

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. 实验代码解析	3
2. 模型结构图	12
3. 损失函数详细描述	13
4. 天池提交分数	15
5. 实验总结	16

1. 实验代码解析

本实验在 Google Colab 环境下进行，实验数据存储在 Google Drive 中,通过以下代码将 Google Drive 挂载到 Colab 环境。

```
[ ] # 将google drive挂载到colab到/content/drive目录
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

图一：代码块 1

以下这段代码首先导入了必要的数据处理和机器学习库，包括 pandas 和 numpy 用于数据处理，torch 和相关模块用于构建和训练神经网络模型，以及 matplotlib 用于数据可视化。随后，通过检测可用的计算设备（GPU 或 CPU），设置了训练过程中使用的设备，以优化计算效率。

接下来，从指定路径加载了训练数据和测试数据，并通过 pandas 库的 read_csv 函数读取 CSV 文件。这一步确保了数据被成功导入，并存储在 train_data 和 test_data 两个数据框中。通过打印数据集的形状和标签分布，对数据的基本结构和类别分布进行了初步的了解，这有助于发现数据集中的类别不平衡等问题，为后续的数据处理和模型训练提供了必要的信息。

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

# 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('/content/drive/MyDrive/Colab Notebooks/assets/train.csv')
test_data = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/assets/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
```

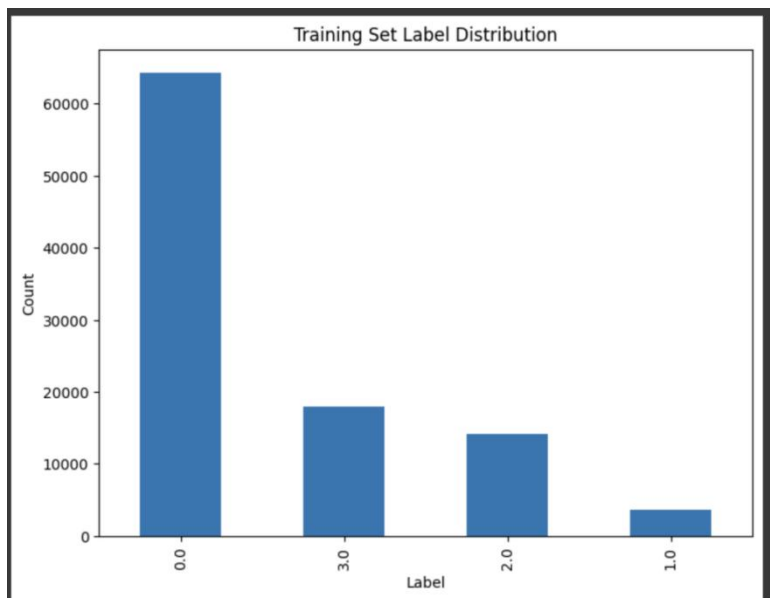
图二: 代码块 2

下面这段代码首先使用 matplotlib 库对训练集标签的分布情况进行可视化。通过 pandas 的 value_counts 方法统计每个标签的样本数量，并绘制成条形图 (bar chart)。条形图展示了不同类别标签在训练集中的分布情况，有助于识别类别不平衡问题，例如某些类别的样本数量明显多于其他类别，这在模型训练时可能导致偏差。

```
# 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()

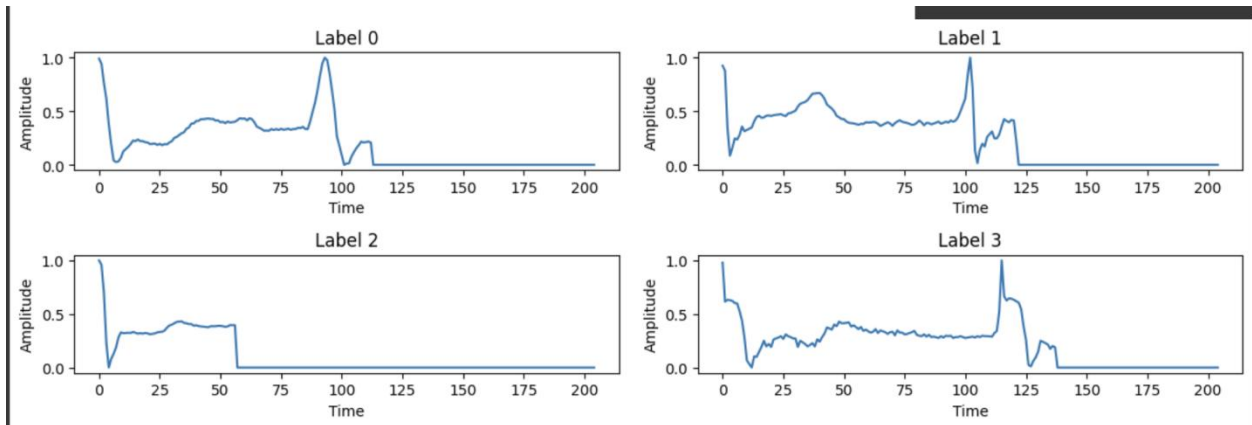
# 绘制心跳信号波形图
plt.figure(figsize=(12, 4))
for i in range(4):
    signal = train_data[train_data['label'] == i]['heartbeat_signals'].values[0]
    signal = np.array(signal.split(','), dtype=np.float32)
    plt.subplot(2, 2, i+1)
    plt.plot(signal)
    plt.title(f'Label {i}')
    plt.xlabel('Time')
    plt.ylabel('Amplitude')
plt.tight_layout()
plt.show()
```

图三: 代码块 3



图四：可视化训练集 label

接下来，代码绘制了不同标签对应的心跳信号波形图。通过遍历所有标签（0 到 3），提取每个标签的一条心跳信号数据，并将信号字符串转换为浮点数数组。使用 subplot 方法将四个标签的波形图绘制在同一图中，不同标签的心跳信号波形展示了其时间序列特征。波形图横轴表示时间，纵轴表示信号幅度，这些图形为理解心跳信号的时序特征及其在不同标签间的差异提供了直观的参考。



图五：可视化不同标签对应的心跳信号

```
# 数据预处理
class HeartbeatDataset(Dataset):
    def __init__(self, data, mode='train'):
        self.data = data
        self.mode = mode

        if self.mode == 'train':
            self.labels = data['label'].values
            self.encoder = LabelEncoder()
            self.labels = self.encoder.fit_transform(self.labels)

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

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

    def __getitem__(self, idx):
        signal = self.signals.iloc[idx]

        if self.mode == 'train':
            label = self.labels[idx]
            return torch.tensor(signal, dtype=torch.float32), torch.tensor(label, dtype=torch.long)
        else:
            return torch.tensor(signal, dtype=torch.float32)

# 创建数据集和数据加载器
train_dataset = HeartbeatDataset(train_data)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```

图六：代码块 4

在上面这段代码中，我们首先定义了一个名为 `HeartbeatDataset` 的自定义数据集类，继承自 `torch.utils.data.Dataset`。这个类主要用于对心跳信号数据进行预处理。构造函数 `__init__` 接收数据和模式（训练或测试），并在训练模式下，对标签进行编码。心跳信号数据通过字符串拆分并转换为浮点数数组，以便模型处理。实现了 `__len__` 方法返回数据集的大小，以及 `__getitem__` 方法获取指定索引的数据项，这些方法使得数据集对象可以被 PyTorch 的数据加载器使用。

随后，代码创建了 `HeartbeatDataset` 类的实例 `train_dataset`，并通过 `DataLoader` 将其包装成数据加载器 `train_loader`，设定了批量大小为 64，并启用了随机打乱（shuffle）功能。这确保了在每个训练迭代中，数据批次是随机的，帮助模型更好地泛化，避免过拟合。这些步骤是深度学习数据预处理和加载的重要环节，为后续的模型训练提供了结构化和高效的数据输入方式。

```
# 定义CNN模型
class HeartbeatClassifier(nn.Module):
    def __init__(self):
        super(HeartbeatClassifier, self).__init__()
        self.conv1 = nn.Conv1d(1, 64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv1d(64, 128, kernel_size=3, padding=1)
        self.conv3 = nn.Conv1d(128, 256, kernel_size=3, padding=1)
        self.pool = nn.MaxPool1d(2)
        self.fc1 = nn.Linear(256 * 25, 512)
        self.fc2 = nn.Linear(512, 4)

    def forward(self, x):
        x = x.unsqueeze(1) # Add channel dimension
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

图七：代码块 5

在上面这段代码中，我们定义了一个卷积神经网络（CNN）模型 HeartbeatClassifier，继承自 torch.nn.Module。这个模型主要用于对一维心跳信号数据进行分类。模型包含三层一维卷积层（conv1、conv2、conv3），每层卷积后接一个最大池化层（pool），用于逐步减少特征图的长度并提取重要特征。卷积层的通道数分别为 64、128 和 256，卷积核大小为 3，填充为 1，保持特征图的长度不变。

在卷积和池化操作之后，模型将多维特征展平为一维向量，并通过两个全连接层（fc1 和 fc2）进行分类。第一个全连接层将特征向量映射到 512 维，第二个全连接层将其进一步映射到 4 个输出类别。每个卷积层和全连接层后都使用了 ReLU 激活函数（F.relu），增加了模型的非线性能力。在 forward 方法中，输入数据先通过 unsqueeze(1) 添加一个通道维度，然后依次经过卷积层、池化层和全连接层，最终输出分类结果。

```
# 训练模型
model = HeartbeatClassifier().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

num_epochs = 100
train_losses = []

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    for signals, labels in train_loader:
        signals, labels = signals.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(signals)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

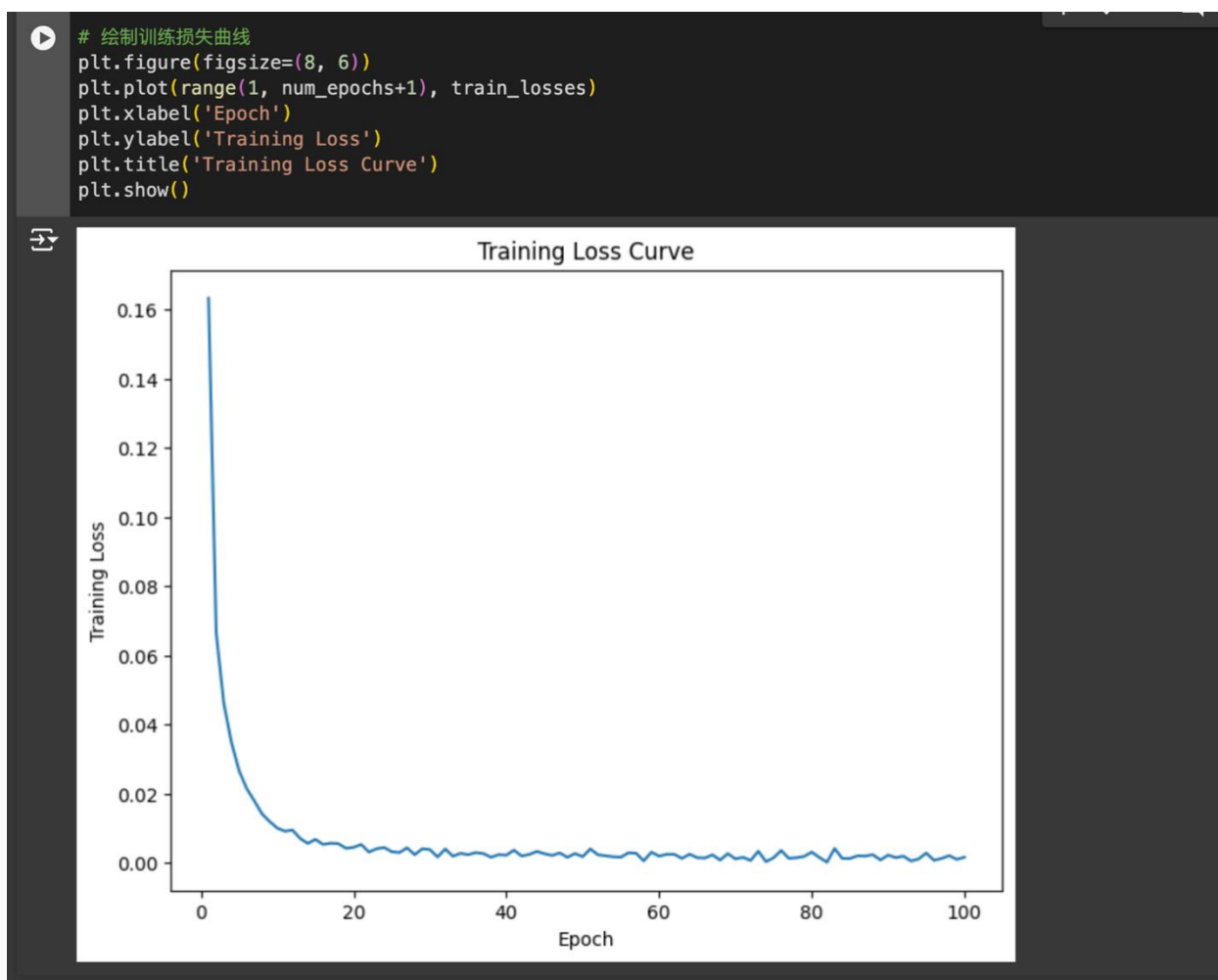
    epoch_loss = running_loss / len(train_loader)
    train_losses.append(epoch_loss)
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}')
```

图八：代码块 6

在上面这段代码中，我们首先实例化了先前定义的 `HeartbeatClassifier` 模型，并将其移动到指定的计算设备（GPU 或 CPU）。接着，定义了交叉熵损失函数（`nn.CrossEntropyLoss`）作为模型的损失函数，并使用 Adam 优化器（`torch.optim.Adam`），学习率设定为 0.001。

模型的训练过程迭代进行了 100 个 epoch 【之后更改为 200 个 epoch，评测结果有改进】。在每个 epoch 中，模型进入训练模式（`model.train()`），并初始化 `running_loss` 用于累计当前 epoch 的损失值。通过 `train_loader` 获取批量数据，每个批次数据都被移动到计算设备。接着，前向传播计算输出结果，通过损失函数计算损失值，然后进行后向传播（`loss.backward()`）以计算梯度，并使用优化器更新模型参数（`optimizer.step()`）。每个批次的损失值累加到 `running_loss` 中。

在每个 epoch 结束时，计算平均损失值（`epoch_loss`），并将其添加到 `train_losses` 列表中，同时打印当前 epoch 的损失值。这些步骤构成了完整的模型训练过程，通过多次迭代逐步优化模型参数，使其在训练数据上的表现不断提高。



图九: 代码块 7

在上面这段代码中，我们使用 matplotlib 库绘制了模型训练过程中的损失曲线，以便直观地展示训练损失值随 epoch 变化的趋势。首先，通过 `plt.figure(figsize=(8, 6))` 设置图形的尺寸为 8x6 英寸。接着，使用 `plt.plot(range(1, num_epochs+1), train_losses)` 函数绘制训练损失曲线，其中横轴表示训练的 epoch 次数，纵轴表示每个 epoch 的训练损失值。横轴和纵轴分别使用 `plt.xlabel('Epoch')` 和 `plt.ylabel('Training Loss')` 添加了描述性的标签，图表的标题则使用 `plt.title('Training Loss Curve')` 进行设置。最后，使用 `plt.show()` 函数显示绘制的损失曲线图。

通过这张训练损失曲线图，我们可以清晰地观察到训练过程中损失值的变化情况。这有助于评估模型的收敛情况，例如损失值是否持续下降，模型是否逐渐稳定，是否存在

过拟合的迹象等。总体而言，这段代码为模型的训练过程提供了重要的可视化支持，使我们能够更好地理解和调整训练策略。

```
[ ] # 保存模型的状态字典 (推荐方式)
torch.save(model.state_dict(), '/content/drive/MyDrive/Colab Notebooks/results/heartbeat_classifier_model_state.pth')

# 加载模型的状态字典
model = HeartbeatClassifier().to(device)
model.load_state_dict(torch.load('/content/drive/MyDrive/Colab Notebooks/results/heartbeat_classifier_model_state.p
model.eval() # 设置模型为评估模式

HeartbeatClassifier(
  (conv1): Conv1d(1, 64, kernel_size=(3,), stride=(1,), padding=(1,))
  (conv2): Conv1d(64, 128, kernel_size=(3,), stride=(1,), padding=(1,))
  (conv3): Conv1d(128, 256, kernel_size=(3,), stride=(1,), padding=(1,))
  (pool): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=6400, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=4, bias=True)
)

# 预测和提交结果
test_dataset = HeartbeatDataset(test_data, mode='test')
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

predictions = []

with torch.no_grad():
    for signals in test_loader:
        signals = signals.to(device)
        outputs = model(signals)
        probs = F.softmax(outputs, dim=1).cpu().numpy()
        predictions.extend(probs)

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]
})

submission.to_csv('/content/drive/MyDrive/Colab Notebooks/results/submission.csv', index=False, mode='w')
```

图十: 代码块 8

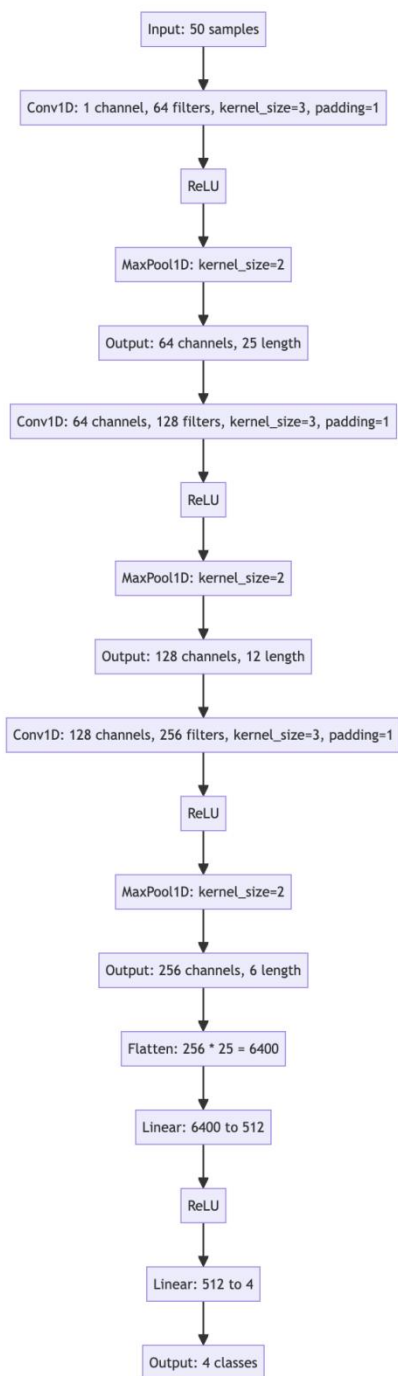
在上面这段代码中，我们首先将测试数据集包装成 `HeartbeatDataset` 对象，并通过 `DataLoader` 创建数据加载器 `test_loader`。在加载测试数据时，我们将 `batch_size` 设置为 64，且不打乱数据 (`shuffle=False`)，以确保预测结果的顺序与输入数据一致。接下来，我们将模型设置为评估模式 (`model.eval()`)，并在不计算梯度的上下文环境中 (`torch.no_grad()`) 进行预测。对每个批次的信号数据，首先将其移动到计算设备 (GPU 或 CPU)，然后通过模型前向传播计算输出。使用 `Softmax` 函数将输出转换为概率分布，并将其转换为 NumPy 数组后，扩展到 `predictions` 列表中。

最后，我们将预测结果组织成一个 DataFrame，每一列对应一个类别的预测概率，并包含测试数据的 ID 列。使用 `csv` 方法将预测结果保存为 CSV 文件，路径为 `/content/drive/MyDrive/Colab Notebooks/results/submission.csv`，以便提交比赛结果。这一步确保了预测结果与竞赛要求的格式一致，并且可以进行评测。

此外，我们在预测和提交结果前，添加了保存和加载模型的步骤。首先，我们使用 `torch.save` 方法保存模型的状态字典（`state_dict`），路径为 `/content/drive/MyDrive/ColabNotebooks/results/heartbeat_classifier_model_state.pth`。然后，在进行预测时，使用 `torch.load` 方法加载保存的模型状态字典，并将模型设置为评估模式（`model.eval()`）。这种方法确保了模型的可复现性和易用性，使我们能够在不同时间和环境下加载并使用训练好的模型进行预测。

实验代码 Github Gist 地址: [链接](#)

2. 模型结构图



图十一：模型结构图

这个卷积神经网络 (CNN) 模型设计用于心跳信号分类任务。假设该模型的输入为长度为 50 的心跳信号样本。首先，信号通过一个一维卷积层，该层使用 64 个卷积核 (过滤器)，每个卷积核的大小为 3，并且在输入信号的两端添加填充以保持输出的长度不变。卷积操作后，经过 ReLU 激活函数，引入非线性特性以增强模型的表达能力。接下来，经过一个最大池化层，将信号的长度减半至 25，从而减少特征图的尺寸，缓解计算量并提取重要特征。

第二层卷积层进一步对信号进行特征提取，这次使用 128 个卷积核，同样经过 ReLU 激活函数和最大池化操作，将特征图的长度进一步减半至 12。第三层卷积层继续在更高维度的特征空间进行操作，使用 256 个卷积核，经过 ReLU 激活函数和最大池化操作后，最终将特征图的长度缩减至 6。通过这三个卷积层和池化层的堆叠，模型逐步提取心跳信号中的复杂特征，并有效地减少了特征图的尺寸。

在卷积和池化操作完成后，特征图被展平为一维向量，向量长度为 256×25 ，即 6400。这个展平后的向量被输入到一个全连接层中，将其维度降至 512，并通过 ReLU 激活函数引入非线性。最后，经过另一个全连接层，将 512 维的向量转换为 4 维，代表 4 个类别的预测输出。整体模型结构通过逐层提取和聚合特征，实现了对心跳信号的有效分类。这样的设计不仅考虑了特征提取的深度和复杂性，还通过池化操作有效控制了计算复杂度，为心跳信号分类任务提供了高效的解决方案。

3. 损失函数详细描述

在本次实验中，我们选择使用交叉熵损失函数 (Cross Entropy Loss) 作为模型的损失函数。交叉熵损失函数是分类问题中最常用的损失函数之一，特别适用于多类别分类任务。在我们的心跳信号分类任务中，模型需要将输入的心跳信号归类为 4 个可能的类别之一，因此交叉熵损失函数是一个合适的选择。

交叉熵损失函数衡量的是模型输出的概率分布与真实标签分布之间的差异。具体来说，交叉熵损失函数计算的是预测概率分布与实际标签分布之间的负对数似然。公式如下：

$$\text{Loss} = - \sum_{i=1}^N y_i \log(\hat{y}_i)$$

图十二：交叉熵损失函数公式

其中， y_i 是实际标签的指示函数（在实际类别的位置为 1，其余位置为 0）， \hat{y}_i 是模型预测的概率。通过最小化交叉熵损失，模型的预测概率分布会尽可能接近实际标签分布，从而提高分类准确率。

在训练过程中，我们采用 Adam 优化器对模型参数进行优化。Adam 优化器是一种自适应学习率优化算法，它结合了动量和 RMSProp 算法的优点，能够在大多数情况下提供较快的收敛速度和较好的性能。我们设定学习率为 0.001，这是一个常见的初始值，能够在实验中表现出较好的效果。

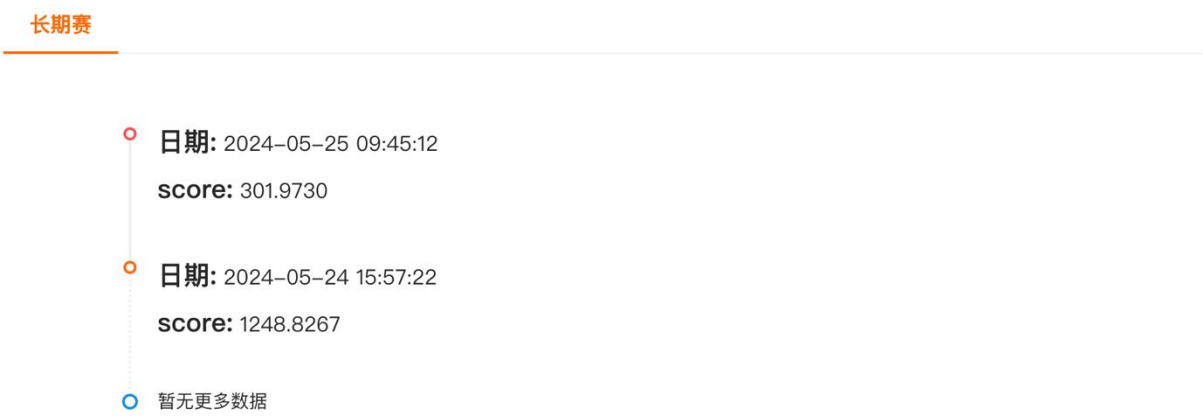
在每个训练周期（epoch）中，我们将模型设置为训练模式（`model.train()`），然后遍历整个训练数据集。对于每一个小批量数据（batch），我们将输入信号和标签加载到 GPU 上，并通过模型前向传播计算输出。接着，计算输出与真实标签之间的交叉熵损失，并通过反向传播计算梯度。最后，使用 Adam 优化器更新模型参数。

每个训练周期结束后，我们计算平均训练损失并记录下来，以监控训练过程中的损失变化情况。最终，打印每个周期的损失值，以便我们可以观察到模型在训练过程中的收敛情况。通过这种方式，我们可以评估模型的性能并进行必要的调整，以确保模型在心跳信号分类任务中的表现达到最佳。

4. 天池提交分数

跑 100 个 epoch 的评测分数:

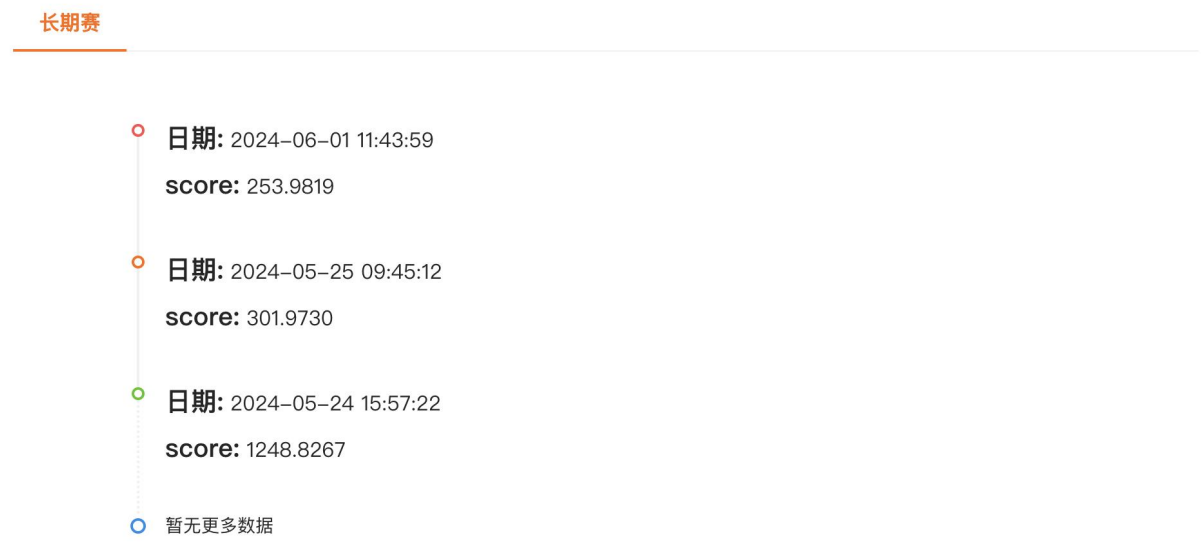
长期赛: 914/301.97



图十三: 天池提交分数 1

跑 200 个 epoch 的评测分数:

长期赛: 673/253.98

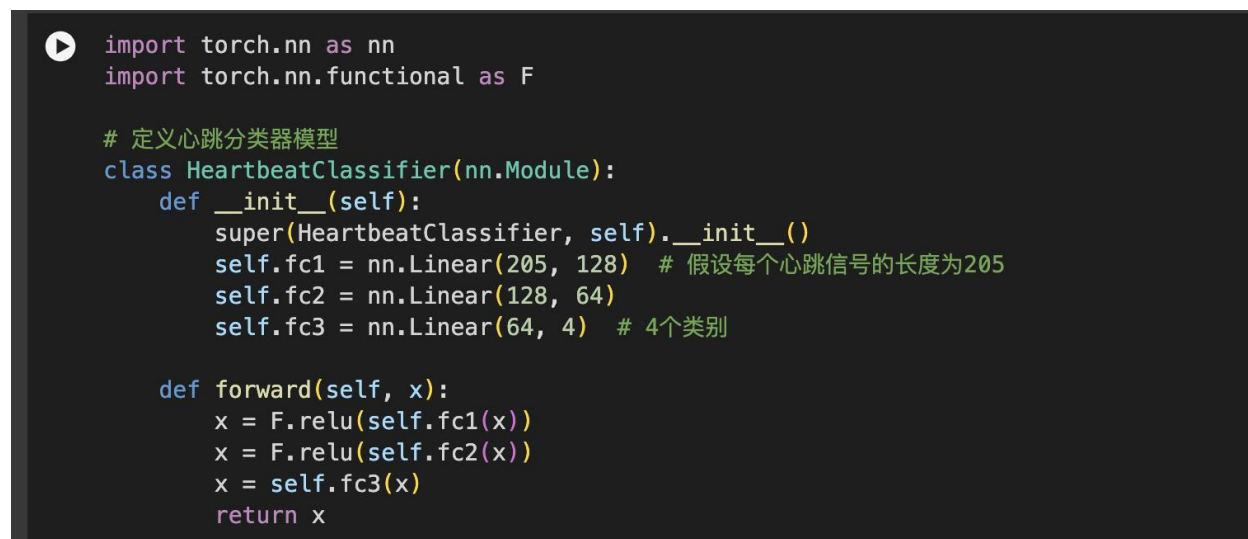


图十四: 天池提交分数 2

5. 实验总结

在本次实验中，我们使用了 Google Colab 作为实验环境。Colab 提供了一种便捷的方式来运行 Python 代码，并且预先安装了大多数常用的 Python 包，使得我们可以专注于模型的开发和调试。在数据加载和预处理过程中，我们首先将心跳信号数据集包装成自定义的 `HeartbeatDataset` 类，并通过 `DataLoader` 创建数据加载器。训练数据集通过标签编码处理为整数形式，并将心跳信号字符串转换为浮点数数组，从而方便后续的模式训练。

在模型结构设计中，我们尝试了两种不同的版本。第一个版本是一个简单的多层感知机 (MLP)，其结构包括三个全连接层。假设每个心跳信号的长度为 205，模型的第一层全连接层输入为 205，输出为 128。第二层全连接层输入为 128，输出为 64，最终输出层将 64 个神经元映射到 4 个类别。虽然这个版本的模型结构相对简单，但它在实验中能够快速进行迭代和调试。然而，这个模型的得分仅为 1200 多分，表明其性能在心跳信号分类任务中并不理想。



```
import torch.nn as nn
import torch.nn.functional as F

# 定义心跳分类器模型
class HeartbeatClassifier(nn.Module):
    def __init__(self):
        super(HeartbeatClassifier, self).__init__()
        self.fc1 = nn.Linear(205, 128) # 假设每个心跳信号的长度为205
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 4) # 4个类别

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

图十五：第一个版本定义的简单 MLP

为了改进模型性能，我们设计了第二个版本的模型，采用了卷积神经网络 (CNN) 结构。CNN 通过卷积层提取输入信号中的局部特征，能够更好地捕捉心跳信号中的重要模式。该模型包括三层一维卷积层和池化层，特征提取后，通过全连接层进行分类。每一

层卷积操作后，特征图的长度逐步减半，从而有效地减少了计算量。特别地，在最后一层卷积和池化操作后，我们得到的特征图尺寸为 $[\text{batch_size}, 256, 25]$ ，因此全连接层的输入尺寸为 $256 * 25$ 。这个设计在模型性能上有显著提升，跑 200 个 epoch，最终得分为 250 多分，证明了其在心跳信号分类任务中的有效性。

```
# 定义CNN模型
class HeartbeatClassifier(nn.Module):
    def __init__(self):
        super(HeartbeatClassifier, self).__init__()
        self.conv1 = nn.Conv1d(1, 64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv1d(64, 128, kernel_size=3, padding=1)
        self.conv3 = nn.Conv1d(128, 256, kernel_size=3, padding=1)
        self.pool = nn.MaxPool1d(2)
        self.fc1 = nn.Linear(256 * 25, 512)
        self.fc2 = nn.Linear(512, 4)

    def forward(self, x):
        x = x.unsqueeze(1) # Add channel dimension
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

图十六: 第二个版本定义的CNN模型

此外，为了确保模型的可复现性和易用性，我们在第二个版本中添加了保存和加载模型状态的步骤。通过使用 `torch.save` 和 `torch.load` 方法，我们可以在不同时间和环境中加载训练好的模型，并直接进行预测。这种方法不仅提高了实验的效率，也确保了模型在实际应用中的灵活性。通过这些优化和改进，我们在实验中得到了更好的结果，同时积累了宝贵的经验，为后续的研究和开发提供了坚实的基础。