PJ2实验报告

一.基础神经网络搭建

在第一部分，我们按照PJ要求搭建基础的用于CIFAR-10分类预测的神经网络。具体训练代码详见train.py，具体测试代码详见test.py。

定义数据集

我们需要定义CIFAR-10数据集的类用于pytorch的训练与预测。为了格式统一，我们需要将图片像素归一化

```
class CIFAR10(Dataset):
    def __init__(self, data_dir, batches=[1,2,3,4,5], transform=None):

        #存储原始数据和标签
        data_list = []
        label_list = []
        for b in batches:
            path = os.path.join(data_dir, f'data_batch_{b}')
            with open(path, 'rb') as f:
                batch = pickle.load(f, encoding='bytes')
                data = batch[b'data']
                labels = batch[b'labels']
            data_list.append(data)
            label_list += labels

        #图片像素归一化
        self.data = np.vstack(data_list).reshape(-1, 3, 32,
32).astype(np.float32) / 255.0
        self.labels = np.array(label_list, dtype=np.int64)
        self.transform = transform

    #定义标签长度的方法
    def __len__(self):
        return len(self.labels)

    #定义获取某个图片的数据和标签的方法
    def __getitem__(self, idx):
        img, label = self.data[idx], self.labels[idx]
        if self.transform:
            img = self.transform(img)
        return img, label
```

定义神经网络结构

我们定义一个简单的神经网络结构用于训练CIFAR-10数据集。具体结构为：卷积层+残差层+卷积层+dropout层+FC层。卷积层使用了Batchnorm方法和ReLU为激活函数，以及maxpooling作为池化方法。

```
class CIFAR10Net(nn.Module):
    def __init__(self):
        super().__init__()
```

```
        #卷积层1
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )

        #residual block层
        self.resblock = ResidualBlock(32)

        #卷积层2
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )

        #drop out层
        self.dropout = nn.Dropout(0.5)

        #full connection层
        self.fc = nn.Linear(64*8*8, 10)

    def forward(self, x):
        x = self.layer1(x)
        x = self.resblock(x)
        x = self.layer2(x)
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        return self.fc(x)
```

在Pytorch中并没有residual connection block的定义，我们需要自己编写。这里skip connection的方法我们采用直接将输入x添加到输出作为最终的输出函数。

```
# 自定义residual connection block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, in_channels, kernel_size=3,
padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(in_channels)
        self.conv2 = nn.Conv2d(in_channels, in_channels, kernel_size=3,
padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(in_channels)

    def forward(self, x):
        identity = x
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))

        #output=F(x)+x,跳跃连接直接加上输入本身
        out += identity
```

```
        return F.relu(out)
```

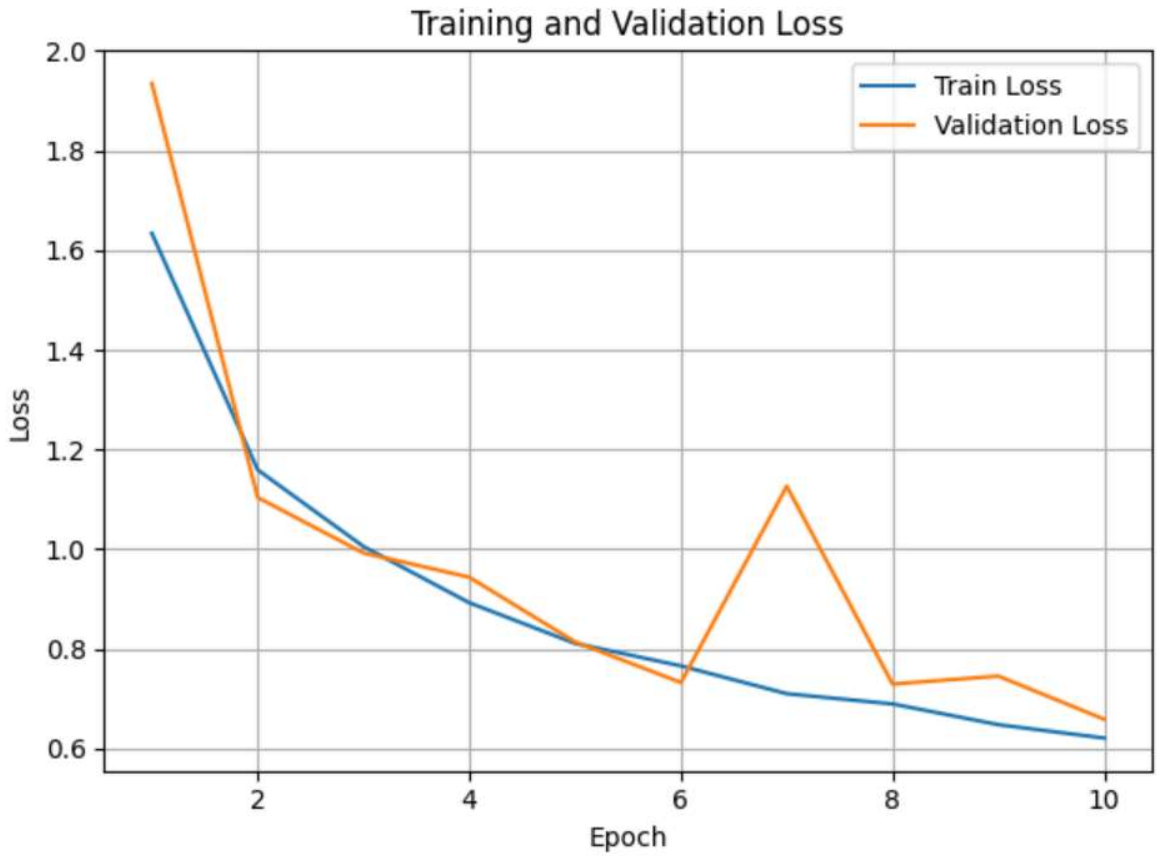
训练神经网络

基础网络的训练我们使用crossentropy作为损失函数，以及adam为优化器，训练轮数为10。我们对训练集采用了test-validation separation，同时记录训练集和验证集的损失，并采用early stopping策略防止训练过拟合。

```
    #损失函数
    criterion = nn.CrossEntropyLoss()

    #优化器
    optimizer = torch.optim.Adam(model.parameters(), lr=3e-3)

    # 训练配置参数
    num_epochs = 10 #训练轮数
    patience = 3  # early stopping 容忍数量
    best_model_path = os.path.join(model_dir, 'best_model.pth') #保存路径
    best_val_loss = float('inf')
    wait = 0
    train_losses = []
    val_losses = []
    for epoch in range(1, num_epochs+1):
        model.train()

        #训练阶段
        train_loss = 0.0
        for imgs, labels in train_loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_loss += loss.item() * imgs.size(0)
        train_loss /= train_size

        # 验证阶段
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for imgs, labels in val_loader:
                imgs, labels = imgs.to(device), labels.to(device)
                outputs = model(imgs)
                loss = criterion(outputs, labels)
                val_loss += loss.item() * imgs.size(0)
        val_loss /= val_size

        # 添加损失到绘图数据中
        train_losses.append(train_loss)
        val_losses.append(val_loss)
        print(f'Epoch {epoch}/{num_epochs} - Train Loss: {train_loss:.4f}, Val
Loss: {val_loss:.4f}')

        # early stopping策略
```

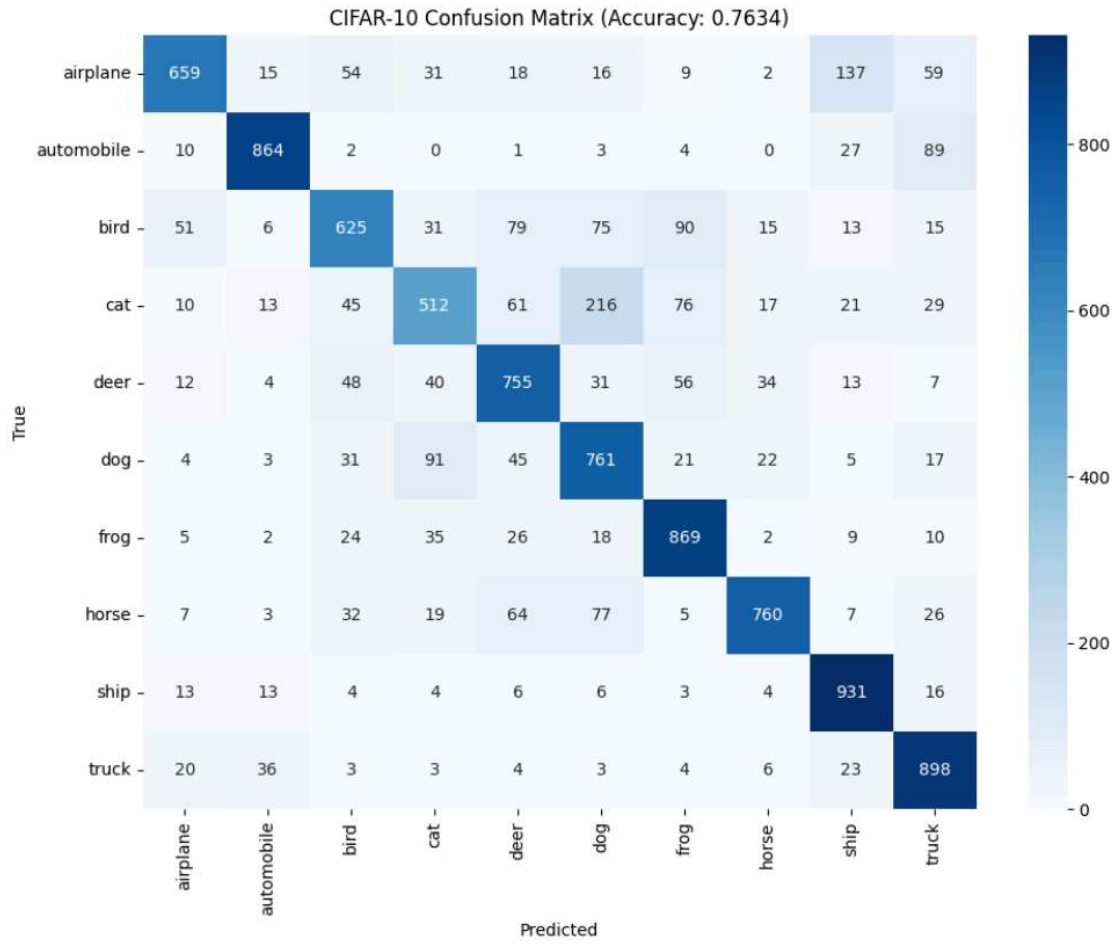
```
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict(), best_model_path)
            print(f'Best model updated at epoch {epoch}, val_loss:
{best_val_loss:.4f}')
            wait = 0
        else:
            wait += 1
            if wait >= patience:
                print('Early stopping triggered')
                break
```

我们对训练结果进行了可视化：



测试神经网络

在测试神经网络时，我们只需要注意按照训练时的方法定义数据集和网络结构进行预测即可。我们绘制了混淆矩阵可视化预测的结果：



二、优化基础神经网络

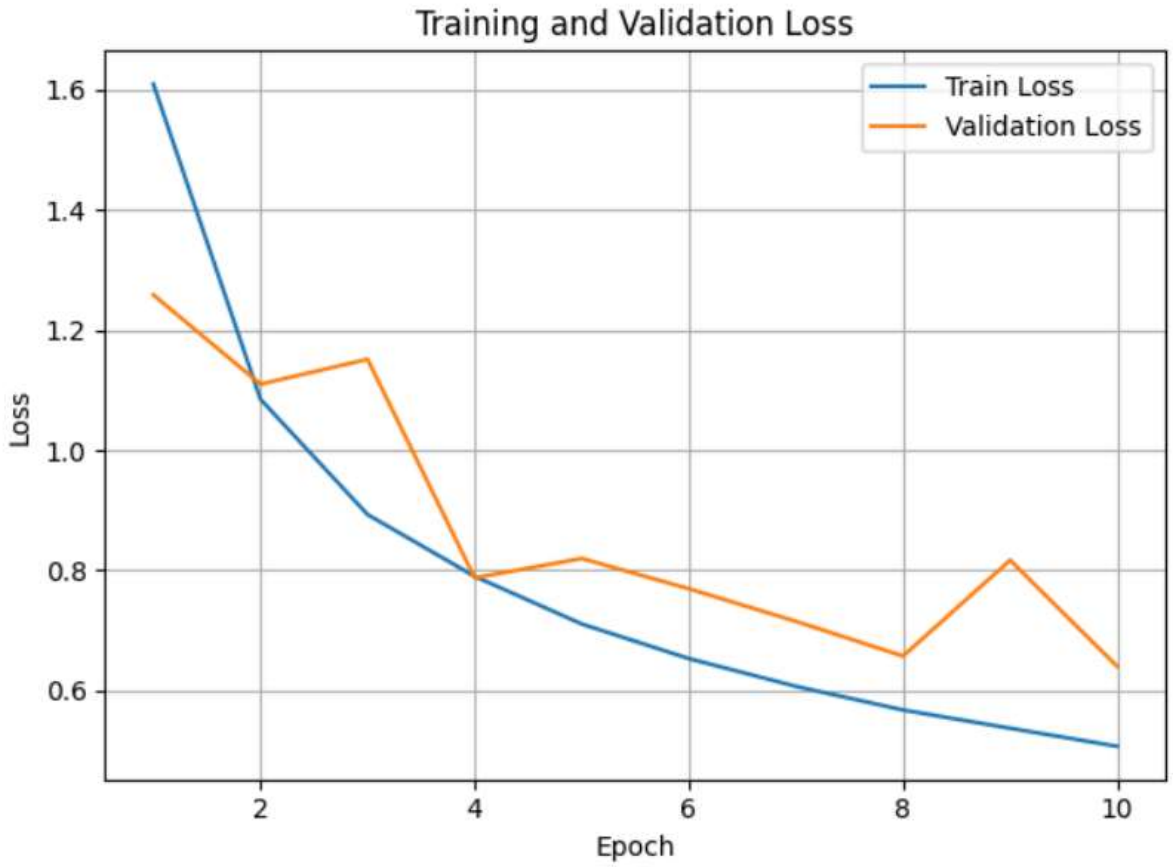
Try different number of neurons

我们额外增加一个卷积层进行训练：注意全连接层的维度也要对应修改

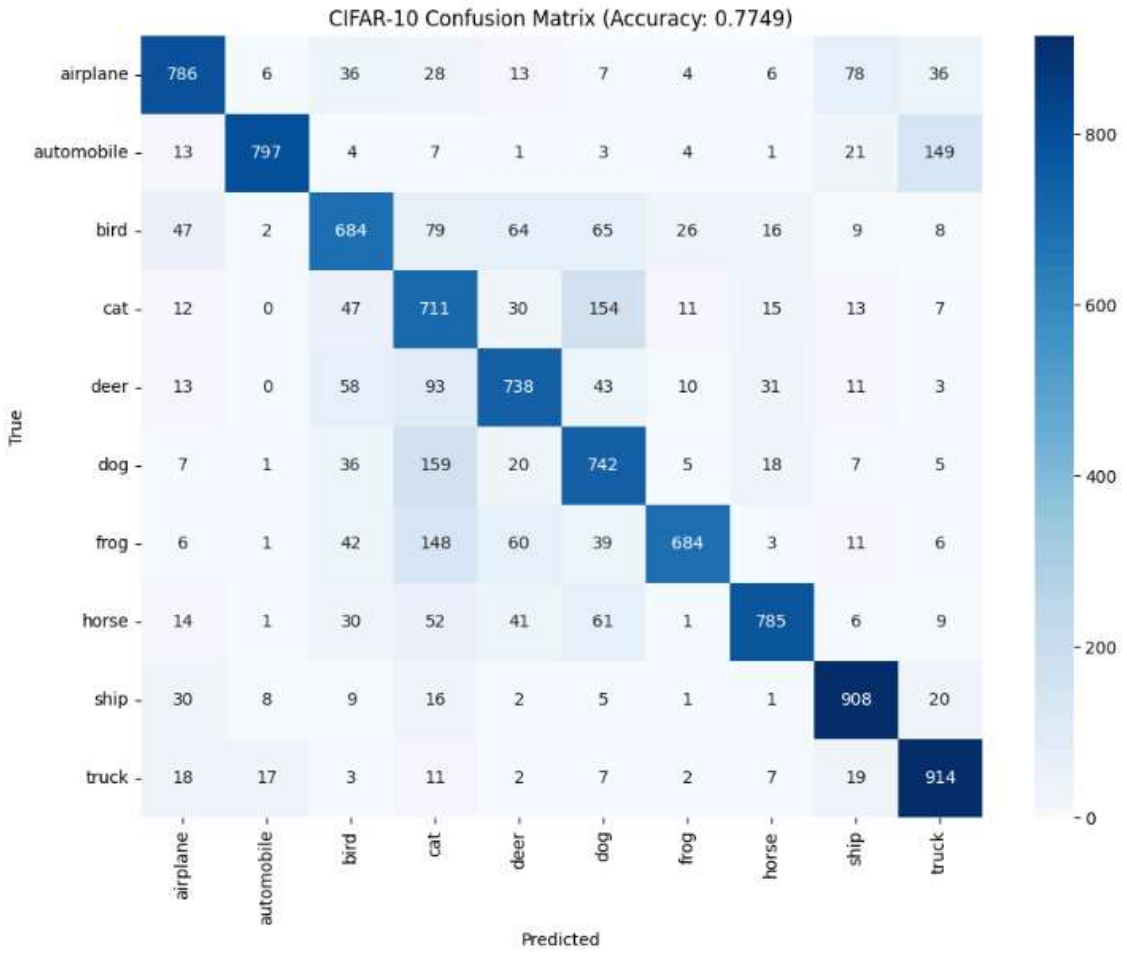
```
class CIFAR10Net(nn.Module):
    def __init__(self):
        super().__init__()

        # 卷积层1
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )

        # Residual block层
        self.resblock = ResidualBlock(32)

        # 卷积层2
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )
```

```
        # 卷积层3
        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )

        # dropout层
        self.dropout = nn.Dropout(0.5)

        # 全连接层
        self.fc = nn.Linear(128 * 4 * 4, 10)

    def forward(self, x):
        x = self.layer1(x)
        x = self.resblock(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        return self.fc(x)
```

训练结果如下：与原始基础模型相比，训练损失有显著下降



预测结果如下：与基础模型相比，预测准确率也有提升

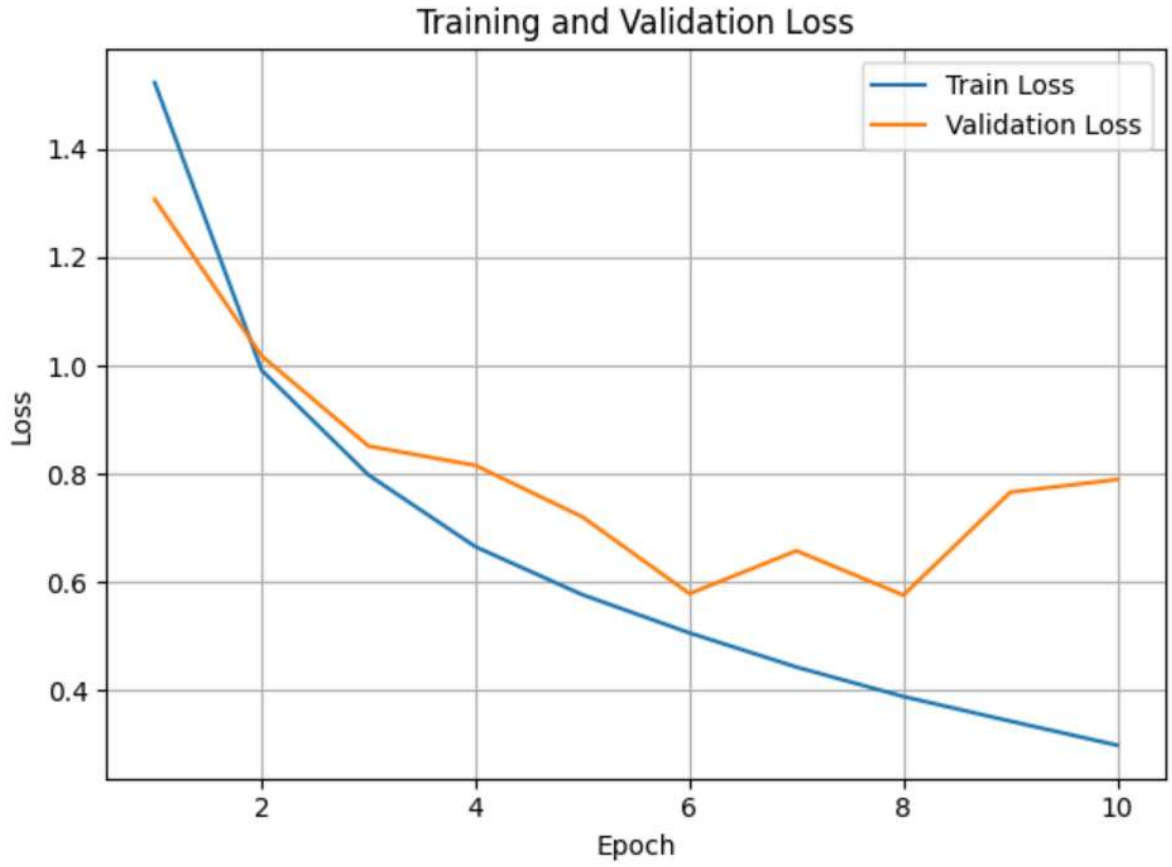


我们再增加一组卷积层:

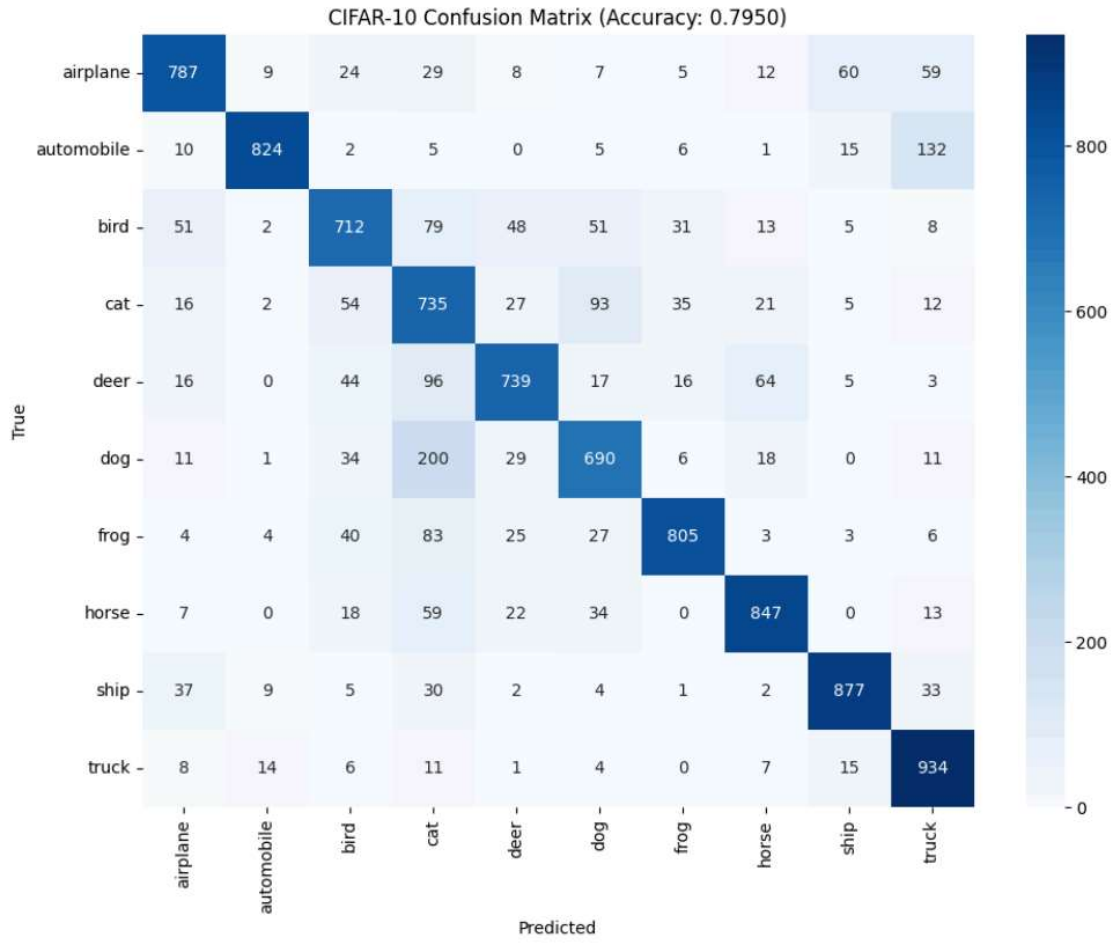
```
class CIFAR10Net(nn.Module):
    def __init__(self):
        super().__init__()

        # 卷积层1
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )

        # ResidualBlock 1
        self.resblock1 = ResidualBlock(32)

        # 卷积层2
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )

        # 卷积层3
        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 128, 3, padding=1),
            nn.BatchNorm2d(128),
```

```
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )

        # 卷积层4
        self.layer4 = nn.Sequential(
            nn.Conv2d(128, 256, 3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2)
        )

        # dropout层
        self.dropout = nn.Dropout(0.5)

        # 全连接层
        self.fc = nn.Linear(256 * 2 * 2, 10)

    def forward(self, x):
        x = self.layer1(x)
        x = self.resblock1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        return self.fc(x)
```

训练结果如下：训练损失再次减小，但验证损失已经开始呈现上升趋势，不过还没达到early stopping的触发步数。随着网络结构的复杂，训练轮数可以适当减小防止过拟合。



预测结果如下：预测效果也有所提升

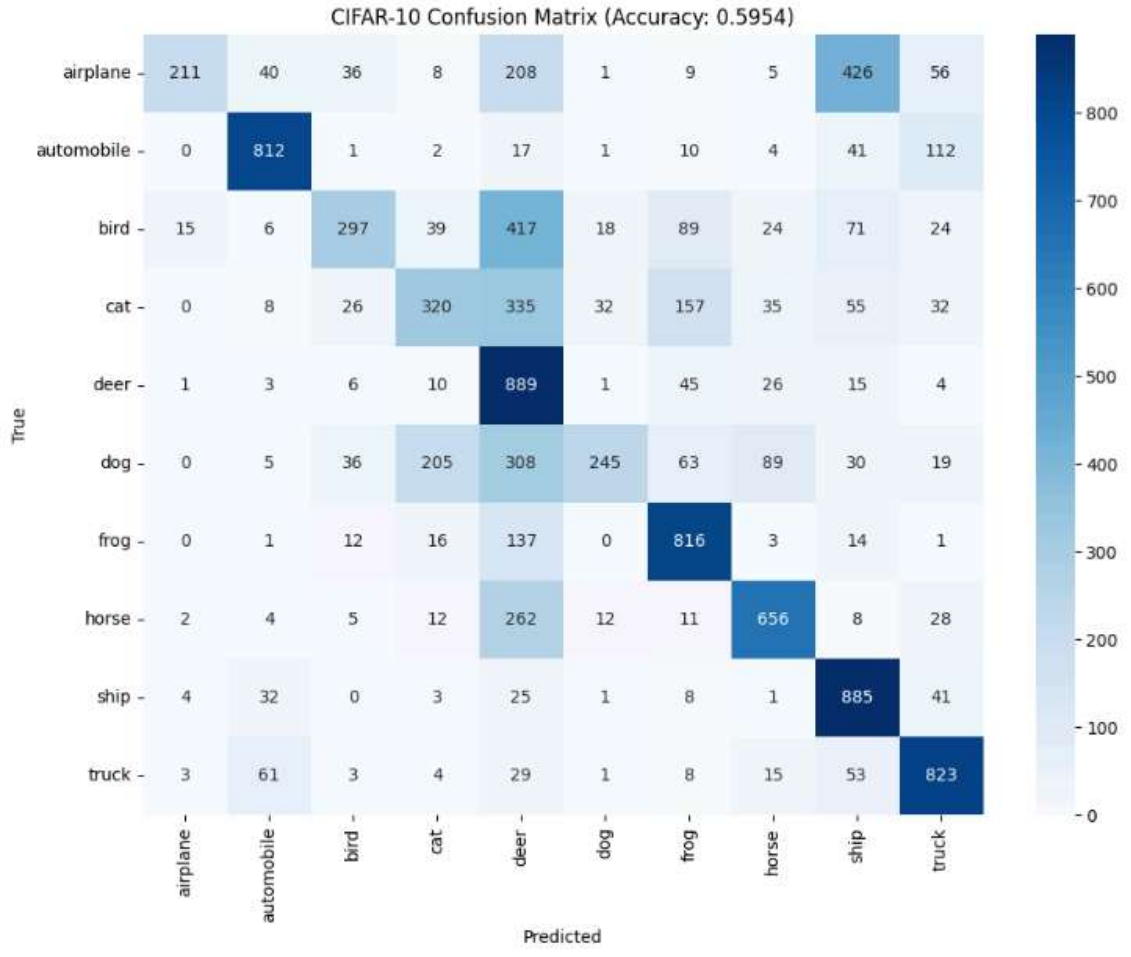


Try different activations

基础模型使用的是ReLU。这里我们首先改为采用sigmoid激活函数。

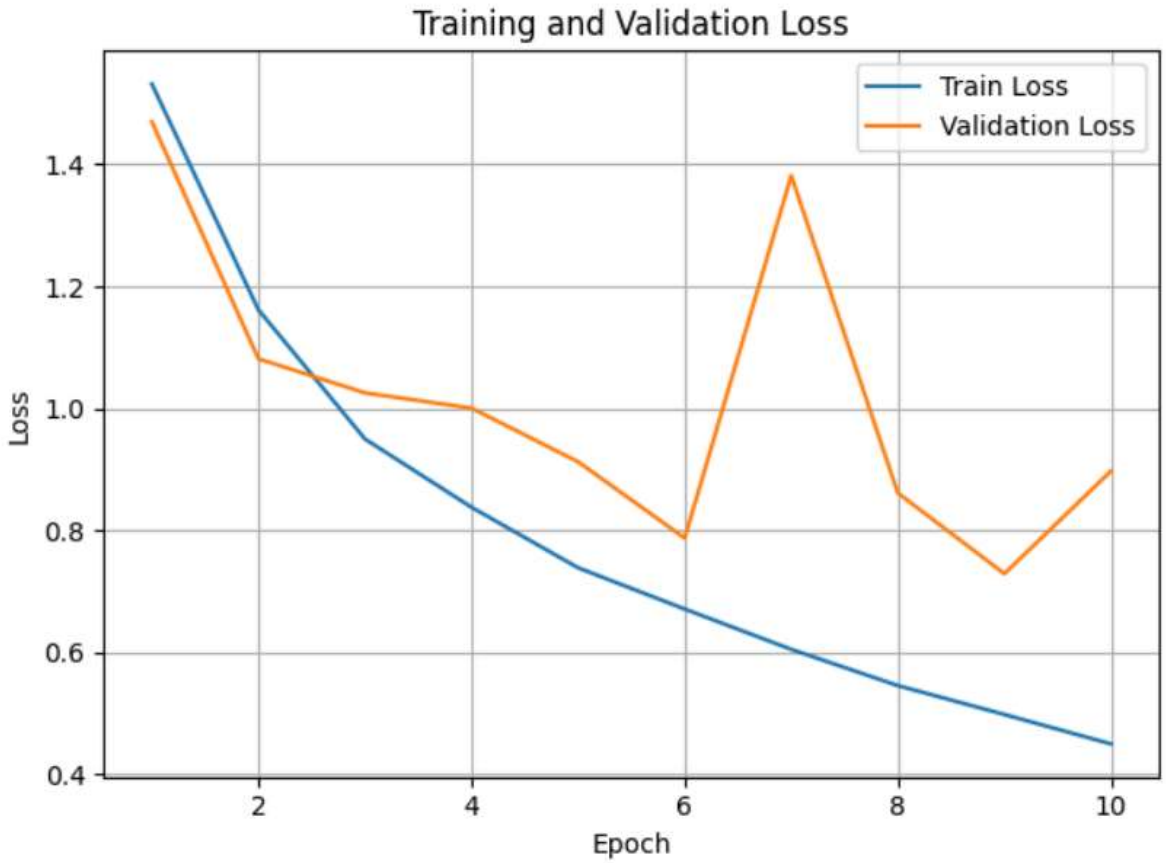
训练结果如下：Sigmoid激活函数训练效果不如ReLU

预测结果如下：Sigmoid激活函数预测效果也不如ReLU

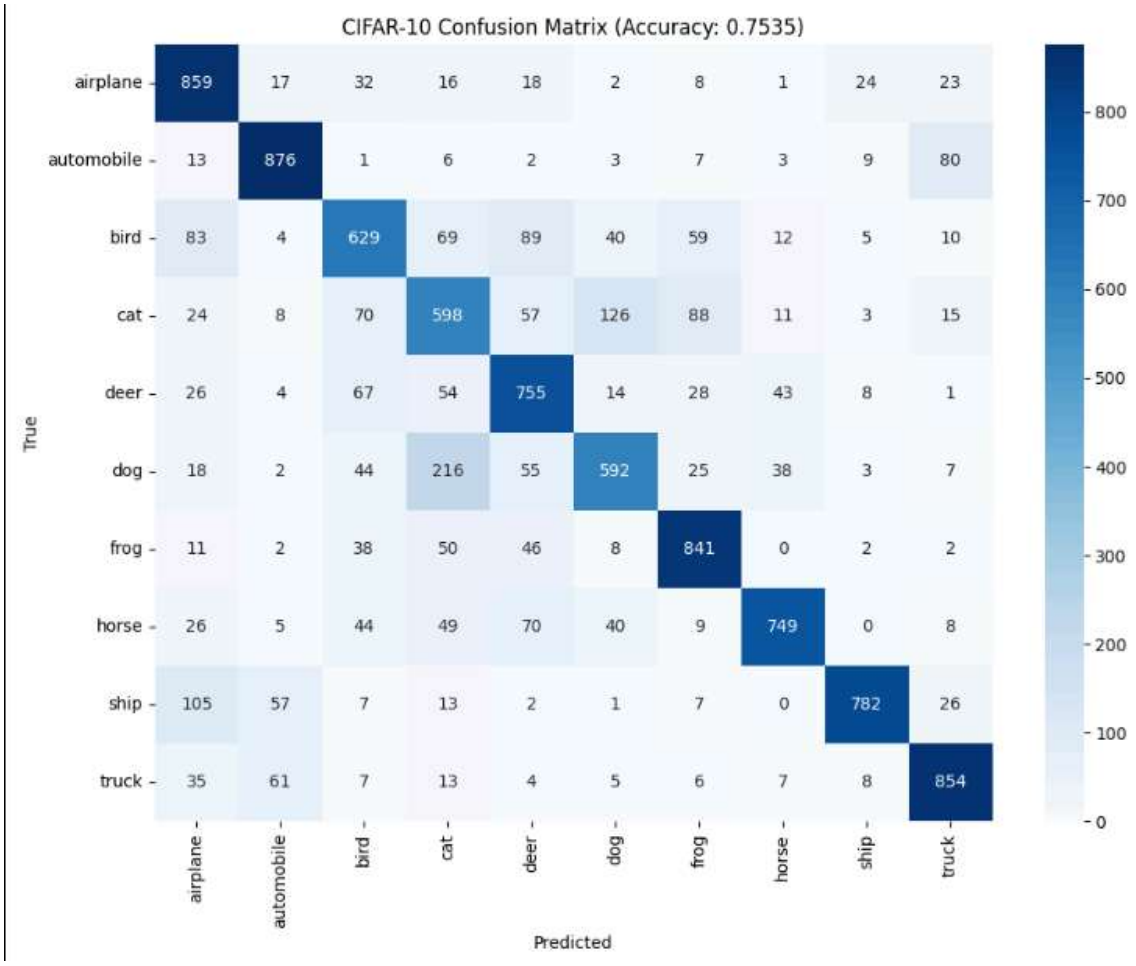


接着我们尝试tanh激活函数。

训练结果如下：tanh激活函数训练效果略逊色于ReLU，但比sigmoid优秀很多。



预测效果如下：同理预测效果略逊色于ReLU，但优秀于Sigmoid。

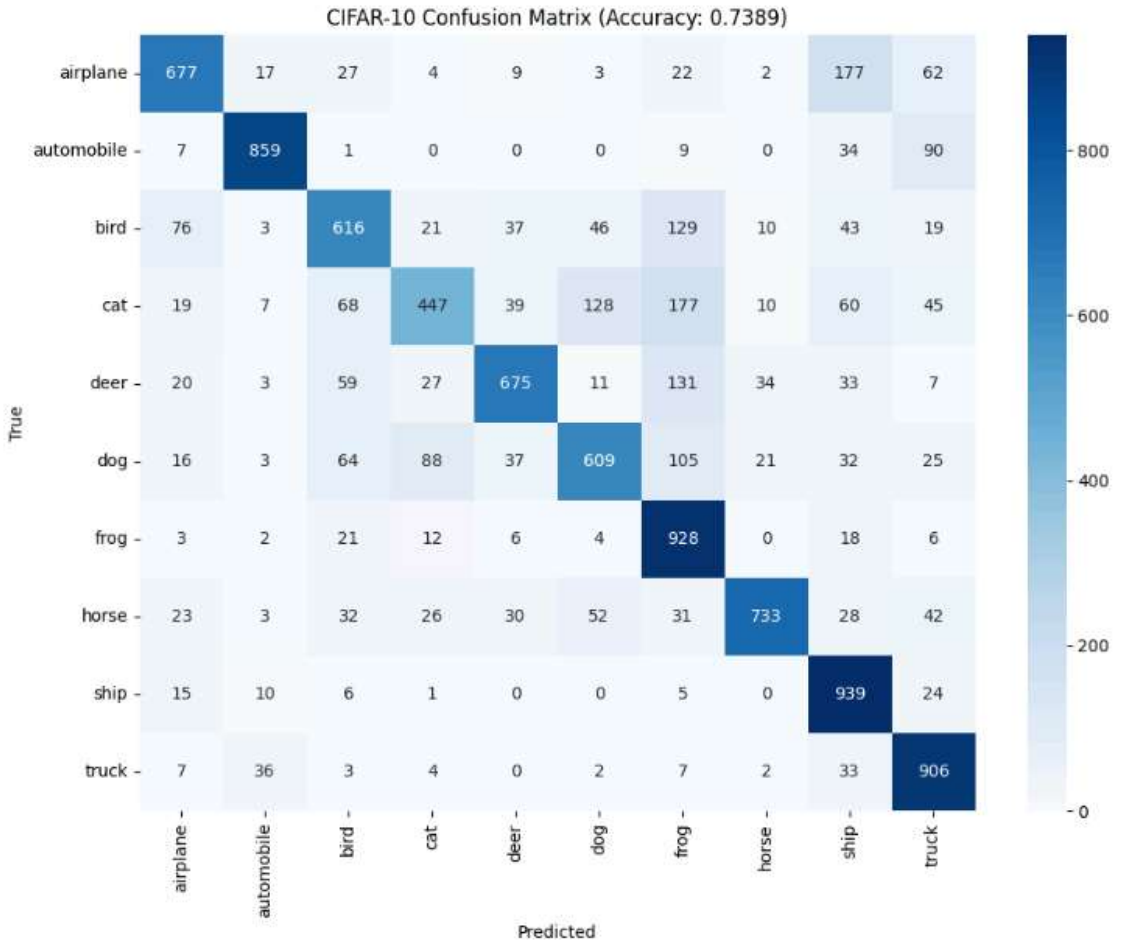


综合上述实验，在tanH，sigmoid，ReLU三个激活函数中，ReLU是最适合的激活函数。

Try different loss functions

尝试使用nn.BCEWithLogitsLoss()。该损失函数是将多分类问题视作多个二分类问题计算损失，需要将标签变为one-hot类型。这里我们不再比较训练效果，因为二者本身计算方式不同，数值并没有可比性。

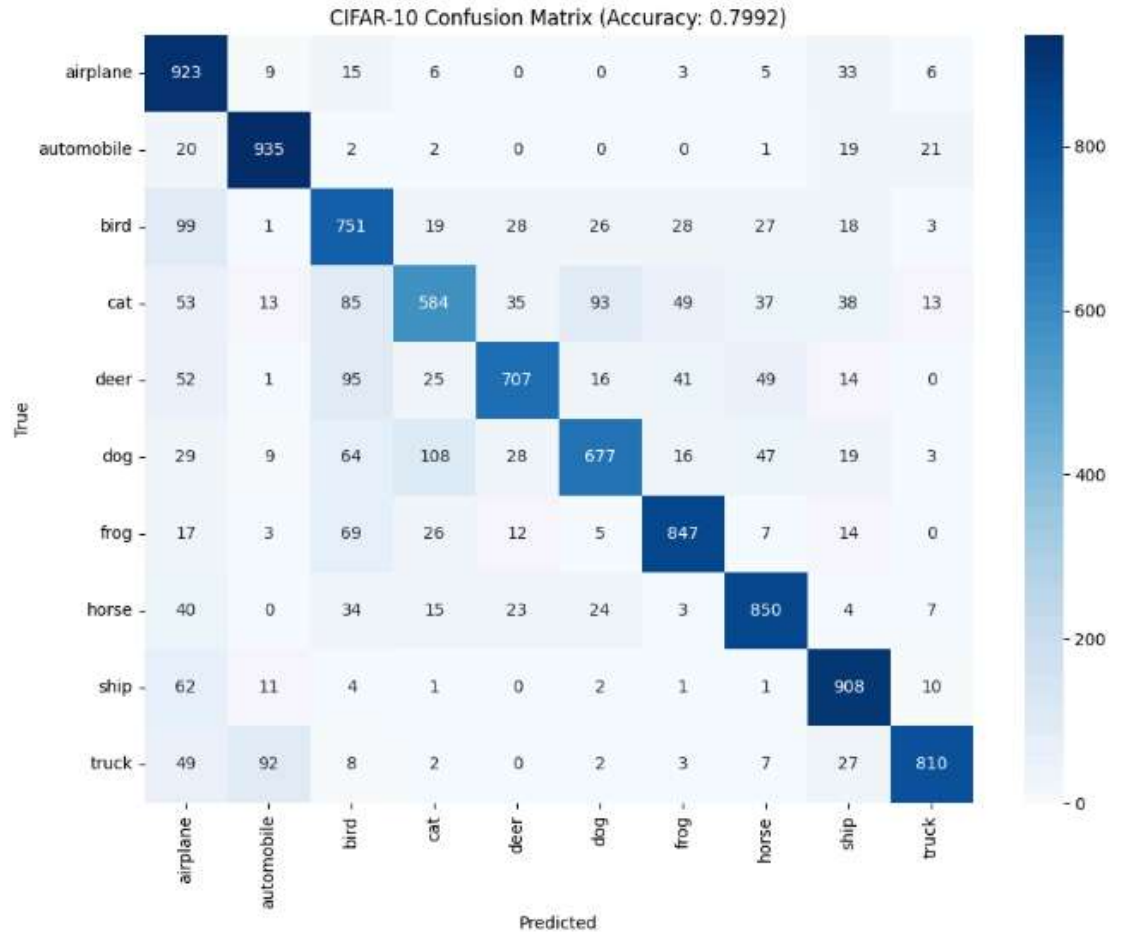
预测结果如下：预测效果却不如交叉熵损失，分析原因可能是该损失函数计算考虑的比较局部，容易导致过拟合现象。



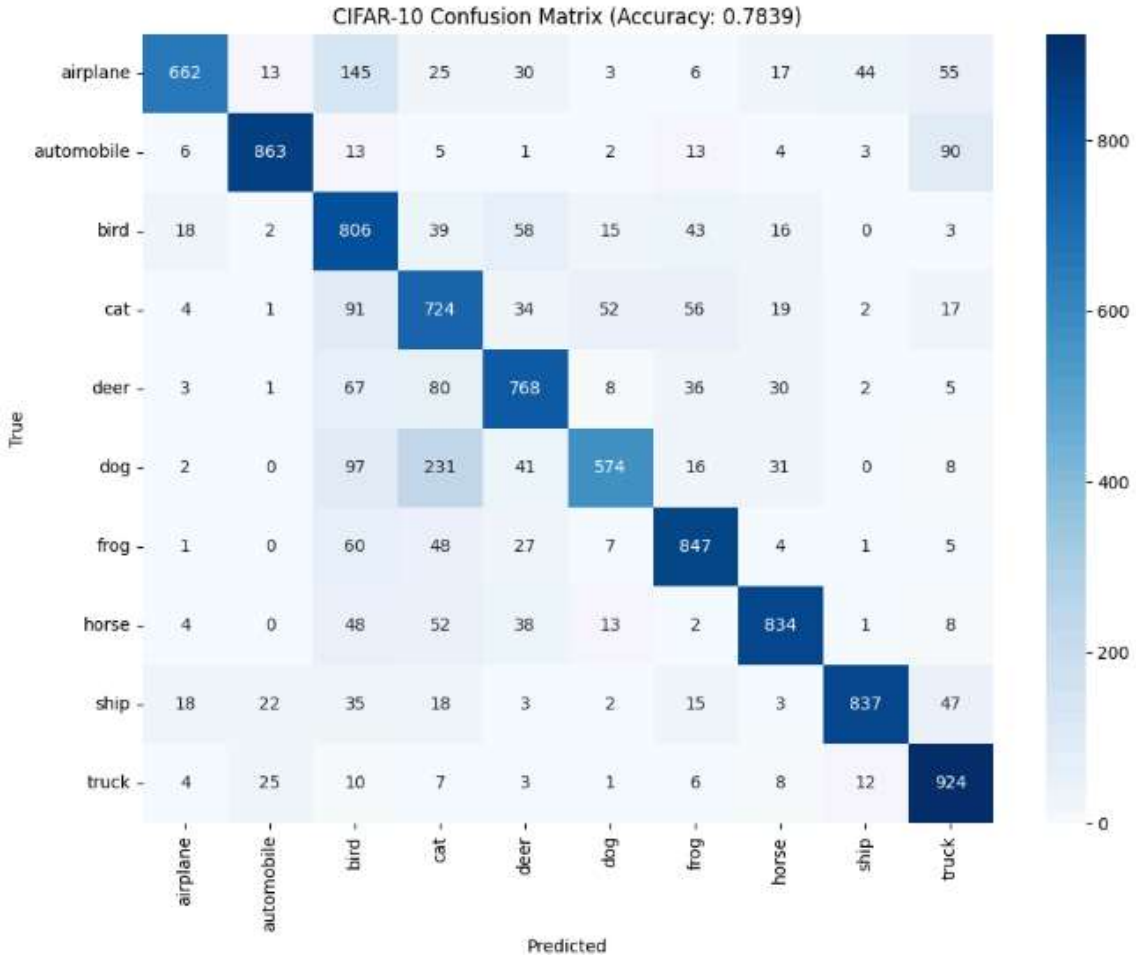
尝试使用LabelSmoothingCrossEntropy，该方法是对交叉熵的改进，通过对原始的交叉熵进行加权求和，可以提升模型泛化能力。实现代码如下，其中nll_loss是原先交叉熵损失的部分。

```
class LabelSmoothingCrossEntropy(nn.Module):
    def __init__(self, smoothing=0.1):
        super().__init__()
        self.smoothing = smoothing

    def forward(self, pred, target):
        log_probs = F.log_softmax(pred, dim=-1)
        nll_loss   = -log_probs.gather(dim=-1,
index=target.unsqueeze(1)).squeeze(1)
        smooth_loss= -log_probs.mean(dim=-1)
        return (1.0 - self.smoothing) * nll_loss + self.smoothing * smooth_loss
```

预测效果如下：效果比普通的交叉熵损失有所提升



考虑加入L2正则化：optimizer = torch.optim.Adam(model.parameters(), lr=3e-3,weight_decay=1e-4)。预测结果如下，效果不如加入正则化时。



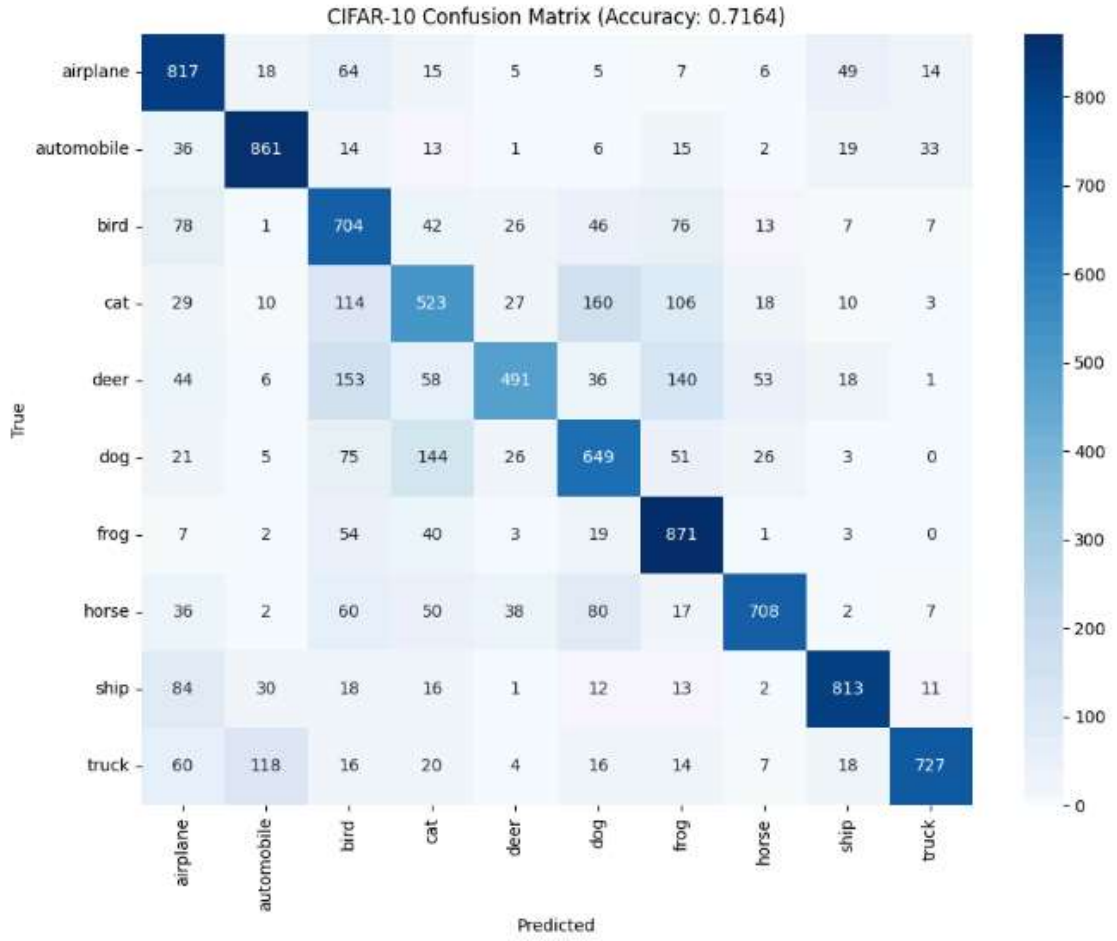
因此本部分得到的最优模型为不使用正则化，使用LabelSmoothingCrossEntropy作为损失函数效果最佳。

Try different optimizers using *torch.optim*

尝试使用moment SGD：

```
    optimizer = torch.optim.SGD(
    model.parameters(),
    lr=1e-3,
    momentum=0.9,
)
```

预测效果如下：经过参数调整始终无法达到Adam的效果，因此本任务中继续使用Adam优化器



三、VGG-A网络与Batch normalization

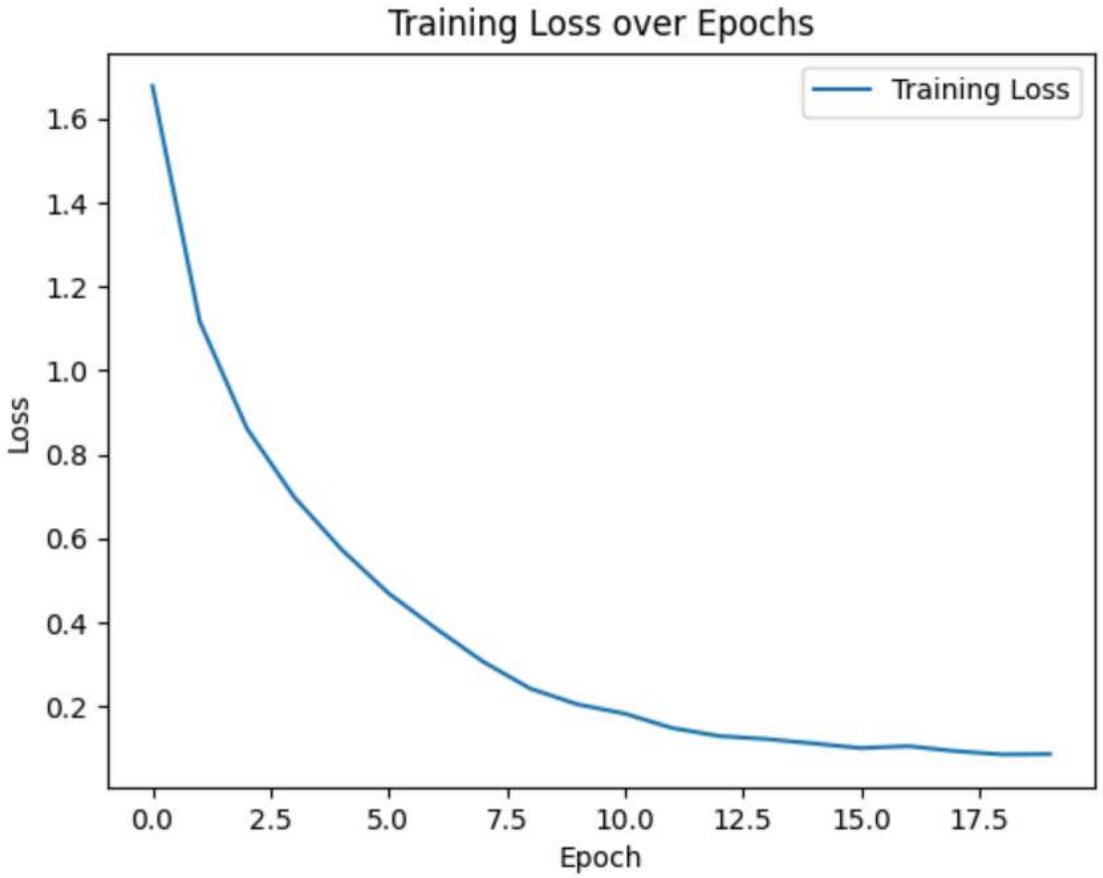
Without BN进行训练

首先我们需要修改vgg.py，补充权重初始化的方法。对卷积层采用He初始化，适合ReLU激活函数的特性。对全连接层采用Xavier初始化，让不同通道的信号流保持均衡。对BN层采用常数初始化，采用保守的初始化策略不作任何引导，完全依赖后续数据的学习。

```
def initialize_weights(module):
    if isinstance(module, nn.Conv2d):
        nn.init.kaiming_normal_(module.weight, mode='fan_out',
nonlinearity='relu')
        if module.bias is not None:
            nn.init.constant_(module.bias, 0)
    elif isinstance(module, nn.Linear):
        nn.init.xavier_normal_(module.weight)
        if module.bias is not None:
            nn.init.constant_(module.bias, 0)
    elif isinstance(module, nn.BatchNorm2d):
        nn.init.constant_(module.weight, 1)
        nn.init.constant_(module.bias, 0)
```

接着编写训练函数VGG_Loss_Landscape.py。主要是训练模板套用的简单代码，这里不过多说明。

训练结果如下：

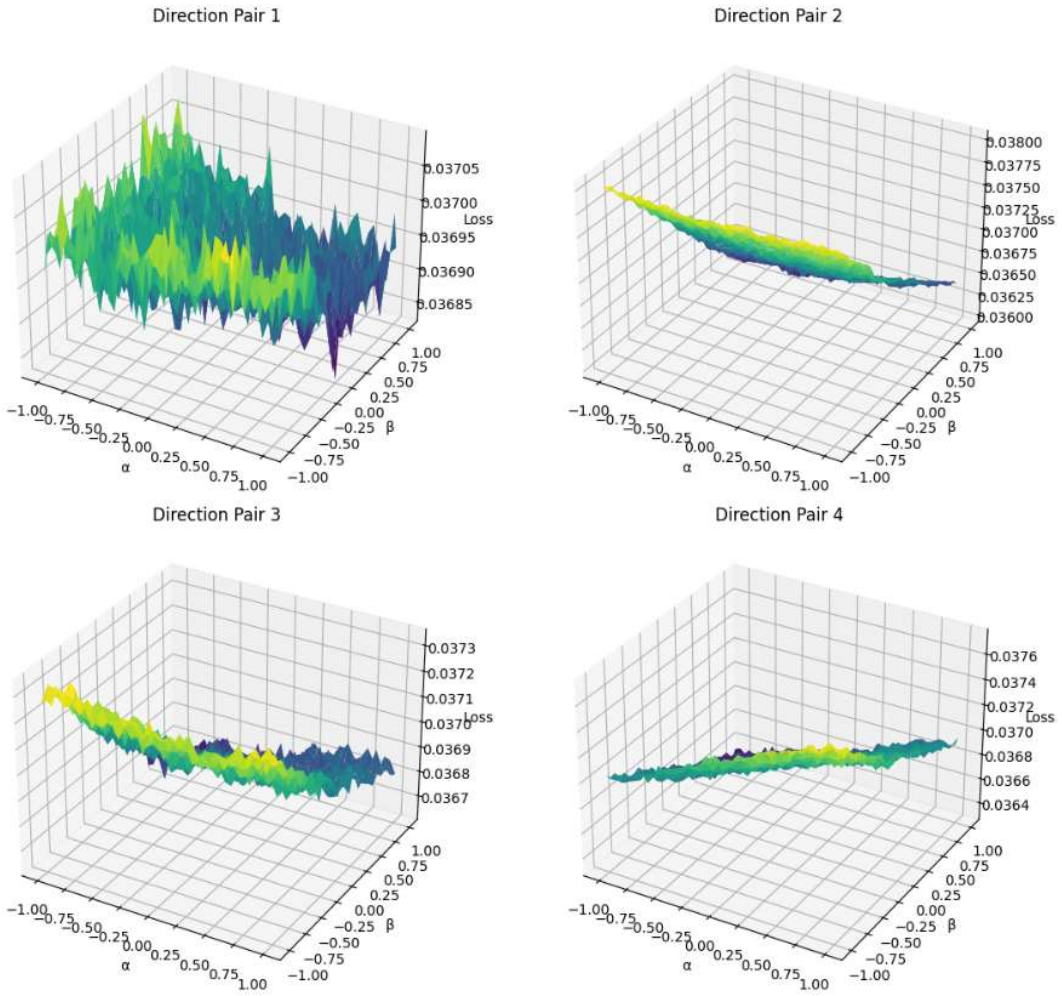


接着我们绘制三维的loss landscape。实现思路为首先将所有参数打包为一个向量，随机抽取两个方向并正交化，绘制当参数取到这个二维空间中某个点及其扰动时模型训练的损失，可以直观地感受训练后参数所处区域的平坦度与曲率。这里我们随机选取四组不同的正交方向以获得更鲁棒的结论。

```
    theta0 =
torch.nn.utils.parameters_to_vector(model.parameters()).detach().cpu()
    dim = theta0.numel()
    alphas = np.linspace(-1.0, 1.0, 41)
    betas  = np.linspace(-1.0, 1.0, 41)
    A, B   = np.meshgrid(alphas, betas)
    fig = plt.figure(figsize=(12,10))
```

```
    axes = [fig.add_subplot(2,2,i+1, projection='3d') for i in range(4)]
    model.eval()
    X0, y0 = next(iter(train_loader))
    X0, y0 = X0.to(device), y0.to(device)
    for k in tqdm(range(4), desc='Plotting Loss Landscapes'):

        # 随机选择四组正交参数方向 δ_k, η_k
        delta = torch.randn(dim)
        delta /= delta.norm()
        eta   = torch.randn(dim)
        eta  -= torch.dot(eta, delta) * delta
        eta  /= eta.norm()

        # 计算 loss_grid
        loss_grid = np.zeros_like(A, dtype=float)
        with torch.no_grad():
            for i in range(A.shape[0]):
                for j in range(A.shape[1]):
                    theta = theta0 + A[i,j]*delta + B[i,j]*eta
                    torch.nn.utils.vector_to_parameters(theta.to(device),
model.parameters())
                    loss_grid[i,j] = criterion(model(X0), y0).item()

        # 绘制子图
        ax = axes[k]
        ax.plot_surface(A, B, loss_grid,
                        rstride=1, cstride=1,
                        cmap='viridis', edgecolor='none')
        ax.set_title(f'Direction Pair {k+1}')
        ax.set_xlabel('α')
        ax.set_ylabel('β')
        ax.set_zlabel('Loss')
```

绘制结果如下：我们看到了几种不同的趋势。可以看到损失表面非常崎岖不平，说明优化器在这样的地形中容易受到小扰动的影响，参数未完全收敛，此时进行梯度下降容易陷入局部最小值。



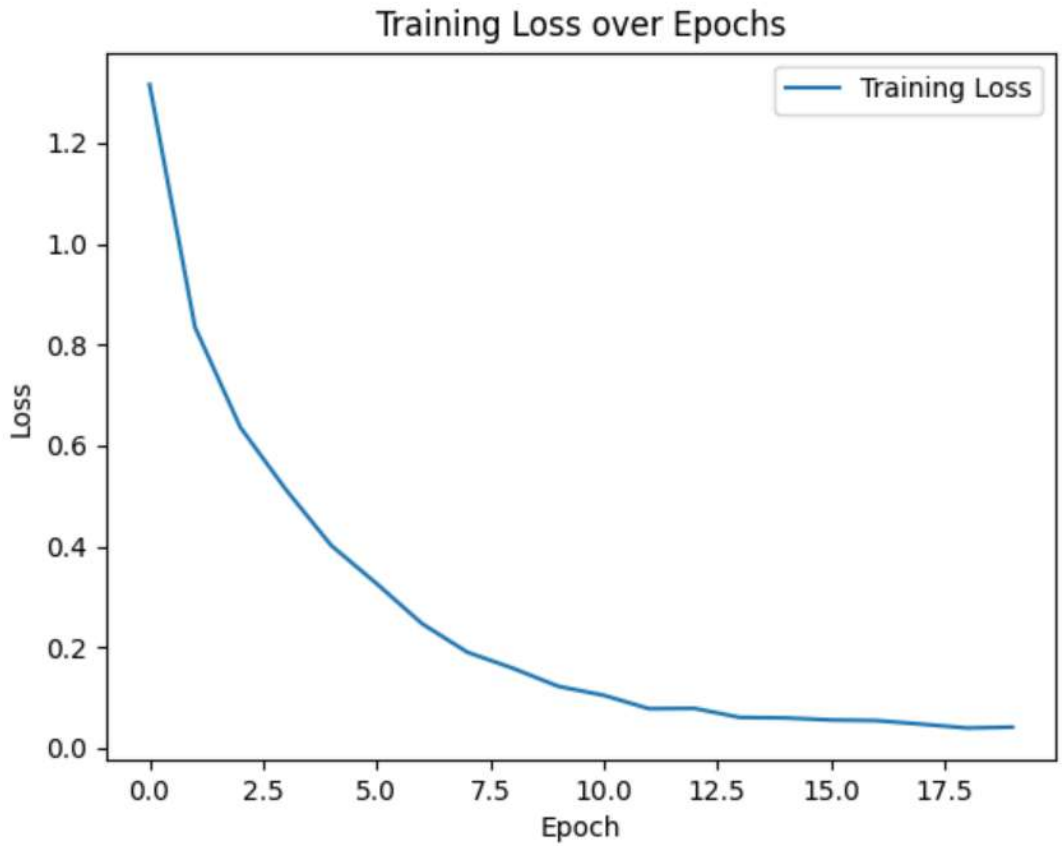
With BN进行训练

在VGG_A_BN类的编写中，我们需要仿照VGG_A网络的结构，并在每个卷积层中加入batch normalization层。

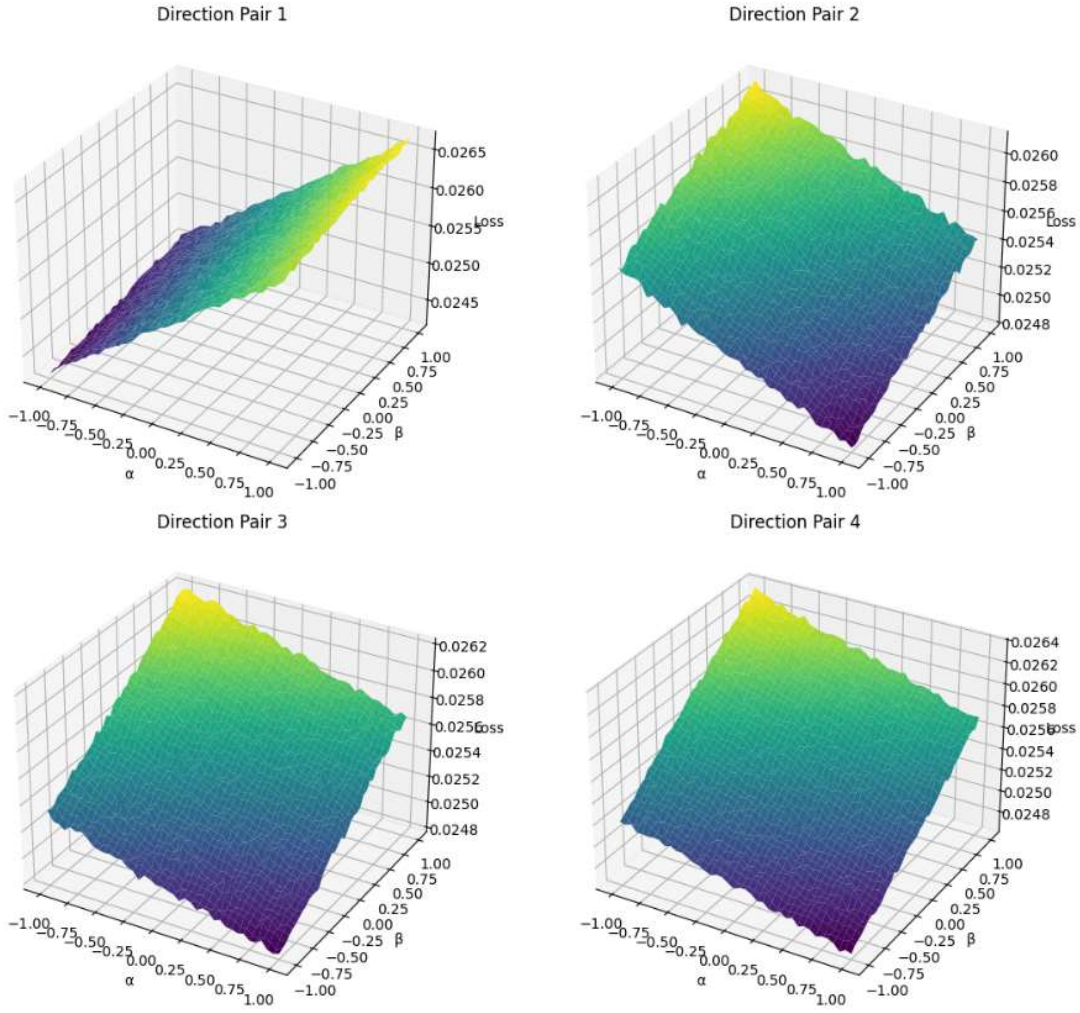
```
class VGG_A_BN(nn.Module):
    def __init__(self, inp_ch=3, num_classes=10, init_weights=True):
        super().__init__()
        self.features = nn.Sequential(
            # stage 1
            nn.Conv2d(inp_ch, 64, 3, padding=1), nn.BatchNorm2d(64),
nn.ReLU(True), nn.MaxPool2d(2),
            # stage 2
            nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.ReLU(True),
nn.MaxPool2d(2),
            # stage 3
            nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256),
nn.ReLU(True),
            nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256),
nn.ReLU(True), nn.MaxPool2d(2),
            # stage 4
            nn.Conv2d(256, 512, 3, padding=1), nn.BatchNorm2d(512),
nn.ReLU(True),
            nn.Conv2d(512, 512, 3, padding=1), nn.BatchNorm2d(512),
nn.ReLU(True), nn.MaxPool2d(2),
            # stage 5
```

```
            nn.Conv2d(512, 512, 3, padding=1), nn.BatchNorm2d(512),
nn.ReLU(True),
            nn.Conv2d(512, 512, 3, padding=1), nn.BatchNorm2d(512),
nn.ReLU(True), nn.MaxPool2d(2)
        )
        self.classifier = nn.Sequential(
            nn.Linear(512, 512), nn.ReLU(True),
            nn.Linear(512, 512), nn.ReLU(True),
            nn.Linear(512, num_classes)
        )
        if init_weights:
            self._init_weights()

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        return self.classifier(x)

    def _init_weights(self):
        self.apply(initialize_weights)
```

训练结果如下：可以看到在BN算法下，训练的损失更小。每层输入分布在训练过程中不断变化，网络需不停地重新适应新的输入分布。BatchNorm 通过对每层输入做归一化，使得后续层接收到的数据分布更稳定，模型更容易学习，收敛更快，训练损失更低。我们将通过绘制loss landscape来解释这一现象。



绘制结果如下：与没有进行batch normalization的VGG_A网络的loss landscape相比，进行BN后损失平面更加平滑，因此在进行梯度下降更新时更不容易跨过最优点或陷入陡峭区域的局部最小值，从而更稳定地降低损失。而且BN也避免了在多层卷积运算过程中可能导致的梯度爆炸或消失现象，整体参数的迭代更加稳定，因此训练效果会比没有进行BN的网络更好。



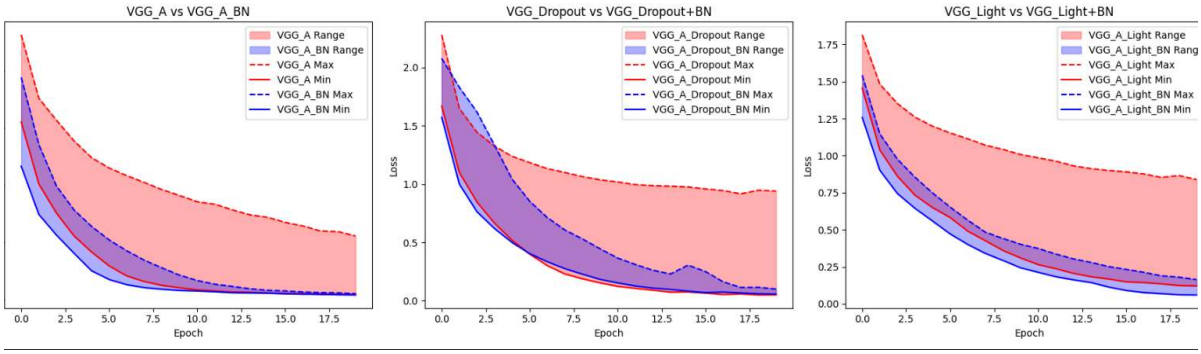
四、Batch normalization算法深度探索

在本块探索中，我们通过训练过程中调整不同学习率对 loss 稳定性的影响，间接评估模型的 Lipschitz 性，并比较使用 BN 和不使用 BN 的模型在这方面的表现。我们分别对VGG_A, VGG_A_light, VGG_A_dropout三个模型进行实验。选择学习率如下：[1e-4, 1e-3, 3e-3, 5e-4]。需要修改main函数如下：

```
def train_curves(ModelClass, label):
    all_losses = []
    for lr in learning_rates:
        print(f"\nTraining {label} with learning rate {lr}")
        model = ModelClass().to(device)
        optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=1e-
4)
        criterion = nn.CrossEntropyLoss()
        best_model_path = os.path.join(models_path,
f'{label}_lr_{lr:.0e}_best.pth')
        losses = train(model, optimizer, criterion,
                       train_loader, val_loader,
                       epochs_n=20,
                       best_model_path=best_model_path)
        loss_save_path = os.path.join(models_path,
f'{label}_loss_lr_{lr:.0e}.txt')
        np.savetxt(loss_save_path, losses, fmt='%.6f')
        all_losses.append(losses)
```

```
    losses_array = np.array(all_losses)
    max_curve = losses_array.max(axis=0)
    min_curve = losses_array.min(axis=0)
    return max_curve, min_curve

if __name__ == '__main__':
    set_random_seeds(2020)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    learning_rates = [1e-4, 1e-3, 3e-3, 5e-4]
    plt.figure(figsize=(18, 5))

    def plot_loss_subplot(idx, model1, model2, name1, name2, title):
        max1, min1 = train_curves(model1, name1)
        max2, min2 = train_curves(model2, name2)
        epochs = np.arange(len(max1))
        ax = plt.subplot(1, 3, idx)
        ax.fill_between(epochs, min1, max1, color='red', alpha=0.3,
label=f'{name1} Range')
        ax.fill_between(epochs, min2, max2, color='blue', alpha=0.3,
label=f'{name2} Range')
        ax.plot(max1, color='red', linestyle='--', label=f'{name1} Max')
        ax.plot(min1, color='red', label=f'{name1} Min')
        ax.plot(max2, color='blue', linestyle='--', label=f'{name2} Max')
        ax.plot(min2, color='blue', label=f'{name2} Min')
        ax.set_title(title)
        ax.set_xlabel('Epoch')
        ax.set_ylabel('Loss')
        ax.legend()

    plot_loss_subplot(1, VGG_A, VGG_A_BN, 'VGG_A', 'VGG_A_BN', 'VGG_A vs
VGG_A_BN')
    plot_loss_subplot(2, VGG_A_Dropout, VGG_A_Dropout_BN, 'VGG_A_Dropout',
'VGG_A_Dropout_BN', 'VGG_Dropout vs VGG_Dropout+BN')
    plot_loss_subplot(3, VGG_A_Light, VGG_A_Light_BN, 'VGG_A_Light',
'VGG_A_Light_BN', 'VGG_Light vs VGG_Light+BN')

    plt.tight_layout()
    save_path = os.path.join(figures_path, 'loss_range_comparison.png')
    plt.savefig(save_path)
    plt.close()
    print(f"Saved comparison plot to: {save_path}")
```

训练结果如下：可以看到在三种不同模型下，batch normalization算法显著降低了max curve与min curve之间的差距，这表明在BN下模型的损失波动小，损失的稳定性强即Lipschitz 性更强，因此BN通过提高损失函数在训练过程中的稳定性来使得训练效果更好。


VGG_A vs VGG_A_BN
VGG_A Range
VGG_A_BN Range
VGG_A Max
VGG_A Min
VGG_A_BN Max
VGG_A_BN Min
Epoch
VGG_Dropout vs VGG_Dropout+BN
VGG_A_Dropout Range
VGG_A_Dropout_BN Range
VGG_A_Dropout Max
VGG_A_Dropout Min
VGG_A_Dropout_BN Max
VGG_A_Dropout_BN Min
Loss
Epoch
VGG_Light vs VGG_Light+BN
VGG_A_Light Range
VGG_A_Light_BN Rang
VGG_A_Light Max
VGG_A_Light Min
VGG_A_Light_BN Max
VGG_A_Light_BN Min
Loss
Epoch