# 具体方法（与文献相对应）

## 1. 逐层解冻（更深度的 Fine‑tuning）（对应文献第一种方法——网络解冻）

在原来只解冻全连接层的基础上，改为解冻 ResNet 后两大 block，使得更多中高级特征可在岩石数据上微调。

# 原来：只优化 fc

- for param in model.parameters():

- param.requires\_grad = False

+ # 先冻结所有层

+ for param in model.parameters():

+ param.requires\_grad = False

+ # 解冻 layer3 和 layer4

+ for name, module in model.named\_children():

+ if name in ['layer3', 'layer4', 'fc']:

+ for param in module.parameters():

+ param.requires\_grad = True

# 损失 & 优化器：将优化器参数改为所有 requires\_grad=True 的参数

- optimizer = optim.Adam(model.fc.parameters(), lr=learning\_rate)

+ params\_to\_optimize = [p for p in model.parameters() if p.requires\_grad]

+ optimizer = optim.Adam(params\_to\_optimize, lr=learning\_rate)

## 2. Mixup 数据增广（对应文献的第二种方法——强化数据增广）

在训练循环里，将每个 batch 的样本做 mixup，再送入网络。

from torch.utils.data import DataLoader

def mixup\_data(x, y, alpha=0.4):

'''返回混合后的 inputs, pairs of targets, and lambda'''

if alpha > 0:

lam = np.random.beta(alpha, alpha)

else:

lam = 1

batch\_size = x.size()[0]

index = torch.randperm(batch\_size).to(x.device)

mixed\_x = lam \* x + (1 - lam) \* x[index, :]

y\_a, y\_b = y, y[index]

return mixed\_x, y\_a, y\_b, lam

# 训练循环内部（phase=='train'处）：

for inputs, labels in tqdm(data\_loaders[phase], desc=phase):

inputs, labels = inputs.to(device), labels.to(device)

+ # mixup

+ inputs, labels\_a, labels\_b, lam = mixup\_data(inputs, labels, alpha=0.4)

optimizer.zero\_grad()

with torch.set\_grad\_enabled(phase=='train'):

outputs = model(inputs)

- loss = criterion(outputs, labels)

+ # mixup loss

+ loss = lam \* criterion(outputs, labels\_a) + (1-lam) \* criterion(outputs, labels\_b)

## 3. CutMix 数据增广（对应文献的第二种方法——强化数据增广）

在训练前对一个 batch 随机执行 CutMix：

import torchvision.transforms.functional as TF

def cutmix\_data(x, y, alpha=1.0):

if alpha > 0:

lam = np.random.beta(alpha, alpha)

else:

lam = 1

B, C, H, W = x.size()

index = torch.randperm(B).to(x.device)

# 生成随机裁剪框

cut\_rat = np.sqrt(1. - lam)

cut\_w = np.int(W \* cut\_rat)

cut\_h = np.int(H \* cut\_rat)

cx = np.random.randint(W)

cy = np.random.randint(H)

bbx1 = np.clip(cx - cut\_w // 2, 0, W)

bby1 = np.clip(cy - cut\_h // 2, 0, H)

bbx2 = np.clip(cx + cut\_w // 2, 0, W)

bby2 = np.clip(cy + cut\_h // 2, 0, H)

x[:, :, bby1:bby2, bbx1:bbx2] = x[index, :, bby1:bby2, bbx1:bbx2]

lam = 1 - ((bbx2 - bbx1) \* (bby2 - bby1) / (W \* H))

y\_a, y\_b = y, y[index]

return x, y\_a, y\_b, lam

# 训练循环内部（phase=='train'处）:

for inputs, labels in tqdm(data\_loaders[phase], desc=phase):

inputs, labels = inputs.to(device), labels.to(device)

+ # CutMix

+ inputs, labels\_a, labels\_b, lam = cutmix\_data(inputs, labels, alpha=1.0)

optimizer.zero\_grad()

with torch.set\_grad\_enabled(phase=='train'):

outputs = model(inputs)

- loss = criterion(outputs, labels)

+ loss = lam \* criterion(outputs, labels\_a) + (1-lam) \* criterion(outputs, labels\_b)

## 4. 更高效网络——EfficientNet（对应文献中的第三种方法——采用更高效的网络结构）

直接用 torchvision 提供的 EfficientNet\_B0，并同样 fine‑tune 最后一层或更多层：

- model = models.resnet50(pretrained=True)

+ from torchvision.models import efficientnet\_b0, EfficientNet\_B0\_Weights

+ # 加载官方预训练权重

+ weights = EfficientNet\_B0\_Weights.DEFAULT

+ model = efficientnet\_b0(weights=weights)

# 冻结所有

for param in model.parameters():

param.requires\_grad = False

# 替换分类头

- in\_features = model.fc.in\_features

- model.fc = nn.Linear(in\_features, num\_classes)

+ in\_features = model.classifier[1].in\_features

+ model.classifier[1] = nn.Linear(in\_features, num\_classes)

model = model.to(device)

criterion = nn.CrossEntropyLoss()

- optimizer = optim.Adam(model.fc.parameters(), lr=learning\_rate)

+ optimizer = optim.Adam(model.classifier[1].parameters(), lr=learning\_rate)

## 5. 学习率调度—余弦退火（Cosine Annealing LR）（对应文献第四种方法——学习率调度）

在训练前构造调度器，并在每个 epoch 结束后 step：

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.fc.parameters(), lr=learning\_rate)

+ from torch.optim.lr\_scheduler import CosineAnnealingLR

+ # T\_max 设为总迭代次数或 epoch 数

+ scheduler = CosineAnnealingLR(optimizer, T\_max=num\_epochs, eta\_min=1e-6)

for epoch in range(num\_epochs):

print(f'Epoch {epoch+1}/{num\_epochs}')

for phase in ['train', 'valid']:

...

# 在每个 epoch 结束后更新学习率

+ scheduler.step()

## 6. 模型融合（Ensemble）（对应文献第七种方法——模型融合）

训练出两个结构（ResNet50 与 EfficientNet）并保存为 resnet.pth、effb0.pth，最后测试时平均它们的 logits：

# 测试前：加载两模型权重

model1 = models.resnet50(pretrained=False)

model1.fc = nn.Linear(model1.fc.in\_features, num\_classes)

model1.load\_state\_dict(torch.load('resnet.pth'))

model1.to(device).eval()

+ from torchvision.models import efficientnet\_b0, EfficientNet\_B0\_Weights

model2 = efficientnet\_b0(weights=None)

model2.classifier[1] = nn.Linear(model2.classifier[1].in\_features, num\_classes)

model2.load\_state\_dict(torch.load('effb0.pth'))

model2.to(device).eval()

preds, labels = [], []

for inputs, labs in tqdm(data\_loaders['test'], desc='test'):

inputs, labs = inputs.to(device), labs.to(device)

with torch.no\_grad():

- outputs = model(inputs)

- \_, batch\_preds = torch.max(outputs, 1)

+ out1 = model1(inputs)

+ out2 = model2(inputs)

+ # 平均 logits 再取最大

+ avg = (out1 + out2) / 2

+ \_, batch\_preds = torch.max(avg, 1)

preds.extend(batch\_preds.cpu().numpy())

labels.extend(labs.cpu().numpy())