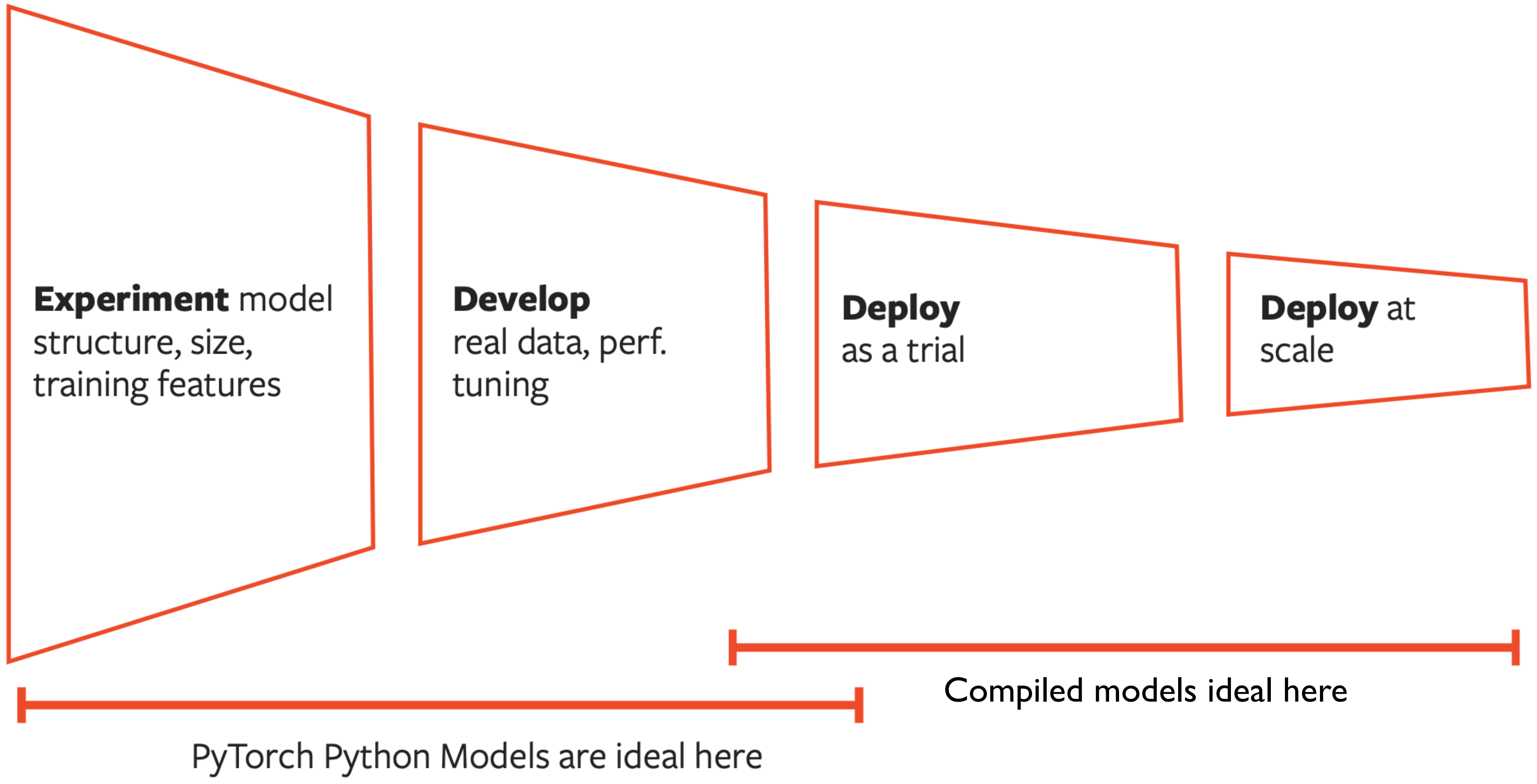
Graph Convolutional Operators in the PyTorch JIT

Lindsey Gray, Thomas Klijnsma (FNAL)
Matthias Fey (TU - Dortmund)

23 October 2020 - 4th CERN IML Workshop

The PyTorch JIT



Experiment model structure, size, training features
Develop real data, perf. tuning
Deploy as a trial
Deploy at scale
Compiled models ideal here
PyTorch Python Models are ideal here

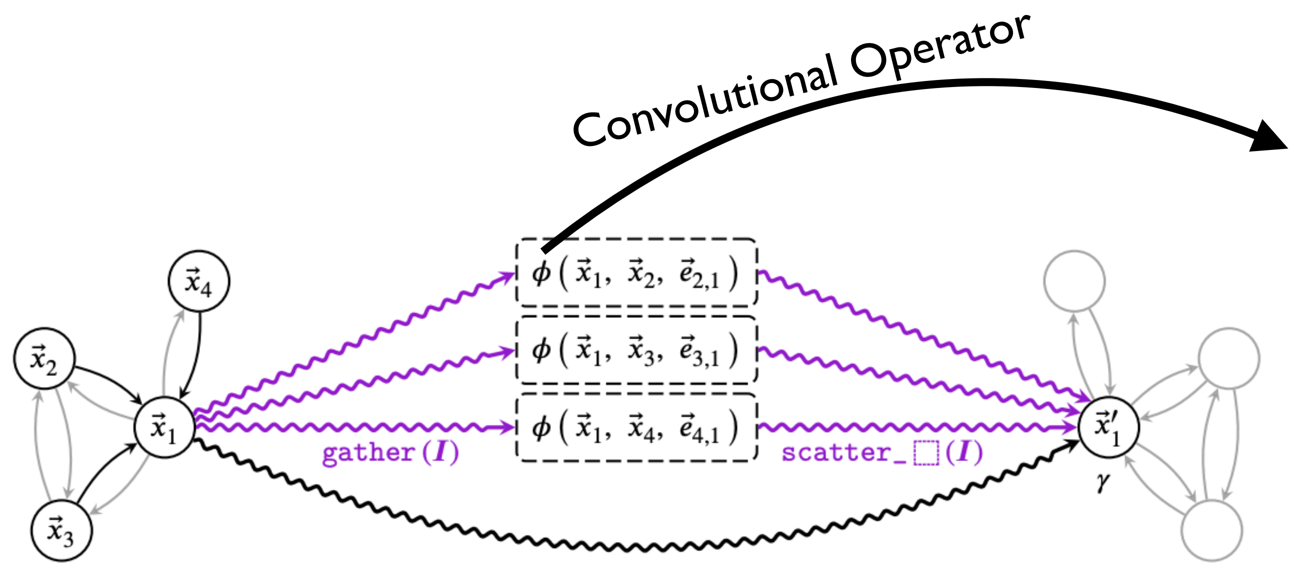
Ease of experimentation Python provides a flexible development platform to create and experiment with models.

Ease of deployment Just-in-time compiled models can be serialized, deployed, and optimized on any platform with a compiler backend.

Suitable for Graph NNs Graph neural networks (GNNs) are extremely fluid differentiable programs with rich control flow.

see: https://program-transformations.github.io/slides/pytorch_neurips.pdf

PyTorch Geometric


Convolutional Operator
$\phi(\vec{x}_1, \vec{x}_2, \vec{e}_{2,1})$
$\phi(\vec{x}_1, \vec{x}_3, \vec{e}_{3,1})$
$\phi(\vec{x}_1, \vec{x}_4, \vec{e}_{4,1})$
gather (I)
scatter_□ (I)
$\vec{x}_1$
$\vec{x}_2$
$\vec{x}_3$
$\vec{x}_4$
$\vec{x}_1'$
γ

```python
import torch
from torch.nn import Sequential as Seq, Linear as Lin, ReLU
from torch_geometric.nn import MessagePassing

class EdgeConv(MessagePassing):
    def __init__(self, F_in, F_out):
        super(EdgeConv, self).__init__(aggr='max')  # "Max" aggregation.
        self.mlp = Seq(Lin(2 * F_in, F_out), ReLU(), Lin(F_out, F_out))

    def forward(self, x, edge_index):
        # x has shape [N, F_in]
        # edge_index has shape [2, E]
        return self.propagate(edge_index, x=x)  # shape [N, F_out]

    def message(self, x_i, x_j):
        # x_i has shape [E, F_in]
        # x_j has shape [E, F_in]
        edge_features = torch.cat([x_i, x_j - x_i], dim=1)  # shape [E, 2 * F_in]
        return self.mlp(edge_features)  # shape [E, F_out]
```

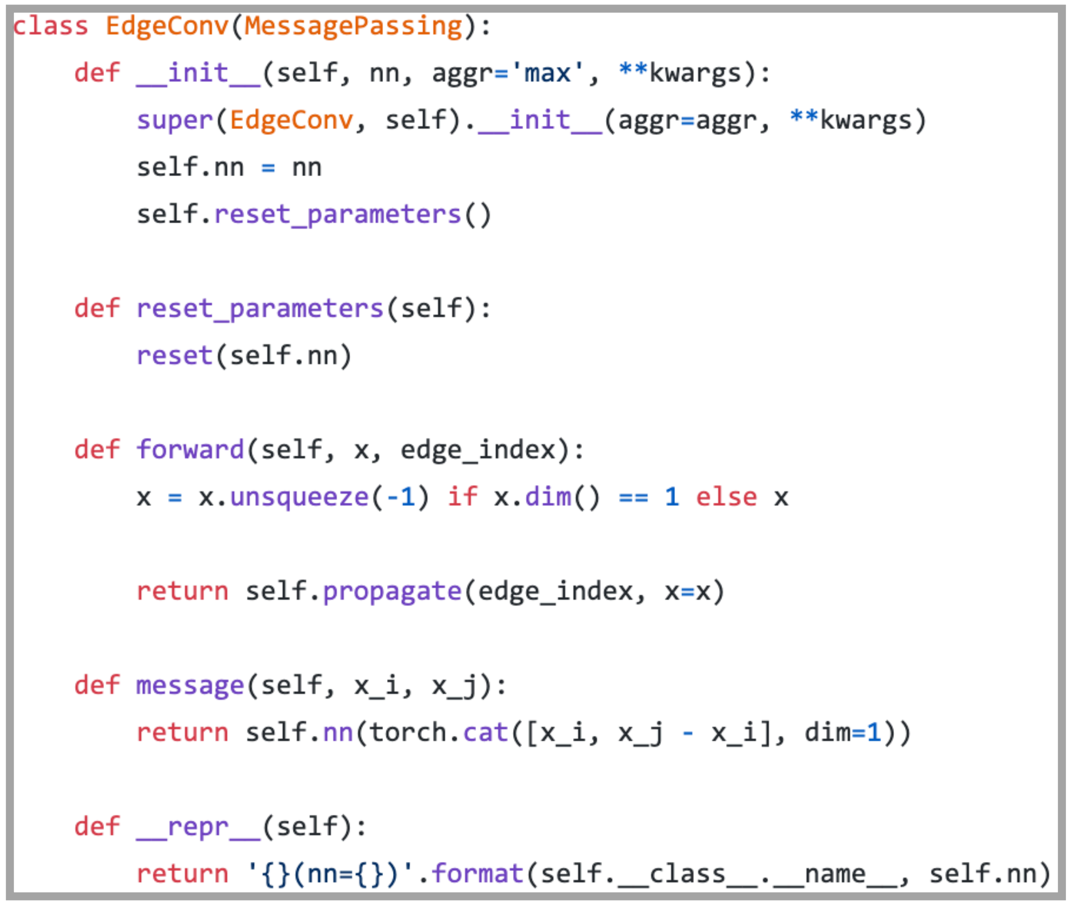
https://pytorch-geometric.readthedocs.io/

Follows PyTorch Principles   Pythonic and experimentation centric with clear API for concisely writing GNNs, backed by well-performing code.

Well Adopted   100s of citations and over 1000 forks to date, with an active community of contributors.

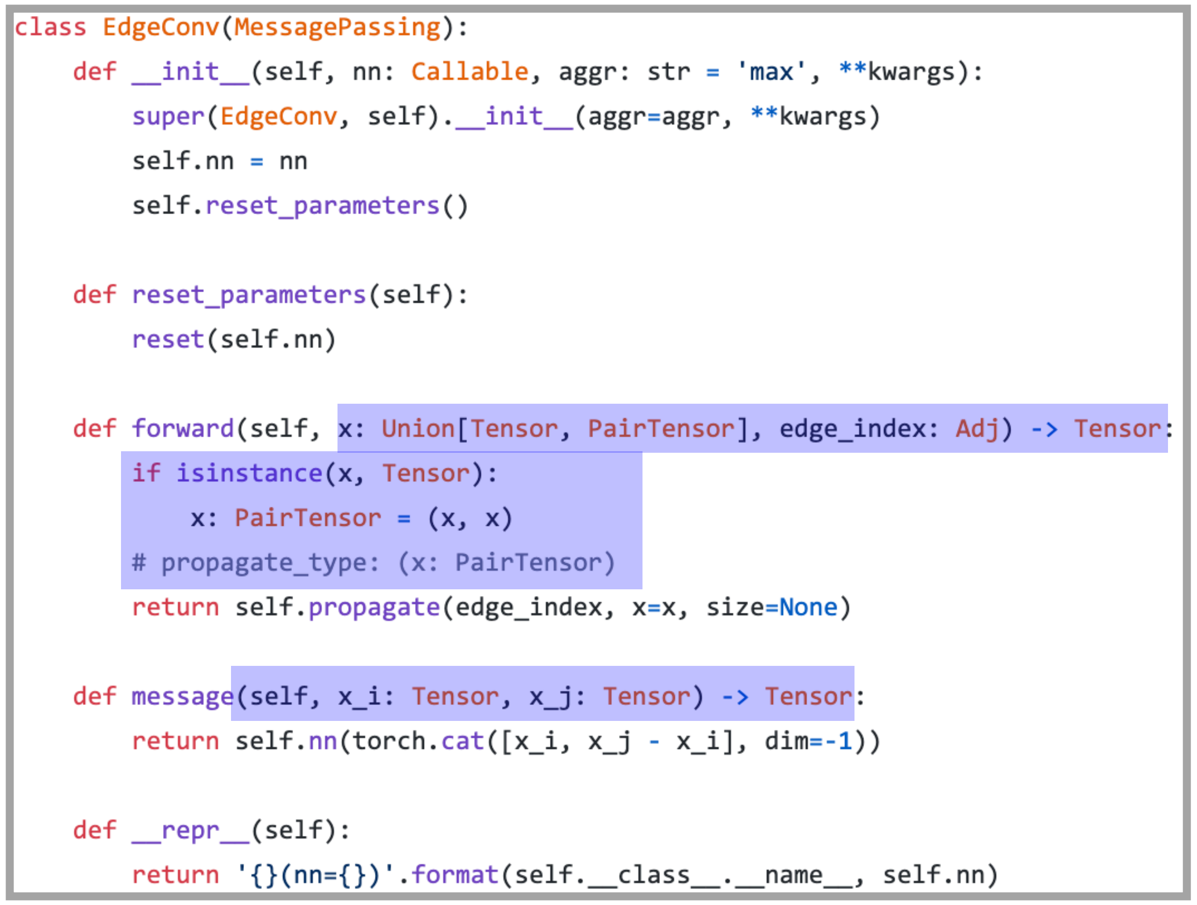
Incredibly flexible   Can implement graph convolutional operators with variety of inputs/outputs. So flexible that Torch-JIT *couldn't* handle it!

JIT'ing PyTorch Geometric Operators



```python
class EdgeConv(MessagePassing):
    def __init__(self, nn, aggr='max', **kwargs):
        super(EdgeConv, self).__init__(aggr=aggr, **kwargs)
        self.nn = nn
        self.reset_parameters()

    def reset_parameters(self):
        reset(self.nn)

    def forward(self, x, edge_index):
        x = x.unsqueeze(-1) if x.dim() == 1 else x

        return self.propagate(edge_index, x=x)

    def message(self, x_i, x_j):
        return self.nn(torch.cat([x_i, x_j - x_i], dim=1))

    def __repr__(self):
        return '{}(nn={})'.format(self.__class__.__name__, self.nn)
```

'taadah' :)



```python
class EdgeConv(MessagePassing):
    def __init__(self, nn: Callable, aggr: str = 'max', **kwargs):
        super(EdgeConv, self).__init__(aggr=aggr, **kwargs)
        self.nn = nn
        self.reset_parameters()

    def reset_parameters(self):
        reset(self.nn)

    def forward(self, x: Union[Tensor, PairTensor], edge_index: Adj) -> Tensor:
        if isinstance(x, Tensor):
            x: PairTensor = (x, x)
        # propagate_type: (x: PairTensor)
        return self.propagate(edge_index, x=x, size=None)

    def message(self, x_i: Tensor, x_j: Tensor) -> Tensor:
        return self.nn(torch.cat([x_i, x_j - x_i], dim=-1))

    def __repr__(self):
        return '{}(nn={})'.format(self.__class__.__name__, self.nn)
```

+ EdgeConv().jittable() call in model code.

Use Python Type-hinting Stay pythonic as possible while making inputs and outputs of operators more concrete so they can be analyzed.

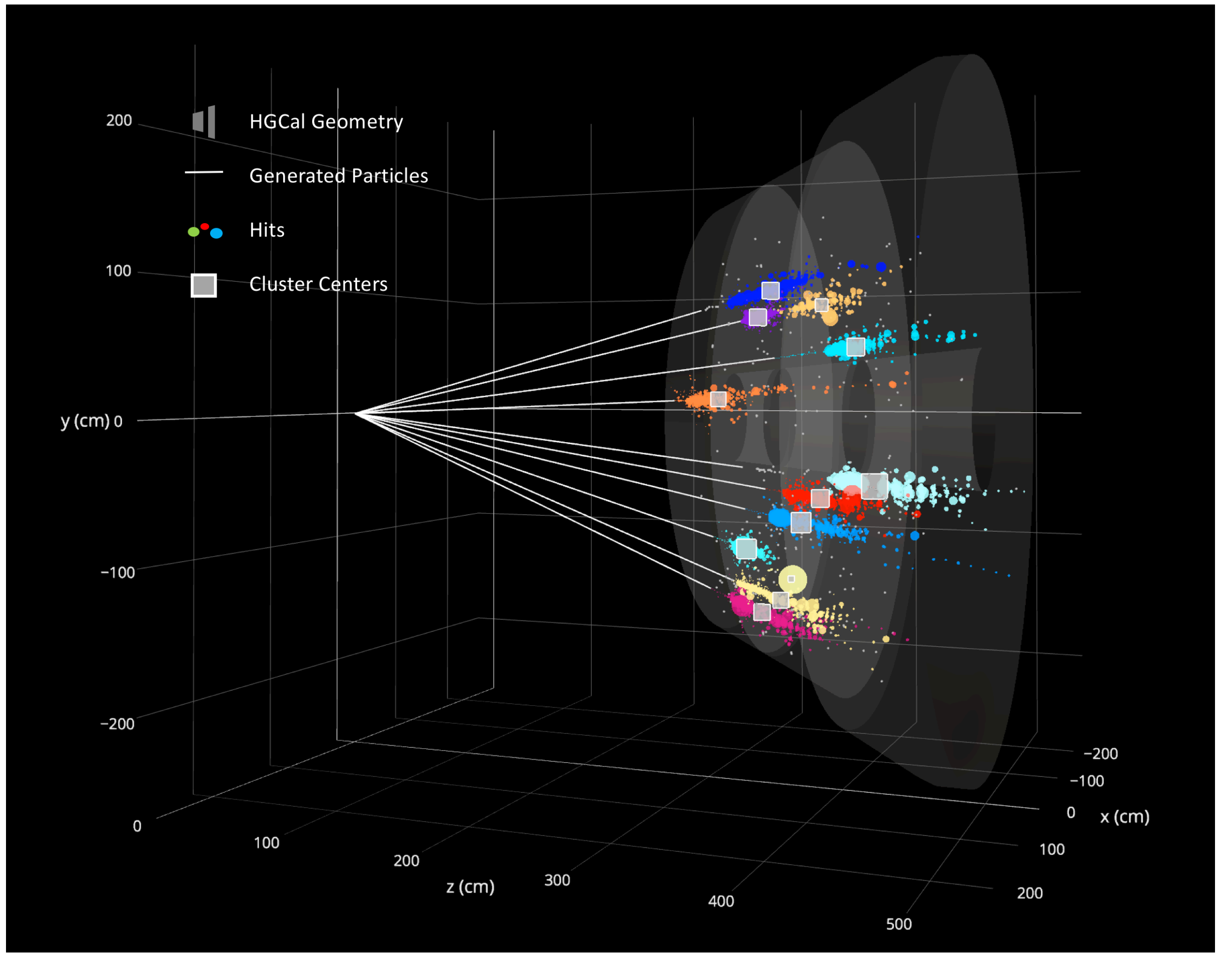
Simple Static Analysis At runtime, dynamically rewrite user code into jit-friendly version of itself while maintaining expected operation.

Minimal Model Changes For a model to be JIT compatible, all that needs to be done is to call '.jittable()' when constructing the operator!

First Results from Deployment

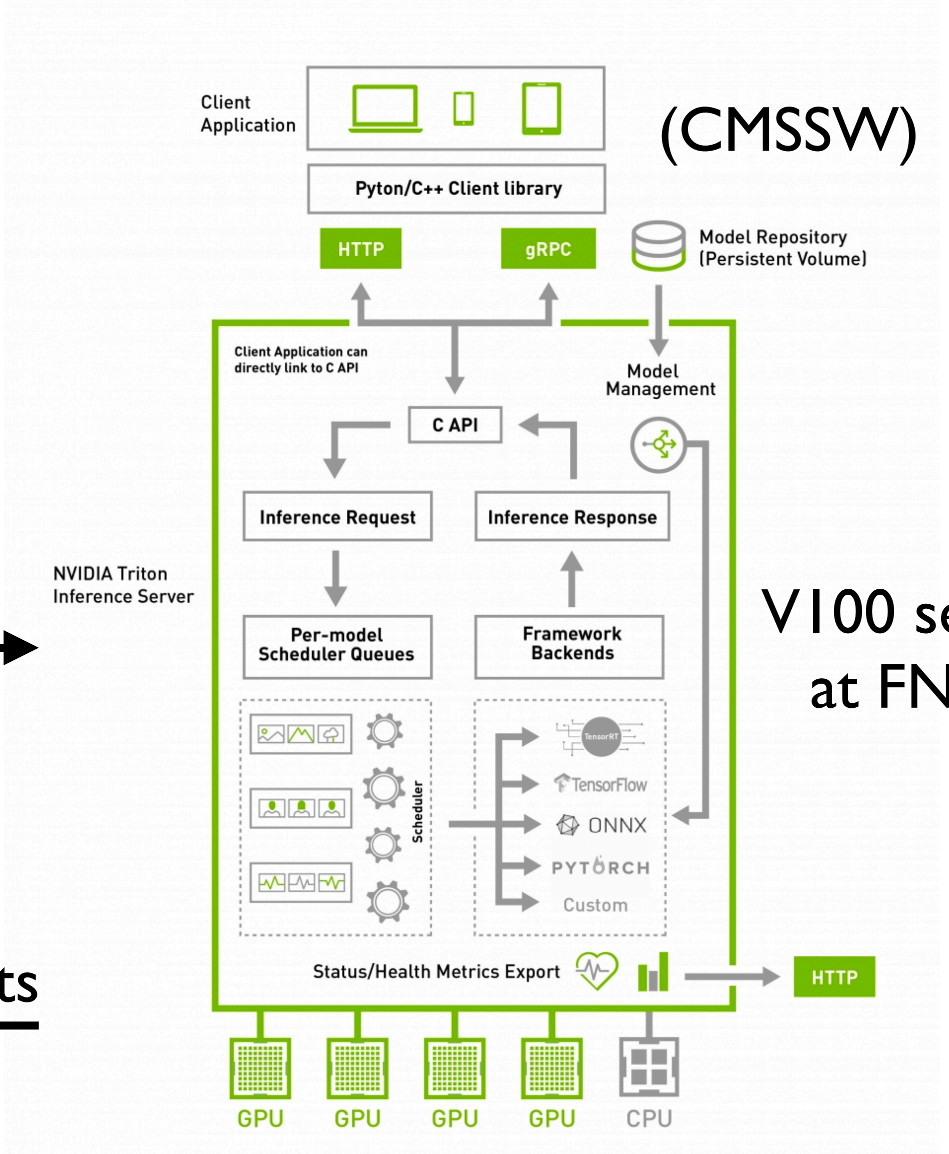
Task - proof of concept clustering for an imaging calorimeter (HGCAL).

Experiment software - CMSSW
No pytorch and no native GPU support for DL frameworks.



hit data →

← cluster assignments



(CMSSW)

V100 server at FNAL

Seamless GPU Integration Via nVidia Triton Inference as a Service (IaaS) engine, hardware used as available. ~100x speedup on GPU for GNN.

Framework independence Did not need to integrate PyTorch into CMSSW (experiment software), instead use abstraction layer for IaaS.

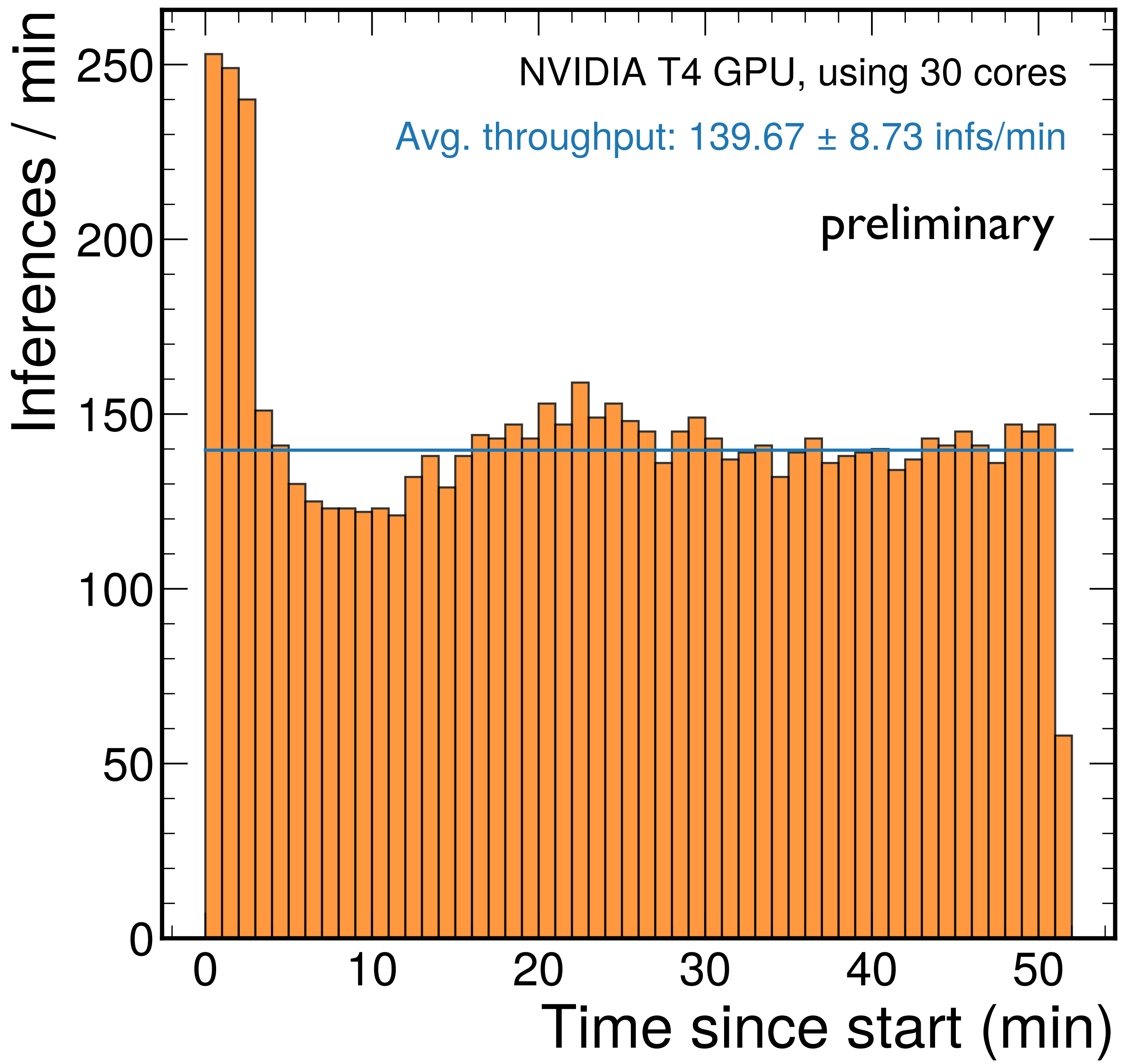
model: https://github.com/tklijnsma/hgcal_ldrd/blob/dev-jittable/src/models/EdgeNetWithCategories.py#L68
expt. sw: https://github.com/cms-sw/cmssw/tree/master/HeterogeneousCore/SonicTriton

Concluding Remarks

- GNNs becoming more widely used tools in HEP
  - highly effective at processing point clouds
  - can successfully describe reconstruction tasks in HEP
  - GNNs in PyTorch widely accessible via PyTorch Geometric
- The PyTorch jit makes deploying models easy and portable
- Moving from GNN research to deployment can be difficult
  - Wrote extensions to PyTorch Geometric which make this transition automated and easy, check out the example
  - All currently implemented (~30) convolutional operators supported
    - straightforward to add more operators if you need something special
- Inference as a service tools like nVidia Triton make deployment of neural networks scalable and easy
  - Additions to PyG layered on top of typical triton container
  - Scalable event processing on GPUs in experiment software

Extras

First Results on Throughput


NVIDIA T4 GPU, using 30 cores
Avg. throughput: 139.67 ± 8.73 infs/min
preliminary
Inferences / min
Time since start (min)