Integration of NuGraph GNN into LArSoft

Giuseppe Cerati (FNAL)

LArSoft Coordination Meeting

Dec. 12, 2023

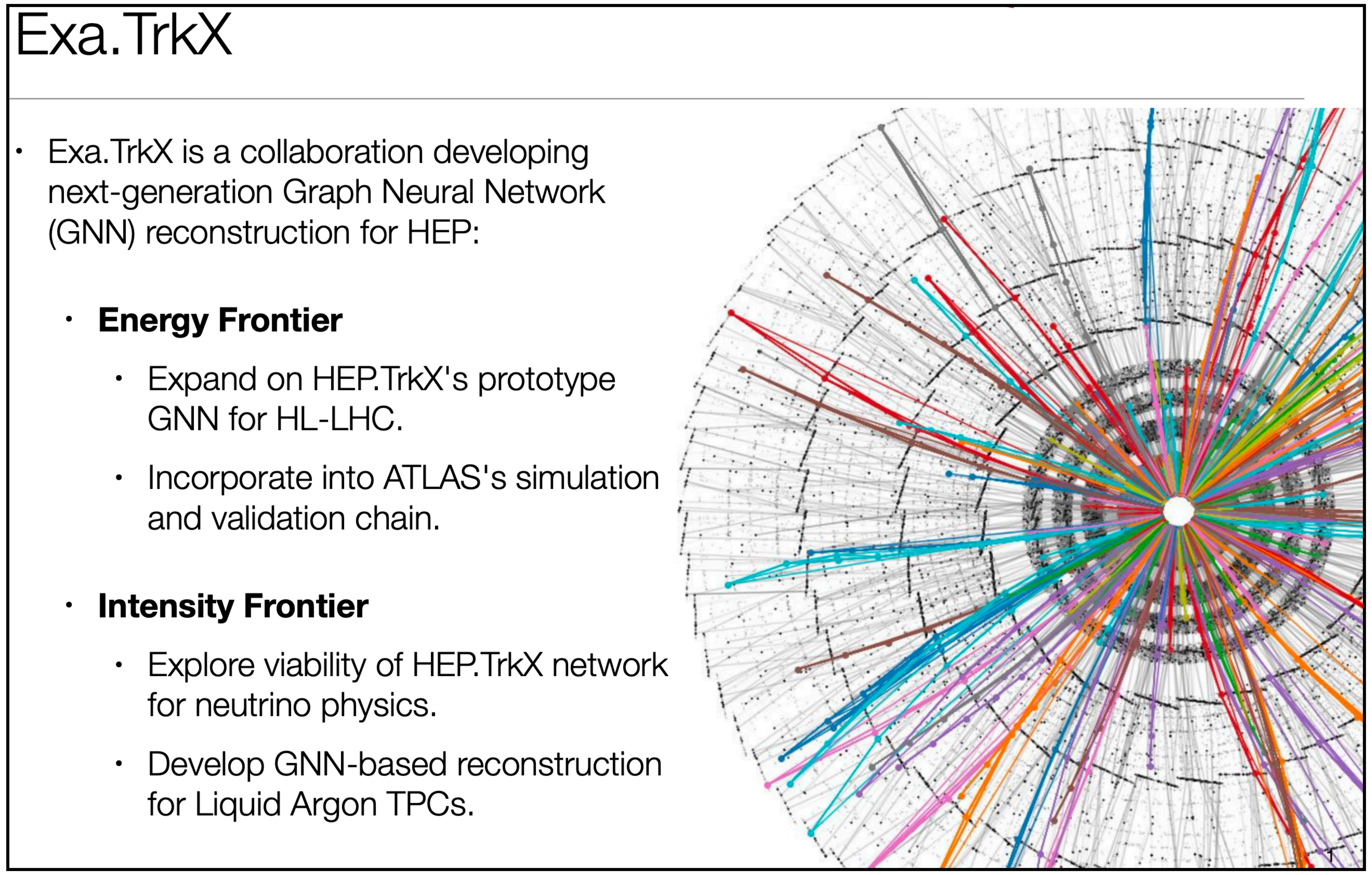
Introduction

- GNNs have been successfully used for tracking application at LHC, can they be used for low-level LArTPC reconstruction?
  - Eur.Phys.J.C 81 (2021) 10, 876 • e-Print: 2103.06995

- I am presenting work by the Exa.TrkX collaboration based on the MicroBooNE open samples
  - LArTPC core team:
    FNAL (GC, J. Kowalkowski), UCincinnati (A. Aurisano, V Hewes)
  - initial results already presented at recent conferences
  - we have a paper in preparation, stay tuned!

- This network architecture is developed to have broad applicability, without being tied to any particular detector geometry.
  - This network was initially developed in the context of the DUNE Far Detector geometry for reconstructing high-multiplicity atmospheric and ντ interactions.
  - Also being deployed on non-LArTPC detector technology!
  - See NuML and pynuml packages

credit: V Hewes



Exa.TrkX

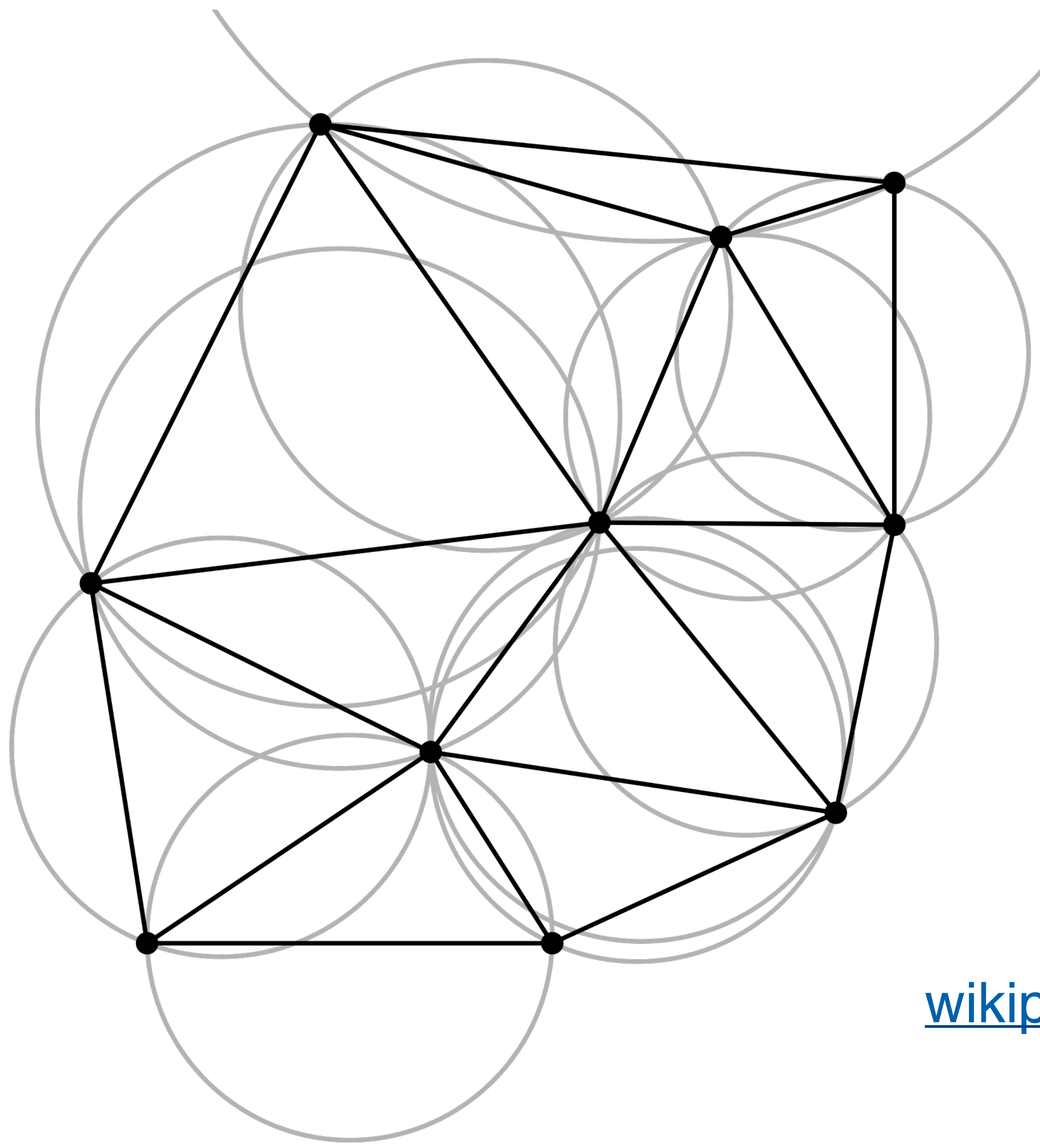
- Exa.TrkX is a collaboration developing next-generation Graph Neural Network (GNN) reconstruction for HEP:
  - **Energy Frontier**
    - Expand on HEP.TrkX's prototype GNN for HL-LHC.
    - Incorporate into ATLAS's simulation and validation chain.
  - **Intensity Frontier**
    - Explore viability of HEP.TrkX network for neutrino physics.
    - Develop GNN-based reconstruction for Liquid Argon TPCs.

Main idea

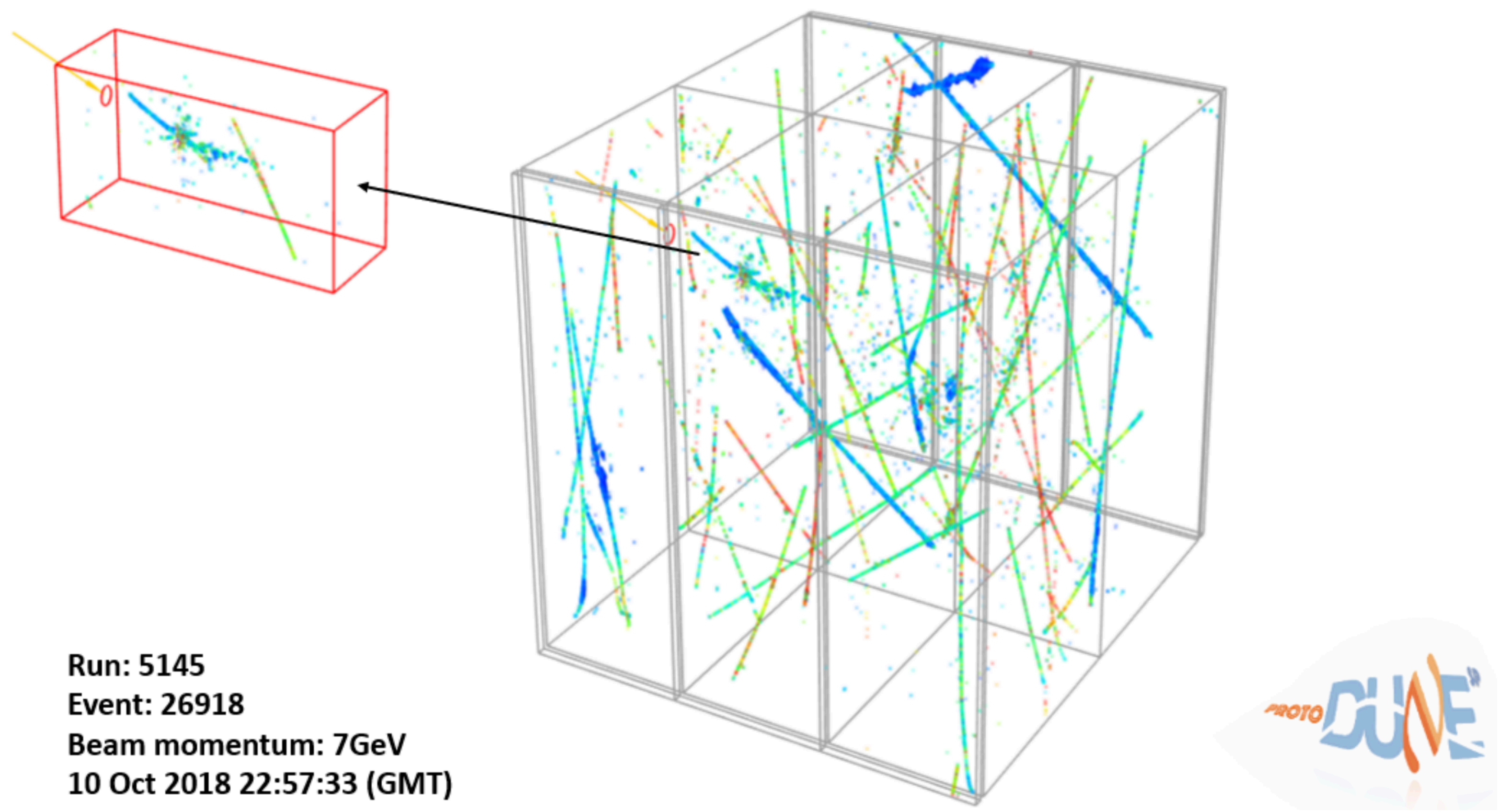
- LArTPC hits can be connected in a graph
  - Naturally sparse representation of the data
  - Low-level information, close to native output of the detector
  - Graphs can also connect hits from different planes, thus making the network “3D-aware”

Inputs and Graph formation

- Main inputs to the GNN are the Hits
  - hits are Gaussian fits to waveforms
  - features: wire, peak time, integral, RMS
  - currently using Hits associated to the neutrino interaction by the “Pandora” algorithm
- Within each plane hits are connected in a graph using Delaunay triangulation
  - fully connected graph, both long and short distance edges, able to jump across unresponsive wire regions
- Hit associations to 3D SpacePoints (currently from “SP solver” algo) are used to create “nexus” connections across graphs in each plane
  - SpacePoints are not connected among themselves
  - No input features for SpacePoints



wikipedia



arXiv:2002.03005

NuGraph2 Network Architecture: Overview

- Initial application for the GNN is semantic hit classification
  - Categories based on the type of particle that produced the hit.

- NuGraph2's core convolution engine is a self-attention message-passing network utilizing a categorical embedding
  - Each particle category is provided with a separate set of embedded features, which are convolved independently.
  - Context information is exchanged between different particle types via a categorical cross-attention mechanism.

- Each message-passing iteration consists of two phases, the planar step and the nexus step:
  - Pass messages internally in each plane.
  - Pass messages up to 3D nexus nodes to share context information.

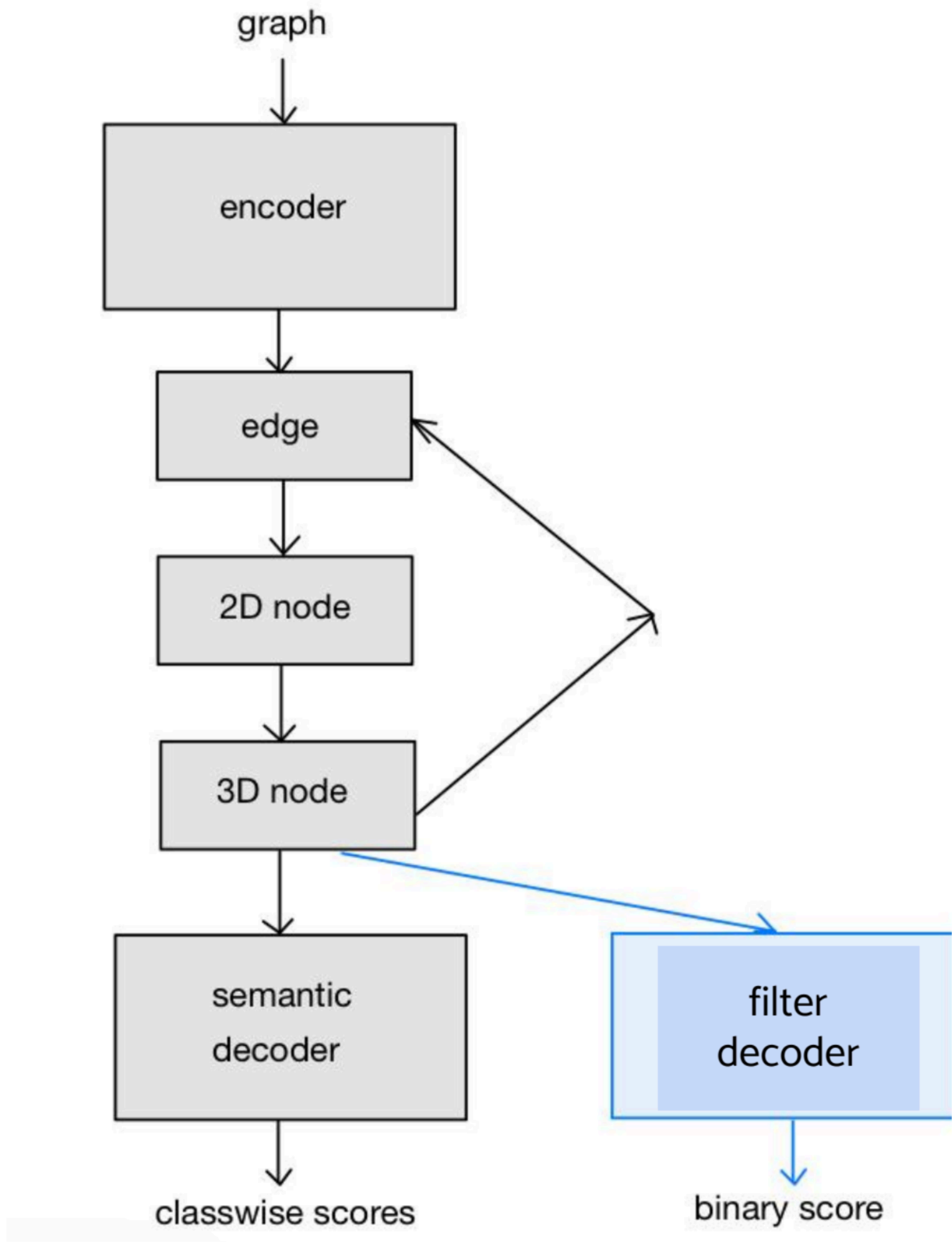
Semantic + Binary Decoders

- The last step at the end of the message passing network is the decoder step
  - Node classifications: semantic and filter
  - Event-level regression: vertexing (in progress)

- Output both class-wise scores from the semantic decoder and a binary score from the filter decoder

- Same learned features are used as input to all decoders

- Different loss functions weighted based on per-task variance (arXiv:1705.07115)



Fermilab

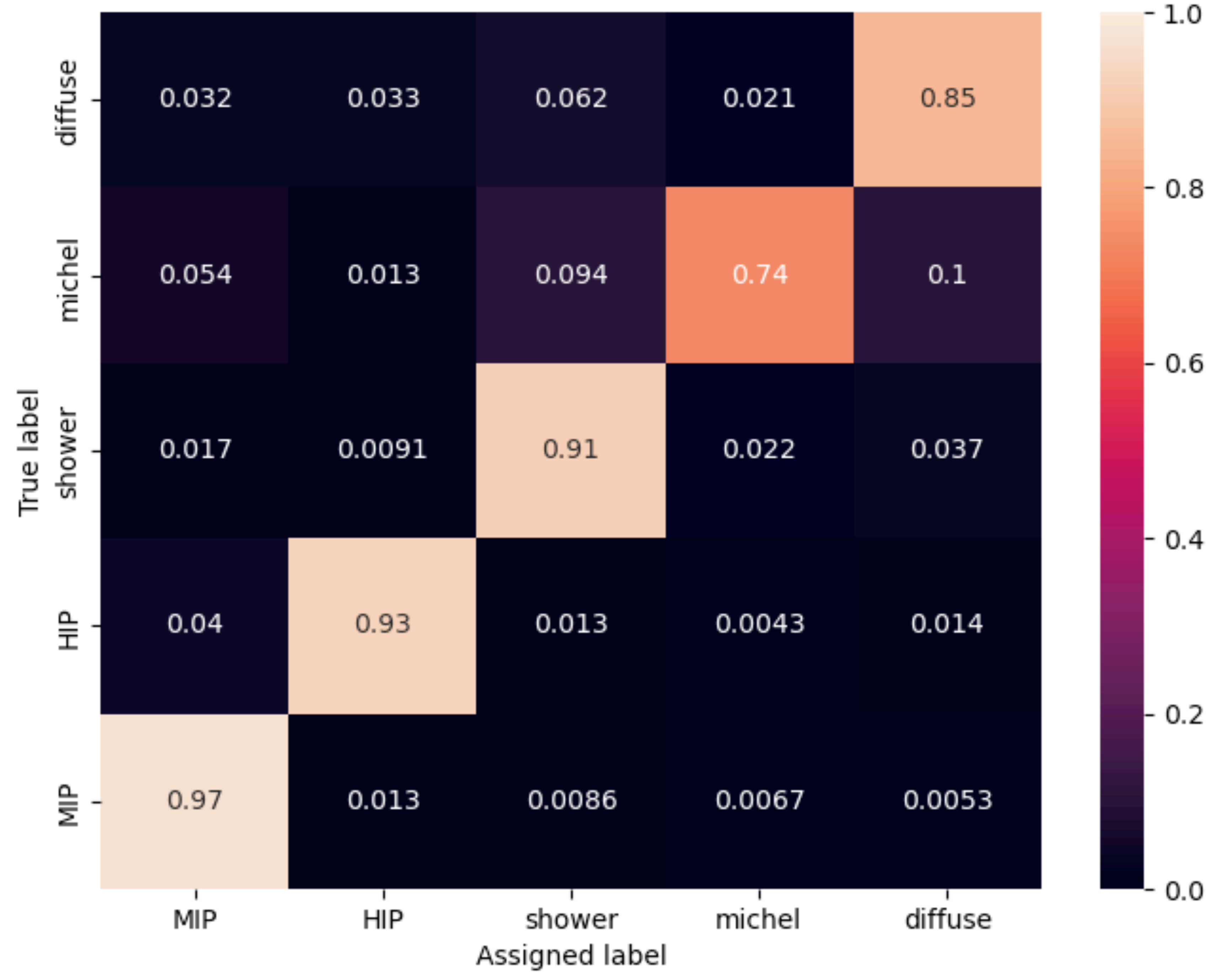
Semantic hit classification

- Decoder trained to classify each neutrino-induced hit according to particle type

- Use five semantic categories:
  - MIP: Minimum ionizing particles (muons, charged pions)
  - HIP: Highly ionizing particles (protons)
  - EM showers (primary electrons, photons)
  - Michel electrons
  - Diffuse activity (Compton scatters, neutrons)

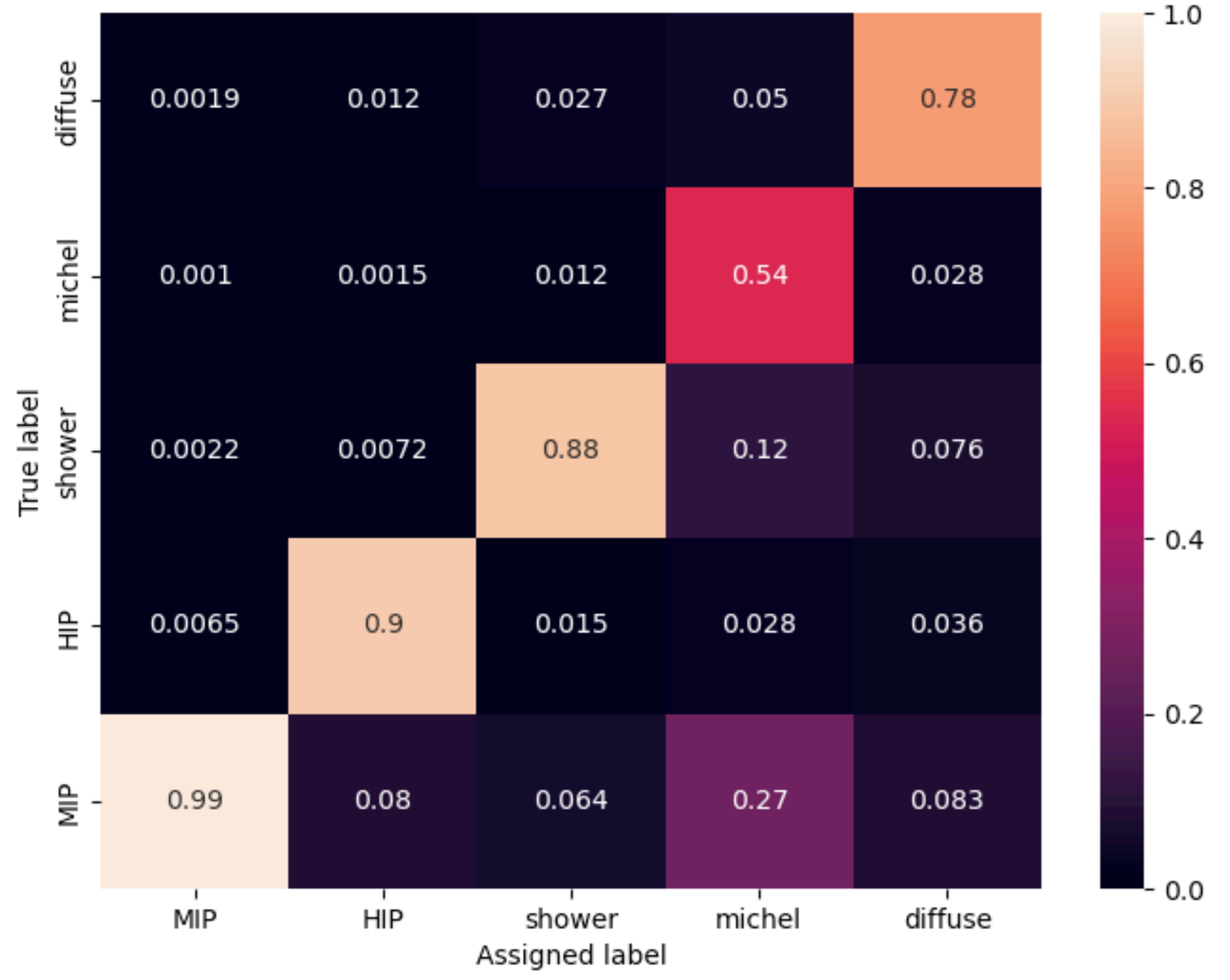
- Performance metrics:
  - recall and precision: ~0.95
  - consistency between planes around 98%
    - compared to ~70% without 3D nexus edges

recall (efficiency)



True label \ Assigned label	MIP	HIP	shower	michel	diffuse
diffuse	0.032	0.033	0.062	0.021	0.85
michel	0.054	0.013	0.094	0.74	0.1
shower	0.017	0.0091	0.91	0.022	0.037
HIP	0.04	0.93	0.013	0.0043	0.014
MIP	0.97	0.013	0.0086	0.0067	0.0053

precision (purity)

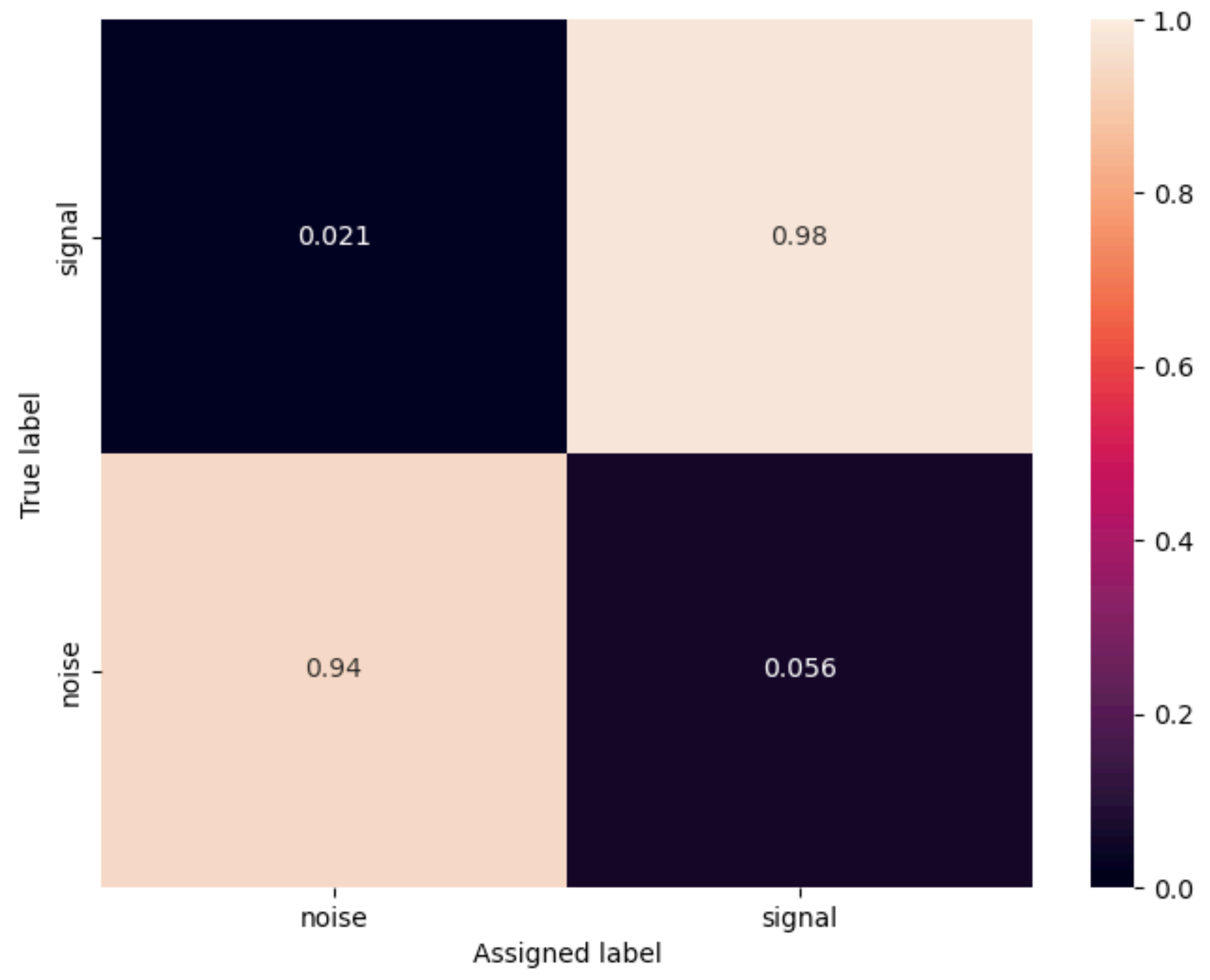


True label \ Assigned label	MIP	HIP	shower	michel	diffuse
diffuse	0.0019	0.012	0.027	0.05	0.78
michel	0.001	0.0015	0.012	0.54	0.028
shower	0.0022	0.0072	0.88	0.12	0.076
HIP	0.0065	0.9	0.015	0.028	0.036
MIP	0.99	0.08	0.064	0.27	0.083

Filter hit classification

- Decoder trained to separate neutrino-induced from noise or cosmic-induced hits
  - Pandora slicing tends to prioritize completeness over purity

- Performance metrics:
  - recall and precision: ~0.98

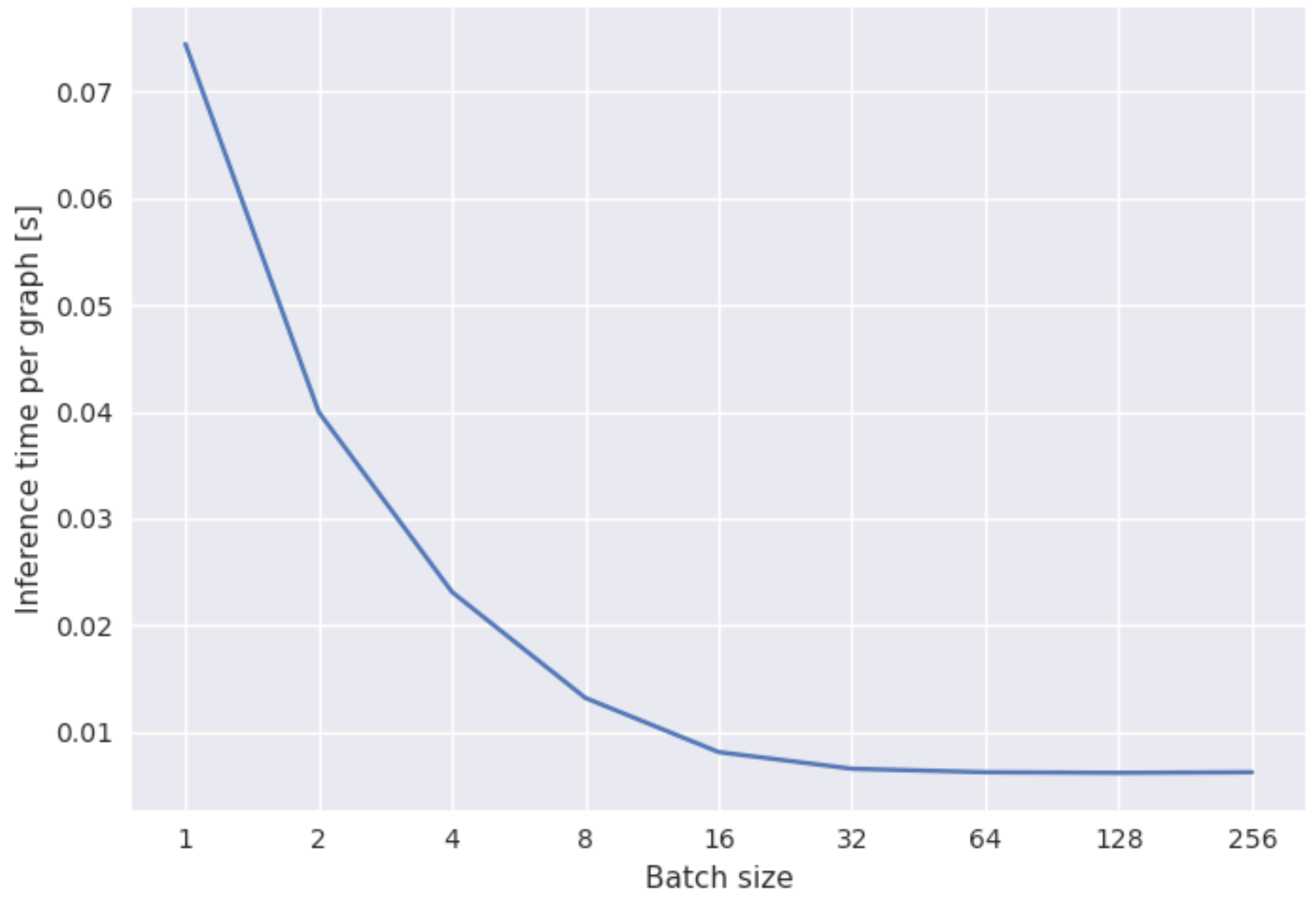


Inference time

- Relatively small network:
  - number of learnable parameters: ~410k
  - max RSS memory on CPU: ~2.5 GB

- Out of the box inference time:
  - 0.12 s/evt on CPU
  - 0.005 s/evt batched on GPU
  - graph construction not included, but also fast

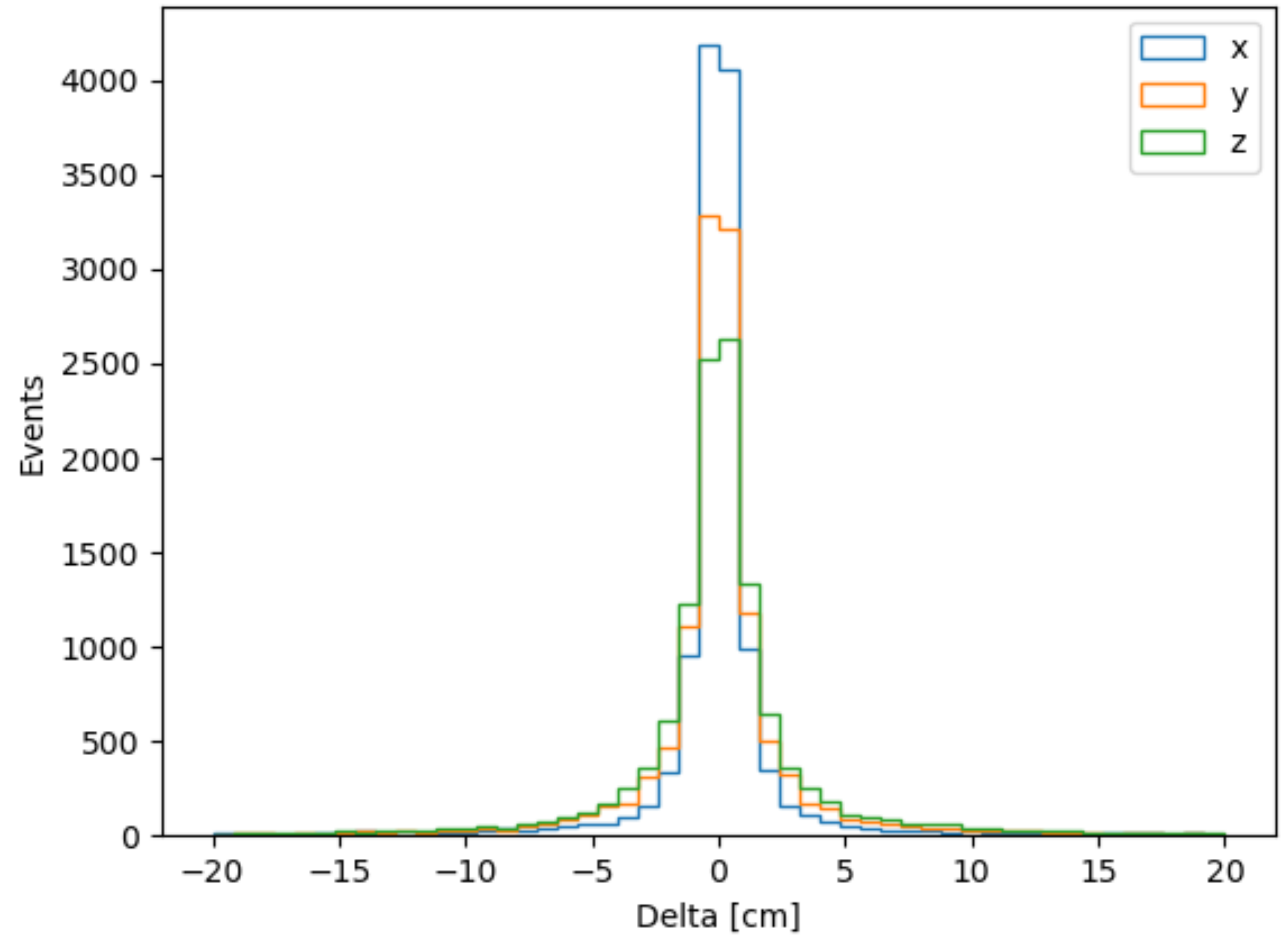
- Implications:
  - can easily run on CPU as part of regular offline processing
  - can run very fast for realtime applications on GPU, or other accelerators

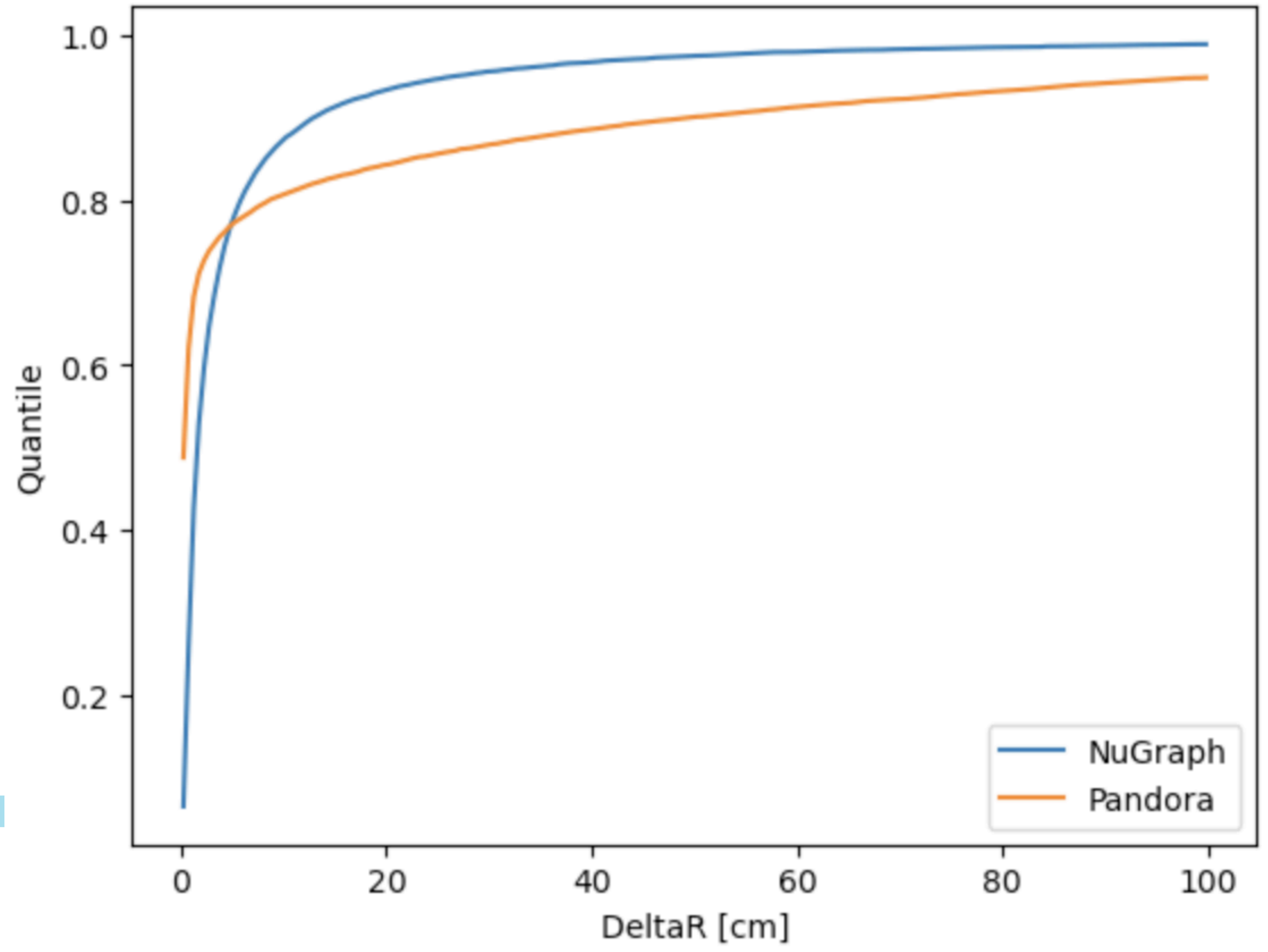


Fermilab

Vertex position classification

- Vertex position decoder using LSTM aggregator
- Preliminary work demonstrates that our GNN is able to identify the neutrino interaction position in 3D
  - currently O(cm) level resolution in each coordinate
- Compared to current vertex reconstruction this version shows worse percentile at low ΔR, but better at larger ΔR
  - worse at finding exact point, better at avoiding catastrophic errors
- Issues related to ground truth definition identified and being fixed, expect to achieve better results soon





Integration in LArSoft: Overview

- Present integration model: libtorch

- Save the GNN model in TorchScript with JIT compiler
  - required a few small changes to the network python code (backup)
- Building libtorch
- Add a new NuGraph package under larrecodnn
  - Creating the graph in LArSoft
  - Run inference from LArSoft
  - Store output in event record
  - Analyze GNN output
  - Code is in feature/cerati_NuGraph branch in forked larrecodnn repo

- Future integration model: NuSonic

Building libtorch

- libtorch is already distributed with LArSoft, but currently older versions are available
  - v1_0_1 (GCC 7.3.0), v1_6_0 and v1_13_1 (more recent GCC versions)
  - our python environment currently uses pytorch v2.1.1 and pytorch-scatter v2.1.2

- Demonstrated working setup with local builds of the above packages

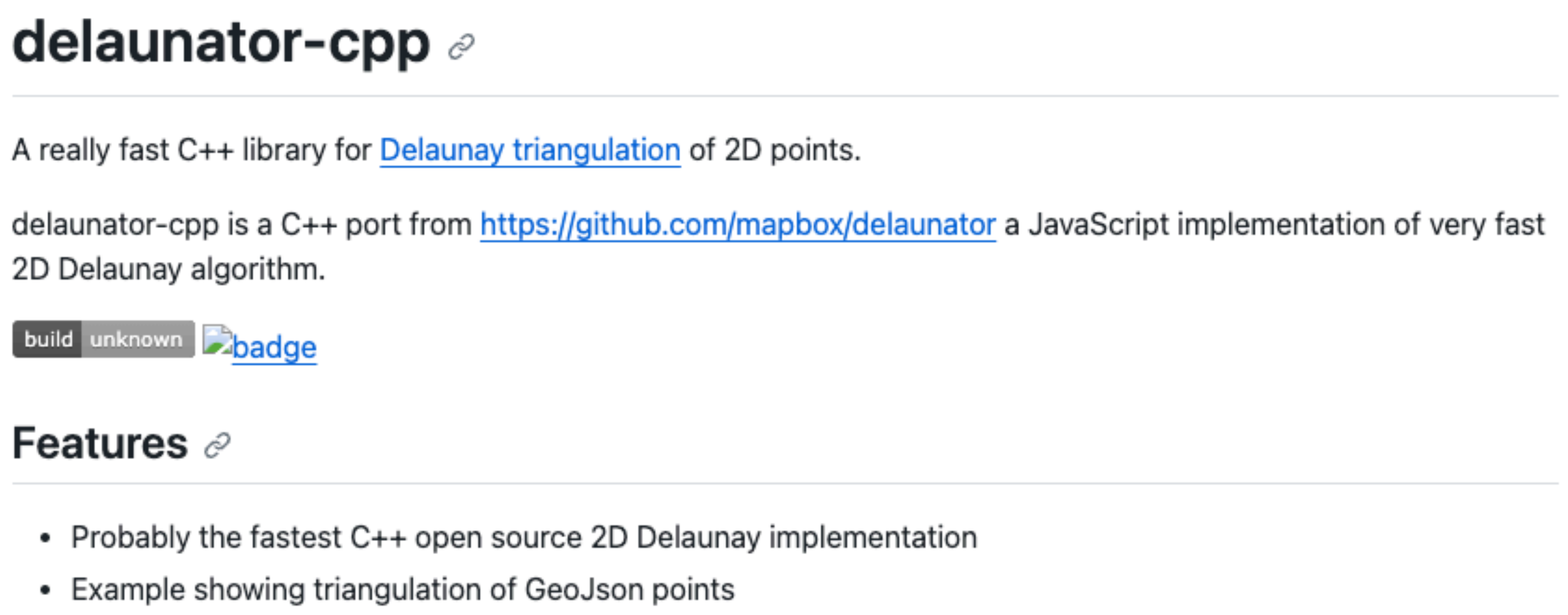
- Requested distribution of built packages in UPS
  - https://cdcvs.fnal.gov/redmine/issues/28425
  - libtorch v2_1_1 is now available on larsoft cvmfs (thanks Lynn!)

- Plan to use this in MicroBooNE MCC9 (GCC 7.3.0)
  - several different options, none straightforward. Deserves a separate discussion.

Creating the graph in LArSoft

- Hits directly available, but features need to be normalized in the same way as in our python setup
- Associations between Hits and SpacePoints (3D nexus edges) are also directly available

- The non-straightforward aspect of graph-building in LArSoft is Delaunay triangulation (2D edges)
- I used “delaunator-cpp” (https://github.com/delfrrr/delaunator-cpp)
  - C++ header-only library, straightforward to import!
  - Able to get identical edges as python setup (after some cleaning of duplicates and sorting)



delaunator-cpp

A really fast C++ library for Delaunay triangulation of 2D points.

delaunator-cpp is a C++ port from https://github.com/mapbox/delaunator a JavaScript implementation of very fast 2D Delaunay algorithm.

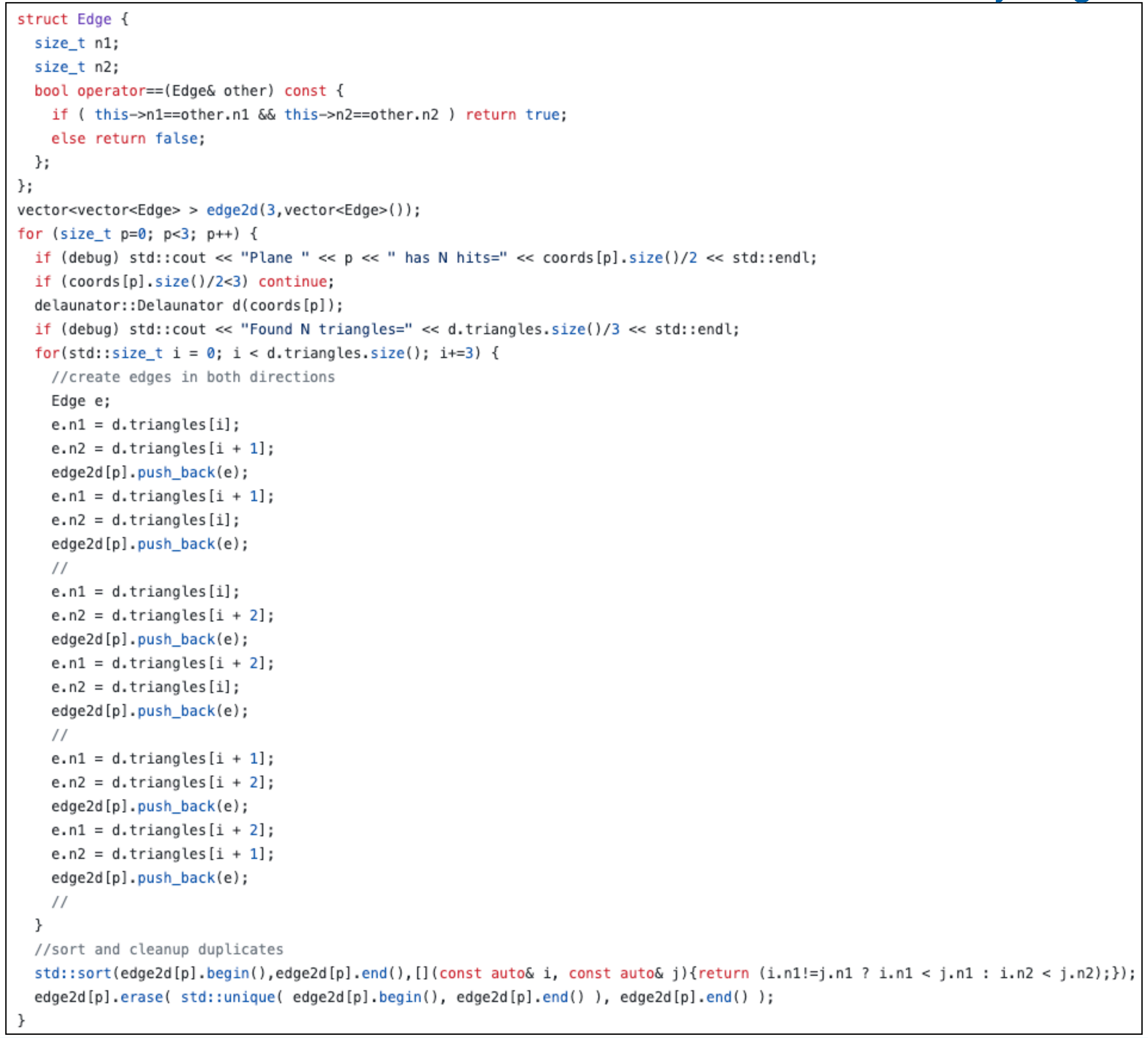
Features

- Probably the fastest C++ open source 2D Delaunay implementation
- Example showing triangulation of GeoJson points

TestInference module - highlights

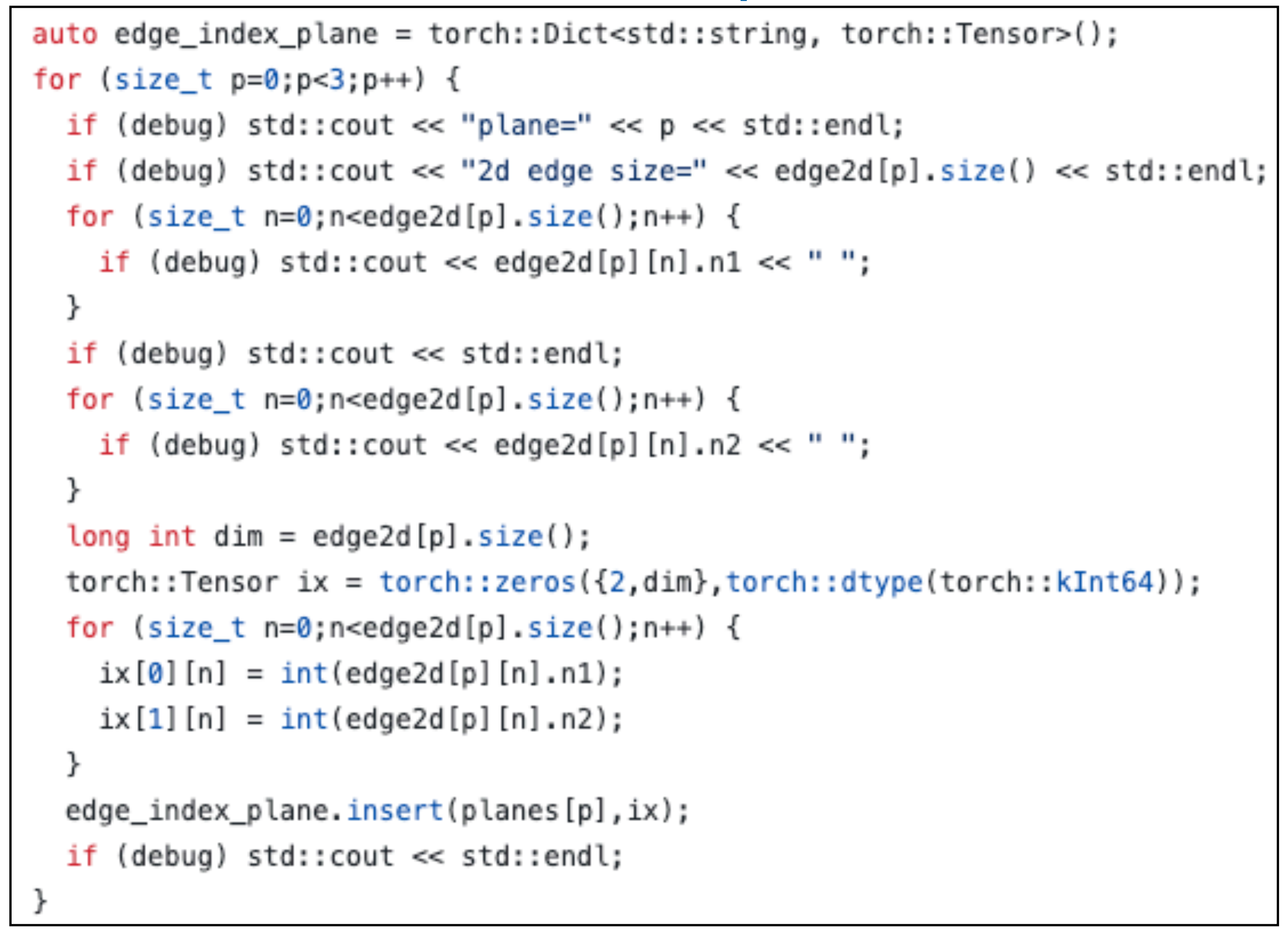
https://github.com/cerati/exatrkxinference/blob/main/exatrkxinference/TestInference_module.cc

create Delaunay edges



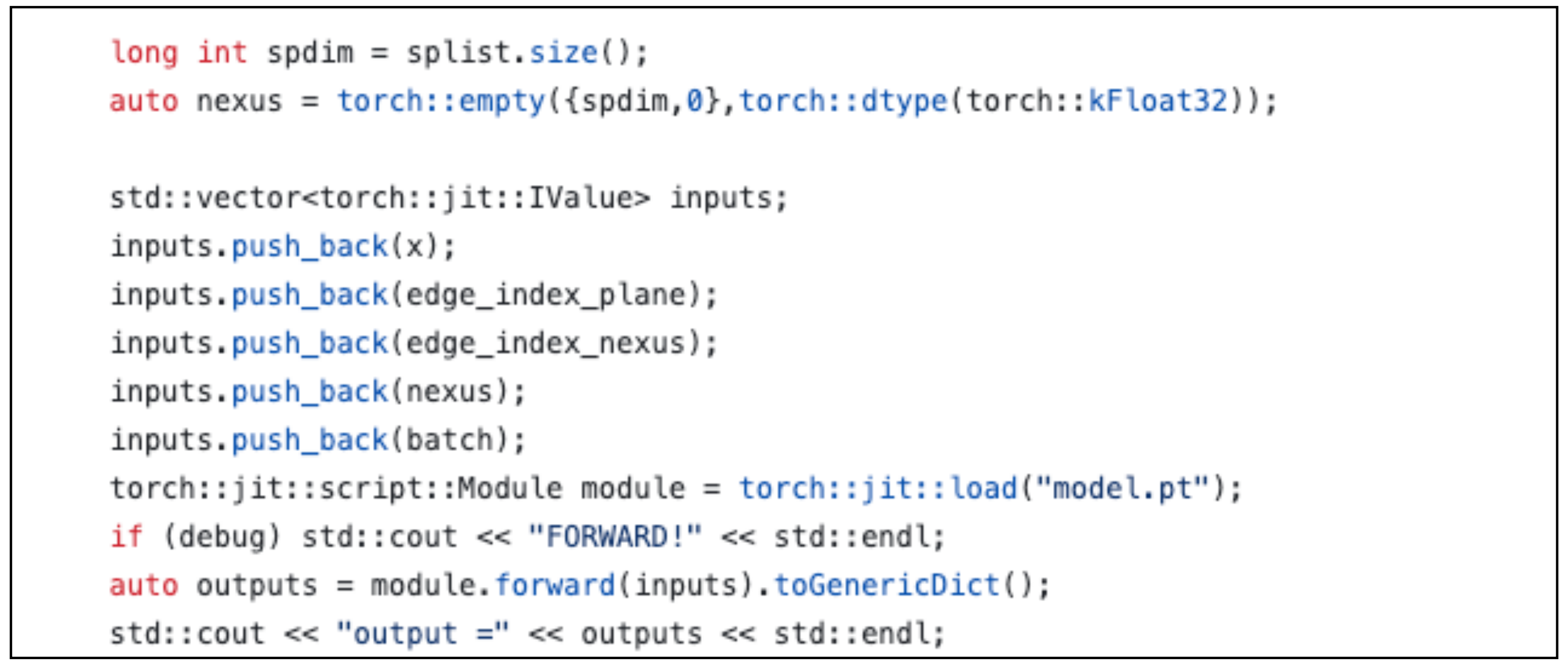
```
struct Edge {
  size_t n1;
  size_t n2;
  bool operator==(Edge& other) const {
    if ( this->n1==other.n1 && this->n2==other.n2 ) return true;
    else return false;
  };
};
vector<vector<Edge> > edge2d(3,vector<Edge>());
for (size_t p=0; p<3; p++) {
  if (debug) std::cout << "Plane " << p << " has N hits=" << coords[p].size()/2 << std::endl;
  if (coords[p].size()/2<3) continue;
  delaunator::Delaunator d(coords[p]);
  if (debug) std::cout << "Found N triangles=" << d.triangles.size()/3 << std::endl;
  for(std::size_t i = 0; i < d.triangles.size(); i+=3) {
    //create edges in both directions
    Edge e;
    e.n1 = d.triangles[i];
    e.n2 = d.triangles[i + 1];
    edge2d[p].push_back(e);
    e.n1 = d.triangles[i + 1];
    e.n2 = d.triangles[i];
    edge2d[p].push_back(e);
    //
    e.n1 = d.triangles[i];
    e.n2 = d.triangles[i + 2];
    edge2d[p].push_back(e);
    e.n1 = d.triangles[i + 2];
    e.n2 = d.triangles[i];
    edge2d[p].push_back(e);
    //
    e.n1 = d.triangles[i + 1];
    e.n2 = d.triangles[i + 2];
    edge2d[p].push_back(e);
    e.n1 = d.triangles[i + 2];
    e.n2 = d.triangles[i + 1];
    edge2d[p].push_back(e);
    //
  }
  //sort and cleanup duplicates
  std::sort(edge2d[p].begin(),edge2d[p].end(),[](const auto& i, const auto& j){return (i.n1!=j.n1 ? i.n1 < j.n1 : i.n2 < j.n2);});
  edge2d[p].erase( std::unique( edge2d[p].begin(), edge2d[p].end() ), edge2d[p].end() );
}
```

fill input torch Tensors



```
auto edge_index_plane = torch::Dict<std::string, torch::Tensor>();
for (size_t p=0;p<3;p++) {
  if (debug) std::cout << "plane=" << p << std::endl;
  if (debug) std::cout << "2d edge size=" << edge2d[p].size() << std::endl;
  for (size_t n=0;n<edge2d[p].size();n++) {
    if (debug) std::cout << edge2d[p][n].n1 << " ";
  }
  if (debug) std::cout << std::endl;
  for (size_t n=0;n<edge2d[p].size();n++) {
    if (debug) std::cout << edge2d[p][n].n2 << " ";
  }
  long int dim = edge2d[p].size();
  torch::Tensor ix = torch::zeros({2,dim},torch::dtype(torch::kInt64));
  for (size_t n=0;n<edge2d[p].size();n++) {
    ix[0][n] = int(edge2d[p][n].n1);
    ix[1][n] = int(edge2d[p][n].n2);
  }
  edge_index_plane.insert(planes[p],ix);
  if (debug) std::cout << std::endl;
}
```

collect inputs, run inference, get output



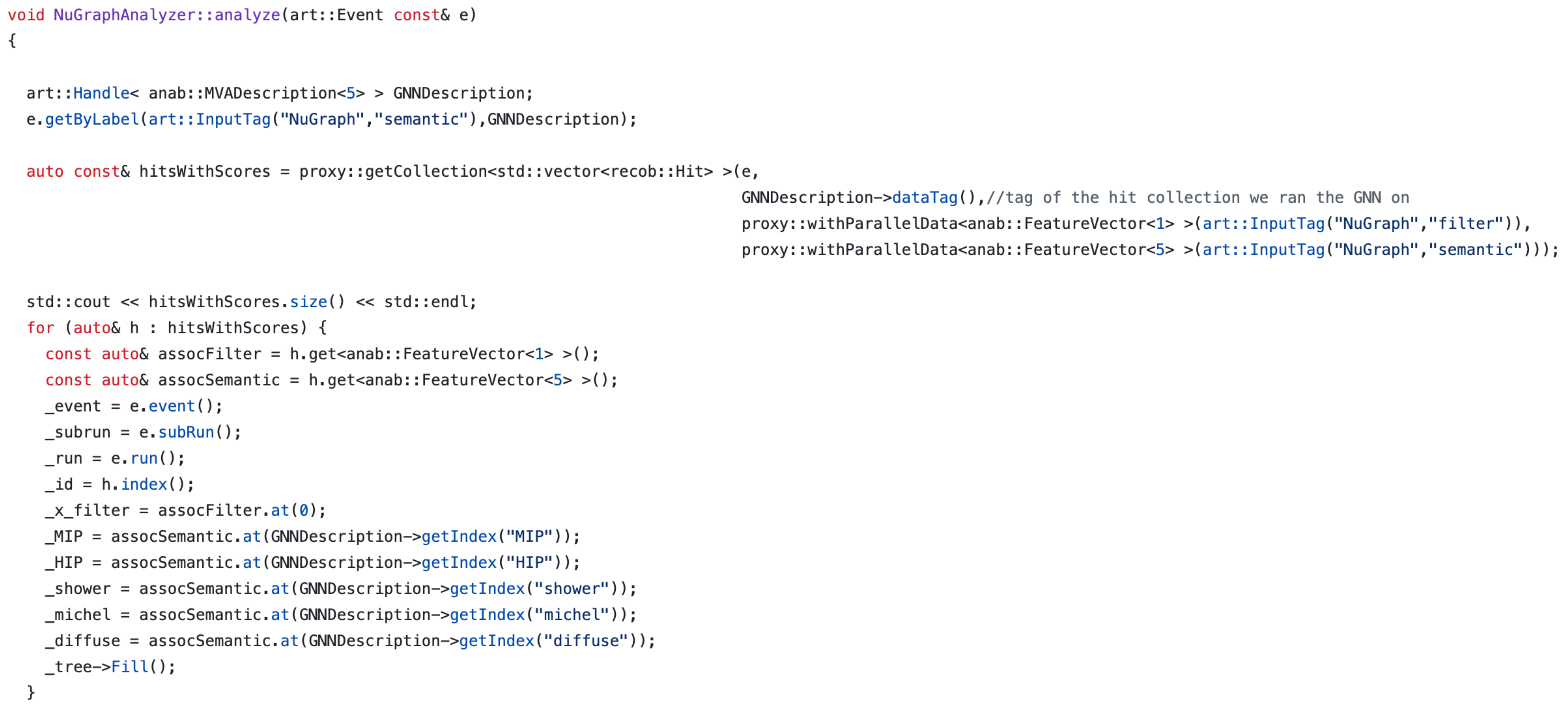
```
long int spdim = splist.size();
auto nexus = torch::empty({spdim,0},torch::dtype(torch::kFloat32));

std::vector<torch::jit::IValue> inputs;
inputs.push_back(x);
inputs.push_back(edge_index_plane);
inputs.push_back(edge_index_nexus);
inputs.push_back(nexus);
inputs.push_back(batch);
torch::jit::script::Module module = torch::jit::load("model.pt");
if (debug) std::cout << "FORWARD!" << std::endl;
auto outputs = module.forward(inputs).toGenericDict();
std::cout << "output =" << outputs << std::endl;
```

Store output in event record

- Use already available classes to store GNN output in event record:
  - FeatureVector<N> and MVADescription<N> for hit-level predictions
    - both defined in lardataobj/AnalysisBase/MVAOutput.h
    - FeatureVector contains the prediction scores, one entry per hit
      - result stored in same order as input hit collection (no Assns)
    - MVADescription contains the input label of the hit collection and a description of each entry in the FeatureVector. One entry per event.
  - Use recob::Vertex for vertex prediction

Analyze GNN output

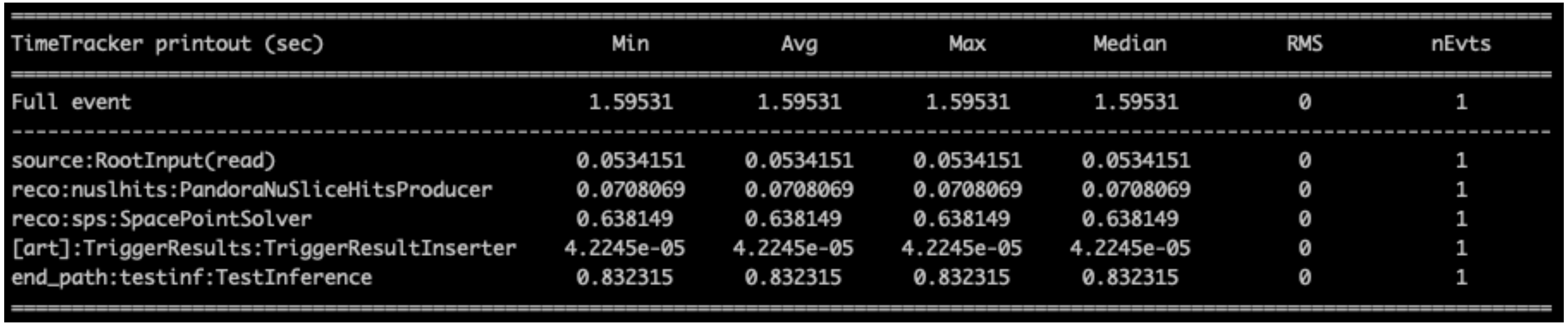
- Simple analyzer that reads back the GNN output from the event record and fills a root TTree with the results
  - leverages larsoft proxies and MVADescription for user-friendly access of results



```
void NuGraphAnalyzer::analyze(art::Event const& e)
{

  art::Handle< anab::MVADescription<5> > GNNDescription;
  e.getByLabel(art::InputTag("NuGraph","semantic"),GNNDescription);

  auto const& hitsWithScores = proxy::getCollection<std::vector<recob::Hit> >(e,
                                                                              GNNDescription->dataTag(),//tag of the hit collection we ran the GNN on
                                                                              proxy::withParallelData<anab::FeatureVector<1> >(art::InputTag("NuGraph","filter")),
                                                                              proxy::withParallelData<anab::FeatureVector<5> >(art::InputTag("NuGraph","semantic")));

  std::cout << hitsWithScores.size() << std::endl;
  for (auto& h : hitsWithScores) {
    const auto& assocFilter = h.get<anab::FeatureVector<1> >();
    const auto& assocSemantic = h.get<anab::FeatureVector<5> >();
    _event = e.event();
    _subrun = e.subRun();
    _run = e.run();
    _id = h.index();
    _x_filter = assocFilter.at(0);
    _MIP = assocSemantic.at(GNNDescription->getIndex("MIP"));
    _HIP = assocSemantic.at(GNNDescription->getIndex("HIP"));
    _shower = assocSemantic.at(GNNDescription->getIndex("shower"));
    _michel = assocSemantic.at(GNNDescription->getIndex("michel"));
    _diffuse = assocSemantic.at(GNNDescription->getIndex("diffuse"));
    _tree->Fill();
  }
```

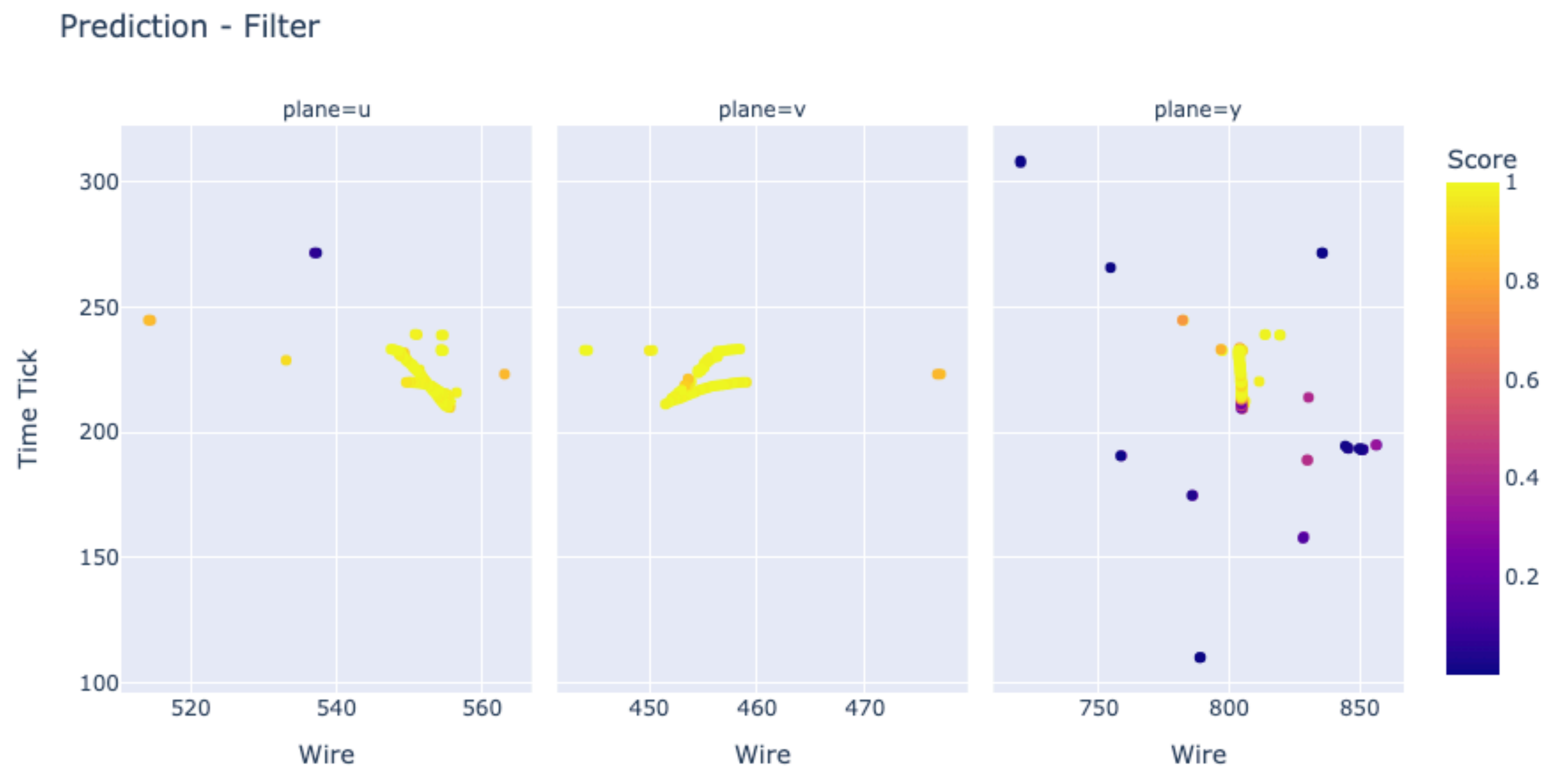
Running a single event

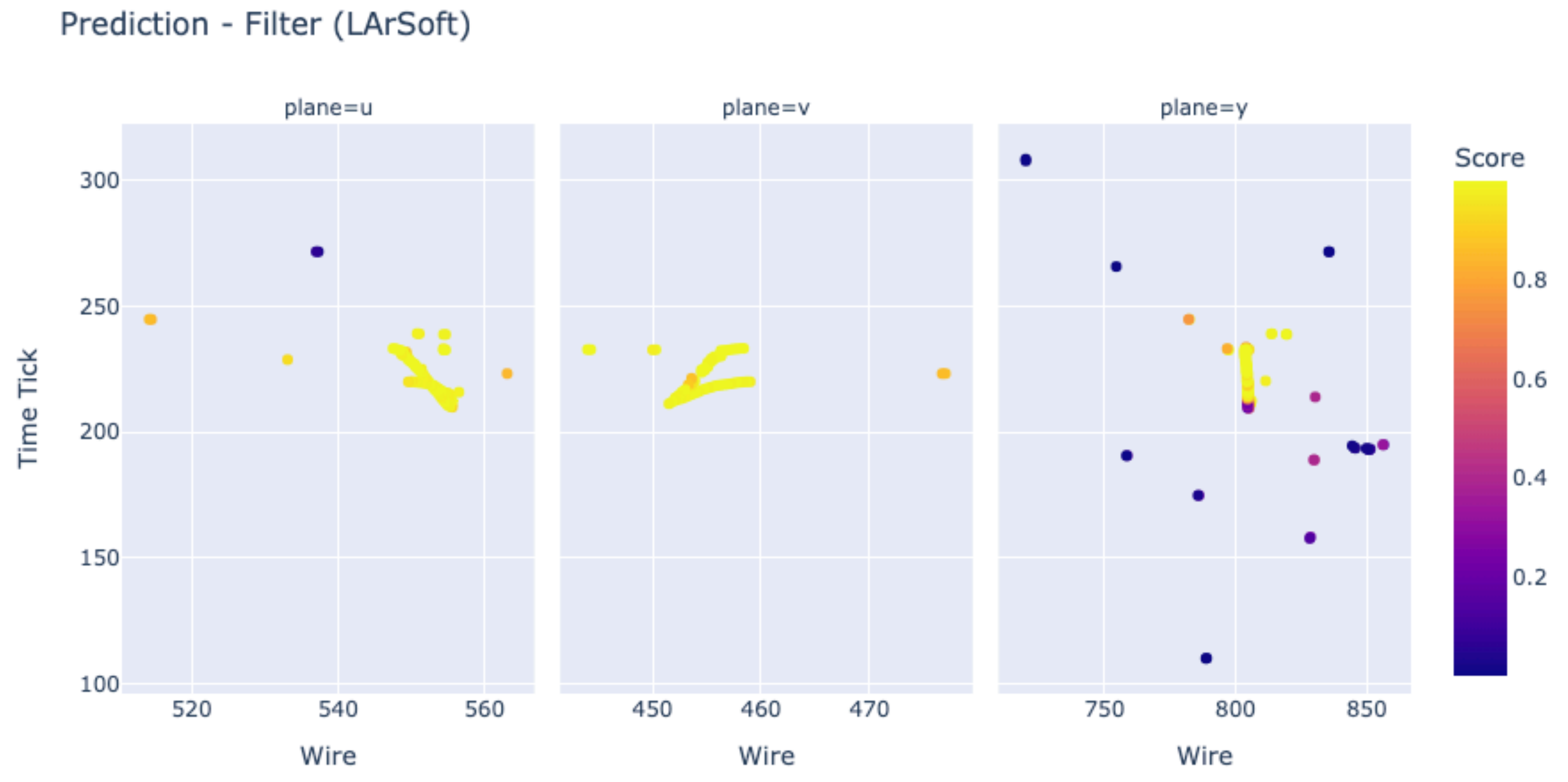
- Inference module takes 0.8 s on a single event, including graph construction
- Other upstream modules are also very fast
  - However, from experience with creating the training dataset, SpacePointSolver may take longer and use significant memory for busy events.
    - Exploring alternatives (e.g. Cluster3D) or a better tuning of the parameters may be required.



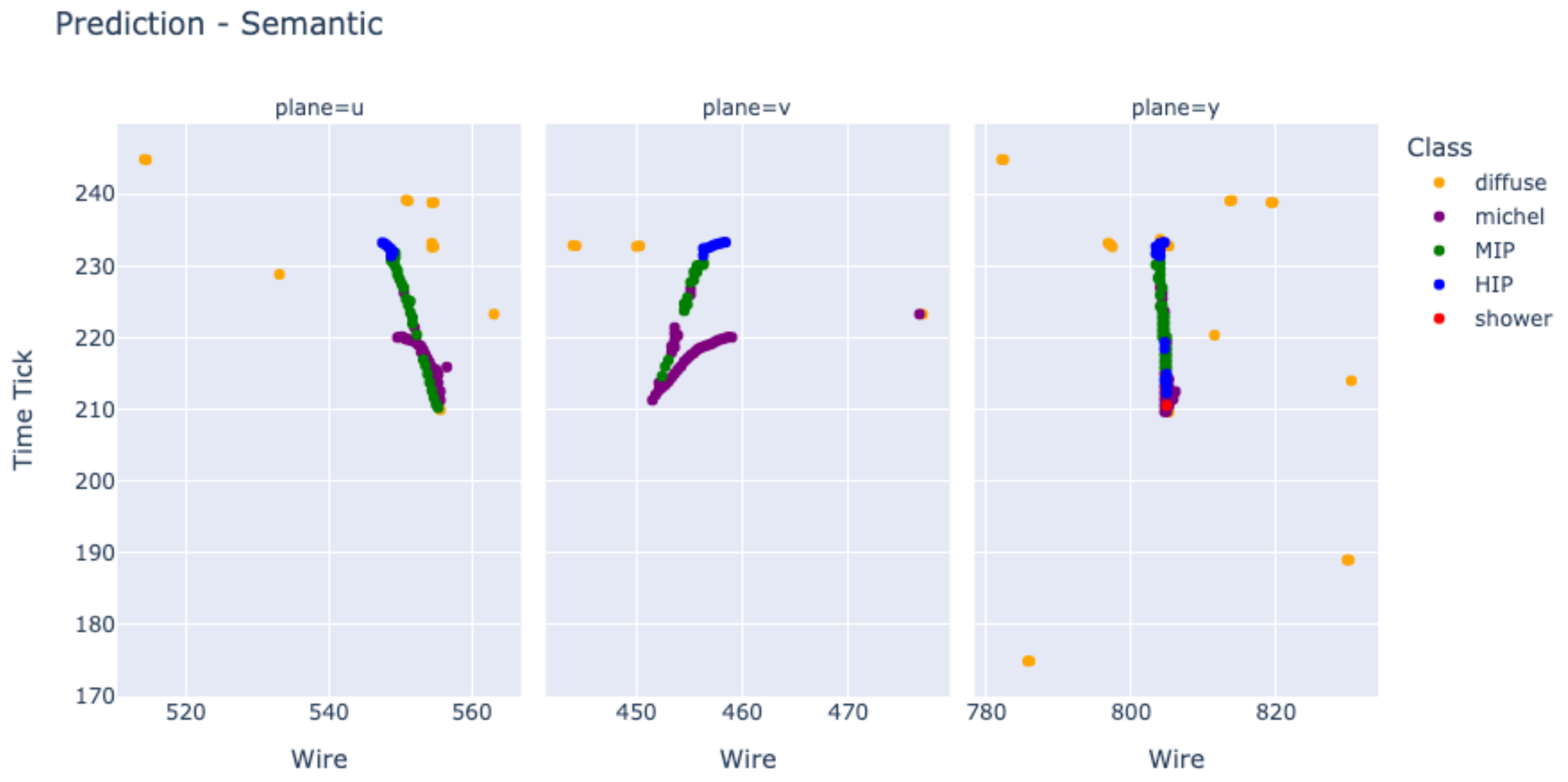
```
=================================================================================================================================
TimeTracker printout (sec)                       Min          Avg          Max          Median         RMS          nEvts
=================================================================================================================================
Full event                                     1.59531      1.59531      1.59531      1.59531          0              1
---------------------------------------------------------------------------------------------------------------------------------
source:RootInput(read)                         0.0534151    0.0534151    0.0534151    0.0534151        0              1
reco:nuslhits:PandoraNuSliceHitsProducer       0.0708069    0.0708069    0.0708069    0.0708069        0              1
reco:sps:SpacePointSolver                      0.638149     0.638149     0.638149     0.638149         0              1
[art]:TriggerResults:TriggerResultInserter    4.2245e-05   4.2245e-05   4.2245e-05   4.2245e-05        0              1
end_path:testinf:TestInference                 0.832315     0.832315     0.832315     0.832315         0              1
=================================================================================================================================
```

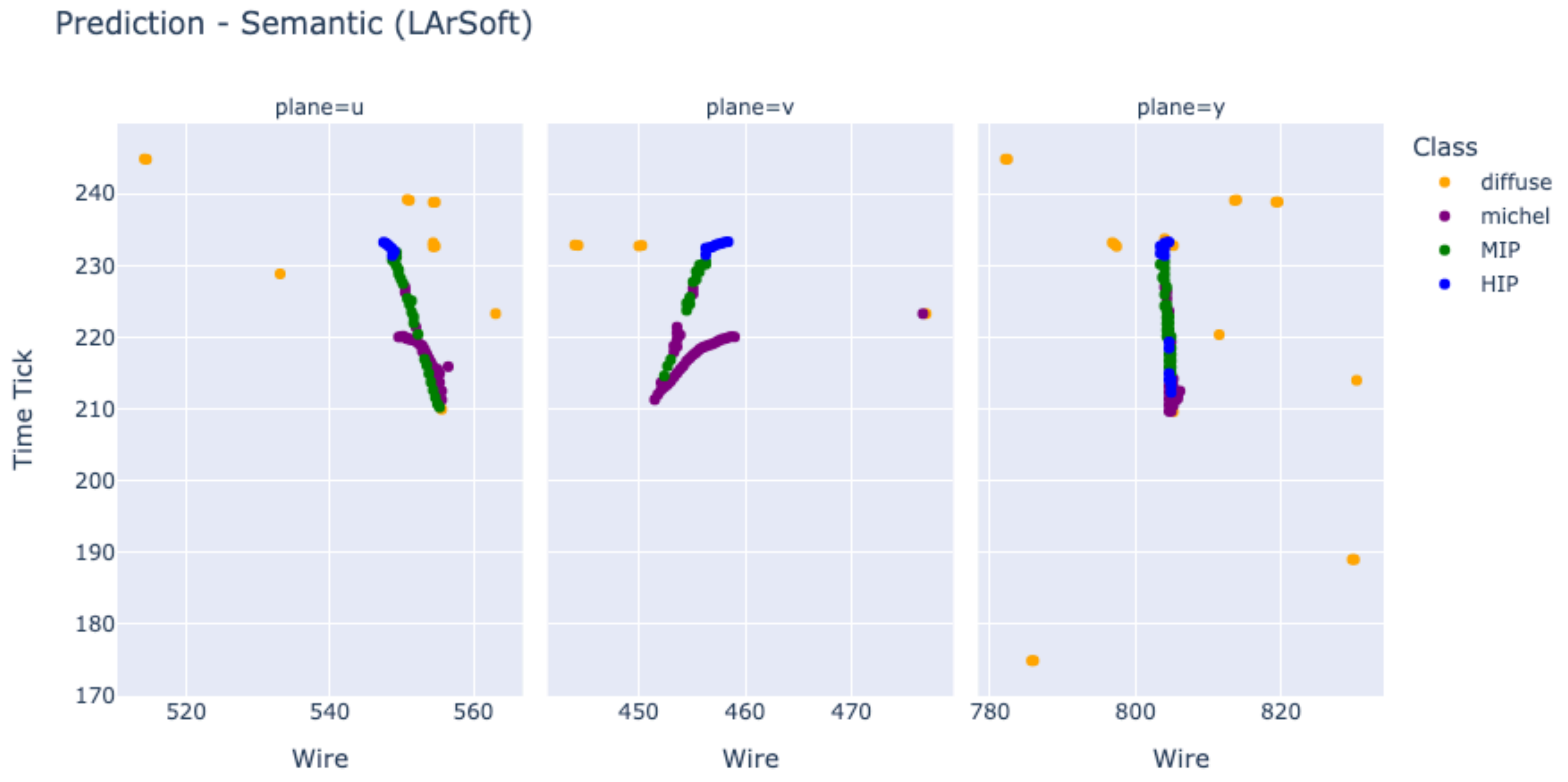
Output - filter decoder





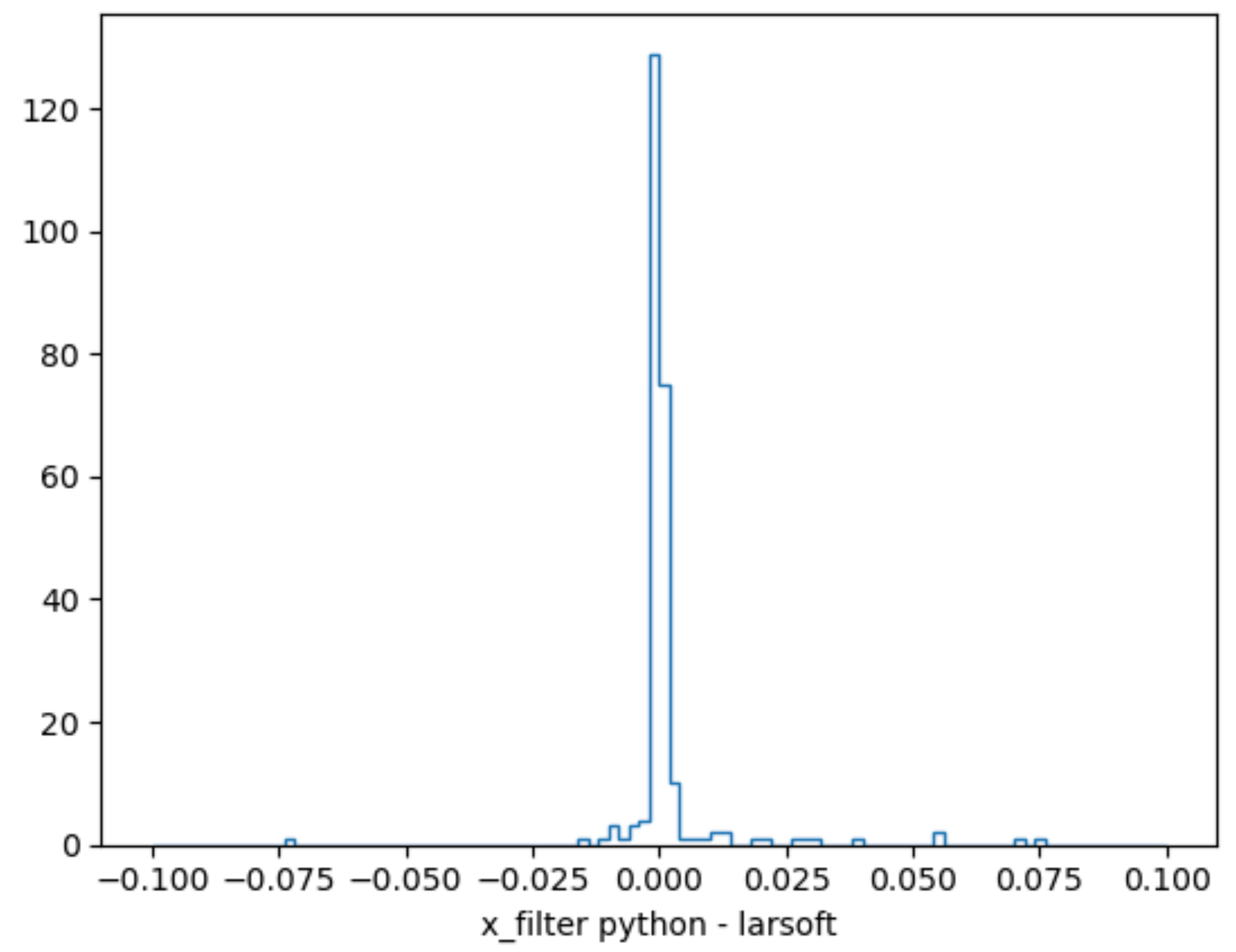
Output - semantic decoder

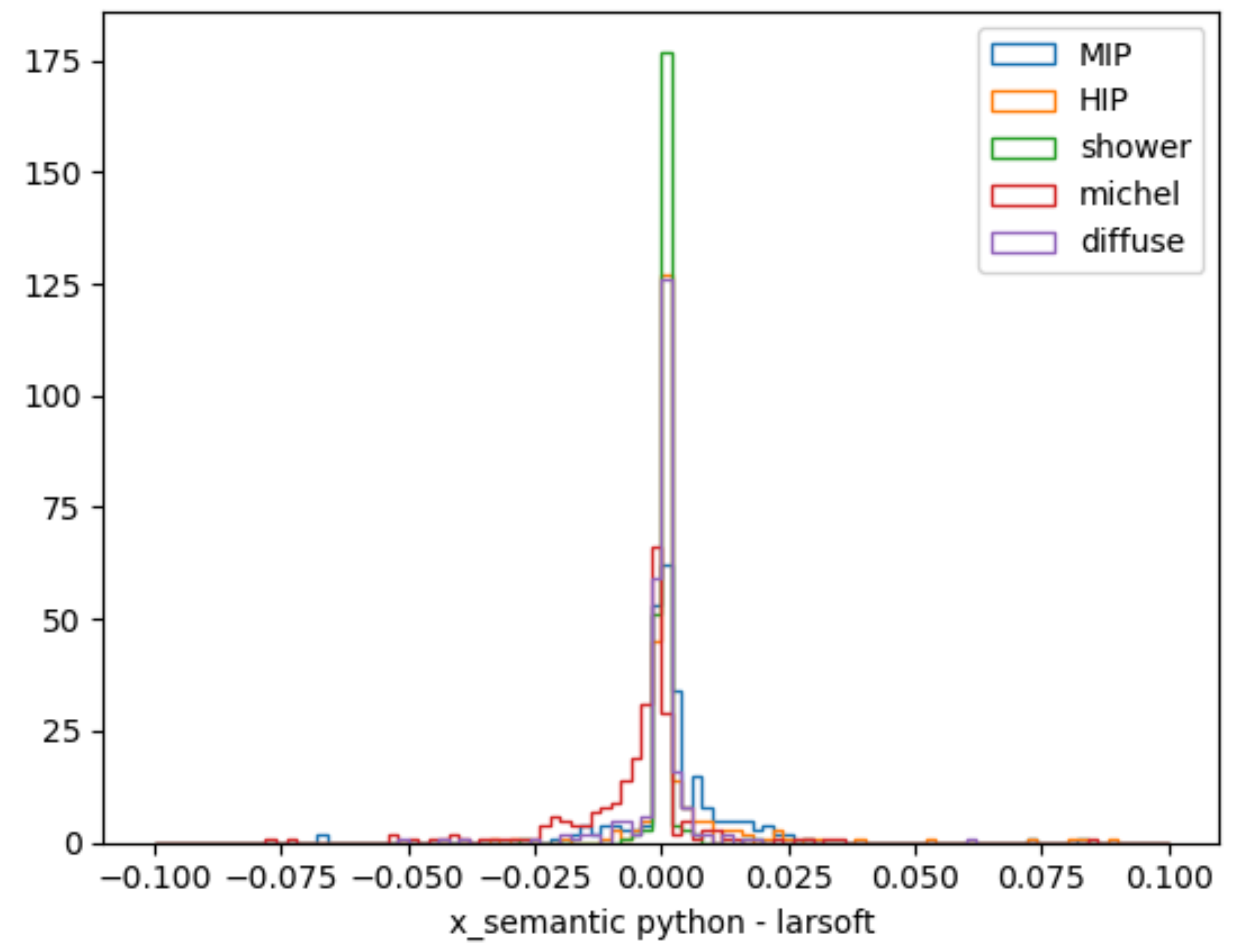




Output differences

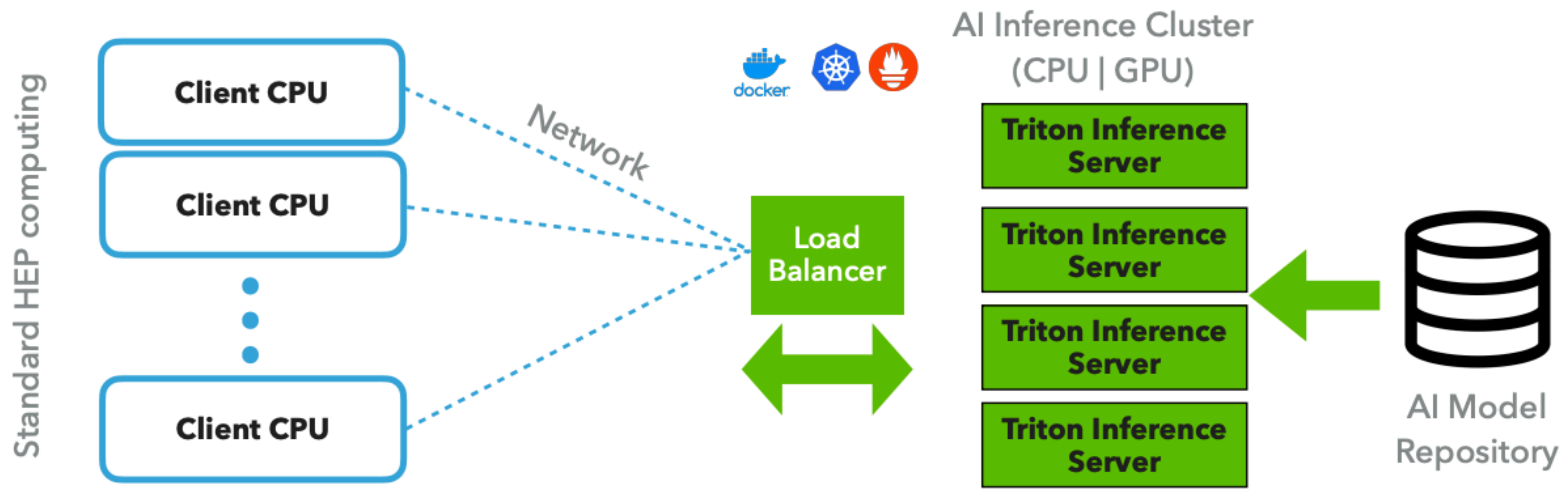
- The output on the tested event is meaningful and has the expected format
- The numerical result is close but not identical to the python setup
  - differences are typically in the 2nd or 3rd significant digit
  - torch version and inputs match (to numerical precision)
  - more work is needed to figure out the origin of the discrepancy, but not a concern for physics results





Other options: Triton/NuSonic

- Using libtorch directly in LArSoft works well for CPU-based workflows (fast inference time on CPU)
- Triton and NuSonic are an option for GPU workflows, or for CPU workflows in case we want to be independent from the distributed pytorch version
- How NuSonic works (arXiv:2009.04509):



- NuSonic supports pytorch models. And the LHC Exa.TrkX network was deployed on Triton as well. However due to constraints on the input format, some changes may be required on our code:
  - all tensors need to be in a single dictionary, while we currently have several different ones
    - see https://github.com/triton-inference-server/server/blob/main/docs/user_guide/model_configuration.md#inputs-and-outputs
- Sonic also should be able to support non-ML algorithms (e.g. Delaunay?) and is also working to support portability languages: see talk by K. Pedro

Conclusions

- NuGraph is a multi-task GNN for LArTPC reconstruction
  - currently trained on MicroBooNE open data
  - for more ideas and applications, see our recent workshop

- Integration in LArSoft through libtorch is in advanced state
  - plan to finalize with larrecodnn as soon as torch_scatter is available in ups

- Future plans include support for NuSonic inference

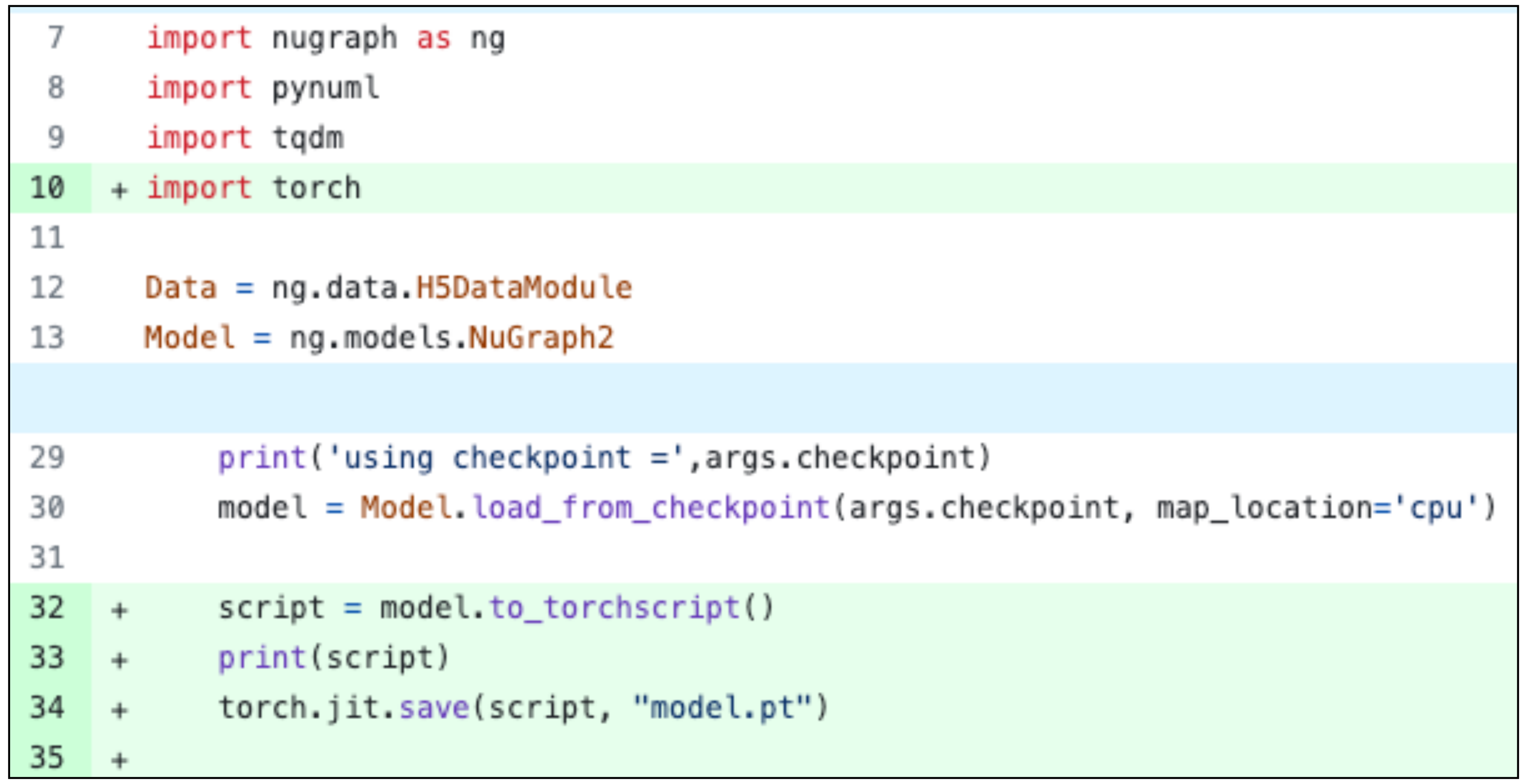
Backup

Saving the model in TorchScript with JIT

- In order to load the module in C++ we need to compile and save it with JIT
  - https://pytorch-geometric.readthedocs.io/en/latest/advanced/jit.html
- A modified (hacked) version of NuGraph does this:
  - https://github.com/exatrkx/NuGraph/compare/feature/cerati_jit-brute-force

- Thanks to V's work NuGraph was almost compatible with TorchScript already. Summary of changes required:
  - Remove checkpointing (not used in inference)
  - Avoid relying on inheritance of decoder classes from common base class
  - And some other changes:
    - add/fix annotations about return types
    - add ".jittable()" to various object instantiations
    - initializing objects to the correct type (i.e. not to None)
    - add "propagate_type" to MessagePassing
    - add "@torch.jit.unused" to functions that are not used
    - loop over self.net.items instead of self.net (with self.net = nn.ModuleDict())

scripts/test.py



```
  import nugraph as ng
  import pynuml
  import tqdm
+ import torch

  Data = ng.data.H5DataModule
  Model = ng.models.NuGraph2

      print('using checkpoint =',args.checkpoint)
      model = Model.load_from_checkpoint(args.checkpoint, map_location='cpu')

+     script = model.to_torchscript()
+     print(script)
+     torch.jit.save(script, "model.pt")
+
```