华南理工大学

South China University of Technology

《统计学习与数据科学》课 程 论 文

(2023-2024 学年暑假学期)

基于卷积神经网络的 ECG 分类

学生姓名： 陶俊

提交日期： 2024 年 8 月 18 日　　学生签名： 陶俊

学　　号	202066138066	座位编号	
学　　院	数学学院	专业班级	数学与应用数学（统计学）
课程名称	统计学习与数据科学课程设计	任课教师	夏立
教师评语：			
本论文成绩评定： _____分			

华南理工大学

South China University of Technology

基于卷积神经网络的 ECG 分类

陶俊

摘要： 在数据信息急剧膨胀的今天，数据挖掘与数据分析的能力已经不再是一个专业人员所具备的特有技能，而是应该成为现代人所掌握的基本技能。本课程设计使用 PTB-XL 数据集完成了一项多标签心电图分类任务，开发了一个机器学习模型，该模型可以预测心电图记录是否属于这些类别中的一个或多个。通过 PTB-XL 数据集的训练，可得到其在测试集上的表现良好，在大多数类别上都取得了较高的 AUROC 和 F1-score。

关键词：ECG，CNN，PTB-XL 数据集

一、 导言

心电图 （ECG） 记录心脏的电活动，通过测量心脏不同位置产生的电信号来反映其功能状态。心电图在临床医学中具有重要意义，对于诊断心脏病、评估心脏功能和监测心脏健康至关重要。随着机器学习和统计学习技术的进步，越来越多的人探索将这些发展应用于心电图的分类和分析，以实现更准确、更及时的心脏病诊断和监测。分析心电图的方法大致可分为传统的机器学习方法和深度学习方法。近年来，深度学习方法在该领域的普及程度激增。卷积神经网络 （CNN） 和递归神经网络 （RNN） 等深度学习技术在自动从 ECG 信号中提取复杂特征方面显示出有希望的结果，从而提高了 ECG 分类任务的准确性。

在本课程项目中，作者使用 PTB-XL 数据集完成一项多标签心电图分类任务，该数据集包含 {“NORM”、“CD”、“HYP”、“MI”、“STTC”} 5 个诊断等级的心电图数据。该任务是一个多标签分类任务，这意味着一个心电记录可以对应多个标签。作者开发了一个机器学习模型，该模型可以预测心电图记录是否属于这些类别中的一个或多个。本文训练的目标即如图 1 所示：

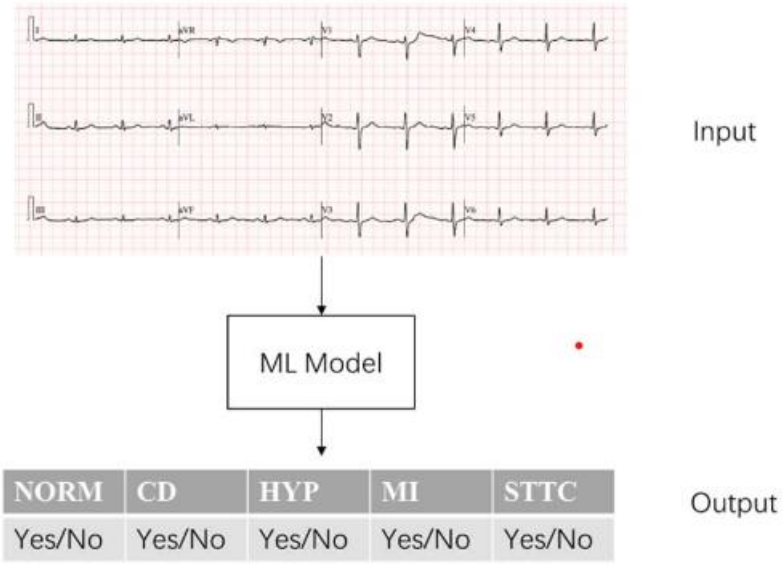

图 1. 算法预期目标

二、 材料和方法

2.1 材料

2.2.1 卷积神经网络基本框架（CNN）

传统神经网络对图像的识别分类具有局限性，无法将所处位置不同的同一物体准确识别出来。而卷积神经网络可以有效克服传统神经网络在图像识别的局限性，并在目标检测、图像生成、视频分析、医学成像、自动驾驶、卫星图像处理、语音识别和自然语言处理等领域有着重大优势和广泛应用。卷积神经网络是一类特殊类型的深度前馈型神经网络，它们在处理具有网格结构数据的时候表现出色。卷积神经网络通过局部连接、权重共享和池化等技术，有效地减少了网络的参数数量，同时保留了图像的空间层次结构。卷积神经网络的基本结构由输入层、卷积层、池化层、全连接层及输出层构成，卷积层和池化层一般会取若干个，采用卷积层和池化层交替设置。卷积层中输出特征面的每个神经元与其输入进行局部连接，并通过对应的连接权值与局部输入进行加权求和再加上偏置值，得到神经元的输入值，该过程为卷积。卷积神经网络通常由以下几个部分组成：

1）输入层：

输入层接收原始图像的数据。图像通常由红、绿、蓝三个颜色通道的数据组成，每个通道的数据由一个矩阵表示，矩阵的每一个元素对应一个像素，代表颜色的深浅，三个矩阵排列起来构成一个张量。

2）卷积层：

卷积神经网络的核心层，用于执行卷积操作，其主要特点是局部连接和参数共享。

卷积操作通过滑动一个小的窗口（卷积核、滤波器）在输入图像上，并计算窗口内像素与卷积核的乘积之和，加上偏置值，然后将结果输出为新的特征图。不同的卷积核可以提取输入图像的不同特征，如边缘、角点、纹理等。

输入层前向传播到卷积层是卷积神经网络前向传播算法的第一步，其过程为：

$$X^{32} = \sigma(Z^{32}) = \sigma(X^1 * W^{32} + b^{32})$$

其中，输入层X是个矩阵，矩阵的值是图片各像素的值。如果样本为 RGB 彩色图片，则输入层X为三个矩阵；若样本为灰度图片，则输入层X为一个矩阵。卷积核W也是个矩阵，一般为方阵，且卷积层的深度必须与输入图像的深度相同。公式上标表示层数，星号表示卷积，b表示偏置，σ为激活函数。公式（1）表示输入层为一个矩阵，卷积核深度为 32，有 32 个偏置，激活函数采用$ReLU$函数。

3）池化层：

模拟人眼对图像的模糊处理。池化层也称为下采样层，用于降低特征图的维度，提取主要特征，这有助于减少计算量和参数数量，控制过拟合，并使特征具有一定的平移不变性。常见的池化操作包括最大池化和平均池化：平均池化能够提取出图像局部特征区域的整体特征，提炼局部特征区域的背景；最大池化可以提取出图像的边缘纹路，放大不同特征区域的内容差异，但可能会引起图像的过度失真。

4）全连接层：

在卷积神经网络的末端，通常有一个或多个全连接层，用于整合前面各层提取的特征，并输出最终的结果。在全连接层中，每个神经元都与前一层的所有神经元相连。

5）输出层：

根据任务的不同，输出层可以设计为分类任务中的$softmax$层，用于输出类别的概率分布，或回归任务中的线性层，用于直接输出预测值。

2.2.2 卷积神经网络算法（CNN）

下面说明卷积神经网络的前向传播和反向传播的算法思想。前向传播是指神经元的输出是上一层神经元输出的加权和，加上一个偏置项，然后通过激活函数进行非线性变换传递到下一层中的过程。 步骤如下：

1） 若第l层是卷积层，则输出为：

$$a^l = ReLU(z^l) = ReLU(a^{l-1} * W^l + b^l)$$

其中a为输入的张量，W为卷积核的参数，b为偏置，z为状态值，作为激活函数的一个输入。

2） 若第l层是池化层，则输出为：

$$a^l = pool(a^{l-1})$$

这里的$pool$是指按池化区域的大小k和池化标准将输入张量缩小的过程。

3） 若第l层是全连接层，则输出为：

$$a^l = \sigma(z^l) = \sigma(a^{l-1}W^l + b^l)$$

其中a为输入的张量，W为每个神经元的连接参数，b为偏置，σ为激活函数。

4） 若第l层是输出层，则输出为：

$$a^l = softmax(z^l) = softmax(a^{l-1}W^l + b^l)$$

其中a为输入的张量，W为每个神经元的连接参数，b为偏置，$softmax$为激活函数。

卷积神经网络的反向传播过程是基于链式法则，计算损失函数对模型参数的梯度，从而更新权重和偏差，减少预测误差的过程。主要步骤如下：

1）若当前为全连接层：$\delta^{i,l} = (W^{l+1})^T\delta^{i,l+1} \odot \sigma'(z^{i,l})$

其中$\delta^{i,l}$为当前第l层的梯度误差，$\odot$表示点乘运算，W是神经元的权重参数，σ'是激活函数的导数，$z^{i,l}$为当前第l层的状态值。

更新权重为：$W^l = W^l - \alpha\sum_{i=1}^{m}\ \delta^{i,l}(a^{i,l-1})^T$

更新偏置为：$b^l = b^l - \alpha\sum_{i=1}^{m}\ \delta^{i,l}$

其中α为学习率，$a^{i,l-1}$为第l-1 层的输出张量，$\delta^{i,l}$为第l层的梯度误差。

2） 若当前为卷积层：$\delta^{i,l} = \delta^{i,l+1} * rot180(W^{l+1}) \odot \sigma'(z^{i,l})$

其中$rot180$表示卷积核被旋转了180度。

更新权重为：$W^l = W^l - \alpha\sum_{i=1}^{m}\ \delta^{i,l} * rot180(a^{i,l-1})$）

更新偏置为：$b^l = b^l - \alpha\sum_{i=1}^{m}\ \sum_{u,v}\ (\delta^{i,l})_{u,v}$

3） 若当前是池化层：$\delta^{i,l} = upsample(\delta^{i,l+1}) \odot \sigma'(z^{i,l})$

其中$upsample$指在反向传播时把$\delta^{i,l}$的所有子矩阵矩阵大小还原成池化之前大小的过程。如果是最大池化，则把$\delta^{i,l}$所有子矩阵的各个池化局域的值放在之前做前向传播算法得到最大值的位置。如果是平均池化，则把$\delta^{i,l}$所有子矩阵的池化局域的值取平均值后放在还原后的子矩阵位置。$upsample$完成了池化误差矩阵放大与误差重新分配的逻辑。

2.2 方法

本课程设计建立的模型是 5 层经典的卷积神经网络。该模型包含 5 层和 1 个 fc 层。每层包含一个批量归一化层、一个 dropout 层（dropout rate=0.2）和一个 Avg pooling 层。对于第一层，它有 12 个输入通道和 16 个输出通道，最后 4 层有越来越多的输出通道，以增强模型的高级特征提取能力。该模型具有更浅的结构和更小的尺寸，实现了更高的准确性，因为池化层有效地减小了模型的大小，并且这些过程增加了模型的泛化能

力和防止模型过度拟合。基本结构如下：

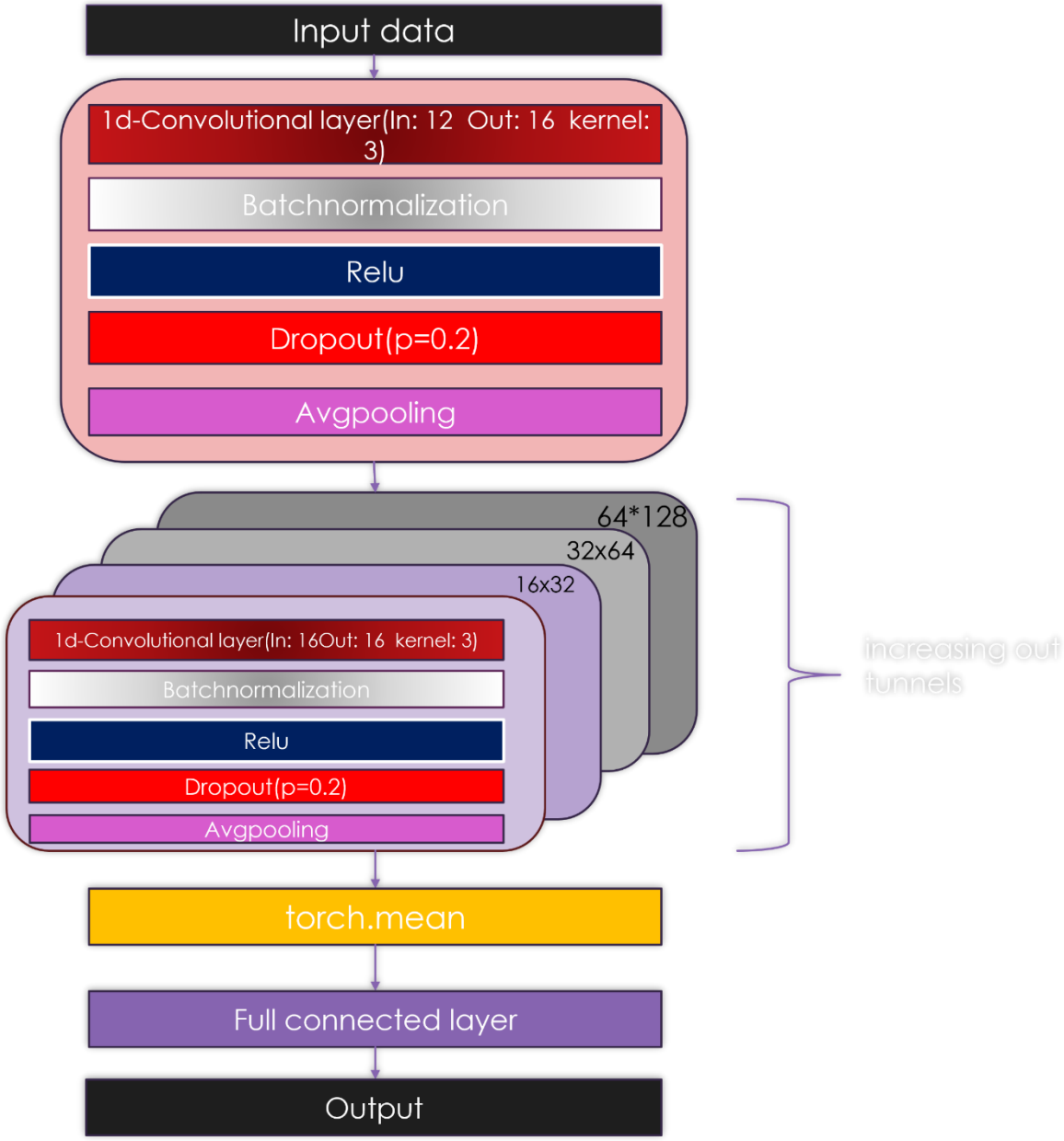

图 2. 经典 5 层卷积神经网络的基本架构

三、 实验与结果

3.1 数据预处理

题目要求使用 PTB-XL 心电图数据集。PTB-XL ECG 数据集是一个大型数据集，包含 10 秒长的 18869 名患者的 21799 个临床 12 导心电图。原始波形数据由多达两名心脏病专家注释，他们将潜在的多个心电图语句分配给每个记录。总共有 71 个不同的心电图语句符合 SCP-ECG 标准，涵盖了诊断、形式和节奏语句。这个数据集其中 52% 为男性，48% 为女性，年龄范围从 0 到 95 岁（中位数 62 岁，分位数范围为 22）。

预处理过程主要是以下几个部分：首先，我们需要处理特征数据，即“X”：患者的 ECG 数据。我们需要从 DAT 文件中读取数据并将其存储在张量中。这是一个相对复杂的过程。为了完成此任务，我们定义了一个名为 PTBXLDataset 的类，并将此过程封装在 PTBXLDataset 类的初始化函数中，将文件夹地址作为参数。在初始化函数中，我们处理文件夹地址。首先，我们定位地址指定的文件夹中的“ptbxl_database.csv”文件。该文件作为目录文件，其中“filename_hr”和“filename_lr”列包含特定数据的文件名，采样率分别为 500 和 100。我们从这两个列中提取元素，并将文件夹地址添加一个级别，以获得特定文件地址。通过使用 for 循环遍历 filename 列中的所有元素，我们可以使用 wfdb 包中的 rdsamp 方法读取所有数据；其次，为了处理标签，我们做的过程与特征数据类似，但添加了两个步骤将其转换为 one hot 标签。读取标签数据并将其存储到列表（这是一个多标签列表，因为一个病人可能有多个标签）之后，我们需要将该列表转换为可以在 pytorch 中处理的一个热标签。它包含两个步骤:首先，使用 Label mapping 列表，我们可以将每个标签编码为表示标签的数字；接着，质量检验过程包括缺失值分析和异常值分析。通过对数据集的检查，我们没有发现缺失值，并且根据 PTBXL 数据集的描述，数据采集过程是在严格的管理和检查下进行的，所以我们可以假设特征数据中没有缺失值和异常值。但对于标签来说，由于多标签样品，可能存在某种反逻辑问题。我们假设样本没有逻辑错误，并平等地处理它们。而对于特征提取过程，我们跳过这个过程，因为深度学习模型的能力可以让我们在不需要人工特征提取的情况下进行端到端的学习。我们只需将序列数据输入到模型中，卷积层就可以自动完成特征提取过程；最后，该项目使用“strat_fold”的值对训练和测试数据集进行拆分，对于每个样本，如果“strat_fold“的值等于“test_fold”，则它属于测试数据集。否则它属于训练数据集。此时，训练集与测试集样本比例大约在 9:1。

完成上述操作后，我们绘制第 5000 个 ECG 信号波形图，并将其展示：

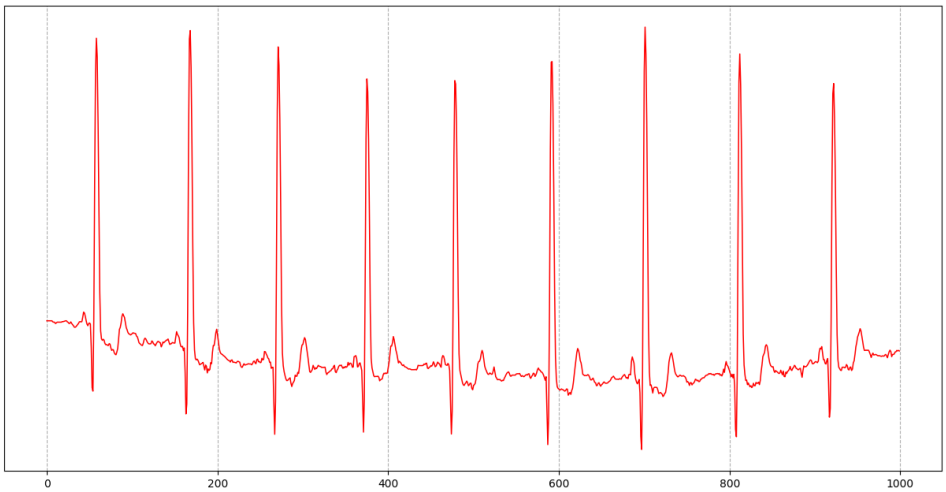

图 3.第 5000 个 ECG 信号波形图

3.2 评价指标

这里本文为评估模型的有效程度，引入了几个参数进行评价：

（1）Loss：MultiLabelSoftMarginLoss 是 PyTorch 中用于多标签分类任务的损失函数。它适用于每个样本可能属于多个类的情况。该损失函数将每个样本的每个类视为一个独立的二进制分类问题，并计算每个类别的二进制交叉熵损失。对于每个样本，MultiLabelSoftMarginLoss 计算预测概率与真实标签之间的交叉熵损失。且对于每个类别，损失函数平均所有样本的交叉熵损失，以获得该类别的损失。最后，再次对所有类别的损失进行平均，得到最终损失值；

（2）AUROC 是一种常用的二分类评价指标，表示 ROC 曲线下面积。ROC 曲线展示了分类模型的真正率（True Positive Rate, TPR）与假正率（False Positive Rate, FPR）之间的关系，AUROC 值越接近 1，说明模型的分类性能越好。**AUROC per class** 是指针对多分类问题中每个类别分别计算的 AUROC（ROC 曲线下面积）。对于多分类问题，可以将每个类别单独视为一个二分类问题（该类别为“正类”，其他类别为“负类”），然后计算该类别的 AUROC；Macro AUROC 是在多分类任务中使用的一种方法，主要通过以下方式计算：针对多分类问题的每个类别，将该类别视为“正类”，其余类别视为“负类”，然后计算出每个类别的 AUROC(其值越接近 1，模型的区分能力越强。0.5 表示模型没有分类能力，相当于随机猜测。低于 0.5 表示模型在反向分类，即把负样本预测为正样本，把正样本预测为负样本)；取这些 AUROC 的平均值：将每个类别的 AUROC 取平均值，得到宏观意义上的 AUROC，即 Macro AUROC；

（3）F1 per class 和 F1 Macro 是用于评估多分类模型性能的两种方式，尤其是

在类别不平衡情况下非常有用。它们都基于 F1-score，这是精确率（Precision）和召回率（Recall）的调和平均数。F1 per class 是指针对每个类别分别计算的 F1-score。具体计算步骤如下：针对每个类别计算 Precision 和 Recall（其中 Precision（精确率）：正确预测为该类别的样本数量占所有被预测为该类别的样本总数的比例，Recall（召回率）：正确预测为该类别的样本数量占该类别的真实样本总数的比例）；计算 F1-score：

$$F_1 = 2 * \frac{Precision * Recall}{Precison + Recall}$$

F1-score 是精确率（Precision）和召回率（Recall）的调和平均数，因此越接近 1，模型的表现越好。F1-score 为 0 表示模型预测完全错误，为 1 表示预测完全正确。

F1 Macro 是对所有类别的 F1 per class 的平均值，具体步骤如下：计算每个类别的 F1-score：按照 F1 per class 的方式计算;取平均值：将所有类别的 F1-score 相加，再除以类别数，得到宏观 F1-score，即 F1 Macro。F1 Macro 是所有类别 F1-score 的平均值，越接近 1，表示模型在所有类别上的整体表现越好。与 F1 per class 类似，接近 1 表示模型的表现较优。F1 Macro 是评估模型整体性能的一个简便指标，而 F1 per class 则提供了更细粒度的模型分析。

3.3 结果展示与分析

这里我们分别设置 epoch，dropout 为不同取值，得出下列表格：

3.3.1 epoch 设置不同

epoch	Loss	AUROC Macro	F1 Macro
10	0.3661	0.9130	0.6769
20	0.3591	0.9155	0.6982
40	0.3598	0.9133	0.6781

表 1.不同 epoch 下的评价指标

实验结果表明，模型在不同的 epoch 设置下的大多数类别上都取得了较高的 AUROC 和 F1-score，证明了其有效性。然而，我们也不难发现，当设置 epoch 取值为 10 时，模型无法完全学习数据的特征，导致欠拟合；当设置 epoch 取值为 20 时，模型有足够的时间学习训练数据的特征，能够较好地拟合训练数据，并在验证集或测试集上表现良好。此时，模型可以达到训练损失和测试损失之间的平衡；当设置 epoch 取值为 40 时，模型开始过度拟合训练数据，即在训练集上的损失持续降低，但在验证集或测试集上性能开始下降。

3.3.2 dropout 设置不同

在经典五层卷积神经网络中，Dropout 是一种常用的正则化技术，能够减少过拟合，增强模型的泛化能力。Dropout 的主要作用是随机地将一部分神经元的输出设为零，这样模型在每次训练时都会学习到不同的特征组合。这里在选定 epoch 为 20 的情况下，分别选定 dropout 为 0.1,0.2 以及 0.3，得到的结果如表 2 所示：

dropout	Loss	AUROC Macro	F1 Macro
0.1	0.3450	0.9180	0.7050
0.2	0.3591	0.9155	0.6982
0.3	0.3750	0.9120	0.6900

表 2. 不同 dropout 下的评价指标

我们得出结论：较低的 Dropout（如 0.1）：适用于模型不容易过拟合的情况，但可能在某些情况下导致过拟合；中等的 Dropout（如 0.2）：通常提供良好的平衡，适用于大多数场景；较高的 Dropout（如 0.3）：适用于模型容易过拟合的情况，但过高的 Dropout 可能导致欠拟合。

3.3.3 训练误差图

我们以 epoch 为 20，dropout 为 0.2，给出一张训练模型图，如图 4 所示：

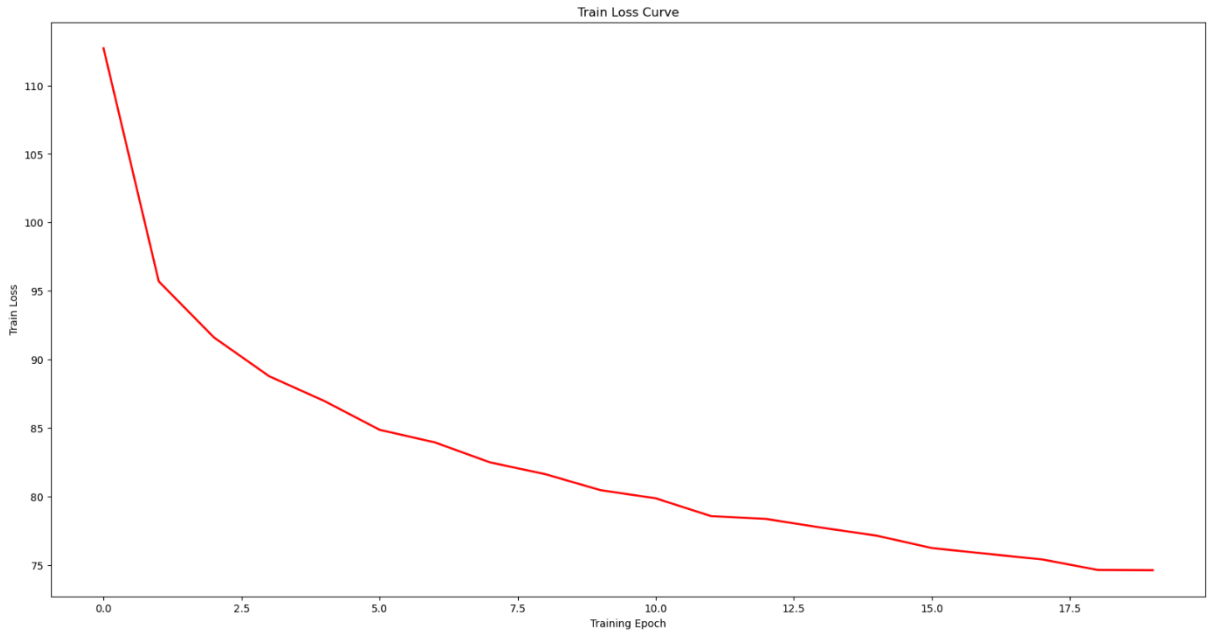

图 4. 训练误差图

不难发现，前期训练误差下降的较为迅速，后期则较为平缓。这也验证了我们 3.3.1 的结果。

3.3.4 课程设计的总结

本课程项目展示了如何使用 PTB-XL 数据集进行多标签心电图分类任务。通过这一项目，我不仅掌握了多标签分类任务的基本方法，还积累了处理医学信号数据的经验。这些经验对未来从事相关领域的研究和应用具有重要意义。

其中遭遇的最大的一个难题是 Pytorch 安装好几次，都无法在 vscode 中导入。经过一系列的环境配置和 Pytorch 的多次下载，还是无法解决问题。经过请教同学以及一系列的尝试，我们才发现可能是因为下载的 Pytorch 是最新版本，自身存在了 bug。后面我下载了 2.2.0 版本，就完美地解决了问题。

另外，这次的课设我做的整体还是显得较为粗糙。究其原因，还是自己投入的时间不够，再加之，自己的代码能力以及对机器学习的相关领域了解的非常浅显。大三升大四这个暑假确实较为繁忙，首先是参加各大高效的夏令营一直到 7 月底，接着八月初开始准备两个从零开始的课程设计，与此同时还准备着开学的缓考科目和预推免相关的考试资料。但好在有着同学，学长学姐和老师们的帮助，让我能够艰难地完成此次课程设计，真的让我受益匪浅。在今后，还是得加强相关能力的训练。

参考文献

[1]郝志峰.数据科学与数学建模.华中科技大学出版社

电子签名：

附录：

```
import torch

import numpy as np
from scipy import io #用于输入输出
import pandas as pd
from torch.utils.data import Dataset, DataLoader #处理数据
from sklearn import preprocessing #进行数据的预处理
from sklearn.preprocessing import MultiLabelBinarizer #处理多标签分类任务
import wfdb #加载和处理 ECG 信号数据
import ast #用于可视化 ECG 信号或结果
import matplotlib.pyplot as plt
class PTBXLDataset(Dataset): #用以加载和处理 PTB-XL 数据集，这样处理的数据可以被直接用来训练 PyTorch 模型或者进行测试
    def __init__(self, path,sampling_rate = 500, train_set =True, transform=None): #transform: 可选的信号变换函数，用于对数据进行预处理
        def load_raw_data(df, sampling_rate, path):
            if sampling_rate == 100:
                data = [wfdb.rdsamp(path+'/'+f) for f in df.filename_lr]
            else:
                data = [wfdb.rdsamp(path+'/'+f) for f in df.filename_hr]
#wfdb.rdsamp 函数用于读取心电信号数据
            data = np.array([signal for signal, meta in data])
            return data


        Y = pd.read_csv(path+'\ptbxl_database.csv', index_col='ecg_id') #标签数据从 ptbxl_database.csv 文件中加载，并解析为字典
        Y.scp_codes = Y.scp_codes.apply(lambda x: ast.literal_eval(x)) #返回一个集合，包含与当前记录相关的诊断类别

        # 导入原始数据
        X = load_raw_data(Y, sampling_rate, path)
```

```
        agg_df = pd.read_csv(path+'\scp_statements.csv', index_col=0)
#index_col=0 表示使用文件的第一列作为 DataFrame 的索引。
        agg_df = agg_df[agg_df.diagnostic == 1] #只记录有限的数据

        def aggregate_diagnostic(y_dic): #用于将每个 ECG 记录的详细诊断代码
（y_dic）转换为诊断超类。
            tmp = []
            for key in y_dic.keys():
                if key in agg_df.index:
                    tmp.append(agg_df.loc[key].diagnostic_class)
            return list(set(tmp))

        # 应用诊断超类
        Y['diagnostic_superclass'] = Y.scp_codes.apply(aggregate_diagnostic) #
存储每个记录的诊断超类

        # 拆分成训练集与测试集
        test_fold = 10
        # 训练
        X_train = X[np.where(Y.strat_fold != test_fold)]
        y_train = Y[(Y.strat_fold != test_fold)].diagnostic_superclass
        # 测试
        X_test = X[np.where(Y.strat_fold == test_fold)]
        y_test = Y[Y.strat_fold == test_fold].diagnostic_superclass

        if train_set:
            self.data = np.array(X_train)
            labels = list(y_train)
        else:
            self.data = np.array(X_test)
            labels = list(y_test)

        label_mapping = {"NORM": 0, "CD": 1, "HYP": 2, "MI": 3, "STTC": 4}
```

```
        self.label = [[label_mapping[label] for label in sample] for sample in
labels] #将诊断超类转换为数字标签。

        mlb = MultiLabelBinarizer()
        self.onehot_label = mlb.fit_transform(self.label)
        self.onehot_label = torch.tensor(self.onehot_label,
dtype=torch.float32)

        self.data = torch.tensor(self.data,
dtype=torch.float32).permute(0,2,1)
        self.data = torch.nan_to_num(self.data, nan=0.0)
        self.X=X

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        data = self.data[idx]
        onehot_label = self.onehot_label[idx]
        return data,onehot_label

#读取数据
path = r"c:\Users\TJ\Desktop\统计软件课程代码\ptb-xl-a-large-publicly-available-
electrocardiography-dataset-1.0.3"
sampling_rate = 100
#创建数据集
train_dataset = PTBXLDataset(path,sampling_rate=100,train_set=True)
test_dataset = PTBXLDataset(path,sampling_rate=100,train_set=False)

#创建数据加载器
fine_tune_loader = DataLoader(train_dataset, batch_size=64, shuffle=True) #用
于训练的 DataLoader，设置了批量大小为 64，并且打乱了数据
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

```
#ECG 绘制第五千个信号波形图
plt.rcParams['figure.figsize'] = (20.0, 10.0)
plt.figure()
plt.plot(train_dataset.X[5000][:,0],color='red', linewidth=1.1)
plt.grid(linestyle='--')
plt.yticks([])
plt.show()

import torch.nn as nn
# 五层经典卷积网络
class CNN_1D_5L(nn.Module):
    def __init__(self, n_in):
        super().__init__()
        self.n_in = n_in
        self.layer1 = nn.Sequential(
            nn.Conv1d(12, 16, (3,), stride=1, padding=4),
            nn.BatchNorm1d(16),#批量归一化
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.AvgPool1d(2,stride=2)#每次池化操作在输入中选择 2 个值，计算它们的平均
值
        )

        self.layer2 = nn.Sequential(
            nn.Conv1d(16, 16, (5,), stride=1, padding=2),
            nn.BatchNorm1d(16),
            nn.ReLU(),
            nn.Dropout(p=0.2),#丢弃部分神经元，防止过拟合
            nn.AvgPool1d(2,stride=2)#池化操作
        )

        self.layer3 = nn.Sequential(
            nn.Conv1d(16, 32, (5,), stride=1, padding=2),
            nn.BatchNorm1d(32),
```

```
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.AvgPool1d(2,stride=2)
        )
        self.layer4 = nn.Sequential(
            nn.Conv1d(32, 64, (5,), stride=1, padding=2),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.AvgPool1d(2,stride=2)
        )
        self.layer5 = nn.Sequential(
            nn.Conv1d(64, 128, (5,), stride=1, padding=2),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.AvgPool1d(2,stride=2)
        )


        self.linear1 = nn.Linear(128, 5) #全连接层，128 个输入，5 个输出


    def forward(self, x):#神经网络的前向传播过程
        x = x.view(-1, 12, self.n_in)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.layer5(x)
        x=torch.mean(x,dim=-1)#将特征图从一个二维的形状（[batch_size, 128, 特征图长度]）转化为一维的形状（[batch_size, 128]）
        return self.linear1(x)#每行包含一个样本在 5 个类别上的预测分数
```

```
#评价函数和训练函数
from torch import nn as nn#专门用于构建神经网络的模块
import torch
from sklearn.metrics import confusion_matrix, f1_score, accuracy_score
from sklearn.metrics import roc_auc_score
import datetime
import torch.nn.functional as F


#评价函数
def evaluate_multilabel_classification(model, dataloader, device, criterion):
    model.eval()
    all_labels = []
    all_predictions = []# 用于存储所有的真实标签和预测结果
    total_loss = 0.0#记录累积损失
    total_samples = 0#记录处理的样本总数

    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)
            predictions = torch.sigmoid(outputs)#将原始输出转化为 [0, 1] 之间的概
率

            loss = criterion(outputs, labels.float())#计算当前批次的损失

            total_loss += loss.item() * inputs.size(0)
            total_samples += inputs.size(0)

            all_labels.append(labels.cpu().numpy())
            all_predictions.append(predictions.cpu().numpy())

    average_loss = total_loss / total_samples#通过累加的损失和总样本数计算平均损失
```

```
    all_labels = np.vstack(all_labels)
    all_predictions = np.vstack(all_predictions)

    # 计算每一个标签的 AUROC
    auroc_list = []
    for i in range(all_labels.shape[1]):
        auroc = roc_auc_score(all_labels[:, i], all_predictions[:, i])
        auroc_list.append(auroc)

    # 计算平均的 AUROC
    auroc_macro = np.mean(auroc_list)

    #计算每一个标签的 F1 分数
    f1_list = []
    for i in range(all_labels.shape[1]):
        f1 = f1_score(all_labels[:, i], (all_predictions[:, i] >
0.5).astype(int))
        f1_list.append(f1)

    #所有类别的平均 F1 分数
    f1_macro = np.mean(f1_list)

    return {"Loss": average_loss, "AUROC per class": auroc_list, "AUROC
Macro": auroc_macro, "F1 per class": f1_list, "F1 Macro": f1_macro}
#训练函数
def train_model_ptbxl(train_loader, test_loader, model, criterion, optimizer,
num_epochs, device='cpu',evaluate=True):
    model.to(device)
    train_loss = []
    test_loss=[]
    macro_auc=[]
    f1_auc=[]
    for epoch in range(num_epochs):
```

```
        model.train()#启用训练模式
        total_loss = 0.0
        start_time = datetime.datetime.now()
        for inputs, labels in train_loader:#从 train_loader 中获取批次数据，并将数据和标签移动到指定设备上
            optimizer.zero_grad()
            outputs = model(inputs.to(device))
            loss = criterion(outputs, labels.to(device))#将输入数据传入模型，计算输出，并使用损失函数计算损失
            loss.backward()
            optimizer.step()#计算梯度并更新模型参数
            total_loss += loss.item()#叠加损失
        end_time = datetime.datetime.now()
        if evaluate:
            model.eval()
            result = evaluate_multilabel_classification(model, test_loader, device, criterion)
            formatted_result = {
                "Loss": "{:.4f}".format(result["Loss"]),
                "AUROC per class": ["{:.4f}".format(auroc) for auroc in result["AUROC per class"]],
                "AUROC Macro": "{:.4f}".format(result["AUROC Macro"]),
                "F1 per class": ["{:.4f}".format(f1) for f1 in result["F1 per class"]],
                "F1 Macro": "{:.4f}".format(result["F1 Macro"])
            }#格式化评估结果
            train_loss.append("{:.4f}".format(total_loss))
            test_loss.append(formatted_result["Loss"])
            macro_auc.append(formatted_result["AUROC Macro"])
            f1_auc.append(formatted_result["F1 Macro"])#记录当前 epoch 的结果
            print(f"Epoch [{epoch + 1}/{num_epochs}]","train_loss:", "{:.4f}".format(total_loss),"test_Loss:", formatted_result["Loss"],
                "AUROC Macro:", formatted_result["AUROC Macro"],
```

```
        "F1 Macro:", formatted_result["F1 Macro"], "Time:", end_time -
start_time)

    return train_loss,test_loss,macro_auc,f1_auc

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")#检测是否
有可用的 GPU
device

#加载模型以及训练参数
model = CNN_1D_5L(1000)
model.to(device)

lr = 0.001#决定了每次迭代时，模型参数沿着梯度方向的调整幅度
optimizer = torch.optim.AdamW(model.parameters(), lr = lr,weight_decay = 1e-
2)# 使用 AdamW 优化器，并设置权重衰减（L2 正则化）为 0.01
criterion = nn.MultiLabelSoftMarginLoss()
epochs = 40
#训练和评价模型
train_loss,test_loss,macro_auc,f1_auc=train_model_ptbxl(fine_tune_loader,
test_loader, model, criterion, optimizer,epochs, device=device)
model.eval()

torch.save(model.state_dict(), 'model_weights_ptbxl.pth')#训练完成后，保存模型的
权重到 model_weights_ptbxl.pth 文件中

result = evaluate_multilabel_classification(model, test_loader, device,
criterion)
formatted_result = {
    "Loss": "{:.4f}".format(result["Loss"]),
    "AUROC per class": ["{:.4f}".format(auroc) for auroc in result["AUROC per
class"]],
    "AUROC Macro": "{:.4f}".format(result["AUROC Macro"]),
    "F1 per class": ["{:.4f}".format(f1) for f1 in result["F1 per class"]],
```

```
    "F1 Macro": "{:.4f}".format(result["F1 Macro"])
}

# 输出结果
print("Formatted Result:")
print("Loss:", formatted_result["Loss"])
print("AUROC per class:", formatted_result["AUROC per class"])
print("AUROC Macro:", formatted_result["AUROC Macro"])
print("F1 per class:", formatted_result["F1 per class"])
print("F1 Macro:", formatted_result["F1 Macro"])
```