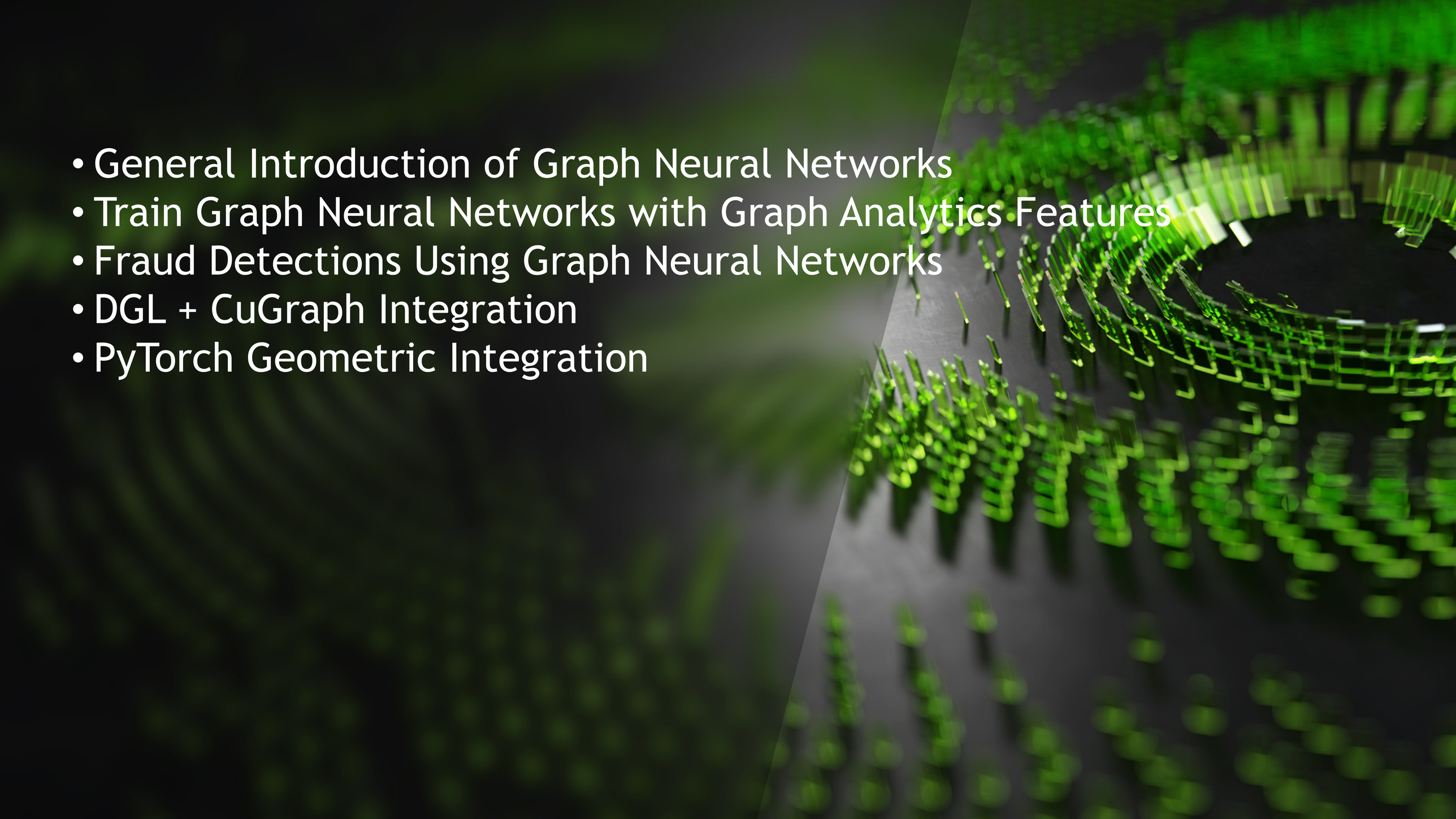
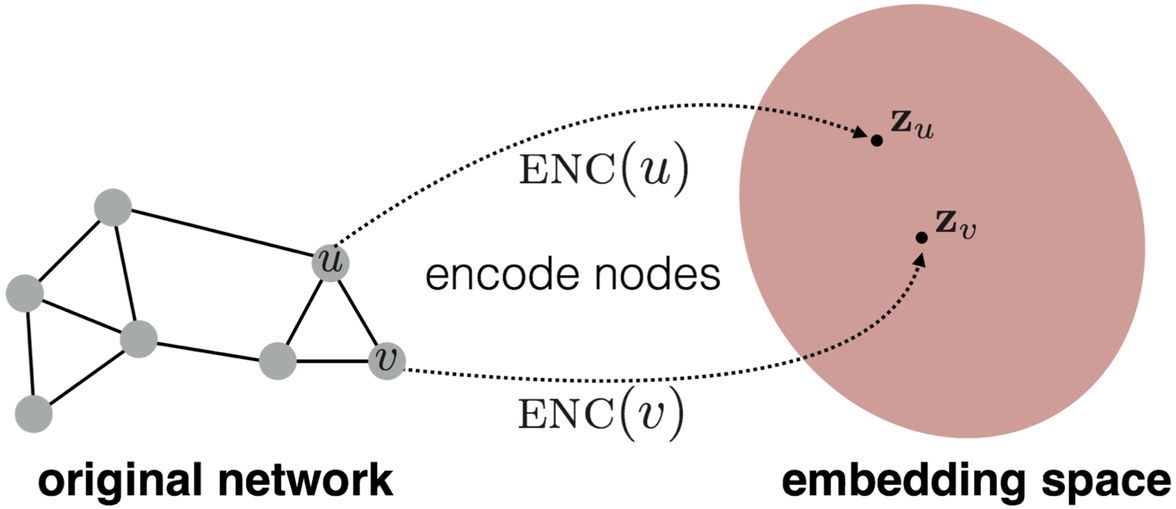
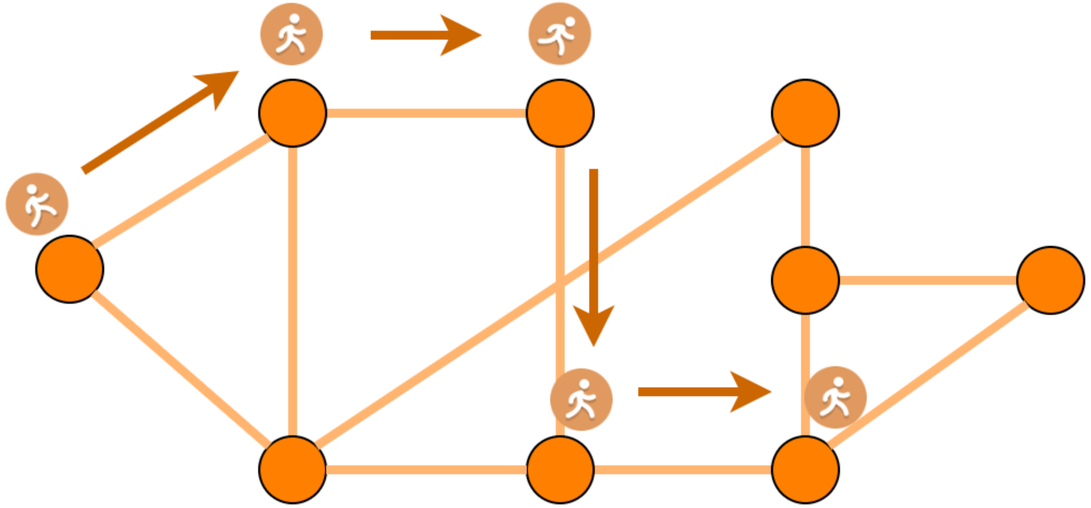
ACCELERATED GNN TRAINING WITH DGL AND RAPIDS CUGRAPH IN A FRAUD DETECTION WORKFLOW

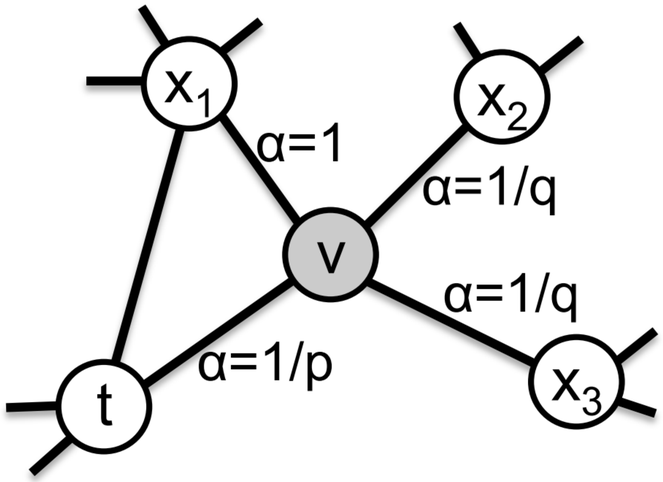
- General Introduction of Graph Neural Networks
- Train Graph Neural Networks with Graph Analytics Features
- Fraud Detections Using Graph Neural Networks
- DGL + CuGraph Integration
- PyTorch Geometric Integration

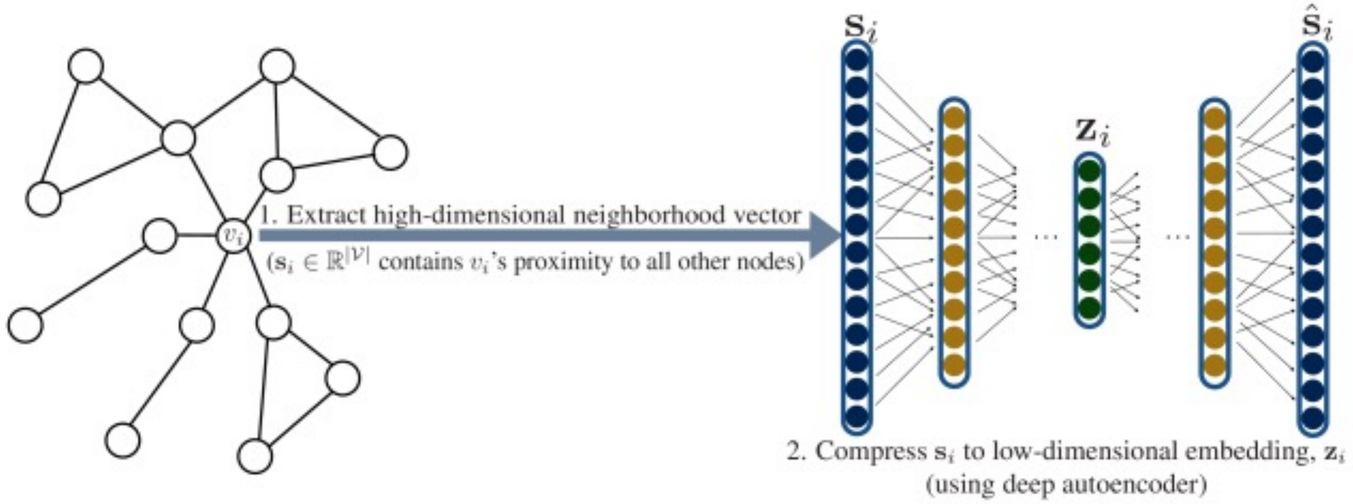


FROM GRAPH EMBEDDING TO GRAPH NEURAL NETWORKS





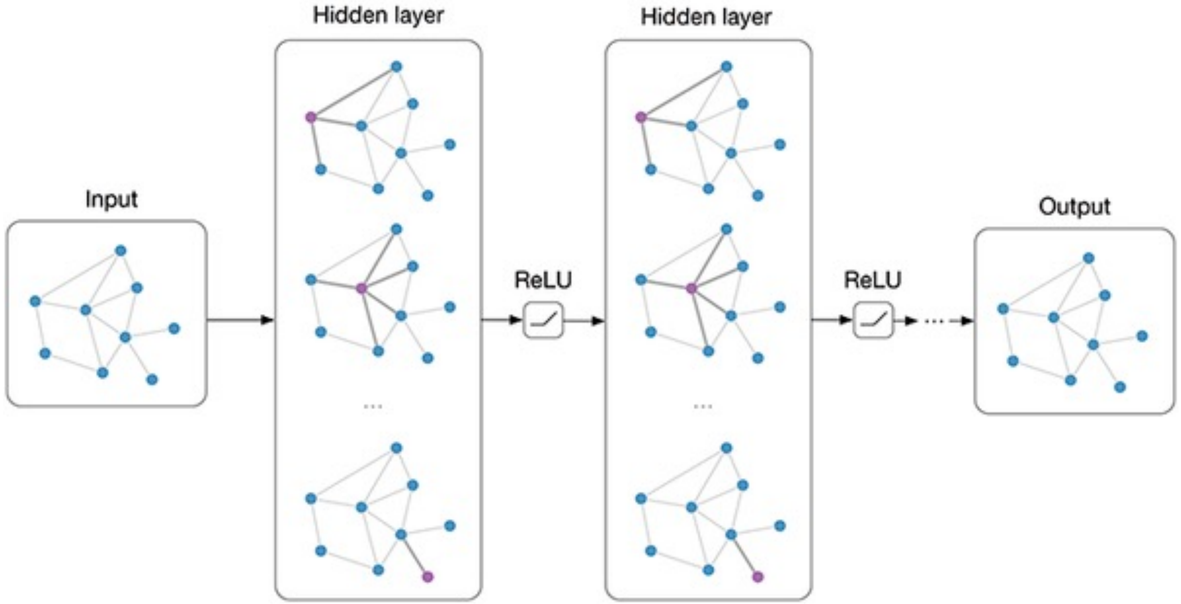




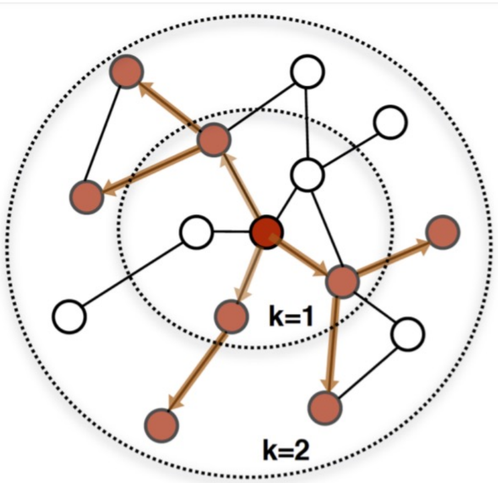
[1] node2vec: Scalable Feature Learning for Networks. A. Grover, J. Leskovec. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2016.
[2] DeepWalk: Online Learning of Social Representations. Perozzi, Bryan and Al-Rfou, Rami and Skiena, Steven (KDD), 2014
[3] https://snap-stanford.github.io/cs224w-notes/machine-learning-with-networks/node-representation-learning

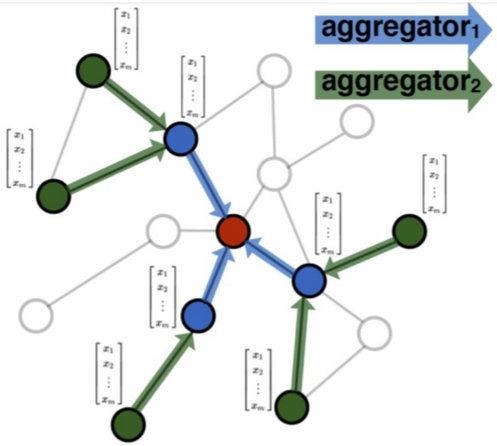
WHAT IS A GRAPH NEURAL NETWORKS?

Original GCN



GraphSAGE



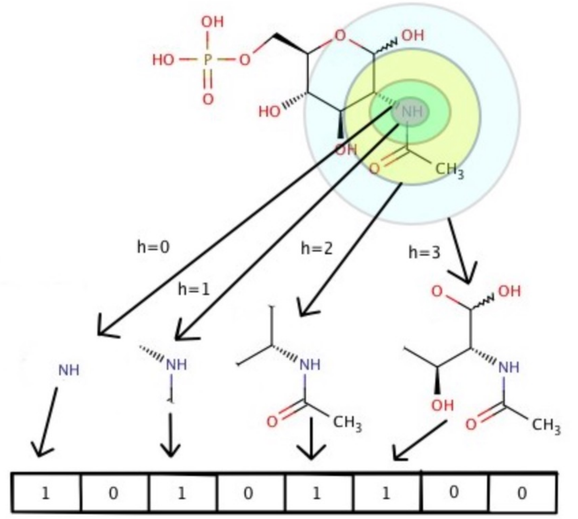


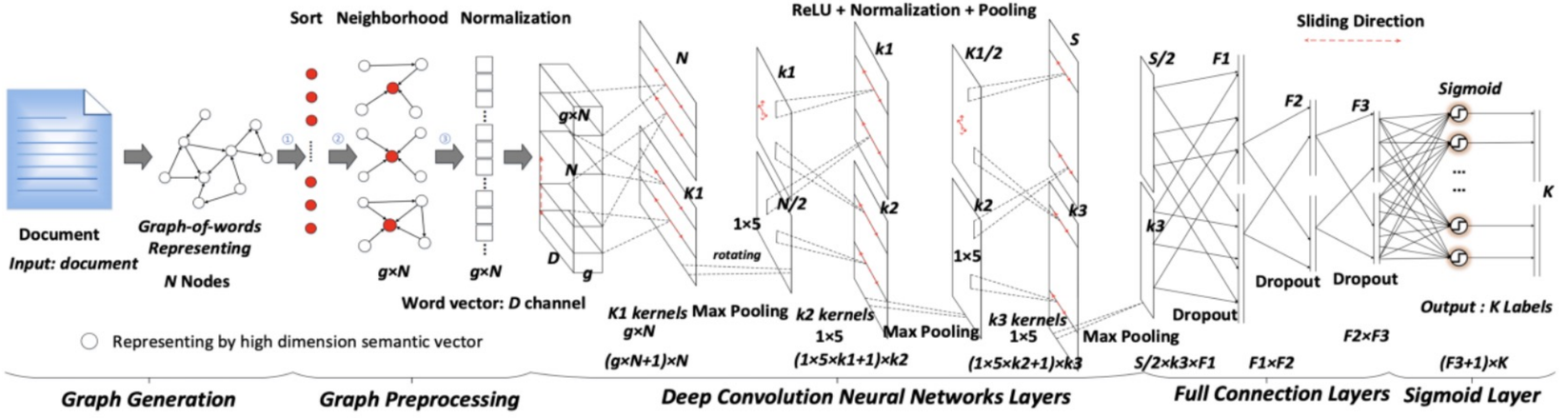
[1] https://arxiv.org/abs/1609.02907
[2] Hamilton, W. L., *et al*.. (2017). Inductive representation learning on large graphs. *NIPS*

EXAMPLES OF GNN APPLICATIONS

(a) (b) (c)

Figure 1: From left to right, the image to be converted into a RAG (a), the image with the superpixel segmentation being shown (b), and the image with the superpixel segmentation and the generated region adjacency graph overlayed on top of it (c).



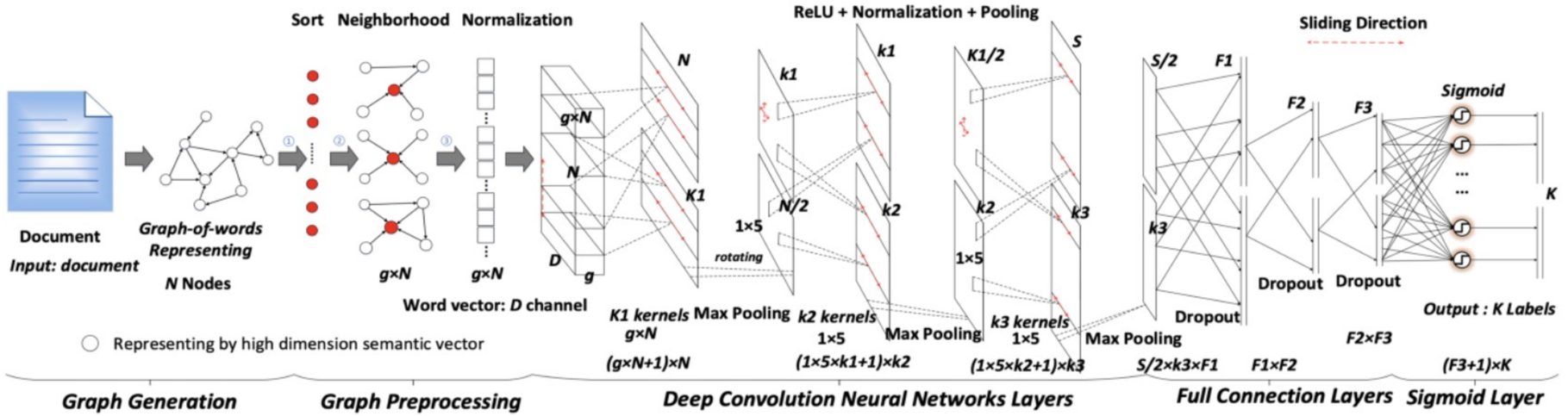


[1]https://arxiv.org/abs/2002.05544 Source

[2]https://towardsdatascience.com/https-medium-com-aishwaryajadhav-applications-of-graph-neural-networks-1420576be574#

[3]https://www.cse.ust.hk/~yqsong/papers/2018-WWW-Text-GraphCNN.pdf

EXAMPLE WORKFLOW OF GNN

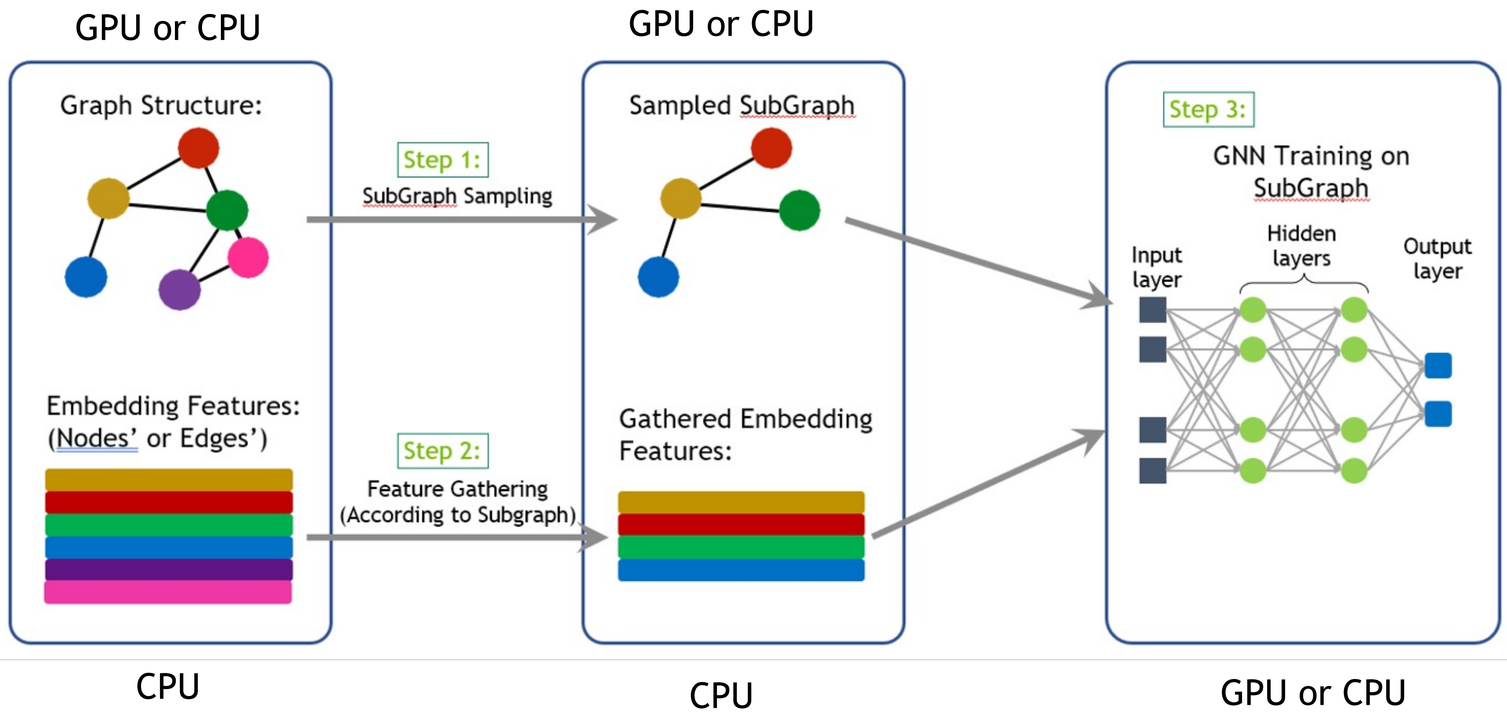


Source

[3]https://www.cse.ust.hk/~yqsong/papers/2018-WWW-Text-GraphCNN.pdf

COMPUTATION PATTERN

- Graph sampling and gathering features can consume 50% - 80% of total training time.



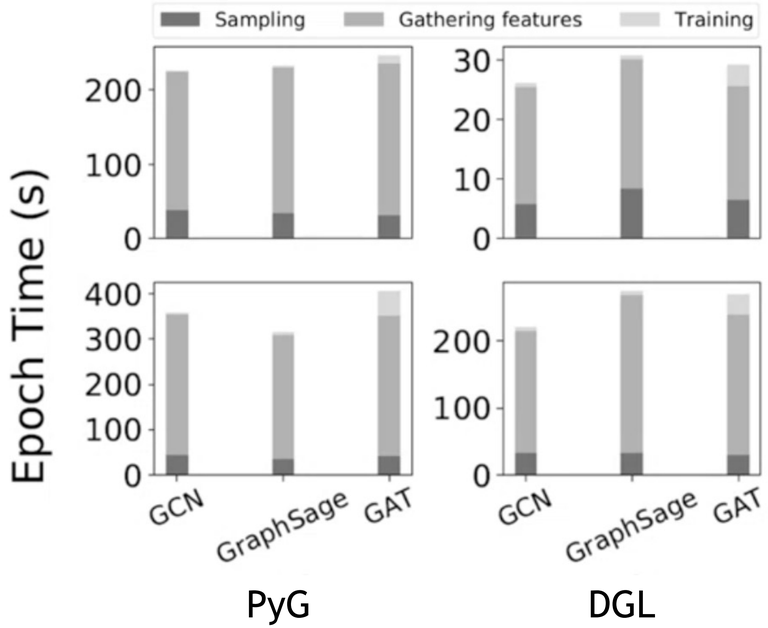
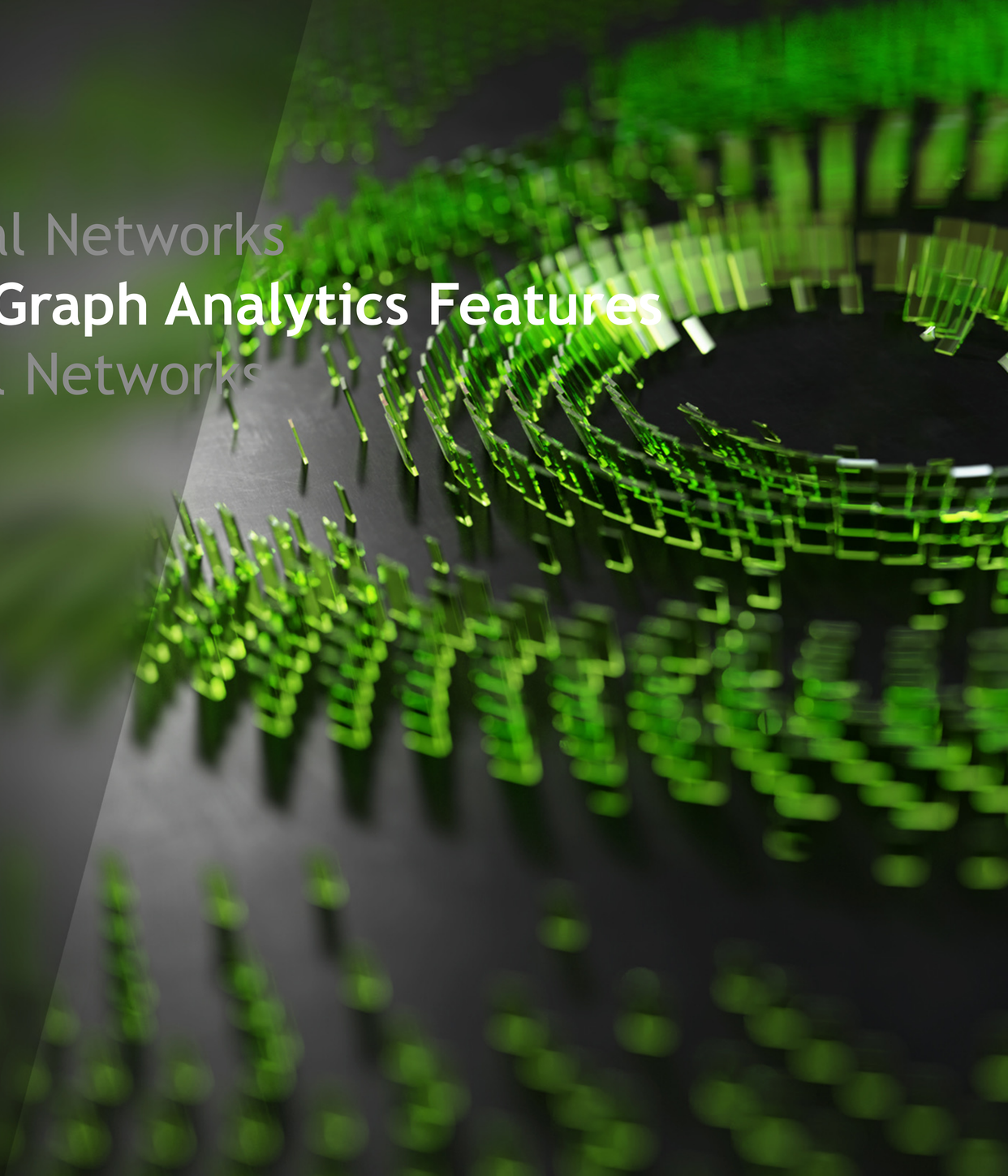


Fig 2 and 3 credit to Xin Yao

- General Introduction of Graph Neural Networks
- **Train Graph Neural Networks with Graph Analytics Features**
- Fraud Detections using Graph Neural Networks
- DGL + CuGraph Integration
- PyTorch Geometric Integration

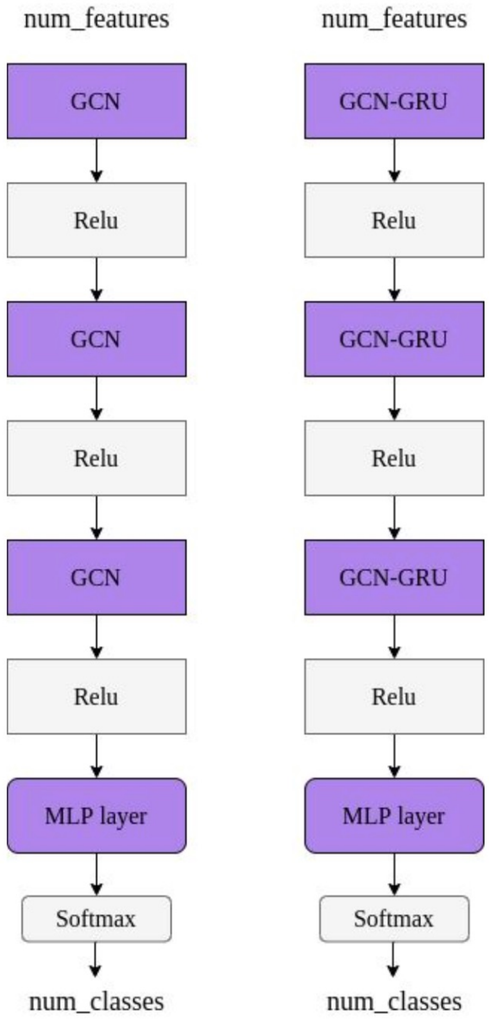


USING GRAPH ANALYTICS INSIDE GNN

- Graph Analytics metrics can be used in GNN to enrich the features of the original data
- CuGraph supports single-GPU and multi-GPU graph analytics computing
- Frequently used Graph Analytics features in GNN are:
  - Pagerank
  - Betweenness centrality
  - Katz centrality
  - Node degree
  - Jaccard Similarity
  - Dice-Sorenson coefficient

BOTNET DETECTION WITH CYBER DATA

Application of Using Graph Analytics Features in GNNs on Netflow Data



1. Duration (mean and std)
2. Unique number of ip addresses contacted
3. Unique number of protocols used (mean and std)
4. Unique number of port used (mean and std)
5. Total Packets (mean and std)
6. Total Bytes (mean and std)
7. Pagerank
8. Betweenness centrality
9. Katz centrality
10. Node degree

CTU-13 dataset

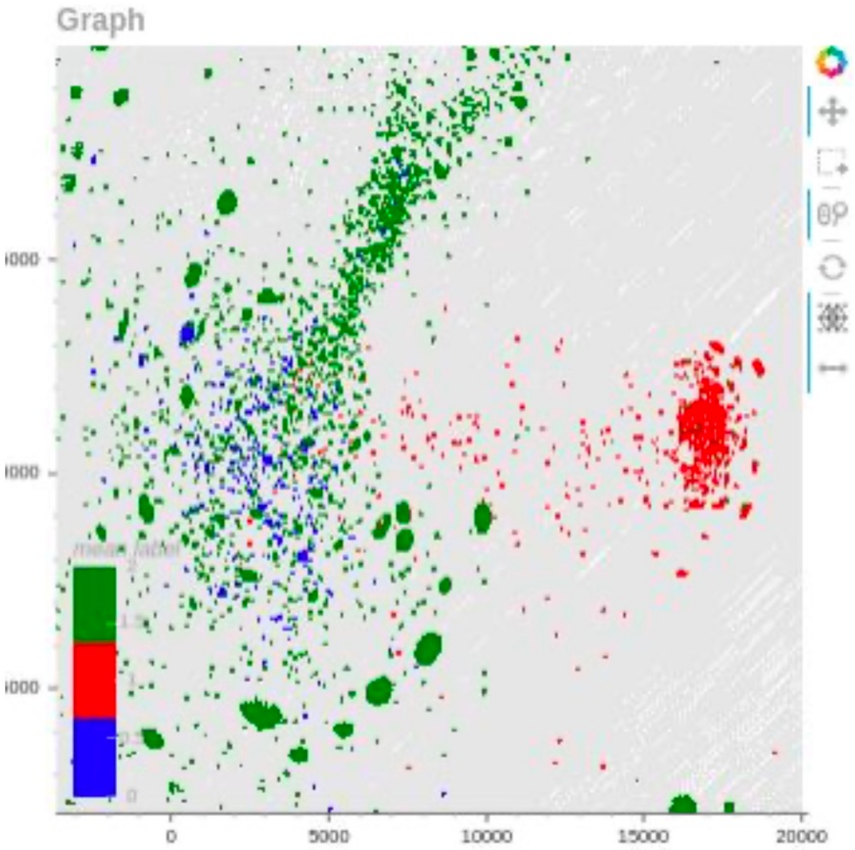
Id	Duration(hrs)	# Packets	#NetFlows	Size	Bot	#Bots
1	6.15	71,971,482	2,824,637	52GB	Neris	1
2	4.21	71,851,300	1,808,123	60GB	Neris	1
3	66.85	167,730,395	4,710,639	121GB	Rbot	1
4	4.21	62,089,135	1,121,077	53GB	Rbot	1
5	11.63	4,481,167	129,833	37.6GB	Virut	1
6	2.18	38,764,357	558,920	30GB	Menti	1
7	0.38	7,467,139	114,078	5.8GB	Sogou	1
8	19.5	155,207,799	2,954,231	123GB	Murlo	1
9	5.18	115,415,321	2,753,885	94GB	Neris	10
10	4.75	90,389,782	1,309,792	73GB	Rbot	10
11	0.26	6,337,202	107,252	5.2GB	Rbot	3
12	1.21	13,212,268	325,472	8.3GB	NSIS.ay	3
13	16.36	50,888,256	1,925,150	34GB	Virut	1

Amount of data on each scenario

Scen.	Total Flows	Botnet Flows	Normal Flows	C&C Flows	Background Flows
1	2,824,636	39,933(1.41%)	30,387(1.07%)	1,026(0.03%)	2,753,290(97.47%)
2	1,808,122	18,839(1.04%)	9,120(0.5%)	2,102(0.11%)	1,778,061(98.33%)
3	4,710,638	26,759(0.56%)	116,887(2.48%)	63(0.001%)	4,566,929(96.94%)
4	1,121,076	1,719(0.15%)	25,268(2.25%)	49(0.004%)	1,094,040(97.58%)
5	129,832	695(0.53%)	4,679(3.6%)	206(1.15%)	124,252(95.7%)
6	558,919	4,431(0.79%)	7,494(1.34%)	199(0.03%)	546,795(97.83%)
7	114,077	37(0.03%)	1,677(1.47%)	26(0.02%)	112,337(98.47%)
8	2,954,230	5,052(0.17%)	72,822(2.46%)	1,074(2.4%)	2,875,282(97.32%)
9	2,753,884	179,880(6.5%)	43,340(1.57%)	5,099(0.18%)	2,525,565(91.7%)
10	1,309,791	106,315(8.11%)	15,847(1.2%)	37(0.002%)	1,187,592(90.67%)
11	107,251	8,161(7.6%)	2,718(2.53%)	3(0.002%)	96,369(89.85%)
12	325,471	2,143(0.65%)	7,628(2.34%)	25(0.007%)	315,675(96.99%)
13	1,925,149	38,791(2.01%)	31,939(1.65%)	1,202(0.06%)	1,853,217(96.26%)

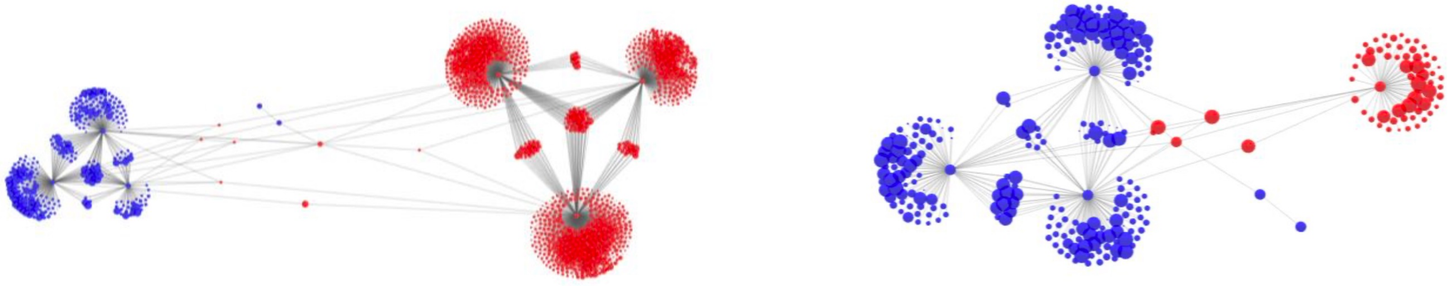
Amount of labeled flows on each capture

Visualization of Botnet Dataset

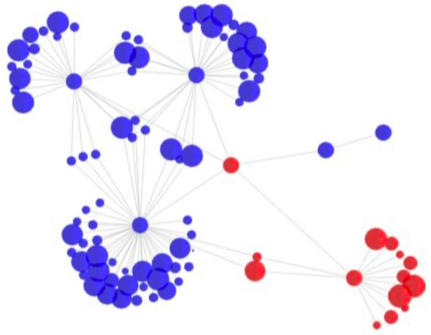


scenario	botnet	normal	background
3	26,759	116,887	4,566,929
4	1,719	25,268	1,094,040
10	106,315	15,847	1,187,592
12	2,143	7,628	315,675
13	38,791	31,939	1,853,217

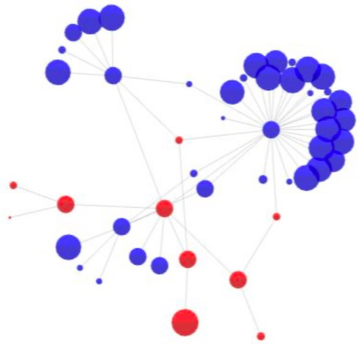
Visualization of botnet clusters



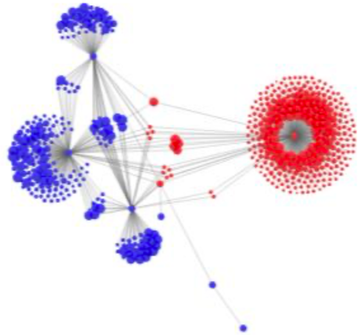
Scenario 12



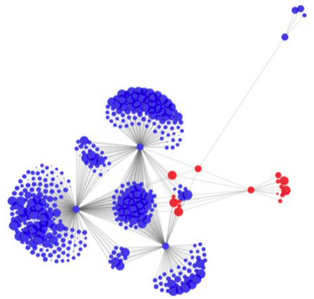
Scenario 7



Scenario 11

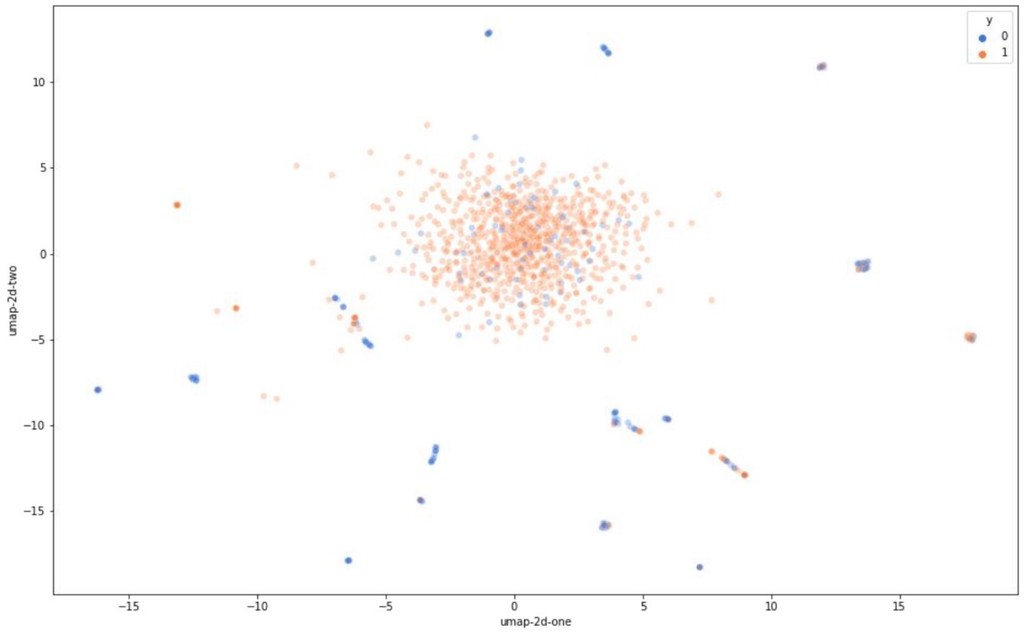


Scenario 13

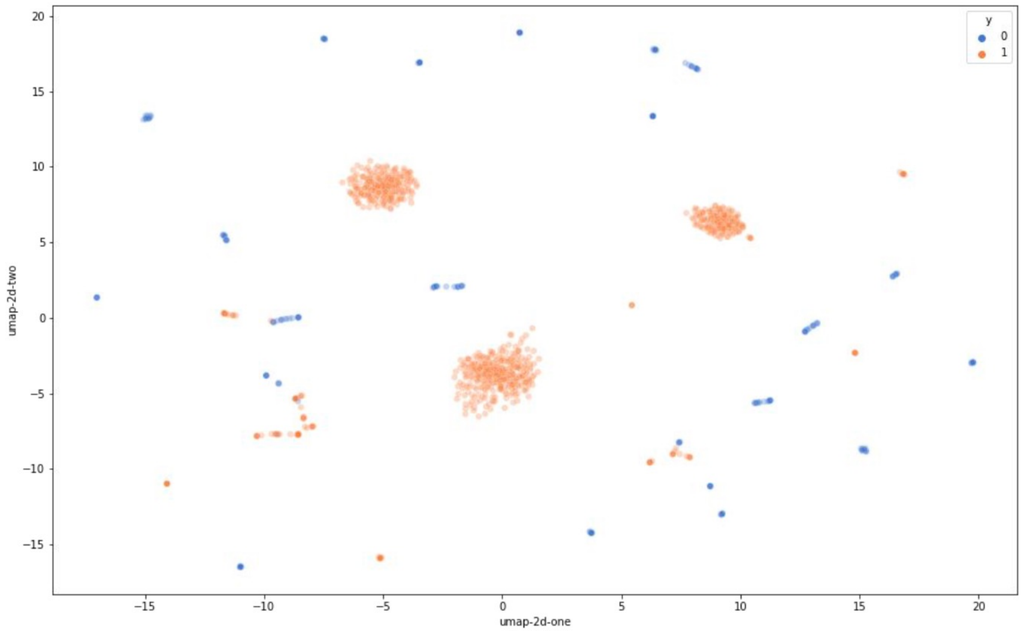


Scenario 4

Feature visualization with UMAP of using graph analytics in GNNs



Scenario 12: netflow features



Scenario 12: netflow + graph features

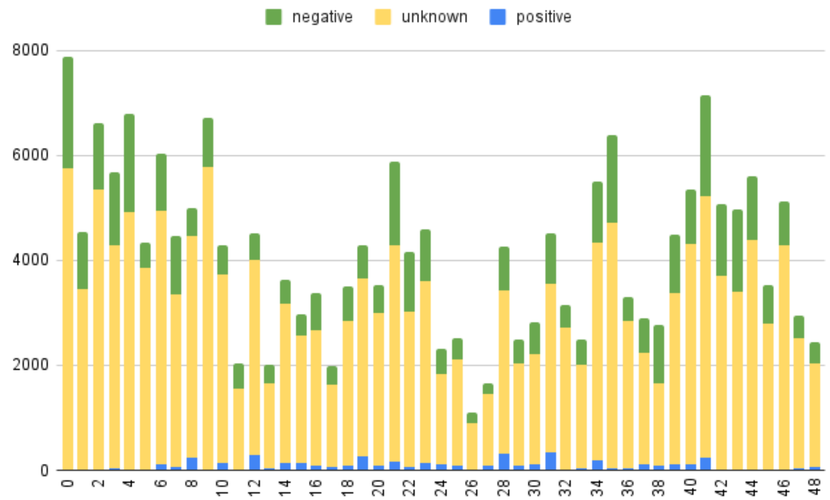
Results

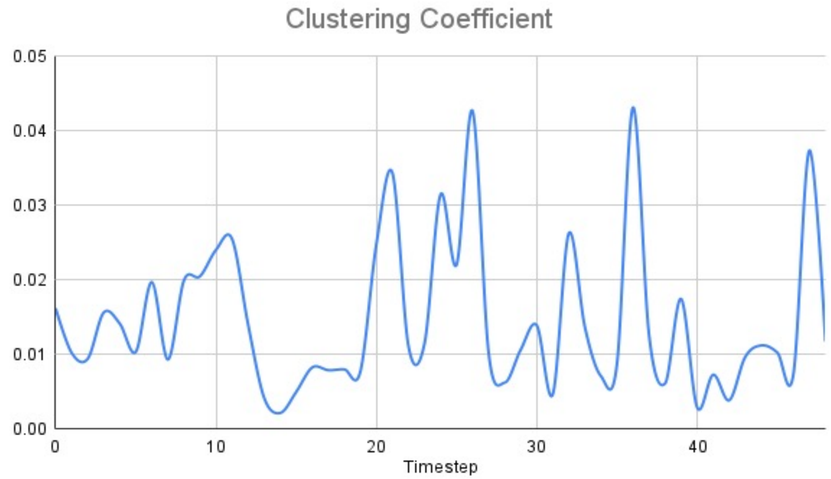
	GCN		GCN-GRU	
Scenario	ROC AUC	PR AUC	ROC AUC	PR AUC
3	**0.99**	**0.99**	**0.99**	**0.99**
4	0.94	0.61	**0.94**	**0.66**
10	0.92	0.83	**0.93**	**0.89**
12	0.89	0.87	**0.92**	**0.88**
13	**0.88**	**0.78**	0.87	0.74

- General Introduction of Graph Neural Networks
- Train Graph Neural Networks with Graph Analytics features
- **Fraud Detections using Graph Neural Networks**
- DGL + CuGraph Integration
- PyTorch Geometric Integration

USING GNN ON ELLIPTIC FOR ANOMALY DETECTION

- Temporal graph that depicts bitcoin transactions, where 203k nodes represent transactions and 234k edges represents payment flows.
- 2% nodes are labelled illicit, 21% nodes are labelled licit, and the remaining are unknown
- Each node has 166 features
  - 94 features represent local information:
    time step, transaction fee, input/output volume etc.
  - 72 aggregated features:
    min/max/std from one-hop forward/backward for the same information



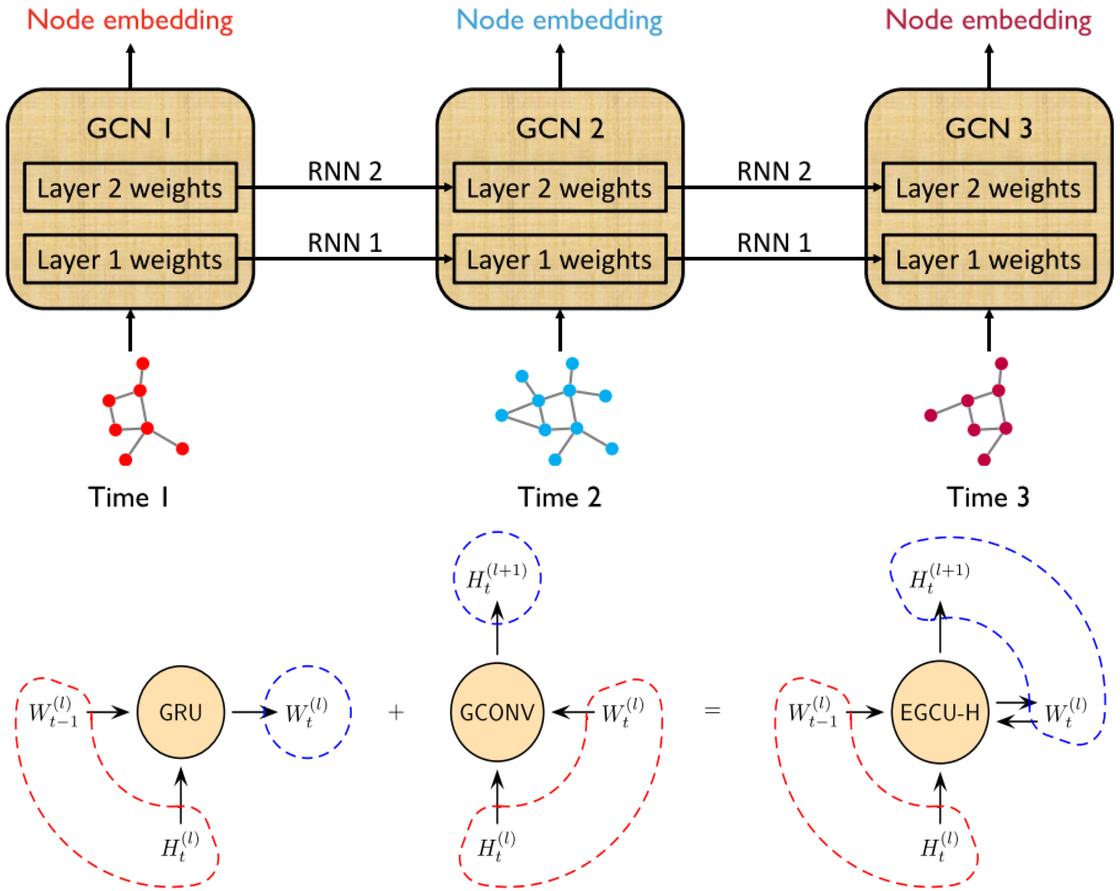


CASE STUDY - EVOLVEGCN

- Variant of GCN for graphs dynamically evolves over time

$$H^{(l+1)} = \sigma(\widehat{A_t} H_t^{(l)} W_t^{(l)})$$

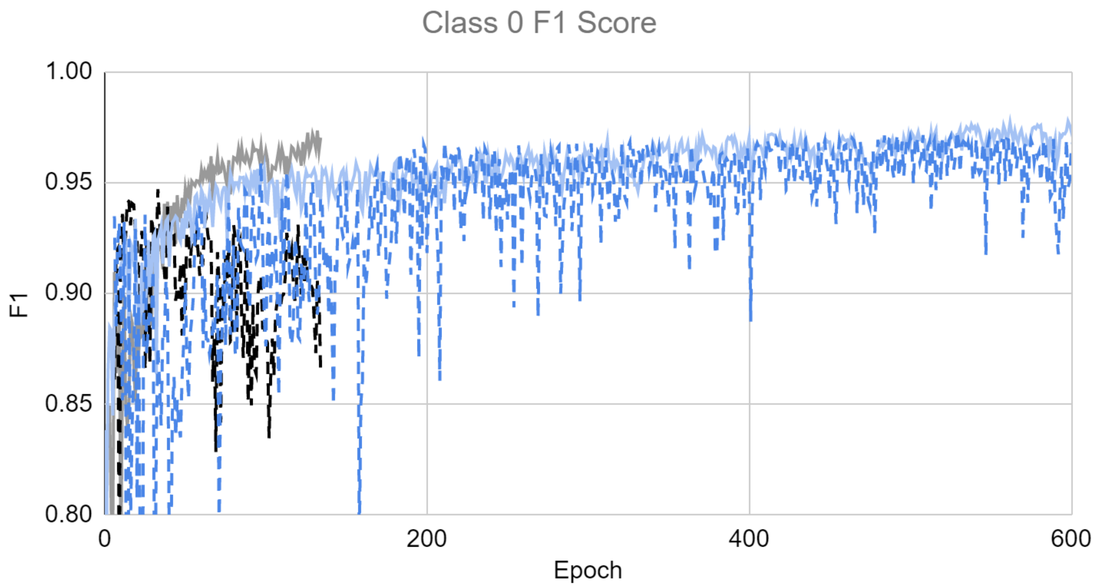
- Update the weight matrix $W_t^{(l)}$ based on current and historical information
- Treat $W_t^{(l)}$ as the hidden state of the dynamical system
- Use RNN to update GCN weight matrices instead of training through GCN

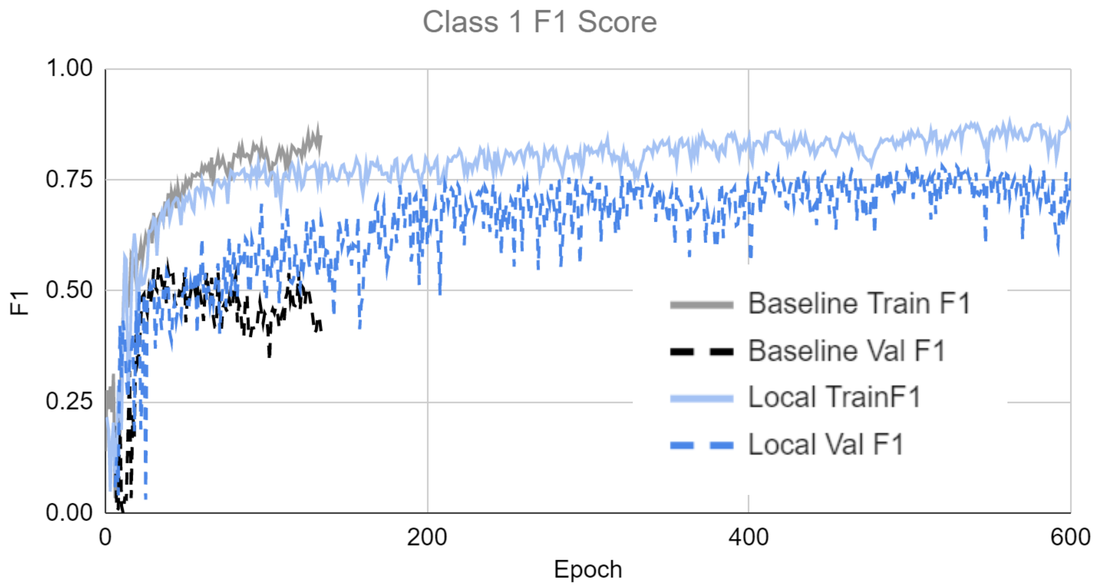


[1] Pareja, Aldo, et al. "Evolvegcn: Evolving graph convolutional networks for dynamic graphs." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 34. No. 04. 2020.

ACCURACY - LOCAL VS. FULL

- Each node has 166 features
  - 94 local information: time step, transaction fee, input/output volume etc.
  - 72 aggregated features: min/max/std from one-hop forward/backward for the same information
- Local features
  - Higher validation F1, slower convergence (134 vs. 605 epochs)

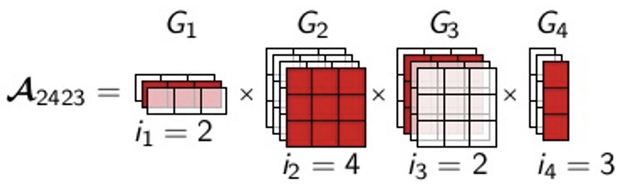


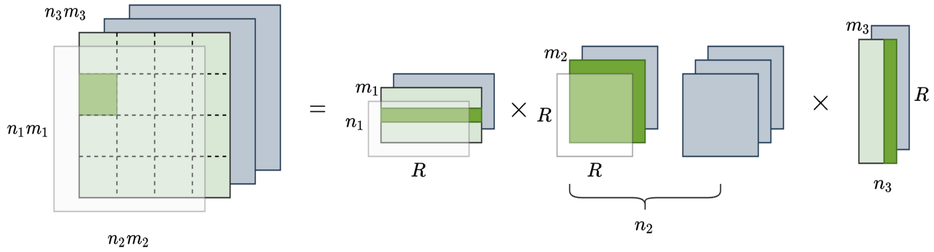


BACKGROUND - COMPRESS EMB. TABLES WITH TT

- Generalize to decomposing matrix [2] (embedding table)
  - Reshape matrix to $\mathcal{W} \in \mathbb{R}^{(m_1 \times n_1) \times (m_2 \times n_2) \times \cdots \times (m_d \times n_d)}$
  - Combine each pair of dimensions as one and decompose the d-way tensor

$$\mathcal{W}((i_1, j_1), (i_2, j_2), \ldots, (i_d, j_d)) = \mathcal{G}_1(:, i_1, j_1, :)\mathcal{G}_2(:, i_2, j_2, :) \ldots \mathcal{G}_d(:, i_d, j_d, :)$$





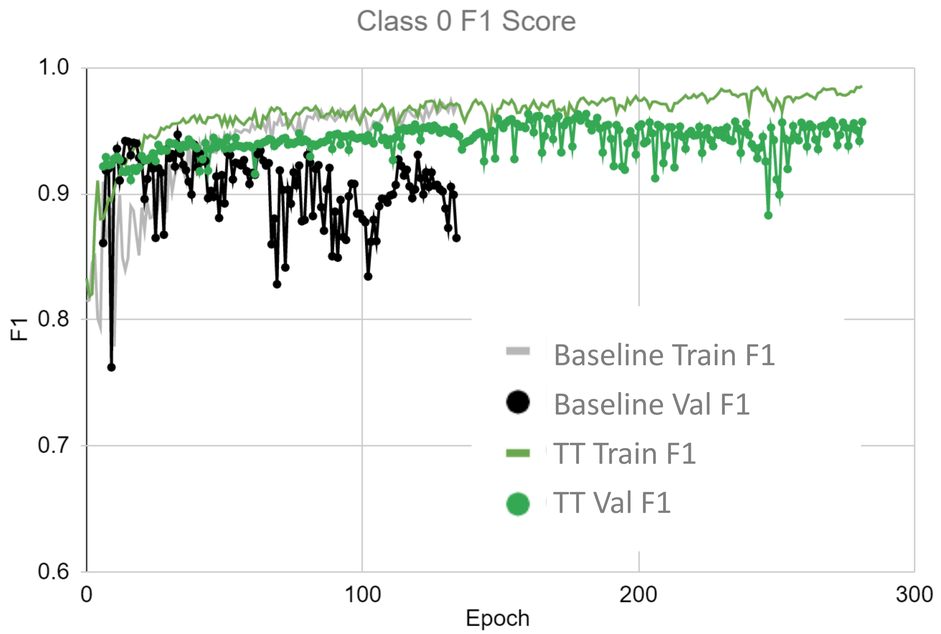
TT-matrix example

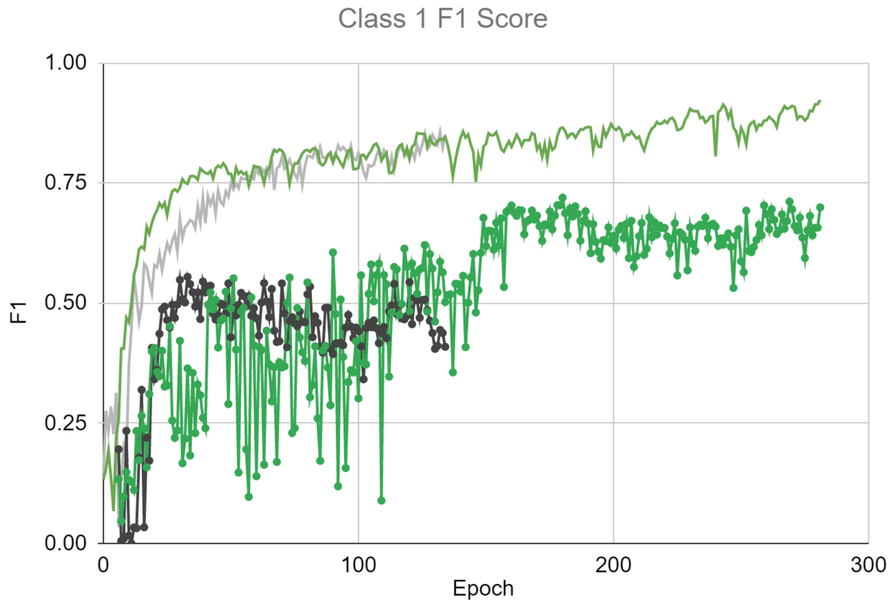
1) Matrix W of size 5,000,000 x 24
2) Factorize dimensions
   5,000,000 = 100 x 200 x 250,
   24 = 4 x 2 x 3
3) Reshape W as a 6-way tensor ((100, 4), (200, 2), (250, 3))
4) Decompose W using 3 TT-cores
5) TT-core Shape:(1, 100, 4, R), (R, 200, 2, R),(R, 250, 3, 1)

[1] Yin, C., Acun, B., Wu, C.J. and Liu, X., 2021. TT-Rec: Tensor Train Compression for Deep Learning Recommendation Models. *Proceedings of Machine Learning and Systems*, *3*.
[2] Novikov, A., Podoprikhin, D., Osokin, A. and Vetrov, D., 2015. Tensorizing neural networks. *arXiv preprint arXiv:1509.06569*.

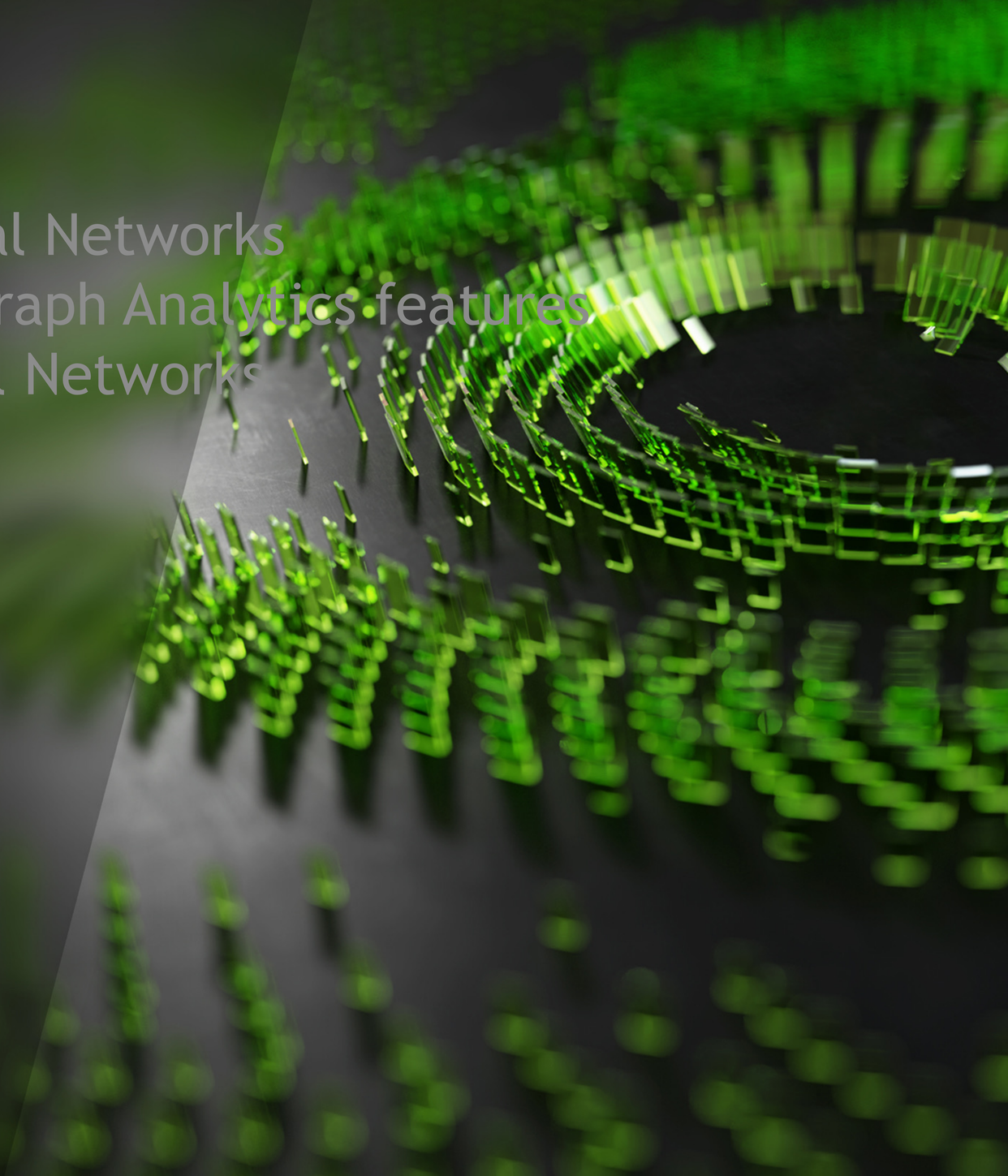
ACCURACY – TT VS. FULL EMB

- Similar training F1 for both classes
- Improve validation F1 for licit class (0) by 0.03, for illicit class (1) by 0.14
- Slower convergence (134 vs. 280 epochs)
- Reduce overfitting by low rank representation





- General Introduction of Graph Neural Networks
- Train Graph Neural Networks with Graph Analytics features
- Fraud Detections using Graph Neural Networks
- **DGL + CuGraph Integration**
- PyTorch Geometric Integration



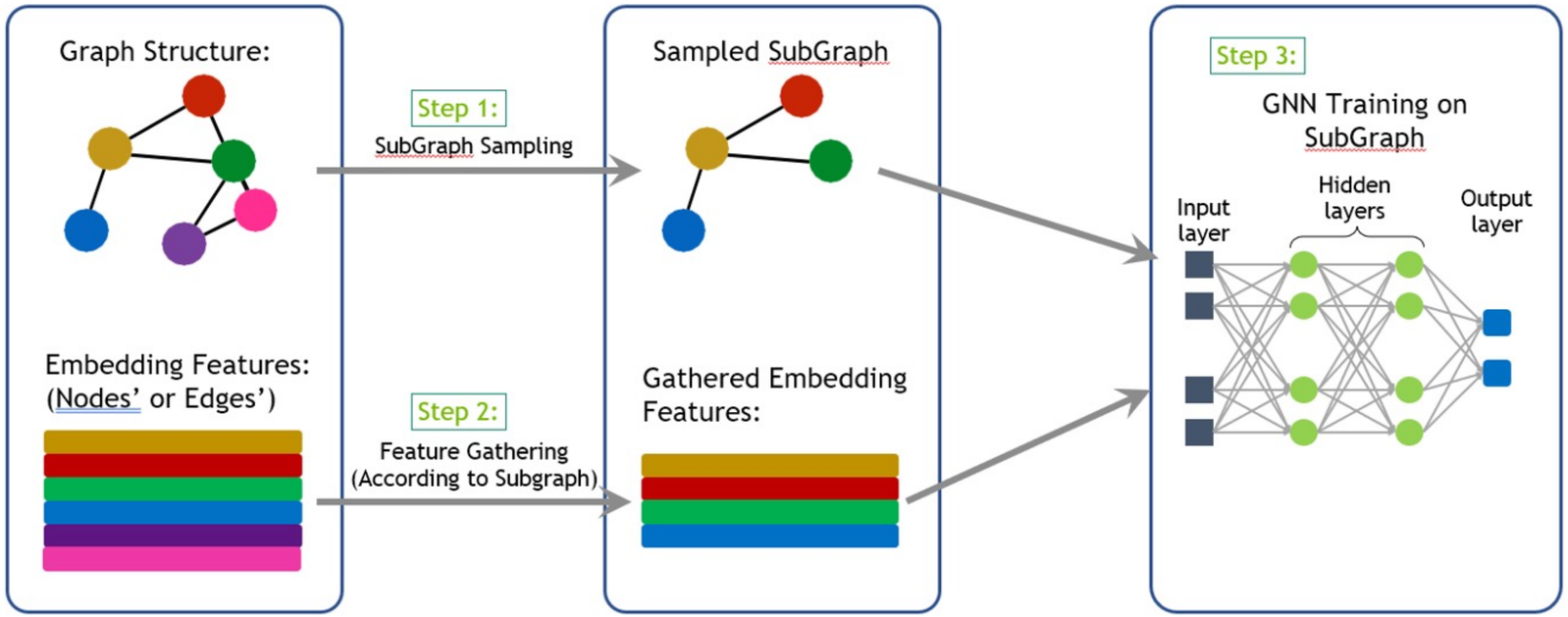
RESULT OF THE FRAUD DETECTIONS WITH GRAPHSAGE

For Integration Work Demo

- In this tutorial, we are using a simple model, GraphSAGE as an example.
- We are doing sampling is with CuGraph functions, and training with DGL and PyG frameworks.
- We used the inductive graph machine learning setting.
- 85% as training data and 15% as testing data.
- We get auc-roc: 0.968.

MOTIVATIONS OF INTEGRATION

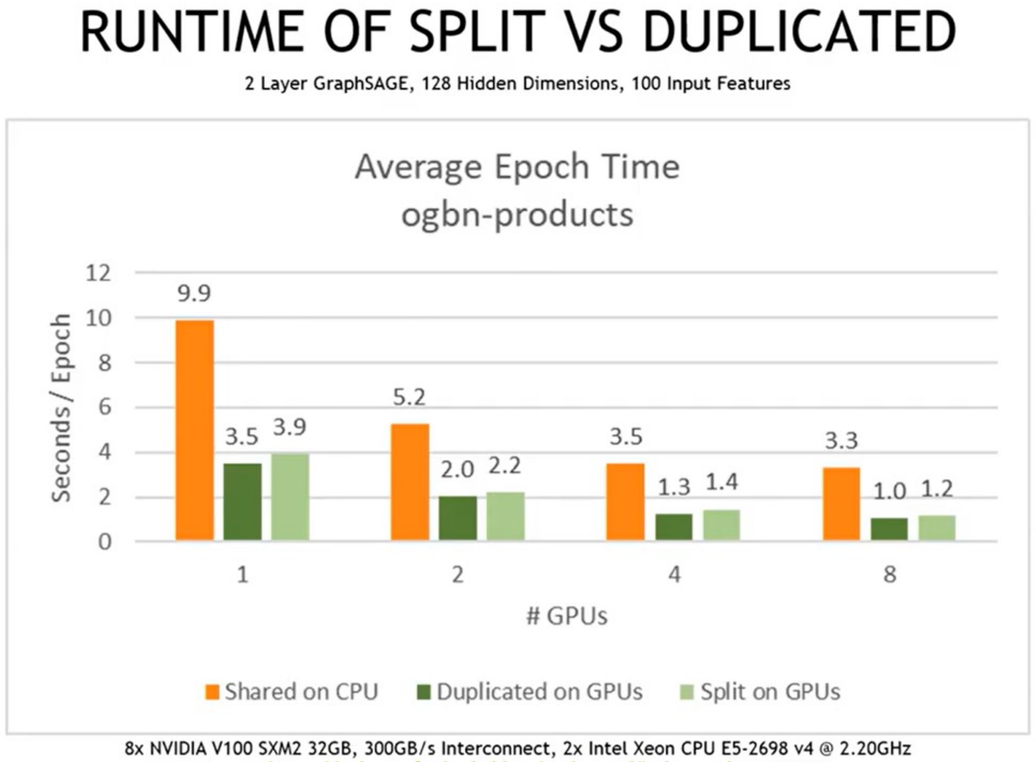
- Avoiding data movement between device and host.
- Store graph across GPUs: Graph structure, embedding features etc. will be all on GPUs.
- Sampling faster in parallel



All on GPU

NVIDIA IN DGL

- New features
  - Moving sampling to GPU
    - Node sampling: 3x faster for node classification
    - Edge sampling: 6x faster for link prediction
  - Moving features to GPU storage
    - Multi-GPU data storage: based on NCCL

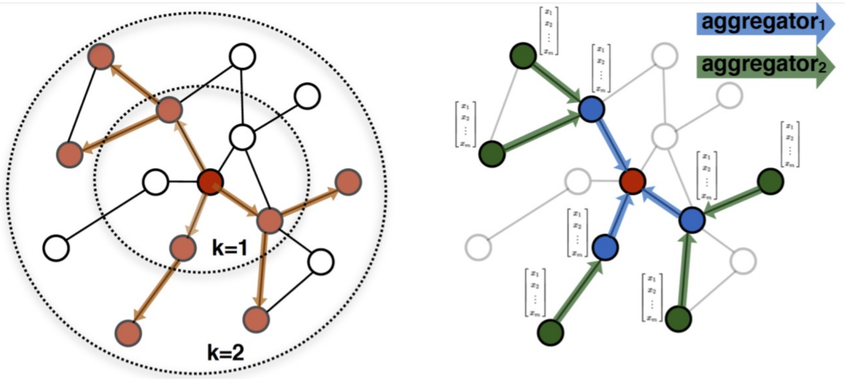


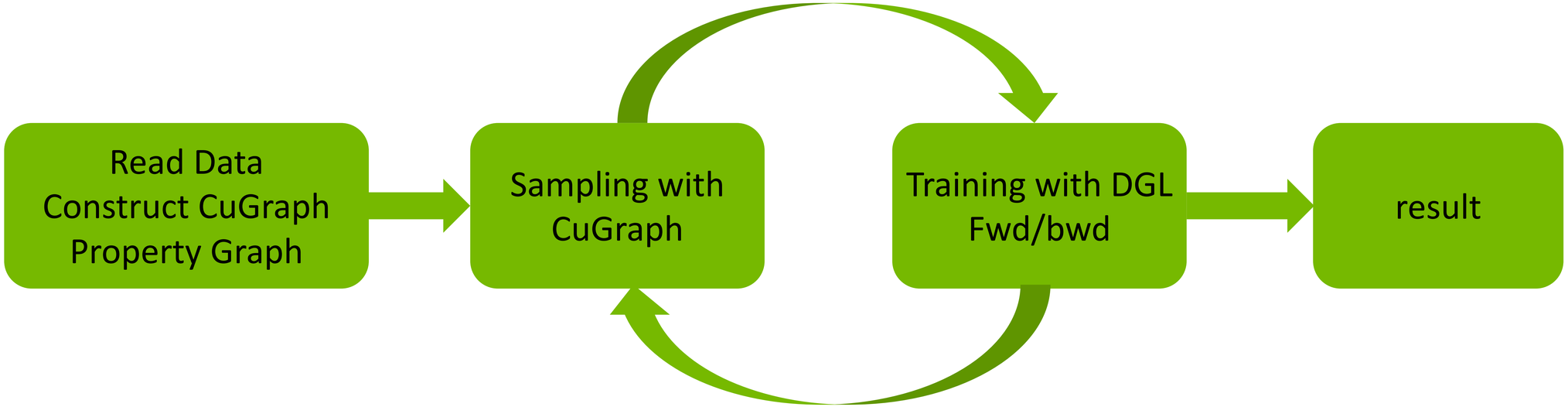
INTEGRATION WITH DGL

Current Status and Goal

The applications using the sampling methods

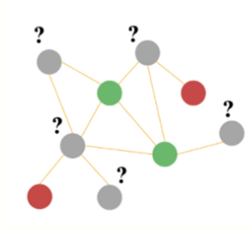
- GraphSAGE
- RGCN (GTC 2023)



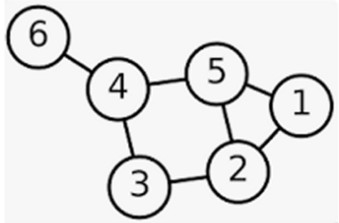


[1] Hamilton, W. L., *et al.*. (2017). Inductive representation learning on large graphs. *NIPS*.

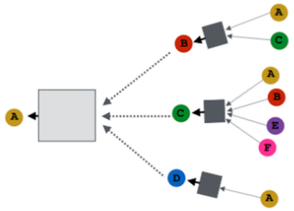
INTRODUCTION TO DGL



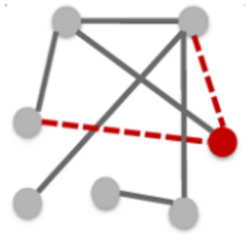
Node Classification with DGL



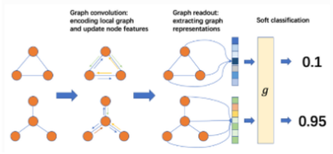
How Does DGL Represent A Graph?



Write your own GNN module



Link Prediction using Graph Neural Networks



Training a GNN for Graph Classification

Make Your Own Dataset

DGL provides a variety of GNNs examples.

It also provides mainstream GNN modules such as: GraphConv, RelGraphConv, SAGEConv, GATConv, GINConv…

Datasets: it provide benchmark datasets for nodes/edges/graphs prediction

class **dgl.data.DGLDataset**(*name, url=None, raw_dir=None, save_dir=None, hash_key=(), force_reload=False, verbose=False*) [source]

The basic DGL dataset for creating graph datasets. This class defines a basic template class for DGL Dataset. The following steps will are executed automatically:

1. Check whether there is a dataset cache on disk (already processed and stored on the disk) by invoking `has_cache()`. If true, goto 5.
2. Call `download()` to download the data.
3. Call `process()` to process the data.
4. Call `save()` to save the processed dataset on disk and goto 6.
5. Call `load()` to load the processed dataset from disk.
6. Done.

DGL Sampling Functions and DataLoader

dgl.sampling

The `dgl.sampling` package contains operators and utilities for sampling from a graph via random walks, neighbor sampling, etc. They are typically used together with the `DataLoader` s in the `dgl.dataloading` package. The user guide Chapter 6: Stochastic Training on Large Graphs gives a holistic explanation on how different components work together.

Random walk

`random_walk`(g, nodes, *[, metapath, length, ...])	Generate random walk traces from an array of startir
`node2vec_random_walk`(g, nodes, p, q, walk_length)	Generate random walk traces from an array of startir
`pack_traces`(traces, types)	Pack the padded traces returned by `random_walk()` i

Neighbor sampling

`sample_neighbors`(g, nodes, fanout[, ...])	Sample neighboring edges of the given nodes and retu
`sample_neighbors_biased`(g, nodes, fanout, bias)	Sample neighboring edges of the given nodes and retu
`select_topk`(g, k, weight[, nodes, edge_dir, ...])	Select the neighboring edges with k-largest (or k-smalle
`PinSAGESampler`(G, ntype, other_type, ...[, ...])	PinSAGE-like neighbor sampler.

class **dgl.dataloading.pytorch.NodeDataLoader**(*g, nids, block_sampler, device=None, use_ddp=False, ddp_seed=0, **kwargs*) [source]

PyTorch dataloader for batch-iterating over a set of nodes, generating the list of message flow graphs (MFGs) as computation dependency of the said minibatch.

Examples

To train a 3-layer GNN for node classification on a set of nodes `train_nid` on a homogeneous graph where each node takes messages from all neighbors (assume the backend is PyTorch):

```
>>> sampler = dgl.dataloading.MultiLayerNeighborSampler([15, 10, 5])
>>> dataloader = dgl.dataloading.NodeDataLoader(
...     g, train_nid, sampler,
...     batch_size=1024, shuffle=True, drop_last=False, num_workers=4)
>>> for input_nodes, output_nodes, blocks in dataloader:
...     train_on(input_nodes, output_nodes, blocks)
```

class **dgl.dataloading.pytorch.EdgeDataLoader**(*g, eids, block_sampler, device='cpu', use_ddp=False, ddp_seed=0, **kwargs*) [source]

PyTorch dataloader for batch-iterating over a set of edges, generating the list of message flow graphs (MFGs) as computation dependency of the said minibatch for edge classification, edge regression, and link prediction.

DGL data format

Examples

```
>>> dataset = dgl.data.rdf.AIFBDataset()
>>> graph = dataset[0]
>>> category = dataset.predict_category
>>> num_classes = dataset.num_classes
>>>
>>> train_mask = g.nodes[category].data.pop('train_mask')
>>> test_mask = g.nodes[category].data.pop('test_mask')
>>> labels = g.nodes[category].data.pop('labels')
```

1 read the data

2 get DGL graph structure

3 add nodes features/ edge features/ labels

Implementation with DGL and DGL+Cugraph integration

Pure DGL implementation

```
g = dgl.graph((edge_index[0],edge_index[1]))
g.ndata['feat'] = node_features.to(torch.float)
g.ndata['label'] = torch.LongTensor(labels)
g = g.to('cuda' if args.mode == 'puregpu' else 'cpu')
train_idx = train_idx.to('cuda' if args.mode == 'puregpu' else 'cpu')
valid_idx = valid_idx.to('cuda' if args.mode == 'puregpu' else 'cpu')

device = torch.device('cpu' if args.mode == 'cpu' else 'cuda')
```

DGL+Cugraph implementation

```
##### creating a Graphstore from cuDF dataframes
import cugraph
pg = cugraph.experimental.PropertyGraph()
# create gs from pg
gs = dgl.contrib.cugraph.CuGraphStorage(pg)
gs.add_edge_data(edge_df, ["src", "dst"], "edge_id")

### Set node type setting to DGL default
gs.add_node_data(node_feat_df, "node_id", 'feat', ntype='_N')
gs.add_node_data(node_label_df, "node_id", 'label', ntype='_N')
```

How the Backend from DGL to Cugraph

- 1 initiate a cugraph Property Graph

  pg = cugraph.experimental.PropertyGraph()

- 2 create a graph storage base on the Property Graph

  gs = dgl.contrib.cugraph.CuGraphStorage(pg)

- 3 add corresponding graph structure data, nodes features and labels.

  gs.add_edge_data(edge_df, ["src", "dst"], "edge_id")

  gs.add_node_data(node_feat_df, "node_id", 'feat', ntype='_N’)

  gs.add_node_data(node_label_df, "node_id", 'label', ntype='_N')

CuGraph Backend

- **CuGraph side implementation**

CuGraphStore, CuFeatureStore classes in CuGraph library store the graph structure and features of nodes and edges; There are corresponding functions in these 2 classes for sampling and fetching.

https://github.com/rapidsai/cugraph/blob/branch-22.08/python/cugraph/cugraph/gnn/graph_store.py

- **DGL side implementation**

CuGraphStorage class in DGL will call functions CuGraph's CuGraphStore class, the return type of the functions are in DGL format.

https://github.com/rapidsai/dgl/blob/master/python/dgl/contrib/cugraph/cugraph_storage.py

- We support integration with PyTorch Geometric as well.

Cugraph GraphStore Backend Implementation

```
def add_node_data(self, df, node_col_name, node_key, ntype=None):
    self.gdata.add_vertex_data(
        df, vertex_col_name=node_col_name, type_name=ntype
    )
    col_names = list(df.columns)
    col_names.remove(node_col_name)
    self.ndata_key_col_d[node_key] += col_names

def add_edge_data(self, df, vertex_col_names, edge_key, etype=None):
    self.gdata.add_edge_data(
        df, vertex_col_names=vertex_col_names, type_name=etype
    )
    col_names = [
        col for col in list(df.columns) if col not in vertex_col_names
    ]
    self.edata_key_col_d[edge_key] += col_names
```

Cugraph GraphStore Backend Implementation

```
def get_node_storage(self, key, ntype=None):

    if ntype is None:
        ntypes = self.ntypes
        if len(self.ntypes) > 1:
            raise ValueError(
                (
                    "Node type name must be specified if there "
                    "are more than one node types."
                )
            )
        ntype = ntypes[0]
    df = self.gdata._vertex_prop_dataframe
    col_names = self.ndata_key_col_d[key]
    return CuFeatureStorage(
        df=df,
        id_col=vid_n,
        _type_=ntype,
        col_names=col_names,
        backend_lib=self.backend_lib,
    )
```

```
def get_edge_storage(self, key, etype=None):
    if etype is None:
        etypes = self.etypes
        if len(self.etypes) > 1:
            raise ValueError(
                (
                    "Edge type name must be specified if there"
                    "are more than one edge types."
                )
            )

        etype = etypes[0]
    col_names = self.edata_key_col_d[key]
    df = self.gdata._edge_prop_dataframe
    return CuFeatureStorage(
        df=df,
        id_col=eid_n,
        _type_=etype,
        col_names=col_names,
        backend_lib=self.backend_lib,
    )
```

Sampling functions and Property functions

- sample_neighbors(nodes, fanout=-1, edge_dir="in", prob=None, replace=False)
- extracted_reverse_subgraph_without_renumbering( )
- extracted_subgraph_without_renumbering( )
- find_edges(edge_ids_cap, etype)
- node_subgraph(nodes=None, create_using=cugraph.Graph, directed=False, multigraph=True)
- egonet(nodes, k)
- randomwalk(nodes, length, prob=None, restart_prob=None)

- ntypes( )
- etypes( )
- ndata( )
- edata( )
- gdata( )
- num_vertices( )
- get_vertex_ids( )

DGL backend implementation

```
class CuGraphStorage:
    """
    Duck-typed version of the DGL GraphStorage class made for cuGraph
    """

    def __init__(self, g):
        # lazy import to prevent creating cuda context
        # till later to help in multiprocessing
        from cugraph.gnn import CuGraphStore

        self.graphstore = CuGraphStore(graph=g)

    def get_node_storage(self, key, ntype=None):
        return self.graphstore.get_node_storage(key, ntype)

    def get_edge_storage(self, key, etype=None):
        return self.graphstore.get_edge_storage(key, etype)
```

- DGL.CugraphStorage calls Cugraph.GraphStore,
- Cugraph.GraphStore calls cugraph functions.

Cugraph GraphStorage in DGL

```
def sample_neighbors(
    self,
    seed_nodes,
    fanout,
    edge_dir="in",
    prob=None,
    exclude_edges=None,
    replace=False,
    output_device=None,
):
    """
    Return a DGLGraph which is a subgraph induced by sampling neighboring
    edges of the given nodes.
    See ``dgl.sampling.sample_neighbors`` for detailed semantics.
    Parameters
    ----------
```

```
if not F.is_tensor(seed_nodes):
    seed_nodes = F.tensor(seed_nodes)
seed_nodes_cap = F.zerocopy_to_dlpack(seed_nodes)

src_cap, dst_cap, edge_id_cap = self.graphstore.sample_neighbors(
    seed_nodes_cap,
    fanout,
    edge_dir=edge_dir,
    prob=prob,
    replace=replace,
)
sampled_graph = dgl.graph(
    (
        F.zerocopy_from_dlpack(src_cap),
        F.zerocopy_from_dlpack(dst_cap),
    )
)
sampled_graph = sampled_graph.astype(seed_nodes.dtype)
sampled_graph.edata["_ID"] = F.zerocopy_from_dlpack(edge_id_cap)

# to device function move the dgl graph to desired devices
if output_device is not None:
    sampled_graph.to_device(output_device)

return sampled_graph
```

Convert functions

We are using zero copy techniques converting the graph types via dlpack

From DGL to CuGraph:

```
def to_cugraph(g):
    """Convert a DGL graph to a :class:`cugraph.Graph` and return.

    Parameters
    ----------
    g : DGLGraph
        A homogeneous graph.

    Returns
    -------
    cugraph.Graph
        The converted cugraph graph.
```

From CuGraph to DGL

```
sampled_graph = dgl.graph(
    (
        F.zerocopy_from_dlpack(src_cap),
        F.zerocopy_from_dlpack(dst_cap),
    )
)
```

Example of louvain

```
def louvain(dgl_g):
    cugraph_g = dgl_g.to_cugraph().to_undirected()
    df, _ = cugraph.louvain(cugraph_g, resolution=3)
    # revert the node ID renumbering by cugraph
    df = cugraph_g.unrenumber(df, 'vertex').sort_values('vertex')
    return torch.utils.dlpack.from_dlpack(df['partition'].to_dlpack()).long()
```

Result on training with pure DGL and DGL + Cugraph Integration

Training on DGL Graph

```
%%time
print('Training...')
train({}, torch.device('cuda'), g, train_idx.to('cuda'), valid_idx.to('cuda'), model)
```

```
Training...
Epoch 00000 | Loss 0.7836 | AUROC 0.9209
Epoch 00001 | Loss 0.3531 | AUROC 0.9365
Epoch 00002 | Loss 0.2865 | AUROC 0.9438
Epoch 00003 | Loss 0.2319 | AUROC 0.9488
Epoch 00004 | Loss 0.2017 | AUROC 0.9524
Epoch 00005 | Loss 0.1874 | AUROC 0.9576
Epoch 00006 | Loss 0.1671 | AUROC 0.9590
Epoch 00007 | Loss 0.1577 | AUROC 0.9630
Epoch 00008 | Loss 0.1489 | AUROC 0.9626
Epoch 00009 | Loss 0.1423 | AUROC 0.9648
CPU times: user 10.9 s, sys: 257 ms, total: 11.1 s
Wall time: 4.79 s
```

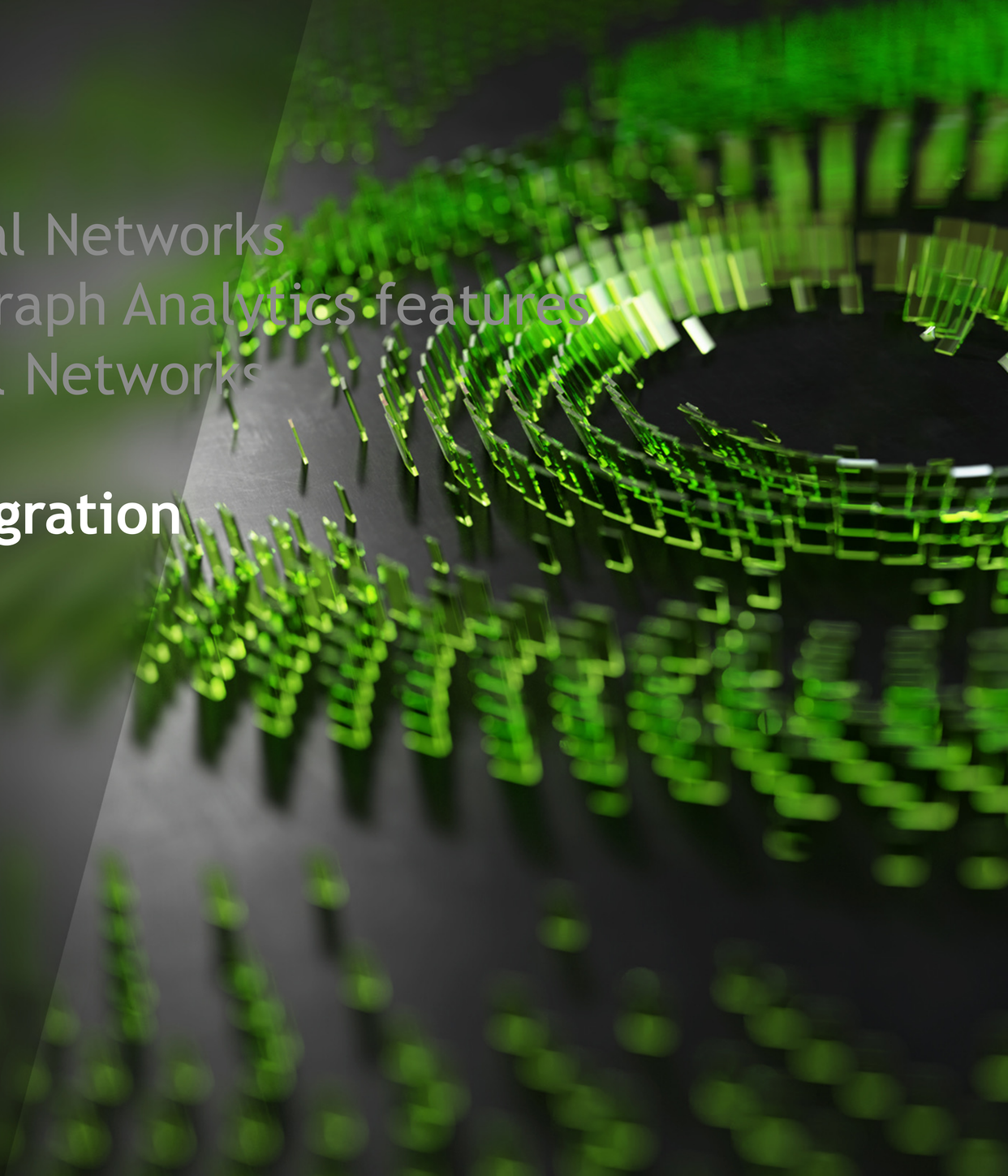
Training on Graph Store

```
%%time
print('Training...')
train({}, torch.device('cuda'), gs, train_idx.to('cuda'), valid_idx.to('cuda'), model)
```

```
Training...
Epoch 00000 | Loss 1.0093 | AUROC 0.9047
Epoch 00001 | Loss 0.3802 | AUROC 0.9287
Epoch 00002 | Loss 0.3079 | AUROC 0.9372
Epoch 00003 | Loss 0.2520 | AUROC 0.9446
Epoch 00004 | Loss 0.2117 | AUROC 0.9502
Epoch 00005 | Loss 0.2024 | AUROC 0.9564
Epoch 00006 | Loss 0.1801 | AUROC 0.9582
Epoch 00007 | Loss 0.1622 | AUROC 0.9623
Epoch 00008 | Loss 0.1503 | AUROC 0.9662
Epoch 00009 | Loss 0.1416 | AUROC 0.9682
CPU times: user 36.1 s, sys: 2.78 s, total: 38.9 s
Wall time: 32.8 s
```

https://github.com/wangxiaoyunNV/kdd/blob/main/example_training_gs_dgl.ipynb

- General Introduction of Graph Neural Networks
- Train Graph Neural Networks with Graph Analytics features
- Fraud Detections using Graph Neural Networks
- DGL + CuGraph Integration
- **PyTorch Geometric + CuGraph Integration**



PyTorch Geometric

- Similar to DGL, PyTorch Geometric provides benchmark datasets; NN modules, and data loading functions

Data Handling of Graphs

A graph is used to model pairwise relations (edges) between objects (nodes). A single graph in PyG is described by an instance of `torch_geometric.data.Data`, which holds the following attributes by default:

- `data.x`: Node feature matrix with shape `[num_nodes, num_node_features]`
- `data.edge_index`: Graph connectivity in COO format with shape `[2, num_edges]` and type `torch.long`
- `data.edge_attr`: Edge feature matrix with shape `[num_edges, num_edge_features]`
- `data.y`: Target to train against (may have arbitrary shape), *e.g.*, node-level targets of shape `[num_nodes, *]` or graph-level targets of shape `[1, *]`
- `data.pos`: Node position matrix with shape `[num_nodes, num_dimensions]`

GAAS – GRAPH-AS-A-SERVICE

Background and goals for GNNs

- Make managing GPU resources for large scale graphs easy and non-intrusive to the GNN framework.
  - Allows for easily dedicating multiple GPUs to large graph data for the duration of the workflow, eliminating resource contention
  - Scaling from single GPU (SG), to multi-GPU (MG), to multi-node/multi-GPU (MNMG) should not require changes to the graph integration code

- Support multiple concurrent clients/processes/threads sharing one or more graphs
  - No need for each client to have access to graph data files, perform ETL, etc.
  - Simplify concurrent programming – synchronization, batch processing details, etc. are implemented server-side

- Simplify MNMG deployments
  - Docker images contain all required, compatible packages
  - Clients use the same GaaS APIs for both SG and MG

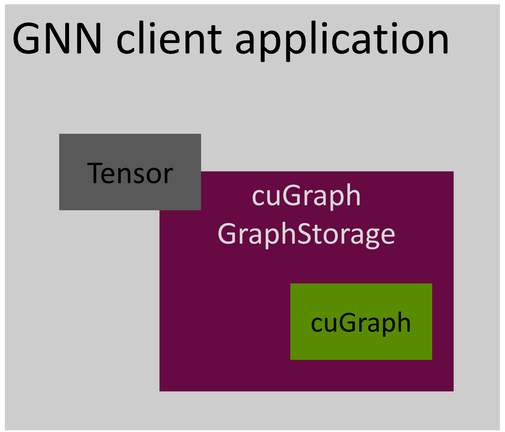
NOTE: GaaS is still a work-in-progress and changes frequently as we add functionality and improve performance.

GAAS - DESIGN PATTERNS

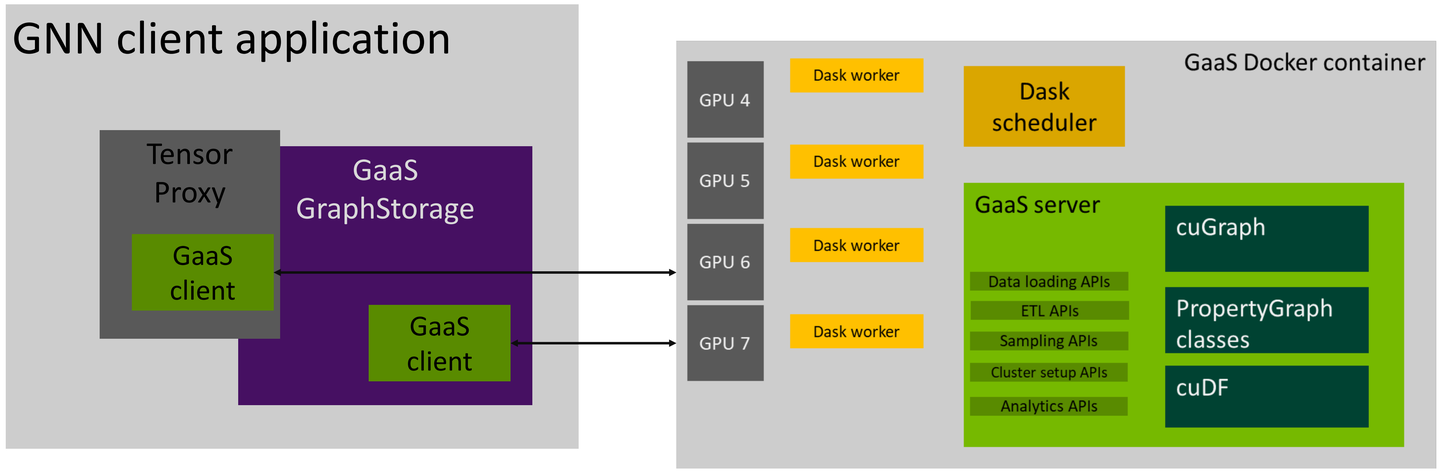
Proxy objects

- Support both cuGraph “local” and “GaaS” modes without code changes
  - Objects for either mode have the same API (duck-typed)
  - “local” mode object accesses cuGraph directly
  - “GaaS” mode object accesses the server via the client API and manages remote object lifecyle using RAII-style implementation
- Simplifies GNN integration by encapsulating integration details

cuGraph “local”

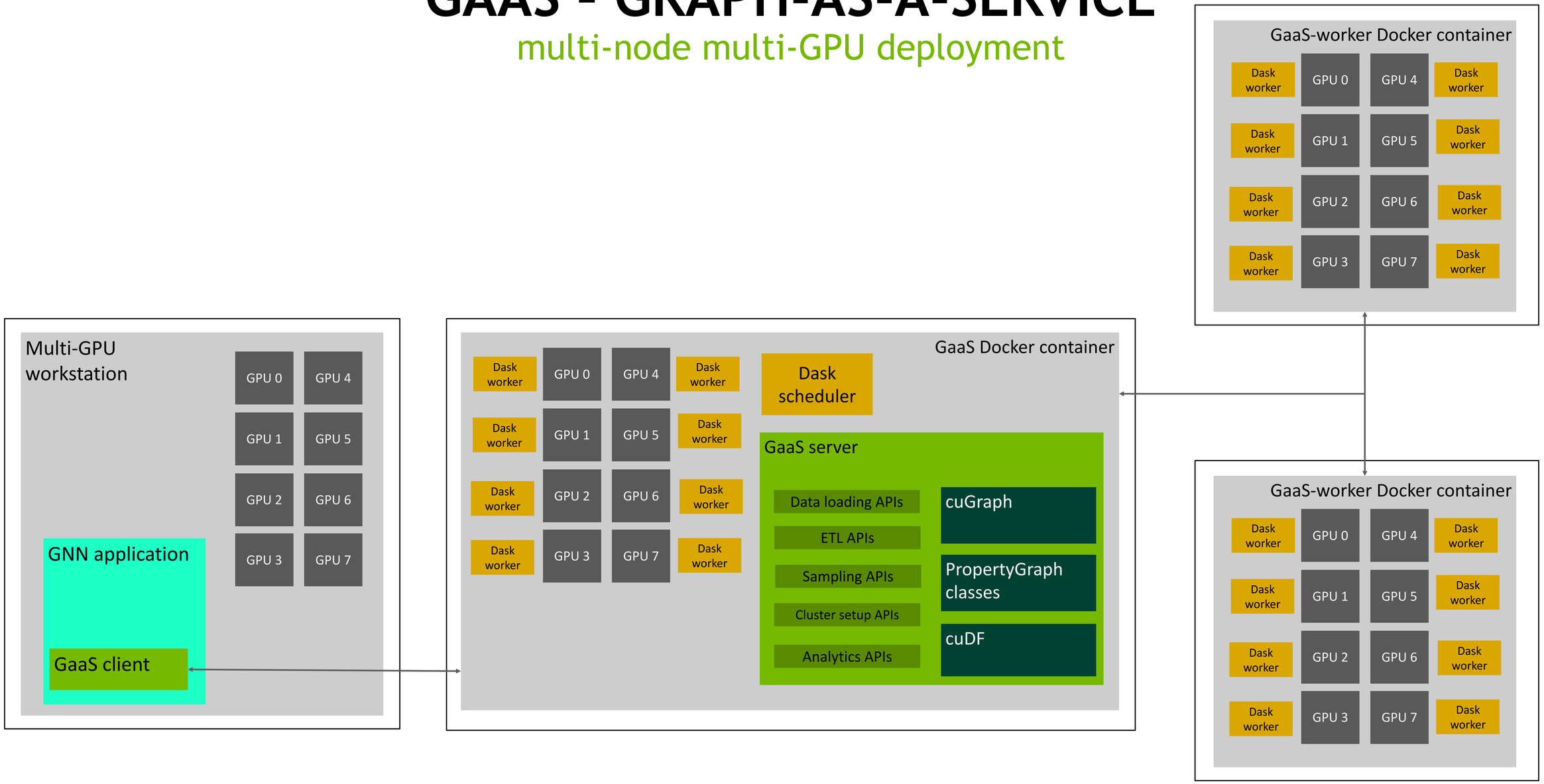


cuGraph “GaaS”



GAAS - GRAPH-AS-A-SERVICE

multi-node multi-GPU deployment

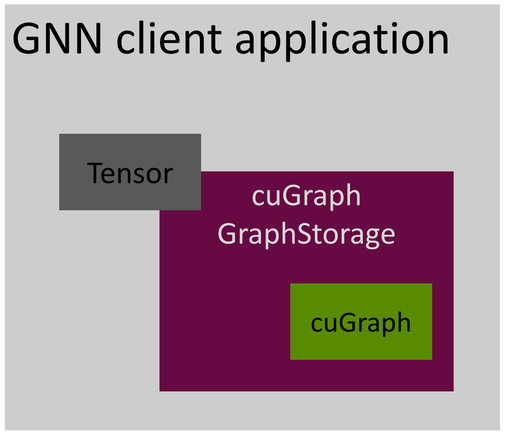


GAAS - DESIGN PATTERNS

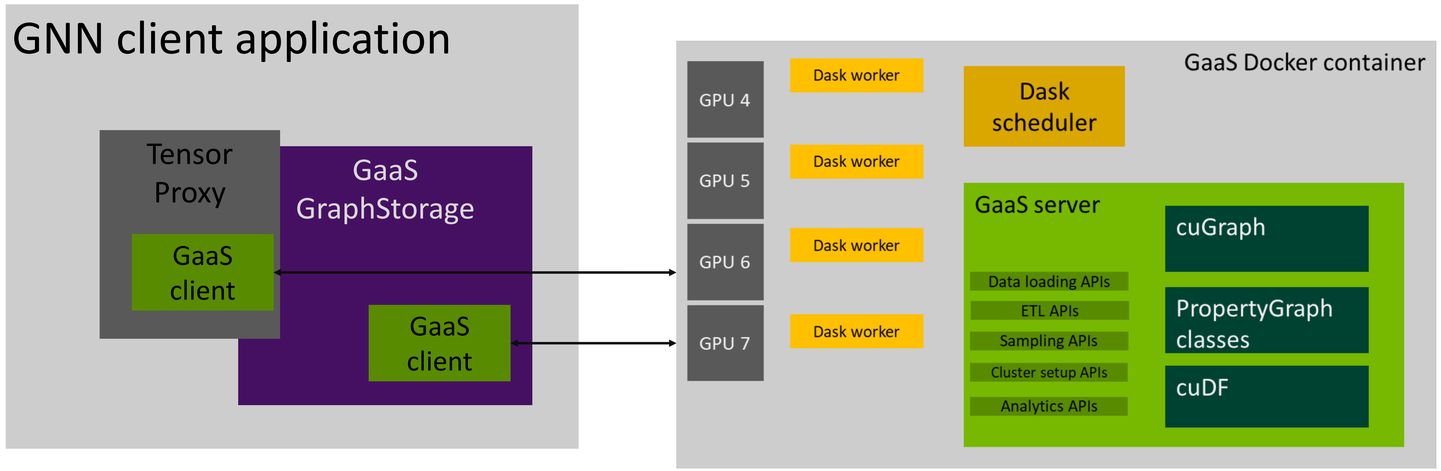
Proxy objects

- Support both cuGraph “local” and “GaaS” modes without code changes
  - Objects for either mode have the same API (duck-typed)
  - “local” mode object accesses cuGraph directly
  - “GaaS” mode object accesses the server via the client API and manages remote object lifecyle using RAII-style implementation
- Simplifies GNN integration by encapsulating integration details

cuGraph “local”



cuGraph “GaaS”



Integration Example

```
import cudf
import cugraph
from cugraph.experimental import PropertyGraph
from cugraph.gnn.gaas_extensions import load_reddit
G = load_reddit(None, '/work/data/reddit')
```

```
from cugraph.gnn.pyg_extensions import CuGraphData
cd = CuGraphData(G, reserved_keys=['id','type'])
```

```
from torch_geometric.loader import LinkNeighborLoader
import numpy as np

train_loader = LinkNeighborLoader(
    data,
    num_neighbors=TRAINING_ARGS['fanout'],
    batch_size=TRAINING_ARGS['batch_size'],
    shuffle=True
)
```

```
class LinkNeighborSampler(NeighborSampler):
    def __init__(self, data, *args, neg_sampling_ratio: float = 0.0, **kwargs):
        super().__init__(data, *args, **kwargs)
        self.neg_sampling_ratio = neg_sampling_ratio

        if issubclass(self.data_cls, (Data, RemoteData)):
            self.num_src_nodes = self.num_dst_nodes = data.num_nodes
        else:
            self.num_src_nodes = data[self.input_type[0]].num_nodes
            self.num_dst_nodes = data[self.input_type[-1]].num_nodes
```

LinkNeighborLoader calls LinkNeighborSampler
Then the sampler will call the cugraph Sampling functions

pyG integration backend

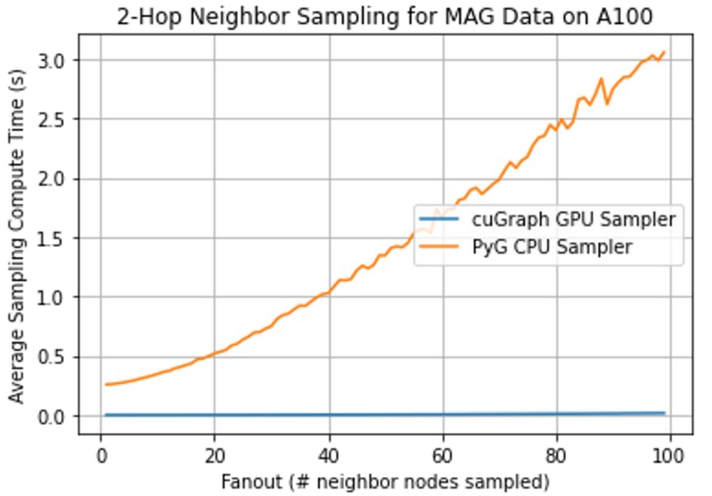
```python
class CuGraphData(BaseData, RemoteData):
    reserved_keys = [
        PropertyGraph.vertex_col_name,
        PropertyGraph.src_col_name,
        PropertyGraph.dst_col_name,
        PropertyGraph.type_col_name,
        PropertyGraph.edge_id_col_name,
        PropertyGraph.vertex_id_col_name,
        PropertyGraph.weight_col_name
    ]
```

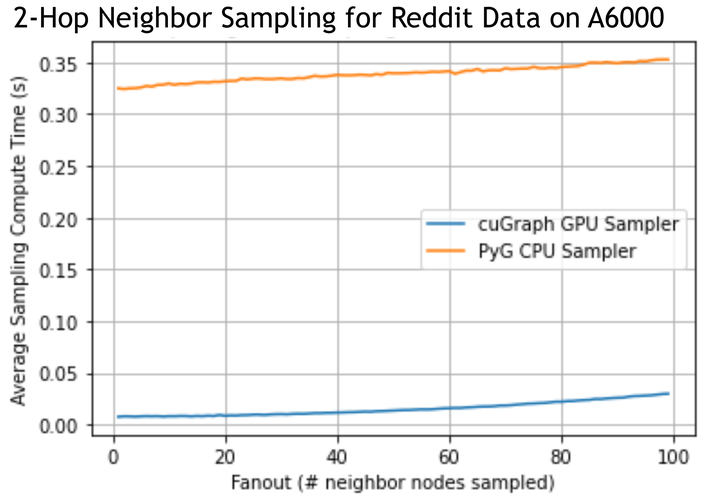
CuGraphData will call RemoteData, which contrains Cugraph data structures
And it contains the sampling functions, and coresponding property functions

```python
def neighbor_sample(
        self,
        index: Tensor,
        num_neighbors: Tensor,
        replace: bool,
        directed: bool) -> Any:

    start_time = datetime.now()

    if not isinstance(index, Tensor):
        index = Tensor(index).to(torch.long)
    if not isinstance(num_neighbors, Tensor):
        num_neighbors = Tensor(num_neighbors).to(torch.long)

    index = index.to(self.device)
    num_neighbors = num_neighbors.to(self.device)

    if self.device == 'cpu':
        index = np.array(index)
        # num_neighbors is required to be on the cpu per cugraph api
        num_neighbors = np.array(num_neighbors)
    else:
        index = cupy.from_dlpack(index.__dlpack__())
        # num_neighbors is required to be on the cpu per cugraph api
        num_neighbors = cupy.from_dlpack(num_neighbors.__dlpack__()).get()
```

PERFORMANCE

- cuGraph Sampling on MAG dataset is several orders of magnitude faster
  - Will soon demonstrate this in training now that we have hetero support
- On reddit data our overhead nullifies most speedup
  - Hoping for better results on MAG
  - Medium term need performance improvements to handle MAG240
  - Eventually need to compute multiple batches at once for max efficiency





CURRENT STATUS

Showed key performance improvements over base PyG

Running about 2x faster on Reddit (with old API)

About to benchmark MAG (with new API)

SG+MG+heterogeneous support available (but not yet merged)

PyG+GaaS needs to be updated for the new PyG API

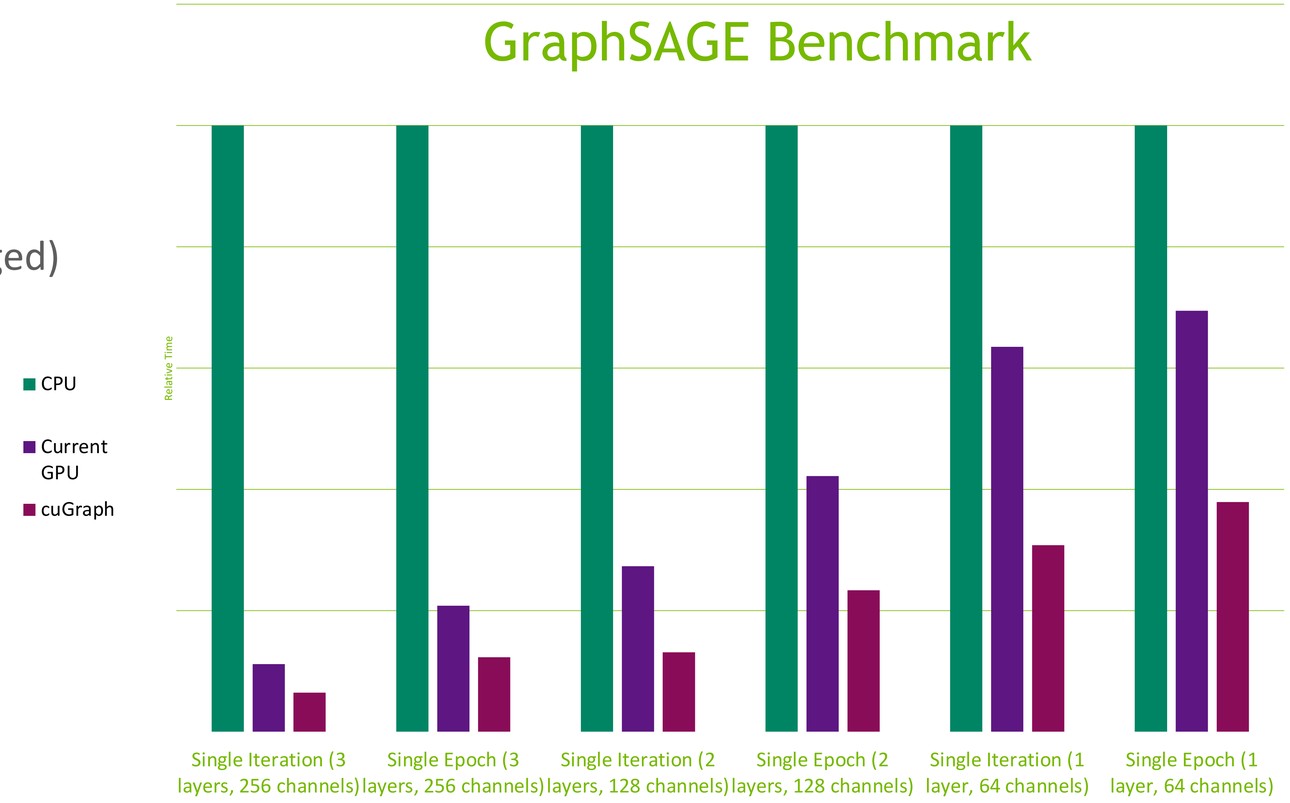


Chart is training time

- Pure CPU
* GPU Training with CPU Sampling
- GPU Training with GPU Sampling from cuGraph

Result of pyG

```
torch.Size([39579, 165]) torch.Size([6985, 165])
 - loss: 0.8107657188463656 - roc: 0.5238861550436414
 - loss: 0.2051875434556261 8 - roc: 0.5277890684027389
 - loss: 0.157265193687867 6 - roc: 0.742887957220662 8
 - loss: 0.1178333249246483 5 - roc: 0.851649820265008 4
 - loss: 0.0901878021131677 2 - roc: 0.87361217482166 6
 - loss: 0.0709033382866432 9 - roc: 0.9034942641676317
 - loss: 0.0600361998716106 2 - roc: 0.915255236270533 2
 - loss: 0.0490124354158919 4 - roc: 0.9338447239906545
 - loss: 0.0444339266850774 5 - roc: 0.945650890403649 4
 - loss: 0.0409567714850883 34 - roc: 0.956638703138780 5
```

We also get similar result here in pyG.


THANK YOU !
NVIDIA