ATLAS EXPERIMENT

CERN

Deep Learning for Particle Tracking
a HEP.TrkX update

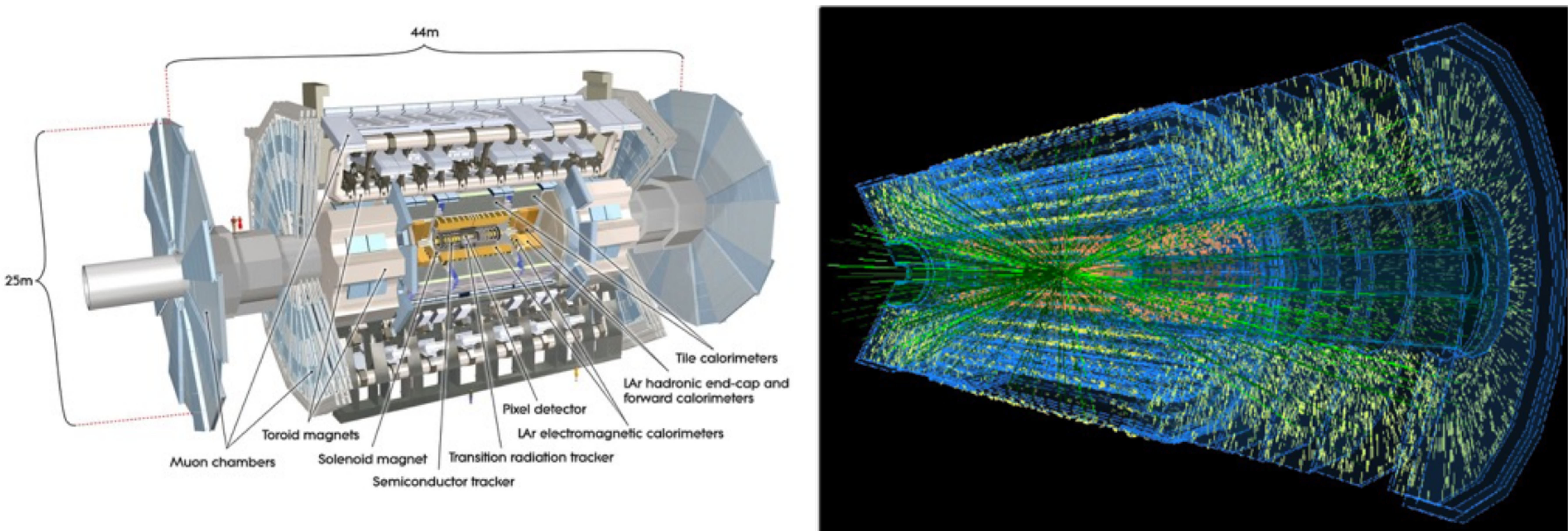
Steve Farrell

LBL-HEP-ML Meetup

Berkeley Lab
Ernest Orlando Lawrence Berkeley National Laboratory

Oct 25, 2017

The HL-LHC tracking problem



- Reconstruct thousands of particle tracks from tens of thousands of spacepoint "hits" per beam collision in highly granular detectors (100M channels)
- Traditional approach builds triplet "seeds" then a combinatorial Kalman Filter to build track candidates
- The HEP.TrkX project is exploring various machine learning ideas to try and tackle this challenging pattern recognition problem
  - Collaboration with FNAL and Caltech

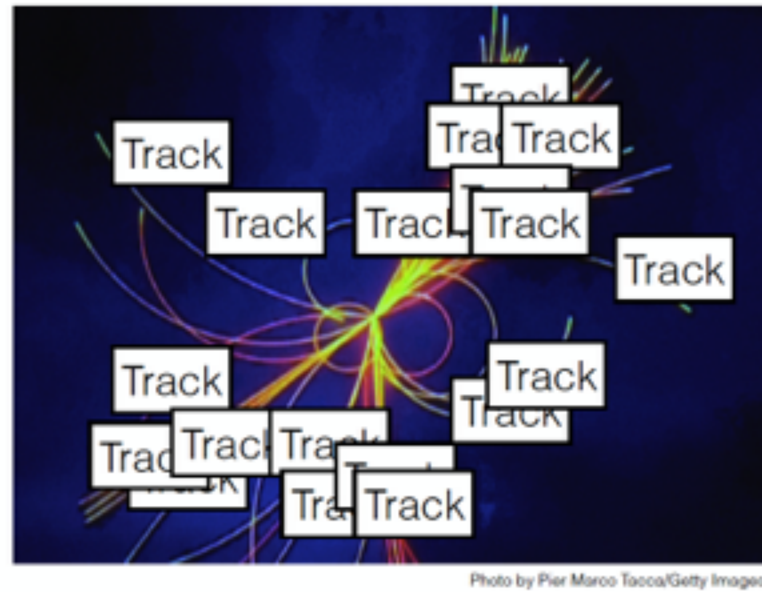
Machine learning for tracking

- Applications
  - Clustering hits into tracks
  - Classifying hits (binary or multi)
  - Classifying track candidates
  - Fitting tracks
- Representations
  - Discrete (image-like) vs. continuous (point-cloud)
  - Hit assignments vs. physics quantities
  - Engineered vs. learned representations

Image-based approaches

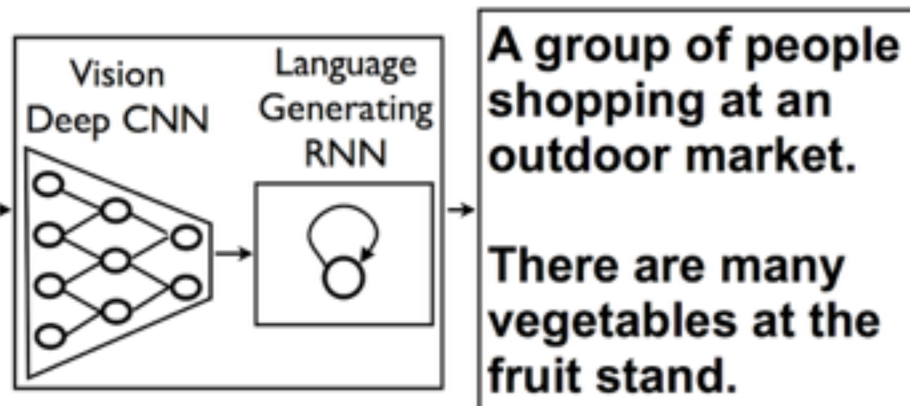
Image segmentation

Our goal (more or less…):



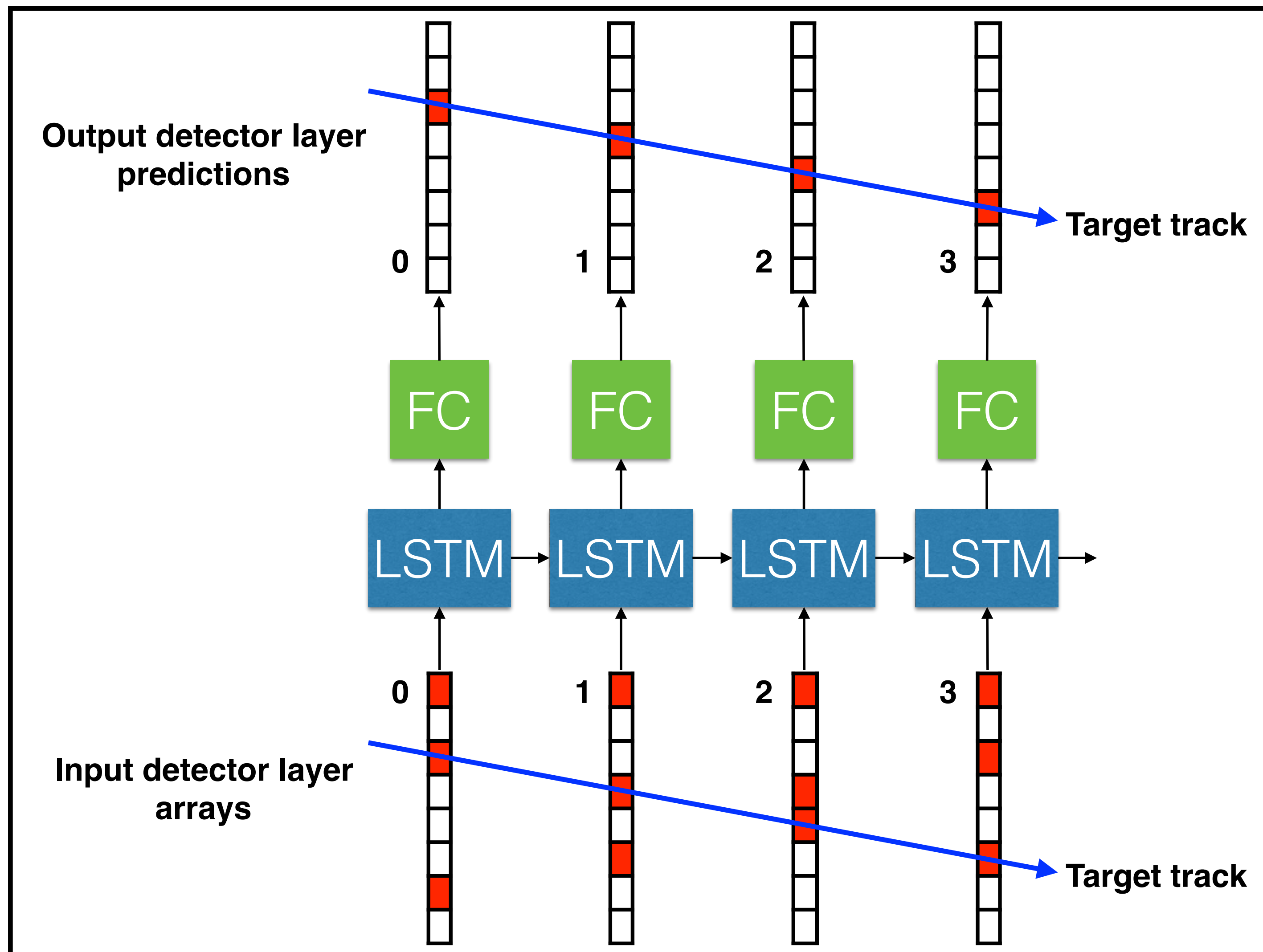
https://arxiv.org/abs/1604.02135

Image captioning

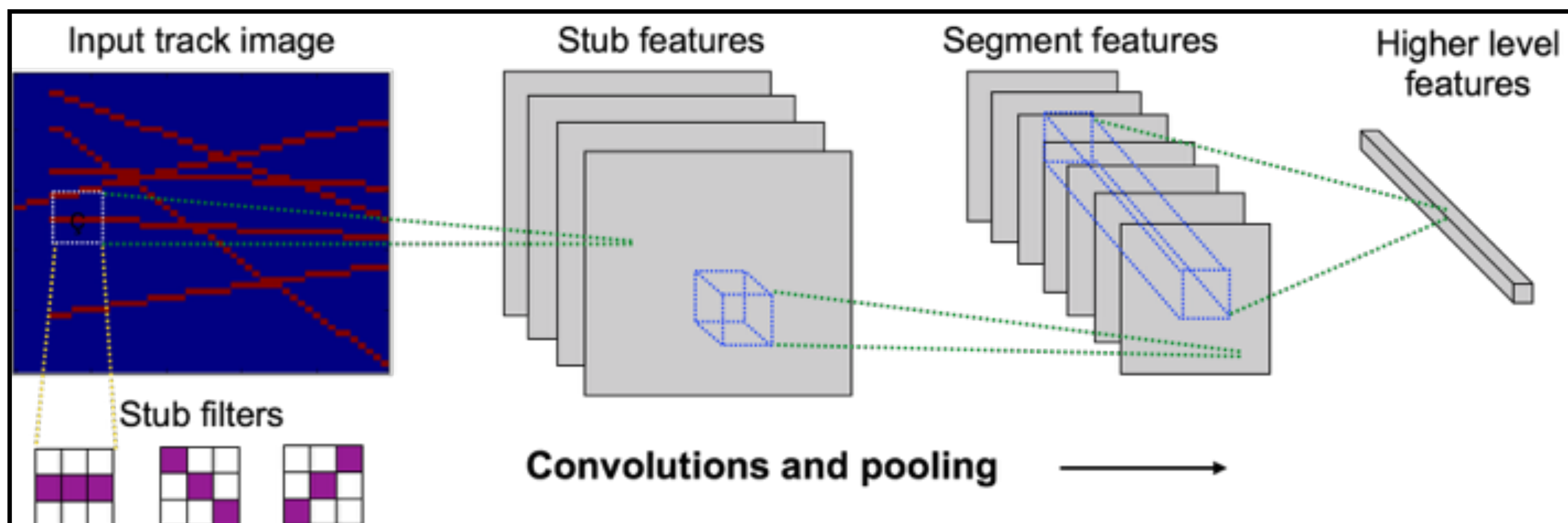


- Analogs in well-known computer vision tasks
  - segmentation
  - captioning
  - object tracking
- Tracks are *patterns* to be discovered
  - local and hierarchical structure
  - symmetries in the geometry and physics
- Detector layers might also be considered *frames* of a video
  - causally related by the particle dynamics

“Semantic segmentation” with RNNs, CNNs

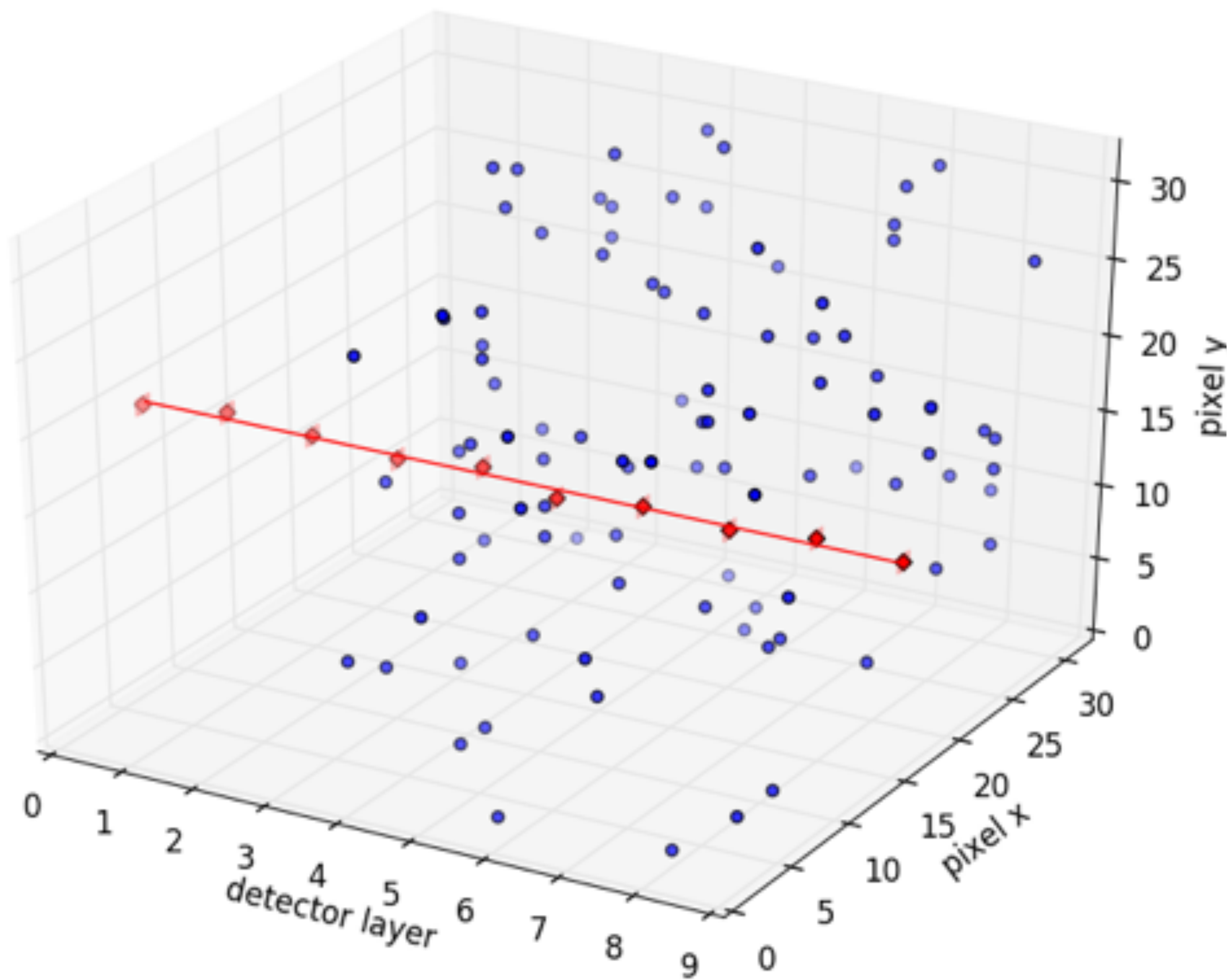


- Recurrent networks can classify pixels layer-by-layer
  - extrapolation + state estimation (like KF)
- Convolutional networks can classify pixels with hierarchical pattern finding
  - extrapolates in all directions at once

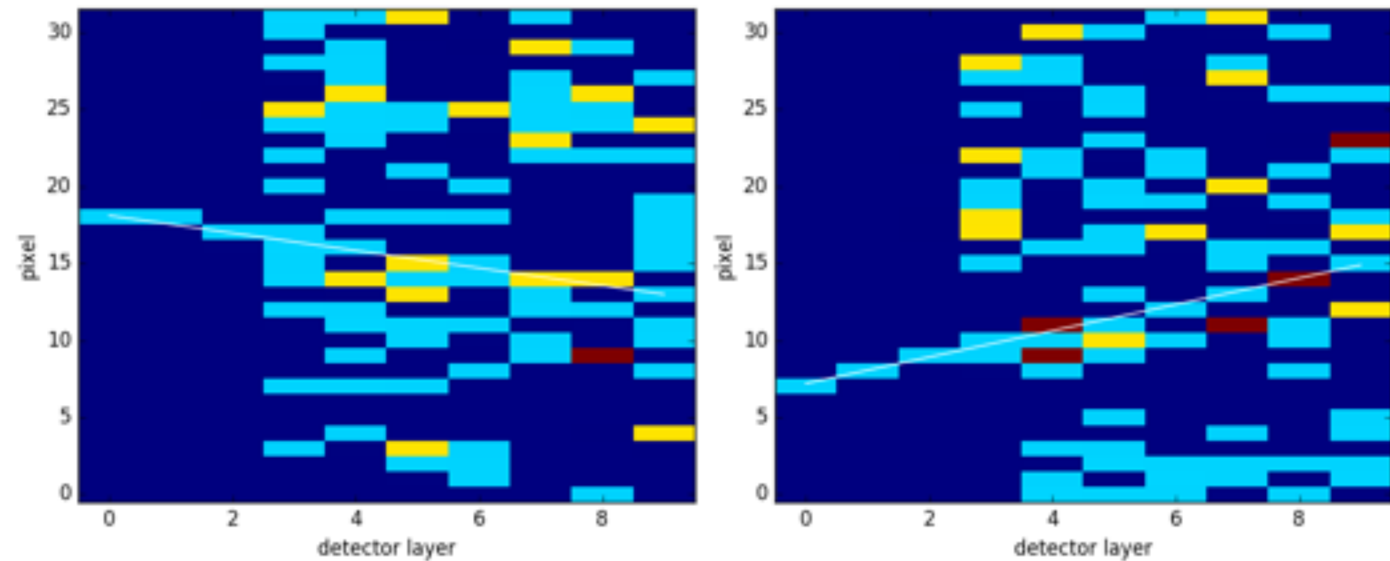


From CTD-WIT 2017
https://doi.org/10.1051/epjconf/201715000003

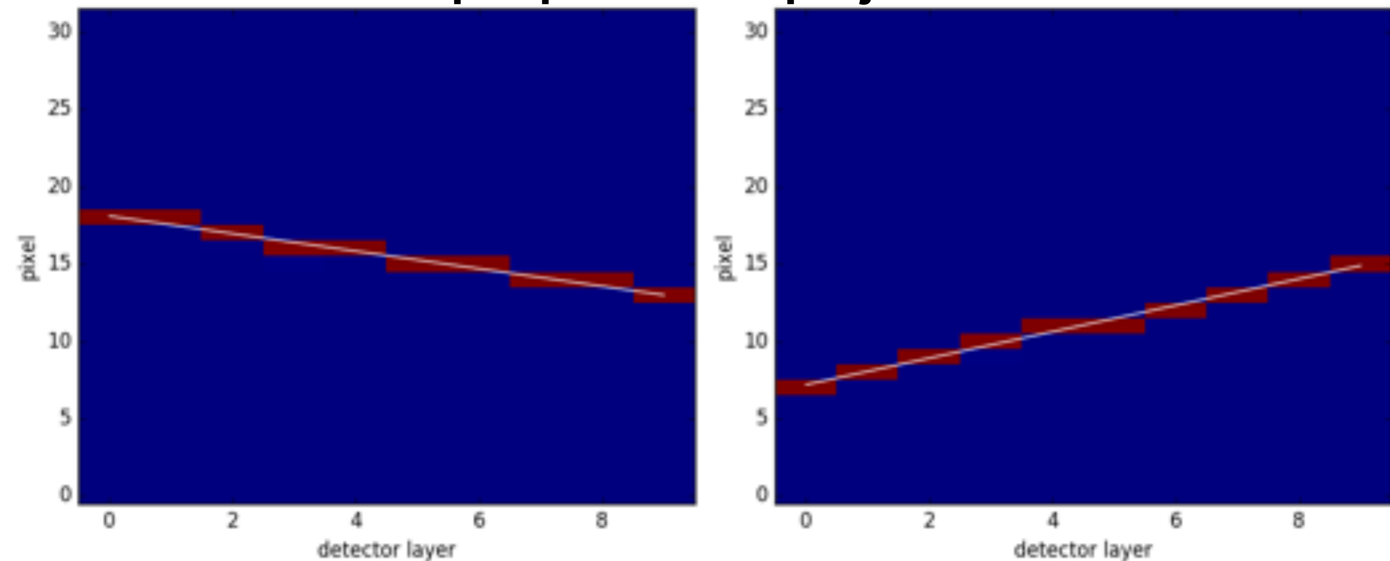
Image segmentation on toy 3D data



Input projections



Output prediction projections

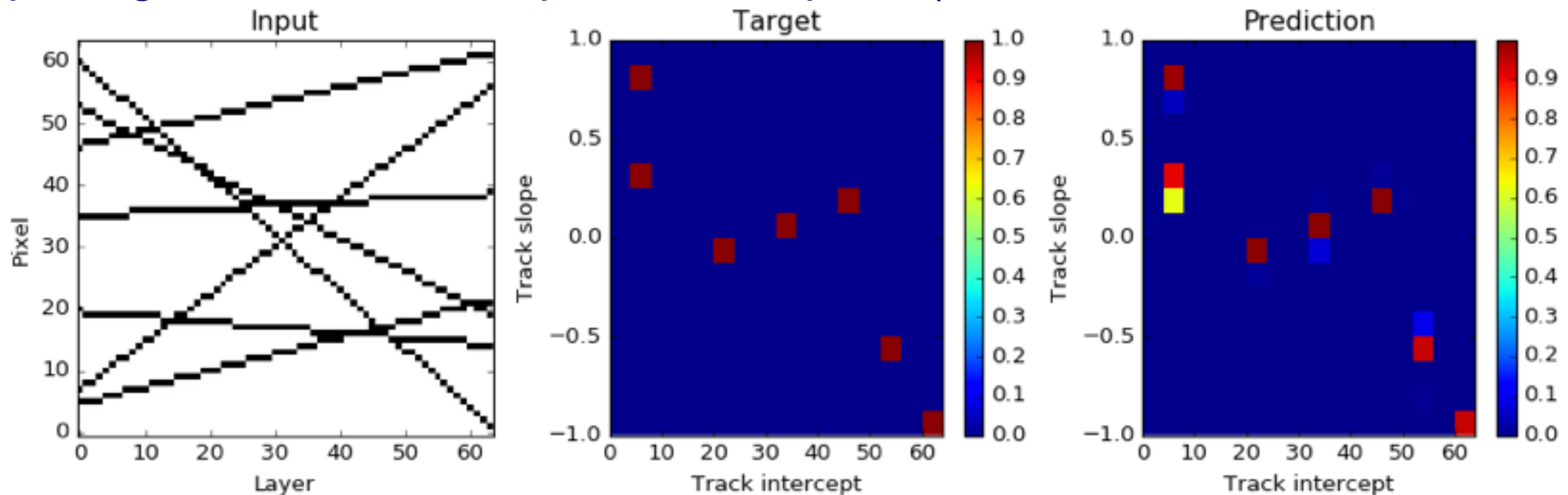


- Simple 3D CNN model
  - 10 layers, 3x3x3 filters
  - no downsampling

From CTD-WIT 2017
https://doi.org/10.1051/epjconf/201715000003

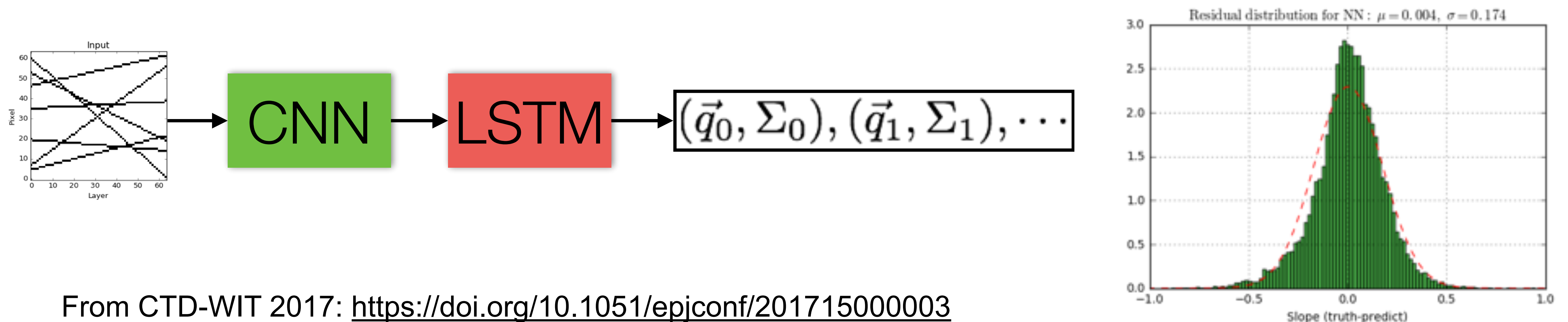
Image labeling/captioning with CNNs, LSTMs

- Map image to binned track parameter space (multi-label classification task)



Shown at DSHEP 2017: https://github.com/HEPTrkX/heptrkx-dshep17/blob/master/cnn/cnn2d_learning.ipynb

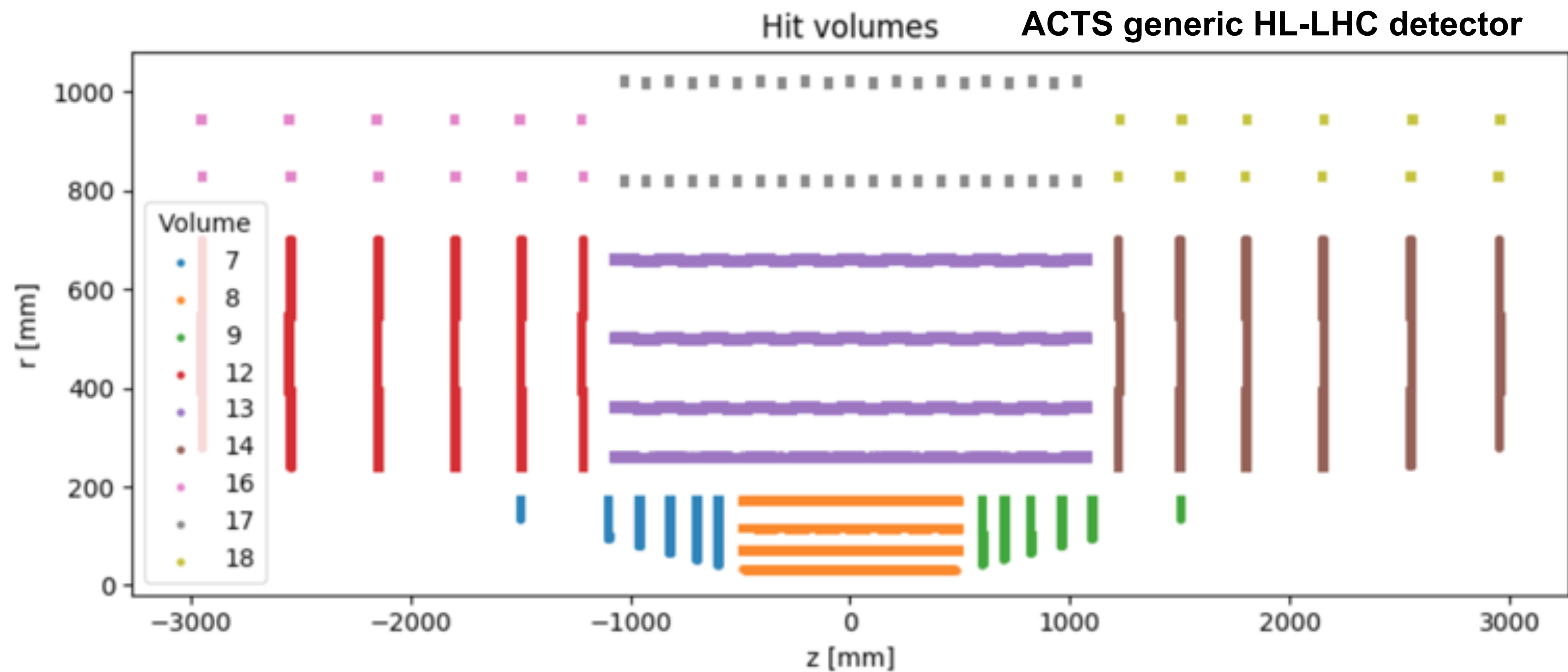
- Produce sequence of discovered track parameters (and uncertainties)



From CTD-WIT 2017: https://doi.org/10.1051/epjconf/201715000003

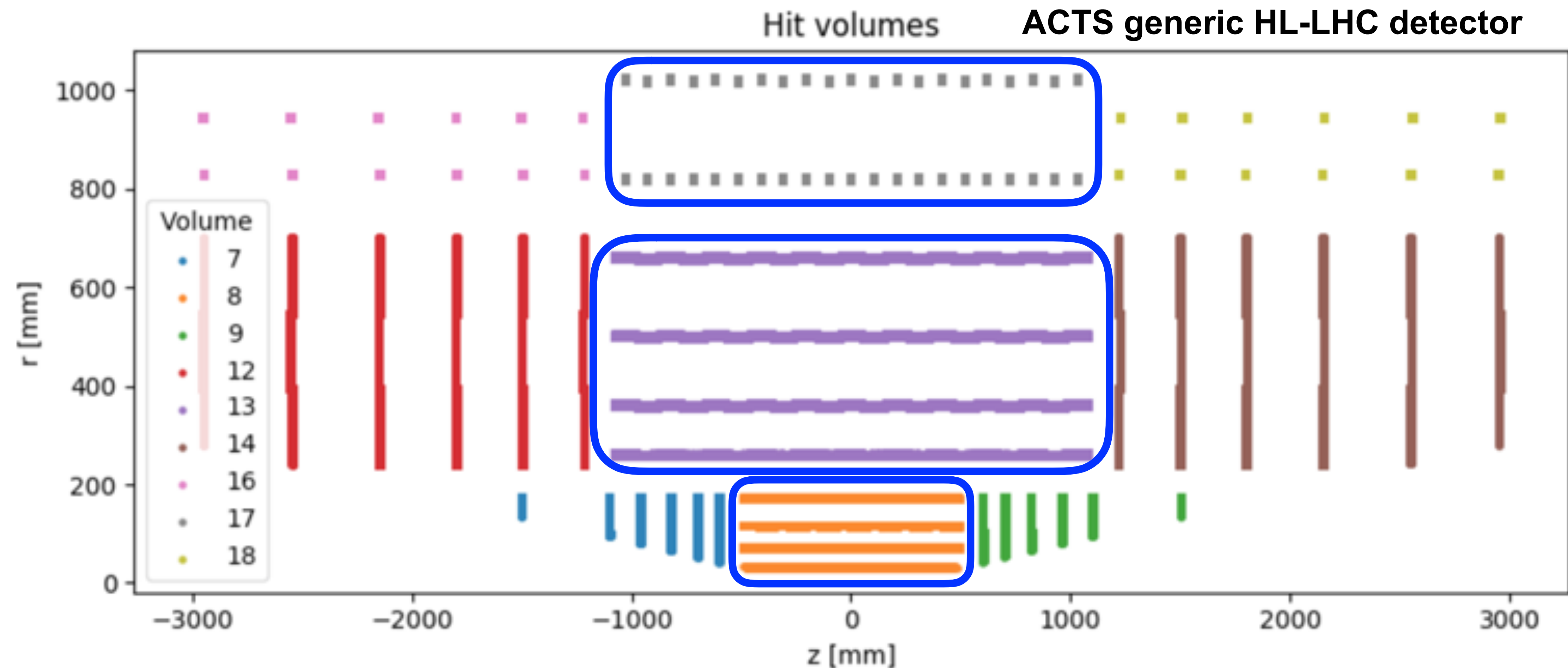
Non-trivial images

- What if the detector is arranged like this?



Non-trivial images

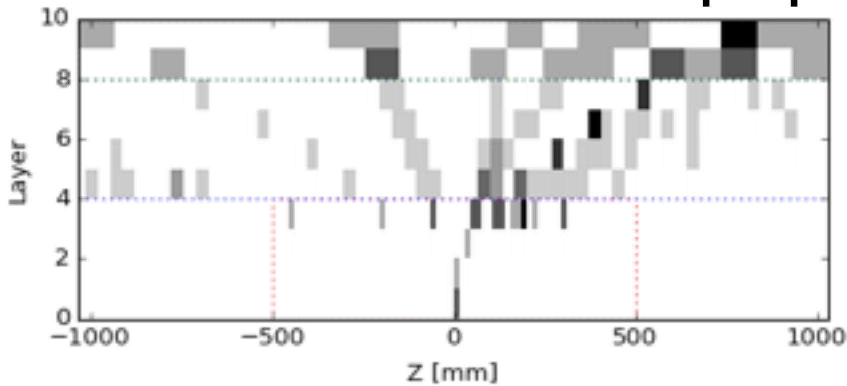
- What if the detector is arranged like this?

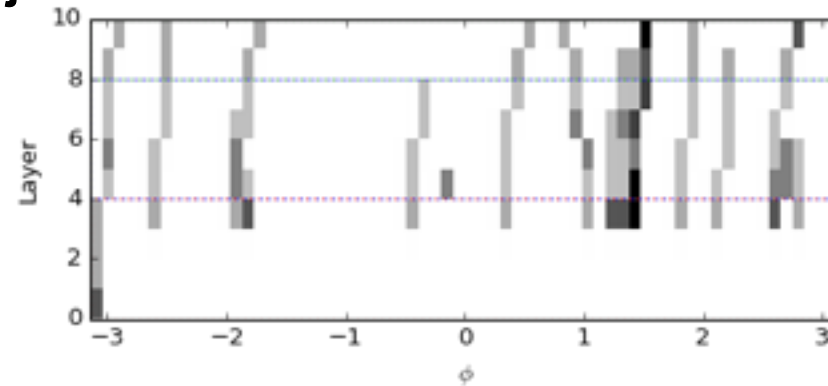


- Construct or bin images in sub-volumes
  - E.g., three volumes in barrel, additional images for endcap disks

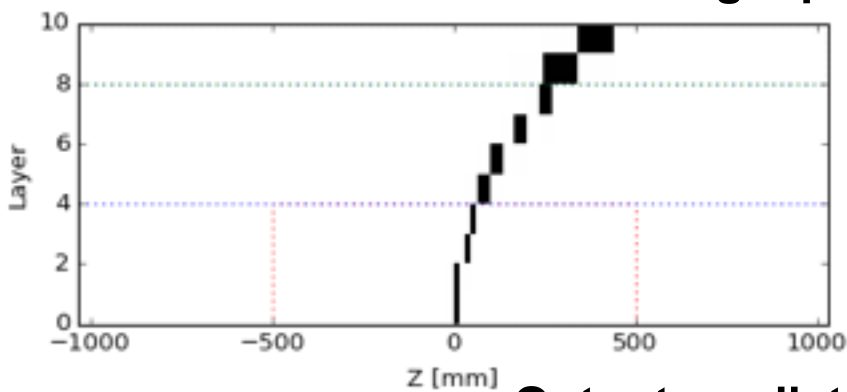
ACTS image segmentation

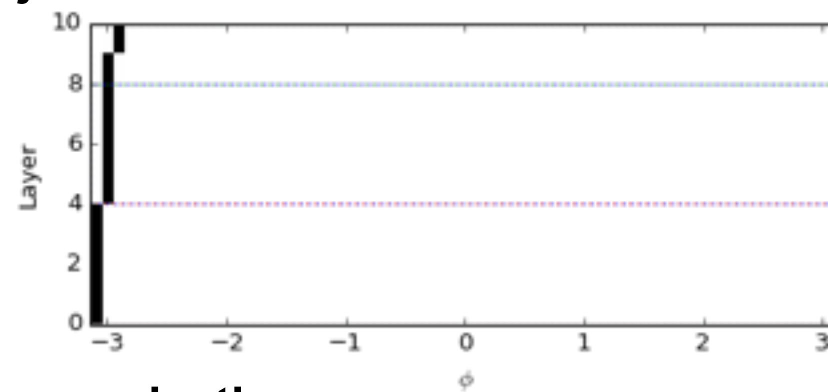
Input projections



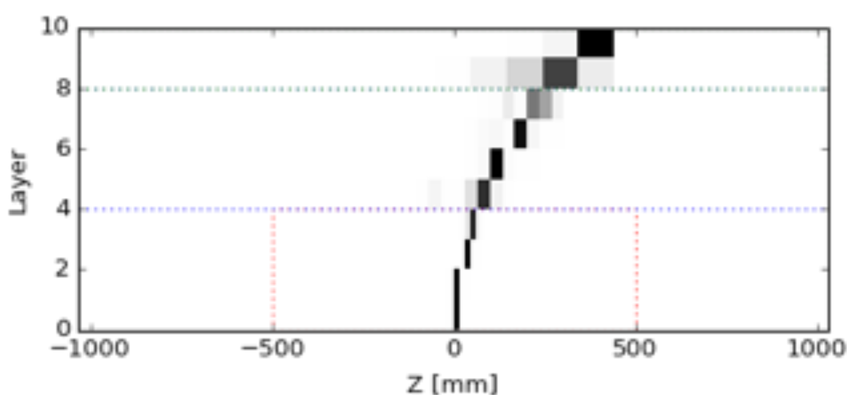


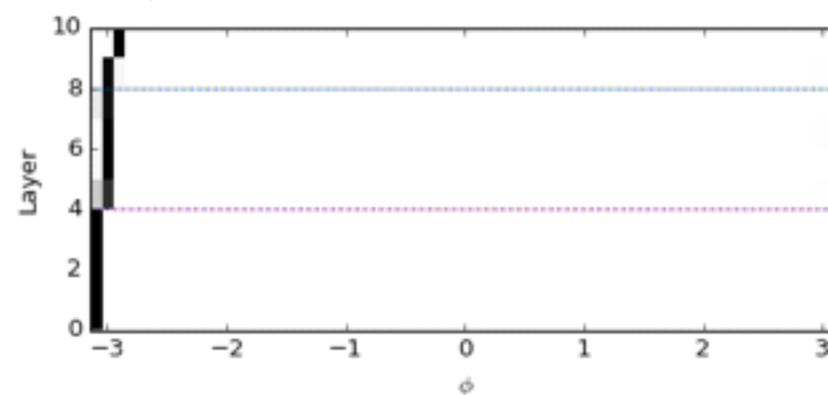
Target projections





Output prediction projections





- LSTM architecture
- seeded single-track finder
  - binary pixel classification
- barrel layers only
- has mediocre performance as-is

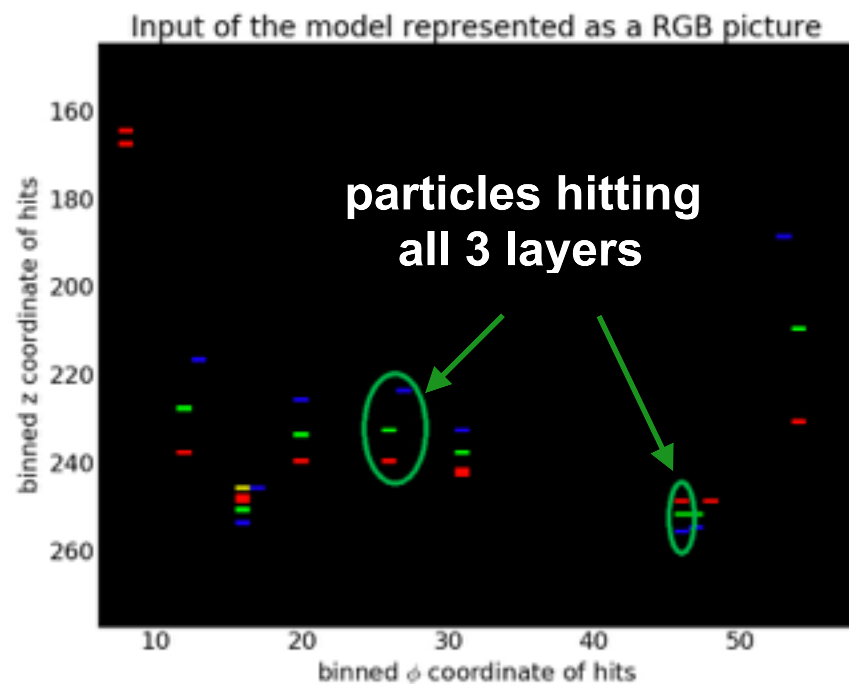
- basic demonstrator of how to extend the toy data case to realistic data
  - but otherwise not a very promising approach as-is

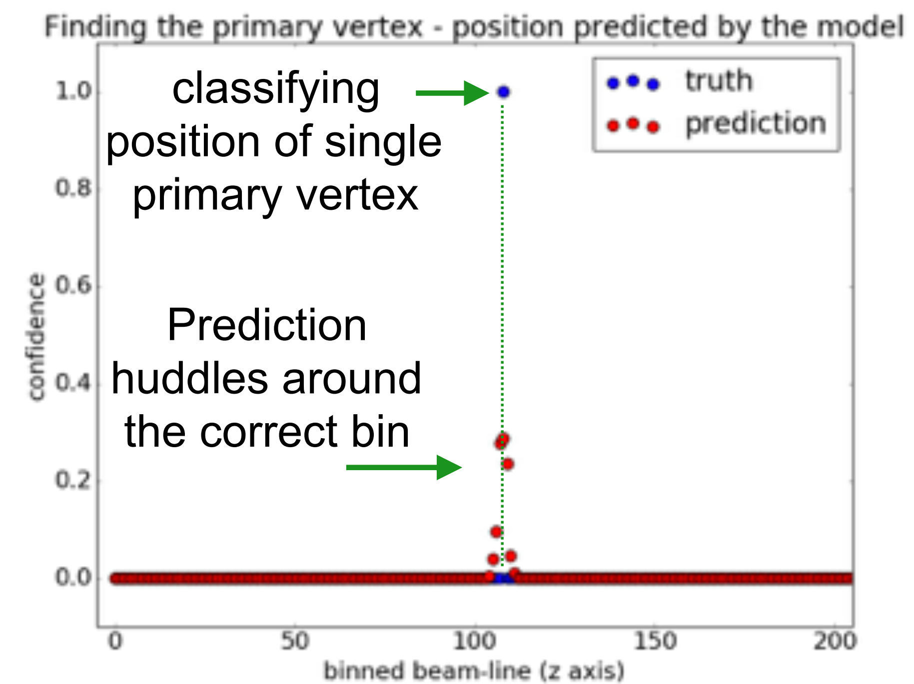
Shown at ACAT 2017

Vertex finding with CNNs

Julien Esseiva's bachelor thesis

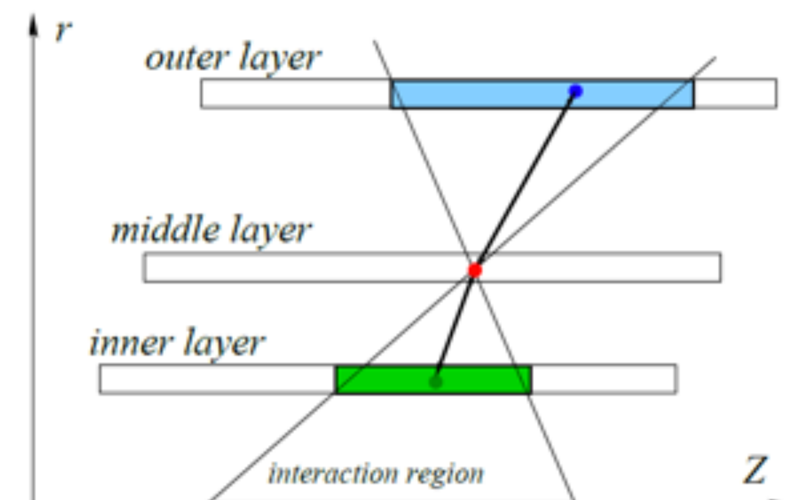
- Represent first 3 barrel layers as an RGB image
- Estimate Z position(s) of the production vertex(es)
  - binned output space; classification or multi-label classification



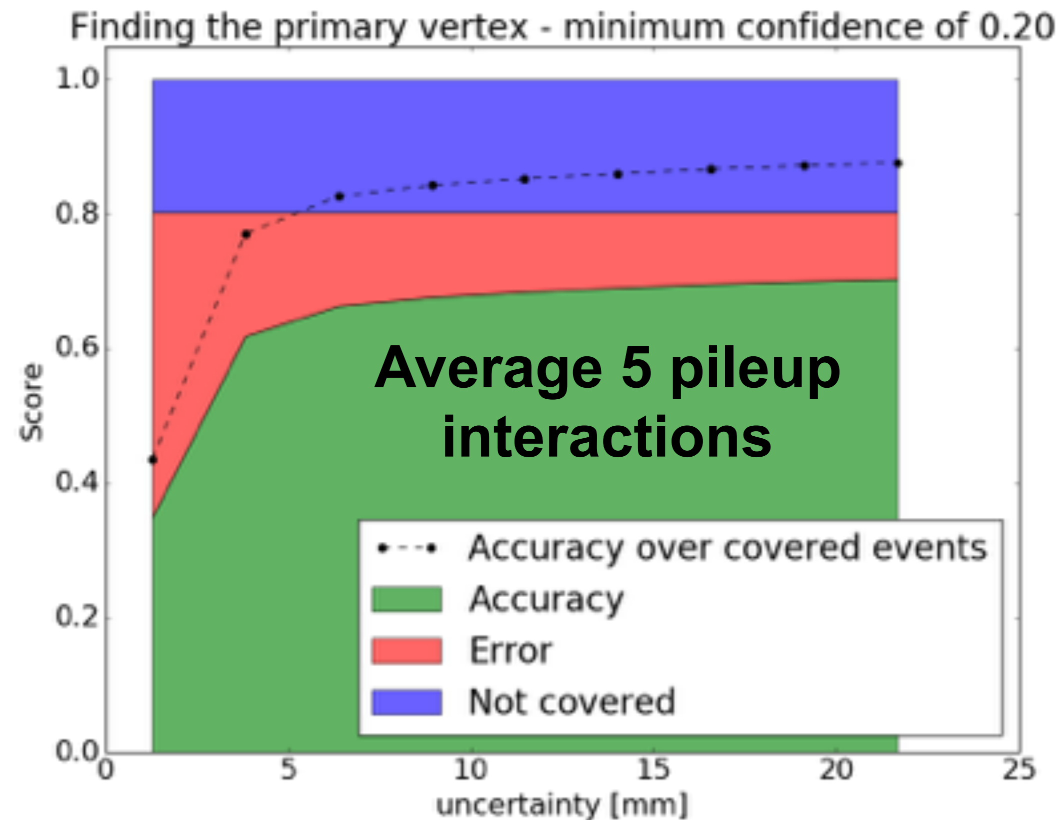


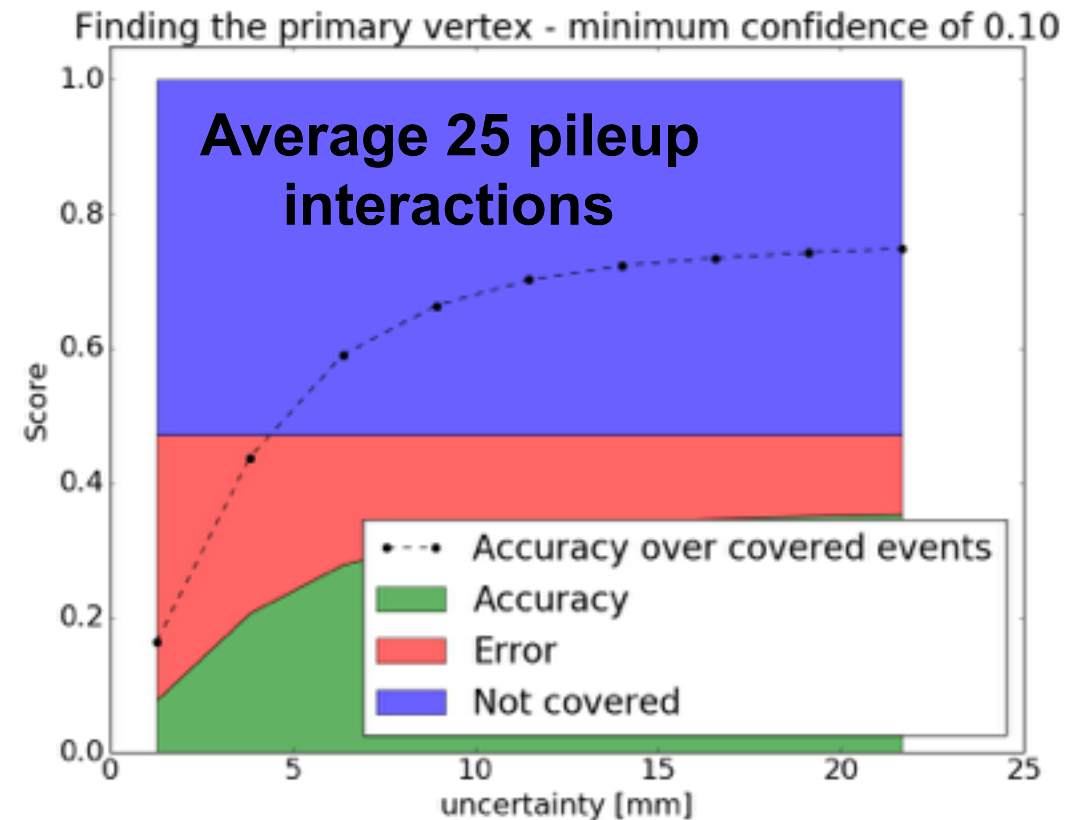
- Can be used to constrain hit combinatorics in hit-triplet (track seed) formation algorithm
  - threshold decision; use only when confident
- Potential online application to speed up track trigger

Shown at ACAT 2017



Vertex finding with CNNs





- Performs ok when finding primary vertex with only handful of pileup vertices
- Mediocre performance at μ=25
- Performs poorly at finding multiple vertices (not shown)
- Probably not good enough yet to be useful
  - Room for improvement

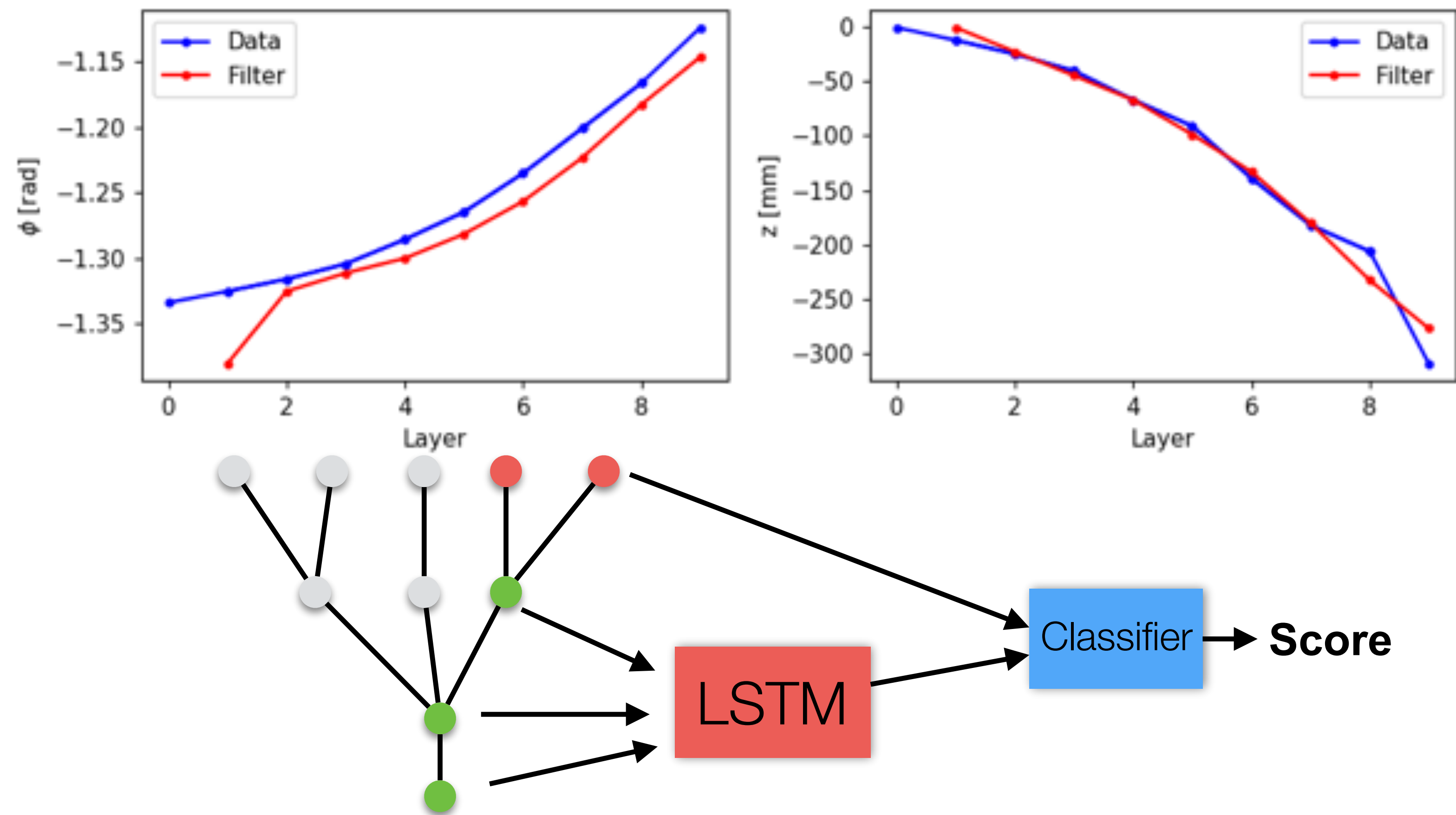
Shown at ACAT 2017

Moving beyond images

- The image formulation brings a number of challenges, particularly when scaling up to realistic data
  - Lossy if binned
  - High dimensionality
  - High sparsity
  - Challenging irregular geometry
- What kinds of ways can you represent spacepoints directly?
  - as a point cloud
  - as a sequence (really a *set*), sorted geometrically
  - as a set of combinatorial search trees
  - as a (directed) graph
- Now things might start to get novel

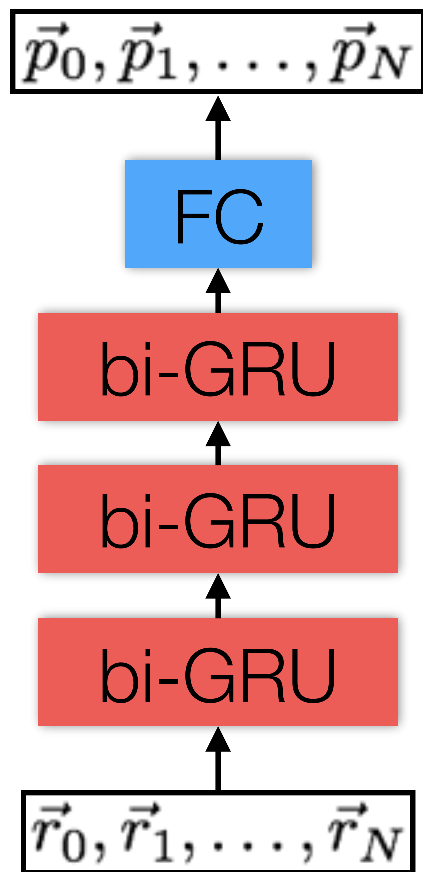
Exploring the tree

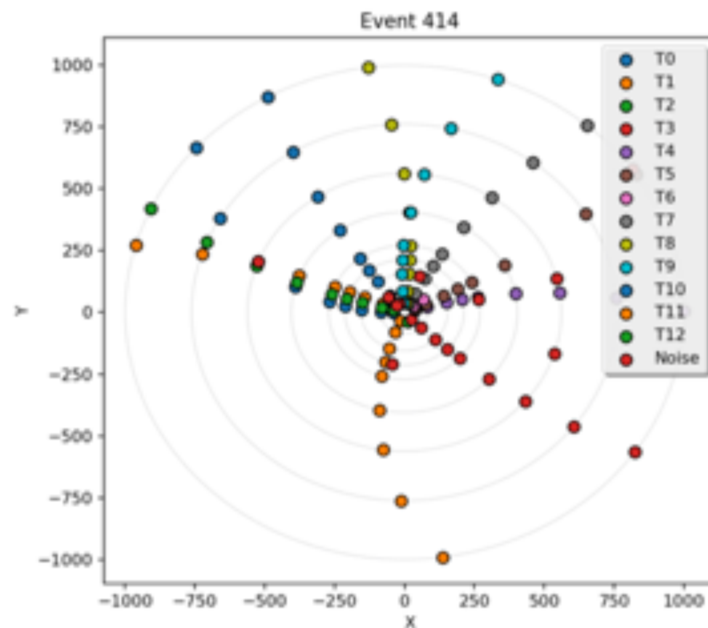
- We've looked at how LSTMs can function as a filter algorithm to learn particle trajectories
- We can put this into a combinatorial tree search to build tracks from seeds
  - Goal: be smarter about choosing hits than Combinatorial Kalman Filter
- Multiple possible approaches to score nodes
  - Predict next-hit location, use guess to score hits
  - Classify track + hit
- Keep top-K candidate nodes and always explore the best one until done

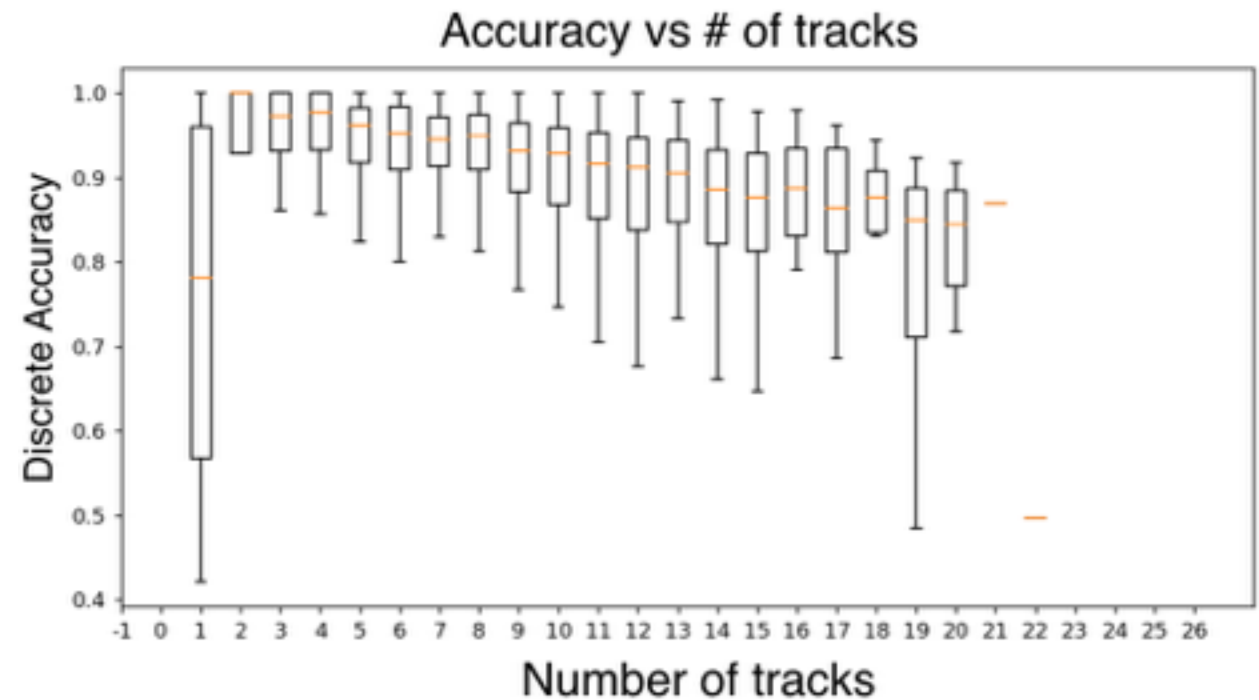


- Starting simple
  - ACTS data with pileup μ=10
  - 3 seed layers
  - Barrel layers only; cleaning up holes, double hits
  - Given correct path so far, train classifier to score 5 closest hits on the next layer
- Shows 100% test set accuracy on this data with fairly simple 2k parameter model
  - Looks promising!
  - Need to train on samples with wrong path as well

Hit sequence to track assignment







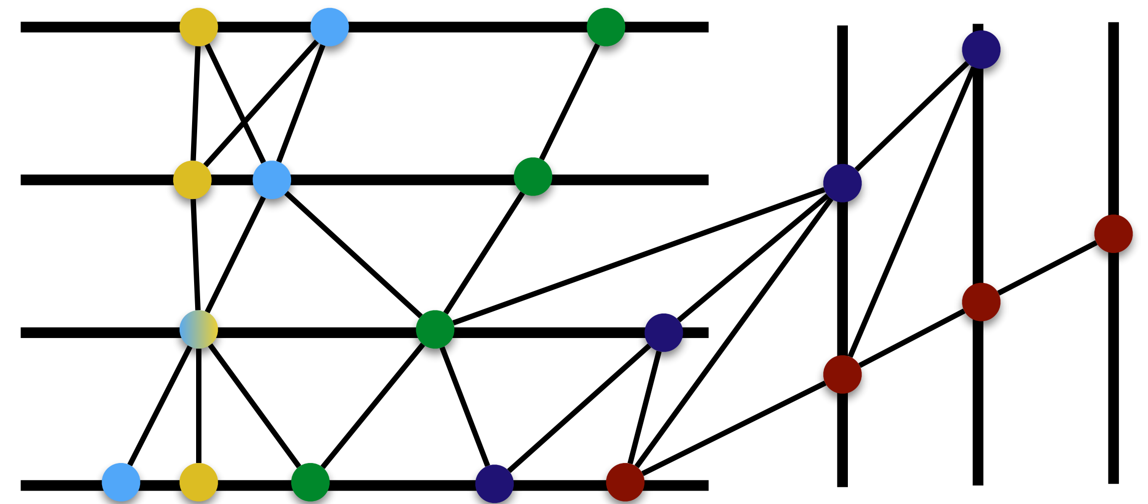
$$\vec{r} = (r, \phi, z)$$
$$\vec{p} = (p_{\text{trk1}}, p_{\text{trk2}}, \ldots, p_{\text{trkMax}})$$

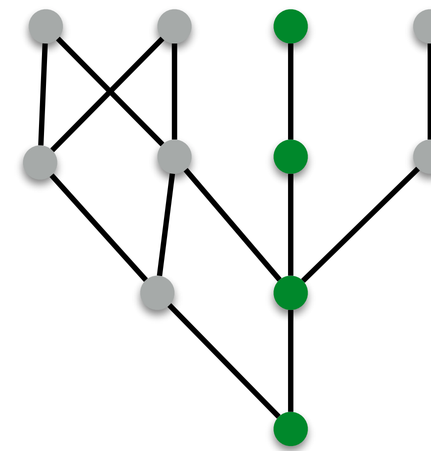
- Sort *all* hits in an event according to position
- Feed hits into a few layers of bi-directional recurrent net (GRU)
- Output is a set of assignment probabilities to track groups
  - Ordering of output track categories is similarly sorted as hits
  - Requires assumed maximum number of tracks
- Assignment matrix is trivially block-identity if tracks never “cross”
  - So the model must focus learning on when to swap assignment order
- Accuracy doesn’t seem to scale well to high occupancy (yet?)

	Track 0	Track 1
Hit 0	1	0
Hit 1	0	1
Hit 2	1	0
Hit 3	0	1

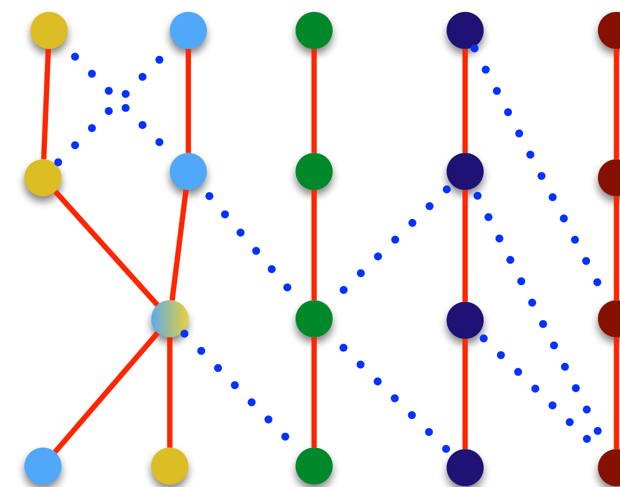
Graph formulation

- Hits on the detector can be arranged in a graph, with edge weights that quantify compatibility
- There has been a fair bit of buzz in the ML community on methods for deep learning on such structured data
  - http://geometricdeeplearning.com/
  - Geometric deep learning: going beyond Euclidean data
  - Neural Message Passing for Quantum Chemistry
  - Semi-Supervised Classification with Graph Convolutional Networks
- There are a variety of possible applications
  - hit classification (binary or multi)
  - hit segment classification
  - hit clustering





Single track hit classification



Segment classification

Graph neural networks

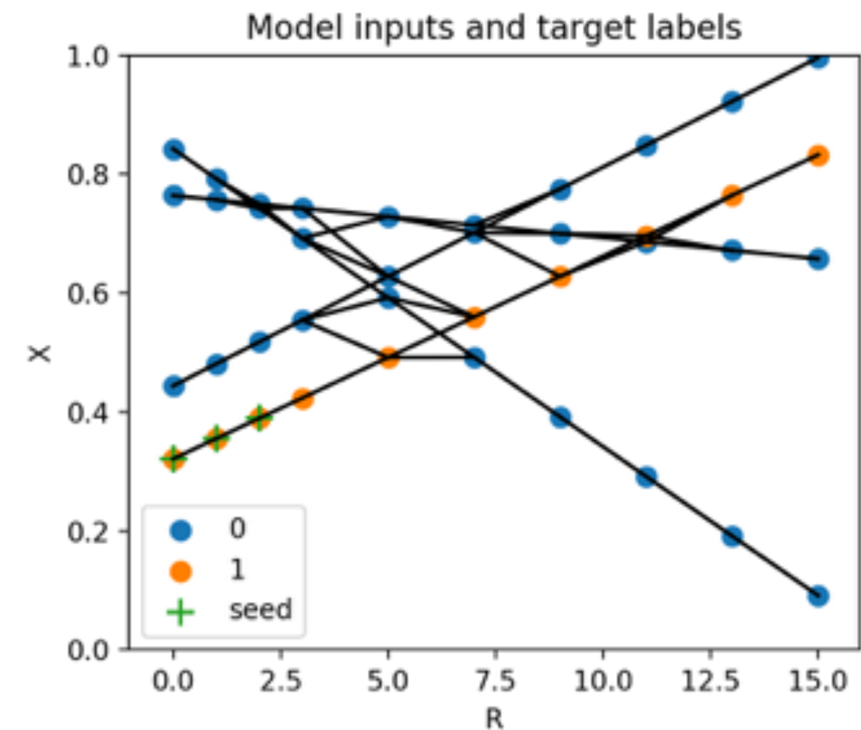
- There are several approaches to define architectures
  - Laplacian spectral graph convolution (no time to discuss in detail)
    - and several simplified parametrized forms
  - Spatial kernel methods
- But the common idea is that a “patch” operation calculates new features for a node by doing a weighted averaging over its neighbor’s features
- A simple graph NN “layer” might look like:

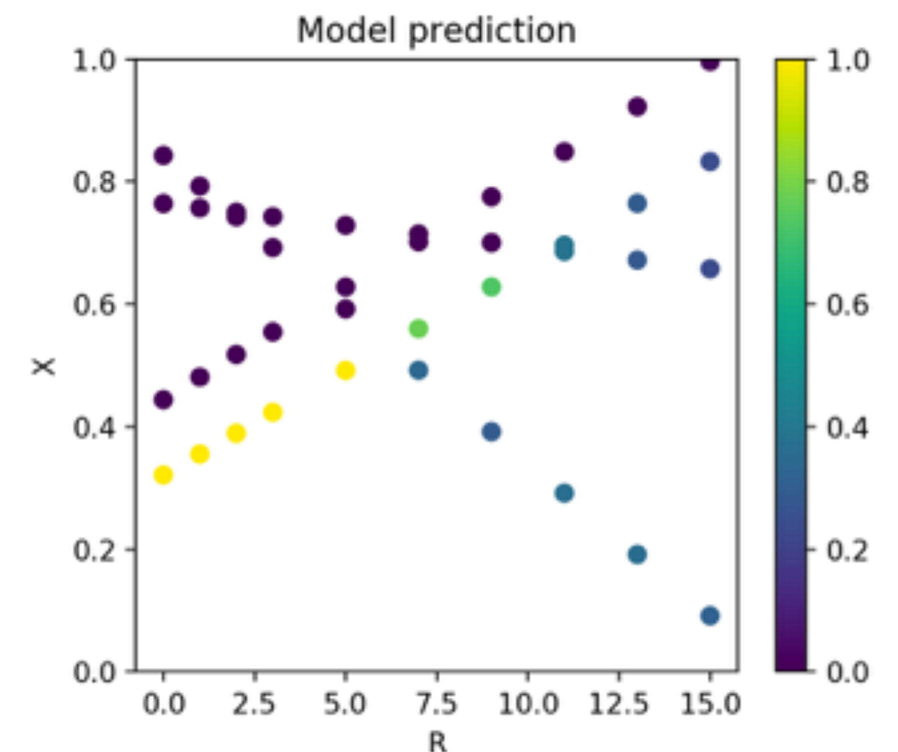
$$X' = \sigma(AXW + B)$$

  - A is N x N adjacency (“similarity”) matrix, possibly normalized in some way
  - X is N x D node features (D features per node)
  - W is D x D’ learned weight matrix
  - B is N x D’ learned bias matrix
- How to get rich
  - Downsampling via graph coarsening
  - Residual/shortcut connections
  - *Learnable* adjacency, e.g. parametrized by some kernel function
  - Multiple adjacencies for modeling distinct types of node relations (edge features)
  - Alternate between layers that calculate edge features and node features

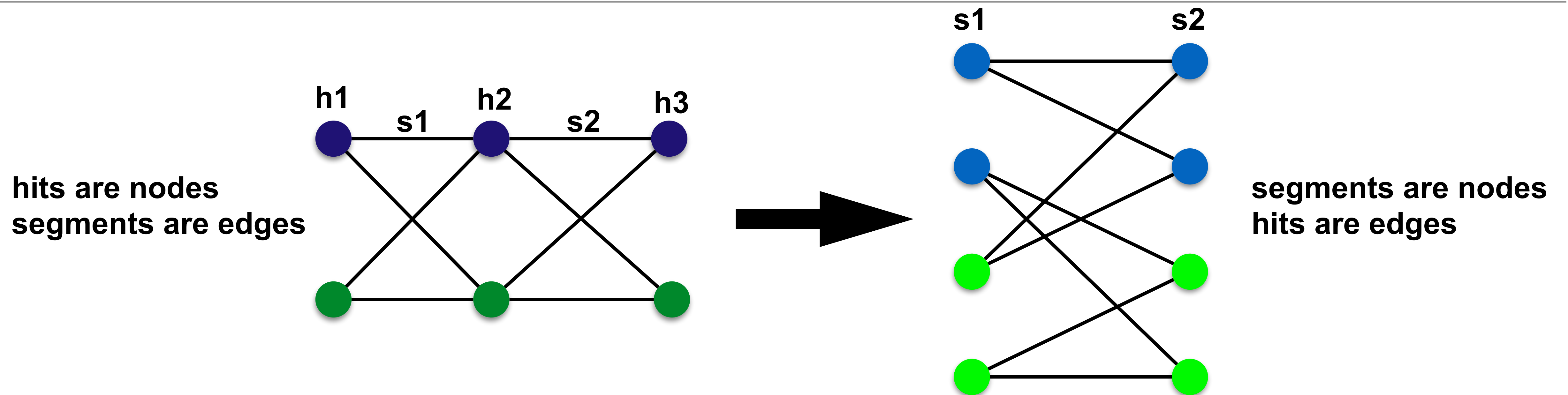
Hit classification with GNNs

- Binary classification of hits to find one seeded target track with 2D toy data
- Inputs
  - Node features: (r, x, is_seed)
  - Edge weights: 1 if hits on adjacent layers and segment defines an allowed line (contained in detector), otherwise 0
- Architecture
  - Graph layers of the form: $X' = \sigma(XW_1 + D^{-1}AXW_2 + B)$
    - D is the diagonal degree matrix (normalizes A), σ is a ReLU
  - Input features also stacked onto every hidden graph layer
- Performance not very good :(
  - Binary similarity not capturing any useful information
  - Model needs help at distinguishing neighbors





Alternative segment-graph formulation

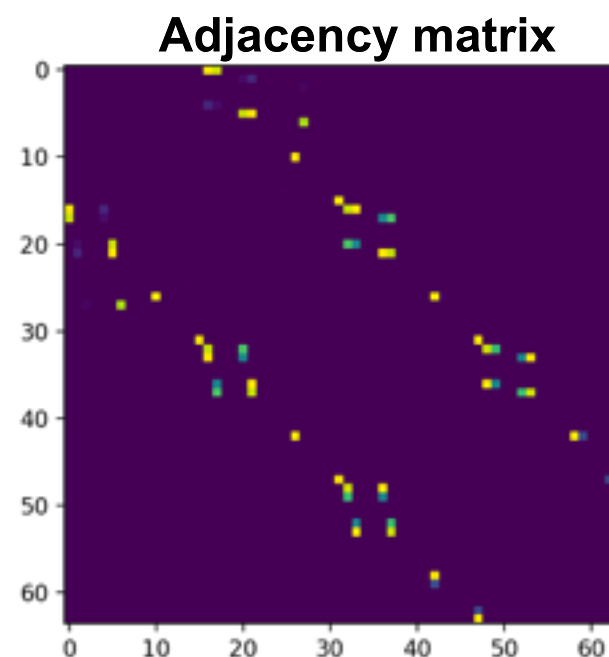


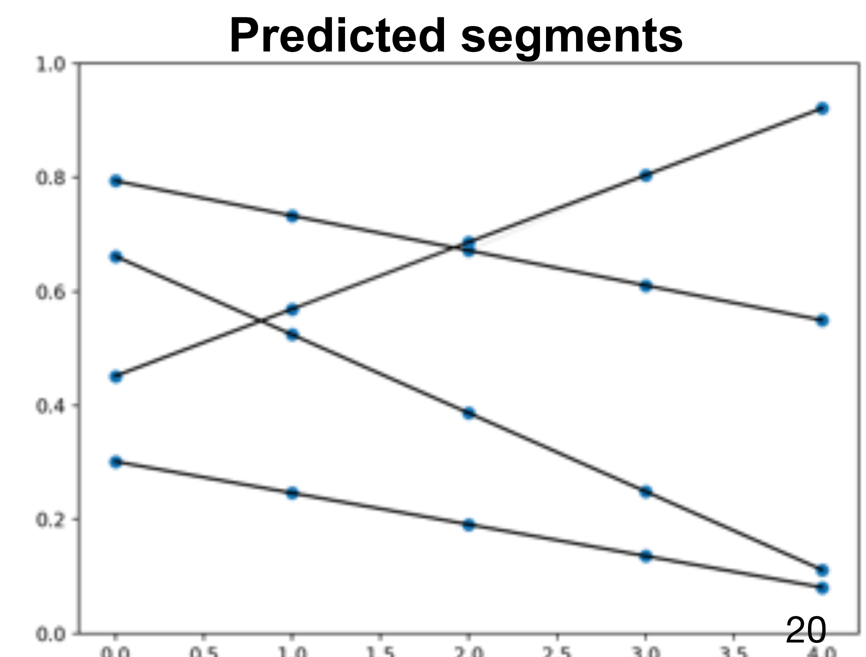
- We can build a “dual” graph which swaps the nodes and edges
- For the tracking problem, then, it’s a graph that relates and learns on the *segments* between hits
- Segments connect to other segments through hits, and we can define similarity in terms of the compatibility of the segments
  - e.g., the change in direction

Segment classification with GNNs

- Binary classification of segments to connect adjacent hits
  - In this case I considered all segments between adjacent layer hits
- Inputs
  - Node features: (r1, r2, x1, x2, slope)
  - Edge weights: Gaus(delta-slope, 0.05) if segments are connected through adjacent layer, otherwise 0
- Architecture similar to before, but I didn't shortcut the inputs to hidden layers
- Performance is great if I make the gaussian kernel sharp enough
  - The correct adjacent segment needs to dominate
  - 96% accuracy
  - 2k parameters

With a sharp enough adjacency, though, the problem is trivial!



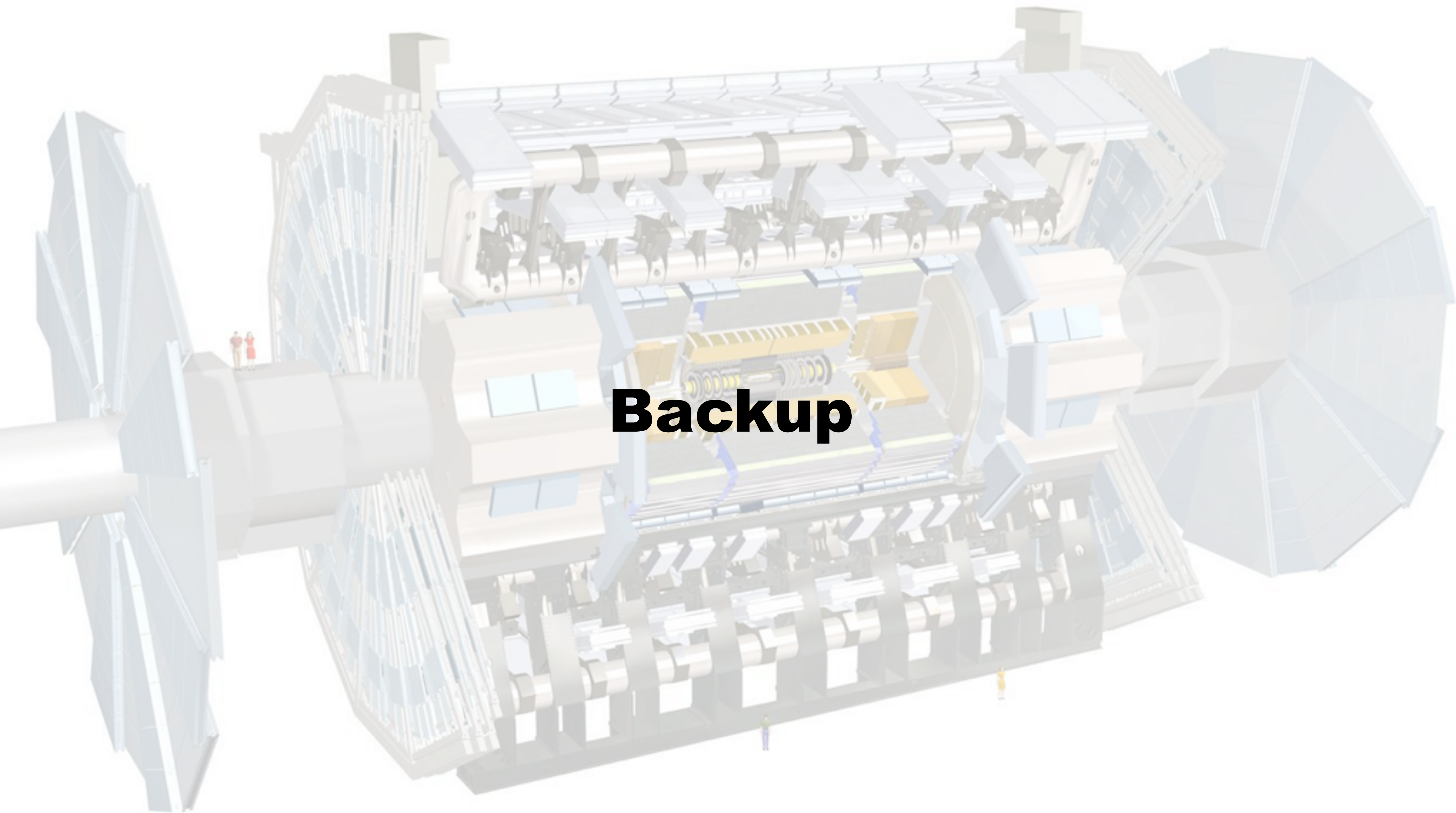


Reflecting on GNNs so far

- Some of this looks promising, and graph NNs may be a useful approach
  - but I wouldn't say these are "good" results (yet)
- Limitations are becoming apparent
  - Basic graph-convolutional architectures do a weighted average over neighbors
    - Isotropic kernels using one similarity measure
  - So it's hard to capture the *specific* relationship between one hit and another
- How might they be improved?
  - Anisotropic, learned kernels
  - Alternating graph representations: hits, segments
  - ???
- This has only been a brief first look. There's a fair bit of investigating yet to do.

Conclusion

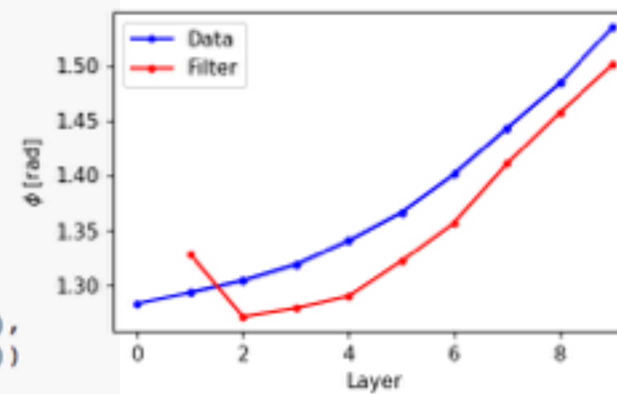
- HEP.TrkX is chugging along trying crazy things
    - Moving away from image-like construction, trying more novel formulations
    - We're working on getting focused, however
        - Working towards common targets
        - Porting ideas to realistic ACTS data
    - We currently suffer from low manpower
        - only 2-3 “active” researchers as far as I can tell
- Areas of work I still think are promising
    - Image-based techniques localized to detector sections
    - ML-assisted combinatorial tree search
    - Graph-based neural networks

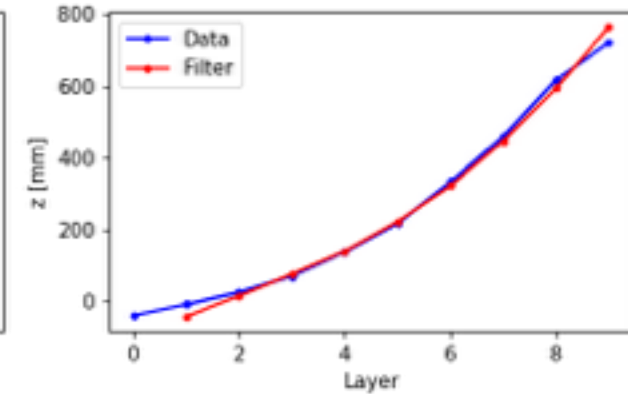


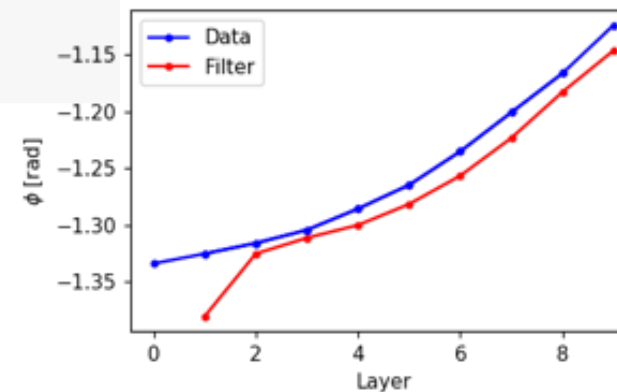
Backup

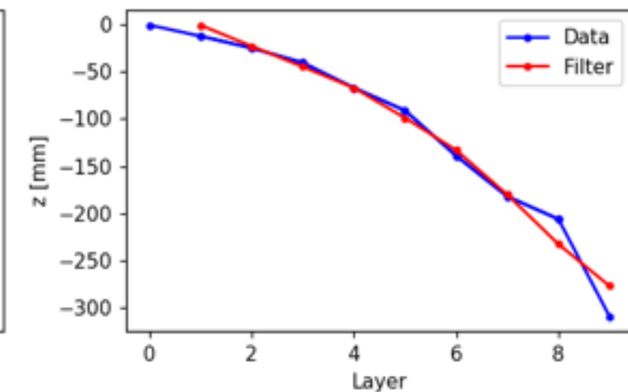
Track fitting with LSTM

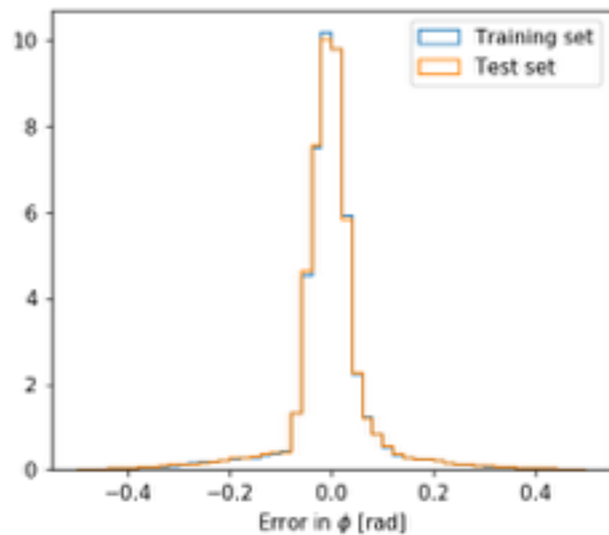
```python
class TrackFilterer(nn.Module):

    def __init__(self, input_dim=3, hidden_dim=5, output_dim=2, n_lstm_layers=1):
        super(TrackFilterer, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, n_lstm_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        input_size = x.size()
        # Initialize the lstm hidden state
        h = (create_tensor_var(self.lstm.num_layers, input_size[0], self.lstm.hidden_size),
             create_tensor_var(self.lstm.num_layers, input_size[0], self.lstm.hidden_size))
        x, h = self.lstm(x, h)
        # Flatten layer axis into batch axis so FC applies independently across layers.
        x = (self.fc(x.contiguous().view(-1, x.size(-1)))
             .view(input_size[0], input_size[1], -1))
        return x
```

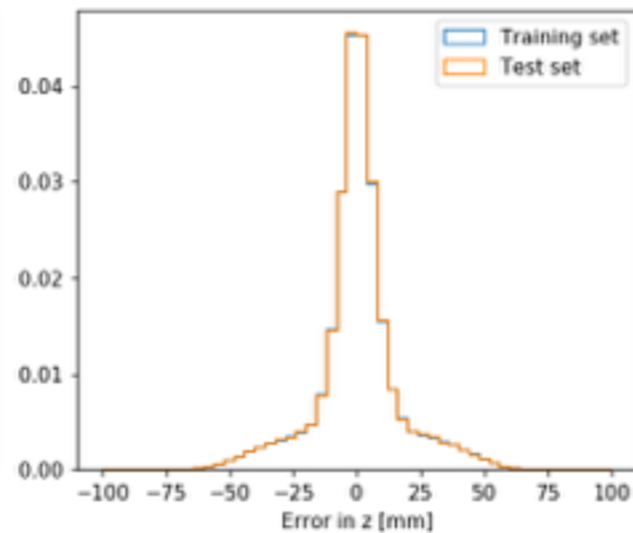


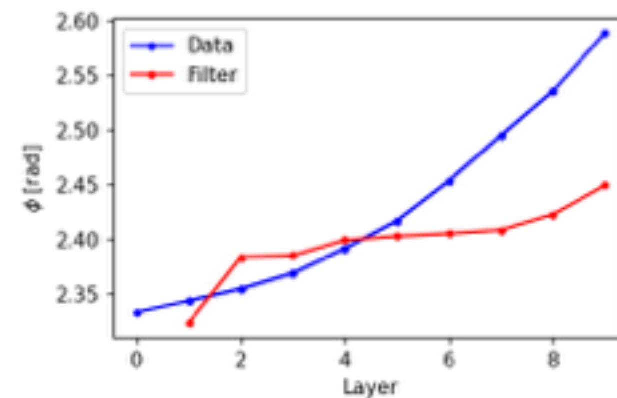


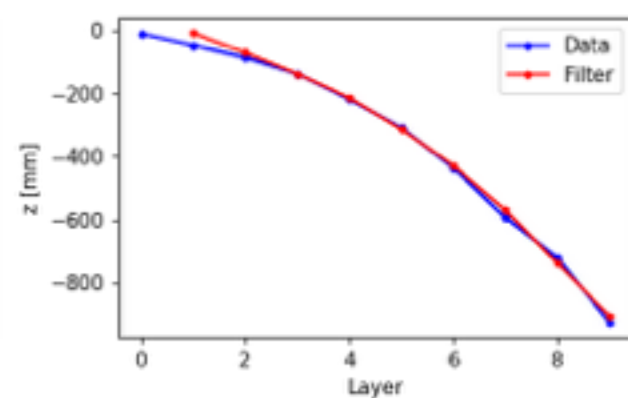












GNNs with PyTorch

```python
class GraphConvSelfInt(nn.Module):
    """
    A graph convolution module with separate explicit self-interaction terms.

    This module takes an input tensor of node features X and adjancency
    matrix A and applies a linear transformation of the form
        X*W1 + A*X*W2 + b
    where (W1, W2) and b are learned weights and biases.
    """
    def __init__(self, input_dim, output_dim):
        super(GraphConvSelfInt, self).__init__()
        self.node_mod = nn.Linear(input_dim, output_dim)
        self.neighbor_mod = nn.Linear(input_dim, output_dim, bias=False)

    def forward(self, x, a):
        node_term = repeat_module(self.node_mod, x)
        neighbor_term = repeat_module(self.neighbor_mod, torch.matmul(a, x))
        return node_term + neighbor_term
```

GNNs with PyTorch

```
class GCNBinaryClassifier(nn.Module):
    """
    A simple graph-convolutional network for binary classification of nodes.

    This model applies a feature extractor to each node,
    followed by a number of graph conv layers,
    followed by a node classifier head.
    """

    def __init__(self, input_dim, hidden_dims, gc_type=GraphConvSelfInt):
        super(GCNBinaryClassifier, self).__init__()
        # Feature extractor layer
        self.feature_extractor = nn.Linear(input_dim, hidden_dims[0])
        # Graph convolution layers
        n_gc_layers = len(hidden_dims) - 1
        self.gc_layers = nn.ModuleList([gc_type(hidden_dims[i], hidden_dims[i+1])
                                        for i in range(n_gc_layers)])
        # Node classifier
        self.classifier = nn.Linear(hidden_dims[-1], 1)

    def forward(self, x, a):
        # Apply feature extraction layer
        x = F.relu(repeat_module(self.feature_extractor, x))
        # Apply graph conv layers
        for gc in self.gc_layers:
            x = F.relu(gc(x, a))
        # Apply node classifier
        return repeat_module(self.classifier, x).squeeze(-1)
```