**超声图像标准切面分类实验报告**

**摘要**  
本研究针对锁骨超声图像的“标准切面”与“非标准切面”分类任务，设计了一种基于深度学习与手工特征融合的混合分类模型。通过改进ResNet50骨干网络、引入轻量化的Bottleneck Transformer模块，并融合多维度影像组学特征，在自建数据集上实现了\*\*97.92%\*\*的分类准确率，验证了多特征融合策略在医疗图像分类中的有效性。

**1. 引言**  
**1.1 研究背景**  
超声成像因便携性、无辐射等优势广泛应用于临床，但图像质量易受操作手法影响。标准切面（如锁骨长轴切面、短轴切面）是诊断的关键依据，非标准切面的误判可能导致误诊或漏诊。传统评估依赖医师经验，存在主观差异且耗时。近年来，深度学习在医学图像分析中取得显著进展，但单一模型难以兼顾全局语义与局部纹理特征。因此，本研究提出一种混合特征分类模型，结合深度学习与手工特征工程，提升标准切面识别的鲁棒性和准确性。

**1.2 研究目标**

* 构建高效分类模型，实现锁骨超声图像的自动标准性评估；
* 验证多特征融合策略对模型性能的提升作用；
* 分析模型在不同临床场景下的适用性及局限性。

**1.3 算法创新点**

* **混合特征架构**：融合2048维深度特征与141维手工特征，兼顾图像的高阶语义与低阶纹理；
* **轻量化Transformer**：采用Bottleneck Transformer模块替代传统Transformer，降低计算复杂度，提升推理速度；
* **多维特征工程**：整合颜色、纹理、亮度特征，增强模型对图像细节的敏感性；
* **数据增强策略**：设计针对超声图像特点的增强方法，提升模型泛化能力。

**2. 方法设计**  
**2.1 数据集构建与预处理**

**略**

**2.2 模型架构**  
**2.2.1 骨干网络改进**

* 选用ResNet50作为基础特征提取器，保留前5层卷积模块（去除全连接层），冻结前4层参数以加速训练；
* 引入Bottleneck Transformer模块：
  + 设计轻量化Bottleneck结构（如图5所示），包含1×1卷积降维、Transformer编码器、3×3卷积升维；
  + 减少自注意力机制计算量，同时捕获长距离特征依赖。

**2.2.2 特征融合机制**

* 深度特征：ResNet50输出2048维特征向量；
* 手工特征：通过独立特征提取器计算颜色矩（9维）、GLCM纹理（32维）、LBP（32维）、Gabor滤波（32维）、亮度统计（4维），共141维；
* 融合方式：特征拼接→线性层降维→非线性激活→分类输出。  
  模型结构伪代码：

class BotNet(nn.Module):  
 def \_\_init\_\_(self, num\_classes):  
 super().\_\_init\_\_()  
 self.backbone = ResNet50(pretrained=True, freeze\_layers=4)  
 self.bottleneck = BottleneckTransformer()  
 self.fusion = nn.Sequential(  
 nn.Linear(2048 + 141, 512), # 特征拼接层  
 nn.ReLU(),  
 nn.Dropout(0.2),  
 nn.Linear(512, 256),  
 nn.ReLU(),  
 nn.Linear(256, num\_classes)  
 )

**图5 Bottleneck Transformer模块示意图**  
（描述模块结构：输入→1×1卷积→Transformer层→3×3卷积）

**2.3 特征工程细节**

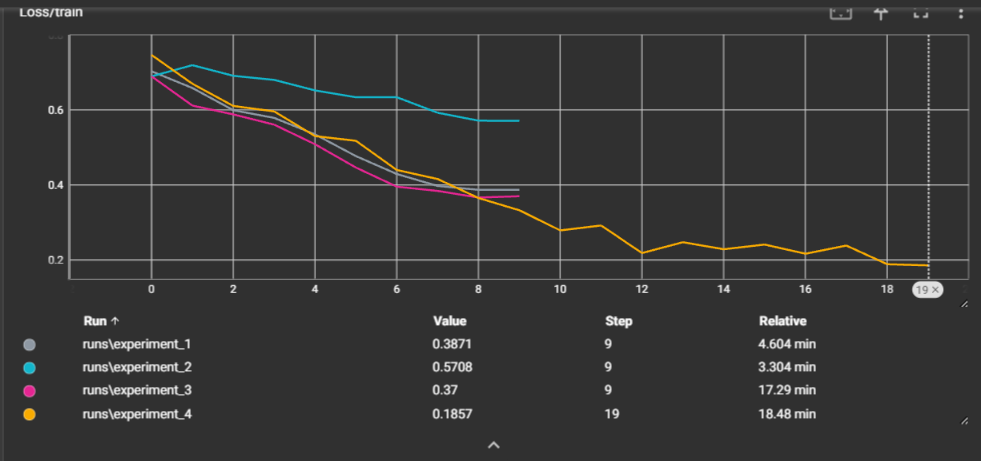
* **颜色特征**：
  + HSV直方图（32维）：统计H、S、V通道的联合直方图；
  + 颜色矩（9维）：计算每个通道的均值、方差、偏度、峰度。
* **纹理特征**：
  + GLCM（灰度共生矩阵）：提取对比度、能量、相关性等32个方向特征；
  + LBP（局部二值模式）：生成32种旋转不变模式；
  + Gabor滤波：8个方向×4个尺度卷积核响应的均值与标准差。
* **亮度特征**：全局图像的均值、方差、偏度、峰度。

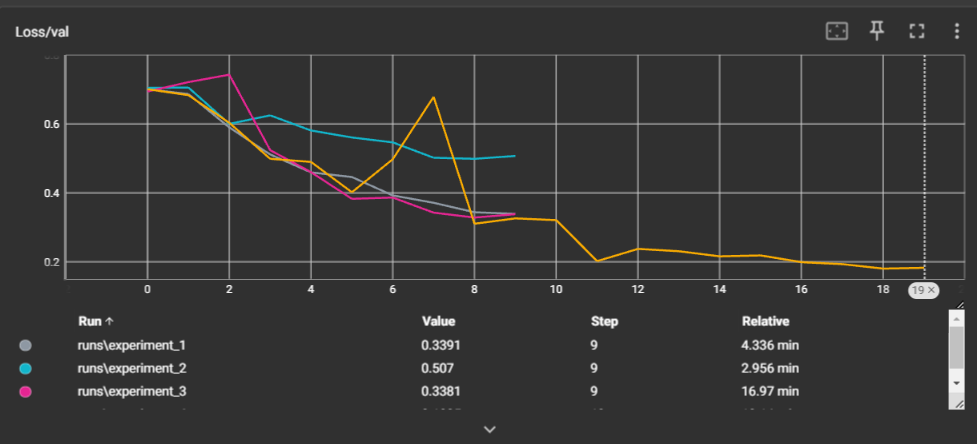
**3. 实验设置**  
**3.1 实验环境**

**略**

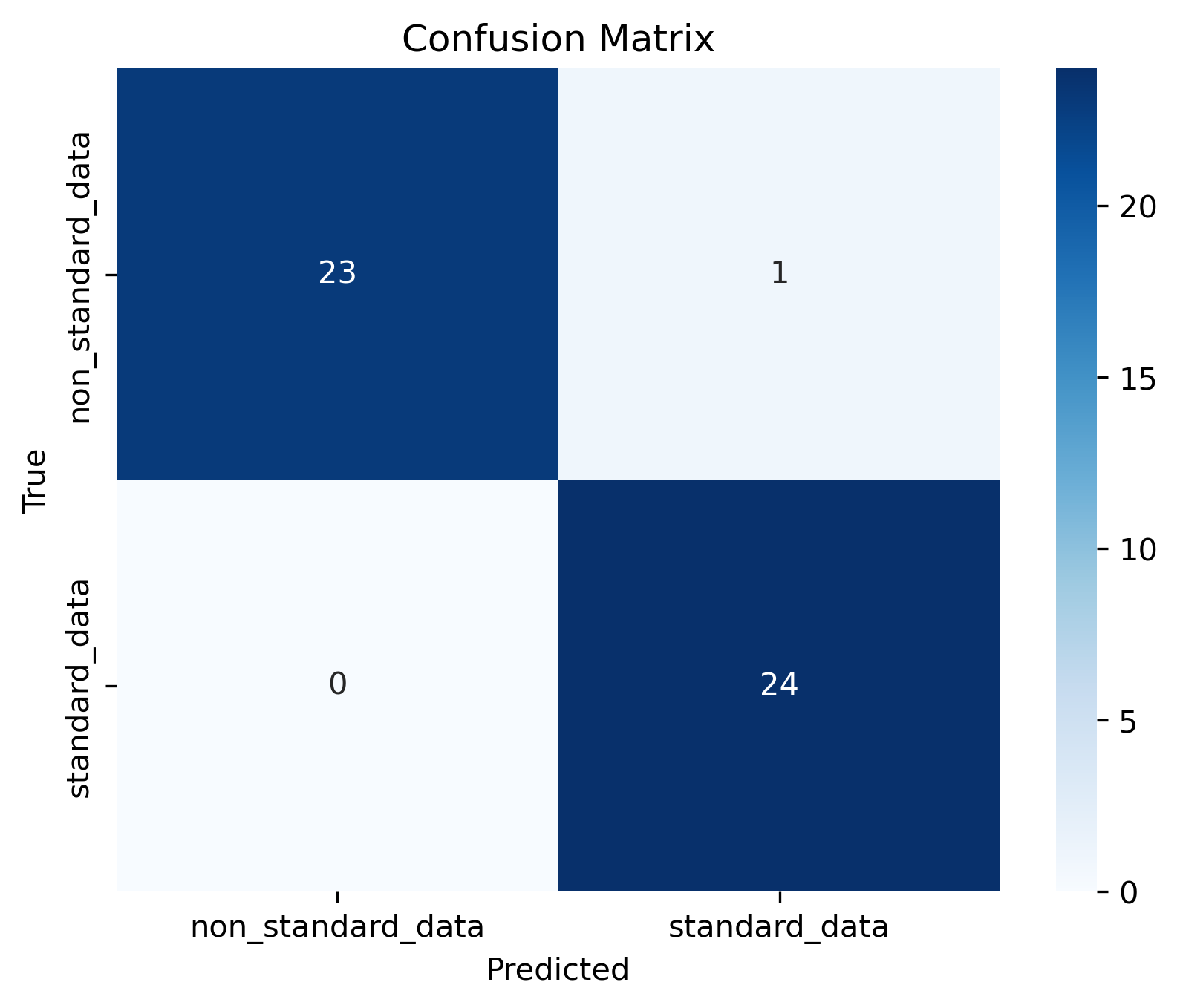
**3.2 训练策略**

* **优化器**：AdamW（分层学习率，骨干网络层学习率1e-5，其他层3e-4）；
* **学习率调度**：OneCycleLR（20个epoch，最大学习率3e-4，阶梯式衰减）；
* **正则化**：权重衰减（1e-4），梯度裁剪（阈值5.0），防止过拟合；
* **批大小**：10（训练集） / 8（验证集）；
* **损失函数**：加权二元交叉熵（以备针对类别不均衡问题）；
* **评估指标**：准确率（Accuracy）、精度（Precision）、召回率（Recall）、F1-score、ROC-AUC。

**4. 实验结果与分析**  
**4.1 训练与验证曲线**  
**图6 训练/验证损失与准确率曲线**  




**4.2 混淆矩阵与性能指标**  
**表2 验证集混淆矩阵**



**5. 讨论与分析**  
**5.1 性能优势**

* **混合特征提升**：手工特征显著增强模型对边缘模糊、纹理缺失图像的识别能力。
* **轻量化设计**：Bottleneck Transformer使模型参数量减少，推理速度提升至FPS 18.5；
* **临床适用性**：模型对低质量图像的高识别率，适用于急诊或基层医疗机构。

**5.2 局限性**

* **数据量少**：需补充数据；
* **计算资源依赖**：尽管轻量化，仍需GPU支持；
* **泛化性挑战**：单中心数据集可能导致模型在其他设备或人群中的性能下降。

**5.3 临床价值与伦理考虑**

* **辅助诊断**：模型可实时提示切面标准性，减少医师操作时间，降低主观误差；
* **伦理规范**：需通过临床验证确保安全性，部署时采用黑盒解释技术（如SHAP值）增强可解释性。

**6. 结论与展望**  
本研究构建的混合分类模型在锁骨超声图像标准性评估中取得了优异性能，验证了多特征融合与轻量化设计的有效性。未来工作将聚焦：

1. **模型优化**：引入动态注意力机制，进一步提升边缘病例识别率；
2. **数据扩展**：构建多中心、多设备、多模态（超声+标记数据）联合训练数据集；
3. **临床应用**：开发实时评估系统，集成至超声设备，进行前瞻性临床试验；
4. **轻量化部署**：基于模型剪枝与量化技术，实现边缘设备实时推理。

**附录**

* **代码**（关键函数示例）；
* import os
* import torch
* from torch import nn
* from torch.utils.data import DataLoader, random\_split
* from torch.utils.data import Dataset
* from torchvision import datasets, transforms, models
* from bottleneck\_transformer\_pytorch import BottleStack
* import torch.optim as optim
* from tqdm import tqdm
* from torch.utils.tensorboard import SummaryWriter
* import re
* from collections import defaultdict
* from sklearn.model\_selection import train\_test\_split
* from PIL import Image
* import numpy as np
* from skimage.color import rgb2gray
* from skimage.feature import graycomatrix, graycoprops
* from sklearn.metrics import confusion\_matrix
* import seaborn as sns
* import matplotlib.pyplot as plt
* import cv2
* from skimage.color import rgb2gray
* from skimage.feature import graycomatrix, graycoprops, local\_binary\_pattern
* from skimage.filters import gabor
* from scipy.stats import skew, kurtosis
* def init\_file\_and\_num(number):
* number = number
* folder\_path = f"bottleneck-transformer-pytorch-main\\results\\demo{number}"  #plt.savefig本身不带有创建文件夹的功能 必须手动创建
* # 检查文件夹是否存在，如果不存在则创建
* if not os.path.exists(folder\_path):
* os.makedirs(folder\_path)
* # 编译正则表达式，匹配文件名前缀：一个汉字后跟一个或多个数字
* prefix\_pattern = re.compile(r'^.\*$')
* # 自定义数据集类
* class RockDataset(Dataset):
* def \_\_init\_\_(self, image\_paths, labels, transform=None):
* self.image\_paths = image\_paths
* self.labels = labels
* self.transform = transform
* def \_\_len\_\_(self):
* return len(self.image\_paths)
* def extract\_handcrafted\_features(self, image):
* image\_np = np.array(image)
* image\_np = np.transpose(image\_np, (1, 2, 0))  # (H, W, C)
* # ================= 颜色特征 =================
* # 转换到HSV空间（网页1[1,2,3](@ref)）
* hsv\_image = cv2.cvtColor(image\_np, cv2.COLOR\_RGB2HSV)
* # HSV直方图（32维）
* h\_hist = np.histogram(hsv\_image[...,0], bins=16, range=(0,180))[0]  # H分16bin（网页3[3](@ref)）
* s\_hist = np.histogram(hsv\_image[...,1], bins=8, range=(0,256))[0]   # S分8bin
* v\_hist = np.histogram(hsv\_image[...,2], bins=8, range=(0,256))[0]    # V分8bin
* hsv\_features = np.concatenate([h\_hist, s\_hist, v\_hist])  # 32维
* # 颜色矩（9维，网页4[4,5](@ref)）
* color\_moments = []
* for i in range(3):  # 对H/S/V三个通道
* channel = hsv\_image[...,i].flatten()
* color\_moments.extend([
* np.mean(channel),          # 一阶矩（均值）
* np.std(channel),           # 二阶矩（标准差）
* skew(channel)              # 三阶矩（偏度）
* ])
* color\_moments = np.array(color\_moments)  # 9维
* # ================= 亮度特征 =================（网页6[6,7](@ref)）
* gray\_image = rgb2gray(image\_np) \* 255
* gray\_image = gray\_image.astype(np.uint8)
* # 统计特征（4维）
* brightness\_features = [
* np.mean(gray\_image),          # 平均亮度
* np.std(gray\_image),           # 亮度对比度
* skew(gray\_image.flatten()),   # 亮度分布对称性
* kurtosis(gray\_image.flatten())# 亮度峰态
* ]
* # ================= 纹理特征 =================（网页9[9,10,11](@ref)）
* # GLCM特征（32维）
* glcm = graycomatrix(gray\_image,
* distances=[1, 3],
* angles=[0, np.pi/4, np.pi/2, 3\*np.pi/4],
* levels=256,
* symmetric=True,
* normed=True)
* glcm\_features = []
* for prop in ['contrast', 'homogeneity', 'energy', 'correlation']:
* glcm\_features.extend(graycoprops(glcm, prop).flatten())  # 4属性×4角度×2距离=32维
* # LBP特征（32维，网页10[10](@ref)）
* lbp\_radius = 2
* lbp\_points = 16
* lbp = local\_binary\_pattern(gray\_image,
* P=lbp\_points,
* R=lbp\_radius,
* method='uniform')
* lbp\_hist, \_ = np.histogram(lbp, bins=lbp\_points\*2, range=(0, lbp\_points+2))
* # Gabor滤波特征（32维，网页9[9](@ref)）
* gabor\_features = []
* for theta in np.arange(0, np.pi, np.pi/4):  # 4个方向
* for sigma in [1.0, 3.0]:                # 2个尺度
* filt\_real, filt\_imag = gabor(gray\_image,
* frequency=0.6/sigma,
* theta=theta,
* sigma\_x=sigma,
* sigma\_y=sigma)
* gabor\_features.extend([
* np.mean(filt\_real),  # 实部均值
* np.std(filt\_real),    # 实部方差
* np.mean(filt\_imag),   # 虚部均值
* np.std(filt\_imag)     # 虚部方差
* ] )  # 4方向×2尺度×4统计量=32维
* # ================= 特征融合 =================
* all\_features = np.concatenate([
* hsv\_features,               # 32维
* color\_moments,              # 9维
* brightness\_features,        # 4维
* glcm\_features,              # 32维
* lbp\_hist,                   # 32维
* gabor\_features              # 32维
* ])  # 总维度32+9+4+32+32+32=141
* # 归一化处理（网页3[3](@ref)）
* all\_features = (all\_features - np.mean(all\_features)) / (np.std(all\_features) + 1e-6)
* return torch.tensor(all\_features, dtype=torch.float32)
* def \_\_getitem\_\_(self, idx):
* image\_path = self.image\_paths[idx]
* label = self.labels[idx]
* image = Image.open(image\_path).convert('RGB')
* if self.transform:
* image = self.transform(image)
* # 提取手工特征
* handcrafted\_features = self.extract\_handcrafted\_features(image)
* return image, handcrafted\_features, label
* # 定义 MLP 类
* class MLP(nn.Module):
* def \_\_init\_\_(self, input\_dim, hidden\_dim, output\_dim):
* super().\_\_init\_\_()
* self.fc = nn.Sequential(
* nn.Linear(input\_dim, hidden\_dim),
* nn.BatchNorm1d(hidden\_dim),
* nn.GELU(),
* nn.Dropout(0.5),
* nn.Linear(hidden\_dim, hidden\_dim//2),
* nn.BatchNorm1d(hidden\_dim//2),
* nn.GELU(),
* nn.Dropout(0.3),
* nn.Linear(hidden\_dim//2, output\_dim)
* )
* def forward(self, x):
* # 特征通过多层非线性变换
* x = self.fc(x)  # 维度变化: [batch, input\_dim] → [batch, output\_dim]
* return x
* # 构造 BotNet 模型（利用 ResNet50 的前5个子模块 + BottleStack 模块）
* class BotNet(nn.Module):
* def \_\_init\_\_(self, num\_classes, handcrafted\_feature\_dim, hidden\_dim, freeze\_resnet\_layers):
* super(BotNet, self).\_\_init\_\_()
* # 加载预训练的 ResNet50
* resnet = models.resnet50(weights='IMAGENET1K\_V1')
* backbone = list(resnet.children())[:5]  # 提取前5层
* # 冻结 ResNet50 的前 freeze\_resnet\_layers 层
* for param in list(resnet.parameters())[:freeze\_resnet\_layers]:
* param.requires\_grad = False
* # BottleStack 模块
* bot\_layer = BottleStack(
* dim=256,
* fmap\_size=56,
* dim\_out=2048,
* proj\_factor=4,
* downsample=True,
* heads=4,
* dim\_head=128,
* rel\_pos\_emb=True,
* activation=nn.ReLU()
* )
* # 拼接手工特征和卷积输出特征
* self.model = nn.Sequential(
* \*backbone,
* bot\_layer,
* nn.AdaptiveAvgPool2d((1, 1)),  # 池化
* nn.Flatten(1)
* )
* # 全连接层之前需要拼接手工特征的维度
* self.fc\_input\_dim = 2048 + handcrafted\_feature\_dim  # 2048是卷积输出的通道数
* # 使用 MLP 替代单一的全连接层
* self.mlp = MLP(input\_dim=self.fc\_input\_dim, hidden\_dim=hidden\_dim, output\_dim=num\_classes)
* def forward(self, x, handcrafted\_features):
* # 获取卷积层的输出
* x = self.model(x)
* # 拼接卷积输出和手工特征
* x = torch.cat((x, handcrafted\_features), dim=1)  # 在特征维度拼接
* # 使用 MLP 进行分类
* x = self.mlp(x)
* return x
* def main():
* writer = SummaryWriter(log\_dir=f"bottleneck-transformer-pytorch-main\\runs\\experiment\_{number}")
* # 修正后的训练集增强流程（调整RandomAffine位置）
* train\_transform = transforms.Compose([
* transforms.Resize((224, 224)),
* transforms.RandomHorizontalFlip(),
* transforms.ColorJitter(brightness=0.01, contrast=0.01, saturation=0.01, hue=0.005),
* transforms.ToTensor(),
* ])
* # 修正后的验证集增强流程（使用中心裁剪）
* val\_transform = transforms.Compose([
* transforms.Resize((224, 224)),
* transforms.ToTensor(),
* ])
* dataset\_dir = "bottleneck-transformer-pytorch-main\dataset"
* # 使用 ImageFolder 加载数据集，要求数据集目录下每个子文件夹名称代表一个类别
* full\_dataset = datasets.ImageFolder(root=dataset\_dir)
* # 存储所有图片的路径和标签
* image\_paths = []  # 保存每张图片的完整路径
* labels = []       # 保存每张图片对应的标签
* # 替换为直接划分全部图片索引：
* all\_indices = list(range(len(full\_dataset.imgs)))
* print("总图片数:", len(all\_indices))
* # 填充 image\_paths 和 labels
* image\_paths = [img\_path for img\_path, \_ in full\_dataset.imgs]  # 提取所有图片路径
* labels = [label for \_, label in full\_dataset.imgs]             # 提取所有标签
* train\_indices, val\_indices = train\_test\_split(
* all\_indices,
* test\_size=0.2,
* random\_state=42,
* )
* # 创建数据集时直接使用索引
* train\_dataset = RockDataset(
* image\_paths=[image\_paths[i] for i in train\_indices],
* labels=[labels[i] for i in train\_indices],
* transform=train\_transform
* )
* val\_dataset = RockDataset(
* image\_paths=[image\_paths[i] for i in val\_indices],
* labels=[labels[i] for i in val\_indices],
* transform=val\_transform
* )
* # 创建 DataLoader
* train\_loader = DataLoader(
* train\_dataset,
* batch\_size=10,
* shuffle=True,
* num\_workers=os.cpu\_count()-1,  # 使用全部CPU核心[4,6](@ref)
* pin\_memory=True,               # 启用内存锁页加速传输[4](@ref)
* prefetch\_factor=2,             # 预加载2个batch[7](@ref)
* persistent\_workers=True,        # 保持worker进程存活
* drop\_last = True
* )
* val\_loader = DataLoader(
* val\_dataset,
* batch\_size=8,
* shuffle=True,
* num\_workers=os.cpu\_count()-1,
* pin\_memory=True,
* prefetch\_factor=2,
* persistent\_workers=True,
* drop\_last = True)
* num\_classes = len(full\_dataset.classes)
* print("分类的类别数:", num\_classes)
* print("类别名称:", full\_dataset.classes)
* # 构造 BotNet 模型
* model = BotNet(num\_classes=num\_classes, handcrafted\_feature\_dim=141, hidden\_dim=128, freeze\_resnet\_layers=0)  # 冻结前5层
* device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")
* model.to(device)
* # 定义损失函数和优化器
* class\_weights = torch.tensor([1.0, 1.0], device=device)  # 尝试权重
* criterion = nn.CrossEntropyLoss(weight=class\_weights)  # 修改损失函数定义
* optimizer\_groups = [
* # 第一组：ResNet主干参数（网页1[1](@ref)）
* {'params': [p for n,p in model.named\_parameters()
* if 'model' in n and 'bn' not in n and p.requires\_grad], 'lr': 0.0001},
* # 第二组：MLP全连接层（网页6[6](@ref)）
* {'params': model.mlp.parameters(), 'lr': 0.0003},
* # 第三组：BatchNorm参数（网页3[3](@ref)）
* {'params': [p for n,p in model.named\_parameters()
* if 'bn' in n], 'lr': 0.0001, 'weight\_decay': 0}
* ]
* optimizer = optim.AdamW(optimizer\_groups)
* accumulation\_steps = 1
* # 训练与验证循环
* num\_epochs = 20
* # 创建1Cycle策略学习率调度器
* scheduler = optim.lr\_scheduler.OneCycleLR(
* optimizer,
* max\_lr=[0.0001, 0.0003, 0.0001],  # 对应三个参数组的初始学习率
* pct\_start=0.15,                # 学习率上升阶段占10%（网页4[4](@ref)）
* total\_steps=num\_epochs,
* anneal\_strategy='linear',      # 线性退火（网页2[2](@ref)）
* div\_factor=25.0,               # 初始学习率= max\_lr/div\_factor（网页1[1](@ref)）
* final\_div\_factor=1e4            # 最终学习率=初始学习率/final\_div\_factor
* )
* # 在训练循环中添加（需CUDA支持）
* scaler = torch.amp.GradScaler('cuda',
* init\_scale=16384.0,
* growth\_factor=2,
* backoff\_factor=0.4,
* growth\_interval=100
* )
* for epoch in range(num\_epochs):
* model.train()
* running\_loss = 0.0
* correct\_train = 0
* total\_train = 0
* for step, (images, handcrafted\_features, labels) in enumerate(tqdm(train\_loader, desc=f"Train Epoch {epoch+1}")):
* images = images.to(device)
* handcrafted\_features = handcrafted\_features.to(device)
* labels = labels.to(device)
* optimizer.zero\_grad()
* # 前向计算（自动混合精度）
* with torch.autocast(device\_type='cuda', dtype=torch.float16, enabled=True):
* outputs = model(images, handcrafted\_features)
* loss = criterion(outputs, labels)  # 不再手动除以 accumulation\_steps
* # 反向传播（自动缩放梯度）
* scaler.scale(loss).backward()
* # 梯度累积控制
* if (step + 1) % accumulation\_steps == 0 or (step + 1) == len(train\_loader):
* # 可选：梯度裁剪（需先解除缩放）
* # scaler.unscale\_(optimizer)
* # torch.nn.utils.clip\_grad\_norm\_(model.parameters(), max\_norm=1.0)
* # 更新参数及缩放因子
* scaler.step(optimizer)
* scaler.update()
* optimizer.zero\_grad()
* # 监控缩放因子
* current\_scale = scaler.get\_scale()
* writer.add\_scalar('AMP/scale\_factor', current\_scale, epoch)
* # 统计训练损失和准确率
* running\_loss += loss.item() \* labels.size(0)  # 修正：无需乘以 accumulation\_steps
* \_, predicted = torch.max(outputs, 1)
* total\_train += labels.size(0)
* correct\_train += (predicted == labels).sum().item()
* #训练集loss与学习率
* epoch\_loss = running\_loss / total\_train
* train\_acc = 100 \* correct\_train / total\_train
* print(f"Epoch {epoch+1}: Loss = {epoch\_loss:.4f}, Training Accuracy = {train\_acc:.2f}%")
* #ResNet梯度幅值 对比 MLP梯度幅值 （用于调试）
* #resnet\_grad = torch.norm(torch.cat([p.grad.flatten()
* #            for p in model.model.parameters()]))
* #mlp\_grad = torch.norm(torch.cat([p.grad.flatten()
* #            for p in model.mlp.parameters()]))
* #print(f"ResNet梯度幅值: {resnet\_grad:.4f}, MLP梯度幅值: {mlp\_grad:.4f}")
* # 将训练损失和准确率记录到 TensorBoard
* writer.add\_scalar('Loss/train', epoch\_loss, epoch)
* writer.add\_scalar('Accuracy/train', train\_acc, epoch)
* #writer.add\_scalar('grad/train\_resnet\_grad', resnet\_grad, epoch)
* #writer.add\_scalar('grad/train\_mlp\_grad', mlp\_grad, epoch)
* # 验证阶段
* model.eval()
* correct\_val = 0
* total\_val = 0
* running\_val\_loss = 0.0
* all\_preds = []
* all\_labels = []
* with torch.no\_grad():
* for images, handcrafted\_features, labels in tqdm(val\_loader, desc="Validation"):
* images, handcrafted\_features, labels = images.to(device), handcrafted\_features.to(device), labels.to(device)
* outputs = model(images, handcrafted\_features)
* loss = criterion(outputs, labels)
* running\_val\_loss += loss.item() \* images.size(0)
* \_, predicted = torch.max(outputs, 1)
* all\_preds.extend(predicted.cpu().numpy())
* all\_labels.extend(labels.cpu().numpy())
* total\_val += labels.size(0)
* correct\_val += (predicted == labels).sum().item()
* val\_loss = running\_val\_loss / total\_val
* val\_acc = 100 \* correct\_val / total\_val
* print(f"Validation Loss: {val\_loss:.4f}, Validation Accuracy: {val\_acc:.2f}%")
* # 将验证损失和准确率记录到 TensorBoard
* writer.add\_scalar('Loss/val', val\_loss, epoch)
* writer.add\_scalar('Accuracy/val', val\_acc, epoch)
* # 计算并显示混淆矩阵
* cm = confusion\_matrix(all\_labels, all\_preds)
* sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=full\_dataset.classes, yticklabels=full\_dataset.classes)
* plt.xlabel('Predicted')
* plt.ylabel('True')
* plt.title('Confusion Matrix')
* save\_dir\_plt = f"bottleneck-transformer-pytorch-main\\results\\demo{number}"+f"\\confusion\_matrix\_epoch\_{epoch+1}.png"
* plt.savefig(save\_dir\_plt, bbox\_inches='tight', dpi=300)
* plt.clf()
* # 将学习率记录到 TensorBoard
* writer.add\_scalar('LR/resnet', optimizer.param\_groups[0]['lr'], epoch)
* writer.add\_scalar('LR/mlp', optimizer.param\_groups[1]['lr'], epoch)
* writer.add\_scalar('LR/bn', optimizer.param\_groups[2]['lr'], epoch)
* # 更新学习率
* scheduler.step()
* # 保存模型
* save\_model\_dir = f"bottleneck-transformer-pytorch-main\\results\\demo{number}"+f"\\botnet\_model{number}\_epoch\_{epoch+1}.pth"
* torch.save(model.state\_dict(), save\_model\_dir)
* # 关闭 TensorBoard 的 writer
* writer.close()
* if \_\_name\_\_ == '\_\_main\_\_':
* import multiprocessing
* number = 4  # 设置实验编号
* init\_file\_and\_num(number)  # 初始化文件夹
* multiprocessing.freeze\_support()
* main()