Graph Neural Networks for charged particle tracking

Undergraduate Thesis

Submitted in partial fulfillment of the requirements of
BITS F422T Thesis

By

Aryaman Jeendgar
ID No. 2019B5AA0767H

Under the supervision of:

Dr. Kilian Lieret
&
Dr. Pratyush Chakraborty

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, HYDERABAD CAMPUS
May 2024

Declaration of Authorship

I, Aryaman Jeendgar, declare that this Undergraduate Thesis titled, 'Graph Neural Networks for charged particle tracking' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 5th May, 2024

Certificate

This is to certify that the thesis entitled, "*Graph Neural Networks for charged particle tracking*" and submitted by Aryaman Jeendgar ID No. 2019B5AA0767H in partial fulfillment of the requirements of BITS F422T Thesis embodies the work done by him under my supervision.

Supervisor
Dr. Kilian Lieret
Research Software Engineer,
Princeton University
Date: May 5, 2024

Co-Supervisor
Dr. Pratyush Chakraborty
Professor,
BITS-Pilani Hyderabad Campus
Date:

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, HYDERABAD CAMPUS

Abstract

Bachelor of Engineering(Hons.) Electrical & Communications Engineering

Graph Neural Networks for charged particle tracking

by Aryaman JEENDGAR

There has been a surge of interest in the applicability of deep-learning techniques for track reconstruction (on the order of $\mathcal{O}(10^k)$ trajectories of particles from their recorded hits in the detector) because of their comparable performance to traditional algorithms in addition to demonstrating better computational scaling.

In this thesis we investigate several research threads around a learned-clustering-based pipeline for charged particle tracking, [12]. We first explore the feasibility of a pre-emptive noise-classification module as part of the pipeline, *i.e.* building a module that can filter out all of the noise tracker hits in the dataset *before* we process the tracker data in any way. We perform a feasibility study followed by an attempt at implementing the suggested scheme using FCNNs with residual connections and an appropriately weighted binary cross entropy loss to remain sensitive to the imbalanced nature of the noise-classification problem.

Further, we perform studies into leveraging a loss from contrastive learning for training the learned clustering. These results are gathered on a simplified subset of the entire dataset.

Acknowledgements

This thesis summarizes the work that I have done for my thesis with Princeton Research Computing at CERN (via IRIS-HEP). I am deeply grateful to my supervisor, Dr. Kilian Lieret for introducing me to this problem and for his invaluable input throughout my fellowship, as an advisor and mentor par excellence.

I would also like to thank Dr. Henry Schreiner because of whom I was able to pursue this excellent thesis opportunity and Dr. G.J. Peter Elmer for all of his help with the logistics of my fellowship.

Finally, I would like to thank my Co-Supervisor, Prof. Pratyush Chakraborty. From being an instructor beyond par in the courses I took under him, to introducing me to the exciting world of optimal control via the problem we worked on with the VRFB battery. At the same time, I'd like to thank the entire EEE department for at BITS Hyderabad for giving me a cutting-edge education in the electrical sciences!

Contents

List of Figures

Abbreviations/Notations

GNN	**G**raph **N**eural **N**etwork
GC	**G**raph **C**onstruction
OC	**O**bject **C**ondensation
EC	**E**dge **C**lassification
FCNN	**F**ully **C**onnected **N**eural **N**etwork
ROC	**R**eceiver **O**perator **C**haracteristics
FPR	**F**alse **P**ositive **R**ate
TPR	**T**rue **P**ositive **R**ate
CKF	**C**ombinatorial **K**alman **F**ilter
KF	**K**alman **F**ormalism
`-oi-`	of-interest

Dedicated to the Indomitable Human Spirit...

Chapter 1

An introduction to charged particle tracking

In this section, I give some general background for charged particle tracking and classical approaches to the same. The goal of track reconstruction is to recover the properties of charged particles from a set of measurements caused by their interaction with special kinds of sensitive detectors. It is in essence, a classification problem (with an arbitrary number of classes — equal to the number of particles that we choose to study plus noise, which is going to be the subject of our study for this report).

CMS and **ATLAS** are particle detectors at CERN. CMS and ATLAS are both independent collaborations within CERN and their development has historically progressed independently of each other.

Yet they both have some core similarities in their design which are relevant to understand certain features of the **TrackML** dataset [1] that we use throughout our work.

The `TrackML` challenge [1] was hosted as by CERN from May 2018 through July 2019 for crowd sourcing the exploration of efficient data-driven methods for charged particle tracking. To that end, the `TrackML` dataset was made public, which relies on a realistic detector model to simulate measured particle hits similar to what is expected for an HL-LHC experiment. The detector model is inspired by the ATLAS and CMS upgrade tracker designs.

The dataset provides the transverse momentum p_T, particle id (with a `particle_id` of zero indiciating a noise hit), detector layer number, and a set of 14 features (cylindrical coordinate (r, ϕ, z) of the hits, pseudorapidity η which is a spatial coordinate describing the angle of particle relative to the beam axis — where a 'hit' is the collision of a pair of protons)

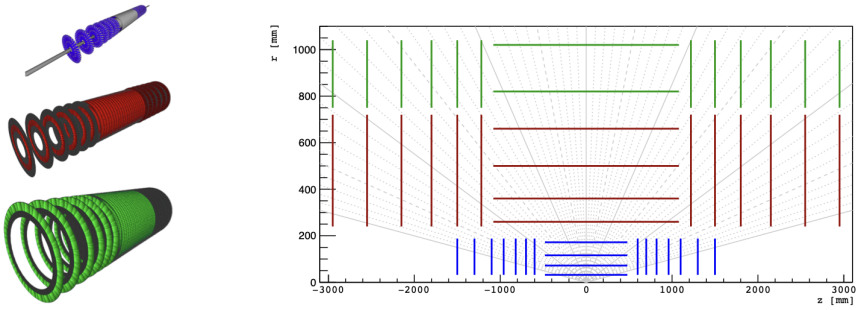


FIGURE 1.1: The `TrackML` detector, *image taken from*, [1]

(A) Pixel detector (B) Strip detector

FIGURE 1.2: Illustration of two-dimensional (a) and a one-dimensional segmentation (b) of a silicon sensor, *Image taken from the ACTS docs*, [18]

The core idea of charged particle detection is to leverage the ionization of nearby material upon its interaction with a charged particle. To this end semiconducting particle detectors are used, typically made of silicon. When a charged particle traverses such a sensor, it ionizes the material in the depletion zone, caused by the interface of two different semiconducting materials. The result are pairs of opposite charges. These charge pairs are separated by an electric field and drift toward the electrodes. At this point, an electric signal is created which can be amplified and read out [17].

The sensors are segmented so that a location can be affixed to the measured signal — segmentation can be performed in either one dimension (strip layers) or in two dimensions (pixel layers). The actual CMS and ATLAS detectors have several of these pixel and strip detector layers. The `TrackML` dataset makes a distinction between each detector layer and pixel and strip features.

1.1 The 'classical' approach, built on *Combinatorial Kalman Filters*

Classical approaches to charged particle tracking typically have the following skeleton, [17] (much of this material is taken from the excellent [18]):

1. *Clustering*: The raw readouts from the detector need to be clustered in order to extract an estimate of where particles intersect with the sensor

2. *Spacepoint formation*: The basic input to most forms of pattern recognition algorithms for tracking are space points, which need to be assembled from the raw measurements. To this end, the raw measurements are combined with information provided by the geometry description, such as the location and rotation of the sensors. In this way, the locations, which are restricted to be local to the sensor surfaces intrinsically, can be converted into three dimensional points in space

3. *Seeding*: Initial track candidates are constructed which will be iteratively improved upon.

4. *Track finding and Track fitting*: Here, the track candidates are extrapolated from the initial seeds, by far the most common algorithm used here is the *Combinatorial Kalman Filter* (CKF), [18].

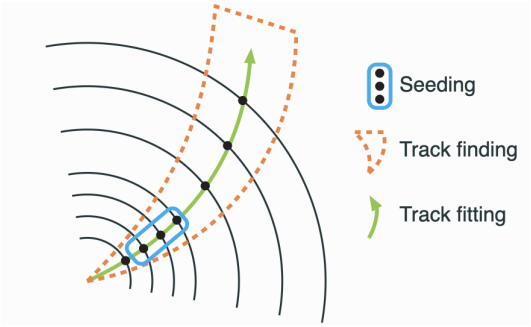


FIGURE 1.3: Illustration of a track reconstruction chain starting from spacepoints to fully formed tracks, *Image taken from the ACTS docs*, [18]

Particularly, all CKF implementations leverage the *Kalman Formalism* (henceforth referred to as KF), which can be used for the task of iteratively calculating track estimates. For the sake of completion, I wish to spend the rest of this section covering the KF from a high-level.

1.1.1 Track parametrization

The most standard way to parametrize tracks is via the so-called *Perigee* parametrization, [18], where the state of a particle is defined as:

$$\tilde{\mathbf{x}} := (l_0, l_1, \phi, \theta, q/p, t)^T$$

Where (l_0, l_1) are the local coordinates of the corresponding surface, (ϕ, θ) are the angles in the transverse and longitudinal direction of the global frame and q/p is product of the charge with the inverse momentum and t is the time.

(A) Local position and global momentum

(B) Angles θ, ϕ in the transverse and longitudinal planes

FIGURE 1.4: Illustration of the parametrization of a particle track with respect to a two-dimensional surface, *Figure taken from ACTS docs*, [18]

An important quantity associated with the above parametrization is the 5×5 covariance matrix that may be computed as such:

$$C := \begin{bmatrix} \sigma^2(l_0) & \text{cov}(l_0, l_1) & \text{cov}(l_0, \phi) & \text{cov}(l_0, \theta) & \text{cov}(l_0, q/p) \\ \cdot & \sigma^2(l_1) & \text{cov}(l_1, \phi) & \text{cov}(l_1, \theta) & \text{cov}(l_1, q/p) \\ \cdot & \cdot & \sigma^2(\phi) & \text{cov}(\phi, \theta) & \text{cov}(\phi, q/p) \\ \cdot & \cdot & \cdot & \sigma^2(\theta) & \text{cov}(\theta, q/p) \\ \cdot & \cdot & \cdot & \cdot & \sigma^2(q/p) \end{bmatrix}$$

1.1.2 Track fitting via KF

The Kalman fit process can be neatly partitioned into three steps, each of which attempts to improve the estimate of the state at the current iteration by leveraging information from prior/current and future iterates

1. *Prediction*: The prediction for the $k-th$ iterate based off of the $(k-1)$-th one is computed in the prediction stage as:
$$\vec{x}_k^{(k-1)} = F_{k-1}\vec{x}_{k-1} + \vec{w}_{k-1}$$
Where, $\mathbf{F}$ is assumed to be a linear transport function that can propagate the state vector from $\vec{x}_{k-1}$ to the k -th iterate (the existence of $\mathbf{F}$ is *technically* only guaranteed in the case of homogenous magnetic fields)

2. *Filtering*: The prediction for the k -th iterate is refined on the basis of a measurement vector $\vec{m}_k$ that can be computed on the basis of information contained in the current state leveraging the gain matrix formalism, as:
$$\vec{x}_k := \vec{x}_k^{(k-1)} + \mathbf{K}_k(\vec{m}_k - H_k\vec{x}_k^{(k-1)})$$

Where $\mathbf{K}_k$ is the *Kalman Gain Matrix* and H_k is a measurement mapping function that transforms a state vector into the 'measurement' space (which could be as simple as a projection matrix). $\mathbf{K}_k$ depends on the covariance matrix C defined above.

3. *Smoothing*: The final smoothing step tries to leverage information about the *prediction* for the future iterate (which can already be constructed from the filtered $\vec{x}_k$) for further refining the current state as:

$$\vec{x}_k^n := \vec{x}_k + A_k(\vec{x}_{k+1}^n - \vec{x}_{k+1}^k)$$

Where $\vec{x}_{k+1}^n$ is the smoothed future iterate and $\vec{x}_{k+1}^k$ is the predicted future iterate *at* the k -th iteration. Here, A_k is the *smoother gain matrix*, which also depends on the covariance matrix C (analogously to K_k).

A very complete implementation of the above pipeline can be found in the **ACTS** library [17], developed and maintained by the ATLAS collaboration.

1.2 Deep learing based approaches

There has been a surge of interest in exploring the applicability of deep learning approaches to charged particle tracking, in particular of *Graph Neural Networks* (GNNs) — this is motivated by the poor computational scaling of the above mentioned CKF based algorithm with pileup (*i.e.* overlapping secondary proton-proton collisions on top of the primary interaction) [2] and at the same time, by the excellent performance and scalability w.r.t. pileup demonstrated by these architectures [10].

The majority of such GNN based approaches have been developed around the edge classification (EC) paradigm.

I give a brief sketch of such an EC-based approach next

1.2.1 Edge-Classification based charged particle tracking

Tracking via edge classification typically involves three stages [8, 5].

1. *Graph Construction*: Silicon tracker hits are mapped to nodes after which an edge-building algorithm forms edges between certain nodes.

2. *Edge-Classification*: An edge-classifier GNN is trained to be able to predict probability that each edge in the above constructed graph corresponds to a true track segment (*i.e.*

the probability that an edge connects two nodes that correspond to tracker hits of the same particle)

3. *Graph segmentation*: A track-building algorithm leverages the edge weights to form full track candidates — this is done either by collecting the connected components or a simple graph traversal.

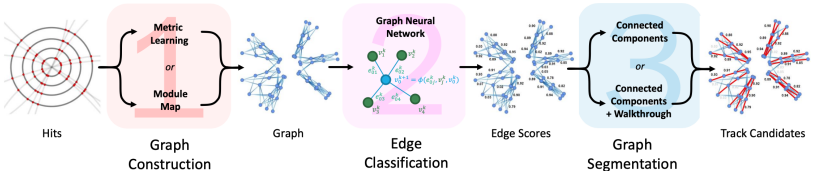


FIGURE 1.5: The `ExaTrkX` pipeline, *Figure taken from [3]*

A very complete implementation of the above scheme on the `TrackML` dataset has publicly made available by the **ExaTrkX** group, it can be found **here**, [4]

1.2.2 The learned clustering pipeline

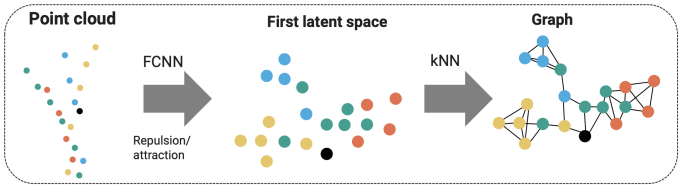
The learned clustering approach is distinct from EC-based pipelines as instead of refining the edges formed after the initial graph construction phase, it further refines corresponding node space and learns to cluster tracker hits from the same particles together. I give a higher-level overview of the steps in the pipeline below [11, 14]:

1. *Graph Construction* (GC): The construction of the initial graph proceeds almost identically to the EC approach, and we do end up with (almost) the same graph but with different features. A FCNN is used to produce learned clustering coordinates for each track hit, after which the graph is built from this latent space using k -nearest neighbours (kNN). The most important ingredient here is are the loss functions used to train this FCNN, we use an attractive, squared L^{att} and a repulsive hinge loss L^{rep}. The net loss is defined as $L_{\text{total}} := L^{\text{att}} + s_{\text{rep}} L^{\text{rep}}$. The attractive loss function rewards points that are 'close enough' to each other while the repulsive loss does the opposite. The ratio between these two in the net loss, s_{rep} is an extremely important hyperparameter for the GC.

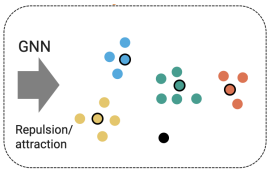
2. *Learned Clustering*: After we have constructed the initial graph, we now have to reconstruct the actual physical tracks. This is done by further refining the latent space that was constructed in the previous GC stage using a GNN. The same loss as the GC-phase is used (linear combination of attractive and repulsive losses).

3. In the final step, we cluster the hits in the refined latent space (as done by the GNN in the learned clustering step) and collect them via DBSCAN.

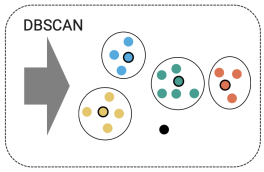
The below diagram captures the entire pipeline succinctly.



(A) Stage-1: Graph Construction



(B) Stage-2: Object Condensation phase



(C) Stage-3: Collect Clusters

FIGURE 1.6: The entire pipeline at a glance, *figure taken from [13]*

Much of the core contributions of this thesis are extending this learned clustering pipeline in various ways, and we will be discussing details of its various moving parts as we go along.

Chapter 2

Exploring pre-emptive noise-classification for the learned clustering pipeline

2.1 Motivation

There are noise hits in the raw point cloud data — a noise hit is a detector signal that does not correspond to an actual particle from the collision.

If we are able to remove the noise hits from the data *before* the GC phase, then that improves both the training and inference time for both the GC and the OC but we also hope to see improvements in the final evaluation metrics.

In the next section I show results for the learned clustering pipeline when run with *perfect* noise classification (using a trivial ground-truth classifier) — the intention with this was to get an experimental 'upper-bound' on the performance gain that we can expect from the addition of a pre-emptive noise classifier.

After that, we provide some results from a study where we implemented such a classifier for the problem using FCNN's.

2.2 Feasibility results

We use the following metrics to assess track reconstruction performance [11]:

1. **Perfect match efficiency** ($\epsilon^{\text{perfect}}$): The number of reconstructed tracks that include all hits of the matched particle and no other hits, normalized to the number of particles.
2. **LHC-style match efficiency** (ϵ^{LHC}): The fraction of reconstructed tracks in which 75% of the hits belong to the same particle, normalized to the number of reconstructed tracks.
3. **Double Majority match efficiency** (ϵ^{DM}): The fraction of reconstructed tracks in which at least 50% of the hits belong to one particle and this particle has less than 50% of its hits outside of the reconstructed track, normalized to the number of particles.
4. Total Validation Loss

Variants of each of these quantities for particles of $p_T > c$ GeV are denoted as: $\epsilon^{\{\text{DM, perfect, LHC}\}}_{p_T>c}$

(**Note** These are results for 'perfect' pre-emptive noise classification)

	$\epsilon^{perfect}_{p_T>0.9}$	$\epsilon^{DM}_{p_T>0.9}$	$\epsilon^{LHC}_{p_T>0.9}$
Noise-Classification	0.85	0.97	0.979
Vanilla	0.76	0.94	0.97

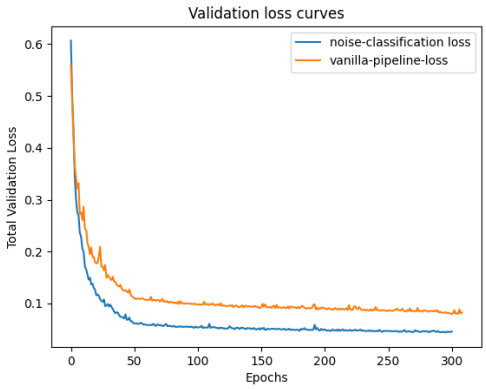


FIGURE 2.1: OC Validation loss curves

Note that the track reconstruction metrics in the above table are computed *after* the final OC runs in both cases with the noise-classifier in the noise-classification run having been applied *before* its corresponding GC run — and the performance of the GC run is *pivotal* for the OC training that it eventually feeds into.

A better measure of the quality of graph construction is the metric `n_edges_frac_segment50_95`. Simply put, it is a metric that computes the number of edges that would need to be constructed by the GC phase to have 95% of the reconstruted tracks to be *sufficiently well connected* (which in essence means that a perfect EC executed on this graph could reconstruct a track that satisfies the above-stated double-majority criterion).

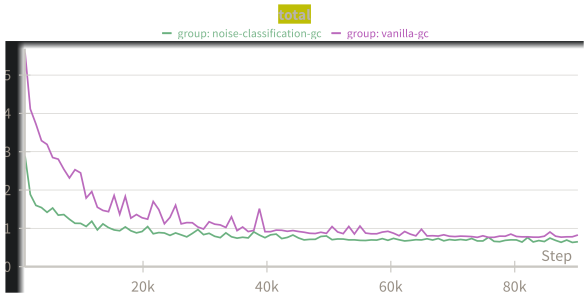


FIGURE 2.2: GC loss curves

But note it is incorrect to compare both of those numbers directly because of the smaller (and 'higher quality') nodes that the noise-classification model is trained on, so to make the comparison more appropriate we scale the `n_edges_frac_segment50_95` metric for the vanilla model by $(1 - \texttt{fraction\_of\_noise})$ (which in the case of the current dataset is ≈ 0.06).

	Noise classification GC	Vanilla GC (scaled by $(1 - \texttt{fraction\_of\_noise})$)
`n_edges_frac_segmen--t50_95` (*lower is better*)	316654	383793

For the purposes of reproduction, I provide some important configuration parameters and details on the runs below:

	Vanilla OC	Noise-Classification OC

`cluster_scanner`	`DBSCANHyperParamScanner`
	`n_trials: 60`
	`n_jobs: 6`
	`keep_best: 30`
		Identical

`loss_fct`	`GraphConstructionHingeEmbeddingLoss`
	`lw_repulsive: 0.05`
	`pt_thld: 0.9`
	`max_num_neighbours: 256`
	`p_attr: 2`
	`p_rep: 2`
	`r_emb: 1`
		Identical

`optimizer`	`Adam` `lr:` 8×10^{-4}	*Identical*

scheduler	LinearLR start_factor: 1 end_factor: 0.1 total_iters: 50	*Identical*
model	GraphTCNForMLGCPipeline node_indim: 22 edge_indim: 44 h_dim: 192 e_dim: 192 hidden_dim: 24 h_outdim: 24 L_hc: 5 alpha_latent: 0.5 n_embedding_coords: 8	*Identical*
num_epochs	300	300
preproc	*Vanilla GC model, trained for 100 epochs*	*Noise-Classification GC model, trained for 100 epochs*

	Vanilla GC	Noise-Classification GC
model	GraphConstructionFCNN alpha: 0.6 depth: 6 hidden_dim: 256 in_dim: 14 out_dim: 8	*Identical*
loss_fct	GraphConstructionHingeEmbeddingLoss lw_repulsive: 0.06 pt_thld: 0.9 max_num_neighbours: 256 p_attr: 2 p_rep: 2 r_emb: 1	*Identical*

`gc_scanner`	`GraphConstruction-KNNScanner` `ks: list(range(10))`	*Identical*
`preproc`	`None`	`NoiseClassifierModel`
`optimizer`	`Adam` `lr:` 7×10^{-4}	*Identical*
`scheduler`	`ExponentialLR` `gamma: 0.985`	*Identical*
`num_epochs`	100	100

2.3 A simple attempt at realization of a noise-classifier

We attempted to realise the above noise-classifier via an FCNN with residual connections and a weighted classification loss.

Note that, an interesting feature of the binary classification task at hand is the imbalanced nature of the classes *i.e.* there are far more non-noise hits than there are noise hits. The typical way to deal with class imbalance in a classification setting is to weight the classification loss appropriately (a good rule of thumb is to scale the terms out in proportion to the imbalance, which is the heuristic that we follow in our weighting).

Another really important point we have to keep in mind while performing the classification is to keep our *false-positive* rate to a minimum since filtering out correct track hits could potentially disproprtionately impact final track reconstruction performance — we quantify this by providing the ROC (Receiver Operating Characteristic) curve for the final model and quantifying its classification efficacy via the RUC curve and its corresponding area under the curve (AUC-RUC)

A final consideration is that ideally, the model should be lightweight enough to allow for fast enough inference at runtime since this will be an additional model that needs to be evaluated (alongside the FCNN for the graph construction and the GNN for the refinement phase)

These are functions that we used for computing the FPR (False Positive Rate) and TPR (True Positive Rate) for the final model (the ROC curve is a `FPR v TPR` plot):

```
def compute_fpr(preds, thresholds, good_mask):
    fpr = []
    for t in thresholds:
        # dividing by 'size' of the mask because *all* of the elements
        # 'chosen' by the mask are non-noise, particles of interest
        # (#FP_predict_over_good)/(#total_good)
```

```
        fpr.append(torch.sum(preds[good_mask] > t) / torch.sum(good_mask))
    return fpr

def compute_tpr(preds, target, thresholds):
    tpr = []
    for t in thresholds:
        # (#total_predict_pos - #total_predict_false_pos)/(#total_predict_pos)
        tpr.append((torch.sum(preds > t) - torch.sum((preds > t)[~target]))
                   / torch.sum(target))
    return tpr
```

Note that we compute these rates for the cases of both the particles 'of interest' (labelled, `FPoiR`) and for *all* particles (labelled, `FPR`) (note that the exact definition of 'particles of interest' is given alongside the definition of the loss in Chapter-3)

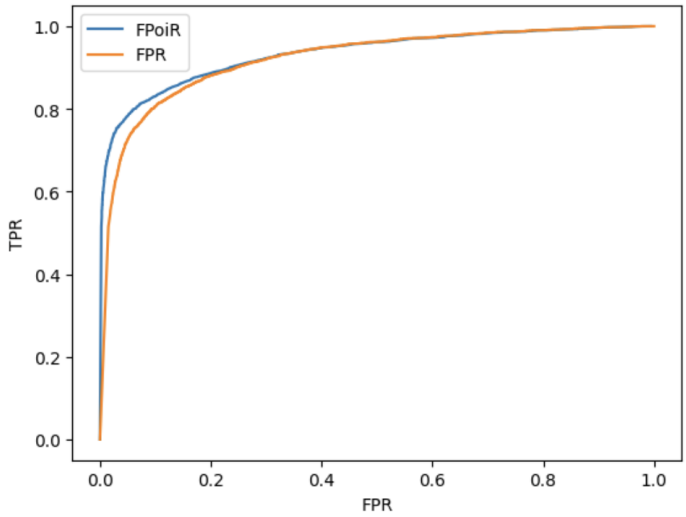
As before, I provide the configuration for being able to reproduce the above results below:

I wrote a dedicated pytorch-lightning module (to be able to override the `get_losses` method, since unlike the other models we've trained so far, this required the use of the binary cross entropy loss) — I also opted to use the existing `GraphConstructionFCNN` class which we have been using so far for the graph-construction phase with a slight modification of its output layer to make it amenable for binary classification tasks (added via the introduction of a `classification: bool` parameter during instantiation)

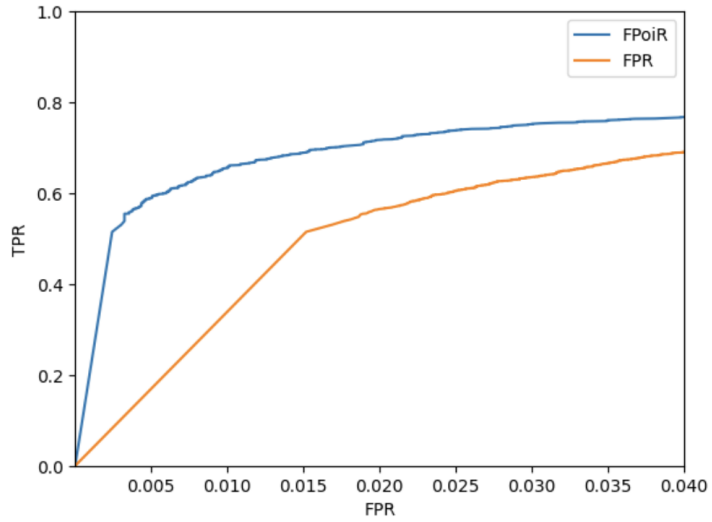
Name of argument	*Instantiated class*

`model`	`GraphConstructionFCNN`
	`alpha: 0.6`
	`depth: 6`
	`hidden_dim: 256`
	`in_dim: 14`
	`out_dim: 2`
	`classification: True`

`loss_fct`	`CEL` (a thin wrapper around `torch.nn.CrossEntropyLoss`) `weight: [0.9348, 0.0652]`
`optimizer`	`torch.optim.Adam` `lr:` 1×10^{-3}
`scheduler`	`torch.optim.lr_scheduler.ExponentialLR` `gamma: 0.985`



(A) RUC curves overlayed for particles of interest and for all particles



(B) Zoomed figure

FIGURE 2.3: RUC curves for the FCNN noise classifier, both have an `AUC=` $\sim 92\%$

For completions sake, I also provide some pre-liminary results that we had from using the `XGBoost` model for the same class (also with a weighted classification loss). The `XGBClassifier` was trained on a very small subset of the entire `TrackML` dataset (only about $\approx$ 80 files $\sim$ 5 million nodes compared to the 7743 $\sim$ 1 billion nodes available in the entire dataset).

Parameters used for `XGBClassifier`:

```
scale_pos_weight = (data.particle_id.shape[0] - num_pos)/num_pos
```

```
{
```

```
    'n_estimators': 5000,
    'max_depth': 100,
    'learning_rate': 1, # default
    'scale_pos_weight': scale_pos_weight,
    'max_delta_step': 5,
    'objective': 'binary:logistic'
}
```

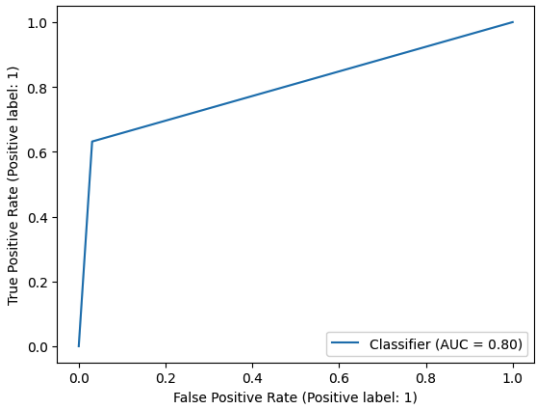


FIGURE 2.4: `FPR` *vs* `TPR`, `XGBoost`

2.4 Key Takeaways and conclusion

For this application, we were interested in keeping `FPR` $< \sim 0.5\%$ for any model that we learnt — and as one can observe from the RUC curves, the FCNN with residual connections and an appropriately weighted binary cross entropy loss is able to learn to remove $\sim 60\%$ of the noise from the dataset.

There is potentially more exploration that could be done along these lines, perhaps with models that are able to capture node-node interactions of the recorded tracker hits

Chapter 3

A contrastive loss for the learned clustering pipeline

3.1 Contrastive learning, background

In contrastive learning, our objective is to be able to learn an embedding space in which 'similar' sample pairs are naturally closer *and* simultaneously, 'dissimilar' ones are far apart.

There is a lot in the way of convenience afforded to us by the contrastive learning framework, in that, it can be readily modified to suit operation in unsupervised, self-supervised (where 'supervision' is introduced via heavy data augmentation of input samples) and supervised (which is the case for our tracking problem) regimes [9]

We already employ a composite contrastive loss as part of the learned clustering pipeline [11, 14] — the details of which we alluded to in Chapter-1. The attractive and repulsive portions of the loss essentially achieve the same effect as would a contrastive loss of ensuring tracker hits from the same hit are well-clustered together while actively repulsing hits belonging to a different particle (we have access to the labels for each of these hits `data.particle_id`).

The attractive portion of the loss is a quadratic function while the repulsive portion is a hinge loss as, [14]:

$$L^{\text{att}} := \frac{1}{|I_{\text{att}}|} \sum_{(i,j) \in I_{\text{att}}} \left|\left| h_i^{\text{GC/OC}} - h_j^{\text{GC/OC}} \right|\right|^2$$

$$I_{\text{att}} := \{(i,j) | 1 \leq i, j \leq N_{\text{hits}}, \pi_i = \pi_j, \pi_i \text{ of interest}\}$$

$$L^{\text{rep}} := \frac{1}{|I_{\text{rep}}|} \sum_{(i,j)\in I_{\text{rep}}} \text{ReLU}(1 - \left|\left|h_i^{\text{GC/OC}} - h_j^{\text{GC/OC}}\right|\right|^2)$$
$$I_{\text{rep}} := \{(i,j) | 1 \leq i,j \leq N_{\text{hits}}, \pi_i \neq \pi_j \text{ or noise}\}$$

Where, π_i denotes the particle of hit i, with *'particles of interest'* being defined as all particles having $p_T > 0.9$ GeV, $|\eta|$¡4.0 (pseudo-rapidity) that have at least three hits.

In these set of experiments, we were interested in exploring if using a classical contrastive loss would be better suited for our contrastive learning step as opposed to the composite loss.

3.2 Our choice of contrastive loss

We chose the `InfoNCE` loss [16, 21] for our experiments as it is particularly easily amenable to a supervised formulation (and also because it has been recently used for solving the tracking problem on the `trackML` dataset, [15] and also for the task of learning representations in jet physics [6]).

Intuitively, the intent behind the `InfoNCE` loss is being able to predict future information in a way that maximally preserves mutual information of the original signals x (future) and the context c (present) defined as:

$$I(x;c) := \sum_{x,c} p(x,c) \log \frac{p(x|c)}{p(x)}$$

i.e. Instead of making future predictions x_{t+k} directly with a generative model $p(x_{t+k}|c_t)$ (as we typically do), we instead model the density ratio that can preserve the mutual information between x_{t+k} and c_t:

$$f_k(x_{t+k}, c_t) \propto \frac{p(x_{t+k}|c_t)}{p(x_{t+k})}$$

Given a set $X = \{x_1, \ldots x_N\}$ of N random samples containing one positive sample from $p(x_{t+k}|c_t)$ and $(N-1)$ negative samples from the proposal distribution $p(x_{t+k})$ — the `InfoNCE` loss is:

$$\mathcal{L}_N := -\mathbb{E}_X \left[\log \frac{f_k(x_{t+k}, c_t)}{\sum_{x_j \in X} f_k(x_j, c_t)} \right]$$

Optimizing this above loss function ends up in estimating the density ratio $f_k(x_{t+k}, c_t)$ that we defined above.

The above loss can be read off as being the categorical cross-entropy of classifying the positive sample correctly with $\frac{f_k}{\sum_X f_k}$ being the prediction of the model.

We write the optimal probability for this loss as $p(d = i|X, c_t)$ with $[d = i]$ being the indicator that the sample x_i is the 'positive' sample. We are interested in the probability that sample x_i was drawn from the conditional distribution $p(x_{t+k}|c_t)$ rather than the proposal distribution $p(x_{t+k})$

$$p(C = \text{pos}|X, \mathbf{c}) = \frac{p(x_{\text{pos}}|\mathbf{c})\Pi_{i=1,\ldots,N;i\neq \text{pos}} p(x_i)}{\sum_{j=1}^{N}\left[p(x_j|c)\Pi_{i=1,\ldots,N;i\neq j} p(x_i)\right]} = \frac{\frac{p(x_{\text{pos}}|c)}{p(x_{\text{pos}})}}{\sum_{j=1}^{N} f(x_j, c)}$$

The optimal value for $f(x_{t+k}, c_t)$ is proportional to $\frac{p(x_{t+k}|c_t)}{x_{t+k}}$ and this is independent of the choice of the number of negative samples $(N - 1)$

The above stated form of the loss is suitable for unsupervised/self-supervised contrastive learning — however, since we operate in the supervised regime, we use a slighty different variation of the `InfoNCE` loss that allows for the incorporation of label information, [15]:

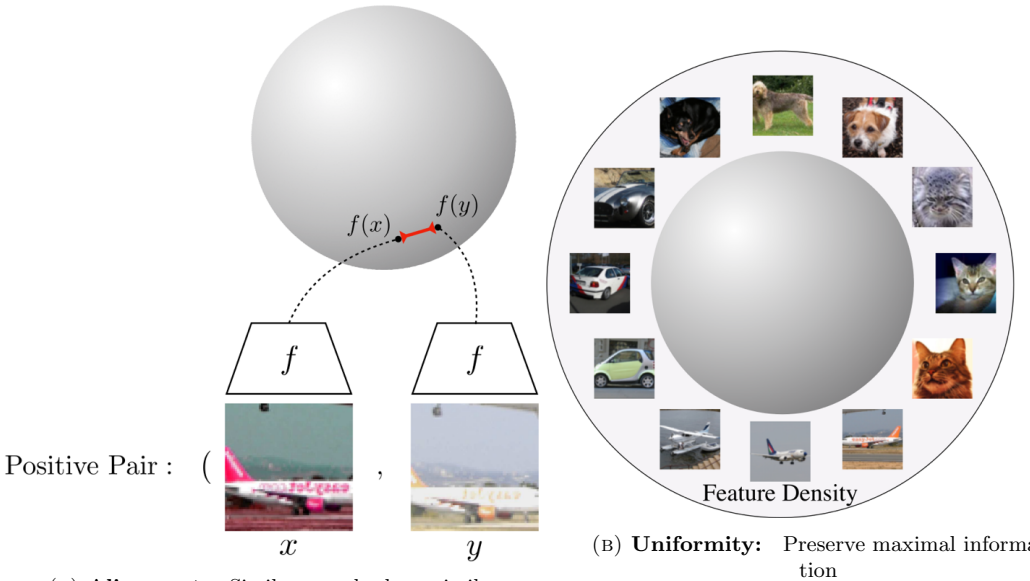
$$\mathcal{L}_{\text{InfoNCE}} = -\sum_{(u,v)\in\mathcal{B}} \log \frac{\text{sim}(\mathbf{h}_u, \mathbf{h}_v^+)}{\exp(\text{sim}(\mathbf{h}_u, \mathbf{h}_v^+)) + \sum_{\mathbf{h}_v^-\in\mathcal{N}} \exp(\text{sim}(\mathbf{h}_u, \mathbf{h}_v^-))} \quad \text{where } \mathcal{B} \text{ is the current batch}$$

Where, $\mathbf{h}_u$ is the current embedding that we are summing over and $\mathbf{h}_u^{+/-}$ are the set of all other embeddings in the batch that have the same (/ different) label as $\mathbf{h}_u$. $\text{sim}(\cdot, \cdot)$ is a function for computing a 'similarity' metric between any two embeddings (analogue to the 'scoring function', $f(x, c) \propto \frac{p(x|c)}{p(x)}$ that we ran into earlier), for our experiments we adopt $\text{sim}(\cdot, \cdot)$ as $\exp(-||\mathbf{h}_u - \mathbf{h}_v||^2/\tau)$, where τ is often referred to as the *temperature* parameter the variation of which has an interesting theoretical effect which I briefly describe next, [19, 20].

3.2.1 The Alignment vs Uniformity problem

Two key properties that are often associated with contrastive losses are that of *Alignment* and *Uniformity*, [20]. This discussion is in the context of the unit-hypersphere $\mathcal{S}^n$ serving as an intermediate feature space — this is often implicitly imposed in problems by constraining feature vectors to having a unit- l_2 norm.

Roughly speaking, alignment is the hope that the contrastive loss would map two samples that form a positive pair to nearby features. While, uniformity is the hope that the feature vectors should be (roughly) uniformly distributed on the unit hypersphere $\mathcal{S}^{m-1}$ to preserve



(A) **Alignment:** Similar samples have similar features

(B) **Uniformity:** Preserve maximal information

FIGURE 3.1: Alignment and Uniformity, *figure taken from [20]*

as much information as possible (because the uniform distribution is the unique distribution that minimizes the expected pairwise potential and hence retains maximal information from the initial space)

There have been attempts to quantify adherence to these properties as metrics (losses) in [20] as:

Definition 1. The alignment loss is defined as the expected distance between positive pairs, [20]:

$$\mathcal{L}_{\text{align}} := \underset{(x,y)\in p_{\text{pos}}}{\mathbb{E}} \left[||f(x) - f(y)||_2^{\alpha}\right], \quad \alpha > 0$$

Defining a similar metric for uniformity is a bit more trickier since it would require choosing a distribution that we optimize this metric over to converge to a uniform distribution (asymptotically correct) and also be easy to represent with a finite number of points — a natural choice for the same is the Gaussian potential kernel (Radial Basis Functions (RBF's)), $G_t : \mathcal{S}^d \times \mathcal{S}^d \to \mathbb{R}_+$:

$$G_t(u, v) := e^{-t||u-v||_2^2} = e^{2t \cdot u^T v - 2t}, \quad t > 0$$

Definition 2. The uniformity loss can now be defined as the log-average-pairwise Gaussian potential, [20]:

$$\mathcal{L}_{\text{uniform}}(f; t) := \log \underset{\substack{i.i.d \\ x,y\sim p_{\text{data}}}}{\mathbb{E}} \left[G_t(u, v)\right], \quad t > 0$$

It can be observed that the choice of the temperature parameter has a significant effect on how much we want our model to value either of these quantities

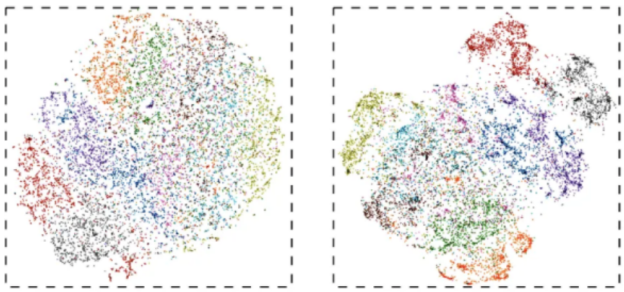


FIGURE 3.2: Alignment vs Uniformity tradeoff with τ

Recall that all standard loss functions can be derived under certain probabilistic assumptions on the distribution of the underlying data *i.e.* all loss functions have an associated *generating* distribution under which they are 'optimal'. For example, the mean-squared loss is optimal when the errors of the underlying data are distributed as a Gaussian, a more niche example would be the hyperbolic secant distribution serving as the generating distribution of the $\log\cosh$, [7].

Hence, varying τ directly affects the properties of the generating distribution corresponding to the `InfoNCE` loss in terms of the aforementioned defined *alignment* and *uniformity* properties.

3.3 Implementation details and results

The major implementation detail to keep in mind for the `InfoNCE` loss was the correct selection of the positive and negative samples for the current hit under consideration — thankfully, there existed helper methods for performing this inside of the `gnn_tracking` library, [12], that we have been extending so far, namely — `gnn_tracking.metrics.losses.info_nce_loss.InfoNCELoss._get_edges` can be used for gaining access to the parameters `(att_edges, rep_edges)` (which are essentially the sets of 'positive' and 'negative' hits, respectively).

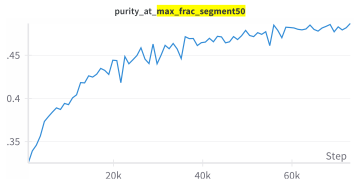
The rest of the implementation reads naturally from the above formula for the supervised variant of the `InfoNCE` loss.

```
def _InfoNCE_loss(
    *,
    x: T,
    att_edges: T,
    rep_edges: T,
    sim_func: str,
```

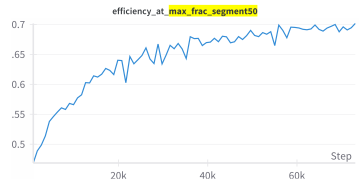
```
    tau: float
):
    if sim_func == 'l2_rbf':
        sigma = 0.75
        # distance between `attractive_edges`
        l2_dist_1 = torch.linalg.norm(x[att_edges[0]] - x[att_edges[1]], ord=2, dim=-1)
        # distance between `repulsive_edges`
        l2_dist_2 = torch.linalg.norm(x[rep_edges[0]] - x[rep_edges[1]], ord=2, dim=-1)
        similarity_1 = torch.exp(-l2_dist_1 / (2 * sigma**2))
        similarity_2 = torch.exp(-l2_dist_2 / (2 * sigma**2))

        max_sim_1 = (similarity_1 / tau).max()
        max_sim_2 = (similarity_2 / tau).max()
        exp_sim_1 = torch.exp(similarity_1 / tau - max_sim_1)
        exp_sim_2 = torch.exp(similarity_2 / tau - max_sim_2)

    num = exp_sim_1
    denom = num + torch.sum(exp_sim_2)
    return torch.mean(-torch.log(num / denom))
```

We performed some initial experiments with the implementation of this new loss on a truth-cut variant of the data (cut on $p_T > 1.5$), and the results are fairly promising!

(A) `max_frac_segment50` curve



(B) `purity_at_max_frac_segment50` curve



(C) `efficiency_at_max_frac_segment50` curve

FIGURE 3.3: Contrastive loss, preliminary results on truth-cut data, $\sim$ 900 files ($\approx$ 50 million nodes)

Here, the key metric is `max_frac_segment50` (with the efficiency and purity metrics being derivative in a certain sense). With this metric (and all of its 'derivative' metrics) we are interested in quantifying how 'well-connected' the output GC graph is, in particular, we are

interested in the fraction of particles in this output graph whose $> 50\%$ hits are connected by *true* edges (edges which connect hits belonging to the same particle) — the `max` in the name of this metric comes from the fact that we scan over multiple k while constructing the actual graph via kNN (as mentioned in Chapter-1) — i.e. the value of this metric will correspond to the highest value of k and simply give the value of that k.

Efficiency, is the number of true edges of interest (edges connecting two hits of which at least one is a particle of interest — where a particle of interest is as defined at the start of this chapter) normalized to the number of possible true edges of interest ($\sum_{\pi \text{ of interest}} \binom{N^{\pi}_{\text{hits}}}{2}$ where N^{π}_{hits} are the number of hits for a particle π) [14].

While purity is the number of true edges normalized to the number of edges of interest [14].

Bibliography

[1] Sabrina Amrouche et al. "The Tracking Machine Learning Challenge: Accuracy Phase". In: *The Springer Series on Challenges in Machine Learning*. Springer International Publishing, Nov. 2019, 231–264. ISBN: 9783030291358. DOI: 10.1007/978-3-030-29135-8_9. URL: http://dx.doi.org/10.1007/978-3-030-29135-8_9.

[2] Daniele Bertolini et al. "Pileup per particle identification". In: *Journal of High Energy Physics* 2014.10 (Oct. 2014). ISSN: 1029-8479. DOI: 10.1007/jhep10(2014)059. URL: http://dx.doi.org/10.1007/JHEP10(2014)059.

[3] Daniel Murnane Sylvain Caillou. *Graph Neural Networks for High Luminosity Track Reconstruction*. https://indico.cern.ch/event/1104699/attachments/2446264/4196018/GNN%20for%20HL-LHC.pdf. 2022.

[4] Exa.TrkX Collaboration. *Tracking with ML*. 2023. URL: https://github.com/HSF-reco-and-software-triggers/Tracking-ML-Exa.TrkX (visited on 04/30/2024).

[5] Gage DeZoort et al. "Charged Particle Tracking via Edge-Classifying Interaction Networks". In: *Computing and Software for Big Science* 5.1 (Nov. 2021). ISSN: 2510-2044. DOI: 10.1007/s41781-021-00073-z. URL: http://dx.doi.org/10.1007/s41781-021-00073-z.

[6] Barry Dillon et al. "Symmetries, safety, and self-supervision". In: *SciPost Physics* 12.6 (June 2022). ISSN: 2542-4653. DOI: 10.21468/scipostphys.12.6.188. URL: http://dx.doi.org/10.21468/SciPostPhys.12.6.188.

[7] Aryaman Jeendgar et al. *LogGENE: A smooth alternative to check loss for Deep Healthcare Inference Tasks*. 2023. arXiv: 2206.09333 [cs.LG].

[8] Xiangyang Ju et al. "Performance of a geometric deep learning pipeline for HL-LHC particle tracking". In: *The European Physical Journal C* 81.10 (Oct. 2021). ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-021-09675-8. URL: http://dx.doi.org/10.1140/epjc/s10052-021-09675-8.

[9] Prannay Khosla et al. *Supervised Contrastive Learning*. 2021. arXiv: 2004.11362 [cs.LG].

[10] Alina Lazar et al. "Accelerating the Inference of the Exa.TrkX Pipeline". In: *Journal of Physics: Conference Series* 2438.1 (2023), p. 012008. DOI: 10.1088/1742-6596/2438/1/012008. URL: https://dx.doi.org/10.1088/1742-6596/2438/1/012008.

[11] Kilian Lieret and Gage DeZoort. *An Object Condensation Pipeline for Charged Particle Tracking at the High Luminosity LHC*. 2023. arXiv: 2309.16754 [physics.data-an].

[12] Kilian Lieret and Gage deZoort. *gnn_tracking: An open-source GNN tracking project*.

[13] Kilian Lieret and Gage DeZoort. *High Pileup Particle Tracking with Learned Clustering*. https://indico.cern.ch/event/1330797/contributions/5796857/. 2024.

[14] Kilian Lieret et al. *High Pileup Particle Tracking with Object Condensation*. 2023. arXiv: 2312.03823 [physics.data-an].

[15] Siqi Miao et al. *Locality-Sensitive Hashing-Based Efficient Point Transformer with Applications in High-Energy Physics*. 2024. arXiv: 2402.12535 [cs.LG].

[16] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. *Representation Learning with Contrastive Predictive Coding*. 2019. arXiv: 1807.03748 [cs.LG].

[17] Andreas Salzburger et al. *A Common Tracking Software Project*. Version 10.0.0. July 2021. DOI: 10.5281/zenodo.5141419. URL: https://github.com/acts-project/acts.

[18] Andreas Salzburger et al. *A Common Tracking Software Project, documentation*. 2024. URL: https://acts.readthedocs.io/en/latest/tracking.html (visited on 05/01/2024).

[19] Feng Wang and Huaping Liu. *Understanding the Behaviour of Contrastive Loss*. 2021. arXiv: 2012.09740 [cs.LG].

[20] Tongzhou Wang and Phillip Isola. *Understanding Contrastive Representation Learning through Alignment and Uniformity on the Hypersphere*. 2022. arXiv: 2005.10242 [cs.LG].

[21] Lilian Weng. *Contrastive Representation Learning*. 2021. URL: https://lilianweng.github.io/posts/2021-05-31-contrastive/ (visited on 04/29/2024).