Hierarchical Clustering with Graph Neural Networks and DGL

NICHOLAS CHOMA

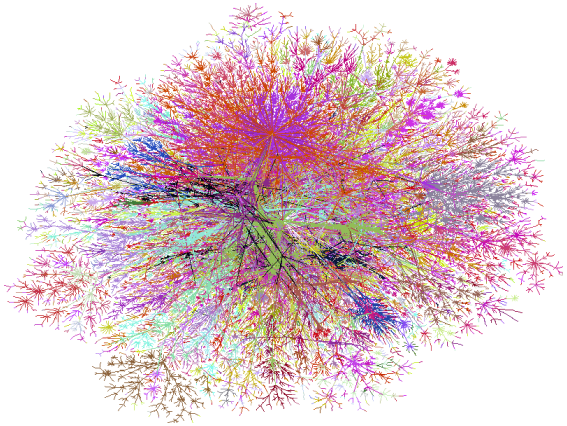
Agenda

- Introduction
- Particle Tracking
- Clustering Basics
- GNN Clustering
- Implementation with DGL
- Results and Next Steps

Agenda

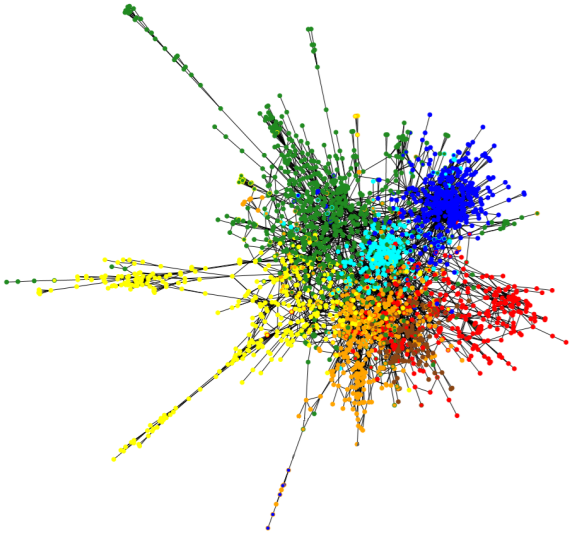
- Introduction
- TrackML
- Clustering Basics
- GNN Clustering
- Implementation with DGL
- Results and Next Steps

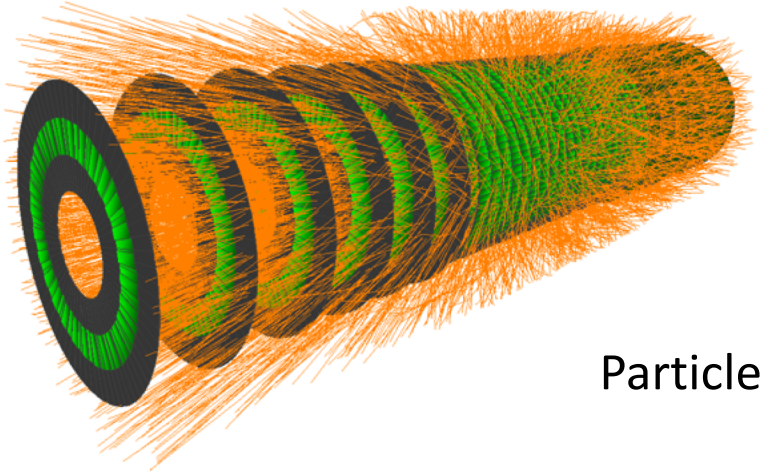
Clustering with GNNs



Sensor / Social Networks

Citation Networks

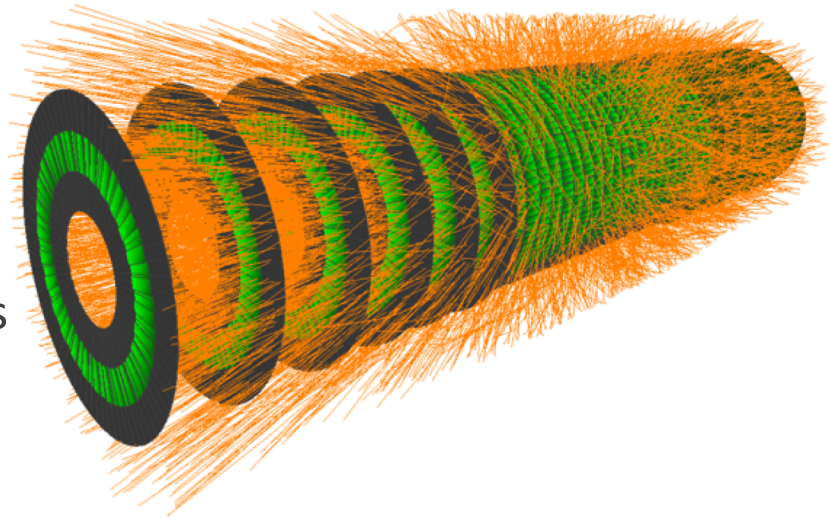




Particle Tracking

Particle Tracking

- Proton bunches circulate at LHC and collide at high energy
- Each collision produces many new particles, which spread outward in a shower
- To identify the types of particles and their kinematic properties, an applied magnetic field bends their trajectories
- These particles are recorded having passed through the detector cells
- From the recorded hits, the goal is to reconstruct the track each particle took through the detector



Agenda

- Introduction
- Particle Tracking
- Clustering Basics
- GNN Clustering
- Implementation with DGL
- Results and Next Steps

Particle Tracking

Given a set of $\approx 10^5$ hits created by $\approx 10^4$ particle tracks, cluster hits such that each cluster is associated with one track.

Input:

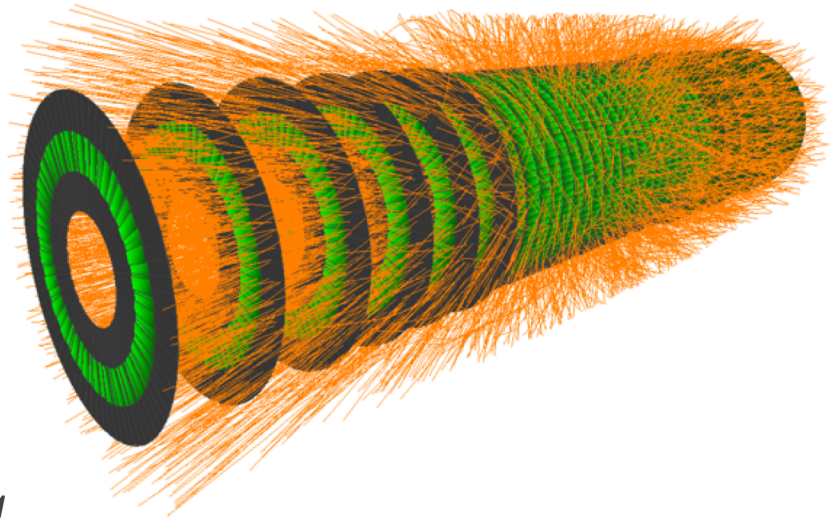
- N hits (x, y, z and detector ID)
- Detector cell pattern for each hit

Output:

- k clusters of the N hits

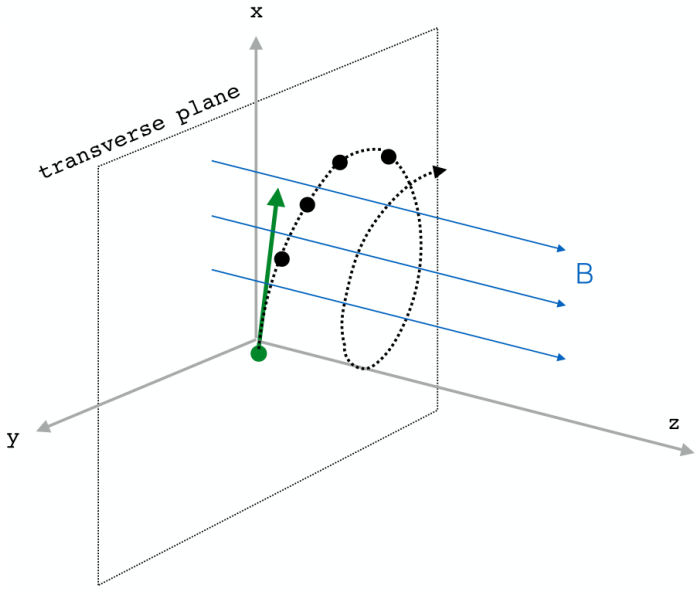
Challenges:

- Variable number of tracks which is not *a priori* known
- Inference must be efficient



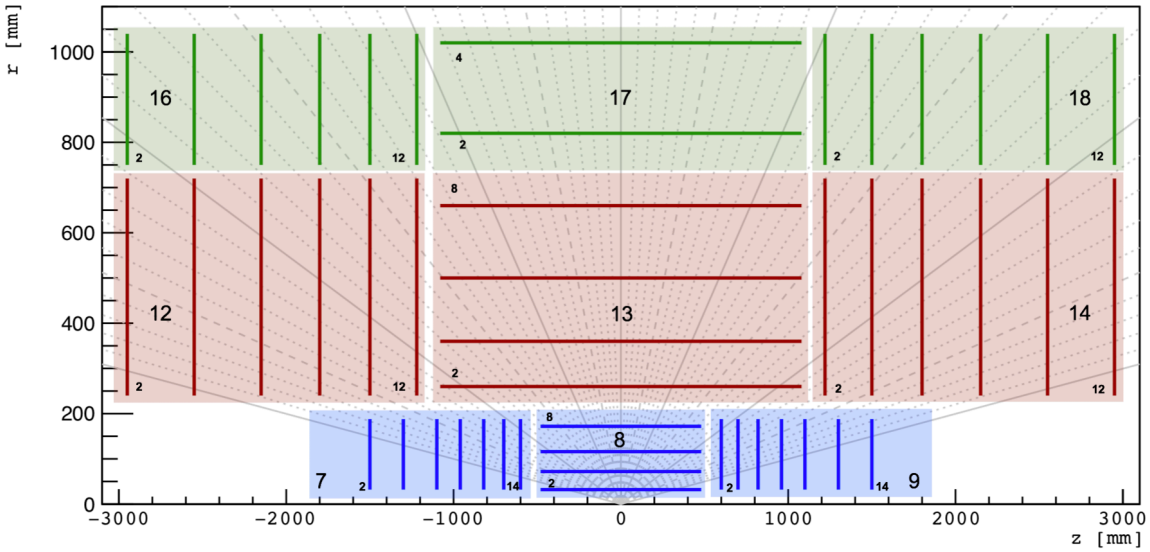
Particle Trajectory

- Particles follow helix trajectories under uniform magnetic field
- Helix radius proportional to particle energy
  - Particles guaranteed to have energy above threshold → reject hit triplets with small radii
- TrackML more complex due to non-uniform magnetic field



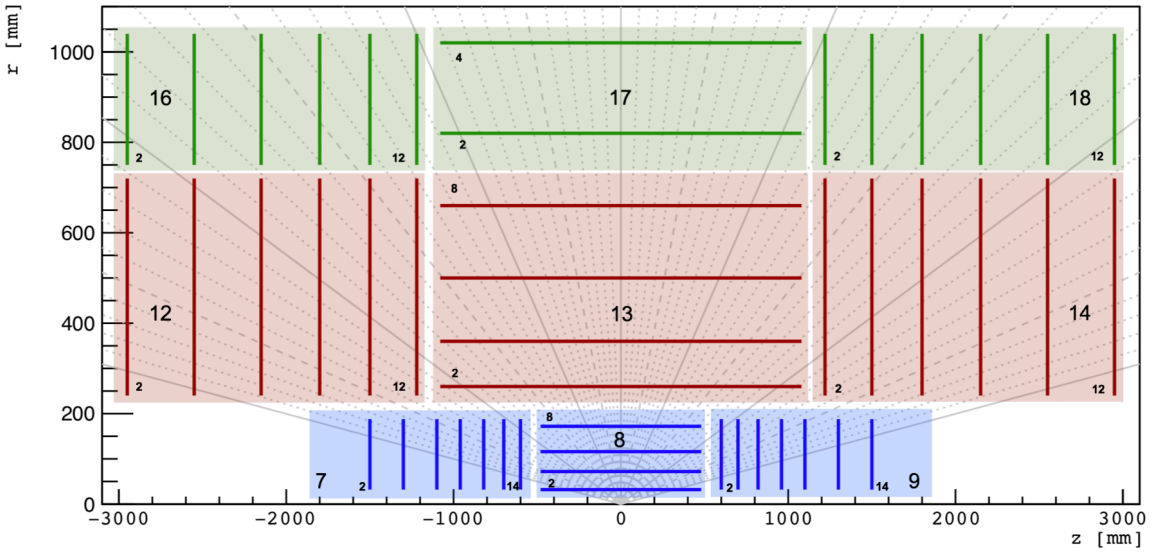
Particle Trajectory

- Particles created near origin of the detector
- No curvature in the (z, r) plane
  - Reject hit pairs which intersect z-axis outside $\approx (-50,50)$ mm



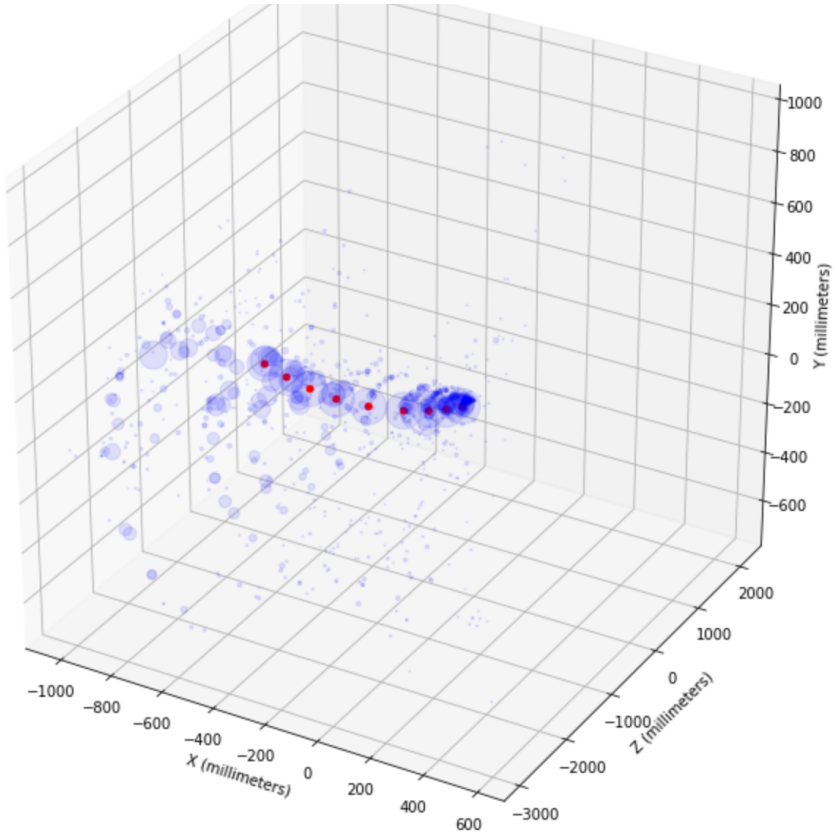
Detector Geometry

- Would like efficient method for determining important nearby layers
- Initial idea: manually enumerate over all nearby layers
  - Find nearby hits by querying nearby layers
- Not all nearby layers have hits from same track (holes are present)



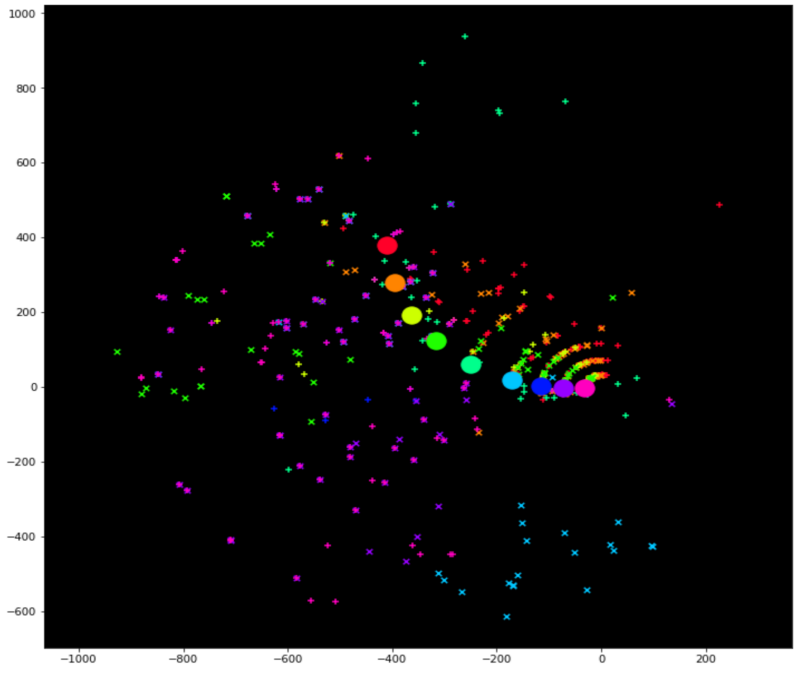
Baseline Method 1

- Identify candidate pairs of hits
  - Score all (!!) hit pairs with MLP on hit and pairwise features
- For each hit not already part of a track
  - Add nearest neighbor according to MLP score
  - Continue adding neighbors, ranked by average linkage to hits in current track
- Refine track selections
  - Reject tracks which do not have helical shape
  - Extend tracks by adding leftover hits
- ≈90% track recovery



Baseline Method 2

- Identify candidate pairs of hits
  - Find hit pairs from nearby detector layers
  - Use logistic regression to score pairs
- Extend all candidate pairs to triplets
  - Overlap allowed, ≈50M triplets
- Using priority queue, build tracks from triplets
  - Remove overlap from queue once a hit is selected for a track
- Refine track selection
  - Extend tracks with leftover hits
  - Combine tracks which fit some criteria for matching
- ≈92% track recovery



Baseline Method Themes

Top solutions to kaggle's TrackML competition generally operate in three stages:

1. Identify candidate pairs of hits
   - Accomplished with heuristics or machine learning
2. Construct tracks based on candidate pairs
3. Refine track selections
   - Reject tracks below certain threshold
   - Extend tracks with any appropriate leftover hits

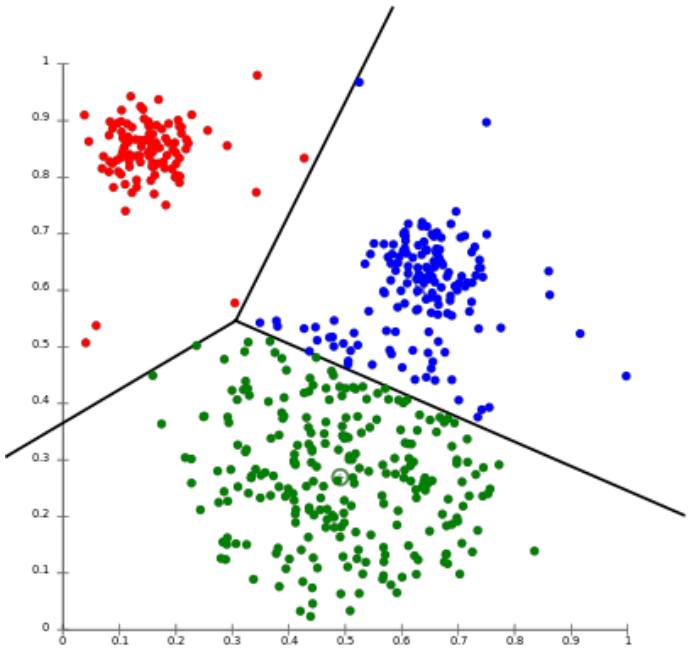
Some machine learning, but could an end-to-end solution do better?

Agenda

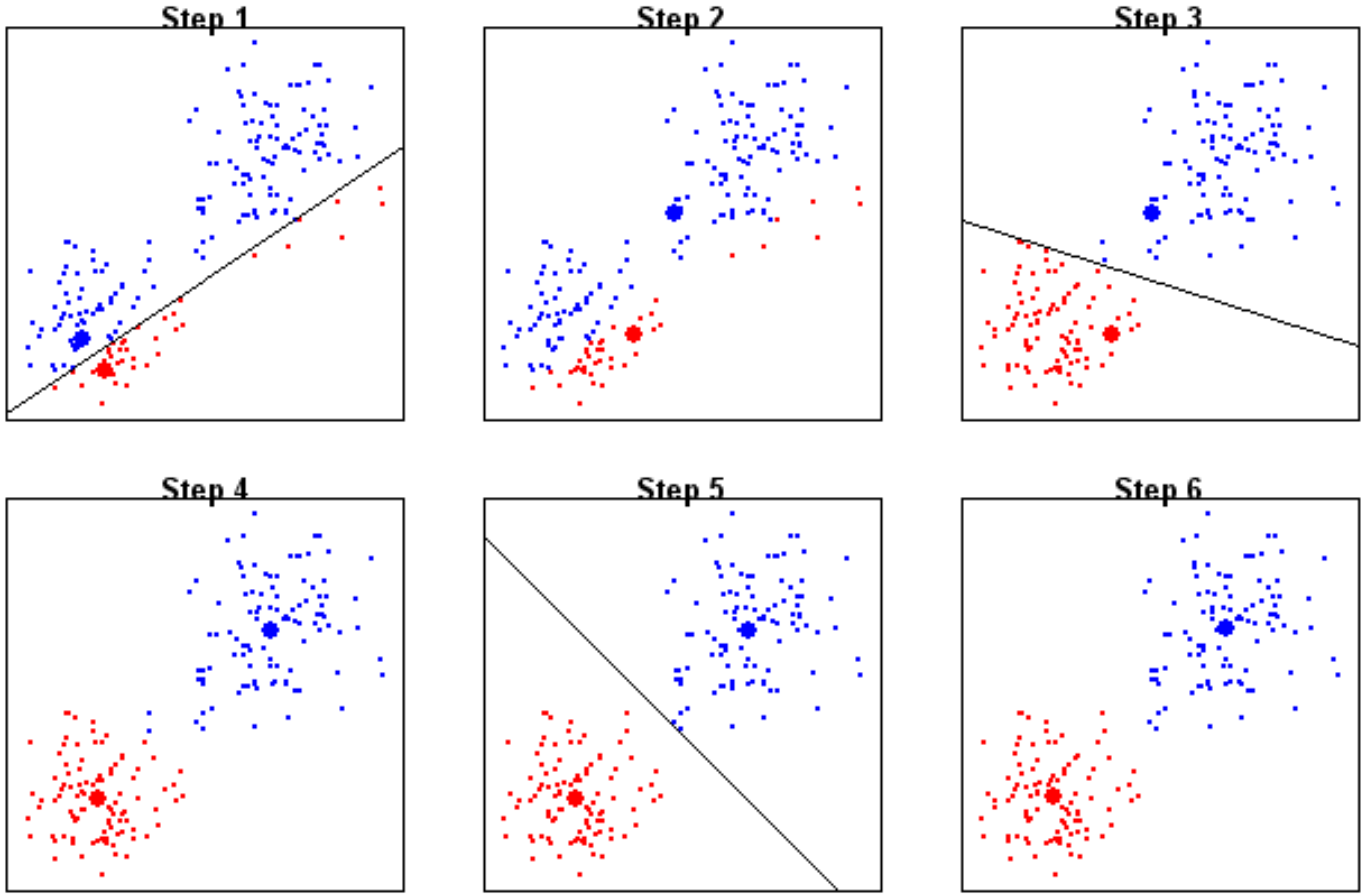
- Introduction
- Particle Tracking
- Clustering Basics
- GNN Clustering
- Implementation with DGL
- Results and Next Steps

k-means

- **Input:** $X = \{x_1, \ldots, x_n\}, x_i \in \mathbb{R}^d$
- **Output:** k clusters of X
- **Hyperparameters**
  - k, the number of clusters
  - i, the number of iterations for convergence
- **Algorithm**
  1. **Assign** points to current cluster centroids
  2. **Update** cluster centroids from assigned points
- **Time complexity**
  - $O(ndki)$

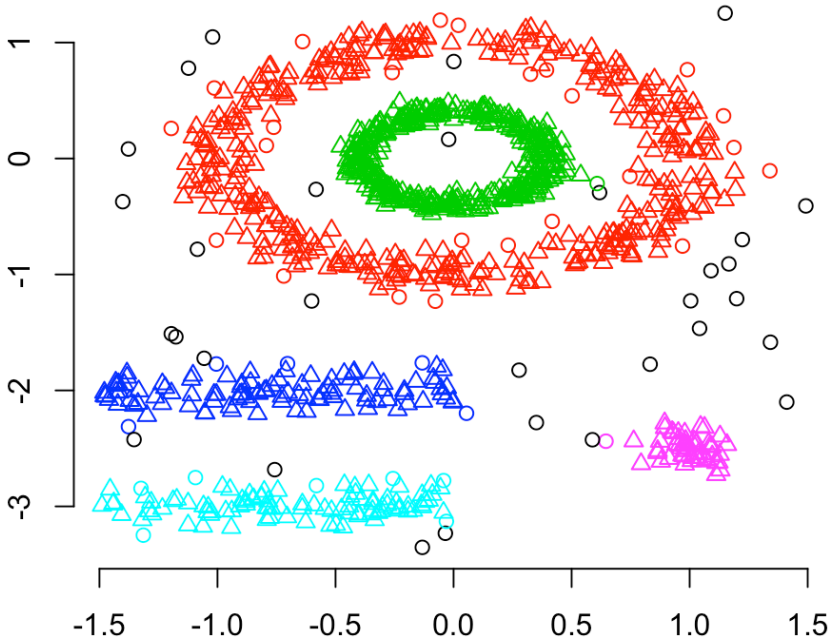


k-means



DBSCAN

- **Input:** $X = \{x_1, \ldots, x_n\}, x_i \in \mathbb{R}^d$
- **Output:** k clusters of X
- **Hyperparameters**
  - ϵ, the maximum distance between two points to be considered within the same neighborhood
  - m, the minimum number of points per cluster
- Related to spectral clustering
  - But no need to pre-specify number of clusters



DBSCAN

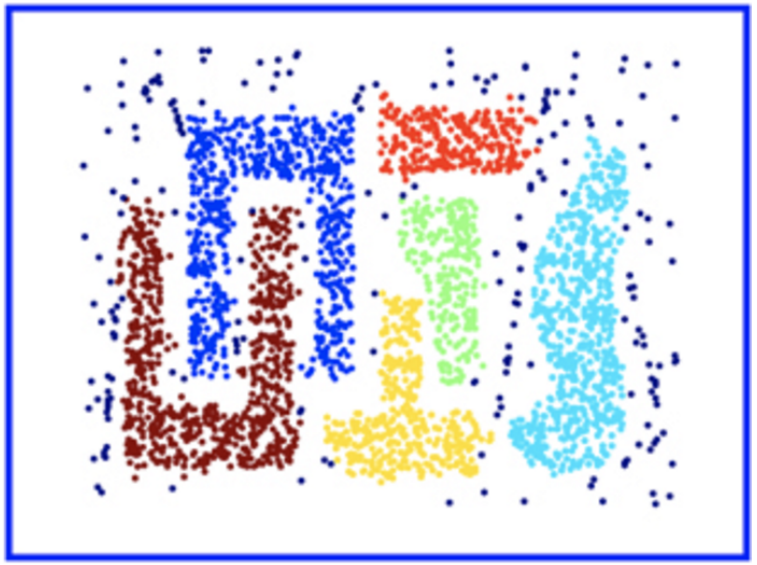
- **Algorithm**

```
C = 0
for x_i in X:
        if label(x_i) is None:
                N = range_query(x_i, epsilon)
                if |N| >= min_points:
                        assign X_i to C
                        assign all points in N to C
                        C += 1
                else:
                        assign x_i to noise
```

- **Time complexity**
  - $O(nd)$

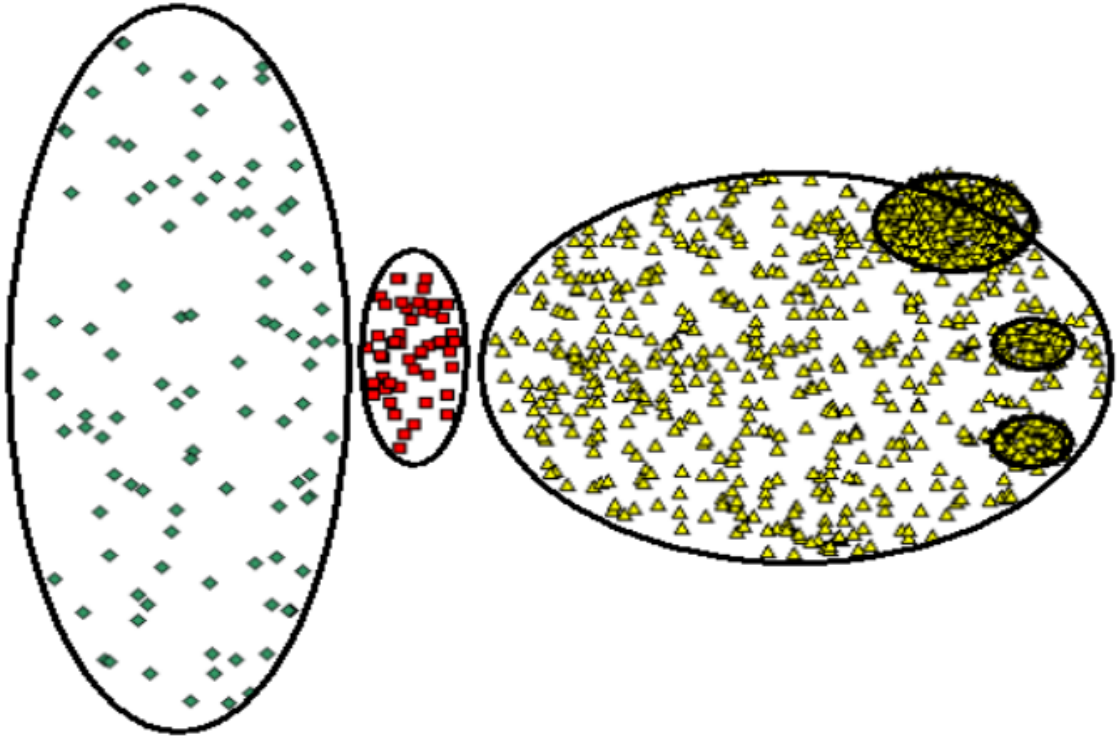
k-means vs. DBSCAN

- DBSCAN succeeds where k-means fails



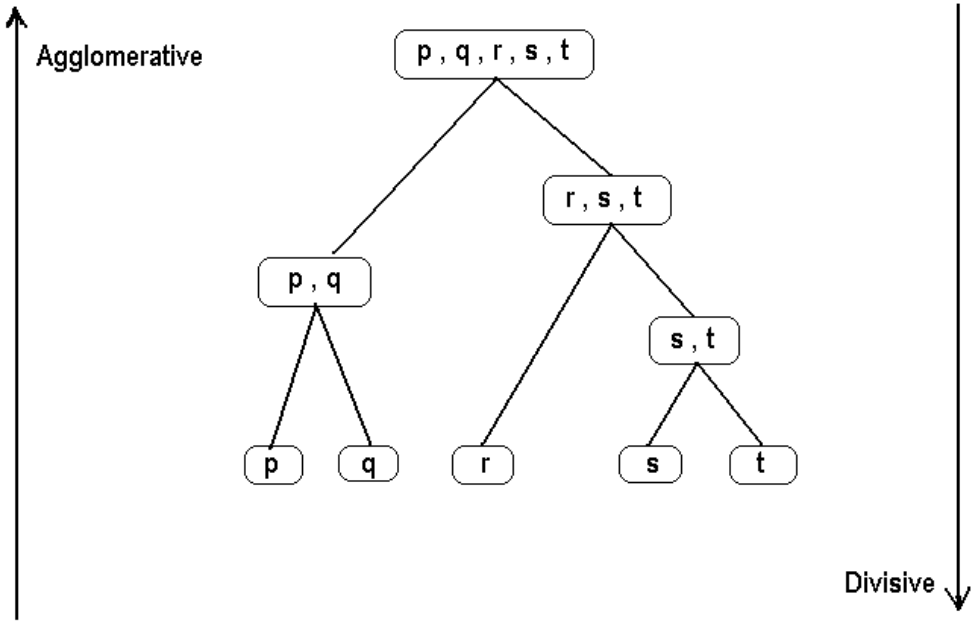
k-means vs. DBSCAN

- DBSCAN fails when cluster density varies drastically



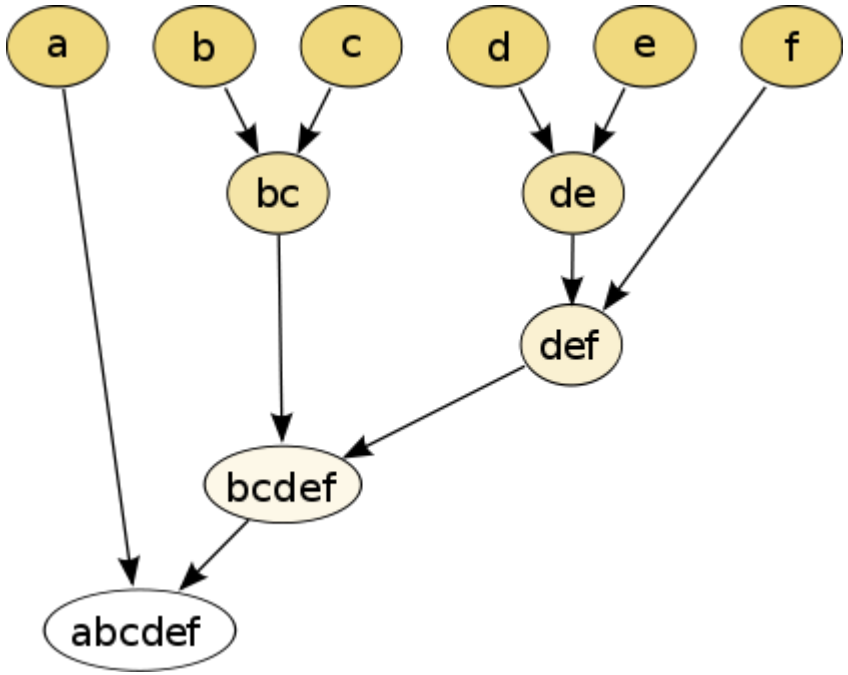
Hierarchical Clustering

- k-means and DBSCAN are *flat* clustering algorithms
  - Both return unstructured sets with no relation between sets
- Hierarchical clustering provides information relating sets to each other
- Two approaches:
  - Bottom-up (agglomerative)
  - Top-down (divisive)



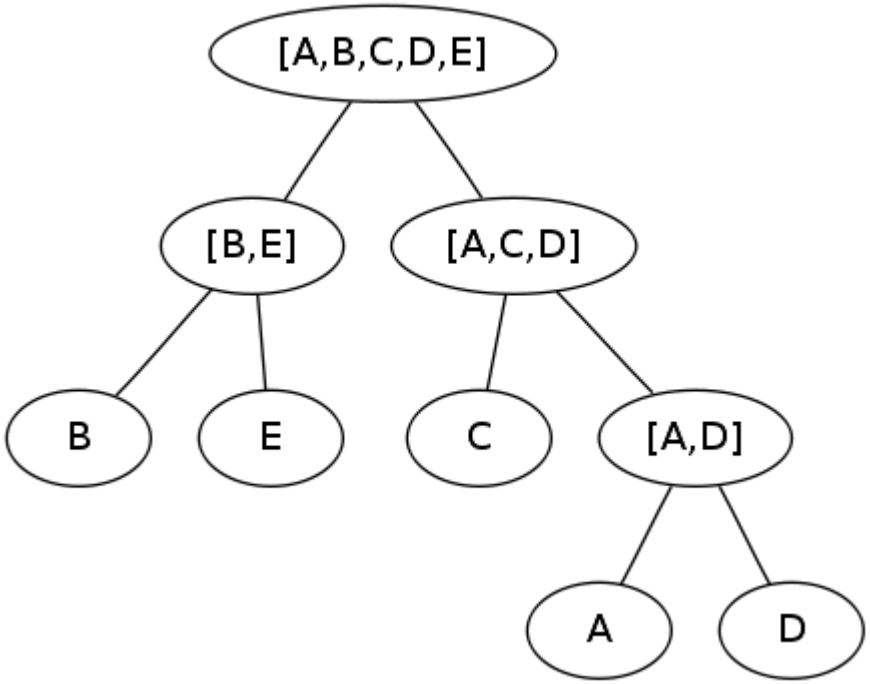
Agglomerative Clustering

- Each point begins in its own cluster
- Recursively merge clusters until every point belongs to one cluster
- Typical implementations operate in $O(n^2)$ time
- Faster methods exist, though rely on assumptions about problem dimensionality



Divisive Clustering

- Split input points with binary decision using e.g. k-means and $k = 2$ into two clusters at each level
- Recursively divide subclusters until either a *stop* criteria is met, or no additional points to split
- $O(n \log n)$ time complexity is easily achieved (with $O(n)$ divide function and even splitting)



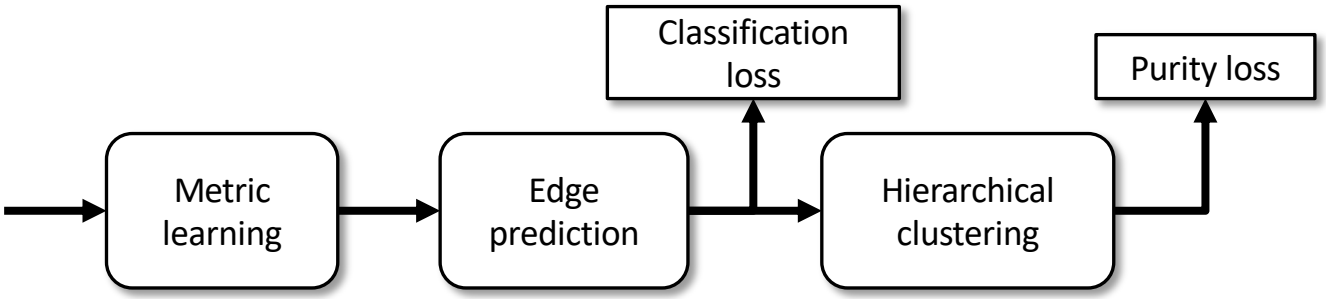
Agenda

- Introduction
- Particle Tracking
- Clustering Basics
- GNN Clustering
- Implementation with DGL
- Results and Next Steps

Recap, TrackML

- Given a set of $\approx 10^5$ hits created by $\approx 10^4$ particle tracks, cluster hits such that each cluster is associated with one track.
- **Input:**
  - N hits (x, y, z and detector ID)
  - Detector cell pattern for each hit
- **Output:**
  - k clusters of the N hits
- **Challenges:**
  - Variable number of tracks which is not *a priori* known
  - Inference must be efficient

Proposed Method

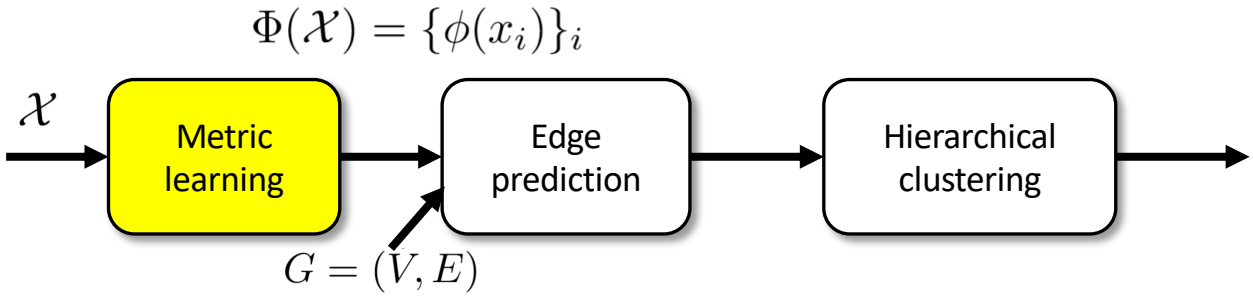


- End-to-end graph neural network model which is trained in stages

1. Construct graph by pre-selecting potential hit pairs, which become edges (hits are vertices)
2. Embed hits using proxy goal of classifying whether hit pairs belong to same track
3. Hierarchically cluster embedded hits into tracks

Construct Graph

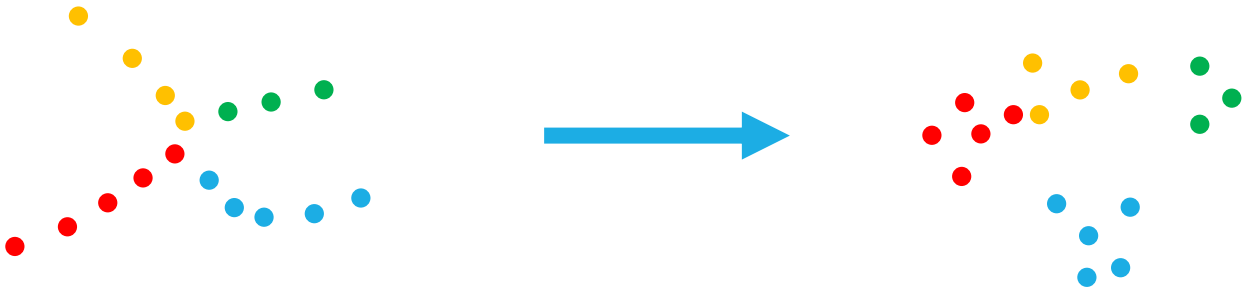
Construct graph by pre-selecting potential hit pairs, which become edges



- Relevant hit pairs are selected in stages
  1. Embed hits into new space with Euclidean distance metric
  2. Build k-d tree using embedded hits and find all nearby hits within ϵ-neighborhood
  3. Refine edge selection using an edge classification model which takes as input edge pairs and pairwise features

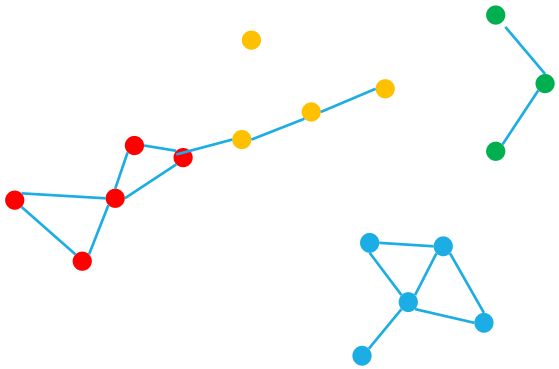
Construct Graph

- Hits are embedded from original feature space to new space with Euclidean distance metric
  - Hits belonging to same track are nearby
  - Hits belonging to different tracks are far apart



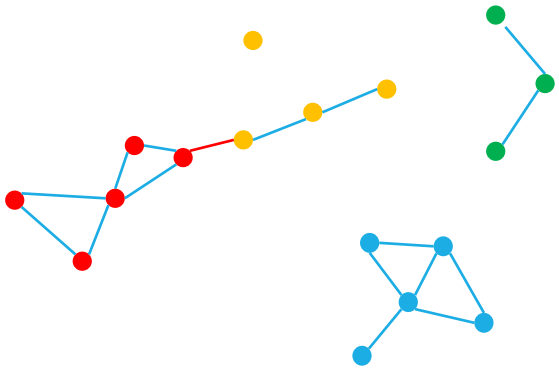
Construct Graph

- From embedded hits, construct k-d tree with Euclidean distance
  - Efficient querying for fast graph construction
- Construct graph by finding hits within ϵ-neighborhood of each hit



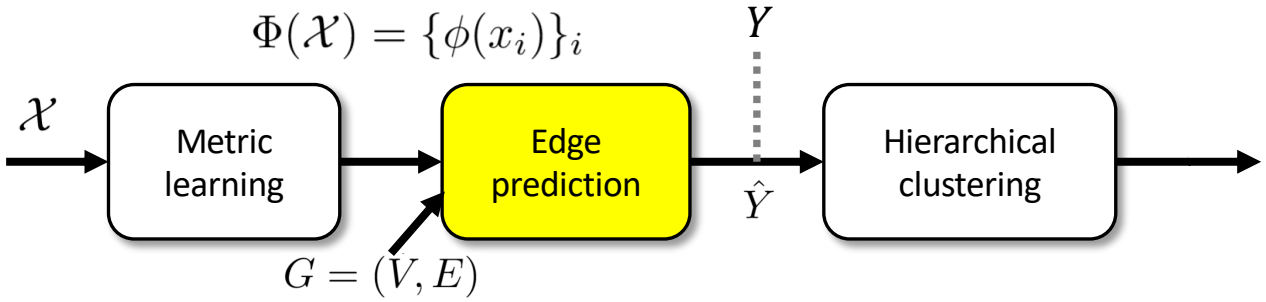
Construct Graph

- Refine edge selection using pairwise hit features, predicted using e.g. an MLP
- Initial graphs have ≈ 100 million edges (1% of dense edges)
- Refined graphs have ≈ 6 million edges (>80% true edges recovered)



GNN Edge prediction

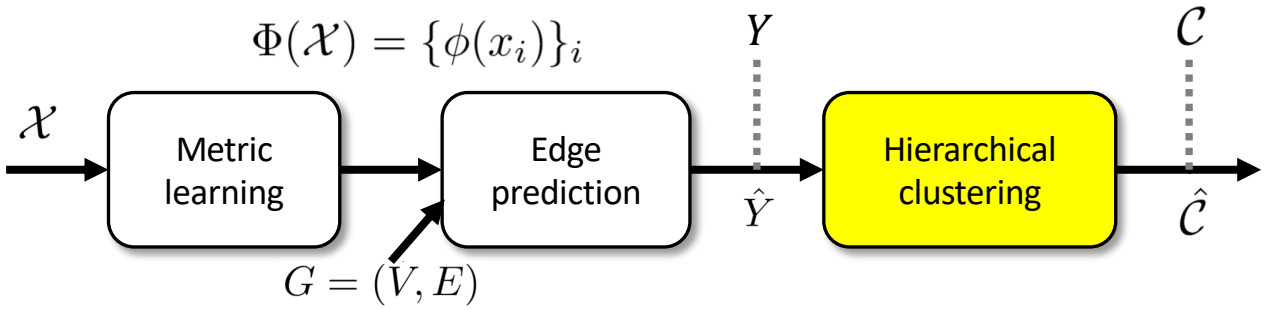
Embed hits by classifying whether hit pairs belong to same track



- Train graph neural network (GNN) model with sparse graph constructed by metric learning stage
- GNN improves hit embedding by using information from each hit's local neighborhood
- Model is a message-passing GNN, where each layer re-computes weighted edges

Clustering

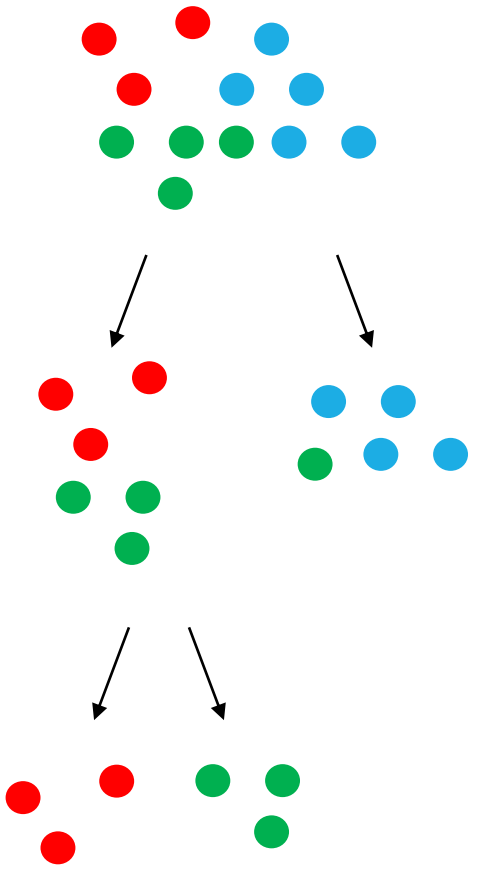
Hierarchically cluster embedded hits into tracks



- Divisive (top-down) hierarchical clustering strategy
- Parameterized by two sub-modules
  - Split module, which divides cluster into two subsets
  - Stop module, which learns whether to continue splitting

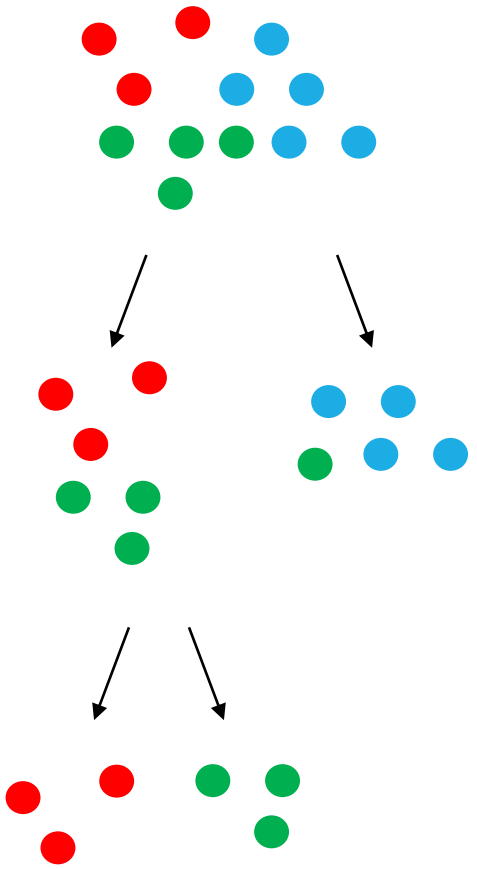
Clustering, split loss

- Split module must
  - divide hits of different tracks into separate clusters
  - perform balanced division to keep overall runtime sub-quadratic
- Parameterized by MLP
  - **Input:** embedded hits $x'_1, \ldots, x'_N$, plus global information for each cluster (mean, variance)
  - **Output:** $p(x'_1), \ldots, p(x'_N)$, the probability that each hit goes left
- Reduce entropy of hits within same track
- Maximize entropy of hits between tracks

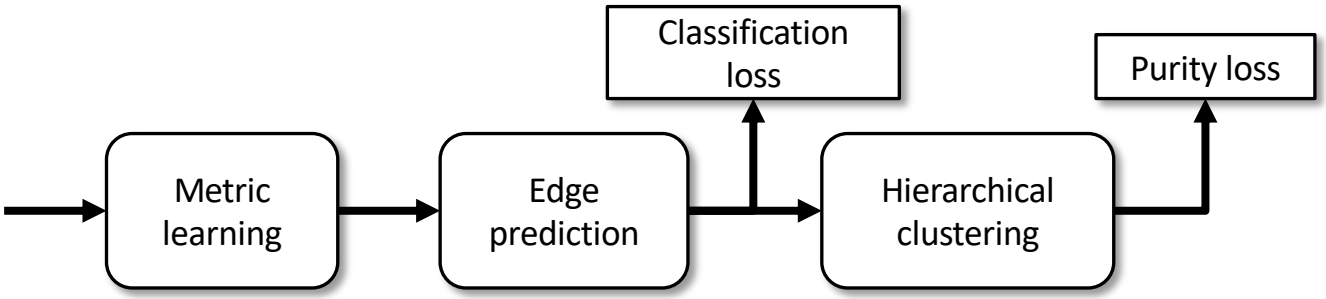


Clustering, stop loss

- Stop module prevents model from further dividing points once doing so will lower score
- Parameterized by MLP
  - **Input:** embedded hits $x'_1, \ldots, x'_N$, plus global information for each cluster (mean, variance)
  - **Output:** $p(c_1)$, …, $p(c_{|C|})$, the probability that each cluster should stop
- Training
  - Compute clustering score for all hits within each node in binary tree
  - If a node's score is higher than that of its weighted children, indicate stop
  - Else indicate continue
- Clustering loss per tree node is a combination of track precision and recall
  - Similar to TrackML scoring, but guaranteed to never be zero exactly



End-to-end clustering



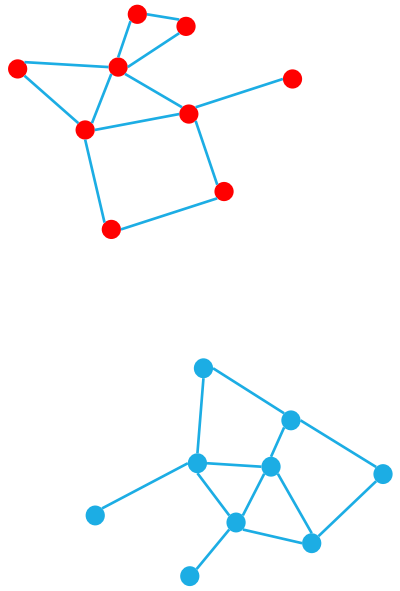
- Resulting architecture can be trained end-to-end
- Fully supervised
- Handle variable number of clusters
- Time complexity is $O(n \log n)$
  - No quadratic step

Agenda

- Introduction
- Particle Tracking
- Clustering Basics
- GNN Clustering
- Implementation with DGL
- Results and Next Steps

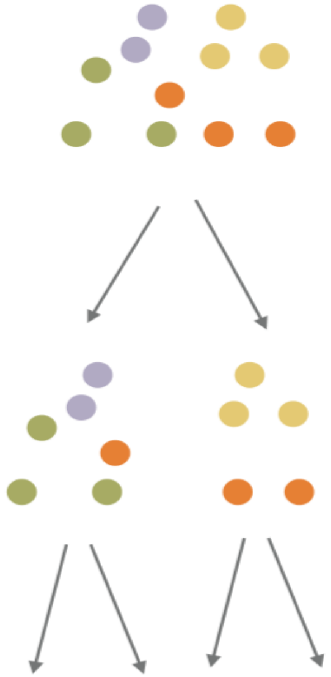
Graph Neural Networks

- Deep Graph Library (DGL) framework with PyTorch back end allows for simplified, fast training with sparse graphs
- Key features
  - Improved support for sparse autograd
  - Batched training with sparse matrices
  - Optimized built-in functions for standard GNN operations
  - Simplified interface which abstracts implementation for hardware



Recursive training

- Recursive training requires keeping track of
  - Unequal-sized clusters
  - Changing number of clusters within each tree node
  - Changing number of tree nodes
- Top of binary tree
  - Few tree nodes
  - Many clusters within each tree node
- Bottom of binary tree
  - Many tree nodes
  - Few clusters within each tree node
- Parallelization strategy must account for changing cluster, tree shape

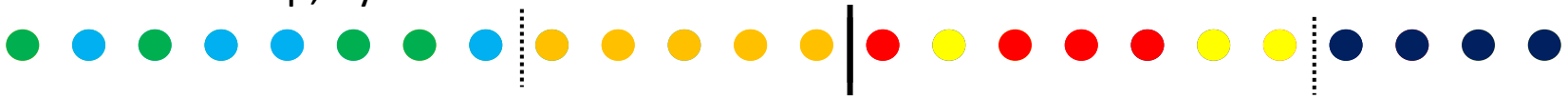


Recursive training, one step

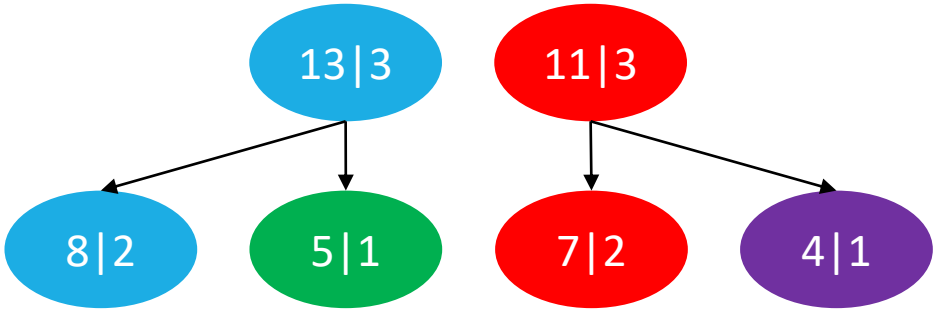
Before division step, by binary tree node

Before division step, by track ID

After division step, by track ID

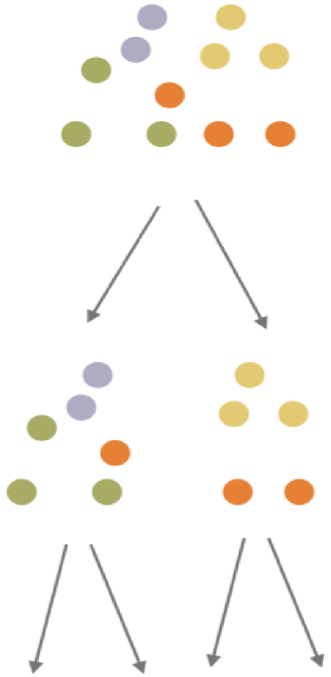


After division step, by binary tree node



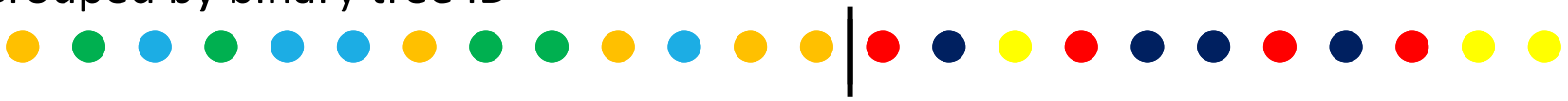
Recursive training

- Parallelize by grouping hits at each split level
  - By track
  - By tree ID
  - By track and tree ID
- DGL allows to compute aggregate statistics over groups of hits in parallel
- Map between different methods of groupings using permutations

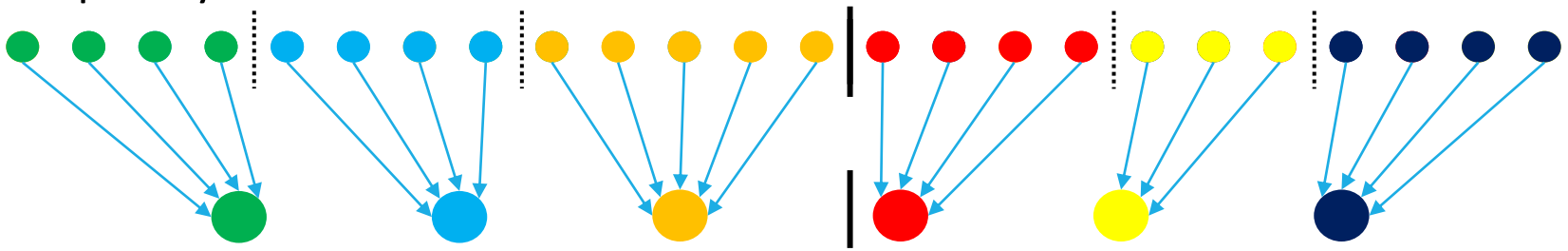


Grouping with permutations

Grouped by binary tree ID



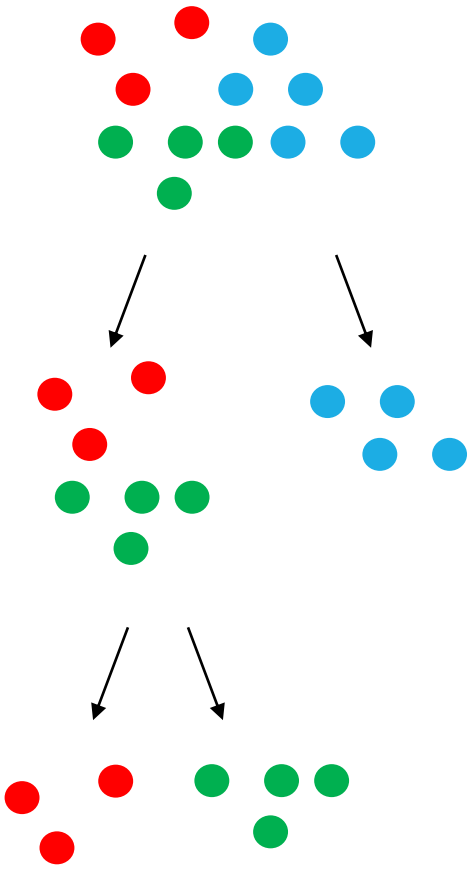
Grouped by track ID



Grouped by track ID and tree ID using DGL's aggregation commands

Additional complexities

- As tree nodes contain fewer hits, some will stop splitting
- Must be able to remove hits belonging to tree IDs that have stopped
- Use DGL's batch, unbatch, and subgraphs commands to handle hit removal
- Update permutations between groupings using argsort



Agenda

- Introduction
- Particle Tracking
- Clustering Basics
- GNN Clustering
- Implementation with DGL
- Results and Next Steps

Results

- Entire architecture may be trained end-to-end
  - Capable of handling variable and a priori unknown number of clusters
- Total complexity is $O(n \log n)$ due to sparse GNN training
  - High throughput during inference
- So far, 88% track reconstruction on 10% tracks per event (randomly sampled)
  - Physics methods achieve 94% reconstruction on full events (10^5 hits)

Next steps

- Improvements in metric learning allow for better graph sparsity
  - Further feature extraction using hit cells
  - Transform input domain which account for detector's irregular magnetic field
- GNN has difficulty optimizing full event sizes
  - Explore graph sampling approaches for better control of minibatch sizes
  - Multiple passes through GNN: parameter sharing at no additional cost in limited GPU memory