GNN Node Classification Using Koopman Operator Theory on GPU

Christopher Brissette, William Hawkins, George M. Slota
Rensselaer Polytechnic Institute, **Contact:** slotag@rpi.edu

Graph Neural Networks

Graph Neural Networks (GNNs) are now ubiquitous in the toolkit of network scientists. Graph convolutional neural networks produce embeddings for each node by aggregating neighborhood feature vectors. We use the Kipf And Welling architecture of

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

for the purpose of node classification. Here, H^l is activation functions at the l^{th} layer, $\hat{A} = A + I_{|V|}$ is the adjacency matrix of underlying graph $G = (V, E)$, $\hat{D}$ is the diagonal degree matrix, σ 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 is to solve for all weights in each layer. This work seeks to accelerate this training using a Koopman operator on GPU.

Contributions

Our work has several contributions towards both the practical usage of a Koopman operator for neural network training as well as the specific training of graph neural network problems.

1. We present a method of Koopman training optimized on GPU with high performance and relatively low memory requirements.
2. We demonstrate our method by greatly accelerating the training of three benchmark node classification problems, at no cost to accuracy or loss.
3. We discuss how our method can be more generally applied to other problems, as well as considerations for hyperparameter selection.

References

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

[2] 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

[3] Diederik P Kingma and Jimmy Ba Adam: A method for stochastic optimization arXiv preprint arXiv:1412.6980, 2014

Patchwork Koopman Approximation

The Koopman operator has been used widely in the literature, mostly for dynamic model decomposition within computational fluid dynamics. We define the Koopman operator on dynamic system F with state $x_t \in \mathbb{R}^n$ at time t along with observable $g : \mathbb{R}^n \to \mathbb{R}$ on the Hilbert space $L^2(\mathbb{R}^n, \mu)$. Our Koopman operator K is the infinite dimensional linear operator such that the following is true for all such functionals g:

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

To predict future states of F, we consider a set of observations $x = [x_0, x_1, \cdots, x_T]$, broken into submatrices $X = [x_0, x_1, \cdots, x_{T-1}]$ and $Y = [x_1, x_2, \cdots, x_T]$, defining 'before' and 'after' states where $x_{t+1} = F(x_t)$. We then have the approximation

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

where $YV\Sigma^{-1}Z^{*}$ is a SVD used to compute the pseudo-inverse. Future states are predicted via $U^s x_t \approx x_{t+s}$. With x_t as the weight history of a neural network trained with standard optimization, we can use this method to predict future weights. A single matrix-vector product can approximate an entire training epoch.

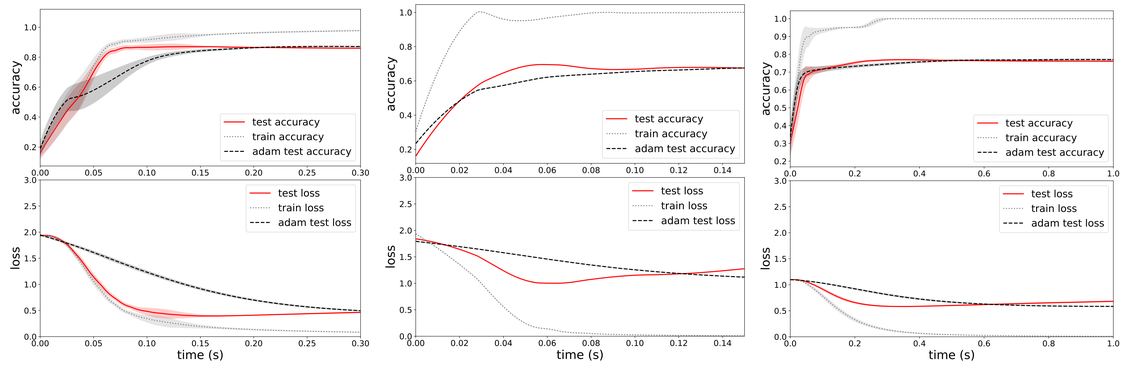
We use the general above approach of Tano et al. and optimize it on GPU (Patchwork Koopman Approximation - PKA) for the purposes of training Graph Neural Networks (GNNs). We select GNNs as an obvious target for PKA, as the memory overheads of storing the state matrix x is minimal when considering that the underlying graphs for GNNs are generally much larger than the GNNs being trained.

Results: Significant Acceleration over Adam

We run PKA and Adam (Kingma and Ba, 2014) on PyTorch Geometric datasets Cora, Pubmed, and Citeseer for node classification on a A100 GPU. Below is the maximum accuracy and minimum loss achieved by Adam and PKA for the three test instances, along with the elapsed time required to achieve those values. We note speedups of 2-3× on average with PKA with no accuracy/loss cost, averaged over 100 training runs.

	Cora		Citeseer		PubMed	
	Adam	PKA	Adam	PKA	Adam	PKA
Accuracy Highpoint	0.874	0.875	0.681	0.687	0.774	0.774
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
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×

We also visualize these results below on Cora, Citeseer, and Puubmed (from left to right) with accuracy (top) and loss (below) plotted against time. Variance across 100 tests is observable via the shaded regions.



Discussion

Hyperparameters

- m - history window for Koopman. We use a value of $m = 4$, meaning that we retain the prior weights for the previous 4 epochs. This keeps memory overheads manageable while still providing enough data for accurate prediction.
- p - forward prediction steps. Using the m sets of weights, we predict $p = 8$ steps forward for our training. Higher values of p become more unstable and less accurate.
- r - finite dimension. While Koopman operators are technically infinite in dimension, we use a finite dimensional SVD for approximation of $U = YX^{\dagger} = YV\Sigma^{-1}Z^{*}$ with dimension of $r = 2$. This provides a good approximation in practice.

Performance

- We observe a consistent speedup of about at least 2× in our testing across our three experimental datasets.
- Our usage of specific GPU subroutines and a small batch size for computing our Koopman operator accounts for this acceleration with minimal memory cost.
- We are the first to demonstrate actual training time acceleration with the usage of Koopman approximation. Our work can also be generalized to any neural network training scenario.

Future Work

Optimizer Usage: Since PKA can be used with any optimizer, exploring alternatives is a compelling avenue for future work. Stochastic gradient descent (SGD) is of particular interest, since its low memory requirements can make using it with PKA equivalent in memory utilization to Adam without PKA.

Hyperparameter Selection: As with most other optimizers, the hyperparameters of PKA might require tuning for given datasets for optimal performance. Dynamic selection of hyperparameters is an obvious open problem.