DynaGraph: Dynamic Graph Neural Networks at Scale

Mingyu Guan
Georgia Institute of Technology
Atlanta, USA

Anand Padmanabha Iyer
Microsoft Research
Redmond, USA

Taesoo Kim
Georgia Institute of Technology
Atlanta, USA

Abstract

In this paper, we present DynaGraph, a system that supports dynamic Graph Neural Networks (GNNs) efficiently. Based on the observation that existing proposals for dynamic GNN architectures combine techniques for structural and temporal information encoding independently, DynaGraph proposes novel techniques that enable cross optimizations across these tasks. It uses cached message passing and timestep fusion to significantly reduce the overhead associated with dynamic GNN processing. It further proposes a simple distributed data-parallel dynamic graph processing strategy that enables scalable dynamic GNN computation. Our evaluation of DynaGraph on a variety of dynamic GNN architectures and use cases shows a speedup of up to 2.7× compared to existing state-of-the-art frameworks.

1 Introduction

The recent past has seen an increasing interest in Graph Neural Networks (GNNs) due to their ability to produce superior results in a wide variety of domains ranging from social networks to chemistry and even medicine [32, 36, 39, 45]. GNNs enable such applications by generating *embeddings* of graph entities—the nodes and edges—using a combination of graph structure and feature information. Today, a wide variety of GNN architectures exist that vary in how the embeddings are generated. To support these GNN architectures, state-of-the-art GNN frameworks have incorporated optimizations that enable them to handle large graphs, with billions of nodes and edges [8, 9, 21, 30, 35, 46, 49–51].

While existing GNN frameworks have enabled scalability, ease of programming, and several other desirable properties, they currently assume that the input graph is *static*: that is, the nodes, edges and features associated with the graph do not change once ingested into the framework. However, real-world graphs are *dynamic* in nature - they evolve over time. For instance, social network graphs evolve as new users join and content is generated, knowledge graphs are constantly augmented with new information, and drug interaction networks evolve over time with new interaction trial results [1, 27, 29]. The ability to process dynamic graphs can be useful for many scenarios that can benefit from GNNs. For instance, traffic forecasting systems can predict future traffic statistics based on historical data flows with the help of GNNs [28, 57, 59]. Thus, supporting dynamic graphs is a requirement for enabling many GNN applications.

GRADES & NDA'22, June 12, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9384-3/22/06.
https://doi.org/10.1145/3534540.3534691

In this paper, we focus on *efficient* dynamic GNN training. The natural way to support dynamic graphs is to view them as a series of static snapshots, as proposed by existing time-evolving graph processing systems [19, 26, 34]. While the main task in time-evolving graph processing systems is to execute graph algorithms on a series of snapshots of the graph, dynamic GNNs have the additional task of capturing the temporal dependencies. Consequently, the general approach in dynamic GNN is to combine spatial embedding techniques that leverage structural information in the graph with temporal information encoding techniques that capture the dynamicity aspect in the generated embeddings [3, 28, 33, 38, 43]. The spatial embedding is enabled by static GNN techniques, such as GCNs [15, 25, 54] and GAT [47], whereas a common approach to capture temporal encoding is to use Recurrent Neural Networks (RNNs) such as LSTMs [17] or GRUs [6]. Hence, existing dynamic GNN proposals combine GNNs and RNNs by stacking them in an alternating fashion [33, 43], integrating GNN and RNN operations [3, 38, 43, 59], or using autoencoding or generative models [13, 14]. Unfortunately, all these approaches lead to significant performance issues and create scalability bottlenecks.

We present DynaGraph[1], a system that enables efficient dynamic GNN training. DynaGraph is based on the observation that the root of inefficiencies in existing dynamic GNN proposals stem from the fact that they combine the structural and temporal embedding operations *independently*. As a result, they miss opportunities to optimize many underlying operations. Thus, DynaGraph focuses on reducing or eliminating such inefficiencies by leveraging the computational structure of spatial and temporal embedding generation and exploiting opportunities to utilize this knowledge *across* them. DynaGraph proposes three techniques to provide efficient training of dynamic GNNs.

First, DynaGraph proposes *cached message passing* as a technique to eliminate redundant neighborhood aggregation. This is based on our observation that the popular approach to combining GNN and RNN is by replacing the matrix multiplication in the RNN using graph convolution operations. We denote this combination as **GraphRNN** cell. As a result, each GraphRNN cell would consist of multiple GNN operations. For instance, a LSTM-based dynamic GNN (GraphLSTM) cell consists of 8 GNN operations. While each gate of GraphRNN ingests two kinds of operations—time-dependent and time-independent, each kind performs neighborhood aggregation independently on different inputs. However, inputs are shared or partially shared across gates and performed neighborhood aggregation on the same snapshot at a timestep. For example, four gates of a GraghLSTM all take node features as inputs for time-dependent GNN operations and hiddens from the previous timestep for time-dependent GNN operations. DynaGraph uses this cross-operation information to build cached message passing, where messages between graph entities are cached at the destination and reused. To

[1] for **Dyna**mic **Graph** Neural Network Processing System.

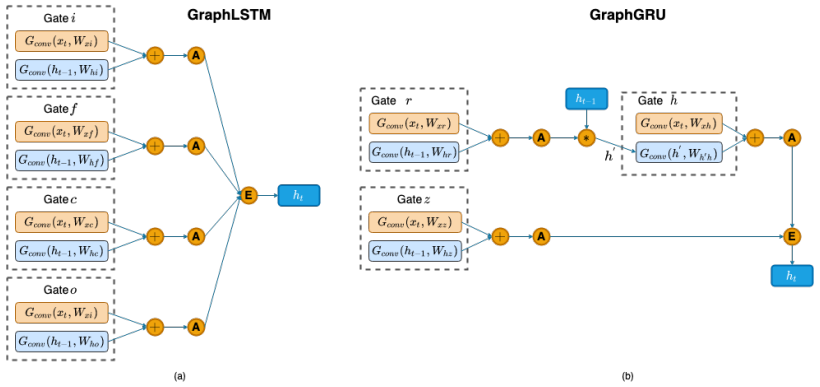

Figure 1: Dynamic GNNs combine techniques for structural information encoding (e.g., GNNs) with techniques for temporal information encoding (e.g., RNNs). While there are many ways to combine GNNs and RNNs, the most adopted approach is to replace the matrix multiplications in the RNN, such as LSTMs or GRUs, with graph convolution G_{conv}. Time-independent graph convolutions in the orange boxes take current nodes representation x_t as inputs, while time-dependent graph convolutions in the blue boxes depend on the hiddens h_{t-1} from previous timesteps.

identify redundancy in such messages, DynaGraph uses a simple interface that uniquely identifies messages across spatial and temporal operations and caches them for future use. It then exposes the cache using the well-known PUT and GET interface.

Second, it uses *time-step fusion* to reduce the underutilization of GPUs for small real-world datasets. We base this on our observation that existing dynamic graphs for various applications typically have a small number of input features, often nodes in the graphs are associated with less than a few 10s of features. Moreover, spatio-temporal graphs, in which node features change over time while the underlying graph structure is static, are widely used in various applications such as traffic forecasting[56] and pandemic forecasting[22]. Due to the static graph structure, the neighborhood aggregation of time-independent GNNs across timesteps can be fused to one to save computation. DynaGraph leverages this information and fuses the inputs for time-independent operations in dynamic GNN, thus reduce redundant neighborhood aggregation and avoiding GPU underutilization.

Third, DynaGraph proposes a data-parallel execution model for dynamic GNN training in a distributed setting. We observe that existing frameworks that support distributed GNN training, such as DGL, use graph partitioning on a single graph (snapshot) as a primary technique to distribute work across machines. However, dynamic GNNs typically have a large number of snapshots of the graph, where they are trained in a sequence manner. Extending the same approach to distributing dynamic GNNs results in poor performance due to partitioning and communication overhead. Instead, DynaGraph proposes partitioning the snapshots at sequence level across the GPUs in the cluster. It then executes the GNN-RNN operation in a data-parallel fashion, thus providing efficient distributed training.

The combination of these techniques enables DynaGraph to outperform existing state-of-the-art GNN processing frameworks for dynamic GNN training. For instance, on a wide range of dynamic GNN architectures, DynaGraph is able to provide up to 2.7× faster training times compared to DGL [50] and PyG Temporal [8, 40], two popular GNN training frameworks.

2 Background

2.1 Dynamic Graph Neural Networks

While most GNNs consider the input graph to be static, real-world graphs tend to be dynamic and vary over time. Examples of dynamic graphs can be seen in almost all domains that GNNs are suited for, ranging from social media to medicine. Consequently, machine learning researchers have investigated dynamic GNN architectures: [28, 57, 59] apply dynamic GNNs for traffic forecasting to predict the future traffic speeds of a sensor network given historic traffic speeds and road networks. Dynamic GNNs have also been used for epidemiological reporting tasks [37, 41]. Other applications of dynamic GNNs include fraud detection [31, 48, 52] and human motion prediction [55].

Dynamic GNNs have focused on two aspects of dynamicity: the dynamicity of the graph structure, and the dynamicity of the input features. The graph structure and the input features can be independently dynamic; for instance, the features may remain static while the structure changes over time, and vice-versa. For example, spatio-temporal GNNs [28, 55, 57], one kind of dynamic GNNs, are designed specifically for graphs where the structure of the graph is static and only the input features change over time.

The general approach in dynamic GNN is to combine techniques for structural information encoding with techniques for temporal information encoding. The common practice is to use a GNN to encode the structural information, while Recurrent Neural Networks (RNNs) or self-attention networks are used to encode temporal information [40, 44].

2.2 Challenges in Dynamic GNN Training

Though dynamic GNNs have been shown to be effective for a wide variety of tasks, efficiently training dynamic GNNs faces many challenges.

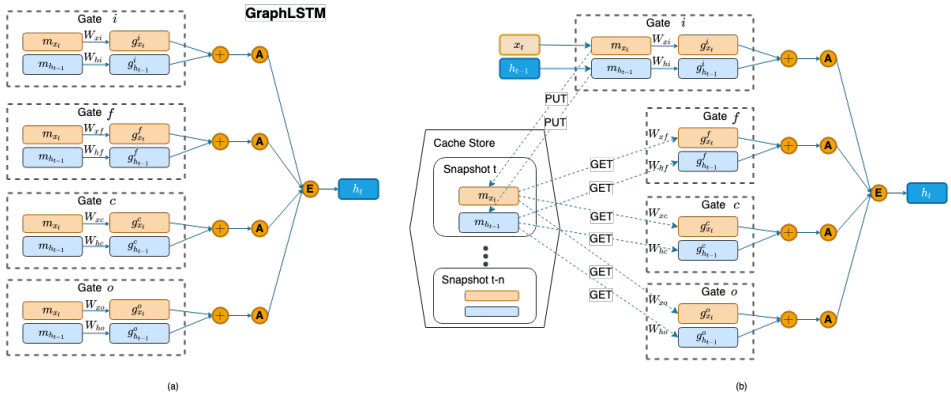

Figure 2: (a) Existing systems apply graph convolution operations independently on gates in each iteration, which results in duplicate messages m_{x_t} and $m_{h_{t-1}}$ being generated and communicated. (b) DynaGraph proposes computing these messages only once, then caching and reusing them across gates to reduce computation and communication. For GraphLSTM, when Only *Gate i* needs to compute the intermediate messages - m_{x_t} and $m_{h_{t-1}}$. DynaGraph stores the m_{x_t} and $m_{h_{t-1}}$ in the cache, which can be reused by other Gates instead of recomputing them.

2.2.1 Challenge #1: Redundant Neighborhood Aggregation

As we discussed earlier (§2.1), dynamic GNNs replace matrix multiplications in each of the gates of RNNs with GNN layers (fig. 1). Just as RNNs consist of two classes of matrix multiplications [58]—those that depend solely on the inputs and those which have recursive dependence on previous states, we can categorize the GNNs in the dynamic GNN into two categories. The first only depends on the input features of the nodes, while the second depends on previous hidden states. There is no redundancy in matrix multiplications of RNNs since they use different weights. However, for dynamic GNNs, graph convolutions operated on the same graph and node features perform the same neighborhood aggregation, before any linear transformation involving weights, resulting in poor performance. Hence, this challenge is unique to dynamic GNNs but not applicable to RNNs.

2.2.2 Challenge #2: Ineffectiveness of Graph/Feature Partitioning in Distributed Training

State-of-the-art distributed GNN frameworks, such as Deep Graph Library (DGL), only supports static GNNs in distributed settings. Since static GNNs only need to process a single graph, DGL partitions both the graph structure and node/edge features across machines and then uses data parallelism to train a static GNN. A natural approach to extending this to dynamic GNNs is to partition each snapshot individually so that the machines can compute them in parallel. However, this scheme is inefficient for two reasons. First, the number of snapshots in a dynamic graph dataset can be very large, ranging from 100s to many 1000s. Partitioning and maintaining a large number of snapshots can be expensive. Second, the graph structure and the node/edge features in each snapshot may vary, which means that partitioning them may lead to poor performance due to communication requirements.

3 Dynamic GNN Training with DynaGraph

DynaGraph is based on our key observation that existing approaches to dynamic GNN processing are by combining GNN and RNN operations *independently* (§2). Consequently, the graph convolutions dominate the execution time. DynaGraph proposes a new, efficient way of dynamic GNN training that reduces the overhead by cross-layer optimizations across GNN and RNN operations. To do so, DynaGraph incorporates many techniques, which we describe in detail in this section.

3.1 Cached Message Passing

Generally, many GNN models can fit into the message passing paradigm proposed by [10], which consists of two main steps: 1) Aggregate neighbor's representation for each node with certain reduce function such as summation and average; 2) Apply a linear projection to the aggregated representation followed by a non-linearity.

In each iteration, each gate of a GraphRNN cell computes two graph convolutions. However, we notice that the convolution operations across all gates result in generating the same intermediate message passing results m_{x_t} for all time-independent graph convolutions and $m_{h_{t-1}}$ for all time-dependent graph convolutions, respectively, where x_t is current node representation and h_{t-1} is previous hidden states [2]. This is because m_{x_t} ($m_{h_{t-1}}$) **only depends on x_t (h_{t-1}) and the graph properties such as structure and edge weights, which is the same for all gates**. As an example, we split graph convolutions in GraphLSTM into two steps in fig. 2 (a): 1) compute intermediate message passing results m_{x_t} ($m_{h_{t-1}}$) and 2) compute final output of graph convolution g_{x_t} ($g_{h_{t-1}}$) using m_{x_t} ($m_{h_{t-1}}$). Here, we notice that across the gates, the same m_{x_t} ($m_{h_{t-1}}$) are generated multiple times during the graph convolution operation. This results in large overhead, especially in distributed settings. Thus, DynaGraph proposes caching these messages and reusing them across gates in an iteration.

Moreover, it is common that dynamic GNNs are trained in a sliding-window fashion using a sequence of snapshots, which predicts either next timestep or multiple future time steps. Unlike other deep learning tasks, it is special that timesteps as labels (ground truth) in current sequence may be inputs in subsequent sequence(s)

[2]The time-dependent graph convolution in Gate h of GraphGRU should be computed separately, as it depends on the results of Gate r as shown in fig. 1.

in time-series forecasting tasks. For example, let S_t be the snapshot at time t, S_1 to S_4 are used as inputs to predict S_5 in the first sequence. Next, the sliding window moves forward one timestep so that S_2 to S_5 are used as inputs to predict S_6 in the second sequence, and so on and so forth. This brings another opportunity to reuse intermediate message passing results across snapshots since the neighborhood aggregation has already been performed on some input snapshots when they were used as labels in previous sequence(s).

As shown in fig. 2 (b), for each snapshot, DynaGraph uses a dedicated cache to store intermediate messages if needed (please see §3.4 for a description of DynaGraph 's API that enables this). The cache store resides in GPU if it has enough memory; otherwise, CPU cache is used for large intermediate messages. The cached messages are uniquely identified by the snapshot index and graph convolution inputs (node features or hiddens). When new messages need to be generated, DynaGraph checks its cache to determine whether there is an existing cached message that can be reused. If so, the cached message can be reused thus avoiding the need for computation and communication; otherwise, the intermediate messages need to be computed from scratch and cached if they can be reused. When the training process on a snapshot as both input and label is finished, the cache store would automatically evict all cached messages for that snapshot. Users can also evict the cached messages earlier by an explicit evict call if they cannot be reused anymore.

3.2 Input Timestep Fusion

Similar to traditional RNNs, gates of GraphRNN operate on time-dependent and time-independent inputs. We notice that the time-independent graph convolution operation operates independently at each timestep. This is necessary for graphs where the underlying structure changes over time as the convolution results depend on it. However, for spatio-temporal graphs, it is not efficient to execute time-independent operations separately at each time step, especially when nodes have low dimensional features, which results in substantial overhead to the execution time. Fortunately, it is relatively straightforward to mitigate this overhead. DynaGraph uses input timestep fusion to achieve this.

For each GraphRNN cell, DynaGraph first concatenates the inputs along the timestep dimension. Then it executes a single graph convolution operation on the concatenated inputs for all timesteps in the sequence. We note that this optimization brings more benefits for small datasets, especially when GPU is underutilized due to small graph size or low dimensional features.

3.3 Distributed Training

Though most available real-world datasets for dynamic GNNs have relatively small graph structures and/or low dimensional features, we see a clear future direction to enable dynamic GNNs for large-scale graphs that have millions or even billions of nodes and edges. In addition, **dynamic graphs can be quite large due to the number of snapshots they contain.** As an example, in the METR-LA dataset, while each individual snapshot is fairly small, the number of snapshots is large making the dataset very large(§5). Moreover, with sequence modeling, dynamic GNNs are trained on a sequence of snapshots in a sliding-window fashion which further increases the size of samples. Thus, DynaGraph supports distributed execution of dynamic GNN training for scalability.

3.3.1 Graph Partitioning

Unlike the state-of-the-art static GNN frameworks, which partitions at snapshot/graph level to train a static GNN, DynaGraph partitions dynamic datasets at a sequence level considering the temporal dimension. For each sequence of snapshots, DynaGraph partitions it across machines as even as possible. We use a simple random partition scheme to distribute data evenly across devices based on the number of nodes, features, and timesteps. Specifically, if node A in the first snapshot reside in a device, node A in the subsequent snapshot(s) of the same sequence also belongs to the same device. This simple partition scheme enables data-parallel distributed execution of dynamic GNN training by ensuring that the same nodes across snapshots in a sequence reside in the same device to reduce network communication of nodes exchanging.

3.3.2 Data-parallel Execution

To enable distributed training for dynamic GNNs, we partition sequences of snapshots randomly and evenly across nodes. Note that if the underlying graph structure is static, we only store one copy of that instead of replicating it in memory; otherwise, graph structures get partitioned and fed into training together with its associated snapshots. Moreover, unlike existing GNN systems, which replicate snapshots in time-series forecasting tasks, DynaGraph uses a sliding window to iterate over the datasets to achieve memory-efficient training. Each GPU computes the forward and backward phase for its minibatch independently and then synchronize the model parameters across them.

3.4 DynaGraph API

```
class GCN(nn.module):
  def __init__(in_feats, out_feats):
    linear = nn.Linear(in_feats, out_feats)
  # generate messages
  def msg_udf(g, feats, u, v): return msg
  # reduce aggregated messages
  def reduce_udf(agg_msg): return sum(agg_msg)
  # compute new representation
  def update(msg_rst):
    return linear(msg_rst)
  def forward(g, feats):
    msg_rst = msg_pass(g, feats, msg_udf, reduce_udf).cache
        ↪ ()
    return update(msg_rst)

gcrn = integrate(GraphLSTM, GCN)
seq2seq_gcrn = stack_seq_model(in_feats, out_feats,
     ↪ num_layers, gcrn)
```

Listing 1: Using DynaGraph 's API to implement a seq2seq GCRN-LSTM model.

DynaGraph exposes all its optimization via a simple API that makes it easy for a developer to leverage it in a new dynamic GNN architectures. The API, shown in table 1, consists of the following five functions:

- **cache** stores intermedia message passing result in global cache if it has not been cached yet; otherwise, do nothing.

API	Description
cache()	cache caller function outputs; do nothing if already cached.
msg_pass(g, feats, msg_udf, reduce_udf)	Computes intermediate message passing results based on user-defined message function *msg_udf* and reduce function *reduce_udf*.
update(msg_rst)	Computes output representation from intermediate message passing results.
integrate(rnn, gnn)	Integrates a GNN into a GraphRNN to create a dynamic GNN.
stack_seq_model(in_feats, out_feats, num_layers, gcrn)	Stacks dynamic GNN layers to an encoder-decoder structure.

Table 1: The simple APIs exposed by DynaGraph for dynamic GNNs.

- **msg_pass** is a user-provided function to generate intermediate message passing results based on user-defined message function and reduce function.
- **update** is a user-provided function to generate hidden representation by applying zero or more element-wise and non-element-wise NN operations on intermediate message passing result.
- **integrate** integrates a GNN into a GraphRNN, such as GraphGRU and GraphLSTM.
- **stack_seq_model** stacks dynamic GNN layers to an encoder-decoder seq2seq model.

Listing 1 outlines how GCRN-LSTM [43] can be implemented in DynaGraph. Using DynaGraph's API, the developer can build dynamic GNNs simply and optimizations discussed in §3 can be done under the hood. To define a GCN layer, the developer first defines the *msg_pass* function that aggregates the representation of its neighborhoods by applying summation (see *reduce_udf*) over the incoming source vertex representation (see *msg_udf*). Note that in *forward* function, *msg_pass* calls *cache()* to cache the outputs if not yet cached; otherwise it performs no operation. Next, the intermediate message passing results are fed into a fully connected layer (see *update*). The developer then can use *integrate* function to integrate GCN into LSTM to create a dynamic GNN layer. Moreover, if a seq2seq structure is needed, *stack_seq_model* can be used to easily stack the dynamic GNN layers to an encoder-decoder structure, which enables optimizations specially for seq2seq structures behind the scenes, such as timesteps fusion and reuse message passing results across snapshots. We discuss DynaGraph's implementation details in §4.

4 Implementation

DynaGraph is implemented on top of Deep Graph Library (DGL) [50], a popular open-source framework for training GNN models. DynaGraph uses DGL as a graph propagation engine for neighborhood aggregation using message passing primitives and other graph related operations, and PyTorch as the neural network execution runtime. We extended DGL in multiple ways to support the optimizations of fusing timesteps, caching and reusing message passing results, as well as distributed Dynamic GNN training. First, we implement the optimization of input timestep fusion for seq2seq models by automatically concatenating time-independent inputs along the feature dimension as a single tensor and splitting the outputs back for each snapshot. Second, we enable reusing intermediate message passing results using caching. DynaGraph caches the outputs of **msg_pass** (see §3.4) if not yet cached. We implement a dictionary on each snapshot to be a mutable mapping to the cached results. When performing the repeated message passing operations, cached results can be extracted and reused to save computational overhead. Last, we implement graph partitioning and data-parallel model for distributed dynamic GNN training. We partition sequences of snapshots randomly and evenly across nodes. We repeat the random partition every epoch to guarantee randomness. We wrap a user-defined dynamic GNN model with *DistributedDataParallel()* from PyTorch for weight synchronization and update.

Graph	Nodes	Features	Snapshots
METR-LA [20]	207	2	34,272
PEMS-BAY [28]	325	2	52,116
METR-LA-Large	423,936	128	34,272
PEMS-BAY-Large	665,600	128	52,116
Chickenpox Hungary [40]	20	4	522
Wikipedia Math [40]	1068	8	731
Windmill Output [40]	319	8	17,472

Table 2: Graph datasets used in evaluating DynaGraph.

5 Evaluation

We evaluate DynaGraph on several real-world graphs and dynamic GNN architectures. The key results from our evaluation are:

- DynaGraph is able to improve performance compared to DGL by up to 2.5× and PyG Temporal by up to 2.7×.
- DynaGraph exhibits superior scaling characteristics compared to DGL, and its benefits increase as the number of GPUs in the cluster increase. DynaGraph's distributed training technique is able to match the published accuracy results for known training tasks.

Experimental Setup: All of our experiments were conducted on a GPU cluster with 8 nodes, each of which has dual 12-core Intel Xeon Gold 6226 CPU, 384 GB of RAM, and two NVIDIA Tesla V100 16 GB GPUs, if not specified. GPUs on the same node are connected via a shared PCIe interconnect, and nodes are connected via a 10 Gbps Ethernet interface. All servers run 64-bit Red Hat Enterprise Linux 7.6 with CUDA library v11.1, PyTorch v1.9.0, DGL v0.7, and Pytorch Geometric v2.0.2.

Datasets & Comparison: We list the five graphs we use in our experiments in table 2. The first two are traffic forecasting datasets from [28]: METR-LA [20], collected from 207 loop detectors on highways in Los Angeles County in aggregated 5-minute intervals for 4 months, and PEMS-Bay [28] collected from 325 traffic sensors in the Bay Area for 6 months. When training on these two datasets, 70%

Dataset	Model	DGL	DynaGraph	Speedup
METR-LA	DCRNN	97.51	68.36	1.43x
	GCRN-GRU	97.04	60.69	1.60x
	GCRN-LSTM	138.26	59.85	2.31x
PEMS-BAY	DCRNN	146.22	102.97	1.42x
	GCRN-GRU	155.12	92.84	1.67x
	GCRN-LSTM	217.23	96.97	2.24x

Table 3: DynaGraph is able to gain up to 2.31× improvement in epoch time (sec) over DGL on a single machine.

Dataset	Model	PyG Temporal	DynaGraph	Speedup
Chickenpox Hungary	TGCN	2.17	0.98	2.21x
	GC-LSTM	2.44	1.06	2.30x
	GCRN-GRU	3.07	1.61	1.91x
	GCRN-LSTM	4.61	1.71	2.70x
Wikipedia Math	TGCN	3.76	1.62	2.32x
	GC-LSTM	3.97	1.80	2.21x
	GCRN-GRU	4.51	2.74	1.65x
	GCRN-LSTM	5.88	2.54	2.31x
Windmill Output	TGCN	-†	-†	-†
	GC-LSTM	89.02	41.99	2.12x
	GCRN-GRU	111.93	68.97	1.62x
	GCRN-LSTM	149.38	65.52	2.28x

†: Not available since baseline runs out of memory.

Table 4: DynaGraph is able to gain up to 2.7× improvement in epoch time (sec) over PyG Temporal on a single machine.

of data is used for training, 20% is used for testing while the remaining 10% for validation, which follows DCRNN [28] paper and DGL settings. The latter three—Chickenpox Hungary [41], Wikipedia Math [40], Windmill Output [40]—were released recently by PyG Temporal. When training on these two datasets, 90% of data is used for training, 10% is used for testing, as used in PyG Temporal paper [40]. We compare DynaGraph against DGL [50] and PyG Temporal [8, 40], two popular state-of-the-art GNN processing frameworks. We note that PyG does not support distributed training, and DGL's distributed training only support static graphs - only a single distributed graph (*DistGraph*) is allowed according to their official document. Hence, we apply the same distributed data-parallel model as DynaGraph on DGL, denoted as **DGL-P**, but without the overhead reduction techniques that it incorporates (timestep fusion and cached message passing). Since available real-world datasets such as METR-LA and PEMS-Bay, are not large enough for the cluster settings we use (e.g., 8 or 16 GPUs), **we increase the size of the dataset by simply replicating the dataset many times so as to obtain large-scale datasets**, i.e., METR-LA-Large and PEMS-Bay-Large, that can benefit from distributed training. For example, in METR-LA-Large, a sequence of 12 snapshots contains more than 5 million nodes in total.

Models: We use five different DGNN models: GCRN-GRU [43], GCRN-LSTM [43], DCRNN [28], TGCN [59], GC-LSTM [3]. These models represent the state-of-the-art dynamic GNNs that have been applied to various tasks. Unless otherwise stated, when compared to DGL, we use a 2-layer seq2seq, encoder-decoder structure to predict multiple timesteps (a sequence of snapshots). Here, we set sequence length, also known as sliding-window size, to be 12, following official examples from DGL. Similarly, when comparing against PyG Temporal, we use a single-layer dynamic GNN followed by a fully connected feedforward layer to predict a single snapshot at next timestep, as used in official PyG Temporal examples. Our main metric for comparison is the epoch time, the time taken by the framework to complete one pass on the entire dynamic graph dataset (all the snapshots). We use a hidden size of 64 for DGL and 32 for PyG Temporal experiments. Minibatch size is set to 64 for single-machine DGL experiments. For PyG Temporal experiments, we follow the cumulative training in official examples. All reported timing results are measured after warm-up and averaged over 10 repetitions.

5.1 Single Machine Performance

We first present the performance of DynaGraph on a single machine. table 3 shows the comparison between DynaGraph and DGL, and table 4 shows the comparison between DynaGraph and PyG Temporal for this experiment.

We observe that DynaGraph is able to outperform comparison systems for all datasets and models. Compared to DGL, DynaGraph's speedups range from 1.42× to 2.31×. Compared to PyG Temporal, DynaGraph is able to attain speedups between 1.62× and 2.7×. We notice that the LSTM-based models benefit more from DynaGraph's techniques compared to GRU-based models. This is because of the increased number of gates in the LSTM-based models, where cached message passing is able to obtain more opportunities for message reuse. The reason for DynaGraph's increased performance against PyG Temporal can be attributed to the different

Dataset	Model	DGL-P	DynaGraph	Speedup
METR-LA-LARGE	DCRNN	2,223	1,576	1.41x
	GCRN-GRU	1,993	1,261	1.58x
	GCRN-LSTM	2,825	1,332	2.12x
PEMS-BAY-LARGE	DCRNN	3,466	2,249	1.54x
	GCRN-GRU	3,570	2,122	1.68x
	GCRN-LSTM	5,151	2,308	2.23x

Table 5: DynaGraph is able to gain up to 2.23× improvement in epoch time (sec) over DGL-P in a distributed setting.

datasets and architectural differences in how GNN operations are implemented compared to DGL. We also notice that the speedups of the same model for different datasets are variant. Specifically, a minor degradation of speedup presents for large datasets like Windmill Output. This is due to heavier data movement between CPUs and GPUs for larger datasets, which cancels out some performance benefits with DynaGraph's techniques.

5.2 Distributed Training Performance

Next, we present the performance of DynaGraph in a distributed setting with data parallelism. As mentioned before, this comparison experiment uses only DGL, since PyG Temporal does not support distributed execution for dynamic GNN implementations yet. Since DGL does not support distributed training of dynamic GNNs but only static graphs (it only allows creation of a single *DistGraph* object), we apply the same distributed data-parallel model as DynaGraph on DGL, denoted as **DGL-P**, but without the overhead reduction techniques it incorporates (timestep fusion and cached message passing). The results of this experiment are shown in table 5. Here, DynaGraph's speedups range up to 2.23× compared to DGL-P. The results indicate that DynaGraph is able to carry over its single-machine performance to a distributed setting. The speedups present a minor degradation compared with the single-machine setting, which can be attributed to overhead of gradient synchronization across machines.

5.3 Scaling Characteristics

This experiment evaluates the strong scaling properties of DynaGraph with its optimization. We choose the METR-LA-Large dataset and train GCRN-GRU and GCRN-LSTM model on it. To understand the scaling properties, we vary the number of machines, thereby varying the number of GPUs used by DynaGraph and DGL-P. We report the average throughput (the number of samples processed per second) in Figure 3.

DynaGraph exhibits near-linear scaling characteristics and maintains its benefits (up to 2.54x) with the increase of the number of machines, compared with DGL-P. This is because DynaGraph's techniques are independent of the number of machines and there is no extra network communication to enable them. We see a minor degradation of speedup, for example, speedup of GCRN-LSTM model is 2.54x for two nodes and 2.23x for eight nodes, due to overhead of gradient synchronization across more machines.

5.4 Accuracy

This experiment shows correctness of our approach. We train various models on DGL and PyG Temporal and evaluate them on 5 datasets. For DGL, DCRNN are evaluated based on Mean Absolute

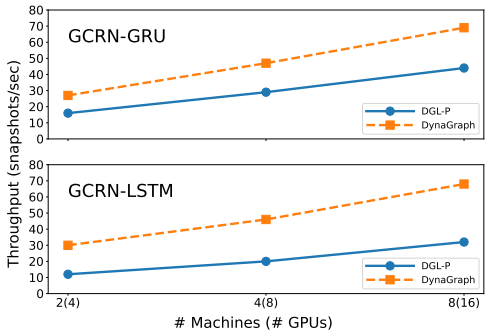

Figure 3: DynaGraph is able to scale more gracefully as the number of GPUs in the cluster increase.

Error (MAE); for PyG Temporal, GC-LSTM, GCRN-GRU and GCRN-LSTM are evaluated based on Mean Squared Error (MSE). Models were trained for 100 epochs to calculate the average MAE or MSE from 10 experimental runs. The choices are made by following the official examples in DGL and PyG Temporal.

As shown in table 6, DynaGraph can achieve the same accuracy as DGL and PyG Temporal. This is because DynaGraph accelerates DGNN training by eliminating redundant computation, which does not affect computational correctness. We observe evident difference between Reported MAE/MSE and Test MAE/MSE for some models. For DCRNN, we attribute this to the difference in detailed implementation of [28] and DGL. For PyG models, this may be because of the differences in the number of node features used, which is not mentioned explicitly in [40]. In our experimental runs, we follow the default setting in the official examples of PyG Temporal as shown in table 2. Nevertheless, DynaGraph can achieve same test accuracy as baselines with its optimizations.

6 Shortcomings

Finally, we discuss cases where DynaGraph fails to provide benefits. A fundamental assumption made by DynaGraph is that the intermediate message passing results can be shared by all GNNs in a GraphRNN cell. However, this is not applicable when an attention mechanism is involved like GAT [47], since there are learnable weights involved when computing the attention score between two neighbors, which makes intermediate message passing results different for GNNs in a GraphRNN cell.

7 Related Work

(Dynamic) Graph Processing Systems: Enabling graph algorithms on large-scale, real-world graphs has long been an active area

Framework	Model	Dataset	Reported MAE/MSE	Test MAE/MSE	DynaGraph MAE/MSE
DGL	DCRNN	METR-LA	3.60	2.712 ± 0.024	2.721 ± 0.021
		PEMS-BAY	2.07	1.618 ± 0.021	1.609 ± 0.018
PyG Temporal	TGCN	Chickenpox Hungary	1.111 ± 0.022	1.149 ± 0.023	1.153 ± 0.018
		Wikipedia Math	0.846 ± 0.020	0.616 ± 0.015	0.617 ± 0.014
		Windmill Output	-†	-‡	-‡
	GC-LSTM	Chickenpox Hungary	1.116 ± 0.023	1.096 ± 0.004	1.101 ± 0.007
		Wikipedia Math	0.852 ± 0.016	0.686 ± 0.011	0.691 ± 0.016
		Windmill Output	-†	1.036 ± 0.012	1.034 ± 0.009
	GCRN-GRU	Chickenpox Hungary	1.132 ± 0.023	1.119 ± 0.010	1.104 ± 0.012
		Wikipedia Math	0.837 ± 0.021	0.706 ± 0.005	0.713 ± 0.008
		Windmill Output	-†	1.021 ± 0.013	1.019 ± 0.015
	GCRN-LSTM	Chickenpox Hungary	1.119 ± 0.022	1.044 ± 0.011	1.038 ± 0.013
		Wikipedia Math	0.868 ± 0.018	0.823 ± 0.021	0.831 ± 0.017
		Windmill Output	-†	1.043 ± 0.021	1.042 ± 0.018

†: Not present in official documents.

‡: Not available since baseline runs out of memory.

Table 6: DynaGraph achieves the same accuracy as DGL and PyG Temporal. We collect Reported MAE/MSE from [28] for DCRNN and [40] for PyG Temporal models. Test MAE/MSE shows actual average test loss from our experimental runs. DynaGraph MAE/MSE shows the accuracy that DynaGraph can achieve with proposed optimizations.

of research, and several systems have the capability to process massive graphs, some even in the range of trillions of edges [4, 5, 19, 53]. In graph processing literature, there has been an increasing interest in processing dynamic, or *time-evolving* graphs, for the purpose of temporal, historic, or real-time analysis [11, 12, 18, 23]. These systems focus on enabling graph algorithms and graph mining, and do not support embedding generation as required in GNNs.

Graph Neural Network Systems: GNNs have become an active area of research in the recent past, both in the machine learning and the systems community. The machine learning community has proposed a number of GNN architectures, each with its own advantages for particular tasks [7, 16, 25, 47]. Recently, many efforts have been made to scale GNNs to large-scale graphs by leveraging accelerators such as GPUs [24]. Some of these proposals are single-machine systems, while others support distributed settings. PyTorch Geometric [8] and DGL [50] are two of the most popular frameworks support GNN training. ROC [21] and P^3 [9] proposed techniques to scale GNN training to real-world graphs.

Dynamic GNNs: Since real-world graphs are dynamic, there has been an increasing interest in supporting dynamic graphs in GNNs. Most of these works take a discrete view of a dynamic graph that is represented by a series of snapshots, and can be classified into three [44]. **Stacked dynamic GNNs** propose using a separate GNN for each snapshot of the graph and then feeding the output to a time-series component such as an RNN. Works in this space differ on the type of RNN used. GCRN-M1 [43] stacks GCN and LSTM, WD-GCN [33] uses a separate LSTM per node. DySAT [42] stacks a GAT and a Transformer. HDGNN [60] focuses on heterogeneous dynamic networks using spectral GCNs and a variety of RNNs. StrGNN [2] uses a stacked dynamic GNN for anomaly detection. **Integrated dynamic GNNs**, the focus of this paper, propose combining GNNs and RNNs in one layer, thus capturing the spatial and temporal modeling in the same layer. GCRN-M2 [43] integrates GCN in an LSTM (i.e., GCRN-LSTM model in this paper). GC-LSTM [3] is similar to GCRN-M2 but only performs a spectral graph convolution on hiddens. T-GCN [59] integrates GCN into GRU but only performs a graph convolution on inputs. Finally, **Dynamic graph autoencoders and generative models**, such as DynGEM [14], use a deep autoencoder to encode snapshots of discrete node-dynamic graphs.

Dynamic GNN Systems: While we are not aware of any systems that are specifically built for dynamic GNNs, existing GNN processing frameworks, such as PyTorch Geometric (PyG) and DGL have some support for dynamic GNNs. Pytorch Geometric Temporal [40] is a library built on PyG [8] for temporal graph neural networks. It provides data loaders and iterators for spatiotemporal datasets. DGL supports implementing dynamic GNNs as a combination of GNN and RNN operations explicitly by the developer. Unfortunately, considering GNN and RNN operations independently results in inefficiencies by ignoring opportunities for optimizations.

8 Conclusion

In this paper, we presented DynaGraph, a system that supports efficient dynamic GNN processing. DynaGraph is based on the observation that existing dynamic GNN architectures combine structural embedding techniques and temporal embedding techniques independently, and that cross-optimizations can lead to a reduction in overhead. Based on this, DynaGraph proposes cached message passing, a technique to reduce the amount of redundant computation and communication, and timestep fusion, a technique to combine time-independent components of the operation. To process large-scale dynamic graphs, it further uses a simple distributed data-parallel processing technique with a sequence-level partition scheme for the snapshots in dynamic GNNs. These techniques enable DynaGraph to achieve significant speedups compared to the state-of-the-art. In our evaluation, DynaGraph is able to outperform DGL and PyG Temporal, two popular GNN processing frameworks, by up to 2.7×.

References

[1] Han Altae-Tran, Bharath Ramsundar, Aneesh S. Pappu, and Vijay Pande. 2017. Low Data Drug Discovery with One-Shot Learning. *ACS Central Science* 3, 4 (2017), 283–293. https://doi.org/10.1021/acscentsci.6b00367 arXiv:https://doi.org/10.1021/acscentsci.6b00367 PMID: 28470045.

[2] Lei Cai, Zhengzhang Chen, Chen Luo, Jiaping Gui, Jingchao Ni, Ding Li, and Haifeng Chen. 2020. Structural Temporal Graph Neural Networks for Anomaly Detection in Dynamic Graphs. arXiv:2005.07427 [cs.LG]

[3] Jinyin Chen, Xueke Wang, and Xuanheng Xu. 2021. GC-LSTM: Graph Convolution Embedded LSTM for Dynamic Link Prediction. arXiv:1812.04206 [cs.SI]

[4] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 1, 15 pages. https://doi.org/10.1145/2741948.2741970

[5] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-scale. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[6] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv:1406.1078 http://arxiv.org/abs/1406.1078

[7] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc., 3844–3852. https://proceedings.neurips.cc/paper/2016/file/04df4d434d481c5bb723be1b6df1ee65-Paper.pdf

[8] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[9] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi

[10] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. *CoRR* abs/1704.01212 (2017). arXiv:1704.01212 http://arxiv.org/abs/1704.01212

[11] Joseph Gonzalez, Reynold Xin, Ankur Dave, Daniel Crankshaw, and Ion Franklin, Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 17–30.

[13] Palash Goyal, Sujit Rokka Chhetri, and Arquimedes Canedo. 2018. dyngraph2vec: Capturing Network Dynamics using Dynamic Graph Representation Learning. *CoRR* abs/1809.02657 (2018). arXiv:1809.02657 http://arxiv.org/abs/1809.02657

[14] Palash Goyal, Sujit Rokka Chhetri, Ninareh Mehrabi, Emilio Ferrara, and Arquimedes Canedo. 2018. DynamicGEM: A Library for Dynamic Graph Embedding Methods. *ArXiv* abs/1811.10734 (2018).

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., 1024–1034. https://proceedings.neurips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf

[16] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data Engineering Bulletin*, Article arXiv:1709.05584 (Sept. 2017), arXiv:1709.05584 pages. arXiv:1709.05584 [cs.SI]

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (11 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735 arXiv:https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf

[18] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 745–761. https://www.usenix.org/conference/osdi18/presentation/iyer

[19] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 337–355. https://www.usenix.org/conference/nsdi21/presentation/iyer

[20] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. 2014. Big Data and Its Technical Challenges. *Commun. ACM* 57, 7 (jul 2014), 86–94. https://doi.org/10.1145/2611567

[21] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 187–198. https://proceedings.mlsys.org/paper/2020/file/fe9fc289c3ff0af142b6d3bead98a923-Paper.pdf

[22] Amol Kapoor, Xue Ben, Luyang Liu, Bryan Perozzi, Matt Barnes, Martin Blais, and Shawn O'Banion. 2020. Examining COVID-19 Forecasting using Spatio-Temporal Graph Neural Networks. *CoRR* abs/2007.03113 (2020). arXiv:2007.03113 https://arxiv.org/abs/2007.03113

[23] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing* (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 239–252. https://doi.org/10.1145/2600212.2600227

[24] Kevin Kiningham, Christopher Re, and Philip Levis. 2020. GRIP: A Graph Neural Network Accelerator Architecture. *arXiv e-prints*, Article arXiv:2007.13828 (July 2020), arXiv:2007.13828 pages. arXiv:2007.13828 [cs.AR]

[25] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations* (Palais des Congrès Neptune, Toulon, France) *(ICLR '17)*. https://openreview.net/forum?id=SJU4ayYgl

[26] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 249–263. https://www.usenix.org/conference/fast19/presentation/kumar

[27] Junying Li, Deng Cai, and Xiaofei He. 2017. Learning Graph-Level Representation for Drug Discovery. *CoRR* abs/1709.03741 (2017). arXiv:1709.03741 http://arxiv.org/abs/1709.03741

[28] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. 2018. Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. In *International Conference on Learning Representations (ICLR '18)*.

[29] Jaechang Lim, Seongok Ryu, Kyubyong Park, Yo Joong Choe, Jiyeon Ham, and Woo Youn Kim. 2019. Predicting Drug–Target Interaction Using a Novel Graph Neural Network with 3D Structure-Embedded Graph Representation. *Journal of Chemical Information and Modeling* 59, 9 (2019), 3981–3988. https://doi.org/10.1021/acs.jcim.9b00387 arXiv:https://doi.org/10.1021/acs.jcim.9b00387 PMID: 31443612.

[30] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN Training on Large Graphs via Computation-Aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 401–415. https://doi.org/10.1145/3419111.3421281

[31] Yixin Liu, Shirui Pan, Yu Guang Wang, Fei Xiong, Liang Wang, and Vincent C. S. Lee. 2021. Anomaly Detection in Dynamic Graphs via Transformer. *CoRR* abs/2106.09876 (2021). arXiv:2106.09876 https://arxiv.org/abs/2106.09876

[32] Yu-Chen Lo, Stefano E. Rensi, Wen Torng, and Russ B. Altman. 2018. Machine learning in chemoinformatics and drug discovery. *Drug Discovery Today* 23, 8 (2018), 1538 – 1546. https://doi.org/10.1016/j.drudis.2018.05.010

[33] Franco Manessi, Alessandro Rozza, and Mario Manzo. 2020. Dynamic graph convolutional networks. *Pattern Recognition* 97 (Jan 2020), 107000. https://doi.org/10.1016/j.patcog.2019.107000

[34] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 25, 16 pages. https://doi.org/10.1145/3302424.3303974

[35] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 533–549. https://www.usenix.org/conference/osdi21/presentation/mohoney

[36] Aditya Pal, Chantat Eksombatchai, Yitong Zhou, Bo Zhao, Charles Rosenberg, and Jure Leskovec. 2020. PinnerSage: Multi-Modal User Embedding Framework for Recommendations at Pinterest. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 2311–2320. https://doi.org/10.1145/3394486.3403280

[37] George Panagopoulos, Giannis Nikolentzos, and Michalis Vazirgiannis. 2020. United We Stand: Transfer Graph Neural Networks for Pandemic Forecasting. *CoRR* abs/2009.08388 (2020). arXiv:2009.08388 https://arxiv.org/abs/2009.08388

[38] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2019. EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs. arXiv:1902.10191 [cs.LG]

[39] Namyong Park, Andrey Kan, Xin Luna Dong, Tong Zhao, and Christos Faloutsos. 2019. Estimating Node Importance in Knowledge Graphs Using Graph Neural Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 596–606. https://doi.org/10.1145/3292500.3330855

[40] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, , Guzman Lopez, Nicolas Collignon, and Rik Sarkar. 2021. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*. 4564–4573.

[41] Benedek Rozemberczki, Paul Scherer, Oliver Kiss, Rik Sarkar, and Tamas Ferenci. 2021. Chickenpox Cases in Hungary: a Benchmark Dataset for Spatiotemporal Signal Processing with Graph Neural Networks. *CoRR* abs/2102.08100 (2021). arXiv:2102.08100 https://arxiv.org/abs/2102.08100

[42] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2019. Dynamic Graph Representation Learning via Self-Attention Networks. arXiv:1812.09430 [cs.LG]

[43] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. 2016. Structured Sequence Modeling with Graph Convolutional Recurrent Networks. arXiv:1612.07659 [stat.ML]

[44] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. 2020. Foundations and modelling of dynamic networks using Dynamic Graph Neural Networks: A survey. *CoRR* abs/2005.07496 (2020). arXiv:2005.07496 https://arxiv.org/abs/2005.07496

[45] Jonathan M. Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M. Donghia, Craig R. MacNair, Shawn French, Lindsey A. Carfrae, Zohar Bloom-Ackermann, Victoria M. Tran, Anush Chiappino-Pepe, Ahmed H. Badran, Ian W. Andrews, Emma J. Chory, George M. Church, Eric D. Brown, Tommi S. Jaakkola, Regina Barzilay, and James J. Collins. 2020. A Deep Learning Approach to Antibiotic Discovery. *Cell* 180, 4 (2020), 688 – 702.e13. https://doi.org/10.1016/j.cell.2020.01.021

[46] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe

[47] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[48] Andrew Z. Wang, Rex Ying, Pan Li, Nikhil Rao, Karthik Subbian, and Jure Leskovec. 2021. Bipartite Dynamic Representations for Abuse Detection. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 3638–3648. https://doi.org/10.1145/3447548.3467141

[49] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: A Flexible and Efficient Distributed Framework for GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3447786.3456229

[50] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv e-prints*, Article arXiv:1909.01315 (Sept. 2019), arXiv:1909.01315 pages. arXiv:1909.01315 [cs.LG]

[51] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 515–531. https://www.usenix.org/conference/osdi21/presentation/wang-yuke

[52] Mark Weber, Giacomo Domeniconi, Jie Chen, Daniel Karl I. Weidele, Claudio Bellei, Tom Robinson, and Charles E. Leiserson. 2019. Anti-Money Laundering in Bitcoin: Experimenting with Graph Convolutional Networks for Financial Forensics. *CoRR* abs/1908.02591 (2019). arXiv:1908.02591 http://arxiv.org/abs/1908.02591

[53] Jingqi Wu, Rong Chen, and Yubin Xia. 2021. *Fast and Accurate Optimizer for Query Processing over Knowledge Graphs*. Association for Computing Machinery, New York, NY, USA, 503–517. https://doi.org/10.1145/3472883.3486991

[54] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks? *CoRR* abs/1810.00826 (2018). arXiv:1810.00826 http://arxiv.org/abs/1810.00826

[55] Sijie Yan, Yuanjun Xiong, and Dahua Lin. 2018. Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition. *CoRR* abs/1801.07455 (2018). arXiv:1801.07455 http://arxiv.org/abs/1801.07455

[56] Bing Yu, Haoteng Yin, and Zhanxing Zhu. 2017. Spatio-temporal Graph Convolutional Neural Network: A Deep Learning Framework for Traffic Forecasting. *CoRR* abs/1709.04875 (2017). arXiv:1709.04875 http://arxiv.org/abs/1709.04875

[57] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. 2018. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294* (2018).

[58] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *USENIX Annual Technical Conference*.

[59] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. 2020. T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction. *IEEE Transactions on Intelligent Transportation Systems* 21, 9 (Sep 2020), 3848–3858. https://doi.org/10.1109/tits.2019.2935152

[60] Fan Zhou, Xovee Xu, Ce Li, Goce Trajcevski, Ting Zhong, and Kunpeng Zhang. 2020. A Heterogeneous Dynamical Graph Neural Networks Approach to Quantify Scientific Impact. arXiv:2003.12042 [cs.SI]