End-to-End Hierarchical Clustering with Graph Neural Networks

Nicholas Choma

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of Master of Science

at

New York University

2019

Contents

Abstract

The objective of this thesis is to develop a data-driven, hierarchical clustering method which is capable of operating on large point cloud datasets, necessitating a runtime which is sub-quadratic. Hierarchical clustering is noteworthy for its ability to produce multiscale views of data, allowing for rich and interpretable representations, and for its ability to cluster when the number of clusters is not specified *a priori*. To date, deep learning methods for clustering have primarily focused on a narrower class of models which cluster using partitioning strategies and require as input the number of clusters to produce. In this work, we introduce the clustering graph neural network, extending previous research into graph neural networks to handle large clustering tasks where the number of clusters is variable and not pre-specified. Our architecture is fast, operating with $O(n \log n)$ time complexity, and we note its amenability to high levels of parallelization. Because each stage is differentiable, we emphasize that our architecture is capable of end-to-end training, leveraging signal throughout the learning pipeline as part of a multi-objective loss function. Finally, we demonstrate the clustering graph neural network on a challenging particle tracking task, which, while unable to outperform highly-tuned and domain-specific baselines, nevertheless achieves high performance while remaining flexible to a wide array of clustering tasks.

Chapter 1

Introduction

Hierarchical clustering methods offers to provide new and important insights throughout a variety of scientific and practical fields. Text and speech analysis, image representations, latent structure in DNA, and a multitude of tasks in particle physics and astrophysics are just a few of the areas in the deep learning revolution is transforming as torrents of new data streams become available. While current techniques provide useful tools for analyzing and detecting underlying structure in data, efficient methods are needed for handling tasks which offer a form of supervision, such as in particle tracking.

In this work, we leverage classical hierarchical clustering frameworks with the emerging field of geometric deep learning to produce the clustering graph neural network. The clustering graph neural network is fast, operating in $O(n \log n)$ time, and easily parallized, efficiently detecting thousands of clusters on a single GPU. Unlike recent deep learning approaches, which tend to operate with a partitioning strategy rather than hierarchical, our architecture has no requriement of prior knowledge of the number of clusters. Thus is capable of handling variable sample sizes, producing a multiscale view of the data as output. The focal application of this work lies in particle tracking, and specifically in the TrackML Challenge, a difficult task in which each sample has two orders of magnitude more clusters than other current deep learning benchmarks for clustering.

We begin with an overview of particle tracking and TrackML Challenge, with emphasis placed on traditional physics-based approaches which leveraging domain structure and expertise to produce efficient and accurate methods. In chapter 2, we outline the clustering graph neural network in detail, explaining along the way important considerations torwards the development of the architecture. Chapter 3 presents our training strategy and numerical results for the TrackML dataset, including intermediate results on the metric learning stage of our pipeline. We briefly mention several future directions of our work in chapter 4 before concluding in chapter 5.

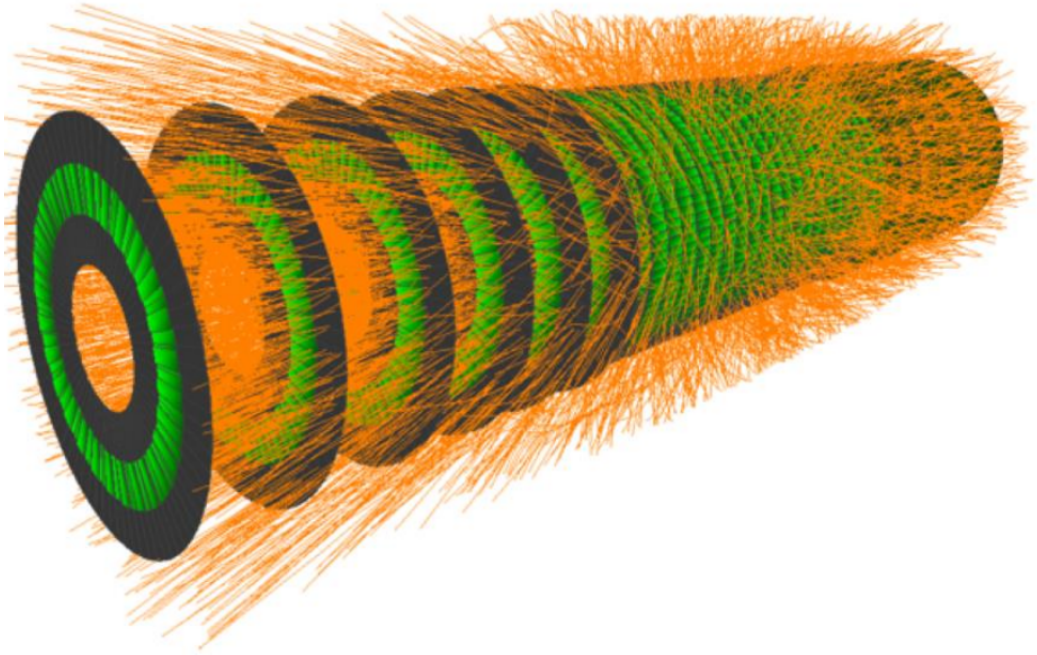

Figure 1-1: A single event from the TrackML dataset, where the orange lines represent particle trajectories through the detector layers. [24]

1.1 TrackML

Within a particle collider such as the Large Hadron Collider (LHC), streams of proton bunches circulate very near the speed of light in opposite directions. At specified locations, such as within the Atlas detector at the LHC, these streams intersect, resulting in thousands of high-energy collisions between the proton bunches. These collisions can be thought of as miniature big bangs, and observing the products of such collisions offers scientists a glimpse into the make-up of our universe.

The primary products of these collisions decay far too rapidly to be observed, but secondary and later particles spread outward in a shower. Eventually, they come into contact with the detector cells, which can be considered as a massive and highly sensitive camera to capture the event. An applied magnetic field bends the trajectory of the particles, which assists in recovering their type and kinematic properties. The task of TrackML, then, is to recover the trajectory of each particle from the information recorded by the detector [24].

1.1.1 Dataset and Challenge

Figure 1-1 illustrates a single event within the TrackML detector. Each sample i within the TrackML dataset consists of $N^{(i)}$ *hits* - an impact site where a particle passed through a detector layer. Each hit has an associated set of features, where consist of:

- An x, y, z coordinate

- An identifier specifying where in the detector the hit impacted
- A set of *cell* features that contain information on how the particle passed through the detector, each of which has a position and magnitude

Hits for sample i can be considered as entities within a general domain:

$$X^{(i)} = \{x_1^{(i)}, \ldots, x_{n_i}^{(i)}\}, x_j^{(i)} \in \mathcal{X}. \tag{1.1}$$

Every sample i also consists of a variable number of clusters

$$C^{(i)} = \{c_1^{(i)}, \ldots, c_{k_i}^{(i)}\}, \tag{1.2}$$

and each cluster $c_j^{(i)}$ is a partition of $\{1, \ldots, n_i\}$. Each sample also has a cluster weighting for each cluster:

$$W^{(i)} = \{w_1^{(i)}, \ldots, w_{k_i}^{(i)}\}, \tag{1.3}$$

and the sum of the weights for the sample sums to 1:

$$\sum_{j=1}^{k} |c_j^{(i)}| \times w_j^{(i)} = 1. \tag{1.4}$$

The TrackML challenge consists of finding the true clusters C, given the input features X. Notably absent from this setup is *apriori* knowledge of the number or size of the clusters within each sample. Given the ground truth clusters C and a set of predicted clusters C', the TrackML score $R(C, C')$ is computed as follows:

$$R(C, C') = \sum_{i=1}^{|C'|} r(C, c_i'), \tag{1.5}$$

where $r(C, c_i')$ is the TrackML score computed on predicted cluster c_i'. The score for each predicted cluster is then computed as follows:

$$r(C, c_i') = \begin{cases} w_j \times |c_j \cap c_i'| & \text{if } \frac{|c_j \cap c_i'|}{|c_j|} > 0.5 \text{ and } \frac{|c_j \cap c_i'|}{|c_i'|} > 0.5 \\ 0 & \text{else,} \end{cases} \tag{1.6}$$

where j is set as

$$j = \operatorname*{argmax}_k |c_k \cap c_i'|. \tag{1.7}$$

That is, the score of one predicted cluster is nonzero only if a large majority of its points belong to the same true cluster, and if the majority of the true cluster is contained within the predicted cluster. A perfect clustering will thus lead to a score of 1, while a random clustering will almost certainly have a score of 0.

1.1.2 Domain and Derived Features

When designing clustering algorithms for TrackML, it is typical to begin by considering pairwise interactions as a starting point [11]. There are several qualities inherent to the collision events which provide insight as to which pairs of hits are unlikely to belong to the same particle track.

Helix radius: Because the detector imposes a magnetic field on the particles, the particle tracks are helical in shape, with radius proportional to the particle's kinetic energy (which remains constant since the detector removes only a negligible amount). Within TrackML, there is a lower bound of 250 mega electronvolts (MeV) kinetic energy for every particle. Thus, considering the magnetic field B of the detector, where $B = 2$ Tesla for TrackML, the minimum helical radius ρ any particle trajectory can have is $\rho_{min} = 250$ mm. Computing the energy p_t in MeV of a particle from its radius in mm and the magnetic field in Tesla is given as

$$p_t = \frac{\rho}{0.6}. \tag{1.8}$$

Considering two points

$$\{v_1 = (x_1, y_1), v_1 = (x_1, y_1)\}, \tag{1.9}$$

and assuming they belong to the same track and that the track passes through the origin (true for nearly every particle within TrackML), ρ is given by

$$\rho = 0.5 \times \frac{\mathrm{d}r}{\sin(\mathrm{d}\phi)} \tag{1.10}$$

where

$$\mathrm{d}r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{1.11}$$

and $\mathrm{d}\phi$ is the positive angle between v_1 and v_2:

$$\mathrm{d}\phi = \arccos\left(\cos[\arctan^2(y_2, x_2) - \arctan^2(y_1, x_1)]\right). \tag{1.12}$$

Thus if any two points have a ρ value less than 250 mm, it is unlikely that these points belong to the same track. Figure 1-2 illustrates an exaggerated trajectory as curved by the magnetic field within the detector.

Propagation through the z-r plane: Since the magnetic field of the detector does not affect the trajectory of a particle along the z-axis, the particle follows a straight path within the z-r plane, where r is given as

$$r = \sqrt{x^2 + y^2}. \tag{1.13}$$

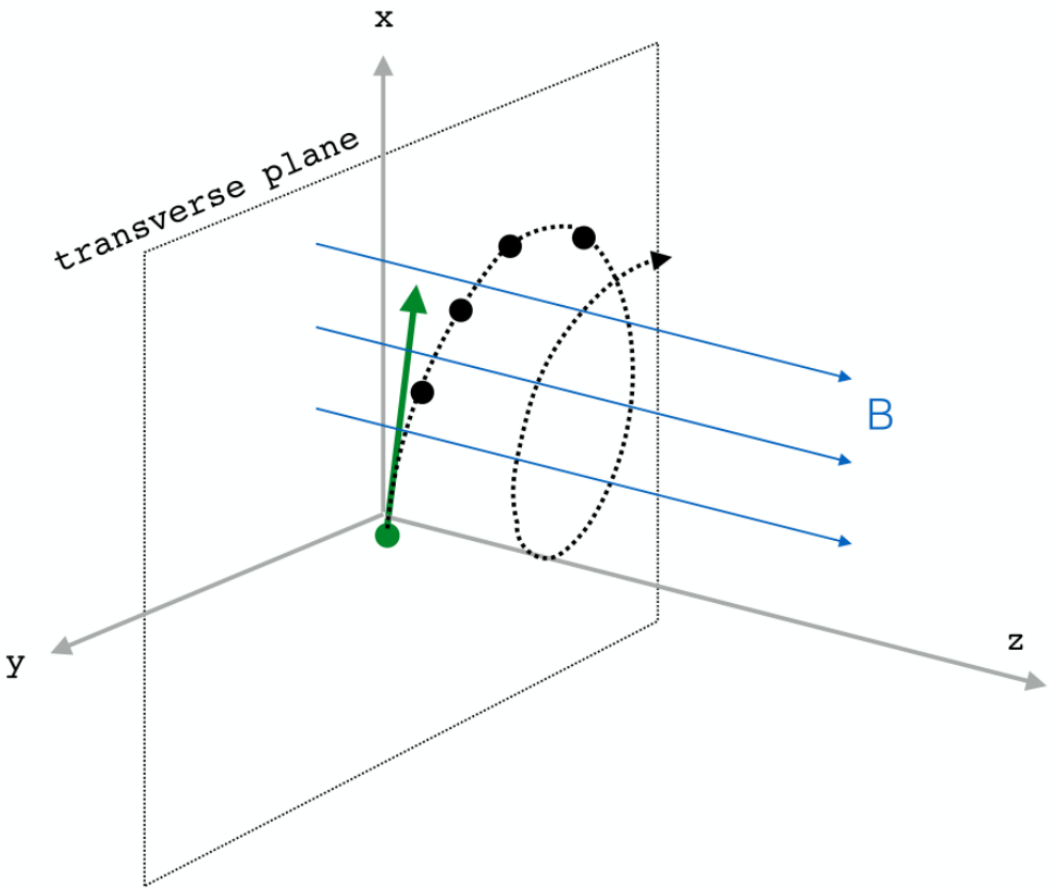

Figure 1-2: A single particle and its trajectory as it passes through the magnetic field of the detector. [24]

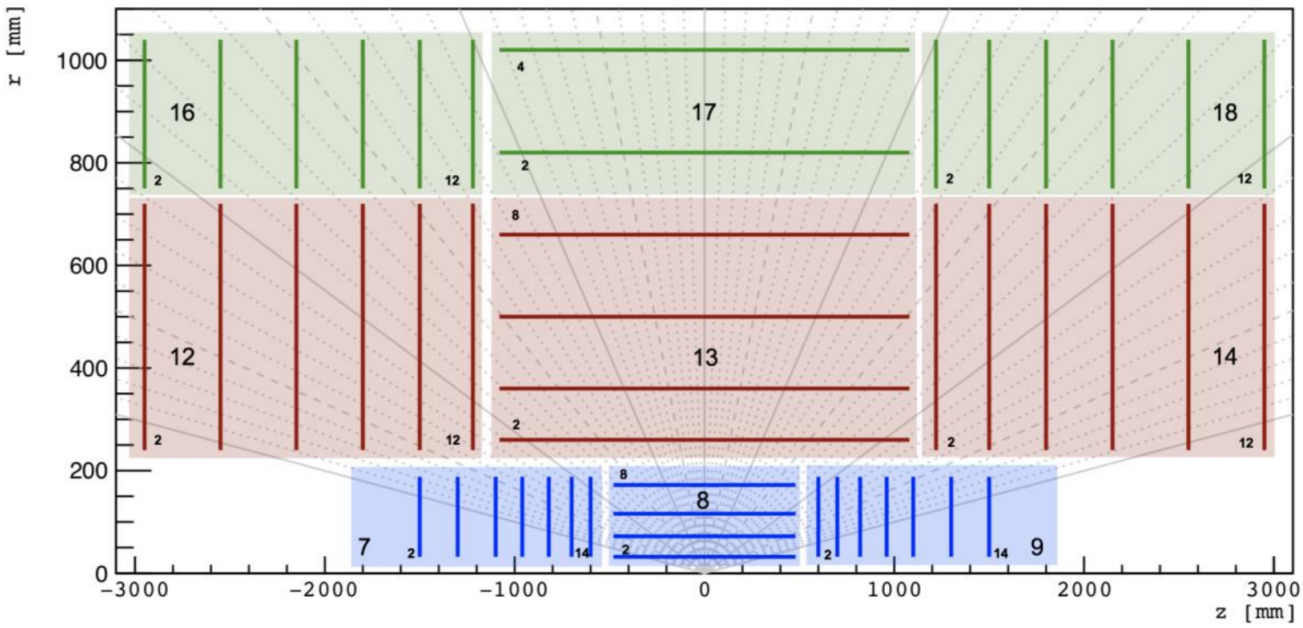

Figure 1-3: The detector and its layers as viewed from the z-r plane. [24]

The particle's propagation is then given as

$$z = z_0 - rc, \tag{1.14}$$

where z_0 is the starting location on the z-axis of the track, and c is given between two hits as

$$c = \frac{\mathrm{d}z}{\mathrm{d}r}. \tag{1.15}$$

Because tracks are likely to originate within 200 mm of the origin along the z-axis, for any z_0 given as

$$|z_0| = |z_1 + r_1 c| > 200, \tag{1.16}$$

the pair of hits likely do not belong to the same track. Figure 1-3 shows the z-r plane of the detector.

These features form the basis of the baseline methods which perform with high accuracy in the TrackML competition. For our purposes, they allow for slightly better accuracy with much faster convergence during the training of our graph neural network model.

1.1.3 Baseline Methods

We use as points of comparison the methods which have performed best in Kaggle's TrackML accuracy phase competition. These methods tend to follow a similar strategy:

1. Pairs of hits are scored to identify candidate pairs which have a higher likelihood

of belonging to the same track.

2. Using these pairwise scores, candidate tracks are constructed.

3. Candidate tracks are iteratively refined through exploitation of domain expertise and using features specific to the competition.

Specifically, we consider the first and second place strategies as baseline comparisons against our method, respectively labeled All Pairs [2] and Layerwise [1] solutions.

Baseline, All Pairs: The All Pairs solution begins by extracting features between pairs of hits, and predicting the probability of each pair belonging to the same track. Here, true pairs are those whose hits belong to the same track, and false pairs are those whose hits do not. This classification task is performed using a multi-layer preceptron, where training is completed in stages using hard negative mining to offset the fact that most pairs of false pairs are easy to classify.

Next, the All Pairs solution builds tracks one hit at a time. A hit is seeded which begins a track, and the track is grown by adding the hit which is closest to the track, measured by average linkage distance using the multi-layer preceptron to all hits already contained in the track. Figure 1-4 illustrates the reconstruction process on one track. No overlap is allowed, so once a hit is inserted to a track, it is unavailable for use in later tracks. Once the average linkage distance of the next hit falls below a threshold, the track stops growing.

The track selection is refined by evaluating how much noise is needed for each candidate track matches a perfect helix pattern. Those which require noise terms above a specified threshold are rejected, and the hits contained in those tracks become available again. Tracks are also extended by incorporating hits from rejected tracks which have a low average linkage distance to the track.

While the All Pairs method does attain an impressive 90% TrackML score, its reliance upon scoring every pair of hits leads to quadratic complexity, which ultimately is far too slow for the throughput required by the High Luminosity Large Hadron Collider (HL-LHC).

Layerwise: The Layerwise solution avoids this quadratic complexity pitfall by scoring only pairs of hits which fall on pairs of detector layers that are considered important. Which pairs of layers are important is determined through trial and error, though the majority of important layer pairs are those that are adjacent.

The Layerwise solution thus scores a small (subquadratic) fraction of hit pairs by again assessing the probability that each pair's hits belong to the same track. This scoring is performed using logistic regression on several derived features, including those mentioned earlier as well as noisy directional information extracted from the hit's cell features.

Next, candidate pairs are extended to triplets, again using average linkage to find for each pair the closest hit. Unlike the All Pairs solution, overlap is allowed at this point, so one hit may belong to many triplets.

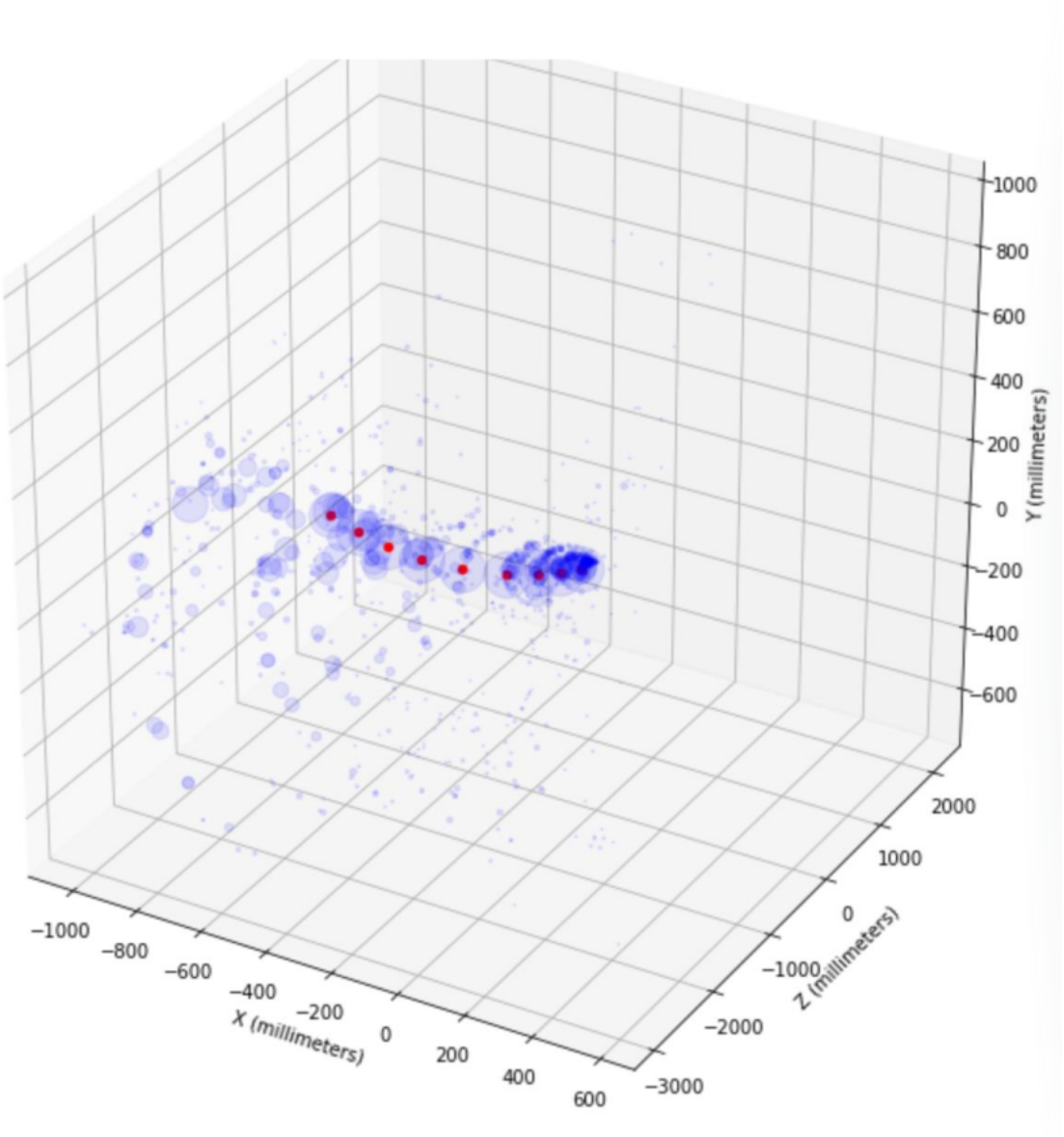

Figure 1-4: Probabilities, represented as blue dots, that each available hit belongs to the current track seed. [2]

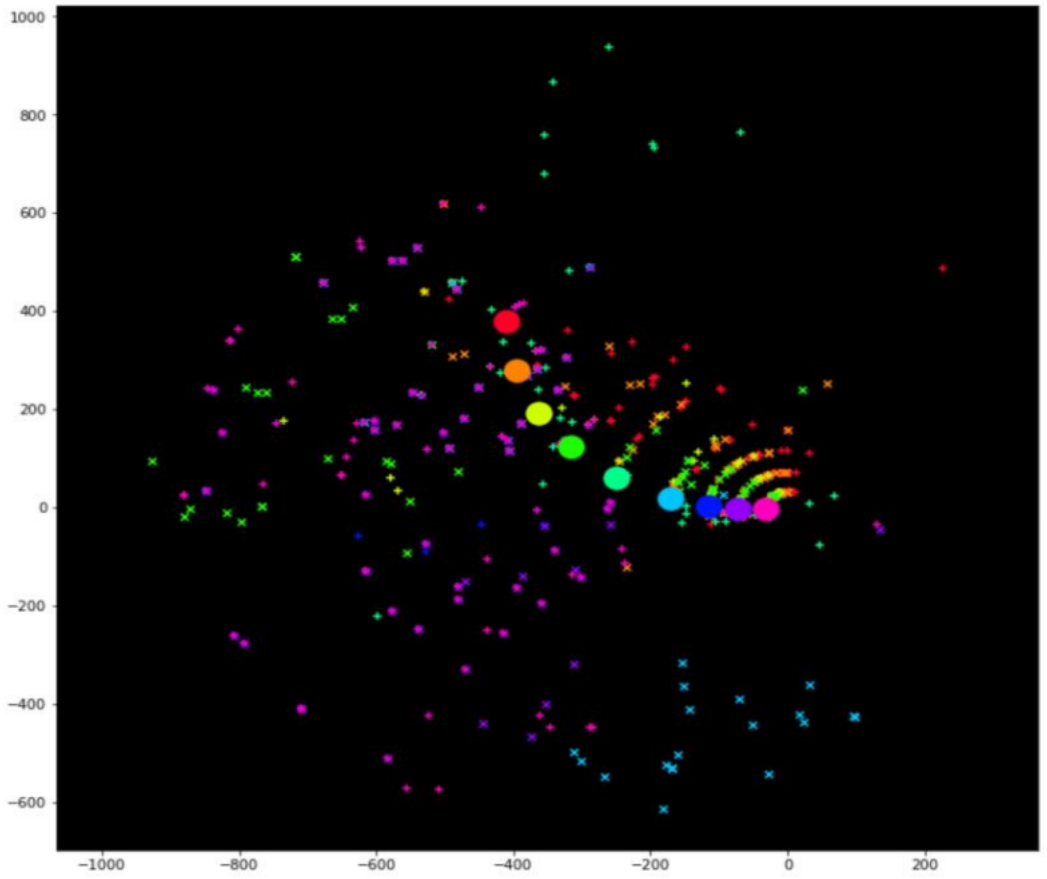

Figure 1-5: A representation of the pairwise solution growing a track with leftover hits. [1]

The triplets are placed into a priority queue indexed by their average linkage distance, and grown into tracks. Once a track has been constructed, all the priority queue is flushed of any triplets which contain hits in the new track. Thus all resulting tracks are without overlap.

Finally, tracks are again extended with leftover hits that have low average linkage distance, and by combining nearby tracks which have similar helix patterns. Figure 1-5 gives a visual representation of this reconstruction process.

This method achieves a 92% TrackML score, and does so efficiently, taking approximately 30 seconds per event using an Intel Core-i5 7th-generation desktop CPU.

1.2 Graph Neural Networks

As graph- and manifold-structured data has become more prevalent with the ubiquity of social, sensor, and biological networks, a new field which unites the deep learning methods operating on such data has emerged, called *Geometric Deep Learning* [5]. Motivated by the remarkable success of deep learning methods in computer vision, geometric deep learning aims to operate on any such data as well as point clouds, 3D shapes, and molecules, leveraging the non-Euclidean interactions and relations between objects, or entities [4]. These methods have been successfully applied to a wide range of tasks within particle physics, quantum chemistry, vision, and computer graphics, to name just a few.

At the center of geometric deep learning are graph neural networks, which have become a staple within deep learning in recent years, and are largely responsible for the many successes on tasks involving non-Euclidean datasets. Graph neural networks were first formulated close to their current form in [25], proposed as a learned method for diffusing information across a graph. While a promising first direction, this method relied on reaching a steady state convergence, the downside of which is a time consuming process with no guarantee of convergence.

Bruna et al. [6] developed spectral graph neural networks, which operate on learned filters in the spectral domain, defined by the eigenvectors of the graph Laplacian. This approach takes direct inspiration from convolutional neural networks, performing Fourier decomposition of graph-structured signals, with filtering performed as multiplication between the Fourier coefficients of the input signal and the learned spectral filter parameters. A major drawback of this method is that, unlike Euclidean-structured data where the fast Fourier transform is available, computing the forward and inverse Fourier transform requires $O(n^2)$ time complexity. Additionally, with no guarantee of spatial localization of the filters, spectral graph neural networks are sensitive to changes in the domain.

Message-passing neural networks (MPNNs) are, by comparison, a simple method which has gained considerable popularity as an efficient and effective method for graph-structured datasets. A wide variety of MPNNs have been proposed [13], all of which follow a similar overarching strategy made up of layers of graph convolution, where graph convolution consists of a message passing function and an update function. The message passing function passes information along the graph, from one

vertex's hidden state to another, aggregating information received typically with a sum or mean operation. Then, the update function concatenates each vertex's hidden state with the information sent to the vertex, to transform the features to a new hidden state via a learned set of weights. After several layers of graph convolution, the vertex features enter the so-called *readout* phase, where the features are combined - again through a sum or mean operation - to form a single feature vector, the output of the MPNN.

1.3 Related Work

1.3.1 Clustering

Clustering strategies generally fall into two broad categories: those which are partitional, or *flat*, and those which are hierarchical. The most popular flat clustering scheme is k-means [20], which is an iterative algorithm that alternates between assigning points to the nearest cluster centroid, and updating the cluster centroid based upon the points assigned to it. k-means requires *apriori* knowledge of the number of clusters, and aims to minimize the average distance between data points and their respective cluster centroids.

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [10] is likewise a flat clustering strategy, but is capable of detecting an arbitrary number of arbitrarily shaped clusters. Like k-means, it is an iterative method wherein a data point is seeded which begins a cluster, and nearby points which are not yet assigned to a cluster are added. This process repeats until there are no additional points nearby to the cluster. If the cluster is larger than a pre-specified minimum cluster size, the cluster is output; otherwise the points are not clustered. While DBSCAN is related to spectral clustering [26], it is far more efficient when compared to the $O(n^3)$ runtime of spectral clustering, and does not require choosing the number of eigenvectors for the spectral embedding, nor the number of clusters to produce from this embedding with k-means. DBSCAN thus has both a fast runtime and does not require prior knowledge of the number of clusters, it serves as a baseline for our experiments with TrackML.

The two primary approaches taken in hierarchical clustering are agglomerative, or bottom-up, strategies [14] [18], and divisive, or top-down, strategies [16]. Agglomerative clustering begins with every data point in its own cluster, and at each iteration combines clusters which are nearby. This process is repeated until every point belongs to the same cluster, at which point the algorithm outputs a multiscale clustering in the form of a binary tree. While fast methods for agglomerative clustering have been proposed [30], they typically rely on strong assumptions for scoring pairwise cluster distance, and are sensitive to noise. Without such assumptions, agglomerative clustering runs with $O(n^2)$ time complexity, thus is prohibitively expensive for use with TrackML.

Divisive hierarchical clustering begins with every data point assigned to same cluster, and at each step divides the current clusters into two subsets. This process

repeats until every data point lies within its own cluster, though the process may be stopped prematurely if so desired. Thus with a divide procedure which runs in $O(n)$ time produces reasonable balanced splits - enough such that the hierarchical tree has $O(\log n)$ depth, this strategy has $O(n \log n)$ time complexity. Because hierarchical clustering does not require *a priori* knowledge of the number of clusters, and because divisive clustering is able to run in $O(n \log n)$ time with light assumptions, this is the strategy we adopt for our learned clustering module.

1.3.2 Learned Clustering

Many recent deep learning approaches to clustering focus on finding good representations for the data points which are then able to be clustered using k-means [32] [31] [8]. However, as Shaham et al. point out in [27], the use of k-means criterion introduce an implicit bias toward solutions which favor convex-shaped clusters, limiting their applications to datasets similar to e.g. benchmark datasets such as Reuters and MNIST. Other methods for learned clustering take a spectral approach. Methods by Tian et al. [29] and Mishne et al. [21] learn to map the graph Laplacian onto a corresponding spectral embedding, then use k-means to form the final clusters. These approaches are successful on standard benchmark datasets, but their reliance on prior knowledge of the number of clusters prohibits their application to TrackML.

Yang et al. [33] employ a recurrent, agglomerative method for hierarchical clustering which, at each step, learns to merge two clusters. While their implementation relies on *a priori* knowledge of the number of clusters, it could be extended with a method similar to our `Stop` module, which removes clusters that should not continue to merge. The hard limitation of this work, and indeed agglomerative clustering strategies, is the need to iteratively combine clusters, a process which is simply too time consuming for TrackML.

Advancements on graph neural networks have led to new developments in graph coarsening strategies, analogous to pooling layers in convolutional neural networks. Unlike pooling in convolutional networks, recent work in graph pooling aims to adaptively coarsen graphs, aimed to handle different graph resolutions and preserve information within local neighborhoods. DiffPool [34] achieves this by assigning n_t points to n_{t+1} pooled units by means of a dense assignment matrix which is encouraged to be sparse. Cangea et al. [7] score the nodes after a graph convolution layer, selecting only the top k to continue to the next layer, where k is a hyperparameter. Gao and Ji use a simiilar formulation in their work on Graph U-Nets [12]. While these efforts have achieved performance improvements in graph classification tasks, they are unable to handle clustering tasks such as TrackML since all rely on setting the number of pooling units as hyperparameters, and in the case of DiffPool, quadratic complexity under modest assumptions.

Chapter 2

Clustering Graph Neural Network

2.1 Problem Setup

Our work focuses on a general clustering task wherein we are given an set of entities and predict which entities belong together within a cluster. Specifically, we are given, for each sample i, a set of input entities

$$X^{(i)} = \{x_1^{(i)}, \ldots, x_{n_i}^{(i)}\}, x_j^{(i)} \in \mathcal{X}. \tag{2.1}$$

Each sample i also consists of a varible number of clusters

$$C^{(i)} = \{c_1^{(i)}, \ldots, c_{k_i}^{(i)}\}, \tag{2.2}$$

and each cluster $c_j^{(i)}$ is a partition of $\{1, \ldots, n_i\}$.

A major challenge of this setup is that the variable number of clusters k for event i is not known at inference time. This is presents two difficulties. First, it rules out many of the standard clustering algorithms such as k-means, as well as previous clustering approaches using graph neural networks in which the number of clusters must be pre-specified [34]. Second, and more subtly, is that there may be a dependency between the number of clusters k and the number of input points n. Since this relationship could potentially be $k = \Omega(n)$ - this is the case for particle tracking - the runtime of our clustering graph neural network must not have any stage which is $O(kn)$. This is largely the constraint which motivates a hierarchical algorithm.

Since we only assume we are given a set of input entities and their ground truth clustering during training, we note that our model does not require knowledge of hierarchical structure. However, where present, ground truth information on a sample's hierarchical structure would greatly simplify training.

2.2 Model Requirements

We consider several important characteristics for our clustering graph neural network:

- **Speed:** A major challenge lies in handling the torrent of data produced by the

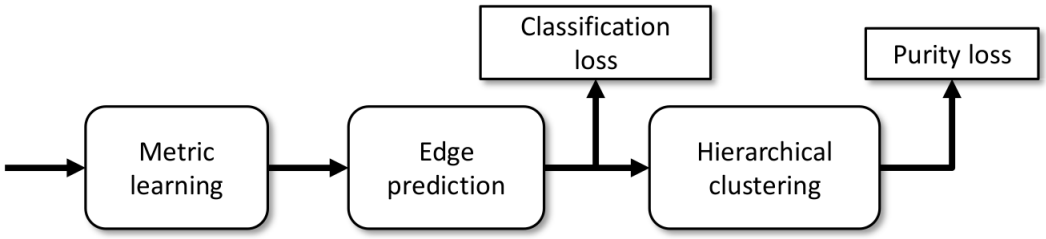

Figure 2-1: An overview of the clustering graph neural network stages.

High Luminosity Large Hadron Collider. Because of this, our model must be fast. Specifically, we require the model to operate, at train and test time, with:

1. $O(n \log n)$ asymptotic complexity
2. Good constant factors, allowing for high sample throughput, particularly at inference time
3. Good parallelization, with strong preference toward easy implementation on GPU hardware

- **End-to-end training:** We set the requirement for an end-to-end framework to better optimize each stage of our pipeline toward the prediction we ultimately care about. This means each stage (except for the last) must produce an output which is differentiable.
- **Inductive bias:** Due to the plethora of domain expertise present in particle physics tasks, we designed our architecture to handle the various features and representations accessible to us within the particle tracking challenge. The ability to exploit this domain expertise leads to smaller model classes for faster training and inference, and potentially superior generalization.

2.3 End-to-end Pipeline

Our clustering graph neural networks consists of three broad stages (figure 2-1), outlined as follows:

1. **Metric learning:** Given an input point cloud, embed the points into a new space which has a Euclidean distance metric. This metric will be used to build a sparse graph for use in the graph neural network.
2. **Graph neural network:** With the previous stage's graph, use a graph neural network model to consider higher-order interactions between points. The graph neural network will output a new representation of the points which is amenable to clustering.
3. **Hierarchical clustering:** With an improved representation for clustering, recursively divide the points until final clusters are reached, thus producing the final output.

2.4 Metric Learning

Since graph neural networks operate over a graph, but datasets such as TrackML consist only of points, we need a method for constructing a relational graph from entities. A simple choice used in recent work on graph neural networks is to use a dense graph, yet this won't be feasible in this case since TrackML's point clouds are significantly larger - about 3 orders of magnitude - than, say, the jet tagging point clouds in [9].

Another choice would be to use underlying domain knowledge of the TrackML detector, which is organized in layers, to find plausible pairs of hits. This is the approach taken in [1], where nearby layers are manually enumerated before extracting pairs based on rejection criteria fitting the underlying physics problem. Yet this is unsatisfying for two reasons. First, it sometimes happens that a particle passing through a layer is not registered in the detector, creating a *hole* in the track which will result in missing edges. It is not entirely impossible for successful reconstruction to occur, though at greater difficulty since the graph neural network will not provide communication to the two track halves. The second issue is that this method would not generalize to new datasets, since it relies on extensive domain knowledge and an understanding of the detector geometry.

Instead, we consider a more general approach where the objective is to find a good distance metric between pairs of points, wherein pairs belonging to the same cluster are nearby, and pairs belonging to different clusters are further apart. Assuming the cost to compute distance between a pair of points is $O(1)$, we can then construct a sparse graph efficiently by performing neighbor or neighborhood queries using e.g. kd-trees.

2.4.1 Graph quality:

The output of the metric learning stage will be a constructed graph $G = (V, E)$, where V is the graph's vertices, one for each point in the point cloud, E its edges. We consider $e_{i,j} \in E$ to be a *true* edge if, for points $x_i \in$ cluster $c_a, x_j \in$ cluster c_b, $c_a == c_b$. Otherwise we consider $e_{i,j}$ to be a *false* edge.

Since our graph neural network will be tasked with providing a good representation of the points for later use in the clustering stage, it is important that G be capable of allowing such good representations. Concretely, this requirement means the graph should contain as many true edges as possible, to allow for good communication among points belonging to the same cluster. Additionally, to keep computational costs low, the graph should contain as few false edges as possible.

Consider E_t and E_f, the true and false sets of edges within E. Graphs are evaluated, then, on two criteria:

- **Purity**

$$g_{purity} = \frac{|E_t|}{|E|}$$

- **Recall**

$$g_{recall} = \frac{|E_t|}{N_t}$$

where

$$N_t = \sum_{i=1}^{K} |c_i|^2. \tag{2.3}$$

That is, N_t is the total number of true edges discoverable for each sample.

2.4.2 Learning setup:

In constructing a dataset for our metric learning stage, we first extract pairs of points

$$X = \{(x_1^{(1)}, x_1^{(2)}), \ldots, (x_n^{(1)}, x_n^{(2)})\}, x_j^{(i)} \in \mathcal{X} \tag{2.4}$$

and their true labels

$$Y = \{y_1, \ldots, y_n\}, y_i \in \{0, 1\}, \tag{2.5}$$

where pairs of points are sampled uniformly from each of the samples in our dataset, and an equal number of positive and negative pairs are chosen to ease training.

2.4.3 Learned embeddings:

Rather than learn a distance metric directly, we instead embed our input points $x_i \in \mathcal{X}$ into a new Euclidean space $\mathbb{R}^d$, where d is low enough that the embedded space is not too sparse when considering all points within each sample. This formulation is an effort to leverage existing algorithms which can perform efficient queries using common distance metrics, something we will need for graph construction.

We embed points using a learned model ϕ, parameterized by θ, which maps points into the new Euclidean space

$$x \in \mathcal{X} \to \phi(\theta, x) \in \mathbb{R}^d. \tag{2.6}$$

In our experiments, ϕ is implemented as a multi-layer preceptron.

This stage is trained using a hinge embedding loss, pulling together points belonging to the same cluster and pushing aparts points which do not. So for a given sample

$$s_i = [(x_i^{(1)}, x_i^{(2)}), y_i], \tag{2.7}$$

we compute loss l as

$$l(s_i) = \max\left(0, 1 - y_i * \left\|\phi(\theta, x_i^{(1)}) - \phi(\theta, x_i^{(1)})\right\|_p\right). \tag{2.8}$$

For our implementation, we use $p = 2$. Figure 2-2 illustrates a mapping which is successful.

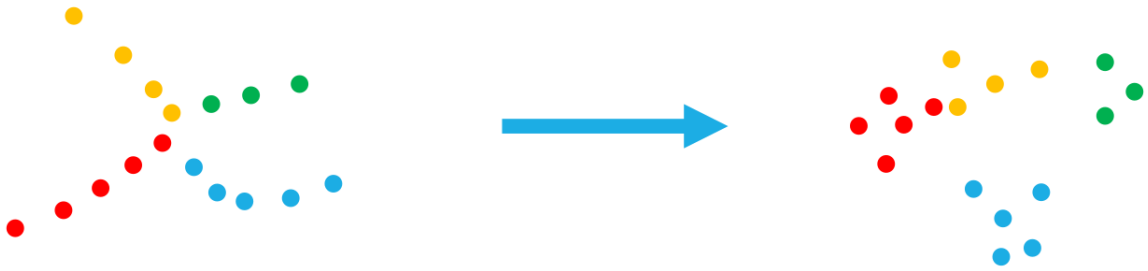

Figure 2-2: *Left:* Points in the original space corresponding to TrackML tracks. *Right:* Points mapped to the new space which has a Euclidean distance metric.

2.4.4 Initial graph construction:

Let

$$X' = [x'_1, \ldots, x'_n] = [\phi(x_1), \ldots, \phi(x_n)] \tag{2.9}$$

Then upon completion of the learned embedding's training, we now have a distance metric d where for points $x_i, x_j \in \mathcal{X}$,

$$d(x_i, x_j) = \left\| x'_i - x'_j \right\|_p . \tag{2.10}$$

With d, we can construct G by querying, for each point x_i, the set of points N_i which are nearby. Then, for every point

$$x_j \in N_i, \tag{2.11}$$

we add a directed edge $e_{i,j}$ which connects vertex v_i to vertex v_j.

For efficient querying, we construct a kd-tree from the embedded points. Once built, each point is queried using one of the following two strategies:

- **k-nearest neighbors**, which finds for each point x_i a neighborhood

$$N^i_{knn} = knnQuery(x_i, k), |N_i| = k \tag{2.12}$$

- **ϵ-ball query**, which finds for each point x_i a neighborhood

$$N^i_{ball} = ballQuery(x_k, \epsilon), \tag{2.13}$$

  where for each neighbor

$$x_j \in N^i_{ball}, d(x_i, x_j) < \epsilon. \tag{2.14}$$

While graphs produced with k nearest neighbor queries have the advantage of being regular, thus allowing for implementation speedups since they allow for grid-like data structures which GPUS favor, ϵ-ball queries tend to work better in practice

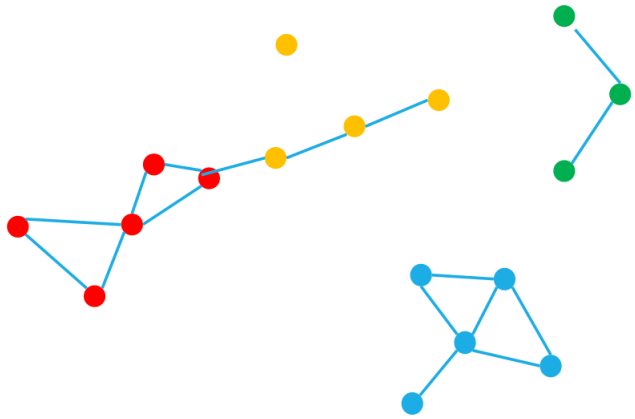

Figure 2-3: Graph formation from the embedded point cloud.

with respect to the aforementioned graph quality criteria. This is likely due to a non-uniform density surrounding each point, allowing ϵ-ball queries to better adapt to their surroundings. Figure 2-3 shows a toy visualization of the edge discovery process.

2.4.5 Edge refinement:

Although graphs produced using the learned embeddings are, in the case of TrackML, quite sparse, further refinement can yield in still much sparser graphs. Within the embedding model, we are only able to consider features derived from each point individually. Since we have now produced a relatively small set of edges, represented as pairs of points, we can now consider models which take as input pairs of points, as well as pairwise features derived from domain expertise.

We thus construct an edge refinement model ϕ', parameterized by θ', which operates on pairs of points x_i, x_j and their pairwise features $z_{i,j}$, and outputs the probability $p_{i,j}$ that the pair belongs to the same cluster.

$$p_{i,j} = \phi'(x_i, x_j, z_{i,j}) \in [0, 1] \tag{2.15}$$

ϕ' is likewise parameterized as a multi-layer preceptron.

With our trained model, we compute $p_{i,j}$ for each $e_{i,j} \in E$ produced during the embedding stage. Then, choosing a threshold hyperparameter $t \in [0, 1]$, we are left with our final edge selection

$$E' = \{e_{i,j} | p_{i,j} > t\}. \tag{2.16}$$

2.5 Graph neural network

The metric learning stage is capable of producing an embedding for each point which relies on information from that point alone. Although we can refine this metric by accounting for pairwise information, this is only feasible in special cases in which we only need to look at a small set of pairs if we want to keep sub-quadratic time complexity. Ultimately we will need a way of grouping points together to form our algorithm's final clustering, and as such it is desireable to start with the best representation possible.

Graph neural networks offer a way to improve over the metric learning stage's metric and pointwise representation by extracting information from neighborhoods rather than individual or pairs of points. This is accomplished through the use of rich filtering operations over irregular grids such as graphs, by means of information aggregation contained in each point's neighborhood to produce a superior representation. Additionally, because the graph we use is weighted and directed, the graph neural network is capable of learning a diverse set of views over each point's neighborhood, differing in both density and in reach.

2.5.1 Learning setup:

The input to our network is a sample, consisting of input entities

$$X = \{x_1'', \ldots, x_n''\}. \tag{2.17}$$

Each x_i'' is a combination of the original input features and the embedded features from the metric learning stage:

$$x_j'' = [x_j, x_j']\}, x_j \in \mathcal{X}, x_j' = \phi(x) \in \mathbb{R}^d \tag{2.18}$$

where $\phi(x)$ is the embedding model used in the metric learning stage, concatenated to the original features of each point. While the graph neural network is capable of learning this representation, providing it as input significantly reduces time during training.

We are also give a graph $G = (V, E)$ from the metric learning stages, where the input entities are vertices and edges are directed and given as input entity pairs from the metric learning stage:

$$E = \{e_1, \ldots, e_m\}, e_i \in (x_j, x_k). \tag{2.19}$$

As in the metric learning stage, we are given the true labels of the edges

$$Y = \{y_1, \ldots, y_m\}, y_i \in \{0, 1\}, \tag{2.20}$$

where the edge $e_i = (x_j, x_k)$ has true label $y_i = 1$ if x_j and x_k belong to the same cluster, and $y_i = 0$ if not. Since G is output using nearest-neighbor or ϵ-ball queries, we choose G to be sparse such that $|E| = m = O(n \log n)$. Our task is then to perform

edge classification on the learned embedding produced by the graph neural network, resulting in still better representation for presentation to the final clustering stage.

2.5.2 Edge weighting:

At each layer of the graph neural network, we update the geometric representation of each event by weighting the edges of the input graph G. This weighting can be thought of as a sparse $n \times n$ adjacency matrix $\boldsymbol{A}$, constructed through a learned kernel $d_{ij}^{(k)}$, belonging to layer k in the graph nerual network. Each kernel consists of a multi-layer preceptron $\Phi^{(k)}$ on pairs of input entities, normalized through a sigmoid operation to produce the edge weight $a_{ij}^{(k)}$:

$$d_{ij}^{(k)} = \Phi^{(k)}([v_i^{(k)}, v_j^{(k)}]) \in \mathbb{R},$$
$$a_{ij}^{(k)} = \frac{1}{1 + e^{-d_{ij}^{(k)}}} \in (0, 1),$$

where $v_i^{(k)}$ are vertex i's features at layer k. At the first layer, we set the vertex features

$$v_i^{(1)} = x_i''. \tag{2.21}$$

We compute a non-zero weight $a_{ij}^{(k)}$ for each discovered edge $e_{ij} \in E$.

While we have experimented with neighborhood normalization through the use of a softmax operation, we found it to hinder performance since information on the size of the neighborhood may be quite useful to a layer. This is in contrast to [9], where neighborhood normalization was absolutely crucial in order for the graph neural network to converge.

2.5.3 Graph convolution:

There are K layers of graph convolution within our graph neural network, and they are applied as follows. Layer k takes as input the d^k-dimensional feature vector v_i at each vertex, combined with all other vertices into an $n \times d^k$ matrix $\boldsymbol{V}^{(k)}$. Upon performing graph convolution, the layer outputs an $n \times d^{k+1}$ matrix $\boldsymbol{V}^{(k+1)}$, consisting of vertices, each represented by a d^{k+1} feature vector.

Graph convolution Ψ is specified by applying the adjacency matrix $\boldsymbol{A}$ to the input, then performing an affine transformation on the one-hop features concatenated with the input features:

$$\Psi(\boldsymbol{V}^{(k)}) = [\boldsymbol{A}^{(k)}\boldsymbol{V}^{(k)}, \boldsymbol{V}^{(k)}](\boldsymbol{w}^{(k)})^\intercal + b^{(k)}\mathbf{1}, \tag{2.22}$$

where $\boldsymbol{w}^{(k)}$ and $b^{(k)}$ are the $2d^{k+1}$ vector of weights and scalar bias, respectively.

Finally, the layer output $\boldsymbol{V}^{(k+1)}$ is produced by applying a non-linear function ρ (in our case, the recified linear unit) to the graph convolution output:

$$\boldsymbol{V}^{(k+1)} = \rho(\Psi(\boldsymbol{V}^{(k)})). \tag{2.23}$$

2.5.4 Evaluation:

Since the desired outcome of the graph neural network is an embedding of the points such that they may be easily clustered, the natural candidate to evaluate loss is to reuse the strategy outlined in the embedding metric learning stage.

After the last layer of the graph neural network, we are left with

$$\boldsymbol{V}^{(K)} \in \mathbb{R}^{n \times d^K}, \tag{2.24}$$

the matrix which holds feature vectors for each vertex. Since we have the true edge labels Y for all of the edges in our graph, we can define the loss as follows:

$$L(\boldsymbol{V}^{(K)}, Y) = \frac{1}{|E|} \sum_{(v_i, v_j) \in e_k \forall e_k \in E} l(v_i, v_j, y_k),$$
$$l(v_i, v_j, y_k) = \max\left(0, 1 - y_k * \left\|\phi(\theta, v_i) - \phi(\theta, v_j)\right\|_p\right),$$

where l is the hinge loss. As in the metric learning stage, we use $p = 2$ for the Euclidean norm.

2.5.5 Input and evaluation graphs:

Although we aim to provide G to the network such that G is sparse, containing as many true edges and as few false edges as possible, we may wish to evaluate our network embedding using a different set of edges. One reason to do this is for the purpose of including more true edges, which the case of TrackML, only requires adding only a constant factor of additional edges before every true edge is evaluated. We may also wish to include a larger neighborhood search to ensure the graph neural network has a more balanced distribution of true and false edges for evaluation. Since we already have an inexpensive method for building graphs developed in the metric learning stage, this extra cost is insignificant when compared with the boost in performance it offers.

2.6 Hierarchical clustering

Since the graph neural network also represents the input entities as points within a Euclidean space, a first consideration for the clustering stage could be a classic algorithm such as k-means. Unfortunately, in the context of TrackML, there are two significant problems with k-means. For one, we are not given k for each sample *a priori*. Thus for algorithms such as k-means or spectral clustering, a small error in an estimation of k may result in significant reconstruction error. Further, even with an oracle which provides the true k, because k has - for TrackML - a linear dependency on the number of entities in each sample n, these algorithms will run in quadratic time.

Traditional clustering algorithms exist which do not have these problems, and

one such candidate is DBSCAN. Since DBSCAN builds clusters based upon the ϵ-neighborhood of each point, it is unnecessary to know the number of clusters *a priori*. Additionally, as long as ϵ is chosen sufficiently small - and indeed the algorithm performs best when it is - the runtime of DBSCAN is sub-quadratic. However, because DBSCAN is not differentiable, we are precluded from training end-to-end as we had initially desired.

There are many important considerations when choosing a clustering strategy for the final stage in our pipeline, and we review them here:

- **Speed:** The clustering strategy must be fast, operating in $O(n \log n)$ time, with good constant factor scaling. It is critical that the clustering strategy respects this asymptotic runtime requirement both in cases where a cluster of size $\theta(n)$, and in cases where there are $\theta(n)$ clusters, each of size $O(1)$. An added bonus is that the method is easily parallelized, preferably for GPU acceleration.

- **Differentiable:** Because we aim to perform end-to-end training, the output of the clustering strategy - as measured by our loss criteria - must be differentiable with respect to the output of the previous, graph neural network stage.

- **Unspecified number of clusters:** We must be capable of handling datasets where, as in the case of TrackML, we are not given the number of clusters *a priori*.

- **Inductive bias on dataset:** While the clusters within TrackML are largely independent of each other, often times clusters resulting from data generating processes are done so in either a direct hierarchical fashion, or with some structure relating clusters to each other. Ideally, our framework should be capable of discovering such relational structure between clusters, leveraging this information in the pursuit of improved performance. Likewise, since there may be additional structure present in the ground truth labels such as a hierarchy, we prefer a clustering strategy whose evaluation criteria are easily modified to handle additional information.

- **Multiscale resolution:** A multiscale view of data can assist not only helping models to learn, but at inference time when interpretability becomes a more centralized focus.

These factors naturally lead to consider a learned, hierarchical clustering framework. And while there are formulations for fast agglomerative hierarchical clustering, the most natural framework for our requirements is that of a divisive, top-down approach. In this case, every point begins in the same cluster, and clusters are recursively split until each point is within its own cluster.

2.6.1 Learning setup:

The input to our clustering stage is a sample, consisting of input entities

$$H = \{h_1, \ldots, h_n\}, h_i = v^{(K)} \in \mathbb{R}^{d^K}. \tag{2.25}$$

That is, the input to the clustering stage is the set of embedded points as output from the previous stage's graph neural network.

The output of the clustering stage is a variable number of clusters

$$C' = \{c'_1, \ldots, c'_r\}, \tag{2.26}$$

where c'_i is a partition of $\{1, \ldots, n\}$.

Recall that the true clusters for an event are given as

$$C = \{c_1, \ldots, c_k\}. \tag{2.27}$$

Because we do not know k during inference, there is no requirement that the number of discovered clusters r is equal to the true number of clusters k, something we must account for in our evaluation strategy.

2.6.2 Clustering framework:

Our clustering framework consists of two modules, one which we call `Split`, and one which we call `Stop`. The algorithm runs as follows:

1. Assign all points H into one large cluster.
2. Run the `Split` module on the points within the cluster, outputting for each point the probability that the point is assigned to split to the right child cluster. A low probability for a given point h_i means that h_i will more likely be assigned to the left child cluster.
3. From the probabilities, assign each point to either the right or left child cluster.
4. For each child cluster, run the `Stop` module on all points contained within the cluster.
5. Aggregate the stop probabilities for points contained within each cluster.
6. Continue recursively on all clusters which have a low probability of stopping. Output all clusters with a high probability of stopping.

Figure 2-4 shows a toy visualization of the clustering processes, wherein different clusters are allowed to stop at different depths of recursion.

2.6.3 Split, Stop modules:

For both the `Split` and the `Stop` modules, we are given a set of input points

$$Z = \{z_1, \ldots, z_M\}, M = |C_i'^{(l)}|, \tag{2.28}$$

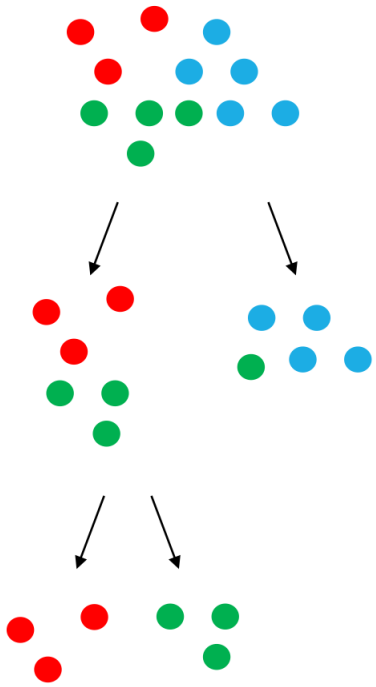

Figure 2-4: The hierarchical clustering stage allows for clusters to split to variable depths of recursion, necessating balanced splits to keep the algorithm's time complexity sub-quadratic.

where $C_i'^{(l)}$ is the ith cluster in recursion depth l. For each point, we compute the probability of taking a binary action to produce the output

$$P = \{p_1, \ldots, p_M\}, p_i \in [0, 1]. \tag{2.29}$$

In the `Split` case, $p_i = 1$ implies that point z_i will be assigned to the right cluster, and $p_i = 0$ implies the left. In the `Stop` case, $p_i = 1$ implies that the point z_i votes for the cluster to stop splitting.

Since the decision to stop splitting is made once per cluster, we aggregate the stop probabilities by computing the mean probability, though other aggregation tactics may be used such as the max. The probability of stopping cluster i at recursion depth l is then given as

$$P_i^{(l)} = \frac{1}{M} \sum_{j=1}^{M} p_j. \tag{2.30}$$

Because the modules are intended to operate over clusters which may be quite different in shape and size, it is necessary to share global information with each point when determining the probability assigned to the point. We take inspiration from the Split model used in [22], which similarly aims to divide a set of points into two exclusive sets.

Our `Split` and `Stop` modules are thus similar to a multi-layer preceptron with

T layers, but with the following modification. Before each layer t, we have a set of feature vectors for each point:

$$H^{(t)} = \{h_1^{(t)}, \ldots, h_M^{(t)}\}, h_j^{(t)} \in \mathbb{R}^{d_t}, \tag{2.31}$$

where $H^{(1)}$ is the set of input points Z, and for all other layers, $H^{(t)}$ is the output of the previous layer $t-1$. In order to share global information with each point, we perform aggregation over the feature vectors by computing the mean

$$\bar{h}^{(t)} = \frac{1}{M} \sum_{j=1}^{M} h_j^{(t)} \tag{2.32}$$

and the variance

$$\hat{h}^{(t)} = \frac{1}{M} \sum_{j=1}^{M} (h_j^{(t)} - \bar{h}^{(t)})^2. \tag{2.33}$$

To compute the output of the layer for each point, we concatenate the global information with the individual point's feature vector, then pass this new representation through a dense neural network layer:

$$h_j^{(t+1)} = \rho\left([h_j^{(t)}, \bar{h}^{(t)}, \hat{h}^{(t)}]^\intercal W^{(t)} + \beta^{(t)}\right), \tag{2.34}$$

where $W^{(t)}$ is a learned $3d_t \times d_{t+1}$ parameter matrix, $\beta^{(t)}$ is a learned d_{t+1} bias vector, and ρ is a non-linear function - in our case, the rectified linear unit.

To compute a probability for each point, first compute an output logit for each point:

$$o_j = (h_j^{(T)})^\intercal \boldsymbol{w} + b, \tag{2.35}$$

where $\boldsymbol{w}$ is a $d_T \times 1$ weight vector and b is a scalar bias. Then we apply the sigmoid function to produce a probability:

$$p_j = \frac{1}{1 + e^{-o_j}}. \tag{2.36}$$

2.6.4 `Split` supervision:

The `Split` module takes as input the set of feature vectors associated to the entities within a recursion tree cluster, and outputs the probability that each entity is assigned to the left or right child subcluster. If one has access to a dataset consisting not only of the ground truth clusters, but also to the underlying hierarchical structure which defines how to split the entities within each parent cluster, training the `Split` module simply becomes a binary classification task. However, in many scenarios, including that of TrackML, we have access only to the final clusters, and as such must use a more sophisticated training strategy.

Keeping with the desiderata outlined earlier, the output of the `Split` module should have the following characteristics:

1. **Separability:** The purpose of the `Split` module is to divide the input points along the ground truth cluster boundaries. Doing so recursively will eventually lead us to recover the true clusters, so we must ensure our training strategy selects for this.

2. **Balance:** Since the `Split` module affects the depth of recursion reached, it has a major influence on the runtime of our clustering framework. Unbalanced splits will quickly lead to quadtratic complexity, or large constant factors in less severe cases. Therefore, we would also like a method for penalizing unbalanced splits during training.

Because we ultimately assign a binary action to each point, sending the point either left or right, a first possibility is to score the module's performance based upon the final clustering and runtime, as defined by a reward function which we implement. We can use REINFORCE [28] to train the `Split` module until the desired performance characteristics are met. Unfortunately, while this approach works well for small sample sizes, our experiments resulted in poor convergence as the number of points in the sample becomes large.

Thus we have developed a supervision strategy which works on principles of entropy. At a high level, we seek to minimize the entropy of points belonging to the same ground truth cluster, and to maximize the entropy of points belonging to different ground truth clusters. Computing these entropy losses is done in the following manner.

We again consider a cluster $C_i'^{(l)}$, the ith cluster at recursion depth l. The set of points assigned to $C_i'^{(l)}$ are

$$Z = \{z_1, \ldots, z_M\}, M = |C_i'^{(l)}|, \tag{2.37}$$

and each point has a probability computed as

$$P = \{p_1, \ldots, p_M\}, \tag{2.38}$$

where p_j is the probability that point j is assigned to the right child subcluster. We are also given access to the ground truth cluster information for the points, which we consider as

$$\tilde{C} = \{\tilde{c}_1, \ldots, \tilde{c}_N\} \tag{2.39}$$

and each cluster $\tilde{c}_j$ is a partition of $\{1, \ldots, M\}$. As an intermediate step, we must also compute the average probability for each ground truth cluster:

$$\tilde{p}_j = \frac{1}{|\tilde{c}_j|} \sum_{k \in \tilde{c}_j} p_k, \tag{2.40}$$

and we will also need the mean probability across all ground truth clusters:

$$\bar{P} = \frac{1}{N} \sum_{j=1}^{N} \tilde{p}_j. \tag{2.41}$$

The first entropy term seeks to ensure points belonging to the same ground truth cluster are assigned to the same child subcluster. We compute one entropy term per cluster as follows:

$$s_j = -p_j \log p_j - (1 - p_j) \log p_j, \tag{2.42}$$

where s_j is the within-cluster entropy for ground truth cluster j. The total within-cluster entropy is taken as the average within-cluster entropy across all ground truth clusters:

$$S_{within} = \frac{1}{N} \sum_{j=1}^{N} s_j. \tag{2.43}$$

Minimizing S_{within} on its own will almost certainly result in the trivial behavior of all points being assigned to the same child subcluster. Thus our second entropy term aims to maximize entropy between ground truth clusters, and compute it as follows:

$$S_{between} = -\bar{P} \log \bar{P} - (1 - \bar{P}) \log \bar{P}. \tag{2.44}$$

$S_{between}$ is maximized when $\bar{P} = 0.5$. If S_{within} is minized, this means exactly half of the ground truth clusters will be assigned to each child subcluster, and all points within each ground truth cluster will stay together. We note that our formulation is invariant to the size of the ground truth clusters, placing an equal importance on each - something desirable for achieving the best TrackML performance.

Finally, we compute loss as the difference between entropy terms:

$$l(C_i^{(l)}) = S_{between} - S_{within}. \tag{2.45}$$

This loss formulation allows for fully supervised training, and because it is differentiable with respect to the output from the graph neural network stage, we are also able to train in an end-to-end fashion.

2.6.5 `Stop` supervision:

The `Stop` module takes as input the set of feature vectors associated to the entities within a recursion tree cluster c_i,

$$Z = \{z_1, \ldots, z_M\}, M = |c_i|, \tag{2.46}$$

and outputs the probability that the cluster has finished dividing

$$p_i = \texttt{Stop}(c_i). \tag{2.47}$$

Once the cluster has stopped dividing, a new cluster label is created, to which all entities within the cluster are assigned. Then the labeled entities are returned by the algorithm.

A major challenge in developing a training strategy for the `Stop` module lies in the fact that its input depends on the performance of the `Split` module. Should the `Split` module perform perfectly, training the `Stop` module would be a simple

task, wherein it learns to stop splitting once a ground truth cluster has been reached. However, since the `Split` module will produce some amount of error, determining when to stop splitting depends on how the final clustering is scored. Our training strategy should thus seek to stop splitting when the clustering score is maximized.

Because the output of the `Stop` module is the probability of taking a binary action, a natural first candidate for training is the REINFORCE algorithm, using the non-differentiable clustering score as reward. However, similar to the `Split` module, convergence only reliably occurs for small sample sizes with few clusters; as the number of clusters approaches even a small fraction of those in TrackML samples, convergence is poor and simple heuristics (e.g., stop splitting once cluster size is below a certain threshold) typically provide better performance.

We instead have developed a method for fully supervised training, which rests on our ability to score each cluster independently of the remaining clusters. Given a set of recursion tree clusters c_p, c_l, and c_r, where c_l and c_r are the left and right subcluster of parent cluster c_p, respectively, we assume that each cluster has an associated score

$$s_i = \texttt{scoreCluster}(c_i) \in (0, 1]. \tag{2.48}$$

Because the child subclusters may be unbalanced, we also assume each cluster c_i has a weight w_i, and that

$$w_p = w_l + w_r. \tag{2.49}$$

We can now assign a recursive score to each cluster, which is the highest score achievable by the cluster itself, or of its weighted descendants:

$$s'_p = \max(s_p w_p, s'_l w_l + s'_r w_r). \tag{2.50}$$

Ground truth labels y_i are then generated on-the-fly for each cluster as follows:

$$y_i = \begin{cases} 1 & \text{if } s_p w_p \geq s'_l w_l + s'_r w_r \\ 0 & \text{else} \end{cases} \tag{2.51}$$

That is, splitting ends when no further splitting leads to a higher weighted score among the parent cluster's descendants.

To score a cluster c, we begin by determining the ground truth cluster c_{gt} which has the highest number of entities in c. We then compute several quantities:

- $N_c = |c|$, the number of entities in the cluster c,
- $N_{gt} = |c_{gt}|$, the number of entities in the ground truth cluster c_{gt},
- N_t, the number of entities belonging to both c and c_{gt}.

With this information, we compute a precision and recall score for c:

$$s_{precision} = \frac{N_t}{N_c}$$
$$s_{recall} = \frac{N_t}{N_{gt}}$$

Finally, the score s for cluster c is computed as

$$s = s_{precision} \times s_{recall}. \tag{2.52}$$

Note that this score is never zero, and equals its maximum value of one exactly when the ground truth cluster is recovered. In our experiments, because TrackML requires clusters of at least size three, we assign a score of zero to any clusters containing fewer than three points.

A consequence of computing ground-truth labels on-the-fly with recursion is that simply because a cluster achieves a higher score than its immediate descendants does not preclude further descendents from achieving a higher yet aggregate score. It is therefore necessary to consider, during training, the entire recursion tree to ensure our scoring is accurate. Assuming the `Split` module produces reasonably balanced subclusters, there is no additional asymptotic cost occurred, though the increased constant factor can be significant.

Chapter 3

Experimentation

Our target application is the TrackML challenge, and as such the experiments we run to demonstrate the clustering graph neural network test various aspects of the dataset. Because previous deep learning approaches are not capable of operating on TrackML data, due either to quadratic time complexity, sub-quadratic time with unfeasibly large constant factors, or to an *a priori* requirement to specify the number of clusters, we do not compare our architecture to previous work within geometric deep learning. Rather, we compare our model with the best performing TrackML entries, and give as points of reference baseline, traditional clustering methods.

3.1 Training

All stages of our pipeline are trained using PyTorch and Nvidia P40 GPUs, and optimized with Adam [17]. Each stage is first trained individually, before training jointly in an end-to-end fashion, providing for marginally better final performance over pure end-to-end training. Learning rates are specified for each section, but all stages use a ten-fold reduce-on-plateau learning rate decay, invoked when validation loss does not improve for ten epochs.

3.1.1 Metric Learning

The metric learning stage as described in section 2.4 uses, for the embedding stage, a three-layer MLP, where the first two layers contain 256 hidden units, and the final layer embeds into each point into $\mathbb{R}^4$, and is trained with a starting learning rate of 0.001. The edge refinement stage consists of four layers, the first three of which contain 2048 hidden units, and the last of which embeds to one hidden unit for edge prediction. The starting learning rate is likewise 0.001.

10 TrackML events are mined to form the training dataset used in both the embedding module and the edge refinement module. All true hit pairs are included from these events, 10 million total, and 10 million false pairs are randomly selected to form a balanced dataset. Once the embedding model has begun to converge, 10 more events are mined in a similar fashion, but the false pairs are chosen as the ones

most difficult for the embedding module to correctly place. Training then continues for both the embedding and edge refinement with the combined 40 million samples.

3.1.2 Graph Neural Network

The graph neural network stage as described in section 2.5 consists of six layers of graph convolution, each with their own learned MLP kernel. The first five stages contain 192 hidden units, while the final stage embeds each hit to $\mathbb{R}^8$. Each kernel is a two-layer MLP, where the first layer has 32 hidden units and the final layer outputs one value as input to a sigmoid activation. Training begins with a learning rate of 0.0005.

The graph neural network is trained using 4000 TrackML events, where each event is randomly sub-sampled to contain only 10% of the original clusters. This sub-sampling is necessary to handle the large event sizes of TrackML, and will in future work be replaced with graph sampling techniques. Since the graph neural network and the hierarchical clustering stage are ultimately trained end-to-end, this dataset is also applied when training the following stage.

PyTorch is at-present limited in handling minibatches and backpropagation with the use of sparse adjacency matrices, and as such, we implement the graph neural network stage using Deep Graph Library (DGL) [3] with PyTorch as backend.

3.1.3 Hierarchical Clustering

The hierarchical clustering network contains a `Split` module and a `Stop` module, each of which is a modified MLP as described in section 2.6. Both modules are identical in network architecture, each consisting of four layers with 256 hidden units each, before performing a binary prediction as output. The initial learning rate for the `Split` module is 0.01 and for the `Stop` module, 0.001.

A major challenge in this form of hierarchical training lies in parallelization. Because the number of clusters for TrackML is $O(n)$, and because divisive clustering requires to start with one cluster of size $O(n)$, few assumptions on the distribution of points can be exploited. Additionally, some clusters may be ordered to stop splitting before others, as illustrated in figure 2-4.

Although this stage does not operate over graph-structured data, we again use DGL for training the hierarchical clustering stage. DGL's abaility to batch non-uniform sets of entities, combined with its ability to aggregate information over such sets, allow for a simple relatively implementation, where the different aggregation strategies are tracked using permutations. Figure 3-1 shows a typical grouping of clusters as represented during recursive training, where color labels the true clusters, and the black vertical line divides the points into two higher-level recursive tree clusters. Figure 3-2 shows an aggregation strategy for computing measures such as the entropy loss for the `Split` model.

Figure 3-1: A set of points after one application of `Split` module. Node color represents true clusters, while the vertical black bar divides the current recursive tree clusters.

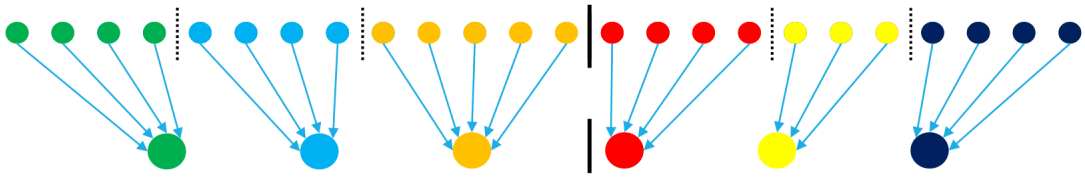

Figure 3-2: Permuting the set of points by their true clusters allows the DGL framework to aggregate information for loss terms, handling in parallel the variable-sized groups.

3.2 Results

As mentioned in section 2.4, the two metrics used for scoring graph quality are that of purity and recall, where purity measures the proportion of true edges among the edges discovered, and recall measures the proportion of true edges discovered of all true edges available. Because the baseline methods rely upon an ability to score pairs of points, we demonstrate the effectiveness of our metric learning stage here. Table 3.1 shows a comparison of the different methods' ability to discover true edges, and the number of edges required to evaluate in order to produce these edges.

Notable in this comparison is the number of pairs required for the All Pairs baseline, which produces a dense scoring matrix for later use during track generation. The Pairwise baseline requires computing over far fewer pairs of points - only 4% f the dense All Pairs method - but still falls short in terms of recall at only 24%, something the Pairwise method must make up for with superior track generation. Our embed module is capable of a three-fold increase in recall over the Pairwise method, with a two-fold increase in purity. Passing these recovered pairs through our pairwise module results in a small drop in recall, with a five-fold increase in recall, allowing for extremely sparse graphs as input to the graph neural network, while retaining most of the true pairs of points.

Table 3.2 displays the performance of our clustering graph neural network, com-

Method	**# Edges (million)**	**Purity**	**Recall**
Baseline, All Pairs	10000	0.001	1.00
Baseline, Pairwise	400	0.02	0.24
Ours, Embed Module	200	0.04	0.81
Ours, Pairwise Module	35	.21	0.78

Table 3.1: Comparison of the various methods and their ability to discover true edges. The purity and recall scores used here are defined in section 2.4.

Method	**TrackML Score**
Baseline, All Pairs	0.90
Baseline, Pairwise	0.92
Baseline, Pairwise (10% tracks)	0.93
DBSCAN	0.20
DBSCAN (10% tracks)	0.26
Clustering GNN (10% tracks)	0.91

Table 3.2: Results of several methods for TrackML track reconstruction, with TrackML's scoring function

pared with the TrackML baselines, and DBSCAN as a point of reference. We report the performance of our method as evaluated using 10% of each sample's tracks, due to current performance limitations of graph neural networks on GPUs. While the baseline methods are capable of outperforming the clustering graph neural network, we emphasize that its performance is still high for the TrackML challenge, and note that its strength lies in its flexibility to adapt to many different regimes, something which the highly-tuned TrackML baselines are unable to do.

Chapter 4

Future Directions

4.1 Graph Sampling

One of the primary limitations of our architecture lies in the graph neural network's $O(nmp)$ memory complexity, where n is the number of entities to be clustered, m is average number of neighbors between entities, and p is the number of hidden units in a given layer. Since changing the size of the neighborhood can have severe consequences on the communication between entities, and assuming we do not wish to use a prohibitively small network, the only reasonable method for reducing this memory cost is to reduce the number of entities n.

In practice, this amounts to implementation of a graph sampling strategy, which would aim to reduce the overall graph size while retaining the local neighborhood structure of a node or set of nodes in question. There are many basic strategies to produce such a sampling [19], and several recent advances in graph sampling offer still more advanced methods for creating unbiased samples [15] [23].

4.2 Enhanced Parallelism

In tandem with graph sampling are enhanced parallelization methods by means of distributed computing. While GPU architectectures are remarkably efficient when it comes to dense matri multiplication, they are typically on par or slower than CPUs when it comes to the sparse message passing operations central to our method. Thus a move to CPU operations would allow for far greater memory allocations with little cost in efficiency, while distributed computing would result in greater training and inference speeds. These efficiency gains would be further enhanced with an implementation of graph sampling.

4.3 Iterative Clustering

Finally, a weakness of our higherchical clustering stage is its inablility to recover once a mistake has been made by the `Split` module. Because each data point is assigned a

probability by the `Split` module, it is possible to score each of the final clusters based upon the aggregate and intermediate probabilities assigned throughout inference. It may thus be beneficial to output clusters with only the highest aggregate scores, re-running data points belonging to clusters with lower aggregate scores, in an iterative fashion until some cutoff has been met. This type of strategy results in a yet wider class of models, one which has the representational power to reproduce the TrackML baseline methods.

Chapter 5

Conclusions

This thesis has introduced the clustering graph neural network, a novel and fast architecture for data-driven hierarchical clustering with end-to-end training. Compared with previous deep learning efforts to cluster on graph and point-cloud data, our method is capable both of operating in sub-quadratic time, and in producing a variable number of clusters which does not need to be pre-specified. We likewise emphasize the ability of our architecture to be heavily parallelized, handling datasets for which each sample is two or more orders of magnitude than those presented in previous work on clustering with graph neural networks.

To demonstrate the clustering graph neural network, we have evaluated its performance on the TrackML Challenge, a particle tracking task which requires efficient and accuracte methods to handle the torrent of data produced by the forthcoming HL-LHC. We have noted the key difficulties within this domain, including a number of clusters which is proportional to the number of data points, and lack of *a priori* knowledge of the number of clusters. We have presented encouraging, if not state-of-the-art, performance which motivates future research in extending our architecture to incorporate graph sampling and model parallelism.

Bibliography

[1] Trackml, 1st place solution. Kaggle Documentation, 2018.

[2] Trackml, 2st place solution. Kaggle Documentation, 2018.

[3] Deep graph library. `https://github.com/dmlc/dgl`, 2019.

[4] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018.

[5] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *CoRR*, abs/1611.08097, 2016.

[6] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. Spectral networks and locally connected networks on graphs. 12 2013.

[7] Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Lio. Towards sparse hierarchical graph classifiers, 11 2018.

[8] Gang Chen. Deep learning with nonparametric clustering. *CoRR*, abs/1501.03084, 2015.

[9] Nicholas Choma, Federico Monti, Lisa Gerhardt, Tomasz Palczewski, Zahra Ronaghi, Mr Prabhat, Wahid Bhimji, Michael Bronstein, Spencer R. Klein, and Joan Bruna. Graph neural networks for icecube signal classification, 09 2018.

[10] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[11] Steven Farrell, Paolo Calafiura, Mayur Mudigonda, Prabhat, Dustin Anderson, Jean-Roch Vlimant, Stephan Zheng, Josh Bendavid, Maria Spiropulu, Giuseppe Cerati, Lindsey Gray, Jim Kowalkowski, Panagiotis Spentzouris, and Aristeidis Tsaris. Novel deep learning methods for track reconstruction, 2018.

[12] Hongyang Gao and Shuiwang Ji. Graph u-net, 2019.

[13] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017.

[14] K. Chidananda Gowda and G. Krishna. Agglomerative clustering using the concept of mutual nearest neighbourhood. *Pattern Recognition*, 10(2):105–112, 1978.

[15] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

[16] Leonard Kaufman and Peter Rousseeuw. *Finding Groups in Data: An Introduction To Cluster Analysis*. 01 1990.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[18] Takio Kurita. An efficient agglomerative clustering algorithm using a heap. *Pattern Recognition*, 24(3):205 – 209, 1991.

[19] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 631–636, New York, NY, USA, 2006. ACM.

[20] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[21] Gal Mishne, Uri Shaham, Alexander Cloninger, and Israel Cohen. Diffusion nets. *CoRR*, abs/1506.07840, 2015.

[22] Alex Nowak and Joan Bruna. Divide and conquer with neural networks. *CoRR*, abs/1611.02401, 2016.

[23] Jihun Oh, Kyunghyun Cho, and Joan Bruna. Advancing graphsage with a data-driven node sampling, 2019.

[24] David Rousseau. Participant document: Particle tracking and the trackml challenge, 2018.

[25] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 2009.

[26] Erich Schubert, Sibylle Hess, and Katharina Morik. The relationship of dbscan to matrix factorization and spectral clustering, 2018.

[27] Uri Shaham, Kelly Stanton, Henry Li, Ronen Basri, Boaz Nadler, and Yuval Kluger. Spectralnet: Spectral clustering using deep neural networks. In *International Conference on Learning Representations*, 2018.

[28] Richard S. Sutton and Andrew G. Barto. Reinforcement learning i: Introduction, 1998.

[29] Fei Tian, Bin Gao, Qing Cui, Enhong Chen, and Tie-Yan Liu. Learning deep representations for graph clustering. In *AAAI*, 2014.

[30] Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. Fast agglomerative clustering for rendering, 2008.

[31] Junyuan Xie, Ross B. Girshick, and Ali Farhadi. Unsupervised deep embedding for clustering analysis. *CoRR*, abs/1511.06335, 2015.

[32] Bo Yang, Xiao Fu, Nicholas D. Sidiropoulos, and Mingyi Hong. Towards k-means-friendly spaces: Simultaneous deep learning and clustering. *CoRR*, abs/1610.04794, 2016.

[33] Jianwei Yang, Devi Parikh, and Dhruv Batra. Joint unsupervised learning of deep representations and image clusters. *CoRR*, abs/1604.03628, 2016.

[34] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *CoRR*, abs/1806.08804, 2018.