NNDL-PJ2

姓名：尹志涛

学号：22307140055

task1-实现CIFAR-10分类神经网络

1. 项目介绍&完成情况

在task1中，我以CIFAR10作为数据集构建了一个分类神经网络。神经网络存在全连接网络、2D卷积层、2D汇聚层以及激活函数。此外，本次任务实现的神经网络包括**batch-norm layer，drop out layer，residual connection，充分保证了神经网络的泛化能力并提高了网络性能。** 在优化部分，我实验了三个不同神经元数量/滤波器数量对网络训练的影响，并测试了两种激活函数**ReLU & Leaky ReLU**，以及两种**Loss function: CrossEntropy & MSE**。

对于optimizer部分，我尝试了两种torch.optim中提供的优化器，并与不同的激活函数/损失函数结合，得到了8种配置下的网络性能。最后，我将卷积层的参数进行了可视化，并尝试对可视化的结果进行分析。

因此，task1的完成情况如下：

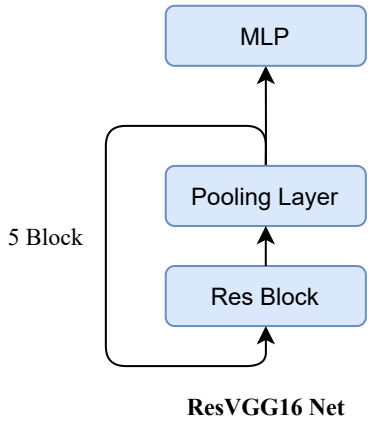
- 要求一： 包含Fully-Connected layer;2D convolutional layer;2D pooling layer;Activations; **完成**
- 要求二：网络至少包含：Batch-Norm layer;Drop out;Residual Connection;Others; **完成**
- 要求三：尝试不同配置下的同一网络架构对网络性能的影响：神经元数量/滤波器数量/损失函数/激活函数； **完成**
- 要求四：实验多种优化器对网络的影响；**完成**
- 要求五：对网络进行可视化，分析与解释。**完成**

2. 代码与权重链接：

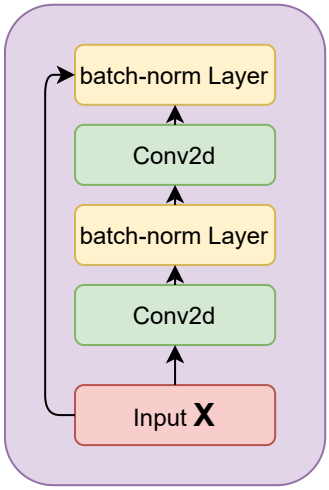
YinZT1/Project2-nndl-2025Spring

链接：https://pan.baidu.com/s/1I2cBCYb8qWi6ENY6Ga1ghA
提取码：1234

3. 网络架构



ResVGG16 Net



Residual Block

左侧为网络架构，右侧为Res Block的详细架构。

为了满足项目的要求，并且受到给定文件的启发，我尝试在task1实现VGG16卷积网络。 VGG-A是11层卷积层，包含11个权重层（8个卷积层 + 3个全连接层）。而VGG16是VGG一系列中综合表现最为出彩的一个架构。其包含16个权重层（13个卷积层 + 3个全连接层）

以下是VGG11和VGG16的层结构对比：

块	VGG11	VGG16
Block 1	1个卷积层（64通道）+ MaxPool	2个卷积层（64通道）+ MaxPool
Block 2	1个卷积层（128通道）+ MaxPool	2个卷积层（128通道）+ MaxPool
Block 3	2个卷积层（256通道）+ MaxPool	3个卷积层（256通道）+ MaxPool
Block 4	2个卷积层（512通道）+ MaxPool	3个卷积层（512通道）+ MaxPool
Block 5	2个卷积层（512通道）+ MaxPool	3个卷积层（512通道）+ MaxPool

块	VGG11	VGG16
全连接层	3层（4096-4096-1000）	3层（4096-4096-1000）

- **VGG11**：每个卷积块的卷积层数较少（1-1-2-2-2），总共8个卷积层。
- **VGG16**：每个卷积块的卷积层数较多（2-2-3-3-3），总共13个卷积层。
- 全连接层部分相同，都是3层（两层4096神经元，最后一层1000神经元，对应ImageNet的1000个类别）。

但传统VGG的实现并没有融合残差连接，因此我将实现的网络架构命名为：**ResVGG16**

3.1 ResVGG16的实现：

- **定义残差块**

残差块的实现思路基本是参照ResNet，

```
import torch.nn as nn

class ResidualBlock(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1, activation_fn_class=nn.ReLU):
        super(ResidualBlock, self).__init__()
        # 主路径：两个3x3卷积层 + 批量归一化 + 激活
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.activation = activation_fn_class()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # 残差连接：匹配维度和下采样
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
```

```
            )

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.activation(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out += self.shortcut(identity)
        out = self.activation(out)
        return out
```

- **实现网络**:

```
class ResVGG16(nn.Module):
    def __init__(self, block=ResidualBlock, num_blocks_list=[2,2,3,3,3],
                 num_classes=10, dropout_p=0.5, activation_fn_class=nn.ReLU,
                 initial_conv_out_channels=64, channels_list=[64,128,256,512,512],
                 classifier_hidden_neurons_list=[512,512]):
        super(ResVGG16, self).__init__()
        self.in_channels = initial_conv_out_channels

        # 初始卷积
        self.conv1 = nn.Conv2d(3, self.in_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(self.in_channels)
        self.initial_activation = activation_fn_class()

        # 5个残差块 + 池化
        self.layer1 = self._make_layer(block, channels_list[0], num_blocks_list[0], stride=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.layer2 = ...
        ...

        # 自适应池化 + 分类器
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        classifier_layers = [nn.Flatten()]
        fc_input_features = channels_list[-1] * block.expansion
        for num_neurons in classifier_hidden_neurons_list:
```

```
            classifier_layers += [nn.Linear(fc_input_features, num_neurons),
                                  activation_fn_class(), nn.Dropout(p=dropout_p)]
            fc_input_features = num_neurons
        classifier_layers.append(nn.Linear(fc_input_features, num_classes))
        self.classifier = nn.Sequential(*classifier_layers)


```

4 实验结果

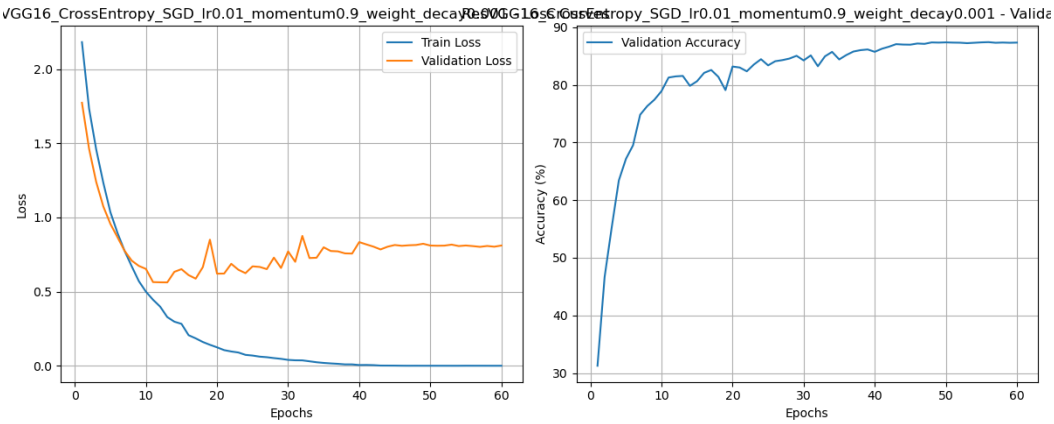
4.1 实验最优的超参数

训练轮数 (Epochs): 60。所有实验使用 `CrossEntropyLoss` 和 `SGD` 优化器，并启用了 `CosineAnnealingLR` 学习率调度器。

模型超参数设计	最高验证准确率 (%)
lr0.1_mom0.9_wd5e-4	10.0
lr0.05_mom0.9_wd5e-4	10.0
lr0.01_mom0.9_wd5e-4	**87.34 - best**
lr0.01_mom0.9_wd1e-4	87.4
lr0.01_mom0.9_wd1e-4	86.5

针对sgd（实际上adam也是如此），较高的学习率会导致参数始终无法更接近极值点。但是较低的参数的收敛速度过慢，通过一系列的训练我发现对于sgd的最佳学习率大约在0.01，而由于adam通常要求更低的学习率，通过测试我发现1e-3的learning rate都无法让adam收敛，最终我选取了learning rate = 1e-4作为adam优化器的学习率。

以下是ResVGG16在最优超参数以及sgd下的损失函数。



4.2 设计不同神经元和filter的神经网络

```
cfg_arch = {
    'VGG16_standard': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M'],
    'VGG16_light': [32, 32, 'M', 64, 64, 'M', 128, 128, 128, 'M', 256, 256, 256, 'M', 256, 256, 256, 'M'],
}
```

我设计了两个VGG16的残差网络，其中 **standard**就是最经典的VGG16，结合了残差块保证了网络的性能。 **light**则是削减了filter数量，并且在全连接网络中，削减了神经元个数。

```
fc_layers_cfg = {
    # (input_multiplier_from_last_conv_filter, [fc_hidden_dims])
    'standard': (512, [4096, 4096]),
    'light': (256, [1024, 512]),
}
```

4.3 最终的实验

我训练了24种配置的组合：

- 神经元/filter配置："VGG16_standard" "VGG16_light"
- 激活函数配置："relu" "leaky_relu" "tanh"
- 损失函数配置："cross_entropy" "nll_loss"
- 优化器配置："adam" "sgd"

训练的结果放在表格中：（由于24个实验对应至少24张图，因此我并没有将Loss图放在报告里）

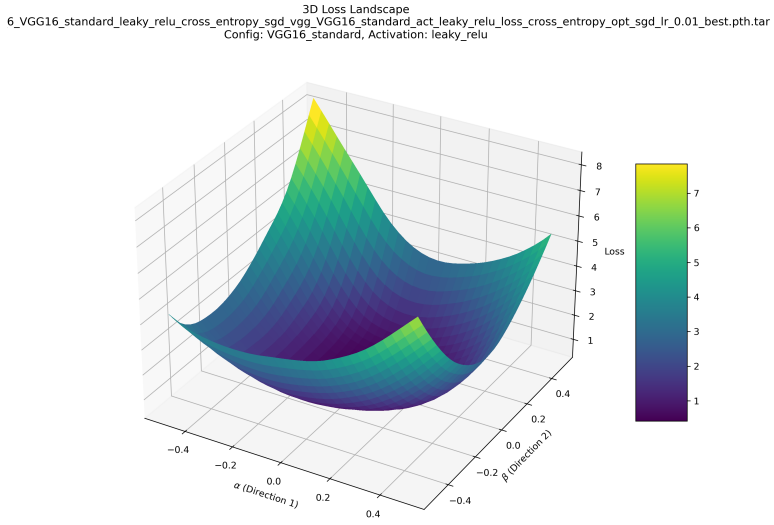
实验编号 (Run)	VGG 配置 (vgg_config)	激活函数 (activation)	损失函数 (loss_fn)	优化器 (optimizer)	最高验证准确率 (Best Validation Acc)
1	VGG16_standard	relu	cross_entropy	adam	86.680%
2	VGG16_standard	relu	cross_entropy	sgd	90.080%
3	VGG16_standard	relu	nll_loss	adam	86.680%
4	VGG16_standard	relu	nll_loss	sgd	90.080%
5	VGG16_standard	leaky_relu	cross_entropy	adam	87.860%
6	VGG16_standard	leaky_relu	cross_entropy	sgd	90.820%
7	VGG16_standard	leaky_relu	nll_loss	adam	87.860%

实验编号 (Run)	VGG 配置 (vgg_config)	激活函数 (activation)	损失函数 (loss_fn)	优化器 (optimizer)	最高验证准确率 (Best Validation Acc)
8	**VGG16_standard**	**leaky_relu**	**nll_loss**	**sgd**	**90.820%**
9	VGG16_standard	tanh	cross_entropy	adam	82.800%
10	VGG16_standard	tanh	cross_entropy	sgd	87.370%
11	VGG16_standard	tanh	nll_loss	adam	82.800%
12	VGG16_standard	tanh	nll_loss	sgd	87.370%
13	VGG16_light	relu	cross_entropy	adam	83.340%
14	VGG16_light	relu	cross_entropy	sgd	88.640%
15	VGG16_light	relu	nll_loss	adam	83.340%
16	VGG16_light	relu	nll_loss	sgd	88.640%
17	VGG16_light	leaky_relu	cross_entropy	adam	84.480%
18	VGG16_light	leaky_relu	cross_entropy	sgd	89.080%
19	VGG16_light	leaky_relu	nll_loss	adam	84.480%
20	**VGG16_light**	**leaky_relu**	**nll_loss**	**sgd**	**89.080%**
21	VGG16_light	tanh	cross_entropy	adam	81.440%
22	VGG16_light	tanh	cross_entropy	sgd	85.760%
23	VGG16_light	tanh	nll_loss	adam	81.440%
24	VGG16_light	tanh	nll_loss	sgd	85.760%

通过实验结果来看，leaky relu在两种网络架构面前都表现得十分好。而nllLoss和交叉熵并没有本质的区别（因为这两就是同一个东西），对于sgd和adam，在CIFAR10分类面前，sgd训练出的网络性能 **优于** adam。

4.4 filter可视化&损失函数地形化

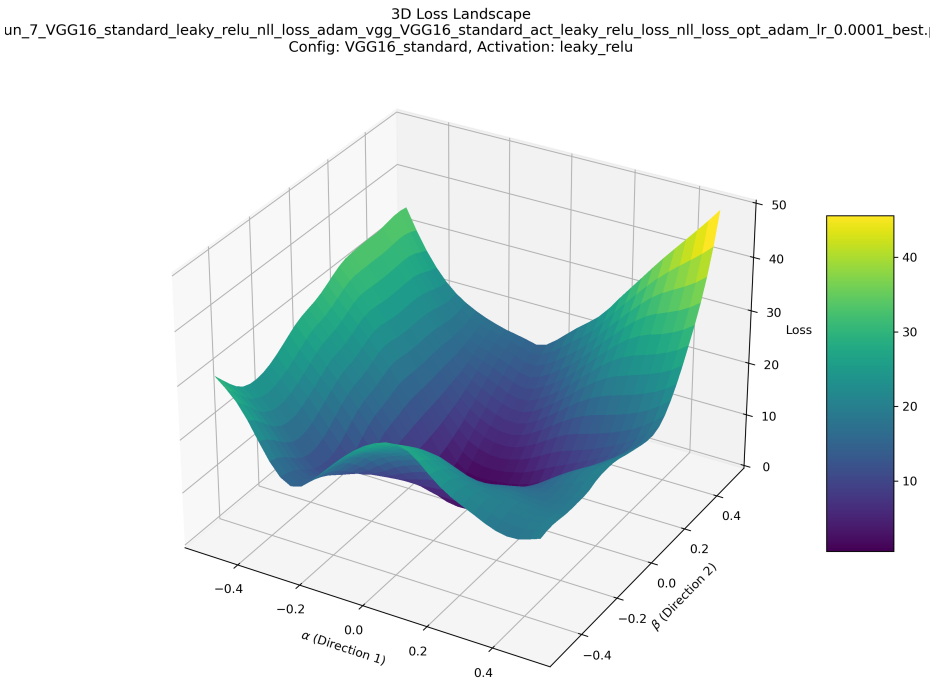
对最优配置VGG16-standard的filter以及损失函数的可视化如下：



可以看到损失函数的3d图的最小值附近是比较平滑的，也就是说较高的lr值更具有收敛的可能性，这与之前的实验结果相契合，因为在测试时，sgd的lr可以设置为0.01，然而当使用adam进行优化时，即使lr缩小10倍变成0.001，训练效率直接变成0，acc固定不变，从实验结果可以推测出来adam的三维LOSS图的极值点一定是比较尖锐的！

对于滤波器部分，我可视化了第一层卷积核的16个filter，从图中可以看出filter的颜色非常多样，这充分证明了作为第一层卷积层，训练的十分成功！因为CIFAR10是彩色图像，第一层卷积核需要对颜色进行一定程度的处理，所以滤波器中颜色非常复杂。此外，可以看到filter的中心颜色都比较暗淡，这可以有效测量出斑点。

为了验证对adam的损失函数的猜测，我可视化了adam对应的损失函数地形图，可以看出，我的猜测是准确的！：



这说明猜测是完全正确的。

task2-探索Batch-Norm对网络的作用

1. 项目介绍 & 完成情况

项目有以下几个要求：

- 分别对包含BatchNorm层和不包含Batch-Norm层的VGG-A进行训练，并且结合不同的optimizer（Adam，SGD）来测试VGG-A在CIFAR10数据集上进行图像分类的性能。
- 对BN进行分析，深度探索BN在哪些方面能够帮助网络/优化网络性能。可以通过以下角度进行分析：地形损失图，梯度变化。

完成情况：

- 我测试了Batch-Norm对VGG-A的影响。
- 我绘出了地形损失图，并对BatchNorm进行了深入的分析。

2. 代码与权重链接

YinZT1/Project2-nndl-2025Spring

链接：https://pan.baidu.com/s/1icAnMExdI1H6ZBAqfTEwDg

提取码：1234

3. 代码实现

VGG-A的基本框架已经给出：

3.1 残差块定义：

```
1. 残差块定义 (ResidualBlock)
pythonclass ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        # 主路径：两个3x3卷积层
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
                               stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # 跳跃连接：当维度不匹配时进行调整
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,
                          stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)  # 残差连接
        out = self.relu(out)
```

```
        return out
```

3.2 BatchNorm层

```
class VGG_A_Res_BN(nn.Module):
    def __init__(self, inp_ch=3, num_classes=10, init_weights=True):
        super().__init__()
        # 特征提取部分：5个阶段的残差块
        self.features = nn.Sequential(
            # Stage 1: 64通道
            ResidualBlock(inp_ch, 64),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Stage 2: 128通道
            ResidualBlock(64, 128),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Stage 3: 256通道，两个残差块
            ResidualBlock(128, 256),
            ResidualBlock(256, 256),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Stage 4: 512通道，两个残差块
            ResidualBlock(256, 512),
            ResidualBlock(512, 512),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Stage 5: 512通道，两个残差块
            ResidualBlock(512, 512),
            ResidualBlock(512, 512),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        # 分类器部分：全连接层 + Dropout + BN
        self.classifier = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(512 * 1 * 1, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(True),
            nn.Dropout(0.5),
            nn.Linear(512, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(True),
            nn.Linear(512, num_classes)
        )

```

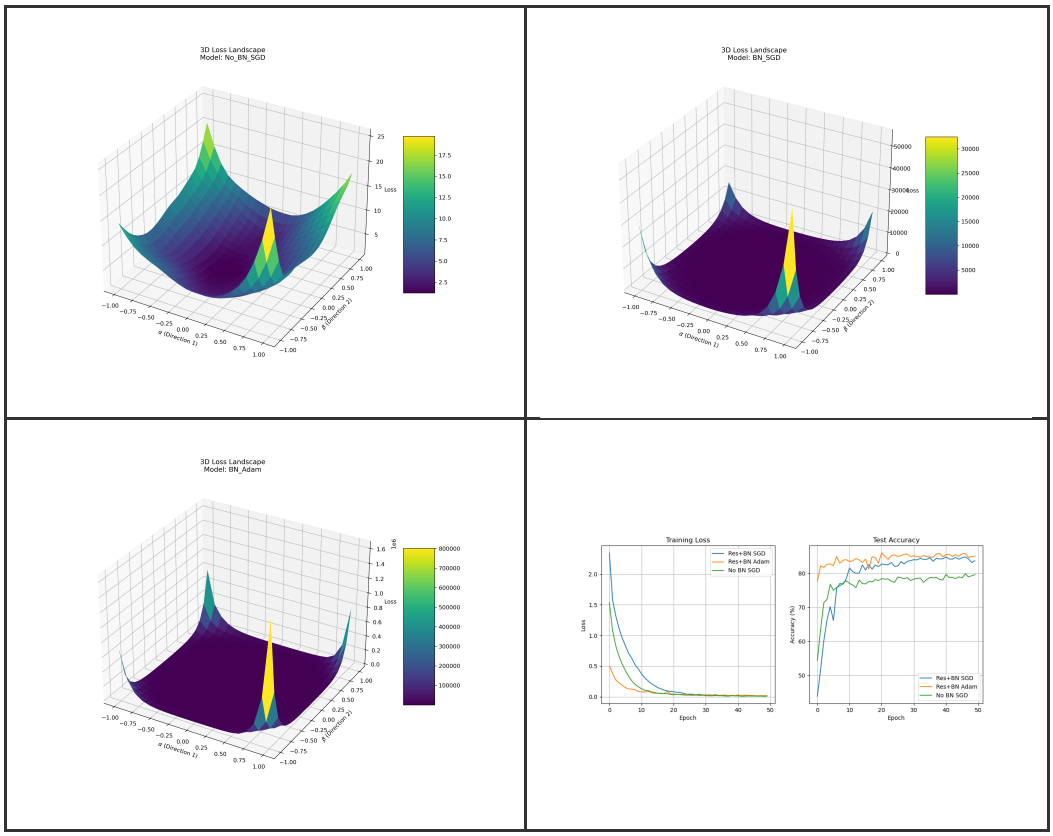
```
    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x.view(-1, 512 * 1 * 1))
        return x
```

实验结果

CONFIG	acc	收敛时间(第k个epoch收敛)
VGGA(不含BN) adam lr 0.001	80.77%	20
VGGA(含BN) adam lr 0.001	83.29%	4
VGGA(含BN与残差块) adam lr 0.001	**86.0%**	**13-best**
VGGA(含BN) sgd lr 0.01	81.50%	18
VGGA(含BN与残差块) sgd lr 0.01	84.64%	15

可以看出最优的配置是adam优化器结合BN与残差连接，在测试集上的acc高达 **86.0%**。

可视化结果



结果分析以及Batch-Norm功能解析：

从训练曲线来看，**Batch Normalization 显著加速了模型的收敛速度**，并且帮助模型达到了**更高的测试准确率**。无论使用 SGD还是 Adam 优化器，加入 BN 层后模型的性能都有明显提升。

No_BN_SGD 的损失地形

- 该地形图的损失值 (Loss) 范围非常大，颜色条显示最高可达 2.0×106 (z轴标签甚至标到 4.0×106)。
- 地形呈现出一个**非常狭窄且极其陡峭的“峡谷”**，谷底代表了低损失区域。离开这个狭窄区域后，损失值会急剧上升。

BN_SGD 的损失地形

- 该地形图的损失值范围相对较小，颜色条显示最高为 20 (z轴标签最高为 25)。
- 与 No_BN_SGD 相比，这个损失地形看起来**更宽阔、坡度更平缓一些**。损失值的变化没有那么剧烈。

这说明Batch Normalization能够平滑损失函数，这意味着梯度方向能更稳定地指向全局或一个好的局部最小值，这一方面可以加速收敛，因为每一次梯度的方向都更加正确，另一方面，对于固定的lr值（如果不实时更新的话），Batch Normalization的应用会让训练结果不容易“**停滞**”，即不会因为lr过大导致跳过最优点，也不容易因lr过小而被困在鞍点中。

Batch Normalization 层的作用总结

综合以上分析，Batch Normalization 层在 VGG-A 模型训练中起到了以下关键作用：

1. **加速训练收敛：** 从训练损失和测试准确率曲线可以看出，加入 BN 层的模型收敛速度远快于没有 BN 层的模型。这是因为 BN 层通过规范化层输入，使得损失函数的梯度更稳定和可预测。
2. **提升模型性能：** 加入 BN 层的模型在测试集上达到了更高的准确率，说明 BN 层有助于模型学习到更好的特征表示，并可能具有一定的正则化效果，提高了模型的泛化能力。
3. 改善损失函数的地形：
    - BN 层使得损失函数的优化路径更加平滑和直接。虽然 `BN_SGD` 的可视化地形在全局看起来陡峭，但它可能形成了一个引导优化器高效到达最优解的清晰路径。
    - 相比之下，没有 BN 层的模型其损失地形可能包含更多使优化变慢的区域 (如平坦区域或不规则的梯度变化)，导致收敛缓慢且性能较差。

4. **稳定训练过程：** BN 层降低了模型对初始化参数的敏感性，并允许使用更高的学习率，从而使训练过程更加稳定。