PointNet for Particle Flow Energy Reconstruction: Documentation

MSE 493

Co-op Technical Report #3

Spring 2024: April 10, 2024

Prepared For:

Stephanie DeRapp

Industry Internship Coordinator, Simon Fraser University

Max Swiatlowski

Research Scientist - Physical Sciences TRIUMF

Prepared By:

Marko Jovanovic

301306143

markoj285@gmail.com

7359 words

Abstract

This technical report presents the development and implementation of JetPointNet, an advanced neural network model based on the PointNet architecture, along with its associated data processing pipeline. It is designed to reconstruct particle flow energies in the ATLAS detector at CERN's Large Hadron Collider. The JetPointNet model innovatively applies deep learning techniques to process and analyze the vast amounts of collision data generated by the ATLAS experiment, with the aim of enhancing the accuracy of energy measurements of particles resulting from proton collisions.

The report details the construction of a robust data preprocessing pipeline that converts complex CERN Root files into structured numpy arrays, preparing the data for the deep learning model. It outlines the adaptation of the PointNet model to accommodate the specific characteristics of particle collision data, integrating custom layers and activation functions to address the challenges inherent to this unique application.

The report establishes the groundwork laid out for JetPointNet and discusses its potential to improve upon current energy reconstruction methods. Intended as onboarding documentation, this report provides insights into the project's objectives, methodology, and initial findings. It also outlines recommendations for future development and research directions to continue this innovative work in particle physics research.

This document serves as a guide for subsequent students who will pick up the project, ensuring a smooth transition and continued progress toward the project's long-term goals.

The link to the code repository is available here:
https://github.com/markojovo/JetPointNet

Table of Contents

Warning: Items in red boxes can be considered warnings that should be paid attention.

Note: Items in yellow boxes can be considered useful information, good to keep in mind.

Recommendation: Items marked in green boxes can be considered recommendations for fixes or future work.

Introduction

The ATLAS experiment, stationed at the Large Hadron Collider (LHC) at CERN, represents a pinnacle in particle physics research. This endeavor involves accelerating packets of protons to velocities nearing **99.9999991%** the speed of light, facilitating their collision. The primary objective of these high-energy proton collisions is to probe the fundamental aspects of matter, investigating the basic constituents of the universe and the forces mediating their interactions.

ATLAS Detector

The ATLAS detector, distinguished as the largest among the detectors operational at the LHC, serves as a critical tool in analyzing the aftermath of these proton collisions. The detector's architecture is an intricate assembly of multiple sensor layers, each designed to perform specific functions in the detection and analysis of particle interactions.

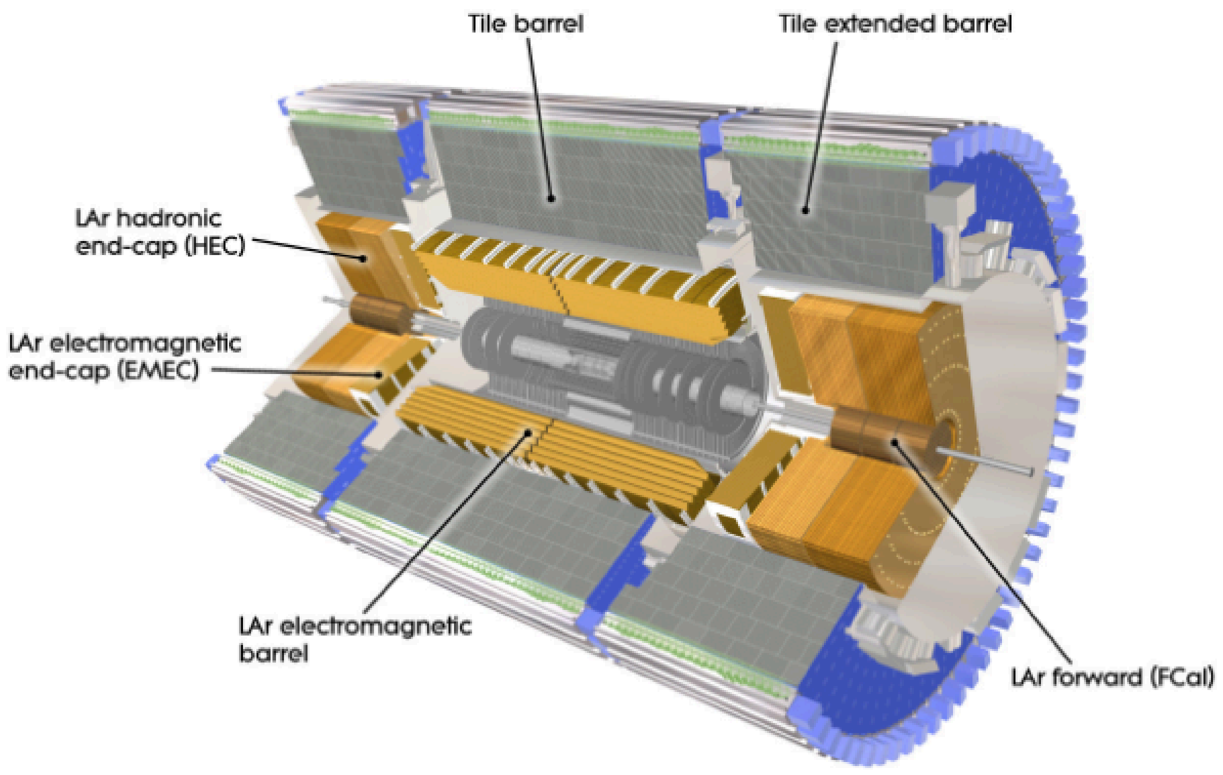

Tile barrel
Tile extended barrel
LAr hadronic end-cap (HEC)
LAr electromagnetic end-cap (EMEC)
LAr electromagnetic barrel
LAr forward (FCal)

Figure 1: The ATLAS detector

The focus of our study centers on two key components within this complex structure: the Inner Tracker (ITk) and the calorimeter cells.

Inner Tracker (ITk)

The Inner Tracker is the closest detector system to the point of collision and is primarily tasked with tracking **charged** particles (it cannot detect neutral particles, that's for the calorimeter layers). It consists of layers of silicon-based sensors that detect the passage of these particles. As charged particles traverse the ITk, they leave behind a trail of hits in these silicon layers. By reconstructing these hits, physicists can determine the trajectories of the particles with high precision. This information is critical for measuring the momentum of the particles, as the curvature of their paths in the detector's magnetic field allows for the calculation of their momentum.

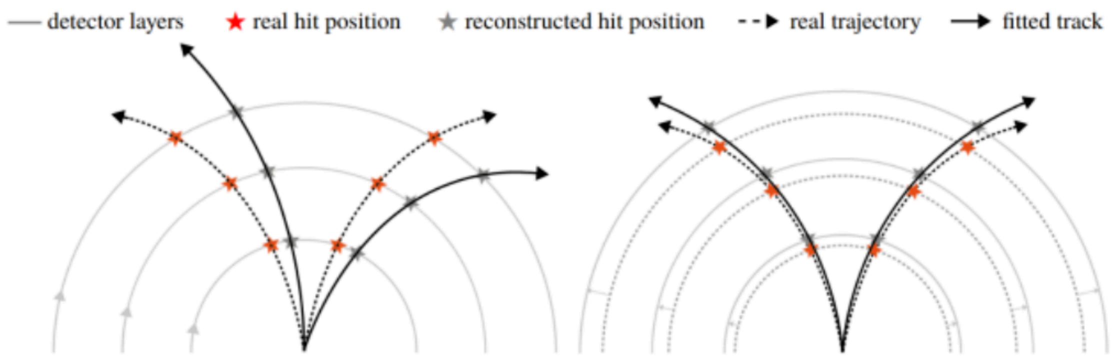


Figure 2: ITk Detected Track Trajectories

Calorimeter System

The calorimeter system surrounds the Inner Tracker and is segmented into two main types: the Electromagnetic Calorimeter (EM Calorimeter) and the Hadronic Calorimeter.

- **Electromagnetic Calorimeter (EM Calorimeter):** This section is specifically designed to measure the energy of particles that interact electromagnetically, such as electrons and photons. It consists of layers of lead and liquid argon. When these particles enter the EM Calorimeter, they initiate a cascade of reactions, creating a shower of secondary particles. The amount of energy deposited in these showers is proportional to the energy of the incoming particle. The EM Calorimeter is optimized to stop and measure these electromagnetic showers, thus providing precise measurements of the energy of electrons and photons.
- **Hadronic Calorimeter:** Located just outside the EM Calorimeter, the Hadronic Calorimeter is designed to measure the energy of hadrons, particles made of quarks and gluons, such as protons, neutrons, and pions. This calorimeter is made of denser materials, such as steel or scintillating tiles, capable of absorbing hadronic particles. Similar to the EM Calorimeter, the Hadronic Calorimeter measures the energy of incoming hadrons by the amount of energy deposited in particle showers initiated by these hadrons.

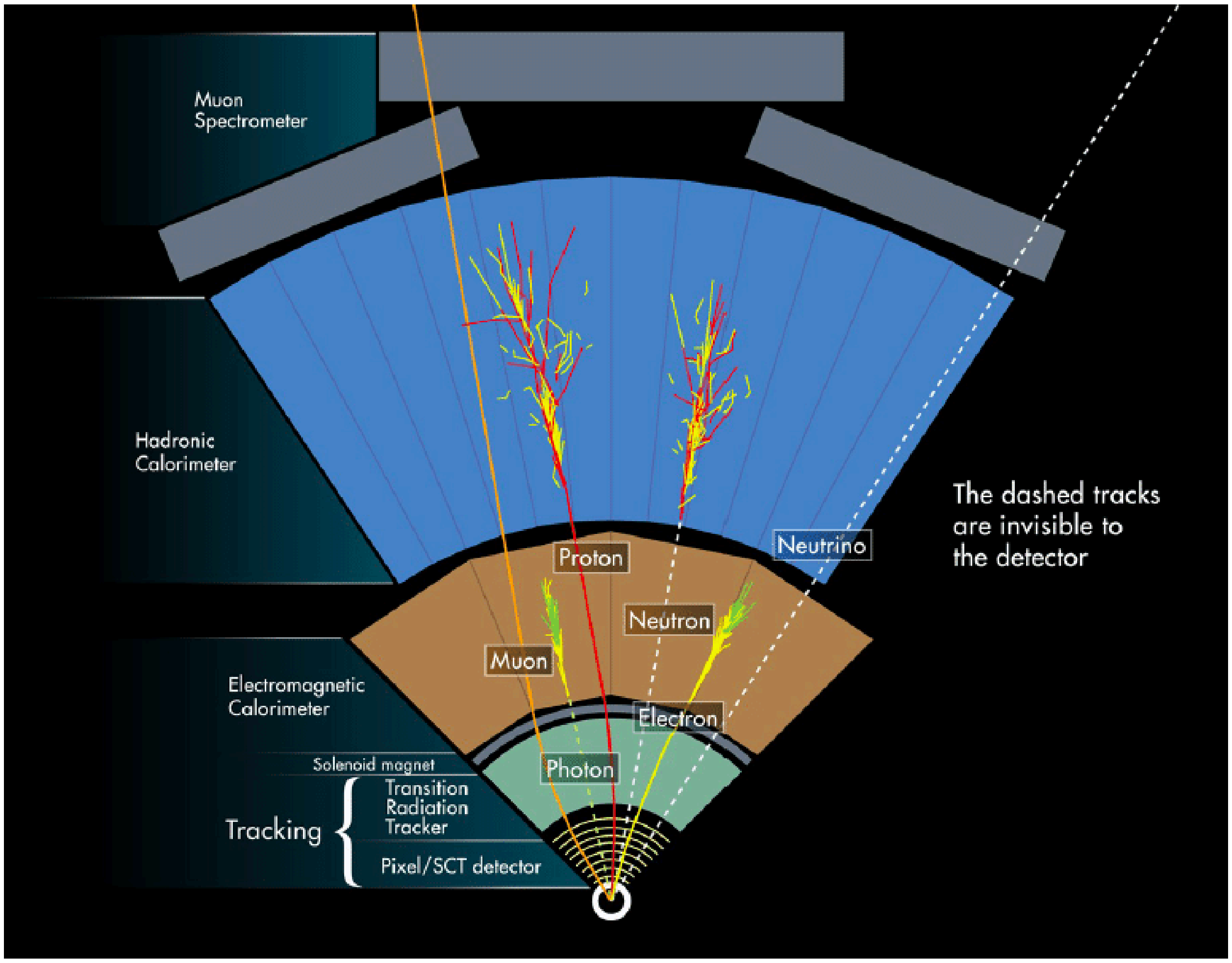


Figure 3: Example particle showers in the ATLAS detector

In the ATLAS detector at CERN, the Inner Tracker (ITk) and the calorimeter systems play crucial roles in analyzing the particles emerging from high-energy proton-proton collisions. These components are designed to measure different aspects of the particles' properties, contributing to the comprehensive understanding of particle interactions.

ATLAS Geometry

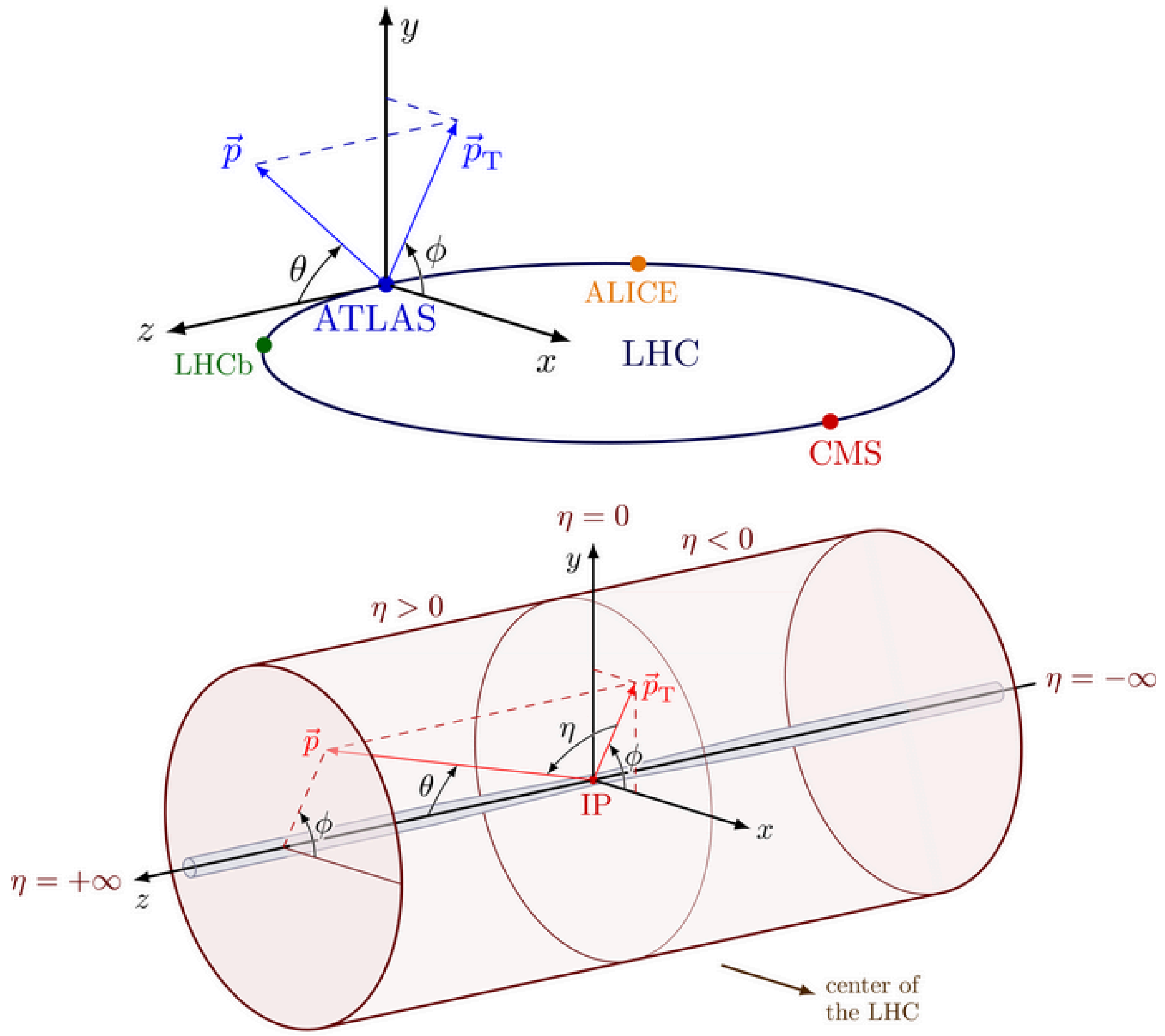


Figure 4: ATLAS detector geometry [1]

The geometric layout of the ATLAS detector is typically described in terms of pseudorapidity (η) and azimuthal angle (ϕ), which are crucial for mapping the three-dimensional trajectories of particles onto a two-dimensional plane. These coordinates are fundamental to the analysis and interpretation of collision events and data formats, aiding in the detailed reconstruction of particle interactions within the detector. The equation for pseudorapidity is as follows:

$$\eta = - ln[tan(\frac{\theta}{2})]$$

Where θ is the polar angle measured from the beam axis. The azimuthal angle ϕ is measured in the plane perpendicular to the beam axis. This can be thought of as a measure of the tangential angle between the beamline axis and the transverse plane.

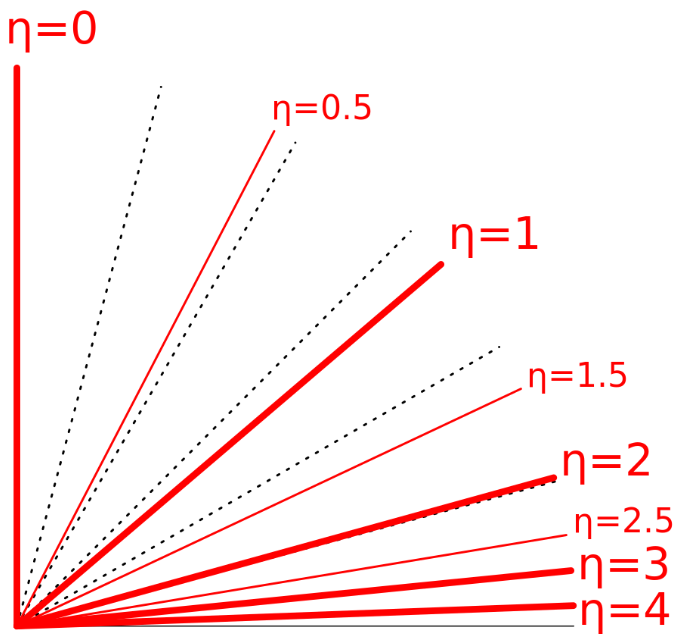


Figure 5: Angular Diagram of Pseudorapidity [2]

Jets

In the exploration of the universe's fundamental elements, the analysis of jets—focused streams of particles formed through the hadronization of quarks and gluons in high-energy phenomena like proton-proton collisions—holds critical importance. Of particular interest are the bottom quark (b-quark) jets, or b-jets, which play a pivotal role in numerous studies, including those investigating the di-Higgs boson production. This process stands at the forefront of examining the Higgs mechanism and the possibilities for new physics that extends beyond the Standard Model's current understanding.

The accurate reconstruction of b-jet energies is essential in these studies, as it influences the detection and quantification of the Higgs bosons. In the context of the di-Higgs scenario, where each of the two Higgs bosons decays into a pair of b-quarks, the precise measurement of the b-jets' energy is instrumental in deducing the Higgs boson's characteristics and validating theoretical models.

The ambition of using neural networks for this is to make significant strides in enhancing the precision of b-jet energy reconstruction.

Problem Statement and Planned Solution

The primary objective in the realm of particle physics investigations at the ATLAS detector is the precise reconstruction of the energies initially carried by particles produced in high-energy collision events. This undertaking demands an exhaustive analysis of the energies deposited across the array of calorimeter cells, augmented by the trajectory data furnished by the Inner Tracker and the spatial arrangement of these cells. Analogous to piecing together an intricate jigsaw puzzle or conducting a forensic investigation at a crime scene, the goal here is to accurately assign the detected energy fragments within specific calorimeter cells to their corresponding particles.

To tackle this challenge, a methodical strategy is essential. The approach involves a thorough scrutiny of the paths carved by charged particles, as recorded by the ITk. The analysis of each tracked particle's trajectory aims to estimate the fraction of energy in a calorimeter cell that belongs to the original tracked particle of focus, and is done by looking at cells in an angular radius around the track. For this purpose, the PointNet deep learning model is selected, renowned for its adeptness in managing point cloud data. This choice is motivated by the similarity between point cloud data and the pattern of energy dispersion in calorimeter cells, where the model is applied to individual tracks within a collision event.

The ambition behind employing this methodology is to outperform the conventional Particle Flow (PFlow) algorithm [4], which has been the benchmark in such analytical endeavors.

The successful deployment and refinement of this deep learning model holds the promise of markedly improving the resolution of measuring particle energies. Enhancements in measurement precision could significantly elevate the quality of experimental results derived from the ATLAS detector and yield deeper insights into the essential mechanisms of particle interactions.

Overall Setup

The following sections of this report delve into the jet energy reconstruction process used in JetPointNet, outlining the codebase and algorithms currently in place. It covers data preprocessing, from CERN Root data to neural network training, particularly focusing on the application of the PointNet model for improved energy measurement resolution.

Data Preprocessing

The jet energy reconstruction workflow begins with the processing of CERN Root files, produced by the MLTree package [5]. These files are compressed dictionaries holding arrays of variable lengths, containing diverse experimental and simulated measurements. The format of these files is not immediately suitable for training deep neural networks due to the varied array sizes and the complex nature of the stored data.

To address this, a data preprocessing pipeline is established to convert CERN Root files into `.npz` files, a Numpy compressed format that is more manageable for neural network training. This pipeline is crucial for reformatting the data into a structure that is both consistent and conducive to machine learning techniques, without losing critical information.

The conversion process involves several steps:

1. Extracting relevant data from the Root files.
2. Organizing this data into a uniform structure.
3. Saving the structured data as `.npz` files for easy access and manipulation in neural network training environments.

Each step is designed to ensure that the dataset remains comprehensive for deep learning purposes, detailing operations in the accompanying codebase for transparency. This preprocessing is essential for utilizing advanced neural network models, like PointNet, efficiently in the subsequent phases of energy reconstruction.

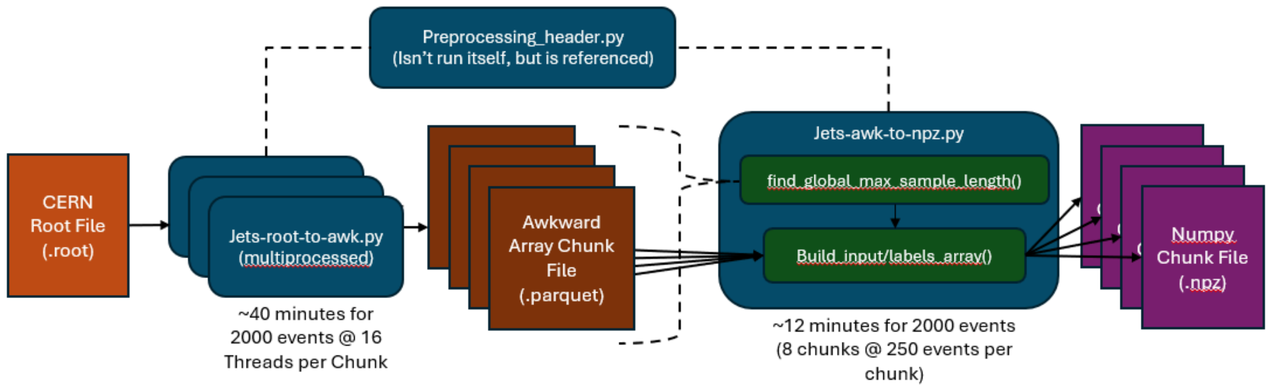


Figure 6: JetPointNet dataset processing pipeline

Note: This is a 2-stage pipeline. The reasoning for this is mainly because of the global_max_sample_length value. We can't know, as we're processing, the size of the largest sample in the dataset (and this is something that may change depending on which processing steps are used), so it lends itself well to using awkward arrays to store this jagged structure. This lets us go through the ROOT file and extract the relevant data then, once the entire ROOT file is processed, go through the awkward arrays to pick out the sample of largest size and then use that for the numpy array. Numpy arrays have the property of having to be of consistent sizes between each sub-array, the size of which we can't know until we've gone through all the samples at least once.

Note: Personally, I've found this setup convenient to make changes with, as you don't need to run through the entire processing pipeline whenever you make changes to the .npz training data (such as labels). There is a lot more information stored in the .parquet files than is used in the .npz training

Recommendation: This processing pipeline can be optimized for efficiency, especially in memory usage, but it was designed with convenience of maintaining and modification in mind. Try to keep things modularized where you can.

Recommendation: This should be able to be easily expanded to handling multiple ROOT files at the same time, one after another. Can be done if needed, maybe for expanded training runs.

CERN Root

The CERN Root files serve as the initial data source for the jet energy reconstruction process, created from either simulation or experimental data. These files are essentially compressed dictionaries forming a tree structure, where each node can contain jagged arrays—arrays of arrays with variable lengths. Data within these files are stored in an "event-wise" manner, meaning each particle collision event is indexed, providing a structured way to access measurements and values for various fields related to that event.

Key characteristics of the CERN Root files include:

- **Event-Wise Storage**: Each event acts as an index, under which data for various measurements are stored. This structure allows for efficient organization and retrieval of data corresponding to specific collision events.
- **Jagged Arrays**: Within each event, there's a variable number of items for different fields, reflecting the diverse outcomes of particle collisions. This variability presents challenges for direct data processing and analysis.
- **Cell Clustering**: Originally, cells detected by the sensors are grouped into clusters, with unclustered cells often omitted. This clustering, along with some initial noise filtering, was performed by previous algorithms. However, for our processing, while we retain the noise-filtered cells, the cluster structure itself is disregarded, and the data is flattened. This step ensures that all relevant cell information is preserved without the constraints of the original cluster organization.
- **Geometry Information**: The geometric details of each cell, crucial for understanding the spatial distribution of energy deposits, are retrieved from a separate CERN Root file tree. This information includes the positioning of cells within the detector, which is vital for parsing the geometric information to the neural network.

Processing CERN Root files involves extracting and reorganizing cell data, disregarding the original cluster structure but retaining essential geometric and noise-filtered information. In this code, we use the “Awkward” library to extract the CERN root data as awkward arrays for the intermediate preprocessing step.

Awkward Arrays

Awkward Arrays are a specialized data structure designed to handle variable-length arrays and nested, hierarchical data, characteristics often encountered in particle physics data like that from the CERN Root files. They excel in managing the jagged arrays present in these files, providing a more flexible and efficient way to work with complex data structures that do not fit neatly into the traditional, fixed-size arrays. By facilitating operations on data with varying dimensions and structures, Awkward Arrays (see Appendix A for some tutorials), make the preprocessing and analysis of particle, track, and calorimeter data more accessible and manageable.

Preprocessing_header.py

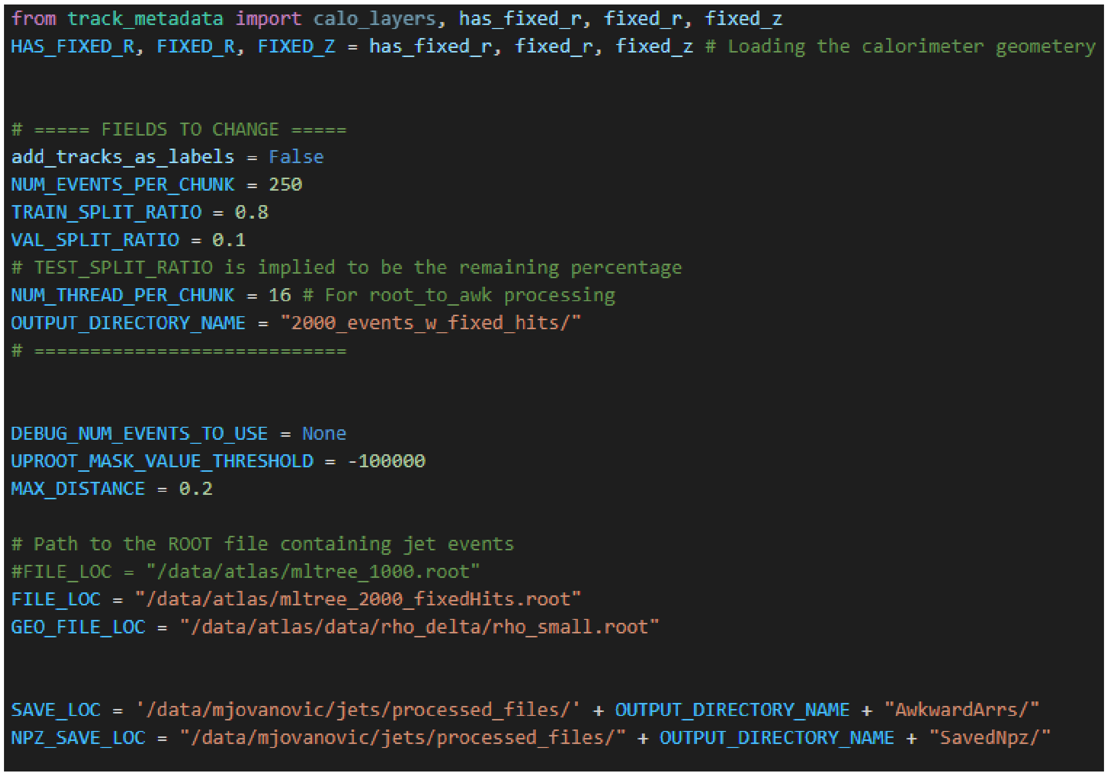
The `preprocessing_header.py` script is crucial for initializing configuration settings for the data preprocessing pipeline in the JetPointNet project. It outlines the geometrical details of the calorimeter by importing `calo_layers`, `has_fixed_r`, `fixed_r`, and `fixed_z` from the `track_metadata` module, which are essential for understanding the structure of the ATLAS detector.

Here are the primary configurations defined:

- `add_tracks_as_labels` determines whether track information is included unmasked in the output, with main tracks labeled as "track_P" and secondary tracks as "0". This is typically left `False` unless track information is explicitly required in the labels.
- `NUM_EVENTS_PER_CHUNK` groups events for batch processing.
- `TRAIN_SPLIT_RATIO` and `VAL_SPLIT_RATIO` split the dataset into training and validation sets.
- `NUM_THREAD_PER_CHUNK` sets the number of threads for parallel processing during the ROOT to Awkward Array conversion.
- `OUTPUT_DIRECTORY_NAME` specifies the directory for storing processed files.

File paths for the source ROOT files and the destination for the Awkward Arrays and `.npz` files are also provided. Additional configurations like `DEBUG_NUM_EVENTS_TO_USE` for debugging purposes and `UPROOT_MASK_VALUE_THRESHOLD` for data masking thresholds are included. These parameters are pivotal for guiding the data transformation processes that follow in the pipeline.

Due to its brevity, simple structure, and use as a configuration file, the total header code is included in the figure below to provide the reader some familiarity with the fields.



```python
from track_metadata import calo_layers, has_fixed_r, fixed_r, fixed_z
HAS_FIXED_R, FIXED_R, FIXED_Z = has_fixed_r, fixed_r, fixed_z # Loading the calorimeter geometery


# ===== FIELDS TO CHANGE =====
add_tracks_as_labels = False
NUM_EVENTS_PER_CHUNK = 250
TRAIN_SPLIT_RATIO = 0.8
VAL_SPLIT_RATIO = 0.1
# TEST_SPLIT_RATIO is implied to be the remaining percentage
NUM_THREAD_PER_CHUNK = 16 # For root_to_awk processing
OUTPUT_DIRECTORY_NAME = "2000_events_w_fixed_hits/"
# ============================


DEBUG_NUM_EVENTS_TO_USE = None
UPROOT_MASK_VALUE_THRESHOLD = -100000
MAX_DISTANCE = 0.2

# Path to the ROOT file containing jet events
#FILE_LOC = "/data/atlas/mltree_1000.root"
FILE_LOC = "/data/atlas/mltree_2000_fixedHits.root"
GEO_FILE_LOC = "/data/atlas/data/rho_delta/rho_small.root"


SAVE_LOC = '/data/mjovanovic/jets/processed_files/' + OUTPUT_DIRECTORY_NAME + "AwkwardArrs/"
NPZ_SAVE_LOC = "/data/mjovanovic/jets/processed_files/" + OUTPUT_DIRECTORY_NAME + "SavedNpz/"
```

Figure 7: processing_header.py

Jets_root_to_awk.py

The `jets_root_to_awk.py` script is designed to convert data from the CERN Root format produced by the MLTree code to Awkward Array format, facilitating subsequent processing stages in the JetPointNet pipeline. It uses the Python library `uproot` to read Root files, `awkward` to handle jagged array data structures, and `numpy` for numerical computations.

The script operates as follows:

1. **Initialization**: Imports necessary modules, defines the processing functions, and sets up a progress tracking mechanism using a Manager from the `multiprocessing` library.
2. **Parallel Processing**: It splits the data into chunks, which are then processed in parallel across multiple threads using `concurrent.futures` and `multiprocessing.Pool`. The number of events per chunk and threads per chunk are configurable parameters defined in the `preprocessing_header`.
3. **Data Splitting**: After processing, the data is divided into training, validation, and test sets based on predefined ratios. This separation is done sample (focused track)-wise, This may cause unlikely situations of samples within the same event that may contain the same overlapping cells, but are separated into the train and validation/test files.
4. **Geometry Extraction**: It retrieves cell geometry information from a specified ROOT file, necessary for aligning cell measurements with their physical location in the detector.
5. **Main Event Processing**: The core of the script where each event is processed to filter and extract relevant cell data and metadata, restructure it into Awkward Arrays, and annotate it with geometric information.
6. **Progress Monitoring**: A separate thread is used to monitor and display the progress of data processing in real time, providing feedback on the script's execution status.
7. **File Handling**: Processed data chunks are saved to disk in the parquet format, with a directory structure organized by train, validation, and test datasets.
8. **Execution Flow**: The script's main block coordinates the execution flow, from loading the data, iterating over chunks, processing, to saving the results, and tracking the overall time elapsed.

The script is intended to be both efficient and modular, leveraging parallel processing to handle large datasets and providing clear progress updates throughout its execution.

Recommendation: Updating the split_and_save_to_disk() function to split event-wise (based on the eventNumber field of each sample in the array) rather than sample-wise (via slicing based on the indices of the array) would be good to do to absolutely ensure that there is no training data leaking into the testing set and vice versa.

Note: The selection criteria for whether a cell (or adjacent track) should be included as part of a focused track's sample is based on the calculate_delta_r() function. Currently it's hard-coded to include any cells or tracks within the MAX_DISTANCE parameter, set to 0.2 in the eta-phi space. This is calculated relative to the focused track's extrapolated hit into the EMB2 or EME2 eta/phi hit coordinates. This being, the extrapolation hit either in the 2nd layer of the electromagnetic calorimeter barrel or that of the endcap, and we use the eta-phi of that hit location for the delta-r calculation. This 0.2 hard-coded maximum distance is used for every track. There is some overlap sometimes, so keep that in mind.

Recommendation: Optimizing the delta-r radius to a variable based on some information about the track would likely greatly increase the quality of sample data provided to the neural network and improve training stability and performance. We had thought of using a calculated sigma value, from a study that had been done from one of Max's colleagues. It was calculated from something like "What eta-phi radius included ~90% of a tracked particle's original energy".

Warning: This processing script flattens the "Event" structure of the ROOT file, and in any file in the .parquets and after, it is stored as one long contiguous array of samples, with each sample corresponding to a singular track of focus and the relevant associated cells and tracks surrounding it, along with some metadata about the track and event. It might be worth removing the flattening of events as it's unnecessary if splitting the train, test, and validation sets by event, but I did it to personally keep things logically coherent between what's saved in the awkward array and the final training output structure.

Note: I use "threads" and "processes" interchangeably in the code and in this document, but for the sake of actually speeding up processing computation, it's integral to use Python's "multi*processing*" functionality. Due to Python's Global Interpreter Lock (GIL), multiple python "threads" cannot run concurrently. "Threading" in python is useful when you have some I/O operations you need to wait on or something, and then you can have the code do other stuff while that waits, but for running truly concurrently, you have to use multiprocessing. This is where it spawns multiple parallel instances of the python process, each with their own allocated memory and process ID.

Recommendation: This code is very memory intensive, for me it ran at about 4-8GB of RAM per process. This isn't bad on the ML-1 server since it has a huge amount of virtual memory, so using 80GB of memory isn't the end of the world, so long as it's during off-hours. This still isn't ideal, and even doing some basic napkin math suggests that there is way more memory being used than is needed. It would at some point be worth optimizing this code for reduced memory usage per process, allowing for more processes to be run in parallel. My gut tells me that a lot of the memory is being used by storing multiple copies of the major awkward array, one for each process. This can likely be shared through some mechanism and stored into one super-array among all the processes.

Track_metadata.py

The `track_metadata.py` script defines key geometrical constants for the ATLAS calorimeter's layers used in the JetPointNet project. It lists the calorimeter layers involved in the tracking of particles and specifies the number of tracking points. Additionally, it provides a dictionary to identify which layers have a fixed radial position (`fixed_r`) and which have a fixed longitudinal position (`fixed_z`). This metadata is essential for locating where particle tracks intersect with calorimeter layers, a critical part of the data preprocessing for jet energy reconstruction.

Jets_awk_to_npz.py

The script `jets_awk_to_npz.py` is an integral component of the data processing pipeline that converts Awkward Arrays stored in Parquet files into Numpy `.npz` format suitable for neural network training. It interfaces with utilities defined in `util_functs.py` and settings from `preprocessing_header.py` to manage the conversion process.

Key operations performed by the script include:

1. **Reading Parquet Files**: The `read_parquet` function reads the Parquet files into Awkward Array format, facilitating further manipulation.
2. **Directory Preparation**: The script ensures the existence of the necessary directories for saving output files by checking and creating folders for training, testing, and validation datasets.
3. **Global Maximum Sample Length Calculation**: It calculates the global maximum sample length across all data. This value is essential to standardize the shape of input arrays for the neural network. Note that sample length in this case refers to the maximum number of points passed into PointNet, so that's the sum of all the number of focused track points, associated cell points, and associated track points.

4. **Data Conversion**: The main loop iterates through each data partition (train, test, val), processing chunk files. For each chunk, it constructs feature arrays and labels for the neural network training, scaling the data appropriately.
5. **Data Saving**: Finally, it saves the processed features and labels as `.npz` files for each data chunk, ensuring they are stored in an organized and accessible manner for subsequent training stages.

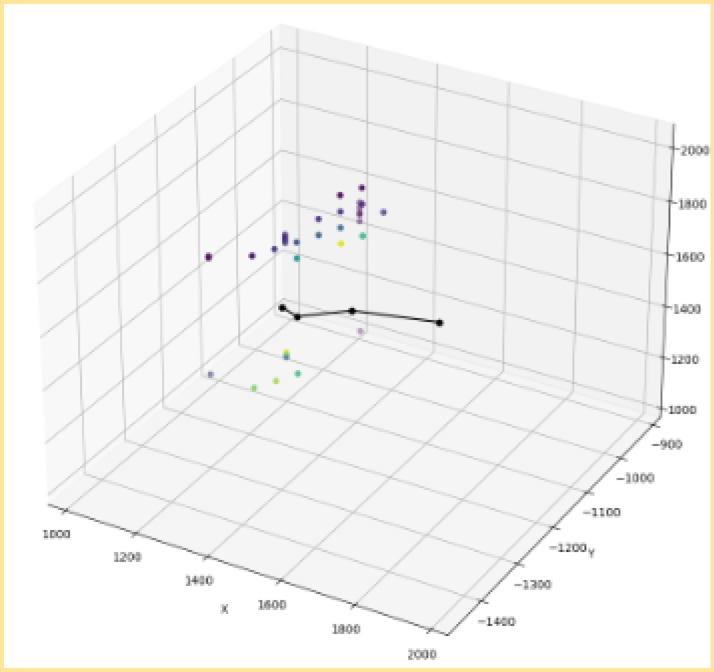
The script is structured to process the data in chunks, allowing for efficient handling of large datasets. This chunked approach is also beneficial for parallel processing if needed. The placeholder `global_max_sample_length` ensures that arrays are uniformly sized, a crucial step before input into machine learning models, as tensorflow does not currently support variable-length inputs, a common trait among currently available deep learning frameworks.

The script's output includes features and labels necessary for training the PointNet model, specifically fractional labels, total labels, and total truth energy, each scaled and prepared for efficient learning.

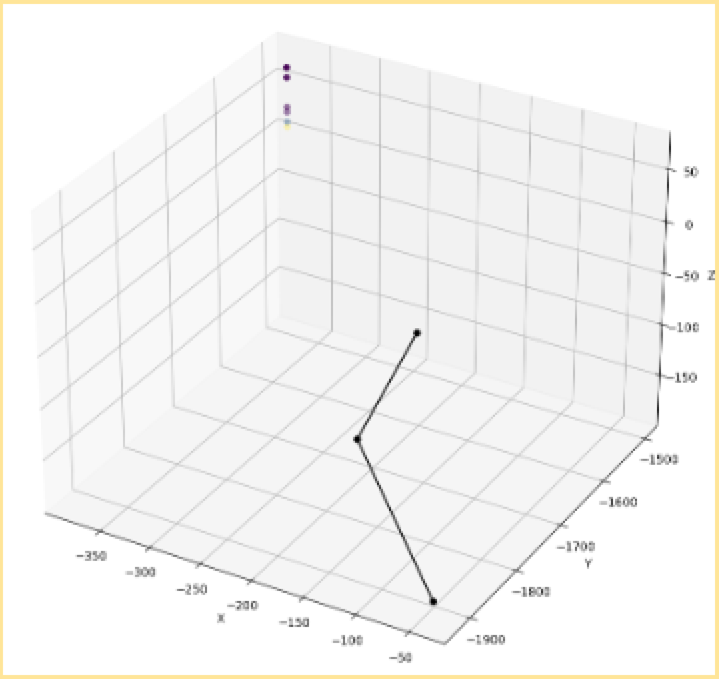
The comment at the end of the script suggests that future enhancements may include better metadata management, particularly regarding the maximum sample length used in processing the Awkward Arrays. This metadata is important for ensuring consistency across the dataset and could be crucial for any post-processing or analysis stages.

Note: Currently, any sample in the awkward .parquet files that has fewer than 100 cells hit are filtered out and excluded from the .npz training data. This was because samples with very few cells tended to be very weird and not good data. This will lead to a size mismatch between the number of samples in the .parquet files and the .npz files.

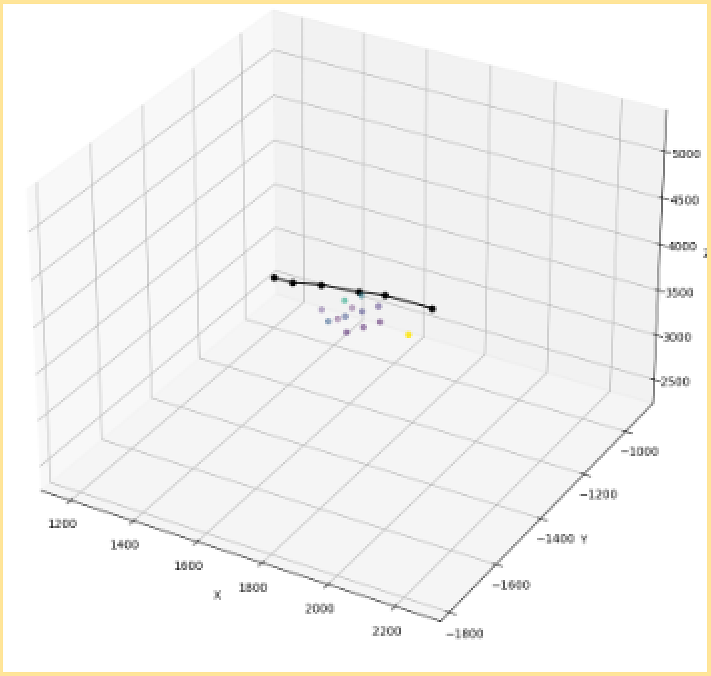
Here are some examples of what they looked like



2000
1800
1600
1400
1200
1000
−900
−1000
−1100
−1200
Y
−1300
−1400
1000
1200
1400
X
1600
1800
2000

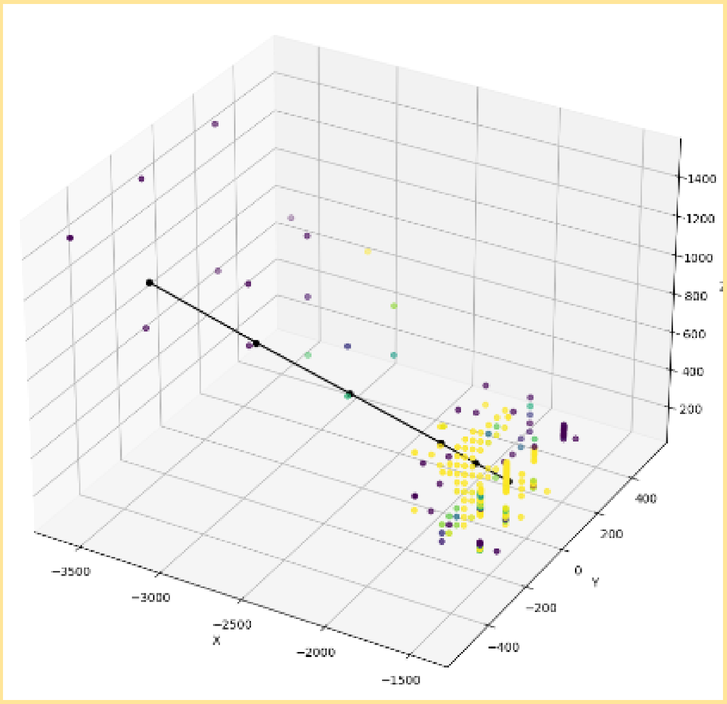


50
0
−50
Z
−100
−150
−1500
−1600
−1700
Y
−1800
−1900
−350
−300
−250
−200
X
−150
−100
−50



PointNet also performed noticeably worse on these types of samples.

Here is a sample that is more in line with what we would expect to see in the detector:



Recommendation: The “ghost tracks” where they would sometimes hit cells on the opposite side of the detector, or would hit few or no cells were often in this “weird data” category. It would be worth investigating at some point but for now the <100 filter seems fine. This problem is most easily visible when plotting the points in these problematic samples in 3D space.

Warning: Currently the global_max_sample_length is set to a hard-coded value (859 currently, as this was the max sample length among the filtered root file used during testing). This may (but should not) cause issues if you move to a different dataset. The worst thing that’ll probably happen is greater RAM / VRAM usage and slowdown. At least if the truncation is working.

Recommendation: Automating or optimizing the global_max_sample_length would be a good idea. It’s used elsewhere in training, so it could be good to have it saved in a .txt during the numpy processing to be referenced later. Also, the size of it determines the size of the input to the neural network, so reducing the size could greatly reduce the neural network size and memory usage. An idea could be to look through the samples and see what's the size that includes 99% of the points, and truncate if any samples are greater than that.

Recommendation: Currently this script is just single-threaded. Could be worth multi-processing if you find yourself inconvenienced by the processing time.

Util_functs.py

The `util_functs.py` script is a collection of utility functions and processing routines for handling data in the JetPointNet pipeline. These functions are primarily concerned with geometric transformations, data manipulation, and preparation of inputs for machine learning models.

Key functions include:

1. **Geometric Conversions**: Functions like `calculate_cartesian_coordinates` and `eta_phi_to_cartesian` transform detector geometry information from the pseudorapidity (η) and azimuthal angle (ϕ) space to Cartesian coordinates. This is crucial for understanding particle trajectories and interactions within the detector.
2. **Intersection Calculations**: The script provides methods to calculate the intersection points of particle tracks with the fixed-radius and fixed-z layers of the ATLAS detector. These intersection points are used to associate energy deposits in the calorimeter cells with the tracks.
3. **Delta R Calculations**: A vectorized function calculates the ΔR, a measure of separation in the η-ϕ space, between pairs of tracks or between tracks and calorimeter cells. This is essential for identifying which cells are associated with which particle tracks.

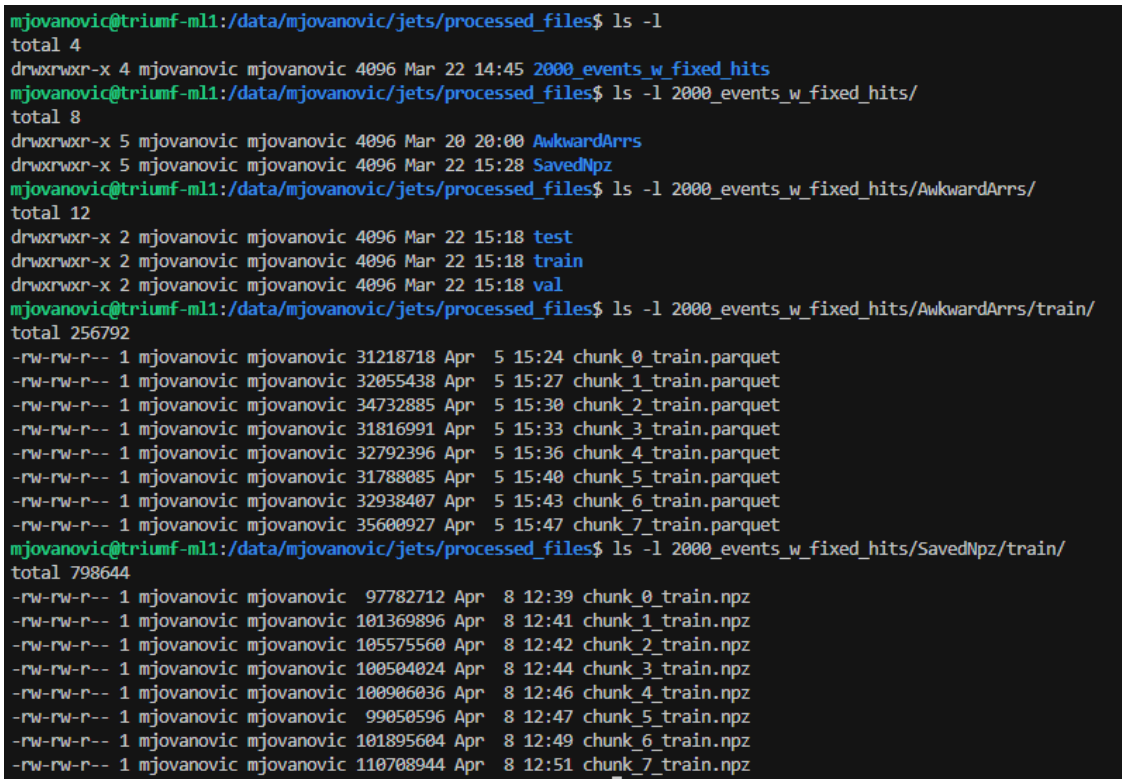
4. **Data Structuring**: Functions such as `calculate_max_sample_length`, `print_events`, `build_input_array`, and `build_labels_array` are responsible for shaping the raw data into structured arrays suitable for input into neural network models. These functions handle tasks such as padding and truncating data to uniform lengths and formatting labels for supervised learning.
5. **Cell and Track Processing**: The script contains several functions to process cell information (`process_and_filter_cells`) and associated track data (`process_associated_tracks`). These functions filter out the relevant information based on geometric criteria and prepare it for subsequent neural network training.
6. **Metadata Addition**: The utility functions also manage the addition of metadata to the track samples, incorporating information like track IDs, η, ϕ, and particle indices. This metadata is crucial for a detailed understanding of each track and its associated energy deposits.

Warning: Util_functs.py houses functions for the entire data processing pipeline and is used at multiple stages.

Recommendation: Could be worth separating functions into different .py modules just to keep things clean and understandable.

Saved File Folders

The processed data structure is organized to support the machine learning workflow for jet energy reconstruction. An example of the saved and processed file structure is shown below, of 2000 events, the data is partitioned into 8 chunks, each containing 250 events. These 250 events are then partitioned further into training, validation and testing data files. This chunking facilitates efficient batch processing for training and evaluation and allows scalability to arbitrarily large datasets.



```
mjovanovic@triumf-ml1:/data/mjovanovic/jets/processed_files$ ls -l
total 4
drwxrwxr-x 4 mjovanovic mjovanovic 4096 Mar 22 14:45 2000_events_w_fixed_hits
mjovanovic@triumf-ml1:/data/mjovanovic/jets/processed_files$ ls -l 2000_events_w_fixed_hits/
total 8
drwxrwxr-x 5 mjovanovic mjovanovic 4096 Mar 20 20:00 AwkwardArrs
drwxrwxr-x 5 mjovanovic mjovanovic 4096 Mar 22 15:28 SavedNpz
mjovanovic@triumf-ml1:/data/mjovanovic/jets/processed_files$ ls -l 2000_events_w_fixed_hits/AwkwardArrs/
total 12
drwxrwxr-x 2 mjovanovic mjovanovic 4096 Mar 22 15:18 test
drwxrwxr-x 2 mjovanovic mjovanovic 4096 Mar 22 15:18 train
drwxrwxr-x 2 mjovanovic mjovanovic 4096 Mar 22 15:18 val
mjovanovic@triumf-ml1:/data/mjovanovic/jets/processed_files$ ls -l 2000_events_w_fixed_hits/AwkwardArrs/train/
total 256792
-rw-rw-r-- 1 mjovanovic mjovanovic 31218718 Apr  5 15:24 chunk_0_train.parquet
-rw-rw-r-- 1 mjovanovic mjovanovic 32055438 Apr  5 15:27 chunk_1_train.parquet
-rw-rw-r-- 1 mjovanovic mjovanovic 34732885 Apr  5 15:30 chunk_2_train.parquet
-rw-rw-r-- 1 mjovanovic mjovanovic 31816991 Apr  5 15:33 chunk_3_train.parquet
-rw-rw-r-- 1 mjovanovic mjovanovic 32792396 Apr  5 15:36 chunk_4_train.parquet
-rw-rw-r-- 1 mjovanovic mjovanovic 31788085 Apr  5 15:40 chunk_5_train.parquet
-rw-rw-r-- 1 mjovanovic mjovanovic 32938407 Apr  5 15:43 chunk_6_train.parquet
-rw-rw-r-- 1 mjovanovic mjovanovic 35600927 Apr  5 15:47 chunk_7_train.parquet
mjovanovic@triumf-ml1:/data/mjovanovic/jets/processed_files$ ls -l 2000_events_w_fixed_hits/SavedNpz/train/
total 798644
-rw-rw-r-- 1 mjovanovic mjovanovic  97782712 Apr  8 12:39 chunk_0_train.npz
-rw-rw-r-- 1 mjovanovic mjovanovic 101369896 Apr  8 12:41 chunk_1_train.npz
-rw-rw-r-- 1 mjovanovic mjovanovic 105575560 Apr  8 12:42 chunk_2_train.npz
-rw-rw-r-- 1 mjovanovic mjovanovic 100504024 Apr  8 12:44 chunk_3_train.npz
-rw-rw-r-- 1 mjovanovic mjovanovic 100906036 Apr  8 12:46 chunk_4_train.npz
-rw-rw-r-- 1 mjovanovic mjovanovic  99050596 Apr  8 12:47 chunk_5_train.npz
-rw-rw-r-- 1 mjovanovic mjovanovic 101895604 Apr  8 12:49 chunk_6_train.npz
-rw-rw-r-- 1 mjovanovic mjovanovic 110708944 Apr  8 12:51 chunk_7_train.npz
```

Figure 8: Example file output structure from a dataset processing run

The structure is as follows:

- A root directory named `2000_events_w_fixed_hits` houses the entire processed dataset.
- Within this root, there are three main subdirectories: `AwkwardArrs`, `SavedNpz`, and `test/train/val` splits corresponding to the different stages of machine learning model development.

- `AwkwardArrs` contains the intermediate `.parquet` files which represent the preprocessed data in an Awkward Array format. This format retains the complex nested structures necessary for detailed particle collision data.
- `SavedNpz` contains the final `.npz` files ready for input into the neural network. Each `.npz` file is named systematically (e.g., `chunk_0_train.npz`) to reflect its contents and role in the pipeline.
- Inside each of the test, train, and val folders under `AwkwardArrs` and `SavedNpz`, files are further chunked, corresponding to the segregated 250-event datasets, ensuring manageable sizes for computational processes.

This systematic organization enables straightforward access and manipulation of data subsets during the machine learning model's training, validation, and testing phases

Awkward and Numpy Array Track/Cell Data Structure

The data structure for tracks and cells within the JetPointNet project is designed to manage the complex spatial and feature-related information captured by the ATLAS detector. At the core of this data handling are the Awkward and NumPy arrays.

Awkward Sample Array

The structure of the Awkward sample arrays can be seen in figures 9 and 10. They contain nested jagged arrays for the various fields relevant to each sample. Note again that this is one continuous awkward array containing all the samples for all the events in the chunk.

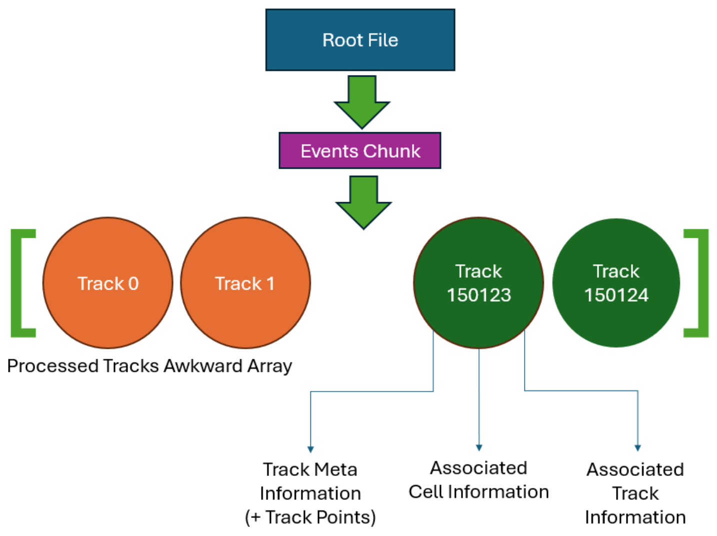
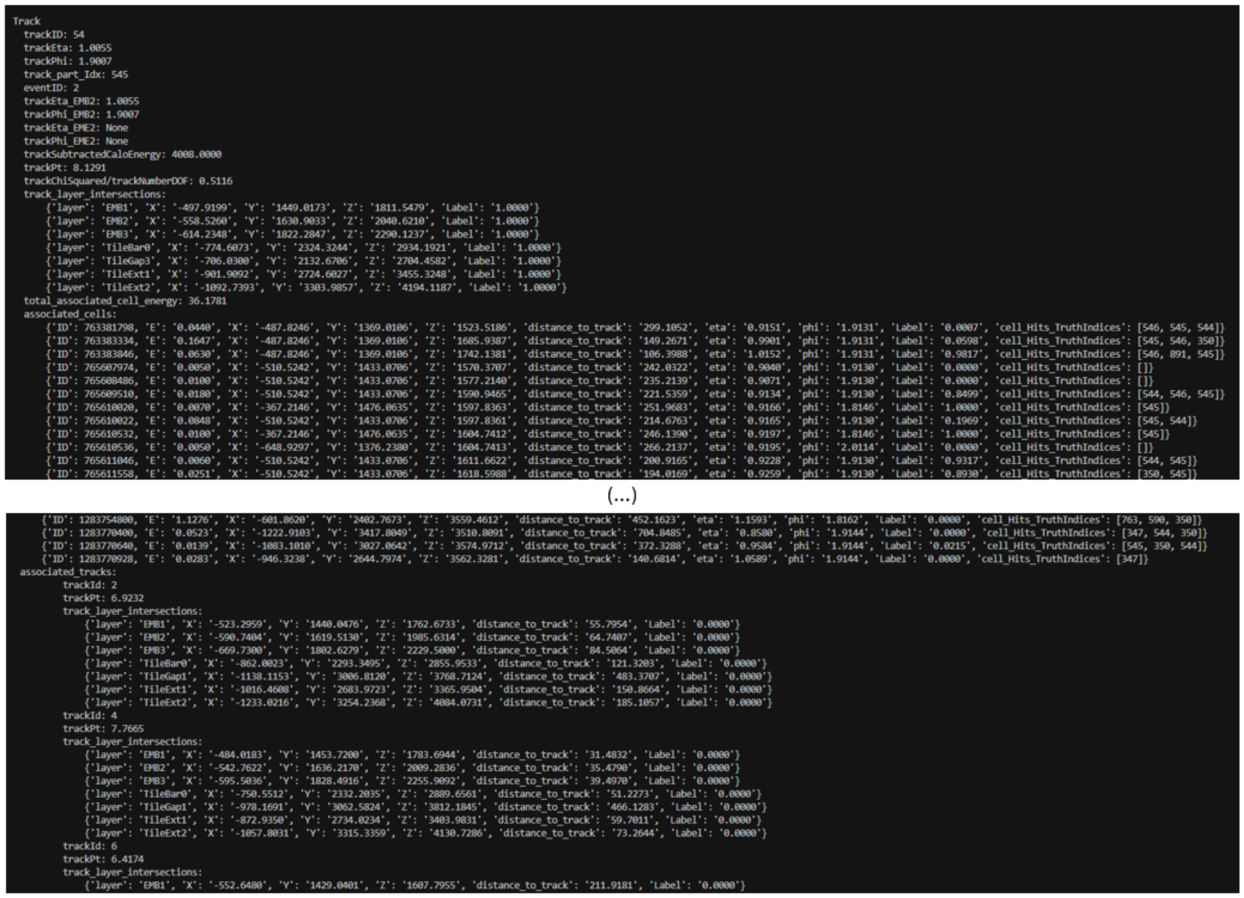

Root File
Events Chunk
Track 0
Track 1
Track 150123
Track 150124
Processed Tracks Awkward Array
Track Meta Information (+ Track Points)
Associated Cell Information
Associated Track Information

Figure 9: Awkward Array Structure



```
Track
  trackID: 54
  trackEta: 1.0055
  trackPhi: 1.9007
  track_part_Idx: 545
  eventID: 2
  trackEta_EMB2: 1.0055
  trackPhi_EMB2: 1.9007
  trackEta_EME2: None
  trackPhi_EME2: None
  trackSubtractedCaloEnergy: 4008.0000
  trackPt: 8.1291
  trackChiSquared/trackNumberDOF: 0.5116
  track_layer_intersections:
      {'layer': 'EMB1', 'X': '-497.9199', 'Y': '1449.0173', 'Z': '1811.5479', 'Label': '1.0000'}
      {'layer': 'EMB2', 'X': '-558.5260', 'Y': '1630.9033', 'Z': '2040.6210', 'Label': '1.0000'}
      {'layer': 'EMB3', 'X': '-614.2348', 'Y': '1822.2847', 'Z': '2290.1237', 'Label': '1.0000'}
      {'layer': 'TileBar0', 'X': '-774.6073', 'Y': '2324.3244', 'Z': '2934.1921', 'Label': '1.0000'}
      {'layer': 'TileGap3', 'X': '-706.0300', 'Y': '2132.6706', 'Z': '2704.4582', 'Label': '1.0000'}
      {'layer': 'TileExt1', 'X': '-901.9092', 'Y': '2724.6027', 'Z': '3455.3248', 'Label': '1.0000'}
      {'layer': 'TileExt2', 'X': '-1092.7393', 'Y': '3303.9857', 'Z': '4194.1187', 'Label': '1.0000'}
  total_associated_cell_energy: 36.1781
  associated_cells:
      {'ID': 763381798, 'E': '0.0440', 'X': '-487.8246', 'Y': '1369.0106', 'Z': '1523.5186', 'distance_to_track': '299.1052', 'eta': '0.9151', 'phi': '1.9131', 'Label': '0.0007', 'cell_Hits_TruthIndices': [546, 545, 544]}
      {'ID': 763383334, 'E': '0.1647', 'X': '-487.8246', 'Y': '1369.0106', 'Z': '1685.9387', 'distance_to_track': '149.2671', 'eta': '0.9901', 'phi': '1.9131', 'Label': '0.0598', 'cell_Hits_TruthIndices': [545, 546, 350]}
      {'ID': 763383846, 'E': '0.0630', 'X': '-487.8246', 'Y': '1369.0106', 'Z': '1742.1381', 'distance_to_track': '106.3988', 'eta': '1.0152', 'phi': '1.9131', 'Label': '0.9817', 'cell_Hits_TruthIndices': [546, 891, 545]}
      {'ID': 765607974, 'E': '0.0050', 'X': '-510.5242', 'Y': '1433.0706', 'Z': '1570.3707', 'distance_to_track': '242.0322', 'eta': '0.9040', 'phi': '1.9130', 'Label': '0.0000', 'cell_Hits_TruthIndices': []}
      {'ID': 765608486, 'E': '0.0100', 'X': '-510.5242', 'Y': '1433.0706', 'Z': '1577.2140', 'distance_to_track': '235.2139', 'eta': '0.9071', 'phi': '1.9130', 'Label': '0.0000', 'cell_Hits_TruthIndices': []}
      {'ID': 765609510, 'E': '0.0180', 'X': '-510.5242', 'Y': '1433.0706', 'Z': '1590.9465', 'distance_to_track': '221.5359', 'eta': '0.9134', 'phi': '1.9130', 'Label': '0.8499', 'cell_Hits_TruthIndices': [544, 546, 545]}
      {'ID': 765610020, 'E': '0.0070', 'X': '-367.2146', 'Y': '1476.0635', 'Z': '1597.8363', 'distance_to_track': '251.9683', 'eta': '0.9166', 'phi': '1.8146', 'Label': '1.0000', 'cell_Hits_TruthIndices': [545]}
      {'ID': 765610022, 'E': '0.0848', 'X': '-510.5242', 'Y': '1433.0706', 'Z': '1597.8361', 'distance_to_track': '214.6763', 'eta': '0.9165', 'phi': '1.9130', 'Label': '0.1969', 'cell_Hits_TruthIndices': [545, 544]}
      {'ID': 765610532, 'E': '0.0100', 'X': '-367.2146', 'Y': '1476.0635', 'Z': '1604.7412', 'distance_to_track': '246.1390', 'eta': '0.9197', 'phi': '1.8146', 'Label': '1.0000', 'cell_Hits_TruthIndices': [545]}
      {'ID': 765610536, 'E': '0.0050', 'X': '-648.9297', 'Y': '1376.2380', 'Z': '1604.7413', 'distance_to_track': '266.2137', 'eta': '0.9195', 'phi': '2.0114', 'Label': '0.0000', 'cell_Hits_TruthIndices': []}
      {'ID': 765611046, 'E': '0.0060', 'X': '-510.5242', 'Y': '1433.0706', 'Z': '1611.6622', 'distance_to_track': '200.9165', 'eta': '0.9228', 'phi': '1.9130', 'Label': '0.9317', 'cell_Hits_TruthIndices': [544, 545]}
      {'ID': 765611558, 'E': '0.0251', 'X': '-510.5242', 'Y': '1433.0706', 'Z': '1618.5988', 'distance_to_track': '194.0169', 'eta': '0.9259', 'phi': '1.9130', 'Label': '0.8930', 'cell_Hits_TruthIndices': [350, 545]}
(...)
      {'ID': 1283754800, 'E': '1.1276', 'X': '-601.8620', 'Y': '2402.7673', 'Z': '3559.4612', 'distance_to_track': '452.1623', 'eta': '1.1593', 'phi': '1.8162', 'Label': '0.0000', 'cell_Hits_TruthIndices': [763, 590, 350]}
      {'ID': 1283770400, 'E': '0.0523', 'X': '-1222.9103', 'Y': '3417.8049', 'Z': '3510.8091', 'distance_to_track': '704.8485', 'eta': '0.8580', 'phi': '1.9144', 'Label': '0.0000', 'cell_Hits_TruthIndices': [347, 544, 350]}
      {'ID': 1283770640, 'E': '0.0139', 'X': '-1083.1010', 'Y': '3027.0642', 'Z': '3574.9712', 'distance_to_track': '372.3288', 'eta': '0.9584', 'phi': '1.9144', 'Label': '0.0215', 'cell_Hits_TruthIndices': [545, 350, 544]}
      {'ID': 1283770928, 'E': '0.0283', 'X': '-946.3238', 'Y': '2644.7974', 'Z': '3562.3281', 'distance_to_track': '140.6814', 'eta': '1.0589', 'phi': '1.9144', 'Label': '0.0000', 'cell_Hits_TruthIndices': [347]}
  associated_tracks:
          trackId: 2
          trackPt: 6.9232
          track_layer_intersections:
              {'layer': 'EMB1', 'X': '-523.2959', 'Y': '1440.0476', 'Z': '1762.6733', 'distance_to_track': '55.7954', 'Label': '0.0000'}
              {'layer': 'EMB2', 'X': '-590.7404', 'Y': '1619.5130', 'Z': '1985.6314', 'distance_to_track': '64.7407', 'Label': '0.0000'}
              {'layer': 'EMB3', 'X': '-669.7300', 'Y': '1802.6279', 'Z': '2229.5000', 'distance_to_track': '84.5064', 'Label': '0.0000'}
              {'layer': 'TileBar0', 'X': '-862.0023', 'Y': '2293.3495', 'Z': '2855.9533', 'distance_to_track': '121.3203', 'Label': '0.0000'}
              {'layer': 'TileGap1', 'X': '-1138.1153', 'Y': '3006.8120', 'Z': '3768.7124', 'distance_to_track': '483.3707', 'Label': '0.0000'}
              {'layer': 'TileExt1', 'X': '-1016.4608', 'Y': '2683.9723', 'Z': '3365.9504', 'distance_to_track': '150.8664', 'Label': '0.0000'}
              {'layer': 'TileExt2', 'X': '-1233.0216', 'Y': '3254.2368', 'Z': '4084.0731', 'distance_to_track': '185.1057', 'Label': '0.0000'}
          trackId: 4
          trackPt: 7.7665
          track_layer_intersections:
              {'layer': 'EMB1', 'X': '-484.0183', 'Y': '1453.7200', 'Z': '1783.6944', 'distance_to_track': '31.4832', 'Label': '0.0000'}
              {'layer': 'EMB2', 'X': '-542.7622', 'Y': '1636.2170', 'Z': '2009.2836', 'distance_to_track': '35.4790', 'Label': '0.0000'}
              {'layer': 'EMB3', 'X': '-595.5036', 'Y': '1828.4916', 'Z': '2255.9092', 'distance_to_track': '39.4970', 'Label': '0.0000'}
              {'layer': 'TileBar0', 'X': '-750.5512', 'Y': '2332.2035', 'Z': '2889.6561', 'distance_to_track': '51.2273', 'Label': '0.0000'}
              {'layer': 'TileGap1', 'X': '-978.1691', 'Y': '3062.5824', 'Z': '3812.1845', 'distance_to_track': '466.1283', 'Label': '0.0000'}
              {'layer': 'TileExt1', 'X': '-872.9350', 'Y': '2734.0234', 'Z': '3403.9831', 'distance_to_track': '59.7011', 'Label': '0.0000'}
              {'layer': 'TileExt2', 'X': '-1057.8031', 'Y': '3315.3359', 'Z': '4130.7286', 'distance_to_track': '73.2644', 'Label': '0.0000'}
          trackId: 6
          trackPt: 6.4174
          track_layer_intersections:
              {'layer': 'EMB1', 'X': '-552.6480', 'Y': '1429.0401', 'Z': '1607.7955', 'distance_to_track': '211.9181', 'Label': '0.0000'}
```

Figure 10: Example Awkward Array Structure, printed via print_events() in util_functs.py

Numpy Data Array

The Numpy arrays are the selected data required for training and testing the neural network. Each sample is represented by an array of MAX_SAMPLE_LENGTH points, where if there's fewer points than the maximum, the extra indices in the numpy array are masked out in both input and label.

Input Data Shape

The shape of the input for a sample is given by (MAX_SAMPLE_LENGTH, num_points, num_feats_per_point). Each point in the input data is represented by an array with the following features:

- `x (mm)`: The x-coordinate in millimeters, detailing the particle's position in the detector's lateral plane.
- `y (mm)`: The y-coordinate in millimeters, indicating the particle's vertical position within the detector.
- `z (mm)`: The z-coordinate in millimeters, capturing the depth of the particle's location along the beamline.
- `minimum_of_distance_to_focused_track (mm)`: The shortest distance in millimeters from the point to the track that is the primary focus of the energy deposit analysis.
- `energy (MeV)`: The energy measured at the point, expressed in megaelectronvolts (MeV). For tracks this is set to the tracked particle's measured momentum value.
- `type`: A categorical value indicating the nature of the point; it's set to `-1` for masked points, `0` for calorimeter cells, `1` for the focused track (the track under analysis), and `2` for other tracks.

The type designation is crucial for guiding the deep learning model's interpretation of the data, differentiating between various kinds of points in the point cloud.

Labeling Scheme

When converting data from Awkward to NumPy arrays (`awk_to_npz.py`), if the option `add_tracks_as_labels` is set to `False`, labels for the tracks are assigned a value of `-1`. This instructs the model to mask these points during the loss computation phase, effectively excluding them from predictions and training gradients. This mechanism ensures that the learning process remains focused on the relevant data points.

For each cell, the `tot_labels` array holds unbounded values representing the absolute amount of true energy that the focused particle has deposited in that cell. Depending on the scale applied during data preprocessing, this energy is represented in either MeV (if scaled by 1000) or GeV (if scaled by 1).

The label for each point is structured into a 1D array of size `NUM_POINTS`, aligning with the PointNet framework's characteristic that each input point's index matches its output label index. This feature enables a direct correlation between the input data and the model's output, facilitating an efficient learning process. Any points that are masked out in the system are assigned a label of `-1`, signaling the model to ignore these during training.

Training

Model Architecture Overview

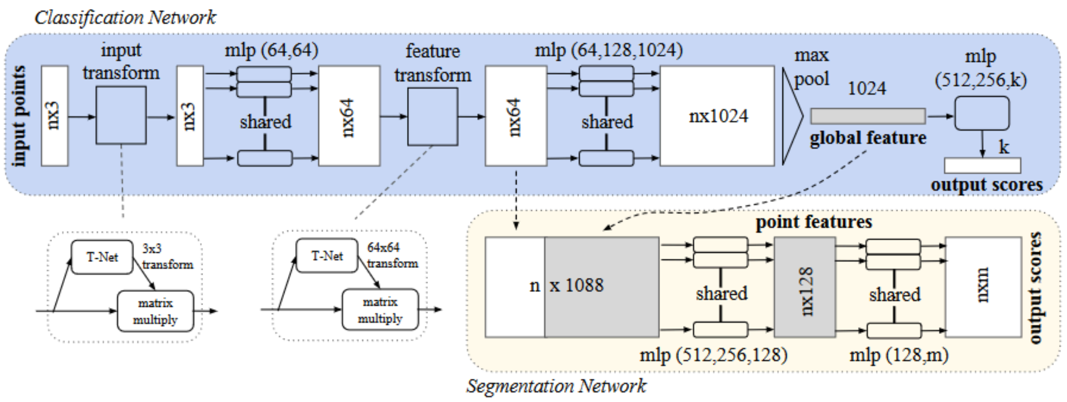


Figure 11: PointNet Architecture [3]

The model being used adapts the base PointNet architecture with a novel addition that multiplies the network's output by the input energies. It uses a TSSR activation function to scale these outputs before the final multiplication, which aims to isolate the energy contribution from a specific particle within each detector cell. To fully grasp the intricacies of this model and its various components, it's highly recommended to review the original PointNet paper.

models/JetPointNet.py

Key components and functionalities of this adapted model include:

- **Custom Layers and Functions**: The model uses a `CustomMaskingLayer` to handle input correctly by ignoring specific types of data points, and an `OrthogonalRegularizer` that encourages orthogonal transformations within the T-Net layers, a mini-network within PointNet that normalizes the input data.
- **T-Nets**: These mini-networks within the architecture apply affine transformations to align the input points and features in a canonical space, which is a critical step for the model's invariance to geometric transformations of the input data.
- **Convolutional MLPs**: The model employs 1D convolutional layers with kernel size 1 (equivalent to shared MLPs) to process the input data. These are used in both the T-Nets and the main network body to extract features.
- **Global Feature Aggregation**: A global max pooling layer is used to condense the per-point features into a global descriptor that encapsulates the entire input point cloud's properties.
- **Activation Function**: A custom activation function, `TSSR_Activation`, is employed, which behaves linearly for small input values and as a square root function for larger input values. The reason for using TSSR rather than a more traditional activation function that is bounded, is that due to measurement inefficiencies there may actually be more of the focused track's energy in the cell than the total amount of energy measured in the cell. The hope is to "soft-bound" the output energies to something scaled with the input in order to stabilize training and prediction outputs and avoid the "mean-seeking" problem.
- **Bypass with Energy Multiplication**: Differing from the original PointNet architecture, the original input energies are bypassed through the network and ultimately multiplied by the TSSR-scaled segmentation outputs. This operation is designed to output the "focused particle energy deposit," indicative of a particular particle's contribution to the total detected energy. This is done to stabilize the outputs so that there's a direct one-step line

relating the cell's energy and the relative deposit from the focused particle. Think in the style of "ResNet" skips.

- **Loss Functions**: Specialized loss functions are used to handle the model's output, including `masked_bce_loss`, `masked_mse_loss`, and `masked_mae_loss`. These loss functions ensure that only the relevant parts of the output are considered during training, effectively ignoring the masked or padded parts of the input.

This specialized version of PointNet is configured to handle the high-dimensional, structured data typical of particle physics experiments, aiming to provide precise predictions of energy deposition patterns. The inclusion of custom layers and functions, such as the TSSR activation and specialized loss functions, reflect an optimization for our particular use case of jet energy reconstruction.

Recommendation: Hyperparameter optimization is an obvious ultimate step. Once you have messed around and gotten an idea of what kind of setup you think will work well, then it is worth setting up a hyperparameter optimization run. From a study I had done on another project, I would recommend using the "Hyperband" algorithm, as it found the best optimum in the fewest steps for the problem being solved. If you have a good base model, this is a good way to get a "free lunch" in terms of performance improvement.

Recommendation: Currently there's little cross-point information transfer mechanisms other than the global feature encoding. What I would recommend trying is attaching some attention mechanism (i.e. transformer encoder) to one or more of the point-separated segmentation stages. These are the ones being computed with a 1D convolution with kernel size of 1. What this does is keep the information between points isolated so that there is a direct mapping between input points and output predictions.

models/pnet_models.py

This is a file of mostly deprecated models, used in previous versions of the code. Can take a look if you are looking for inspiration or history of what was previously explored.

Jets_train.py

The JetPointNet training script sets up the environment and orchestrates the training of the PointNetSegmentation model using a dataset of processed jet collision events. Here's an overview of the script's workflow:

1. **Environment Setup**: Sets the specific GPU to be used for training and defines the maximum sample length (MAX_SAMPLE_LENGTH) as a constant for pre-processed data. Note again, this is hard-coded and was based on the dataset used in testing. Would be worth automating this so nothing breaks
2. **Custom Callbacks**: Implements a `BatchNormalizationMomentumScheduler` for adjusting Batch Normalization layers' momentum during training, and a `SaveModel` callback to save the model after each epoch. Currently the `BatchNormalizationMomentumScheduler` is omitted,
3. **Data Loading and Generator Functions**: Includes functions to load data from `.npz` files and to generate data batches for training, ensuring continuous data flow to the model during training.
4. **Training Utilities**:
    - A step calculator to determine the number of steps per epoch based on the batch size and data volume.
    - A model-saving function to persist the model after each epoch.
    - A learning rate scheduler to adjust the learning rate throughout training epochs.
5. **Model Initialization**: The PointNetSegmentation model is instantiated with the specified input shape and classes, followed by the Adam optimizer setup with an initial learning rate.

6. **Model Compilation**: The model is compiled using the custom `masked_mse_loss` function, and `masked_mae_loss` is used as a metric.
7. **Training Execution**: Triggers the model training process with defined callbacks including the learning rate scheduler and CSV logger to track the training progress.
8. **Output Information**: Prints the model summary, parameter count, size in megabytes, and training directory path for reference.
9. **Training Process**: Executes the model.fit() function with the training and validation data generators, specifying the number of steps per epoch and callbacks for logging and learning rate scheduling.
10. **Post-Training**: Calculates and prints the total training time in hours.

This training script is built to handle the complex nature of jet collision data and is specifically tailored for energy reconstruction in particle physics experiments using deep learning.

Recommendation: Read about what others have done, explore model architecture blocks, training parameters, loss functions, etc. Just be sure to have a consistent metric to compare the performance of the model after every setup change, to see if you are actually improving things..

Warning: Avoid putting actual data processing into the data loading and generator functions. If you need the dataset to be modified, do that in the preprocessing scripts and save it to disk. It will save you hours of headache as compared to having a heavy-to-compute data loader setup. This is a huge potential bottleneck that you want to keep clean and efficient.

Jets_test.ipynb

The jets-test.ipynb evaluation script for the JetPointNet model is designed to accurately determine the model's proficiency in reconstructing the total energy of particles within test datasets. It begins by setting up the computation environment to bypass GPU acceleration. The trained model's parameters are retrieved from the designated storage location using the `load_weights` function provided by the Keras framework.

The `load_data_from_npz` function is integral to the evaluation process, extracting features and labels, including the true total energy deposited by particles into the cells, from `.npz` format files. The script concentrates on the absolute Mean Absolute Error (MAE) as the key metric for performance evaluation.

For every sample in the dataset, the following steps are taken:

1. Data is loaded, including features and labels.
2. The model's predictions on the features are obtained.
3. The total predicted energy for the sample is calculated by summing up the predicted energy deposits across all cells.
4. The Mean Absolute Percentage Error (MAPE) is computed by contrasting the total predicted energy with the total actual energy collected from the cell deposits, omitting any masked values.

A visualization component, the `plot_energies` function, is incorporated to graphically compare predicted and actual energy values for a select set of samples, providing an immediate visual assessment of model accuracy.

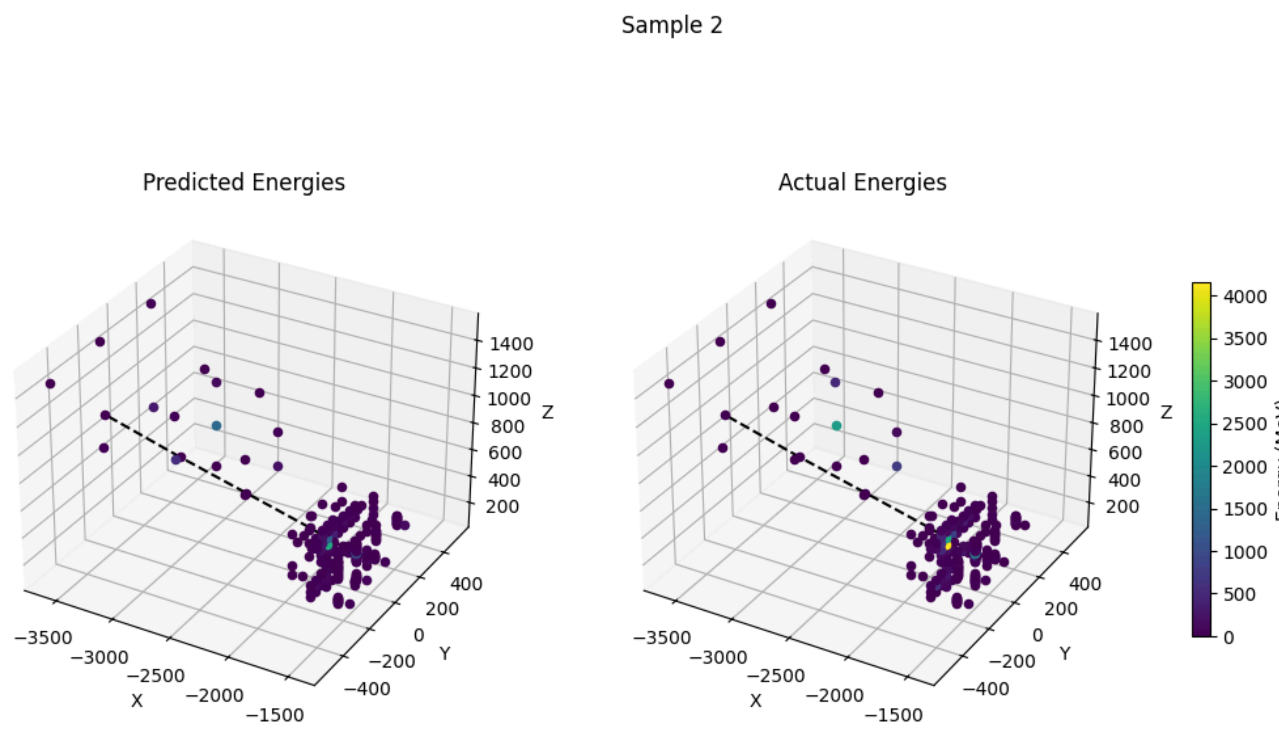


Figure 12: Example output plot:

Tracked particle is dotted line, cell colours represent energy contribution

Total Predicted Energy: 16.567 GeV

Total True Energy: 29.393 GeV

Percentage Difference: 43.63 %

The script concludes by computing the average MAPE for all samples to gauge the model's overall accuracy in predicting the total energy of particles. This calculated average MAPE is a critical measure, providing direct feedback on the model's capability in the context of total energy reconstruction.

In terms of performance, while quantitatively, it leaves some to be desired, qualitatively it seems to perform well in identifying which cells are the deposits of importance. This is promising, as it means that the bulk of work to be done for now is in optimizing fine-tuning the algorithm’s performance.

Conclusion

The body of work undertaken in developing the JetPointNet model has laid a strong foundation for the complex task of energy reconstruction in the ATLAS detector. The key findings include the successful development of a data preprocessing pipeline that translates CERN Root files into a format suitable for deep learning applications, and the adaptation of the PointNet architecture to handle the unique demands of particle physics data.

The configured model addresses the initial objectives by providing a framework capable of interpreting the intricate patterns of energy deposits within the detector. While the full extent of the model's accuracy is pending further results, the work completed signifies a step forward in employing deep learning for particle detection and energy reconstruction tasks.

For future work, it is recommended to focus on fine-tuning the model's parameters and exploring the effects of different network architectures on prediction accuracy. Moreover, optimizing the data preprocessing pipeline to handle larger datasets more efficiently could be beneficial. These efforts will aim to solidify JetPointNet's role as a tool in particle physics analysis and to potentially surpass the performance of traditional algorithms used within the field.

References

[1] I. Neutelings, "CMS coordinate system," TikZ.net, https://tikz.net/axis3d_cms/ (accessed Apr. 9, 2024).

[2] "Pseudorapidity," *Wikipedia*, Jan. 24, 2023. https://en.wikipedia.org/wiki/Pseudorapidity# (accessed Apr. 10, 2024).

[3] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," 2017. [Online]. Available: arXiv:1612.00593.

[4] F. Beaudette, "The CMS Particle Flow Algorithm," in *Proceedings of Calorimetry for High Energy Frontiers - CHEF 2013*, Paris, France, April 22-25, 2013.

[5] M. J. Swiatlowski, M. Hodgkinson, et al., "MLTree: ATLAS Athena package for calorimeter cluster imaging," ATLAS Jetetmiss, CERN GitLab. [Online]. Available: https://gitlab.cern.ch/atlas-jetetmiss/pflow/commontools/MLTree

Appendix A: Awkward Array Tutorial

For a range of tutorials on Awkward Array, you can visit the Awkward Array documentation page. Additionally, for a specific tutorial on using Awkward Array within high-energy physics, you can explore the HSF Training page. These resources offer detailed guidance and examples to help you work with this powerful library.

Appendix B: Uproot Tutorial

https://github.com/jpivarski-talks/2021-07-06-pyhep-uproot-awkward-tutorial
And the corresponding video lecture
https://www.youtube.com/watch?v=s47Nz0h0vcg