# GNN Node Classification Using Koopman Operator Theory on GPU

Christopher Brissette[1,2], William Hawkins[1,3], and George M. Slota[1,4]

[1] Rensselaer Polytechnic Institute, Troy NY
[2] brissc@rpi.edu, [3] hawkiw2@rpi.edu, [4] slotag@rpi.edu

**Abstract.** Koopman operator theory provides a framework for approximating nonlinear dynamical systems with linear operators. Some research has suggested treating deep neural network (DNN) weights as a dynamical system and utilizing discrete Koopman operators to accelerate network training. We extend former work on this topic in two ways. First, we apply this operation to its most promising use-case of graph neural networks (GNN), demonstrating that the method may be generalized to learning tasks beyond DNNs. Second, we implement Koopman approximations on GPU, significantly accelerating previous CPU-based work. We present an algorithm we call "*Patchwork Koopman Approximation*" for accelerating GNN training, and we find that performing Koopman approximation can provide a speedup of over three times that of Adam for the same accuracy and loss on the Cora, Citeseer, and PubMed node classification benchmark datasets.

## 1 Introduction

In recent years, Graph Neural Networks (GNNs) have become standard in the toolkit of network scientists. GNNs are particularly adept at node label and network classification problems, edge prediction, graph generation, and clustering, among many other core network science problems (Wu et al. [2020]). Due to this rising interest in GNNs and Graph Convolutional Networks (GCNs) specifically, many methods for accelerating training have been proposed (see Liu et al. [2022], Zhang et al. [2019], Liang et al. [2021] for recent surveys). These methods include graph sparsification, graph coarsening, and most notably graph pooling. The concept behind each of these methods is to alter the graph topology during training in order to resolve issues with the "neighborhood blow-up" associated with many real world networks. While these methods are topological simplifications in nature, GNNs use the same standard backpropagation-based optimizers as traditional neural network architectures (e.g., deep neural networks (DNNs), convolutional neural networks (CNNs), recurrent neural networks (RNNs), etc.). It then makes sense that acceleration in optimizers for traditional neural networks may lead to acceleration in GNNs. For this purpose, we introduce the Koopman operator, first examined by Koopman [1931].

The Koopman operator is of historical importance in the field of functional analysis. It provides a method for analyzing finite dimensional nonlinear dy-

namics with an infinite dimensional linear system. While the study of this operator was broadly relegated to the whiteboards of functional analysts for many decades, it has experienced a renaissance in the applied community in the last decade (e.g., Brunton et al. [2021], Kutz et al. [2016], Mezić [2013]). This is largely due to the popularity of a method from computational fluid dynamics known as the dynamic mode decomposition (DMD) being interpreted as a finite dimensional approximation of the Koopman operator [Schmid, 2010].

The utility of the Koopman operator is multifaceted, but for the case of neural network training, it is important for one main reason. The discrete Koopman operator allows for accurate predictions in the state evolution of a nonlinear dynamical system at the cost of a single matrix-vector multiplication. Previous work (Dogra and Redman [2020], Tano et al. [2020]) has made this connection and shown that by treating the weights of a neural network as the state variables in a nonlinear dynamical system, Koopman operator theory and DMD may be used to train neural networks. The prior work has also shown that the weights output by Koopman training approximate those of a network fully trained with the underlying optimizer. Despite this, previous publications on the subject either lack the speed to be useful to practitioners, are only shown to be useful near the optimal solution of a DNN, or require incredibly high memory overheads. Furthermore, prior work focuses solely on DNN architectures and do not test on GNNs. We address all of these shortcomings in this manuscript.

As we will discuss, memory overheads of Koopman training, which are proportional to the size of the neural network, can make its practical use limited for modern massive-scale DNNs. However, for GNNs, where the scale of the neural network can be small relative to the scale of the input graph itself, these overheads are relatively negligible. *This makes GNNs an extremely promising target for Koopman training.* Additionally, as Koopman training is simply an optimizer for neural network weights, it can be combined with any of the more specialized acceleration techniques for GNNs such as coarsening, sparsification, and pooling, without modification.

**Our Contribution:** We present a fast method for Koopman training entirely on GPU which performs well in terms of performance acceleration and memory requirements. We apply this method to three standard node classification problems using GCNs and show that Koopman training can produce accurate results in this domain with much faster training times relative to using Adam optimization (Kingma and Ba [2014]). We additionally discuss best practices for hyperparameter selection to ensure performance and numerical stability.

## 2 Background

### 2.1 GNNs / GCNs

GNNs and GCNs rose to popularity in the machine learning community after the publication of a popular paper by Kipf and Welling [2016], detailing a convolutional architecture for learning on graphs. The Kipf and Welling architecture

remains popular, and it will be the architecture referred to interchangeably by the acronyms GNN and GCN for the remainder of this manuscript.

The fundamental idea behind GCNs, and specifically GCNs for node classification, is to produce embeddings for each node based on aggregates of neighboring feature vectors. The difficulty in this technique is learning the aggregation function for each neighborhood. For this purpose, Kipf and Welling suggested the following architecture.

$$H^{l+1} = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^l W^l\right)$$

Here $H^l$ is the matrix of activation functions at the $l^{th}$ layer and $\hat{A} = A + I_{|V|}$ is the adjacency matrix of the underlying graph $G = (V, E)$ with added self-loops. Additionally, $\hat{D}$ is the diagonal degree matrix given by the sums of rows of $\hat{A}$, $\sigma$ is an element-wise activation function, and $W^l$ is a learnable matrix of weight parameters for layer $l$. For a node $u \in V$, this function considers a weighted sum of the activations $H^l$ of its neighbors $v \in \mathcal{N}(u)$, multiplies that by a weight matrix $W^l$, and applies an element-wise activation function. The goal of training the GCN is to solve for all weights in each layer.

Without additional acceleration techniques, this method can be particularly slow, due to the exponential increase in $k$-hop nieghborhood size (i.e., the "neighborhood blow-up" problem). To deal with this, topological simplification methods such as graph pooling are utilized. There are many methods for graph pooling; however, the common idea is to sample from a reduced graph for training instead of using the entire graph topology. This is an active area of research, and techniques range from explicit spectral methods to heuristics (see Gilmer et al. [2017], Ying et al. [2018], Bianchi et al. [2020], Lee et al. [2019]).

### 2.2 Koopman operator

While there is a wealth of information about the Koopman operator available in the literature, we present a specific definition for our application. Denote by $x_t \in \mathbb{R}^n$, the state of a dynamical system at time $t \in \mathbb{R}$. Additionally, take $F$ to define our dynamical system such that $F(x_t) = x_{t+1}$. Also consider any observable $g : \mathbb{R}^n \to \mathbb{R}$ on the Hilbert space $L^2(\mathbb{R}^n, \mu)$. The Koopman operator $K$ is the infinite dimensional linear operator such that the following is true for all such functionals $g$.

$$Kg(x_t) = g(F(x_t))$$

This means $K$ is a linear operator which preserves all measurements of our dynamical system at time $t+1$, given its state at time $t$. Because of this, it may be used to predict future states. Since we do not have access to storage for infinite dimensional matrices, practitioners who wish to use Koopman operator theory must rely on finite rank approximations of $K$. These finite rank approximations may take several forms, including the finite section method and the dynamic mode decomposition. While the dynamic mode decomposition is noteworthy,

and its variants have spawned a wealth of research, it requires the computation of eigenpairs for potentially non-symmetric matrices. As such, it is not amenable to fast parallelism on GPU. For this reason, we will be focusing on the finite section method for Koopman approximation.

The finite section method is relatively straightforward. Given a set of observations of the dynamical system as a matrix $x = [x_0, x_1, \cdots, x_T]$, this matrix may be broken into two further matrices $X = [x_0, x_1, \cdots, x_{T-1}]$ and $Y = [x_1, x_2, \cdots, x_T]$. These matrices define the "before and after" states, where we know $x_{t+1} = F(x_t)$. Then, the finite section approximation $U$ of $K$ is given by the following, where $\dagger$ denotes the pseudo-inverse of a matrix.

$$U = YX^{\dagger}$$

From this, future states may be predicted using the equation $U^s x_t \approx x_{t+s}$. The intuition behind this method is that, by properties of the pseudo-inverse, $\|Ux_t - x_{t+1}\|_2$ is minimized for all $(x_t, x_{t+1})$ in $x$. This means that on our observed subspace we match the nonlinear dynamics as closely as possible with respect to the $L2$-norm.

### 2.3 Koopman training

The concept of Koopman training is simple: treat the weights (parameters) in a neural network as the variables in a dynamical system and apply a Koopman operator at some time step to avoid back-propagation. Functionally, this amounts to tracking the weights of the neural network for some number of time steps $m$, then solving for a finite dimensional Koopman operator approximation $U$, and finally predicting $p$ steps ahead using $U^p$. The advantage to this method is that matrix-vector multiplication is much faster than back-propagation.

We discuss both of the former works studying Koopman training of neural networks by Dogra and Redman [2020] as well as the work by Tano et al. [2020]. These papers present very different approaches to Koopman training. In the paper by Dogra and Redman, Koopman training is applied near the end of optimization when weight evolution is slow. Starting at some time $t_1$, they begin tracking weights until another time $t_2$. These weights are then used to obtain $U$ via the finite section method. $U$ is then used to approximate the final state of the network after many steps. In the paper, they predict an impressive 2500 steps ahead and show that their final network weights are very close to those of the traditionally trained network.

Tano et al. take a very different approach. Primarily, their method utilizes the dynamic mode decomposition as opposed to finite sections. This has two effects. First, it makes their method slower than the work of Dogra and Redman, since now an eigen-decomposition must be computed on top of the pseudo-inverse. Second, however, it can provide improved accuracy, since DMD allows for the pruning of spurious modes which may create instabilities in predictions. Beyond that, the authors do not wait to use their Koopman operator as Dogra and Redman do. Instead, they alternate during training, performing $m$ steps of

standard optimization before predicting $p$ steps forward in time and returning to standard training for another $m$ steps. This allows for acceleration throughout the entire training process, not just near a minimum.

Both methods are shown to train their test networks well; however, there are gaps for further research. As noted by both groups, these implementations are CPU only, but they require a GPU implementation to be useful to practitioners. In fact, Tano et al. note that their DMD-based method is ultimately slower in terms of clock time than standard optimization. Furthermore, all test instances in both papers were on DNN architectures, and it is yet to be shown how similar methods may generalize to learning tasks such as GCNs.

## 3 Methods

### 3.1 Algorithm

Our Koopman training algorithm is similar to Tano et al., in that it alternates between standard optimization and Koopman prediction. See Algorithm 1. In one phase (Line 8), training is performed using a standard optimization technique, such as stochastic gradient descent (Bottou et al. [1991]), or adaptive moment estimation (Adam) (Kingma and Ba [2014]). This training is performed on some input network $M$ for some number of pre-determined steps $m$, and the weights of the network are stored as a column $W_i \in \mathbb{R}^\omega$ in the weight history matrix $W \in \mathbb{R}^{\omega \times m}$, where $\omega$ is the number of learnable parameters in the network. We additionally use hyperparameters $r$ and $p$, which are the finite dimension for SVD truncation and number of steps predicted forward, respectively. We summarize these variables in Table 1.

Table 1: Variables for Koopman training (Alg. 1) and prediction (Alg. 2).

| Variable | Definition |
|---|---|
| $m$ | Number of standard training steps |
| $p$ | Number of steps predicted forward |
| $r$ | Finite dimension for SVD truncation |
| $M$ | Input neural network |
| $\omega$ | Number of learnable parameters in the network |
| $W$ | Weight history matrix for prior steps |

In the second phase of Koopman training, seen in Algorithm 2 (and called from Line 6 in Algorithm 1), the weight history matrix is used to form the matrices $X = [W_0, W_1, \cdots, W_{m-1}]$ and $Y = [W_1, W_2, \cdots, W_m]$. Then the singular value decomposition (SVD) is computed for $X = Z\Sigma V^*$ and its pseudo-inverse is computed from that decomposition. This yields our finite section approximation of the Koopman operator $U$ as the following.

$$U = YV\Sigma^{-1}Z^*$$

It should be noted that computing the SVD is an incredibly expensive operation with a work complexity of $O\left(\omega(m-1)^2\right)$. One may notice that Algorithm 2

**Algorithm 1** : Koopman training $(M, m, p, r, \texttt{epochs})$

$\omega \leftarrow \texttt{param_num}(M)$
$W \leftarrow \texttt{zeros}(\mathbb{R}^{\omega \times m})$
$W \leftarrow \texttt{pad}(W, \omega \bmod 32)$
**for** $i \in [1..\texttt{epochs}]$ **do**
  **if** $i \bmod m = 0$ **then**
    $\texttt{weights}(M) \leftarrow \texttt{koopman_prediction}(W, m, p, r)$
  **else**
    $\texttt{weights}(M) \leftarrow \texttt{train_epoch(M)}$
  $W_{(i \bmod m)} \leftarrow \texttt{weights}(M)$
**return** $M$

reshapes the data before performing SVD. This is because we subdivide both $X$ and $Y$ into small matrices for efficient batched computations on GPU. This yields a piece-wise approximation of $U$ across subsets of weights in the network. In the language of Dogra and Redman, this is a type of sub-node level Koopman operator. We call this "*Patchwork Koopman Approximation*" (PKA). Using Patchwork Koopman Approximation for training is discussed in further detail in the "Implementation" section that follows.

**Algorithm 2** : Koopman prediction $(W, m, p, r)$

$X \leftarrow \texttt{reshape}\left([W_0, W_1, \cdots, W_{m-1}]\right)$
$Y \leftarrow \texttt{reshape}\left([W_1, W_2, \cdots, W_m]\right)$
$(Z, \Sigma, V^*) \leftarrow \texttt{batched_svd}(X)$
$Z \leftarrow \texttt{trunc}(Z, r)$
$\Sigma \leftarrow \texttt{trunc}(\Sigma, r)$
$V^* \leftarrow \texttt{trunc}(V^*, r)$
$U \leftarrow YV\Sigma^{-1}Z^*$
$v \leftarrow \texttt{reshape}(W_m)$
**for** $j \in [1..p]$ **do**
  $v \leftarrow Uv$
**return** $v$

After $U$ is computed, training is projected forward a pre-determined number of steps $p$ by the equation $U^p W_m = W_{m+p}$. Afterwards, the weight history matrix is cleared, $W_{m+p}$ replaces the old vector $W_0$, and we return to standard optimization for another $m$ steps before repeating the process again. This is repeated for some number of epochs, or some early stopping criteria.

### 3.2 Implementation

A number of optimizations are made to the baseline Algorithm 1 in order to improve its performance on GPU. For starters, the expensive $O\left(\omega(m-1)^2\right)$ SVD is broken into many subproblems. In Figure 1, it can be seen that Algorithm 2 reshapes the first $m-1$ columns of $W$ into matrices of size $32 \times (m-1)$. This

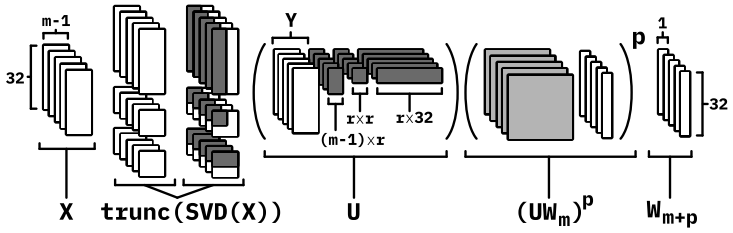



Fig. 1: Visualization of Algorithm 2.

is done in order to make use of the batched SVD operation available via `CUDA`'s `gesvdjBatched` function. This function allows for multiple singular value decompositions to be performed at once, so long as the shape of each individual matrix is smaller than or equal to $32 \times 32$ and all sub-matrices are the same size. Because of this, $U$, in our case, does not stand for the Koopman operator of the entire model. Instead, $U$ can be thought of as a Patchwork Koopman operator of smaller matrices $U = \{U_i\}$, where each matrix is an approximation of the full Koopman operator on a subspace with maximum dimension 32. In order to meet these requirements, zeros may need to be added to pad the size of $W$ such that its column number is divisible by 32 (see the function $\texttt{pad}(\cdot)$ in Algorithm 1).

We further optimize our method selecting a finite dimension $r \leq m$ and use it to truncate the SVD $(Z, \Sigma, V^*)$. This is akin to using principal component analysis (PCA) to reduce the dimensionality of $X$ before computing the pseudo-inverse. This has two benefits. First, for $r < m$, this can drastically reduce both the total work required as well as the per-thread work. Additionally, for small $r$, this prunes minuscule singular values from $\Sigma$. If not pruned, these values may cause instabilities in the method. After the truncation step, $U = YV\Sigma^{-1}Z^*$ is computed. The influence of $r$ on our patchwork Koopman implementation is discussed later. Unfortunately, there are no readily available methods for batched eigen-pair computations of non-symmetric matrices on GPU. This relegates us to using the finite section method, and it means that fast implementations of more robust Koopman approximations such as Exact DMD (Tu et al. [2014]) and ResDMD (Colbrook and Townsend [2021]) are currently infeasible.

All of our algorithms were implemented using the `PyTorch` (Paszke et al. [2019]) and `CuPy` (Nishino and Loomis [2017]) libraries in `Python`. `CuPy` is GPU-optimized implementation of the functionality contained within `NumPy` and `SciPy`. While both of these libraries have access to functions utilizing `gesvdjBatched`, we used the `PyTorch` batched SVD function and $\texttt{torch.linalg.svd}(\cdot)$. `CuPy` was used for all other linear algebraic operations.

## 4 Experiments

We consider the `PyTorch` Geometric Planetoid datasets of `Cora`, `PubMed`, and `Citeseer` (Bollacker et al. [1998], McCallum et al. [2000], Yang et al. [2016]) for GCN node classification tasks. We run our experiments using a NVIDIA 40 GB

A100 GPU on the `zepy` server in RPI's High Performance Combinatorics and Graph Analytics Laboratory. We consider a GNN architecture with two convolutions. The outputs of all convolution layers are 64, their activation function is `ReLu`, and the output is put through a `softmax` layer. Standard training iterations were performed with the `Adam` optimizer in `PyTorch` with the default learning rate of 0.001. The network architecture was selected because the weight numbers would be easily divisible for patchwork Koopman, and in testing they were expressive enough to train well for our classification problems.

For the results that we will show, we have selected $m = 4$, $p = 8$, and $r = 2$ as the PKA hyperparameters to balance stability and time-to-solution. We will discuss the selection of these values later. We primarily compare our method against the baseline `Adam` optimization without any PKA prediction. All results we will show are the average of 100 training runs with 400 maximum epochs.

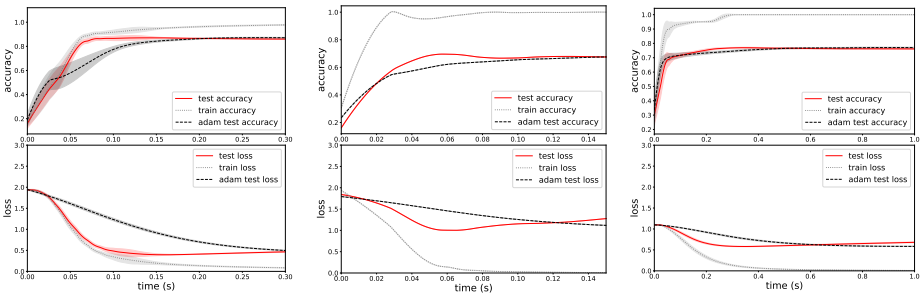
### 4.1 Results

We first run both Adam and PKA training across all datasets to observe the average highpoint for test accuracy and lowpoint for test loss. We give these results in Table 2. We make several observations. Primarily, we note that PKA on average equals or outperforms Adam in all tests with respect to loss and accuracy. Further, we note that PKA performs training 2-3× faster than Adam, with respect to the maximum accuracy or minimum loss achieved.

Table 2: The maximum accuracy achieved and minimum loss achieved by Adam and PKA for the three test instances, along with the elapsed time required to achieve those values.

| | `Cora` | | `Citeseer` | | `PubMed` | |
|---|---|---|---|---|---|---|
| | Adam | PKA | Adam | PKA | Adam | PKA |
| Accuracy Highpoint | 0.874 | 0.875 | 0.681 | 0.687 | 0.774 | 0.774 |
| Highpoint time (s) | 0.291 | 0.140 | 0.143 | 0.078 | 1.007 | 0.365 |
| Loss Lowpoint | 0.419 | 0.399 | 1.053 | 1.020 | 0.583 | 0.579 |
| Lowpoint Time (s) | 0.553 | 0.210 | 0.236 | 0.098 | 1.182 | 0.353 |
| Accuracy Speedup | | 2.08× | | 1.83× | | 2.76× |
| Loss Speedup | | 2.63× | | 2.41× | | 3.35× |

In Figure 2, we visualize the training behavior of Adam and PKA with respect to accuracy and loss over time. The figure gives the averaged accuracy and loss over 100 training runs with respect to elapsed time. We also give the variance across runs via the shaded envelopes for each line. We note that PKA generally trains slower than Adam initially, but PKA surpasses Adam once a (presumed) threshold is passed where the dynamics of the network weights becomes more reliably predictable.




(a) `Cora` (b) `Citeseer` (c) `PubMed`

Fig. 2: Visualization of accuracy (top) and loss (bottom) for the three test instances during training. Plotted is the interpolated average training and test accuracy/loss versus time for all test instances relative to Adam. Averages are calculated across 100 total runs, with the variance among runs visualized via the shaded region for each line.

### 4.2 Discussion

**Performance:** We note that our method offers notable speedup relative to Adam over the considered test instances, while equaling or improving on generalization performance. The ability of PKA to accurately capture the dynamics of the network weights and usefully predict the weights of future epochs is surprisingly impressive, particularly when combined with the observed speedups. We are the first work that actually demonstrates such speedups in practice.

Beyond speedups, our method requires minimal memory relative to prior work. Any Koopman training method will require a minimum of $O(m\omega)$ floats for tracking weights and $O((m-1)\omega + 2(m-1)^2)$ for computing the SVD. In Dogra and Redman [2020], a very large $m$ is required, as they only use Koopman approximation for the final steps of training near convergence and tracking the state for a long time horizon is needed to ensure accuracy. Large $m$ values quickly become unfeasible on GPU. As mentioned, we use a method similar to Tano et al. [2020] which alternates between Koopman training and standard training techniques. This requires a much smaller $m$ value that is more amendable to use in limited GPU memory.

**Hyperparameter Selection:** We performed a preliminary parametric study over PKA hyperparameters $m$, $p$, and $r$, varied over $\{4, 8, 16, 32\}$, $\{4, 8, 16, 32\}$, and $\{1, 2, 3, 4, 5\}$, respectively. The values for $m$ had to be less than 32 to allow use of batched SVD with CUDA. We observed similar performance to those shown in Table 2 and Figure 2 across a wide breadth of the parameter space. We unfortunately cannot show the full set of results due to space, but we can discuss our primary observations.

Generally, we found that larger values of $p$ or $r$ can result in instability of PKA. Because we are iterating matrices on vectors in Algorithm 2, we can loosely analyze this method in terms of power iteration. As $p$ increases, $U^p v$ approaches the eigenvector associated with $\|U\|_2$ for a random vector $v$. Because of this,

one avenue for improving stability is to reduce $p$. This means that it is generally best practice to not "over-predict" when using patchwork Koopman. We found $p$ values of 8 or less to reliably resulted in convergence for our experiments.

Alternatively, when $\|U\|_2$ is comparatively large, it will require fewer iterations $p$ to achieve a similar loss increase. This suggests one may reduce $\|U\|_2$ in order to improve stability. In our implementation this is done through increasing the parameter $r$. Recalling that $U_i = Y_i X_i^\dagger$, we know $\|U_i\|_2 \leq \|Y_i\|_2 \|X_i^\dagger\|_2$, therefore, $\|U_i\|_2$ can be controlled by reducing $\|Y_i\|_2$ or $\|X_i^\dagger\|_2$, respectively. Because $r$ prunes the smallest singular values from $X_i$, that means it also prunes the largest singular values from $X_i^\dagger$, thus minimizing $\|X_i^\dagger\|_2$. It should additionally be noted that $r$ should not be too low as to not lose out on crucial information for prediction. For our test instances, the suggested minimum $r$ value is 2, which is also what we used for our experiments.

Parameter $m$ controls how many prior sets of weights are stored for prediction. A larger $m$ therefore increases memory usage and the cost of the batched SVD. However, a larger $m$, especially relative to $p$, is also observed to increase convergence stability. The recommended value of of $m$ should be only as large as is necessary to guarantee convergence, to get the best performance. For our datasets, the value fitting this threshold was $m = 4$.

**Future Work:** Since patchwork Koopman training can be used in conjunction with any optimizer, it is a compelling avenue for further research, and there are many open problems. Perhaps the most important open problem for practical usage of PKA is a dynamic selection of hyperparameters, to avoid manual tuning efforts. Though we note that this is a general and long-standing issue plaguing most optimizers used in neural network training. Another avenue of future work is implementing batched Exact DMD or ResDMD on GPU. This is akin to removing the noise from the approximation, though at a computational cost. In lieu of that, we expect that there are more clever ways to divide the network weights, such that each Koopman sub-operator $U_i$ yields a better approximation of its underlying nonlinear dynamics. Additionally, this manuscript focused on GNN training for node classification, and prior work has written about training DNNs, but other relevant architectures and applications have yet to be explored.

## 5 Conclusions

We presented an algorithm on GPU which uses a Koopman operator approximation to accelerate optimizers for GCN node classification. Our implementation is the first to present on-GPU speedups for Koopman training methods over the entire training window. We found that our operator, which we call "*Patchwork Koopman Approximation*", can be computed efficiently with the use of popular libraries such as `PyTorch` and `CUDA`'s `gesvdjBatched` functionality. We performed a study over the `Cora`, `Citeseer`, and `PubMed` becnhmark datasets, and we found that our method boasts a speedup of over two times that of Adam in most cases, while achieving the same or better test accuracy and loss.

# Bibliography


Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. Spectral clustering with graph neural networks for graph pooling. In *International conference on machine learning*, pages 874–883. PMLR, 2020.

Kurt D Bollacker, Steve Lawrence, and C Lee Giles. Citeseer: An autonomous web agent for automatic retrieval and identification of interesting publications. In *Proceedings of the second international conference on Autonomous agents*, pages 116–123, 1998.

Léon Bottou et al. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8):12, 1991.

Steven L Brunton, Marko Budišić, Eurika Kaiser, and J Nathan Kutz. Modern koopman theory for dynamical systems. *arXiv preprint arXiv:2102.12086*, 2021.

Matthew J Colbrook and Alex Townsend. Rigorous data-driven computation of spectral properties of koopman operators for dynamical systems. *arXiv preprint arXiv:2111.14889*, 2021.

Akshunna S Dogra and William Redman. Optimizing neural networks via koopman operator theory. *Advances in Neural Information Processing Systems*, 33:2087–2097, 2020.

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. PMLR, 2017.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Bernard O Koopman. Hamiltonian systems and transformation in hilbert space. *Proceedings of the National Academy of Sciences*, 17(5):315–318, 1931.

J Nathan Kutz, Steven L Brunton, Bingni W Brunton, and Joshua L Proctor. *Dynamic mode decomposition: data-driven modeling of complex systems*. SIAM, 2016.

Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling. In *International conference on machine learning*, pages 3734–3743. PMLR, 2019.

Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021.

Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, Dongrui Fan, Shirui Pan, and Yuan Xie. Survey on graph neural network acceleration: An algorithmic perspective. *arXiv preprint arXiv:2202.04822*, 2022.

Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3:127–163, 2000.

Igor Mezić. Analysis of fluid flows via spectral properties of the koopman operator. *Annual Review of Fluid Mechanics*, 45:357–378, 2013.

ROYUD Nishino and Shohei Hido Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st confernce on neural information processing systems*, 151(7), 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Peter J Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of fluid mechanics*, 656:5–28, 2010.

Mauricio E Tano, Gavin D Portwood, and Jean C Ragusa. Accelerating training in artificial neural networks with dynamic mode decomposition. *arXiv preprint arXiv:2006.14371*, 2020.

Jonathan H Tu, Clarence W Rowley, Dirk M Luchtenburg, Steven L Brunton, and J Nathan Kutz. On dynamic mode decomposition: Theory and applications. *Journal of Computational Dynamics*, 1(2):391–421, 2014.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

Zhilin Yang, William Cohen, and Ruslan Salakhudinov. Revisiting semi-supervised learning with graph embeddings. In *International conference on machine learning*, pages 40–48. PMLR, 2016.

Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems*, 31, 2018.

Qianru Zhang, Meng Zhang, Tinghuan Chen, Zhifei Sun, Yuzhe Ma, and Bei Yu. Recent advances in convolutional neural network acceleration. *Neurocomputing*, 323:37–51, 2019.