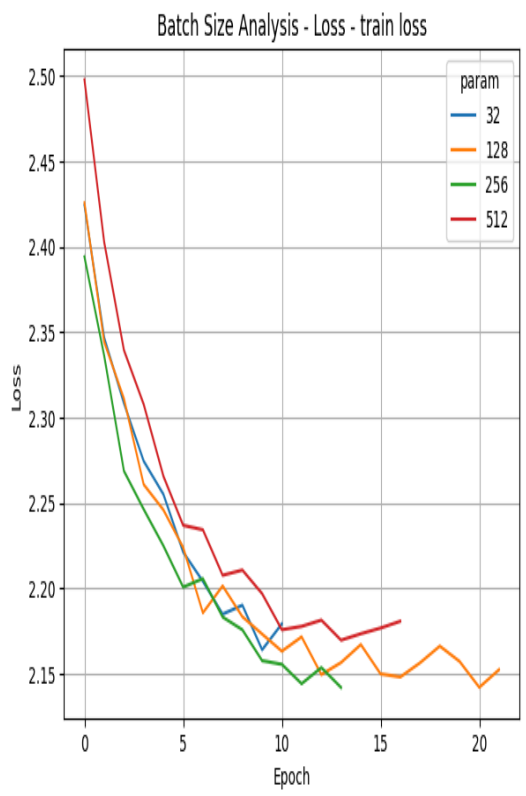
```
Testing: temperature param: 0.01 ===
/home/ubuntu/.local/lib/python3.10/site-packages/torch/amp/grad_scaler.py:132: UserWarning
  warnings.warn(
Epoch 1/40
Train Loss: 5.456578, Train Acc: 0.000000 | Val Loss: 4.227163, Val Acc: 0.000000
Epoch 2/40
Train Loss: 3.713610, Train Acc: 0.000000 | Val Loss: 3.464168, Val Acc: 0.000000
Epoch 3/40
Train Loss: 3.169740, Train Acc: 0.000000 | Val Loss: 3.072734, Val Acc: 0.000000
Epoch 4/40
Train Loss: 2.884534, Train Acc: 0.000000 | Val Loss: 2.860430, Val Acc: 0.000000
Epoch 5/40
Train Loss: 2.675055, Train Acc: 0.000000 | Val Loss: 2.658999, Val Acc: 0.000000
Epoch 6/40
Train Loss: 2.544094, Train Acc: 0.000000 | Val Loss: 2.584806, Val Acc: 0.000000
```

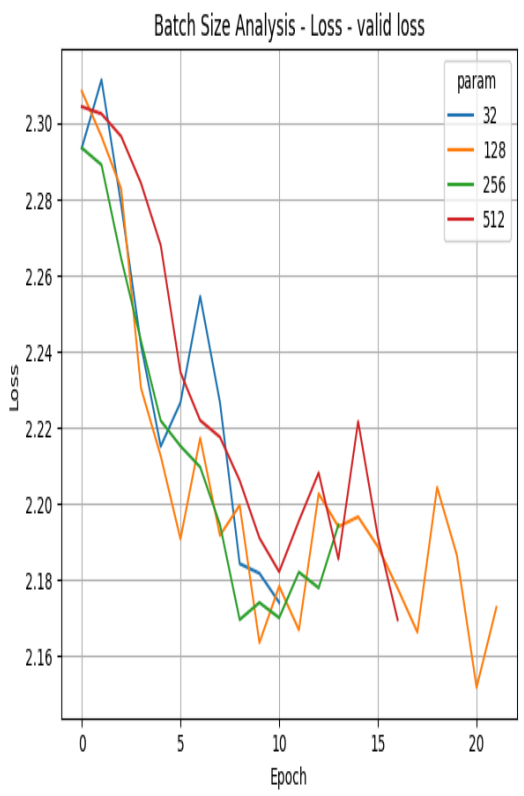
`batch_size` 与正负样本比例有关（当前一个视图只有一个正样本，增大 `batch_size` 引入的全部都是负样本），那么小 `batch_size` 可以让模型更新次数变多，对局部特征的把握能力更好，但是由于负样本很少，对正负样本的区分能力相对会较弱一些；大 `batch_size` 由于引入了更多的负样本，模型在对比学习过程中区分正负样本的能力会更强一些，但由于样本太多，表现在整体上就会导致损失下降更缓慢

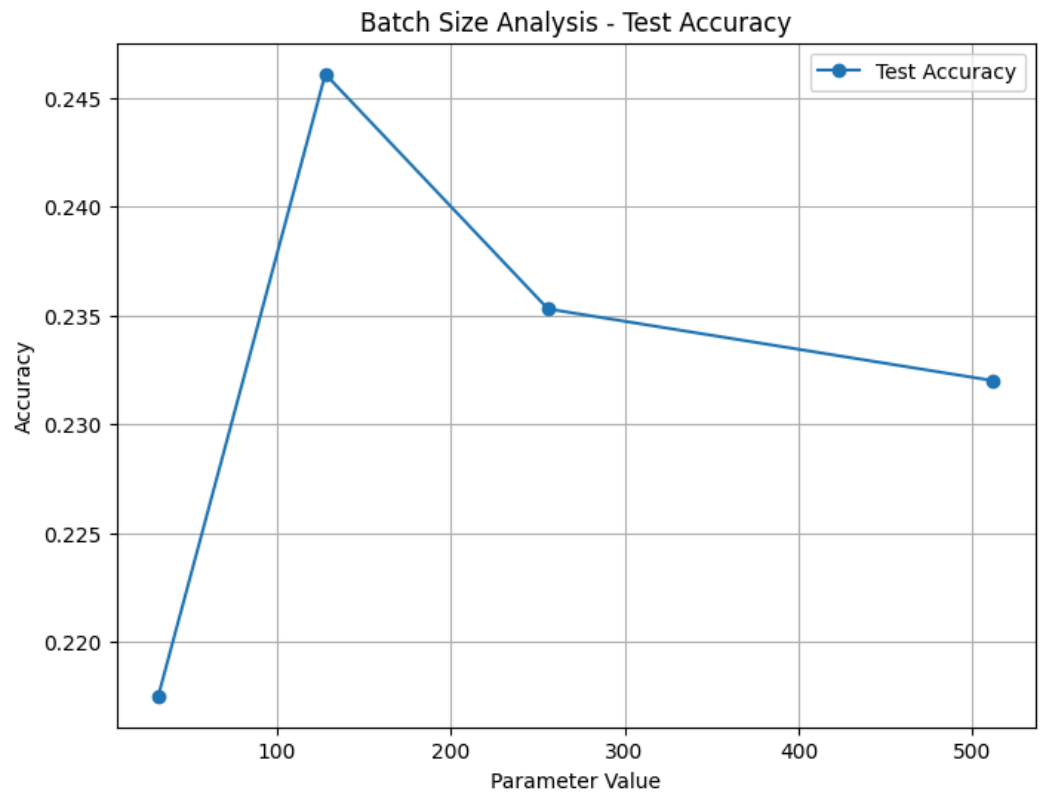




到了微调阶段，小 `batch_size` 由于在对比学习阶段区分正负样本的能力就不是很足，此时的泛化能力也会受到影响（比如自监督学习阶段可能陷入了局部最优解）；而大 `batch_size` 虽然区分正负样本的能力更强，但它太依赖于全局特征，对于图像具体细节特征的关注度可能就更弱一些，同样也就影响了微调分类





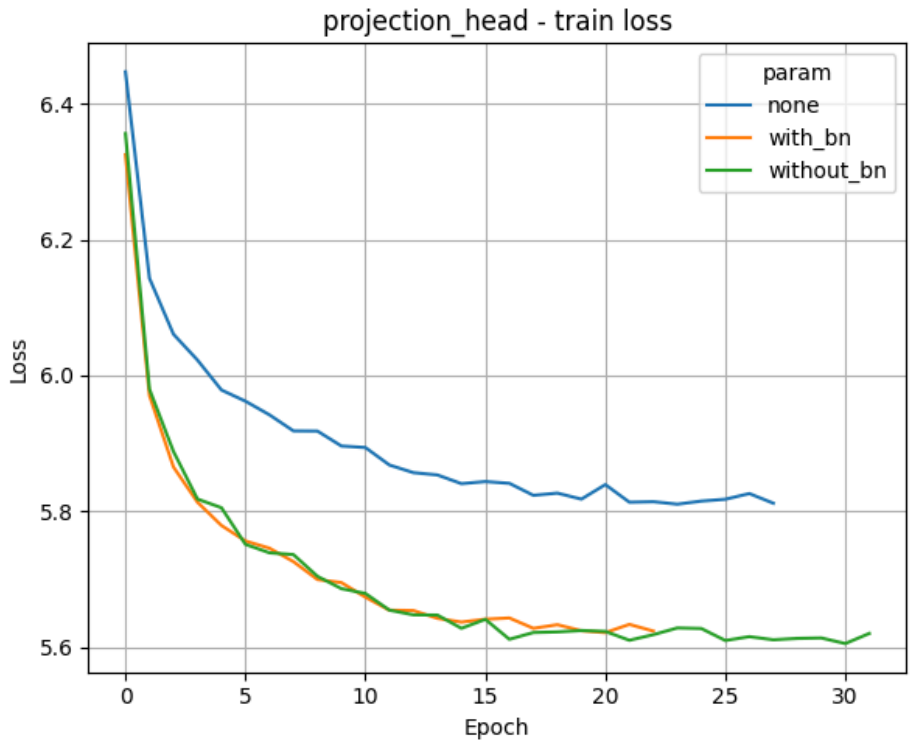


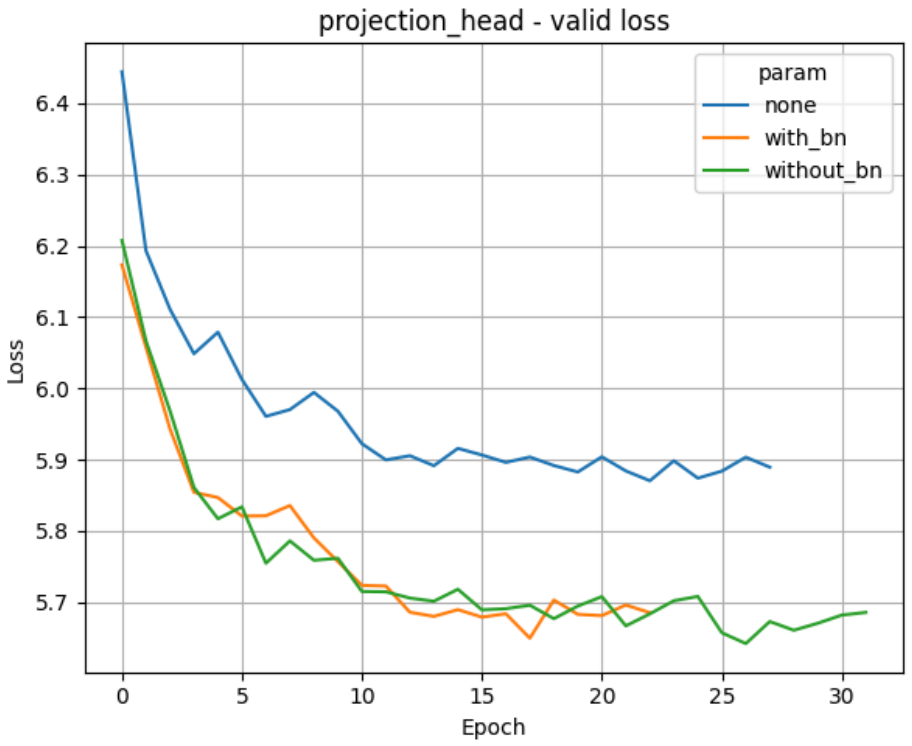
Projection head 结构的影响

投影头将编码器提取的特征映射到应用对比损失的空间，这里是为了让正样本在特征空间中更近、负样本更远

当没有投影头时，直接使用基础编码器的输出，特征表示可能因为没有经过合适的线性/非线性变换而不够有效（搭建模型骨架时移除了最后的整个分类层），无法很好地区分正负样本对，从而导致损失较高

`BatchNorm` 和非 `BatchNorm` 的投影头在对比学习阶段的表现几乎没有区别，因为对比学习主要依赖于特征的方向（通过余弦相似性计算），而 `BatchNorm` 主要影响特征的尺度和分布（让余弦相似性的计算更加稳定）





在微调阶段，基础编码器被冻结且投影头被替换为单层线性层分类头，性能主要取决于编码器在对比学习阶段学习到的特征表示质量

这里无 `BatchNorm` 的投影头性能比较高，可能是因为其在对比学习阶段没有进行特征归一化，使得编码器学习到的特征表示保留了更多更适合分类任务的原始信息，于是在微调阶段分类头就能够直接利用这些特征进行分类