Санкт-Петербургский государственный университет

Никольская Анастасия Николаевна

Выпускная квалификационная работа

Разработка системы локального трекинга в коллайдерных экспериментах с применением методов глубокого обучения

Уровень образования: магистратура

Направление 01.03.02 «Фундаментальная информатика и информационные технологии»

Основная образовательная программа «Распределенные вычислительные технологии»

Научный руководитель,
Доктор технических наук,
профессор
Дегтярёв А.Б.
Рецензент:Стрельцова О.И.
Консультант: Ососков Г.А.

Санкт-Петербург

2020

Содержание

Введение

В современном мире всё чаще возникают исследовательские задачи, требующие массивного использования экспериментальной и вычислительной техники. Такие задачи производят огромные объемы данных, которые необходимо правильно обрабатывать и интерпретировать. В результате, работа с большим данными играет одну из ключевых ролей в современных исследованиях, поэтому разработка быстрых и точных систем обработки информации становится всё более актуальной. Так, существующие на сегодняшний день эксперименты в области физики высоких энергий, производят гигантские потоки информации, достигшие уже экзабайтного уровня, и поэтому требуют специальных компьютерных и сетевых систем для распределенного сбора, фильтрации и обработки данных [1].

Один из методов наблюдения и изучения внутренней структуры вещества на самом низком уровне заключается в ускорении частиц вещества с последующим соударением с помощью специальных установок – коллайдеров, или ускорителей частиц. Современная физика достигла значительных успехов с помощью высокоэнергетических экспериментов, выполненных на ускорителях и коллайдерах, таких как HERA, LEP, SLC, Tevatron и др. Одним из самых известных коллайдеров является Большой адронный коллайдер (БАК), известность которому принес впечатляющий масштаб проводимых экспериментов , особенно эксперименты CMS и ATLAS, результаты которых по обнаружению бозона Хиггса были удостоины Нобелевской премии за 2012 год [2].

Другая немаловажная область исследований, кроме адрон-адронных и электрон-позитронных столкновений, – это изучение процессов столкновений ядер тяжелых ионов. В настоящее время в России строится ускорительный комплекс NICA, проект которого входит в список наиболее приоритетных для российской науки. Одна из главных задач – исследование кварк-глюонной

плазмы, особого состояния вещества, характерного для ранних этапов развития вселенной.

Для исследования физических свойств материи необходимо выполнить и зафиксировать событие по взаимодействию тяжелых ионов при их столкновению друг с другом в коллайдерных экспериментах, где ускоряются встречные пучки ионов, или по их столкновению с неподвижной мишенью в экспериментах с фиксированной мишенью. После этого ключевой задачей становится «расшифровка » данных , зарегистрированных в эксперименте, что называется реконструкцией событий. Эта задача состоит в проведении процедуры восстановления траекторий, или треков, элементарных частиц в трековых детекторах, в которых и происходит регистрация событий. Однако такое восстановление сопряжено со многими трудностями, связанными с данными: высокая сложность структуры распознаваемых образов, число фоновых событий на несколько порядков выше числа полезных, высокий уровень шума. Кроме того, важную роль могут играть несовершенство некоторых типов детекторов и неоднородность магнитного поля. В современных экспериментах, характеризуемых высочайшими темпами поступления данных, классические методы трекинга, такие как фильтрация по Калману, не могут удовлетворить требованиям исследователей, предъявляемых к скорости работы алгоритма, в виду плохой масштабируемости исходного метода, несмотря на высокие показатели эффективности работы фильтра. Т.е. фильтр способен выполнить процедуру восстановления события с высокой точностью, но темпы поступления новых данных так высоки, что алгоритм не способен обработать все это за разумное время, а сохранить все данные в сыром формате не представляется возможным.

Таким образом, современная экспериментальная физика высоких энергий требует не только гигантских комплексов для распределённых вычислений и соответствующей инфраструктуры, но и разработки новых

подходов для эффективной реконструкции событий в рамках этих комплексов. Сюда относятся методы машинного обучения для поиска закономерностей в данных, прогнозов и фильтрации. Наиболее эффективными инструментами в данном случае являются глубокие нейронные сети из-за высокой способности обобщения, обучения и самообучения.

Модели глубоких нейронных сетей используются для решения проблемы распознавания треков для локального (трек за треком) и глобального (все треки в событии одновременно) распознавания. Другое преимущество глубоких нейронных сетей - способность обнаруживать скрытые нелинейные зависимости в данных. Кроме того, как будет показано далее, одну и ту же нейросетевую модель можно использовать для различных экспериментов без существенных изменений.

Постановка задачи

Целью работы является разработка нейронной сети и набора программных средств для обучения и тестирования модели, обеспечивающей предсказание траектории частицы в экспериментах BESIII и BM@N.

Для достижения поставленной цели необходимо:

1) Изучить предметную область
2) Разработать и реализовать методы предварительной обработки данных
3) Предобработать данные
4) Выбрать архитектуру модели и параметры обучения
5) Выполнить тренировку выбранной модели
6) Проанализировать результаты

Объектом исследования являются данные Монте-Карло моделирования, симулирующие движение частиц в трековых детекторах на основе Газовых электронных Умножителей (ГЭУ) со стриповым съемом информации в экспериментах BESIII и BM@N.

Предметом исследования является реконструкция треков элементарных частиц в смоделированных событиях.

Глава 1. Трековые детекторы

При анализе данных эксперимента в физике высоких энергий необходимо с максимальной возможной точностью определить различные кинетические характеристики частиц, включающие импульс, направление движения, кривизну движения и т.д. при их столкновении. Для определения этих характеристик используются специальные детекторы. Ускоренные заряженные частицы при взаимодействиях с мишенью или другой частицей производят множество вторичных частиц, поэтому важной задачей при восстановлении события такого взаимодействия является распознавание траекторий – треков вторичных частиц с целью последующего определения их физических характеристик, - импульсов, зарядов и углов разлета их точки взаимодействия, называемой вершиной.

Таким образом, практически в любом эксперименте физики высоких энергий важнейшей частью установки являются трековые детекторы, позволяющие определить координаты прохождения частицы сквозь пространство детекторов и восстановить траекторию ее движения.. Алгоритм восстановления трека называют трекингом.

Координатное разрешение трековых детекторов оказывается непосредственно связанным с диапазонами импульсов, поддерживаемыми установками, что является одной из важнейших характеристик как частиц, так и самих экспериментов. Другим параметром, влияющим на импульсное разрешение, является количество точек в пространстве, по которым восстанавливается трек. Далее эти точки мы будем называть хитами (от англ. hit — удар, попадание). Количество хитов также определяется конструкцией детектора и количеством чувствительных элементов на пути частицы.

Для восстановления трека необходимо определить, какие срабатывания элементов детектора (далее - станций) относятся к одной и той же частице, определить координаты соответствующих точек и затем аппроксимировать их

заданной моделью трека. При этом модели треков могут быть довольно простыми, так, в случае однородного магнитного поля они описываются спиралями или окружностями. Однако на практике необходимо учитывать различные эффекты, связанные с неоднородным магнитным полем, рассеянием и ионизационными потерями энергии частиц во время движения и т.д., в результате чего модель существенно усложняется.

Объектом исследования в данной работе является информация, полученная с основных трековых детекторов в экспериментах BESIII и BM@N – ГЭУ (англ. *GEM*) детекторов.

ГЭУ детектор представляет собой набор камер, наполненных газовой смесью и в общем виде работает как стандартный газовый детектор [3].

Принцип работы такого детектора состоит из четырёх этапов:

1. Частица, пролетая через объем камеры, сталкивается с атомами вещества. В результате каждого столкновения производятся электроны (происходит электронно-газовая ионизация)
2. Под действием электрического поля внутри камеры электрон перемещается к чувствительной плоскости.
3. Последовательно проходя через усиливающие ГЭУ каскады, каждый электрон порождает большое число новых электронов («электронная лавина»), которые также движутся к чувствительной плоскости и этим усиливают сигнал.

Существует два типа съема данных с GEM детекторов, –стриповый и пэдовый. Более распространенный тип - с микростриповым съемом информации - показан на рис. 1. Наведенный от лавин электронов сигнал фиксируется двумя слоями стрипов на считывающей плоскости. Засвеченные стрипы представляют собой стриповый кластер.

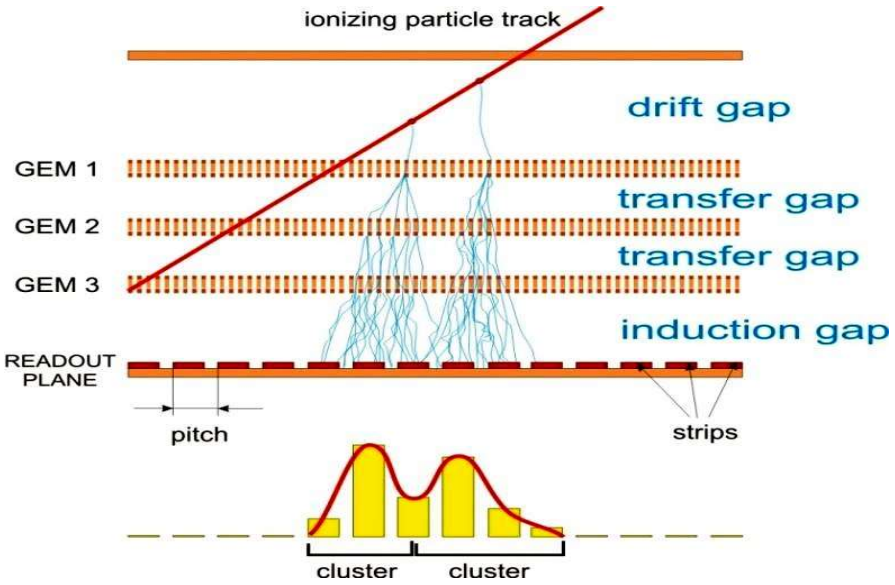

Рис.1. - Схема ГЭУ, показаны лавины электронов и стриповые кластеры

После того, как электронные лавины для каждого трека формируют энергетические кластеры, вычисляется локальный максимум для каждого события и выбираются те стрипы, через которые предположительно прошла частица. Для идентификации координаты точки прохождения элементарной частицы в *GEM* детекторе каждая станция имеет 2 стриповых слоя – прямые и наклонные стрипы.

Главный недостаток такой конструкции считывающей плоскости – наличие большого количества ложных срабатываний, которые дальше будут называться фейками (от англ. fake) в случае, если лавин больше одной, и общее число фейков может достигать квадрата числа реальных пересечений.

Этого недостатка лишен второй способ съема информации, - пэдовый, - двухкоординатный, когда в станциях детектора используется только одна координатная плоскость с детектирующими элементами на ней в виде маленьких прямоугольников – пэдов (англ. pad - площадка). Лавина от проходящей частицы в зависимости от угла её прохождения активирует один или несколько пэдов, так что их центр тяжести дает сразу две координаты. Пэдовый способ удобен, но значительно дороже стрипового и, главное, вывод

информации из тысяч педов осуществляется с помощью проводников, что вносит слишком много нежелательного вещества в рабочий объем детектора.

Поэтому далее мы будем рассматривать только GEM детекторы со стриповым съемом информации.

Типичный пример события на этом детекторе можно видеть на рис. 4. Задача трекинга сводится к тому, чтобы по набору точек, составляющему как шумовую (фиктивные хиты), так и смысловую (истинные хиты) составляющие, определить подлинные траектории частиц. Пример трекинга можно видеть на рис.2.

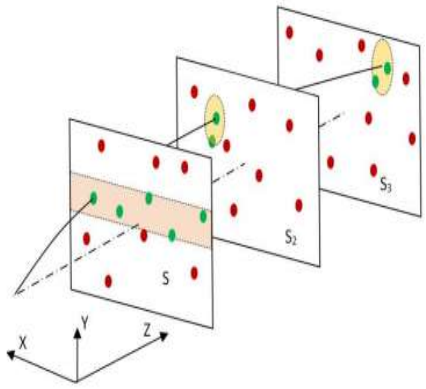

Рис. 2 - Схема поиска кандидатов с помощью KD дерева

Как правило, стриповые слои располагают относительно друг друга таким образом, чтобы угол между стрипами был достаточно малым, что позволяет избавиться от части фейков, которые при этом выносится за границы рассматриваемой области. Несмотря на это, число фиктивных пересечений все равно остается достаточно большим – Для n истинных хитов – $n^2 - n$ фейков. На рисунке 3 представлен процесс возникновения фейков и то, как добавление стерео-угла позволяет вынести их за границы чувствительных областей детектора.

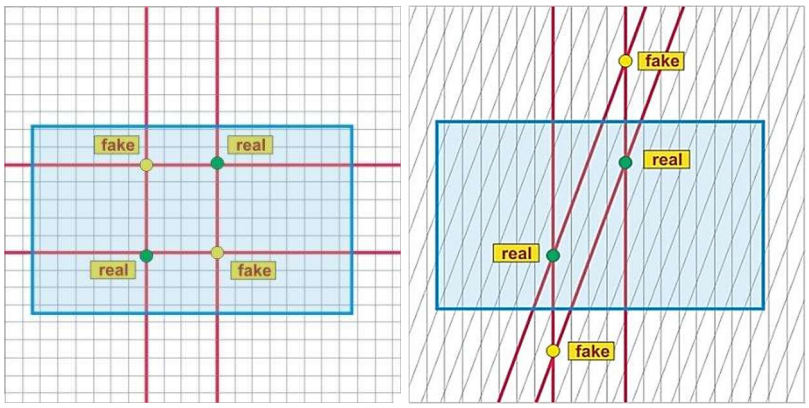

Рисунок 3. – Процесс рождения ложных стриповых пересечений – фейков.

Классические алгоритмы, способные решить комбинаторную задачу такой сложности, плохо подвергаются параллелизации, что ведет к серьезному кризису таких алгоритмов для трекинга и интересу исследователей к разработке новых систем.

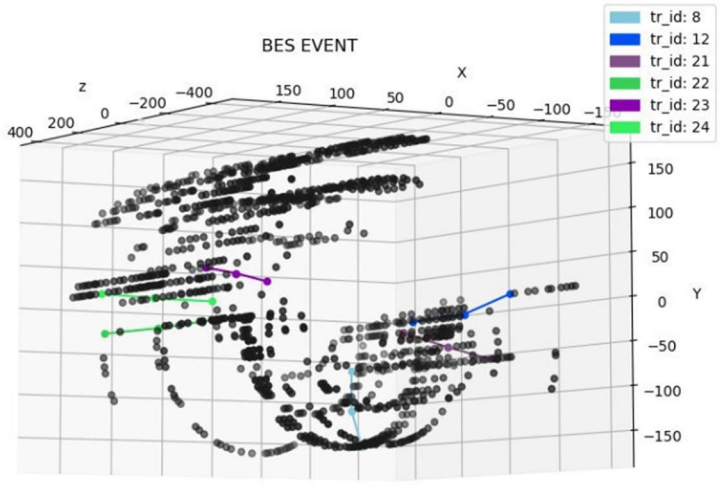

Рис.4 - Пример события BESIII. Черные точки - подделки, цветные - отслеживают попадания со связями треков

На данный момент восстановления совпадений CGEM использовалось моделирование методом Монте-Карло BESIII CGEM с последующим алгоритмом кластеризации.

1.1. Эксперимент BESIII

BESIII – это эксперимент в области физики частиц, проводящийся в Институте Физики Высоких Энергий на базе коллайдера BEPC-II в Пейджинге

и проводимый с 2018 и до 2022 года [4]. Задача эксперимента в том, чтобы пролить свет на природу взаимодействий частиц в стандартной модели, описывающей сильные и слабые взаимодействия [5] с помощью изучения свойств тау-лептонов, очарованных частиц и состояний чармония, которые образуются в электрон-позитронных столкновениях.

В результате взаимодействия в каждом событии, как правило, образуется до 20 заряженных треков, однако большая часть событий содержит 2-8 треков. Восстановление треков заряженных частиц играет важную роль в программе эксперимента [6]. В детекторе BESIII они восстанавливаются в дрейфовой камере (см. рис. 5). При проходе заряженной частицы через рабочий объем детектора ионизируется заполняющая его газовая смесь. Электроны под действием внешнего электрического поля дрейфуют к считывающей плоскости и проходят через три каскада газового усиления в GEM-пленках. В результате на считывающих электродах наводится электрический импульс, который усиливается и регистрируется. Считывание осуществляется двумя слоями микрополосок (стрипов), расположенных на считывающих плоскостях под углами в 30 и 45 градусов с шагом 650 микрон.

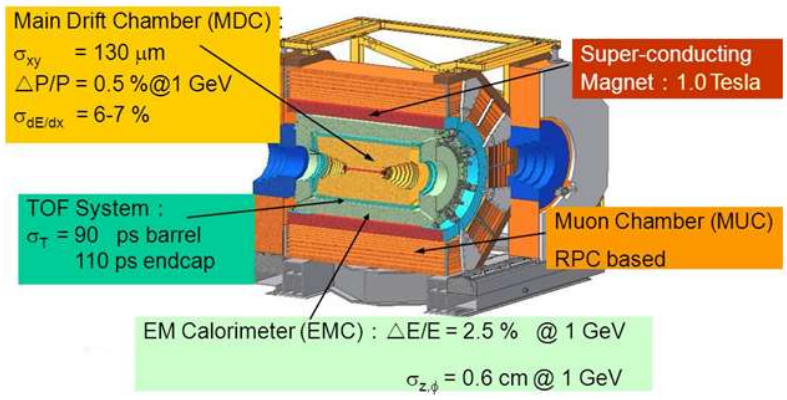

Рис. 5. - Схема детекторного комплекса эксперимента BESIII

Одна частица может формировать сигнал одновременно на нескольких расположенных рядом стрипах, поэтому на первом этапе трекинга восстанавливаются кластеры сработавших стрипов в каждом слое. Координата

кластера может восстанавливаться из сработавших стрипов как средняя координата (бинарный режим), средневзвешенная с зарядом, а также может определяться по времени их срабатывания. На втором этапе хиты восстанавливаются комбинаторно. В данной работе используется кластеризация в бинарном режиме [7].

Как уже было сказано, стриповая конструкция считывающей плоскости приводит к появлению большого количества фейков, если число треков больше одного. В общем случае число ложных пересечений пропорционально квадрату числа треков.

CGEM-IT – внутренний детекторный комплекс эксперимента BESIII типа ГЭУ, состоит из трех вложенных цилиндров [8] (см. рис. 6). При этом, в отличие от предыдущих детекторов данной конструкции [1], ГЭУ и электродные пластины формируют цилиндры только на этапе сборки.

На рисунке 3 изображена схема всего комплекса эксперимента.

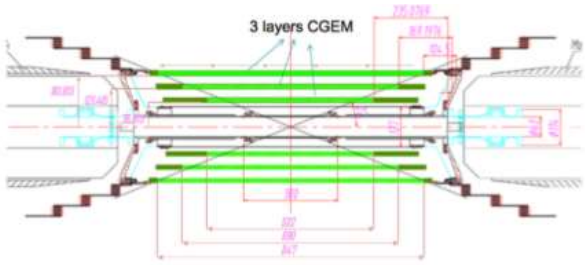

Рисунок 6. – Схема главного трекового детектора в эксперименте BESIII (измерения в мм)

1.2. Эксперимент BM@N мегапроекта NICA

NICA (Nuclotron based Ion Collider fAcility) – новый ускорительный комплекс, который строится на базе Объединённого института ядерных исследований (Дубна, Россия) для исследования свойств плотной барионной материи, воссоздания кварк-глюонной плазмы – особого состояния вещества, свойственного раннему периоду существования вселенной.

BM@N – эксперимент проекта с фиксированной мишенью с выведенным из нуклотрона пучком частиц для изучения свойств барионной материи, которая образуется при столкновении тяжелых ионов при энергиях пучка от 2 до 6 A·GEV. Детекторный комплекс эксперимента включает в себя различные подсистемы: внутреннюю, внешнюю трековые системы, системы определения параметров столкновения, триггерные счетчики и системы идентификации частиц. В данной работе рассматривается информация, получаемая с одной из систем - основного трекового детектора в эксперименте – ГЭУ-детектора. На рисунке 7 изображена схема всего комплекса эксперимента.

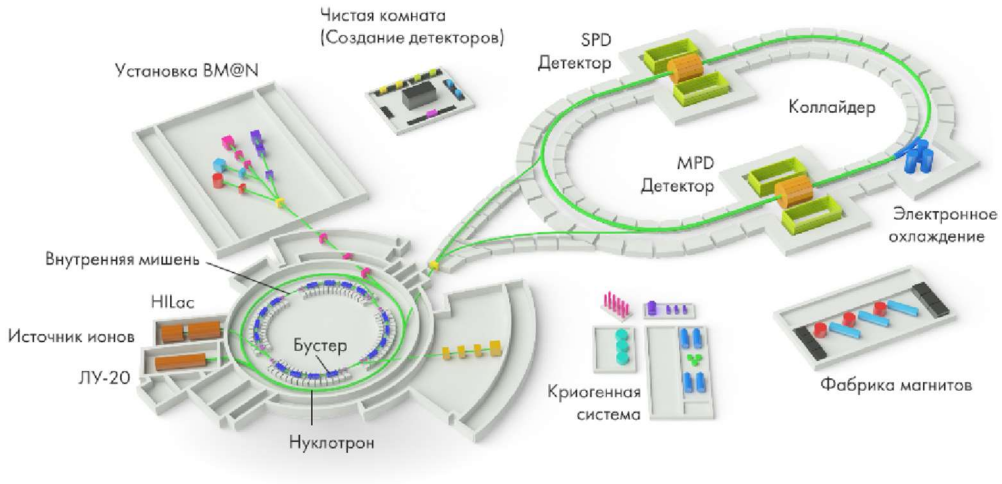

Рис. 7. - Схема мегапроекта NICA

BM@N ГЭУ детектор состоит из набора последовательных камер, наполненных газовой смесью ArCO2 (70/30), также с микростриповым съемом информации. Всего станций 6, и в отличие от эксперимента BESIII, они расположены последовательно, а не вложены друг в друга (см. рис.8).

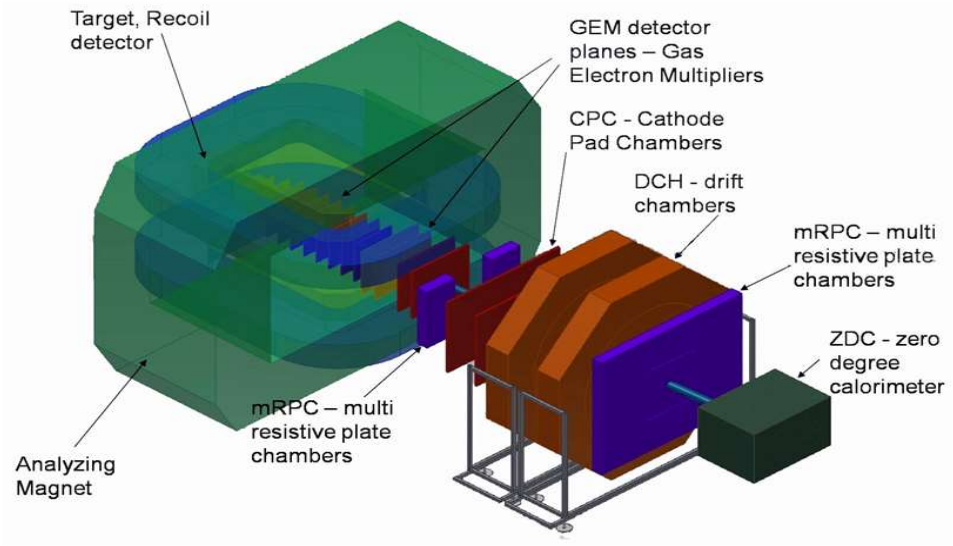

Рис. 8. - ГЭУ-детектор BM@N

Таким образом, главным отличием эксперимента BESIII от эксперимента BM@N является то, что регистрация частиц производится в 4π-геометрии, в результате чего регистрируются практически все частицы, участвующие в событии. В эксперименте BM@N же треки регистрируются только в узком конусе по направлению исходного движения пучка, сталкивающегося с мишенью перед чувствительными плоскостями (рис. 9).

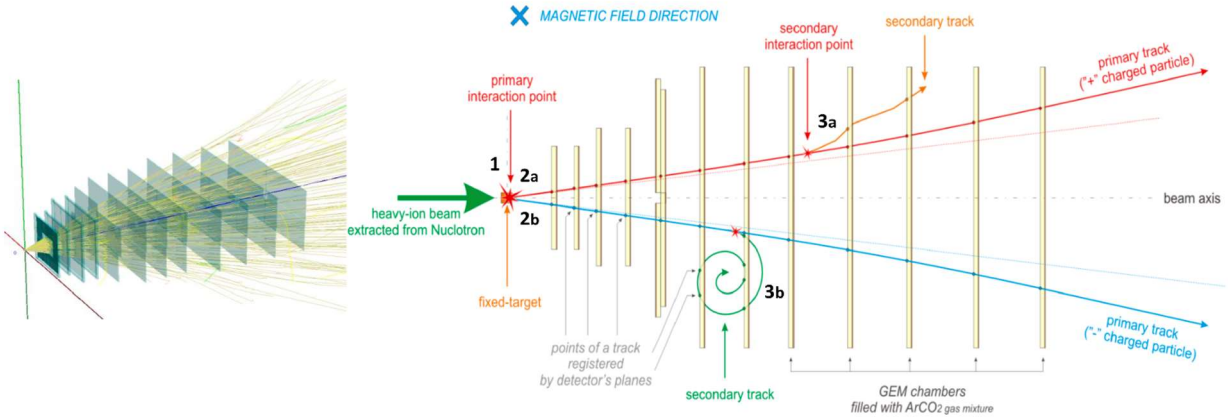

Рис.9. - Схема регистрации события

Пример события этого эксперимента можно видеть на рис.10. Как можно увидеть, в событиях также много шумового сигнала, при этом на первых трёх

станциях уровень шума выше, так как используются кремниевые считывающие плоскости – у них выше разрешение, но и ложных срабатываний тоже больше.

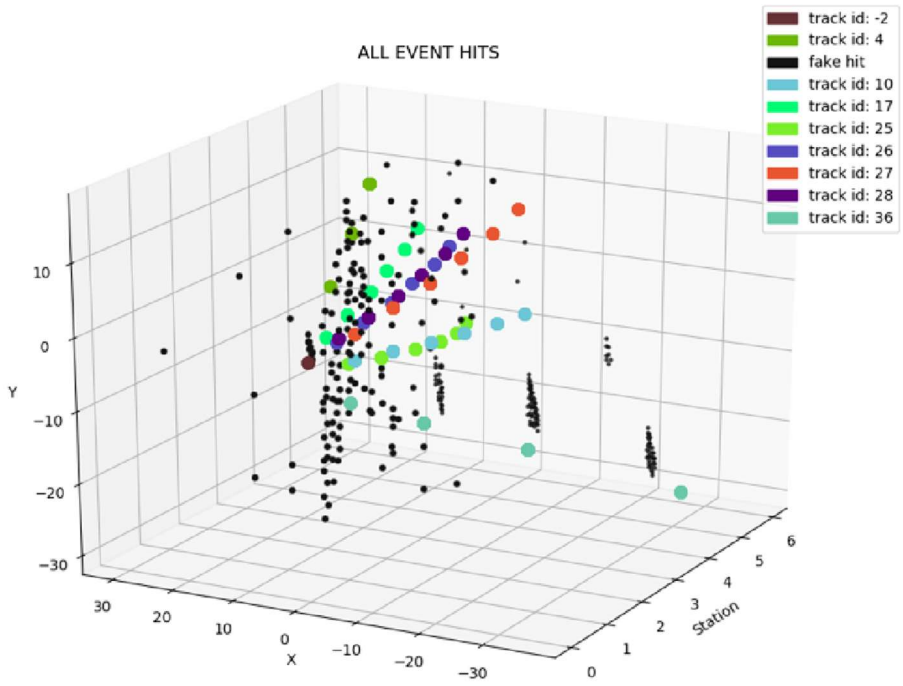

Рис. 10. Пример события BM@N. Чёрным отмечены фейковые хиты, цветным – хиты различных треков

Глава 2. Обзор литературы

Трекинг или распознавание треков - это процесс восстановления траекторий частиц в детекторе ФВЭ путем прослеживания и соединения точек-хитов (хит – это реконструированный отклик детектора), которые каждая частица оставляет, проходя через плоскости детектора. Процедура трекинга включает в себя фазы:

- сидинга (от англ. seed - зерно) – процесс формирования начальных состояний треков, определяющих направление или начальную траекторию кандидата в треки (трека-кандидата);

- построения треков, реконструкция треков – это процесс кластеризации хитов исходного события по признаку принадлежности к определенной траектории;

- подгонки – это процесс определения физических параметров исходной траектории частицы, необходимых для её идентификации;

- отбора треков – этот процесс выполняется уже после после построения всех кандидатов в треки и предназначен для того, чтобы отсеять шумовые треки. Как правило, для этого применяется критерий Хи-квадрат.

Проблема восстановления траекторий частиц имеет практически вековую историю, Началась она еще в эпоху пузырьковых камер, когда события регистрировались на стереофотографиях и вводились в компьютер вручную, полуавтоматами или с помощью сканирующих устройств типа «Спиральный измеритель», в котором оператор ставил точку в вершину события, откуда шло сканирование снимка по спирали.

Когда пришла эра электронных экспериментов, данные измерений стали оцифровываться и сразу поступать прямо в компьютер. После многоэтапной фильтрации и процедур алайнмента, наступало время трекинга.

В разное время использовались различные подходы, такие как:

1) Конформные отображения
2) Преобразования Хафа
3) Методы отслеживания трека
4) Метод наименьших квадратов
5) Фильтр Калмана
6) Нейросетевые подходы

Рассмотрим эти подходы.

2.1. Метод конформного отображения

Рассмотрим пример, характерный для детекторов с фиксированной мишенью и однородным магнитным полем, когда трек в горизонтальной проекции имеет вид окружности. На основе предположения о таком виде трека разработан метод конформного отображения. В данном методе окружности отображаются в прямые в координатах $u - v$ по формуле:

$$u = \frac{x}{x^2 + y^2}, \qquad v = \frac{y}{x^2 + y^2},$$

где окружности определяются уравнением окружности $(x - a)^2 + (y - b)^2 = r^2 = a^2 + b^2$. Прямые в плоскости *u*-*v* определяются следующим уравнением:

$$v = \frac{1}{2b} - u\frac{a}{b}.$$

Для больших значений *r*, то есть для треков с высоким значением импульса, прямые линии проходят близко к началу координат, и трек-кандидаты могут быть получены с помощью преобразования измерений из плоскости $u - v$ в полярную систему координат. Для того, чтобы отобрать треки из множества кандидатов, строится гистограмма распределения по угловой координате и выбираются пики в этой гистограмме [9].

Однако данный подход применим только при наличии однородного магнитного поля, что не позволяет использовать его во многих экспериментах.

2.2. Преобразование Хафа

Преобразование Хафа [8] позволяет покрыть более общее множество треков-кандидатов по сравнением с конформным отображением. Его отличие заключается в том, что оно применимо не только для прямых, проходящих вблизи начала координат, например при отсутствии магнитного поля или после "выпрямления" координат.

Это преобразование основано на уравнении прямой линии в плоскости *x-y*, где $y = cx + d$, и его преобразовании в пространство параметров трека частицы (далее пространство параметров), но уже в плоскости *c-d*, $d = -xc + y$. В новом пространстве точки, принадлежащие линии, соответствуют всем возможным прямым, проходящим через точку (*x, y*) в плоскости *x-y*. Поэтому точки, лежащие вдоль прямой в плоскости $x - y$, порождают линии в c-d, пересекающиеся в некоторой точке, которая определяет параметры прямой в исходном пространстве. В результате, при построении гистограммы в новом пространстве можно определить параметры прямой, проходящей через точки в исходном пространстве.

Как можно заметить, отличие конформного отображения от метода Хафа в том, что в последнем случае получаемое пространство двумерно [9]. Однако в результате теряется эффективность при попытке перейти в пространство параметров с большей размерностью.

2.3. Метод прослеживания по дорожке

Одним из методов локального трекинга является метод прослеживания по дорожке (track road). Первый его этап – сбор измерений, возможно, принадлежащих одной заряженной частице. При этом используется интерполяция, построенная с помощью модели трека, например, формы траектории. Интерполяция позволяет создать «коридор» - область, точки внутри которой формируют треки-кандидаты. Для оценки правильности гипотезы треков используются количество точек и качество посадки треков.

Еще один вариант метода заключается в последовательной экстраполяции треков-кандидатов по слоям детектора, начиная с так называемых сидов (“seed”). Сиды представляют собой короткие отрезки треков, сконструированные по некоторым правилам. Так, они могут быть сконструированы вблизи области взаимодействия, где измерения имеют высокую точность, или во внешней, так как там ниже плотность измерений. При экстраполяции используется просто ближайшая к прогнозируемому треку точка.

2.4. Подгонка треков методом наименьших квадратов

Кроме построения трека-кандидата необходимо провести его подгонку (“fitting”). Этот этап позволяет оценить параметры трека-кандидата, соответствующие кинематическим характеристикам частицы, на основе информации, содержащейся в различных измерениях его хитов. Поскольку положения частицы на разных станциях детектора является стохастической величиной, оценка также представляет собой статистическую процедуру и позволяет получить оценку неопределенности параметров с помощью ковариационной матрицы.

Большая часть реализаций трекинга использует какой-либо линейный метод наименьших квадратов, в зависимости от модели трека. Глобальный метод наименьших квадратов является оптимальным для линейной модели, когда функция f_{ik}, описывающая трек от i-й до k-й станции детектора является линейной функцией от вектора состояния q_i, если плотности вероятностей гауссовы. Рассмотрим совокупность хитов трека как динамическую систему. В таком случае решение задачи разбивается на несколько этапов. Первый этап заключается в получении зависимости вектора измерений m_k от начального состояния частицы q_0, на k-м слое детектора в виде (2).

$$m_k = d_k(q_0) + \gamma_k$$

где $d_k = h_k \circ f_{k|k-1} \circ \ldots \circ f_{2|1} \circ f_{1|0}$, $f_{k|k-1}$– функция, описывающая трек между станциями k и k-1.

Стохастический γ_kсодержит все кратные кулоновские рассеяния до слоя k, а также погрешность измерения m_k. Также необходима линеаризация модели трека, что приводит к появлению якобиана D_k:

$$D_k = H_k F_{k|k-1} \ldots F_{2|1} F_{1|0}$$

где H – якобиан h, F – якобиан f.

Наблюдения, состояния, шумовые компоненты представляются в векторном виде:

$$m = (m_1, \ldots m_n)^T, d = (d_1, \ldots, d_n)^T, D = (D_1, \ldots, D_n)^T, \gamma = (\gamma_1, \ldots, \gamma_2)^T$$

где n – общее число измерений, то есть число слоёв детектора.

Тогда модель можно представить как:

$$m = d(q_0) + \gamma = D(q_0) + c + \gamma$$

где c – константа.

Метод наименьших квадратов позволяет получить глобальную оценку метода наименьших квадратов:

$$\tilde{q}_0 = D^T G D^{-1} D^T G (m - c)$$

D^{-1}является недиагональной ковариационной матрицей γ.

К минусам методов наименьших квадратов можно отнести то, что они недостаточно робастны. Так, в случае многократного рассеивания предсказанный трек может значительно отклониться от реального трека. Кроме того, большое количество измерений приводит к высокой вычислительной сложности в результате большого количества обращений матриц.

2.5. Фильтр Калмана

Одним из самых популярных методов трекинга является рекурсивный фильтр Калмана [9] принцип которого заключается в рекурсивном представлении метода наименьших квадратов, так как в нём используется минимизация χ^2 измерений на треке. При этом трек восстанавливается последовательно, от станции к станции.

Главным отличием и преимуществом данного метода является то, что в нём происходит переход из пространства измерений большой размерности в пространство параметров трека, определяемое вектором состояния с малым числом компонент, что значительно уменьшает размер ковариационной матрицы при обновлении состояния. Использование критерия χ^2 также значительно ускоряет трекинг, позволяя определять, будет ли добавлено измерение к треку-кандидату или нет, избегая повторного подгонки трека. Кроме того, после фильтрации измерений всего трека известно состояние вдоль всей траектории частицы.

Метод фильтра Калмана состоит из трех этапов:

1. предсказание следующего состояния системы на основе текущего;
2. фильтрация реальных хитов на следующей станции на основе предсказанного состояния и уравнений фильтра;
3. сглаживание сформированного трека-кандидата проходом в обратную сторону вдоль сформированной траектории.

Предсказание вектора состояния и его ковариационной матрицы на k-ой станции детектора получается с помощью уравнений (3) и (4):

$$\vec{x}_k^{k-1} = f_k(\vec{x}_{k-1}),$$

$$C_k^{k-1} = F_k C_{k-1} F_k^T + Q_k$$

где $\vec{x}_k^{k-1}$ – предсказанный вектор состояния на основе текущего состояния;

$\vec{x}_k^k \equiv \vec{x}_k$ – значение вектора состояния после фильтрации;

$\vec{x}_{n-1}^n$ – вектор состояния после сглаживания;

Q_k – шум процесса;

C_k^{k-1} –ковариационная матрица.

Для подгонки трека и первого предсказания необходима начальная оценка вектора состояния $\vec{x}_0$. При отсутствии оценок, полученных от других детекторов, начальный вектор состояния строится с помощью подбора трек-кандидатов (сидинг).

После предсказания вектора состояния на k-м слое детектора, необходимо рассчитать отклонение – это расстояние между хитом m_k и хитом, полученным из предсказанного вектора состояния. Предсказанное отклонение считается по формуле:

$$r_k^{k-1} = m_k - h_k\left(\vec{x}_k^{k-1}\right),$$

где m_k – измерение, хит;

h_k – измерительная модель, описывающая функциональную зависимость хитов от вектора состояния $\vec{x}_k^{k-1}$.

Ожидаемая дисперсия, R_k^{k-1} может быть получена из вклада ковариационной матрицы состояния C_k^{k-1} и измерительной ошибки, V_k:

$$R_k^{k-1} = V_k + H_k C_k^{k-1} H_k^T,$$

где R_k^{k-1} – дисперсия состояния;

V_k – матрица ошибок измерения;

H_k – якобиан преобразования от измерений к состоянию.

Подгонка фильтром Калмана заключается в поиске оптимального вектора состояния, минимизирующего значение χ_+^2:

$$(\chi_+^2)_k^{k-1} = \left(r_k^{k-1}\right)^T \left(R_k^{k-1}\right)^{-1} r_k^{k-1}.$$

На рисунке 11 изображена процедура предсказания и фильтрации с помощью фильтра Калмана на примере прослеживания трека вдоль оси *OZ* [10].

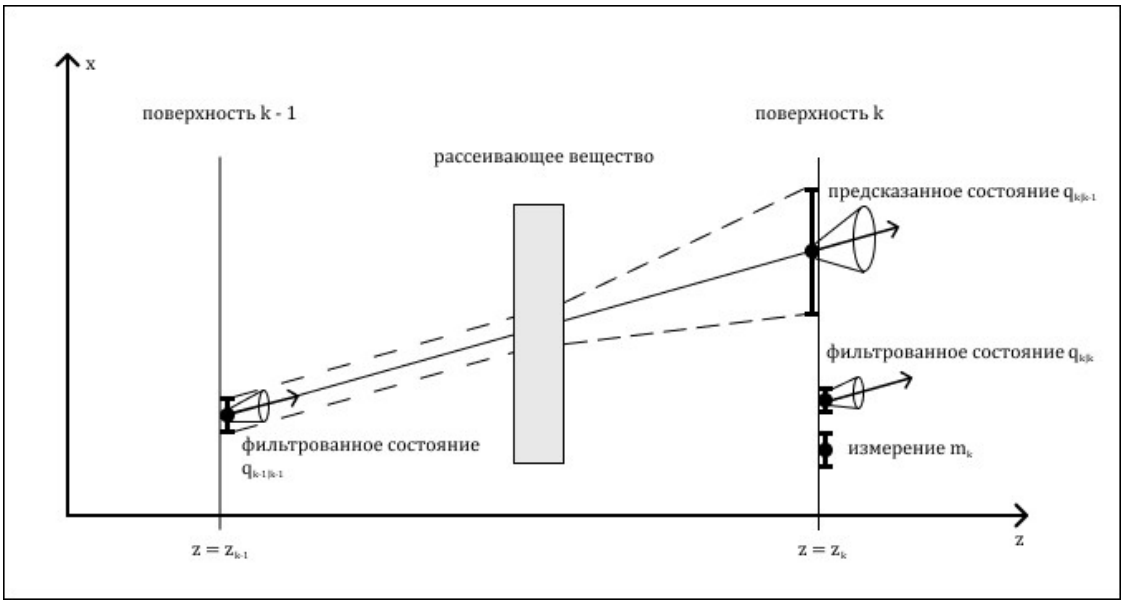

Рисунок 11. – Иллюстрация работы фильтра Калмана

Если имеется большое количество треков, близких друг к другу, или высокий вклад шума, хит, ближайший к прогнозируемому треку, не обязательно принадлежит к рассматриваемому треку. Для таких ситуаций, описанную выше процедуру можно обобщить до комбинаторного фильтра Калмана (англ. – *combinatorial Kalman filter, CKF*) [10].

В *CKF* несколько гипотез о треке принимаются одновременно до тех пор, пока одна из них не будет принята с достаточной уверенностью. Если в первом слое после сидинга имеется несколько совместимых измерений, генерируется несколько ветвей фильтра Калмана, каждая из которых содержит уникальное совместимое измерение в конце ветви. Чтобы справиться с

потенциальной неэффективностью детектора, также создается ветвь с отсутствующим измерением. Все ветви экстраполируются на следующую станцию детектора, содержащую, по меньшей мере, один совместимый хит, и создаются новые ветви для каждой комбинации предсказанных состояний, совместимых с данным хитом.

Описанная процедура приводит к комбинаторному дереву из множества фильтров Калмана, работающих параллельно. Ветвления удаляются, если общее качество ветвления (в пересчете на общую величину χ^2 трек-кандидата до рассматриваемой станции) падает ниже определенного значения, или если пройдено слишком много последовательных уровней без совместимых хитов. В конце концов, ветвь с наивысшим качеством критерия χ^2 из оставшихся сохраняется [11].

Несмотря на все свои преимущества, фильтр Калмана требует выполнения дорогостоящей с вычислительной точки зрения процедуры инициализации, заключающейся в полном переборе всех возможных вариантов трек-кандидатов для первых трех станций детектора. Существует целый ряд методов, позволяющих упростить и ускорить процедуру инициализации фильтра Калмана, о них речь пойдет ниже.

Все это делает фильтр Калмана громоздким и требующим большого количества оперативной памяти и вычислительных мощностей.

Фильтр Калмана активно применяется и в наши дни, однако из-за большого количества измерений и хитов во многих экспериментах слишком велико для достаточно быстрой обработки данных событий. Так, ппланируемый к запуску эксперимент MPD на строящимся в Дубне коллайдере NICA требует создания специального метода для быстрого отбора событий с интересующими исследователей характеристиками для последующей передачи в фильтр Калмана. В связи с этим ученые

разрабатывают различные методы сидинга или уменьшения комбинаторного пространства для ФК [12].

Современные узлы, используемые для вычислений в экспериментах, распределённые и многопроцессорные, что требует от алгоритмов хорошей способности к распараллеливанию, которой фильтр Калмана не обладает. Алгоритмы с использованием нейронных сетей по этой причине всё чаще применяются учёными при работе с такими объемами данных.

2.6. Клеточный автомат для поиска трек-кандидатов

Клеточный автомат - это дискретная сетка, клетки которой в каждый момент времени могут принимать одно состояние из некоторого конечного множества. Смена состояний определяется заданными заранее правилами перехода, например, принятие состояния 1 или 0 в зависимости от числа соседей и т.п.

Клеточные автоматы обладают следующими свойствами:

– параллельность (так как все клетки обновляются независимо друг от друга);

– локальность (новое состояние зависит только от старого состояния и некоторой окрестности клетки);

– однородность (так как все клетки обновляются по одним и тем же правилам).

На рисунке 12 изображена работа КА для поиска трек-кандидата.

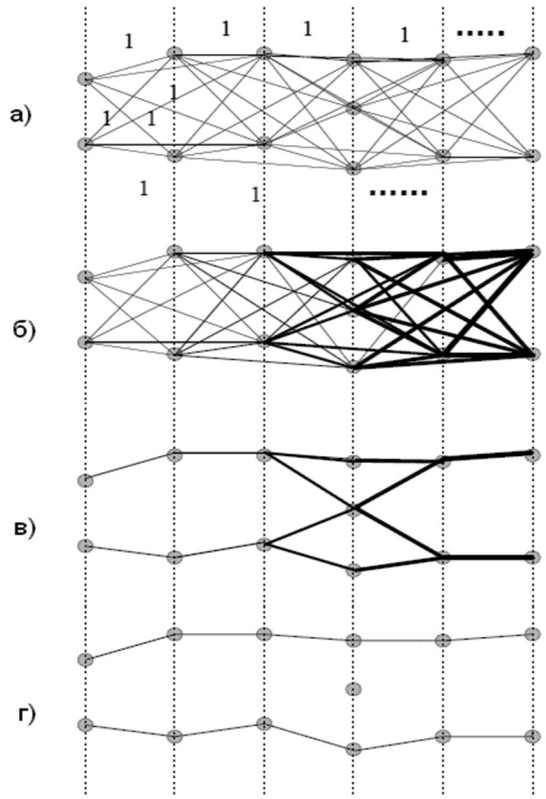

Рисунок 12. – КА для поиска трек-кандидатов в эксперименте *HERA-B* [12]: инициализация КА (а), конец эволюции (б), сбор трек-кандидатов (в) и удаление ложных сегментов (г)

В [14] были предложены правила построения КА, отсеивающего шумовые точки для распознавания связных групп хитов и заполнения пропусков в процессе эволюции клеток. Эти правила, однако, были всего лишь обобщением широко известного КА «жизнь» [13].

В клеточном автомате для трекинга клетка определяется как сегмент, объединяющий два измерения на соседних станциях детектора (с учетом неэффективности оборудования допускается соединения с пропусками – со следующей станцией вместо ближайшей). Трек должен представлять собой гладкую кривую, поэтому соседство определяется близостью направлений.

При инициализации клетки, представляющие сегменты с допустимыми углами, получают состояние 1. В процессе эволюции, если у клетки-соседа на предыдущем слое то же состояние, значение состояния клетки увеличивается на единицу. Эволюция продолжается, пока все клетки не будут иметь разные

состояния. После окончания эволюции трек-кандидаты собираются из полученных сегментов с помощью движения от последней станции детектора к первой и соединения клеток с соседями с предыдущими значениями. Лишние ветки, в случае их появления, отсеиваются по критерию гладкости. На заключительной стадии кандидаты отсеиваются с помощью χ^2 выбора треков с большим числом сегментов.

Применение КА не избавляет от использования фильтра Калмана, кроме того, исходные коды программы недоступны для широкого использования, что сильно ограничивает настройку и адаптацию алгоритма под нужды различных экспериментов ФВЭ.

2.7. Нейронные сети Хопфилда

В процессе развития экспериментальной физики увеличивалась и энергия в проводимых экспериментах, что привело к необходимости обрабатывать события с большим числом треков, поэтому классические методы перестали удовлетворять учёных. Чтобы выполнять трекинг в таких условиях, ученые обратили внимание на модели искусственных нейронных сетей (ИНС). Такие методы имеют много полезных свойств, таких как способность к обучению, устойчивость к шумам, легкая адаптация программ к выполнению параллельной обработки и широко применяются в различных задачах распознавания образов. Самые ранние обзоры применения нейронных сетей в физике высоких энергий можно найти в [14,15].

В работе [17] была сформулирована задача коммивояжера, который ищет кратчайший путь через *N* городов, расположенных в наборе известных координат, как задачу минимизации энергетической функции сети Хопфилда [16]. Подобную постановку задачи использовали исследователи Денби и Петерсон для разработки метода сегментов для трекинга [17,18].

Рассмотрим эту постановку. Пусть имеется множество *N* экспериментальных точек на плоскости. Требуется провести непрерывные

гладкие кривые (треки) так, чтобы они прошли через эти N точек. Предполагается, что треки не имеют изломов и разветвлений.

Введём нейроны v_{ij}, определяющие, принадлежит ли данный направленный сегмент v_{ij} от точки i к точке j треку или нет. Начальные состояния нейронов выбираются как $v_{ij} = 1$, если соответствующий направленный сегмент принадлежит треку; в противном случае $v_{ij} = 0$. В процессе эволюции сети состояния нейронов определяют уровень активности нейрона в диапазоне [0,1], т.е. в случае $v_{ij} > v_{min}$ нейрон считается активным [21].

Функция энергии в [14, 15] определялась как:

$$E = E_{cost} + E_{constraint}.$$

Пусть r_{ij} – это длина сегмента между измерениями *i* и *j*, тогда первый член функции энергии выражается:

$$E_{cost} = \{-\frac{1}{2}\sum_{ijkl} \delta_{jk} \frac{\theta_{ijl}}{r_{ij} + r_{jl}} v_{ij} v_{kl}\,, \quad \text{если} \quad \theta_{ijl} \le \alpha_{con},$$

$$0, \qquad\qquad \text{если} \quad \theta_{ijl} > \alpha_{con},$$

где θ_{ijl} – угол между сегментами, соединяющими точки *i* с *j* и *j* с *l*;

α_{con} – пороговая константа, рассчитываемая с учетом геометрических особенностей детектора;

m – целочисленный показатель степени, который подбирается вручную.

Таким образом, первый член функции энергии поощряет короткие смежные сегменты с малым углом между ними.

Второй член$E_{constraint}$ состоит из суммы двух частей. Первая часть запрещает разветвления (бифуркации) трека:

$$T_{ijkl}^{(1)} = \frac{\alpha}{2}\left[\delta_{ik}(1-\delta_{jl}) + \delta_{jl}(1-\delta_{ik})\right] = \frac{\alpha}{2}\left[\sum_{l\neq j} v_{ij}v_{jl} + \sum_{k\neq i} v_{ij}v_{kj}\right].$$

Вторая часть имеет смысл баланса между числом активных нейронов и числом экспериментальных точек:

$$T^{(2)} = \frac{\beta}{2}\left[\sum_{kl} v_{kl} - N\right]^2.$$

Данный подход был успешно применен для распознавания треков в эксперименте *EXCHARM* [19].

Однако данный метод оказывается неустойчивым к зашумлению и резкому увеличению размерности входа и матрицы нейронов при добавлении даже одного измерения. Число нейронов сети равно $N(N-1)$, что требует больших вычислительных затрат на поиск оптимального стационарного состояния сети и делает метод практически непригодным для экспериментов ФВЭ с тяжелыми ионами и высокой множественностью.

2.8. Эластичные нейронные сети

Термин «эластичные нейронные сети» был введён Дурбином и Вилшоу (ДВ) также для решения задачи коммивояжера [20].

Задача решалась авторами следующим образом: в центр плоскости с множеством размеченных городов помещался маленький эластичный замкнутый контур – окружность с *n* нейронами на ней. В процессе эволюции контур растягивался до тех пор, пока не образует оптимальный обход всех

городов под действием двух сил: одна двигает нейроны к ближайшему из городов, а вторая – отталкивает от соседей по контуру. Метод ДВ позже был обобщен для решения задачи реконструкции треков частиц в эксперименте с дрейфовыми камерами – NEMO [24].

Предлагая метод эластичного трекинга, Джиласси и Харландер исходили из идеи гибкого шаблона. Шаблон представляет собой уравнение трека, описывающее кривую, которая зависит от вариаций параметров таким образом, чтобы пройти как можно ближе к хитам трека [26]. Данный подход с физической точки зрения описывается как взаимодействие положительно заряженного шаблона трека и отрицательно заряженных хитов этого трека. Энергия их взаимодействия тем меньше, чем лучше гибкий шаблон пройдет по точкам измерений трека.

Пусть заряд для шаблона трека распределен с плотностью $\rho_T(r)$, а заряд множества хитов трека – $\rho(r')$, тогда задача реконструкции трека описывается уравнением:

$$E = \int dr' dr \rho_T(r) V(r - r') \rho(r') \rightarrow min$$

где V – потенциал Лоренца, E – энергия взаимодействия.

Эффективное применение эластичных методов сильно затруднено необходимостью выбора начального приближения. Так, если применить для поиска первых трёх хитов применить преобразование Хафа, то точности в измерении кривизны трека, полученной таким методом, будет недостаточно для определения надежно продолжаемого шаблона [22].

2.9. Глубокие нейронные сети

Массивное развитие ИНС привело к появлению различных архитектур: от свёрточных до графовых. Некоторые из них применяются для трекинга

частиц [25,28],. Так, для трекинга в эксперименте NICA до сих пор применялись два различных подхода –графовый и с использованием рекуррентных нейронных сетей.

Первый подход заключается в построении графового представления события, когда срабатывания детектора представляются как узлы графа. Идея представления массива данных в виде графа для последующей обработки специальной нейронной сетью была представлена в статье [28] в 2016 году. Эта идея была адаптирована для решения задачи трекинга проектом HEPTrkX [24] и далее развита в проекте NICA [25]. Хиты события соединяются ребрами так, чтобы соединялись только хиты на разных станциях, после чего веса рёбер вычисляются с помощью полносвязной нейронной сети. Треки же составляются из рёбер с весами выше некоторого порога. Из-за большого количества фейковых хитов в событии потребовалось введение инверсного графа: в нём ребра исходного графа представляются узлами и наоборот. При этом информация о кривизне треков оказывается заложена в рёбрах графа, что упрощает распознавание треков среди фейков и шумов. На рисунке 13 можно увидеть результат трекинга для одного события в графовом представлении.

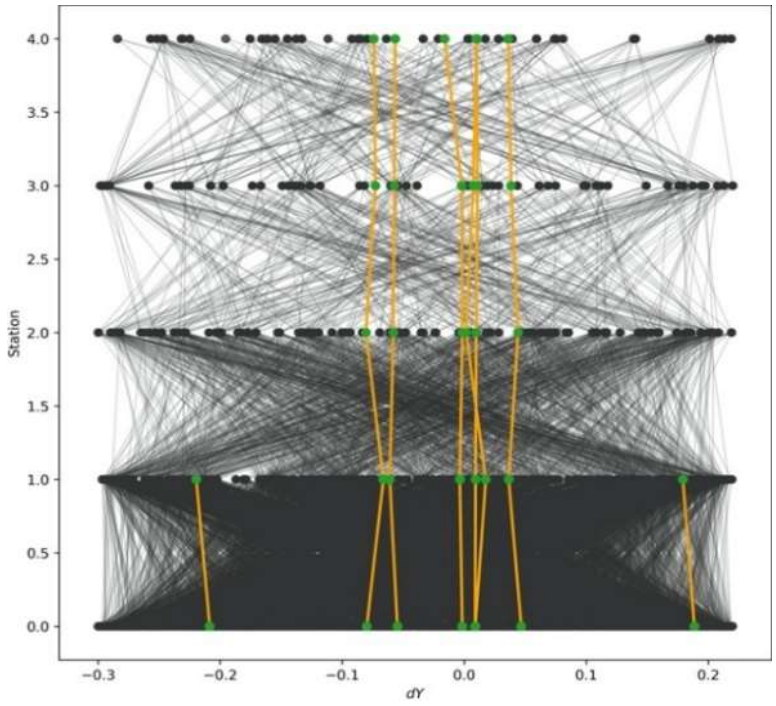

Рис. 13. Графическое представление треков с помощью графа события. Черные узлы и ребра соответствуют фейкам , зеленые узлы и желтые ребра - найденным трекам.

Данная модель, как можно увидеть, реализует глобальный подход, когда всё событие обрабатывается одновременно. Это приводит, с одной стороны, к хорошим результатам, с другой стороны, требует высоких затрат памяти на построение графа события и дальнейшего трекинга.

Рекуррентные сети предназначены для обработки информации, представленной временными рядами [27]. Этот вид нейронных сетей широко используется в задачах моделирования денежных потоков, речи, предсказании различных показателей [28]. Для трекинга с помощью рекуррентной нейронной сети используется *GRU* (*Gated Recurrent Unit*), устройство которого можно видеть на рис. 14. Как можно увидеть, при моделировании с помощью такого нейрона значение состояния на текущем шаге зависит от вектора входа и векторов состояний предыдущих шагов. При этом GRU обладает памятью, что позволяет моделировать достаточно длинные последовательности для задач трекинга.

Для трекинга в *BM@N* характерен большой дисбаланс между реальными и фейковыми хитами, что приводит к высокой комбинаторной сложности поиска хитов трека на каждом этапе. Исходя из этого, последовательно было разработано несколько рекуррентных моделей, учитывающих эти особенности.

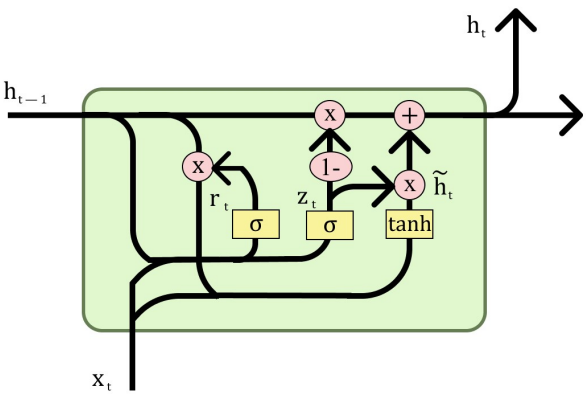

Рисунок 14. – Схема работы *GRU* нейрона

Первым решением был двухступенчатый трекинг, состоящий из предварительной обработки данных путем направленного поиска трек-кандидатов в трехмерной системе координат с помощью *KD*-дерева и дальнейшего разделения кандидатов на реальные и ложные треки с помощью глубокой рекуррентной сети-классификатора [29]. В результате была получена эффективность распознавания, близкая к 100%, однако построение KD-дерева приводило к высоким временным затратам.

Второй подход заключался в предсказании без использования KD-дерева. В новом подходе была добавлена регрессионная часть для предсказания области поиска следующего хита трека-кандидата в виде эллипса [30]. Схему модели, названной TrackNET, можно видеть на рисунке 15. Модель использует рекуррентную архитектуру для извлечения временных признаков из данных и свёрточную, чтобы обогатить входной набор признаков, используя скрытые зависимости координат хитов. Таким образом, эта модель экстраполирует начальное состояние трека на последующие станции путем последовательного предсказания эллипса, в котором ищется следующее возможное попадание, и одновременно оценивает вероятности принадлежности полученных последовательностей к реальным трекам.

Один из недостатков данной модели заключался в том, что невозможно классифицировать сиды, то есть первый отрезок трека, так как первый отрезок определяет только направление, но не кривизну. Также невозможно предсказать эллипсы при максимальной длине трека на входе и соответственно вычислить ошибку предсказания (так как хитов на следующей станции просто нет). В результате было необходимо использовать сразу три модели для обработки кандидатов различной длины.

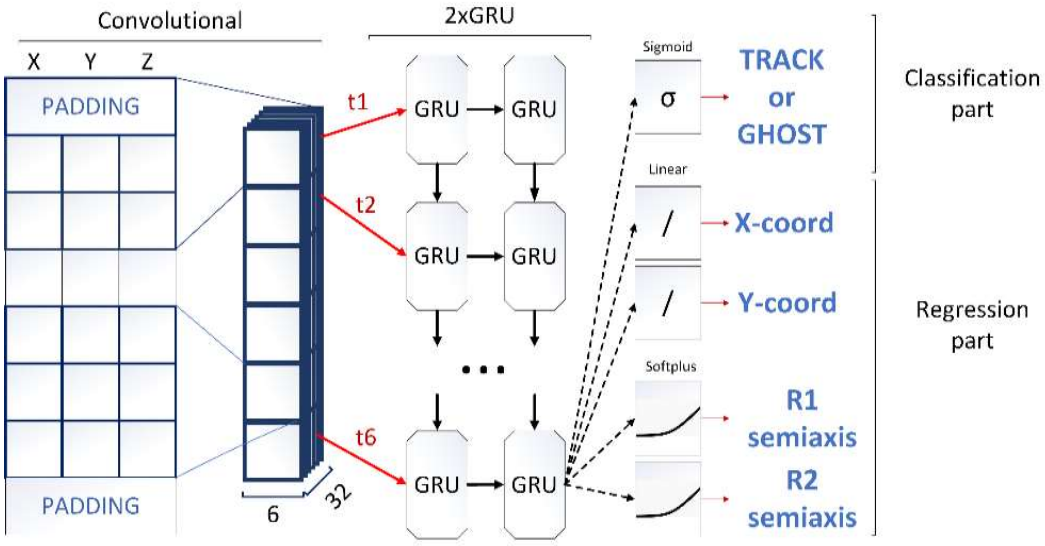

Рис.15. Схема модели TrackNETv1

Для устранения ограничений первой версии TrackNET была разработана модель без классификатора –TrackNETv2 [30]. Треки-кандидаты в последнем случае создаются последовательно с помощью выбора хитов, попавших в эллипсы, и даже если ранние эллипсы содержат неправильные хиты (фейковые или части других треков), последующее построение эллипсов приводит к отбрасыванию таких кандидатов. TrackNETv2 показала высокие результаты для эксперимента BM@N [30].

Глава 3. Разработка модели TrackNETv3 для локального трекинга

3.1. Применение TrackNETv2 к данным коллайдерных экспериментов на примере BESIII

В результате сравнительного анализа существующих подходов для трекинга было решено использовать TrackNETv2 как основу для разработки новой модели. Для разработки новой системы набор данных, смоделированных для эксперимента BESIII, был выбран основным. Особенность их не только в цилиндрической форме станций, как было сказано ранее, но и в малом по сравнению с BM@N числе станций. Однако для

сравнения старого и нового подходов было решено использовать и данные, смоделированные для эксперимента BM@N.

Исходная модель TrackNETv2 была обучена более чем на 250 тысячах событий и проверена на более чем 100 тысячах событий. Среднее количество хитов в одном событии было около 50, но некоторые из них содержали до 300 хитов, при том, что из-за низкой энергии в 1GeV число треков в событии не превышало 20. После обучения модель была протестирована на новых данных с детектора BESIII CGEM-IT, которые не были доступны модели во время обучения и оптимизации гиперпараметров.

Фаза тестирования показала высокую степень чувствительности: для каждого реального начала трека обнаруживается реальное попадание в последний хит с вероятностью 99%. В то же время точность невысока, всего 1 процент, что означает, что модель отфильтровывает очень небольшое количество кандидатов, которые не являются настоящими треками.

Таким образом, TrackNETv2 показала слабую способность фильтровать короткие треки. Возможное улучшение результатов возможно при добавлении z-координаты следующего хита к координатам хитов на текущей станции, или добавлении классификатора, позволяющего оценить вероятность принадлежности кандидата к множеству реальных треков.

3.2. Разработка классификатора треков-кандидатов

Учитывая низкую точность исходной модели для реконструкции треков с низкой энергией, было решено разработать новую часть поверх TrackNETv2, чтобы классифицировать, является ли реконструированный трек реальным или нет. Цель этой части - отфильтровать синтетические треки-кандидаты, так как в реальности неизвестно, какие хиты образуют трек, а какие нет, и берутся сочетания всех хитов на уже пройденных станциях. При этом новый классификатор должен получать на вход не только информацию, которой распоряжается TrackNETv2, но и информацию о следующем отобранном хите.

Классификатор в данном случае выступает в роли аналога критерия Хи-квадрат в фильтре Калмана, но, в отличие от классического подхода, не требует подгонки трека для получения параметров траектории, потому что выучивает все необходимые зависимости из исходных данных.

3.2.1. Классификатор на основе внутренних признаков TrackNETv2

Модель получает на вход внутренние признаки TrackNETv2 и предсказанное завершение (последний хит, попавший в эллипс) для классификации трека. Данная архитектура позволяет использовать признаки, уже извлеченные для первых хитов, и добавлять к ним только координаты последнего хита, что должно приводить как к быстрому достижению желаемых результатов, так и к высокой точности. Однако такая архитектура требует дополнительного обучения и подготовки новых данных для каждой новой версии исходной модели. Кроме того, данную модель невозможно объединить в одну модель с TrackNETv2, так как она использует на вход данные, которые еще не получены на момент формирования входных данных основной модели.

Каждый сигнал подается на полносвязанный слой (FC - Fully Connected), затем результирующие векторы объединяются (Concat). Результирующий сигнал поступает на вход следующего полносвязного слоя (FC). Все описанные выше слои используют функцию активации ReLU [34]. Выход последнего уровня FC нейронной сети имеет функцию активации Softmax [33], а произведенный сигнал (выход) можно интерпретировать как вероятность того, является ли трек истинным треком или нет (рис. 17).

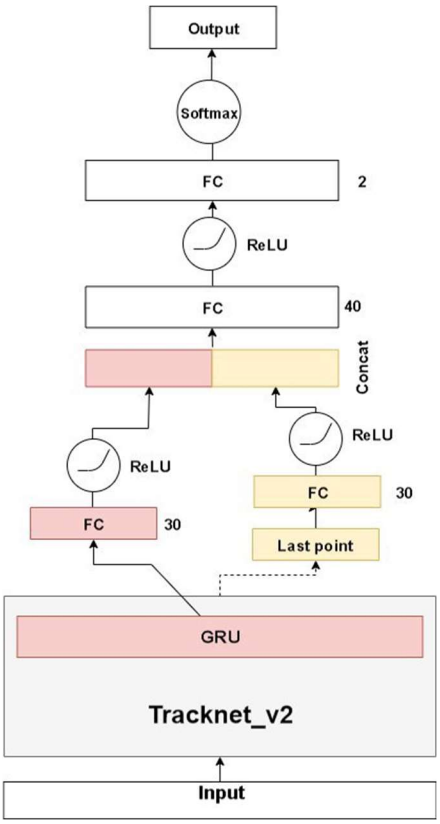

Рис. 17. - Архитектура новой модели

Модель TrackNETv2 с последующим использованием классификатора называется TrackNETv2.1.

3.2.2. Классификатор на основе координат трека

Внутреннее состояние модели может не сохранять полную информацию о таких параметрах трека, как кривизна или длина, непосредственные координаты на станциях каждого хита, поэтому часть полезных признаков может теряться. Более того, в результате добавления следующего хита информация о последнем отрезке может быть не связанной с информацией о кривизне и направлении остальной части трека.

Кроме того, если для классификации использовать непосредственные параметры трека, то каждая новая итерация основной модели требует относительно небольшого обучения классификатора, а также две модели можно обучать и использовать одновременно. Однако простая архитектура может привести к тому, что зависимости между хитами на двух разных

станциях могут теряться, и в случае большого числа станций классификация может быть неудовлетворительной.

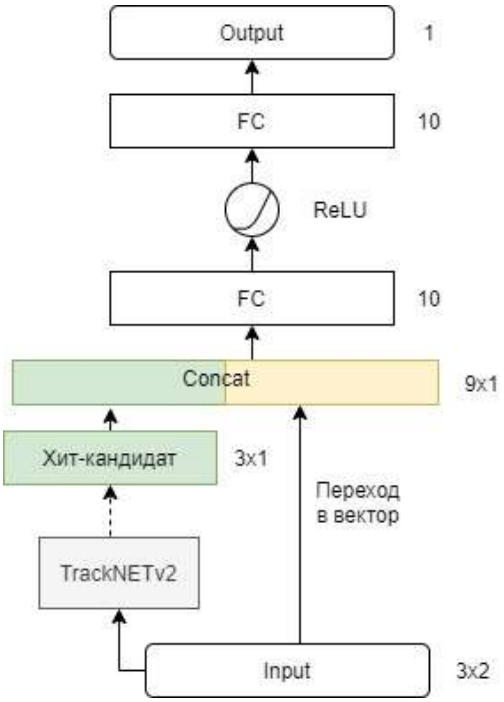

Рис. 18. - Схема классификатора на основе координат трека

Координаты трека-кандидата объединяются в вектор, после чего подаются на полносвязанный уровень (FC - Fully Connected). Результирующий сигнал поступает на вход следующего полносвязного слоя (FC). Все описанные выше слои используют функцию активации ReLU. Выход последнего уровня FC нейронной сети имеет функцию активации Sigmoid, а произведенный сигнал (выход) можно интерпретировать как вероятность того, является ли трек истинным треком или нет (рис. 18).

3.3. Процедура обучения

Для обучения классификатора необходимо учитывать дисбаланс классов. Дисбаланс классов приводит к тому, что вклад позитивных примеров в обучение модели очень низкий по сравнению с негативными, и предсказания значительно теряют в качестве.

Даже при предварительной фильтрации на каждый реальный трек-кандидат приходится около 10 таких, которые являются фейковыми. Это

связано с природой данных – стриповые детекторы производят большое количество ложных срабатываний. Кроме того, многие фейковые треки-кандидаты практически не отличаются от реальных. При обучении TrackNETv2 это не учитывается, так как используются только реальные треки, но при подготовке данных для классификатора это свойство коллайдерных экспериментов приобретает чрезвычайную важность.

Самым простым способом борьбы с дисбалансом классов является отбрасывание части фейковых треков [34]. Однако при использовании этого способа набор данных обедняется, и необходимо значительно увеличивать исходный набор данных, чтобы после отбрасывания обучение оставалось эффективным.

Это делает особенно важным выбор функции потерь – ее неправильный выбор часто приводит к тому, что сеть переобучается или встает в ступор в независимости от объема входящих данных. Важно отметить, что выбор функции потерь влияет на скорость обучения любой выбранной архитектуры нейронной сети.

Дисбаланс классов в различных областях, и одним из способов борьбы с ним традиционно используется взвешивание функции ошибки, так что вклад положительных объектов становится сравним со вкладом негативных. Наиболее часто для бинарной классификации (когда в данных присутствует только два класса) используется бинарная кросс-энтропия, и исследователями создано довольно большое количество её модификаций [35].

Кросс-энтропия измеряет расхождение между двумя вероятностными распределениями. Если кросс-энтропия велика, это означает, что разница между двумя распределениями велика, а если кросс-энтропия мала, то распределения похожи друг на друга. Она определяется как:

$$H\ (P,Q) = -\sum_{x} P(x) log\ Q(x)$$
$$= -(y \cdot log\ (p)\ + (1-y) \cdot log\ (1-p)\ )$$

При бинарной классификации каждая предсказанная вероятность сравнивается с фактическим значением класса (0 или 1), и вычисляется оценка, которая штрафует вероятность на основе расстояния от ожидаемого значения. Добавление весов перед каждым слагаемым позволяет увеличить вклад того или иного класса, в противном случае, при таком дисбалансе, как в настоящем случае, модель будет предсказывать практически только отрицательный класс. Поэтому к кросс-энтропии добавляется множитель α для положительного класса и 1-α для отрицательного для балансировки вклада негативных и позитивных примеров в обучение.

При классификации классов с выбранным способом трекинга рассматриваются все треки-кандидаты, попавшие в “коридор” из эллипсов, однако один коридор может включать несколько треков-кандидатов, реальным треком среди которых может быть только один. При этом фейковые треки могут практически не отличаться от реальных. Кроме того, возможны реальные треки с малой кривизной, которые очень сильно отличаются от также реальных треков, но с меньшим импульсом, и, следовательно, большой кривизной. Это, а также то, что многие синтетические треки не имеют физического смысла и поэтому очень отличаются от реальных, приводит к большим трудностям при обучении. Так, модель стремится к предсказаниям с высокой уверенностью, треки же, не столь характерные для своих классов, теряют значение.

Для решения данных проблем была разработана Focal Loss на базе обычной кросс-энтропии [36]. Для данной функции необходимо рассмотреть следующие категории примеров из выборки:

1. Легкие положительные/отрицательные: образцы, классифицируемые как положительные / отрицательные.
2. Сложные положительные / отрицательные результаты: образцы ошибочно классифицируются как отрицательные / положительные.

Focal Loss позволяет учитывать обе категории с помощью соотношения:

$$FL(p) = -\alpha(1-p)^{\gamma} y \cdot log(p) + (1-y) \cdot (1-\alpha) p^{\gamma} log\,(1-p)$$

где p – предсказанная вероятность принадлежности к положительному классу,

y – истинный класс примера,

α – вес для положительного класса,

γ-параметр контроля сложных примеров

В результате для лёгких примеров значение функции ошибки уменьшается, и обучение в-основном концентрируется на сложных примерах. Гамма является гиперпараметром, в зависимости от которого идет взвешивание вниз для легких примеров. В [36] показано , что γ=2 даёт наилучшие результаты в большинстве случаев.

В рамках данной работы были рассмотрены две эти функции потерь.

Глава 4. Разработка программного решения

4.1. Используемые технологии. библиотека Ariadne

В настоящее время, решение задач обработки данных и машинного обучения чаще всего использует язык Python [36]. Это один из наиболее гибких языков программирования, поддерживающий множество различных парадигм. Кроме того, для этого языка существует множество библиотек, предназначенных для решения различных задач, в том числе машинного и глубокого обучения.

При том, что в настоящее время наблюдается интерес к решению различных задач из области теоретической физики с использованием машинного обучения, а также глубокого обучения, на данный момент не существует развитых библиотек, специализированных для таких задач. Поэтому данное исследование велось, в том числе, как часть разработки библиотеки Ariadne[1]– открытого проекта для решения различных задач экспериментальной физики с использованием нейронных сетей.

При исследовании были разработаны блок трансформаций пространства трекинга и самих треков: очистки, перевода координат, разбиения на наборы данных и т.д., а также различных инструментов для поддержания пайплайна подготовки данных, обучения моделей и их использования. Разработанные модели также стали частью библиотеки.

Библиотека основана на Pytorch – открытом фреймворке для глубокого обучения с поддержкой обучения на GPU. Фреймворк является одним из наиболее популярных на настоящее время, предоставляет многие блоки, используемые в современных моделях, также на его легко создавать пайплайны подготовки, загрузки данных и обучения моделей. Классы, доступные в фреймворке, используются для обучения и тестирования модели, а также для создания самих моделей.

Кроме того, библиотека позволяет конфигурировать все элементы с помощью gin-config, и предоставляет готовые скрипты для подготовки данных и обучения модели. Исследователям необходимо лишь создать свои классы препроцессора, набора данных и модели, наследуясь от уже существующих.

Однако трековые данные сильно отличаются от стандартных, например, изображений или текста, поэтому требуют разработки различных средств, позволяющих проводить машинное обучения. При подготовке данной работы

[1] https://github.com/t3hseus/ariadne

были разработаны некоторые необходимые модули, описание которых будет дано ниже.

4.2. Разработка модуля трансформаций

Различные эксперименты ФВЭ производят данные, вид и распределения которых могут серьёзно отличаться для разных экспериментов. Даже два эксперимента, смоделированные данные для которых рассматриваются в данной работе, имеют различный вид из-за разной конфигурации детекторных станций. Таким образом, при разработке системы нейросетевого трекинга необходимо предусмотреть средство, позволяющее преобразовывать данные в такие, которые подходят для обработки.

В рамках библиотеки Ariadne был разработан модуль трансформаций, который удовлетворяет следующим требованиям:

- трансформации должны подходить для табличного представления хитов;
- трансформации должны возвращать преобразованное табличное представление хитов;
- должна быть возможность проводить трансформации последовательно;
- модуль должен быть расширяемых для данных различных экспериментов и т.д.

Исходя из вышеперечисленного, было решено добавить модуль трансформаций на стадию подготовки данных.

Всего трансформации включают три базовых трансформации: BaseFilter, BaseConverter, BaseScaler с общим классом-предком BaseTransformer.

На рисунке 19 можно увидеть диаграмму базовых классов. Классы, производные от них, в данной диаграмме не учитываются.

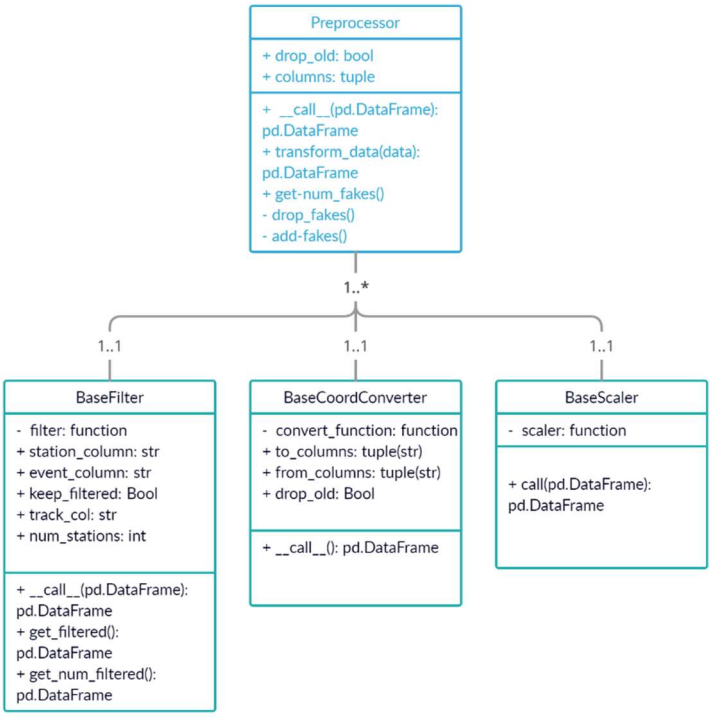

Рис. 19 - Диаграмма базовых классов трансформаций

Каждый из этих классов позволяет создавать объекты с различными функциями преобразования данных.

При вызове фильтра (BaseFilter) с заданной функцией фильтра эта функция применяется к данным, сгруппированным по столбцам event_col и track_col, после чего возвращается разгруппированный набор из групп, удовлетворяющих условию фильтра. Так, необходимо удалять треки длиной меньше трёх, так как для них невозможно предсказание.

При вызове трансформации координат (BaseCoordConverter), столбцы from_coords преобразуются в столбцы to_coords по заданной формуле, например, формуле перевода декартовых координат в цилиндрические без какой-либо группировки.

При вызове трансформации масштабирования (BaseScaler) заданные в атрибуте columns преобразуются к необходимому масштабу по формуле,

реализованной в функции scaler. Так, например, для масштабирования по минимуму/максимуму определяются эти величины для каждого столбца и все его значения преобразуются по формуле:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Так как для различных экспериментов часто нужны одинаковые преобразования данных, были реализованы наследники вышеперечисленных классов с предварительно заданными функциями преобразования, среди них:

DropFakes – удаляет все строчки набора данных, трек которых задан как -1 (фейковый)

DropShort – удаляет все треки длиной меньше заданного числа

DropSpinningTracks – удаляет все треки, хиты которых были зарегистрированы на одной станции более одного раза

Normalize – нормализует значения в заданных столбцов так, чтобы получить нормальное распределение со средним в нуле и стандартным отклонением 1

MinMaxScale – нормализует данные по минимуму и максимуму

ConstraintsNormalize – нормализует данные по заданным ограничением вместо вычисления минимума и максимума по выборке

ToCylindrical/ToCartesian – приводят координаты от декартовых к цилиндрическим и обратно

Для создания композиций последовательных трансформаций данных был создан класс Compose. При инициализации он получает список объектов трансформаций, при вызове последовательно вызывается каждая из них и применяется к данным с выхода предыдущей. Первая трансформация применяется к входным данным.

Таким образом, был получен модуль трансформаций, которые можно легко использовать, задав необходимые столбцы и функции для преобразования, также этот набор легко расширяется, а для одних и тех же данных можно получить различные конечные представления, задав разные списки трансформаций, которые будет к ним применяться.

4.3. Проектирование стадии подготовки данных

Библиотека Ariadne должна быть одинаково удобна для данных, полученных из разных источников, кроме того, часто объёмы данных очень велики, что затрудняет проведение экспериментов каждый раз с нуля. В связи с этим было решено разработать модуль подготовки данных, который исследователь может использовать, задавая необходимые этапы преобразования от исходных файлов в текстовом или ином виде и до формата, пригодного для чтения в классе Dataset при обучении модели. Так, необходимые данных, подготовка которых может занимать часы и даже дни, можно предобработать и при значительно сократить обучение и тестирование различных модификаций нейронных сетей или других моделей.

Для этого был создан скрипт preprocess.py, в который передаётся экземпляр потомка класса Preprocessor, после чего внутри скрипта считывается набор данных, делится на порции меньшего размера и каждая порция обрабатывается, как это необходимо в задаче исследователя. Далее эти порции объединяются в конечный набор и сохраняются.

Класс, наследуемый от Preprocessor, определяет правила, по которым будут обрабатываться данные. Логика, определяющая, как набор будет делиться на порции, содержится в методе generate_chunks, например, это группировка по событию или треку. Подготовка каждой порции должна проводиться в методе preprocess_chunk, например, преобразование координат и т.д. Также возможно преобразование порций в методе postprocess_chunks. Соединение порций в общий набор и сохранение на диске реализуется с

помощью метода save_on_disk. Все классы, реализующие Preprocessor, должны реализовать данные методы.

Например, подготовка данных для обучения TrackNETv2 на данных BESIII привела к следующему виду препроцессора (см. приложение 2).

Набор данных разбивается на порции по событиям. Трансформации данные вызываются в методе preprocess_chunks, а вся остальная логика реализована в методе postprocess_chunks. Она реализует следующий алгоритм:

Для каждой порции данных:

1) Сгруппировать порцию по трекам;

2) Для каждой группы:

   2.1) Сохранить в список входов TrackNETv2 массив значений координат по r,phi,z всех хитов трека, кроме последнего

   2.2) Сохранить в список целевых хитов значения координат r, z последнего хита трека

   2.3) Сохранить в список длин число объектов в группе.

3) Соединить списки в соответствующие массивы

В методе save_on_disk результаты для всех порций объединяются и сохраняются на диск в виде npz-файла.

Соответствующие классы были реализованы и для подготовки данных для TrackNETv3, и для подготовки данных TrackNETv2 и TrackNETv3 для данных BM@N.

4.4. Разработка стадии инференса

В этой стадии не производится оптимизации, параметры модели замораживаются. Кроме того, стадия инференса отличается от тренировки

тем, что события обрабатываются одно за другим целиком без удаления фейковых хитов, лишь после приведения координат к нужному виду.

Стадия инференса соответствует работе системы трекинга в «боевых» условиях, поэтому отличается от стадии обучения. Для её реализации необходимо предусмотреть:

1) Чтение исходных данных
2) Преобразования координат в необходимый для работы модели вид
3) Получения первичных отрезков из данных на первых двух станциях
4) Предсказания следующего хита с помощью модели
5) Удаления и сохранения в список кандидатов хитов без продолжения
6) Добавления полученных хитов к кандидатам предыдущей станции.

Стадия инференса также позволяет оценить реальное качество работы моделей. При получении очередного кандидата в треки можно определить, является ли он одним из реальных треков или нет, и после обработки события можно получить конечный процент найденных треков от общего их числа или от числа отобранных треков-кандидатов.

Необходимо отметить, что общая схема инференса подходит для локального трекинга на различных экспериментах. Так, хоть в эксперименте BESIII всего три станции, общая схема остается такой же, но заканчивается на первой итерации (станции). В зависимости же от модели и выбранного способа определения следующего хита трека реализация может отличаться.

Остановимся на поиске следующего хита. В случае с TrackNETv2 область, где он находится, определяется с помощью эллипса, после чего отбираются хиты, в него попавшие. В оригинальной статье поиск проводился с помощью определения расстояния от центра эллипса до всех хитов станции и дальнейшей фильтрации по расстоянию. Однако такой поиск ближайших

хитов и определение того, попадают ли они в данный эллипс, приводит к неудовлетворительной скорости предсказания затратам из-за большого количества хитов на каждой станции. В связи с этим, было решено переработать поиск ближайших хитов с помощью реализации быстрого поиска в индексе FAISS [37]. Данный алгоритм разработан в Facebook Research для кластеризации и поиска ближайших соседей в векторных пространствах. Он позволяет проводить эффективный поиск ближайших соседей в индексе, который составляется из заданного набора векторов, кроме того, возможен поиск с использованием GPU и разделением поиска на несколько GPU. При поиске возвращаются как индексы ближайших соседей, так и расстояния до них.

Для задачи трекинга FAISS используется следующим образом (см. приложение 4):

1) Все хиты события сохраняются в индекс
2) При получении массива центров эллипсов проводится поиск заданного числа ближайших соседей по индексу. Для задач, описанных в данной работе, достаточно искать 10 ближайших соседей.
3) Выбираются хиты, соответствующие событиям
4) По формуле принадлежности к эллипсу получается маска с 1 для тех хитов, которые попадают в эллипсы, и 0 для тех, которые не попадают в эллипсы

Хиты, отфильтрованные таким образом, попадают на следующий этап инференса.

Глава 5. Подготовка данных

5.1. Подготовка данных для тестирования и тренировки

Одной из важнейших стадий подготовки обучающей выборки для любого алгоритма машинного обучения является анализ и очистка исходных данных. Для BESIII подготовка исходных данных состоит их трех стадий.

1. Удаление сильно закручивающихся треков – треки, хиты которых были зафиксированы на одной станции более одного раза. Данные треки принадлежат частицам с низким импульсом, например, электронам, и в данном эксперименте не представляют интереса.
2. Удаление коротких треков. Короткими считаются треки, которые зафиксированы менее чем на двух станциях. Такие треки не содержат полезной информации и не могут быть продолжены.
3. Перевод в цилиндрические координаты и нормализация.

В эксперименте BM@N все детекторные станции простые и расположены друг за другом, но детектор BESIII CGEM-IT имеет цилиндрическую конструкцию. Чтобы удобно представить данные детектора CGEM-IT для TrackNETv2 и избежать изменения архитектуры нейросети и функций ошибок, необходимо преобразовать координаты в цилиндрические (рис.20). Далее координаты необходимо нормализовать для уменьшения последствий преобразования координат.

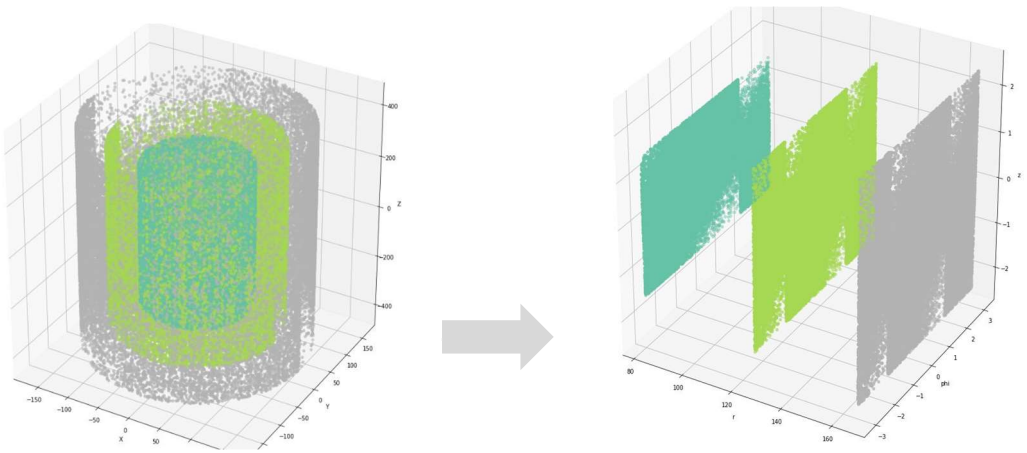

Рис. 20 -Преобразование хитов события в цилиндрические координаты

Из-за различий в конфигурации детекторов данные для BM@N не нуждаются в переводе в цилиндрические координаты, однако необходимо дополнить подготовку следующим этапом удаления треков с пропусками. Некоторые треки в эксперименте BM@N начинаются со второй или третьей станции или не имеют хитов на одной из промежуточных станций. Такие треки не подходят для предсказания, так как модель продолжает треки последовательно.

Авторами TrackNETv2 была предложена процедура обучения модели с помощью распределения треков по так называемым бакетам (от англ. buckets - вёдра). В эксперименте BM@N присутствуют треки разной длины, от 3 до 9, кроме того, необходимо, чтобы модель была способна предсказывать и неполные треки. При этом в данных содержится большее количество длинных треков по сравнением с количеством коротких. Поэтому часть длинных треков обрезалась до более коротких. Целые треки и части длинных треков одной длины формировали бакеты, которые и использовались в обучении. Эта процедура позволяла получить набор треков, сбалансированный по длине, однако имела несколько критичных проблем.

Во-первых, треки, полученные из нескольких точек более длинного трека, не обладают теми же свойствами, что реальные треки этой длинны из-за разного начального импульса. В результате обучение может приводить к нежелательным результатам: так, сеть может так и не научиться предсказывать действительно короткие треки, так как их количество все ещё мало по отношению к трекам с большим значением начального импульса.

Во-вторых, при использовании бакетов используется информация только для последнего хита кандидата, что как уменьшает возможности для обучения классификатора, так и приводит к тому, что TrackNETv2 достаточно качественно предсказывает лишь последний хит трека.

В связи с вышеизложенным было решено адаптировать при обучении схему многие-ко-многим, более естественную для рекуррентных нейронных сетей [32]. Данная процедура заключается в том, что по входному массиву хитов предсказываются несколько выходов – в нашем случае эллипсов (см. рис. 21).

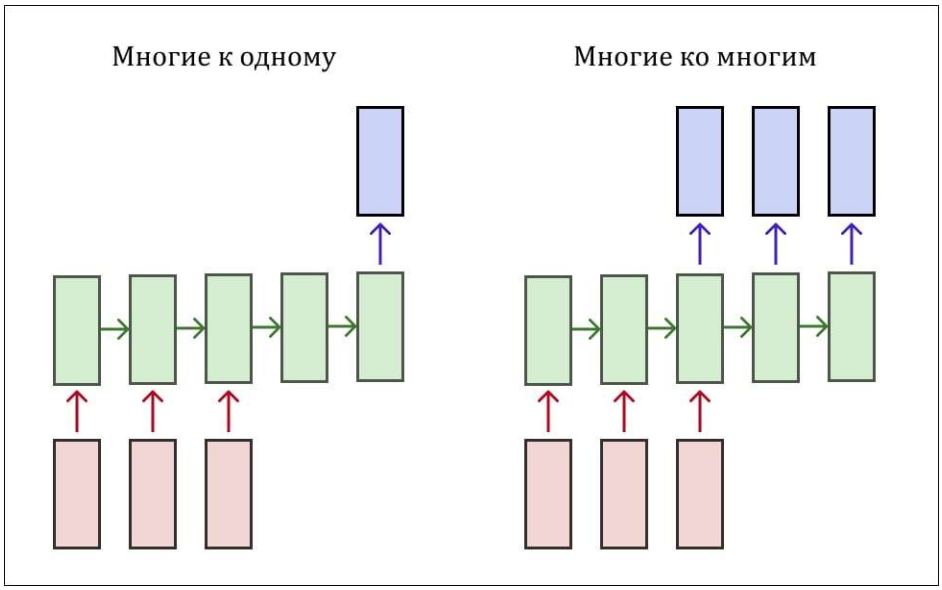

Рис.21 - Иллюстрация предсказания многие-к-одному (реализовалась в исходной схеме обучения TrackNETv2) и многие-ко-многим (реализуется в новой схеме)

В этой процедуре необходимо трансформировать TrackNetLoss так, чтобы она учитывала все выходы от третьей станции до последней. Для этого задаётся маска, зануляющая все лишние выходы сети: после достижения длины трека и до третьей станции. При расчёте ошибки значения ошибок для каждого из выходов сети умножаются на эту маску, после чего усредняются. В результате учитываются все хиты трека, а не только последний, и можно посчитать градиент функции ошибки для оптимизации.

5.2. Подготовка данных для классификатора

Базовая модель и классификатор первого вида не могут быть обучены вместе, так как этот классификатор использует скрытые признаки TrackNETv2 в качестве входных данных, и эти входные данные должны быть неизменными при обучении. Для чистоты эксперимента второй классификатор обучается на

той же выборке, что и первый. Следовательно, необходимо использовать базовую модель в режиме инференса. Это означает, что базовая модель не обучается, а события обрабатываются также, как в «боевых» условиях. Для обучения классификатора необходимо использовать предварительно обученную модель TrackNETv2, разбив проверочный набор на два разных подмножества – одно для обучения классификаторов, а второе для их валидации. Для тестирования связки из двух моделей используется отдельное множество событий, не использовавшееся ранее. Это необходимо для получения «честной» процедуры обучения и проверки классификатора.

Рассмотрим процедуру подготовки данных для классификатора. Эта процедура соответствует стадии инференса для TrackNETv2. Если TrackNETv2 неправильно находит последний хит реального трека или продлевает фальшивый трек, его предсказание отмечается как False, иначе True, и эти метки используются как желаемые классы. Вывод ГРУ и найденная точка сохраняются и передаются в классификатор во время его обучения.

Процедура для BM@N отличается от процедуры подготовки данных для BESIII главным образом за счёт того, что треки могут иметь число хитов меньше, чем число станций детектора – разброс энергий в данном эксперименте значительно выше. Таким образом, для того, чтобы подготовить данные для BESIII, достаточно провести одну итерацию общей процедуры.

Кроме того, классификатор должен быть способен к предсказанию класса для треков разной длины, а базовая модель должна качественно предсказывать следующий хит на разных этапах. Таким образом, стадия инференса TrackNETv2 трансформируется следующим образом:

- Получаются все сочетания хитов на первых двух станциях. Эти сочетания формируют треки-кандидаты.

- Полученные кандидаты подаются на вход базовой модели. Все хиты, попавшие в предсказанные TrackNETv2 эллипсы, объединяются с хитами кандидата, формируя новые кандидаты.
- Если в эллипсы не попало ни одного хита, производится проверка, принадлежит ли эллипс станции. Если эллипс не пересекает плоскость станции и длина кандидата превышает минимальную, то кандидат добавляется к уже отобранным кандидатам, в противном случае отбрасывается.
- В случае, если полученный кандидат является полным реальным треком, ему присваивается позитивный класс, в противном случае, негативный.
- Если реальное завершение трека не попало в эллипс, оно также формирует трек-кандидат с позитивным классом

Выбор такой процедуры обусловлен тем, что реальное завершение трека не обязательно является ближайшим к предсказанному центру эллипса. Кроме того, такая процедура позволяет решить проблему майнинга негативов – если брать только ближайший к центру хит, то теряется большое количество сложных примеров, если же не отфильтровывать хиты, не попавшие в эллипсы, негативные кандидаты будут очень сильно отличаться от положительных, что также приводит к проблемам обучения.

Глава 6. Эксперименты и результаты

6.1. Оценка результатов трекинга

Для обучения классификаторов события делятся на тренировочные и валидационные, после чего все кандидаты подаются на вход модели порциями (батчами) заданного размера.

Для каждого события во время тестирования все модели используются в режиме инференса.

При этом на вход базовой модели подаются все сформированные треки-кандидаты с первых двух станций, а на вход классификатора подаются дополненные кандидаты, попавшие в эллипсы.

Для оценки результатов использовались следующие метрики:

Precision, или точность, характеризует число реальных треков-кандидатов среди тех, для которых был предсказан положительный класс. Эта метрика приведена в формуле:

$$Precision = \frac{n_{predreal|real}}{n_{predreal}} = \frac{TP}{TP + FP},$$

где $n_{predreal|real}$- число истинных треков, для которых предсказан позитивный класс;

$n_{predreal}$-число треков, для которых предсказан позитивный класс;

TP/TN(True positive/True negative) – число истинно положительных/отрицательных примеров,

FP/FN(False positive/False negative) – число ложноположительных/ложноотрицательных примеров

Efficiency, или эффективность, характеризует число реальных треков, которым был присвоен положительный класс, по отношению к полному числу реальных треков с помощью уравнения:

$$Efficiency = \frac{n_{predreal|real}}{n_{real}} = \frac{TP}{TP + FN}$$

Количество нейронов в слое, а также само количество слоёв может привести как к улучшению результатов, так и к их ухудшению в результате

переобучения. В связи с этим были рассмотрены модели с разными модификациями сложности.

6.2. Сравнение результатов экспериментов

6.2.1. BESIII

Рассмотрим результаты вычислений для данного эксперимента. Далее будут использоваться те настройки модель классификатора, функция потерь, параметры классификатора, которые покажут лучшие результаты для этого эксперимента.

Для начала необходимо исследовать влияние выбора функции потерь на предсказания. Это и дальнейшее сравнения проводится для одной и той же базовой модификации TrackNETv2, на основе которого обучены базовые модификации классификаторов. Обучение производится на одном наборе данных, все параметры, кроме описываемых в каждом сравнении, фиксируются.

Таблица 1 иллюстрирует зависимость результатов от функции потерь и модели классификатора.

Таблица 1. Влияние выбора функции потерь на качество трекинга

Модель	Функция потерь	Precision	Recall
Classifier-GRU	Binary Crossentropy	0.54	0.95
Classifier-GRU	Focal Loss	0.68	0.94
Classifier-Coords	Binary Crossentropy	0.66	0.94
Classifier-Coords	Focal Loss	0.42	0.98

Рисунок 22 позволяет сравнить зависимость скорости сходимости обучения от выбора функции потерь. Как видно на рис 22, Focal Loss требует

большего количества эпох по сравнению с кросс-энтропией, но, как видно в таблице, позволяет достичь лучших результатов. Для дальнейших экспериментов была выбрана именно эта функция потерь.

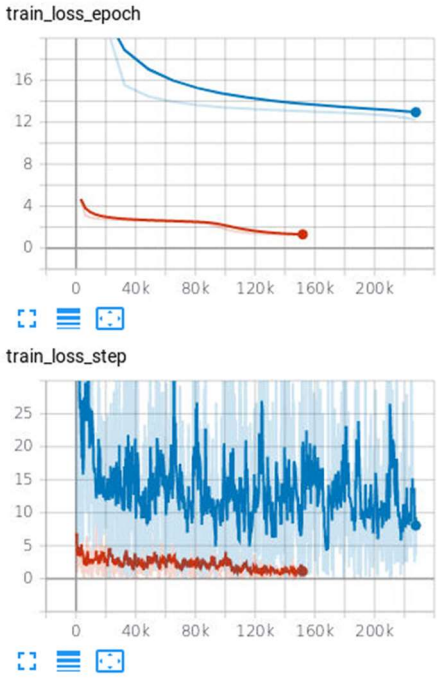

Рис.22 - Сходимость обучения для Focal Loss и Binary Crossentropy. Синим – зависимость значения функции ошибки от времени для кросс-энтропии, красным – для FocalLoss

Далее сравним различные модификации классификаторов (см. приложение 2). В таблице 2 можно видеть результаты обучения различных модификаций после 20 эпох с функцией ошибки Focal Loss.

Таблица 2. Влияние количества нейронов в классификаторе на качество предсказания

Модель	Число нейронов	Precision	Recall
Classifier-Small	1.8 K	0.553	0.97
Classifier	2.4 K	0.677	0.98
Classifier-Big	3.9 K	0.58	0.96

Проведем аналогичное сравнение для второго классификатора. В таблице 3 можно видеть результаты обучения различных модификаций после 20 эпох с функцией ошибки Focal Loss. Как видно в таблице 3, для любых

модификаций этой модели точность гораздо ниже по сравнению со всеми модификациями классификатора на основе признаков скрытого слоя TrackNETv2. Хотя с его помощью можно достичь высоких значений чувствительности, было принято решение в дальнейшем использовать именно классификатор на основе скрытых признаков TrackNETv2.

Таблица 3. Влияние количества нейронов в классификаторе на основе координат на качество предсказания

Модель	Число нейронов	Precision	Recall
Classifier-Small	1 K	0.41	0.98
Classifier-Big	4.1 K	0.58	0.96

Таким образом, в качестве финальной модели был выбран классификатор на основе скрытых признаков модели с функцией ошибки FocalLoss.

Результаты трекинга сильно зависят от выбора порогового значения предсказанной вероятности, при превышении которого трек-кандидат будет классфицирован как положительный. Увеличение порога позволяет отсеять часть фейковых кандидатов, но и ведет к потерям реальных треков. Поскольку нейросетевой трекинг предполагается использовать как генератор для фильтра Калмана, наиболее важно получить максимальный precision при возможном сохранение высокого значения эффективности. Зависимость данных метрик от значения порога можно видеть на рис.23.

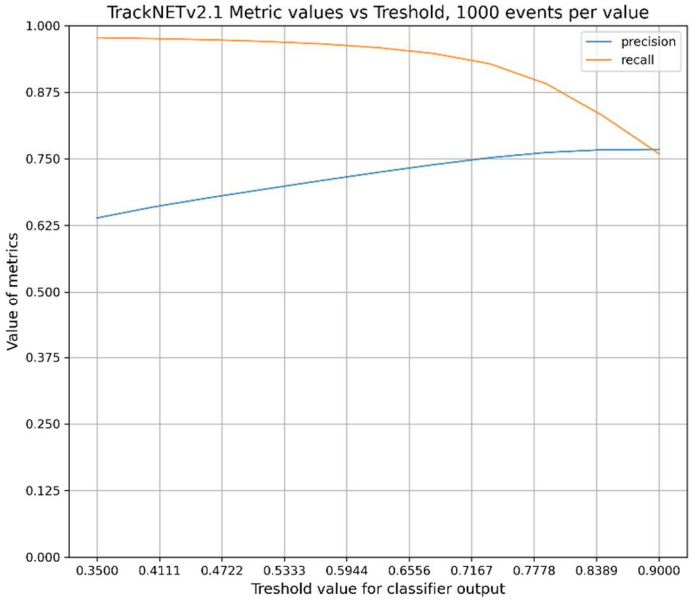

Рис. 23 Зависимость качества трекинга от выбора порогового значения вероятности для классификатора. Желтым – зависимость для эффективности, синим – для точности

6.2.2. BM@N

Рассмотрим результаты различных экспериментов для данного эксперимента. Базовая модель достигает удовлетворительных результатов на данном эксперименте и без использования классификатора, однако необходимо оценить способность классификатора к обобщению и на более сложные эксперименты.

Базовая модель при обучении при обучении для этого эксперимента достигает значений точности в 0.98 и чувствительности в 0.54. И хотя эти значения уже удовлетворяют требованиям физиков, они могут быть улучшены с помощью классификации.

Для начала также необходимо оценить значение функции потерь.

Таблица 4 иллюстрирует зависимость результатов от функции потерь и модели классификатора. Как видно в таблице, выбор FocalLoss позволяет позволяет достичь лучших результатов и для этого эксперимента.

Таблица 4. Влияние выбора функции потерь на качество трекинга в BM@N

Модель	Функция потерь	Precision	Recall
Classifier-GRU	Binary Crossentropy	0.56	0.95
Classifier-GRU	Focal Loss	0.51	0.96

Как видно, использование классификатора также позволило достичь более высокого значения Precision по сравнению с использованием только базовой модели, что подтверждает актуальность перехода на новый метод для локального трекинга.

6.3. Анализ результатов

Для более глубокого анализа результатов использована зависимость эффективности от импульса треков. Зависимость от импульса наиболее интересна физикам, так как позволяет оценить, удовлетворяет ли модель их требованиям. Так, для относительно низких значений импульса эффективность менее важна, так как такие частицы не представляют интереса для исследователей на данных детекторах.

Важнейшей метрикой здесь является зависимость от поперечного импульса (ГэВ / с), которая характерна для энергии частицы:

$$p_t = \sqrt{p_x^2 + p_y^2}$$

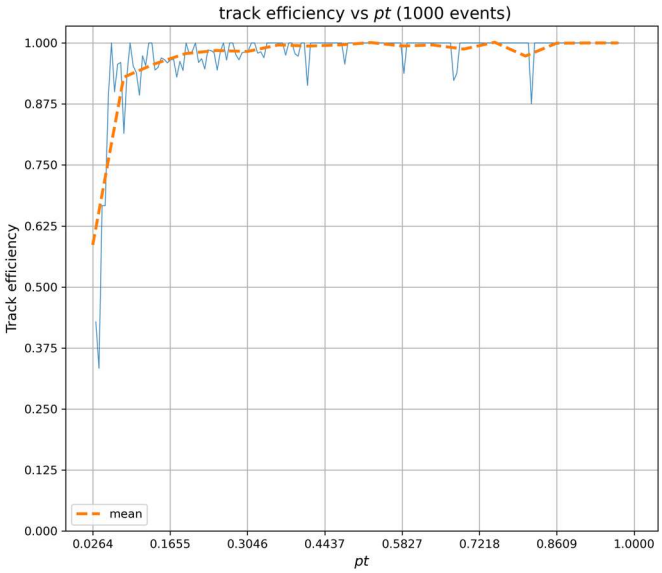

Рис. 24. – Зависимость эффективности распознавания от импульса

Другой важной метрикой является зависимость от полярного угла, что характерно для кривизны пути:

$$\cos(\theta) = \frac{p_z}{\sqrt{p_x^2 + p_y^2 + p_z^2}}$$

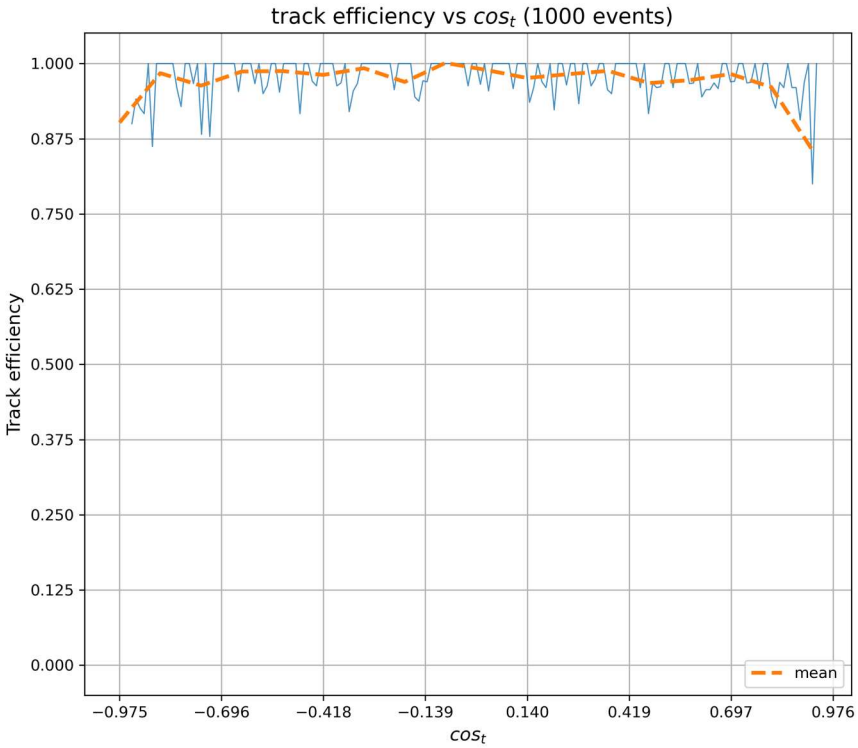

Рис. 25. – Зависимость эффективности распознавания от угла трека

Еще одна метрика - зависимость от азимутального угла:

$$\varphi = arctan2\,(p\_x, p\_y)$$

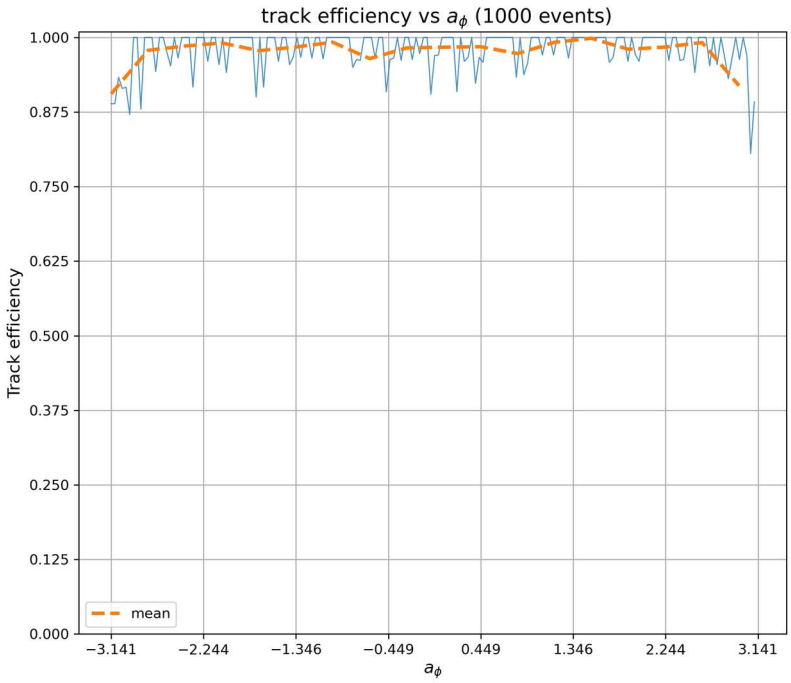

Рис. 25. Зависимость эффективности распознавания от

Рисунки 24-26 иллюстрируют распределения результатов для различных значений энергии частиц в эксперименте BESIII. Как вы можете видеть на рисунке 24 и 25, нет никакой зависимости эффективности от момента и полярного угла частицы. Эти две характеристики связаны, потому что частица с меньшей начальной энергией имеет трек с большей кривизной. Рисунки 26, 24 и 25 показывают, что модель способна обрабатывать частицы с различными начальными значениями энергии и массы.

Еще одна большая проблема связана с архитектурой эксперимента. Нам нужно использовать цилиндрические координаты, поэтому некоторые треки пересекают границу 2π. Рисунок 26 показывает, что эта особнность эксперимента не влияет на TrackNETv3.

Как вы можете видеть на рисунках 24-26, наша модель показывает удовлетворительные результаты для всех этих зависимостей.

Заключение

В данной работе была решена задача трекинга частиц в экспериментах с детекторами на основе ГЭУ на примере BESIII и BM@N. Особенность этих экспериментов заключается в том, что детекторы в них регистрируют не только пролетающие сквозь своё внутреннее пространство частицы, но и большое количество вторичных частиц или попросту шума, называемых фейками. В результате методы типа фильтра Калмана перестают удовлетворять требованиям таких экспериментов по скорости.

В данной работе был развит нейросетевой подход для трекинга частиц. Была предложена модификация модели TrackNETv2. Так, исследования показали, что TrackNETv2 не может фильтровать ложные треки для детекторов с небольшим количеством станций обнаружения, как, например, 3 в эксперименте BESIII. Всего один процент ложных треков отбрасывался, что не удовлетворяло требованиям физиков по качеству трекинга. Эта модель была изменена путем добавления части классификатора для фильтрации фейковых треков-кандидатов, что расширило ее возможности обобщения. Также была существенно переработана процедура обучения и тестирования данной модели.

Разработанный подход был также адаптирован для эксперимента BM@N, что привело к увеличению качества трекинга на данном эксперименте и подтвердило целесообразность использования данного метода локального трекинга не только в экспериментах с низкими энергиями взаимодействия, но и в экспериментах с более высокими энергиями.

Для разработки системы локального трекинга использовалась и дополнялась открытая библиотека для нейросетевого трекинга Ariadne. При решении задач, описанных в данной работе, библиотека была дополнена модулями для подготовки данных, их трансформации, режима инференса. Кроме того, как базовая модель TrackNETv2, так и новая модель TrackNETv3

вошли в набор готовых моделей этой библиотеки, а методы подготовки данных для экспериментов BM@N и BESIII также стали частью этой библиотеки.

Список источников

1. J.Grey, David T.Liu, M.Nieto-Santiseban, A.Szalay, D.J.DeWitt, G.Heber, Scientific data management in the coming decade, ACM SIGMOD Record, Volume 34, Issue 4, December 2005

2. Нобелевскую премию по физике-2013 дали за бозон Хиггса // Русская служба Би-би-си, 8 октября 2013

3. F.Sauli,GEM:A new concept for electron amplification in gas detectors,Nucl. Instr. And Meth. A 386 (1997) 53l-534

4. Ablikim M. et al. [BESIII Collaboration] Design and Construction of the BESIII Detector // Nucl. Instrum. Meth. A 614. — 2010. — P. 345–399.BESIII Collaboration, Design and Construction of the BESIII Detector, arXiv:0911.4960v1

5. Asner D. M. et al. Physics at BES-III // Int. J. Mod. Phys. A 24. — 2009. — S1-794.

6. Denisenko I., Ososkov G. Primary processing of hits in cylindrical GEM tracker at the BESIII experiment // AIP Conf. Proc. 2163. — 2019. — P. 030002.

7. CGEM-ITgroup ,The Cylindrical GEM Inner Tracker of the BESIII experiment: prototype test beam results, Instrumentation for Colliding Beam Physics (INSTR-17) (27February2017-3March2017)

8. Method and means for recognizing complex patterns: US Patent 3069654 / Hough P. V.; Filing Date: 25/03/1960; Publication Date: 18/12/1962; Assignee: U.S. Patent and Trademark Office

9. Strandlie A. Track and vertex reconstruction: From classical to adaptive methods / A. Strandlie, R. Frühwirth // Reviews of Modern Physics, 2010. – Vol. 82. – No. 2. – P. 1419

10. Frühwirth R. Application of Kalman filtering to track and vertex fitting // Nuclear Instruments and Methods in Physics Research Section A: Accelerators,

Spectrometers, Detectors and Associated Equipment. – 1987. – Vol. 262. – No. 2–3. – P. 444–450.

11. Mankel R. A Concurrent track evolution algorithm for pattern recognition in the HERA-B main tracking system // Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. – 1997. – Vol. 395. – No. 2. – P. 169–184.

12. Abt I. CATS: a cellular automaton for tracking in silicon for the HERA-B vertex detector / I. Abt et al. // Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. – 2002. – Vol. 489. – No. 1-3. – P. 389–405.

13. Gardner M. Mathematical Games: The fantastic combinations of John Conway's new solitaire game «life» // Scientific American, 1970. – Vol. 223. – No. 4. – P. 120–123

14. Kisel I. V. Application of neural networks in experimental physics / I. V. Kisel, V. N. Neskromnyj, G. A. Ososkov // Fizika Ehlementarnykh Chastits i Atomnogo Yadra. – 1993. – Vol. 24. – No. 6. – P. 1551–1595.

15. Denby B. Neural networks in high energy physics: a ten year perspective // Computer Physics Communications. – 1999. – Vol. 119. – No. 2-3. – P. 219–231.

16. Hopfield J. J. "Neural" computation of decisions in optimization problems / J. J. Hopfield, D. W. Tank // Biological cybernetics. – 1985. – Vol. 52. – No. 3. – P. 141–152

17. Peterson C. Track finding with neural networks // Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. – 1989. – Vol. 279. – No. 3. – P. 537–545.

18. Denby B. Neural networks and cellular automata in experimental high energy physics // Computer Physics Communications. – 1988. – Vol. 49. – No. 3. – P. 429–448.

19. Ососков Г. А. и др. Использование нейронных сетей для улучшения интерпретации эксперимента EXCHARM // Математическое моделирование. – 1999. – Т. 11. – №. 10. – С. 116–126

20. Durbin R. An analogue approach to the travelling salesman problem using an elastic net method / R. Durbin, D. Willshaw // Nature. – 1987. – Vol. 326. – No. 6114. – P. 689

21. Kisel I. Elastic net for broken multiple scattered tracks / V. Kovalenko, I. Kisel // Computer physics communications. – 1996. – Vol. 98. – No. 1-2. – P. 45-51.].

22. Ососков Г. А. Современные методы обработки экспериментальных данных в физике высоких энергий / Г. А. Ососков, А. Полянский, И. В. Пузынин // Физика элементарных частиц и атомного ядра. – 2002. – Т. 33. – №. 3.

23. Geometric deep learning: going beyond Euclidean data /Michael M. Bronstein, Joan Bruna, Yann LeCun et al. // CoRR. — 2016. — Vol. abs/1611.08097. — 1611.08097.

24. Farrell Steven et al. Novel deep learning methods for track reconstruction // 4th International Workshop Connecting The Dots 2018 (CTD2018) Seattle, Washington, USA, March 20-22, 2018. — 2018. — 1810.06111.

25. Shchavelev E. et al. Tracking for BM@N GEM Detector on the Basis of Graph Neural Network / E. Shchavelev, P. Goncharov, G. Ososkov, D. Baranov //Proceedings of the 27th Symposium on Nuclear Electronics and Computing (NEC 2019), Budva, Montenegro. – 2019. – Vol. 2507. – P. 280-284.

26. Gyulassy M. Elastic tracking and neural network algorithms for complex pattern recognition / M. Gyulassy, M. Harlander // Computer Physics Communications. – 1991. – Vol. 66. – No. 1. – P. 31–46.

27. Goodfellow I. at al. Deep Learning. MIT Press. 2016. P. 180–184. ISBN 978-0-26203561-3

28. Baranov D. Novel approach to the particle track reconstruction based on deep learning methods / D. Baranov, S. Mitsyn, G. Ososkov, P. Goncharov, A. Tsytrinov // Selected Papers of the 26th International Symposium on Nuclear Electronics and Computing (NEC 2017), Budva, Montenegro, September 25–29, 2017. – CEUR Proceedings. – Vol. 2023. – P. 37–45

29. Baranov D. Catch and Prolong: recurrent neural network for seeking track-candidates / D. Baranov, G. Ososkov, P. Goncharov, A. Tsytrinov // The XXII International Scientific Conference of Young Scientists and Specialists (AYSS-2018). – EPJ Web of Conferences. – EDP Sciences, 2019. – Vol. 201. – P. 05001

30. Baranov D. THE PARTICLE TRACK RECONSTRUCTION BASED ON DEEP NEURAL NETWORKS / D. Baranov, S. Mitsyn, P. Goncharov, G. Ososkov // 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018). – EPJ Web of Conferences. – EDP Sciences, 2019 (in print), препринт arxiv 1812.03859

31. van Rossum G. Python tutorial. — 1995.

32. Николенко С. И., Кадурин А. А., Архангельская Е. О. Глубокое обучение, Питер, 2019 - 480 с

32. CGEM-ITgroup ,The Cylindrical GEM Inner Tracker of the BESIII experiment: prototype test beam results, Instrumentation for Colliding Beam Physics (INSTR-17) (27 February 2017-3 March 2017)

33. C. A. Goodfellow I., Bengio Y., “Deep learning,” (MIT Press, 2016) Chap. Softmax Units for Multinoulli Output Distributions, p. 180–184.

34. Ling, Charles X., and Chenghui Li. "Data mining for direct marketing: Problems and solutions." Kdd. Vol. 98. 1998

35. Murphy, Kevin (2012). Machine Learning: A Probabilistic Perspective. MIT. ISBN 978-0262018029

36. Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollár, Focal Loss for Dense Object Detection. ICCV 2017

37. J.Johnson, M.Douze, Billion-scale similarity search with GPUs, arXiv preprint arXiv:1702.08734, 2017

Приложение 1 Листинги трансформаций

```
import logging
from typing import Optional, List
from copy import deepcopy

import gin
import pandas as pd
import numpy as np
from sklearn.preprocessing import (
    StandardScaler,
    MinMaxScaler,
    Normalizer
)

LOGGER = logging.getLogger('ariadne.transforms')

class Compose:
    """Composes several transforms together. Mostly copied from torchvision.
    Args:
        transforms (list of ``Transform`` objects): list of
            transforms to compose.

    Example:
        >>> Compose([
        >>>     transforms.StandardScale(),
        >>>     transforms.ToCylindrical(),
        >>> ])
    """

    def __init__(self, transforms):
        self.transforms = transforms

    def __call__(self, data: pd.DataFrame, preserve_index=True):
        if preserve_index:
            data['index'] = data.index
        for t in self.transforms:
            data = t(data)
            if data.empty:
                LOGGER.warning(f'{t.__class__.__name__} returned empty data.
'
                               'Skipping all further transforms')
                return data
        return data

    def __repr__(self):
        """
        Returns:
            str: formatted strings with class_names,
                parameters and some statistics for each class
        """
        transforms_str = ',\n'.join(
            [f'  {t.__class__.__name__}' for t in self.transforms])
        fmt_str = f'{self.__name__}(\n{transforms_str}\n)'
        return fmt_str


class BaseTransformer(object):
    """Base class for transforms
    # Args:
         columns (list or tuple, ['x', 'y', 'z'] by default): Columns to
transform
```

```
        drop_old (boolean, True by default): If True, original data is
discarded,
                                             else preserved in columns with
suffix '_old'
        keep_fakes (boolean, True by default): If True, hits with no track
are preserved
    """

    def __init__(self, drop_old=False, columns=('x', 'y', 'z'),
track_col='track', event_col='event',
                 station_col='station'):
        self.drop_old = drop_old
        self.columns = columns
        self.fakes = None
        self.event_column = event_col
        self.track_column = track_col
        self.station_column = station_col
        assert len(columns) == 3, "Columns must be list or tuple of length 3"

    def transform_data(self, data, normed):
        for i in range(3):
            if not self.drop_old:
                data.loc[:, self.columns[i] + '_old'] = 1
                data.loc[:, self.columns[i] + '_old'] = deepcopy(data.loc[:,
self.columns[i]])

            data.loc[:, self.columns[i]] = normed[i]
        return data

    def drop_fakes(self, data):
        return data.loc[data[self.track_column] != -1, :]

    def get_num_fakes(self):
        if self.fakes:
            return len(self.fakes)
        return 0

    def add_fakes(self, data, fakes):
        return pd.concat([data, fakes], axis=0, ignore_index=True)



class BaseScaler(BaseTransformer):
    """Base class for scalers.
     # Args:
        scaler (function or method pd.DataFrame -> iterable of pd.Series):
scaler with fit_predict method
        columns (list or tuple, ['x', 'y', 'z'] by default): Columns to
scale
        drop_old (boolean, True by default): If True, unscaled data is
discarded,
                                             else preserved in columns with
suffix '_old'
    """

    def __init__(self, scaler, drop_old=True, columns=('x', 'y', 'z')):
        super().__init__(drop_old=drop_old, columns=columns)
        self.scaler = scaler

    def __call__(self, data):
        """
        # Args:
            data (pd.DataFrame):  to clean up.
```

```
        # Returns:
            data (pd.DataFrame): transformed dataframe
        """
        norms = pd.DataFrame(self.scaler.fit_transform(data[[self.columns[0],
self.columns[1], self.columns[2]]]))
        data = self.transform_data(data=data, normed=norms)
        return data

    def __repr__(self):
        return (f'{"-" * 30}\n'
                f'{self.__class__.__name__} with scaler: {self.scaler}'
                f'{"-" * 30}\n')


class BaseFilter(BaseTransformer):
    """Base class for all filtering transforms
     # Args:
         filter_rule (function or method pd.DataFrame -> iterable of
pd.Series): Function, which
                                    convertes data, returned value must be
iterable with pd.Series values (list etc)
         keep_fakes (boolean, True by default): If True, hits with no track
are preserved
         station_col (string, 'station' by default): column with station
identifiers
         event_col (string, 'event' by default): column with event
identifiers
         track_col (string, 'track' by default): column with station
identifiers
    """

    def __init__(self, filter_rule, num_stations=None, keep_filtered=True,
station_col='station', track_col='track',
                 event_col='event'):
        super().__init__(station_col=station_col, track_col=track_col,
event_col=event_col)
        self.num_stations = num_stations
        self._broken_tracks = None
        self._num_broken_tracks = None
        self.filter_rule = filter_rule
        self.keep_filtered = keep_filtered


    def __call__(self, data):
        """
        # Args:
            data (pd.DataFrame):  to clean up.
        # Returns:
            data (pd.DataFrame): transformed dataframe
        """
        fakes = data.loc[data[self.track_column] == -1, :]
        data = self.drop_fakes(data)
        tracks = data.groupby([self.event_column, self.track_column])
        if self.num_stations is None:
            self.num_stations = tracks.size().max()
        good_tracks = tracks.filter(self.filter_rule)
        broken = list(data.loc[~data.index.isin(good_tracks.index)].index)
        self._broken_tracks = data.loc[broken, [self.event_column,
self.track_column, self.station_column]]
        self._num_broken_tracks = len(self._broken_tracks[[self.event_column,
self.track_column]].drop_duplicates())
        if self.keep_filtered:
            data.loc[~data.index.isin(good_tracks.index), 'track'] = -1
```

```
        else:
            data = data.loc[data.index.isin(good_tracks.index), :]
        return self.add_fakes(data, fakes)

    def get_broken(self):
        return self._broken_tracks

    def get_num_broken(self):
        return self._num_broken_tracks

    def __repr__(self):
        return (f'{"-" * 30}\n'
                f'{self.__class__.__name__} with filter_rule:
{self.filter_rule}\n'
                f'{"-" * 30}\n')



class BaseCoordConverter(BaseTransformer):
    """Base class for coordinate convertions

    # Args:
        convert_function(function or method pd.DataFrame -> iterable of
pd.Series): Function, which
        convertes data, returned value must be iterable with pd.Series
values (list etc)
        drop_old (boolean, False by default): If True, old columns are
discarded from data
        from_columns (list or tuple of length 3, ['x', 'y', 'z'] by
default): list of original features
        to_columns (list or tuple of length 3, ['r', 'phi', 'z'] by
default): list of features to convert to
    """

    def __init__(self, convert_function, drop_old=False, from_columns=('x',
'y','z'),
                 to_columns=('r', 'phi', 'z'), postfix='general_convert'):
        assert len(from_columns) == 3, 'To convert coordinates, you need 3
old columns'
        assert len(to_columns) == 3, 'To convert coordinates, you need 3 new
columns'
        super().__init__(drop_old=drop_old, columns=to_columns)
        self.convert_function = convert_function
        self.range_ = {}
        self.from_columns = from_columns
        self.to_columns = to_columns
        self.postfix = postfix


    def __call__(self, data):
        """
        # Args:
            data (pd.DataFrame):  to clean up.
        # Returns:
            data (pd.DataFrame): transformed dataframe
        """
        # assert type(data) == pd.core.frame.DataFrame, "unsupported data
format"
        self.get_ranges(data, self.from_columns)
        converted = self.convert_function(data)
        if not self.drop_old:
            for col in self.from_columns:
                data[col + '_' + self.postfix] = data[col]
```

```
        if self.drop_old:
            for col in self.from_columns:
                del data[col]
        for i in range(len(self.to_columns)):
            data.loc[:, self.to_columns[i]] = converted[i]
        self.get_ranges(data, self.to_columns)
        return data

    def get_ranges(self, data, columns):
        for col in columns:
            try:
                self.range_[col] = (min(data[col]), max(data[col]))
            except:
                'This column is not used'

    def get_ranges_str(self):
        return '\n'.join([f'{i}: from {j[0]} to {j[1]}' for i, j in
self.range_.items()])

    def __repr__(self):
        return(f'{"-" * 30}\n'
               f'{self.__class__.__name__} with convert_function: '
               f'{self.convert_function}\n'
               f'{"-" * 30}\n')

@gin.configurable
class PreserveOriginal:
    """Preserves original state of given columns.
    May be needed to use original state of column
    in future transforms or tests

    # Args:
         columns (list or tuple, None by default): Columns to keep state of.
    """

    def __init__(self, columns=None):
        self.columns = columns

    def __call__(self, data):
        """
        # Args:
            data (pd.DataFrame):  to clean up.
        # Returns:
            data (pd.DataFrame): transformed dataframe
        """
        if not self.columns:
            self.columns = data.columns
        for col in self.columns:
            data.loc[:, col + '_original'] = data.loc[:, col]
        return data

    def __repr__(self):
        return (f"{'-' * 30}\n"
               f'{self.__class__.__name__} ceeping original state of columns:
{self.columns} \n'
               f"{'-' * 30}\n")

@gin.configurable
class BakeStationValues:
    """Coverts z coordinate of hit to given value for each station station.
    Nessesary for BM@N, becaule strips have different depths.
     # Args:
         scaler (function or method pd.DataFrame -> iterable of pd.Series):
```

```
scaler with fit_predict method
        columns (list or tuple, ['x', 'y', 'z'] by default): Columns to
scale
        drop_old (boolean, True by default): If True, unscaled data is
discarded,
                                             else preserved in columns with
suffix '_old'
    """

    def __init__(self, values, col='z', station_col='station'):
        self.station_values = values
        self.station_column = station_col
        self.column = col

    def __call__(self, data):
        """
        # Args:
            data (pd.DataFrame):  to clean up.
        # Returns:
            data (pd.DataFrame): transformed dataframe
        """
        for i, value in self.station_values.items():
            data.loc[data['station'] == i, 'z'] = value
        return data

    def __repr__(self):
        return (f'{"-" * 30}\n'
                f'{self.__class__.__name__} with scaler: {self.scaler}'
                f'{"-" * 30}\n')


@gin.configurable
class StandardScale(BaseScaler):
    """Standardizes coordinates by removing the mean and scaling to unit
variance
    # Args:
        drop_old (boolean, True by default): If True, unscaled features are
dropped from dataframe
        with_mean (boolean, True by default): If True, center the data before
scaling
        with_std (boolean, True by default): If True, scale the data to unit
variance (or equivalently, unit standard deviation).
        columns (list or tuple, ('x', 'y', 'z') by default): Columns to
Standardize
    """

    def __init__(self, drop_old=True, with_mean=True, with_std=True,
columns=('x', 'y', 'z')):
        self.with_mean = with_mean
        self.with_std = with_std
        self.scaler = StandardScaler(with_mean, with_std)
        super().__init__(self.scaler, drop_old, columns)

    def __repr__(self):
        return (f"{'-' * 30} \n"
                f"{self.__class__.__name__} with scaling_parameters:
drop_old={self.drop_old}, "
                f"with_mean={self.with_mean},with_std={self.with_std} \n"
                f"{'-' * 30} \n"
                f" Mean: {self.scaler.mean_} \n Var: {self.scaler.var_} \n
Scale: {self.scaler.scale_} ")
```

```
@gin.configurable
class MinMaxScale(BaseScaler):
    """Transforms features by scaling each feature to a given range.
     # Args:
        drop_old (boolean, True by default): If True, unscaled features are
dropped from dataframe
        feature_range (Tuple (min,max), default (0,1)): Desired range of
transformed data.
        columns (list or tuple, ('x', 'y', 'z') by default): Columns to
Standardize
    """

    def __init__(self, drop_old=True, feature_range=(0, 1), columns=('x',
'y', 'z')):

        assert feature_range[0] < feature_range[1], 'minimum is not smaller
value then maximum'
        self.feature_range = feature_range
        self.scaler = MinMaxScaler(feature_range=feature_range)
        super().__init__(self.scaler, drop_old, columns=columns)

    def __repr__(self):
        return (f"{'-' * 30}\n"
                f"{self.__class__.__name__} with parameters: "
                f"drop_old={self.drop_old},
feature_range={self.feature_range} \n"
                f"{'-' * 30}\n"
                f" Data min: {self.scaler.data_min_} \n Data max:
{self.scaler.data_max_} \n Scale: {self.scaler.scale_}")


@gin.configurable
class Normalize(BaseScaler):
    """Normalizes samples individually to unit norm.
    Each sample (i.e. each row of the data matrix) with at least one non zero
component is rescaled independently of
    other samples so that its norm (l1, l2 or inf) equals one.

      # Args:
        drop_old (boolean, True by default): If True, unscaled features are
dropped from dataframe
        norm ('l1', 'l2', or 'max' ('l2' by default)): The norm to use to
normalize each non zero sample.
                                If norm='max' is used, values will be rescaled
by the maximum of the absolute values.
        columns (list or tuple, ('x', 'y', 'z') by default): Columns to
Standardize
    """

    def __init__(self, drop_old=True, norm='l2', columns=('x', 'y', 'z')):
        self.norm = norm
        self.scaler = Normalizer(norm=norm)
        super().__init__(self.scaler, drop_old, columns)

    def __repr__(self):
        return (f"{'-' * 30}\n"
                f"{self.__class__.__name__} with parameters:
drop_old={self.drop_old}, norm={self.norm} \n"
                f"{'-' * 30}\n")


@gin.configurable
class ConstraintsNormalize(BaseTransformer):
```

```
    """Normalizes samples using station given characteristics  or computes
them by call.
    If you need to compute characteristics, you can use MinMaxScale too
(maybe better)
    Each station can have its own constraints or global constrains.
      Args:
        drop_old (boolean, True by default): If True, unscaled features are
dropped from dataframe
        columns (list or tuple, ('x', 'y', 'z') by default): Columns to scale
        margin (number, positive): margin applied to stations (min = min-
margin, max=max+margin)
        constraints (dict, None by deault) If None, constraints are computed
using dataset statistics.
        use_global_constraints (boolean, True by default) If True, all data
is scaled using given global constraints.

    If use_global_constraints is True and constraints is not None,
constraints must be {column:(min,max)},
    else it must be {station: {column:(min,max)}}.

    Station keys for constraints must be in dataset. Number of constraints
for each column must be 2.
    Number of constraints must be the same as number of columns
    """

    def __init__(self, drop_old=True, columns=('x', 'y', 'z'), margin=1e-3,
use_global_constraints=True,
                 constraints=None):
        super().__init__(drop_old=drop_old, columns=columns)
        assert margin > 0, 'Margin is not positive'
        self.margin = margin
        self.use_global_constraints = use_global_constraints
        self.constraints = constraints
        if constraints is not None:
            if use_global_constraints:
                for col in columns:
                    assert col in constraints.keys(), f'{col} is not in
constraint keys {constraints.keys()}'
                    assert len(constraints[col]) == 2, f'Not applicable
number of constraints for column {col}'
                    assert constraints[col][0] < constraints[col][
                        1], f'Minimum is not smaller than maximum for column
{col}'
            else:
                for key, constraint in constraints.items():
                    for col in columns:
                        assert col in constraint.keys(), f'{col} is not in
constraint keys for station {key}'
                        assert len(constraint[
                                       col]) == 2, f'Not applicable number of
constraints for column {col} and station {key}'
                        assert constraint[col][0] < constraint[col][
                            1], f'Minimum is not smaller than maximum for
column {col} and station {key}'


    def __call__(self, data):
        """
        # Args:
            data (pd.DataFrame):  to clean up.
        # Returns:
            data (pd.DataFrame): transformed dataframe
        """
```

```
        if self.constraints is None:
            self.constraints = self.get_stations_constraints(data)
        if self.use_global_constraints:
            global_constrains = {}
            for col in self.columns:
                global_min = self.constraints[col][0]
                global_max = self.constraints[col][1]
                assert global_min < global_max, f"global_min should be <
global_max {global_min} < {global_max}"
                global_constrains[col] = (global_min, global_max)
            x_norm, y_norm, z_norm = self.normalize(data, global_constrains)
            data = super().transform_data(data, [x_norm, y_norm, z_norm])
        else:
            ##assert all([station in data['station'].unique() for station in
             #           self.constraints.keys()]), "Some station keys in
constraints are not presented in data. Keys: " \
            #
f"{data['station'].unique()}; data keys: {self.constraints.keys()}"

            for station in list(data['station'].unique()):
                #print(data['station'].unique())
                group = data.loc[data['station'] == station,]
                x_norm, y_norm, z_norm = self.normalize(group,
self.constraints[station])
                data.loc[data['station'] == station, :] = \
                    self.transform_data_by_group(data['station'] == station,
data,
                                                 [x_norm, y_norm,
z_norm])

        return data

    def transform_data_by_group(self, grouping, data, normed):
        for i in range(3):
            assert self.drop_old, "Saving old data is not supported for now"
            data.loc[grouping, self.columns[i]] = normed[i]
        return data

    def get_stations_constraints(self, df):
        groups = df['station'].unique()

        station_constraints = {}
        for station_num in groups:
            group = df.loc[df['station'] == station_num,]
            min_x, max_x = min(group[self.columns[0]]) - self.margin,
max(group[self.columns[0]]) + self.margin
            min_y, max_y = min(group[self.columns[1]]) - self.margin,
max(group[self.columns[1]]) + self.margin
            min_z, max_z = min(group[self.columns[2]]) - self.margin,
max(group[self.columns[2]]) + self.margin
            station_constraints[station_num] = {self.columns[0]: (min_x,
max_x),
                                                self.columns[1]: (min_y,
max_y),
                                                self.columns[2]: (min_z,
max_z)}
        return station_constraints

    def normalize(self, df, constraints):
        x_min, x_max = constraints[self.columns[0]]
        y_min, y_max = constraints[self.columns[1]]
        z_min, z_max = constraints[self.columns[2]]
        assert all(df[self.columns[0]].between(x_min, x_max)), \
```

```
            f'Some values in column {self.columns[0]} are not in
{constraints[self.columns[0]]}'
        x_norm = 2 * (df[self.columns[0]] - x_min) / (x_max - x_min) - 1
        assert all(df[self.columns[1]].between(y_min, y_max)), \
            f'Some values in column {self.columns[1]} are not in
{constraints[self.columns[1]]}'
        y_norm = 2 * (df[self.columns[1]] - y_min) / (y_max - y_min) - 1
        assert all(df[self.columns[2]].between(z_min, z_max)), \
            f'Some values in column {self.columns[2]} are not in
{constraints[self.columns[2]]}'
        z_norm = 2 * (df[self.columns[2]] - z_min) / (z_max - z_min) - 1
        return x_norm, y_norm, z_norm

    def __repr__(self):
        return (f"{'-' * 30}\n"
                f"{self.__class__.__name__} with parameters: "
                f"drop_old={self.drop_old}, "
                f"use_global_constraints={self.use_global_constraints}\n"
                f"{' '*20}margin={self.margin}, columns={self.columns}\n"
                f"{'-'*30}\nconstraints are: {self.constraints}")


@gin.configurable
class DropShort(BaseFilter):
    """Drops tracks with num of points less then given from data.
      # Args:
        num_stations (int, default None): Desired number of stations
(points). If None, maximum stations number for one track is taken from data.
        keep_fakes (bool, default True): If True, points with no tracks are
preserved, else they are deleted from data.
        station_column (str, 'station' by default): Event column in data
        track_column (str, 'track' by default): Track column in data
        event_column (str, 'event' by default): Station column in data
    """

    def __init__(self, num_stations=None, keep_filtered=True,
station_col='station', track_col='track', event_col='event'):
        self.num_stations = num_stations
        self.broken_tracks_ = None
        self.num_broken_tracks_ = None
        self.filter = lambda x: len(x) >= self.num_stations
        super().__init__(self.filter, num_stations=num_stations,
station_col=station_col, track_col=track_col,
                         event_col=event_col, keep_filtered=keep_filtered)


    def __repr__(self):
        return(f'{"-" * 30}\n'
               f'{self.__class__.__name__} with parameters:
num_stations={self.num_stations}, '
               f'    track_column={self.track_column},
station_column={self.station_column}, '
               f'event_column={self.event_column}\n'
               f'{"-" * 30}\n'
               f'Number of broken tracks: {self.get_num_broken()} \n')


@gin.configurable
class DropSpinningTracks(BaseFilter):
    """Drops tracks with points on same stations (e.g. (2,2,2) or (1,2,1)).
      # Args:
        keep_fakes (bool, True by default ): If True, points with no tracks
are preserved, else they are deleted from data.
```

```
        station_col (str, 'station' by default): Event column in data
        track_col(str, 'track' by default): Track column in data
        event_col (str, 'event' by default): Station column in data
    """

    def __init__(self, keep_filtered=True, station_col='station',
track_col='track', event_col='event'):

        self.filter = lambda x: x[self.station_column].unique().shape[0] ==
x[self.station_column].shape[0]
        super().__init__(self.filter, station_col=station_col,
track_col=track_col, event_col=event_col,
                         keep_filtered=keep_filtered)

    def __repr__(self):
        return(f'{"-" * 30}\n'
               f'{self.__class__.__name__} with parameters:'
               f'    track_column={self.track_column},
station_column={self.station_column}, event_column={self.event_column}\n'
               f'{"-" * 30}\n'
               f'Number of broken tracks: {self.get_num_broken()} \n')

@gin.configurable
class DropTracksWithHoles(BaseFilter):
    """Drops tracks with points on same stations (e.g. (2,2,2) or (1,2,1)).
      # Args:
        keep_fakes (bool, True by default ): If True, points with no tracks
are preserved, else they are deleted from data.
        station_col (str, 'station' by default): Event column in data
        track_col(str, 'track' by default): Track column in data
        event_col (str, 'event' by default): Station column in data
    """

    def __init__(self,
                 keep_filtered=True,
                 station_col='station',
                 track_col='track',
                 event_col='event',
                 min_station_num=0):

        self.filter = lambda x: x[self.station_column].values.shape[0] == \
                                len(np.arange(min_station_num,
int(x[self.station_column].max())))+1
        super().__init__(self.filter, station_col=station_col,
track_col=track_col, event_col=event_col,
                         keep_filtered=keep_filtered)

    def __repr__(self):
        return(f'{"-" * 30}\n'
               f'{self.__class__.__name__} with parameters:'
               f'    track_column={self.track_column},
station_column={self.station_column}, event_column={self.event_column}\n'
               f'{"-" * 30}\n'
               f'Number of broken tracks: {self.get_num_broken()} \n')


@gin.configurable
class DropFakes(BaseTransformer):
    """Drops points without tracks (marked as -1).
    Args:
        track_col (str, 'track' by default): Track column in data
    """
```

```
    def __init__(self, track_col='track'):
        super().__init__(track_col=track_col)
        self._num_fakes = None
        self.track_col = track_col


    def __call__(self, data):
        """
        # Args:
            data (pd.DataFrame):  to clean up.
        # Returns:
            data (pd.DataFrame): transformed dataframe
        """
        data = self.drop_fakes(data)
        return data

    def __repr__(self):
        return (f'{"-" * 30}\n'
                f'{self.__class__.__name__} with parameters:
track_col={self.track_col}'
                f'{"-" * 30}\n'
                f'Number of misses: {self.get_num_fakes()} \n')


@gin.configurable
class ToCylindrical(BaseCoordConverter):
    """Convertes data to polar coordinates. Note that cartesian coordinates
are used in reversed order!
       Formula used: r = sqrt(x^2 + y^2), phi = atan2(x,y)

       # Args:
           drop_old (boolean, False by default): If True, old coordinate
features are deleted from data
           cart_columns (list or tuple of length 3,  ['x', 'y', 'z'] by
default ): columns of x, y and z in cartesian coordiates
           polar_columns = (list or tuple of length 3, ['r','phi', 'z'] by
default):  columns of r and phi (and redundant z) in cylindrical coordinates
       New "z" column (same value for each station) will be r for cylindrical
chamber.
    """

    def __init__(self, drop_old=False, cart_columns=('x', 'y', 'z'),
polar_columns=('r', 'phi', 'z'), postfix='before_cyl'):
        super().__init__(self.convert, drop_old=drop_old,
from_columns=cart_columns, to_columns=polar_columns, postfix=postfix)

    def convert(self, data):
        r = np.sqrt(data[self.from_columns[0]] ** 2 +
data[self.from_columns[1]] ** 2)
        phi = np.arctan2(data[self.from_columns[0]],
data[self.from_columns[1]])
        z = data[self.from_columns[2]]
        return (r, phi, z)

    def __repr__(self):
        return (f'{"-" * 30}\n'
                f'{self.__class__.__name__} with parameters:
drop_old={self.drop_old}, '
                f'from_columns={self.from_columns},
to_columns={self.to_columns}\n'
                f'{"-" * 30}\n'
                f' Ranges: {self.get_ranges_str()} ')
```

```
@gin.configurable
class ToCartesian(BaseCoordConverter):
    """Converts coordinates to cartesian. Formula is: y = r * cos(phi), x = r
* sin(phi).
    Note that always resulting columns are x,y,z. z column after convertion
has same values as before.
      # Args:
        drop_old (boolean, True by default): If True, unscaled features are
dropped from dataframe
        cart_columns (list or tuple of length 3,  ['x', 'y', 'z'] by default
): columns of x and y in cartesian coordiates
        polar_columns = (list or tuple of length 3, ['r','phi','z'] by
default):  columns of r and phi in cylindrical coordiates

    """

    def __init__(self, drop_old=True, cart_columns=('x', 'y', 'z'),
polar_columns=('r', 'phi', 'z')):
        self.from_columns = polar_columns
        self.to_columns = cart_columns
        super().__init__(self.convert, drop_old=drop_old,
from_columns=self.from_columns, to_columns=self.to_columns)

    def convert(self, data):
        y_new = data[self.from_columns[0]] *
np.cos(data[self.from_columns[1]])
        x_new = data[self.from_columns[0]] *
np.sin(data[self.from_columns[1]])
        z_new = data[self.from_columns[2]]
        return (x_new, y_new, z_new)

    def __repr__(self):
        return (f'{"-" * 30}\n'
                f'{self.__class__.__name__} with parameters: '
                f'drop_old={self.drop_old}, phi_col={self.to_columns[1]},
r_col={self.to_columns[0]}\n'
                f'{"-" * 30}\n'
                f'Ranges: {self.get_ranges_str()}')


@gin.configurable
class ToBuckets(BaseTransformer):
    """Data may contains from tracks with varying lengths.
    To prepare-hydra-wombat a train dataset in a proper way, we have to
    split data on so-called buckets. Each bucket includes
    tracks based on their length, as we can't predict the
    6'th point of the track with length 4, but we can predict
    3-d point

        # Args:
            flat (boolean, True by default): If True, converted data is
single dataframe
                                with additional column, else it is dict of
dataframes
            random_state (int, 42 by default): seed for the RandomState
            shuffle (boolean, False by default): whether or not shuffle
output dataset.
            keep_fakes (boolean, True by default):  . If True,
                     points without tracks are preserved.
            event_col (string, 'event' by default): Column with event data.
            track_col (string, 'event' by default): Column with track
numbers.
```

```
    """

    def __init__(self, flat=True, shuffle=False, max_stations=None,
random_state=42,
                 event_col='event', track_col='track', keep_fakes=False,
max_bucket_size=None):
        super().__init__(event_col=event_col, track_col=track_col)
        self.flat = flat
        self.shuffle = shuffle
        self.random_state = random_state
        self.max_num_stations=max_stations
        self.keep_fakes = keep_fakes
        self.max_bucket_size = max_bucket_size

    def __call__(self, df):
        """
        # Args:
            data (pd.DataFrame): data to clean up.
        # Returns:
            data (pd.DataFrame or dict(len:pd.DataFrame): transformed
dataframe,
            if flat is True, returns dataframe with specific column, else
dict with bucket dataframes
        """
        # assert type(data) == pd.core.frame.DataFrame, "unsupported data
format"
        df['index'] = df.index
        rs = np.random.RandomState(self.random_state)
        groupby = df.groupby([self.event_column, self.track_column])
        maxlen = groupby.size().max()
        if self.max_num_stations is None:
            self.max_num_stations = maxlen
        minlen = max(groupby.size().min(), 3)
        subbuckets = {}
        res = {}
        val_cnt = groupby.size().unique()  # get all unique track lens (in
BES3 all are 3)
        val_cnt = range(minlen, max(maxlen+1, val_cnt.max()))
        print(val_cnt)
        for length in val_cnt:
            this_len = groupby.filter(lambda x: x.shape[0] == length)
            if len(this_len) > 0:
                this_len_groups = this_len.groupby([self.event_column,
self.track_column])
                bucket_index =
np.stack(list(this_len_groups['index'].agg(lambda x: list(x.values))),
axis=0)
            else:
                bucket_index = []
            subbuckets[length] = bucket_index
        print(subbuckets)
        # approximate size of the each bucket
        bsize = len(df) // (self.max_num_stations - 2)
        print(bsize)
        if self.max_bucket_size is not None:
            bsize = min(bsize, self.max_bucket_size)
        buckets = {i: [] for i in range(3, self.max_num_stations+1)}
        print(buckets)
        # reverse loop until two points
        for n_points in range(self.max_num_stations, minlen-1, -1):
            print(n_points, maxlen)
            # while bucket is not full
```

```
            k = n_points
            if n_points not in buckets.keys():
                continue
            while len(buckets[n_points]) < bsize:
                if k < n_points or k > maxlen:
                    break
                if self.shuffle:
                    rs.shuffle(subbuckets[k])
                # if we can't extract all data from subbucket
                # without bsize overflow
                if len(buckets[n_points]) + len(subbuckets[k]) > bsize:
                    n_extract = bsize - len(buckets[n_points])
                    # extract n_extract samples
                    buckets[n_points].extend(subbuckets[k][:n_extract,
:n_points])
                    print(buckets[n_points])
                    # remove them from original subbucket
                    subbuckets[k] = subbuckets[k][n_extract:]
                    print(subbuckets)
                else:
                    if len(subbuckets[k]) == 0:
                        k += 1
                        continue
                    buckets[n_points].extend(subbuckets[k][:, :n_points])
                    # remove all data from the original list
                    subbuckets[k] = []
                    # increment index
                k += 1
                print(buckets)
                if all([len(subbuckets[i]) == 0 for i in range(n_points,
maxlen+1) if i in subbuckets.keys()]):
                    break

            if all([len(subbuckets[i]) == 0 for i in subbuckets.keys()]):
                break
        #append unappended items
        for i, k in subbuckets.items():
            if len(k) > 0:
                try:
                    append_len = int(len(k)/(i-2))
                    begin = 0
                    for j in range(i-1, min(list(buckets.keys()))-1, -1):
                        print(k[begin:begin+append_len])
                        buckets[j].extend(k[begin:begin+append_len, :j])
                        begin = begin+append_len
                except:
                    print('alert!')
        buckets = {k: np.concatenate(i) for k, i in buckets.items() if len(i)
> 0}
        self.buckets_ = buckets
        if self.flat is True:
            res = df.copy()
            res['bucket'] = 0
            for i, bucket in buckets.items():
                res.loc[bucket, 'bucket'] = i
        else:
            res = {i: df.loc[bucket] for i, bucket in buckets.items()}
        return res

    def get_bucket_index(self):
        """
        # Returns: dict(len: indexes) - dict with lens and list of indexes in
bucket
```

```
        """
        return self.buckets_

    def get_buckets_sizes(self):
        """

        # Returns:
            {bucket:len} dict with length of data in bucket
        """
        return {i: len(j) for i, j in self.buckets_.items()}

    def __repr__(self):
        return (f'{"-" * 30}\n'
                f'{self.__class__.__name__} with parameters:
flat={self.flat}, '
                f'random_state={self.random_state}, shuffle={self.shuffle},
keep_fakes={self.keep_fakes}\n'
                f'{"-" * 30}\n')
```

Приложение 2. Листинги подготовки данных

```
def preprocess(
        target_processor: DataProcessor.__class__,
        output_dir: str,
        ignore_asserts: bool,
        random_seed=None,
):
    os.makedirs(output_dir, exist_ok=True)
    setup_logger(output_dir, target_processor.__name__)

    LOGGER.info("GOT config: \n======config======\n %s
\n========config=======" % gin.config_str())
    if random_seed is not None:
        LOGGER.info('Setting random seed to %d', random_seed)
        seed_everything(random_seed)
    for data_df, basename in parse():
        LOGGER.info("[Preprocess]: started processing a df with %d rows:" %
len(data_df))
        processor: DataProcessor = target_processor(data_df=data_df,
                                                    output_dir=output_dir)

        generator = processor.generate_chunks_iterable()

        preprocessed_chunks = []
        try:
            for (idx, df_chunk) in tqdm(generator):
                try:
                    data_chunk = processor.construct_chunk(df_chunk)
                except AssertionError as ex:
                    if ignore_asserts:
                        LOGGER.warning("GOT ASSERT %r on idx %d" % (ex, idx))
                        continue
                    else:
                        raise ex
                preprocessed_chunks.append(
                    processor.preprocess_chunk(chunk=data_chunk,
idx=basename)
                )
        except KeyboardInterrupt as ex:
            LOGGER.warning("BREAKING by interrupt. got %d processed chunks" %
len(preprocessed_chunks))
        processed_data = processor.postprocess_chunks(preprocessed_chunks)
        processor.save_on_disk(processed_data)



@gin.configurable(denylist=['df_chunk_data'])
class TracknetDataChunk(DataChunk):
    def __init__(self, df_chunk_data: pd.DataFrame):
        super().__init__(df_chunk_data)

class ProcessedTracknetData(ProcessedData):
    def __init__(self,
                 output_name: str,
                 processed_data: List[ProcessedDataChunk]
                 ):
        super().__init__(processed_data)
        self.processed_data = processed_data
        self.output_name = output_name

class ProcessedTracknetDataChunk(ProcessedDataChunk):
```

```
    def __init__(self,
                 processed_object: Optional,
                 output_name: str):
        super().__init__(processed_object)
        self.processed_object = processed_object
        self.output_name = output_name

@gin.configurable(denylist=['data_df'])
class TrackNetProcessor(DataProcessor):
    def __init__(self,
                 output_dir: str,
                 data_df: pd.DataFrame,
                 name_suffix: str,
                 transforms: List[BaseTransformer] = None):
        super().__init__(
            processor_name='TrackNet_v2_Processor',
            output_dir=output_dir,
            data_df=data_df,
            transforms=transforms)
        self.output_name = os.path.join(self.output_dir,
f'tracknet_{name_suffix}')

    def generate_chunks_iterable(self) -> Iterable[TracknetDataChunk]:
        return self.data_df.groupby('event')

    def construct_chunk(self,
                        chunk_df: pd.DataFrame) -> TracknetDataChunk:
        processed = self.transformer(chunk_df)
        return TracknetDataChunk(processed)

    def preprocess_chunk(self,
                         chunk: TracknetDataChunk,
                         idx: str) -> ProcessedTracknetDataChunk:
        chunk_df = chunk.df_chunk_data

        if chunk_df.empty:
            return ProcessedTracknetDataChunk(None, '')

        chunk_id = int(chunk_df.event.values[0])
        output_name = f'{self.output_dir}/tracknet_{idx.replace(".txt", "")}'
        return ProcessedTracknetDataChunk(chunk_df, output_name)


    def postprocess_chunks(self,
                           chunks: List[ProcessedTracknetDataChunk]) ->
ProcessedTracknetData:
        for chunk in chunks:
            if chunk.processed_object is None:
                continue
            chunk_data_x = []
            chunk_data_y = []
            chunk_data_len = []
            df = chunk.processed_object
            grouped_df = df[df['track'] != -1].groupby('track')
            for i, data in grouped_df:
                chunk_data_x.append(data[['r', 'phi', 'z']].values[:-1])
                chunk_data_y.append(data[['phi', 'z']].values[-1])
                chunk_data_len.append(2)
            chunk_data_x = np.stack(chunk_data_x, axis=0)
            chunk_data_y = np.stack(chunk_data_y, axis=0)
            chunk_data = {
                'x': {
                    'inputs': chunk_data_x,
```

```
                    'input_lengths': chunk_data_len},
                'y': chunk_data_y}
            chunk.processed_object = chunk_data
        return ProcessedTracknetData(chunks[0].output_name,chunks)


    def save_on_disk(self,
                     processed_data: ProcessedTracknetData):
        all_data_inputs = []
        all_data_y = []
        all_data_len = []
        for data_chunk in processed_data.processed_data:
            if data_chunk.processed_object is None:
                continue

all_data_inputs.append(data_chunk.processed_object['x']['inputs'])

all_data_len.append(data_chunk.processed_object['x']['input_lengths'])
            all_data_y.append(data_chunk.processed_object['y'])
        all_data_inputs = np.concatenate(all_data_inputs).astype('float32')
        all_data_y = np.concatenate(all_data_y).astype('float32')
        all_data_len = np.concatenate(all_data_len)
        np.savez(
            processed_data.output_name,
            inputs=all_data_inputs,
            input_lengths=all_data_len, y=all_data_y
        )
        LOGGER.info(f'Saved to: {processed_data.output_name}.npz')



@gin.configurable(denylist=['data_df'])
class TrackNetProcessorWithMask(DataProcessor):
    def __init__(self,
                 output_dir: str,
                 data_df: pd.DataFrame,
                 name_suffix: str,
                 transforms: List[BaseTransformer] = None,
                 columns=('x', 'y', 'z'),
                 max_len=6):
        super().__init__(
            processor_name='TrackNet_v2_Processor',
            output_dir=output_dir,
            data_df=data_df,
            transforms=transforms)
        self.output_name = os.path.join(self.output_dir,
f'masked_tracknet_{name_suffix}')
        self.columns = columns
        self.max_len = max_len
        self.det_indices = [0,1]

    def generate_chunks_iterable(self) -> Iterable[TracknetDataChunk]:
        if len(self.det_indices) > 1:
            self.data_df.loc[self.data_df.det == 1, 'station'] =
self.data_df.loc[self.data_df.det == 1, 'station'].values + 3
            self.data_df = self.data_df.loc[self.data_df['station'] > 1, :]
            self.data_df.loc[:, 'station'] = self.data_df.loc[:,
'station'].values - 2
        return self.data_df.groupby('event')

    def construct_chunk(self,
                        chunk_df: pd.DataFrame) -> TracknetDataChunk:
        processed = self.transformer(chunk_df)
```

```
        return TracknetDataChunk(processed)

    def preprocess_chunk(self,
                         chunk: TracknetDataChunk,
                         idx: str) -> ProcessedTracknetDataChunk:
        chunk_df = chunk.df_chunk_data

        if chunk_df.empty:
            return ProcessedTracknetDataChunk(None, '')
        chunk_id = int(chunk_df.event.values[0])
        output_name =
f'{self.output_dir}/masked_tracknet_{idx.replace(".txt", "")}'
        return ProcessedTracknetDataChunk(chunk_df, self.output_name)


    def postprocess_chunks(self,
                           chunks: List[ProcessedTracknetDataChunk]) ->
ProcessedTracknetData:
        for chunk in chunks:
            if chunk.processed_object is None:
                continue
            chunk_data_x = []
            df = chunk.processed_object
            grouped_df = df[df['track'] != -1].groupby('track')
            for i, data in grouped_df:
                chunk_data_x.append(data[list(self.columns)].values)
            chunk_data_x.extend(chunk_data_x)
            chunk.processed_object = chunk_data_x
        return ProcessedTracknetData(self.output_name,chunks)


    def save_on_disk(self,
                     processed_data: ProcessedTracknetData):
        all_data_inputs = []
        for data_chunk in processed_data.processed_data:
            if data_chunk.processed_object is None:
                continue
            all_data_inputs.extend(data_chunk.processed_object)
        try:
            temp_inputs = np.load(self.output_name+'.npy', allow_pickle=True)
            all_data_inputs = np.concatenate((temp_inputs,
np.array(all_data_inputs, dtype=object)))
        except:
            print('new array is created')
        np.save(self.output_name, all_data_inputs, allow_pickle=True)
        temp_inputs = np.load(self.output_name+'.npy', allow_pickle=True)
        print(len(temp_inputs))
        LOGGER.info(f'Saved to: {self.output_name}.npy as object-pickle')
```

Приложение 3. Листинги моделей

```
@gin.configurable
class TrackNETv2(nn.Module):
    """Builds TrackNETv2 model

    # Arguments
        input_features: number of input features (channels)
        rnn_type: type of the rnn unit, one of [`lstm`, `gru`]
    """
```

```
    def __init__(self,
                 input_features=4,
                 conv_features=32,
                 rnn_type='gru',
                 batch_first=True):
        super().__init__()
        self.input_features = input_features
        rnn_type = rnn_type.upper()
        if rnn_type not in ALLOWED_RNN_TYPES:
            raise ValueError(f'RNN type {rnn_type} is not supported. '
                             f'Choose one of {ALLOWED_RNN_TYPES}')
        _rnn_layer = getattr(nn, rnn_type)
        self.conv = nn.Sequential(
            nn.Conv1d(input_features, conv_features,
                      kernel_size=3,
                      stride=1,
                      padding=1,
                      bias=False),
            nn.ReLU(),
            nn.BatchNorm1d(conv_features)
        )
        self.rnn = _rnn_layer(
            input_size=conv_features,
            hidden_size=conv_features,
            num_layers=2,
            batch_first=batch_first
        )
        # outputs
        self.xy_coords = nn.Sequential(
            nn.Linear(conv_features, 2)
        )
        self.r1_r2 = nn.Sequential(
            nn.Linear(conv_features, 2),
            nn.Softplus()
        )

    def forward(self, inputs, input_lengths, return_gru_states=False):
        # BxTxC -> BxCxT
        inputs = inputs.transpose(1, 2).float()
        input_lengths = input_lengths.int().cpu()
        x = self.conv(inputs)
        # BxCxT -> BxTxC
        x = x.transpose(1, 2)
        # Pack padded batch of sequences for RNN module
        packed = torch.nn.utils.rnn.pack_padded_sequence(
        x, input_lengths, enforce_sorted=False, batch_first=True)
        # forward pass trough rnn
        x, _ = self.rnn(packed)
        # unpack padding
        gru_outs, _ = torch.nn.utils.rnn.pad_packed_sequence(x,
batch_first=True)
        # get result using only the output on the last timestep
        xy_coords = self.xy_coords(gru_outs)
        r1_r2 = self.r1_r2(gru_outs)
        outputs = torch.cat([xy_coords, r1_r2], dim=-1)
        if return_gru_states:
            return outputs, gru_outs
        return outputs


@gin.configurable
class TrackNetClassifier(nn.Module):
    """Builds TrackNETv2_1 classifier model
```

```
    # Arguments
        gru_size: number of features in gru output of base model
        coord_size: number of predicted point coords
        num_classes: number of classes to use (real/fake candidate etc)
    """
    def __init__(self, gru_size=32, coord_size=2, num_gru_states=1,
z_values=None):
        super().__init__()
        self.gru_feat_block =
nn.Sequential(nn.Linear(gru_size*num_gru_states, 30),
                                        #nn.BatchNorm1d(30),
                                        nn.ReLU(),
                                        nn.Linear(30, 15)
                                        )
        self.coord_feat_block = nn.Sequential(
            nn.Linear(coord_size, 20),
            #nn.BatchNorm1d(30),
            nn.ReLU(),
            nn.Linear(20, 15)
        )
        self.classifier = nn.Sequential(nn.Linear(30, 20),
                                         #nn.BatchNorm1d(20),
                                         nn.ReLU(),
                                         nn.Linear(20, 1))

    def forward(self, gru_features, coord_features):
        """
        # Arguments
        gru_x: GRU output of base model
        coord_x: coordinates of predicted point (found as last hit)
        """
        # BxTxC -> BxCxT
        gru_features = gru_features.contiguous().view(gru_features.size()[0],
-1)
        x1 = self.gru_feat_block(gru_features.float())
        x2 = self.coord_feat_block(coord_features.float())
        x = torch.cat((x1, x2), dim=1)
        x = self.classifier(x).float()
        return x


@gin.configurable
class TrackNetV22Classifier(nn.Module):
    """Builds TrackNETv2_2 classifier model

    # Arguments
        gru_size: number of features in gru output of base model
        coord_size: number of predicted point coords
        num_classes: number of classes to use (real/fake candidate etc)
    """
    def __init__(self, gru_size=32, coord_size=2, num_classes=2):
        super().__init__()
        self.classifier = nn.Sequential(
            nn.Linear(9, 10),
            nn.ReLU(),
            nn.Linear(10, 1),
        )

    def forward(self, coord_features):
        """
        # Arguments
        gru_x: GRU output of base model
```

```
        coord_x: coordinates of predicted point (found as last hit)
        """
        # BxTxC -> BxCxT
        x = self.classifier(coord_features).float()
        return x
```

Приложение 5. Листинги инференса

```
@gin.configurable()
def faiss_test(tracknet_ckpt_path_dict,
               classifier_ckpt_path_dict,
               max_num_events=1000,
               input_dir='output/cgem_t_plain_valid_v48_valid',
               file_mask='tracknet_all_3.npz',
               last_station_file_mask='tracknet_all_3_last_station.npz',
               tracknet_input_features=3,
               tracknet_conv_features=32,
               use_classifier=True,
               draw_figures=True,
               plot_treshold=True,
               treshold_min=0.5,
               treshold_max=0.95,
               treshold_step=0.05
               ):
    path_to_tracknet_ckpt = get_checkpoint_path(**tracknet_ckpt_path_dict)
    path_to_classifier_ckpt =
get_checkpoint_path(**classifier_ckpt_path_dict)

    model =
weights_update(model=TrackNETv2(input_features=tracknet_input_features,

conv_features=tracknet_conv_features,
                                            rnn_type='gru',
                                            batch_first=True),
                           checkpoint=torch.load(path_to_tracknet_ckpt))
    model.to(DEVICE)
    class_model = weights_update(model=TrackNetClassifierBig(),

checkpoint=torch.load(path_to_classifier_ckpt))
    class_model.to(DEVICE)

    events = load_data(input_dir, file_mask, None)
    all_last_station_coordinates = load_data(input_dir,
last_station_file_mask, None)

    last_station_hits = all_last_station_coordinates['hits']
    last_station_hits_events = all_last_station_coordinates['events']

    all_tracks_df = pd.DataFrame(columns=['event', 'track', 'hit_0_id',
'hit_1_id', 'hit_2_id', 'px', 'py', 'pz', 'pred'])
    reco_df = all_tracks_df.copy()

    def handle_event_to_df_faiss(batch_input,
                                 batch_len,
                                 batch_target,
                                 batch_real_flag,
                                 batch_last_station,
                                 batch_momentum,
                                 treshold=0.5):
        t0 = time.time()
```

```
        all_last_y = batch_last_station.astype('float32')
        with torch.no_grad():
            temp_dict = {}
            temp_df = pd.DataFrame(columns=['found', 'found_right_point',
'is_real_track', 'px', 'py','pz', 'p'])
            if batch_input.ndim < 3:
                batch_input = np.expand_dims(batch_input, axis=0)
                batch_len = np.expand_dims(batch_len, axis=0)
                batch_target = np.expand_dims(batch_target, axis=0)
                batch_real_flag = np.expand_dims(batch_real_flag, axis=0)
            if all_last_y.ndim < 2:
                all_last_y = np.expand_dims(all_last_y, axis=0)
            test_pred, last_gru_output =
model(inputs=torch.from_numpy(batch_input).to(DEVICE),

input_lengths=torch.from_numpy(batch_len).to(DEVICE),
                                                    return_gru_states=True)
            num_real_tracks = batch_real_flag.sum()
            t1 = time.time()
            nearest_points, is_point_in_ellipse =
find_nearest_hit_old(test_pred[:, -1].detach().cpu().numpy(), all_last_y)
            t2 = time.time()
            if use_classifier:
                pred_classes =
class_model(last_gru_output[is_point_in_ellipse][:,-1],

torch.from_numpy(nearest_points[is_point_in_ellipse].astype('float')).to(DEVI
CE))
                confidence = deepcopy(F.sigmoid(pred_classes))
                pred_classes = (F.sigmoid(pred_classes) >
treshold).squeeze().detach().cpu().numpy()
            is_prediction_true = np.logical_and(batch_target, nearest_points)
            is_prediction_true = is_prediction_true.all(1)
            if use_classifier:
                found_points = is_point_in_ellipse
                found_points[is_point_in_ellipse] = (pred_classes == 1)
            else:
                found_points = is_point_in_ellipse
            found_right_points = found_points & (is_prediction_true == 1) &
(batch_real_flag == 1)
            temp_dict['found'] = found_points
            temp_dict['found_right_point'] = found_right_points
            temp_dict['is_real_track'] = batch_real_flag
            temp_dict['px'] = batch_momentum[:, 0]
            temp_dict['py'] = batch_momentum[:, 1]
            temp_dict['pz'] = batch_momentum[:, 2]
            temp_dict['p'] = np.linalg.norm(batch_momentum, axis=1)
            temp_df = pd.DataFrame(temp_dict)
            t3 = time.time()
            return temp_df, t2-t1, t3-t0


    result_df_faiss = pd.DataFrame(columns=['found', 'found_right_point',
'is_real_track', 'px', 'py', 'pz', 'p'])

    all_time_no_faiss = 0
    all_time_faiss = 0
    search_time_no_faiss = 0
    search_time_faiss = 0
    times = {i: [] for i in range(1, 20)}

    num_tresholds = int((treshold_max - treshold_min) / treshold_step)
    tresholds = np.linspace(treshold_min, treshold_max, num_tresholds)
```

```
    metrics = {'precision': [], 'recall': []}
    treshold = treshold_min
    for i in tqdm(range(num_tresholds)):
        num_events = 0
        for batch_event in np.unique(events['events'])[:max_num_events]:
            num_events += 1
            batch_x = events['x'][events['events'] == batch_event]
            batch_len = events['len'][events['events'] == batch_event]
            batch_y = events['y'][events['events'] == batch_event]
            batch_labels = events['is_real'][events['events'] == batch_event]
            batch_momentums = events['momentums'][events['events'] ==
batch_event]
            last_station_data = last_station_hits[last_station_hits_events ==
batch_event]
            df_faiss, elapsed_time_faiss, total_elapsed_faiss =
handle_event_to_df_faiss(batch_x,

batch_len,

batch_y,

batch_labels,

last_station_data,

batch_momentums,

treshold=treshold)
            #print('\n df_faiss \n', df_faiss[['found', 'found_right_point',
'is_real_track']])
            search_time_faiss += elapsed_time_faiss
            all_time_faiss += total_elapsed_faiss
            times[int(batch_labels.sum())].append(total_elapsed_faiss)
            result_df_faiss = pd.concat([result_df_faiss, df_faiss], axis=0)

        real_tracks =
deepcopy(result_df_faiss.loc[result_df_faiss['is_real_track'] == 1, ])
        recall = result_df_faiss['found_right_point'].sum() /
(result_df_faiss['is_real_track'].sum() + 1e-6)
        precision = result_df_faiss['found_right_point'].sum() /
(float(result_df_faiss['found'].sum()) + 1e-6)
        metrics['precision'].append(precision)
        metrics['recall'].append(recall)

        LOGGER.info('\n ===> FAISS: \n'
                    'Test set results: \n'
                    f'------> Treshold: {treshold}, {num_events} events\n'
                    f'Precision: {precision} \n'
                    f'Recall:    {recall} \n'
                    f'{"-"*10} '
                    f'\nSearch time: {search_time_faiss} sec \n'
                    f'Search time per batch: {(search_time_faiss /
num_events)} sec \n'
                    f'{"-"*10} '
                    f'Total time: {all_time_faiss} sec \n'
                    ""f'Total time per batch: {(all_time_faiss/num_events)}
sec')

        result_df_faiss['pt'] = LA.norm(result_df_faiss[['px','py']].values,
axis=1)
        result_df_faiss['cos_t'] = (result_df_faiss[['pz']].values /
                                    LA.norm(result_df_faiss[['px', 'py',
```

```
'pz']].values, axis=1, keepdims=True))
        result_df_faiss['a_phi'] = np.arctan2(result_df_faiss[['px']].values,
result_df_faiss[['py']].values)


        real_tracks_result_df =
result_df_faiss[result_df_faiss['is_real_track'].astype('int') == 1]
        found_tracks_result_df =
result_df_faiss[result_df_faiss['found'].astype('int') == 1]
        tracks_pred_true =
result_df_faiss[result_df_faiss['found_right_point'].astype('int') == 1]

        if draw_figures:
            draw_for_col(real_tracks_result_df, tracks_pred_true, 'pt',
'$pt$', num_events, 175, style='plot')
            draw_for_col(real_tracks_result_df, tracks_pred_true, 'a_phi',
'$a_\\phi$', num_events, 175, style='plot')
            draw_for_col(real_tracks_result_df, tracks_pred_true,'cos_t',
'$cos_t$', num_events, 175, style='plot')
            draw_for_col(found_tracks_result_df, tracks_pred_true, 'pt',
'$pt$', num_events, 175, metric='precision', style='plot')
            draw_for_col(found_tracks_result_df, tracks_pred_true, 'a_phi',
'$a_\\phi$', num_events, 175, style='plot', metric='precision')
            draw_for_col(found_tracks_result_df, tracks_pred_true, 'cos_t',
'$cos_t$', num_events, 175, style='plot', metric='precision')
            times_mean = {k: np.mean(v) for k, v in times.items()}
            times_std = {k: np.std(v) for k, v in times.items()}
            draw_from_data(title=f"TrackNETv2.1 Mean Processing Time vs
Multiplicity (ms), treshold {treshold}",
                           data_x=list(times_mean.keys()),
                           data_y=[v * 1000 for v in times_mean.values()],
                           data_y_err=[v * 1000 for v in times_std.values()],
                           axis_x="multiplicity",
                           mean_label='mean'
                           )
        treshold += treshold_step

    if use_classifier and plot_treshold:
        draw_treshold_plots(title=f"TrackNETv2.1 Metric values vs Treshold,
{max_num_events} events per value",
                            data_x=tresholds,
                            data_y_dict=metrics,
                            model_name='tracknet_v2_1',
                            total_events=max_num_events)
    if not use_classifier and plot_treshold:
        LOGGER.warning('You can apply treshold only on classifier logits, not
TrackNETv2. Skipping plotting')



def main(argv):
    del argv
    gin.parse_config(open(FLAGS.config))
    LOGGER.setLevel(FLAGS.log)
    LOGGER.info("CONFIG: %s" % gin.config_str())
    faiss_test()

if __name__ == '__main__':
    app.run(main)
```