Project-2 report

李一帆 19307110499

April 2022

在此次实验中，我完成了包括 bonus 的所有内容，探索了部分不在问题清单中的额外内容。在 CIFAR10 数据集上的最佳模型在测试集上的准确率达到了 0.9528。

目录

1 CIFAR10

1.1 基本模型架构

在最开始，我参考了 VGG Net 和 ResNet[1]的网络设计，构建了一个深层 conv model。为了贴合 CIFAR10 数据集的 32×32 的输入特征，将 ResNet 开始的 kernel_size=7, stride=2 更替为 kernel_size=3（这一点在论文中也有提及）, stride=1，否则会损失较多的信息。且移除了紧接着的 MaxPooling 层，避免由于传递较小特征图导致深层网络迭代后信息丢失。

该 plain net 的结构见附录 A。

1.1.1 数据预处理

数据无意义归一化

在导入数据后，通过以下命令：

```
rgb_mean = np.mean(trainset.train_data, axis=(0, 1, 2))/255
rgb_std = np.std(trainset.train_data, axis=(0, 1, 2))/255
rgb_mean ,rgb_std
```

可以得到以下结果：

```
(array([0.49139968, 0.48215841, 0.44653091]),
 array([0.24703223, 0.24348513, 0.26158784]))
```

这是训练集数据的均值和方差。为了防止网络学习到该信息造成效果变差，将训练集和测试集数据都按照此进行归一化。

1.1.2 Residual Connection

Residual connection 的加入有利于缓解深层网络向后传播的梯度消失问题，因此在原先的 Plain Net 基础上加入 residual connection，搭建 Res Net（结构见附录 A），并使用消融实验对照 residual connection 的作用。

使用的超参数组合为：batch_size=128, epoch=20, learning_rate=1e-2，并使用 SGD 梯度下降算法。最后得到的对照结果为：

- Plain Net: 0.8319
- Res Net: 0.8541

从结果上来看，Res Net 的模型效果要比 Plain Net 好上一些，当然，这是在只循环 20 轮的前提下，增大循环轮数，差异可能更明显。

因此，保留 residual connection 这个优化算法，使用 Res Net 的结构进行后续实验。

1.1.3 默认超参数选取

在后面的实验中，默认选取以下超参数：

除此之外，还需选择一个初始学习率。在现有的模型基础上，尝试多种学习率，对比选用最佳的学习率作为后面实验的超参数。实验结果如下：

可以看到不同的学习率的模型效果差距还是很大的，接下来就选取 $1e-2$ 作为后续实验的**默认学习率**。

超参数	取值
batch_size	128
epoch	20
梯度下降算法	SGD
momentum	0.9
weight_decay	5e-4
drop_prob	0.5

学习率	1e-4	5e-4	1e-3	5e-3	1e-2	5e-2	1e-1	5e-1
准确率	0.6521	0.7259	0.7688	0.8381	0.8541	0.803	0.7877	0.2976

1.1.4 Different number of filters/layers

Classifier

在 *Deep Residual Learning for Image Recognition*，作者的 resnet 虽然是基于 VGG Net 改进的，但除了更深的网络深度和 residual connection 机制外，在 Res Net 上还有一个改动是将原本 VGG 的多层全连接隐藏层大幅缩减为了一层全连接输出层。Classifier 对网络是否会有增益，在现有的模型结构上加入 classifier(结构见附录 A) 观察效果：

1. Original Res Net: 0.8541
2. Res Net with classifier: 0.8241

加入 classifier 之后模型的表现反而变差了，这很有可能是由于加入的全连接层带来了一些的参数增长，而 20 轮的迭代则不够其收敛。将迭代轮数增加到 200 轮，测试结果：

1. Original Res Net: 0.86
2. Res Net with classifier: 0.884

从结果上来看，确实证实了前面的猜想，加入 classifier 后需要更加多轮的训练来收敛参数，但是网络的效果也变得稍微好了一些。

此外，从训练图来看：

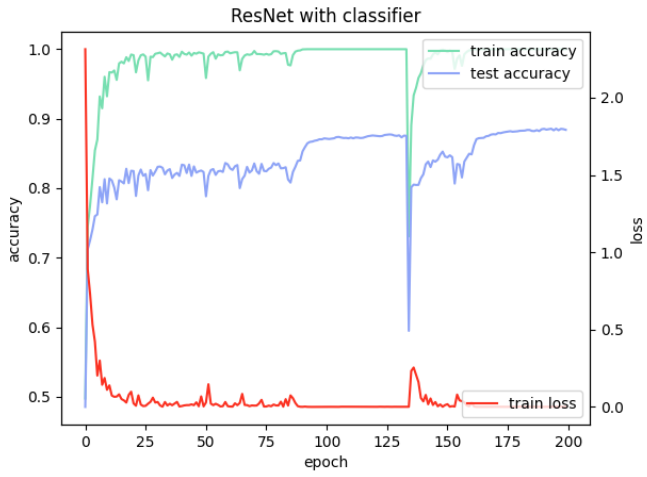

图 1: Res Net with classifier

发现在约 135 轮时，出现了模型短暂的大幅减益。从 loss 曲线上来看，猜测可能是到了后期，learning rate 的调整不恰当使得参数突然‘跳出’了最优区域，后续会继续探索是否要更换优化算法以及从 learning_rate 的选取上进一步优化。

Filters

实验测试不同数量的卷积滤波器组成的模型的效果。

目前使用的模型的卷积层共有 9 层，尝试削减最后一个 Res Block，其他结构保持不变，具体结构见 **ResNetconv4**，最终结果为 0.7478，程序总运行时间为 696s.

类似地，尝试增加卷积滤波器的数量。考虑到参数量的增加，我将这个新的 Res Block 添加到原结构的第 4 个 block 后，具体结构见 **ResNetconv6**，最终结果为 0.8186，程序总运行时间为 1079s. 其训练过程：

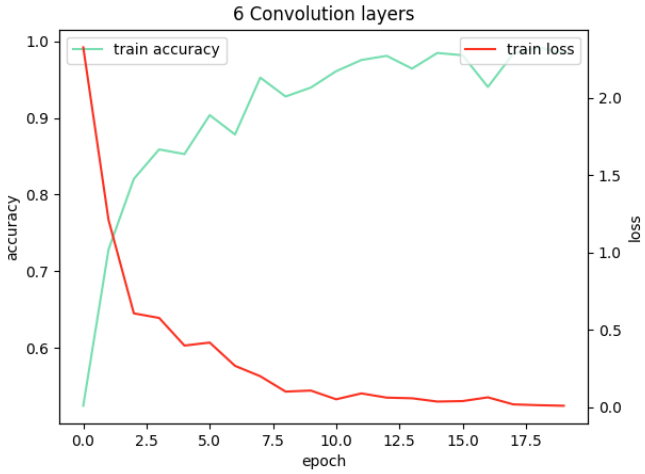

图 2: Increase the number of filters

从中可以看到，模型的收敛也稍慢了一些。有没有可能 6 layer 的表现效果比原先的模型稍差一些只是因为迭代轮数较少，模型未收敛呢？因此将训练轮数扩增到 200 轮，最终的训练的模型结果为 0.8792，且所需训练时间为 3h。

最初 5 个 Res Block 的模型训练时间约为 900s，增加卷积滤波器数量的最直接的影响就是明显降低了模型训练运行速度和参数收敛速度。从准确率和运行时间两个角度权衡，最终选择 filter 数量适中的网络结构。

1.2 消融实验

在这一部分，我将进一步探索 *batch normalization*、*drop out*、*parameter initialization* 对模型的影响。

1.2.1 Batch Normalization

Batch normalization 是在神经网络中常出现的优化层，它的具体作用也将在后面进一步探索，在此不再赘述。测试有无 batch normalization 对模型效果的影响：

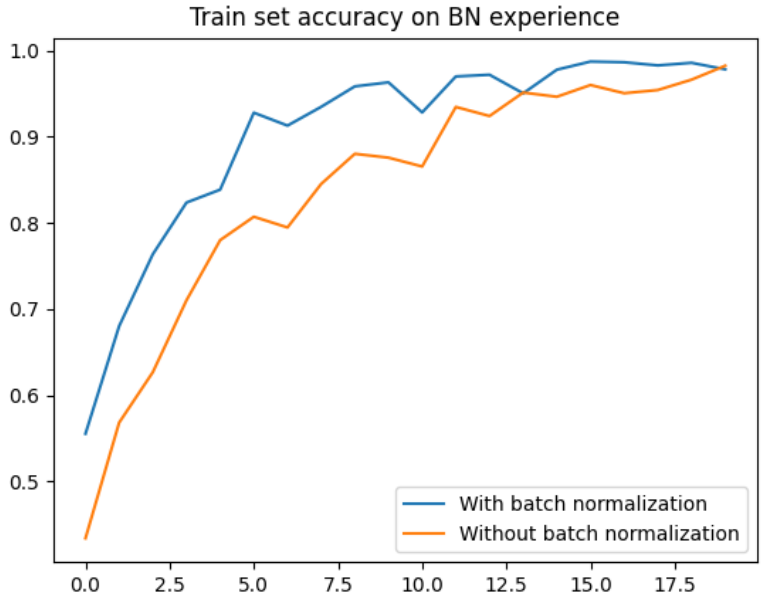

图 3: Batch normalization experiment

从结果上来看，有 batch normalization 层的网络明显收敛效果更好。对 BN 层效果和内在原理更深一步的探索将会在第二部分实现。

1.2.2 Drop Out

Drop out 是有效的缓解模型过拟合的手段，它通过在每一轮的训练按照一定概率随机地使部分神经元失效的手段来减少神经元之间复杂的共适应关系，提高模型的鲁棒性。测试不同 drop out probability 对模型性能的影响，其中当 drop out probability 为 0 时，等价于无 drop out 优化算法。测试效果如下：

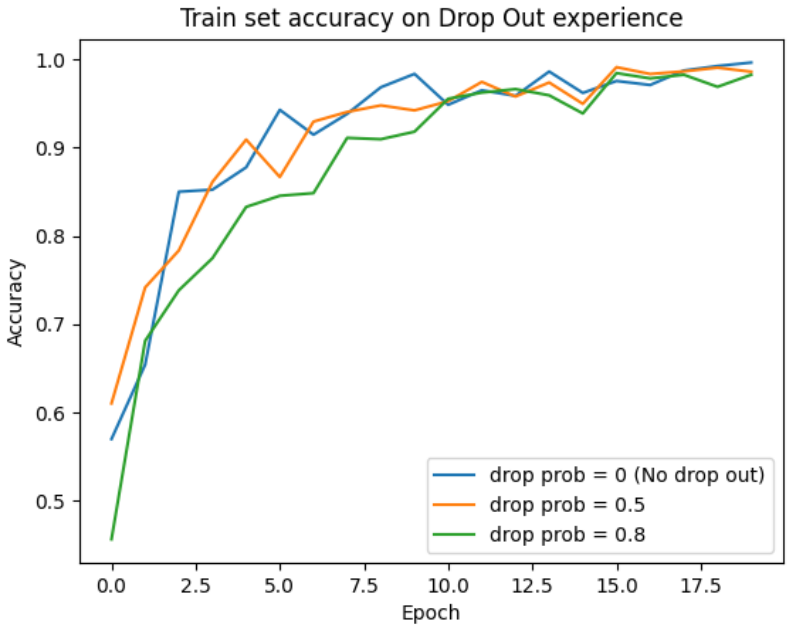

图 4: Different drop out probability experiment

从训练过程来看，在一开始的迭代过程中，当 drop out probability 越小时，模型的效果反而越好。我认为这是由于 drop out 在每一步迭代丢弃了某些信息，使得网络要学习到相同量的信息需要更长的时间，但不代表网络收敛后的模型效果不好，从图中也可以看出，在仅仅 10 轮迭代后，更大的 drop out probability 的模型表现马上就追赶上来了，并没有因为丢失部分特定信息而降低模型的准确率。

1.2.3 参数初始化

参数初始化作为模型训练的起点，决定了模型训练的初始位置。选择的好坏很大程度影响收敛的速度与最终的训练结果。在此之前的模型参数均使用的是 pytorch 默认初始化方法（查询源代码可知其卷积层和线性层的 weight 使用的均为 Kaiming Uniform 方法，BN 层则使用的是 Uniform 方法）。接下来探索是否有更佳的参数初始化方法。

参数初始化的方法有很多，综合考虑各方法的优劣以及目前模型使用的 ReLU 激活函数，依旧先尝试使用 HE 初始化方法，但使用 normal 来替换 [2]：

参数	初始化方法	超参数
BatchNorm.Weight	constant	1.0
BatchNorm.Bias	constant	0.0
Conv2d.Weight	Kaiming normal	\
Linear.Weight	Kaiming normal	\
Linear.Bias	constant	0.0

最终结果为 0.7912。效果并没有预期好。

再使用简单的 normal 分布来初始化卷积层和线性层权重。均值均取为 0，使用不同的方差实验结果如下：

方差	模型训练集准确率
0.001	0.8267
0.01	0.8341
0.1	0.7104

最佳的参数下，模型效果有稍微的提升，但这么多方法尝试下来，其实最终效果差距都不大，猜测主要原因是由于网络结构中含有 BN 层，减小了网络对初始值尺度的依赖，因此不同初始化的方法作用差距并不大，只要初始化的方差足够小，不会出现梯度爆炸问题，就满足网络要求了。

1.3 Optimization Strategies

1.3.1 Loss Function

Regularization

在此前的模型训练结果中，无一例外地发生了严重的过拟合。使用 l_2 正则化的方法对模型参数进行约束。

尝试使用不同的罚项系数进行实验，结果如下：

最后两个参数下，模型已经不能收敛。

加入正则化后，loss 下降的速度会变慢，准确率 Accuracy 的上升速度也会变慢。但在合理的范围内，正则化系数越大，模型的过拟合问题越小（但不代表模型效果越好），以下是权重为 1e-4 和 1e-2 的训练过程图，可以明显看到权重更大的模型训练误差和测试误差越接近：

正则化系数	准确率
1e-4	0.8338
1e-3	0.8354
1e-2	0.6907
1e-1	0.1676
1	\
5	\

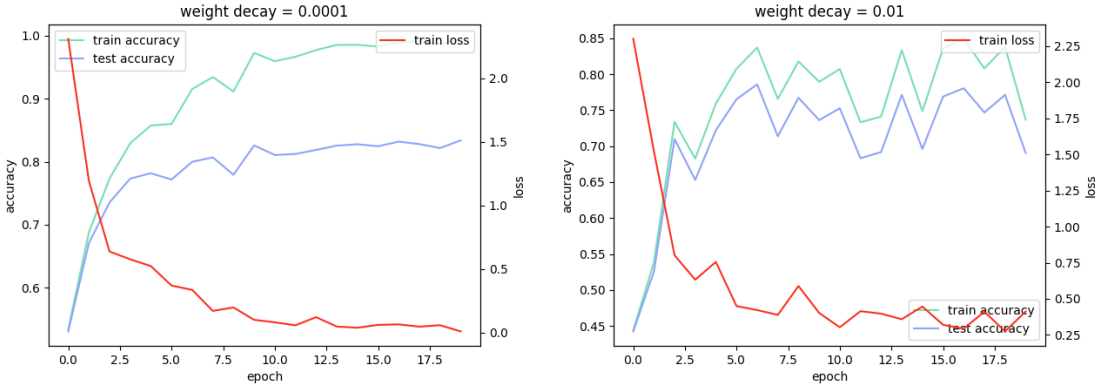

图 5: Effect of different weight decay on model over-fitting

Label Smoothing

在 **When does label smoothing help?**[3]一文中，深入研究了标签平滑对深度神经网络的影响。标签平滑是对损失函数的修正。在普通的 NN 训练中，网络容易对预测变得‘过于自信’，这可能会降低它们的泛化能力，同时由于大型数据集通常会包含标签错误的数据，因此让神经网络对所谓‘正确答案’持怀疑态度可以一定程度上减少围绕错误答案的建模。因此，标签平滑可以起到正则化和提高网络鲁棒性的作用。

在实现上，令 ϵ 为标签平滑调整因子（论文中建议为 0.1，代码实现也采用了该值），将目标标签的概率调整为 $1-\epsilon$，并将这个 ϵ 平摊到其他标签概率上。

实现上，在原先使用的 Cross Entropy 损失函数进行实验，修正过后的损失函数为:

$$H(y,p) = -(1-\epsilon)logp_t - \epsilon/(K-1)\sum_{i\neq t} logp_i$$

修正后的模型训练准确率为 0.869，比起修正前有很大提升。这也印证了标签平滑对模型的增益效果。

Different Loss Function

此前，使用的损失函数为交叉熵函数，是在分类问题中非常常用的目标函数。但是在分类问题中，还有更多可选择的目标函数。考虑到该分类问题的特性以及各损失函数的优化版本，除了之前使用的交叉熵函数，还挑出了 *Smooth L1-loss*, *KL Divergence Loss* 进行实验。所有损失函数实现均在 **lossfunc.py** 中.

Smooth L1 loss 解决了 L_1 和 L_2 损失的缺陷。当 x 较大时，使用 L_2 损失的梯度也非常大，使得在训练初期，预测值和 ground truth 差距较大时的梯度十分大，训练不稳定。而到了训练后期，预测值和 ground truth 差距较大时的梯度非常小时，使用 L_1 损失函数的梯度却仍然为 1，损失函数容易在稳定值附近波动，难以收敛到较高精度。Smooth L1 Loss 结合了二者的优点，避开缺陷:

$$loss(x, y) = \frac{1}{n}\sum_{i} z_i$$

$$where\ z_i = \begin{cases} 0.5(x_i - y_i)^2 & ,if\ |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5 & ,otherwise \end{cases}$$

KL 散度又叫相对熵，适用于衡量两个分布之间的距离：

$$loss(p, q) = \sum_{i=1}^{N} p_i(logp_i - logq_i)$$

其中 p_i 为预测的类别概率，q_i 为修正过后的 ground truth 概率。

模型准确率结果如下:

损失函数	准确率
Smooth l1 loss	0.433
KL divergence	0.867
Cross Entropy	0.8674

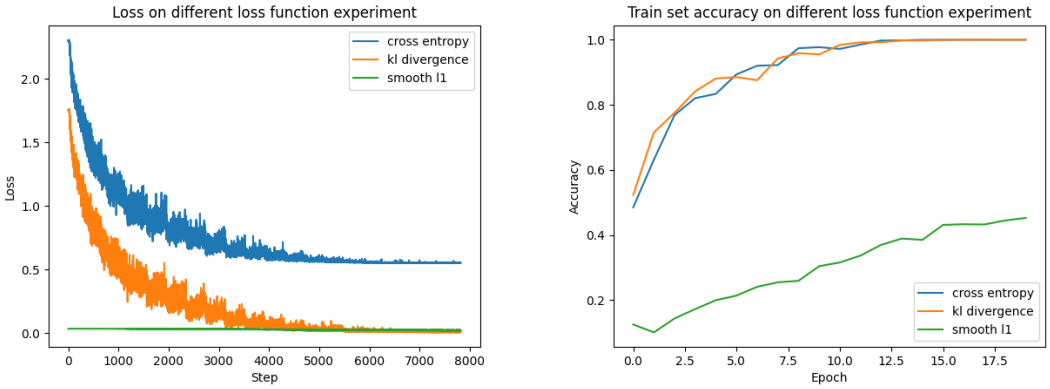

图 6: Train results of different loss functions

Smooth l1 loss 的 loss 变换并不是恒为 0，只是由于数值太小被压缩到了 0 附近。

其实会发现，使用 cross entropy 的模型结果和使用 KL divergence 的模型结果非常相近，就连 loss 和 train set accuracy 的变化曲线形状都十分接近。从数学意义上来说，A 和 B 的交叉熵 $=A$ 与 B 的 KL 散度（即相对熵）$-A$ 的熵。在该模型训练问题上，A 就是模型学习的标签分布，而 B 就是训练数据的标签分布。可以认为训练数据的分布 A 是给定的，也就是说 A 的熵是固定的。那么**最小化 KL 散度损失就等价于最小化交叉熵**。为了分布理解和推导，后续还是使用修正过后的交叉熵作为模型损失函数。

1.3.2 Different activation function

尝试使用不同的激活函数进行模型的训练。待试验的激活函数有 sigmoid 型的 Tanh 函数、ReLU 型的 ReLU 函数（目前使用）、ELU 函数、LeakyReLU 函数。观察实验结果并分析其优劣。

可以看到，sigmoid 型的激活函数比起 ReLU 型的激活函数 loss 下降的要慢很多，模型过拟合问题也更加严重，这是由于 sigmoid 型函数自身的梯度较小，导致参数更新缓慢，且容

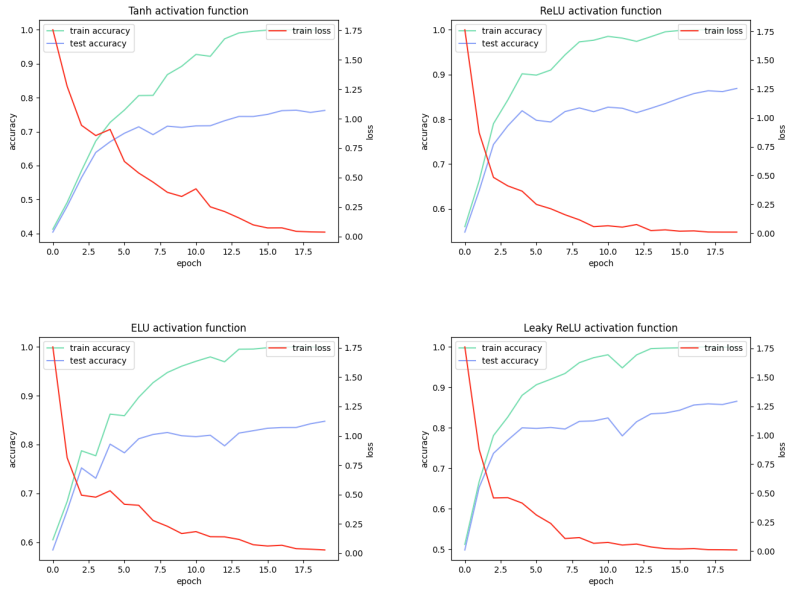

图 7: Different activation function

易产生梯度消失。ELU 激活函数和 leaky ReLU 激活函数均为 ReLU 的修正激活函数。ELU 函数可以避免死亡 ReLU 问题，同时使输出的平均值接近 0，以 0 为中心；通过减少偏置偏移的影响，使正常梯度更接近于单位自然梯度，从而使均值向零加速学习；但是由于 ELU 单元中含有指数运算，会降低程序的运行时间。Leaky ReLU 也能达到类似的效果，且其运算更加简单，因此比 ELU 函数更加快速。

但是在上图中却发现，三种 ReLU 型函数的结果没有太大区别，ELU 和 Leaky ReLU 并没有像我们分析那样比 ReLU 要好上一些。思考原因应该是由于我们的网络中在激活函数之前有一个 BN 层，已经通过对输入值归一化从而避免了输入值偏移的影响，因此 ELU 和 Leaky ReLU 在这里并不能产生更进一步的效果了。

此外，不同的激活函数所需的运行时间如下：

激活函数	准确率	运行时间/s
Tanh	0.7625	926
ReLU	0.869	914
ELU	0.8478	931
Leaky ReLU	0.8658	916

可以看到，Tanh 和 ELU 由于含有大量的指数运算，运行时间相对较长。在 ReLU 型函数效果差不多的前提下，ReLU 由于其计算最为简单，运行时间是最短的。因此，后续将依旧使用 ReLU 函数作为激活函数。

在后续的实验中，还使用到了 GELU 函数，效果也不错，甚至优于 ReLU 函数。

1.3.3 Different Optimizer

由于优化器不同，模型性能可能会差异很大，选择不合适的优化器可能会对机器学习项目产生很大的负面影响。这使得选择优化器成为构建、测试和部署模型过程中的关键一环。选择优化器的问题在于没有一个可以解决所有问题的单一优化器。实际上，优化器的性能高度依赖于项目本身和设置。

在这里，我将常见的优化器主要分为三种：梯度下降法、动量优化法和自适应学习率法。梯度下降法是比较基础的，仅依赖梯度信息的优化器，一般来说很难满足较大网络训练的需求。动量优化法有 SGD with Momentum（SGDM）和 Nesterov Accelerated Gradient（NAG）。自适应学习率法主要有 Adaptive Gradient Algorithm（AdaGrad）、Root Mean Square Propagation（RMSProp）、Adaptive moment estimation（Adam）。Adam 是前两者的优化版，更进一步，Adam 还有 AdamW（加入 weight decay）、AdaMax（对学习率上限提供了一个更简单的范围）、Nadam（类似于带有 Nesterov 动量项的 Adam）等变体。因此，基于以上考虑，最终选用 SGDM、NAG、Adam 和 AdamW 进行实验比较。结果如下：

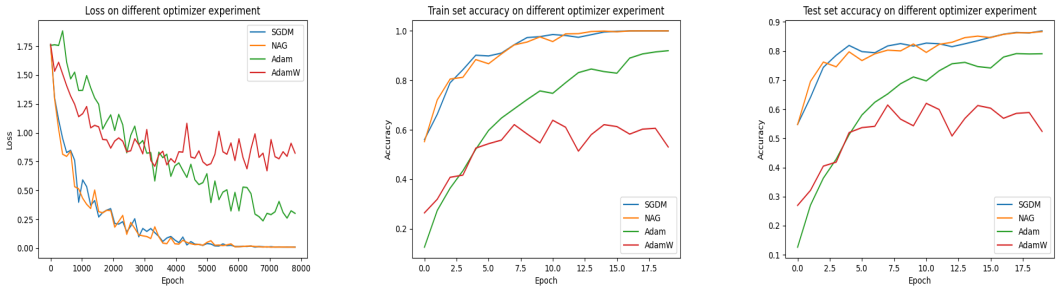

图 8: Different optimizer function

总的来说，使用 SGD 的优化器表现暂好于使用 Adam 优化器。Momentum 和 Nesterov 的效果不相上下，但从运行速度上来说，SGDM 更胜一筹。因此，选用 SGDM 作为模型的优化器。

1.4 Other Optimization

关于我自己额外做的一些提升模型效果的优化。

1.4.1 Data Augmentation

到目前为止，无论之前尝试了再多的优化方法和各种消融实验，模型的过拟合问题一直非常严重。回想第一个 project，当时的所有实验结果中，数据增强是减缓我的模型过拟合最佳的手段。因此，在此次模型中，也尝试加入数据增强的方法。

尝试图像缩放，随机裁剪，随机水平翻转，随机竖直翻转，随机旋转，色彩抖动等手段。具体实现见 **data_aug.py**。

以下是各组实验的结果：

数据增强方法	**准确率**
horizontal flip	0.8721
horizontal flip & random crop	0.8948
horizontal flip & vertical flip & random crop	0.7875
horizontal flip & random crop & rotation	0.8834
horizontal flip & random crop & rotation & color jitter	0.8848

从结果可以发现，只要选择了合适的数据增强方法，出来的效果都差不多。该优化方法确实能大幅提高模型的准确性，同时对过拟合有非常好的缓解效果，以下是最终的模型训练过程：

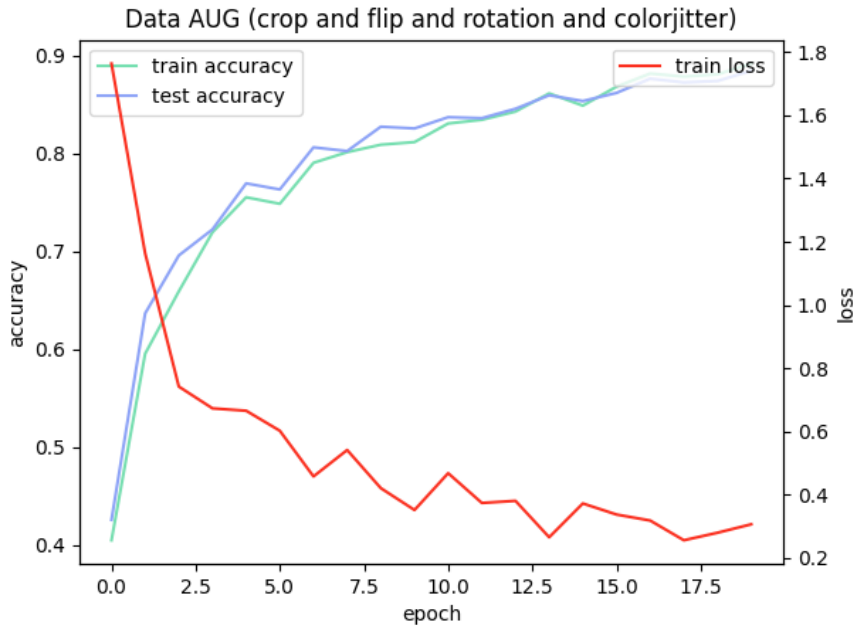

图 9: After adding data augmentation

可以看到对模型过拟合问题缓解的非常好!

1.4.2 CutMix

除了上面对单张图片独立进行数据增强，还有很多数据增强的方法可供选择。在 2019 年的 *CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features*[4]这篇论文中提出了一种新的数据增强方法 CutMix。这种方法是 Cutout 和 Mixup 的结合体。

Mixup 指的是将随机的两张样本按比例混合，分类的结果也同样按比例分配。Cutout 指的是随机的将样本中的部分区域裁减掉，并且填充 0 像素，分类结果保持不变。而对于 CutMix，Cut 指的是切割出图像的一小块，Mix 指的是随机选择另一张图片的相同区域贴到这一小块上，同时按照切割的比例混合二者的 label。比起前两种数据增强的方法，后者具有以下优点：

- 在训练过程中不会出现非信息像素，从而能够提高训练效率.
- 保留了 regional dropout 的优势，能够关注目标的 non-discriminative parts.
- 不会有图像混合后不自然的情形，能够提升模型分类的表现.
- 通过要求模型从局部视图识别对象，对 cut 区域中添加其他样本的信息，能够进一步增强模型的定位能力.

具体的数学变化操作如下：

$$\overline{x} = M \odot x_A + (1 - M) \odot x_B$$
$$\overline{y} = \lambda y_A + (1 - \lambda) y_B$$

其中 M 是一个 mask，取值为 0 或 1.x_A, x_B 为两张图片，y_A, y_B 是它们对应的 label。$\overline{x}, \overline{y}$ 是 cutmix 后得到的图像和 label。λ 服从 $\beta(\alpha, \alpha)$ 分布。

M 通过随机生成的一个 bounding box 得来。假设这个 bbox 的参数为 r_x, r_y, r_w, r_h，由以下随机生成：

$$
\begin{aligned}
r_x &\sim Unif(0, W) \\
r_y &\sim Unif(0, H) \\
r_w &= W\sqrt{1-\lambda} \\
r_h &= H\sqrt{1-\lambda}
\end{aligned}
$$

M 在 bbox 的区域内为 0，其他区域内为 1.

代码详见 **cutmix.py**。模型训练结果如下：

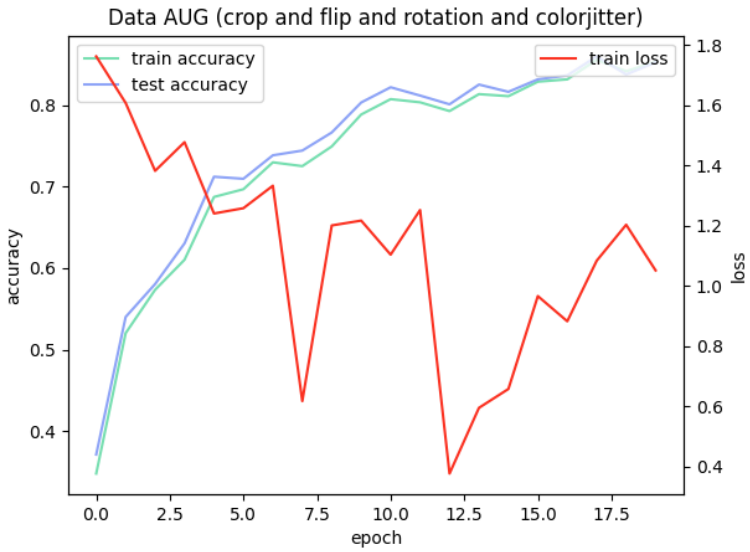

图 10: After adding CutMix

模型效果看起来好像并没有比加之前好多少，甚至模型准确率有所下降。且 loss 曲线看起来很乱。在结合上面数学公式思考后，认为这是由于 mixup 带来的，无论在混合的两类中选择哪一类，另外一类都会都会造成很大的 loss，且 loss 的值大小受到每次 cut 的区域大小 λ 的影响，因此 loss 整体的波动会很大。因此在这种特殊情况下，loss 收敛快慢和波动大小并不能代表模型的好坏。至于 CutMix 的效果，我将训练轮数增加到 40 轮再次对比效果：

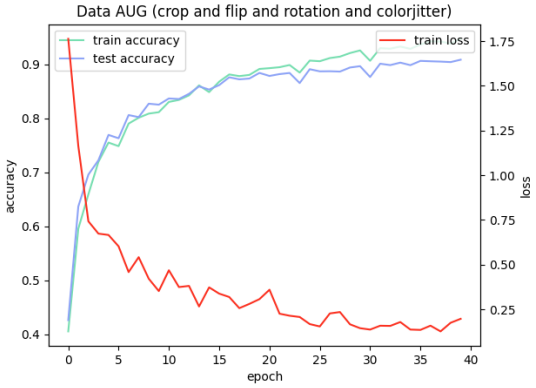

图 11: No CutMix

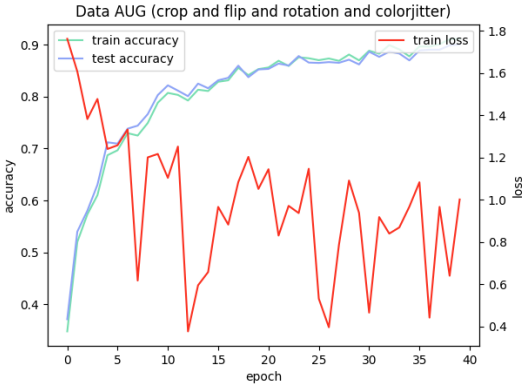

图 12: CutMix

训练 40 轮后，不加 CutMix 的模型准确率为 0.909，加了 CutMix 的模型准确率为 0.9009，

二者几乎没有区别。但是观察上图两条准确率曲线会发现，不加 CutMix 的模型到后期训练集准确率曲线逐渐与测试集准确率曲线拉开差距，而加了 CutMix 之后的模型在后期依旧没有出现过拟合问题。这证明 CutMix 确实能显著提高模型的鲁棒性。

1.4.3 Learning Rate Annealing

在之前的几次增加模型训练轮数至 200 轮时，无一例外都在后期出现了准确率的突降和 loss 的突增现象。我认为这是由于 learning rate 到了后期，对于模型的训练太大了，因此有时候可能会从局部最优的谷底逃出来。一种有效平缓训练曲线的方法就是退火，即随着训练的轮数增加慢慢降低学习率，使得模型的参数最终能收敛到一个比较精确的值上。

在此部分实验中，我参考了 *SGDR: Stochastic Gradient Descent Warm Restarts*[5]这篇论文提出的方法。使用余弦退火的方式并加上 warm restart 来调整学习率。余弦退火可以使得学习率逐渐下降，让模型接近最小值。但有时候模型的训练有可能陷入局部最小值，warm restart 则可以通过突然提高学习率，来“跳出”局部最小值并找到通向全局最小值的路径。

该方法有两个超参数需要设定: T_0 为第一次 restart 的 epoch 数目，T_{mult} 为每一次 restart 之后，下一个 restart 增加的距离。分别使用论文中收敛最快的 $T_0 = 10, T_{mult} = 2$ 和收敛后准确率最高的 $T_0 = 200, T_{mult} = 1$ 参数组合进行实验:

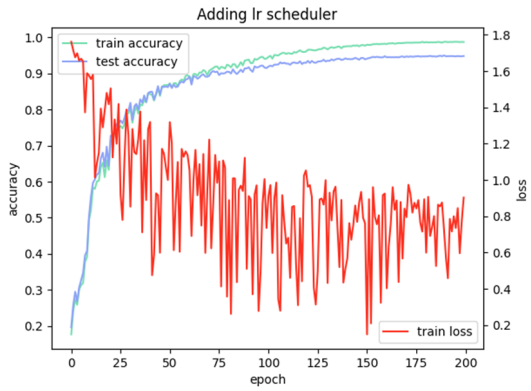

图 13: T_0=200, T_mult=1

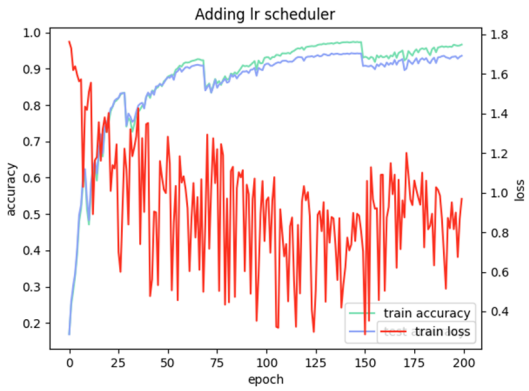

图 14: T_0=200, T_mult=2

相对来说，左图的下降策略要更适合我的模型一些，右图的抖动有点太大了。此外，我还尝试了 mutlsteplr 方法，但效果没有那么好，这里就不再赘述了。

1.5 Network Interpretation

在这一部分，尝试可视化训练出来的最佳网络的特征图，对神经网络做出解释。

先对 5 个 stage 的卷积核进行可视化。在每层选出 18 个卷积核，使用热力图进行可视化，颜色越浅代表数值越大:

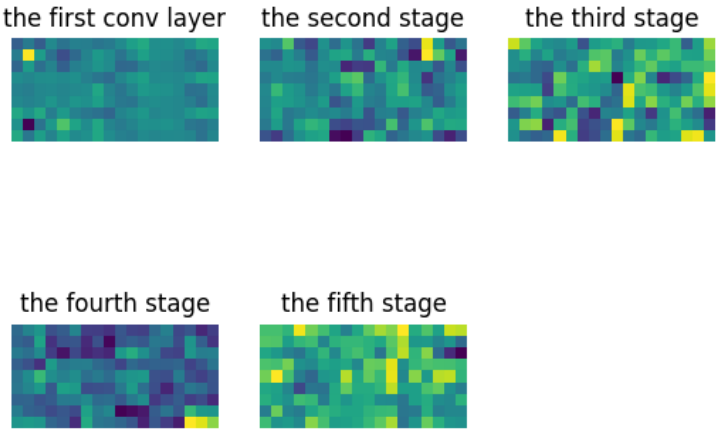

图 15: Visualization of convolution kernel

可以看到信息比较抽象。将图像输入网络，提取每一层输出的特征图。以下是输入的图像，类别为 automobile：

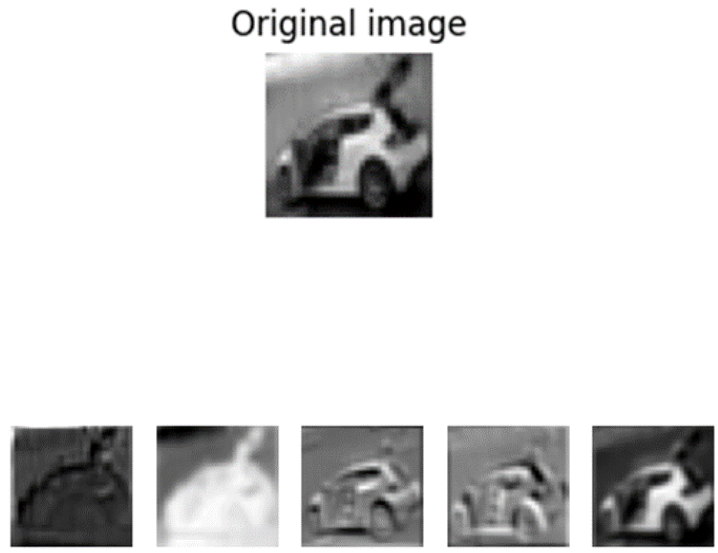

图 16: Output of stage 1 conv layer

图 17: Output of stage 2 conv layer

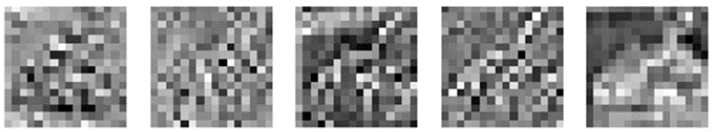

图 18: Output of stage 3 conv layer

前两个 stage 的卷积核对图像边界信息比较敏感，尤其是第二个 stage，和高通滤波器的效果非常相像。不同的卷积核提取的边界信息也不一样，第一个 stage 中，有的卷积核提取的是 automobile 的外轮廓信息，有的卷积核提取的是 automobile 的内部轮廓信息。到了第三个 stage 开始，模型提取的信息就开始变得抽象了。

1.5.1 Grad-CAM

进一步地，我想知道神经网络究竟关注哪些区域。在这一部分实验，我参考了 *Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization*[6]这篇论文的方法，使用 Grad-CAM 来可视化模型关注区域。

论文实现原理如下：首先网络进行正向传播，得到特征层 A（论文中主要指最后一个卷积层的输出）和模型预测值 y。假设 y^c 是模型针对这个数据的预测值，想要知道模型针对这个类别的感兴趣区域。于是对 y^c 进行反向传播，得到向后传播至特征层 $\hat{A}$，代表 A 中每个元素对 y^c 的贡献。再用每个通道的重要程度对 A 进行加权求和并进行 ReLU 计算就可以得到 Grad-CAM。数学公式如下：

$$L^c_{Grad-CAM} = ReLU(\sum_k \alpha^c_k A^k), \qquad \alpha^c_k = \frac{1}{Z}\sum_i \sum_j \frac{\partial y^c}{\partial A^k_{ij}}$$

其中 k 为特征层 A 中的第 k 个通道，c 为预测类别，y^c 为模型针对类别 c 预测的分数，Z 为归一化常数。得到热图后和原图进行叠加得到最终可视化结果，实现代码见 **grad_cam.py**。

从测试集中挑选一些数据样本进行可视化，结果如下：

第一张 horse 的图像模型关注的是头部，第二张 dog 的图像模型关注的是面部和耳朵，第三章 frog 的图像模型关注的是背部的花纹以及背景的草。还是比较有意思的。这次使用的数据集图像比较小，很多特征信息都是有用的，因此模型关注区域占比都比较大。参考一些其他较大的图像背景下神经网络模型的 Grad-CAM 结果，能更加看出这个可视化方法对模型的解释性，比如论文中的示例模型可视化结果：

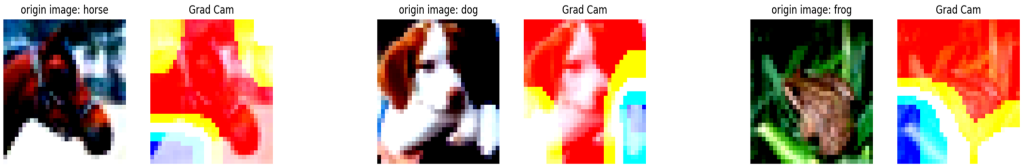

图 19: My Grad-CAM result

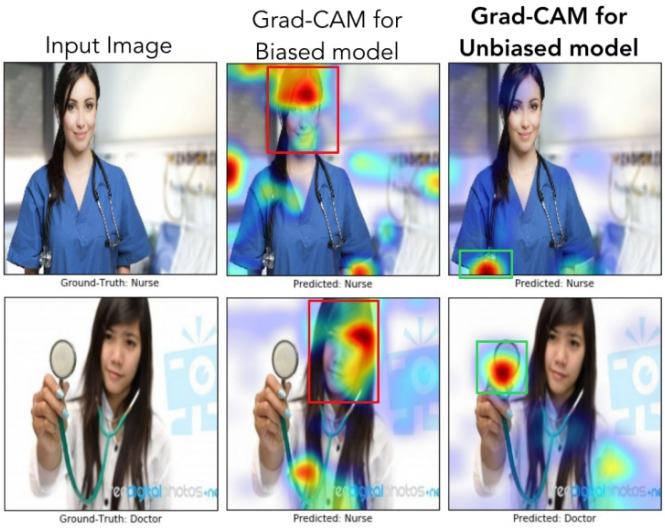

图 20: Grad-CAM example

1.5.2 Best Model

最终，使用以上优化方法和调整超参数之后，我的最佳模型在 165 个 epoch 训练后在测试集上达到了 95.28% 的准确率，所需训练时间不到 1.5h。模型的参数保存在 **best.pth** 中。以下是模型的结构、使用的优化方法及超参数：

model	ResNetC	**GPU**	GeForce GTX 1080 Ti
batch size	128	**initial learning rate**	0.1
epoch	165	**random seed**	2022
drop out probability	0.5	**criterion**	KL divergency
ϵ **in label smoothing**	0.1	**optimizer**	SGD with momentum
momentum	0.9	**learning rate scheduler**	cosine annealing with warm restart
T_0	205	T_{mult}	1
activation function	GELU	**data argumentation**	crop&horizontal flip&rotation&colorjitter
minimal learning rate	1e-3	**cutmix**	$\alpha = 1$

2 Batch Normalization

在本节，主要完成了 VGG-A 模型上的 Batch Normalization 添加，并参考论文 *How does batch normalization help optimization?*[7]中提出的方法，探究 BN 在网络中所起的作用。

2.1 Batch Normalization 的实现

在 VGG-A 模型的基础上，添加 batch normalization 优化。代码上的实现只需要在每一层的激活函数前使用 *nn.BatchNorm1d()* 或 *nn.BatchNorm2d()* 模块即可。实现后，网络的参数量由 9750922 增加至 9758474。

2.2 Loss Landscape

VGG 网络添加 BN 层前后的 loss landscape 如下：

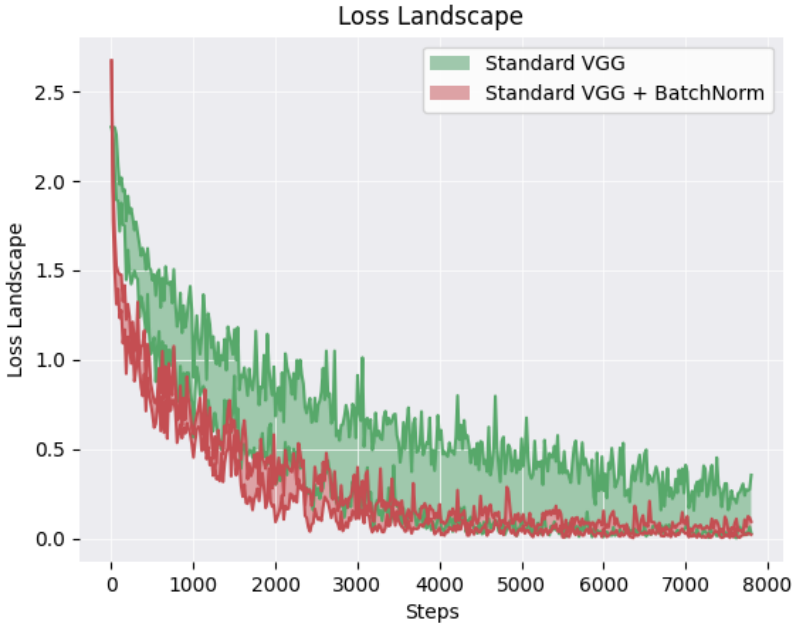

图 21: loss landscape compare

可以看到，加了 BN 层之后的模型 loss 下降数速度远快于未加 BN 层的模型，且不同学习率下 loss 的方差也远小于不加 BN 的模型。这说明 BN 层确实使得 loss 函数变得平滑了。

2.3 Gradient Predictiveness

计算第 i 步和第 $i-1$ 步的损失函数梯度，并在不同学习率下多次试验。在实现过程中，我记录了最后一层的梯度，分别在学习率为 [1e-3, 2e-3, 1e-4, 5e-4] 下训练模型。类似 loss landscape 的实现，分别选出两个模型下梯度距离最大和最小的值记录，可视化这两条曲线及区间。最终结果如下：

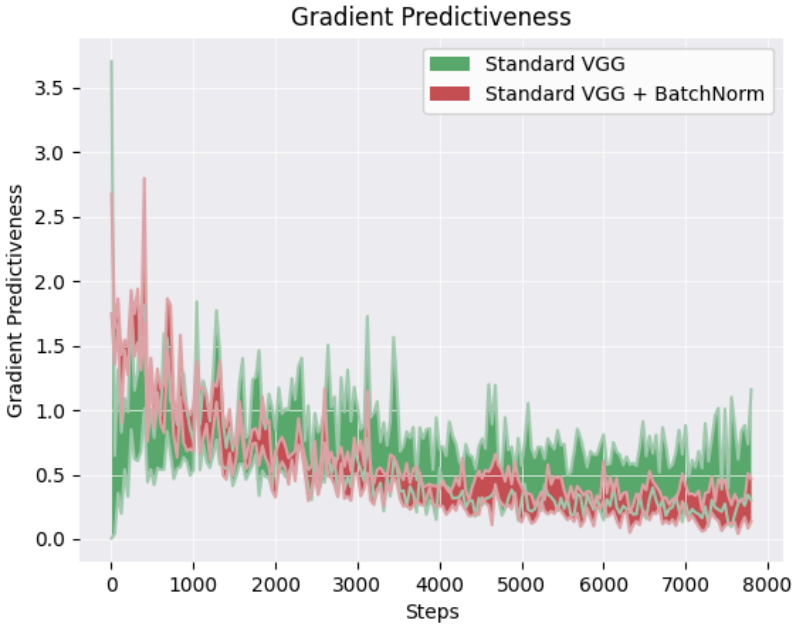

图 22: gradient predictiveness compare

可以看到加了 BN 层后，梯度距离不仅变小了，不同学习率下的梯度距离方差也变小了。范数变小可以说明模型对下一步的预测性提高，对于模型的收敛速度大大提升，波动性减小。方差变小也可以说明模型的收敛更加稳定了，且加了 BN 层之后使得模型对学习率取值的依赖大大降低。

除此之外，可以观察到在模型刚开始训练的时候，加了 BN 层的模型 loss 梯度反而要高于未加 BN 层的模型，这说明加了 BN 层的模型在一开始使用了更大的前进步伐，有利于更快的收敛，而到了后期，则谨慎更新，有利于模型参数的收敛，减少波动性。

2.4 "Effective" β-smoothness

继续使用 β-smoothness 衡量损失梯度的稳定性。和前面的实验一样，用两个模型在不同的学习率下（[1e-3, 2e-3, 1e-4, 5e-4]）进行训练，β-smoothness 计算则使用最大的梯度距离二范数除以移动距离：

$$\beta = \max \frac{||\nabla l(w_1) - \nabla l(w_2)||_2}{||w_1 - w_2||_2}$$

上式的意义代表 weight 的微小改变能给 loss 的梯度带来多大变化，换言之，衡量的是一阶梯度的平滑性。

最终结果可视化如下：

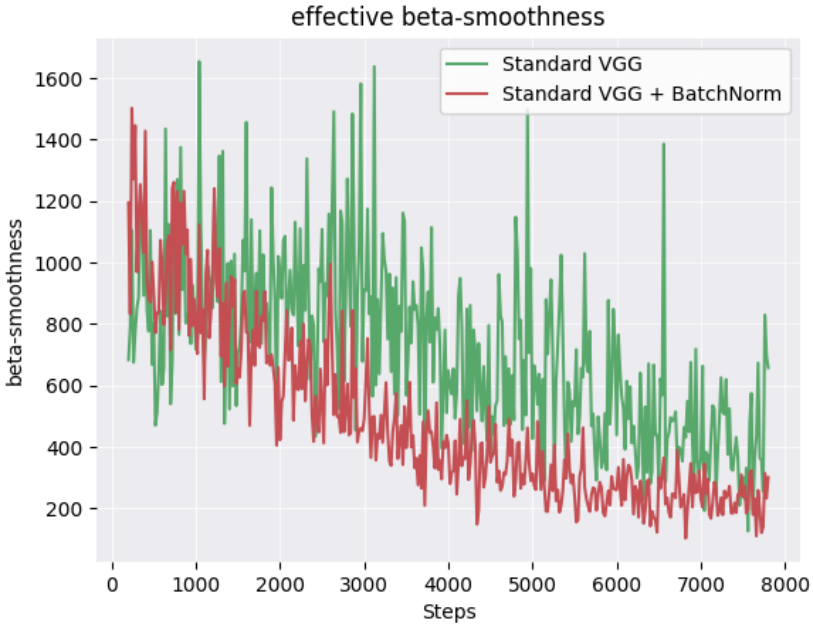

图 23: β-smoothness compare

可以明显看到加了 BN 层后的模型 β-smoothness 系数要小于未加 BN 层的模型，说明其一阶梯度要比不加 BN 层的模型平滑的多，且对学习率更加不敏感，提高了模型的适应性。

3 DessiLBI

3.1 LeNet on MNIST

使用 DessiLBI[8]训练 MNIST 数据集上的 LeNet。网络使用的超参数如下:

超参数	**取值**
batch size	128
epochs	20
learning rate	1e-2
kappa	1
interval	20
mu	20

训练过程如下:

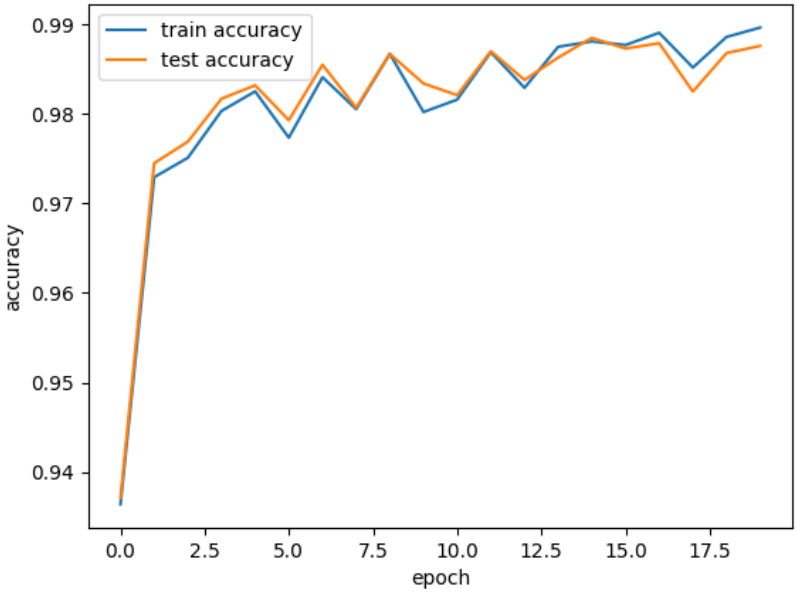

图 24: LeNet on MNIST with DessiLBI

可以看到使用了 DessiLBI 的模型对过拟合问题控制的很好。

训练完模型之后，对模型隐藏层进行剪枝，percent 为 80. 剪枝结果如下：

	准确率
剪枝前	0.9876
剪枝一层（conv3）	0.8939
剪枝两层（conv3 & fc1）	0.8719

可以看到尽管剪枝的比例很大，但模型的准确率损失并不是很大。尤其是对全连接层的剪枝几乎不对模型准确率有太大影响。

可视化 conv3 卷积核剪枝前后的结果（仅取其中一百个核）：

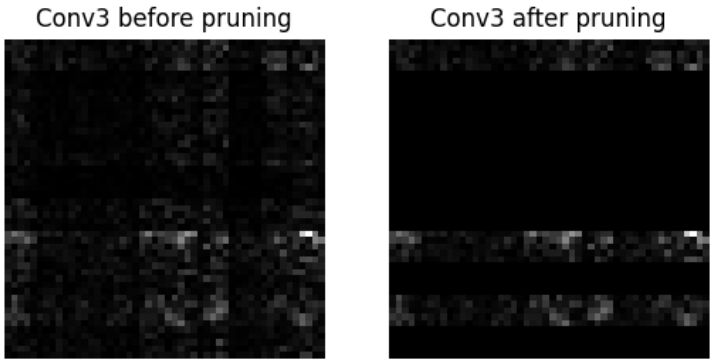

图 25: Conv3 before and after pruning

可以比较明显地看到，剪枝掉的部分都是激活比较浅的权重，这也能说明为什么剪枝前后模型的效果并没有受太大影响。

3.2 DessiLBI with Adam

参考 Adam[9]在 pytorch 上的实现，对 DessiLBI 优化器进行修改，使之结合 Adam 的更新方式。

使用结合后的优化算法在 MNIST 和 CIFAR10 数据集上进行测试。

首先是 CIFAR10。我选用了 VGG_BatchNorm 作为模型，交叉熵作为目标函数，以及以下超参数：

超参数	**取值**
batch size	128
epoch	20
learning rate	1e-3
kappa	1
interval	20
mu	20

之后对模型进行剪枝。首先对实验剪枝比例的影响。随机挑选一个参数层（第三个卷积层）：

从结果上来看，在剪枝比例一开始增加的时候，模型的准确率有小幅度的上升（0.1-1），说明此时模型剪枝的确实都是不太重要的权重。之后，随着模型稀疏性的提高，准确率开始下降。到了之前对 MNIST 剪枝的比例 80 时，模型几乎失去了分类能力。同样地，可视化其中 100 个卷积核的结果：

percent	accuracy
剪枝前	0.8265
0.1	0.8218
0.5	0.8234
1	0.8235
10	0.7321
20	0.6585
50	0.2753
80	0.112

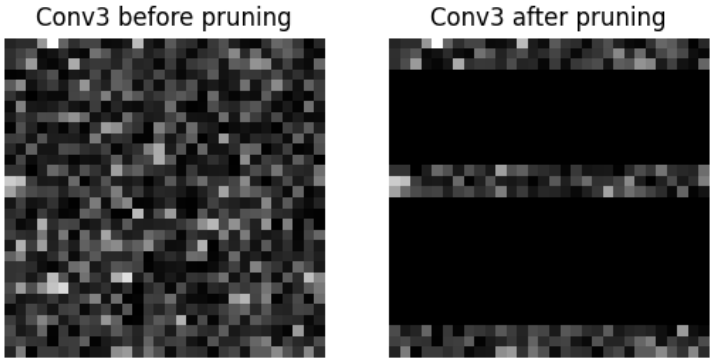

图 26: The third convolution before and after pruning with 80 percent

可以看到，该模型第三个卷积层的权重都很大，这也是为什么大幅剪枝之后效果骤降的原因。猜想是由于 CIFAR10 的图片比 MNIST 的图片结构更加复杂，模型需要记住更加多的特征导致的。

之后，对不同的隐藏层进行剪枝实验，对比其对模型效果的影响，使用的剪枝比例为 20。

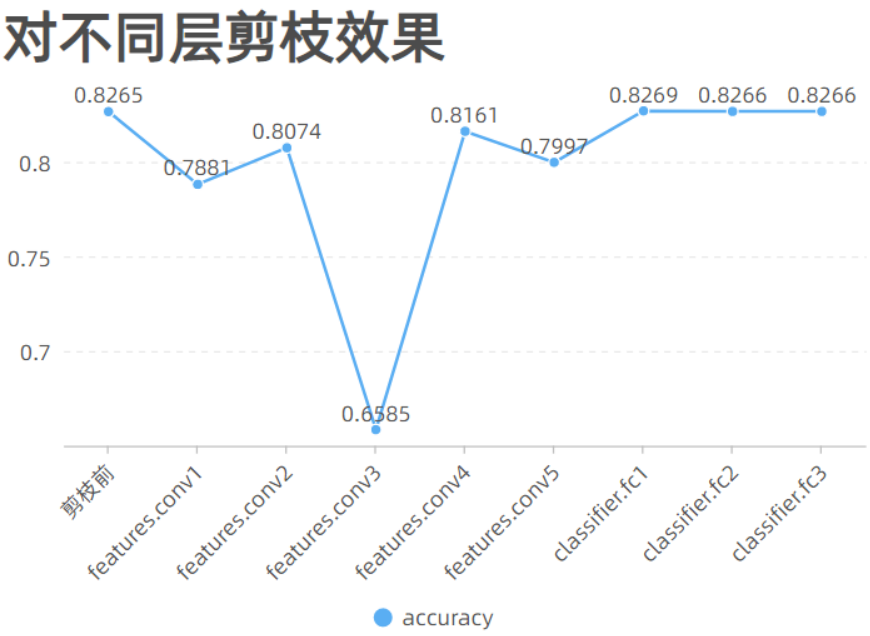

图 27: 以比例 20 对模型的不同层进行剪枝

可以看到对不同层的剪枝产生的效果不同。对卷积层剪枝的效果比较明显，对全连接层剪枝之后，模型效果甚至有微小的上升。这说明其实是可以兼顾模型稀疏性和准确性的，只要选择好剪枝的隐藏层和剪枝的比例，就可以既降低模型参数量，又不对模型的效果造成太大损失。

在 MNIST 数据集上测试结合后的优化器效果，模型结构和前面的报告一致。

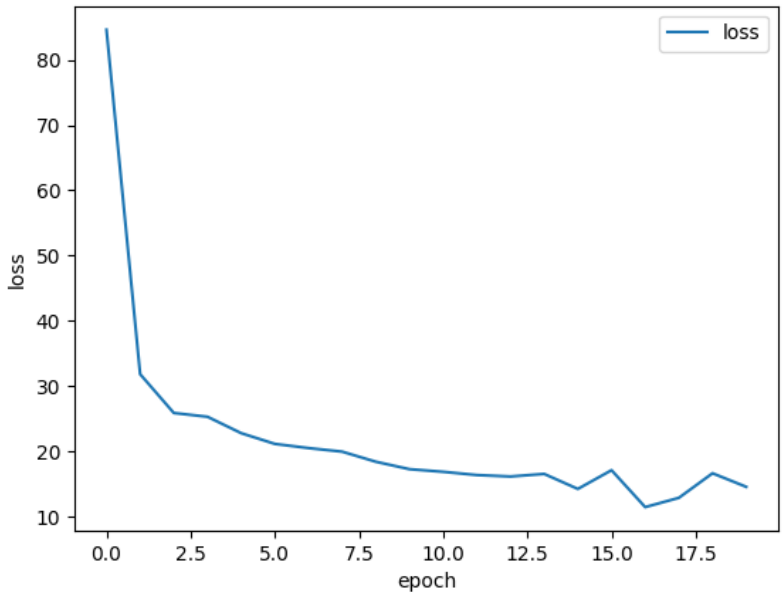

图 28: DessiLBI with Adam on MNIST

依旧是使用 80 的 percent 进行剪枝，剪枝之后结果如下：

可以看到比先前的 DessiLBI 效果还好。

	准确率
未剪枝	0.9819
conv3	0.9776
conv3 & fc1	0.9772

参考文献

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. 1.1

[2] ——, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015. 1.2.3

[3] R. Müller, S. Kornblith, and G. E. Hinton, "When does label smoothing help?" in *NeurIPS*, 2019. 1.3.1

[4] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. J. Yoo, "Cutmix: Regularization strategy to train strong classifiers with localizable features," *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6022–6031, 2019. 1.4.2

[5] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv: Learning*, 2017. 1.4.3

[6] R. R. Selvaraju, A. Das, R. Vedantam, M. Cogswell, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," *International Journal of Computer Vision*, vol. 128, pp. 336–359, 2019. 1.5.1

[7] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?" in *NeurIPS*, 2018. 2

[8] Y. Fu, C. Liu, D. Li, X. Sun, J. Zeng, and Y. Yao, "Dessilbi: Exploring structural sparsity of deep networks via differential inclusion paths," *ArXiv*, vol. abs/2007.02010, 2020. 3.1

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2015. 3.2

4 Appendix A

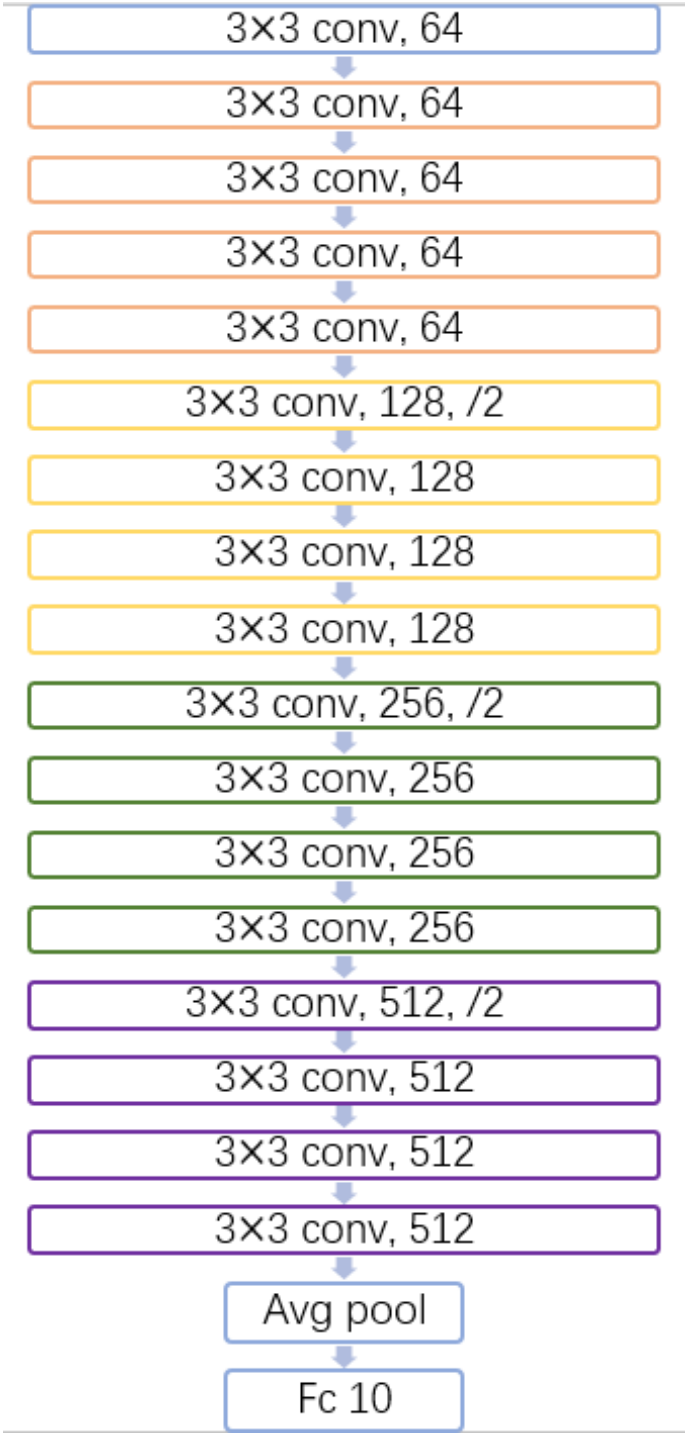

图 29: Plain Net structure

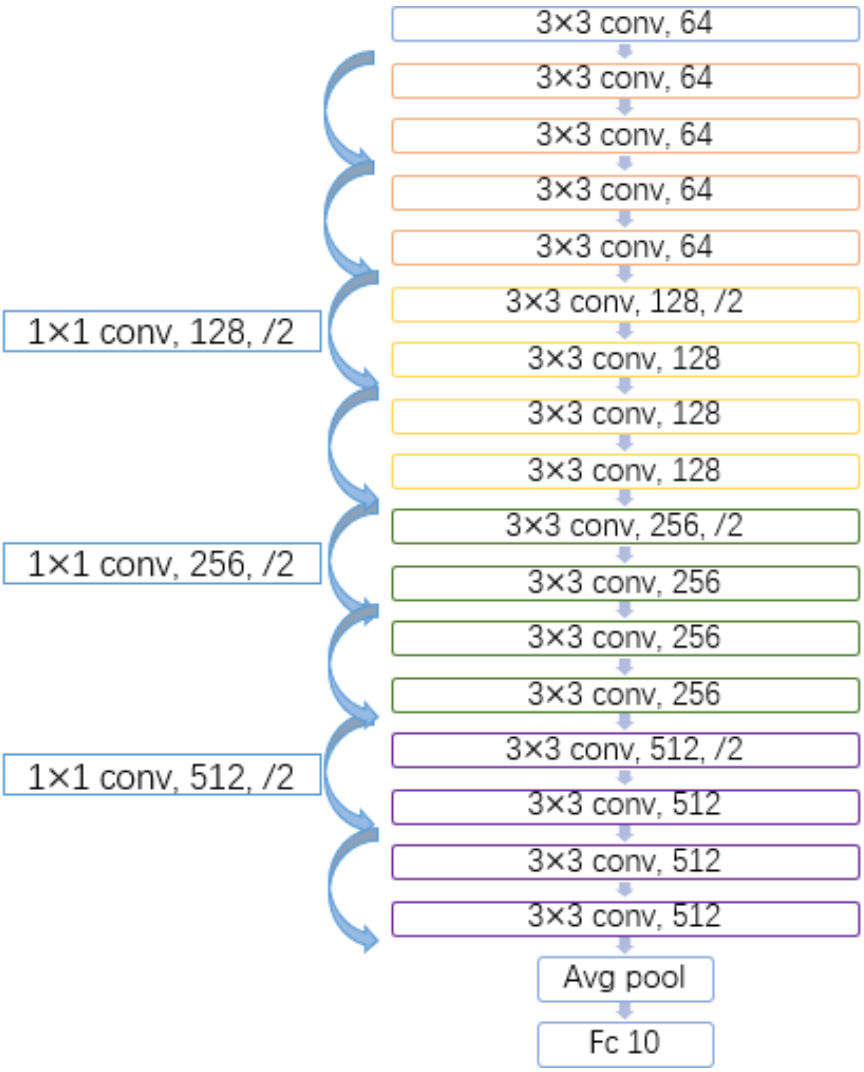

图 30: ResNet-18 structure

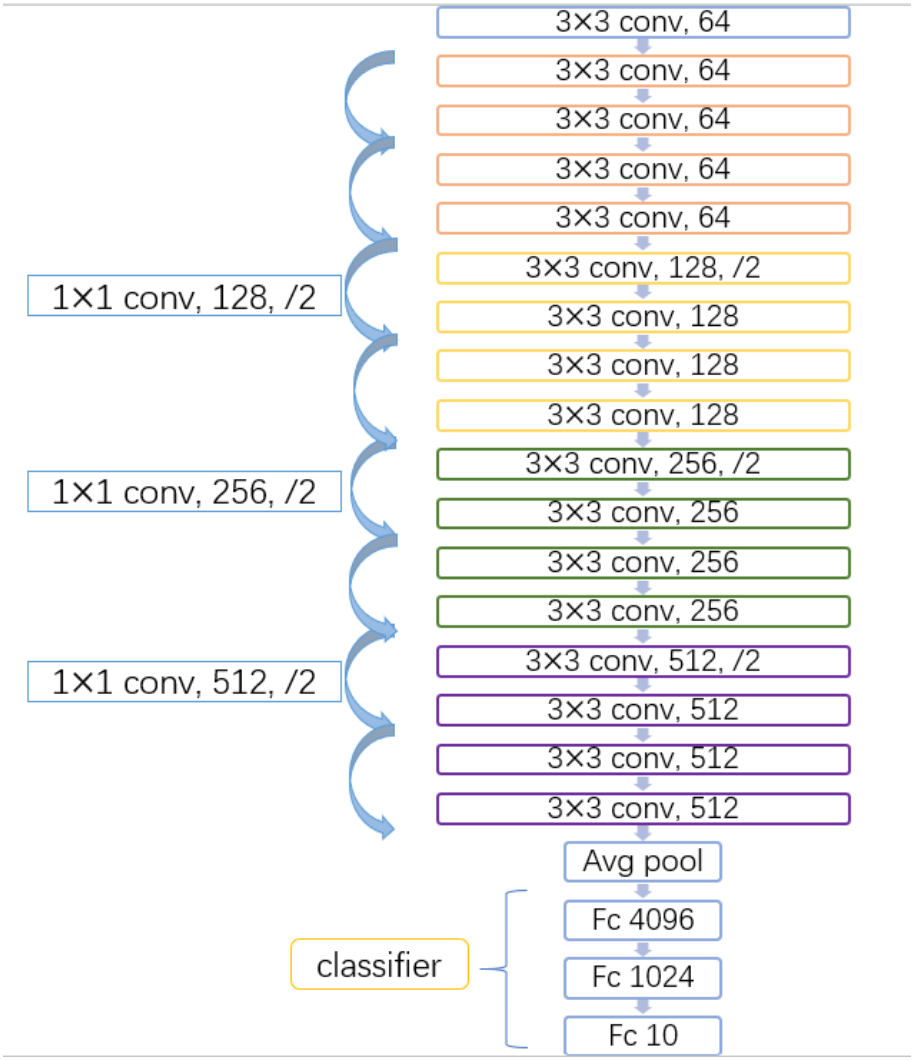

图 31: ResNet with classifier structure