Augmenting Recurrent Graph Neural Networks with a Cache

Guixiang Ma
guixiang.ma@intel.com
Intel Labs

Vy A. Vo
vy.vo@intel.com
Intel Labs

Theodore L. Willke
ted.willke@intel.com
Intel Labs

Nesreen K. Ahmed
nesreen.k.ahmed@intel.com
Intel Labs

ABSTRACT

While graph neural networks (GNNs) provide a powerful way to learn structured representations, it remains challenging to learn long-range dependencies in graphs. Recurrent GNNs only partly address this problem. In this paper, we propose a general approach for augmenting recurrent GNNs with a cache memory to improve their expressivity, especially for modeling long-range dependencies. Specifically, we first introduce a method of augmenting recurrent GNNs with a cache of previous hidden states. Then we further propose a general Cache-GNN framework by adding additional modules, including attention mechanism and positional/structural encoders, to improve the expressivity. We show that the Cache-GNNs outperforms other models on synthetic datasets as well as tasks on real-world datasets that require long-range information.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; *Bio-inspired approaches*; *Machine learning algorithms.*

KEYWORDS

graph neural networks, memory, attention

ACM Reference Format:
Guixiang Ma, Vy A. Vo, Theodore L. Willke, and Nesreen K. Ahmed. 2023. Augmenting Recurrent Graph Neural Networks with a Cache. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23), August 6–10, 2023, Long Beach, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3580305.3599260

1 INTRODUCTION

Graph neural networks (GNNs) provide a family of methods for modeling data with complex and irregular relationships. While they can handle highly regular inputs (e.g. 2D grids of image pixels or a 1D line of temporal steps), their power lies in modeling information over data with arbitrary relational structure [15, 50]. In the typical message passing paradigm, for a give node v, information from the set its neighbors $\mathcal{N}_v$ is propagated to update the representation of node v. This propagation process is repeated recursively for L iterations, where each iteration represents a GNN message passing layer [66]. Due to the locality bias in this paradigm and the issues with over-smoothing [35] and over-squashing [1], it is challenging for GNNs to learn about relationships between distant nodes [4] — which we will explain next in details. This can pose problems for many real-world datasets depending on the size and structure of the graph and the nature of the task [13]. This also makes it difficult to model any long-range temporal relationships.

To address these issues, several proposals suggest using some form of recurrent unit to enrich the intermediate features used in the propagation step [36, 37]. We refer to these as *recurrent GNNs*, since they combine the approach of GNNs with components of recurrent neural networks (RNNs). There are some clear benefits to these proposals. First, using recurrent units allow GNNs to output sequences. This is critical for many tasks that have time-varying inputs, such as video [46] and text [53, 65]. However, they are not limited to sequential outputs and can pool the representations to output node-level or graph-level representations [36]. Second, they can also be used to learn representations from dynamic graphs that change over time [25]. Third, enriched intermediate representations should improve the ability of GNNs to model long-range dependencies, and was shown to alleviate issues of over-squashing to some degree compared to GNNs [1].

The logic behind this third claim can be seen by examining the effective reach, or receptive field, of a given node after it has been processed by a GNN model. When aggregating for multiple iterations, the node can capture information from its distant hop neighborhood [60]. Ideally, stacking L GNN layers should allow the node to aggregate from its L-hop neighborhood. However, as L increases, the receptive field of a given node increases exponentially. This makes learning long-range dependencies in GNNs an even more challenging problem than in RNNs, where recursion causes a linear, rather than exponential, increase in the information that needs to be captured by some fixed-length vector [1]. This has been dubbed the *over-squashing* bottleneck. Previous work has shown that a recurrent GNN performs better than, for example, a convolutional approach, likely due to its increased representational capacity [1]. In particular, gated recurrent units (GRU) [8] that remember some internal representation over T time-steps are more capable of modeling long-range dependencies [4].

This suggests that memory mechanisms may be key to capturing long-range dependencies in GNNs. Humans rely on multiple memory systems for everyday learning and decision making [47, 54]. This inspired us to investigate methods to augment recurrent GNNs with another form of memory, especially one that would further enrich the intermediate representations of the network. Prior work on memory-augmented GNNs use the memory to store information from more distant nodes [59] farther back in time [39, 40, 49]. Memory mechanisms are helpful when dealing with dynamic graphs and sequential data, as has been shown in GNN applications ranging from algorithmic reasoning [51] to dialog systems [58].

We propose a general approach for augmenting recurrent GNNs with a cache memory to improve their expressivity, especially for modeling long-range dependencies. The paper is organized as follows. First, we describe a method of augmenting recurrent GNNs with a cache of previous hidden states. We then show that this

KDD '23, August 6–10, 2023, Long Beach, CA, USA

ACM ISBN 979-8-4007-0103-0/23/08.
https://doi.org/10.1145/3580305.3599260

memory augmentation specifically improves performance for far apart nodes. Second, we propose our general Cache-GNN model by adding additional modules to improve expressivity. We then show its performance on several tasks, including the Long-Range Graph Benchmark [13], and compare it to many baselines. Finally, we conduct ablation experiments to understand the contribution of each module.

Contributions. We summarize our main contributions thusly:

- We propose augmenting recurrent GNNs with a non-differentiable cache-memory to store previous hidden states, and observe that adding a cache-memory improves model performance on several long-range modeling tasks.
- We propose a general, flexible, modular framework, called Cache-GNN, that is effectively utilizing attention, recurrence, and memory mechanisms to capture long-range dependencies in graphs.
- We extensively evaluate the performance of the proposed framework on five different graph learning tasks, namely link prediction, node classification, graph classification, graph regression, and network reachability.

2 NEURAL CACHE RECURRENT GNN

2.1 How GNNs can benefit from memory

Different forms of memory can increase the representational capacity of GNNs. In some memory-augmented GNNs, an external store retains task-specific content, such as a knowledge graph used for question answering [45] or spatial position in an image [26]. These forms of memory are non-differentiable and are simply read and processed by the GNN, rather than modified in some way. They resemble retrieval-augmented methods in other areas of machine learning [21, 64]. These external memory mechanisms can be distinguished from memory storage of *internal* representations, such as the hidden state of a gated recurrent unit.

In RNNs, memory augmentation has been shown to improve their expressive capacity [43]. Some differentiable memory structures for RNNs, such as stacks and queues, have been specifically experimented with for their ability to learn hierarchical structure [19, 52]. The well-known differentiable memory controllers of the Neural Turing Machine and Differentiable Neural Computer, on the other hand, forego structure and directly learn how to read and write to memory [17, 18]. However, they can be unstable during training and have been shown to depend on the initialization of the contents of memory [9].

In addition, there have been other attempts to improve long-range modeling in GNNs by adding fully-connected nodes to an otherwise sparsely connected graph. A densely connected node circumvents the usual locality bias through graph rewiring [5, 14, 38], enabling each node to receive some global information about the graph. This approach has been framed as adding a *memory node* to the graph [59]. Indeed, Gilmer et al. [14] proposed to use a densely connected node that serves as a global scratch space where each node both reads from and writes to in every step of message passing, in order to allow information to travel long distances in the graph.

However, a densely connected node may not be an optimal form of memory for learning long-range relationships. For large graphs, there may be many long-range relationships that cannot be adequately represented in the feature dimension of the single, densely connected node. Indeed, there may be several subgraphs or substructures, each of which would be ideally represented by their own node selectively connected to their constituents [57]. Another approach is to operate over a fully-connected graph to learn the required edges. The recently popular graph Transformers learn these edges by all-to-all attention [11, 30, 48]. However, it is well-known that the attention mechanism scales poorly with an increasing number of nodes, with complexity on the order of $O(N^2)$ for N nodes. Previous work has even proposed that recurrent memory mechanisms can be viewed as a 'cousin' to Transformer self-attention that has reduced complexity at the expense of inducing a locality bias [32].

Here we study a simple, non-differentiable form of memory augmentation for recurrent GNNs. We call this form of memory a neural cache that stores recent hidden state representations (i.e., from previous timesteps $t \in T$), to effectively expanding the representational capacity of the node. Neural cache has been used in language modeling [16] and was shown to be extremely informative for prediction. By keeping the memory non-differentiable, we keep the number of learnable parameters to a minimum, and allows the model to scale effortlessly when increasing the number of memory cells. A cache also avoids making structural assumptions about the data, as is inherent with a stack or queue.

2.2 Cache memory for recurrent GNNs

Given a graph $G = (V, E)$, where V is the set of nodes, $|V|$ denotes the number of nodes in the graph, E is the set of edges, where $|E| \ll |V|^2$. We also use $\mathbf{A}$ to denote the graph adjacency matrix.

We now define the propagation rule for a single node in a generic recurrent GNN, computed over a fixed number of iterative timesteps $t \in T$. Propagation consists of an aggregation step over the node's neighbors (Eq. 1), followed by an update step for the current hidden state (Eq. 2).

$$\mathbf{a}_v^t = \mathbf{A}_v\{h_1^{t-1} \cdots \mathbf{h}_{|V|}^{t-1}\} + \mathbf{b} \tag{1}$$

$$\mathbf{h}_v^t = C_{RNN}(\mathbf{a}_v^t, \mathbf{h}_v^{t-1}) \tag{2}$$

Here $\mathbf{h}_v^t$ denotes the hidden representation for node v at timestep t. C_{RNN} can be defined by any set of recurrent neural network equations where the input is a_v^t. For example, prior GNN work [36, 37] has used the Gated Recurrent Unit (GRU) equations [7] or the Long Short-Term Memory (LSTM) unit equations [22].

Next, we redefine the update equation (Eq. 2) so that it incorporates the cached hidden representations. The cache S stores the most recent hidden representations from previous timesteps, $\{\mathbf{h}_v^j \cdots \mathbf{h}_v^{t-2}\}$ (Figure 1). If all previous hidden representations are stored, then $j = 1$. The size of the memory cache $|S|$ can be freely determined by the experimenter, but here we set this value to its theoretical maximum, which is $T - 2$. Now the current hidden state

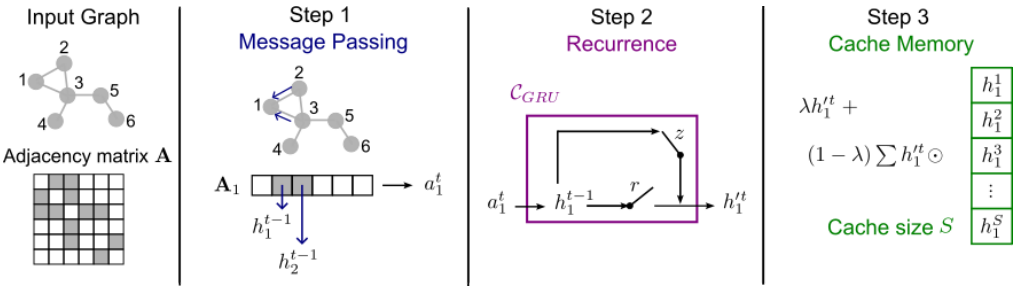

Figure 1: The cache mechanism for recurrent GNNs. On the far left, we show an example graph and its adjacency matrix. Then from left to right, we show the steps to calculate the hidden state of node 1. Although not illustrated here, every step is repeated for all nodes before moving to the next step. (1) One iteration of message passing occurs. (2) The result of message passing is passed as input to a recurrent unit, such as the Gated Recurrent Unit [7] shown here. The output is an intermediate hidden state h' without any information from the cache. (3) The intermediate h' is combined with the node's previous hidden states stored in the cache, using a learnable hyperparameter λ.

of the cache-augmented recurrent GNN is given by:

$$\mathbf{h}_v'^t = C_{RNN}(\mathbf{a}_v^t, \mathbf{h}_v^{t-1}) \tag{3}$$

$$\mathbf{h}_v^t = \lambda \mathbf{h}_v'^t + (1-\lambda) \sum_{i=1}^{|S|} (\mathbf{h}_v'^t \odot \mathbf{h}_v^i) \tag{4}$$

where the λ hyperparameter interpolates between the intermediate hidden state representation $\mathbf{h}_v'^t$ and its Hadamard product with the past hidden representations stored in the cache. The addition of the cache changes the space complexity of a recurrent GNN from $|V|d$, where d is the hidden dimension, to $|V|d|S|$. Since $|S|$ is a hyperparameter, this storage cost can be pre-allocated.

2.3 Evaluation and Results

Cache-memory experimental setup. To investigate the effects of the proposed cache memory, we evaluate two different models on three tasks that can reveal model performance across both short-range and long-range distances between nodes. (1) The NeighborsMatch task [1] is a synthetic benchmark task that requires the GNN to predict which other graph nodes have the same number of neighbors as the target node. For this task, models are trained for 2000 epochs, $1e-3$ learning rate and batch size 1024. (2) The Reachability Analysis task [41] is constructed from compiler graphs, and requires a binary prediction of whether a node is reachable from the root node of the graph. For the reachability task, models are trained for 500 epochs at a learning rate of $1e-3$ and a batch size of 64. (3)The PascalVOC-SP is an image graph dataset from the long-range graph learning benchmark in [13], where each image is represented as a graph, and each node corresponds to a region of the image with some semantic segmentation label. For this node classification task, we use the same data splits and experimental setup as [13] and train the models for 1000 epochs at a learning rate of $1e-4$ with a batch size of 32. Full details of the datasets are given in Section 4.2.

We evaluate two different choices of recurrent units – the GRU used in the Gated GNN (GG-NN) work [36], and the graph LSTM (G-LSTM) as proposed in [3]. We test these baseline recurrent GNNs before adding the cache memory. We also evaluate two other candidates that could attend to long-range information through convolution (gated convolutional network, GCN, [28]) or attention (graph attention network, GAT, [56]). Finally, we examine a version of the GCN that is augmented with a global memory node [59] (MemGCN) to compare it with our cache memory model. We trained all models in an end-to-end manner using Adam optimizer with the default PyTorch configuration. We repeated each experiment three times and report the average and standard deviation.

Recurrent GNNs improve with cache memory. We find that the addition of the cache to either the GG-NN or G-LSTM improves accuracy on the NeighborsMatch task, especially when the nodes are farther apart (Figure 4, solid blue lines). Both of the cache models outperform the baseline models (Table 1).

The selective improvement at long range distances is even more notable in the Reachability Analysis experiment, where the performance gap between the recurrent GNN (GG-NN) and the cache-memory version of the model (Cache+GG-NN) grows between 15 and 30 node hops (Figure 3). As before, the cache-memory models outperform all of the other baseline models (Figure 3 and Table 2). With this task, however, the performance gap is even evident at shorter ranges.

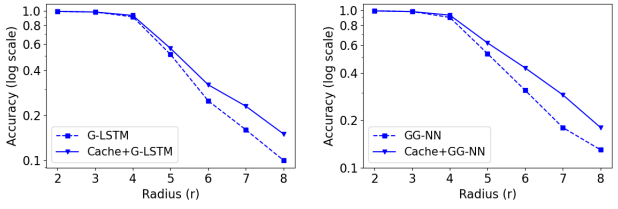

Figure 2: Performance of G-LSTM (left) and Gated GNN (right) vs. Cache-memory models on NeighborsMatch dataset.

Performance analysis over varying time steps or layers. To investigate the effects of the cache-memory on the recurrent GNNs with

Table 1: Accuracy of different GNN models vs. Cache-GNN for different problem radius (r) in NeighborsMatch dataset.

Model	$r = 2$	$r = 3$	$r = 4$	$r = 5$	$r = 6$	$r = 7$	$r = 8$
GCN	0.98 ± 0.01	0.96 ± 0.01	0.63 ± 0.02	0.13 ± 0.03	0.08 ± 0.03	0.06 ± 0.03	0.05 ± 0.03
MemGCN	0.98 ± 0.01	0.97 ± 0.01	0.71 ± 0.02	0.27 ± 0.03	0.16 ± 0.03	0.08 ± 0.03	0.07 ± 0.03
GAT	0.98 ± 0.01	0.97 ± 0.01	0.88 ± 0.02	0.34 ± 0.02	0.16 ± 0.02	0.12 ± 0.02	0.08 ± 0.03
G-LSTM	0.99 ± 0.01	0.98 ± 0.01	0.91 ± 0.01	0.51 ± 0.02	0.25 ± 0.02	0.16 ± 0.02	0.10 ± 0.03
GG-NN	0.99 ± 0.01	0.98 ± 0.01	0.90 ± 0.01	0.53 ± 0.02	0.31 ± 0.02	0.18 ± 0.03	0.13 ± 0.03
Cache+G-LSTM	0.99 ± 0.01	0.98 ± 0.01	0.93 ± 0.01	0.56 ± 0.02	0.32 ± 0.02	0.23 ± 0.02	0.15 ± 0.03
Cache+GG-NN	**0.99 ± 0.01**	**0.98 ± 0.01**	**0.93 ± 0.01**	**0.62 ± 0.02**	**0.43 ± 0.02**	**0.29 ± 0.02**	**0.18 ± 0.03**

Table 2: Performance of GNNs vs. Cache-memory recurrent GNNs on Reachability Analysis and PascalVOC-SP datasets

Model	Reachability **Precision ↑**	PascalVOC-SP **F1 score ↑**
GCN	0.826 ± 0.011	0.125 ± 0.011
MemGCN	0.845 ± 0.010	0.196 ± 0.018
GAT	0.833 ± 0.011	0.183 ± 0.034
G-LSTM	0.862 ± 0.010	0.284 ± 0.019
GG-NN	0.865 ± 0.009	0.285 ± 0.024
Cache+G-LSTM	0.881 ± 0.009	0.290 ± 0.019
Cache+GG-NN	**0.892 ± 0.009**	**0.294 ± 0.023**

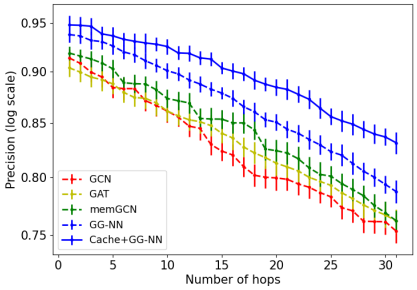

Figure 3: Classification precision for predicting reachable nodes at various numbers of hops.

different numbers of time steps (T) or layers (L), we evaluate the models with T ranging from 1 to 20, and number of layers ranging from 1 to 10, respectively. Figure 4 shows the results on the PascalVOC-SP dataset. As shown in the figure, as the number of time steps increases, the performance of all the models climbs first until reaching their own peaks and then starts to decline. This indicates that while storing and using the past hidden states in memory could help improve the learning capabilities of the recurrent GNNs, incorporating too much prior history information could hinder the performance improvement by the memory. Meanwhile, we observe that the optimal number of time steps and layers of the model both increased when adding the Cache-memory augmentation to GG-NN and G-LSTM. This implies that with the cache memory, both of the two recurrent GNN models are able to leverage useful information from a longer history for this node classification task.

According to the plots in the second row, the optimal numbers of layers for the two models have also been increased due to the cache-memory augmentation. This indicate that the cache-memory enables the models to reach a larger receptive field [1] of each node, which benefits the graph learning tasks that require longer-range information within the graph.

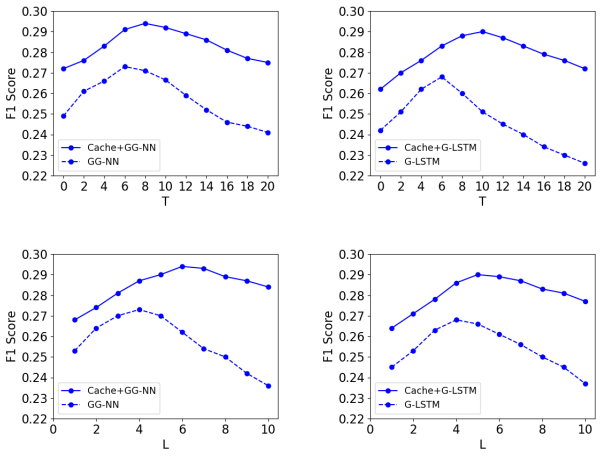

Figure 4: Performance of the Cache-memory recurrent GNNs vs. recurrent GNNs with different numbers of time steps (T) on the PascalVOC-SP.

3 PROPOSED FRAMEWORK

Above, we show that augmenting a recurrent GNN with a cache memory improves its long-range modeling performance on three graph learning tasks. To further improve upon this model, we consider two additional components that have been shown to improve the expressivity and long-range modeling capabilities of GNNs. In this section, we propose a general Cache-GNN framework that incorporates the cache-memory model with an attention mechanism and positional/structural encoders. This modular approach is illustrated in Figure 5. We then evaluate this general model against other baseline models on real-world data from the Long Range Graph Benchmark [13], including graph Transformers that also use attention mechanisms and positional/structural encoders.

Attention. Attention has been widely studied in neural network models as a way to focus on task-relevant representations

and ignore irrelevant or noisy representations. In particular, the attention mechanisms used in graph neural networks are often used to assign a relevance score to the nodes in the graph to highlight elements with task-relevant information. This can be particularly important in real-world datasets with noisy components [31]. The graph attention network (GAT) [56] uses attention to learn different weights on neighboring nodes – i.e. attention operates on the sparse adjacency matrix. Graph Transformers take this further by taking a weighted combination of all nodes – i.e. attention operates on a dense matrix where all nodes are connected to one another [12, 30, 48]. Here we consider adding a general attention mechanism to weight the information that is aggregated from adjacent nodes. The attention mechanism is a general strategy that allows for either sparse attention and dense attention, depending on the mask used on the adjacency matrix.

Positional and structural encodings. Positional encodings (PE) and structural encodings (SE) have been explored recently as strategies to help increase the expressivity and the generalizability of GNNs ([12, 48, 63]). Positional encodings provide information about the position of a given node in space within the graph, whereas the structural encodings provide information about the structure of graphs or subgraphs. Thus, if two nodes are close to each other within a graph, their PEs should also be close, and if two nodes share similar subgraphs, their SEs should be close. These encodings can be used to create bridges between distant nodes in graphs and Here we use the learnable encoders from [48] to generate Laplacian PE (LapPE) and random walk SE (RWSE) from the input graphs. We then concatenate the LapPE/RWSE with the original feature vector for each node and use it as the input node representation to the GNN module (Figure 5). This was implemented using publicly available code associated with [48].

Message passing for the general Cache-GNN model. As before, let $\mathbf{h}_v^t$ denote the node representation for node v at timestep t. Now $\tilde{\mathbf{A}}$ represents some adjacency matrix. Below we define the message passing equations for the best-performing version of our general model, which relies on GRUs as in [36]. That is, we use the cache-memory model with $C_{GRU}(\mathbf{a}_v^t, \mathbf{h}_v^{t-1})$, which is given by:

$$\mathbf{z}_v^t = \sigma(\mathbf{W}^z \mathbf{a}_v^t + \mathbf{U}^z \mathbf{h}_v^{t-1}) \tag{5}$$

$$\mathbf{r}_v^t = \sigma(\mathbf{W}^r \mathbf{a}_v^t + \mathbf{U}^r \mathbf{h}_v^{t-1}) \tag{6}$$

$$\tilde{\mathbf{h}_v^t} = tanh(\mathbf{W}\mathbf{a}_v^t + \mathbf{U}(\mathbf{r}_v^t \odot \mathbf{h}_v^{t-1})) \tag{7}$$

$$\mathbf{h'}^t_v = (1 - \mathbf{z}_v^t) \odot \mathbf{h}_v^{t-1} + \mathbf{z}_v^t \odot \tilde{\mathbf{h}_v^t} \tag{8}$$

$$\mathbf{h}_v^t = \lambda \mathbf{h'}^t_v + (1 - \lambda) \sum_{i=1}^{|S|} (\mathbf{h'}^t_v \odot \mathbf{h}_v^i) \tag{9}$$

Together, Equations 5 through 8 are simply the equations of the GRU, where Eq. 5 defines the update gate $\mathbf{z}$ and Eq. 6 defines the reset gate $\mathbf{r}$. The usage of the cache memory is the same as in Section 2 – that is, Eq. 9 is the same as Eq. 4.

To incorporate attention, we redefine the typical aggregation step given in Eq.(1) as Eq. (10):

$$\mathbf{a}_v^t = \tilde{\mathbf{A}}_v^T [\alpha_{v1} \mathbf{h}_1^{t-1} \cdots \alpha_{v|V|} \mathbf{h}_{|V|}^{t-1}]^T + \mathbf{b} \tag{10}$$

such that the hidden state of each node in $\tilde{\mathbf{A}}_\mathbf{v}$ is weighted by a learned attention weight α. To get each attention weights for nodes $u \in \tilde{N}(v)$, we compute softmax attention

$$\alpha_{vu} = \begin{cases} exp(e_{vu}) / \sum_{w \in \tilde{N}(v)} exp(e_{vw}) & \text{if } u \in \tilde{N}(v) \\ 0 & \text{if } u \notin \tilde{N}(v) \end{cases} \tag{11}$$

where $\tilde{N}(v)$ is some neighborhood of node v consistent with the adjacency matrix $\tilde{A}$. Then e_{vu} are coefficients that indicate the importance of node u's features to node v:

$$e_{vu} = f(\mathbf{W'}\mathbf{h}_v^{t-1}, \mathbf{W'}\mathbf{h}_u^{t-1}) \tag{12}$$

where $\mathbf{W'}$ is a learnable weight matrix and f is a single-layer feed-forward neural network, parameterized by a weight vector $\vec{\delta}$ followed by an application of the LeakyReLU nonlinearity. Then for $u \in \tilde{N}(v)$ (the top case in Eq. 11), the coefficients computed by the attention mechanism are:

$$\frac{exp(LeakyReLU(\vec{\delta}^\top [\mathbf{W'}\mathbf{h}_v^{t-1} \parallel \mathbf{W'}\mathbf{h}_u^{t-1}]))}{\sum_{w \in \tilde{N}(v)} exp(LeakyReLU(\vec{\delta}^\top [\mathbf{W'}\mathbf{h}_v^{t-1} \parallel \mathbf{W'}\mathbf{h}_w^{t-1}]))} \tag{13}$$

where $.^\top$ represents transposition and $\parallel$ is the concatenation operation.

To stabilize the learning process, we use multi-head attention in our model [56], where k is the number of attention heads. The aggregation from Eq. 10 is then reformulated as:

$$\mathbf{a}_v^t = \frac{1}{K} \sum_{k=1}^{K} \tilde{\mathbf{A}}_v^T [\alpha_{v1}^k \mathbf{h}_1^{t-1} \cdots \alpha_{v|V|}^k \mathbf{h}_{|V|}^{t-1}]^T + \mathbf{b} \tag{14}$$

Note that the model described above is a general attention model. When $\tilde{\mathbf{A}} = \mathbf{A}$, it is a sparse attention model which only attends to the neighbors of node v (i.e., $\tilde{N}(v) = N(v)$ is the set of neighboring nodes of node v, excluding v). Note that we do not need to include self-attention to node v because it is already treated separately in the recurrent equations. When $\tilde{\mathbf{A}}$ is an all-ones matrix (except for the diagonal), then the set of neighboring nodes is $\tilde{N}(v) = \{V\} - \{v\}$, creating a dense attention model. Under these definitions, attention is applied to all the other nodes in the graph during the aggregation. This general attention formulation allows for flexibility when applying this framework in different graph learning tasks. Together Equations 5-9 and 11-14 define each layer block in our general cache-GNN model (Figure 5).

Complexity analysis. Between the cache and the proposed attention components, the most expensive addition is the dense attention model, which takes the space complexity from scaling over the number of edges $O(|E|)$ to all pairwise connections between nodes, $O(|V|^2)$. A more in-depth complexity analysis is given in Appendix A.4.

4 EXPERIMENTS & RESULTS

4.1 Baselines & Experimental Setup

Graph Convolutional Networks. From the category of graph convolutional networks, we select GCN [28], CGNII [6], GINE [60], and GatedGCN [3] as baselines.

Recurrent GNNs. From recurrent GNNs, we select the Gated GNN (GG-NN) proposed in [36], the Graph LSTM model introduced in [3], and the Implicit GNN (IGNN) [20] as baselines.

Graph Transformers. From the Transformer class, we select the fully connected Transformer [55] and SAN [29] models.

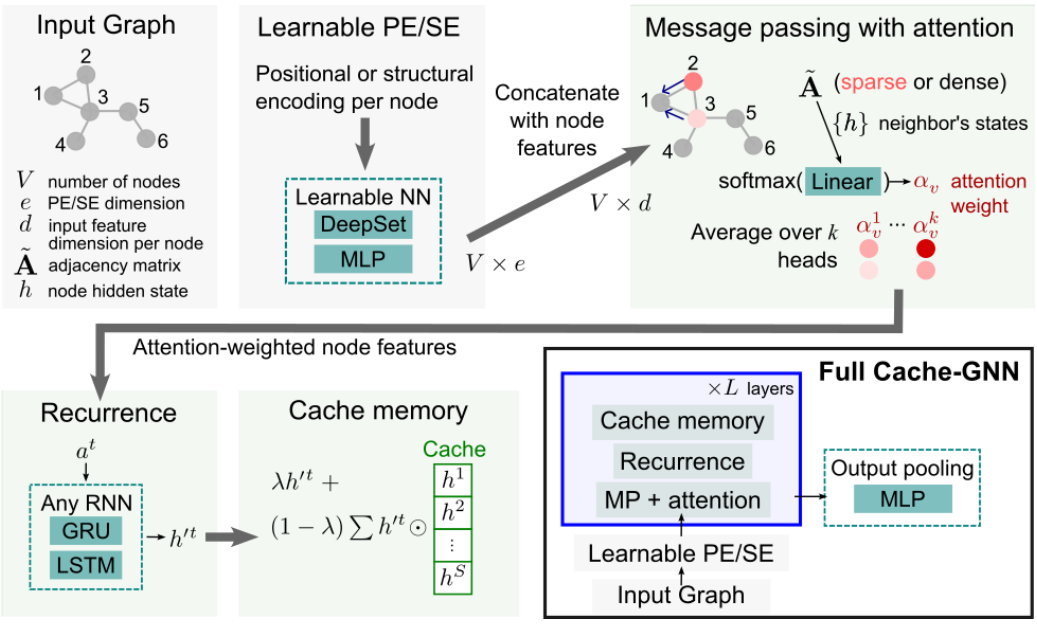

Figure 5: A schematic of the overall cache-GNN model. From the input graph, positional or structural encodings (PE/SE) can be computed for each node. These are passed through some learnable neural network, resulting in learned PE/SE features that are concatenated with the original node features from the input graph. This becomes the input to the message passing step, where the node features are weighted by a softmax attention mechanism. This attention mechanism can operate over the sparse graph, i.e. the neighborhood as defined in the input graph, or on a densely connected graph, as typical for graph transformer models. The outputs of the message passing step are given to the recurrent units to estimate intermediate hidden states for each node. Finally, this information is combined with the cache. These last two steps reuse the mechanisms from Figure 1. All of the modules in green form a single cache-GNN layer, which can be repeatedly stacked before being passed to some output model that transforms the hidden representations to task-specific predictions.

Experimental Setup. Similar to prior work on the long-range graph benchmark [13], we use 500k model parameter budget for all methods to maintain a fair comparison, and we follow the same settings and data splits as [13] for the experiments on the LRGB datasets. More details about the experimental setups can be found in the Appendix.

4.2 Datasets & Benchmarks

Long-Range Graph Benchmark. The Long-Range Graph Benchmark [13] was proposed to evaluate GNNs on their ability to model higher-order structural information, noting that previous benchmarks often relied only on local structure. The benchmark consists of 5 real-world datasets. The tasks may require the GNN to operate over large graphs with many nodes, to understand the global structure of the graph, or to compute relationships between distant nodes. Details on the tasks are given below in Section 4.3.

Reachability Analysis. The DeepDataFlow datasets [10] contains a set of graphs constructed from real-world LLVM-IR files for compiler analysis. We use 10,000 graphs from this dataset for the reachability analysis, which predicts a binary label indicating if a node is reachable from the root node of the graph. Due to a large class imbalance, we evaluate performance by computing the precision of the predicted labels on the reachable class only [10].

NeighborsMatch Dataset. [1] introduced a synthetic benchmark problem that requires long-range information. The goal is to predict which of the other graph nodes has the same number of neighbors as the target node. We use the same settings as [1] to generate 32,000 graphs per problem radius ranging from 2 to 8, and use 60% for training and 20% each for validation and test. We set the number of layers in the GNNs to $r + 1$, where r is the problem radius.

More details about the datasets, experimental setup and hyperparameter optimization can be found in the Appendix.

4.3 Machine Learning Tasks

Node Classification. PascalVOC-SP and COCO-SP are computer vision tasks from the Long-Range Graph Benchmark that require the GNN to predict the semantic segmentation label of parts of an image. The task on NeighborsMatch dataset is also a node classification task, which predicts which of the other nodes in the graph has the same number of neighbors as the target node.

Table 3 shows the results of our proposed Cache-GNN framework, the baseline GNNs, and transformer models in the node classification tasks on PascalVOC-SP and COCO-SP. As we can see from the table, our proposed Cache-GNN outperforms all the baseline models including the graph transformers. In particular, the best performance comes from our Cache-GNN model with LapPE and dense attention. It outperforms the baseline GNNs such as

the GCN by 173%, and the GG-NN by 22% on the PascalVOC-SP. The transformers such as SAN+LapPE also outperform the baseline GNNs, though it achieves lower F1 scores than Cache-GNN. Among the graph convnets, the best performing model is Gated GCN.

While we have demonstrated the superior performance of the cache-memory recurrent GNNs over baseline GNNs on the NeighborsMatch task in Section 2, we now evaluate our full Cache-GNN framework on the NeighborsMatch, and compare the performance with the graph transformer model: SAN+LapPE. Table 4 shows the testing accuracy of the models over different problem radii on NeighborsMatch. Our Cache-GNN+LapPE with dense attention achieves the best performance consistently across the problem radii, especially when $r > 5$. Notably, our Cache-GNN with sparse attention also outperforms the transformer model when $r > 5$. This indicates that even with sparse attention, our Cache-GNN can achieve better performance in predicting the labels for the distant nodes than the transformer model that uses dense attention.

These results demonstrate the superior capabilities of our proposed Cache-GNN framework in learning long-range relationships for the node classification tasks.

Table 3: Node classification results of the GNN models vs. transformer models for two benchmark datasets (PascalVOC-SP and COCO-SP), as measured by the macro weighted F1 Score. Red: best performing method. Blue: second best performing method. Magenta: third best performing method.

Model	PascalVOC-SP F1 score ↑	COCO-SP F1 score ↑
GCN	0.1268 ± 0.0060	0.0841 ± 0.0010
GCNII	0.1698 ± 0.0080	0.1404 ± 0.0011
GINE	0.1265 ± 0.0076	0.1339 ± 0.0044
IGNN	0.1725 ± 0.0124	0.1923 ± 0.0036
GatedGCN	0.2873 ± 0.0219	0.2641 ± 0.0045
GatedGCN+LapPE	0.2860 ± 0.0085	0.2574 ± 0.0034
GG-NN	0.2855 ± 0.0236	0.2612 ± 0.0042
G-LSTM	0.2843 ± 0.0196	0.2556 ± 0.0041
Cache-GNN+LapPE (Sparse)	0.3315 ± 0.0104	0.2724 ± 0.0035
Cache-GNN+LapPE (Dense)	**0.3462 ± 0.0085**	**0.2793 ± 0.0033**
Transformer+LapPE	0.2694 ± 0.0098	0.2618 ± 0.0031
SAN+LapPE	0.3230 ± 0.0039	0.2592 ± 0.0158
SAN+RWSE	0.3216 ± 0.0027	0.2434 ± 0.0156

Table 4: Testing accuracy of the Cache-GNN vs. graph transformer model over different problem radii on the NeighborsMatch dataset (mean ± std).

Model	$r = 5$	$r = 6$	$r = 7$	$r = 8$
Cache-GNN+LapPE(sparse)	0.66 ± 0.01	0.48 ± 0.01	0.34 ± 0.02	0.21 ± 0.03
Cache-GNN+LapPE(dense)	0.68 ± 0.01	0.50 ± 0.01	0.37 ± 0.02	0.23 ± 0.02
SAN+LapPE	0.64 ± 0.01	0.46 ± 0.01	0.31 ± 0.02	0.20 ± 0.02

Graph Classification & Regression. The two graph-level tasks from the LRGB use the same chemistry dataset of peptides, which are large molecules that are ideal for testing long-range dependencies. While their amino acid chains are shorter than proteins, their size allows them to fit in a GPU mini-batch. The graphs are constructed as 1D chains without any higher-dimensional information in the graph structure. The graph classification task, Peptides-func, requires the GNN to predict one of 10 class labels of the peptide. The graph regression task, Peptides-struct, predicts the aggregated 3D properties of the peptide, such as length or sphericity.

As reported in Table 5, our proposed Cache-GNN outperforms all the baseline GNN and transformer models on these two tasks, where the Cache-GNN with dense attention is the best performing method, and the Cache-GNN with sparse attention is the second best performing method. The third best performance comes from the transformers. Graph convnets perform worse than Cache-GNN and transformers, and the best performing methods among them are GatedGCN+RWSE and GCN for the two tasks respectively.

Table 5: Graph classification results on Peptides-func, as measured by by average precision (AP) (left), and graph regression results on Peptides-struct, as measured by Mean Absolute Error (MAE). Message passing GNN models are given on top, and graph transformer performance from [13] on bottom. Red: best performing method. Blue: second best performing method. Magenta: third best performing method.

Model	Peptides-func AP ↑	Peptides-struct MAE ↓
GCN	0.5930 ± 0.0023	0.3496 ± 0.0013
GCNII	0.5543 ± 0.0078	0.3471 ± 0.0010
GINE	0.5498 ± 0.0079	0.3547 ± 0.0045
GatedGCN	0.5864 ± 0.0077	0.3420 ± 0.0013
GatedGCN+RWSE	0.6069 ± 0.0035	0.3357 ± 0.0006
GG-NN	0.5900 ± 0.0063	0.3398 ± 0.0014
G-LSTM	0.5836 ± 0.0072	0.3370 ±0.0012
Cache-GNN+LapPE (Sparse)	0.6483 ± 0.0080	0.2436 ± 0.0014
Cache-GNN+LapPE (Dense)	**0.6671 ± 0.0056**	**0.2358 ± 0.0013**
Transformer+LapPE	0.6326 ± 0.0126	0.2529 ± 0.0016
SAN+LapPE	0.6384 ± 0.0121	0.2683 ± 0.0043
SAN+RWSE	0.6439 ± 0.0075	0.2545 ± 0.0012

Link Prediction. This task is performed on quantum chemistry graphs from the PCQM4M dataset [23]. The model must predict which pairs of distant nodes (>5 hop distance) that will contact each other in 3D space.

Table 6 shows the results of the models on this task. We observe that our Cache-GNN with RWSE & dense attention is the best performing method among all models, and our Cache-GNN+LapPE & dense attention is the second best. Among the baseline GNNs, the recurrent RNNs are the best, and they even outperform some of the graph transformer models. For instance, the GG-NN achieves better results than the Transformer+LapPE on three of the evaluation metrics. On the other hand, the best performing Graph Convenets is GatedGCN, though it performs worse than the best recurrent GNNs. The best performing model in transformers is the SAN with LapPE/RWSE. These results demonstrate the superior performance of our Cache-GNN in the link prediction task.

Network Reachability. This task is performed on the DeepDataFlow datasets [10] that contains a set of graphs constructed from real-world LLVM-IR files for compiler analysis. The task is to predict a binary label indicating if a node is reachable from the root node of the graph.

Table 6: Link prediction results on PCQM-Contact dataset for GNN models vs. graph transformers — Performance metrics: Hits@1, Hits@3, Hits@10, and Mean Reciprocal Rank (MRR). Red: best performing method. Blue: second best performing method. Magenta: third best performing method.

Model	Hits@1 ↑	Hits@3 ↑	Hits@10 ↑	MRR ↑
GCN	0.1321±0.0007	0.3791±0.0004	0.8256±0.0006	0.3234±0.0006
GCNII	0.1325±0.0009	0.3607±0.0003	0.8116±0.0009	0.3161±0.0004
GINE	0.1337 ± 0.0013	0.3642 ± 0.0043	0.8147 ± 0.0062	0.3180 ± 0.0027
GatedGCN	0.1279 ± 0.0018	0.3783 ± 0.0004	0.8433 ± 0.0011	0.3218 ± 0.0011
GatedGCN+RWSE	0.1288 ± 0.0013	0.3808 ± 0.0006	0.8517 ± 0.0039	0.3174 ± 0.0020
GG-NN	0.1345 ± 0.0011	0.3794 ± 0.0007	0.8448 ± 0.0013	0.3256 ± 0.0012
G-LSTM	0.1328 ± 0.0015	0.3675 ± 0.0010	0.8389 ± 0.0016	0.3166 ± 0.0012
Cache-GNN+RWSE (Sparse)	0.1394 ± 0.0012	0.3926 ± 0.0010	0.8610 ± 0.0012	0.3395 ± 0.0009
Cache-GNN+LapPE (Sparse)	0.1369 ± 0.0012	0.3895 ± 0.0011	0.8588 ± 0.0013	0.3358 ± 0.0010
Cache-GNN+RWSE (Dense)	0.1463 ± 0.0011	0.4102 ± 0.0008	0.8693 ± 0.0008	0.3488 ± 0.0008
Cache-GNN+LapPE (Dense)	0.1449 ± 0.0011	0.4094 ± 0.0008	0.8684 ± 0.0008	0.3467 ± 0.0007
Transformer+LapPE	0.1221 ± 0.0011	0.3679 ± 0.0033	0.8517 ± 0.0039	0.3174 ± 0.0020
SAN+LapPE	0.1355 ± 0.0017	0.4004 ± 0.0021	0.8478 ± 0.0044	0.3350 ± 0.0003
SAN+RWSE	0.1312 ± 0.0016	0.4030 ± 0.0008	0.8550 ± 0.0024	0.3341 ± 0.0006

Table 7 shows the overall performance of our Cache-GNN with sparse/dense attention vs. the transformer model SAN+LapPE. We can see that both the Cache-GNN models outperform the transformer in the overall precision for this task. We further compare the Cache-GNN with dense attention vs. the transformer model for predicting reachable nodes at various numbers of hops. As shown in Figure 6, compared to the SAN model, the Cache-GNN consistently achieves higher precision in predicting nodes with different hops, especially for the node hops greater than 20, indicating the advantage of Cache-GNN over transformer for capturing long-range relationships.

Table 7: Node classification precision for reachability task, averaged across hops.

Model	Precision
Cache-GNN+LapPE(sparse)	0.919 ± 0.008
Cache-GNN+LapPE(dense)	0.932 ± 0.007
SAN+LapPE	0.917 ± 0.007

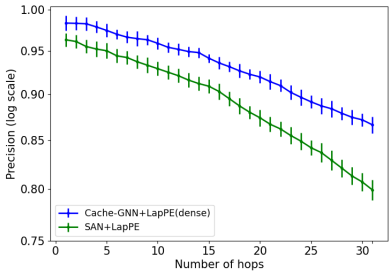

Figure 6: Classification precision of Cache-GNN vs. graph transformer on DeepDataFlow dataset for predicting reachable nodes at various numbers of hops.

4.4 Ablation Study

Recurrence: graph GRU vs. graph LSTM.

A key component in the Cache-GNN framework is the recurrence module. To investigate the effects of using different recurrent units for this module on the performance of Cache-GNN, we compare the full model with either graph GRU or graph LSTM as the recurrence component on 4 tasks from the Long-Range Graph Benchmark, which includes node classification, graph classification and graph regression. Table 8 shows the results of the two models. We can see that the Cache-GNN+LapPE(Sparse) with graph GRU outperforms the Cache-GNN+LapPE(Sparse) with graph LSTM on all the four datasets, indicating that having graph GRU in Cache-GNN is more powerful for these long-range graph learning tasks.

Table 8: The impact of the recurrence module (graph GRU/graph LSTM) on the performance of Cache-GNN+LapPE (Sparse) on PascalVOC-SP, COCO-SP, Peptides-func, and Peptides-struct. Bold indicates the best performing method.

Recurrence	PascalVOC-SP F1 score ↑	COCO-SP F1 score ↑	Peptides-func AP ↑	Peptides-struct MAE ↓
Graph-GRU	**0.3315 ± 0.0104**	**0.2724 ± 0.0035**	**0.6483 ± 0.0080**	**0.2436 ± 0.0014**
Graph-LSTM	0.3241 ± 0.0094	0.2615 ± 0.0038	0.6316 ± 0.0075	0.2510 ± 0.0012

Positional vs. Structural Encoders.

To investigate the contribution of positional and structural encoders to Cache-GNN performance on long-range graph problems, we conduct an ablation study and compare the full model with either LapPE or RWSE to a model without any encoders. All models compared here use a sparse attention matrix. Table 9 shows the results of the models on PascalVOC-SP, COCO-SP, Peptides-func, and Peptides-struct for the corresponding learning tasks. As we can see, having LapPE or RWSE in the sparse Cache-GNN help improve the performance on all the four datasets. Cache-GNN+LapPE outperforms Cache-GNN+RWSE on most of the datasets, suggesting that the

LapPE is more helpful for enhancing Cache-GNN's learning capabilities for the node classification, graph classification/regression tasks on these datasets.

From Table 6, we observe that the Cache-GNN+RWSE outperforms Cache-GNN+LapPE on the PCQM-Contact dataset across all the evaluation metrics, indicating that the structural information captured by the random walk in RWSE is more helpful for this link prediction task than the positional information by LapPE. Since our proposed Cache-GNN is a general framework, it allows for using various forms of positional or structural encodings. In practical applications, it would be beneficial to select specific forms of encodings that are the most relevant to the tasks.

Table 9: The impact of using an encoder (LapPE/RWSE) on the performance of Cache-GNN (Sparse) on PascalVOC-SP, COCO-SP, Peptides-func, and Peptides-struct. Bold is used to indicate the best performing method.

Encoder	PascalVOC-SP F1 score ↑	COCO-SP F1 score ↑	Peptides-func AP ↑	Peptides-struct MAE ↓
No encoder	0.3128 ± 0.0132	0.2673 ± 0.0038	0.6234 ± 0.0089	0.2865 ± 0.0021
LapPE	**0.3315 ± 0.0104**	0.2724 ± 0.0035	**0.6483 ± 0.0080**	**0.2436 ± 0.0014**
RWSE	0.3153 ± 0.0112	**0.2834 ± 0.0034**	0.6302 ± 0.0076	0.2739 ± 0.0015

Table 10: The impact of attention mechanism on the performance of Cache-GNN+LapPE on PascalVOC-SP, COCO-SP, Peptides-func, and Peptides-struct. We use *dense* to denote a fully connected graph, *sparse* to denote the observed graph, and *no attention* to denote using the observed graph (sparse) without learning attention weights. Bold is used to indicate the best performing method.

Attention	PascalVOC-SP F1 score ↑	COCO-SP F1 score ↑	Peptides-func AP ↑	Peptides-struct MAE ↓
No attention	0.3162 ± 0.0113	0.2683 ± 0.0032	0.6395 ± 0.0088	0.2786 ± 0.0018
Dense	**0.3462 ± 0.0085**	**0.2793 ± 0.0033**	**0.6671 ± 0.0056**	**0.2358 ± 0.0013**
Sparse	0.3315 ± 0.0104	0.2724 ± 0.0035	0.6483 ± 0.0080	0.2436 ± 0.0014

Sparse vs. Dense Attention.

We now investigate the influence of sparse or dense attention on the performance of the Cache-GNN. We evaluate the model with no attention, sparse attention, or dense attention on the same tasks as the PE/SE study above. We chose to use the LapPE as the encoder. As shown in Table 10, both the sparse and dense attentions help improve the performance of Cache-GNN in all tasks. In particular, the Cache-GNN+LapPE model with dense attention achieve the best results on all the datasets. This indicates that incorporating the weighted information of the neighborhood as well as the other distant nodes in the graph into the aggregation helps the Cache-GNN to capture more relevant information for the long-range learning tasks. As we discussed above, even with only sparse attention, the Cache-GNN can perform better than or similarly to the graph transformer models on the Long Range Graph Benchmark tasks. This implies the promising advantages of Cache-GNNs over graph transformers in practical applications, especially when graphs are very large and sparse, where it is hard for the graph transformers to scale due to the all-to-all attention mechanisms they use.

5 RELATED WORK

Earlier work on graph learning proposed recursive message passing that continued until convergence, and required a different learning algorithm which placed constraints on the neural network parameters [50]. This was extended to use more modern RNN practices and learning algorithms by iterating for a fixed number of timesteps and relying on backpropagation through time [36]. We adopt the latter approach in this work.

Our work is related to other efforts to augment neural networks with different types of memory mechanisms. Some work has added other types of memory modules to RNNs, such as a stack memory (i.e. a pushdown automaton) [24, 62] or a cache of previous hidden states [16, 42]. Our proposal is related to the continuous neural cache [16] in that we store an untransformed history of the most recent hidden states in a memory – however, we do not store an additional copy of the input. Other work augments neural networks with differentiable memory mechanisms [17, 18]. While these differentiable memory mechanisms have not been adapted to work with GNNs, others have proposed a variety of memory mechanisms for different GNN tasks (see [40] for review). This includes anything from a static knowledge base [45] to some form of key-value memory [27, 39]. Finally, a recent work adds a cache to GNNs that process dynamic graphs to accelerate the updating of representations, rather than to enrich the hidden representations and improve performance on long-range tasks [33].

Our general Cache-GNN model uses an attention mechanism that can operate over either a sparse or dense neighborhood (see [31] for review of attention in GNNs). Attention to a sparse neighborhood has been utilized in other GNN proposals [56]. Other papers have proposed some graph version of a transformer which involves the densification of the input graph [11, 30, 44, 48, 61]. Like this work, these graph transformers aim to enhance graph learning, especially for long-range dependencies. This graph densification can be viewed as a type of graph rewiring approach to improve message passing [57].

6 CONCLUSION AND FUTURE DIRECTIONS

In this paper, we proposed augmenting recurrent GNNs with a cache-memory to store previous hidden states, and we introduced a general Cache-GNN framework that utilizes attention, recurrence and memory mechanisms to capture long-range dependencies in graphs. Evaluations on both synthetic and real-world datasets demonstrated the effectiveness of the proposed approach and its superior performance on long-range graph learning tasks compared to other GNNs and graph transformer models. However, the tasks that we show here focus on using memory for capturing information across the graph, rather than across time. Future work should investigate the use of Cache-GNNs to model dynamic graphs. Some recent work [34] started to explore a cache-based system for optimizing GNNs, however, they mainly focus on optimizing the speed of GNNs instead of their performance for dynamic graphs. Our current proposal in Cache-GNN stores all recent timesteps, which may be suboptimal when modeling long temporal sequences. Some mechanism to control what is stored in memory may be required.

REFERENCES

[1] Uri Alon and Eran Yahav. 2020. On the bottleneck of graph neural networks and its practical implications. *arXiv preprint arXiv:2006.05205* (2020).

[2] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems* 31 (2018).

[3] Xavier Bresson and Thomas Laurent. 2017. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553* (2017).

[4] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. 2021. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. arXiv:2104.13478 (May 2021). http://arxiv.org/abs/2104.13478

[5] Rickard Brüel-Gabrielsson, Mikhail Yurochkin, and Justin Solomon. 2022. Rewiring with Positional Encodings for Graph Neural Networks. arXiv:2201.12674 (Oct 2022). http://arxiv.org/abs/2201.12674

[6] Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. 2020. Simple and deep graph convolutional networks. In *International conference on machine learning*. PMLR, 1725–1735.

[7] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[8] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[9] Mark Collier and Joeran Beel. 2018. Implementing Neural Turing Machines. In *Artificial Neural Networks and Machine Learning – ICANN 2018*, Věra Kůrková, Yannis Manolopoulos, Barbara Hammer, Lazaros Iliadis, and Ilias Maglogiannis (Eds.). Springer International Publishing, 94–104.

[10] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. 2021. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In *International Conference on Machine Learning*. PMLR, 2244–2253.

[11] Vijay Prakash Dwivedi and Xavier Bresson. 2021. A Generalization of Transformer Networks to Graphs. In *AAAI 2021 Workshop on Deep Learning on Graphs: Methods and Applications*. arXiv. https://doi.org/10.48550/arXiv.2012.09699

[12] Vijay Prakash Dwivedi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2021. Graph neural networks with learnable structural and positional representations. *arXiv preprint arXiv:2110.07875* (2021).

[13] Vijay Prakash Dwivedi, Ladislav Rampášek, Mikhail Galkin, Ali Parviz, Guy Wolf, Anh Tuan Luu, and Dominique Beaini. 2022. Long Range Graph Benchmark. *arXiv:2206.08164* (Jun 2022). http://arxiv.org/abs/2206.08164

[14] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning*. PMLR, 1263–1272. https://proceedings.mlr.press/v70/gilmer17a.html

[15] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, Vol. 2. IEEE, 729–734.

[16] Edouard Grave, Armand Joulin, and Nicolas Usunier. 2017. Improving Neural Language Models with a Continuous Cache. (2017).

[17] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing Machines. *arXiv:1410.5401* (Oct 2014). http://arxiv.org/abs/1410.5401

[18] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (Oct 2016), 471–476. https://doi.org/10.1038/nature20101

[19] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to Transduce with Unbounded Memory. *arXiv:1506.02516* (Nov 2015). http://arxiv.org/abs/1506.02516

[20] Fangda Gu, Heng Chang, Wenwu Zhu, Somayeh Sojoudi, and Laurent El Ghaoui. 2020. Implicit graph neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 11984–11995.

[21] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. REALM: Retrieval-Augmented Language Model Pre-Training. In *Proceedings of the 37 th International Conference on Machine Learning*. Vienna, Austria, 10.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[23] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. In *Proceedings of the 35th Conference on Neural Information Processing Systems Track on Datasets and Benchmarks*. arXiv. http://arxiv.org/abs/2103.09430

[24] Armand Joulin and Tomas Mikolov. 2015. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. *arXiv:1503.01007* (Jun 2015). http://arxiv.org/abs/1503.01007

[25] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation Learning for Dynamic Graphs: A Survey. arXiv:1905.11485 (Apr 2020). https://doi.org/10.48550/arXiv.1905.11485

[26] Mahmoud Khademi. 2020. Multimodal Neural Graph Memory Networks for Visual Question Answering. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 7177–7188. https://doi.org/10.18653/v1/2020.acl-main.643

[27] Amir Hosein Khasahmadi, Kaveh Hassani, Parsa Moradi, Leo Lee, and Quaid Morris. 2020. Memory-based graph networks. (2020), 16. arXiv:2002.09518

[28] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[29] Devin Kreuzer, Dominique Beaini, Will Hamilton, Vincent Létourneau, and Prudencio Tossou. 2021. Rethinking graph transformers with spectral attention. *Advances in Neural Information Processing Systems* 34 (2021), 21618–21629.

[30] Devin Kreuzer, Dominique Beaini, Will Hamilton, Vincent Létourneau, and Prudencio Tossou. 2021. Rethinking Graph Transformers with Spectral Attention. In *Advances in Neural Information Processing Systems*, Vol. 34. Curran Associates, Inc., 21618–21629. https://proceedings.neurips.cc/paper/2021/hash/b4fd1d2cb085390fbbadae65e07876a7-Abstract.html

[31] John Boaz Lee, Ryan A Rossi, Sungchul Kim, Nesreen K Ahmed, and Eunyee Koh. 2019. Attention models in graphs: A survey. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 13, 6 (2019), 1–25.

[32] Omer Levy, Kenton Lee, Nicholas FitzGerald, and Luke Zettlemoyer. 2018. Long Short-Term Memory as a Dynamically Computed Element-wise Weighted Sum. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Melbourne, Australia, 732–739. https://doi.org/10.18653/v1/P18-2116

[33] Haoyang Li and Lei Chen. 2021. Cache-based GNN System for Dynamic Graphs. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. ACM, Virtual Event Queensland Australia, 937–946. https://doi.org/10.1145/3459637.3482237

[34] Haoyang Li and Lei Chen. 2021. Cache-based gnn system for dynamic graphs. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 937–946.

[35] Qimai Li, Zhichao Han, and Xiao-Ming Wu. 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI conference on artificial intelligence*.

[36] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[37] Xiaodan Liang, Xiaohui Shen, Jiashi Feng, Liang Lin, and Shuicheng Yan. 2016. Semantic Object Parsing with Graph LSTM. In *European Conference on Computer Vision (ECCV)*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, 125–143. https://doi.org/10.1007/978-3-319-46448-0_8

[38] Xin Liu, Jiayang Cheng, Yangqiu Song, and Xin Jiang. 2022. Boosting Graph Structure Learning with Dummy Nodes. arXiv:2206.08561 (Jun 2022). http://arxiv.org/abs/2206.08561

[39] Chen Ma, Liheng Ma, Yingxue Zhang, Jianing Sun, Xue Liu, and Mark Coates. 2020. Memory Augmented Graph Neural Networks for Sequential Recommendation. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 04 (April 2020), 5045–5052. https://doi.org/10.1609/aaai.v34i04.5945

[40] Guixiang Ma, Vy Vo, Theodore Willke, and Nesreen K. Ahmed. 2022. Memory-Augmented Graph Neural Networks: A Neuroscience Perspective. arXiv:2209.10818 (Sep 2022). https://doi.org/10.48550/arXiv.2209.10818

[41] Guixiang Ma, Yao Xiao, Mihai Capotă, Theodore L. Willke, Shahin Nazarian, Paul Bogdan, and Nesreen K. Ahmed. 2021. Learning Code Representations Using Multifractal-based Graph Networks. In *2021 IEEE International Conference on Big Data (Big Data)*. 1858–1866. https://doi.org/10.1109/BigData52589.2021.9671685

[42] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. *arXiv:1609.07843* (Sep 2016). http://arxiv.org/abs/1609.07843

[43] William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A. Smith, and Eran Yahav. 2020. A Formal Hierarchy of RNN Architectures, Vol. Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.acl-main.43

[44] Grégoire Mialon, Dexiong Chen, Margot Selosse, and Julien Mairal. 2021. GraphiT: Encoding Graph Structure in Transformers. arXiv:2106.05667 (Jun 2021). http://arxiv.org/abs/2106.05667

[45] Seungwhan Moon, Pararth Shah, Anuj Kumar, and Rajen Subba. 2019. Memory Graph Networks for Explainable Memory-grounded Question Answering. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*. Association for Computational Linguistics, Hong Kong, China, 728–736. https://doi.org/10.18653/v1/K19-1068

[46] Andrei Nicolicioiu, Iulia Duta, and Marius Leordeanu. 2019. Recurrent Space-time Graph Neural Networks. In *33rd Conference on Neural Information Processing Systems*.

[47] Dale Purves (Ed.). 2004. *Neuroscience, 3rd ed.* Sinauer Associates. xix, 773 pages.

[48] Ladislav Rampášek, Mikhail Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. 2022. Recipe for a General, Powerful, Scalable Graph Transformer. arXiv:2205.12454 (Jul 2022). http://arxiv.org/abs/2205.12454
[49] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. *arXiv:2006.10637* (Oct. 2020).
[50] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.
[51] Heiko Strathmann, Mohammadamin Barekatain, Charles Blundell, and Petar Veličković. 2021. Persistent Message Passing. *arXiv:2103.01043* (Mar 2021). http://arxiv.org/abs/2103.01043
[52] Mirac Suzgun, Sebastian Gehrmann, Yonatan Belinkov, and Stuart M. Shieber. 2019. Memory-Augmented Recurrent Neural Networks Can Learn Generalized Dyck Languages. *arXiv:1911.03329* (Nov 2019). http://arxiv.org/abs/1911.03329
[53] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics. http://arxiv.org/abs/1503.00075
[54] Endel Tulving. 1985. How many memory systems are there? *American Psychologist* 40, 4 (1985), 385–398.
[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[56] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).
[57] Petar Veličković. 2022. Message passing all the way up. In *ICLR 2022 Workshop on Geometrical and Topological Representation Learning*.
[58] Jie Wu, Ian G Harris, and Hongzhi Zhao. 2022. GraphMemDialog: Optimizing End-to-End Task-Oriented Dialog Systems Using Graph Memory Networks. *Proceedings of the AAAI Conference on Artificial Intelligence* 36, 10 (Jun 2022), 11504–11512. https://doi.org/10.1609/aaai.v36i10.21403
[59] Tao Xiong, Liang Zhu, Ruofan Wu, and Yuan Qi. 2020. Memory Augmented Design of Graph Neural Networks. (Sept. 2020). https://openreview.net/forum?id=K6YbHUIWHOy
[60] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).
[61] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. 2021. Do Transformers Really Perform Badly for Graph Representation?. In *Advances in Neural Information Processing Systems*, Vol. 34. Curran Associates, Inc., 28877–28888. https://proceedings.neurips.cc/paper/2021/hash/f1c1592588411002af340cbaedd6fc33-Abstract.html
[62] Dani Yogatama, Yishu Miao, Gabor Melis, Wang Ling, Adhiguna Kuncoro, Chris Dyer, and Phil Blunsom. 2018. Memory architectures in recurrent neural network language models. (2018).
[63] Jiaxuan You, Rex Ying, and Jure Leskovec. 2019. Position-aware Graph Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning*. PMLR, 7134–7143. https://proceedings.mlr.press/v97/you19b.html
[64] Hamed Zamani, Fernando Diaz, Mostafa Dehghani, Donald Metzler, and Michael Bendersky. 2022. Retrieval-Enhanced Machine Learning. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2875–2886. https://doi.org/10.1145/3477495.3531722
[65] Yue Zhang, Qi Liu, and Linfeng Song. 2018. Sentence-State LSTM for Text Representation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Melbourne, Australia, 317–327. https://doi.org/10.18653/v1/P18-1030
[66] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2018. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434* (2018).

A APPENDIX

A.1 Detailed Descriptions of Datasets

NeighborsMatch. [1] introduce this synthetic benchmark problem that requires long-range information. For some example graph in the dataset, each node has an alphabetical label (e.g. A, B, C...) and a number of neighbors n. Given some target node, the task is to predict which of the other graph nodes has the same number of neighbors n and return the corresponding alphabetical label (e.g. node C). Every example in the dataset has a different mapping between the label and number of neighbors, so solving this problem requires message propagation and matching for every graph in the dataset. To control the receptive field size required to solve the problem, each target node is the root of a binary tree of some depth r, a.k.a. the problem radius. We use the same settings as [1] to generate 32,000 graphs, and use 60% for training and 20% each for validation and test.

Reachability analysis. The DeepDataFlow datasets [10] contains a set of graphs constructed from a corpus of real-world LLVM-IR files from various sources (e.g. C, C++, OpenCL, etc.) for compiler analysis. We use 10,000 graphs from this dataset for the control reachability analysis, which predicts if a node (i.e., instruction) is reachable from the root node (i.e., starting code instruction) of the graph. This set of graphs have an average of 1370 nodes and 2390 edges, and each node comes with inst2vec [2] features and a binary label indicating if it is reachable from the root. We evaluate performance on this task by computing the precision of the predicted labels on the reachable class only, as in [10, 41]. This is necessary because of the large class imbalance between reachable and unreachable nodes. Models are trained for 500 epochs at a learning rate of 0.001, and we use a batch size of 64.

A.2 Hyperparameter Optimization

We tuned parameters such as the embedding size and λ on the validation set. For the NeighborsMatch dataset, we follow the settings in [1] and set the number of layers in the GNNs to $r + 1$, where r is the problem radius. For the Reachability Analysis datasets, we treat the number of graph layers as a parameter and vary it from 1 to 10 on the validation set to obtain the optimal number. We also tune the embedding size of each GNN model on the validation set by doing grid search from {16, 32, 64, 128}. For the LRGB datasets, we follow the same settings and data splits as [13]. For the recurrent GNNs and our proposed cache-memory recurrent GNN models, we vary the number of time steps from 1 to 20 on the validation set to obtain the optimal number. Both the MemGCN and our Cache-memory models have a hyperparameter λ which controls the weight they put on the component related to external memory. We tuned λ for both models on each validation set by doing grid search from {0.1, 0.2, ...,0.9}.

A.3 Parameter Sensitivity Analysis

We provided parameter sensitivity analysis in Figure 4 for the number of timesteps T and number of layers L, and we studied the impact of our proposed cache GNN model with two state-of-the-art recurrent GNN models (Gated GNN with GRU, and Graph LSTM). Below, we have also added a sensitivity analysis for the parameter λ that controls the contribution from the cache memory to the model.

Performance of Cache-memory recurrent GNNs with different values for on PascalVOC-SP dataset. The performance metric is the average F1 score of multiple runs.

A.4 Complexity analysis

Below we give a more complete analysis of the space and time complexity of different components of the CacheGNN framework (Table 11). Let $|V|$ be the number of nodes, $|E|$ the number of edges, T the number of timesteps, and d the hidden state dimension.

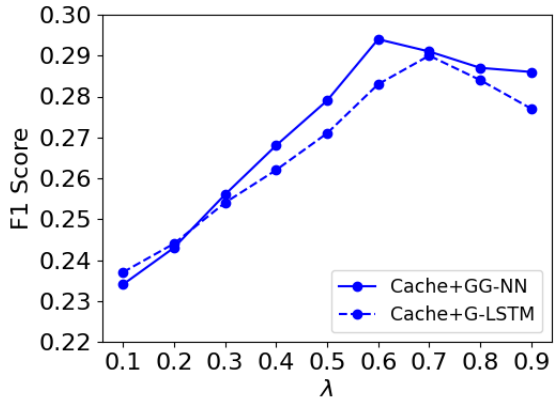

Figure 7: Performance of Cache-memory recurrent GNNs with different values for λ on PascalVOC-SP dataset.

Table 11: Complexity of different CacheGNN components compared to the base recurrent GNN.

Component	Space Complexity	Time complexity
Recurrent GG-NN	$O(\|E\| + \|V\|d + d^2)$	$O(\|V\|d^2 + \|E\|d)T$
Cache	$O(\|E\| + \|V\|dT + d^2)$	$O(\|V\|d^2 + \|E\|d + \|V\|d)T$
Sparse attention	$O(\|E\| + \|V\|d + d^2)$	$O(\|V\|d^2 + \|E\|d)T$
Dense attention	$O(\|V\|^2d + \|V\|dT + d^2)$	$O(\|V\|d^2 + \|V\|^2d)T$

The base model complexity depends on both the complexity of the message passing steps and the complexity of the recurrent cell. The message passing steps require storing the edges, $|E|$, as well as a hidden state for every node, $|V|d$. The space complexity of the recurrent cell scales with the size of the weight matrices, adding d^2 for a final storage complexity for the GG-NN of $O(|E| + |V|d + d^2)$. The time complexity for message passing is dependent on the number of edges and the hidden dimension, $|E|d$. Matrix multiplication with the weights occurs for every node, yielding $|V|d^2$. Finally, the basic (message passing, recurrent cell) pattern is repeated for T timesteps, yielding a final time complexity of $O(|V|d^2 + |E|d)T$. Recall that we experimented with $T \leq 20$ and found that larger values do not provide any additional performance when learning on static graphs. Therefore an asymptotic analysis of complexity will never be dominated by T. We can thus assume that $|E| > |V| > d >> T$ for all the tasks and settings in this work.

The cache stores a hidden state for every single node and timestep, resulting in an extra storage complexity that scales linearly with these, $|V|dT$ instead of $|V|d$. The additional computation is an elementwise product with the states in the cache, so the cache simply adds $|V|dT$ to the time complexity. Assuming $|V| > d$ and knowing that $|V| >> T$ and $d >> T$, the asymptotic analysis shows that the cache adds computational complexity linear to the number of nodes.

Compare this to the changes in complexity associated with adding an attention mechanism, a commonly used method for trying to learn long-range dependencies in graphs. The sparse attention mechanism adds an attention weight matrix multiplication for every node to store an attention coefficient for each edge. However, the additional weight matrix and scalar coefficient per edge can be folded into the existing $d^2 + |E|$ terms, so there is no change in space complexity from the recurrent GNN. As noted in the Graph Attention Network paper, this complexity is on par with graph convolutional methods [56]. Yet the additional complexity of a dense attention mechanism is even worse – the terms that rely on the number of edges $|E|$ need to be replaced by $|V|^2$ to account for the all-to-all node connections.

To summarize, an asymptotic analysis shows that the cache only scales linearly with the number of nodes, while dense attention mechanisms scale quadratically.