表情识别上机编程作业报告

2020010768 无 05 付宇辉

一、 运行基准模型

代码结构

```
├─files
│       dataset.py     自定义数据集，用于加载数据
│       main.py        训练，验证，测试模型
│       model.py       定义模型结构
│       utils.py       一些功能性代码
│       predict.py     模型最终的性能测试
```

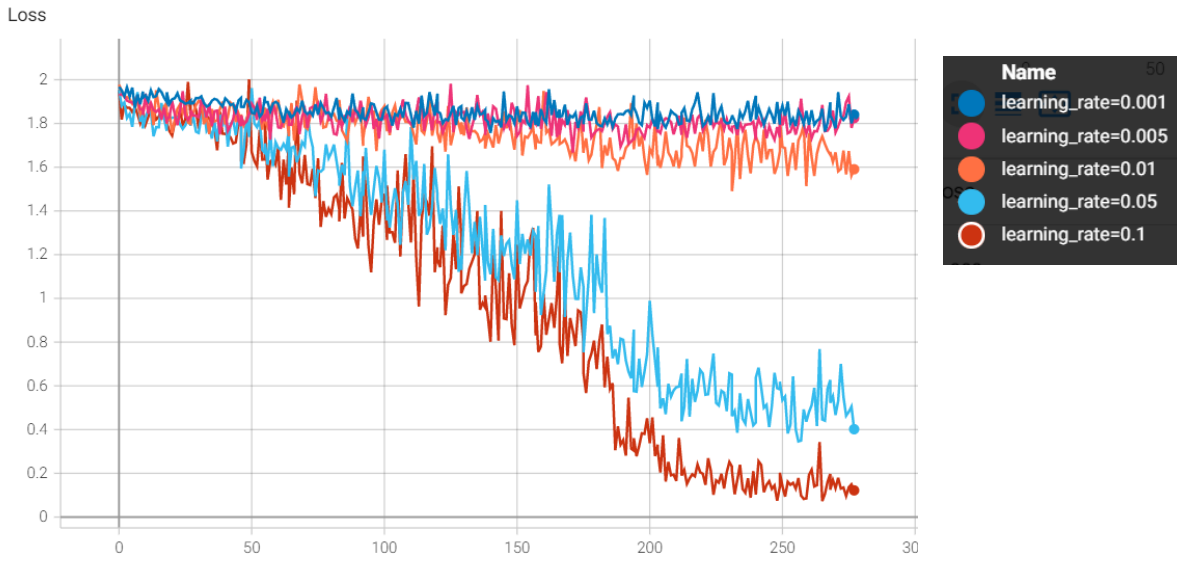
在全部使用默认参数的情况下的输出结果为：

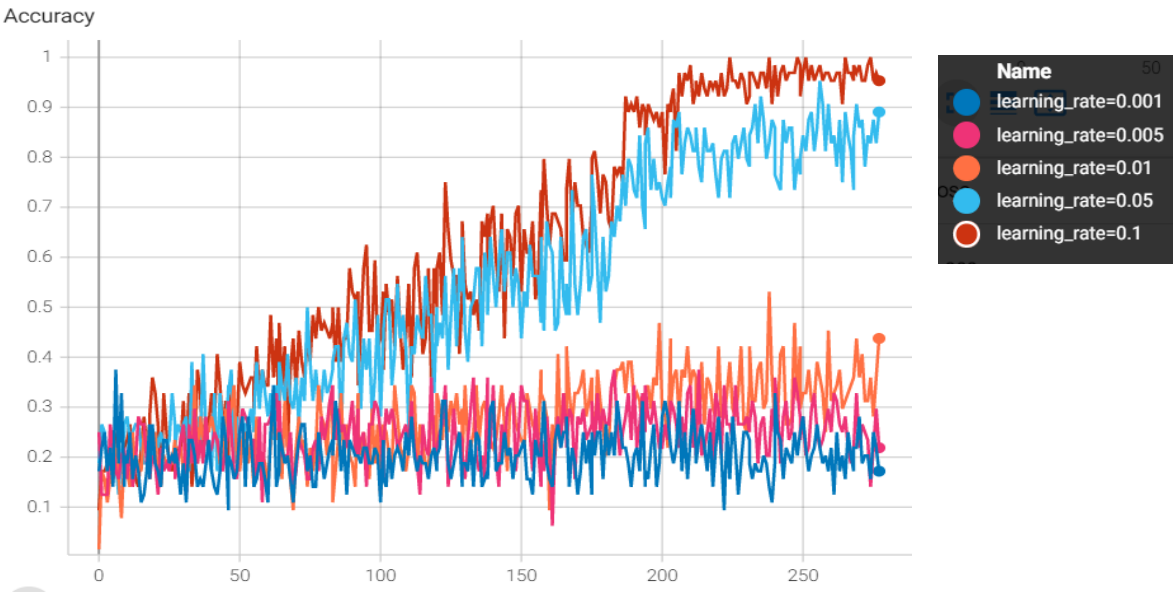
val accuracy:28.952381134033203%
test accuracy:32.21757125854492%

二、 实验结果分析

分析基础代码的结果（图是由 torch 中 tensorboard 包画的，中期报告的代码丢失，所以现在代码中并没有一开始的作图代码，之后的图全部是 matplotlib 画的）

训练过程中 loss 和准确率的变化，以及 learning_rate 对其的影响如下图所示





样本的分布特征

用 PCA 对样本进行降维（代码位于 utils.py 中）

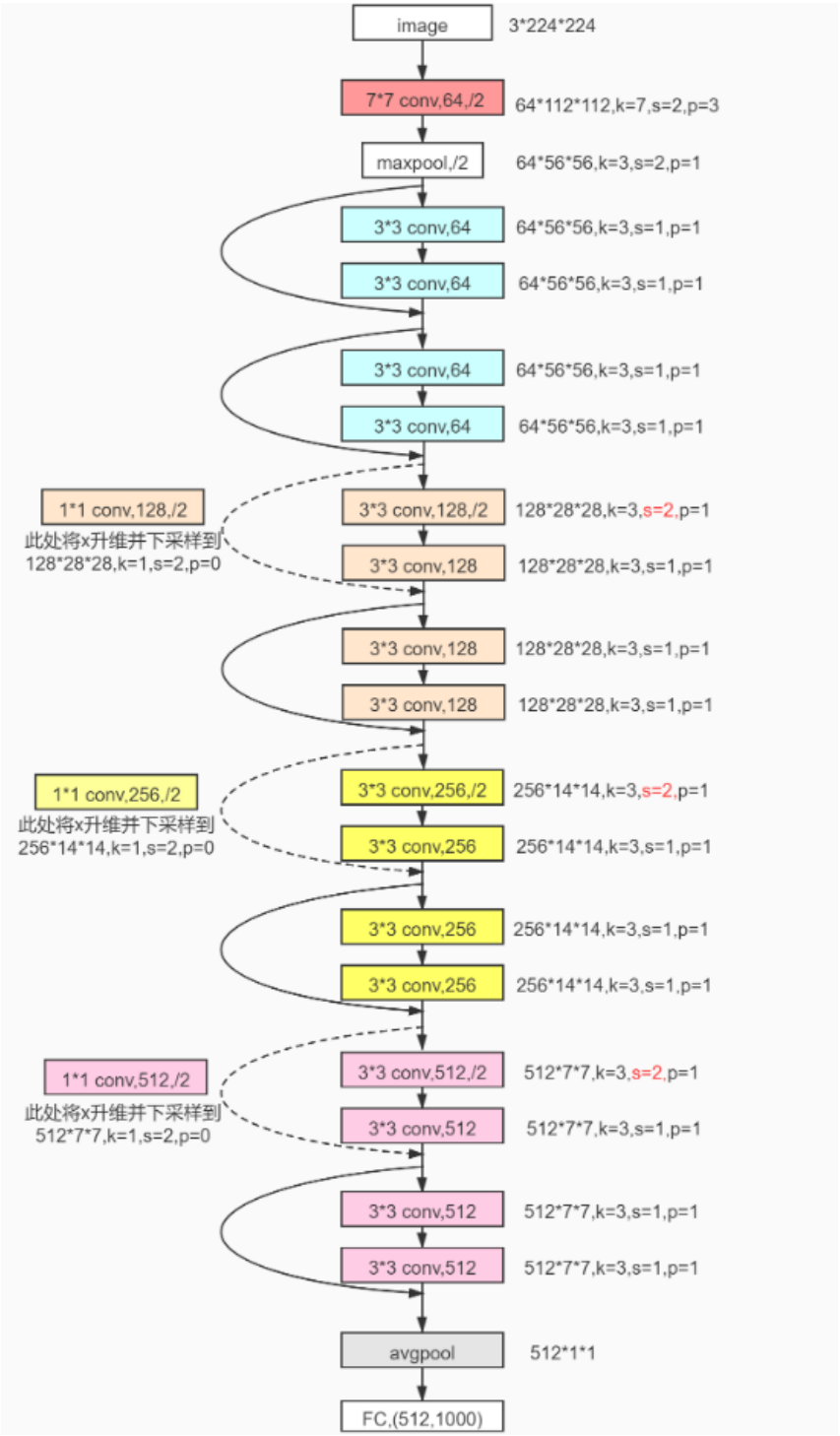
```
def PCA(X: torch.Tensor, k: int):
    """
    X: data
    k: dims which you want
    """
    X = X.cpu().numpy()
    n_samples, n_features = X.shape
    mean = np.array([np.mean(X[:, i]) for i in range(n_features)])
    norm_X = X - mean
    scatter_matrix = np.dot(np.transpose(norm_X), norm_X)
    eig_val, eig_vec = np.linalg.eig(scatter_matrix)
    eig_pairs = [(np.abs(eig_val[i]), eig_vec[:, i]) for i in
range(n_features)]
    eig_pairs.sort(reverse=True, key=lambda ls: ls[0])
    feature = np.array([elem[1] for elem in eig_pairs[:k]])
    data = np.dot(norm_X, np.transpose(feature))
    return data


def PCA_draw(data, label):
    features = PCA(data, 2)
    label = list(label.cpu().numpy())
    plt.scatter(features[:, 0], features[:, 1], c=label, cmap='rainbow')
```

baseline 中样本特征分析的结果已经在**中期报告**中有所呈现，此处不再展示。

三、 模型的改善

1. 使用 Resnet18

由于训练集过小，为了避免过拟合等问题，我们选择参数较小和层数较浅的 Resnet18 主干网络进行改进。Resnet18 的结构如下。



模型的构建在 model.py 当中

前向传播的部分代码如下：

```
    def _forward_impl(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        self.feature = x
        x = self.fc(x)

        return x

    def forward(self, x):
        return self._forward_impl(x)
```

其余环境都不变，只是将模型换为 Resnet18，参数设置为：learning_rate 0.1 epochs 30, batch_size 64

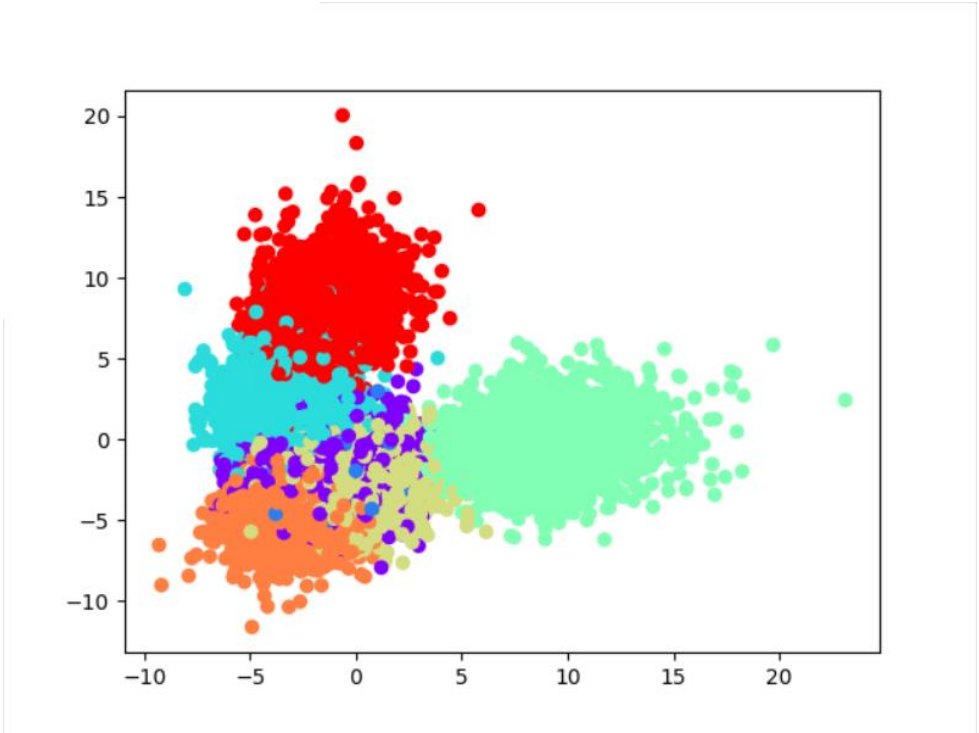
训练结果：

==========

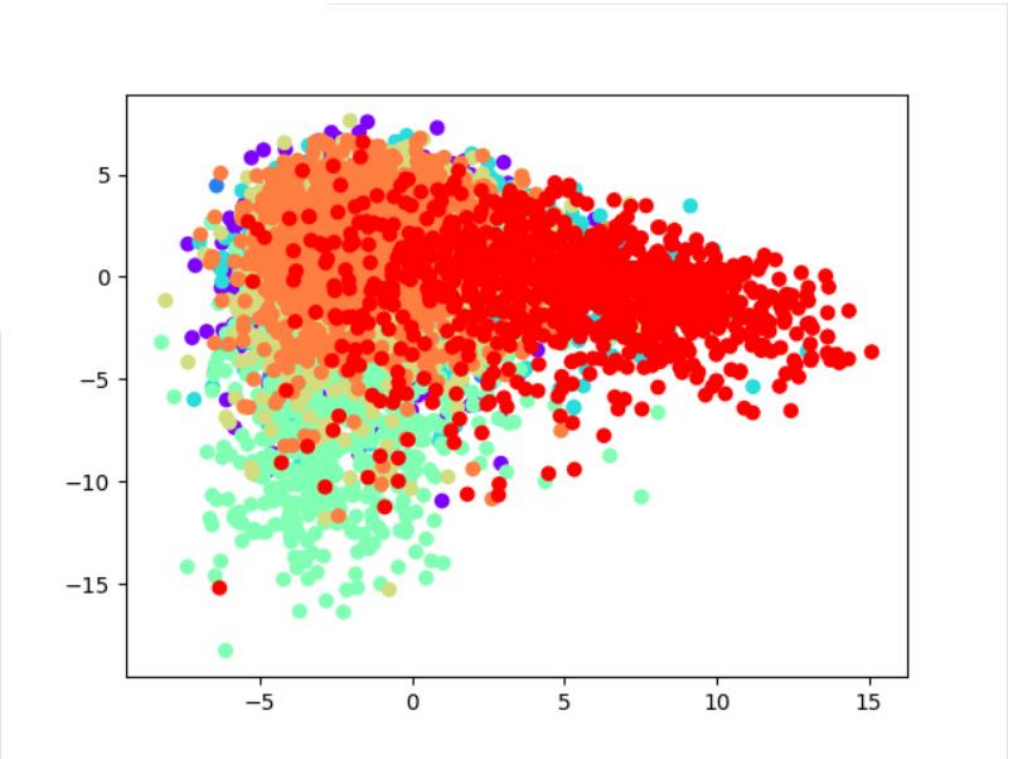
val accuracy:48.72108843537415%

test accuracy:51.89679218967922%

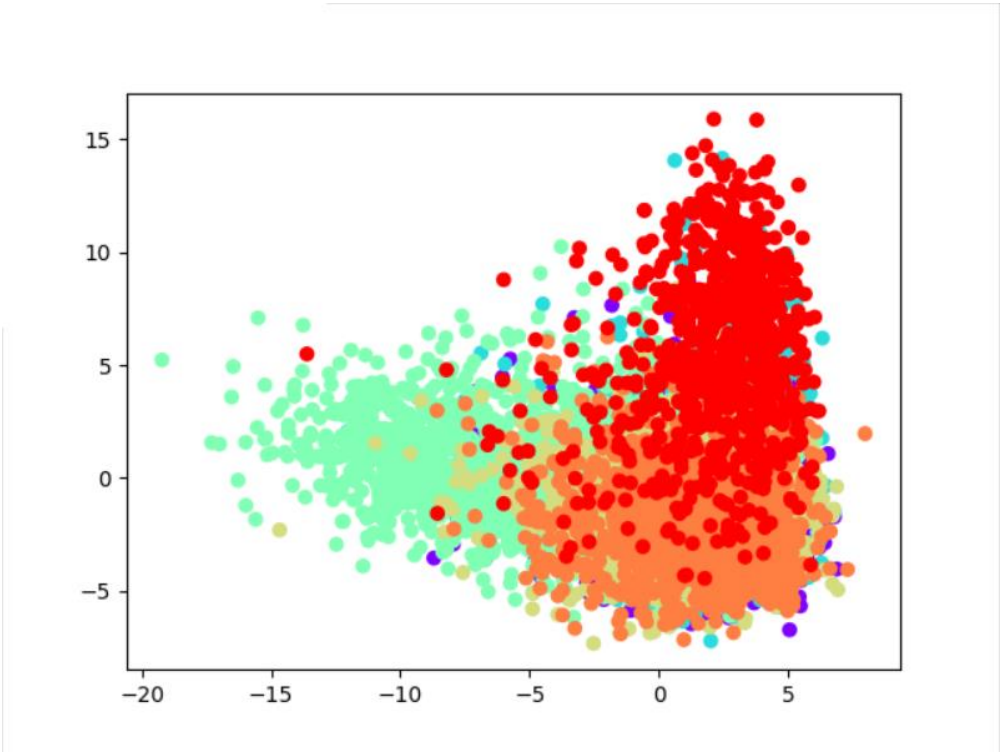
在 learning_rate 0.1 batch_size 64 epochs 30 的情况下，对训练集提取样本特征(即上述代码中的 self.feature)，同样使用的是 PCA 方法，将 feature 压缩为二维向量，不同颜色代表不同的类别。由于此时，在训练集上的准确率已经达到了 99%左右，对训练集拟合的已经是非常好了，所以提取的特征应该是近似线性可分的。可以看到训练后的模型提取的训练集特征非常明显的划分开来，之后的全连接层很好的能够将不同类别识别出来。(可能图上不好看出有七种类别，这是由于训练集各个类别的图片个数差距还是有点的，并且各个散点可能出现覆盖情况，导致部分类别没办法很好的展示在图中)



对验证集提取样本特征如下，我们可以看到，特征在训练集上并没有很好的分开，导致最后的分类准确率不是特别高。



对测试集提取样本特征如下，与验证集一样，提取的特征并没有很好的分开。



综上，模型对验证集和测试集的提取的特征并没有将不同类别的样本分开，所以正确率只有 50%左右。

2. 增加动量和正则化

我们知道，SGD 可以通过增加动量使得模型较好地避免收敛到局部最值的情况，通过增加惩罚项可以较好地减少 overfitting 的效果。

设置优化器为 SGD，参数变为 learning_rate 0.1　weight_decay 1e-4　momentum 0.9，其余超参数均为默认且保持不变。

果然和我们料想的差不多，验证集和测试集的准确率都相较于不使用动量和正则化的情况要高，至此我们又提高了测试集准确率 3%左右。

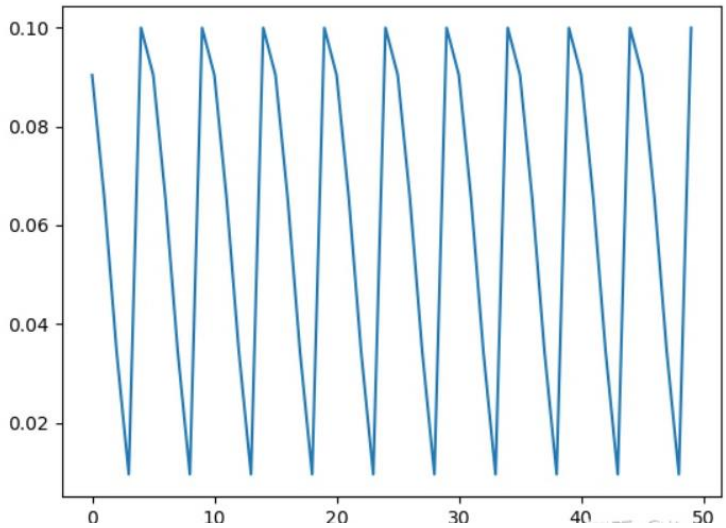
```
============================
val accuracy:50.87074829931972%
test accuracy:54.03068340306834%
```

3. 换用学习率策略为余弦退火

我发现原本的学习率策略并没有很好的考虑到走出局部极值后，由于学习率过小，没办法再次收敛到其他的极值或者最值，所以我换用了余弦退火的学习率动态策略。

CosineAnnealingLR（torch 中实现）

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\frac{T_{cur}}{T_i}\pi\right)$$




```
CosineAnnealingLR(optimizer, T_max=config.epochs, eta_min=1e-5)
```

在尝试过多组参数之后，发现准确率并没有提高，但是我认为余弦退火的学习率策略还是符合我的想法，但是可能训练迭代的次数不太够，之后我仍然保持这个改动，继续下面的改进。

参数 learning_rate 0.1 batch_size 128 epochs 50

训练结果：

4. 数据增强

将训练集的数据进行一些处理，使得训练数据的情况多样化，提高模型的泛化能力。我选择逐步增加数据预处理的方式（torch 当中均有实现）。

将训练集增加随机水平翻转 RomdomHorizontalFilp，再进行训练，

其他参数与上保持不变，部分超参数为 learning_rate 0.1 batch_size 128 epochs 40

训练结果：

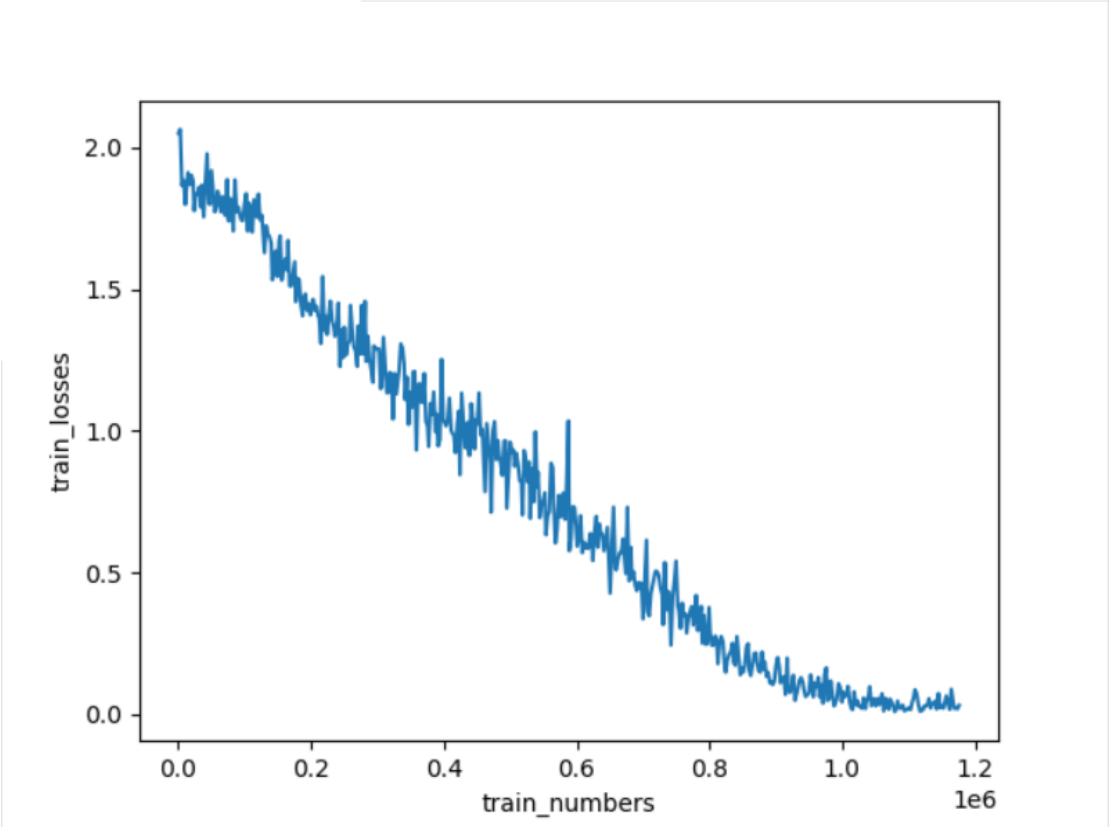
```
==========================
val accuracy:51.85034013605442%
```

我们可以看到，在验证集上的准确率达到了到目前为止的最高水平，说明数据增强的效果使非常 nice 的。

继续增加随机旋转处理 RandomRotation(30)

其他参数与上保持不变，部分超参数为 learning_rate 0.1 batch_size 128 epochs 100

Loss 曲线如下，非常的平滑，这与我们选择余弦退火的学习率策略是密不可分的。



验证集结果如下：

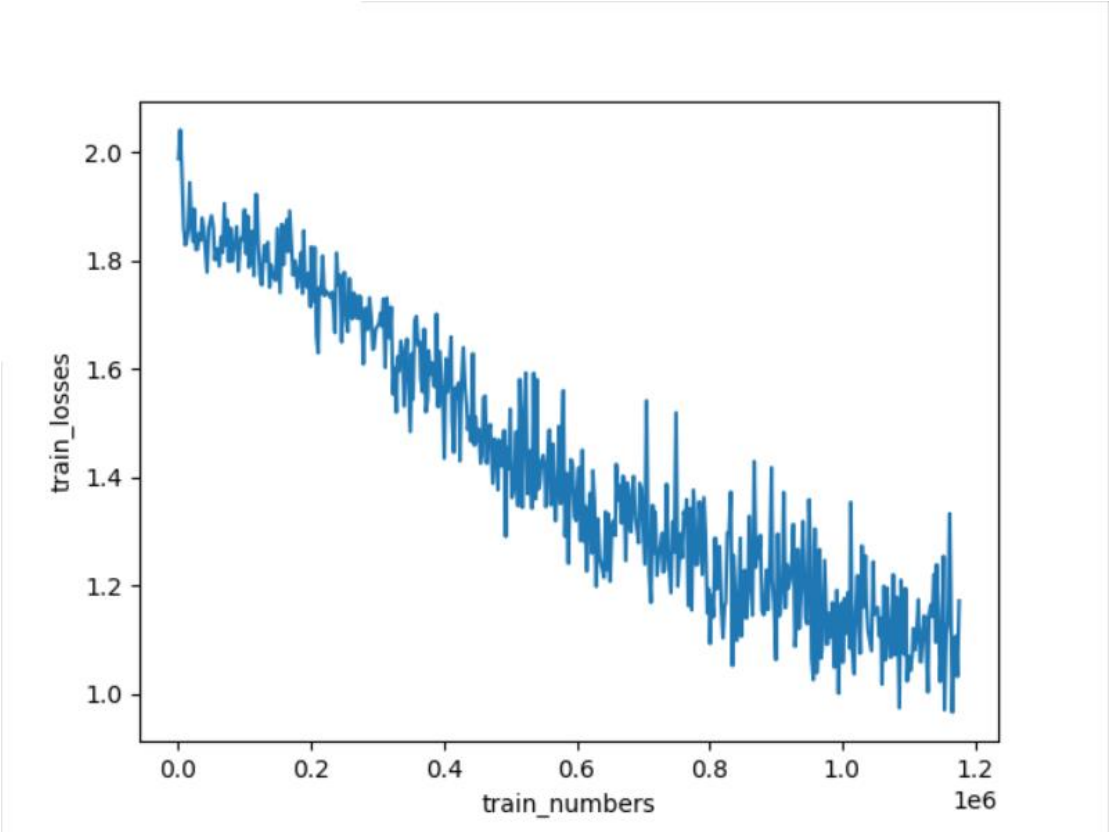
```
==========================
val accuracy:56.625850340136054%
```

结果真的非常棒，只是增加了一个数据预处理的方式，结果相对上述的情况提升了 5% 左右的准确率，这让我非常相信，数据预处理可以大大增加模型的泛化能力和性能。

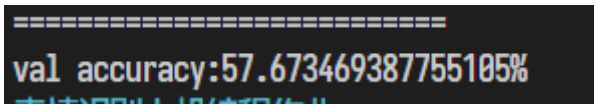
继续增加随机裁剪 RandomResizedCrop(config.image_size)，然后再进行训练

其他参数与上保持不变，部分超参数为 learning_rate 0.1 batch_size 128 epochs 100

Loss 曲线如下：



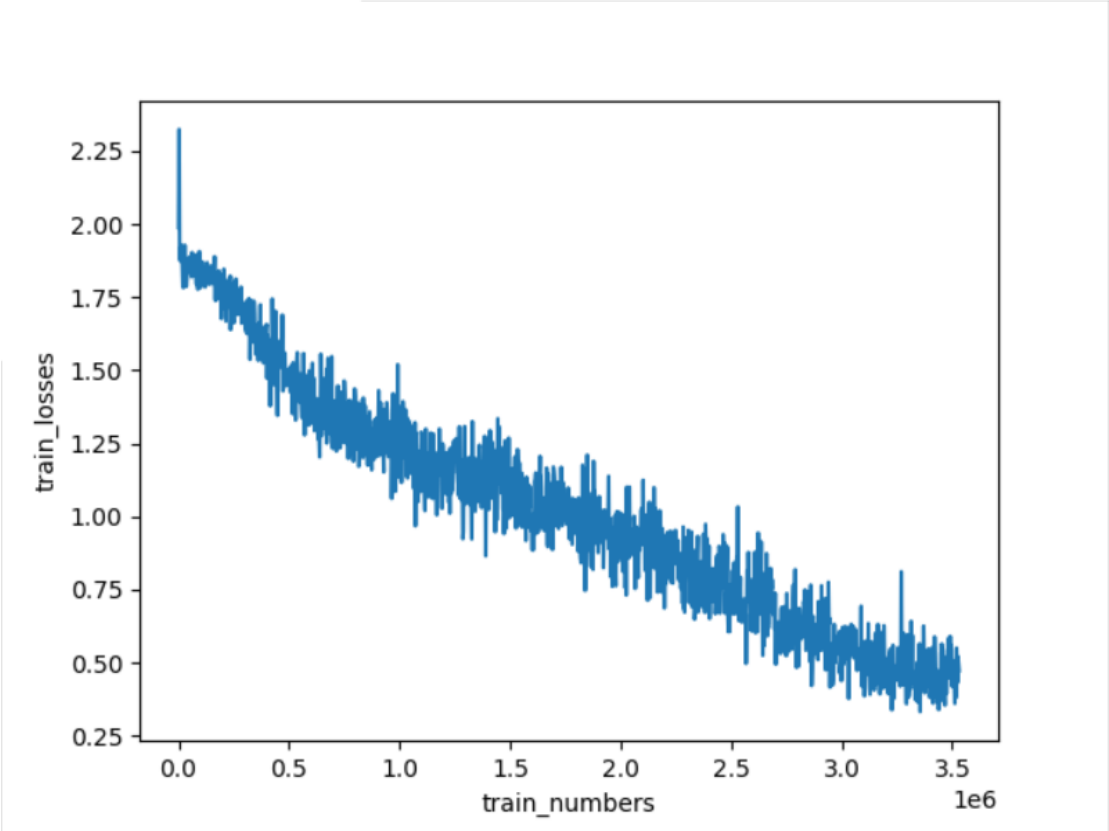
验证集结果：



我们可以看到 loss 并没有下降到较低的水平，说明在训练过程中，损失函数并没有完全收敛，可能是学习率上限不够大的问题，或者是训练迭代次数不够的问题。我认为并没有达到现在这种情况的最好效果，所以我才用增大训练周期的办法，再次进行训练。

故加大 epoch 为 300 再次进行训练：

训练的 loss 曲线如下图，我们可以看到现在的 loss，相对于 epoch 为 100 时，下降到较低的水平了，我有理由相信，此时在验证集上的准确率相对于以上情形都高。



验证集上的准确率为

```
============================
val accuracy:59.53741496598639%
```

果不其然我们再次将验证集上的准确率提高了 2%左右，提升已经可以说是相当不错了。

到目前为止，我们在通过 predict.py 测试集上检验一下模型改善的结果

```
============================
val accuracy:59.53741496598639%
test accuracy:63.165969316596936%
```

可以发现，测试集的准确率提升到了 63%，相比于一开始的 baseline 准确率 32%提升了大约 30%左右。但是准确率刚刚提升到 60%以上远远不够呀（doge），接下来继续优化模型的性能。

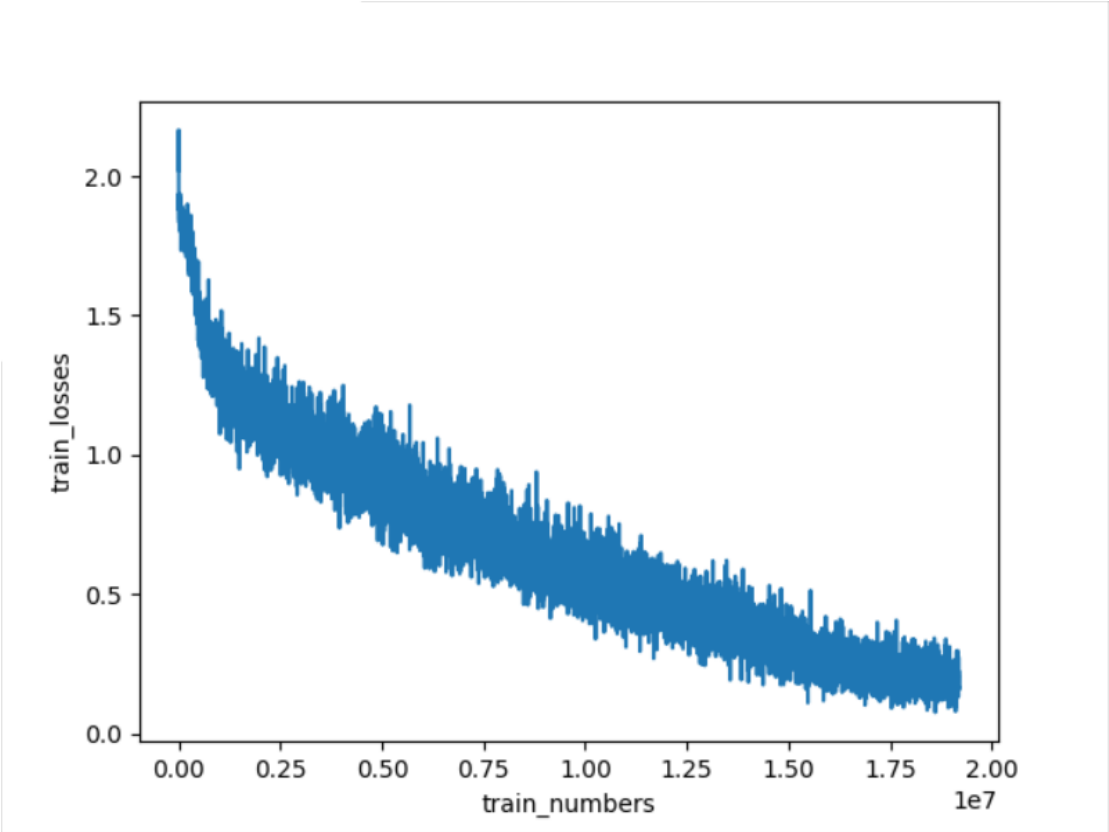
5. 扩充训练集

我们作业当中的训练集共有 11880 张图片，验证集共有 7350 张图片，测试集共有 7170

张图片，训练集比例较小，所以尝试把验证集加入训练集当中，进行训练

其余超参数均不变，learning_rate 0.1 batch_size 128 epochs 1000

Loss 曲线如下



虽然有些抖动，但是不难看出 loss 总体水平一直下降。

通过 predict.py 来计算在测试集上的准确率：

```
==========================
val accuracy:99.22448979591837%
test accuracy:66.41562064156207%
```

我们可以看到啊，这个结果是非常的 amazing：我们将验证集加入到训练集当中，可以看到模型将验证集拟合的非常好，同时我们增加训练集，也让测试集的正确率再次提了 3% 左右，效果还算是不错的。

四、 总结

这次作业，我体验了从模型构建到优化的整体过程，知道了一个模型的优化是非常耗时耗力、十分不易的。

我从模型结构的改变到参数选择，到一些数据增强和学习率，优化器调整等小的 trick

出发优化模型，整体效果还是不错的。其中大多数比较失败的训练的模型并没有记录到报告里（比如增加 trick 之后，慢慢尝试参数的过程）。

除此之外，由于不同类别的图片数量不太平衡，也可能导致训练效果的不太好，但是由于时间和精力的原因并没有尝试。还有一些方法并没有尝试，如 mixup、label smoothing、dropout 等避免 overfitting 的方法，也没有去尝试去设计新的损失函数等等。

整体报告到此，谢谢老师和助教在学习过程当中提供的答疑和帮助！