CNN solutions for Pixel-Level ID

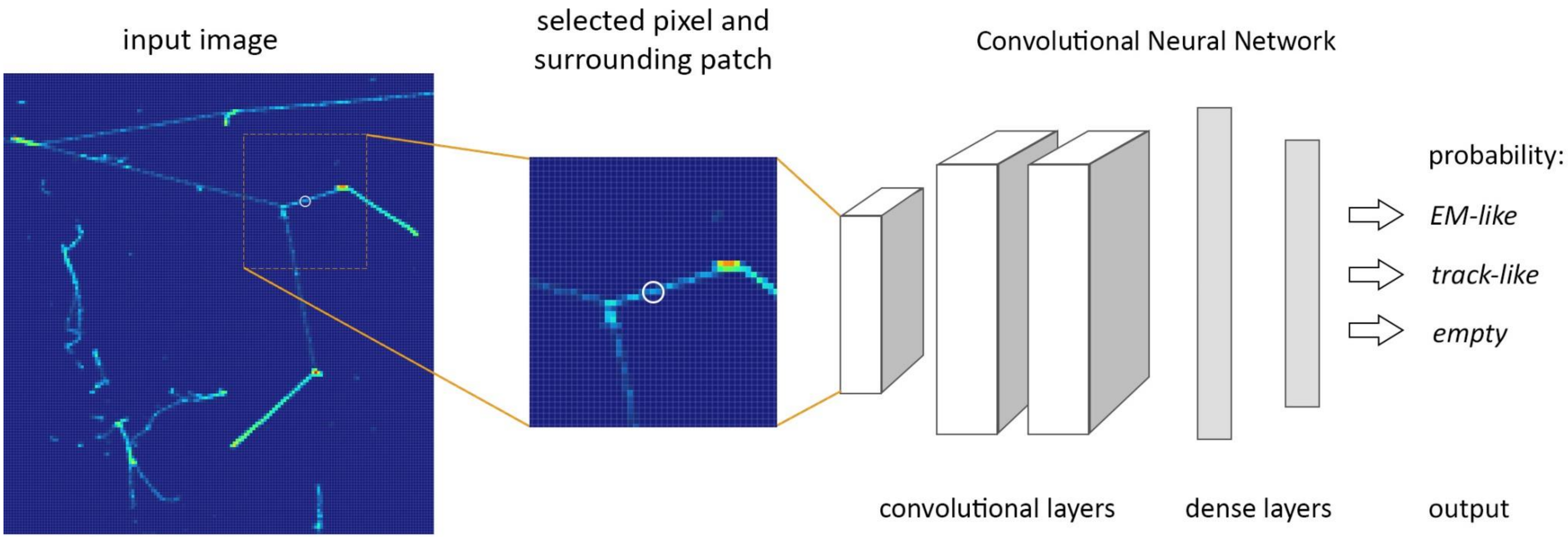
Aaron Higuera (UH, US)
Piotr Płoński (WUT, PL)
Aidan Reynolds (Oxford, UK)
Andrea Scarpelli (APC, France)
Daniel Smith (BU, US)
Dorota Stefan (CERN/NCBJ, CH/PL)
Robert Sulej (FNAL/NCBJ, CH/PL/US…)
Leigh Whitehead (CERN, CH)

Support on Tensorflow from Lynn Garen and LArSoft team

Outline

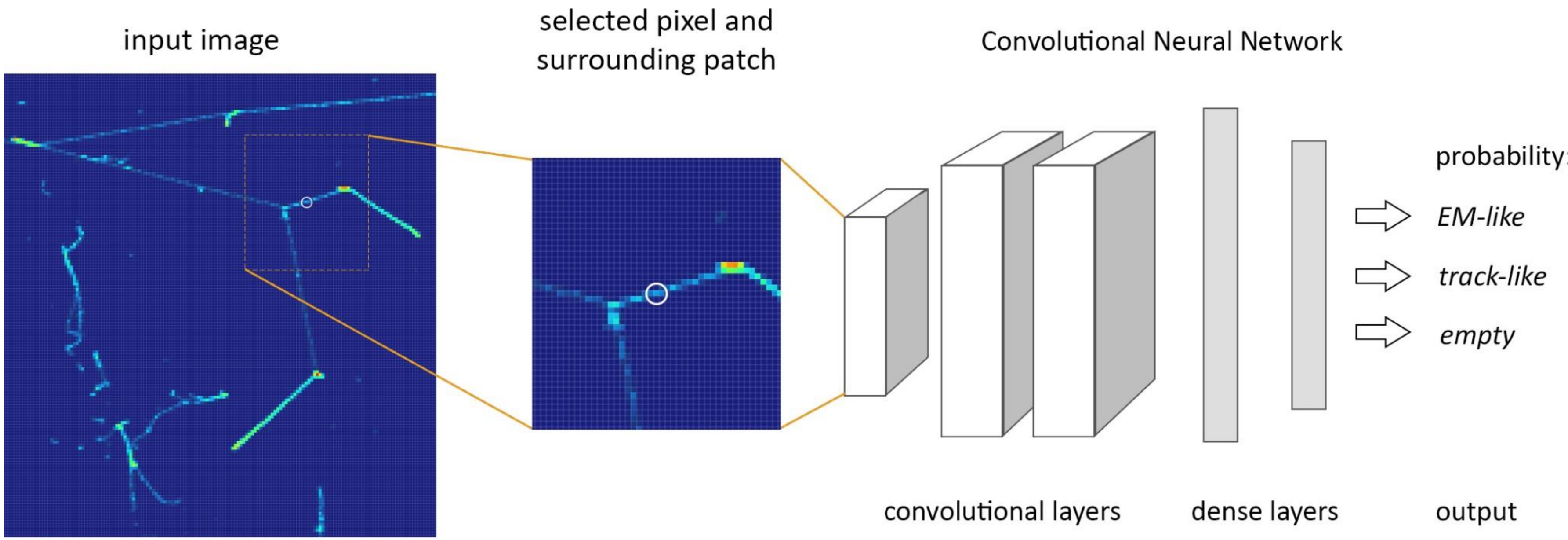
- what we are doing
- how, and what could be gained from common tools
- path forward

Pixel-level activity classification: basic idea



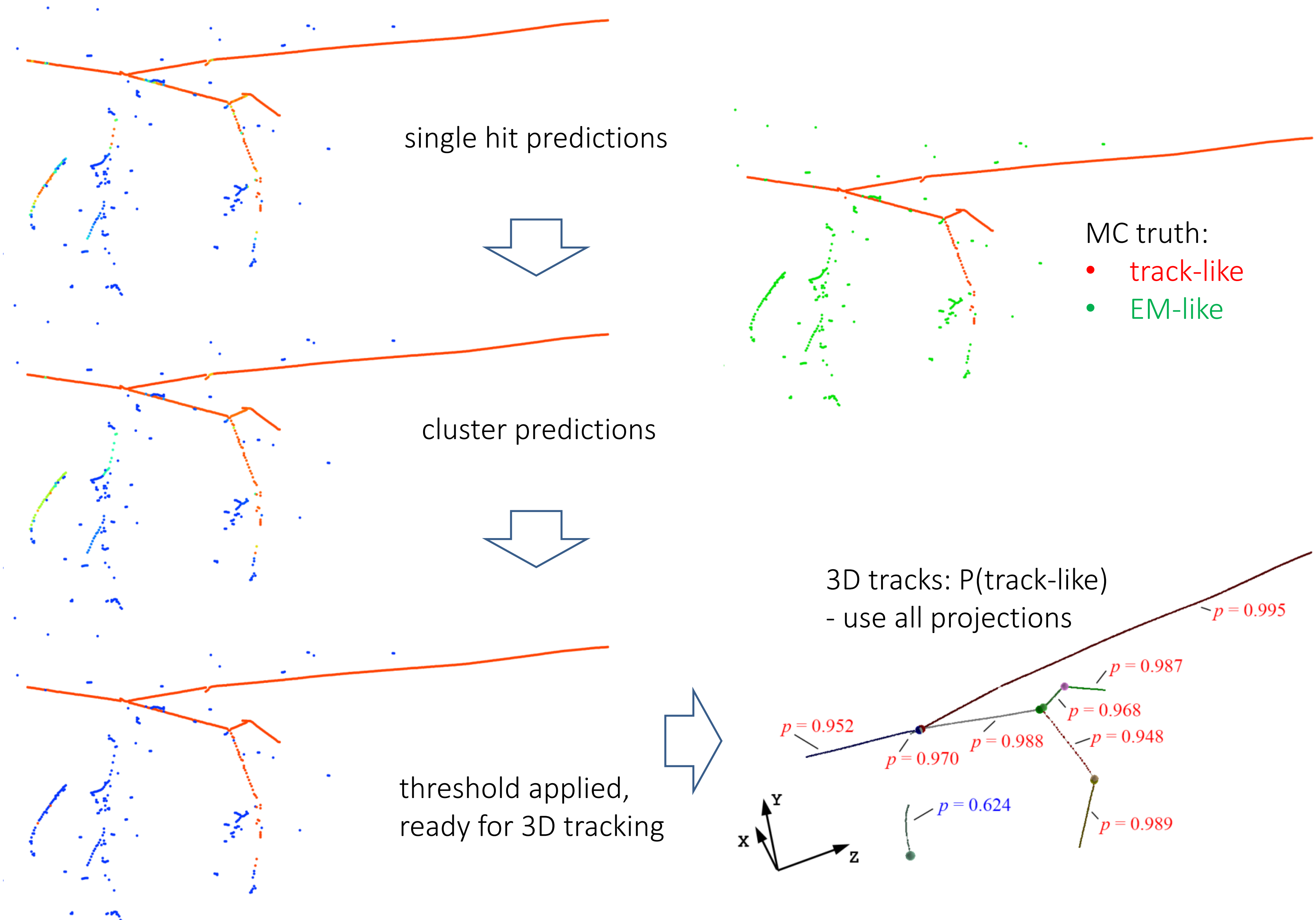
- **Identify low-level features, independent from long range / full event context**
  - EM-like / track-like discrimination
  - Michel electron activity identification
  - Vertex identification & classification (interaction / decay ID, gamma conversion, …)
- Support standard reconstruction or additional input to higher level ML
- Several direct applications to physics & calibration
- Use raw ADC: no hit-related inefficiencies

Pixel-level activity classification: low level features



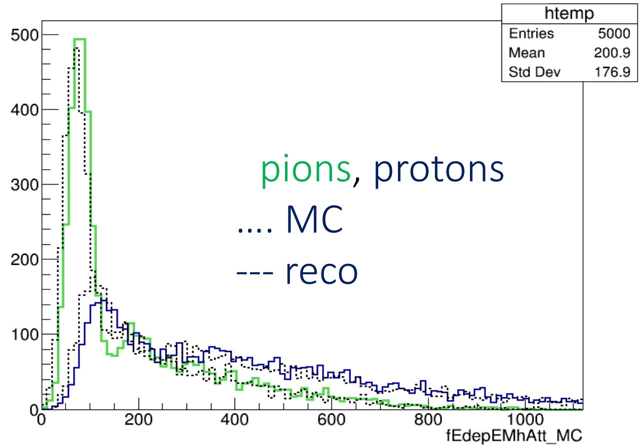
- …independent from long range / full event context
  - limit the patch rage to the minimum: avoid effects of <u>incorrect sim of inetactions</u>
  - there is still ionization shape (somewhat handled with drift downsampling)
    - → accurate detector sim (it was for LArIAT, may not be that easy for 3x1x1)
    - → or explore more GAN, VAE, …
  - noise is also a factor, however easier to simulate or take from data

EM/track in ProtoDUNE: CNN combined with reconstruction

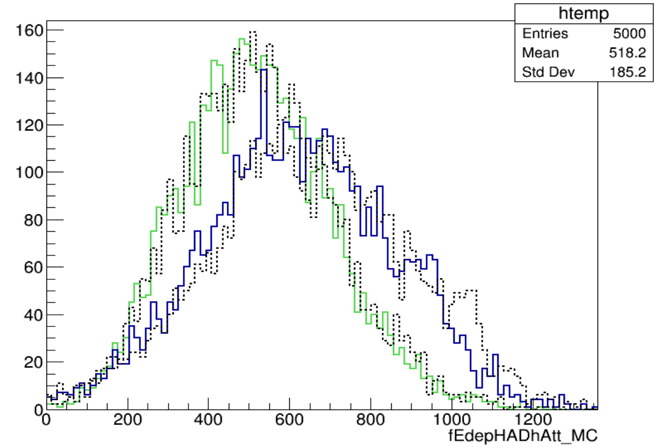


EM/track in ProtoDUNE: CNN combined with reconstruction

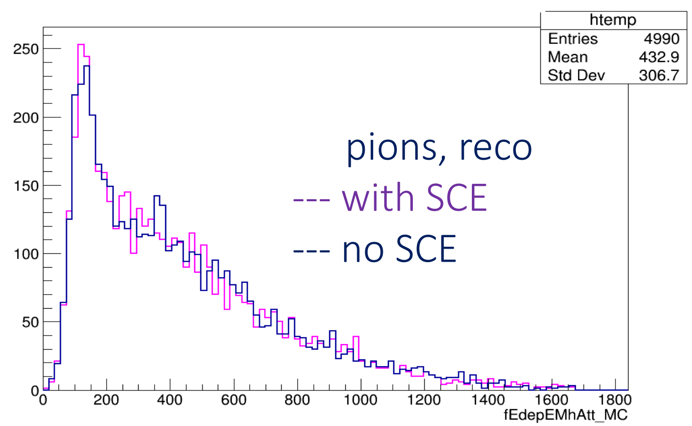
EM component



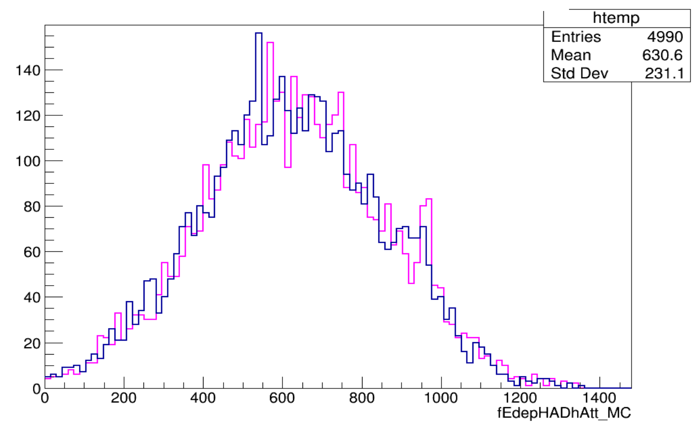
Hadronic component



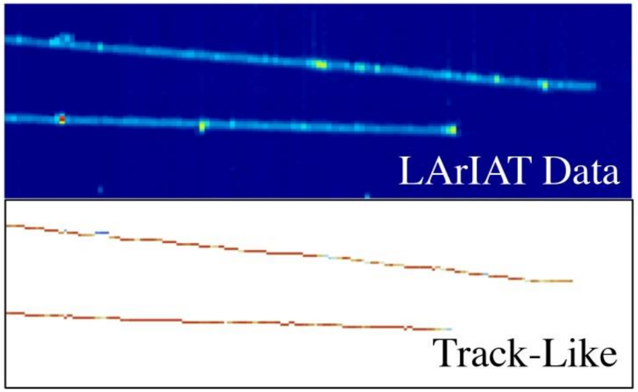
EM component

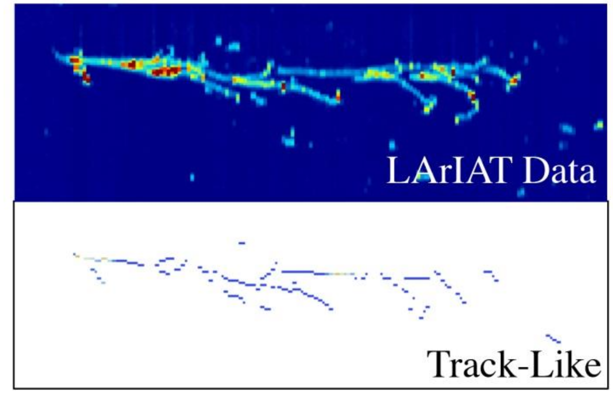


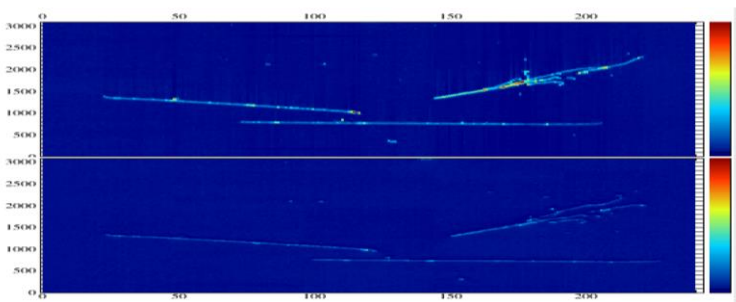
Hadronic component

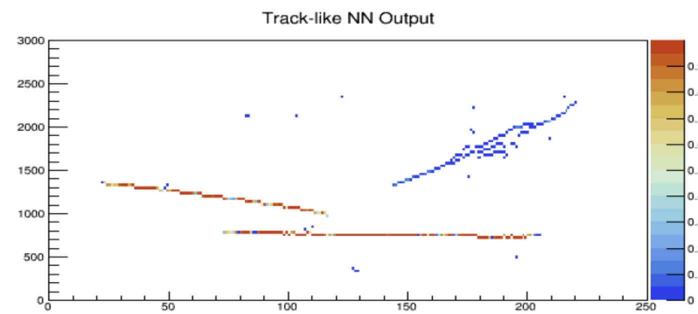


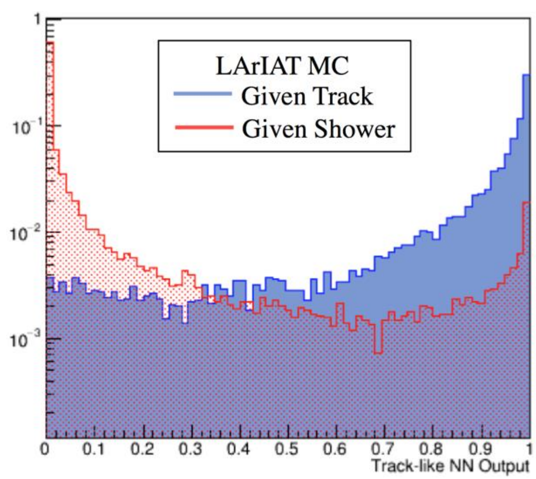
EM/track in LArIAT: test on real data

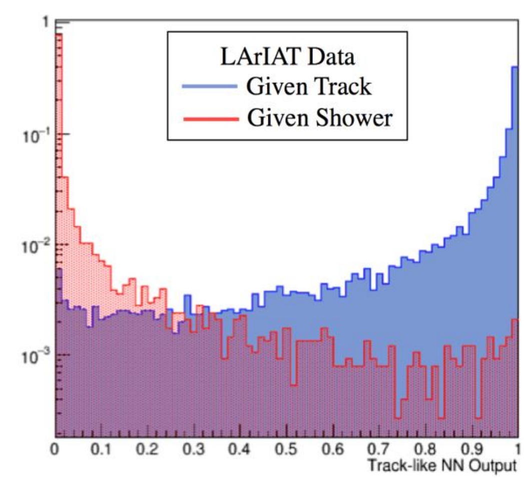






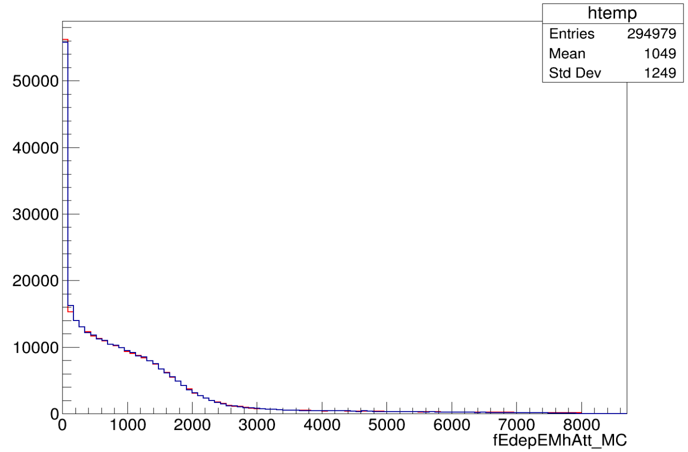




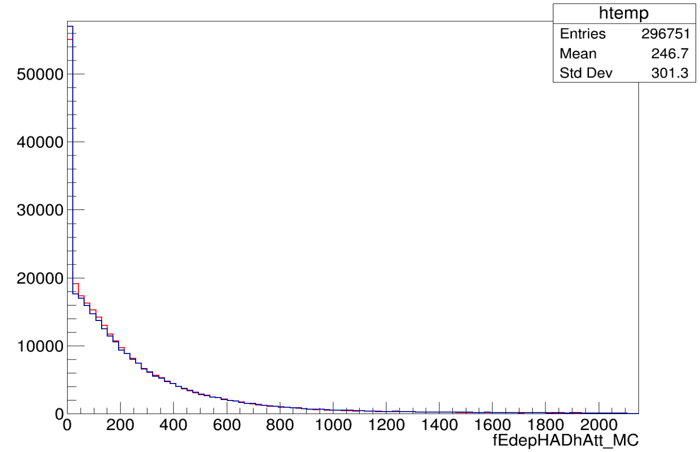


EM/track in neutrino events

EM component

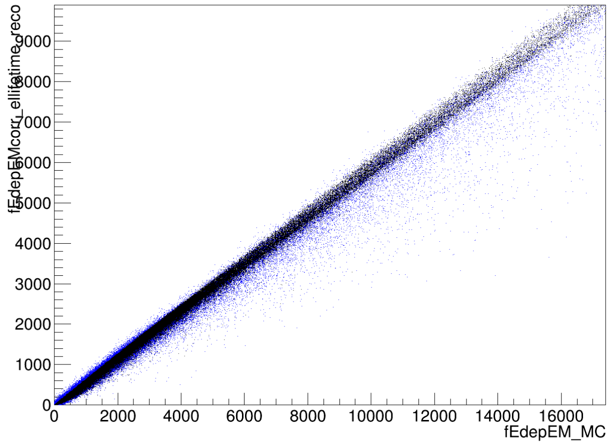


Hadronic component



EM component

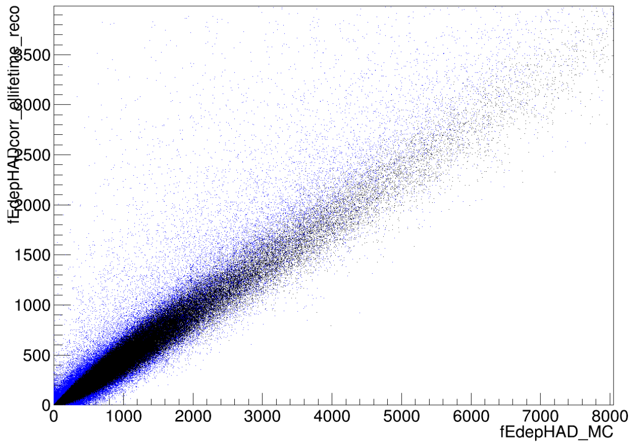
reco



MC

Hadronic component

reco



MC

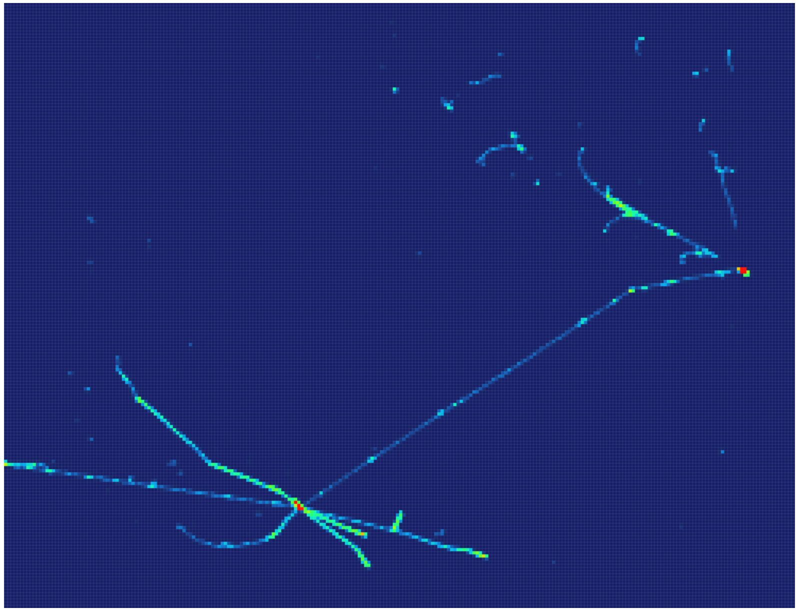
dual-phase 3x1x1 prototype data is waiting

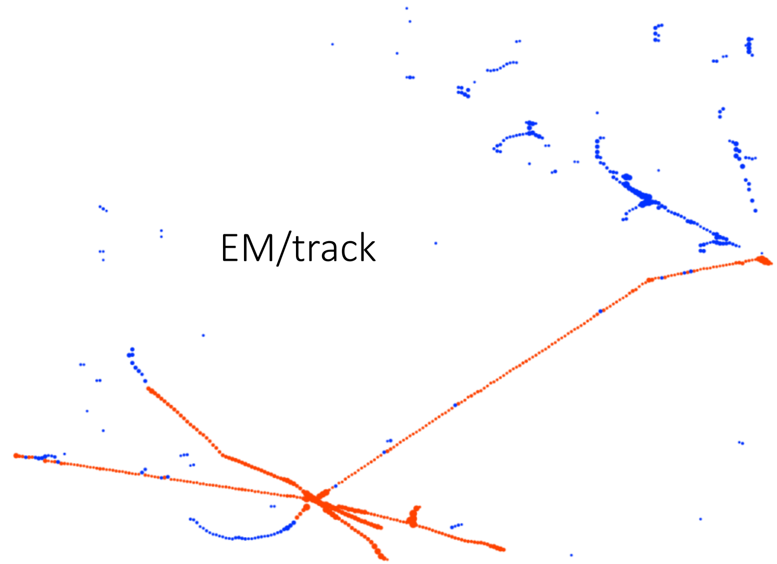
- models were already prepared for dual-phase MC
- need to do the work on data… this is the target, not MC

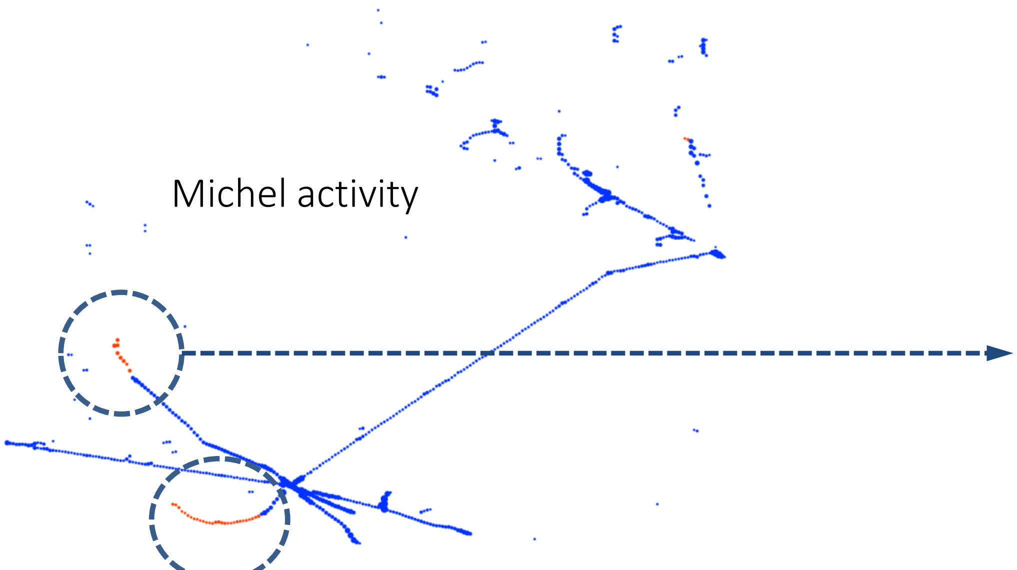
→ do MC-data overlay for most realism of noise → physics week now
→ investigate signal shape differences: back to GAN idea?

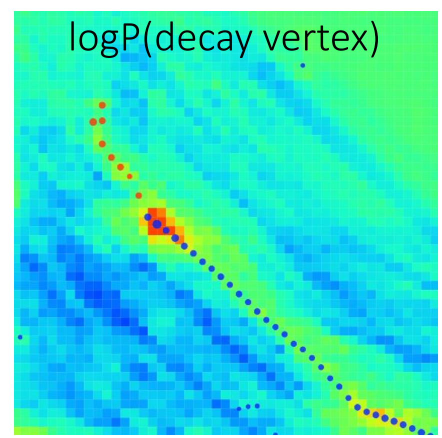
As for all high and low level pattern recognition cases:
→ need to control nuisance parameters and feature differences in MC/data

Michel electron selection: see more on DUNE physics week









Vertex location / classification

- decay / interation vertex ID used also for NDK work (Aaron)
- more vertex ID: EM shower vertex (Leigh)

Moving to technical issues…

- EM/track and Michel selection applied at all hit locations
  - single CNN model, improvments from **architecture/trainning customization**
  - *production mode* needs really fast calculation
  - few Michel examples, many EM/track per event
  - majority of „easy” cases, difficult are less frequent but most important to solve

- Vertex ID applied to points/hits in selected regions
  - 1-to-few training examples per event

→ speed: training, production

→ training set optimization

→ architecture optimization

→ data representation / generators

Usual workflow for CNN preparation

LArSoft

G4 sim, detsim, signal processing

(and any required reco)

Theano used previously, quite faster, but no C++ API

Python, Keras API +Tensorflow backend

Model prototyping, training, tests

does not need to be CNN/CVN

save graph description, model coefficients

LArSoft

Load the graph, apply CNN

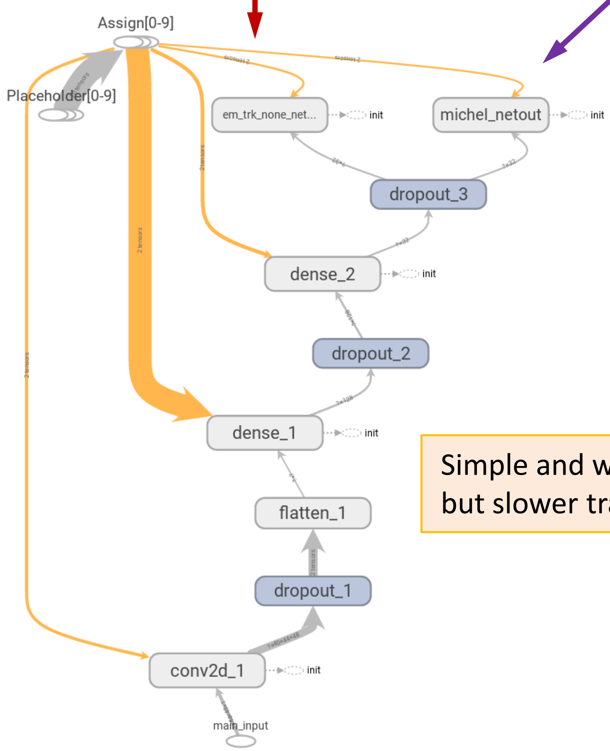
(model is fixed, „inference mode”)

- graph needs only a simple I/O interface,
- straightforward to port to any framework
- one can mix even different backends: Caffe / TF can read the sam format

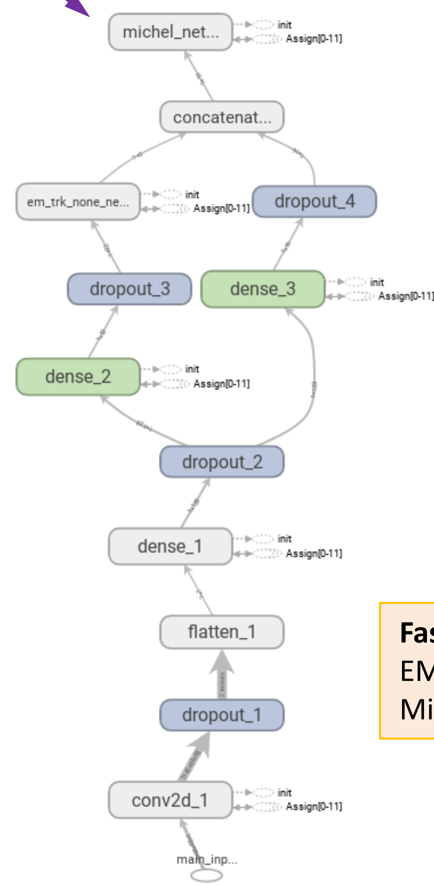
Graphs

EM – track – none layer, multi-class softmax / cross-entropy

Michel-or-not layer, binary sigmoid / MSE



Simple and work very well, but slower training



Fast learner!
EM/track regularizes the Michel output

(Theano + our old, simple API could handle „sequential" graph, single output layer only)

Graphs

→ description of CNN ideally suited for parallelization:

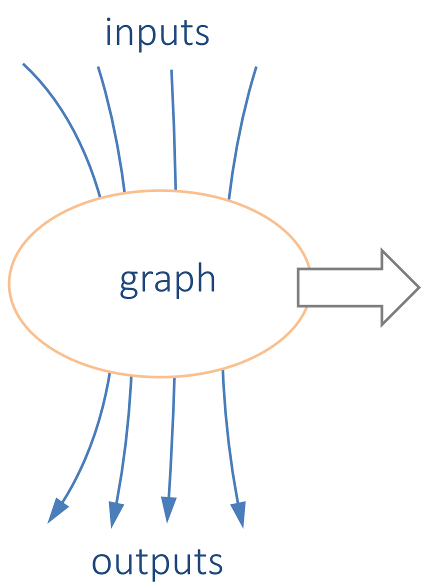
-- input / output data as a tensors

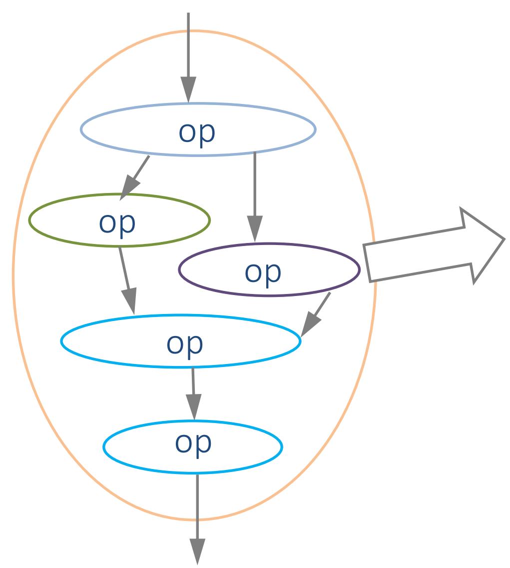
-- batch of input tensors processed in parallel with a single graph

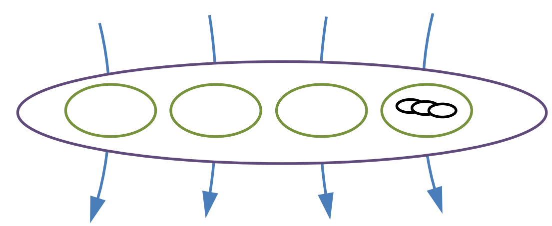
→ calculations decomposed into a basic building blocks to profit from parallelization & vectorization

→ TF and Keras toolkits provide API to handle complex graphs in a few lines, no need to see all details

→ not for „if-else” heavy solutions







single operation can be parallelized (e.g. kernel applied to every element in tensor)

each input-output can be calculated in parallel to others

operations inside graph can run in parallel

CPU/GPU can apply simple math instructions to multiple operands

Graphs

Can describe every custom detail with TF:

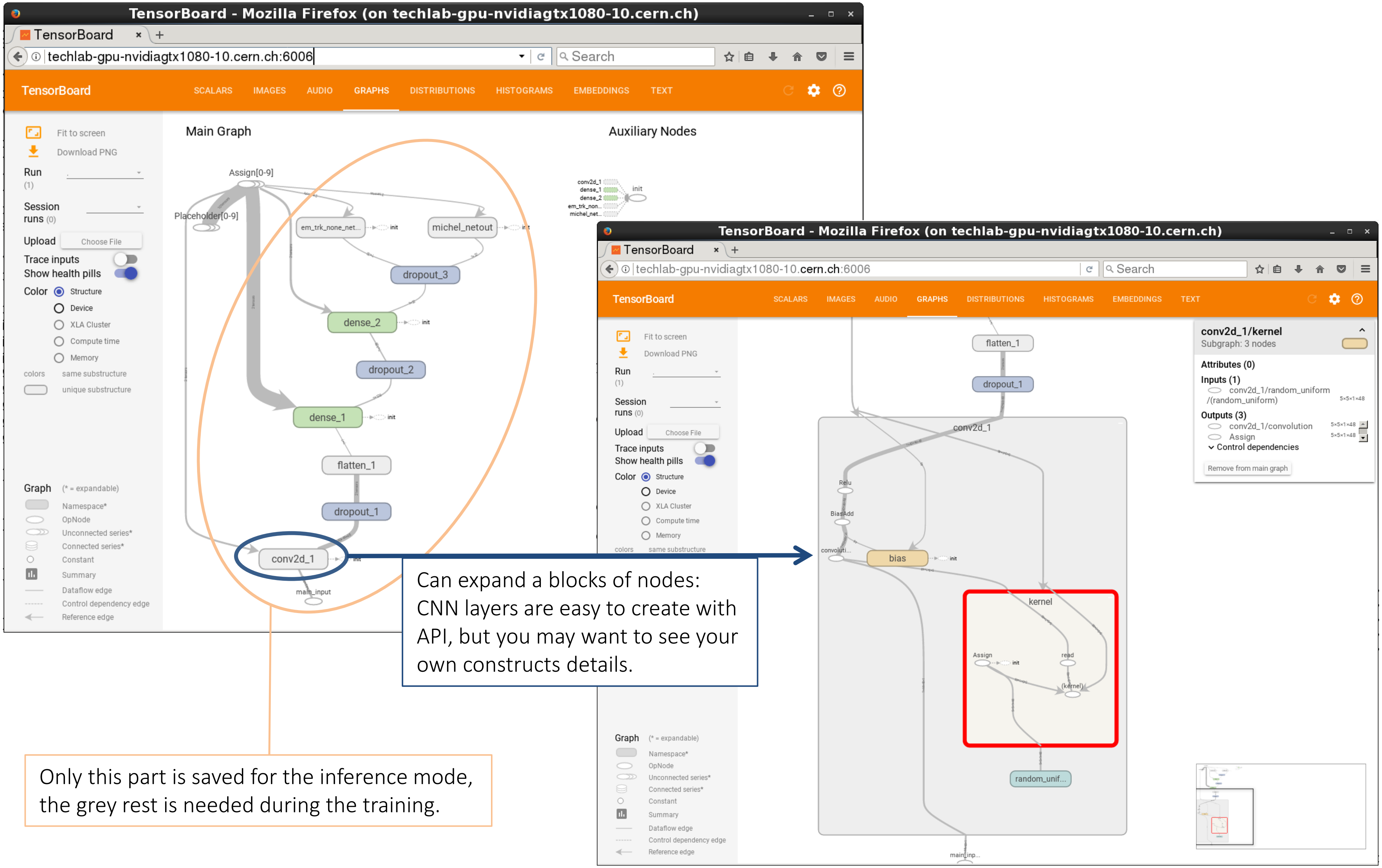
```
import tensorflow as tf

with tf.Session() as sess:
    a = tf.Variable(5.0, name='a')
    b = tf.Variable(6.0, name='b')
    c = tf.multiply(a, b, name="c")

    sess.run(tf.global_variables_initializer())

    print a.eval() # 5.0
    print b.eval() # 6.0
    print c.eval() # 30.0

    tf.train.write_graph(sess.graph_def,
            'models/', 'graph.pb', as_text=False)
```

…but usually use Keras as a high level API for TF:

```
main_input = Input(shape=(img_rows, img_cols, 1),
                   name='main_input')

x = Conv2D(nb_filters1, (nb_conv1, nb_conv1),
           padding='valid', data_format='channels_last',
           activation=convactfn1)(main_input)

x = Dropout(drop1)(x)
x = Flatten()(x)

x = Dense(densesize1, activation=actfn1)(x)
x = Dropout(drop2)(x)

x = Dense(densesize2, activation=actfn2)(x)
x = Dropout(drop2)(x)

em_trk_none = Dense(3, activation='softmax,,
                    name='em_trk_none_netout')(x)
michel = Dense(1, activation='sigmoid',
               name='michel_netout')(x)

sgd = SGD(lr=0.01, decay=1e-4, momentum=0.9, nesterov=True)
model = Model(inputs=[main_input],
              outputs=[em_trk_none, michel])
model.compile(optimizer=sgd,
     loss={'em_trk_none_netout': 'categorical_crossentropy',
           'michel_netout': 'mean_squared_error'},
     loss_weights={'em_trk_none_netout': 0.1,
                   'michel_netout': 1.})
```

C++ API for TF allows building graphs on the fly, but what we really need is **loading graphs worked out in Python: this is available now in UPS.**

Graphs

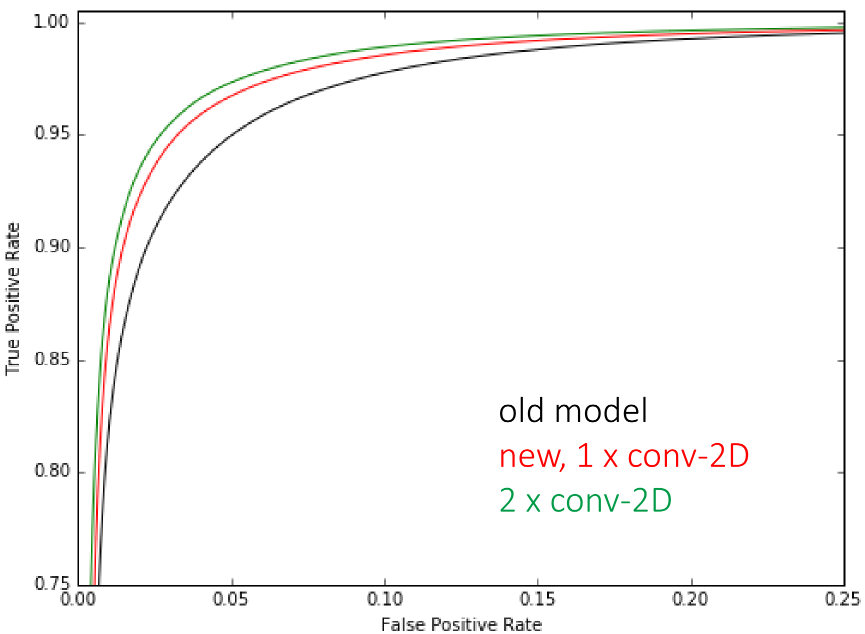


Can expand a blocks of nodes:
CNN layers are easy to create with API, but you may want to see your own constructs details.

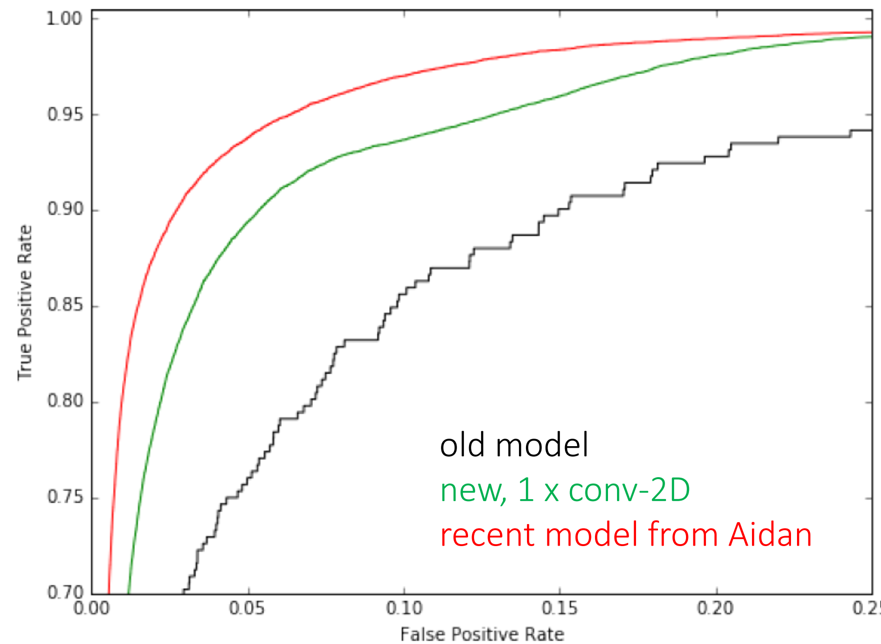
Only this part is saved for the inference mode, the grey rest is needed during the training.

Performance with adjusted architecture

EM-like selection ROC

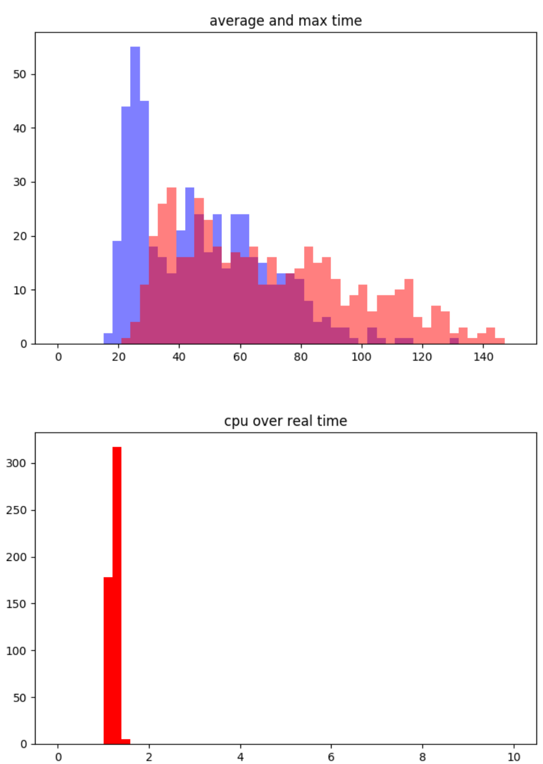


Michel-like selection ROC

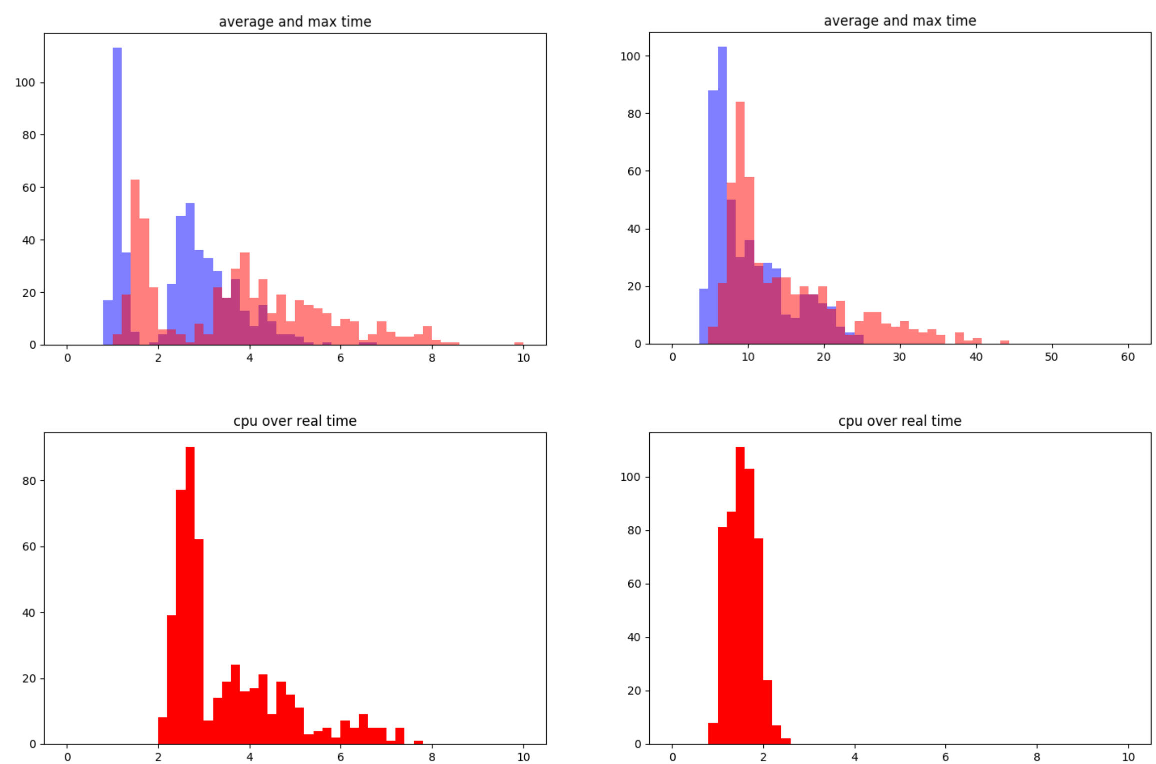


Running on the grid

- old code: almost no parallelization, little vectorization
- **Tensorflow – extremely flexible parallelization, code optimized, but: still cannot use full vectorization (no AVX on some grid CPUs)**
- need convenient **mechanism to select code that fits hardware: old CPU / GPU / KNL / …**
- parallelization / vectorization may be very dependent on backend: **target also Caffe / TF selection**



old model / code



same size model / new code / two rounds of grid submissions

On a ProtoDUNE full event: 6 APA's,   beam event + ~60-70 cosmic tracks

Simple c++ implementation of CNN, 4 CPU

```
==========================================================
TimeTracker printout (sec)                            Avg
==========================================================
Full event                                          1029.31
----------------------------------------------------------
source:RootInput(read)                            0.00172957
reco:rns:RandomNumberSaver                        0.000820898
reco:caldata:DataPrepModule                         7.14426
reco:gaushit:GausHitFinder                          65.1564
reco:hitfd:HitFinder35t                             107.047
reco:linecluster:LineCluster                        1.5267
reco:emtrkmichelid:EmTrackMichelId                  606.107
reco:pmtrack:PMAlgTrackMaker  (beam+cosmics)        152.281
reco:pandora:StandardPandora  (only cosmics)        72.9939
reco:TriggerResults:TriggerResultInserter         0.000111717
end_path:out1:RootOutput                          2.9803e-05
end_path:out1:RootOutput(write)                     17.0451
==========================================================
MemoryTracker summary (base-10 MB units used)

  Peak virtual memory usage (VmPeak)   : 3884.21 MB
  Peak resident set size usage (VmHWM) : 2863.01 MB
==========================================================

TimeReport ---------- Time  Summary ---[sec]----
TimeReport CPU = 1321.828051 Real = 1037.665077
```

Tensorflow, using AVX, 4 CPU

```
==========================================================
TimeTracker printout (sec)                            Avg
==========================================================
Full event                                          486.511
----------------------------------------------------------
source:RootInput(read)                            0.00148615
reco:rns:RandomNumberSaver                        0.000784336
reco:caldata:DataPrepModule                         7.04595
reco:gaushit:GausHitFinder                          65.6555
reco:hitfd:HitFinder35t                             115.82
reco:linecluster:LineCluster                        1.75606
reco:emtrkmichelid:EmTrackMichelId                  50.716
reco:pmtrack:PMAlgTrackMaker                        152.856
reco:pandora:StandardPandora                        76.3032
reco:TriggerResults:TriggerResultInserter         0.000106771
end_path:out1:RootOutput                          4.4151e-05
end_path:out1:RootOutput(write)                     16.3541
==========================================================
MemoryTracker summary (base-10 MB units used)

  Peak virtual memory usage (VmPeak)   : 4463.87 MB
  Peak resident set size usage (VmHWM) : 3052.71 MB
==========================================================

TimeReport ---------- Time  Summary ---[sec]----
TimeReport CPU = 620.489671 Real = 494.307292
```

→ CNN runs 10x faster, no longer bottleneck, but it is not the only thing in the reco chain

→ parallelization has an overhead, likely allocating 2 or 4 CPU on the fermigrid should be optimal

→ the larger batch of inputs, the faster processing, ...and more memory needed

→ **optimally, feature maps should be done once / event, not redone for each patch...**

Upcoming work: more brain power on training data handling

1. EM / track / Michel: simple workflow enough

→ ADC / PDG maps in ROOT histograms, then „patch" preparation for the training

→ training set composition „optimized" manually (otherwise heavy weapon for a simple task)

→ straigt-forward data augmentation (flips only)

2. More towards CNN/reco combined: disambiguation and segmentation, work waiting for the Ph.D. student

→ ADC / TrackID maps in ROOT histograms, will need multiple resolution „glimps"

→ training batches **should** be generated o the fly

→ likely will go for automated opimization of training set

3. primary vertex location: main focus now due to the DUNE TDR

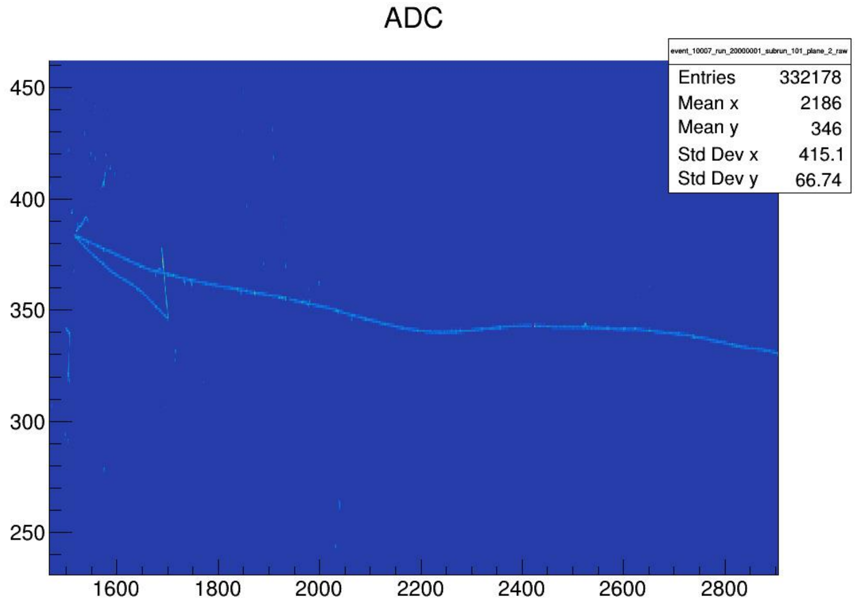
→ ADC maps in ROOT histograms, 1 byte / pixel, ~150-180 kB / DUNE LBL ν event – ready

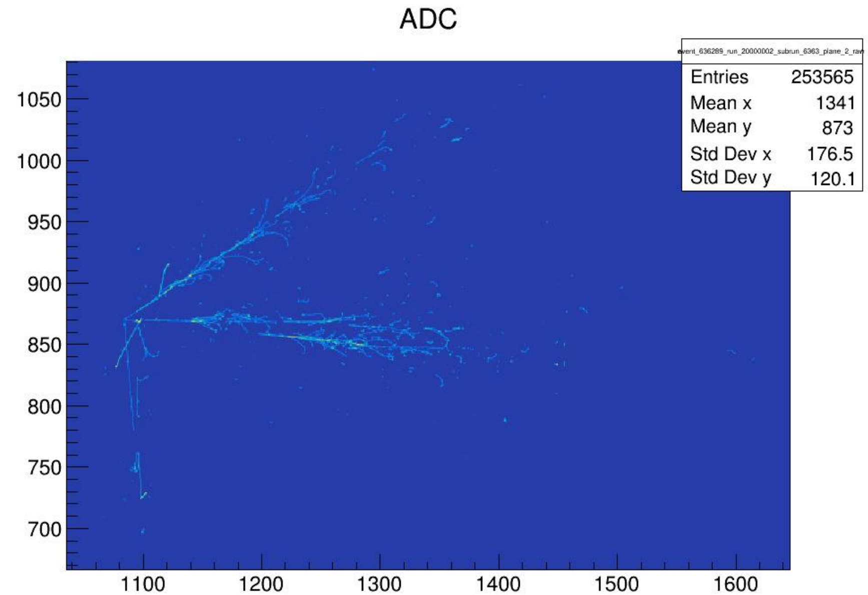
→ multiple resolution „glimps", recurrent CNN: batches **MUST** be generated o the fly – ready

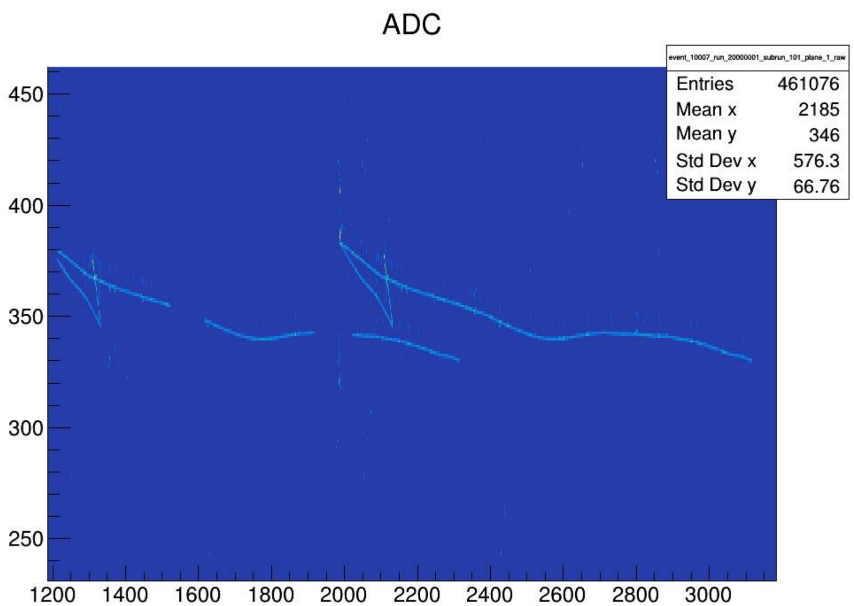
→ automated opimization of training batches – ongoing

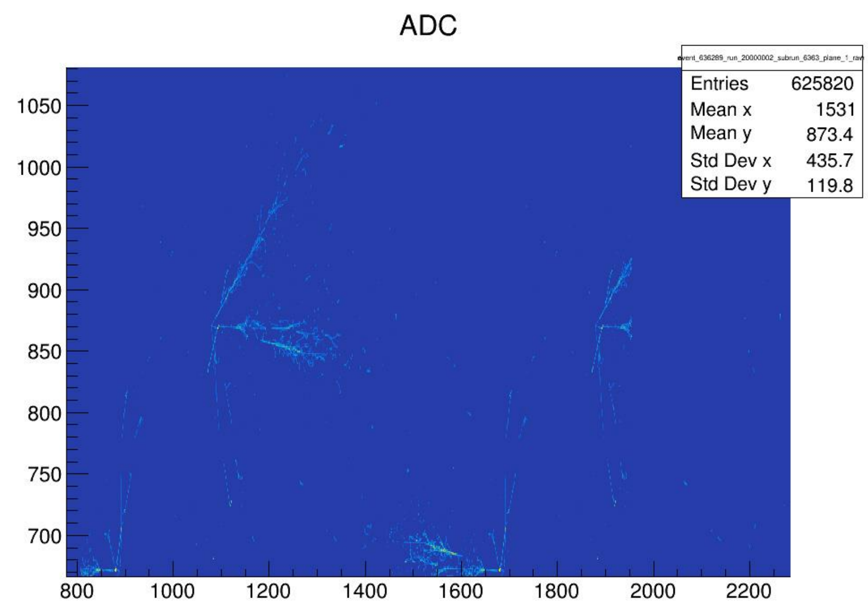
→ NOTE non-trivial issues with the wire wrapping and „global image" – done in a hacky way

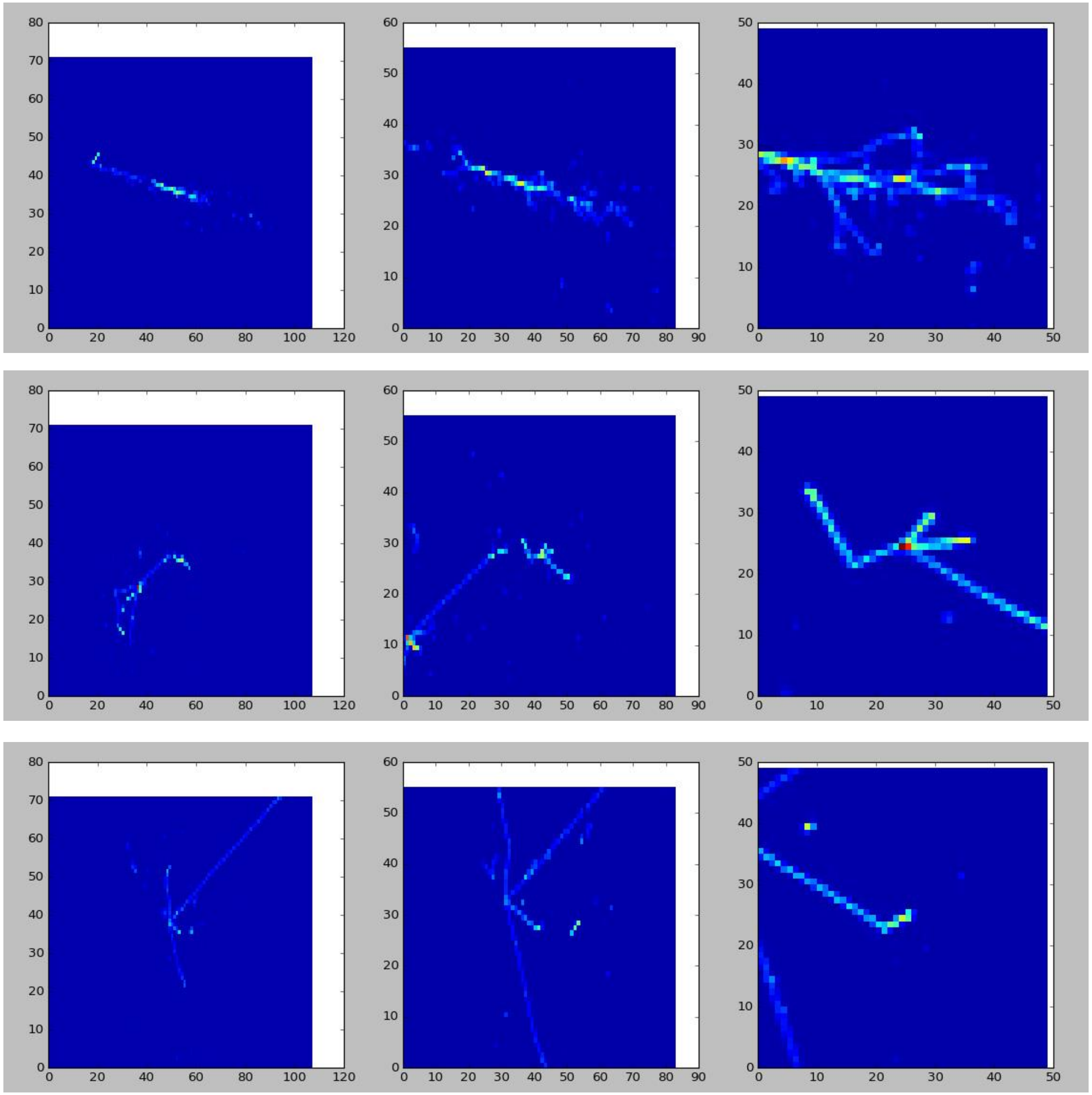
Wrapped wires can be very annoying…











Training data representation

For practical work:

→ need >1M events in memory, can swap from disk occasionally

→ need fast unpacking for the batch generation purposes

→ no strong downsampling (not below the detector effective resolution)

Items to try today?

→ load Caffe model with TF/Keras, …or vice versa and include Caffe backend in EM/track inference

→ modify CVN for 2-view dual-phase neutrinos

→ look what event formats others are using