Implicit SVD for Graph Representation Learning

Sami Abu-El-Haija *  
USC Information Sciences Institute  
sami@haija.org

Hesham Mostafa, Marcel Nassar  
Intel Labs  
{hesham.mostafa,marcel.nassar}@intel.com

Valentino Crespi, Greg Ver Steeg, Aram Galstyan  
USC Information Sciences Institute  
{vcrespi,gregv,galstyan}@isi.edu

Abstract

Recent improvements in the performance of state-of-the-art (SOTA) methods for Graph Representational Learning (GRL) have come at the cost of significant computational resource requirements for training, e.g., for calculating gradients via backprop over many data epochs. Meanwhile, Singular Value Decomposition (SVD) can find closed-form solutions to convex problems, using merely a handful of epochs. In this paper, we make GRL more computationally tractable for those with modest hardware. We design a framework that computes SVD of implicitly defined matrices, and apply this framework to several GRL tasks. For each task, we derive a linear approximation of a SOTA model, where we design (expensive-to-store) matrix $M$ and train the model, in closed-form, via SVD of $M$, without calculating entries of $M$. By converging to a unique point in one step, and without calculating gradients, our models show competitive empirical test performance over various graphs such as article citation and biological interaction networks. More importantly, SVD can initialize a deeper model, that is architected to be non-linear almost everywhere, though behaves linearly when its parameters reside on a hyperplane, onto which SVD initializes. The deeper model can then be fine-tuned within only a few epochs. Overall, our procedure trains hundreds of times faster than state-of-the-art methods, while competing on empirical test performance. We open-source our implementation at: https://github.com/samihaija/isvd

1 Introduction

Truncated Singular Value Decomposition (SVD) provides solutions to a variety of mathematical problems, including computing a matrix rank, its pseudo-inverse, or mapping its rows and columns onto the orthonormal singular bases for low-rank approximations. Machine Learning (ML) software frameworks (such as TensorFlow) offer efficient SVD implementations, as SVD can estimate solutions for a variety of tasks, e.g., in computer vision [Turk and Pentland, 1991], weather prediction [Molteni et al., 1996], recommendation [Koren et al., 2009], language [Deerwester et al., 1990; Levy and Goldberg, 2014], and more-relevantly, graph representation learning (GRL) [Qiu et al., 2018].

SVD’s benefits include training models, without calculating gradients, to arrive at globally-unique solutions, optimizing Frobenius-norm objectives (§2.3), without requiring hyperparameters for the learning process, such as the choice of the learning algorithm, step-size, regularization coefficient, etc. Typically, one constructs a design matrix $M$, such that, its decomposition provides a solution to a task of interest. Unfortunately, existing popular ML frameworks [Abadi et al., 2016; Paszke et al., 2019] cannot calculate the SVD of an arbitrary linear matrix given its computation graph: they

*Part of this work was done during internship at Intel Labs.

compute the matrix (entry-wise) then its decomposition. This limits the scalability of these libraries in several cases of interest, such as in GRL, when explicit calculation of the matrix is prohibitive due to memory constraints. These limitations render SVD as impractical for achieving state-of-the-art (SOTA) for tasks at hand. This has been circumvented by [Qiu et al., 2018] by sampling M entry-wise, but this produces sub-optimal estimation error and experimentally degrades the empirical test performance ([7] Experiments).

We design a software library that allows symbolic definition of M, via composition of matrix operations, and we implement an SVD algorithm that can decompose M from said symbolic representation, without need to compute M. This is valuable for many GRL tasks, where the design matrix M is too large, e.g., quadratic in the input size. With our implementation, we show that SVD can perform learning, orders of magnitudes faster than current alternatives.

Currently, SOTA GRL models are generally graph neural networks trained to optimize cross-entropy objectives. Their inter-layer non-linearities place their (many) parameters onto a non-convex objective surface where convergence is rarely verified. Nonetheless, these models can be convexified ([3]) and trained via SVD, if we remove nonlinearities between layers and swap the cross-entropy objective with Frobenius norm minimization. Undoubtedly, such linearization incurs a drop of accuracy on M, this is too large, e.g., quadratic in the input size. With our implementation, we show that SVD can perform learning, orders-of-magnitudes faster than current alternatives.

1. We open-source a flexible python software library that allows symbolic definition of matrices and computes their SVD without explicitly calculating them.
2. We linearize popular GRL models, and train them via SVD of design matrices.
3. We show that fine-tuning a few parameters on-top of the SVD initialization sets state-of-the-art on many GRL tasks while, overall, training orders-of-magnitudes faster.

2 Preliminaries & notation

We denote a graph with n nodes and m edges with an adjacency matrix $A \in \mathbb{R}^{n \times n}$ and additionally, if nodes have (d-dimensional) features, with a feature matrix $X \in \mathbb{R}^{n \times d}$. If nodes $i, j \in [n]$ are connected then $A_{ij}$ is set to their edge weight and otherwise $A_{ij} = 0$. Further, denote the (row-wise normalized) transition matrix as $T = D^{-1}A$ and denote the symmetrically normalized adjacency with self-connections as $\hat{A} = (D + I)^{-\frac{1}{2}}(A + I)(D + I)^{-\frac{1}{2}}$ where I is identity matrix.

We review model classes: (1) network embedding and (2) message passing that we define as follows. The first inputs a graph $(A, X)$ and outputs node embedding matrix $Z \in \mathbb{R}^{n \times z}$ with z-dimensions per node. Z is then used for an upstream task, e.g., link prediction. The second class utilizes a function $H : \mathbb{R}^{n \times n} \times \mathbb{R}^{n \times d} \rightarrow \mathbb{R}^{n \times z}$ where the function $H(A, X)$ is usually directly trained on the upstream task, e.g., node classification. In general, the first class is transductive while the second is inductive.

2.1 Network embedding models based on DeepWalk & node2vec

The seminal work of DeepWalk [Perozzi et al., 2014] embeds nodes of a network using a two-step process: (i) simulate random walks on the graph – each walk generating a sequence of node IDs then (ii) pass the walks (node IDs) to a language word embedding algorithm, e.g. word2vec [Mikolov et al., 2013], as-if each walk is a sentence. This work was extended by node2vec [Grover and Leskovec, 2016] among others. It has been shown by [Abu-El-Haija et al., 2018] that the learning outcome of

$\nabla_{\theta} J = 0$, where $J$ is mean train objective and $\theta$ are model parameters.
the two-step process of DeepWalk is equivalent, in expectation, to optimizing a single objective:

$$
\min_{Z=\{L,R\}} \sum_{(i,j) \in [n] \times [n]} \left[ -\mathbb{E}_{q \sim \mathcal{Q}} [T^q] \circ \log \sigma(LR^T) - \lambda(1 - A) \circ \log(1 - \sigma(LR^T)) \right]_{ij},
$$

(1)

where L, R ∈ ℝnx are named by word2vec as the input and output embedding matrices, o is Hadamard product, and the log(.) and the standard logistic σ(.) = (1 + exp(.) )−1 are applied element-wise. The objective above is weighted cross-entropy where the (left) positive term weighs the dot-product L1,R1 by the (expected) number of random walks simulated from i and passing through j, and the (right) negative term weighs non-edges (1 − A) by scalar λ ∈ ℝ+. The context distribution Q stems from step (ii) of the process. In particular, word2vec accepts hyperparameter context window size C for its stochastic sampling: when it samples a center token (node ID), it then samples its context tokens that are up-to distance c from the center. The integer c is sampled from a coin flip uniform on the integers [1, 2, ..., C] as detailed by Sec.3.1 of [Levy et al., 2015]. Therefore, Pq(q | C) ∝ C−q+1. Since q has support on [C], then Pq(q | C) = 2(C+1)/C C−q+1.

2.2 Message passing graph networks for (semi-)supervised node classification

We are also interested in a class of (message passing) graph network models taking the general form:

$$
\text{for } l = 0, 1, \ldots, L: \quad H^{(l+1)} = \sigma_l \left( g(A)H^{(l)}W^{(l)} \right); \quad H^{(0)} = X; \quad H = H^{(L)};
$$

(2)

where L is the number of layers, W(l)’s are trainable parameters, σl’s denote element-wise activations (e.g. logistic or ReLu), and q is some (possibly trainable) transformation of adjacency matrix. GCN [Kipf and Welling, 2017] set g(A) = A, GAT [Veličković et al., 2018] set g(A) = A o MultiHeadedAttention and GIN [Xu et al., 2019] as g(A) = A + (1 + ϵ)I with ϵ > 0. For node classification, it is common to set σL = softmax (applied row-wise), specify the size of W_L, s.t. H ∈ ℝnx where y is number of classes, and optimize cross-entropy objective:

$$
\min_{\{W_l\}_{l=1}^{L}} \left[ -Y \circ \log H - (1 - Y) \circ \log(1 - H) \right], \quad \text{where } Y \text{ is a binary matrix with one-hot rows indicating node labels. In semi-supervised settings where not all nodes are labeled, before measuring the objective, subset of rows can be kept in Y and H that correspond to labeled nodes.}
$$

2.3 Truncated Singular Value Decomposition (SVD)

SVD is an algorithm that approximates any matrix M ∈ ℝrxc as a product of three matrices:

$$
\text{SVD}_k(M) \triangleq \arg \min_{U,S,V} \| M - USV^T \|_F \text{ subject to } U^T U = V^T V = I_k; \quad S = \text{diag}(s_1, \ldots, s_k).
$$

The orthonormal matrices U ∈ ℝrxk and V ∈ ℝck,k, respectively, are known as the left- and right-singular bases. The values along diagonal matrix S ∈ ℝkk are known as the singular values.

Due to theorem of [Eckart and Young, 1936], SVD recovers the best rank-k approximation of input M, as measured by the Frobenius norm ||.||_F. Further, if k ≥ rank(M) ⇒ ||.||_F = 0.

Popular SVD implementations follow Random Matrix Theory algorithm of [Halko et al., 2009]. The prototype algorithm starts with a random matrix and repeatedly multiplies it by M and by M^T, interleaving these multiplications with orthonormalization. Our SVD implementation (in Appendix) also follows the prototype of [Halko et al., 2009], but with two modifications: (i) we replace the recommended orthonormalization step from QR decomposition to Cholesky decomposition, giving us significant computational speedups and (ii) our implementation accepts symbolic representation of M (§4), in lieu of its explicit value (in contrast to TensorFlow and PyTorch, requiring explicit M).

In §3, we derive linear first-order approximations of models reviewed in §1, §2.2 and explain how SVD can train them. In §5, we show how they can be used as initializations of non-linear models.

---

4Derivation is in [Abu-El-Haija et al., 2018]. Unfortunately, matrix in Eq. (1) is dense with O(n^2) nonzeros.
3 Convex first-order approximations of GRL models

3.1 Convexification of Network Embedding Models

We can interpret objective \([\ell]\) as self-supervised learning, since node labels are absent. Specifically, given a node \(i \in [n]\), the task is to predict its neighborhood as weighted by the row vector \(E_q[T_i]\), representing the subgraph \(T_i\) around \(i\). Another interpretation is that Eq. \([\ell]\) is a decomposition objective: multiplying the tall-and-thin matrices, as \(LR^T \in \mathbb{R}^{n \times n}\), should give a larger value at \((LR^T)_{ij} = L_j^T R_i\) when nodes \(i\) and \(j\) are well-connected but a lower value when \((i, j)\) is not an edge. We propose a matrix such that its decomposition can incorporate the above interpretations:

\[
\hat{M}^{(NE)} = \mathbb{E}_q[C \left[T^q\right] - \lambda (1 - A) = \left( \frac{2}{(C + 1)C} \right) \sum_{q=1}^{C} \left( \frac{C - q + 1}{C} \right) T^q - \lambda (1 - A) \tag{3}
\]

If nodes \(i, j\) are nearby, shares a lot of connections, and/or in the same community, then entry \(\hat{M}^{(NE)}_{ij}\) should be positive. If they are far apart, then \(\hat{M}^{(NE)}_{ij} = -\lambda\). To embed the nodes onto a low-rank space that approximates this information, one can decompose \(\hat{M}^{(NE)}\) into two thin matrices \((L, R)\):

\[
LR^T \approx \hat{M}^{(NE)} \iff (LR^T)_{i,j} = (L_i, R_j) \approx \hat{M}^{(NE)}_{ij} \text{ for all } i, j \in [n]. \tag{4}
\]

SVD gives low-rank approximations that minimize the Frobenius norm of error \(\|\hat{M} - LR^T\|_F\). The remaining challenge is computational burden: the right term \((1 - A)\), a.k.a. graph compliment, has \(\approx n^2\) non-zero entries and the left term has non-zero at entry \((i, j)\) if nodes \(i, j\) are within distance \(C\) away, as \(q\) has support on \(\lceil C \rceil\) -- for reference Facebook network has an average distance of 4 \cite{Backstrom2012} i.e. yielding \(T^q\) with \(O(n^2)\) nonzero entries. Nonetheless, Section \(\S 4\) presents a framework for decomposing \(\hat{M}\) from its symbolic representation, without explicitly computing its entries. Before moving forward, we note that one can replace \(T\) in Eq. \(2\) by its symmetrically normalized counterpart \(\tilde{A}\), recovering a basis where \(L = R\). This symmetric modeling might be empirically preferred for undirected graphs. Learning can be performed via SVD. Specifically, the node at the \(ith\) row and the node at the \(jth\) column will be embedded, respectively, in \(L_i\) and \(R_j\) computed as:

\[
U, S, V \leftarrow \text{SVD}_{k}(\hat{M}^{(NE)}); \quad L \leftarrow US^\frac{1}{2}; \quad R \leftarrow VS^\frac{1}{2}\tag{5}
\]

In this \(k\)-dim space of rows and columns, Euclidean measures are plausible: Inference of nodes’ similarity at row \(i\) and column \(j\) can be modeled as \(f(i, j) = (L_i, R_j) = U_i^T SV_j = \langle U_i, V_j \rangle_S\).

3.2 Convexification of message passing graph networks

Removing all \(\sigma\)’s from Eq. \(2\) and setting \(g(A) = \hat{A}\) gives outputs of layers 1, 2, and \(L\), respectively, as:

\[
\hat{A}XW^{(1)} \quad , \quad \hat{A}^2XW^{(2)} \quad , \quad \text{and} \quad \hat{A}^LXW^{(1)}W^{(2)} \ldots W^{(L)}. \tag{6}
\]

Without non-linearities, adjacent parameter matrices can be absorbed into one another. Further, the model output can concatenate all layers, like JKNets \cite{Xu2018}, giving final model output of:

\[
\hat{H}^{(NC)}_{\text{linearized}} = \left[ X \mid \hat{A}X \mid \hat{A}^2X \mid \ldots \mid \hat{A}^LX \right] \hat{W} = \hat{M}^{(NC)}\hat{W}, \tag{7}
\]

where the linearized model implicitly constructs design matrix \(\hat{M}^{(NC)} \in \mathbb{R}^{n \times F}\) and multiplies it with parameter \(\hat{W} \in \mathbb{R}^{F \times \nu}\) -- here, \(F = d + dL\). Crafting design matrices is a creative process \(\S 5.3\).

Learning can be performed by minimizing the Frobenius norm: \(\|H^{(NC)} - Y\|_F = \|\hat{M}^{(NC)}\hat{W} - Y\|_F\). Moore-Penrose Inverse (a.k.a. the psuedoinverse) provides one such minimizer:

\[
\hat{W}^* = \text{argmin}_{\hat{W}} \left\| \hat{M}\hat{W} - Y \right\|_F = \hat{M}^\dagger Y \approx VS^+U^T Y, \tag{8}
\]

with \(U, S, V \leftarrow \text{SVD}_{k}(M)\). Notation \(S^+\) reciprocates non-zero entries of diagonal \(S\) \cite{Golub1996}. Multiplications in the right-most term should, for efficiency, be executed right-to-left.

\(^3\mathbb{E}_q[T_i]\) is a distribution on \([n]\): entry \(j\) equals prob. of walk starting at \(i\) ending at \(j\) if walk length \(\sim U[C]\).
Figure 1: Symbolic Matrix Representation. **Left:** code using our framework to implicitly construct the design matrix $\hat{M} = \hat{M}^{(NE)}$ with our framework. **Center:** DAG corresponding to the code. **Right:** An equivalent automatically-optimized DAG (via lazy-cache, Fig. 2) requiring fewer floating point operations. The first 3 lines of code create explicit input matrices (that fit in memory): adjacency $A$, diagonal degree $D$, and transition $T$. Matrices are imported into our framework with $F.\text{leaf}$ (depicted on computation DAGs in blue). Our classes overloads standard methods (+, -, *, 0, **) to construct computation nodes (intermediate in grey). The output node (in red) needs not be exactly calculated yet can be efficiently multiplied by any matrix by recursive downward traversal.

The pseudoinverse $\hat{M}^\dagger \approx V S^+ U^\top$ recovers the $\hat{W}^*$ with least norm (§6 Theorem 1). The $\approx$ becomes $=$ when $k \geq \text{rank}(\hat{M})$.

In semi-supervised settings, one can take rows subset of either (i) $Y$ and $U$, or of (ii) $Y$ and $M$, keeping only rows that correspond to labeled nodes. Option (i) is supported by existing frameworks (e.g., tf.gather()) and our symbolic framework §4 supports (ii) by implicit row (or column) gather – i.e., calculating SVD of submatrix of $M$ without explicitly computing $M$ nor the submatrix. **Inference** over a (possibly new) graph $(A, X)$ can be calculated by (i) (implicitly) creating the design matrix $\hat{M}$ corresponding to $(A, X)$ then (ii) multiplying by the explicitly calculated $\hat{W}^*$. As explained in §4 $\hat{M}$ need not to be explicitly calculated for computing multiplications.

### 4 Symbolic matrix representation

To compute the SVD of any matrix $M$ using algorithm prototypes presented by Halko et al. [2009], it suffices to provide functions that can multiply arbitrary vectors with $M$ and $M^\top$, and explicit calculation of $M$ is not required. Our software framework can symbolically represent $M$ as a directed acyclic graph (DAG) of computations. On this DAG, each node can be one of two kinds:

1. **Leaf node** (no incoming edges) that explicitly holds a matrix. Multiplications against leaf nodes are directly executed via an underlying math framework (we utilize TensorFlow).
2. **Symbolic node** that only implicitly represents a matrix as as a function of other DAG nodes. Multiplications are recursively computed, traversing incoming edges, until leaf nodes.

For instance, suppose leaf DAG nodes $M_1$ and $M_2$, respectively, explicitly contain row vector $\in \mathbb{R}^{1 \times n}$ and column vector $\in \mathbb{R}^{n \times 1}$. Then, their (symbolic) product DAG node $M = M_2 \otimes M_1$ is $\in \mathbb{R}^{n \times n}$. Although storing $M$ explicitly requires $O(n^2)$ space, multiplications against $M$ can remain within $O(n)$ space if efficiently implemented as $\langle M, . \rangle = \langle M_2, \langle M_1, . \rangle \rangle$. Figure 1 shows code snippet for composing DAG to represent symbolic node $M^{(NE)}$ (Eq. 5), from leaf nodes initialized with in-memory matrices. Appendix lists symbolic nodes and their implementations.

### 5 SVD initialization for deeper models fine-tuned via cross-entropy

#### 5.1 Edge function for network embedding as a (1-dimensional) Gaussian kernel

SVD provides decent solutions to link prediction tasks. Computing $U, S, V \leftarrow \text{SVD}(M^{(NE)})$ is much faster than training SOTA models for link prediction, yet, simple edge-scoring function $f(i, j) = \langle U_i, V_j \rangle_{\mathbb{R}^d}$ yields competitive empirical (test) performance. We propose $f$ with $\theta = \{\mu, s\}$:

$$f_{\mu,s}(i, j) = \mathbb{E}_{x \sim \mathcal{N}(\mu, s)} \langle U_i, V_j \rangle_{\mathbb{R}^d} = U_i^\top \mathbb{E}_x [S^*] V_j = U_i^\top \left( \int_\Omega S^* \mathcal{N}(x | \mu, s) \, dx \right) V_j,$$

$$f(i, j) = \mathbb{E}_{x \sim \mathcal{N}(\mu, s)} \langle U_i, V_j \rangle_{\mathbb{R}^d} = U_i^\top \mathbb{E}_x [S^*] V_j = U_i^\top \left( \int_\Omega S^* \mathcal{N}(x | \mu, s) \, dx \right) V_j,$$

$$f_{\mu,s}(i, j) = \mathbb{E}_{x \sim \mathcal{N}(\mu, s)} \langle U_i, V_j \rangle_{\mathbb{R}^d} = U_i^\top \mathbb{E}_x [S^*] V_j = U_i^\top \left( \int_\Omega S^* \mathcal{N}(x | \mu, s) \, dx \right) V_j,$$
where $\mathcal{N}$ is the truncated normal distribution (we truncate to $\Omega = [0.5, 2]$). The integral can be approximated by discretization and applying softmax (see §A.4). The parameters $\mu \in \mathbb{R}, \sigma \in \mathbb{R}_{>0}$ can be optimized on cross-entropy objective for link-prediction:

$$\min_{\mu, \sigma} - \mathbb{E}_{(i,j) \in \mathcal{A}} \left[ \log \left( \sigma(f_{\mu, s}(i, j)) \right) \right] - k_{(n)} \mathbb{E}_{(i,j) \notin \mathcal{A}} \left[ \log \left( 1 - \sigma(f_{\mu, s}(i, j)) \right) \right], \quad (10)$$

where the left- and right-terms, respectively, encourage $f$ to score high for edges, and the low for non-edges. $k_{(n)} \in \mathbb{N}_{>0}$ controls the ratio of negatives to positives per batch (we use $k_{(n)} = 10$). If the optimization sets $\mu = 1$ and $s \approx 0$, then $f$ reduces to no-op. In fact, we initialize it as such, and we observe that $f$ converges within one epoch, on graphs we experimented on. If it converges as $\mu < 1$, VS $\mu > 1$, respectively, then $f$ would effectively squash, VS enlarge, the spectral gap.

5.2 Split-ReLu (deep) graph network for node classification (NC)

$\hat{W}^*$ from SVD (Eq.8) can initialize an $L$-layer graph network with input: $H(0) = X$, with:

**message passing (MP)**

$$H^{(l+1)} = \left[ \hat{A} H^{(l)} W^{(l)}_{(p)} \right]_+ - \left[ \hat{A} H^{(l)} W^{(l)}_{(n)} \right]_+, \quad (11)$$

**output**

$$H = \sum_{l=0}^{L} \left[ H^{(l)} W^{(l)}_{(op)} \right]_+ - \left[ H^{(l)} W^{(l)}_{(on)} \right]_+, \quad (12)$$

**initialize MP**

$$W^{(l)}_{(p)} \leftarrow I; \quad W^{(l)}_{(n)} \leftarrow -I; \quad (13)$$

**initialize output**

$$W^{(l)}_{(op)} \leftarrow \hat{W}^*[d_{(l+1)}]; \quad W^{(l)}_{(on)} \leftarrow -\hat{W}^*[d_{(l+1)}]; \quad (14)$$

Element-wise $[\cdot]_+ = \max(0, \cdot)$. Further, $W_{[i,:;j]}$ denotes rows from $(i)^{th}$ until $(j-1)^{th}$ of $W$.

The deep network layers (Eq. [11][12]) use our Split-ReLu layer which we formalize as:

$$\text{SplitReLu}(X; W_{(p)}, W_{(n)}) = [X W_{(p)}]_+ - [X W_{(n)}]_+, \quad (15)$$

where the subtraction is calculated entry-wise. The layer has twice as many parameters as standard fully-connected (FC) ReLu layer. In fact, learning algorithms can recover FC ReLu from SplitReLu by assigning $W_{(n)} = 0$. More importantly, the layer behaves as linear in $X$ when $W_{(p)} = -W_{(n)}$. On this hyperplane, this linear behavior allows us to establish the equivalency: the (non-linear) model $H$ is equivalent to the linear $H_{\text{linearized}}$ at initialization (Eq. [13][14] due to Theorem 2. Following the initialization, model can be fine-tuned on cross-entropy objective as in §2.2.

5.3 Creative Add-ons for node classification (NC) models

**Label re-use (LR):** Let $\tilde{M}^{(NC)}_{LR} \triangleq \left[ \tilde{M}^{(NC)}_{\text{train}} \right] \left[ (\hat{A} - (D + I)^{-1})Y_{\text{train}} \right] \left[ (\hat{A} - (D + I)^{-1})^2 Y_{\text{train}} \right]$. This follows the motivation of Wang and Leskovec [2020], Huang et al. [2021], Wang [2021] and their empirical results on ogbn-arxiv dataset, where $Y_{\text{train}} \in \mathbb{R}^{N \times p}$ contains one-hot vectors at rows corresponding to labeled nodes but contain zero vectors for unlabeled (test) nodes. Our scheme is similar to concatenating $Y_{\text{train}}$ into $X$, but with care to prevent label leakage from row $i$ of $Y$ to row $i$ of $\tilde{M}$, as we zero-out the diagonal of the adjacency multiplied by $Y_{\text{train}}$.

**Pseudo-Dropout (PD):** Dropout [Srivastava et al., 2014] reduces overfitting of models. It can be related to data augmentation, as each example is presented multiple times. At each time, it appears with a different set of dropped-out features — input or latent feature values, chosen at random, get replaced with zeros. As such, we can replicate the design matrix as: $\hat{M}^\top \leftarrow \left[ \hat{M}^\top \right] \left[ \text{PD}(\hat{M})^\top \right]$. This row-wise concatenation maintains the width of $\hat{M}$ and therefore the number of model parameters.

In the above add-ons, concatenations, as well as PD, can be implicit or explicit (see §A.3).
Table 1: Dataset Statistics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
<th>Source</th>
<th>Task</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPI</td>
<td>3,852 proteins</td>
<td>20,881 chem. interactions</td>
<td>node2vec</td>
<td>LP</td>
<td>X</td>
</tr>
<tr>
<td>FB</td>
<td>4,039 users</td>
<td>88,234 friendships</td>
<td>SNAP</td>
<td>LP</td>
<td>X</td>
</tr>
<tr>
<td>AstroPh</td>
<td>17,903 researchers</td>
<td>197,031 co-authorships</td>
<td>SNAP</td>
<td>LP</td>
<td>X</td>
</tr>
<tr>
<td>HepTh</td>
<td>8,638 researchers</td>
<td>24,827 co-authorships</td>
<td>SNAP</td>
<td>LP</td>
<td>X</td>
</tr>
<tr>
<td>Cora</td>
<td>2,708 articles</td>
<td>5,429 citations</td>
<td>Planetoid</td>
<td>SSC</td>
<td>✔</td>
</tr>
<tr>
<td>Citeseer</td>
<td>3,327 articles</td>
<td>4,732 citations</td>
<td>Planetoid</td>
<td>SSC</td>
<td>✔</td>
</tr>
<tr>
<td>Pubmed</td>
<td>19,717 articles</td>
<td>44,338 citations</td>
<td>Planetoid</td>
<td>SSC</td>
<td>✔</td>
</tr>
<tr>
<td>ogbn-ArXiv</td>
<td>169,343 papers</td>
<td>1,166,243 citations</td>
<td>OGB</td>
<td>SSC</td>
<td>✔</td>
</tr>
<tr>
<td>ogbl-DDI</td>
<td>4,267 drugs</td>
<td>1,334,889 interactions</td>
<td>OGB</td>
<td>LP</td>
<td>✔</td>
</tr>
</tbody>
</table>

6 Analysis & Discussion

Theorem 1. (Min. Norm) If system \( \hat{M}\hat{W} = Y \) is underdetermined\(^6\) with rows of \( \hat{M} \) being linearly independent, then solution space \( \hat{W}^* = \{ \hat{W} | \hat{M}\hat{W} = Y \} \) has infinitely many solutions. Then, for \( k \geq \text{rank}(\hat{M}) \), matrix \( \hat{W}^* \), recovered by Eq.8 satisfies: \( \hat{W}^* = \arg\min_{\hat{W} \in \hat{W}^*} ||\hat{W}||_F^2 \).

Theorem 1 implies that, even though one can design a wide \( \hat{M}^{(NC)} \) (Eq.7), i.e., with many layers, the recovered parameters with least norm should be less prone to overfitting. Recall that this is the goal of L2 regularization. Analysis and proofs are in the Appendix.

Theorem 2. (Non-linear init) The initialization Eq. 13 & 14 yields \( H^{(NC)} \) linearized \( = H \) via Eq. 13 & 14. Cross-entropy objective can then fine-tune \( \theta \).

This end-to-end process, of (i) computing SVD bases and (ii) training the network \( f_\theta \) on singular values, advances SOTA on competitive benchmarks, with (i) converging (quickly) to a unique solution and (ii) containing merely a few parameters \( \theta \) — see §7.

7 Applications & Experiments

We download and experiment on 9 datasets summarized in Table1

We attempt link prediction (LP) tasks on smaller graph datasets (< 1 million edges) of: Protein-Protein Interactions (PPI) graph from Grover and Leskovec [2016]; as well as ego-Facebook (FB), AstroPh, HepTh from Stanford SNAP [Leskovec and Krevl, 2014]. For these datasets, we use the train-test splits of Abu-El-Haija et al. [2018]. We also attempt semi-supervised node classification (SSC) tasks on smaller graphs of Cora, Citeseer, Pubmed, all obtained from Planetoid [Yang et al., 2016]. For these smaller datasets, we only train and test using the SVD basis (without finetuning).

Further, we attempt on slightly-larger datasets (> 1 million edges) from Stanford’s Open Graph Benchmark [OGB, Hu et al. 2020]. We use the official train-test-validation splits and evaluator of OGB. We attempt LP and SSC, respectively, on Drug Drug Interactions (ogbl-DDI) and ArXiv citation network (ogbn-ArXiv). For these larger datasets, we use the SVD basis as an initialization that we finetune, as described in §5. For time comparisons, we train all models on Tesla K80.

7.1 Test performance & runtime on smaller datasets from: Planetoid & Stanford SNAP

For SSC over Planetoid’s datasets, both \( A \) and \( X \) are given. Additionally, only a handful of nodes are labeled. The goal is to classify the unlabeled test nodes. Table 2 summarizes the results. For baselines, we download code of GAT [Veličković et al., 2018], MixHop [Abu-El-Haija et al., 2019], GCNII [Chen et al., 2020] and re-ran them with instrumentation to record training time. However, for baselines Planetoid [Yang et al., 2016] and GCN [Kipf and Welling, 2017], we copied numbers

\(^6\)E.g., if the number of labeled examples i.e. height of \( M \) and \( Y \) is smaller than the width of \( M \).
Table 2: Test accuracy (& train time) on citation graphs for task: *semi-supervised node classification*.

<table>
<thead>
<tr>
<th>Graph dataset:</th>
<th>Baselines:</th>
<th></th>
<th>Our models:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>acc</td>
<td>tr.time</td>
<td>acc</td>
<td>tr.time</td>
</tr>
<tr>
<td>Planetoid</td>
<td>75.7</td>
<td>(13s)</td>
<td>64.7</td>
<td>(26s)</td>
</tr>
<tr>
<td>GCN</td>
<td>81.5</td>
<td>(4s)</td>
<td>70.3</td>
<td>(7s)</td>
</tr>
<tr>
<td>GAT</td>
<td>83.2</td>
<td>(1m)</td>
<td>72.4</td>
<td>(3m)</td>
</tr>
<tr>
<td>MixHop</td>
<td>81.9</td>
<td>(26s)</td>
<td>71.4</td>
<td>(31s)</td>
</tr>
<tr>
<td>GCNII</td>
<td>85.5</td>
<td>(2m)</td>
<td>73.4</td>
<td>(3m)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iSVD_{100}(\hat{M}^{(NC)})</td>
<td>82.0 ± 0.13 (0.1s)</td>
<td>71.4 ± 0.22 (0.1s)</td>
<td>78.9 ± 0.31 (0.3s)</td>
<td></td>
</tr>
<tr>
<td>+ dropout</td>
<td>82.5 ± 0.46 (0.1s)</td>
<td>71.5 ± 0.53 (0.1s)</td>
<td>78.9 ± 0.59 (0.2s)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Test ROC-AUC (& train time) on Stanford SNAP graphs for task: *link prediction*.

<table>
<thead>
<tr>
<th>Graph dataset:</th>
<th>Baselines:</th>
<th></th>
<th>Our models:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AUC</td>
<td>tr.time</td>
<td>AUC</td>
<td>tr.time</td>
</tr>
<tr>
<td>WYS</td>
<td>99.4</td>
<td>(54s)</td>
<td>97.9</td>
<td>(32m)</td>
</tr>
<tr>
<td>n2v</td>
<td>99.0</td>
<td>(30s)</td>
<td>97.8</td>
<td>(2m)</td>
</tr>
<tr>
<td>NetMF</td>
<td>97.6</td>
<td>(5s)</td>
<td>96.8</td>
<td>(9m)</td>
</tr>
<tr>
<td>NetMF</td>
<td>97.0</td>
<td>(4s)</td>
<td>81.9</td>
<td>(4m)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iSVD_{32}(\hat{M}^{(NE)})</td>
<td>99.1 ± 1e-6 (0.2s)</td>
<td>94.4 ± 4e-4 (0.5s)</td>
<td>90.5 ± 0.1 (0.1s)</td>
<td>89.3 ± 0.01 (0.1s)</td>
</tr>
<tr>
<td>iSVD_{256}(\hat{M}^{(NE)})</td>
<td>99.3 ± 9e-6 (2s)</td>
<td>98.0 ± 0.01 (7s)</td>
<td>90.1 ± 0.54 (2s)</td>
<td>89.3 ± 0.48 (1s)</td>
</tr>
</tbody>
</table>

From [Kipf and Welling, 2017]. For **our models**, the row labeled iSVD_{100}(\hat{M}^{(NC)}) , we run our implicit SVD twice per graph. The first run incorporates structural information: we (implicitly) construct \( \hat{M}^{(NE)} \) with \( \lambda = 0.05 \) and \( C = 3 \), then obtain \( [L, R] \leftarrow \text{SVD}_{\text{diag}}(\hat{M}^{(NE)}) \) per Eq. 5. Then, we concatenate \( L \) and \( R \) into \( X \). Then, we PCA the resulting matrix to 1000 dimensions, which forms our new \( X \). The second SVD run is to train the classification model parameters \( \hat{W} \). From the PCA-ed \( X \), we construct the implicit matrix \( \hat{M}^{(NC)} \) with \( L = 15 \) layers and obtain \( \hat{W}^* = VS^T [U^TY_{\text{[train]}} \text{per in RHS of Eq. 8} \) for our second “+ dropout” model variant, we (implicit) augment the data by \( \hat{M}^{(NC)} = [\hat{M}^{(NC)}] \to [\hat{M}^{(NC)}] \text{PD}([\hat{M}^{(NC)}]) \to \hat{W}^* = VS^T [U^TY_{\text{[train]}} \text{then similarly learn as: SVD}_{100}(\hat{M}^{(NC)}_{\text{[train]}}) \to \hat{W}^* = VS^T [U^TY_{\text{[train]}} \text{Discussion: Our method is competitive yet trains faster than SOTA. In fact, the only method that reliably beats ours on all dataset is GCNII, but its training time is about one-thousand-times longer. For LP over SNAP and node2vec datasets, the training adjacency A is given but not X. The split includes test positive and negative edges, which are used to measure a ranking metric: ROC-AUC, where the metric increases when test positive edges are ranked above than negatives. Table 3 summarizes the results. For **baselines**, we download code of WYS [Abu-El-Haija et al., 2018] and we use the efficient implementation of PyTorch-Geometric [Fey and Lenssen, 2019] for node2vec (n2v) and we download code of [Qiu et al., 2018] and run it with their two variants, denoting their first variant as NetMF (for exact, explicitly computing the design matrix) as and their second variant as NetMF (for approximate, sampling matrix entry-wise) – their code runs SVD after computing either variant. For the first of **our models**, we compute \( U, S, V \leftarrow \text{SVD}_{32}(\hat{M}^{(NE)}) \) and score every test edge as \( U^YSV_j \). For the second, we first run SVD_{256} on half of the training edges, determine the “best” rank \( \in \{8, 16, 32, 128, 256\} \) by measuring the AUC on the remaining half of training edges, then using this best rank, recompute the SVD on the entire training set, then finally score the test edges. **Discussion:** Our method is competitive on SOTA while training much faster. Both NetMF and WYS explicitly calculate a dense matrix before factorization. On the other hand, NetMF approximates the matrix entry-wise, trading accuracy for training time. In our case, we have the best...
We summarize experiments ogbl-DDI and ogbn-ArXiv, respectively, in Tables 4 and 5. For baselines, we copy numbers from the public leaderboard, where the competition is fierce. We then follow links to avoid computing sub-expressions if already calculated (§A.3.4). Speed-ups are shown in Fig. 2.

### 7.2 Experiments on Stanford’s OGB datasets

We summarize experiments ogbl-DDI and ogbn-ArXiv, respectively, in Tables 4 and 5. For baselines, we copy numbers from the public leaderboard, where the competition is fierce. We then follow links to avoid computing sub-expressions if already calculated (§A.3.4). Speed-ups are shown in Fig. 2.

For our models on ogbl-DDI, we (a) first calculate \( \mathbf{U}, \mathbf{S}, \mathbf{V} \leftarrow \text{SVD}_{100}(\mathbf{M}^{\text{NE}}) \) built only from training edges and score test edge \((i, j)\) using \( \mathbf{U}^\top_{\text{train}} \mathbf{S} \mathbf{V}_j \). Then, we (b) then finetune \( f_{\mu,s}(\mathbf{s}) \) (per §3.1 Eq. 9 & 10) for only a single epoch and score using \( f_{\mu,s}(i, j) \). Then, we (c) update the SVD basis to include edges from the validation partition and also score using \( f_{\mu,s}(i, j) \). We report the results for the three steps. For the last step, the rules of OGB allows using the validation set for training, but only after the hyperparameters are finalized. The SVD has no hyperparameters (except the rank, which was already determined by the first step). More importantly, this simulates a realistic situation: it is cheaper to obtain SVD (of an implicit matrix) than back-propagate through a model. For a time-evolving graph, one could run the SVD more often than doing gradient-descent epochs on a model. For our models on ogbn-ArXiv, we (a) compute \( \mathbf{U}, \mathbf{S}, \mathbf{V} \leftarrow \text{SVD}_{250}(\mathbf{M}_{LR}^{\text{NC}}) \) where the implicit matrix is defined in §5.3. We (b) repeat this process where we replicate \( \mathbf{M}_{LR}^{\text{NC}} \) once: in the second replica, we replace the \( \mathbf{Y}_{\text{train}} \) matrix with zeros (as-if, we drop-out the label with 50% probability). We (c) repeat the process where we concatenate two replicas of \( \mathbf{M}_{LR}^{\text{NC}} \) into the design matrix, each with different dropout seed. We (d) fine-tune the last model over 15 epochs using stochastic GTTF (Markowitz et al. [2021]). Discussion: Our method competes or sets SOTA, while training much faster.

#### Time improvements

We replaced the recommended orthonormalization of [Halko et al., 2009] from QR decomposition, to Cholesky decomposition (§A.2). Further, we implemented caching to avoid computing sub-expressions if already calculated (§A.3.4). Speed-ups are shown in Fig. 2.

### Table 4: Test Hits@20 for link prediction over Drug-Drug Interactions Network (ogbl-ddi).

<table>
<thead>
<tr>
<th>Baselines:</th>
<th>ogbl-DDI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Graph dataset:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DEA+JKNet</td>
<td>Yang et al., 2021</td>
</tr>
<tr>
<td>LRGa+n2v</td>
<td>Hsu and Chen, 2021</td>
</tr>
<tr>
<td>MAD</td>
<td>Luo et al., 2021</td>
</tr>
<tr>
<td>LRGa+GCN</td>
<td>Puny et al., 2020</td>
</tr>
<tr>
<td>GCN+JKNet</td>
<td>Xu et al., 2018</td>
</tr>
<tr>
<td>Our models:</td>
<td></td>
</tr>
<tr>
<td>(a) iSVD_{100}(\mathbf{M}^{\text{NE}})</td>
<td></td>
</tr>
<tr>
<td>(b) + finetune ( f_{\mu,s}(\mathbf{s}) ) (§3.1 Eq. 9 &amp; 10) sets ( \mu = 1.15 )</td>
<td></td>
</tr>
<tr>
<td>(c) + update ( \mathbf{U}, \mathbf{S}, \mathbf{V} ) (on validation, keeps ( f_{\mu,s} ) fixed)</td>
<td></td>
</tr>
</tbody>
</table>

### Table 5: Test classification accuracy over ogbn-arxiv.

<table>
<thead>
<tr>
<th>Baselines:</th>
<th>ogbn-ArXiv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Graph dataset:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>GAT+1R+KD</td>
<td>Ren, 2020</td>
</tr>
<tr>
<td>GAT+LR</td>
<td>Niu, 2020</td>
</tr>
<tr>
<td>AGDN</td>
<td>Sun and Wu, 2020</td>
</tr>
<tr>
<td>GAT+c&amp;S</td>
<td>Huang et al., 2021</td>
</tr>
<tr>
<td>GCNII</td>
<td>Chen et al., 2020</td>
</tr>
<tr>
<td>Our models:</td>
<td></td>
</tr>
<tr>
<td>(a) iSVD_{250}(\mathbf{M}_{LR}^{\text{NC}}) (§3.2 Eq. 8)</td>
<td></td>
</tr>
<tr>
<td>(b) + dropout(LR) (§5.3)</td>
<td></td>
</tr>
<tr>
<td>(c) + dropout(\mathbf{M}_{LR}^{\text{NC}}) (§5.3)</td>
<td></td>
</tr>
<tr>
<td>(d) + finetune ( \mathbf{H} ) (§5.2 Eq. 11)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: SVD runtime configs of (lazy caching; orthonormalization) as a ratio of SVD’s common default (QR decomposition)
8 Related work

Applications: SVD was used to project rows & columns of matrix \( M \) onto an embedding space. \( M \) can be the Laplacian of a homogenous graph [Belkin and Niyogi, 2003], Adjacency of user-item bipartite graph [Koren et al., 2009], or stats [Deerwester et al., 1990, Levy and Goldberg, 2014] for word-to-document. We differ: our \( M \) is a function of (leaf) matrices — useful when \( M \) is expensive to store (e.g., quadratic). While Qiu et al. [2018] circumvents this by entry-wise sampling the (otherwise \( n^2\)-dense) \( M \), our SVD implementation could decompose exactly \( M \) without calculating it.

Symbolic Software Frameworks: including Theano [Al-Rfou et al., 2016], TensorFlow [Abadi et al., 2016] and PyTorch [Paszke et al., 2019], allow chaining operations to compose a computation (directed acyclic) graph (DAG). They can efficiently run the DAG upwards by evaluating (all entries of) matrix \( M \in \mathbb{R}^{r \times c} \) at any DAG node. Our DAG differs: instead of calculating \( M \), it provides product function \( u_M(.) = M \times . \). – The graph is run downwards (reverse direction of edges).

Matrix-free SVD: For many matrices of interest, multiplying against \( M \) is computationally cheaper than explicitly storing \( M \) entry-wise. As such, many researchers implement \( SVD(u_M) \), e.g. Calvetti et al. [1994], Wu and Simon [2000], Bose et al. [2019]. We differ in the programming flexibility: the earlier methods expect the practitioner to directly implement \( u_M \). On the other hand, our framework allows composition of \( M \) via operations native to the practitioner (e.g., @, +, concat), and \( u_M \) is automatically defined.

Fast Graph Learning: We have applied our framework for fast graph learning, but so as sampling-based approaches including [Chen et al., 2018, Chiang et al., 2019, Zeng et al., 2020, Markowitz et al., 2021]. We differ in that ours can be used to obtain an initial closed-form solution (very quickly) and can be fine-tuned afterwards using any of the aforementioned approaches. Additionally, Graphs shows one general application. Our framework might be useful in other areas utilizing SVD.

9 Conclusion, our limitations & possible negative societal impact

We develop a software framework for symbolically representing matrices and compute their SVD. Without computing gradients, this trains convexified models over GRL tasks, showing empirical metrics competitive with SOTA while training significantly faster. Further, convexified model parameters can initialize (deeper) neural networks that can be fine-tuned with cross entropy. Practitioners adopting our framework would now spend more effort in crafting design matrices instead of running experiments or tuning hyperparameters of the learning algorithm. We hope our framework makes high performance graph modeling more accessible by reducing reliance on energy-intensive computation.

Limitations of our work: From a representational prospective, our space of functions is smaller than TensorFlow’s, as our DAG must be a linear transformation of its input matrices, e.g., unable to encode element-wise transformations and hence demanding first-order linear approximation of models. As a result, our gradient-free learning can be performed using SVD. Further, our framework only works when the leaf nodes (e.g., sparse adjacency matrix) fit in memory of one machine.

Possible societal impacts: First, our convexified models directly learn parameters in the feature space i.e. they are more explainable than deeper counterparts. Explainability is a double-edged sword: it gives better ability to interpret the model’s behavior, but also allows for malicious users, e.g., to craft attacks on the system (if they can replicate the model parameters). Further, we apply our method on graphs. It is possible to train our models to detect sensitive attributes of social networks (e.g., ethnicity). However, such ethical concerns exists with any modeling technique over graphs.

Acknowledgments and Disclosure of Funding

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and the Army Contracting Command- Aberdeen Proving Grounds (ACC-APG) under Contract Number W911NF-18-C-0020.
References


Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. In International Conference on Web Search and Data Mining, WSDM, pages 459–467, 2018.


A Appendix

A.1 Hyperparameters

We list our hyperparameters so that others can replicate our results. Nonetheless, our code has a README.md file with instructions on how to run our code per dataset.

A.1.1 Implicit matrix hyperparameters

Planetoid datasets [Yang et al., 2016] (Cora, CiteSeer, PubMed): we (implicitly) represent $\hat{M}^{(NE)}$ with negative coefficient $\lambda = 0.05$ and context window $C = 3$; we construct $\hat{M}^{(NC)}$ with number of layers $L = 15$. In one option, we use no dropout (first line of “our models” in Table 2) and in another (second line), we use dropout as described in §5.3.

SNAP and node2vec datasets (PPI, FB, AstroPh, HepTh): we (implicitly) represent $\hat{M}^{(NE)}$ with negative coefficient $\lambda = 0.02$ and context window $C = 10$.

Stanford OGB Drug-Drug Interactions (ogbl-DDI): we (implicitly) represent $\hat{M}^{(NE)}$ with negative coefficient $\lambda = 1$ and context window $C = 5$.

Stanford OGB ArXiv (ogbn-ArXiv): Unlike Planetoid datasets, we see in arXiv that appending the decomposition of $\hat{M}^{(NE)}$ onto $X$ does not improve the validation performance. Nonetheless, we construct $\hat{M}^{(NC)}$ with $L = 2$: we apply two forms of dropout, as explained in §7.2.

Justifications: For Stanford OGB datasets, we make the selection based on the validation set. For the other (smaller) datasets, we choose them based on one datasets in each scenario (Cora for node classification and PPI for link prediction) and keep them fixed, for all datasets in the scenario – hyperparameter tuning (on validation) per dataset may increase test performance, however, our goal is to make a general framework that helps across many GRL tasks, more-so than getting bold on specific tasks.

A.1.2 Finetuning hyperparameters

For Stanford OGB datasets (since competition is challenging), we use SVD to initialize a network that is finetuned on cross-entropy objectives. For ogbl-DDI, we finetune model in §5.1 using Adam optimizer with 1,000 positives and 10,000 negatives per batch, over one epoch, with learning rate of $1e^{-2}$. For ogbn-ArXiv, we finetune model in §5.2 using Adam optimizer using GTTF [Markowitz et al., 2021] with fanouts=[4, 4] (i.e., for every labeled node in batch, GTTF samples 4 neighbors, and 4 of their neighbors). With batch size 500, we train with learning rate $= 1e^{-3}$ for 7 epochs and with learning rate $= 1e^{-4}$ for 8 epochs (i.e., 15 epochs total).

A.2 SVD Implementation

The prototype algorithm for SVD described by [Halko et al., 2009] suggests using QR-decomposition to for orthonormalizing, but we rather use Cholesky decomposition which is faster to calculate (our framework is written on top of TensorFlow 2.0 [Abadi et al., 2016]). Fig. 2 in the main paper shows the time comparison.

A.3 Symbolic representation

Each node is a python class instance, that inherits base-class SymbolicPF (where PF stands for product function. Calculating SVD of (implicit) matrix $M \in \mathbb{R}^{r \times c}$ is achievable, without explicitly knowing entries of $M$, if we are able to multiply $\hat{M}$ by arbitrary $G$.

1. shape() property must return a tuple of the shape of the underlying implicit matrix e.g. = $(r, c)$.
2. dot(G) must right-multiply an arbitrary explicit matrix $G \in \mathbb{R}^{c \times \cdot}$ with (implicit) $\hat{M}$ as $\hat{M} \times G$ returning explicit matrix $\in \mathbb{R}^{r \times \cdot}$.

$^7 Q \leftarrow \text{orthonorm}(Q)$ yields $\hat{Q}$ such that $\hat{Q}^T \hat{Q} = I$ and $\text{span}(\hat{Q}) = \text{span}(Q)$. 

14
Algorithm 1 Computes rank-\(k\) SVD of implicit matrix \(\hat{\mathbf{M}}\), defined symbolically as in §A.3. Follows prototype of Halko et al. [2009] but uses an alternative routine for orthonormalization.

1: input: SVD rank \(k \in \mathbb{N}_+\); implicit matrix \(\hat{\mathbf{M}} \in \mathbb{R}^{r \times c}\) implementing functions in §A.3.
2: procedure iSVD(\(\hat{\mathbf{M}}, k\))
3: \((r, c) \leftarrow \hat{\mathbf{M}}.\text{shape}()\)  ▷ Random matrix sampled from standard normal. Shape: \((c \times 2k)\)
4: \(Q \leftarrow \sim \mathcal{N}(0, 1)_{c \times 2k}\)
5: for \(i \leftarrow 1 \text{ to iterations} \) do
6: \(Q \leftarrow \text{orthonorm}(\hat{\mathbf{M}}.\text{dot}(Q))\)  ▷ \((r \times 2k)\)
7: \(Q \leftarrow \text{orthonorm}(\hat{\mathbf{M}}.T().\text{dot}(Q))\)  ▷ \((c \times 2k)\)
8: \(B \leftarrow \hat{\mathbf{M}}.T().\text{dot}(Q)^T\)  ▷ \((2k \times c)\)
9: \(Q \leftarrow \text{orthonorm}(\hat{\mathbf{M}}.\text{dot}(Q))\)  ▷ \((r \times 2k)\)
10: \(U, s, V^T \leftarrow \text{tf.linalg.svd}(B)\)
11: \(U \leftarrow Q \times U\)
12: return \([U[:, k], s[: k], V[:, k]]\)


1: input: matrix \(Q\).
2: procedure orthonormHalko(Q)
3: \(\hat{Q}, R \leftarrow \text{QRdecomposition}(Q)\)
4: return \(\hat{Q}\)

1: input: matrix \(Q\).
2: procedure orthonormOurs(Q)
3: \(L \leftarrow \text{Choleskydecomposition}(Q^TQ)\)
4: return \(\hat{Q} \leftarrow Q \times (L^{-1})^T\)

3. \(\text{T()}\) must return an instance of SymbolicPF which is the (implicit) transpose of \(\hat{\mathbf{M}}\) i.e. with \(.\text{shape}() = (c, r)\)

The names of the functions were purposefully chosen to match common naming conventions, such as of numpy. We now list our current implementations of PF classes used for symbolically representing the design matrices discussed in the paper (\(\hat{\mathbf{M}}^{(\text{NE})}\) and \(\hat{\mathbf{M}}^{(\text{NC})}\)).

A.3.1 Leaf nodes

The leaf nodes explicitly hold the underlying matrix. Next, we show an implementation for a dense matrix (left), and another for a sparse matrix (right):

```
class DenseMatrixPF(SymbolicPF):
    """(implicit) matrix is explicitly stored dense tensor."""
    def __init__(self, m):
        self.m = tf.convert_to_tensor(m)
    def dot(self, g):
        return tf.matmul(self.m, g)
    @property
    def shape(self):
        return self.m.shape
    @property
    def T(self):
        return DenseMatrixPF(tf.transpose(self.m))

class SparseMatrixPF(SymbolicPF):
    """(implicit) matrix is explicitly stored dense tensor."""
    def __init__(self, m):
        self.m = scipy_csr_to_tf_sparse(m)
    def dot(self, g):
        return tf.sparse.sparse_dense_matmul(self.m, g)
    @property
    def shape(self):
        return self.m.shape
    @property
    def T(self):
        return SparseMatrixPF(tf.sparse.transpose(self.m))
```
A.3.2 Symbolic nodes

Symbolic nodes implicitly hold a matrix. Specifically, their constructors (__init__) accept one-or-more other (leaf or) symbolic nodes and their implementations of .shape() and .dot() invokes those of the nodes passed to their constructor. Let us take a few examples:

```python
class SumPF(SymbolicPF):
    def __init__(self, pfs):
        self.pfs = pfs
        for pf in pfs:
            assert pf.shape == pfs[0].shape

def dot(self, m):
    sum_ = self.pfs[0].dot(m)
    for pf in self.pfs[1:]:
        sum_ += pf.dot(m)
    return sum_

@property
def T(self):
    return SumPF([f.T for f in self.pfs])

@property
def shape(self):
    return self.pfs[0].shape
```

```python
class ProductPF(SymbolicPF):
    def __init__(self, pfs):
        self.pfs = pfs
        for i in range(len(pfs) - 1):
            assert pfs[i].shape[1] == pfs[i+1].shape[0]

@property
def shape(self):
    return (self.pfs[0].shape[0], self.pfs[-1].shape[1])

@property
def T(self):
    return ProductPF(reversed([f.T for f in self.pfs]))

def dot(self, m, cache=None):
    product = m
    for pf in reversed(self.pfs):
        product = pf.dot(product)
    return product
```

Our attached code (will be uploaded to github, after code review process) has concrete implementations for other PFs. For example, GatherRowsPF and GatherColsPF, respectively, for (implicitly) selecting row and column slices; as well as, concatenating matrices, and multiplying by scalar. Further, the actual implementation has further details than displayed, for instance, for implementing lazy caching §A.3.4.

A.3.3 Syntactic sugar

Additionally, we provide a couple of top-level functions e.g. `fsvd.sum()`, `fsvd.gather()`, with interfaces similar to numpy and tensorflow (but expects implicit PF as input and outputs the same). Further, we override python operators (e.g., +, -, *, **, @) on the base SymbolicPF that instantiates symbolic node implementations, as:

```python
class SymbolicPF:
    # ....
    def __add__(self, other):
        return SumPF([self, other])
    def __matmul__(self, other):
        return ProductPF([self, other])
    def __sub__(self, other):
        return SumPF([self, TimesScalarPF(-1, other)])
    def __mul__(self, scalar):
        return TimesScalarPF(scalar, self)
    def __rmul__(self, scalar):
        return TimesScalarPF(scalar, self)
    def __pow__(self, integer):
        return ProductPF([self] * int(integer))
```
This allows composing implicit matrices using syntax commonly-used for calculating explicit matrices. We hope this notation could ease the adoption of our framework.

A.3.4 Optimization by lazy caching

One can come up with multiple equivalent mathematical expressions. For instance, for matrices A, B, C, the expressions A × B + A^2C and A × (B + AC) are equivalent. However, the first one should be cheaper to compute.

At this point, we have not implemented such equivalency-finding: we do not substitute expressions with their more-efficient equivalents. Rather, we implement (simple) optimization by lazy-caching. Specifically, when computing product \( \hat{M}^\top G \), as the dot() methods are recursively called, we populate a cache (python dict) of intermediate products with key-value pairs:

- key: ordered list of (references of) SymbolicPF instances that were multiplied by \( G \);
- value: the product of the list times \( G \).

While this optimization scheme is suboptimal, and trades memory for computation, we observe up to 4X speedups on our tasks (refer to Fig. 2). Nonetheless, in the future, we hope to utilize an open-source expression optimizer, such as TensorFlow’s\cite{Lattner2021}.

A.4 Approximating the integral of Gaussian 1d kernel

The integral from Equation 9 can be approximated using softmax, as:

\[
\int_\Omega S^x \mathcal{N}(x | \mu, s) dx = \int_\Omega S^x \exp \left( -\frac{1}{2} \left( \frac{x - \mu}{s} \right)^2 \right) dx \approx \sum_{x \in \Omega} S^x \exp \left( -\frac{1}{2} \left( \frac{x - \mu}{s} \right)^2 \right) dx
\]

where the first equality expands the definition of truncated normal, i.e. divide by partition function, to make the mass within \( \Omega \) sum to 1. In our experiments, we use \( \Omega = [0.5, 2] \). The \( \approx \) comes as we use discretized \( \Omega = \{0.5, 0.505, 0.51, 0.515, \ldots, 2.0\} \) (i.e., with 301 entries). Finally, the last expression contains a constant tensor (we create it only once) containing \( S \) raised to every power in \( \Omega \), stored along (tensor) axis which gets multiplied (via tensor product i.e. broadcasting) against softmax vector (also of 301 entries, corresponding to \( \Omega \)). We parameterize with two scalars \( \mu, \sigma \in \mathbb{R} \) i.e. implying \( \sigma = \log \frac{1}{2s} \).

A.5 Analysis, theorems and proofs

A.5.1 Norm regularization of wide models

Proof of Theorem 1 is common in university-level linear algebra courses, but here for completeness. It implies that if \( \hat{M}^{\text{RC}} \) is too wide, then we need not to worry much about overfitting.

\textbf{Proof.} of Theorem 1

Assume \( Y = y \) is a column vector (the proof can be generalized to matrix \( Y \) by repeated column-wise application\footnote{Minimizer of Frobenius norm is composed, column-wise, of minimizers \( \arg\min_{\hat{W}_{:,j}} ||W_{:,j}||_2^2, \forall j \).}). SVD\( k(M) \) for \( k \geq \text{rank}(\hat{M}) \), recovers the solution:

\[
\hat{W}^* = \hat{M}^\top y = \hat{M}^\top (\hat{M}\hat{M}^\top)^{-1} y. \tag{16}
\]

The Gram matrix \( \hat{M}\hat{M}^\top \) is nonsingular as the rows of \( \hat{M} \) are linearly independent. To prove the claim let us first verify that \( \hat{M}W^* \in V^* \):

\[
\hat{M}W^* = \hat{M}\hat{M}^\top (\hat{M}\hat{M}^\top)^{-1} y = y.
\]
Let \( \hat{W}_p \in \hat{W}^* \). We must show that \( \|\hat{W}^*\|_2 \leq \|\hat{W}_p\|_2 \). Since \( \hat{M}\hat{W}_p = y \) and \( \hat{M}\hat{W}^* = y \), their subtraction gives:

\[
\hat{M}(\hat{W}_p - \hat{W}^*) = 0.
\]

It follows that \( (\hat{W}_p - \hat{W}^*) \perp \hat{W}^* \):

\[
(\hat{W}_p - \hat{W}^*)^\top \hat{W}^* = (\hat{W}_p - \hat{W}^*)^\top \hat{M}^\top (\hat{M}\hat{M}^\top)^{-1}y
\]

\[
= (\hat{M}(\hat{W}_p - \hat{W}^*))^\top (\hat{M}\hat{M}^\top)^{-1}y = 0
\]

Finally, using Pythagoras Theorem (due to \( \perp \)):

\[
\|\hat{W}_p\|_2^2 = \|\hat{W}^* + \hat{W}_p - \hat{W}^*\|_2^2
\]

\[
= \|\hat{W}^*\|_2^2 + \|\hat{W}_p - \hat{W}^*\|_2^2 \geq \|\hat{W}^*\|_2^2
\]

\( \square \)

A.5.2 At initialization, deep model is identical to linear (convexified) model

**Proof.** of Theorem\(^2\)

The layer-to-layer “positive” and “negative” weight matrices are initialized as: \( W^{(l)}_{(op)} = -W^{(l)}_{(on)} = I \). Therefore, at initialization:

\[
H^{(l+1)} = \left[ \hat{A}H^{(l)}W^{(l)}_{(op)} \right]_+ - \left[ \hat{A}H^{(l)}W^{(l)}_{(on)} \right]_+ = \left[ \hat{A}H^{(l)} \right]_+ - \left[ -\hat{A}H^{(l)} \right]_+
\]

\[
= I \left[ \hat{A}H^{(l)} \geq 0 \right] \circ \left( \hat{A}H^{(l)} \right) - I \left[ -\hat{A}H^{(l)} \geq 0 \right] \circ \left( -\hat{A}H^{(l)} \right)
\]

\[
= I \left[ \hat{A}H^{(l)} \geq 0 \right] \circ \left( \hat{A}H^{(l)} \right) + I \left[ -\hat{A}H^{(l)} \geq 0 \right] \circ \left( \hat{A}H^{(l)} \right)
\]

\[
= \left( I \left[ \hat{A}H^{(l)} \geq 0 \right] + I \left[ -\hat{A}H^{(l)} \geq 0 \right] \right) \circ \left( \hat{A}H^{(l)} \right)
\]

\[
= \hat{A}H^{(l)},
\]

where the first line comes from the initialization; the second line is an alternative definition of relu: the indicator function \( I \) is evaluated element-wise and evaluates to 1 in positions its argument is true and to 0 otherwise; the third line absorbs the two negatives into a positive; the fourth by factorizing; and the last by noticing that exactly one of the two indicator functions evaluates to 1 almost everywhere, except at the boundary condition i.e. at locations where \( \hat{A}H^{(l)} = 0 \) but there the right-term makes the Hadamard product 0 regardless. It follows that, since \( H^{(0)} = X \), then \( H^{(1)} = \hat{A}X \), \( H^{(2)} = \hat{A}^2X \), \( \ldots \), \( H^{(L)} = \hat{A}^LX \).

The layer-to-output positive and negative matrices are initialized as: \( W^{(l)}_{(op)} = -W^{(l)}_{(on)} = \hat{W}^{*}_{[dl:d(l+1)]} \). Therefore, at initialization, the final output of the model is:

\[
H = \sum_{l=0}^{L} \left[ H^{(l)}W^{(l)}_{(op)} \right]_+ - \left[ H^{(l)}W^{(l)}_{(on)} \right]_+ = \sum_{l=0}^{L} \left[ H^{(l)}\hat{W}^{*}_{[dl:d(l+1)]} \right]_+ - \left[ -H^{(l)}\hat{W}^{*}_{[dl:d(l+1)]} \right]_+
\]

\[
= \sum_{l=0}^{L} H^{(l)}\hat{W}^{*}_{[dl:d(l+1)]},
\]

where the first line comes from the definition and the initialization. The second line can be arrived by following exactly the same steps as above: expanding the re-writing the ReLu using indicator notation, absorbing the negative, factorizing, then lastly unioning the two indicators that are mutually exclusive.
almost everywhere. Finally, the last summation can be expressed as a block-wise multiplication between two (partitioned) matrices:

\[
H = \left[ H^{(0)} \mid H^{(1)} \mid \ldots \mid H^{(L)} \right] \begin{bmatrix}
\hat{W}^{[0:d]} \\
\hat{W}^{[d:2d]} \\
\vdots \\
\hat{W}^{[dL:d(L+1)]}
\end{bmatrix} = \left[ X \mid \hat{A}X \mid \ldots \mid \hat{A}^L X \right] \hat{W}^* = \hat{M}^{(NC)} \hat{W}^* = H^{(NC)}_{\text{linearized}}
\]

### A.5.3 Computational Complexity and Approximation Error

**Theorem 3.** (Linear Time) Implicit SVD (Alg. 7) trains our convexified GRL models in time linear in the graph size.

**Proof.** of Theorem 3 for our two model families:

1. For rank-\(k\) SVD of \(\hat{M}^{(NE)}\): Let cost of running \(\hat{M}^{(NC)}\).dot() be \(T_{\text{mult}}\). The run-time to compute SVD, as derived in Section 1.4.2 of [Halko et al., 2009], is:

\[
O(kT_{\text{mult}} + (r + c)k^2). \tag{18}
\]

Since \(\hat{M}^{(NE)}\) can be defined as \(C\) (context window size) multiplications with sparse \(n \times n\) matrix \(T\) with \(m\) non-zero entries, then running iSVD of \(\hat{M}^{(NE)}\) costs:

\[
O(kmC + nk^2) \tag{19}
\]

2. For rank-\(k\) SVD over \(\hat{M}^{(NC)}\): Suppose feature matrix contains \(d\)-dimensional rows. One can calculate \(\hat{M}^{(NC)} \in \mathbb{R}^{n \times Ld}\) with \(L\) sparse multiplies in \(O(Lmd)\). Calculating and running SVD of \(\hat{M}^{(KN)}\) costs total of:

\[
O(ndL \log(k) + (n + dL)k^2 + Lmd). \tag{20}
\]

Therefore, training time is linear in \(n\) and \(m\). □

Contrast with methods of WYS [Abu-El-Haija et al., 2018] and NetMF [Qiu et al., 2018], which require assembling a dense \(n \times n\) matrix requiring \(O(n^2)\) time to decompose. One wonders: how far are we from the optimal SVD with a linear-time algorithm? The following bounds the error.

**Theorem 4.** (Exponentially-decaying Approx. Error) Rank-\(k\) randomized SVD algorithm of [Halko et al., 2009] gives an approximation error that can be brought down, exponentially-fast, to no more than twice of the approximation error of the optimal (true) SVD.

**Proof.** of Theorem 4 is in Theorem 1.2 of [Halko et al., 2009] □

Consequently, compared to \(\hat{\text{NetMF}}\) of [Qiu et al., 2018], which incurs unnecessary estimation error as they sample the matrix entry-wise, our estimation error can be brought-down exponentially by increasing the \(\text{iterations}\) parameter of Alg. 1. In particular, as long as we can compute products against the matrix, we can decompose it, almost as good as if we had the individual matrix entries.