Nina Otter, Mason A Porter, Ulrike Tillmann, Peter Grindod, and Heather A Harrington. A roadmap for the computation of persistent homology. *EPJ Data Science*, 6(17), 2017.

Arkady Pikovsky, Jurgen Kurths, Michael Rosenblum, and Jürgen Kurths. *Synchronization: a universal concept in nonlinear sciences*. Number 12. Cambridge university press, 2003.

Yousef Saad and Henk A Van Der Vorst. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1-2):1–33, 2000.

Mrinal Sarkar and Neelima Gupte. Phase synchronization in the two-dimensional kuramoto model: Vortices and duality. *Physical Review E*, 103(3):032204, 2021.

Ivan Sosnovik and Ivan Oseledets. Neural networks for topology optimization. *Russian Journal of Numerical Analysis and Mathematical Modelling*, 34(4):215–223, 2019.

Salma Tarmoun, Guilherme Franca, Benjamin D Haeffele, and Rene Vidal. Understanding the dynamics of gradient flow in overparameterized linear models. In *International Conference on Machine Learning*, pp. 10153–10161. PMLR, 2021.

Tijmen Tieleman, Geoffrey Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *arXiv preprint arXiv:1812.04202*, 2018.

A EXTENDED DERIVATIONS

A.1 ESTIMATING $\overline{M}$

To apply equation 14 we need to estimate the value of $\overline{M}$ for different random samples for $\boldsymbol{w}(t)$. $\overline{M}$ can be written in terms of the moments of the random variable $\boldsymbol{W} = \{\boldsymbol{w}\}$ using the Taylor expansion of $\mathcal{L}$ around $\boldsymbol{w}_i = 0$ plugged into equation 3

$$\frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial \boldsymbol{w}_i} \approx \sum_{k=0}^{\infty} \frac{1}{k!} \left[\boldsymbol{w}^T \frac{\partial}{\partial \boldsymbol{v}} \right]^k \frac{\partial \mathcal{L}(\boldsymbol{v})}{\partial \boldsymbol{v}_i} \bigg|_{\boldsymbol{v} \to 0} \tag{21}$$

$$\overline{M}_{ij}(t) = \sum_{p,q=1}^{\infty} \frac{1}{p!q!} \sum_{\{i_a\}} \mathbb{E}_P \left[\boldsymbol{w}_{i_1} \dots \boldsymbol{w}_{i_{p+q}} \right] \frac{\partial^{p+1} \mathcal{L}}{\partial \boldsymbol{v}_{i_1} \dots \partial \boldsymbol{v}_{i_p} \partial \boldsymbol{v}_i} \frac{\partial^{q+1} \mathcal{L}}{\partial \boldsymbol{v}_{i_{p+1}} \dots \partial \boldsymbol{v}_{i_{p+q}} \partial \boldsymbol{v}_j} \bigg|_{\boldsymbol{v} \to 0} \tag{22}$$

where the sum over $\{i_a\}$ means all choices for the set of indices $i_1 \dots i_{p+q}$. Equation 22 states that we need to know the order $p+q$ moments $\mathbb{E}_P[\boldsymbol{w}_{i_1} \dots \boldsymbol{w}_{i_{p+q}}]$ of $\boldsymbol{W}$ to calculate $\overline{M}$. This is doable in some cases. For example, we can use a normal distribution is used to initialize $\boldsymbol{w}(t=0) \in \mathbb{R}^n$. The local minima of $\mathcal{L}$ reachable by GD need to be in a finite domain for $\boldsymbol{w}$. Therefore, we can always rescale and redefine $\boldsymbol{w}$ such that its domain satisfies $\|\boldsymbol{w}\|_2 \leq 1$. This way we can initialize with normalized vectors $\boldsymbol{w}(0)^T \boldsymbol{w}(0) = 1$ using $\sigma = 1/\sqrt{n}$ and get

$$\boldsymbol{w}_{(i)j}(0) = \mathcal{N}\left(0, n^{-1/2}\right), \qquad \mathbb{E}_P \left[\prod_{a=1}^{p} \boldsymbol{w}_{i_a} \right] = \delta_{i_1 \dots i_p} \frac{(p-1)!!}{n^{p/2}} \text{even}(p) \tag{23}$$

where $\delta_{i \dots j}$ is the Kronecker delta, $(p-1)!! = \prod_{k=0}^{[p/2]} (p-1-2k)$ and $\text{even}(p) = 1$ if p is even and 0 if it's odd. When the number of variables $n \gg 1$ is large, order $p > 2$ moments are suppressed by factors of $n^{-p/2}$. Plugging equation 23 into equation 22 and defining the Hessian $\mathcal{H}_{ij}(\boldsymbol{w}) \equiv \partial^2 \mathcal{L}(\boldsymbol{w}) / \partial \boldsymbol{w}_i \partial \boldsymbol{w}_j$ we have

$$\overline{M}_{ij}(t=0) = \frac{1}{n} \left[\mathcal{H}^2 \right]_{ij} \Big|_{\boldsymbol{w} \to 0} + O(n^{-2}), \tag{24}$$

which yields $\overline{K} \approx (\mathcal{H}/h_{max})^{-1}$ where h_{max} is the maximum eigenvalue of the Hessian at initialization (note, the $\sigma^2 = 1/n$ factor cancels out). Here the assumption is that the derivatives $\partial^p_{\boldsymbol{w}} \mathcal{L}$ are not n dependent after rescaling the domain such that $\|\boldsymbol{w}\| \leq 1$. In the experiments in this paper this condition is satisfied.

A.2 COMPUTATIONALLY EFFICIENT IMPLEMENTATION

Define $M_0 \equiv (1-\xi)M/m_{max}$ with $\xi \ll 1$. To get an $O(qn^2)$ approximation for $\kappa = M_0^{-1/4}$ we can take the first q terms in the binomial expansion

$$M_0^{-1/4} = [I + (M_0 - I)]^{-1/4} = \sum_{m=0}^{\infty} \frac{\prod_{i=0}^{m} (-1/4 - i)}{m!} (M_0 - I)^m \tag{25}$$

Since M_0 is positive semi-definite and its largest eigenvalue is one, the sum in equation 25 can be truncated after $q \sim O(1)$ terms. The first q terms of equation 25 can be implemented as a q layer GCN with aggregation function $f(M) = I - M_0$ and residual connections. We choose q to be as small as 1 or 2, as larger q may actually slow down the optimization. Note that computing $f(M)^q \theta$ is $O(qn^2)$ because we do not need to first compute $f(M)^q$ (which is $O(qn^3)$) and instead use $\boldsymbol{v}_{i+1} f(M) \boldsymbol{v}_i$ ($O(n^2)$) q times with $\boldsymbol{v}_1 = \theta$.

Lastly, to evaluate M_0 we need to estimate the leading eigenvalue m_{max}. We can use a similar expansion for approximating m_{max}. Given any vector $\boldsymbol{v} \in \mathbb{R}^n$ and using the spectral expansion we have

$$M^q \boldsymbol{v} = \sum_i m_i^q (\psi_i^T \boldsymbol{v}) \psi_i \tag{26}$$

Since M is positive semi-definite, for $q > 1$ the leading eigenvector ψ_{max} quickly dominates the sum and we have $M^q \boldsymbol{v} \approx m_{max}^q (\psi_{max} \boldsymbol{v}) \psi_{max}$. Additionally, using the generalized Perron-Frobenius theorem (Berman & Plemmons (1994)) , the components of the leading eigenvector ψ_{max} should be mostly positive. Therefore, a good choice for $\boldsymbol{v}$, which may be close to the actual ψ_{max}, is $\boldsymbol{v} = \mathbf{1}/\sqrt{n}$, where $\mathbf{1} = (1, \ldots, 1)$. This yields

$$\mathbf{1}^T M^p \mathbf{1} \approx m_{max}^p (\psi_i^T \mathbf{1})^2, \qquad\qquad m_{max}^{p-1} \approx \frac{\mathbf{1}^T M^p \mathbf{1}}{\mathbf{1}^T M \mathbf{1}} \tag{27}$$

This is a crude approximation, but it serves our goal by being computationally efficient. Note that, while computing M^q is $O(qn^3)$, because $M\mathbf{1}$ is $O(n^2)$, computing $M^q\mathbf{1}$ is $O(qn^2)$. Therefore, we choose $p = 2$ and write

$$m_{max} \approx \frac{\mathbf{1}^T M^2 \mathbf{1}}{\mathbf{1}^T M \mathbf{1}} = \frac{\sum_{ij} M_{ij}^2}{\sum_{ij} M_{ij}} \tag{28}$$

It is worth noting that if we think of M as the weighted adjacency matrix of a graph with n vertices, the vector component $D_i = [M\mathbf{1}]_i = \sum_j M_{ij}$ is the weighted degree of node i. Hence equation 28 states $M/m_{max} \approx \left(\sum_i D_i / \|D\|^2\right) M$. This is reminiscent of the graph diffusion operator $D^{-1}M$ and $D^{-1/2}MD^{-1/2}$ which are used as the aggregation functions in GCN (here $D_{ij} = D_i \delta_{ij}$ is the diagonal degree matrix).

Equation 9 decomposes the rate of convergence into a part M arising from the loss function and a part K capturing the effect of the neural architecture. To understand the interplay between these two matrices, we will first work out the toy example of convex optimization.

A.3 Connection with NTK

Note that, like M, the NTK K is positive-semi-definite because $K = BB^T$ with $B_{ia} \equiv \sqrt{\hat{\varepsilon}_a} \partial \boldsymbol{w}_i / \partial \theta_a$, and so for any vector $\boldsymbol{v} \in \mathbb{R}^n$ we have $\boldsymbol{v}^T K \boldsymbol{v} = \|\boldsymbol{v}^T B\|^2 \geq 0$. To see that K is indeed the NTK, note that in neural network for a supervised problem on dataset $\boldsymbol{Z} = \{z_i\}$ the neural network output $f(\theta; z_n)$ is a function of z_n. Therefore, the supervised NTK has two more indices $K_{ij,nm} = \sum_a \partial f_i(\theta; x_n)/\partial\theta_a \partial f_j(\theta; x_m)/\partial\theta_a$. However, in our problem the neural network $\boldsymbol{w}(\theta)$ is a separate reparametrization of the trainable parameters in $\mathcal{L}(\boldsymbol{w}; \boldsymbol{Z})$, which doesn't take the dataset $\boldsymbol{Z}$ as explicit input. Therefore, the NTK for $\boldsymbol{w}(\theta)$ does not have sample indices n, m. The dataset $\boldsymbol{Z}$ only appears explicitly in $\mathcal{L}(\boldsymbol{w}; \boldsymbol{Z})$, which includes a loss function and an architecture defining the model used to learn the system.

$M(\boldsymbol{w})$ is purely a function of the optimization loss function (and potential neural models in it) and the value of $\boldsymbol{w}$. In particular, at a given value of $\boldsymbol{w}$, $M(\boldsymbol{w})$ does not explicitly depend on the architecture of the NR $\boldsymbol{w}(\theta)$. $K(\theta)$, on the other hand, directly depends on $\boldsymbol{w}(\theta)$.

A.4 Reparameterization with GNN

Due to the non-convex nature of the optimization problem, reaching a good local optimum or an equilibrium state can be computationally expensive. The core idea of our approach to accelerate optimization is to parametrize the node features X as the output of a deep neural network.

Specifically, we start with a high-dimensional random embedding $G_0 \in \mathbb{R}^{N \times h_0}$. Then we apply GNN layers to the random embedding sequentially:

$$G_1 = \sigma(f(A) Z W^{(1)}), \quad G_2 = \sigma(f(A) G_1 W^{(2)}), \quad X = \sigma(GW + b) \tag{29}$$

where $f(A) = \tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2}$ is the propagation rule in the GNN Kipf & Welling (2016). Propagation rule is critical to the representation power of GNNs Dehmamy et al. (2019). Here we use the normalized adjacency matrix $\tilde{A} = A + I$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ is the degree matrix of $\tilde{A}$. The new embeddings $G_1 \in \mathbb{R}^{N \times h_1}$ and $G_2 \in \mathbb{R}^{N \times h_2}$ are reparametrized via GNNs. $W^{(1)}$ and $W^{(2)}$ are the trainable weight matrices. We concatenate G_0, G_1 and G_2 along the feature dimension to obtain the node embedding $G = [G_0, G_1, G_2] \in \mathbb{R}^{N \times h}$ where $h = h_0 + h_2 + h_2$.

Finally, we apply a fully connected network to project the h-dimensional embedding G to the original d dimension features X. We allow G_0 to be freely trainable and $W \in \mathbb{R}^{h \times d}$ is the projection

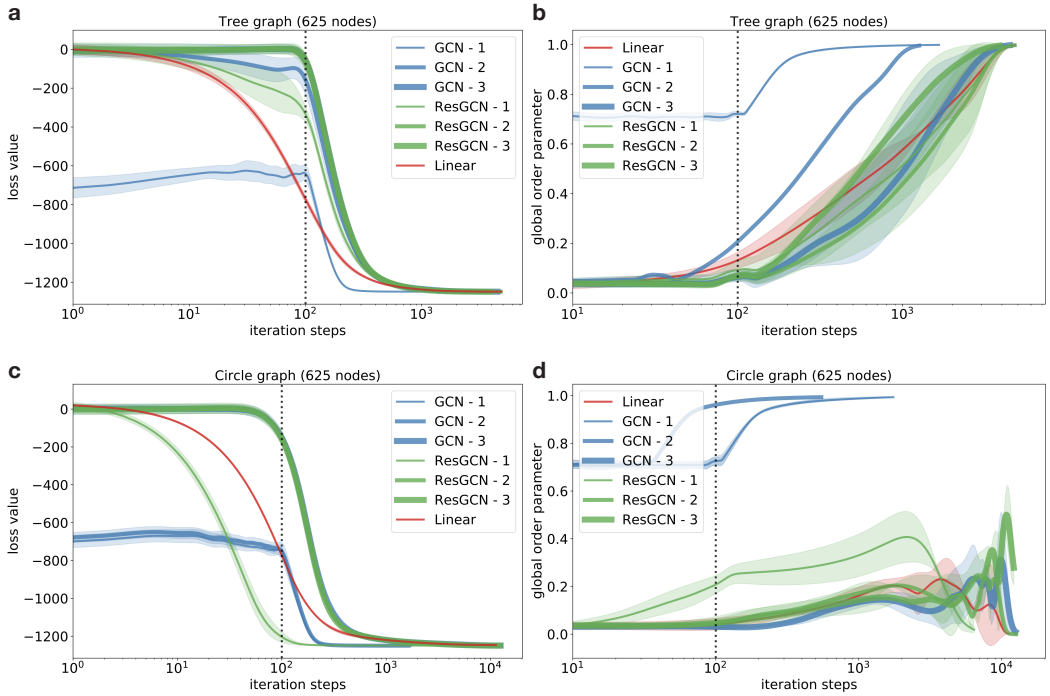

Figure 6: Kuramoto model on (a,b) tree and (c,d) circle graph. a,c) loss curves evolution, while b,d) global order parameter in each iteration steps

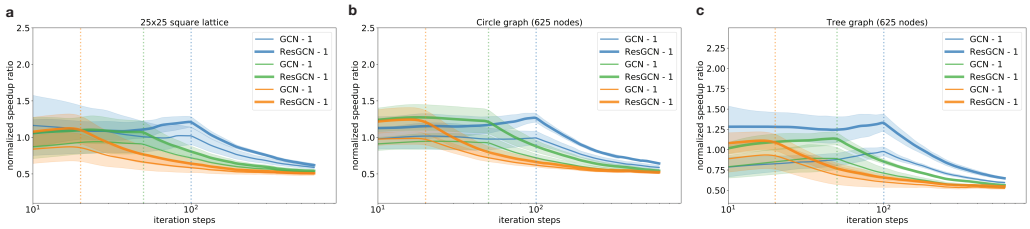

Figure 7: Kuramoto-model on different graph structures. We use the GCN model and switch to the Linear model after different iteration steps: orange - 20 steps, green - 50 steps, and blue - 100 steps. The plot shows that each GCN iteration step takes longer.

weight matrix. The set of trainable parameters is $\theta = \{G_0, W^{(1)}, W^{(2)}, W, b\}$. In this way, the original optimization problem over X becomes the optimization over θ. We initialize all the parameters θ randomly and optimize them through back-propagation.

B Experiment Details and Additional Results

B.1 Kuramoto Oscillator

Relation to the XY model The loss equation 17 is also identical to the Hamiltonian (energy function) of a the classical XY model (Kosterlitz & Thouless (1973)), a set of 2D spins s_i with interaction energy given by $\mathcal{L} = \sum_{i,j} A_{ij} s_i \cdot s_j = \sum_{i,j} A_{ji} \cos(\phi_j - \phi_i)$. In the XY model, we are also interested in the minima of the energy.

Method In our model, we first initialize random phases between 0 and 2π from a uniform distribution for each oscillator in a h dimensional space that results in $N \times h$ dimensional vector N is the number of oscillators. Then we use this vector as input to the GCN model, which applies

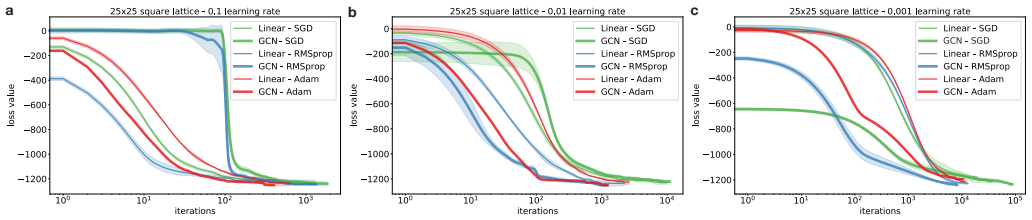

Figure 8: Kuramoto-model on 25×25 square lattice. We use the GCN and Linear model with different optimizers and learning rates. The Adam optimizer in all cases over-performs the other optimizers.

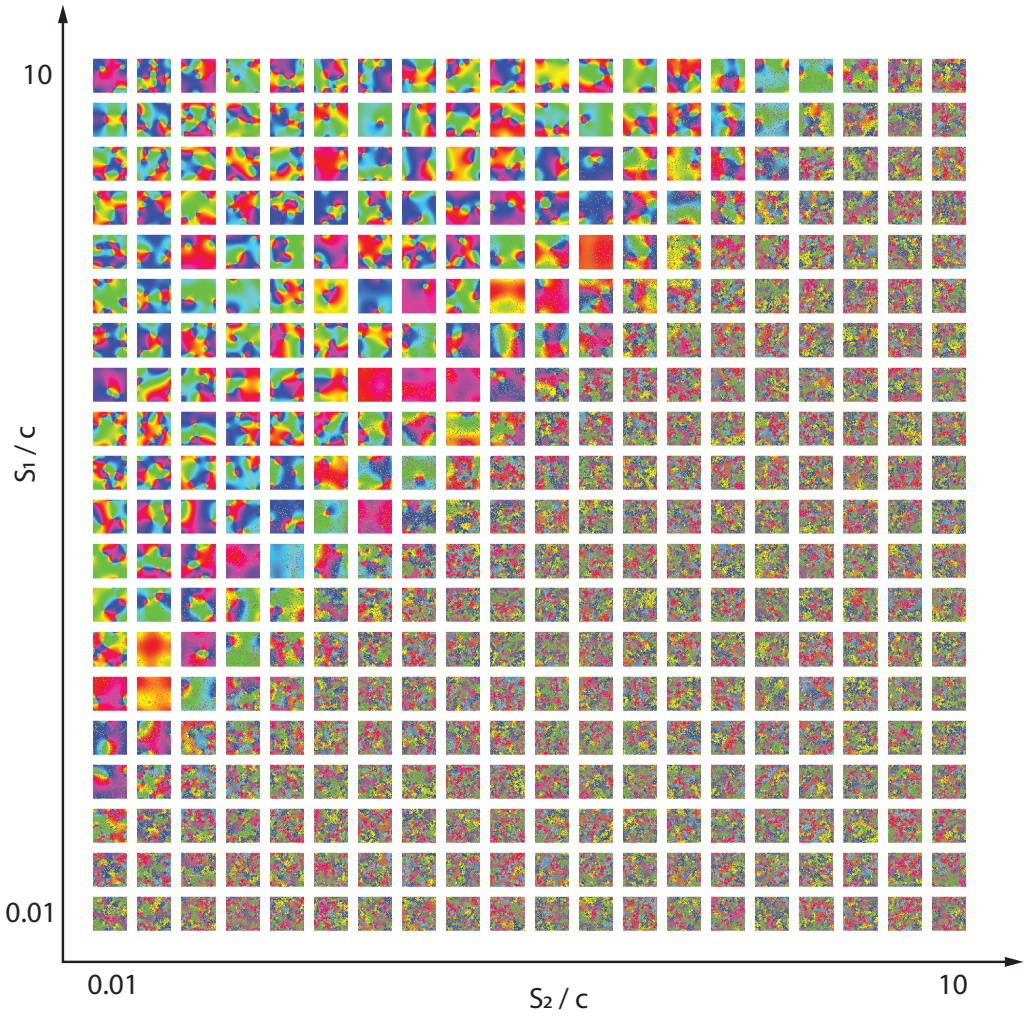

Figure 9: Hopf-Kuramoto model on a square lattice (50×50) - phase pattern. We can distinguish two main patterns - organized states are on the left part while disorganized states are on the right part of the figure. In the experiments $c = 1$ for the simplicity.

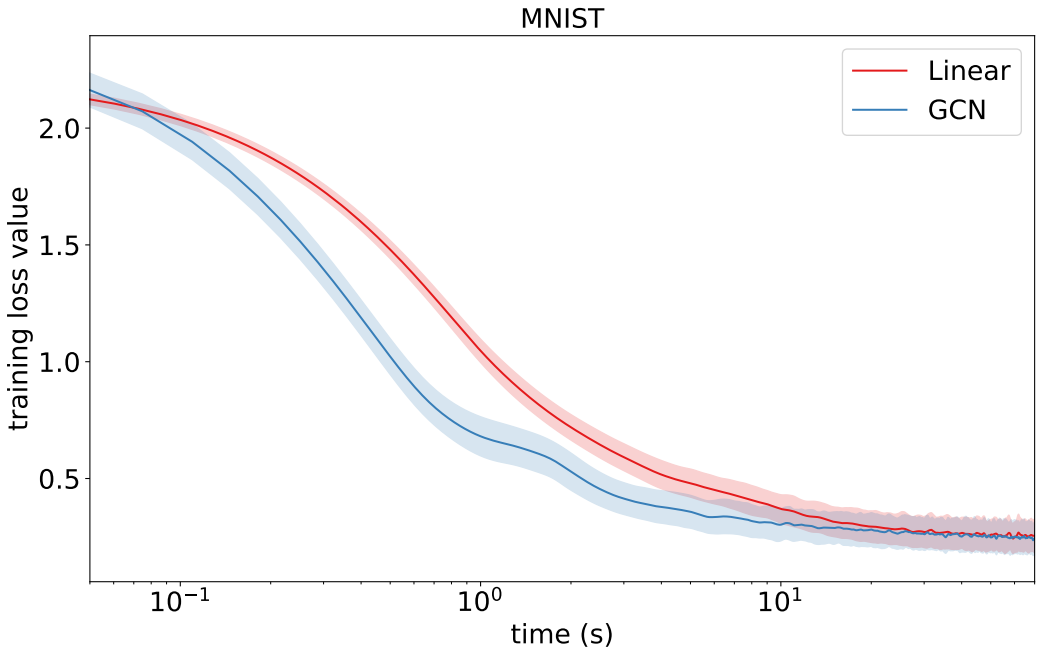

Figure 10: MNIST image classification. Red curve shows a linear model (92.68% accuracy). Blue curve is the reparametrized model, where at step 50 we switch from GCN to Linear model (92.71% accuracy).

$D^{-1/2}\hat{A}D^{-1/2}$, $\hat{A} = A + I$ propagation rule, with LeakyRelu activation. The final output dimension is $N \times 1$, where the elements are the phases of oscillators constrained between 0 and 2π. In all experiments for h hyperparameter we chose 10. Different h values for different graph sizes may give different results. Choosing large h reduces the speedup significantly. We used Adam optimizer by 0.01 learning rate.

B.2 MNIST IMAGE CLASSIFICATION

Here, we introduce our reparametrization model for image classification on the MNIST image dataset. First, we test a simple linear model as a baseline and compare the performance to the GCN model. We use a cross-entropy loss function, Adam optimizer with a 0.001 learning rate in the experiments, and Softmax nonlinear activation function in the GCN model. We train our models on 100 batch size and 20 epochs. In the GCN model, we build the $\boldsymbol{H}$ matrix (introduced in the eq. 15) from the covariance matrix of images and use it as a propagation rule. In the early stages of the optimization, we use the GCN model until a plateau appears on the loss curve then train the model further by a linear model. We found that the optimal GCN to linear model transition is around 50 iterations steps. Also, we discovered that wider GCN layers achieve better performance; thus, we chose 500 for the initially hidden dimension. According to the previous experiments, the GCN model, persistent homology (B.3), and Kuramoto (B.1) model speedups the convergence in the early stages (Fig. 10)

B.3 PERSISTENT HOMOLOGY

Overview. Homology describes the general characteristics of data in a metric space, and is categorized by the order of its features. Zero order features correspond to connected components, first order features have shapes like ”holes” and higher order features are described as ”voids”.

A practical way to compute homology of a topological space is through forming simplicial complexes from its points. This enables not only fast homology computation with linear algebra, but also approximating the topological space with its subsets.

In order to construct a simplicial complex, a filtration parameter is needed to specify the scope of connectivity. Intuitively, this defines the resolution of the homological features obtained. A feature is considered "persistent" if it exists across a wide range of filtration values. In order words, persistent homology seeks features that are scale-invariant, which serve as the best descriptors of the topological space.

There are different ways to build simplicial complexes from given data points and filtration values. Czech complex, the most classic model, guarantees approximation of a topological space with a subset of points. However, it is computationally heavy and thus rarely used in practice. Instead, other models like the Vietoris-Rips complex, which approximates the Czech complex, are preferred for their efficiency (Otter et al., 2017). Vietoris-Rips complex is also used in the point cloud optimization experiment of ours and Gabrielsson et al. (2020); Carriere et al. (2021).

Algorithm Implementation. Instead of optimizing the coordinates of the point cloud directly, we reparameterize the point cloud as the output of the GCN model. To optimize the network weights, we chose identity matrix with dimension of the point cloud size as the fixed input.

To apply GCN, we need the adjacency matrix of the point cloud. Even though the point cloud does not have any edges, we can manually generate edges by constructing a simplicial complex from it. The filtration value is chosen around the midpoint between the maximum and minimum of the feature birth filtration value of the initial random point cloud, which works well in practice.

Before the optimization process begins, we first fit the network to re-produce the initial random point cloud distribution. This is done by minimizing MSE loss on the network output and the regression target.

Then, we begin to optimize the output with the same loss function in Gabrielsson et al. (2020); Carriere et al. (2021), which consists of topological and distance penalties. The GCN model can significantly accelerates convergence at the start of training, but this effect diminishes quickly. Therefore, we switch the GCN to the linear model once the its acceleration slows down. We used this hybrid approach in all of our experiments.

Hyperparameter Tuning. We conducted extensive hyperparameter search to fine tune the GCN model, in terms of varying hidden dimensions, learning rates and optimizers. We chose the setting of 200 point cloud with range 2.0 for all the tuning experiments.

Fig. 11 shows the model convergence with different hidden dimensions. We see that loss converges faster with one layer of GCN instead of two. Also, convergence is delayed when the dimension of GCN becomes too large. Overall, one layer GCN model with $h1, h2 = 8, 6$ generally excels in performance, and is used in all other experiments.

Fig. 12,13,14 shows the performance of the GCN model with different prefit learning rates, train learning rates and optimizers. From the results, a lower prefit learning rate of 0.01 combined with a training learning rate below 0.01 generally converges to lower loss and yields better speedup. For all the optimizers, default parameters from the Tensorflow model are used alongside varying learning rates and the same optimizer is used in both training and prefitting. Adam optimizer is much more effective than RSMProp and SGD on accelerating convergence. For SGD, prefitting with learning rate 0.05 and 0.1 causes the loss to explode in a few iterations, thus the corresponding results are left as blank spaces.

Detailed Runtime Comparison. Fig. 15 and 16 shows how training, initial point cloud fitting and total time evolve over different point cloud sizes and ranges. Training time decreases significantly with increasing range, especially from 1.0 to 2.0. This effect becomes more obvious with density normalized runtime. On the other hand, prefitting time increases exponentially with both point cloud range and size. Overall, the total time matches the trend of training time, however the speed-up is halved compared to training due to the addition of prefitting time.

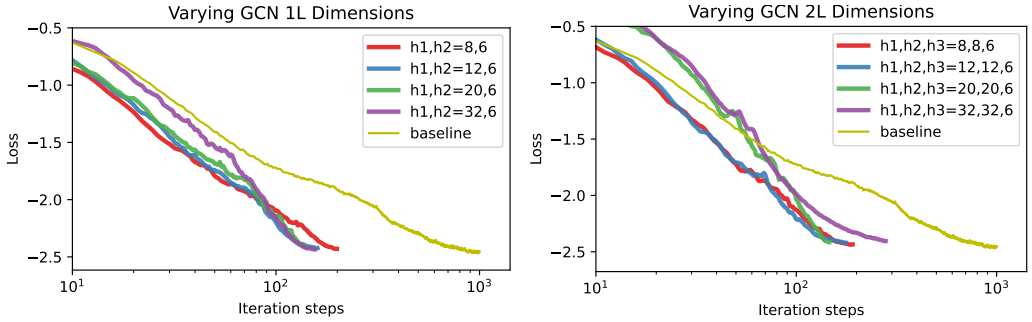

Figure 11: GCN Hyperparameter Comparison. We recommend using one layer GCN model with $h1, h2 = 8, 6$.

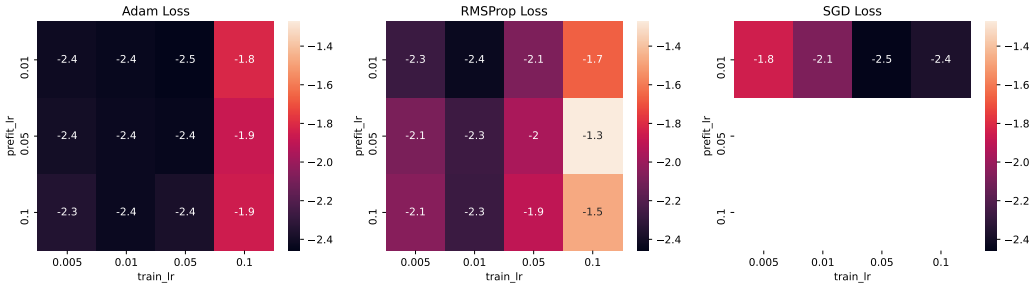

Figure 12: Converged loss of different learning rates and optimizers.

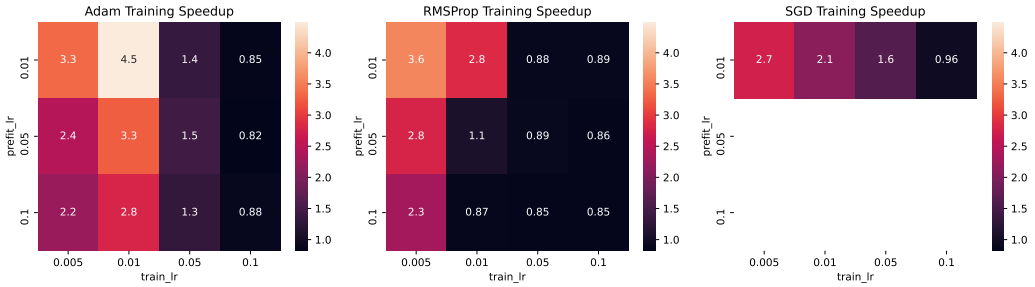

Figure 13: Training Speedup of different learning rates and optimizers.

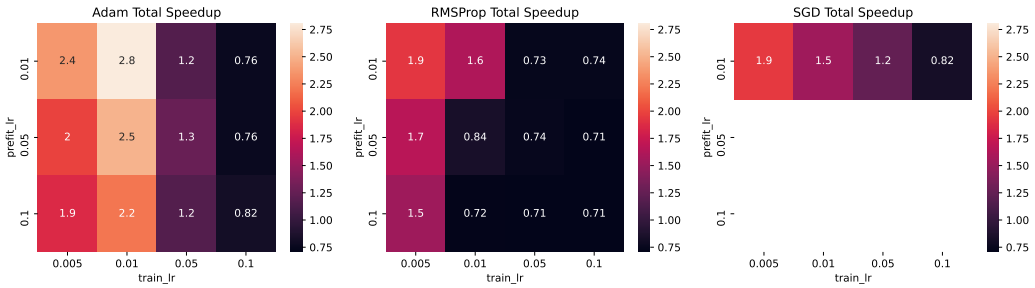

Figure 14: Total Speedup of different learning rates and optimizers.

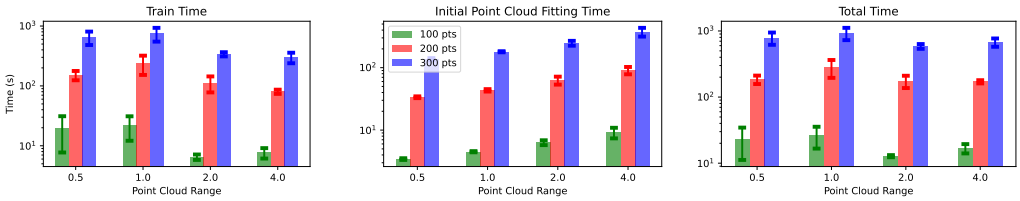

Figure 15: Training, prefitting and total time

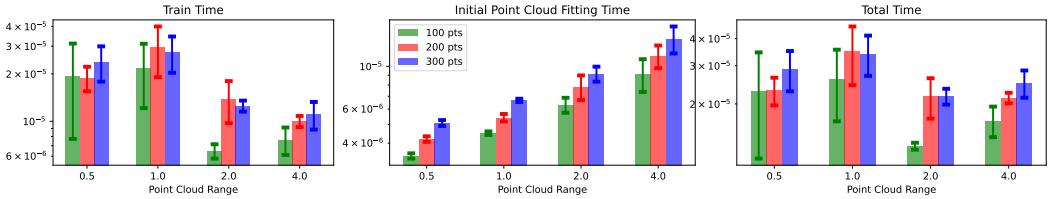

Figure 16: Density Normalized training, prefitting and total time. We normalize the runtime by N^3, where N is the point cloud size. This is because persistence diagram computation has time complexity of $\mathcal{O}(N^3)$

.