Consensus Optimization Graph Neural Networks

Olga Solodova[1] **Ryan P. Adams**[1]

[1]Department of Computer Science, Princeton University, Princeton, New Jersey, USA

Abstract

Implicit graph neural networks (GNNs) cast inference as optimization, computing node embeddings as the solution to a graph-structured optimization problem. By coupling node embeddings through a shared global objective, implicit GNNs naturally capture long-range dependencies, addressing a key limitation of standard message-passing GNNs. Another appealing property of these models is that inference can be performed in a decentralized and asynchronous manner, making them well suited for deployment in multi-agent systems. However, training is costly, as it requires solving optimization problems in both the forward and backward pass. In this work, we focus on graph-level prediction and introduce an approach for directly computing graph-level representations in implicit GNNs that bypasses the need for node embeddings. We do this by reformulating the internal optimization problem to operate over a shared graph embedding. This significantly reduces both computation and memory costs, while retaining compatibility with decentralized and asynchronous inference. Experiments across multiple benchmarks show that the shared-embedding approach consistently matches the predictive performance of using node embeddings, providing a simpler and more efficient alternative for graph-level tasks.

1 INTRODUCTION

Graph neural networks (GNNs) are a standard framework for learning on graph-structured data, in domains spanning molecular chemistry [Gilmer et al., 2017], social networks [Fan et al., 2019], and multi-agent robotics [Blumenkamp et al., 2022]. A central challenge in graph learning is modeling long-range dependencies between distant nodes, since conventional GNN architectures compute node embeddings using message passing. To propagate information beyond local neighborhoods, GNNs typically stack message-passing layers, thereby progressively enlarging each node's receptive field. However, increasing depth in this way often leads to over-smoothing or oversquashing of node embeddings, which can degrade downstream task performance [Rusch et al., 2023, Alon and Yahav, 2021].

Implicit GNNs have recently emerged as an alternative approach to modeling long-range dependencies [Gu et al., 2020, Liu et al., 2021a, Baker et al., 2023, Zhu et al., 2021, Yang et al., 2021]. Rather than specifying a finite sequence of layers, these models define node embeddings implicitly as the solution to a graph-structured optimization problem (optimization GNNs) or fixed-point equation (fixed-point GNNs). By jointly optimizing node embeddings under a shared global objective, implicit GNNs enable information to propagate globally while avoiding pathologies associated with deep message-passing architectures.

Beyond capturing long-range dependencies, implicit GNNs have another appealing property: under mild assumptions, they are provably robust to decentralized and asynchronous inference [Solodova et al., 2025]. This makes them well suited for real-world deployment in multi-agent systems—an area of growing interest for GNN-based methods—where centralized control is typically infeasible and synchrony is difficult to enforce. Decentralized execution is a property all message passing GNNs share since message passing inherently operates over local node neighborhoods; however, in standard feed-forward architectures, node updates must execute synchronously per layer. In contrast to conventional GNNs, implicit GNNs admit decentralized inference procedures that are inherently robust to asynchrony.

Despite these advantages, training implicit GNNs is computationally expensive in both memory and runtime. The forward pass requires solving an iterative optimization problem rather than performing a standard feed-forward evaluation, and parameter optimization relies on implicit differentiation,

Proceedings of the 42nd Conference on Uncertainty in Artificial Intelligence (UAI 2026), PMLR 337:6445–6466.

which entails solving a linear system. In both cases, solvers operate over variables whose dimension equals that of the combined node embeddings, which scales with the number of nodes. This leads to per-iteration costs that scale at least linearly with graph size.

In this work, we focus on graph-level prediction tasks, and propose modifying the optimization objective in optimization GNNs to operate over a single (consensus) graph embedding instead of node embeddings. A graph-level representation is obtained directly as the solution to a consensus optimization problem, rather than by aggregating node embeddings; we call the resulting class of models consensus optimization GNNs. As a result of this reformulation, the size of the optimization variable in the forward and backward passes no longer scales with the number of nodes, significantly reducing the cost of training. To validate this approach, we compare performance of the energy GNN architecture Solodova et al. [2025] to a modified *consensus* energy GNN, where node embeddings are replaced with a shared graph embedding. Empirical results on a range of datasets demonstrate that using a shared graph embedding matches the performance of using aggregated node embeddings while reducing training cost.

We further show that consensus optimization GNNs remain amenable to decentralized and asynchronous inference under the same assumptions used for optimization GNNs that compute node embeddings. We also extend the convergence analysis of Solodova et al. [2025] to address computation of graph-level representations from node embeddings under decentralized and asynchronous inference. While their work establishes convergence of node embeddings in this setting, they do not consider how to aggregate these embeddings into graph-level representations. Drawing on the consensus optimization literature, we discuss mechanisms for pooling node embeddings that are compatible with decentralized and asynchronous execution.

Our contributions are summarized as follows:

1. We propose recasting the optimization in optimization GNNs for graph-level tasks to operate over a shared graph embedding, achieving substantial reductions in memory and compute during training without performance degradation.

2. We show that the consensus reformulation is compatible with decentralized and asynchronous inference.

3. For optimization GNNs that use node embeddings, we show that adding a pooling layer to aggregate optimized embeddings prior to prediction preserves robustness to decentralized and asynchronous inference.

2 BACKGROUND AND RELATED WORK

Implicit GNNs define node embeddings implicitly as the solution to a graph-structured optimization problem with learnable parameters, rather than being obtained by stacking a finite number of message-passing layers. They can be divided into two classes: fixed-point GNNs, which obtain embeddings as the fixed-point of a contractive message passing function [Scarselli et al., 2009, Gu et al., 2020, Liu et al., 2021a, Baker et al., 2023], and optimization GNNs Ma et al. [2021], Zhu et al. [2021], Liu et al. [2021b], Solodova et al. [2025], which compute embeddings by minimizing a scalar function defined over the graph. Embeddings can be obtained via iterative local updates; in this sense, implicit GNNs can be interpreted as weight-tied 'infinite-layer' GNNs where effective depth is determined by the number of iterations to reach equilibrium. Training implicit GNNs is computationally expensive as a result of framing the forward pass as an optimization problem and using implicit differentiation in the backward pass. A number of works mitigate these costs by using mini-batch training strategies, specialized or approximate solvers, or architectures that admit closed-form solutions (see Appendix A). Our work reduces cost in a fundamentally different way through a structural change to the underlying optimization problem.

Since embeddings are obtained through iterative local computation, implicit GNNs have a natural connection to distributed algorithms over networks. Each node repeatedly updates a local variable using information from its neighbors; this is precisely the computational model studied in the distributed computation literature, which encompasses fixed-point iterations, gradient descent, and general operator-splitting methods as special cases [Bertsekas and Tsitsiklis, 1989]. This connection underlies a number of works: for example, APPNP [Gasteiger et al., 2019] derives its propagation scheme from personalized PageRank iteration; Ma et al. [2021], Zhu et al. [2021] interpret a broad class of GNNs as performing iterative optimization; and Yang et al. [2021] construct GNN architectures by unrolling classical distributed algorithms.

A key consequence of this correspondence is that convergence theory from the distributed computation literature applies directly to implicit GNNs; this has been exploited to show that inference can be executed in a fully decentralized and asynchronous manner under mild assumptions [Solodova et al., 2025]. Building on this perspective, we situate optimization GNNs within the framework of decentralized consensus optimization, which addresses problems where agents cooperate to minimize a shared objective using local computation and communication. We directly leverage the theoretical foundations of this literature, which establish convergence guarantees under practical constraints such as asynchrony, time-varying topologies, and communication delays [Nedić et al., 2018].

Notation For a graph $\mathcal{G}$ we use $\mathcal{V} = \{1, ..., n\}$ to denote the node set and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ to denote the edge set. The graph may have node and edge features $\boldsymbol{X} \in \mathbb{R}^{n \times p}$ and $\boldsymbol{E} \in \mathbb{R}^{|\mathcal{E}| \times r}$; we overload notation and use $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{X}, \boldsymbol{E})$ to denote the graph and its associated data. We use $\mathcal{N}_i$ to denote the set of neighbors of node i.

3 CONSENSUS OPTIMIZATION GRAPH NEURAL NETWORKS

Optimization GNNs compute node embeddings $\boldsymbol{H} \in \mathbb{R}^{n \times k}$ as the solution to a convex optimization problem of the form

$$\boldsymbol{H}^* = \arg\min_{\boldsymbol{H}} E_\theta(\mathcal{G}, \boldsymbol{H}), \tag{1}$$

where $E_\theta : \mathcal{G} \times \mathbb{R}^{n \times k} \to \mathbb{R}$ is a scalar-valued convex 'energy' function with parameters θ. The energy decomposes as a sum of per-node objectives that depend only on local neighborhoods:

$$E_\theta(\mathcal{G}, \boldsymbol{H}) = \sum_{i=1}^{n} e_\theta\left(\boldsymbol{h}_i, \boldsymbol{x}_i, \{\boldsymbol{h}_j, \boldsymbol{x}_j, \boldsymbol{e}_{ij}\}_{j \in \mathcal{N}_i}\right). \tag{2}$$

Optimization GNN architectures differ in their parameterization of e_θ, but all share the general structure above. While it may seem restrictive to assume E_θ is convex in $\boldsymbol{H}$, Solodova et al. [2025] suggest a flexible class of parameterizations for e_θ by employing partially input-convex neural networks (PICNNs) [Amos et al., 2017], with the resulting architecture referred to as an energy GNN. PICNNs are multi-layer neural networks which constrain specific parameters such that the output is convex with respect to a subset of the inputs; for e_θ, the embeddings $\boldsymbol{h}$ are convex inputs. The use of PICNNs yields a flexible class of convex functions for the graph energy E_θ. We describe the architecture of E_θ when using PICNNs in Appendix B.

Once optimal embeddings $\boldsymbol{H}^*$ are computed, node predictions are obtained by applying a readout function (e.g. a neural network) independently to each embedding. For graph-level predictions, embeddings are aggregated to obtain a graph-level representation prior to applying the readout.

We propose a simpler approach for graph-level tasks that bypasses node embeddings entirely and removes the need for a pooling layer. Specifically, we modify the optimization problem in (1) to use a single embedding $\boldsymbol{h} \in \mathbb{R}^k$:

$$\boldsymbol{h}^* = \arg\min_{\boldsymbol{h}} E_\theta(\mathcal{G}, \boldsymbol{h}) \tag{3}$$

where E_θ has the same form as in (2) except all nodes share a single embedding $\boldsymbol{h}$ rather than using individual embeddings $\boldsymbol{h}_i$. Writing out E_θ, we now have

$$E_\theta(\mathcal{G}, \boldsymbol{h}) = \sum_{i=1}^{n} e_\theta\left(\boldsymbol{h}, \boldsymbol{x}_i, \{\boldsymbol{x}_j, \boldsymbol{e}_{ij}\}_{j \in \mathcal{N}_i}\right). \tag{4}$$

Solving the optimization problem above directly yields a graph-level embedding $\boldsymbol{h}^*$ without requiring node embeddings or a pooling layer.

This formulation is directly connected to consensus optimization, which considers problems of the form

$$\min_{\boldsymbol{h}} F(\boldsymbol{h}) := \sum_{i=1}^{n} f_i(\boldsymbol{h}) \tag{5}$$

where each $f_i : \mathbb{R}^k \to \mathbb{R}$ is local to agent i. This matches the optimization problem in (3) if we specify

$$f_i(\boldsymbol{h}) := e_\theta\left(\boldsymbol{h}, \boldsymbol{x}_i, \{\boldsymbol{x}_j, \boldsymbol{e}_{ij}\}_{j \in \mathcal{N}_i}\right). \tag{6}$$

As such, we can view the modified E_θ in equation 4 as a graph-structured instance of a consensus optimization objective, where each local objective f_i is parameterized uniformly across nodes by θ, and depends on node i's local neighborhood $\mathcal{N}_i$. Drawing on this connection, we call optimization GNNs which compute a shared embedding $\boldsymbol{h}$ consensus optimization GNNs. While the consensus optimization literature typically assumes the f_i's are fixed and known, here the shared parameters θ are learned from data, allowing the model to adapt the landscape of the global objective E_θ to downstream tasks.

It is worth mentioning that the consensus reformulation is particularly natural for *optimization-based* implicit GNNs, because replacing the optimization variable with a shared embedding is a well-defined and simple modification. For fixed-point implicit GNNs, which compute embeddings satisfying $\boldsymbol{H}^* = f_\theta(\boldsymbol{H}^*; \mathcal{G})$, an analogous reformulation is less straightforward, as the fixed-point iteration is defined node-wise and does not naturally admit a single shared state variable. As a representative example, consider the IGNN architecture [Gu et al., 2020], in which node embeddings are computed as a fixed point satisfying $\boldsymbol{H}^* = \phi(\boldsymbol{A}\boldsymbol{H}^*\boldsymbol{W}_1 + \boldsymbol{X}\boldsymbol{W}_2)$, where ϕ is a non-linear activation function, $\boldsymbol{W}_1 \in \mathbb{R}^{k \times k}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{p \times k}$ are parameters and $\boldsymbol{A}$ denotes the adjacency matrix. The architecture is intrinsically defined over node-level representations; replacing $\boldsymbol{H}$ with a single shared vector $\boldsymbol{h}$ would require a more substantial and non-obvious architectural redesign rather than a simple reformulation.

In section 5, we use results from the decentralized consensus literature to show that in the consensus approach, E_θ can be minimized in a decentralized (and asynchronous) manner despite $\boldsymbol{h}$ being a global optimization variable.

4 TRAINING

Training implicit GNNs is more costly than standard feed-forward GNNs because embeddings are defined as the solution to an optimization problem, typically obtained using an iterative solver. Furthermore, to compute derivatives with

respect to parameters θ, the backward pass uses implicit differentiation, which requires solving a linear system whose dimension equals that of the embedding space. We briefly summarize the result of using implicit differentiation below; a more detailed derivation is in Appendix D. Using $\boldsymbol{z} \in \mathbb{R}^d$ to denote embeddings in a unified way, where $\boldsymbol{z} = \boldsymbol{H} \in \mathbb{R}^{nk}$ in the node-level case and $\boldsymbol{z} = \boldsymbol{h} \in \mathbb{R}^k$ in the consensus case, for an objective $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$, computing gradients $\frac{d\mathcal{L}}{d\theta}$ requires solving the following $d \times d$ linear system:

$$\frac{\partial^2 E_\theta}{\partial \boldsymbol{z}^2}(\boldsymbol{z}^*)\lambda = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}(\boldsymbol{z}^*) \tag{7}$$

In general, both the forward optimization and the backward linear solve rely on numerical methods whose memory requirements and per-iteration compute scale at least linearly with the dimension of $\boldsymbol{z}$. Consequently, replacing node embeddings with a shared embedding reduces these solver-side costs by at least a factor of n.

Methods that explicitly form and manipulate the Hessian of E_θ benefit most from using a shared embedding. Linear systems involving the Hessian appear in both the backward pass to solve (7), and in the forward pass when using second order methods to minimize E_θ. Materializing the dense Hessian and using direct solvers requires $O(d^2)$ memory and $O(d^3)$ operations; when using node embeddings we have $d = nk$, so these costs scale quadratically and cubically with the number of nodes. By exploiting the block-sparse structure of the Hessian when using node embeddings with sparse graphs, memory and compute costs can at best be reduced to $O(|\mathcal{E}|k^2)$ and $O(|\mathcal{E}|k^3)$ when using direct solvers. This best case assumes no fill-in during factorization of the Hessian, which is only guaranteed for tree-structured graphs; in the worst case, the cost remains $O(n^2k^2)$ and $O(n^3k^3)$. The dependence on graph size is also present when using iterative linear solvers which use Hessian-vector products (e.g., conjugate gradient), with per-iteration cost scaling as $O(|\mathcal{E}|k)$ when exploiting sparsity. In contrast, when using a shared embedding, the cost in memory and compute of solving these linear systems depends only on the embedding dimension k when using either direct or iterative solvers.

We note that in the discussion above, we focus on operations isolated to the solver; the cost of evaluating the energy E_θ, its gradient, and Hessian-vector products is $O(|\mathcal{E}|k)$ in both formulations. However, the cost of evaluating the Hessian using forward-over-reverse automatic differentiation requires $O(n)$ fewer operations and $O(n)$ less memory when using shared embeddings, since d Hessian-vector products are required. Custom graph-aware computation of the Hessian can reduce this discrepancy for sparse graphs by computing only the non-zero entries, but the worst case cost (in the case of dense graphs) cannot be reduced. In contrast, Hessian evaluation when using a shared embedding is $O(|\mathcal{E}|k^2)$ (i.e. the cost of evaluating k Hessian-vector products) without requiring specialized implementations and regardless of graph structure. In practice, this makes it feasible to use the full Hessian in dense direct linear solves for the backward pass linear system, and exact Newton updates (using direct solvers to compute search directions) for optimization of E_θ. These methods often converge faster, are more numerically stable, and avoid the tuning and approximation errors of iterative solvers, but are prohibitively expensive when using node embeddings on moderately sized graphs. For methods not using the Hessian, the savings in memory and compute arise primarily from the reduced dimension of the optimization variable in the objective E_θ and reduced size of the linear system in equation 7.

5 DECENTRALIZED AND ASYNCHRONOUS INFERENCE

When using node embeddings, the decomposition of E_θ over local neighborhoods allows the optimization problem in (1) to be solved via decentralized gradient descent, where each node maintains and updates its own embedding using only information from neighbors. We show that using a consensus embedding does not sacrifice this property, and likewise admits decentralized optimization procedures that are robust to asynchrony.

5.1 E_θ WITH NODE EMBEDDINGS

Let e_θ^i denote node i's local objective, i.e., the i^{th} term in equation 2, and let $\boldsymbol{g}_{ij}$ denote the gradient of e_θ^i with respect to $\boldsymbol{h}_j$. Omitting node and edge features for clarity, we have that for each node i at time t,

$$\boldsymbol{g}_{ij}^t := \nabla_{\boldsymbol{h}_j} e_\theta^i\left(\boldsymbol{h}_i^t, \{\boldsymbol{h}_j^t\}_{j\in\mathcal{N}_i}\right), \tag{8}$$

$$\boldsymbol{h}_i^{t+1} = \boldsymbol{h}_i^t - \alpha \textstyle\sum_{j\in\mathcal{N}_i\cup\{i\}} \boldsymbol{g}_{ji}^t, \tag{9}$$

where $\alpha > 0$ is the step size. The quantities $\boldsymbol{h}_j^t$ and $\boldsymbol{g}_{ji}^t$ are communicated by neighbors $j \in \mathcal{N}_i$.

This decentralized procedure can be executed asynchronously, where nodes update at different times and use potentially stale neighbor information. Assuming E_θ is strongly convex in $\boldsymbol{H}$ and each local objective e_θ^i has Lipschitz continuous gradients, Solodova et al. [2025] prove that the embeddings converge to the unique solution of (1) under *partial asynchrony*—a standard model of asynchrony in distributed computation that assumes bounded time between successive node updates and bounded staleness of neighbor information (Assumption C.1). Note that E_θ can be made strongly convex in $\boldsymbol{H}$ with the addition of a quadratic regularizer $\beta||\boldsymbol{h}_i||_2^2$ to each node i's local objective. A more detailed description of partial asynchronism and the convergence result from Solodova et al. [2025] are provided in Appendix C.

5.2 E_θ WITH A CONSENSUS EMBEDDING

The shared embedding $\boldsymbol{h}^*$ can be computed using standard techniques for decentralized consensus optimization. In particular, the consensus optimization problem in (4) can equivalently be expressed as

$$\min_{\boldsymbol{h}_1,\ldots,\boldsymbol{h}_n} \sum_{i=1}^{n} e_\theta^i(\boldsymbol{h}_i) \quad \text{s.t.} \quad \boldsymbol{h}_1 = \boldsymbol{h}_2 = \cdots = \boldsymbol{h}_n \tag{10}$$

where each agent maintains a local copy $\boldsymbol{h}_i \in \mathbb{R}^k$ of the optimization variable. Optimization algorithms for problems of this form specify how agents cooperate via local communication to find the minimizer $\boldsymbol{h}^*$ of the global objective.

A simple method for solving (10) is decentralized gradient descent (DGD) [Nedić and Ozdaglar, 2009]. DGD assumes a symmetric non-negative doubly stochastic averaging matrix $\boldsymbol{W} \in \mathbb{R}^{n\times n}$ whose sparsity is consistent with that of the underlying communication graph. A common choice is to use Metropolis-Hasting weights, where $\boldsymbol{W}$ is defined as

$$\boldsymbol{W}_{ij} = \begin{cases} \dfrac{1}{1+\max(|\mathcal{N}_i|,|\mathcal{N}_j|)}, & \text{if } j \in \mathcal{N}_i, \\ 1 - \displaystyle\sum_{j\in\mathcal{N}_i} W_{ij}, & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases}$$

DGD applies local gradient steps followed by neighbor averaging, where for each node i at time t,

$$\boldsymbol{h}_i^{t+1} = \boldsymbol{h}_i^t - \alpha\nabla_{\boldsymbol{h}} f_i(\boldsymbol{h}_i^t) + \textstyle\sum_{j\in\mathcal{N}_i} \boldsymbol{W}_{ij}(\boldsymbol{h}_i^t - \boldsymbol{h}_j^t) \tag{11}$$

where $\alpha > 0$. In words, the gradient term pulls $\boldsymbol{h}_i$ to minimize f_i, and the mixing term pulls $\boldsymbol{h}_i$ to its neighbors' values $\boldsymbol{h}_j$. All quantities in the iteration are obtainable through local communication, so the overall update step is decentralized.

When using diminishing step sizes α, each $\boldsymbol{h}_i^*$ obtained by executing DGD converges linearly to the optimal solution $\boldsymbol{h}^*$ of the optimization problem in (5) under the assumptions that each f_i is strongly convex and has Lipschitz continuous gradients [Yuan et al., 2016]. Under the same assumptions, DGD converges linearly to a neighborhood of the optimal solution when using constant step size, where the optimality gap depends on the connectivity of the graph and the step size. When executed under partial asynchrony (Assumption C.1) DGD retains the same convergence guarantees [Wu et al., 2026].

Beyond DGD, many other methods exist for asynchronous and decentralized consensus optimization, which may be better suited to particular deployment settings [Wu et al., 2018, Peng et al., 2016, Tian et al., 2020, Eisen et al., 2017, Nedić et al., 2018]. One well-studied alternative is the gradient-push method [Nedić and Olshevsky, 2015, Assran and Rabbat, 2021], which, unlike DGD, is applicable when the underlying communication graph is directed and time-varying. This arises in many practical settings such as mobile sensor or multi-agent networks with intermittent or asymmetric communication. Motivated by these more general scenarios and anticipating future applications of consensus optimization GNNs in multi-agent environments, we provide a brief summary of the gradient-push method in Appendix E.3.

6 DECENTRALIZED POOLING OF NODE EMBEDDINGS

In this section, we switch our focus to optimization GNNs operating over node embeddings and discuss how embeddings can be aggregated into graph-level representations in a decentralized setting. The convergence result in Solodova et al. [2025] establishes that the solution to equation 1 can be computed via decentralized and asynchronous gradient descent. This is sufficient for node-level prediction, where each node computes predictions from its own embedding $\boldsymbol{h}_i^*$. However, the analysis does not consider the problem of computing graph-level representations under decentralized and asynchronous inference. In their experiments, graph-level tasks are recast as node-level prediction problems, requiring each node to produce the same graph-level output from its own embedding. This avoids the more natural approach of pooling node embeddings into a graph-level representations, and instead imposes an artificial constraint: either all nodes must converge to identical embeddings, or the readout function must map distinct embeddings to identical outputs.

We draw on the consensus optimization literature to eliminate this constraint, and show that mean-pooling can be done in a decentralized and asynchronous manner. Note that during decentralized optimization of E_θ, node embeddings evolve over time, and individual nodes do not know when the global optimization has converged. Consequently, we require a dynamic average consensus protocol [Zhu and Martínez, 2010] which tracks the average of time-varying signals.

Decentralized dynamic average consensus Let $\boldsymbol{m}_i^t \in \mathbb{R}^k$ denote node i's estimate of the mean at time t, initialized as $\boldsymbol{m}_i^0 = \boldsymbol{h}_i^0$, and let $\Delta\boldsymbol{h}_i^t = \boldsymbol{h}_i^t - \boldsymbol{h}_i^{t-1}$ denote the change in node i's embedding at time t as a result of the gradient step in (9). Using the averaging matrix $\boldsymbol{W}$ defined in section 5.2, each node i updates its estimate $\boldsymbol{m}_i$ at time t as follows:

$$\boldsymbol{m}_i^{t+1} = \Delta\boldsymbol{h}_i^{t+1} + \textstyle\sum_{j\in\mathcal{N}_i\cup\{i\}} \boldsymbol{W}_{ij}\boldsymbol{m}_j^t \tag{12}$$

where node i receives $\boldsymbol{W}_{ij}\boldsymbol{m}_j^t$ from neighbors. This update shares the same structure as in DGD, consisting of a consensus term and a local perturbation occurring at node i. If perturbations $\Delta\boldsymbol{h}_i^t$ eventually go to zero, the mean estimates $\boldsymbol{m}_i^t$ for each node converge to the average of the steady-state values $1/n \sum_{k=1}^n \boldsymbol{h}_k^t$. For a detailed convergence proof,

we refer the reader to Zhu and Martínez [2010]. In our case, since embeddings converge to the minimizer $\boldsymbol{H}^*$, each node's local estimate converges to $1/n \sum_{k=1}^{n} \boldsymbol{h}_k^*$.

We note that proving convergence relies on the fact that the stationary distribution of irreducible doubly stochastic matrices is uniform, such that $\lim_{t\to\infty} \boldsymbol{W}^t = 1/n \mathbf{1}\mathbf{1}^T$. Under partially asynchronous execution, the *effective* averaging matrix $\boldsymbol{W}$ used at time t may not be doubly stochastic and convergence is not guaranteed. In this setting, we can instead use the perturbed push-sum protocol, which only requires that $\boldsymbol{W}$ be column stochastic to guarantee convergence [Nedić and Olshevsky, 2015]; this guarantee also holds under partially asynchrony [Assran and Rabbat, 2021]. We discuss the protocol in more detail in Appendix E.1.

Other pooling approaches Beyond the mean, a natural question is whether other aggregates of the node embeddings can be computed in the decentralized asynchronous setting. A sufficient condition for an aggregate of time-varying values to be trackable via push-sum-based protocols is that it decomposes as a sum of per-node terms. Non-linear aggregates such as the maximum, minimum, or median do not admit such an additive decomposition, and are difficult to track for time-varying values without additional coordination. We also note that sums, a common aggregation mechanism in GNNs, are not directly computable unless the total network size n is known to each node, which also requires additional coordination. We provide a more detailed discussion of decentralized and asynchronous pooling approaches in Appendix E.1.

7 LIMITATIONS

The central distinction between the consensus and non-consensus approaches is how graph structure enters the energy objective E_θ and the mechanism by which individual nodes influence the graph-level embedding.

The non-consensus model couples node-level embeddings through the graph energy E_θ, allowing individual embeddings to reflect multi-hop information. This is in contrast with the consensus model, where the shared embedding is the solution to an optimization problem over the multiset of one-hop local observations. Under this formulation, it follows that graphs with identical multisets of one-hop neighborhoods induce the same embedding (regardless of global arrangement), making them indistinguishable to the consensus model. Notably, this is a consequence of defining each node's contribution to the consensus energy objective in (4) as a function of information in its one-hop neighborhood, not a fundamental limitation of consensus optimization as a framework. Richer local objectives incorporating multi-hop or positional information can integrate higher-order structure in the shared embedding.

An advantage of the non-consensus approach is that it allows incorporation of multi-hop information in individual node embeddings, even if local objectives use only one-hop information. However, there are practical limits to how effectively information can propagate over long graph distances, which can make this advantage less salient. Long-range sensitivity is characterized by the derivative of the solution node embeddings $\boldsymbol{H}^* = h^*(\boldsymbol{x}) \in \mathbb{R}^{nk}$ with respect to the input features $\boldsymbol{x}$, where by implicit differentiation we have

$$\frac{\partial h^\star}{\partial \boldsymbol{x}} = \left(\frac{\partial^2 E_\theta}{\partial \boldsymbol{H}^2}\right)^{-1} \frac{\partial^2 E_\theta}{\partial \boldsymbol{x} \partial \boldsymbol{H}}.$$

The mixed derivative ($\frac{\partial^2 E_\theta}{\partial \boldsymbol{x} \partial \boldsymbol{H}}$) is local (with sparsity dictated by the 2-hop adjacency matrix), while long-range influence is mediated by the inverse Hessian. The Hessian is block-sparse and its inverse is generally dense, so information propagation is not limited to a fixed number of hops. However, for all node embeddings to be influenced by an informative feature localized at a small set of nodes, the inverse Hessian must preferentially amplify graph-spanning eigenvectors. On large locally connected graphs, this typically requires the Hessian to have graph-spanning eigenvectors with relatively small eigenvalues compared to eigenvectors associated with local variation, increasing the spectral anisotropy of the Hessian. As this anisotropy grows, the Hessian tends to become increasingly ill-conditioned, leading to stiff forward optimization and ill-conditioned linear systems in the backward implicit differentiation solve. This tends to lead to unstable training, making it practically difficult to learn an energy function capable of effectively propagating highly localized informative signals across large graphs.

As a result, when task-relevant information is localized to a small number of nodes, the corresponding signal may remain concentrated in only a few node embeddings and subsequently be diluted by graph-level pooling. In contrast, the consensus formulation can straightforwardly preserve sparse informative signals in the shared embedding. Local objectives for uninformative nodes can be learned to be flat in the class-discriminative directions of the embedding space without the overall Hessian — a sum of local $k \times k$ Hessians — becoming poorly conditioned.

8 EXPERIMENTS

To validate the use of a consensus embedding in optimization GNNs, we compare performance of the energy GNN architecture [Solodova et al., 2025] to its consensus counterpart (which we call a consensus energy GNN). We focus on the energy GNN architecture due to its flexible parameterization and strong performance relative to other optimization GNNs. We use the same architecture for E_θ for both models, where in the consensus case we simply modify E_θ to operate over a single graph embedding. For the

Table 1: Classification accuracy (%) for consensus vs non-consensus energy GNN on benchmark datasets. Mean and standard deviations are computed over 5 seeds and 10 folds of the data.

DATA SET	MUTAG	PROTEINS	COX2	PTC	NCI1
# GRAPHS	188	1113	467	344	4110
AVG # NODES	17.9	39.1	41.2	25.5	29.9
AVG # EDGES	19.8	72.8	43.4	14.7	32.3
NON-CONSENSUS (DIRECT)	93.2 ± 4.7	-	85.1 ± 5.3	68.5 ± 7.1	77.3 ± 2.0
CONSENSUS (DIRECT)	91.7 ± 7.4	77.9 ± 2.7	84.9 ± 3.8	69.8 ± 7.0	74.6 ± 1.8
NON-CONSENSUS (CG)	93.8 ± 4.5	77.4 ± 3.3	84.9 ± 4.9	67.9 ± 7.0	76.9 ± 2.0
CONSENSUS (CG)	92.5 ± 7.4	76.6 ± 3.1	85.0 ± 4.2	69.4 ± 6.8	73.3 ± 2.0

Table 2: Test performance of consensus and non-consensus energy GNN on benchmark datasets. Metrics are ROC-AUC (ogbg-molhiv), AP (peptides-func), and MAE (peptides-struct). Mean and standard deviation are computed over 5 seeds.

DATA SET	OGBG-MOLHIV	PEPTIDES-STRUCT	PEPTIDES-FUNC
# GRAPHS	41127	15535	15535
AVG # NODES	25.5	150.9	150.9
AVG # EDGES	27.5	153.7	153.7
NON-CONSENSUS (CG)	0.717 ± 0.016	0.316 ± 0.015	0.393 ± 0.016
CONSENSUS (DIRECT)	0.713 ± 0.037	0.317 ± 0.011	0.427 ± 0.002

original energy GNN, we compute a graph embedding via $1/n \sum_{i=1}^{n} f_\psi(\boldsymbol{h}_i^*)$, where f_ψ is a linear layer. In both models, we include a final three-layer neural network to compute predictions from the graph embedding. We use Newton's method to optimize E_θ in the forward pass, and use the same batch size for both models in each experiment.

We compare performance on several benchmark GNN datasets, and on two synthetic tasks more aligned with applications in multi-agent systems, where the data consist of spatial graphs. The first is the synthetic MNIST "terrain" classification task introduced in Solodova et al. [2025], and the second is an intersection-over-union (IoU) prediction task we introduce in this work. Further architecture, training, and dataset details are in Appendix F.

8.1 GNN BENCHMARK DATASETS

We compare performance on MUTAG, PROTEINS, COX2, PTC and NCI1 from the TUDataset collection[Morris et al., 2020], ogbg-molhiv [Hu et al., 2020], and peptides-func and peptides-struct [Dwivedi et al., 2022] (which use the same graphs but different prediction targets). Peptides-struct is a graph regression task, and the others are classification tasks. For the TU datasets, we use 10-fold cross-validation and report classification accuracy with mean and standard deviations over all folds and 3 seeds (Table 1). For ogbg-molhiv and the peptides datasets, we report mean and standard deviations of test set performance for 5 seeds (Table 2).

On the TU datasets, we compare two choices for solving the linear systems that arise in the backward pass (equation 7) and forward pass (to compute Newton search directions): DIRECT denotes use of direct solvers, and CG denotes conjugate gradient. Task performance is statistically indistinguishable between the consensus and non-consensus models on all datasets except NCI1, and is largely insensitive to solver choice. However, the consensus approach consistently reduces training time per epoch, with larger gains on datasets containing larger graphs (Table 3). On PROTEINS, which contains the largest graphs, we run out of memory when attempting to use direct solvers.

Since solver choice has little impact on task performance, for the ogbg-molhiv and peptides datasets, we use direct solvers for the consensus approach; the non-consensus model requires CG due to memory constraints. On ogbg-molhiv and peptides-struct, task performance between the two models is not statistically different; for peptides-func, the consensus model performs significantly better. For these larger datasets, training with node embeddings is an order of magnitude slower (Table 4).

Appendix F.6 contains absolute epoch times and peak GPU memory usage, and experiments with doubled parameter counts and increased embedding dimension. For datasets with the largest graphs (peptides and PROTEINS) we run

out of memory when attempting to train the larger non-consensus model. Increasing model size generally improves performance for both approaches, but consensus remains substantially more efficient: even the larger consensus model trains faster than the smaller non-consensus model. Thus, under a fixed training budget, consensus enables higher-capacity architectures that often yield improved task performance. We also note that for NCI1, where the consensus approach achieved worse performance with smaller model size, increasing size closes the gap in performance.

Table 3: Approximate relative increase in time per epoch when using the non-consensus versus consensus energy GNN on MUTAG, PROTEINS, COX2, PTC, and NCI1.

	MUTAG	PROTEINS	COX2	PTC	NCI1
DIRECT	1.3X	-	2.3X	1.8X	7.2X
CG	1.6X	6.4X	1.6X	2.1X	2.4X

Table 4: Approximate relative increase in time per epoch when using the non-consensus versus consensus energy GNN on ogbg-molhiv, peptides-func, and peptides-struct.

OGBG-MOLHIV	PEPTIDES-FUNC	PEPTIDES-STRUCT
8X	12X	10X

8.2 MNIST "TERRAIN"

In the MNIST "terrain" dataset, each graph is associated with a different binary MNIST image, and the prediction target corresponds to the digit in the image. The graphs are constructed by randomly placing $n = 10$ nodes at different pixels in the associated image, and nodes are connected by an edge if they are within 4 pixels of one another. Each node has features consisting of the coordinate and value of the pixel they occupy. Edge features corresponding to distances between nodes are also included. We show examples of graphs from this dataset in Figure 1.

We use direct solvers in the forward and backward pass linear systems for both models, and report accuracy on the test data in Table 5, with mean and standard deviation over 5 random seeds. The consensus approach achieves the same accuracy as non-consensus, but requires less time per epoch (Table 6); the difference in epoch times is less pronounced on this dataset because the graphs are small. We include absolute times in Appendix F.6.

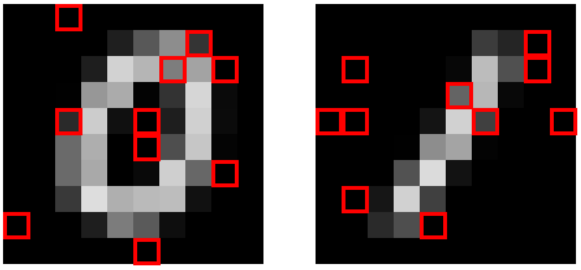

Figure 1: Example graphs for mnist-terrain experiment. Highlighted cells are those occupied by nodes. Edges (not depicted) connect nodes within 4 pixels of one another.

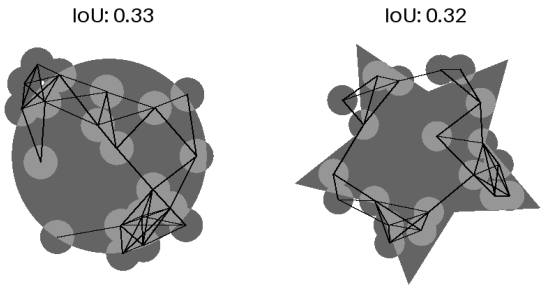

Figure 2: Example graphs for IoU experiment with circle target shape (left) and star target shape (right). Prediction targets are shown above the graph.

8.3 INTERSECTION OVER UNION

For the IoU task, we construct random graphs by sampling node positions in the unit square, and connect nodes by an edge if their distance from one another is $\leq r_{comm}$, where r_{comm} denotes an inter-node communication radius. Each node has spatial extent given by a disk of radius r_{extent}, and we define the shape $\mathcal{S}_i$ associated with a graph as the union of these disks. Given a target shape $\mathcal{T}$, the prediction target for a graph $\mathcal{G}_i$ is the IoU between $\mathcal{S}_i$ and $\mathcal{T}$. Nodes only have access to distances to their neighbors, and do not observe absolute positions (i.e. no node features are present, and edge features consist of inter-node distances). Accordingly, the IoU target is defined to be invariant to rigid transformations, and is computed by centering the graph at the geometric mean of the target $\mathcal{T}$ and maximizing IoU over rigid transforms of $\mathcal{T}$. In other words, the nodes aim to predict how 'close' they are to forming $\mathcal{T}$, without reference to a particular coordinate frame. Knowledge of the target $\mathcal{T}$ is provided only implicitly through the training objective. We construct two datasets with circle and star target shapes, each consisting of $50,000$ graphs with 20 nodes. We reserve $5,000$ graphs as test data. Figure 2 shows examples of graphs from each dataset. Additional dataset details are in Appendix F.3.

Table 5: Task performance for consensus vs non-consensus GNN on MNIST-terrain (classification accuracy) and IoU (MAE). IoU targets and predictions were scaled from $[0, 1]$ to $[0, 100]$. Mean and standard deviation are over 5 seeds.

DATA SET	NON-CONSENSUS	CONSENSUS
MNIST	96.4 ± 0.4	96.5 ± 0.3
IOU CIRCLE	1.49 ± 0.02	1.69 ± 0.01
IOU STAR	1.51 ± 0.03	1.81 ± 0.01

Table 6: Approximate relative increase in time per epoch when using the non-consensus versus consensus energy GNN on MNIST-terrain and IoU datasets.

MNIST-TERRAIN	IOU CIRCLE	IOU STAR
1.3X	2.7X	2.4X

The IoU task is inspired by the problem of shape formation in robotics [Rubenstein et al., 2014, Sun et al., 2023], where agents coordinate to form a specified shape under local communication constraints and typically are assumed not to have access to global coordinates. While we do not solve the control problem, predicting IoU provides agents with a global quality signal that can be used as feedback in decentralized shape formation.

We use direct solvers in the forward and backward pass linear systems for both models, and report the MAE on the test data in Table 5, with mean and standard deviation over 5 random seeds. Note that since IoU lies in $[0, 1]$, we scale both targets and predictions by 100, effectively expressing IoU as a percentage. Both models perform well, with mean MAE between 1.5-2.0. The non-consensus energy GNN achieves better performance on each dataset; this is likely because multi-hop structural information is important for IoU prediction, and in these experiments the consensus energy objective E_θ uses only one-hop neighborhood information. However, the time per epoch is twice as high in the non-consensus case (Table 6). As with the other experiments, raw epoch times are in Appendix F.6.

9 CONCLUSION AND FUTURE WORK

In this work, we introduce a new approach to computing graph level representations in optimization-based implicit GNNs. Optimization GNNs compute node embeddings as the solution to a graph-structured optimization problem and follow the standard approach of aggregating these embeddings to produce a graph level representation. Instead, we define a consensus optimization problem directly over a shared graph embedding. This simple reformulation reduces the dimensionality of the optimization variable—and the linear systems arising in implicit differentiation—by a factor of n, yielding substantial savings in memory and compute in solvers used during training. We show empirically that using a consensus graph embedding results in comparable task performance to using aggregated node embeddings across a wide range of datasets, while significantly reducing the time required for training, particularly for datasets with larger graphs. These results suggest that for optimization GNNs, using a consensus embedding is an effective and practical alternative for computing graph-level representations compared to aggregating node embeddings.

Optimization GNNs operating over node embeddings also have the attractive property of being amenable to decentralized and asynchronous inference, which is particularly relevant in multi-agent deployment settings. By drawing on results from the decentralized consensus optimization literature, we demonstrate that the consensus reformulation does not sacrifice this property. We also extend the convergence analysis of Solodova et al. [2025] — which established convergence of node embeddings under decentralized asynchronous inference — to show that node embeddings can be aggregated into graph-level representations in this setting via consensus averaging protocols.

More broadly, our results further highlight the close connection between implicit GNNs and distributed optimization. A natural direction for future work is extending consensus optimization GNNs to time-varying graphs by drawing on existing tools from this literature.

Acknowledgements

This work was partially supported by NSF grants IIS-2007278 and OAC-2118201. We thank Sowmya Thanvantri for her help with the IoU experiments.

References

Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations*, 2021.

Brandon Amos, Lei Xu, and J. Zico Kolter. Input convex neural networks. In *International Conference on Machine Learning*, pages 146–155, 2017.

Mahmoud S. Assran and Michael G. Rabbat. Asynchronous gradient push. *IEEE Transactions on Automatic Control*, 66(1):168–183, 2021.

Justin Baker, Qingsong Wang, Cory D. Hauck, and Bao Wang. Implicit graph neural networks: A monotone operator viewpoint. In *International Conference on Machine Learning*, pages 1521–1548, 2023.

Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989.

Jan Blumenkamp, Steven D. Morad, Jennifer Gielis, Qingbiao Li, and Amanda Prorok. A framework for real-world multi-robot systems running decentralized GNN-based policies. In *International Conference on Robotics and Automation*, pages 8772–8778, 2022.

Florence Bénézit, Vincent Blondel, Patrick Thiran, John Tsitsiklis, and Martin Vetterli. Weighted gossip: Distributed averaging using non-doubly stochastic matrices. In *2010 IEEE International Symposium on Information Theory*, pages 1753–1757, 2010. doi: 10.1109/ISIT.2010.5513273.

Qi Chen, Yifei Wang, Yisen Wang, Jianlong Chang, Qi Tian, Jiansheng Yang, and Zhouchen Lin. Efficient and scalable implicit graph neural networks with virtual equilibrium. In *2022 IEEE International Conference on Big Data (Big Data)*, 2022.

Vijay Prakash Dwivedi, Ladislav Rampášek, Mikhail Galkin, Ali Parviz, Guy Wolf, Anh Tuan Luu, and Dominique Beaini. Long range graph benchmark. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

Mark Eisen, Aryan Mokhtari, and Alejandro Ribeiro. Decentralized quasi-Newton methods. *IEEE Transactions on Signal Processing*, 65(10):2613–2628, 2017.

Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The Web Conference*, pages 417–426, 2019.

Johannes Gasteiger, Aleksandar Bojchevski, and Stephan Günnemann. Combining neural networks with personalized pagerank for classification on graphs. In *International Conference on Learning Representations*, 2019.

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272, 2017.

Fangda Gu, Heng Chang, Wenwu Zhu, Somayeh Sojoudi, and Laurent El Ghaoui. Implicit graph neural networks. In *Advances in Neural Information Processing Systems*, volume 33, pages 11984–11995, 2020.

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 2017.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

Thomas N. Kipf, , and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

Juncheng Liu, Kenji Kawaguchi, Bryan Hooi, Yiwei Wang, and Xiaokui Xiao. EIGNN: Efficient infinite-depth graph neural networks. In *Advances in Neural Information Processing Systems*, 2021a.

Juncheng Liu, Bryan Hooi, Kenji Kawaguchi, Yiwei Wang, Chaosheng Dong, and Xiaokui Xiao. Scalable and effective implicit graph neural networks on large graphs. In *The Twelfth International Conference on Learning Representations*, 2024.

Xiaorui Liu, Wei Jin, Yao Ma, Yaxin Li, Liu Hua, Yiqi Wang, Ming Yan, and Jiliang Tang. Elastic graph neural networks. In *Proceedings of the 38th International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR, 2021b.

Yao Ma, Xiaorui Liu, Tong Zhao, Yozen Liu, Jiliang Tang, and Neil Shah. A unified view on graph neural networks as graph signal denoising. *arXiv preprint arXiv:2112.11609*, 2021.

Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. TU-Dataset: A collection of benchmark datasets for learning with graphs. In *ICML Workshop on Graph Representation Learning and Beyond*, 2020.

Angelia Nedić and Alex Olshevsky. Distributed optimization over time-varying directed graphs. *IEEE Transactions on Automatic Control*, 60(3):601–615, 2015.

Angelia Nedić and Asuman Ozdaglar. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 54(1):48–61, 2009.

Angelia Nedić, Alex Olshevsky, and Michael G. Rabbat. Network topology and communication-computation tradeoffs in decentralized optimization. *Proceedings of the IEEE*, 106(5):953–976, 2018.

Zhimin Peng, Yangyang Xu, Ming Yan, and Wotao Yin. ARock: An algorithmic framework for asynchronous parallel coordinate updates. *SIAM Journal on Scientific Computing*, 38(5):A2851–A2879, 2016.

Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.

T. Konstantin Rusch, Michael M. Bronstein, and Siddhartha Mishra. A survey on oversmoothing in graph neural networks. *arXiv preprint arXiv:2303.10993*, 2023.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

Olga Solodova, Nick Richardson, Deniz Oktay, and Ryan P. Adams. Graph neural networks gone hogwild. In *International Conference on Learning Representations*, 2025.

Guibin Sun, Rui Zhou, Zhao Ma, Yongqi Li, Roderich Groß, Zhang Chen, and Shiyu Zhao. Mean-shift exploration in shape assembly of robot swarms. *Nature Communications*, 14(1):3476, 2023.

Ye Tian, Ying Sun, and Gesualdo Scutari. Achieving linear convergence in distributed asynchronous multiagent optimization. *IEEE Transactions on Automatic Control*, 65 (12):5264–5279, 2020.

Tianyu Wu, Kun Yuan, Qing Ling, Wotao Yin, and Ali H. Sayed. Decentralized consensus optimization with asynchrony and delays. *IEEE Transactions on Signal and Information Processing over Networks*, 4(4):794–807, 2018.

Xuyang Wu, Changxin Liu, Sindri Magnússon, and Mikael Johansson. Asynchronous distributed optimization with delay-free parameters. *IEEE Transactions on Automatic Control*, 71(1):259–274, 2026.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

Yongyi Yang, Tang Liu, Yangkun Wang, Jinjing Zhou, Quan Gan, Zhewei Wei, Zheng Zhang, Zengfeng Huang, and David Wipf. Graph neural networks inspired by classical iterative algorithms. *arXiv preprint arXiv:2103.06064*, 2021.

Kun Yuan, Qing Ling, and Wotao Yin. On the convergence of decentralized gradient descent. *SIAM Journal on Optimization*, 26(3):1835–1854, 2016.

Meiqi Zhu, Xiao Wang, Chuan Shi, Houye Ji, and Peng Cui. Interpreting and unifying graph neural networks with an optimization framework. In *The Web Conference*, pages 1215–1226, 2021.

Minghui Zhu and Sonia Martínez. Discrete-time dynamic average consensus. *Automatica*, 46(2):322–329, 2010.

Consensus Optimization Graph Neural Networks (Supplementary Material)

Olga Solodova[1] **Ryan P. Adams**[1]

[1]Department of Computer Science, Princeton University, Princeton, New Jersey, USA

A REDUCING TRAINING COST IN IMPLICIT GNNS

Several works have proposed approaches to reducing the cost of training implicit GNNs. These focus on fixed-point GNNs, which are the subclass of implicit GNNs that have gained the most attention from the community and are most widely used in practice. EIGNN [Liu et al., 2021a] propose a fixed-point GNN architecture which admits a closed-form solution for the forward pass via eigendecomposition of the graph adjacency matrix, and leverages the same eigendecomposition to compute implicit gradients efficiently during the backward pass. However, as the eigendecomposition costs $O(n^3)$, this approach is best suited to small and medium-sized graphs; for larger graphs, approximate truncated eigendecompositions are required, introducing a small approximation error in exchange for tractability. Other work focuses on developing mini-batch training strategies and more efficient solvers for fixed-point GNNs; Liu et al. [2024] propose a mini-batch training method and a stochastic solver to obtain unbiased approximate solutions with fewer iterations, enabling training on larger graphs; and Chen et al. [2022] uses the latest equilibrium of in-batch nodes and their 1-hop neighbors to warm-start and accelerate the root-finding process in mini-batch training.

Our work takes a fundamentally different approach: for graph-level prediction tasks, we change the structure of the underlying optimization problem to operate over a single embedding, such that the dimension of the optimization variable is independent of graph size. This means the cost of the forward pass and backward pass is independent of graph size, in contrast to other implicit GNNs which operate over node embeddings. This removes the need for specialized solvers or training strategies, since the forward pass optimization problem and backward pass linear system that arises as a result of using implicit differentiation are both low dimensional and can be solved efficiently with standard methods.

B ENERGY GNN ARCHITECTURE: PARAMETERIZING E_θ USING PARTIALLY-INPUT CONVEX NEURAL NETWORKS

B.1 PARTIALLY INPUT-CONVEX NEURAL NETWORKS

PICNNs [Amos et al., 2017] are neural networks with inputs $\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{y} \in \mathbb{R}^m$, where weights are constrained such that the network is convex in $\boldsymbol{y}$. An L-layer PICNN $f_\theta(\boldsymbol{x}, \boldsymbol{y})$ is defined by the following recurrences:

$$\boldsymbol{u}_{\ell+1} = \tilde{g}_\ell \left(\tilde{\boldsymbol{W}}_\ell \boldsymbol{u}_\ell + \tilde{\boldsymbol{b}}_\ell \right) \tag{13}$$

$$\boldsymbol{z}_{\ell+1} = g_\ell \left(\boldsymbol{W}_\ell^{(z)} (\boldsymbol{z}_\ell \circ [\boldsymbol{W}_\ell^{(zu)} \boldsymbol{u}_\ell + \boldsymbol{b}_\ell^{(z)}]_+) + \boldsymbol{W}_\ell^{(y)} (\boldsymbol{y} \circ (\boldsymbol{W}_\ell^{(yu)} \boldsymbol{u}_\ell + \boldsymbol{b}_\ell^{(y)})) + \boldsymbol{W}_\ell^{(u)} \boldsymbol{u}_\ell + \boldsymbol{b}_\ell \right) \tag{14}$$

$$f_\theta(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{z}_L, \quad \boldsymbol{u}_0 = \boldsymbol{x}, \quad \boldsymbol{z}_0 = \boldsymbol{0} \tag{15}$$

Provided the $\boldsymbol{W}^{(z)}$ are elementwise nonnegative for all layers ℓ, and the activation functions g_ℓ are non-decreasing in each argument, it follows that f_θ is convex in $\boldsymbol{y}$.

B.2 ENERGY GNN ARCHITECTURE

In the energy GNN architecture [Solodova et al., 2025], E_θ is comprised of two PICNNs. In particular,

$$E_\theta(\mathcal{G}, \boldsymbol{H}) = \textstyle\sum_{i=1}^n e_\theta^i \qquad \text{where} \tag{16}$$

$$e_\theta^i = u_{\theta_u}(\boldsymbol{m}_i, \boldsymbol{h}_i, \boldsymbol{x}_i) + (\beta/2)||\boldsymbol{h}_i||_2^2 \tag{17}$$

$$\boldsymbol{m}_i = \textstyle\sum_{j \in \mathcal{N}_i} m_{\theta_m}(\boldsymbol{h}_i, \boldsymbol{h}_j, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{e}_{ij}) \tag{18}$$

where the message function m_{θ_m} in 18 corresponds to a PICNN which has node embeddings $\boldsymbol{h}_i, \boldsymbol{h}_j$ as its convex inputs (all other features are non-convex inputs), and function u_{θ_u} in 17 corresponds to a PICNN which is convex in $\boldsymbol{h}_i$ and $\boldsymbol{m}_i$, and nonconvex in the node features. The aggregate message $\boldsymbol{m}_i$ enters u_{θ_u} by initializing $\boldsymbol{z}_0 = \boldsymbol{m}_i$ in the PICNN for u_{θ_u} (instead of initializing $\boldsymbol{z}_0 = \boldsymbol{0}$ as in (15)). The composition of these functions is convex; therefore E_θ (which is a sum of convex functions) is convex in $\boldsymbol{H}$. The aggregation in equation 18 is only constrained to be a non-negative sum over neighbors, and therefore can be replaced with, e.g., a mean or a sum weighted by the entries of the symmetric renormalized adjacency matrix. The neighbor aggregation can also incorporate an attention mechanism, where neighbor contributions to the sum in equation 18 are scaled by neighbor-specific attention weights computed as a function of any of the non-convex inputs.

This architecture for E_θ offers significantly more flexibility than other instances of optimization GNNs [Zhu et al., 2021, Ma et al., 2021], which define E_θ as

$$E_\theta(\mathcal{G}, \boldsymbol{H}) = \gamma||\boldsymbol{H} - g_\theta(\boldsymbol{X})||_F^2 + \beta \text{tr}(\boldsymbol{H}^T \boldsymbol{L} \boldsymbol{H}), \tag{19}$$

where $g_\theta : \mathbb{R}^p \to \mathbb{R}^k$ and is applied independently to each node feature, $\boldsymbol{L}$ is a generalized incidence matrix (assumed to be symmetric and positive semi-definite with same sparsity as the adjacency matrix $\boldsymbol{A}$, e.g. the graph Laplacian), γ and β are constants, and tr() is the trace. Solodova et al. [2025] demonstrate empirically that the energy GNN architecture achieves stronger performance across a range of tasks compared to using E_θ as defined above.

C DECENTRALIZED INFERENCE UNDER PARTIAL ASYNCHRONY

C.1 PARTIAL ASYNCHRONY

Under asynchronous execution, nodes perform embedding updates at different times and using possibly stale information from neighbors. Partial asynchronism as defined by Bertsekas and Tsitsiklis [1989] is a commonly used model of asynchrony in distributed computing, which we summarize here.

Let the set $T^i \subseteq \{0, 1, 2, \dots\}$ denote times at which each node i is updated from the perspective of a global clock. For each $t \in T^i$ let $0 \leq \tau_j^i(t) \leq t$ denote the time associated with node i's view of node j at time t. Partial asynchronism places bounds on the time between updates for each node and on the staleness of information from neighbors, and assumes each node i maintains the most up to date version of its own state. These assumptions are formalized below.

Assumption C.1 (Partial Asynchronism)**.** *There exists an integer $B > 0$ such that:*

1. *(Bounded time to next update) For every node i and for every $t \geq 0$, at least one of the elements of the set $\{t, t+1, \dots, t+B-1\}$ belongs to T^i.*
2. *(Bounded staleness) $t - B < \tau_j^i(t) \leq t$, for all i, j, and all $t \geq 0$ belonging to T^i.*
3. *$\tau_i^i(t) = t$ for all $i = 1, 2, \dots, n$ and $t \in T^i$.*

C.2 COMPUTING NODE EMBEDDINGS IN OPTIMIZATION GNNS UNDER PARTIAL ASYNCHRONY

Solodova et al. [2025] assume the formal model of partial asynchronism above in proving convergence of node embeddings to the solution of the optimization problem in (1) under decentralized and asynchronous inference. Re-writing equations (8) and (9) to account for asynchronous execution under this model, we obtain that for a node i at time $t \in T^i$:

$$\boldsymbol{g}_{ij}(t) := \nabla_{\boldsymbol{h}_j} e_\theta^i(\boldsymbol{h}_i(t), \{\boldsymbol{h}_j(\tau_j^i(t)) | j \in \mathcal{N}_i\}) \tag{20}$$

$$\boldsymbol{h}_i(t+1) = \boldsymbol{h}_i(t) - \alpha \textstyle\sum_{j \in \mathcal{N}_i \cup \{i\}} \boldsymbol{g}_{ji}(\tau_j^i(t)) \tag{21}$$

At times $t \notin T^i$, node i sets $\boldsymbol{h}_i(t+1) = \boldsymbol{h}_i(t)$. In words, Equation 21 states that at time t node i uses the gradient $\boldsymbol{g}_{ji}(\tau_j^i(t)$ from node $j \in \mathcal{N}_i$ that was computed at time $\tau_j^i(t)$. In turn, when node j computed this gradient at time $\tau_j^i(t)$, it used embedding information obtained from its neighbors $j' \in \mathcal{N}_j$ at time $\tau_{j'}^j(\tau_j^i(t))$.

Proposition C.2. *If E_θ is strongly convex and separable per node, and is twice differentiable (w.r.t. $\boldsymbol{H}$) with a Hessian of bounded norm, then for a sufficiently small step size and the bounded staleness conditions in C.1, the optimization procedure of Equations 20 and 21 will converge to the solution of 1 when executed under partial asynchrony.*

We refer the reader to Solodova et al. [2025] for a detailed proof. Proposition C.2 uses the fact that with the addition of a penalty term $(\beta/2)||\boldsymbol{h}_i||_2^2$ to each node i's local objective e_θ^i in (2), the graph energy E_θ is strongly convex in $\boldsymbol{H}$.

D IMPLICIT DIFFERENTIATION

To avoid backpropagating through the optimization procedure in the forward pass, implicit GNNs compute derivatives by implicitly differentiating the optimality conditions of the optimization problem used to obtain node embeddings. At a nominal parameter value θ^*, let $\boldsymbol{z}^* \in \mathbb{R}^d$ denote the corresponding solution to the optimization problems (1) or (3), so that the first-order optimality condition is satisfied:

$$g(\boldsymbol{z}^*, \theta^*) = \frac{\partial E_{\theta^*}}{\partial \boldsymbol{z}}(\boldsymbol{z}^*) = \mathbf{0}. \tag{22}$$

Given an objective $\mathcal{L} : \mathbb{R}^p \mapsto \mathbb{R}$, the desired quantity is the total derivative of $\mathcal{L}$ with respect to the parameters θ. We can define a local solution mapping $h^*(\theta) = \boldsymbol{z}$ to obtain the following expression for the total derivative, with all quantities evaluated at $(\theta^*, \boldsymbol{z}^*)$:

$$\frac{d\mathcal{L}}{d\theta} = \frac{\partial \mathcal{L}}{\partial h^*}\frac{dh^*}{d\theta} + \frac{\partial \mathcal{L}}{\partial \theta}. \tag{23}$$

We compute $\frac{\partial \mathcal{L}}{\partial h^*}$ and $\frac{\partial \mathcal{L}}{\partial \theta}$ using normal automatic differentiation, and the solution Jacobian $\frac{dh^*}{d\theta}$ using implicit differentiation. At the solution $\boldsymbol{z}^*$, we have:

$$\frac{d}{d\theta} g(h^*(\theta), \theta) = \mathbf{0} \tag{24}$$

$$\frac{\partial g}{\partial h^*}\frac{dh^*}{d\theta} + \frac{\partial g}{\partial \theta} = \mathbf{0} \tag{25}$$

$$\frac{\partial g}{\partial h^*}\frac{dh^*}{d\theta} = -\frac{\partial g}{\partial \theta}. \tag{26}$$

Provided $\frac{\partial g}{\partial h^*}$ is invertible, we can rewrite the solution Jacobian as:

$$\frac{dh^*}{d\theta} = -(\frac{\partial g}{\partial h^*})^{-1}\frac{\partial g}{\partial \theta}, \tag{27}$$

and substitute this expression into (23) as follows:

$$\frac{d\mathcal{L}}{d\theta} = -\frac{\partial \mathcal{L}}{\partial h^*}(\frac{\partial g}{\partial h^*})^{-1}\frac{\partial g}{\partial \theta} + \frac{\partial \mathcal{L}}{\partial \theta}. \tag{28}$$

Rather than computing the inverse $(\frac{\partial g}{\partial h^*})^{-1}$ explicitly, we solve the linear system

$$\frac{\partial g}{\partial h^*}^T \lambda = -\frac{\partial \mathcal{L}}{\partial h^*}^T \tag{29}$$

and substitute the variable λ into (28). Note that $\frac{\partial g}{\partial h^*}$ is the Hessian of E_θ evaluated at the solution embeddings; the size of the linear system above is $nk \times nk$ when using per-node embedding case, compared to $k \times k$ when using shared consensus embeddings.

E METHODS FOR DECENTRALIZED CONSENSUS AVERAGES AND CONSENSUS OPTIMIZATION

E.1 PERTURBED PUSH-SUM FOR CONSENSUS AVERAGES

A widely used method for computing averages in a decentralized manner is the push-sum algorithm [Bénézit et al., 2010]; the *perturbed* push-sum algorithm is tailored for cases when the values being averaged are time-varying Nedić and Olshevsky [2015]. These methods make use of column stochastic averaging matrices, in contrast to the dynamic average consensus protocol discussed in Section 6 which assumes doubly stochastic matrices. Under partially asynchronous execution of average consensus protocols which assume doubly stochastic averaging matrix, the effective averaging matrix used at time t may not be doubly stochastic, and each node's local estimate of the mean is not guaranteed to converge to the true mean. In contrast, column stochasticity can be enforced during partially asynchronous execution, since it requires only that each node normalizes its outgoing messages. This relaxation on the structure of the averaging matrix allows push-sum protocols to be executed under partial asynchrony while retaining convergence guarantees.

Let $\boldsymbol{P}(t) \in \mathbb{R}^{nxn}$ denote the column-stochastic averaging matrix used by the protocol at time t, and $\mathcal{N}_i(t)$ denote the set of neighbors of node i communicates with at time t. A common choice is to use $\boldsymbol{P}(t)$ defined as

$$\boldsymbol{P}_{ij}(t) = \begin{cases} 1/|\mathcal{N}_j(t) \cup \{j\}| & \text{if } i \in \mathcal{N}_j(t) \cup \{j\}, \\ 0 & \text{otherwise.} \end{cases} \tag{30}$$

so that each node j distributes its value uniformly to its neighbors (including itself). In the general push-sum protocol, the mixing matrix $\boldsymbol{P}(t)$ may vary with time and is defined over a directed communication graph.

In executing the synchronous perturbed push-sum protocol, each agent i maintains variables $\boldsymbol{x}_i(t) \in \mathbb{R}^k$, $\boldsymbol{w}_i(t) \in \mathbb{R}^k$, $\boldsymbol{z}_i(t) \in \mathbb{R}^k$, and $y_i(t) \in \mathbb{R}$. At time $t = 0$, each node i initializes $\boldsymbol{h}_i(0)$ arbitrarily and sets $\boldsymbol{x}_i(0) = \boldsymbol{h}_i(0)$, $y_i(0) = 1$. When node i updates at time t, it performs the following updates:

$$\begin{aligned} \boldsymbol{w}_i(t+1) &= \sum_{j \in \mathcal{N}_i \cup \{i\}} \boldsymbol{P}_{ij}(t) \boldsymbol{x}_j(t) \\ y_i(t+1) &= \sum_{j \in \mathcal{N}_i \cup \{i\}} \boldsymbol{P}_{ij}(t) y_j(t) \\ \boldsymbol{z}_i(t+1) &= \frac{\boldsymbol{w}_i(t+1)}{y_i(t+1)} \\ \boldsymbol{x}_i(t+1) &= \boldsymbol{w}_i(t+1) + \epsilon_i(t+1) \end{aligned} \tag{31}$$

where $\epsilon_i(t) \in \mathbb{R}^k$ summarizes a local change in node i's value at time $t+1$ and the values $\boldsymbol{P}_{ij}(t)\boldsymbol{x}_j(t)$ and $\boldsymbol{P}_{ij}(t)y_j(t)$ are obtained from neighbors. Intuitively, $\boldsymbol{w}_i(t)$ and $y_i(t)$ track weighted sums, while their ratio $\boldsymbol{z}_i(t)$ provides a local estimate of the global average. If perturbations eventually go to zero, i.e.,

$$\lim_{t \to \infty} \epsilon_i(t) = \mathbf{0} \qquad \forall i \in \mathcal{V}, \tag{32}$$

then the estimates $\boldsymbol{z}_i(t)$ converge to the average of the steady-state values $\boldsymbol{h}_i(t)$:

$$\lim_{t \to \infty} ||\boldsymbol{z}_i(t) - 1/n \sum_{k=1}^{n} \boldsymbol{h}_k(t)||_1 = 0 \qquad \forall i \in \mathcal{V}. \tag{33}$$

This result holds under both partial asynchrony and communication delays (corresponding to Assumption C.1), provided that the underlying graph is strongly connected. Moreover, this algorithm results in geometric convergence of local estimates $\boldsymbol{z}_i(t)$ to the true mean. We refer the reader to [Assran and Rabbat, 2021] for more details and convergence analysis.

In using the protocol to compute the average of node embeddings in optimization GNNs, the perturbation $\epsilon_i(t)$ is attributed to node i's local gradient step at time t (i.e. the step defined in $refeqn : g_{iu}pdate)) : Since\boldsymbol{H}(t)$ converges to the minimizer $\boldsymbol{H}^*$ (by Proposition C.2), the estimate $\boldsymbol{z}_i(t)$ held by each node i eventually converges to the mean of the optimized embeddings $1/n \sum_{i=1}^{n} \boldsymbol{h}_i^*$.

To build intuition for push-sum, consider the case where perturbations $\epsilon(t) = \mathbf{0}$, and the mixing matrix $\boldsymbol{P}$ is time-invariant, doubly stochastic, non-negative and aperiodic (e.g. using $\boldsymbol{P} = \boldsymbol{W}$ where $\boldsymbol{W}$ is the averaging matrix defined in (**??**)). In this setting, we have that $\lim_{t\to\infty} \boldsymbol{P}^t = {}^1/_n \mathbf{1}\mathbf{1}^T$, and thus the iteration $\boldsymbol{x}(t+1) = \boldsymbol{P}\boldsymbol{x}(t)$ converges directly to the average of the initial values ${}^1/_n \mathbf{1}\mathbf{1}^T \boldsymbol{x}(0)$. In contrast, when $\boldsymbol{P}$ is only column stochastic, the iteration converges to a weighted average determined by the stationary distribution π of $\boldsymbol{P}$, i.e. $\lim_{t\to\infty} \boldsymbol{P}^t = \mathbf{1}\pi^T$. Push-sum corrects this bias by evolving an auxiliary variable $y(t)$ with the same linear dynamics and normalizing by it, thereby canceling the unknown weights and recovering the exact average.

E.2 OTHER APPROACHES FOR POOLING NODE EMBEDDINGS

Beyond the mean, other aggregates of node embeddings $\boldsymbol{h}_i$ can also be computed in the decentralized asynchronous setting. We first note that since node embeddings change during optimization of E_θ, algorithms designed for aggregation of *fixed* values are not directly applicable without using a termination detection protocol. Distributed termination detection protocols require additional coordination overhead; moreover, at deployment the underlying optimization problem E_θ may change over time (e.g., due to new observations), meaning the protocol would need to be repeatedly restarted.

Instead, we require algorithms that continuously track the aggregate as the embeddings change, analogously to dynamic average consensus and perturbed push sum. A sufficient condition for an aggregate to be trackable via push-sum-based protocols is that it decomposes as a sum of per-node terms, i.e., it takes the form ${}^1/_n \sum_{i=1}^n f(\boldsymbol{h}_i^t)$ for some function f applied locally at each node. The key property is the linearity of the aggregation operator (not of f itself, which may be arbitrarily nonlinear) since perturbations to any single node's value contribute additively to the global aggregate and can therefore be tracked incrementally via the push-sum mechanism. Consequently, any statistic expressible as a combination of averages (e.g. polynomial moments of any order or higher-order cumulants such as variance) is trackable in this setting.

In contrast, non-linear aggregates such as the maximum, minimum, or median do not admit such an additive decomposition. For example, max-tracking is feasible only when values are monotonically non-decreasing — which is not the case during gradient-based optimization of embeddings — since previously broadcast maxima are never invalidated. The same difficulty applies to other order-dependent statistics more generally. Aggregates decomposable as means of locally-computed features are therefore the most natural fit for the decentralized asynchronous setting.

E.3 GRADIENT-PUSH FOR CONSENSUS OPTIMIZATION

A well-studied method for solving the general consensus optimization problem in (5) is the gradient-push method [Nedić and Olshevsky, 2015], which can be viewed as the perturbed push-sum algorithm discussed in Appendix E.1 with particular choice for perturbations $\epsilon_i(t)$. Unlike the DGD algorithm discussed in 5.2, this algorithm is applicable in settings where the communication graph is time-varying and communication links are directed.

As in the push-sum protocol, each agent i maintains variables $\boldsymbol{x}_i(t) \in \mathbb{R}^k$, $\boldsymbol{w}_i(t) \in \mathbb{R}^k$, $\boldsymbol{z}_i(t) \in \mathbb{R}^k$, and $y_i(t) \in \mathbb{R}$. Embeddings $\boldsymbol{h}_i(0) \in \mathbb{R}^k$ are initialized arbitrarily, and we set $y_i(0) = 1$ and $\boldsymbol{x}_i(0) = \boldsymbol{h}_i(0)$ for all agents i. Let $\boldsymbol{P}(t)$ be a column-stochastic mixing matrix as defined in (30). At time $t \in T^i$, node i performs the following update:

$$
\begin{aligned}
\boldsymbol{w}_i(t+1) &= \sum_{j \in \mathcal{N}_i \cup \{i\}} \boldsymbol{P}_{ij}(t) \boldsymbol{x}_j(t) \\
y_i(t+1) &= \sum_{j \in \mathcal{N}_i \cup \{i\}} \boldsymbol{P}_{ij}(t) y_j(t) \\
\boldsymbol{z}_i(t+1) &= \frac{\boldsymbol{w}_i(t+1)}{y_i(t+1)} \\
\boldsymbol{x}_i(t+1) &= \boldsymbol{w}_i(t+1) - \alpha \nabla_{\boldsymbol{h}} e_\theta^i(\boldsymbol{z}_i(t+1))
\end{aligned} \tag{34}
$$

where $\boldsymbol{z}_i(t)$ serves as node i's current estimate of the global minimizer $\boldsymbol{h}^*$. Assuming the functions e_θ^i have Lipschitz continuous gradients and are strongly convex, and the underlying graph $\mathcal{G}$ is strongly connected, the local estimates $\boldsymbol{z}_i(t)$ held by all nodes converge to the minimizer $\boldsymbol{h}^*$ of the global objective E_θ when using diminishing step sizes Nedić and Olshevsky [2015]. Under the same assumptions, local estimates $\boldsymbol{z}_i(t)$ converge to an unbiased global minimizer $\boldsymbol{h}^*$ of E_θ when the protocol is executed under partial asynchrony (corresponding to Assumption C.1). We refer the reader to Assran and Rabbat [2021] for a detailed proof.

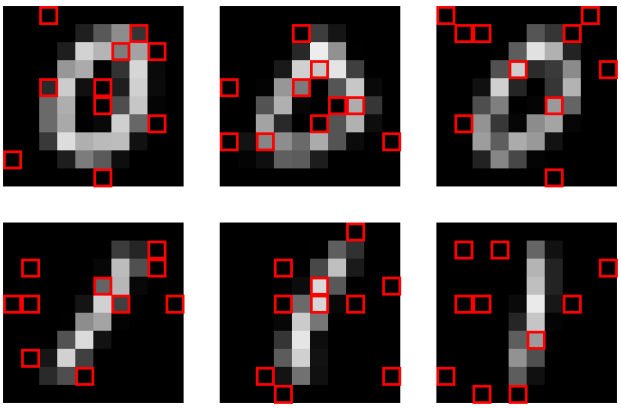

Figure 3: Example graphs for mnist-terrain experiment. Highlighted cells are those occupied by nodes. Edges (not depicted) connect nodes within 4 pixels of one another.

F EXPERIMENTAL DETAILS

F.1 GNN BENCHMARK DATASETS

The benchmark datasets we report performance for are MUTAG, PROTEINS, COX2, PTC, NCI1, peptides-func, and ogbg-molhiv for graph classification and peptides-struct for graph regression. We summarize these datasets below.

TU Datasets MUTAG, PROTEINS, COX2, PTC, and NCI1 are datasets from the TUDataset collection Morris et al. [2020]. In PROTEINS each graph represents a protein, where nodes correspond to secondary structural elements and are connected by an edge if they are within a threshold distance of one another. Nodes are associated with one-hot encoded features corresponding to one of 3 possible secondary structures. In MTUAG, COX2, PTC, and NCI1 each graph represents a small molecule, where nodes correspond to atoms and edges correspond to bonds. Nodes are associated with one-hot encoded features corresponding to the atom type, and in MUTAG and PTC there are additionally one-hot encoded edge features corresponding to the bond type. Each of these datasets correspond to binary graph classification tasks where the binary target denotes whether the molecule exhibits a binary property.

Peptides-func & Peptides-struct The peptides-func and peptides-struct datasets are from the Long-Range Graph Benchmark dataset collection [Dwivedi et al., 2022] and consist of the same 15535 graphs, with different prediction targets. Each graph corresponds to a peptide; nodes correspond to heavy atoms (and are associated with 9 categorical node features representing atom properties) and edges correspond to bonds (and are associated with 3 categorical edge features representing bond properties). For peptides-func, the prediction task is multi-label graph classification (10 classes) of the peptide function. For peptides-struct, the prediction task is graph regression of various peptide properties (11 regression targets).

ogbg-molhiv ogbg-molhiv is a dataset from the Open Graph Benchmark collection [Hu et al., 2020], where each graph corresponds to a molecule and binary prediction targets correspond to whether the molecule inhibits HIV replication. Nodes correspond to atoms (and are associated with 9 categorical features representing atom type and properties) and edges correspond to bonds (and are associated with 3 categorical edge features representing bond properties).

F.2 MNIST "TERRAIN" DATASET

For the MNIST "Terrain" dataset, we follow Solodova et al. [2025] in constructing the dataset. We resize binary MNIST images to 10x10 pixels using OpenCV's cv2.INTER_AREA interpolation, and normalize pixel values to be in $[0, 1]$. For each image, we sample a graph by sampling $n = 10$ pixel coordinates in the image for nodes, and connect nodes within 4 pixels of one another. Node features consist of the coordinates and value of the pixel occupied by each node, and edge features are distances between nodes. The prediction target for each graph corresponds to the digit of the underlying image. The dataset consists of a total of 14.780 graphs, with 12.665 graphs for training and 2115 for testing (the test partition is consistent with graphs in the test set in the standard MNIST dataset). Additional examples of graphs from this dataset are

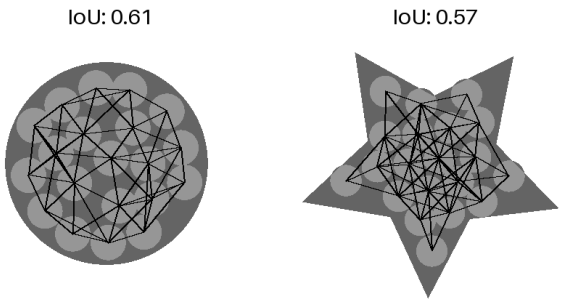

Figure 4: Example high IoU graphs for IoU experiment with circle target shape (left) and star target shape (right). Prediction targets are shown above the graph.

depicted in Figure 3.

F.3 IOU DATASET

We generate IoU datasets by sampling connected graphs in the unit square . Nodes are connected by an edge if they are within communication distance $r_{comm} = 4r_{extent}$, where r_{extent} denotes the radius of a disk defining each node's spatial extent. We define the graph-induced shape as the union of these disks. Node features are absent; edge features are inter-node distances.

Prediction targets are IoUs between this union shape and a fixed target (circle or 5-pointed star). Since nodes lack global coordinates, IoU is computed modulo rigid transformations. For the circle, this reduces to centering at the mean position; for the star, we additionally maximize over rotations. The target scale is chosen so that nodes can approximately fit within the shape without significant overlap.

To obtain a broad range of IoU values, we discretize the IoU interval and populate bins with a roughly interpolated uniform–Gaussian distribution. To generate graphs with high IoU, we render the nodes on a rasterized image of the target shape and approximately maximize the IoU via gradient descent on node positions. An example configuration of a graph with a high IoU prediction target achieved in this manner is shown in 4 for each dataset.

Each dataset contains 50,000 graphs with 20 nodes, with 5.000 graphs reserved for testing.

F.4 ARCHITECTURE DETAILS

For both the consensus and non-consensus energy GNN, we use the same architecture for E_θ, except that for the non-consensus architecture, we concatenate the embeddings for neighboring nodes when computing messages. In particular, for the non-consensus architecture, we use

$$E_\theta(\mathcal{G}, \boldsymbol{H}) = \textstyle\sum_{i=1}^n e_\theta^i \qquad \text{where} \tag{35}$$

$$e_\theta^i = u_{\theta_u}(\boldsymbol{m}_i, \boldsymbol{h}_i, \boldsymbol{x}_i) + (\beta/2)||\boldsymbol{h}_i||_2^2 \tag{36}$$

$$\boldsymbol{m}_i = \textstyle\sum_{j \in \mathcal{N}_i} m_{\theta_m}([\boldsymbol{h}_i||\boldsymbol{h}_j], [\boldsymbol{x}_i||\boldsymbol{x}_j||\boldsymbol{e}_{ij}]) \tag{37}$$

while for the consensus architecture we use

$$E_\theta(\mathcal{G}, \boldsymbol{H}) = \textstyle\sum_{i=1}^n e_\theta^i \qquad \text{where} \tag{38}$$

$$e_\theta^i = u_{\theta_u}(\boldsymbol{m}_i, \boldsymbol{h}, \boldsymbol{x}_i) + (\beta/2)||\boldsymbol{h}||_2^2 \tag{39}$$

$$\boldsymbol{m}_i = \textstyle\sum_{j \in \mathcal{N}_i} m_{\theta_m}(\boldsymbol{h}, [\boldsymbol{x}_i||\boldsymbol{x}_j||\boldsymbol{e}_{ij}]) \tag{40}$$

with $||$ denoting concatenation. In both models we set $\beta = 0.05$ and use softplus activations in the PICNNs for m_{θ_m} and u_{θ_u}. We also use softplus for enforcing positivity in the PICNN (for weights operating on the convex inputs). We add a self-loop when computing messages $\boldsymbol{m}_i$, where the self-loop contribution is computed by an independently parameterized PICNN (instead of using the same parameters as for neighbors), which takes as input $\boldsymbol{h}_i, \boldsymbol{x}_i$ for the non-consensus architecture, and $\boldsymbol{h}, \boldsymbol{x}_i$ for the consensus architecture.

For the non-consensus architecture, we use the same g_ψ and compute final graph-level predictions via $\hat{\boldsymbol{y}} = g_\psi(1/n \sum_{i=1}^{n} f_\phi(\boldsymbol{h}_i^*))$ where f_ϕ is a linear readout applied to optimized node embeddings and g_ψ is a three-layer MLP with softplus activations. For the consensus architecture, we compute predictions as $\hat{\boldsymbol{y}} = g_\psi(\boldsymbol{h}^*)$. We report results for two model sizes in our experiments, operating over $k = 4$ and $k = 16$ dimensional embeddings, respectively. In both cases, f_ϕ is linear with dimension k, and g_ψ is a three layer MLP with layer sizes $(k, k,$ <output dimension>). When $k = 4$ the message PICNN m_{θ_m} layer sizes are $(4, 4, 4)$, and readout PICNN u_{θ_u} layer sizes are $(4, 4, 1)$. When $k = 16$ the message PICNN m_{θ_m} layer sizes are $(8, 8, 8)$, and readout PICNN u_{θ_u} layer sizes are $(8, 8, 1)$.

F.5 TRAINING DETAILS

For all experiments, we use Newton's method in the forward pass to minimize E_θ. We compare performance and time per epoch when using two choices of solvers for the linear systems in the forward and backward passes - a direct solver and conjugate gradient.

In the forward pass, we consider Newton's method to have converged if the gradient magnitude is less than 1e-4, and use a maximum of 10 iterations. When using conjugate gradient to solve the linear systems in the forward and backward passes, we likewise use a convergence tolerance of 1e-4, and use a maximum of 30 iterations.

For all experiments, we use Adam with weight decay regularization to optimize model parameters during training with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. We use an initial step size of 0.01 which is decayed exponentially with decay rate 0.98 every 300 gradient steps. We train for a maximum of 500 epochs and use early stopping with a patience of 100 epochs. For the datasets from the TUDataset collection (MUTAG, PROTEINS, COX2, PTC, NCI1) we use a batch size of 50, and for the remaining datasets we use a batch size of 100. All experiments are performed on a single NVIDIA RTX 2080 Ti GPU.

F.6 ADDITIONAL RESULTS

In Tables 10 and 12, we report peak-GPU memory for the benchmark GNN datasets. Peak memory is measured using nvidia-smi, so the reported values should be interpreted as coarse process-level peaks. However, they provide a consistent comparison across consensus and non-consensus runs. These results empirically validate the discussion and analysis in Section 4 and show that the consensus model requires significantly less memory. This is consistent with the embedding and solver-side dimensionality reduction from nk to k. Larger graphs still increase memory in both models because evaluating local objectives, gradients, and HVPs scales with the number of nodes and edges. However, consensus fares better when graph size increases, since it avoids increase in memory in the solvers. With consensus embeddings, the direct linear systems are only k-dimensional per graph, so direct and CG have similar memory footprints. For node-level embeddings, direct solve requires significantly more memory than CG since the solver requires $O(n)$ more memory to store and use the Hessian. We also intuitively observe that increasing the embedding dimension k is much more expensive for the non-consensus model: increasing k by one adds one degree of freedom per node, increasing the overall embedding dimension by n.

Table 7: Classification accuracy (%) for consensus vs non-consensus energy GNN on benchmark datasets, using two model sizes (small and large; see for architecture details) and direct solvers or conjugate gradient for linear systems in the forward and backward pass during training. Mean and standard deviations are computed over 10 folds of the dataset and 3 random parameter seeds. Entries denoted by 'OOM' indicate that we ran out of memory when attempting to train with the given solver. Note that entries corresponding to the small model are copied from the main text and are included for reference. We additionally include performance for several competitive baseline models - GCN [Kipf et al., 2017], GIN [Xu et al., 2019], GraphSAGE [Hamilton et al., 2017], WL subtree [Shervashidze et al., 2011], and IGNN Gu et al. [2020] - using values reported in Xu et al. [2019], Gu et al. [2020].

.

DATA SET	MUTAG	PROTEINS	COX2	PTC	NCI1
# GRAPHS	188	1113	467	344	4110
AVG # NODES	17.9	39.1	41.2	25.5	29.9
AVG # EDGES	19.8	72.8	43.4	14.7	32.3
NON-CONSENSUS SMALL (DIRECT)	93.2 ± 4.7	OOM	85.1 ± 5.3	68.5 ± 7.1	77.3 ± 2.0
CONSENSUS SMALL (DIRECT)	91.7 ± 7.4	77.9 ± 2.7	84.9 ± 3.8	69.8 ± 7.0	74.6 ± 1.8
NON-CONSENSUS SMALL (CG)	93.8 ± 4.5	77.4 ± 3.3	84.9 ± 4.9	67.9 ± 7.0	76.9 ± 2.0
CONSENSUS SMALL (CG)	92.5 ± 7.4	76.6 ± 3.1	85.0 ± 4.2	69.4 ± 6.8	73.3 ± 2.0
NON-CONSENSUS LARGE (DIRECT)	94.7 ± 4.7	OOM	88.3 ± 3.7	69.0 ± 5.8	78.6 ± 2.4
CONSENSUS LARGE (DIRECT)	94.0 ± 5.6	78.4 ± 3.0	87.3 ± 4.0	70.3 ± 7.4	77.8 ± 2.4
NON-CONSENSUS LARGE (CG)	93.8 ± 5.0	OOM	88.5 ± 4.0	69.2 ± 6.4	78.0 ± 2.0
CONSENSUS LARGE (CG)	93.5 ± 5.2	78.5 ± 2.9	86.6 ± 4.1	70.1 ± 7.1	76.7 ± 1.9
GCN (5 LAYER)	85.6 ± 5.8	76.0 ± 3.2	-	64.2 ± 4.3	80.2 ± 2.0
GRAPHSAGE (5 LAYER)	85.1 ± 7.6	75.9 ± 3.2	-	63.9 ± 7.7	77.7 ± 1.5
GIN (5 LAYER)	89.4 ± 5.6	76.2 ± 2.8	-	64.6 ± 7.0	82.7 ± 1.7
WL SUBTREE	90.4 ± 5.7	75.0 ± 3.1	-	59.9 ± 4.3	86.0 ± 1.8
IGNN (3 LAYER)	89.3 ± 6.7	77.7 ± 3.4	86.9 ± 4.0	70.1 ± 3.4	80.5 ± 1.9

Table 8: Task performance for consensus vs non-consensus GNN on benchmark datasets using two model sizes (small and large; see for architecture details) and direct solvers (for the consensus model) and conjugate gradient (for the non-consensus model) for linear systems in the forward and backward pass during training. We report ROC-AUC for ogbg-molhiv, AP for peptides-func, and MAE for peptides-struct, with mean and and standard deviations over 5 random parameter seeds. Entries denoted by 'OOM' indicate that we ran out of memory when attempting to train. Note that entries corresponding to the small model are copied from the main text and are included for reference. We additionally include performance for several competitive baseline models, including GCN [Kipf et al., 2017], GIN [Xu et al., 2019], and a fully connected transformer with Laplacian positional encodings. For the peptides datasets, we copy values reported in Dwivedi et al. [2022] (where the authors use $\approx$ 500K parameters for each model). For ogbg-molhiv, we copy values reported in Hu et al. [2020].

DATA SET	OGBG-MOLHIV	PEPTIDES-STRUCT	PEPTIDES-FUNC
# GRAPHS	41127	15535	15535
AVG # NODES	25.5	150.9	150.9
AVG # EDGES	27.5	153.7	153.7
CONSENSUS SMALL (DIRECT)	0.713 ± 0.037	0.317 ± 0.011	0.427 ± 0.002
NON-CONSENSUS SMALL (CG)	0.717 ± 0.016	0.316 ± 0.015	0.393 ± 0.016
CONSENSUS LARGE (DIRECT)	0.737 ± 0.012	0.284 ± 0.007	0.512 ± 0.025
NON-CONSENSUS LARGE (CG)	0.740 ± 0.032	OOM	OOM
GCN (2 LAYERS)	-	0.395 ± 0.002	0.457 ± 0.006
GIN (2 LAYERS)	-	0.388 ± 0.001	0.500 ± 0.004
GCN (5 LAYERS)	0.761 ± 0.010	0.350 ± 0.001	0.593 ± 0.002
GIN (5 LAYERS)	0.756 ± 0.014	0.355 ± 0.004	0.549 ± 0.008
TRANSFORMER + LAPLCACIANPE	-	0.253 ± 0.002	0.633 ± 0.013

Table 9: Raw time per epoch for consensus vs non-consensus energy GNN on benchmark datasets. Entries denoted by 'OOM' indicate that we ran out of memory when attempting to train the model.

DATA SET	MUTAG	PROTEINS	COX2	PTC	NCI1
NONCONSENSUS SMALL (DIRECT)	0.143 ± 0.006	OOM	0.623 ± 0.094	0.409 ± 0.074	26.4 ± 3.7
CONSENSUS SMALL (DIRECT)	0.113 ± 0.005	2.1 ± 0.3	0.269 ± 0.016	0.224 ± 0.013	3.7 ± 0.1
NONCONSENSUS SMALL (CG)	0.240 ± 0.029	21.3 ± 1.8	0.592 ± 0.151	0.631 ± 0.025	10.9 ± 1.3
CONSENSUS SMALL (CG)	0.149 ± 0.011	3.4 ± 0.4	0.347 ± 0.019	0.300 ± 0.029	4.6 ± 0.5
NONCONSENSUS LARGE (DIRECT)	0.250 ± 0.005	OOM	0.809 ± 0.089	0.498 ± 0.092	115 ± 8.5
CONSENSUS LARGE (DIRECT)	0.115 ± 0.006	4.5 ± 0.5	0.280 ± 0.016	0.230 ± 0.015	3.7 ± 0.3
NONCONSENSUS LARGE (CG)	0.268 ± 0.032	OOM	1.903 ± 0.060	1.922 ± 0.073	22.6 ± 4.8
CONSENSUS LARGE (CG)	0.167 ± 0.010	8.3 ± 1.9	0.446 ± 0.039	0.382 ± 0.041	7.7 ± 0.5

Table 10: Peak GPU memory (GB) for consensus vs non-consensus energy GNN on benchmark datasets. Entries denoted by 'OOM' indicate that we ran out of memory ($>$ 11GB) when attempting to train the model.

DATA SET	MUTAG	PROTEINS	COX2	PTC	NCI1
MAX # NODES	28	620	50	64	111
MAX # EDGES	33	2049	60	70	120
NONCONSENSUS SMALL (DIRECT)	0.45	OOM	1.22	1.22	3.28
CONSENSUS SMALL (DIRECT)	0.45	1.27	0.45	0.46	0.47
NONCONSENSUS SMALL (CG)	0.45	2.78	0.45	0.45	0.72
CONSENSUS SMALL (CG)	0.45	1.24	0.45	0.46	0.48
NONCONSENSUS LARGE (DIRECT)	1.22	OOM	3.27	4.29	4.29
CONSENSUS LARGE (DIRECT)	0.45	1.79	0.46	0.46	0.47
NONCONSENSUS LARGE (CG)	0.45	OOM	0.70	0.70	0.73
CONSENSUS LARGE (CG)	0.45	1.75	0.46	0.46	0.48

Table 11: Raw time per epoch for consensus vs non-consensus energy GNN on benchmark datasets. Entries denoted by 'OOM' indicate that we ran out of memory when attempting to train the model.

DATA SET	OGBG-MOLHIV	PEPTIDES-FUNC	PEPTIDES-STRUCT
NONCONSENSUS SMALL (CG)	171.2 ± 8.1	108.3 ± 12.8	96.7 ± 16.8
CONSENSUS SMALL (DIRECT)	21.8 ± 3.6	8.8 ± 1.7	9.5 ± 1.1
NONCONSENSUS LARGE (CG)	264.9 ± 14.6	OOM	OOM
CONSENSUS LARGE (DIRECT)	55.9 ± 2.1	17.7 ± 1.6	19.9 ± 2.2

Table 12: Peak GPU memory (GB) for consensus vs non-consensus energy GNN on benchmark datasets. Entries denoted by 'OOM' indicate that we ran out of memory ($>$ 11GB) when attempting to train the model.

DATA SET	OGBG-MOLHIV	PEPTIDES-FUNC	PEPTIDES-STRUCT
MAX # NODES	222	444	444
MAX # EDGES	240	907	907
NONCONSENSUS SMALL (CG)	1.23	2.77	2.77
CONSENSUS SMALL (DIRECT)	0.76	1.25	1.25
NONCONSENSUS LARGE (CG)	8.31	OOM	OOM
CONSENSUS LARGE (DIRECT)	1.78	3.29	3.29

Table 13: Raw time per epoch for consensus vs non-consensus GNN on the mnist-terrain and IoU datasets.

DATA SET	MNIST-TERRAIN	IOU (CIRCLE)	IOU (STAR)
NON-CONSENSUS (DIRECT)	7.2 ± 0.4	38.9 ± 4.5	37.9 ± 3.9
CONSENSUS (DIRECT)	5.3 ± 0.6	14.5 ± 0.8	15.8 ± 0.1