JUNO 能量重建实验报告

Team: PMTMender

Members: 刘明昊 沈若寒 赵海萌

小组主要分工：

- 刘明昊：Ghost Hunter Legacy，特征工程，主体架构
- 沈若寒：特征工程，性能优化
- 赵海萌：Ghost Hunter Legacy，机器学习，撰写报告

评测平台上的mhliu0001，hmzhao均为本小组成员。

注意！ 本报告仅是一个简短的说明文档和操作手册，详细实现思路与细节可以参见`model.ipynb, train.ipynb, final.ipynb`中的Markdown说明。

摘要

本项目以JUNO中微子探测装置为背景，试图通过物理分析、信号处理与机器学习等技术，从PMT波形中重建出中微子事件的能量。通过数据预处理、特征工程和多级LightGBM的统计学习，我们的算法能在约1小时的时间内，以很高的精度重构出事件的能量，同时具备很好的可解释性。高效率、高精度、可解释的能量重建手段有助于我们理解中微子质量顺序的难题。

目录

文件结构

本项目的文件结构如下：

```
|-- project-1-junosap-pmtmender
    |-- README.md
    |-- requirements.txt
    |-- Makefile
    |-- utils.py
    |-- waveform.py
    |-- fetch.sh
    |-- prompt.sh
    |-- train.ipynb
    |-- final.ipynb
    |-- model.ipynb
```

其中`README.md`为本实验报告，`requirements.txt`罗列了本项目的依赖包版本，`Makefile`定义了处理流水线，`waveform.py`进行数据预处理（所需时间较长，在我们的服务器上需要50分钟左右），`train.ipynb`训练模型，`final.ipynb`生成预测答案。`utils.py`中定义了一些常用函数，`fetch.sh`用于下载数据集，`prompt.sh`用于指定答案的文件名以及log。

注意：`model.ipynb`是一个示例代码，使用一个训练集来训练决策树，可以在有data的前提下单独运行，不需要数据预处理。阅读它可以使读者更好地了解我们的算法。

执行方式

注意！本项目对内存的需求较大，推荐在96GB以上内存的服务器上运行。

我们的运行环境是：

系统：Ubuntu 18.04，5.4.0-80-generic

CPU：Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz

内存；128GB

在执行前请确保依赖包都已安装并符合版本要求，执行

```
pip install -r requirements.txt
```

以安装依赖包。

在项目目录下用`shell`执行代码

```
make
```

可以完整地执行整个项目的流程，下载训练集、预处理数据、训练模型、生成预测答案。

注意：`make`到最终生成答案的阶段，需要手动输入答案的序号以及一个简短的log文字。答案的序号必须是正整数，不能与已有的序号重复。

执行代码

```
make data
make train
make model
make ans
```

可分别下载数据、预处理数据、训练模型、生成预测答案。预测答案位于`./ans`中。

执行代码

```
make clean
```

可以清理预处理数据和训练模型，但不会清理下载的数据和预测答案。

若要单独执行预处理数据，可以执行

```
python3 waveform.py
```

若要单独训练模型和得出答案，可以执行`train.ipynb`和`final.ipynb`。

`model.ipynb`也可以单独运行，不需要运行耗时较长的`waveform.py`，便可以得到决策树。再运行`final.ipynb`（需要将一个代码块换成另一个代码块，见中间的注释），也可以得到答案。

整体思路

我们面临的问题是要从波形中重构出事件能量。考虑到我们拥有大量的训练数据，这是一个典型的机器学习回归问题。其中的困难主要在于**特征工程**：即如何从波形中提取出有效的特征，用于训练预测能量的模型。

为了保证较高的效率、可复现性与可解释性，避免过大的计算资源需求，我们没有采取以BERT为首的一系列深度神经网络，而是采用了Kaggle等数据科学竞赛中常用的传统机器学习算法LightGBM，其具有**效率高、迭代快、效果好**的特点，方便我们进行调整并测试不同的特征工程方法与超参数。

那么余下的问题便是特征工程，如何从巨量数据的波形中提取出有效的特征。考虑到击中每个波形的PE个数以及PETime是一个重要的中间量，我们首先将整个波形-能量的任务拆分成两步：波形->每个波形的PE个数和PETime平均值->能量。

对于第一步，我们设计了手作算法。对于第二步，我们训练了LightGBM决策树，利用PE个数总数、平均值、标准差，以及PETime平均值、标准差这五个feature，得到最终的能量。

手作算法

在使用手作算法之前，我们先去除没有PE处的暗噪声。取918为阈值，对于小于918的信号用918去减，对于大于等于918的信号置0。

手作算法的主要实现位于`utils.py`中的`getPePerWF`函数。基本思路是对波形取`np.argmax`来寻找PE的位置，然后减去这个PE产生的效果，再去除噪声。重复上述过程，直到波形的积分值足够小。

其中有三点需要解释：

1. **PE产生的效果**：这里取的是以argmax为峰的横坐标，18为纵坐标，宽度为16的三角脉冲波。这是一个很简单的处理，如果能知道具体的PMT对于单光子的响应函数，便能够取得更好的效果。
2. **去除噪声**：指在原波形减去三角脉冲波后，去除三角脉冲波不为0处的噪声。将8作为阈值，小于8的信号均视为噪声，置为0。不仅要考虑这一次减去的三角脉冲波不为0的区域，还要考虑之前减去的三角脉冲波不为0的区域。
3. **波形的积分值足够小**：指积分值小于
   $150-8\max\{$波形纵坐标超过0的点的数量，$16-$波形纵坐标超过0的点的数量$\}$。这样的原因是，波形纵坐标超过0的点很有可能因为噪声的影响，纵坐标变小；波形可能并非完整的波形，PE产生的脉冲波的两侧的点可能在除噪声的过程中被置0了。这样的判断条件可以同时考虑到两者的影响。

我们还去除了部分的暗噪声，方法是判断argmax是否大于600或小于150。我们观察到，PETime过大或过小的基本上都是暗噪声，因此使用这种方法来去除。

在没有判断暗噪声的情况下，用手作算法处理final-2.h5，PE总数的准确率达到了89%。作图发现，真实的PE总数（未去除暗噪声）与计算得到的PE总数（去除了暗噪声）大致在一条直线上。这均能证明手作算法的有效性。

为了数据的对齐，对于一个波形，将所有的argmax取平均值，作为`meanPETime`输出。

手作算法具体的实现细节，可以参见`model.ipynb`中的描述，以及`utils.py`中的源代码。

特征工程

我们使用了五个特征：对于每个事件，再对每个被触发的PMT，取PE个数总数、平均值、标准差，以及PETime平均值（是手作算法得到的argmax）平均值、标准差。

PE个数总数与能量直接相关，但能量与PE总数的关系又与事件的位置有关系。当事件位于边缘时，产生的光子更有可能被PMT接受，因此PE总数与能量的比值会变大。

假设PMT的分布是均匀的，通过PE个数平均值、标准差、PETime平均值、标准差四个参数，可以反映出事件位于边缘的程度。事件越靠近边缘，在PE总数不变的情况下，这四个参数分别会变大、变大、变小、变大。因此，这五个参数有潜力推断出能量。

我们采用LightGBM训练决策树，使用全部的训练集，取10%的数据作为验证集，取较小的Bagging fraction，max_depth以防止过拟合。最终的训练结果与真实结果的散点图可以在`train.ipynb`中画出。可以看到，除了少数离群点外，其余的点均大致处于一条直线上。

优化方法

训练过程中的速度瓶颈主要在于手作算法`utils.getPePerWF`，最初在一个数据集上运行需要40分钟。根据我们在一阶段的经验，这里有相当大的优化空间。

Numpy并行化

最初的算法只能一个一个地处理波形，产生的大量的循环。我们的核心目的就是同时处理多个波形，来减少循环的数量。

对于一批次中的不同波形，它们可能具有非常不同的特征。比如，大部分的波形都只含有小于5个的PE，而少部分波形能含有大几十个的PE。如果我们以完全一样的方式来处理这些波形，就会造成大量的计算浪费。因此，即使我们同时处理它们，也需要在过程中动态地进行数据的舍去。

我们的解决方法是通过创建数组`label`来控制还有哪些波形需要计算。每次循环都会更新`label`，然后程序根据`label`留下还需要计算的数据，并通过`label`写入那些累积变量。这就要求`label`是一个`int`数组而非`bool`数组，因为它不仅需要表示一次循环中的舍去，还需要表达数据在最原始输入中的位置。

多进程并行化

即使`numpy`化大大加快了速度，但程序只能使用单核。程序多为判断与求和，`numpy`中的这些函数没有多核优化，`numexpr`也难以加速。程序还是顺序无关的。因此，多进程就是合理又必要的加速手段。

我们将数据集先用`np.array_split`分割成小块，然后用`utils.getPePerWF`处理这些小块，最后将这些小块通过`np.concatenate`完成输出。

我们试图通过调节块数量以及进程数来进一步优化性能。结果发现块数量对性能没有明显影响；进程数大于4后每个核心的利用率就到不了100%，说明内存性能开始成为瓶颈，但继续增加进程数性能却还有微小的提升。

最终，处理一个数据集仅需34s。

多进程读取

由于hdf5文件在读取时需要解压缩，而这个解压缩只能单核运行，导致读取时无法充分利用硬盘。我们将文件分为小块并读取，虽然未能实现与进程数线性的提升，但依然提速了约50%。