import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, Dataset

from torchvision import transforms

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

import cv2

from tqdm import tqdm

import random

import pennylane as qml

from sklearn.metrics import confusion\_matrix

import pandas as pd

from matplotlib.colors import LinearSegmentedColormap

import torch.nn.init as init

import os

# ==================== 字体设置 ====================

plt.rcParams["font.family"] = ["SimHei", "Microsoft YaHei"]

plt.rcParams['axes.unicode\_minus'] = False # 解决负号显示问题

# ==================== 路径设置 ====================

ROOT\_DIR = r"D:\DALUNWEN"

DATASET\_DIR = os.path.join(ROOT\_DIR, "dataset")

TRAIN\_DIR = os.path.join(DATASET\_DIR, "train")

VAL\_DIR = os.path.join(DATASET\_DIR, "val")

TEST\_DIR = os.path.join(DATASET\_DIR, "test")

PRETRAINED\_PATH = os.path.join(ROOT\_DIR, "pretrained", "medical\_shadow\_pretrained.pth")

OUTPUT\_DIR = os.path.join(ROOT\_DIR, "output", "A1")

MODEL\_DIR = os.path.join(OUTPUT\_DIR, "models")

VISUALIZATION\_DIR = os.path.join(OUTPUT\_DIR, "visualizations")

LOG\_PATH = os.path.join(OUTPUT\_DIR, "training\_log.csv")

# 创建输出目录

os.makedirs(MODEL\_DIR, exist\_ok=True)

os.makedirs(VISUALIZATION\_DIR, exist\_ok=True)

# ==================== 随机种子设置 ====================

def set\_seed(seed=42):

random.seed(seed)

np.random.seed(seed)

torch.manual\_seed(seed)

torch.cuda.manual\_seed(seed)

torch.backends.cudnn.deterministic = True

torch.backends.cudnn.benchmark = False

os.environ['PYTHONHASHSEED'] = str(seed)

set\_seed()

# ==================== CBAM结构 ====================

class CBAM(nn.Module):

"""卷积块注意力模块：通道注意力+空间注意力"""

def \_\_init\_\_(self, in\_channels, reduction\_ratio=16, kernel\_size=7):

super().\_\_init\_\_()

# 1. 通道注意力

self.channel\_attn = nn.Sequential(

nn.AdaptiveAvgPool2d(1), # 全局平均池化获取通道特征

nn.Conv2d(in\_channels, in\_channels // reduction\_ratio, kernel\_size=1),

nn.ReLU(inplace=True),

nn.Conv2d(in\_channels // reduction\_ratio, in\_channels, kernel\_size=1),

nn.Sigmoid() # 输出通道权重（0-1）

)

# 2. 空间注意力

self.spatial\_attn = nn.Sequential(

nn.Conv2d(2, 1, kernel\_size=kernel\_size, padding=kernel\_size // 2, bias=False),

nn.Sigmoid() # 输出空间权重（0-1）

)

def forward(self, x):

# 通道注意力加权

channel\_weight = self.channel\_attn(x)

x = x \* channel\_weight # 逐通道加权

# 空间注意力加权（基于通道池化特征）

avg\_out = torch.mean(x, dim=1, keepdim=True) # 通道平均

max\_out, \_ = torch.max(x, dim=1, keepdim=True) # 通道最大

spatial\_map = torch.cat([avg\_out, max\_out], dim=1) # 拼接为2通道

spatial\_weight = self.spatial\_attn(spatial\_map)

x = x \* spatial\_weight # 逐空间位置加权

return x

# ==================== 量子特征蒸馏器 ====================

class QuantumDistiller(nn.Module):

"""量子特征蒸馏器：量子比特纠缠层+动态门控"""

def \_\_init\_\_(self, in\_channels, out\_channels, pretrained=None, gate\_threshold=0.05, debug=False):

super().\_\_init\_\_()

self.in\_channels = in\_channels

self.out\_channels = out\_channels

self.n\_qubits = min(8, in\_channels) # 量子比特数根据输入通道数动态调整（不超过8）

self.gate\_threshold = gate\_threshold

self.debug = debug # 控制调试输出

self.debug\_counter = 0

# 经典预处理

self.preprocess = nn.Sequential(

nn.Linear(in\_channels, 64),

nn.ReLU(),

nn.Linear(64, self.n\_qubits)

)

# 经典后处理

self.postprocess = nn.Sequential(

nn.Linear(self.n\_qubits, 64),

nn.ReLU(),

nn.Linear(64, out\_channels)

)

# 局部特征转换

self.local\_transform = nn.Conv2d(in\_channels, out\_channels, kernel\_size=1)

# 量子电路初始化（支持批处理）

self.quantum\_circuit = self.create\_quantum\_circuit()

self.quantum\_weights = nn.Parameter(0.1 \* torch.randn(3 \* self.n\_qubits)) # 量子门参数

self.channel\_weights = nn.Parameter(torch.ones(in\_channels) / in\_channels) # 通道权重

# 加载预训练权重

if pretrained and os.path.exists(pretrained):

try:

self.load\_pretrained\_weights(pretrained)

except Exception as e:

print(f"警告：加载预训练权重失败 {pretrained}，错误: {e}")

print("将从头训练量子层")

elif pretrained:

print(f"警告：预训练权重文件不存在 {pretrained}，将从头训练量子层")

# 权重初始化

self.\_initialize\_weights()

def \_initialize\_weights(self):

"""改进的权重初始化"""

for m in self.modules():

if isinstance(m, nn.Conv2d):

init.kaiming\_normal\_(m.weight, mode='fan\_out', nonlinearity='relu')

elif isinstance(m, nn.BatchNorm2d):

init.constant\_(m.weight, 1)

init.constant\_(m.bias, 0)

elif isinstance(m, nn.Linear):

init.xavier\_normal\_(m.weight)

if m.bias is not None:

init.constant\_(m.bias, 0)

def create\_quantum\_circuit(self):

"""量子比特纠缠电路（支持批处理输入）"""

dev = qml.device("default.qubit", wires=self.n\_qubits)

@qml.qnode(dev, interface="torch", diff\_method="backprop")

def circuit(inputs, weights):

# 输入编码（支持批处理）

for i in range(self.n\_qubits):

qml.RY(np.pi \* inputs[:, i], wires=i) # 注意这里的[:, i]处理批处理

# 变分层1

for i in range(self.n\_qubits):

qml.RY(weights[i], wires=i)

# 纠缠层

for i in range(0, self.n\_qubits - 1, 2):

qml.CNOT(wires=[i, i + 1])

# 变分层2

for i in range(self.n\_qubits):

qml.RY(weights[self.n\_qubits + i], wires=i)

# 测量

return [qml.expval(qml.PauliZ(i)) for i in range(self.n\_qubits)]

return circuit

def load\_pretrained\_weights(self, pretrained\_path):

"""加载量子层预训练权重"""

state\_dict = torch.load(pretrained\_path, map\_location=torch.device('cpu'))

quantum\_state\_dict = {k: v for k, v in state\_dict.items() if 'quantum' in k or 'channel\_weights' in k}

self.load\_state\_dict(quantum\_state\_dict, strict=False)

print(f"已加载预训练权重：{pretrained\_path}")

def forward(self, x):

batch\_size, \_, H, W = x.shape

# 全局特征提取

x\_global = torch.mean(x, dim=(2, 3)) # [B, C]

# 量子特征处理（批处理优化）

weighted\_input = x\_global \* self.channel\_weights # 通道加权 [B, C]

x\_quantum = self.quantum\_circuit(weighted\_input, self.quantum\_weights) # [n\_qubits, B]

x\_quantum = torch.stack(x\_quantum).T.float() # 转置为 [B, n\_qubits]

# 动态门控

quantum\_strength = torch.norm(x\_quantum, p=2, dim=1, keepdim=True) # [B, 1]

gate = torch.sigmoid(quantum\_strength - self.gate\_threshold) # 使用sigmoid平滑门控

# 量子特征扩展到空间维度

x\_global\_processed = self.postprocess(x\_quantum) # [B, out\_channels]

x\_global\_spatial = x\_global\_processed.view(batch\_size, self.out\_channels, 1, 1)

x\_global\_spatial = nn.functional.adaptive\_avg\_pool2d(x\_global\_spatial, (H, W))

# 局部特征转换

x\_local = self.local\_transform(x) # [B, out\_channels, H, W]

# 记录监控指标

self.quantum\_strength = quantum\_strength.mean().item()

self.gate\_activation = gate.mean().item()

# 确保门控维度与特征图匹配

if gate.dim() == 2:

gate = gate.unsqueeze(-1).unsqueeze(-1) # [B, 1] -> [B, 1, 1, 1]

# 确保尺寸匹配

if x\_local.size() != x\_global\_spatial.size():

x\_global\_spatial = nn.functional.interpolate(

x\_global\_spatial, size=x\_local.size()[2:], mode='bilinear', align\_corners=False

)

# 调试输出（可控）

if self.debug and self.debug\_counter % 100 == 0:

print(

f"量子蒸馏器调试: 输入尺寸={x.shape}, 局部特征尺寸={x\_local.shape}, 全局特征尺寸={x\_global\_spatial.shape}, 门控尺寸={gate.shape}")

self.debug\_counter += 1

return x\_local + gate \* x\_global\_spatial, x\_quantum, quantum\_strength

# ==================== 量子特征映射器（新增） ====================

class QuantumFeatureMapper(nn.Module):

"""量子特征映射器：通过经典计算模拟量子特性，实现稳定的特征增强"""

def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=3):

super().\_\_init\_\_()

self.in\_channels = in\_channels

self.out\_channels = out\_channels

# 1. 量子态映射（模拟叠加态）

self.superposition = nn.Sequential(

nn.Conv2d(in\_channels, 64, kernel\_size=1),

nn.BatchNorm2d(64),

nn.Tanh() # 限制特征范围至[-1,1]，模拟量子态振幅

)

# 2. 纠缠关联（模拟量子纠缠，强化空间关联性）

self.entanglement = nn.Sequential(

nn.Conv2d(64, 64, kernel\_size=kernel\_size, padding=kernel\_size // 2, groups=8), # 分组卷积模拟局部纠缠

nn.BatchNorm2d(64),

nn.Tanh()

)

# 3. 经典投影（模拟量子测量）

self.projection = nn.Sequential(

nn.Conv2d(64, out\_channels, kernel\_size=1),

nn.BatchNorm2d(out\_channels),

nn.Softmax(dim=1) # 模拟测量概率分布

)

# 4. 残差连接（控制量子特征贡献度）

self.residual\_scale = nn.Parameter(torch.tensor(0.2)) # 固定初始权重0.2

# 权重初始化

self.\_initialize\_weights()

def \_initialize\_weights(self):

"""初始化卷积层权重"""

for m in self.modules():

if isinstance(m, nn.Conv2d):

nn.init.kaiming\_normal\_(m.weight, mode='fan\_out', nonlinearity='relu')

if m.bias is not None:

nn.init.zeros\_(m.bias)

elif isinstance(m, nn.BatchNorm2d):

nn.init.ones\_(m.weight)

nn.init.zeros\_(m.bias)

def forward(self, x):

# x: 输入特征图 [B, C, H, W]

# 1. 模拟量子叠加态（特征维度扩展与归一化）

x\_super = self.superposition(x) # [B, 64, H, W]

# 2. 模拟量子纠缠（强化空间局部关联）

x\_entangle = self.entanglement(x\_super) # [B, 64, H, W]

# 3. 模拟量子测量（投影回经典特征空间）

quantum\_feature = self.projection(x\_entangle) # [B, out\_channels, H, W]

# 4. 残差融合（固定权重控制贡献度）

if x.size(1) != self.out\_channels:

x = nn.Conv2d(self.in\_channels, self.out\_channels, kernel\_size=1)(x) # 特征维度匹配

fused\_feature = x + self.residual\_scale \* quantum\_feature # 残差连接

return {

'fused': fused\_feature,

'quantum\_feature': quantum\_feature # 中间量子特征（用于可视化）

}

# ==================== UNet输出卷积 ====================

class OutConv(nn.Module):

"""输出卷积层，将特征映射到目标类别数"""

def \_\_init\_\_(self, in\_channels, out\_channels):

super(OutConv, self).\_\_init\_\_()

self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size=1)

def forward(self, x):

return self.conv(x)

# ==================== UNet骨干网络 ====================

class DoubleConv(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels, mid\_channels=None):

super().\_\_init\_\_()

if not mid\_channels:

mid\_channels = out\_channels

self.double\_conv = nn.Sequential(

nn.Conv2d(in\_channels, mid\_channels, kernel\_size=3, padding=1, bias=False),

nn.BatchNorm2d(mid\_channels),

nn.ReLU(inplace=True),

nn.Conv2d(mid\_channels, out\_channels, kernel\_size=3, padding=1, bias=False),

nn.BatchNorm2d(out\_channels),

nn.ReLU(inplace=True)

)

# 根据通道数动态调整CBAM的reduction\_ratio

reduction\_ratio = max(4, out\_channels // 32) # 避免过小的压缩比

self.cbam = CBAM(out\_channels, reduction\_ratio=reduction\_ratio)

def forward(self, x):

x = self.double\_conv(x)

x = self.cbam(x) # 应用卷积块内的CBAM

return x

class Down(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels):

super().\_\_init\_\_()

self.maxpool = nn.MaxPool2d(2) # 分离出maxpool以便添加CBAM

# 根据通道数动态调整CBAM参数

reduction\_ratio = max(4, in\_channels // 32)

self.cbam\_after\_pool = CBAM(in\_channels, reduction\_ratio=reduction\_ratio)

# 卷积块（已包含CBAM）

self.conv = DoubleConv(in\_channels, out\_channels)

def forward(self, x):

x = self.maxpool(x)

x = self.cbam\_after\_pool(x) # 下采样后添加CBAM

x = self.conv(x) # 卷积块内已有CBAM

return x

class Up(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels, bilinear=True):

super().\_\_init\_\_()

if bilinear:

self.up = nn.Upsample(scale\_factor=2, mode='bilinear', align\_corners=True)

self.conv = DoubleConv(in\_channels, out\_channels, in\_channels // 2)

# 动态调整CBAM参数

cbam\_channels = in\_channels // 2

else:

self.up = nn.ConvTranspose2d(in\_channels // 2, in\_channels // 2, kernel\_size=2, stride=2)

self.conv = DoubleConv(in\_channels, out\_channels)

# 动态调整CBAM参数

cbam\_channels = in\_channels // 2

reduction\_ratio = max(4, cbam\_channels // 32)

self.cbam\_after\_up = CBAM(cbam\_channels, reduction\_ratio=reduction\_ratio)

# 解码器额外增强CBAM

final\_reduction = max(4, out\_channels // 32)

self.cbam\_final = CBAM(out\_channels, reduction\_ratio=final\_reduction)

def forward(self, x1, x2):

x1 = self.up(x1)

x1 = self.cbam\_after\_up(x1) # 上采样后添加CBAM

# 处理尺寸差异

diffY = x2.size()[2] - x1.size()[2]

diffX = x2.size()[3] - x1.size()[3]

x1 = nn.functional.pad(x1, [diffX // 2, diffX - diffX // 2,

diffY // 2, diffY - diffY // 2])

x = torch.cat([x2, x1], dim=1)

x = self.conv(x) # 卷积块内已有CBAM

x = self.cbam\_final(x) # 最终CBAM增强

return x

class QuantumEnhancedUNet(nn.Module):

"""量子增强UNet，在解码器每一步添加QuantumFeatureMapper"""

def \_\_init\_\_(self, n\_channels=3, n\_classes=1, bilinear=True, pretrained=None, debug=False):

super().\_\_init\_\_()

self.n\_channels = n\_channels

self.n\_classes = n\_classes

self.bilinear = bilinear

self.debug = debug

# 编码器 - 每层都包含CBAM

self.inc = DoubleConv(n\_channels, 64) # 包含CBAM

self.down1 = Down(64, 128) # 包含2个CBAM

self.down2 = Down(128, 256) # 包含2个CBAM

self.down3 = Down(256, 512) # 包含2个CBAM

factor = 2 if bilinear else 1

self.down4 = Down(512, 1024 // factor) # 包含2个CBAM

# 量子特征蒸馏器

self.quantum\_distiller = QuantumDistiller(

in\_channels=1024 // factor,

out\_channels=1024 // factor,

pretrained=pretrained,

debug=debug

)

# 解码器 - 每层都包含CBAM

self.up1 = Up(1024, 512 // factor, bilinear) # 包含3个CBAM

self.up2 = Up(512, 256 // factor, bilinear) # 包含3个CBAM

self.up3 = Up(256, 128 // factor, bilinear) # 包含3个CBAM

self.up4 = Up(128, 64, bilinear) # 包含3个CBAM

# 在解码器每一步添加QuantumFeatureMapper（新增部分）

self.mapper1 = QuantumFeatureMapper(in\_channels=512 // factor, out\_channels=512 // factor)

self.mapper2 = QuantumFeatureMapper(in\_channels=256 // factor, out\_channels=256 // factor)

self.mapper3 = QuantumFeatureMapper(in\_channels=128 // factor, out\_channels=128 // factor)

self.mapper4 = QuantumFeatureMapper(in\_channels=64, out\_channels=64)

self.outc = OutConv(64, n\_classes)

def forward(self, x):

# 编码路径

x1 = self.inc(x)

x2 = self.down1(x1)

x3 = self.down2(x2)

x4 = self.down3(x3)

x5 = self.down4(x4)

if self.debug:

print(f"UNet编码路径: x1={x1.shape}, x2={x2.shape}, x3={x3.shape}, x4={x4.shape}, x5={x5.shape}")

# 量子特征蒸馏

x5, x\_quantum, quantum\_strength = self.quantum\_distiller(x5)

# 解码路径（每一步添加QuantumFeatureMapper）

x = self.up1(x5, x4)

x = self.mapper1(x)['fused'] # 解码器第一步添加映射器

x = self.up2(x, x3)

x = self.mapper2(x)['fused'] # 解码器第二步添加映射器

x = self.up3(x, x2)

x = self.mapper3(x)['fused'] # 解码器第三步添加映射器

x = self.up4(x, x1)

x = self.mapper4(x)['fused'] # 解码器第四步添加映射器

logits = self.outc(x)

return {

'logits': logits,

'quantum\_features': x\_quantum,

'quantum\_strength': quantum\_strength,

'gate\_activation': self.quantum\_distiller.gate\_activation

}

# ==================== 数据预处理与加载 ====================

class Compose:

def \_\_init\_\_(self, transforms):

self.transforms = transforms

def \_\_call\_\_(self, img, mask):

for t in self.transforms:

img, mask = t(img, mask)

return img, mask

class Resize:

def \_\_init\_\_(self, size):

self.size = size

def \_\_call\_\_(self, img, mask):

img = img.resize(self.size, Image.BICUBIC)

mask = mask.resize(self.size, Image.NEAREST)

return img, mask

class RandomHorizontalFlip:

def \_\_init\_\_(self, p=0.5):

self.p = p

def \_\_call\_\_(self, img, mask):

if random.random() < self.p:

img = img.transpose(Image.FLIP\_LEFT\_RIGHT)

mask = mask.transpose(Image.FLIP\_LEFT\_RIGHT)

return img, mask

class RandomVerticalFlip:

def \_\_init\_\_(self, p=0.5):

self.p = p

def \_\_call\_\_(self, img, mask):

if random.random() < self.p:

img = img.transpose(Image.FLIP\_TOP\_BOTTOM)

mask = mask.transpose(Image.FLIP\_TOP\_BOTTOM)

return img, mask

class RandomRotation:

def \_\_init\_\_(self, degrees):

self.degrees = degrees

def \_\_call\_\_(self, img, mask):

angle = random.uniform(-self.degrees, self.degrees)

img = img.rotate(angle, Image.BICUBIC)

mask = mask.rotate(angle, Image.NEAREST)

return img, mask

class RandomCrop:

def \_\_init\_\_(self, size):

self.size = size

def \_\_call\_\_(self, img, mask):

w, h = img.size

th, tw = self.size

if w == tw and h == th:

return img, mask

i = random.randint(0, h - th)

j = random.randint(0, w - tw)

img = img.crop((j, i, j + tw, i + th))

mask = mask.crop((j, i, j + tw, i + th))

return img, mask

class ToTensor:

def \_\_call\_\_(self, img, mask):

img = transforms.functional.to\_tensor(img)

mask = torch.from\_numpy(np.array(mask, dtype=np.float32)).unsqueeze(0)

mask = torch.clamp(mask, 0, 1)

return img, mask

class Normalize:

def \_\_init\_\_(self, mean, std):

self.mean = mean

self.std = std

def \_\_call\_\_(self, img, mask):

img = transforms.functional.normalize(img, self.mean, self.std)

return img, mask

class ShadowDataset(Dataset):

"""改进的数据集加载器，支持跳过损坏文件"""

def \_\_init\_\_(self, root\_dir, split="train", transform=None):

self.root\_dir = root\_dir

self.split = split

self.transform = transform

self.images\_dir = os.path.join(root\_dir, split, "images")

self.masks\_dir = os.path.join(root\_dir, split, "masks")

print(f"检查数据集路径: {self.images\_dir}")

print(f"检查掩码路径: {self.masks\_dir}")

if not os.path.exists(self.images\_dir):

raise FileNotFoundError(f"图像目录不存在: {self.images\_dir}")

if not os.path.exists(self.masks\_dir):

raise FileNotFoundError(f"掩码目录不存在: {self.masks\_dir}")

# 加载并验证有效图像

self.valid\_images = []

image\_files = sorted([f for f in os.listdir(self.images\_dir)

if f.lower().endswith(('.tif', '.png', '.jpg', '.jpeg'))])

mask\_files = sorted([f for f in os.listdir(self.masks\_dir)

if f.lower().endswith(('.png', '.jpg', '.jpeg'))])

# 构建图像和掩码的基础名称映射

image\_bases = {os.path.splitext(f)[0]: f for f in image\_files}

mask\_bases = {os.path.splitext(f)[0]: f for f in mask\_files}

# 验证并收集有效文件对

for base\_name in image\_bases.keys() & mask\_bases.keys():

img\_file = image\_bases[base\_name]

mask\_file = mask\_bases[base\_name]

# 验证图像文件是否有效

img\_path = os.path.join(self.images\_dir, img\_file)

mask\_path = os.path.join(self.masks\_dir, mask\_file)

try:

# 快速验证图像是否可以打开

with Image.open(img\_path) as img:

img.verify()

with Image.open(mask\_path) as mask:

mask.verify()

self.valid\_images.append((img\_file, mask\_file))

except Exception as e:

print(f"跳过无效文件对: {img\_file}, {mask\_file}, 错误: {e}")

print(f"{split} 集加载完成，有效图像-掩码对: {len(self.valid\_images)}")

if len(self.valid\_images) == 0:

raise ValueError(f"{split} 集无有效图像-掩码对")

def \_\_len\_\_(self):

return len(self.valid\_images)

def \_\_getitem\_\_(self, idx):

img\_name, mask\_name = self.valid\_images[idx]

img\_path = os.path.join(self.images\_dir, img\_name)

mask\_path = os.path.join(self.masks\_dir, mask\_name)

try:

img = Image.open(img\_path).convert('RGB')

mask = Image.open(mask\_path).convert('L')

# 应用数据转换

if self.transform is not None:

img, mask = self.transform(img, mask)

# 验证数据类型

if not isinstance(img, torch.Tensor):

img = transforms.functional.to\_tensor(img)

if not isinstance(mask, torch.Tensor):

mask = torch.from\_numpy(np.array(mask, dtype=np.float32)).unsqueeze(0)

# 确保掩码值在0-1范围内

mask = torch.clamp(mask, 0, 1)

return img, mask

except Exception as e:

print(f"警告：加载文件失败: 图像={img\_name}, 掩码={mask\_name}, 错误={e}")

# 替换为随机有效样本而非返回None

fallback\_idx = random.randint(0, len(self) - 1)

return self.\_\_getitem\_\_(fallback\_idx)

# ==================== 评价指标 ====================

def calculate\_iou(pred, target, threshold=0.5):

"""计算IoU"""

pred = (torch.sigmoid(pred) > threshold).float()

intersection = (pred \* target).sum(dim=(1, 2, 3))

union = (pred + target).sum(dim=(1, 2, 3)) - intersection

iou = (intersection + 1e-6) / (union + 1e-6)

return iou.mean().item()

def calculate\_dice(pred, target, threshold=0.5):

"""计算Dice系数"""

pred = (torch.sigmoid(pred) > threshold).float()

intersection = (pred \* target).sum(dim=(1, 2, 3))

dice = (2 \* intersection + 1e-6) / (pred.sum(dim=(1, 2, 3)) + target.sum(dim=(1, 2, 3)) + 1e-6)

return dice.mean().item()

def calculate\_metrics(pred, target, threshold=0.5):

"""计算多种评价指标"""

pred = (torch.sigmoid(pred) > threshold).float()

target = target.float()

# 计算真阳性、假阳性、真阴性、假阴性

tp = (pred \* target).sum().item()

fp = (pred \* (1 - target)).sum().item()

tn = ((1 - pred) \* (1 - target)).sum().item()

fn = ((1 - pred) \* target).sum().item()

# 计算精确率、召回率和F1分数

precision = (tp + 1e-8) / (tp + fp + 1e-8)

recall = (tp + 1e-8) / (tp + fn + 1e-8)

f1 = 2 \* (precision \* recall) / (precision + recall + 1e-8)

return {

'precision': precision,

'recall': recall,

'f1': f1,

'tp': tp,

'fp': fp,

'tn': tn,

'fn': fn

}

# ==================== 损失函数 ====================

class DiceLoss(nn.Module):

def \_\_init\_\_(self, smooth=1.0):

super().\_\_init\_\_()

self.smooth = smooth

def forward(self, pred, target):

pred = torch.sigmoid(pred)

intersection = (pred \* target).sum(dim=(2, 3))

union = pred.sum(dim=(2, 3)) + target.sum(dim=(2, 3))

dice = (2 \* intersection + self.smooth) / (union + self.smooth)

return 1 - dice.mean()

class BCEWithLogitsLoss(nn.Module):

def \_\_init\_\_(self, weight=None):

super().\_\_init\_\_()

self.bce\_loss = nn.BCEWithLogitsLoss(weight=weight)

def forward(self, pred, target):

return self.bce\_loss(pred, target)

class CombinedLoss(nn.Module):

def \_\_init\_\_(self, dice\_weight=0.5, bce\_weight=0.5, smooth=1.0):

super().\_\_init\_\_()

self.dice\_loss = DiceLoss(smooth=smooth)

self.bce\_loss = BCEWithLogitsLoss()

self.dice\_weight = dice\_weight

self.bce\_weight = bce\_weight

def forward(self, pred, target):

target = torch.clamp(target, 0, 1)

return self.dice\_weight \* self.dice\_loss(pred, target) + self.bce\_weight \* self.bce\_loss(pred, target)

# ==================== 量子特征热力图可视化 ====================

def visualize\_quantum\_heatmap(model, dataloader, device, save\_dir, epoch, num\_samples=5):

"""可视化量子特征热力图（优化为CVPR风格：关联分割贡献）"""

model.eval()

with torch.no\_grad():

for i, (images, masks) in enumerate(dataloader):

if i >= num\_samples:

break

images = images.to(device)

masks = masks.to(device)

outputs = model(images)

logits = outputs['logits']

quantum\_features = outputs['quantum\_features']

for j in range(min(images.size(0), 2)): # 每个批次最多可视化2个样本

# 数据准备

img = images[j].cpu().permute(1, 2, 0).numpy()

img = (img - img.min()) / (img.max() - img.min()) # 归一化

mask = masks[j].cpu().squeeze().numpy()

pred = torch.sigmoid(logits[j]).cpu().squeeze().numpy()

q\_features = quantum\_features[j].cpu().numpy()

# 热力图生成

H, W = mask.shape # 获取掩码尺寸（与图像一致）

q\_heatmap = np.zeros((H, W), dtype=np.float32) # 初始化二维热力图

# 热力图关联分割贡献

# 1. 量子特征强度归一化（0~1）

q\_feat\_norm = (q\_features - q\_features.min()) / (q\_features.max() - q\_features.min() + 1e-8)

avg\_q\_strength = q\_feat\_norm.mean() # 量子特征平均强度（量化贡献）

# 2. 分割预测置信度（0~1，越高说明预测越可信）

pred\_confidence = pred # 直接使用模型预测掩码的置信度

# 3. 热力图 = 量子特征强度 × 预测置信度（关联任务：量子对可信预测的贡献）

for i\_pixel in range(H):

for j\_pixel in range(W):

q\_heatmap[i\_pixel, j\_pixel] = avg\_q\_strength \* pred\_confidence[i\_pixel, j\_pixel]

# 4. 避开掩码区域（仅显示背景中量子特征的作用）

q\_heatmap[mask >= 0.5] = 0

# 处理无效量子特征

if np.isnan(q\_heatmap).any() or (q\_heatmap.max() - q\_heatmap.min() < 1e-6):

q\_heatmap = np.zeros\_like(q\_heatmap)

print(f"样本 {i \* 2 + j} 量子特征无效，已使用零矩阵替代")

else:

# 归一化

q\_min, q\_max = q\_heatmap.min(), q\_heatmap.max()

q\_heatmap = (q\_heatmap - q\_min) / (q\_max - q\_min)

# 避开掩码区域

masked\_heatmap = np.ma.masked\_where(mask < 0.5, q\_heatmap)

# 绘图逻辑

fig, axes = plt.subplots(1, 4, figsize=(20, 5), gridspec\_kw={'width\_ratios': [1, 1, 1, 1.2]})

# 原始图像

axes[0].imshow(img)

axes[0].set\_title("原始图像", fontsize=14)

axes[0].axis('off')

# 真实掩码

axes[1].imshow(mask, cmap='gray')

axes[1].set\_title("真实掩码", fontsize=14)

axes[1].axis('off')

# 预测掩码

axes[2].imshow(pred, cmap='viridis')

axes[2].set\_title("预测掩码", fontsize=14)

axes[2].axis('off')

# 量子热力图

axes[3].imshow(img)

# CVPR风格配色

quantum\_cmap = LinearSegmentedColormap.from\_list(

"cvpr\_heatmap",

[(0.0, 0.0, 1.0), # 低强度（蓝）

(0.0, 1.0, 1.0), # 中低（青）

(1.0, 1.0, 0.0), # 中高（黄）

(1.0, 0.0, 0.0)] # 高强度（红）

)

heatmap\_im = axes[3].imshow(masked\_heatmap, cmap=quantum\_cmap, alpha=0.6)

# 标题添加量化指标

axes[3].set\_title(f"量子特征热力图（贡献度：{avg\_q\_strength:.2f}）", fontsize=14)

axes[3].axis('off')

# 颜色条

cbar = fig.colorbar(heatmap\_im, ax=axes[3], orientation='vertical', pad=0.02)

cbar.set\_label('量子特征叠加强度', fontsize=12)

cbar.ax.tick\_params(labelsize=10)

plt.tight\_layout()

plt.savefig(

os.path.join(save\_dir, f"quantum\_heatmap\_epoch\_{epoch}\_sample\_{i \* 2 + j}.png"),

dpi=300, bbox\_inches='tight'

)

plt.close()

# ==================== 学习率预热 ====================

class LRSchedulerWithWarmup:

"""学习率调度器，带预热阶段"""

def \_\_init\_\_(self, optimizer, warmup\_epochs=5, total\_epochs=100, start\_lr=1e-5, max\_lr=1e-3, min\_lr=1e-6):

self.optimizer = optimizer

self.warmup\_epochs = warmup\_epochs

self.total\_epochs = total\_epochs

self.start\_lr = start\_lr

self.max\_lr = max\_lr

self.min\_lr = min\_lr

self.current\_epoch = 0

# 初始化学习率

for param\_group in self.optimizer.param\_groups:

param\_group['lr'] = self.start\_lr

def step(self):

self.current\_epoch += 1

if self.current\_epoch <= self.warmup\_epochs:

# 预热阶段：线性增加学习率

lr = self.start\_lr + (self.max\_lr - self.start\_lr) \* (self.current\_epoch / self.warmup\_epochs)

else:

# 预热后：余弦退火衰减

progress = (self.current\_epoch - self.warmup\_epochs) / (self.total\_epochs - self.warmup\_epochs)

lr = self.min\_lr + 0.5 \* (self.max\_lr - self.min\_lr) \* (1 + np.cos(np.pi \* progress))

# 更新学习率

for param\_group in self.optimizer.param\_groups:

param\_group['lr'] = lr

return lr

# ==================== 训练函数 ====================

def train\_model(debug=False):

# 设置设备

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

print(f"使用设备: {device}")

# 定义数据变换

train\_transform = Compose([

Resize((224, 224)), # 统一调整为224x224

RandomHorizontalFlip(p=0.5),

RandomVerticalFlip(p=0.5),

RandomRotation(degrees=15),

ToTensor(),

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

val\_transform = Compose([

Resize((224, 224)), # 统一调整为224x224

ToTensor(),

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

# 加载数据集

print("加载训练集...")

train\_dataset = ShadowDataset(root\_dir=DATASET\_DIR, split="train", transform=train\_transform)

print("加载验证集...")

val\_dataset = ShadowDataset(root\_dir=DATASET\_DIR, split="val", transform=val\_transform)

# 创建数据加载器

train\_loader = DataLoader(train\_dataset, batch\_size=16, shuffle=True, num\_workers=4, pin\_memory=True)

val\_loader = DataLoader(val\_dataset, batch\_size=16, shuffle=False, num\_workers=4, pin\_memory=True)

# 初始化模型

model = QuantumEnhancedUNet(

n\_channels=3,

n\_classes=1,

pretrained=PRETRAINED\_PATH,

debug=debug

).to(device)

# 定义损失函数和优化器

criterion = CombinedLoss(dice\_weight=0.5, bce\_weight=0.5)

optimizer = optim.Adam(model.parameters(), lr=0.001, weight\_decay=1e-5)

# 学习率调度器（带预热）

scheduler = LRSchedulerWithWarmup(

optimizer,

warmup\_epochs=5,

total\_epochs=100,

start\_lr=1e-5,

max\_lr=1e-3,

min\_lr=1e-6

)

# 记录最佳模型

best\_val\_iou = 0.0

best\_epoch = 0

# 早停机制

patience = 15

no\_improve\_epochs = 0

# 创建日志文件

with open(LOG\_PATH, 'w') as f:

f.write(

"Epoch,Train Loss,Train IoU,Train Dice,Val Loss,Val IoU,Val Dice,Val Precision,Val Recall,Val F1,Quantum Strength,Gate Activation,Learning Rate\n")

# 训练循环

for epoch in range(1, 101): # 100个epochs

print(f"\n===== Epoch {epoch}/{100} =====")

# 训练阶段

model.train()

train\_loss = 0.0

train\_iou = 0.0

train\_dice = 0.0

quantum\_strengths = []

gate\_activations = []

train\_progress = tqdm(train\_loader, desc="训练中")

for images, masks in train\_progress:

images = images.to(device)

masks = masks.to(device)

# 前向传播

outputs = model(images)

logits = outputs['logits']

loss = criterion(logits, masks)

# 反向传播

optimizer.zero\_grad()

loss.backward()

# 梯度裁剪

torch.nn.utils.clip\_grad\_norm\_(model.parameters(), max\_norm=1.0)

optimizer.step()

# 记录指标

batch\_size = images.size(0)

train\_loss += loss.item() \* batch\_size

train\_iou += calculate\_iou(logits, masks) \* batch\_size

train\_dice += calculate\_dice(logits, masks) \* batch\_size

# 记录量子特征强度和门控激活

quantum\_strengths.append(outputs['quantum\_strength'].mean().item())

gate\_activations.append(outputs['gate\_activation'])

# 更新进度条

train\_progress.set\_postfix({

'loss': loss.item(),

'iou': calculate\_iou(logits, masks),

'dice': calculate\_dice(logits, masks),

'q\_strength': outputs['quantum\_strength'].mean().item(),

'gate': outputs['gate\_activation']

})

# 计算平均指标

train\_loss /= len(train\_dataset)

train\_iou /= len(train\_dataset)

train\_dice /= len(train\_dataset)

avg\_quantum\_strength = sum(quantum\_strengths) / len(quantum\_strengths) if quantum\_strengths else 0

avg\_gate\_activation = sum(gate\_activations) / len(gate\_activations) if gate\_activations else 0

print(

f"训练: Loss={train\_loss:.4f}, IoU={train\_iou:.4f}, Dice={train\_dice:.4f}, 量子强度={avg\_quantum\_strength:.4f}, 门控激活={avg\_gate\_activation:.4f}")

# 验证阶段

model.eval()

val\_loss = 0.0

val\_iou = 0.0

val\_dice = 0.0

val\_precision = 0.0

val\_recall = 0.0

val\_f1 = 0.0

val\_progress = tqdm(val\_loader, desc="验证中")

with torch.no\_grad():

for images, masks in val\_progress:

images = images.to(device)

masks = masks.to(device)

# 前向传播

outputs = model(images)

logits = outputs['logits']

loss = criterion(logits, masks)

# 计算指标

batch\_size = images.size(0)

val\_loss += loss.item() \* batch\_size

val\_iou += calculate\_iou(logits, masks) \* batch\_size

val\_dice += calculate\_dice(logits, masks) \* batch\_size

# 计算详细指标

metrics = calculate\_metrics(logits, masks)

val\_precision += metrics['precision'] \* batch\_size

val\_recall += metrics['recall'] \* batch\_size

val\_f1 += metrics['f1'] \* batch\_size

# 更新进度条

val\_progress.set\_postfix({

'loss': loss.item(),

'iou': calculate\_iou(logits, masks),

'dice': calculate\_dice(logits, masks)

})

# 计算平均指标

val\_loss /= len(val\_dataset)

val\_iou /= len(val\_dataset)

val\_dice /= len(val\_dataset)

val\_precision /= len(val\_dataset)

val\_recall /= len(val\_dataset)

val\_f1 /= len(val\_dataset)

# 获取当前学习率

current\_lr = optimizer.param\_groups[0]['lr']

print(

f"验证: Loss={val\_loss:.4f}, IoU={val\_iou:.4f}, Dice={val\_dice:.4f}, Precision={val\_precision:.4f}, Recall={val\_recall:.4f}, F1={val\_f1:.4f}, LR={current\_lr:.6f}")

# 学习率调整

scheduler.step()

# 保存日志

with open(LOG\_PATH, 'a') as f:

f.write(

f"{epoch},{train\_loss},{train\_iou},{train\_dice},{val\_loss},{val\_iou},{val\_dice},{val\_precision},{val\_recall},{val\_f1},{avg\_quantum\_strength},{avg\_gate\_activation},{current\_lr}\n")

# 保存最佳模型

if val\_iou > best\_val\_iou:

best\_val\_iou = val\_iou

best\_epoch = epoch

torch.save(model.state\_dict(), os.path.join(MODEL\_DIR, f"best\_model.pth"))

print(f"保存最佳模型: Epoch {epoch}, Val IoU: {val\_iou:.4f}")

no\_improve\_epochs = 0 # 重置早停计数器

else:

no\_improve\_epochs += 1

print(f"早停计数器: {no\_improve\_epochs}/{patience}")

if no\_improve\_epochs >= patience:

print(f"早停于Epoch {epoch}，验证指标不再提升")

break

# 保存当前模型

torch.save(model.state\_dict(), os.path.join(MODEL\_DIR, f"model\_epoch\_{epoch}.pth"))

# 可视化量子特征

if epoch % 10 == 0: # 每10个epoch可视化一次

visualize\_quantum\_heatmap(model, val\_loader, device, VISUALIZATION\_DIR, epoch)

print(f"训练完成！最佳模型在第 {best\_epoch} 个epoch，验证IoU为 {best\_val\_iou:.4f}")

# 加载最佳模型并在测试集上评估

checkpoint = torch.load(os.path.join(MODEL\_DIR, f"best\_model.pth"), map\_location=device)

model.load\_state\_dict(checkpoint)

model.eval()

# 准备测试集

print("加载测试集...")

test\_transform = Compose([

Resize((224, 224)),

ToTensor(),

Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

test\_dataset = ShadowDataset(root\_dir=DATASET\_DIR, split="test", transform=test\_transform)

test\_loader = DataLoader(test\_dataset, batch\_size=16, shuffle=False, num\_workers=4, pin\_memory=True)

print("在测试集上评估...")

test\_loss = 0.0

test\_iou = 0.0

test\_dice = 0.0

test\_precision = 0.0

test\_recall = 0.0

test\_f1 = 0.0

test\_progress = tqdm(test\_loader, desc="测试中")

with torch.no\_grad():

for images, masks in test\_progress:

images = images.to(device)

masks = masks.to(device)

outputs = model(images)

logits = outputs['logits']

loss = criterion(logits, masks)

batch\_size = images.size(0)

test\_loss += loss.item() \* batch\_size

test\_iou += calculate\_iou(logits, masks) \* batch\_size

test\_dice += calculate\_dice(logits, masks) \* batch\_size

metrics = calculate\_metrics(logits, masks)

test\_precision += metrics['precision'] \* batch\_size

test\_recall += metrics['recall'] \* batch\_size

test\_f1 += metrics['f1'] \* batch\_size

test\_progress.set\_postfix({

'loss': loss.item(),

'iou': calculate\_iou(logits, masks),

'dice': calculate\_dice(logits, masks)

})

test\_loss /= len(test\_dataset)

test\_iou /= len(test\_dataset)

test\_dice /= len(test\_dataset)

test\_precision /= len(test\_dataset)

test\_recall /= len(test\_dataset)

test\_f1 /= len(test\_dataset)

print(

f"测试结果: Loss={test\_loss:.4f}, IoU={test\_iou:.4f}, Dice={test\_dice:.4f}, Precision={test\_precision:.4f}, Recall={test\_recall:.4f}, F1={test\_f1:.4f}")

# 保存测试结果

with open(os.path.join(OUTPUT\_DIR, "test\_results.txt"), 'w') as f:

f.write(f"测试结果:\n")

f.write(f"Loss: {test\_loss:.4f}\n")

f.write(f"IoU: {test\_iou:.4f}\n")

f.write(f"Dice: {test\_dice:.4f}\n")

f.write(f"Precision: {test\_precision:.4f}\n")

f.write(f"Recall: {test\_recall:.4f}\n")

f.write(f"F1: {test\_f1:.4f}\n")

# 运行训练

if \_\_name\_\_ == "\_\_main\_\_":

# 设置debug=True可以启用调试输出

train\_model(debug=False)