基于迁移学习的PPG信号内瘘狭窄检测

基于光电容积脉搏波（PPG）信号和迁移学习的动静脉内瘘（AVF）狭窄程度分类系统。

1、项目概述

本项目基于PyTorch实现，采用预训练的PPG信号编码器模型，通过迁移学习技术应用于动静脉内瘘狭窄三分类任务。系统输入为600点PPG信号片段（采样频率50Hz，时长12秒），输出分类结果：

- 类别0：正常（无狭窄）
- 类别1：轻度狭窄
- 类别2：重度狭窄

该模型融合了残差GRU网络、多尺度卷积、Cross-Gating机制和渐进式特征适应层等创新结构，有效提升了内瘘狭窄检测的准确性和稳定性。

2、主要特点

- **迁移学习框架**：利用在大规模PPG数据集上预训练的编码器，迁移到内瘘狭窄分类任务，有效解决医疗场景下样本量不足问题
- **网络结构**：
  - 残差GRU：通过残差连接和双向结构捕获PPG信号中的时序关系，避免深层网络中的梯度消失问题
  - 多尺度卷积：同时使用小、中、大尺度卷积核提取不同尺度特征，增强对短期、中期和长期模式的感知能力
  - 跨门控机制（Cross-Gating）：增强特征表示能力和跨任务学习能力，在源域和目标域间构建有效的信息桥梁
  - 渐进式适应层：平滑实现源域到目标域的特征迁移，缓解域偏移问题
  - 通道注意力机制：自适应学习特征通道重要性，突出关键生理特征的贡献
- **全面评估体系**：
  - 5折交叉验证确保结果稳定性
  - 多分类评价指标（准确率、F1分数、精确率、召回率）
  - 二分类评价指标（敏感性、特异性、F1）
  - 可视化

3、环境依赖

- Python >= 3.7
- PyTorch >= 1.12.0
- NumPy
- Pandas
- Scikit-learn
- Matplotlib
- Seaborn
- SciPy
- tqdm

可通过以下命令安装所需依赖：

```
pip install torch numpy pandas scikit-learn matplotlib seaborn scipy tqdm
```

4、目录结构

```
PPG-AVF-Stenosis-Detection/
├── data.py              # 数据加载与预处理模块
├── model.py             # 模型架构定义
├── train_eval.py        # 训练与评估函数
├── utils.py             # 评估指标与可视化工具
├── run.py               # 命令行接口
├── README.md            # 项目说明文档
├── dataset/             # PPG信号CSV文件
|   ├── xxx-0.csv            # 正常样本
|   ├── xxx-1.csv            # 重度狭窄样本
|   └── xxx-2.csv            # 轻度狭窄样本
├── pretrained_models/  # 预训练编码器模型
|   └── ppg_pretrain_encoder_best.pth  # 预训练模型权重
└── results/             # 训练结果与可视化
    ├── fold_1/          # 第1折交叉验证结果
    ├── fold_2/          # 第2折交叉验证结果
    ├── ...
    └── summary_results.json  # 汇总结果
```

目录文件说明

本项目包含以下关键Python文件，每个文件都有特定功能：

`data.py`

数据加载与预处理模块，主要功能：

- `AVFDataset` 类：负责加载CSV文件并预处理PPG信号
- 实现带通滤波、标准化和数据质量控制
- 提供数据集随机分割与批量加载功能

`model.py`

模型架构定义模块，包含：

- `Encoder` 类：PPG信号编码器，包含多尺度卷积和残差GRU
- `ResidualGRU` 类：实现带残差连接的双向GRU
- `CrossGatingModule` 类：实现特征交叉增强模块
- `ProgressiveAdaptationLayer` 类：实现渐进式特征适应层
- `ImprovedAVFClassifier` 类：最终的内瘘狭窄分类模型

`train_eval.py`

训练与评估功能模块，实现：

- `load_pretrained_model` 函数：加载预训练编码器
- `evaluate_model` 函数：计算多种评估指标
- `train_transfer_model` 函数：实现迁移学习训练流程
- `main` 函数：实现5折交叉验证和结果汇总

`utils.py`

工具函数模块，提供：

- `generate_classification_report` 函数：生成分类报告
- `plot_confusion_matrix` 函数：绘制混淆矩阵
- `plot_overall_confusion_matrix` 函数：绘制总体混淆矩阵
- `visualize_feature_importance` 函数：可视化特征重要性

`run.py`

命令行接口模块，功能包括：

- 解析命令行参数
- 验证输入参数有效性
- 记录训练配置
- 启动训练流程

5、数据准备

数据格式要求

本模型需要以下格式的PPG信号数据：

1. CSV文件，每个文件仅包含一列标记为'PPG'的信号数据
2. 每个文件包含600个数据点（采样率50Hz，对应12秒片段）
3. 文件命名格式为xxxx-`[类别标签].csv`，如：xxx-0.csv`（正常）`、xxx-1.csv`（重度狭窄）`、xxx-2.csv`（轻度狭窄）`（即类别标签从文件名最后一个'-'之后的数字中提取）

数据预处理流程

在`data.py`中实现了完整的数据预处理流程：

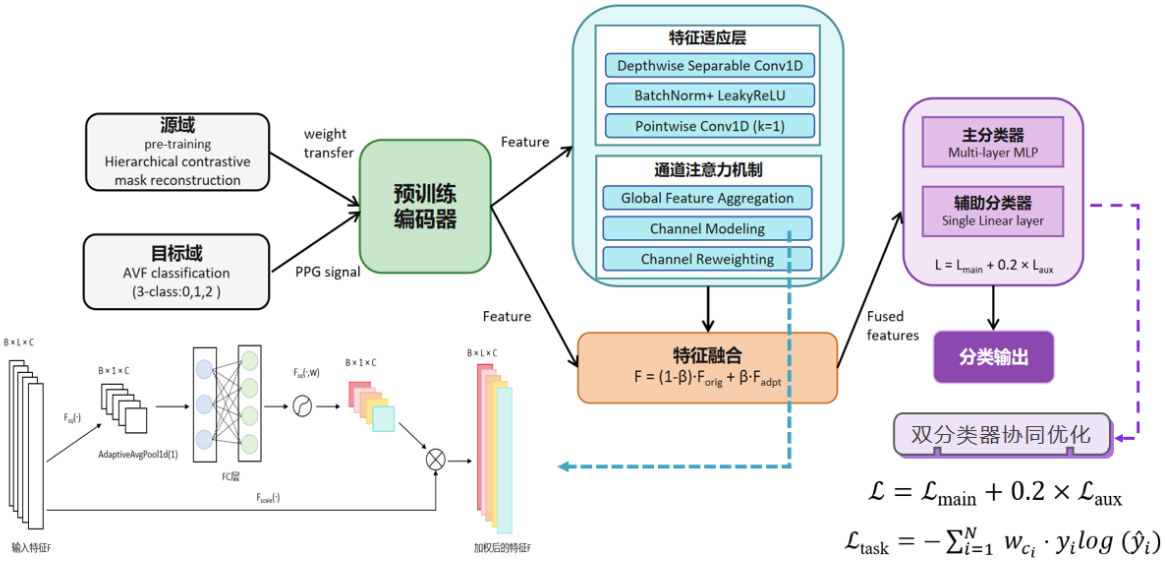
1. **信号过滤**：应用带通滤波器（0.5-10Hz）去除基线漂移和高频噪声
2. **标准化**：零均值单位方差标准化，确保不同病人间信号可比
3. **质量控制**：
    - 自动跳过包含NaN值的信号
    - 剔除方差过小（几乎无变化）的信号
    - 检查信号长度是否满足要求（600点）

数据加载类

`AVFDataset`类负责加载和预处理数据：

```
dataset = AVFDataset(
    data_folder="/path/to/dataset",    #数据集存放路径
    apply_filter=True  # 是否应用滤波
)
```

6、模型架构详解



编码器（Encoder）

编码器是迁移学习的核心组件，主要包含：

1. **投影层**：1×1卷积将输入通道从1扩展到32，显著增加表达能力而不增加计算复杂度

```python
self.projection_conv = nn.Sequential(
    nn.Conv1d(1, 32, kernel_size=1),
    nn.BatchNorm1d(32),
    nn.ReLU()
)
```

2. **多尺度卷积模块**：并行提取不同时间尺度特征，有效捕获PPG信号中的短期、中期和长期变化模式
   - 小尺度卷积（3×1核）：捕获局部特征，适合检测快速变化和尖峰
   - 中尺度卷积（5×1核）：捕获中等范围特征，平衡局部和全局信息
   - 大尺度卷积（7×1核）：捕获全局特征，有助于识别缓慢变化的趋势

```python
# 多尺度卷积示例（小尺度）
self.conv_small = nn.Sequential(
    nn.Conv1d(32, 64, kernel_size=3, padding=1),
    nn.BatchNorm1d(64),
    nn.ReLU()
)
```

```

3. **残差GRU**：双向GRU带残差连接，增强时序建模能力，缓解梯度消失问题，加速收敛过程
   ```python
   self.residual_gru = ResidualGRU(
       input_size=128,
       hidden_size=hidden_size//2,  # 因为是双向的，所以隐藏层大小减半
       num_layers=num_layers,
       dropout=dropout
   )
```
```

分类器（ImprovedAVFClassifier）

分类器模型结构包含：

1. **特征提取层**：预训练的Encoder模块，借助大规模数据学习的通用表征能力
2. **渐进式适应层**：平滑迁移源域到目标域的特征空间，逐步将预训练特征调整至下游任务分布，缓解特征偏移问题。
3. **主分类头**：最终分类层，采用逐层压缩与归一化策略，增强特征表达非线性能力的同时，降低过拟合风险，稳定训练过程。
4. **辅助分类头**：在训练早期提供额外监督信号，有助于稳定梯度传播、辅助主分支优化收敛。

渐进式特征适应层

渐进式特征适应层结合了深度可分离卷积、通道注意力机制和可学习的混合参数，实现了预训练特征到目标任务的平滑过渡：

```
class ProgressiveAdaptationLayer(nn.Module):
    def __init__(self, hidden_dim=128, dropout_rate=0.3):
        super(ProgressiveAdaptationLayer, self).__init__()

        # 渐进式特征适应模块：使用深度可分离卷积（depthwise + pointwise）以降低参数量和计算复杂度
        self.adaptation = nn.Sequential(
            nn.Conv1d(hidden_dim, hidden_dim, kernel_size=3, padding=1, groups=hidden_dim),  # Depthwise卷积：每个通道单独卷积
            nn.BatchNorm1d(hidden_dim),                # 批标准化，加速收敛、稳定训练
            nn.LeakyReLU(0.2),                         # 激活函数，带负斜率以避免“死神经元”
            nn.Conv1d(hidden_dim, hidden_dim, kernel_size=1),  # Pointwise卷积：整合通道间信息
            nn.BatchNorm1d(hidden_dim),                # 再次标准化
            nn.LeakyReLU(0.2),                         # 激活
            nn.Dropout(dropout_rate)                   # Dropout正则化，防止过拟合
        )

        # 通道注意力模块：通过通道平均池化+两层卷积学习每个通道的重要性
        self.channel_attention = nn.Sequential(
            nn.AdaptiveAvgPool1d(1),                   # 将每个通道压缩为1个数：全局平均池化
            nn.Conv1d(hidden_dim, hidden_dim // 8, kernel_size=1),  # 降维映射
            nn.ReLU(),                                 # 非线性激活
            nn.Conv1d(hidden_dim // 8, hidden_dim, kernel_size=1),  # 升维映射
            nn.Sigmoid()                               # 输出每个通道的注意力权重（0~1）
        )

        # alpha为一个可学习的标量参数，用于控制原始特征与适应特征的混合程度
        self.alpha = nn.Parameter(torch.tensor(0.5))  # 初始值设为0.5

    def forward(self, x, training_progress=1.0):
        # 输入x维度：[batch_size, seq_len, hidden_dim]
        # 转换为Conv1d期望的格式：[batch_size, hidden_dim, seq_len]
        x = x.permute(0, 2, 1)
```

```
32
33          # 对输入特征执行适应性卷积变换，生成“适应特征”
34          adapted = self.adaptation(x)
35
36          # 通过通道注意力机制增强关键通道特征，提升判别性
37          attention = self.channel_attention(adapted)  # 输出维度：[batch_size,
    hidden_dim, 1]
38          adapted = adapted * attention                # 通道加权融合
39
40          # 通过sigmoid(alpha)将其约束在0~1，并与训练进度training_progress相乘
41          # training_progress可用于控制训练前期少量适应，后期增加适应比例
42          mix_ratio = torch.sigmoid(self.alpha) * training_progress
43
44          # 最终输出是原始输入x与适应特征adapted的加权混合
45          output = (1 - mix_ratio) * x + mix_ratio * adapted
46
47          return output  # 输出维度：[batch_size, hidden_dim, seq_len]
48
```

设计优势：

1. **深度可分离卷积**：减少参数量的同时保持表达能力，适合小样本迁移学习
2. **通道注意力机制**：自适应识别并增强重要生理特征通道，抑制噪声通道
3. **自适应混合比例**：可学习的alpha参数结合训练进度，动态调整原始特征与适应特征的平衡
4. **渐进式训练策略**：随着训练推进逐步增加适应特征的权重，确保平稳过渡

迁移学习策略

模型在训练过程中采用以下迁移学习策略：

1. **编码器冻结阶段**：训练初期冻结预训练编码器，仅训练分类层，避免预训练知识过早丢失
2. **逐步解冻策略**：随着训练进行，逐步解冻编码器层，自底向上细调适应目标任务

```
1  def update_epoch(self, current, total):
2      # 记录当前训练轮数
3      self.current_epoch = current
4
5      # 记录总训练轮数
6      self.total_epochs = total
7
8   # 计算当前训练进度（0 ~ 1），用于控制逐步解冻的强度
9      progress = current / total
10
11     # 从训练进度超过50%开始，逐步解冻模型编码器层
12     # 若 progress < 0.5，unfreeze_progress 为 0，保持冻结状态；
13     # 若 progress >= 0.5，线性增长，最终达到1（完全解冻）
14     self.unfreeze_progress = max(0, (progress - 0.5) * 2)
15
```

3. **学习率分层设置**：编码器和分类层使用不同学习率，确保预训练知识稳定迁移

```
optimizer = optim.AdamW([
    {'params': [p for n, p in model.named_parameters() if 'encoder' not in n], 'lr': initial_lr},
    {'params': model.encoder.parameters(), 'lr': initial_lr * 0.1}  # 编码器使用更小学习率
], weight_decay=1e-4)
```

7、训练流程

交叉验证流程

本项目使用5折交叉验证保证结果稳定性：

1. **数据集分割**：使用StratifiedKFold保持每折中类别分布一致

```
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
for fold, (train_idx, val_idx) in enumerate(skf.split(data, labels)):
    # 训练和验证
```

2. **模型训练**：每折训练一个模型，使用余弦退火学习率

```
scheduler = CosineAnnealingWarmRestarts(
    optimizer,
    T_0=10,  # 初始周期
    T_mult=2,  # 周期倍增因子
    eta_min=1e-6  # 最小学习率
)
```

3. **最佳模型保存**：每折保存验证F1分数最高的模型

```
if val_result['f1_score'] > best_val_f1_macro:
    best_val_f1_macro = val_result['f1_score']
    torch.save({
        'model_state_dict': model.state_dict(),
        'val_f1': best_val_f1_macro,
        'epoch': epoch
    }, best_model_save_path)
```

4. **结果汇总**：汇总所有折的结果，计算平均性能指标

```
print(f"平均准确率: {np.mean(accuracies):.4f} ± {np.std(accuracies):.4f}")
print(f"平均宏平均F1分数: {np.mean(macro_f1_scores):.4f} ± {np.std(macro_f1_scores):.4f}")
```

损失函数

训练中使用主分类器和辅助分类器的组合损失：

```
main_loss = criterion(main_outputs, labels)
aux_loss = criterion(aux_outputs, labels)
loss = main_loss + 0.2 * aux_loss  # 辅助损失权重为0.2
```

8、评估指标详解

三分类指标

针对三分类任务（正常、轻度狭窄、重度狭窄）的评估指标：

1. **准确率（Accuracy）**：正确分类的样本比例

```
accuracy = accuracy_score(y_true, y_pred)
```

2. **F1分数**：精确率和召回率的调和平均
   - 宏平均F1：所有类别F1的平均值
   - 各类别F1：每个类别单独的F1分数

```
macro_f1 = f1_score(y_true, y_pred, average='macro')
class_f1 = f1_score(y_true, y_pred, average=None)
```

3. **精确率和召回率**：

```
precision = precision_score(y_true, y_pred, average='macro')
recall = recall_score(y_true, y_pred, average='macro')
```

````
### 二分类指标

将正常（0）视为阴性，轻度狭窄（1）和重度狭窄（2）合并为阳性的评估指标：

1. **敏感性（Sensitivity）**：识别出患者的能力
   ```python
   # 敏感性即为阳性样本的召回率
   binary_sensitivity = recall_score(binary_y_true, binary_y_pred, pos_label=1)
````

2. **特异性（Specificity）**：识别出健康人的能力

```
# 特异性 = TN / (TN + FP)
tn, fp, fn, tp = confusion_matrix(binary_y_true, binary_y_pred).ravel()
specificity = tn / (tn + fp)
```

3. **二分类F1分数**：在二分类任务下的F1分数

```
binary_f1 = f1_score(binary_y_true, binary_y_pred, pos_label=1)
```

9、使用说明

模型训练

1. **准备环境**：安装所需依赖

```
pip install torch numpy pandas scikit-learn matplotlib seaborn scipy tqdm
```

2. **准备数据**：按要求格式组织PPG数据
```

3. **执行训练**：

```
python run.py --data_folder ./dataset \
              --pretrained_model
./pretrained_models/ppg_pretrain_encoder_best.pth \
              --save_dir ./results
```

命令行参数

- `--data_folder`：包含PPG信号CSV文件的数据目录
- `--pretrained_model`：预训练编码器模型路径
- `--save_dir`：结果保存目录
- `--seed`：随机种子，用于结果复现（默认：42）
- `--epochs`：训练轮数（默认：100）
- `--lr`：初始学习率（默认：0.001）
- `--batch_size`：批次大小（默认：32）

10、模型推理

使用训练好的模型对新数据进行预测的步骤：

模型加载

```python
import torch
from model import ImprovedAVFClassifier, Encoder

def load_model(model_path, num_classes=3):
    # 初始化编码器
    encoder = Encoder(input_size=600, hidden_size=128)

    # 创建分类器
    model = ImprovedAVFClassifier(
        pretrained_encoder=encoder,
        hidden_dim=128,
        num_classes=num_classes,
        dropout_rate=0.5
    )

    # 加载模型权重
    checkpoint = torch.load(model_path, map_location='cpu')
    if 'model_state_dict' in checkpoint:
        model.load_state_dict(checkpoint['model_state_dict'])
    else:
        model.load_state_dict(checkpoint)

    # 切换到评估模式
    model.eval()

    return model
```

11、输出文件

训练过程会生成以下输出文件：

每折结果（存储在`fold_{i}`文件夹）

- `avf_classifier_best.pth`：最佳模型权重
- `classification_report.txt`：分类性能报告

汇总结果（存储在`save_dir`目录）

- `cross_validation_results.csv`：交叉验证结果表格
- `summary_results.json`：汇总统计数据