Fully-Decentralized Training of GNNs using Layer-wise Self-Supervision

Lodovico Giaretta[0000−0002−0223−8907] and Šarūnas Girdzijauskas[0000−0003−4516−7317]

KTH Royal Institute of Technology, Stockholm, Sweden
{lodovico,sarunasg}@kth.se

Abstract. In existing literature, GNN training has been performed mostly in centralized, and sometimes federated, settings. In this work, we consider a fully-decentralized data-private scenario, where each node has limited knowledge of the surrounding graph. We propose the first architecture that enables GNN training in this fully-decentralized setting, by carefully combining several techniques, including decoupled learning, self-supervision and Gossip Learning. We implement two simulation tools to experimentally evaluate our solution. The results show that the proposed technique can be effectively used in scenarios where centralized or federated approaches are unfeasible or undesirable.

Keywords: Graph Neural Networks · Decentralized Learning · Self-Supervised Learning · Gossip Learning · Decoupled Learning.

1 Introduction

The rise of IoT, mobile devices and connected appliances has caused a fast increase in the data available for training deep learning models to improve existing digital services or provide innovative ones. However, the traditional centralized approach of data collection, storage and processing is no longer suitable to this task. The volume and velocity of data produced by these sources causes scalability issues, especially in terms of data transfer, while the broad scope and high granularity of these heterogeneous data streams raises questions about data privacy, data security and trust. Thus, there is an increasing need for truly decentralized ML training solutions, where the data is stored privately on the original devices and the training is performed over a scalable, peer-to-peer network.

Recent years have also seen the rise of graph-structured data, where the data points are no longer independent inputs, but instead connected to each other by explicit links, representing domain-specific relations which the ML models must exploit to provide accurate predictions. This led to a growing interest in the field of Graph Representation Learning (GRL)[4], and in particular Graph Neural Networks (GNNs)[21]. However, the vast majority of the existing studies have focused on a traditional, centralized training setting.

A few works have investigated the training of GNNs on edge devices using Federated Learning[15]. However, Federated Learning requires a central trusted

aggregator to combine the contributions of different devices, and thus cannot fully eliminate the issues of scalability, trust and privacy. Furthermore, these techniques assume that each device has a sufficiently large view of the graph to perform a complete GNN training step, which is especially problematic for deep GNNs that have a large receptive field.

By contrast, we consider the fully-decentralized training scenario that arises in some edge and IoT settings, such as traffic prediction on smart city sensors at road intersections, or predicting illness spreading based on mobile phone proximity data. In these scenarios, each edge device represents an individual node in the graph to be learned, and is not allowed to have any knowledge of the graph other than the identity and features of its direct neighbours. Each interaction is only known to the two nodes that performed it, and the feature of a node are only known to those that have interacted directly with it. Thus, this scenario guarantees that each device in only responsible for the smallest possible local view and minimizes the dissemination of the raw graph data. Furthermore, the training must happen in a peer-to-peer fashion, without any central aggregator, and without requiring a device to be aware of all other devices in the network, thus ensuring scalability and fault tolerance even in dynamic environments.

Unfortunately, this fully-decentralized scenario introduces new, significant challenges, that cannot be addressed by simply taking existing federated solutions and modifying the centralized parts. Specifically, a *naive* GNN training implementation in this scenario requires a prohibitive amount of communication to implement forward and backward message passing over the broad receptive field of deep GNNs. Furthermore, it is plagued by synchronization issues across the devices. To the best of our knowledge, no previous research has attempted to tackle this type of fully-decentralized GNN training scenario, and thus no solutions exist in the literature to overcome these issues.

Contribution In this work, we introduce DecGNN, the first fully-decentralized GNN training architecture capable of removing the communication and synchronization burdens that arise in the target scenario. We achieve this by carefully combining techniques from several different research directions in a novel way.

We provide two different tools to simulate our decentralized architecture, providing different levels of fidelity and enabling the analysis of different aspects of the system. With the help of these tools, we evaluate DecGNN, which presents a small gap to its centralized counterpart in downstream classification tasks, thus showing that our approach is promising and can be effectively employed when a centralized or federated solution is undesirable.

We also extensively discuss several key open questions and research directions that might close the gap between our architecture and centralized solutions and allow its deployment in real-world environments.

2 Related Work

2.1 Decoupled Greedy Learning

Our work is closely related to *decoupled greedy learning*[9,2,3], a family of techniques used to efficiently train large-scale models. It identifies *forward* and *backward locking* constraints[9] as the main factors slowing down distributed deep model training. The former represents the fact that the forward step of a layer can only begin once all the up-stream layers have completed their own forward layers. The latter instead occurs when the backward step of a layer needs to wait for the backward step of all down-stream layers to complete.

The literature on decoupled greedy learning focuses on traditional, datacenter-scale training, where model parallelism is often used, and where these constraints result in some of the hardware accelerators, or even entire servers, idly waiting for the inputs needed to run the layers assigned to them. Thus, these constraints effectively limit the level of parallelism that can be achieved in the training of deep models. In our case, the issue is not limited to idling and synchronization, but also includes the large number of messages that need to be exchanged between the different nodes at each layer.

The backward locking issue can be solved by introducing a new local source of gradients for each layer. Previous work has employed auxiliary trainable networks, which either predict the expected gradient[9], or predict the down-stream task labels to generate backward gradients[2,3]. In our solution, we instead use a self-supervised loss function that is independent of the down-stream task and does not introduce additional trainable weights. For the forward locking issue, [2] suggests the use of replay buffers between layers, to hold and reuse previous activations, the same approach that we employ in this work.

2.2 Federated Training of GNNs

Federated Learning[15,22] is the most popular approach for collaborative training on massively-decentralized data. In it, each data-owning device performs forward and backward model calculations on its local data to produce a gradient. A central aggregator then collects all these gradients, merges them into the model, and broadcasts the updated model to all participants. In this context, several works have explored the use of Federated Learning for training GNNs[12,20,8]. However, two key differences exist between these studies and our own.

First, these studies assume that each device holds either one or more complete graphs, or a large enough subgraph to allow the local computation of the forward and backward passes of a multi-layer GNN. Thus, network communication is only needed to aggregate the locally-computed gradients via the central aggregator. In our scenario, each individual device represents a single node in the logical graph and has a very limited view of its surroundings, insufficient to perform the training without additional peer-to-peer collaboration during the forward and backward passes.

Second, while Federated Learning, when combined with effective privacy preservation techniques, can enable a certain degree of data privacy, it still relies on a central aggregator and thus shares some of the issues of a traditional centralized training. One of these is trust, as the aggregator may use its privileged position to attempt to leak data from the received gradients, or simply to bias the aggregation and thus training procedure. Another issue is scalability, as the aggregator must scale its networking and memory to support all participants and represents a single point of failure that can completely stop the training process if taken down. We therefore employ Gossip Learning[17,7], a fully-decentralized alternative to Federated Learning that relies on robust and scalable peer-to-peer communications instead of a central, trusted aggregator.

3 DecGNN Architecture

3.1 Scenario

Two different graphs exist in DecGNN. The first is the *communication graph*, which forms the infrastructure over which the decentralized training is performed. This graph is composed of a large set of physical devices, which are able to communicate freely with each other, for example via the Internet or a local network. As such, this first graph is effectively fully-connected.

The second is the domain-specific *logical graph* on which the self-supervised GNN is trained. The nodes of this graph match one-to-one with the physical devices. However, this graph is sparse, containing only a subset of all possible edges, according to a domain-specific definition of what it means for two nodes to be connected. While any pair of devices can communicate with each other to exchange information during training, the GNN message passing only happens on top of the logical graph.

As an example, consider a mobile fitness app. Each device is a smartphone, matching one-to-one with a node, representing a user. Each device stores the features of its user, such as age and gender, and a list of friends together with whom the user has exercised. A GNN could be trained on this graph to predict future exercising behaviour. Alternatively, wireless communications could be used by the smartphones to automatically detect proximity and interactions between individuals during a pandemic, and the information could be used to track the spreading of the illness.

As mentioned, one of the advantages of DecGNN is that it does not require global knowledge of either the logical or communication graph. More specifically, on the logical graph, each node only stores its one-hop neighbourhood, i.e. only the edges incident to itself and the features of its direct neighbours. Furthermore, each node is only aware of a random subset of the communication graph.

3.2 Decoupled Learning via Layer-wise Self-Supervision

In the given scenario, decentralizing the training of a GNN is challenging, as shown in fig. 1. The embedding of a node at layer i depends on those of each

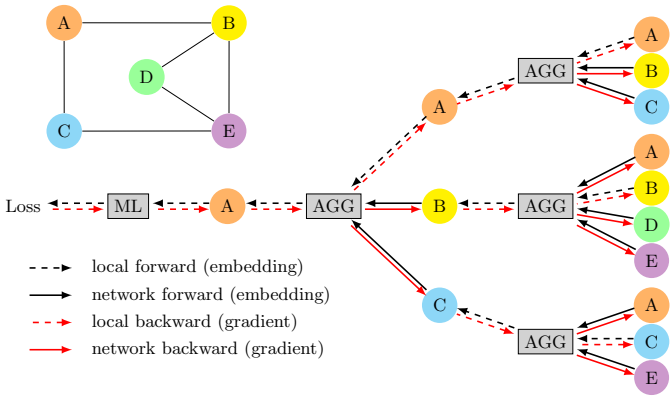

Fig. 1. Computational tree of the forward and backward passes for computing a node embedding in a 2-layer GNN. Note the large number of embeddings and gradients that would be exchanged across the network in a naive decentralized implementation.

neighbour at layer $i-1$, and so the forward step of layer i must wait for the completion of the forward step of layer $i-1$ at all neighbours. The backward pass is even more challenging: the gradients from layer $i+1$ of all neighbours must be waited for and accumulated to update the layer-i weights, and to then compute separate gradients to send to each individual neighbour for layer $i-1$. These *forward* and *backward locking* introduce significant synchronization and network traffic, rendering a naive decentralized training procedure extremely inefficient.

To counter these issues, we decouple the learning at each layer and at each node, removing all dependencies and synchronization points. The key technique we employ to achieve this is layer-wise self-supervision, with the architecture shown in fig. 2. First, we insert gradient stops between all layers of the GNN. Thus, each layer acts as an independent model, treating the embeddings produced by the previous layer as static, non-trainable feature vectors. In this way, no gradients flow between different nodes, and no communication or synchronization is necessary during the backward pass. Unfortunately, this approach completely removes the external training signal provided by a down-stream loss function. To compensate for this, each layer is equipped with its own auxiliary loss function.

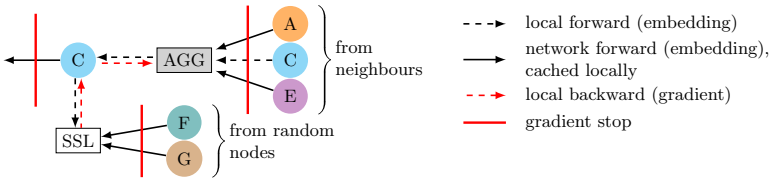

Fig. 2. Structure of an individual DecGNN layer within a node

We opt for a self-supervised auxiliary loss[18,10], rather than a supervised one, as the latter could potentially discard patterns that cannot be directly used to perform the supervised task, but that could become useful in subsequent layers, after additional non-linear transformations. Self-supervised loss functions are instead designed to ensure that the produced embeddings capture all relevant phenomena to reconstruct the original data distribution[10] By using a self-supervised loss in each layer, in theory, the final representations produced by DecGNN should contain all signals useful to any down-stream task.

Within an individual node, each layer only produces a single output vector: the representation of the local device for that layer. This is not sufficient information to produce gradients through a self-supervised loss. *Negative samples* are necessary. These may be used explicitly, as is the case with *contrastive* loss functions[10], or implicitly by requiring the inputs to be fed in uniformly-distributed batched, as is the case with *non-contrastive* losses[18] In this work we employ a non-contrastive loss, and thus each node, in order to produce gradients to update the weights layer i, must feed its i-th self supervised loss function with a batch containing a uniform random sample of the node embeddings produced by layer i across the whole graph.

We thus employ a push-based random sampling approach, where at the end of each local training iteration each node will push its own embeddings for each layer, that it just produced, to a number of randomly-chosen devices. The receiving devices will then be able to use those embeddings, along with those received from others, to build a batch for each layer in their next local training iteration.

3.3 Asynchronicity, Staleness and Network Traffic

One key feature of this decoupled learning approach is its full asynchronicity. All inputs necessary to perform a training step on layer i – the embeddings of the neighbours at layer $i-1$ and the layer i embeddings from random nodes – are stored locally, to ensure that each node can act even when not receiving all updates in a timely fashion.

For neighbours, the most recent embedding received from their forward steps is cached, thus allowing each node to perform its own forward pass at any point without having to wait for fresh information, at the expense of using partially stale information. Random node embeddings are stored in a buffer of length K with a first-in, first-out (FIFO) policy, so that newly received vectors replace the oldest and most stale ones. This allows the entire buffer to be used to maximize the size of the batch fed to the loss function, without requiring K vectors to be received at each step, at the cost of some stale vectors being reused in multiple subsequent gradient computations.

The ability of the model to train even when using partially stale information can also be exploited to reduce network communication requirements. More specifically, if nodes do not require the receipt of K random vector to operate, then it is not necessary for them to send their embeddings to K random peers at each iteration. Instead, they may reduce the network pressure by only sending them to $K' < K$ peers, with the choice of K' trading off between embedding

staleness and network traffic. Similarly, nodes with many neighbours in the logical graph may not need to broadcast their embeddings to all of them at each iteration. Instead, they may use a round-robin policy to update a subset of them each time, again achieving a staleness-traffic tradeoff.

3.4 Gossip Learning

The decoupled learning architecture presented above solves the GNN-specific issue of long-range forward and backward dependencies across nodes. However, decentralized training of ML models more in general presents another challenge: how can all devices converge to the same model parameters without the need for a central aggregator or coordinator?

To tackle this challenge, we employ Gossip Learning[17], a very robust[7] and scalable protocol for decentralized training, based on peer-to-peer exchanges of the model weights. Each node maintains a FIFO buffer with the latest two models received from peers. In each iteration, it merges the two models in its buffer by averaging their weights, performs one step of local training of the resulting aggregated model, and then forwards it to a randomly-chosen peer. From a different perspective, each model can be seen as performing a random walk over the fully-connected communication graph. At each step, it learns not only the information on that individual device, but also all the information that was known to the previous visitor of the same device. Thus, Gossip Learning allows very fast and efficient information propagation and thus results in quick convergence.

3.5 Peer Sampling

For DecGNN to work, each device must be able to select peers uniformly at random from the whole network. This is needed for both sharing model weights in Gossip Learning and spreading embedding vectors to be used in the self-supervised loss. However, it is unrealistic to expect each device to maintain a list of all other participants. This would not scale to large deployments, and maintaining accurate information would be very challenging in a dynamic environment, where devices may join and leave at any moment.

Instead, we employ a *peer sampling* protocol[11] Also known as *decentralized membership management* protocols, these are peer-to-peer protocols that allow each device to build a small, uniformly-distributed random sample of the overall network. These samples are frequently exchanged between peers, typically piggy-backed to existing communications, giving each device the chance to know about and contact any other peer. Furthermore, the samples are typically timestamped so that stale entries representing nodes that left the network can be quickly identified and removed.

In DecGNN we employ a simple push-based peer sampling protocol, with the sharing of samples piggy-backed to the existing Gossip Learning messages.

3.6 Memory Complexity

As discussed in section 3.3, the frequent use of buffering enables fully-asynchronous training and can reduce network pressure. Unfortunately, it can also leads to significant memory pressure. In DecGNN, each device v needs to store:

- The contact information (e.g. IP address and network port) of all its neighbours and of the random network sample from the peer sampling protocol. This only has a negligible contribution on the overall memory footprint, due to the small size of each entry.
- Three copies of the model weights: two in the FIFO buffer, received from peers, and the combined one used for training. This is not specific to our architecture, but rather a common requirement in all instances of Gossip Learning. With an L-layer GNN and D-dimensional feature and embedding spaces, this results in roughly $3 \cdot D^2 \cdot L$ weights.
- Output embeddings of each GNN layer from K random peers, and input embeddings (or features in the case of layer 1) for each GNN layer from all N_v neighbours of node v. With the same assumptions as before, this results in $(N_v + K) \cdot L$ embedding vectors, or $(N_v + K) \cdot L \cdot D$ scalars.

Overall, this leads to a memory usage of $O\left((N_v + K + D) \cdot D \cdot L\right)$ scalars. One one hand, this asymptotic complexity does not appear problematic. N_v and K are not expected to grow significantly with network size and even advanced GNNs are relatively shallow models, with L typically chosen from $\{2, 3\}$, so scalability to large networks and complex GNNs is not an issue. On the other hand, even small graphs may present relatively large values of N_v and K, and the formula does not account for significant constant factors, such as the need to store three copies of the model. Thus, it might be challenging to deploy DecGNN on small, extremely memory-constrained IoT devices.

3.7 Communication Complexity

From a network perspective, each node needs to send $K' + N' + 1$ messages after each iteration. $K' + N'$ messages contain the output embeddings of the node from each of the L layers, and are addressed to K' random peers and N' neighbours. The size of each of these messages is $D \cdot L$ scalars. One additional message is used to gossip the model weights and to share a portion of the local random peer sample. With the latter having negligible footprint, as discussed in section 3.6, the size of this message is roughly $D^2 \cdot L$.

Fortunately, as mentioned in section 3.3, the need for DecGNN to robustly handle stale information during training provides a simple way to reduce the network pressure of the training, by controlling the number $K' <= K$ of random peers and the number $N' <= N_v$ of neighbours that will receive the embeddings of node v after every iteration.

4 Implementation

To efficiently evaluate DecGNN, we implemented two different test programs, which we make available as open source software[1]. One is a complete simulator of a decentralized environment, reproducing all components of the proposed architecture. The other represents a centralized approximation, which only captures certain aspects of the decentralized system, but which enables fast study iterations and experimentation with certain parameters.

4.1 Full Simulator

The complete simulator is a carefully-optimized multi-threaded C++ program targeting Linux-based operating systems and using LibTorch, the C++ frontend of PyTorch, to enable efficient deep learning. Based on experiments run on AMD Zen2 CPUs at a frequency of 2.25 GHz, it is capable of simulating over 100 devices per hardware core, when each device is set to perform one training step per second. Thus, scalability to large numbers of simulated devices can be achieved by increasing the number of cores used and/or decreasing the training frequency of each device to fit more in a single core.

The memory, computation and network communication of each device are simulated individually, providing a near-perfect replica of a real-world deployment. The only limitation is the lack of IP stack simulation: all messages are reliably delivered to recipient within the span of one iteration. This limitation is equivalent to stating that the target real-world environment presents reliable network communications and that the time between training iterations is much larger than the typical network communication time. The second statement is typically true, as smartphones and other power-constrained IoT devices are not expected to perform continuous, high-frequency training. The first statement holds if TCP communications are used, but not if UDP is employed. The use of TCP communications may be reasonable in this scenario, especially considering that the messages between devices may be larger than the what a single UDP packet can carry.

4.2 Approximate Emulator

The emulator consists of a simple Python script which employs the PyTorch library and supports GPU acceleration. To minimize the memory footprint and to enable effective acceleration, which requires the use of few, large matrix operations rather than many smaller ones, the emulator forgoes several of aspects of the DecGNN architecture. A single copy of the model is stored and updated at each iteration, thus removing the Gossip Learning component and resulting in all node embeddings always being up to date, never stale. The emulator still maintains the sparse negative sampling and the layer-wise training, but the various layers are trained sequentially, rather than in parallel, and the sampling is based on global information, rather than a peer sampling protocol.

[1] Link blinded for peer review

Despite its limitations, this emulator is useful to quickly obtain performance estimates on multiple datasets, and also allows a partial "ablation study" of the architecture, to assess how the lack of global information and the decoupled layer-wise training affect the overall quality of the resulting embedding.

5 Evaluation

We evaluate DecGNN using a mix of full and approximate simulations, using both the programs we developed. The goal of this evaluation is not to provide a comprehensive study of all hyperparameters, identify the best-performing model, or reach state-of-the-art results. Instead, the goal is to compare the centralized and decentralized training of a simple GNN model to measure the performance penalty incurred by the latter and evaluate the tradeoff offered by DecGNN with respect to decentralization and prediction quality.

5.1 Setup

For a fair comparison, we train the same GNN architecture in both centralized and decentralized settings, using the same self-supervised loss function for both, applied at the final layer in the centralized case and at every layer in the decentralized one, as dicussed in section 3.2. We then separately train a logistic regressor on top of these embeddings to predict ground-truth node labels. We use the average 5-fold cross-validation accuracy on this task as a proxy for the quality of the embeddings produced by the self-supervised GNNs.

In terms of GNN architecture, we utilize a simple 2-layer GCN[14] with embedding dimension $D = 128$. For the loss function, we use a variant of VICReg (Variance-Invariance-Covariance Regularizaion)[1] modified to operate on a single embedding matrix, rather than to compare embeddings from two different input augmentations. As such, it forgoes the invariance terms while maintaining the others unmodified. Based on early exploratory analysis, this provided better results than other options tested.

The hyperparameters of DecGNN are set as follows. The size of the random embeddings buffer is set as $K = 20$, while $K' = 5$. The same values are also used for the peer sampling protocol, in terms of size of the sample and size of the subsample exchanged at each iteration. Backpropagation is only performed once at least 5 random embeddings have been received from peers.

Table 1. Dataset characteristics

Name	Nodes	Edges	Classes
Cora ML	2,995	16,316	7
Citeseer	4,230	10,674	6
DBLP	17,716	105,734	4

Table 1 summarizes the characteristics of the datasets employed. All datasets were retrieve via the Pytorch Geometric library, specifically using the `torch_geometric.datasets.CitationFull` loader.

5.2 Results

The first and third column of table 2 show the results obtained from DecGNN using the complete C++ simulator on the three datasets, compared with an equivalent centralized implementation. The results for each dataset is the average over 5 separate runs. There is a clear gap between the performance of the decentralized model compared to its centralized counterpart. This is intuitively expected. In the centralized scenario, the self-supervised loss can operate on the entire embedding matrix, thus using global knowledge of the graph to push the embeddings towards an optimal distribution. Furthermore, the gradients computed by the loss travel backwards across all layers, optimizing the whole network to perform its task. On the other hand, in DecGNN, the loss operating on each node only sees a small portion of the network and cannot affect the representations of previous layers. Furthermore, asynchronicity and staleness may also hinder the decentralized training, by adding noise to the embeddings fed to the model and loss.

However, the gap is relatively small, and therefore DecGNN may be a viable solution in those scenarios where centralized training is not feasible or desirable. Considering that, due to limited time and resources, no in-depth tuning of the architecture was performed, these results show that DecGNN is a promising technique, allowing a GNN to learn the graph phenomena without the need for global knowledge, centralized training or orchestration, or expensive decentralized backpropagation.

The second column of the table represents the results obtained when performing an approximate emulation of the decentralized architecture, as discussed in section 4.2. In two of three datasets, these are slightly higher than the true decentralized model, while on the third they are marginally lower. Overall, also taking into consideration the larger run-to-run variance compared to the centralized solution, it appears that the approximate Python emulator provides a slightly optimistic prediction of the decentralized performance.

Table 2. Quality of the self-supervised embeddings produced by DecGNN vs centralized training of the same GNN architecture, measured using the accuracy score on a down-stream classification task. Results over five runs.

Dataset	DecGNN		Centralized
	Full Simulation	Approximate Simulation	
Cora ML	76.2 ± 1.7	79.4 ± 1.5	81.1 ± 0.5
Citeseer	79.1 ± 1.1	78.8 ± 1.0	81.5 ± 0.7
DBLP	71.8 ± 1.4	75.9 ± 1.0	79.1 ± 0.6

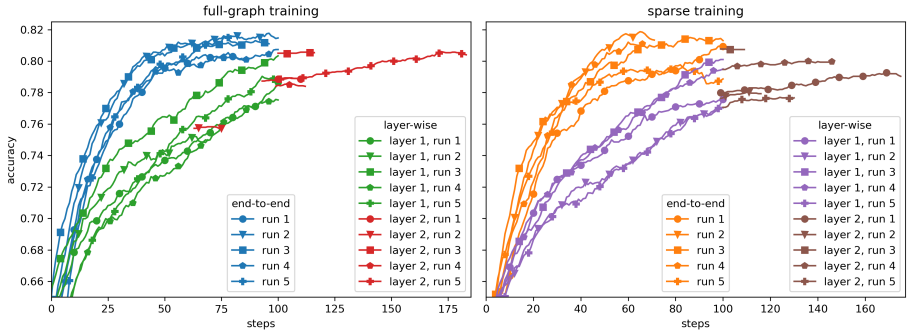

Fig. 3. Performance of various emulator configurations on Cora ML.

Figure 3 delves deeper into the Cora ML dataset, showcasing the range of configuration that can be quickly tested with a less accurate, centralized, GPU-accelerated emulator. It shows each of the 5 runs on the dataset, for a range of configurations:

- traditional end-to-end training, where all layers are trained jointly and the loss function has access to the full graph as a single batch;
- end-to-end but sparse training, where all layers are trained jointly but the loss emulates the knowledge of just a few random embeddings at each node;
- decoupled training of each layer, with the layers trained sequentially starting from the best accuracy of the previous ones, but with each loss function having access to the full graph;
- decoupled and sparse training, which is the best approximation of the fully-decentralized environment of DecGNN.

For the latter two cases, the graph plots the accuracy curve of each layer.

It is clear from the figure that restricting the input of the loss function at each node to a small random subset of the graph does not affect significantly the average accuracy, although it appears to be the cause of the increased run-to-run variance observed in table 2. On the other hand, the figure indicates that at least part of the gap between centralized and decentralized training can be attributed to the decoupled, layer-wise training. It appears that, on the datasets tested, the first layer follows a slightly slower learning curve compared to end-to-end training. The second layer immediately provides a small but visible uptick in accuracy, but then proceeds almost flatly, achieving only minor improvements over additional epochs.

Still, as mentioned, these results are in most cases slightly higher than what is achieved in a more realistic decentralized simulation. Thus, the remaining drop in accuracy can be attributed to those aspects that are not captured by the approximate emulation. One aspect is the staleness of the embeddings being fed to the model in the decentralized setting, which may increase the amount of noise in the training and hamper the learning process. Another is the simultaneous,

rather than sequential, training of the layers. The reason for this choice is that, in a truly decentralized setting, it is not trivial to establish when a certain layer has achieved peak accuracy and to coordinate the shift to the next layer, as is instead trivially done in a centralized setting.

6 Limitations and Future Work

While the results presented above look very promising, more work is needed to fully explore the potential of DecGNN and maximize its capabilities. In fact, given the number of techniques that had to be combined to achieve fully-decentralized GNN training, and thus given the many different aspects that can be further researched and analyzed in detail, we believe that this work may represent the first of a long list, with the final goal of closing the gap to centralized alternatives and achieving real-world usage of DecGNN.

Large-Scale and Advanced Evaluation First, this work only tested DecGNN with a simple GCN architecture with relatively small datasets. With more time and more computational resources, a future study may survey the leading edge of the GRL field, in terms of use cases, large datasets and advanced model architectures, evaluating the quality and convergence speed of DecGNN in a wide range of scenarios, on more realistic data and with better performing models.

Architecture and Hyperparameter Tuning As mentioned, DecGNN combines several components to achieve its goals. Each of these components comes with a number of tunable hyperparameters, and in some cases even completely different architectural options. As it would be impossible to thoroughly analyze each of these choices within the scope of a single study, in this work we have chosen what we consider reasonable default values, based on existing literature.

We here provide four notable examples of potential architectural changes that may need exploration. **First**, the dissemination of random node embeddings, instead of being achieved through random peer sampling, could be performed with random walks over the logical graph. If properly biased, these walks may improve the distribution of the samples, ensuring that key anchor nodes are sampled more frequently and that close neighbours present similar samples, for better convergence in the embedding space. **Second**, the decoupled learning architecture could be relaxed, allowing partial and limited dissemination of backward gradients. This could improve the coupling within the network and guide the early layers to produce activations more suitable to build high-level embeddings, without requiring the high degree of locking and large number of messages that a full backward pass would entail. **Third**, due to the model mixing performed by the Gossip Learning process, the Adam optimizer[13], which is well-known to be very effective, or any other moment-based optimizer cannot be used: at each iteration, in fact, the model to optimize may differ significantly from the one at the previous iteration, rendering any stored moments invalid. In this work, a simple SGD optimizer was used, but in the future optimizers specifically tailored for Gossip Learning should be developed to achieve better convergence.

Fourth, as discussed in section 5.2, it may be beneficial to train the various GNN layers sequentially, rather than simultaneously. However, this is not trivial in a fully-decentralized setting. Thus, future work may look into approaches to track the quality of the embeddings in a decentralized fashion and enable a smooth, consensus-driven transition between the training of different layers. The mechanisms employed to achieve this could then also be extended to detect and handle *data drift*, which may arise due to changes in the set of participating devices, or due to changes in the data stored on those devices, such as when not only the communication graph, but also the logical graph is dynamic.

Memory and Network Pressure Another potential area for future exploration concerns approaches to reduce the memory and network pressure of DecGNN. As discussed in section 3.6, memory requirements may be the main limitation to the deployment of DecGNN on constrained IoT devices. Network requirements, on the other hand, can be lowered at the cost of increased staleness by reducing the number of messages sent per iteration, as mentioned in section 3.7.

A different direction would be to instead reduce the size of each message, by compressing node embeddings and model weights. This could be achieved by reducing the precision of each scalar, such as by switching to half-precision floating point formats, and/or by employing sparsification techniques[19]. The weights and embeddings could also be stored in compressed format by the receiving device, thus reducing both the network and memory pressure. As each layer is treated as an independent model, the device would need to decompress only the embeddings and weights relative to one layer at a time, during the training step. Note that the addition of these compression would not modify the training process itself, which would still rely on traditional 32 bits precision operations. As an additional optimization, quantized training could be used to further reduce memory and computation requirements[5].

Privacy Considerations Finally, while ensuring data privacy was one of the motivations to develop DecGNN, this work has not focused on the rigorous analysis of privacy guarantees, which is left for future work. The current architecture minimizes the knowledge of the raw logical graph, by requiring each node to only know its own edges and the features of its direct neighbours. However, several studies have shown that deep learning models can easily leak private training data[16]. Thus, it is necessary to assess the privacy impact of sharing node embeddings and the weights of the model being trained. If needed, well-established privacy-preservation techniques, such as Differential Privacy[6], should be combined with DecGNN to provide stronger privacy guarantees.

7 Conclusions

In existing literature, GNN training has been performed almost exclusively in centralized, and sometimes federated, settings, which may raise issues of scalability, fault tolerance, trust and privacy. In this work, we envisioned DecGNN,

a fully-decentralized, data-private architecture for GNN training, which naturally addresses these issues. Our evaluation showed that our direction is promising, with only a relatively small performance gap to centralized settings. Furthermore, we laid out a clear path for future research to close this gap, hopefully laying the foundation for a line of work that will enable large-scale, fully-decentralized training of GNNs in a real-world setting.

8 Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement Innovative Training Networks (ITN) - RAIS No 813162.

References

1. Bardes, A., Ponce, J., LeCun, Y.: Vicreg: Variance-invariance-covariance regularization for self-supervised learning. arXiv preprint arXiv:2105.04906 (2021)
2. Belilovsky, E., Eickenberg, M., Oyallon, E.: Decoupled greedy learning of cnns. In: International Conference on Machine Learning. pp. 736–745. PMLR (2020)
3. Belilovsky, E., Leconte, L., Caccia, L., Eickenberg, M., Oyallon, E.: Decoupled greedy learning of cnns for synchronous and asynchronous distributed learning. arXiv preprint arXiv:2106.06401 (2021)
4. Chen, F., Wang, Y.C., Wang, B., Kuo, C.C.J.: Graph representation learning: a survey. APSIPA Transactions on Signal and Information Processing **9**, e15 (2020)
5. Cheng, Y., Wang, D., Zhou, P., Zhang, T.: A survey of model compression and acceleration for deep neural networks. arXiv preprint arXiv:1710.09282 (2017)
6. Dwork, C.: Differential privacy: A survey of results. In: Theory and Applications of Models of Computation: 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings 5. pp. 1–19. Springer (2008)
7. Giaretta, L., Girdzijauskas, Š.: Gossip learning: Off the beaten path. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 1117–1124. IEEE (2019)
8. He, C., Balasubramanian, K., Ceyani, E., Yang, C., Xie, H., Sun, L., He, L., Yang, L., Yu, P.S., Rong, Y., et al.: Fedgraphnn: A federated learning system and benchmark for graph neural networks. arXiv preprint arXiv:2104.07145 (2021)
9. Jaderberg, M., Czarnecki, W.M., Osindero, S., Vinyals, O., Graves, A., Silver, D., Kavukcuoglu, K.: Decoupled neural interfaces using synthetic gradients. In: International conference on machine learning. pp. 1627–1635. PMLR (2017)
10. Jaiswal, A., Babu, A.R., Zadeh, M.Z., Banerjee, D., Makedon, F.: A survey on contrastive self-supervised learning. Technologies **9**(1), 2 (2020)
11. Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.M., Van Steen, M.: Gossip-based peer sampling. ACM Transactions on Computer Systems (TOCS) **25**(3), 8–es (2007)
12. Jiang, M., Jung, T., Karl, R., Zhao, T.: Federated dynamic gnn with secure aggregation. arXiv preprint arXiv:2009.07351 (2020)
13. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
14. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)

15. McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, B.A.: Communication-efficient learning of deep networks from decentralized data. In: Artificial intelligence and statistics. pp. 1273–1282. PMLR (2017)
16. Mireshghallah, F., Taram, M., Vepakomma, P., Singh, A., Raskar, R., Esmaeilzadeh, H.: Privacy in deep learning: A survey. arXiv preprint arXiv:2004.12254 (2020)
17. Ormándi, R., Hegedűs, I., Jelasity, M.: Gossip learning with linear models on fully distributed data. Concurrency and Computation: Practice and Experience **25**(4), 556–571 (2013)
18. Tian, Y., Chen, X., Ganguli, S.: Understanding self-supervised learning dynamics without contrastive pairs. In: Meila, M., Zhang, T. (eds.) Proceedings of the 38th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 139, pp. 10268–10278. PMLR (18–24 Jul 2021), https://proceedings.mlr.press/v139/tian21a.html
19. Wangni, J., Wang, J., Liu, J., Zhang, T.: Gradient sparsification for communication-efficient distributed optimization. Advances in Neural Information Processing Systems **31** (2018)
20. Wu, C., Wu, F., Cao, Y., Huang, Y., Xie, X.: Fedgnn: Federated graph neural network for privacy-preserving recommendation. arXiv preprint arXiv:2102.04925 (2021)
21. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. IEEE transactions on neural networks and learning systems **32**(1), 4–24 (2020)
22. Zhang, C., Xie, Y., Bai, H., Yu, B., Li, W., Gao, Y.: A survey on federated learning. Knowledge-Based Systems **216**, 106775 (2021)