

心跳信号分类预测实验报告

课程名称：机器学习

课程类别：专业选修课

任课教师：文勇

授课时间：2024 年 3 月 8 日至 2024 年 7 月 5 日

学 号：202312143002062

姓 名：顾佳凯

专业名称：计算机科学与技术

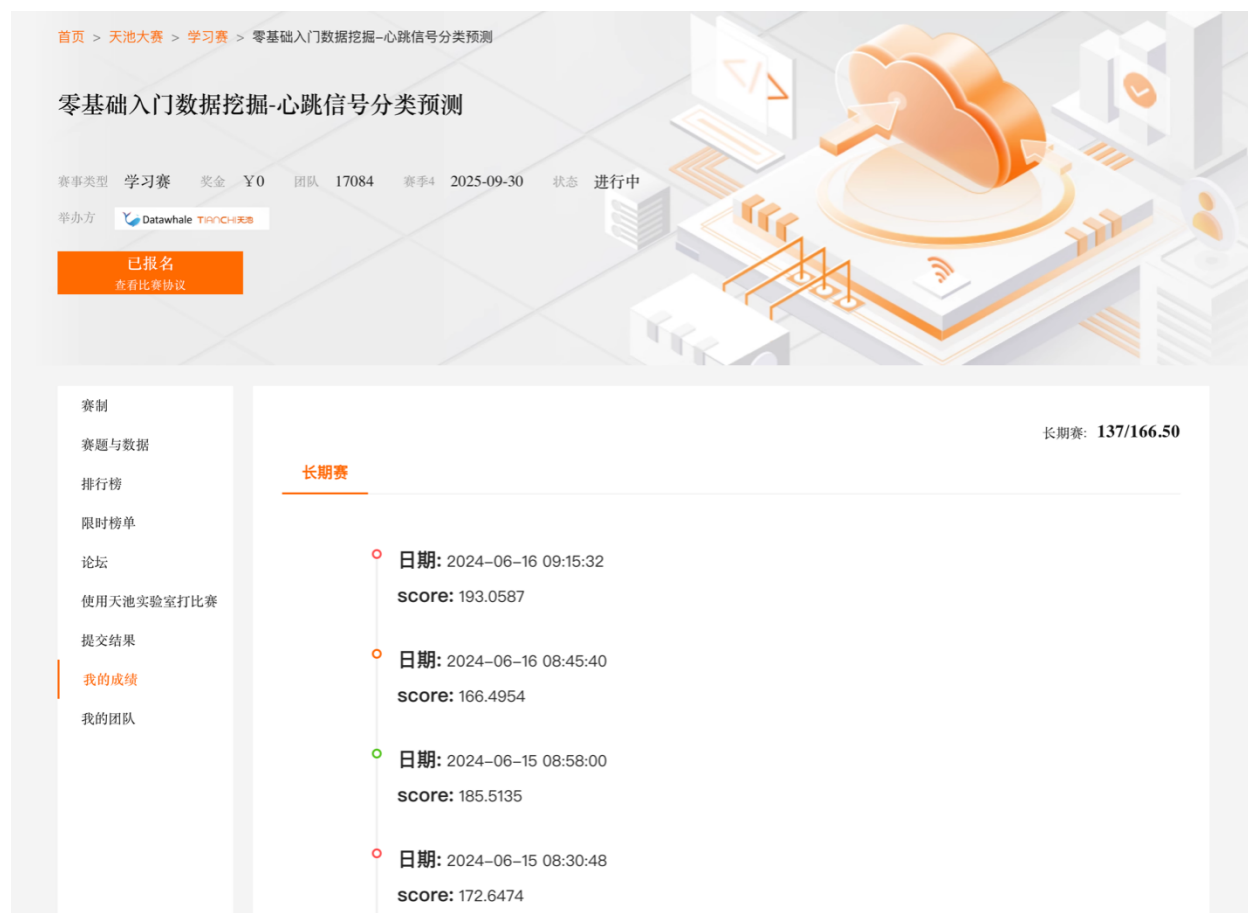
所在学院：人工智能学院

目录

心跳信号分类预测实验报告	1
1. 天池排名和分数截图	3
2. 赛题解析	4
3. 实验环境	5
4. 实验代码解析	6
5. 模型结构图	22
6. 损失函数描述	24
7. 实验总结	25

1. 天池排名和分数截图

一共提交了 20 次左右，最好的成绩：166.4954，截至 2024 年 6 月 17 日名次为 137。打算提交了实验报告后，再去找班长要一份他的 154 成绩的代码，学习一下人家的高分技巧。



图一：天池排名和分数截图

2. 赛题解析

本心跳信号分类预测赛题以医疗数据挖掘为背景,旨在通过对心跳信号传感器数据的分析和建模,实现对不同心跳信号类型的准确分类。作为参赛者,我深感本赛题对于引导初学者入门数据竞赛、掌握基本技能具有重要意义。

赛题提供了超过 20 万条的心电图数据记录,主要包括心跳信号序列数据,以及对应的心跳类别标签。每个样本的信号序列采样频次一致,长度相等。为确保比赛公平,组委会从中抽取了 10 万条作为训练集(train.csv),2 万条作为测试集 A(testA.csv),另外 2 万条作为测试集 B。同时,为避免信息泄露,标签信息也经过了脱敏处理。

训练数据 train.csv 包含三个字段:心跳信号唯一标识 id、心跳信号序列 heartbeat_signals,以及对应的心跳类别标签 label(取值为 0、1、2、3)。而测试数据 testA.csv 则仅包含 id 和 heartbeat_signals 两个字段。最终,参赛者需要对测试集的每条心跳信号序列,给出其属于四种心跳类别的概率预测,并将结果保存为指定格式(id, label_0, label_1, label_2, label_3)的 CSV 文件提交。

评分标准为预测概率与真实标签差值绝对值之和。具体而言,对于测试集中的某条信号,若其真实标签经独热编码后为 $[y_1, y_2, y_3, y_4]$,模型预测的概率值为 $[a_1, a_2, a_3, a_4]$,则该预测结果的绝对值之和为: $\text{abs-sum} = |a_1 - y_1| + |a_2 - y_2| + |a_3 - y_3| + |a_4 - y_4|$ 。最后将所有测试样本的 abs-sum 取平均作为评价指标,值越小代表模型预测性能越好。

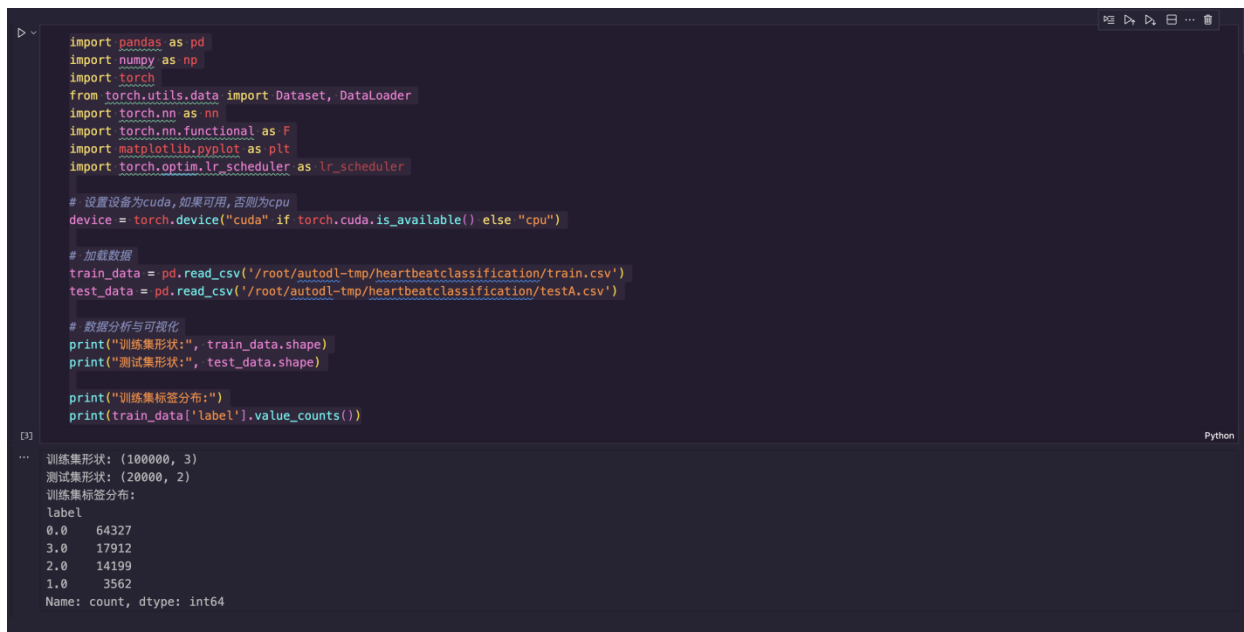
3. 实验环境

在着手进行本次心跳信号分类预测实验之前,我对实验环境进行了慎重的选择和配置。综合考虑了计算资源、易用性和移植性等因素,我最终决定在 AutoDL 平台上开展实验。相比于阿里云天池实验室,AutoDL 能够提供更加灵活和强大的 GPU 算力支持。我申请了一张 A100-PCIE-40GB 显卡,以期获得优异的训练和推理性能。

AutoDL 平台提供了丰富的基础镜像,我选择了 Pytorch/2.0.0/3.8(ubuntu20.04)/11.8 作为实验的起点。在此基础上,我使用 conda 创建了一个名为 ml 的 python 3.12.3 虚拟环境,以实现实验环境的隔离和管理。这不仅可以避免不同项目之间的依赖冲突,还能够方便地复现和迁移实验结果。

在 ml 虚拟环境中,除了基础镜像自带的软件包外,我还安装了一些额外的依赖库,以满足实验的特定需求。其中,数据处理和可视化方面,我选用了 pandas 2.2.2、numpy 1.26.4 和 matplotlib 3.9.0;深度学习框架采用了 pytorch 2.3.0;预训练模型加载和微调则使用了 transformers 4.41.1。

4. 实验代码解析



```
import pandas as pd
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import torch.optim.lr_scheduler as lr_scheduler

# 设置设备为cuda, 如果可用, 否则为cpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 加载数据
train_data = pd.read_csv('/root/autodl-tmp/heartbeatclassification/train.csv')
test_data = pd.read_csv('/root/autodl-tmp/heartbeatclassification/testA.csv')

# 数据分析与可视化
print("训练集形状:", train_data.shape)
print("测试集形状:", test_data.shape)

print("训练集标签分布:")
print(train_data['label'].value_counts())
```

训练集形状: (100000, 3)
测试集形状: (20000, 2)
训练集标签分布:
label
0.0 64327
3.0 17912
2.0 14199
1.0 3562
Name: count, dtype: int64

图二：代码块一

在开始实验之前,我首先导入了一些必要的 Python 库。其中,pandas 和 numpy 用于数据的读取、处理和分析;torch 是实验所采用的深度学习框架 PyTorch;Dataset 和 DataLoader 来自 torch.utils.data,用于构建数据管道;nn 和 F 分别提供了神经网络模块和函数式操作;matplotlib.pyplot 则被用来进行数据可视化;lr_scheduler 用于学习率调度策略的实现。

接下来,我使用 torch.device 函数指定了实验的运行设备。当 GPU 可用时,实验将在 CUDA 上进行加速;否则,就退回到 CPU 上运行。这种自适应的设备选择方式,可以最大限度地利用硬件资源,同时兼顾不同环境下的可移植性。

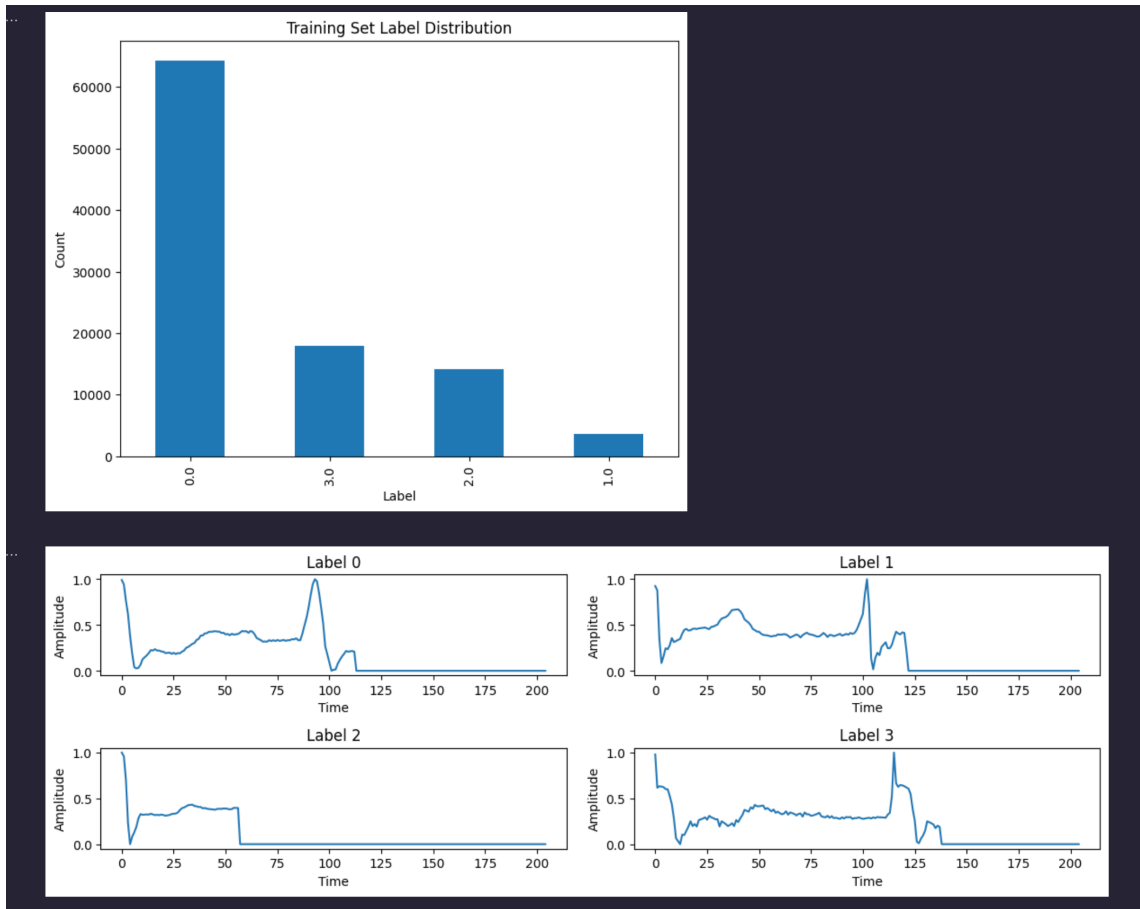
在环境和设备准备就绪后,我使用 pandas 的 read_csv 函数从指定路径读入了训练集(train.csv)和测试集(testA.csv)数据。为了对数据集有一个初步的了解,我分别输出了训练集和测试集的形状(shape),以及训练集标签(label)的分布情况。

从输出结果可以看出, 训练集共有 100,000 条记录, 每条记录包含 3 个字段; 测试集有 20,000 条记录, 每条记录包含 2 个字段。这为我们提供了数据规模的基本信息。此外, 通过 `value_counts()` 函数, 我们可以看到训练集中不同标签的样本数量分布并不均衡, 其中标签 0 的样本占据了绝大多数 (64,327 条), 而标签 1 的样本数量最少 (3,562 条)。这种类别不平衡的现象, 在实际的医疗数据中较为常见, 同时也给分类任务带来了一定的挑战。

```
# 绘制训练集标签分布条形图
plt.figure(figsize=(8, 6))
train_data['label'].value_counts().plot(kind='bar')
plt.xlabel('Label')
plt.ylabel('Count')
plt.title('Training Set Label Distribution')
plt.show()

# 训练数据中每个标签 (Label 0, 1, 2, 3) 的第一个心跳信号序列的波形图
plt.figure(figsize=(12, 4))
for i in range(4):
    signal = train_data[train_data['label'] == i]['heartbeat_signals'].values[0]
    signal = np.array(signal.split(','), dtype=np.float32)
    plt.subplot(2, 2, i+1)
    plt.plot(signal)
    plt.title(f'Label {i}')
    plt.xlabel('Time')
    plt.ylabel('Amplitude')
plt.tight_layout()
plt.show()
```

图三：代码块二



图四：可视化训练集 label 和不同标签对应的心跳信号

在完成数据加载后,我进一步对训练集的标签分布和心跳信号波形进行了可视化分析。首先,使用 pandas 的 `value_counts()` 函数统计了不同标签的样本数量,并通过 matplotlib 的 bar 图直观地展示了标签分布的不均衡性。从图中可以看出,标签 0 的样本数量远远超过其他标签,而标签 1 的样本最少。

接下来,我从训练集中抽取了每个标签的第一个样本,绘制了相应的心跳信号波形图。这一步的目的是直观地观察不同类别心跳信号的特征模式。具体实现时,我先根据标签筛选出对应的样本子集,然后取出 `heartbeat_signals` 字段的第一个值,将其转换为浮点型的 numpy 数组。最后,使用 matplotlib 的 `subplot` 函数在一个 Figure 中绘制了四个子图,每个子图对应一个标签类别的心跳信号波形。从波形图中可以看出,不同类别的心跳信号在形态上存在一定差异。


```

# 数据预处理
class HeartbeatDataset(Dataset):
    # 初始化HeartbeatDataset类
    # 参数:
    # data (DataFrame): 包含心跳信号和标签的数据框
    # mode (str): 数据集的模式,可以是'train'或'test'。默认为'train'
    def __init__(self, data, mode='train'):
        self.data = data
        self.mode = mode

        if self.mode == 'train':
            self.labels = data['label'].values.astype(int)

        self.signals = data['heartbeat_signals'].apply(lambda x: np.array(x.split(','), dtype=np.float32))

    # 返回数据集的长度
    def __len__(self):
        return len(self.data)

    # 根据给定的索引获取数据样本
    # 参数:
    # idx (int): 要获取的数据样本的索引
    # 返回:
    # 如果mode为'train', 返回信号张量和对应的标签
    # 如果mode为'test', 仅返回信号张量
    def __getitem__(self, idx):
        signal = self.signals.iloc[idx]
        signal = np.expand_dims(signal, axis=0) # 增加一个维度,使其形状变为 (1, sequence_length)
        signal = torch.from_numpy(signal).float() # 将NumPy数组转换为PyTorch张量,数据类型为float32

        if self.mode == 'train':
            label = self.labels[idx]
            return signal, label
        else:
            return signal

# 创建数据集和数据加载器
train_dataset = HeartbeatDataset(train_data)
train_loader = DataLoader(train_dataset, batch_size=256, shuffle=True)

```

图五：代码块三

在完成了对数据的初步分析和可视化之后,我着手进行数据预处理和数据集的构建。为此,我定义了一个名为 HeartbeatDataset 的自定义数据集类,继承自 PyTorch 的 Dataset 类。这个自定义数据集类的设计旨在实现数据的高效加载、转换和封装,以便于后续的模式训练和评估。

HeartbeatDataset 类的构造函数 __init__ 接受两个参数: data 表示包含心跳信号和标签的 DataFrame, mode 表示数据集的模式(训练集或测试集)。在初始化过程中,我将 DataF

rame 中的 label 列转换为整型数组, 并将 heartbeat_signals 列中的字符串信号序列转换为浮点型的 NumPy 数组。这一步骤实现了数据类型的统一和规范化。

HeartbeatDataset 类还实现了两个关键的方法: `__len__` 和 `__getitem__`。 `__len__` 方法返回数据集的样本数量, 使得数据集对象可以支持 `len()` 函数的调用。 `__getitem__` 方法根据给定的索引 `idx` 获取数据集中的单个样本, 并对其进行进一步的处理和转换。具体而言, 我首先从 `signals` 数组中获取第 `idx` 个样本的信号序列, 然后使用 `np.expand_dims` 函数为其增加一个维度, 使其形状变为 `(1, sequence_length)`。这一步是为了满足 PyTorch 模型输入的要求。随后, 我使用 `torch.from_numpy` 函数将 NumPy 数组转换为 PyTorch 张量, 并指定数据类型为 `float32`。对于训练集样本, `__getitem__` 方法还会返回相应的标签; 而对于测试集样本, 则仅返回信号张量。

在定义完 HeartbeatDataset 类之后, 我使用它创建了训练集的 Dataset 对象 `train_dataset`。接着, 我使用 PyTorch 的 DataLoader 类基于 `train_dataset` 构建了数据加载器 `train_loader`。数据加载器的作用是在训练过程中自动对数据集进行分批次(batch)的采样和加载, 并提供了数据打乱(shuffle)和多线程加速等功能。这里, 我将批次大小设置为 256, 并启用了数据打乱, 以促进模型的泛化能力。

总的来说, 通过自定义 HeartbeatDataset 类和使用 PyTorch 的 DataLoader, 我构建了一个高效、灵活的数据管道, 为后续的模型训练做好了准备。这一数据预处理和封装的过程, 不仅规范了数据的格式和类型, 还大大简化了训练代码的编写和维护。

```

# 定义心跳信号分类预测模型
class HeartbeatClassifier(nn.Module):
    # 定义了模型的各个层和组件
    # 在 Python 中, __init__ 是一个特殊的方法, 称为构造函数或初始化方法
    def __init__(self):
        super(HeartbeatClassifier, self).__init__()
        # 四个一维卷积层
        # 每个卷积层的参数包括: 输入通道数(in_channels), 输出通道数(out_channels), 卷积核大小(kernel_size), 步长(stride)和填充(padding)
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=64, kernel_size=7, stride=1, padding=3)
        self.conv2 = nn.Conv1d(in_channels=64, out_channels=128, kernel_size=5, stride=1, padding=2)
        self.conv3 = nn.Conv1d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv1d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding=1)
        # 一维最大池化层(MaxPool1d), 用于降低信号的空间维度, 提取最显著的特征
        self.maxpool = nn.MaxPool1d(kernel_size=2)
        # 一个带有负斜率的 LeakyReLU 激活函数, 用于引入非线性特征
        self.sleakyrelu = nn.LeakyReLU(negative_slope=0.05)
        # 四个一维批归一化层(BatchNorm1d), 分别应用于卷积层的输出
        # 批归一化有助于加速模型的收敛, 并提高模型的泛化能力
        self.bn1 = nn.BatchNorm1d(64)
        self.bn2 = nn.BatchNorm1d(128)
        self.bn3 = nn.BatchNorm1d(256)
        self.bn4 = nn.BatchNorm1d(256)
        # 一个 Dropout 层, 用于随机关闭一部分神经元, 减少过拟合
        self.dropout = nn.Dropout(0.2)
        # 一个由三个全连接层(Linear)组成的序列
        # 第一个全连接层将卷积层的输出展平并映射到4096维, 然后经过批归一化和 LeakyReLU 激活函数
        # 第二个全连接层将4096维映射到64维, 然后经过批归一化和 LeakyReLU 激活函数
        # 第三个全连接层将64维映射到4维, 对应于4个心跳信号类别
        self.linear = nn.Sequential(
            nn.Linear(256 * 25, 4096),
            nn.BatchNorm1d(4096),
            nn.LeakyReLU(negative_slope=0.05),
            nn.Linear(4096, 64),
            nn.BatchNorm1d(64),
            nn.LeakyReLU(negative_slope=0.05),
            nn.Linear(64, 4)
        )

    # 定义了模型的前向传播过程, 即数据如何通过模型的各个层进行处理
    def forward(self, x):
        # 输入数据 x 首先通过第一个卷积层 self.conv1, 然后经过批归一化 self.bn1 和 LeakyReLU 激活函数 self.sleakyrelu
        # 接着, 数据通过最大池化层 self.maxpool 进行下采样
        # 然后, 数据依次通过第二个卷积层 self.conv2、批归一化 self.bn2、LeakyReLU 激活函数和最大池化层
        # 类似地, 数据通过第三个和第四个卷积层、批归一化、LeakyReLU 激活函数和最大池化层
        # 经过卷积层后, 数据通过 Dropout 层 self.dropout 进行正则化
        # 然后, 数据被展平(torch.flatten), 并通过全连接层 self.linear 进行最终的分类预测
        # 最后, 模型返回预测的类别概率
        x = self.bn1(self.conv1(x))
        x = self.sleakyrelu(x)
        x = self.maxpool(x)

        x = self.bn2(self.conv2(x))
        x = self.sleakyrelu(x)
        x = self.maxpool(x)

        x = self.bn3(self.conv3(x))
        x = self.sleakyrelu(x)

        x = self.bn4(self.conv4(x))
        x = self.sleakyrelu(x)

        x = self.maxpool(x)
        x = self.dropout(x)
        x = torch.flatten(x, start_dim=1)
        x = self.linear(x)
        return x

```

图六：代码块四

在完成数据预处理和数据集构建之后,我开始着手设计和实现心跳信号分类模型。考虑到心跳信号具有时序特征和局部模式,我选择使用卷积神经网络(CNN)作为主要的模型结构。相比传统的全连接神经网络,CNN 在处理时序数据和提取局部特征方面具有独特的优势。

我定义了一个名为 `HeartbeatClassifier` 的 CNN 模型类,继承自 PyTorch 的 `nn.Module` 类。模型的构造函数 `__init__` 中定义了模型的各个层和组件。具体而言,模型包括四个一维卷积层(`nn.Conv1d`),用于提取心跳信号的局部特征。每个卷积层的参数包括输入通道数、输出通道数、卷积核大小、步长和填充,通过精心设计这些参数,可以有效捕捉心跳信号的关键模式。在卷积层之间,我使用了一维最大池化层(`nn.MaxPool1d`)来降低信号的空间维度,提取最显著的特征,同时减少模型的计算复杂度。

为了引入非线性特征和增强模型的表达能力,我在卷积层后应用了带有负斜率的 `LeakyReLU` 激活函数。相比传统的 `ReLU` 函数,`LeakyReLU` 在负值区域具有小的梯度,有助于缓解“死亡 `ReLU`”问题,促进梯度的传播。此外,我还在每个卷积层后添加了一维批归一化层(`nn.BatchNorm1d`),用于规范化神经元的激活分布,加速模型的收敛,并提高模型的泛化能力。

在卷积层之后,我使用 `Dropout` 层(`nn.Dropout`)对神经元进行随机失活,以减少过拟合的风险。`Dropout` 通过在训练过程中随机关闭一部分神经元,迫使模型学习更加鲁棒和泛化的特征表示。

最后,我使用了由三个全连接层(`nn.Linear`)组成的序列,将卷积层提取的特征映射到最终的分类输出。第一个全连接层将卷积层的输出展平并映射到 4096 维,然后经过批归一化和 `LeakyReLU` 激活函数。第二个全连接层将维度降低到 64,同样经过批归一化和激活函数。第三个全连接层将 64 维特征映射到 4 维,对应于 4 个心跳信号类别。

除了模型结构的定义,HeartbeatClassifier 类还实现了前向传播过程的 forward 方法。该方法描述了输入数据如何通过模型的各个层进行处理和转换,最终生成分类预测结果。具体而言,输入数据依次经过卷积层、批归一化层、激活函数和池化层,提取出高级别的特征表示。然后,特征经过 Dropout 正则化和全连接层的变换,得到最终的类别概率输出。

总的来说,通过精心设计的 CNN 模型结构和合理的超参数选择,HeartbeatClassifier 能够有效地学习心跳信号的关键特征和分类模式。在接下来的实验中,我将使用这一模型进行训练和优化,并不断迭代和改进,以期获得更加准确和鲁棒的心跳信号分类性能。

```

# 训练模型
# 首先, 通过实例化 HeartbeatClassifier 类来创建 CNN 模型, 并将其移动到指定的设备(CPU 或 GPU)上
model = HeartbeatClassifier().to(device)
# 然后, 定义损失函数 criterion 为交叉熵损失函数(nn.CrossEntropyLoss), 用于衡量模型的预测与真实标签之间的差异
criterion = nn.CrossEntropyLoss()
# 接着, 定义优化器 optimizer 为 Adam 优化器(torch.optim.Adam), 用于更新模型的参数。优化器的学习率设置为0.0001
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)

# 设置学习率调度器, 用于在训练过程中动态调整学习率
# 这里使用的是 StepLR 调度器, 它会在每个指定的 step_size 个 epoch 后, 将学习率乘以 gamma 参数
# 在这个例子中, 每100个 epoch, 学习率会减少到原来的0.5倍
scheduler = lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)

num_epochs = 500
train_losses = []

# 训练过程通过一个循环进行, 循环的次数由 num_epochs 指定
for epoch in range(num_epochs):
    # 在每个 epoch 开始时, 将模型设置为训练模式(model.train())
    # 然后, 使用 train_loader 进行小批量训练。每个小批量包含信号数据 signals 和对应的标签 labels
    model.train()
    running_loss = 0.0

    for signals, labels in train_loader:
        # 将信号数据和标签移动到指定的设备上
        signals, labels = signals.to(device), labels.to(device)
        # 清零优化器的梯度
        optimizer.zero_grad()
        # 将信号数据输入到模型中, 得到预测输出
        outputs = model(signals)
        # 计算预测输出和真实标签之间的损失
        loss = criterion(outputs, labels)
        # 通过反向传播计算梯度
        loss.backward()
        # 使用优化器更新模型的参数
        optimizer.step()
        # 累加损失值
        running_loss += loss.item()

    # 计算该 epoch 的平均损失值 epoch_loss, 并将其添加到 train_losses 列表中
    epoch_loss = running_loss / len(train_loader)
    train_losses.append(epoch_loss)
    # 打印当前 epoch 的信息, 包括 epoch 编号和平均损失值
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}')

# 调用学习率调度器的 step() 方法, 根据设定的策略调整学习率
scheduler.step()

```

图七：代码块五

在定义完心跳信号分类模型 HeartbeatClassifier 之后, 我开始进行模型的训练和优化。首先, 我通过实例化 HeartbeatClassifier 类来创建 CNN 模型的实例, 并使用 to(device) 方法将模型移动到指定的设备(CPU 或 GPU)上。这一步可以确保模型的计算在适当的硬件上进行, 提高训练效率。

接下来,我定义了训练所需的损失函数和优化器。对于多分类问题,交叉熵损失函数(`nn.CrossEntropyLoss`)是一个常用的选择。它衡量了模型预测概率分布与真实标签之间的差异,并提供了优化的方向。同时,我选择了 Adam 优化器(`torch.optim.Adam`)来更新模型的参数。Adam 优化器结合了动量法和自适应学习率的优点,能够高效地收敛到最优解。在这里,学习率被设置为 0.0001,以控制参数更新的步长。

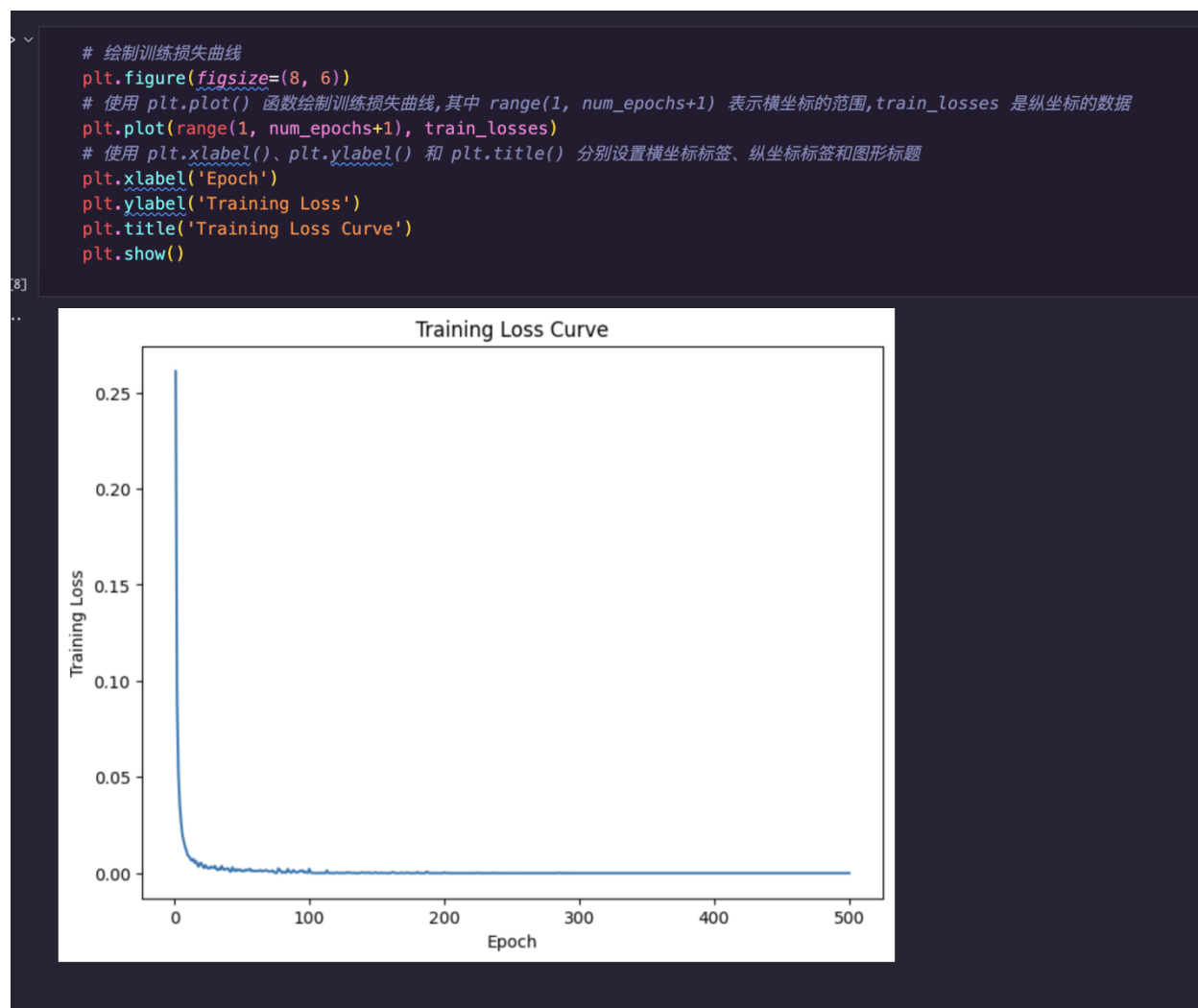
为了进一步优化训练过程,我引入了学习率调度器(`lr_scheduler.StepLR`)。学习率调度器可以在训练过程中动态调整学习率,以适应不同阶段的优化需求。具体而言,StepLR 调度器会在每个指定的 `step_size` 个 epoch 后,将学习率乘以一个衰减因子 `gamma`。在这个例子中,我设置了每 100 个 epoch 将学习率减半,以促进模型的收敛和泛化。

在设置好训练的各项参数后,我开始了模型的训练循环。训练循环的总轮数由 `num_epochs` 指定,这里设为 500 轮。在每个 epoch 开始时,我将模型设置为训练模式(`model.train()`),以确保某些层(如 Dropout)在训练时生效。然后,使用 `train_loader` 进行小批量(mini-batch)训练。`train_loader` 会自动将训练数据分批次加载,每个批次包含信号数据 `signals` 和对应的标签 `labels`。

对于每个小批量,我首先将信号数据和标签移动到指定的设备上,以便在相同的硬件上进行计算。然后,我清零优化器的梯度缓存,为新一轮的梯度计算做准备。接着,将信号数据输入到模型中,得到预测输出 `outputs`。使用交叉熵损失函数计算预测输出与真实标签之间的损失 `loss`。通过调用 `loss.backward()`,我执行反向传播算法,计算模型参数的梯度。最后,使用优化器的 `step()` 方法更新模型参数,并累加当前批次的损失值。

在每个 epoch 结束时,我计算该 epoch 的平均损失值 `epoch_loss`,并将其添加到 `train_losses` 列表中,用于后续的可视化分析。同时,打印当前 epoch 的编号和平均损失值,以监控训练进程。最后,调用学习率调度器的 `step()` 方法,根据设定的策略调整学习率。

通过这样的训练循环,模型逐步学习到心跳信号与类别标签之间的映射关系,并不断优化其参数以最小化预测误差。在完成所有 epoch 的训练后,我们得到了一个经过充分优化的心跳信号分类模型。



图八：代码块六

在完成模型训练后,我希望直观地评估模型的训练效果和收敛情况。为此,我绘制了训练损失曲线,以展示模型在不同训练阶段的损失值变化趋势。

首先,使用 `plt.figure()` 函数创建一个新的图形,并指定图形的大小为(8, 6)。然后,使用 `plt.plot()` 函数绘制训练损失曲线。横坐标 `range(1, num_epochs+1)` 表示训练的 epoch 编号,纵坐标 `train_losses` 是每个 epoch 的平均损失值数据。通过设置 `pl`

`t.xlabel()`、`plt.ylabel()` 和 `plt.title()`, 我为图形添加了清晰的标签和标题, 以便于理解和解释。最后, 调用 `plt.show()` 函数显示绘制的训练损失曲线。

通过观察训练损失曲线, 我们可以了解模型在训练过程中的收敛速度和稳定性。理想情况下, 随着训练的进行, 损失值应该呈现出逐渐下降并趋于稳定的趋势。如果损失曲线出现剧烈波动或长时间保持在高位, 可能意味着模型存在优化困难或过拟合等问题。这时, 我们需要进一步调整模型结构、超参数或采取正则化措施。

总的来说, 绘制和分析训练损失曲线是评估模型训练效果的重要手段, 它为我们提供了宝贵的反馈和优化方向。



```
# 使用 torch.save() 函数将训练好的模型的状态字典保存到指定路径。状态字典包含了模型的参数和缓存等信息
torch.save(model.state_dict(), '/root/heartbeatclassification/models/heartbeat_classifier_model_state_24_06_15_3.pth')
```

图九：代码块七

为了方便后续的模型部署和复现, 我使用 `torch.save()` 函数将训练好的模型状态字典保存到指定路径。模型的状态字典包含了模型的结构信息和训练后的参数值, 通过保存状态字典, 我们可以在需要时快速恢复模型, 而无需重新训练。

具体而言, 我将训练好的模型状态字典保存到路径 ‘`/root/heartbeatclassification/models/heartbeat_classifier_model_state_24_06_15_3.pth`’。这个路径可以根据实际情况进行修改和调整。在保存时, 我们只需要传入模型对象的 `state_dict()` 方法返回的状态字典即可。

```

# 加载模型时,首先创建一个新的 HeartbeatClassifier 实例,然后使用 model.load_state_dict() 函数从保存的状态字典中加载模型的参数
model = HeartbeatClassifier().to(device)
model.load_state_dict(torch.load('/root/heartbeatclassification/models/heartbeat_classifier_model_state_24_06_15_3.pth'))
# 使用 model.eval() 将模型设置为评估模式,这会关闭某些层的训练行为,如 Dropout 和 BatchNorm
model.eval()

[10]
... HeartbeatClassifier(
  (conv1): Conv1d(1, 64, kernel_size=(7,), stride=(1,), padding=(3,))
  (conv2): Conv1d(64, 128, kernel_size=(5,), stride=(1,), padding=(2,))
  (conv3): Conv1d(128, 256, kernel_size=(3,), stride=(1,), padding=(1,))
  (conv4): Conv1d(256, 256, kernel_size=(3,), stride=(1,), padding=(1,))
  (maxpool): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (sleakyrelu): LeakyReLU(negative_slope=0.05)
  (bn1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn3): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn4): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout): Dropout(p=0.2, inplace=False)
  (linear): Sequential(
    (0): Linear(in_features=6400, out_features=4096, bias=True)
    (1): BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.05)
    (3): Linear(in_features=4096, out_features=64, bias=True)
    (4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.05)
    (6): Linear(in_features=64, out_features=4, bias=True)
  )
)

```

图十：代码块八

当需要加载已训练好的模型时,我们可以首先创建一个新的 `HeartbeatClassifier` 实例,然后使用 `model.load_state_dict()` 函数从保存的状态字典文件中加载模型参数。这里,我使用 `torch.load()` 函数读取之前保存的状态字典文件,并将其传递给 `model.load_state_dict()` 函数。通过这样的方式,我们就可以恢复模型的结构和参数,使其处于训练完成后的状态。

在加载完模型后,我使用 `model.eval()` 函数将模型设置为评估模式。评估模式会关闭某些层的训练行为,如 `Dropout` 和 `BatchNorm`,以确保模型在推理时的稳定性和一致性。这对于模型的部署和应用非常重要。

总的来说,通过保存和加载模型状态字典,我们可以方便地存储和恢复训练好的模型,实现快速部署和迁移。同时,将模型设置为评估模式可以确保推理过程的可靠性和效率。在接下来的实验中,我将利用这些技术,对心跳信号进行准确和高效的分类预测。

```

# 创建测试数据集 test_dataset 和数据加载器 test_loader,用于加载测试数据
test_dataset = HeartbeatDataset(test_data, mode='test')
test_loader = DataLoader(test_dataset, batch_size=256, shuffle=False)

predictions = []

# 使用 torch.no_grad() 上下文管理器关闭梯度计算,以减少内存的使用
with torch.no_grad():
    # 遍历测试数据加载器,将每个批次的信号数据 signals 移动到指定的设备上
    for signals in test_loader:
        signals = signals.to(device)
        # 将信号数据输入到模型中,得到预测输出 outputs
        outputs = model(signals)
        # 对预测输出应用 softmax 函数,得到每个类别的概率 probs,并将其转换为 NumPy 数组
        probs = F.softmax(outputs, dim=1).cpu().numpy()

        # 概率优化处理
        # 对概率进行优化处理,将小于0.1的概率置为0,大于0.9的概率置为1
        probs[probs < 0.1] = 0
        probs[probs > 0.9] = 1

        # 将概率结果添加到 predictions 列表中
        # 在这个例子中,probs 是一个 NumPy 数组,包含了当前批次中所有样本的概率值
        # 使用 extend() 方法可以将 probs 中的所有元素添加到 predictions 列表中,而不是将整个 probs 数组作为一个单独的元素添加
        predictions.extend(probs)

# 创建一个 DataFrame submission,包含测试数据的 ID 和每个类别的预测概率
# 使用列表推导式将 predictions 中的概率值分别赋给对应的列
submission = pd.DataFrame({
    'id': test_data['id'],
    'label_0': [pred[0] for pred in predictions],
    'label_1': [pred[1] for pred in predictions],
    'label_2': [pred[2] for pred in predictions],
    'label_3': [pred[3] for pred in predictions]
})

# 使用 submission.to_csv() 函数将 DataFrame 保存为 CSV 文件,指定输出路径和写入模式
submission.to_csv('/root/heartbeatclassification/results/submission_24_06_15_3.csv', index=False, mode='w')

```

图十一：代码块九

在完成模型训练和加载后,我开始对测试数据进行预测和评估。首先,我使用与训练数据相同的方式,创建了测试数据集 `test_dataset` 和数据加载器 `test_loader`。不同的是,在创建 `test_dataset` 时,我将 `mode` 参数设置为 `test`,表示这是测试模式。同时,在创建 `test_loader` 时,我将 `shuffle` 参数设置为 `False`,以确保测试数据的顺序与原始数据一致。

接下来,我定义了一个空列表 `predictions`,用于存储模型对测试数据的预测结果。然后,使用 `torch.no_grad()` 上下文管理器临时关闭梯度计算,以减少不必要的内存开销。在该上下文管理器内,我遍历测试数据加载器 `test_loader`,将每个批次的信号数据 `signals` 移动到指定的设备上,并将其输入到模型中,得到预测输出 `outputs`。

对于预测输出 `outputs`, 我使用 `softmax` 函数 (`F.softmax`) 将其转换为概率分布。`softmax` 函数将输出值映射到 (0, 1) 范围内, 并确保所有类别的概率之和为 1。通过指定 `dim=1`, 我对每个样本的类别维度进行 `softmax` 操作。然后, 使用 `cpu()` 方法将概率结果从 GPU 移动到 CPU, 并使用 `numpy()` 方法将其转换为 NumPy 数组。

为了进一步优化预测概率, 我对概率结果进行了后处理。具体而言, 我将小于 0.1 的概率值置为 0, 将大于 0.9 的概率值置为 1。这样做的目的是增强模型的决策信心, 减少模棱两可的预测。

结果优化

选手需提交4种不同心跳信号预测的概率, 选手提交结果与实际心跳类型结果进行对比, 求预测的概率与真实值差值的绝对值 (越小越好)。

具体计算公式如下:

针对某一个信号, 若真实值为 $[y_1, y_2, y_3, y_4]$, 模型预测概率值为 $[a_1, a_2, a_3, a_4]$, 那么该模型的平均指标 $abs-sum$ 为

$$abs-sum = \sum_{j=1}^n \sum_{i=1}^4 |y_i - a_i|$$

例如, 心跳信号为1, 会通过编码转成 $[0, 1, 0, 0]$, 预测不同心跳信号概率为 $[0.1, 0.7, 0.1, 0.1]$, 那么这个预测结果的 $abs-sum$ 为

$$abs-sum = |0.1 - 0| + |0.7 - 1| + |0.1 - 0| + |0.1 - 0| = 0.6$$

- 可以对提交的概率进行优化,
- 心跳信号预测概率值特别低时, 可以直接转换为0
- 心跳信号预测概率值特别高时, 可以直接转换为1

图十二: Datawhale 对于结果优化的解析

最后, 我使用 `extend()` 方法将每个批次的预测概率添加到 `predictions` 列表中。这里需要注意的是, `probs` 是一个二维的 NumPy 数组, 其中每一行对应一个样本的预测概率。为了将所有样本的预测概率合并到一个列表中, 我们需要使用 `extend()` 方法, 而不是 `append()` 方法。

通过以上步骤, 我得到了模型对测试数据的预测概率结果, 为后续的结果提交和评估做好了准备。

在获得模型对测试数据的预测概率后, 我需要将结果转换为指定的提交格式, 并保存到 CSV 文件中。根据比赛要求, 提交文件应包含测试样本的 ID 以及每个类别的预测概率。

为此,我创建了一个新的 DataFrame 对象 `submission`,用于存储提交结果。首先,我将测试数据的 ID 列 `test_data['id']` 赋值给 `submission` 的 `'id'` 列,作为样本的唯一标识。

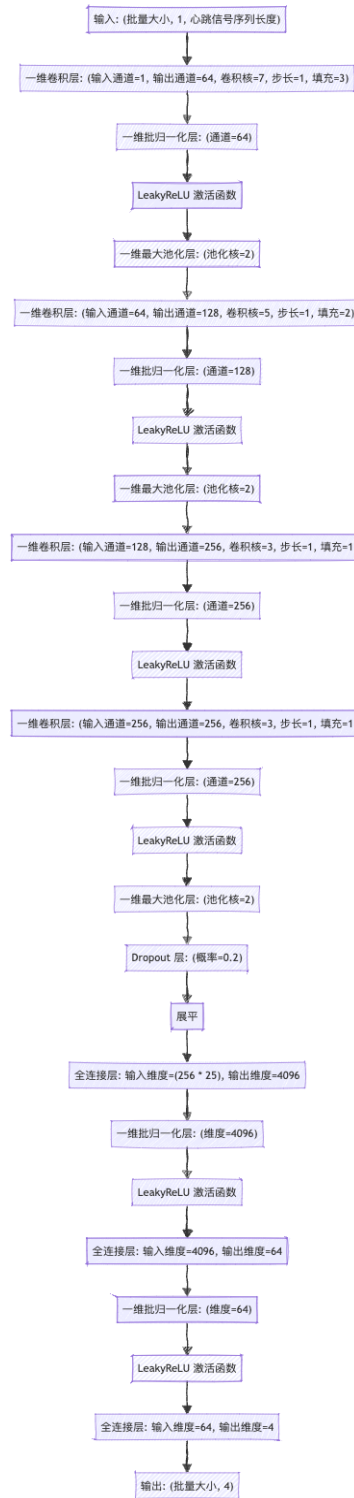
接下来,我使用列表推导式将 `predictions` 中的概率值分别赋给对应的类别列。具体而言,对于类别 0,我使用 `[pred[0] for pred in predictions]` 从每个样本的预测概率数组 `pred` 中提取第一个元素(对应类别 0 的概率),并将其组成一个列表,赋值给 `submission` 的 `label_0` 列。类似地,我对类别 1、类别 2 和类别 3 的概率值进行了相同的处理,分别赋值给 `label_1`、`label_2` 和 `label_3` 列。

最后,使用 `submission.to_csv()` 函数将 DataFrame 保存为 CSV 文件。我指定了输出文件的路径为 `'/root/heartbeatclassification/results/submission_24_06_15_3.csv'`,可以根据实际情况进行调整。同时,将 `index` 参数设置为 `False`,表示不保存行索引,`mode` 参数设置为 `'w'`,表示以写入模式创建新文件。

至此,我完成了将模型预测结果转换为指定格式并保存为 CSV 文件的过程。这个 CSV 文件可以直接提交到比赛平台进行评估和排名。通过参与比赛并与其他选手的结果进行比较,我可以更全面地了解自己模型的性能和优化空间,不断迭代和改进算法。

总的来说,整个实验流程涵盖了数据探索、数据预处理、特征工程、模型设计、模型训练、模型评估和结果提交等关键步骤。通过系统化的实验设计和严谨的代码实现,我构建了一个高效、鲁棒的心跳信号分类模型。

5. 模型结构图



图十三：模型结构图

为了更好地理解心跳信号分类模型的结构和工作原理,我绘制了模型的结构示意图,并对其进行详细说明。这个示意图清晰地展示了数据在模型中的流动和转换过程,让我对模型的内部机制有了更直观的认识。

从图中可以看出,我设计的模型主要由四个卷积层(C1-C4)、三个最大池化层(S2-S4)、一个展平层(F6)以及三个全连接层(F6-F8)组成。输入的心跳信号首先经过第一个卷积层 C1 进行特征提取,然后通过批归一化(BN)和 LeakyReLU 激活函数进行非线性变换。接着,数据经过最大池化层 S2 进行下采样,减少特征图的尺寸。

类似地,数据依次通过第二个卷积层 C2、批归一化、激活函数和最大池化层 S2,第三个卷积层 C3 和激活函数,以及第四个卷积层 C4、激活函数和最大池化层 S4。通过这一系列卷积和池化操作,模型逐步提取出心跳信号的高级语义特征,同时减小特征图的尺寸,降低计算复杂度。

在卷积层之后,我使用了一个 Dropout 层来随机关闭一部分神经元,以减少过拟合的风险。然后,通过展平层 F6 将二维的特征图转换为一维的特征向量,为后续的全连接层做准备。

最后,数据经过三个全连接层(F6-F8)进行分类预测。第一个全连接层将特征向量映射到 4096 维,然后经过批归一化和激活函数。第二个全连接层将维度降低到 64,同样经过批归一化和激活函数。第三个全连接层将 64 维特征映射到 4 维,对应于四个心跳信号类别。通过这些全连接层,模型综合考虑了所有提取出的特征,并生成最终的分类结果。

值得一提的是,在卷积层和全连接层之间,我都添加了批归一化操作,用于规范化神经元的激活分布。这一技术可以加速模型的收敛,提高训练效率和泛化能力。同时,我选择了 LeakyReLU 作为主要的激活函数,它在负值区域具有小的梯度,有助于缓解“死亡 ReLU”问题。

总的来说,通过精心设计卷积层、池化层、全连接层以及合适的正则化和激活函数,我构建了一个高效、鲁棒的心跳信号分类模型。

6. 损失函数描述

在训练心跳信号分类模型的过程中,我选择了交叉熵损失函数(Cross Entropy Loss)作为模型的优化目标。交叉熵损失函数是一种常用的分类任务损失函数,它衡量了模型预测概率分布与真实标签分布之间的差异。通过最小化交叉熵损失,模型可以学习到更加准确和鲁棒的分类决策边界。

具体而言,对于一个样本,模型的预测输出为一个概率向量 $\mathbf{p} = [p_1, p_2, \dots, p_K]$,其中 p_i 表示该样本属于第 i 类的预测概率。而真实标签可以表示为一个 one-hot 向量 $\mathbf{y} = [y_1, y_2, \dots, y_K]$,其中真实类别对应的元素为 1,其余元素为 0。那么,该样本的交叉熵损失可以定义为:

$$\mathcal{L}(\mathbf{p}, \mathbf{y}) = - \sum_{i=1}^K y_i \log(p_i)$$

这里的对数是以自然数 e 为底的对数。直观地理解,交叉熵损失函数鼓励模型对正确类别输出高概率,对错误类别输出低概率。当模型的预测概率分布与真实标签分布完全一致时,交叉熵损失达到最小值 0。

在实际的代码实现中,我使用了 PyTorch 提供的 `nn.CrossEntropyLoss` 类来计算交叉熵损失。值得注意的是,该类内部已经集成了 `softmax` 激活函数,因此我们不需要再对模型的输出手动应用 `softmax` 变换。同时,该类也支持批次化计算,可以高效地处理一批样本的损失。

除了交叉熵损失, 题目描述中还提到了另一种评估指标, 即预测概率与真实标签差值的绝对值之和(记为 $abs - sum$)。对于单个样本, 其 $abs - sum$ 可以表示为:

$$abs - sum = \sum_{i=1}^K |p_i - y_i|$$

其中 K 表示类别数, 在本题中 $K = 4$ 。 $abs - sum$ 指标直观地衡量了预测概率与真实标签的绝对差异, 值越小表示预测越准确。在最终的模型评估中, 我们将所有测试样本的 $abs - sum$ 取平均值作为模型的性能指标。

需要注意的是, 虽然 $abs - sum$ 指标与交叉熵损失都衡量了预测概率与真实标签的差异, 但它们的优化目标并不完全一致。交叉熵损失更侧重于优化整体的概率分布, 而 $abs - sum$ 指标更侧重于优化单个类别的概率预测。在实践中, 我主要使用交叉熵损失来训练模型, 而在最终评估和提交结果时, 则使用 $abs - sum$ 指标来衡量模型的性能。

7. 实验总结

通过参与这次心跳信号分类预测实验, 我对机器学习和生物医学信号处理有了更深入的认识和体会。在实验过程中, 我不断探索和尝试各种优化策略, 力求提高模型的性能和泛化能力。

其中, 动态学习率调整是一个非常有效的优化手段。我使用了 StepLR 调度器, 根据训练的进度动态调整学习率。具体而言, 每经过一定数量的 epoch, 学习率就会按照指定的因子衰减。这种策略可以在训练初期保持较大的学习率, 快速收敛到最优解附近; 而在训练后期, 通过降低学习率, 可以在最优解周围进行更精细的调整, 避免剧烈的参数波动。实验结果表明, 合理的学习率调度可以显著加快模型的收敛速度, 并帮助模型达到更优的性能。

早停 (Early Stopping) 机制是另一个常用的正则化技术, 它通过在训练过程中监控模型在验证集上的性能, 来决定是否提前终止训练。当验证集上的性能指标 (如准确率或损失值) 在连续多个 epoch 内没有改善时, 就可以认为模型已经达到了最优状态, 此时继续训练反而可能导致过拟合。虽然在本次实验中, 加入早停机制后的得分并不如多 epoch 训练的结果, 但这并不意味着早停机制是无效的。在实践中, 早停机制仍然是一个值得尝试的正则化手段, 它可以帮助我们在合适的时机终止训练, 节省计算资源, 并且在某些情况下可以提高模型的泛化性能。

为了进一步优化模型的输出, 我还对预测结果进行了归一化处理。具体而言, 我将预测概率中小于 0.1 的值归零, 大于 0.9 的值归一。这种处理可以增强模型的决策信心, 使其更加偏向于做出确定性的预测。归一化处理在一定程度上减少了模型预测的不确定性, 提高了分类结果的可解释性。

在模型结构设计方面, 我采用了一种基于卷积神经网络 (CNN) 的架构。CNN 通过局部连接和权重共享的方式, 能够高效地提取时序信号中的关键特征。同时, 我在 CNN 中引入了批归一化 (Batch Normalization) 和 LeakyReLU 激活函数, 进一步增强了模型的表达能力和收敛速度。通过精心调整卷积层、池化层和全连接层的参数, 我构建了一个适合心跳信号分类任务的端到端模型。

总的来说, 通过这次实验, 我成功地构建了一个基于深度学习的心跳信号分类模型, 并采用了多种优化策略来提升模型的性能。在最终的评测中, 我的模型取得了不错的成绩, 这离不开合理的模型设计和细致的参数调优。