import os  
import time  
from copy import deepcopy  
import torch  
import torch.nn as nn  
from torch.utils.data import DataLoader, Subset  
from torchvision.datasets import ImageFolder  
from torchvision import transforms  
from torchvision.models import efficientnet\_v2\_s  
from torch.cuda.amp import autocast, GradScaler  
import numpy as np  
from sklearn.model\_selection import train\_test\_split  
from torch.utils.tensorboard import SummaryWriter  
  
  
writer = SummaryWriter(log\_dir='runs/dog\_breed\_experiment\_10')  
device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  
os.environ['PYTORCH\_CUDA\_ALLOC\_CONF'] = 'max\_split\_size\_mb:128' # 防止内存碎片  
# ==============================  
# 数据增强及预处理  
# ==============================  
transform\_train = transforms.Compose([  
 transforms.RandomResizedCrop(224, scale=(0.5, 1.0)), # 扩大裁剪范围  
 transforms.RandomHorizontalFlip(p=0.6),  
 transforms.RandomVerticalFlip(p=0.3),  
 transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.4),  
 transforms.RandomAffine(degrees=20, translate=(0.15, 0.15)),  
 transforms.RandomApply([transforms.GaussianBlur(5)], p=0.4),  
 transforms.RandomSolarize(threshold=128, p=0.2), # 新增光斑效果  
 transforms.ToTensor(),  
 transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),  
 transforms.RandomErasing(p=0.4, scale=(0.03, 0.15), value='random') # 随机颜色擦除  
])  
  
transform\_test = transforms.Compose([  
 transforms.Resize(256),  
 transforms.CenterCrop(224),  
 transforms.ToTensor(),  
 transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])  
])  
  
  
class ModelEMA:  
 def \_\_init\_\_(self, model, decay=0.999, total\_epochs=50):  
 self.ema = deepcopy(model).eval()  
 self.initial\_decay = decay # 初始衰减率  
 self.min\_decay = 0.995 # 最低衰减率  
 self.total\_epochs = total\_epochs  
 for param in self.ema.parameters():  
 param.requires\_grad\_(False)  
  
 # 初始参数同步  
 with torch.no\_grad():  
 for ema\_p, model\_p in zip(self.ema.parameters(), model.parameters()):  
 ema\_p.copy\_(model\_p)  
  
 def update(self, model, current\_epoch):  
 # 线性衰减策略：从initial\_decay降到min\_decay  
 decay = self.initial\_decay - (self.initial\_decay - self.min\_decay) \* (current\_epoch / self.total\_epochs)  
 decay = max(decay, self.min\_decay) # 确保不低于最小值  
 with torch.no\_grad():  
 for ema\_param, model\_param in zip(self.ema.parameters(), model.parameters()):  
 # 处理半精度参数  
 model\_param\_fp32 = model\_param.detach().float()  
 ema\_param\_fp32 = ema\_param.float()  
 ema\_param\_fp32.mul\_(decay).add\_(model\_param\_fp32, alpha=1 - decay)  
 ema\_param.copy\_(ema\_param\_fp32)  
  
  
# ==============================  
# 加载 Stanford Dogs 数据集（Kaggle版本）  
# ==============================  
data\_dir = './images/images' # 请将数据集解压后的文件夹路径填写在此处  
  
# 分别创建两个 ImageFolder 实例，分别设置训练和验证时的变换  
full\_dataset\_train = ImageFolder(root=data\_dir, transform=transform\_train)  
full\_dataset\_val = ImageFolder(root=data\_dir, transform=transform\_test)  
  
# 获取所有样本标签用于分层划分  
all\_targets = full\_dataset\_train.targets  
  
# 使用 train\_test\_split 进行分层划分，80%作为训练集，20%作为验证集  
train\_idx, val\_idx = train\_test\_split(  
 list(range(len(all\_targets))),  
 test\_size=0.2,  
 stratify=all\_targets  
)  
  
train\_set = Subset(full\_dataset\_train, train\_idx)  
val\_set = Subset(full\_dataset\_val, val\_idx)  
  
print(f"训练集: {len(train\_set)} 个样本, 验证集: {len(val\_set)} 个样本")  
  
# ==============================  
# 定义网络模型（以 efficientnet\_v2\_s 为例）  
# ==============================  
def get\_net(devices):  
 model = efficientnet\_v2\_s(weights='DEFAULT')  
  
 # 冻结stem和前4个block  
 for param in model.features[:4].parameters():  
 param.requires\_grad = False  
  
 # 修改分类头（保持特征维度）  
 model.classifier = torch.nn.Sequential(  
 torch.nn.Dropout(p=0.5), # 增加Dropout比例  
 torch.nn.Linear(1280, 1024),  
 torch.nn.BatchNorm1d(1024),  
 torch.nn.SiLU(inplace=True),  
 torch.nn.Dropout(p=0.3),  
 torch.nn.Linear(1024, 512),  
 torch.nn.LayerNorm(512), # 改用LayerNorm  
 torch.nn.SiLU(inplace=True),  
 torch.nn.Linear(512, 120)  
 )  
 return model.to(devices)  
  
pretrained\_net = get\_net(device)  
  
# ==============================  
# 定义优化器和学习率调度器（使用 ReduceLROnPlateau 策略）  
# ==============================  
# 修改优化器参数  
optimizer = torch.optim.AdamW(  
 [  
 {'params': pretrained\_net.features[4:].parameters(), 'lr': 2e-5, 'weight\_decay': 0.001},  
 {'params': pretrained\_net.classifier[:-4].parameters(), 'lr': 5e-4, 'weight\_decay': 0.003},  
 {'params': pretrained\_net.classifier[-4:].parameters(), 'lr': 1e-3, 'weight\_decay': 0.005}  
 ],  
 betas=(0.95, 0.999),  
 eps=1e-8 # 增加数值稳定性  
)  
  
# 修改调度器参数  
plateau\_scheduler = torch.optim.lr\_scheduler.ReduceLROnPlateau(  
 optimizer,  
 mode='max',  
 factor=0.3, # 更温和的衰减幅度  
 patience=3, # 缩短观察窗口  
 threshold=0.001 # 更敏感的阈值  
)  
  
cosine\_scheduler = torch.optim.lr\_scheduler.CosineAnnealingWarmRestarts(  
 optimizer,  
 T\_0=12, # 延长余弦周期到12个epoch  
 T\_mult=2, # 周期倍增  
 eta\_min=1e-6  
)  
  
# ==============================  
# 定义评价函数  
# ==============================  
def evaluate\_accuracy(data\_iter, net, loss\_fn, device=None, use\_ema=False, ema\_model=None):  
 """  
 计算验证集的准确率和损失  
 :param use\_ema: 是否使用EMA模型  
 :param ema\_model: EMA模型实例  
 """  
 if device is None and isinstance(net, nn.Module):  
 device = next(net.parameters()).device  
  
 # 选择模型  
 if use\_ema:  
 if ema\_model is None:  
 raise ValueError("ema\_model must be provided when use\_ema is True")  
 model = ema\_model.ema  
 else:  
 model = net  
  
 model.eval()  
  
 acc\_sum, loss\_sum, n = 0.0, 0.0, 0  
 with torch.no\_grad():  
 for X, y in data\_iter:  
 X, y = X.to(device), y.to(device)  
 with autocast():  
 output = model(X)  
 loss = loss\_fn(output, y)  
 acc\_sum += (output.argmax(dim=1) == y).float().sum().item()  
 loss\_sum += loss.item() \* y.size(0)  
 n += y.shape[0]  
 return acc\_sum / n, loss\_sum / n  
  
  
  
  
def train(train\_iter, test\_iter, net, loss, optimizer, device, num\_epochs,  
 best\_acc=0.0, accum\_steps=4):  
 best\_metric = 0.0  
 patience = 12  
 no\_improve = 0  
 net = net.to(device)  
 scaler = GradScaler()  
 ema = ModelEMA(net)  
 # 记录当前使用的调度器  
 current\_scheduler = "cosine"  
 print(f'training on {device} with accum\_steps={accum\_steps}')  
  
 for epoch in range(num\_epochs):  
 net.train()  
 train\_l\_sum, train\_acc\_sum, n = 0.0, 0.0, 0  
 epoch\_start = time.time()  
 total\_batches = len(train\_iter)  
  
 # 初始化进度跟踪变量  
 batch\_times = []  
 progress\_bar\_length = 30  
  
 for batch\_idx, (X, y) in enumerate(train\_iter):  
 batch\_start = time.time()  
 X, y = X.to(device), y.to(device)  
  
 # 混合精度前向  
 with autocast():  
 y\_hat = net(X)  
 l = loss(y\_hat, y) / accum\_steps  
  
 unscaled\_loss = l.detach().clone() \* accum\_steps  
 scaler.scale(l).backward()  
  
 # 梯度累积条件判断  
 if (batch\_idx + 1) % accum\_steps == 0 or (batch\_idx + 1) == total\_batches:  
 scaler.unscale\_(optimizer)  
 torch.nn.utils.clip\_grad\_norm\_(net.parameters(), max\_norm=2.0) # 添加梯度裁剪  
 scaler.step(optimizer)  
 scaler.update()  
 optimizer.zero\_grad()  
 ema.update(net,epoch)  
  
 # 统计指标  
 batch\_acc = (y\_hat.argmax(dim=1) == y).sum().item() / y.size(0)  
 train\_l\_sum += unscaled\_loss.item() \* y.size(0)  
 train\_acc\_sum += (y\_hat.argmax(dim=1) == y).sum().item()  
 n += y.size(0)  
  
 # 计算进度和时间预估  
 batch\_time = time.time() - batch\_start  
 batch\_times.append(batch\_time)  
 avg\_batch\_time = np.mean(batch\_times[-10:])  
  
 # 进度计算  
 completed = batch\_idx + 1  
 progress = completed / total\_batches  
 elapsed = time.time() - epoch\_start  
 remaining = avg\_batch\_time \* (total\_batches - completed)  
  
 # 构建进度条  
 filled\_length = int(progress\_bar\_length \* progress)  
 progress\_bar = '█' \* filled\_length + '-' \* (progress\_bar\_length - filled\_length)  
  
 # 实时显示信息  
 info = (f"Epoch {epoch + 1}/{num\_epochs} |{progress\_bar}| "  
 f"{completed}/{total\_batches} batches "  
 f"[{elapsed:.0f}s<{remaining:.0f}s, {1 / avg\_batch\_time:.1f}batches/s] "  
 f"Loss: {unscaled\_loss.item():.4f} Acc: {batch\_acc:.4f}")  
 print("\r" + info, end="")  
  
 # 完成一个epoch后换行  
 epoch\_time = time.time() - epoch\_start  
 print(f"\rEpoch {epoch + 1} completed in {epoch\_time:.1f}s".ljust(120))  
  
 # 验证阶段  
 test\_acc, test\_loss = evaluate\_accuracy(test\_iter, net, loss, device, use\_ema=False, ema\_model=ema)  
 print(f" Train Loss: {train\_l\_sum / n:.4f} Train Acc: {train\_acc\_sum / n:.4f}")  
 print(f" Val Loss (Original): {test\_loss:.4f} Val Acc (Original): {test\_acc:.4f}")  
 ema\_test\_acc, ema\_test\_loss = evaluate\_accuracy(test\_iter, net, loss, device, use\_ema=True, ema\_model=ema)  
 print(f" Val Loss (EMA): {ema\_test\_loss:.4f} Val Acc (EMA): {ema\_test\_acc:.4f}")  
  
 # 动态切换调度策略  
 if epoch < 15: # 前5个epoch使用cosine  
 cosine\_scheduler.step(epoch + 0.5) # 需要传入当前epoch  
 current\_scheduler = "cosine"  
 else: # 第5个epoch之后使用plateau  
 if epoch == 15:  
 # 将plateau的基准学习率设为当前值  
 for i, group in enumerate(optimizer.param\_groups):  
 group['initial\_lr'] = group['lr']  
 plateau\_scheduler.base\_lrs = [group['lr'] for group in optimizer.param\_groups]  
 plateau\_scheduler.step(test\_acc)  
 current\_scheduler = "plateau"  
 # 打印学习率信息（调试用）  
 current\_lrs = [f"{g['lr']:.2e}({current\_scheduler})" for g in optimizer.param\_groups]  
 print(f" Current lrs: {current\_lrs}\n")  
 # 记录到TensorBoard  
 writer.add\_scalar('Loss/train', train\_l\_sum / n, epoch)  
 writer.add\_scalar('Accuracy/train', train\_acc\_sum / n, epoch)  
 writer.add\_scalar('Loss/val', test\_loss, epoch) # 新增验证损失  
 writer.add\_scalar('Accuracy/val', test\_acc, epoch)  
 writer.add\_scalar('Metrics/LR\_Group0', optimizer.param\_groups[0]['lr'], epoch)  
 writer.add\_scalar('Metrics/LR\_Group1', optimizer.param\_groups[1]['lr'], epoch)  
 writer.add\_scalar('Metrics/TrainVal\_Gap', train\_acc\_sum / n - test\_acc, epoch) # 过拟合指标  
 writer.add\_scalar('Loss/val\_ema', ema\_test\_loss, epoch)  
 writer.add\_scalar('Accuracy/val\_ema', ema\_test\_acc, epoch)  
  
 # 模型保存逻辑  
 if test\_acc > best\_acc:  
 best\_acc = test\_acc  
 print(f" New best accuracy! Saving the model...")  
 torch.save(net.state\_dict(), 'best\_model.pth')  
 # 保存最佳EMA模型  
 if ema\_test\_acc > best\_acc:  
 best\_acc = ema\_test\_acc  
 print(f" New best ema accuracy! Saving the ema model...")  
 torch.save(ema.ema.state\_dict(), 'best\_model\_ema.pth')  
  
 # 使用加权指标（准确率为主，loss为辅）  
 current\_metric = test\_acc \* 0.8 + (1 - test\_loss) \* 0.2  
 if current\_metric > best\_metric:  
 best\_metric = current\_metric  
 no\_improve = 0  
 else:  
 no\_improve += 1  
 if no\_improve >= patience:  
 print(f"早停触发：连续{patience}个epoch综合指标无提升")  
 break  
  
 # 打印学习率信息  
 current\_lrs = [f"{g['lr']:.2e}" for g in optimizer.param\_groups]  
 print(f" Current lrs: {current\_lrs}\n")  
  
 # 关闭TensorBoard Writer  
 writer.close()  
 return best\_acc  
  
  
def train\_fine\_tuning(net, optimizer, batch\_size=64, num\_epochs=50):  
 train\_iter = DataLoader(  
 train\_set,  
 batch\_size,  
 shuffle=True,  
 pin\_memory=True,  
 )  
 val\_iter = DataLoader(  
 val\_set,  
 batch\_size,  
 shuffle=False,  
 pin\_memory=True,  
 )  
 loss\_fn = nn.CrossEntropyLoss(label\_smoothing=0.15)  
 best\_acc = train(train\_iter, val\_iter, net, loss\_fn, optimizer, device, num\_epochs)  
 return best\_acc  
  
# ==============================  
# 启动微调训练  
# ==============================  
if \_\_name\_\_ == '\_\_main\_\_':  
 train\_fine\_tuning(pretrained\_net, optimizer)