A Degeneracy Framework for Scalable Graph Autoencoders

Guillaume Salha[1,2] , **Romain Hennequin**[1] , **Viet Anh Tran**[1] and **Michalis Vazirgiannis**[2]
[1]Deezer Research & Development, Paris, France
[2]École Polytechnique, Palaiseau, France
research@deezer.com

Abstract

In this paper, we present a general framework to scale graph autoencoders (AE) and graph variational autoencoders (VAE). This framework leverages graph degeneracy concepts to train models only from a dense subset of nodes instead of using the entire graph. Together with a simple yet effective propagation mechanism, our approach significantly improves scalability and training speed while preserving performance. We evaluate and discuss our method on several variants of existing graph AE and VAE, providing the first application of these models to large graphs with up to millions of nodes and edges. We achieve empirically competitive results w.r.t. several popular scalable node embedding methods, which emphasizes the relevance of pursuing further research towards more scalable graph AE and VAE.

1 Introduction

Graphs have become ubiquitous in the Machine Learning community, thanks to their ability to efficiently represent the relationships among items in various disciplines. Social networks, biological molecules and communication networks are some of the most famous real-world examples of data usually represented as graphs. Extracting meaningful information from such structure is a challenging task, which has initiated considerable research efforts, aiming at tackling several learning problems such as link prediction, influence maximization and node clustering.

In particular, over the last decade there has been an increasing interest in extending and applying Deep Learning methods to graph structures. [Gori *et al.*, 2005; Scarselli *et al.*, 2009] firstly introduced graph neural network architectures, and were later joined by numerous contributions to generalize CNNs and the convolution operation to graphs, leveraging spectral graph theory [Bruna *et al.*, 2014], its approximations [Defferrard *et al.*, 2016; Kipf and Welling, 2016a] or spatial-based approaches [Hamilton *et al.*, 2017]. Attempts at extending RNNs, GANs, attention mechanisms or word2vec-like methods for node embeddings also recently emerged in the literature ; for complete references, we refer to [Wu *et al.*, 2019]'s survey on Deep Learning for graphs.

In this paper, we focus on the graph extensions of autoencoders and variational autoencoders. Introduced in the 1980's [Rumelhart *et al.*, 1986], autoencoders (AE) regained a significant popularity in the last decade through neural network frameworks [Baldi, 2012] as efficient tools to learn reduced encoding representations of input data in an unsupervised way. Furthermore, variational autoencoders (VAE) [Kingma and Welling, 2013], described as extensions of AE but actually based on quite different mathematical foundations, also recently emerged as a successful approach for unsupervised learning from complex distributions, assuming the input data is the observed part of a larger joint model involving low-dimensional latent variables, optimized via variational inference approximations. [Tschannen *et al.*, 2018] review the wide recent advances in VAE-based representation learning. In this paper we show that, during the last three years, many efforts have been devoted to the generalization of such models to graphs. Graph AE and VAE appear as elegant node embedding tools i.e. ways to learn a low dimensional vector space representation of nodes, with promising applications to link prediction, node clustering, matrix completion and graph generation. However, most existing models suffer from scalability issues and all existing experiments are limited to graphs with at most a few thousand nodes. The question of how to scale graph AE and VAE to larger graphs remains widely open, and we propose to address it in this paper. More precisely, our contribution is threefold:

- We introduce a general framework to scale graph AE and VAE models, by optimizing the reconstruction loss (for AE) or variational lower bound (for VAE) only from a dense subset of nodes, and then propagate representations in the entire graph. These nodes are selected using graph degeneracy concepts. Such approach considerably improves scalability while preserving performance.
- We apply this framework to large real-world data and discuss empirical results on ten variants of graph AE or VAE models for two learning tasks. To the best of our knowledge, this is the first application of these models to graphs with up to millions of nodes and edges.
- We show that these scaled models have competitive performances w.r.t. several popular scalable node embedding methods. It emphasizes the relevance of pursuing further research towards scalable graph autoencoders.

This paper is organized as follows. In Section 2, we provide an overview of graph AE/VAE and of their extensions, applications and limits. In Section 3, we present our degeneracy framework and how we reconstruct the latent space from an autoencoder only trained on a subset of nodes. We interpret our experimental analysis and discuss possible extensions of our approach in Section 4, and we conclude in Section 5.

2 Preliminaries

In this section, we recall some key concepts related to graph AE and VAE. Throughout this paper, we consider an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $|\mathcal{V}| = n$ nodes and $|\mathcal{E}| = m$ edges, without self-loops. We denote by A the adjacency matrix of $\mathcal{G}$, weighted or not. Nodes can possibly have features vectors of size d, stacked up in an $n \times d$ matrix X. Otherwise, X is the identity matrix I.

2.1 Graph Autoencoders (GAE)

In the last three years, several attempts at transposing autoencoders to graph structures with [Kipf and Welling, 2016b] or without [Wang *et al.*, 2016] node features have been presented. Their goal is to learn, in an unsupervised way, a low dimensional node embedding/latent vector space (*encoding*), from which reconstructing the graph topology (*decoding*) is possible. In its most general form, the $n \times f$ matrix Z of all latent space vectors z_i, where f is the dimension of the latent space, is the output of a Graph Neural Network (GNN) applied on A and, potentially, X. To reconstruct A from Z, one could resort to another GNN. However, [Kipf and Welling, 2016b] and several extensions of their model implement a simpler inner product decoder between latent variables, along with a sigmoid activation $\sigma(\cdot)$ or, if A is weighted, some more complex thresholding. The drawback of this simple decoding is that it involves the multiplication of the two dense matrices Z and Z^T, which has a quadratic complexity $O(fn^2)$ w.r.t. the number of nodes. To sum up, with $\hat{A}$ the reconstruction:

$$\hat{A} = \sigma(ZZ^T) \quad \text{with} \quad Z = \text{GNN(X,A)}.$$

The model is trained by minimizing the reconstruction loss $\|A - \hat{A}\|_F$ of the graph structure where $\|\cdot\|_F$ denotes the Frobenius matrix norm, or alternatively a weighted cross entropy loss, by stochastic gradient descent.

2.2 Graph Convolutional Networks (GCN)

[Kipf and Welling, 2016b], and a majority of following works, assume that the GNN encoder is a Graph Convolutional Network (GCN). Introduced by [Kipf and Welling, 2016a], GCNs leverage both 1) the features information X, and 2) the graph structure summarized in A. In a GCN with L layers, with $H^{(0)} = X$ and $H^{(L)} = Z$, each layer returns:

$$H^{(l+1)} = \text{ReLU}(D^{-1/2}(A + I)D^{-1/2}H^{(l)}W^{(l)})$$

i.e. it averages the feature vectors from $H^{(l)}$ of the neighbors of a given node (and itself, thus the I), with a ReLU activation $\text{ReLU}(x) = \max(x, 0)$. D denotes the diagonal degree matrix of $A + I$, so $D^{-1/2}(A + I)D^{-1/2}$ is its symmetric normalization. ReLU is absent from output layer. Weights matrices $W^{(l)}$, of potentially different dimensions, are trained by stochastic gradient descent. Implementing GCN encoders is mainly driven by complexity purposes. Indeed, the cost of computing each hidden layer is linear w.r.t. m [Kipf and Welling, 2016a], and its training efficiency can also be improved via importance sampling [Chen *et al.*, 2018]. However recent works, e.g. [Xu *et al.*, 2019], highlight some fundamental limits of the simple GCN heuristics. It incites to resort to more powerful albeit more complex GNN encoders, such as [Bruna *et al.*, 2014] computing actual spectral graph convolutions, a model later extended by [Defferrard *et al.*, 2016], approximating smooth filters in the spectral domain with Chebyshev polynomials (GCN being a faster first-order approximation of [Defferrard *et al.*, 2016]). In this paper, we show that our scalable degeneracy framework adequately facilitates the training of such more complex encoders.

2.3 Variational Graph Autoencoders (VGAE)

[Kipf and Welling, 2016b] also introduced Variational Graph Autoencoders (VGAE). They assume a probabilistic model on the graph structure involving some latent variables z_i of length f for each node $i \in \mathcal{V}$, later interpreted as latent representations of nodes in an embedding space of dimension f. More precisely, with Z the $n \times f$ latent variables matrix, the inference model (*encoder*) is defined as $q(Z|X, A) = \prod_{i=1}^{n} q(z_i|X, A)$ where $q(z_i|X, A) = \mathcal{N}(z_i|\mu_i, \text{diag}(\sigma_i^2))$. Parameters of Gaussian distributions are learned using two two-layer GCN. Therefore, μ, the matrix of mean vectors μ_i, is defined as $\mu = \text{GCN}_\mu(X, A)$. Also, $\log\sigma = \text{GCN}_\sigma(X, A)$, and both GCNs share the same weights in first layer. Then, as for GAE, a generative model (*decoder*) aiming at reconstructing A is defined as the inner product between latent variables: $p(A|Z) = \prod_{i=1}^{n}\prod_{j=1}^{n} p(A_{ij}|z_i, z_j)$ where $p(A_{ij} = 1|z_i, z_j) = \sigma(z_i^T z_j)$ and $\sigma(\cdot)$ is the sigmoid function. As explained for GAE, such reconstruction has a limiting quadratic complexity w.r.t. n. [Kipf and Welling, 2016b] optimize weights of GCN by maximizing a tractable variational lower bound (ELBO) of the model's likelihood:

$$\mathcal{L} = \mathbb{E}_{q(Z|X,A)}\Big[\log p(A|Z)\Big] - \mathcal{D}_{KL}(q(Z|X, A)||p(Z)),$$

where $\mathcal{D}_{KL}(\cdot, \cdot)$ is the Kullback-Leibler divergence. They perform full-batch gradient descent, using the *reparameterization trick* [Kingma and Welling, 2013], and choosing a Gaussian prior $p(Z) = \prod_i p(z_i) = \prod_i \mathcal{N}(z_i|0, I)$.

2.4 Applications, Extensions and Limits

GAE and VGAE have been successfully applied for various graph learning tasks, such as link prediction [Kipf and Welling, 2016b], clustering [Wang *et al.*, 2017] and matrix completion for recommendation [Berg *et al.*, 2018]. Extensions of these models also recently tackled multi-task learning problems [Tran, 2018], added adversarial training schemes enforcing the latent representation to match the prior [Pan *et al.*, 2018] or proposed RNN graph autoencoders to learn graph-level embeddings [Taheri *et al.*, 2018].

We also note the existence of several applications of graph VAE to biochemical data and small molecular graphs [Ma *et al.*, 2018]. Most of them put the emphasis on plausible

graph generation using the decoder. Among these works, [Simonovsky and Komodakis, 2018] introduced a model able to reconstruct both 1) the topological graph information, 2) node-level features, and 3) edge-level features. However, it involves a graph matching step in $O(n^4)$ complexity that, while being acceptable for molecules with tens of nodes, prevents the model to scale.

Overall, all existing experiments are restricted to small or medium-size graphs with up to a few thousand nodes and edges. Most models suffer from scalability issues, either by training complex GNN models or by using dense inner product decoding in $O(fn^2)$ complexity as in [Kipf and Welling, 2016b]. This problem has already been raised and partially addressed but without applications to large graphs. For instance, [Grover *et al.*, 2018] proposed Graphite that replaces the standard decoder by more scalable reverse message passing schemes, but only report results on [Kipf and Welling, 2016b]'s medium-size graphs (3K to 20K nodes). To sum up, graph AE and VAE showed very promising results on various tasks for small and medium-size datasets, but the question of their extension to very large graphs remains widely open.

3 Scaling up Graph AE/VAE with Degeneracy

In this section, we introduce a flexible framework, aiming at scaling existing graph autoencoders (variational or not) to large graphs. Here, we assume that nodes are featureless, i.e. that models only learn from the graph structure. Node features will be re-introduced in section 4.

3.1 Overview of the Framework

To deal with large graphs, the key idea of our framework is to optimize the reconstruction loss (for AE) or the variational lower bound (for VAE) only from a wisely selected subset of nodes, instead of using the entire graph $\mathcal{G}$ which would be intractable. More precisely, we proceed as follows:

1. Firstly, we identify the nodes on which the AE/VAE model should be trained, by computing a k-core decomposition of the graph. The selected subgraph is the so-called k-degenerate version of the original one. We justify this choice in section 3.2 and explain how we choose the value of k.
2. Then, we train a graph autoencoder (GAE, VGAE or any variant) on this k-degenerate subgraph. Hence, we only derive latent representation vectors (embeddings) for the nodes included in this subgraph.
3. Regarding the nodes of $\mathcal{G}$ that are not in this subgraph, we infer their latent representations using a simple and fast propagation heuristic, presented in section 3.3.

In a nutshell, training the autoencoder (step 2) still has a potentially high complexity, but now the input graph is much smaller, making the training tractable. Moreover, we will show that steps 1 and 3 have linear running times w.r.t. m. Therefore, our strategy significantly improves speed and scalability and, as we later experimentally verify, is able to effectively process large graphs with millions of nodes and edges.

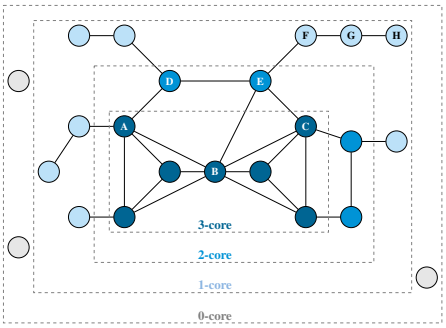

Figure 1: A graph $\mathcal{G}$ of degeneracy 3 and its cores. Some nodes are labeled for the purpose of section 3.3.

Algorithm 1 k-core Decomposition

Input: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: Set of k-cores $\mathcal{C} = \{\mathcal{C}_0, \mathcal{C}_1, ..., \mathcal{C}_{\delta^*(\mathcal{G})}\}$

Initialize $\mathcal{C} = \{\mathcal{V}\}$ and $k = \min_{v \in \mathcal{V}} d(v)$
for $i = 1$ **to** n **do**
  $v =$ node with smallest degree in $\mathcal{G}$
  if $d(v) > k$ **then**
    Append $\mathcal{V}$ to $\mathcal{C}$
    $k = d(v)$
  end if
  $\mathcal{V} = \mathcal{V} \setminus \{v\}$ and remove edges linked to v
end for

3.2 Graph Degeneracy

In this subsection, we detail the first step of our framework, i.e. the identification of a representative subgraph on which the autoencoder should be trained. Our method resorts to the k-core decomposition, a powerful tool to analyze the structure of a graph. Formally, the k-core, or k-degenerate version of graph $\mathcal{G}$, is the largest subgraph of $\mathcal{G}$ for which every node has a degree of at least k within the sub-graph. Therefore, in a k-core, each node is connected to at least k nodes, that are themselves connected to at least k nodes. Moreover, the degeneracy $\delta^*(\mathcal{G})$ of a graph is the maximum k for which the k-core is not empty. Nodes from each core k, denoted $\mathcal{C}_k \in \mathcal{V}$, form a nested chain i.e. $\mathcal{C}_{\delta^*(\mathcal{G})} \subseteq \mathcal{C}_{\delta^*(\mathcal{G})-1} \subseteq ... \subseteq \mathcal{C}_0 = \mathcal{V}$. Figure 1 illustrates an example of core decomposition.

In step 2, we therefore train an autoencoder, either only on the $\delta^*(\mathcal{G})$-degenerate version of $\mathcal{G}$, or on a larger k-degenerate subgraph, i.e. for a $k < \delta^*(\mathcal{G})$. Our justification for this strategy is twofold. The first reason is computational: the k-core decomposition can be computed in a linear running time for an undirected graph [Batagelj and Zaversnik, 2003]. More precisely, to construct a k-core, the strategy is to recursively remove all nodes with degree lower than k and their edges from $\mathcal{G}$ until no node can be removed, as described in Algorithm 1. It involves sorting nodes by degrees, in $O(n)$ time using a variant of bin-sort, and going through all nodes and edges once (see [Batagelj and Zaversnik, 2003] for details). Time complexity is $O(\max(m, n))$ with $\max(m, n) = m$ in most real-world graphs, and same space complexity with sparse matrices. Our second reason to rely on k-degenerate graphs is that, despite being simple, they

have been proven to be very useful tools to extract representative subgraphs over the past years, including for node clustering [Giatsidis *et al.*, 2014], keyword extraction in graph-of-words [Tixier *et al.*, 2016] and graph similarity via core-based kernels [Nikolentzos *et al.*, 2018]. We refer to [Malliaros *et al.*, 2019] for an exhaustive overview of the history, theory, extensions and applications of core decomposition.

On the Selection of k. To select k, one must face an inherent performance/speed trade-off, as illustrated in section 4. Besides, on large graphs, training AE/VAE is usually impossible on lowest cores due to overly large memory requirements. In our experiments, we adopt a simple strategy when dealing with large graphs, and train models on the lowest computationally tractable cores, i.e. on largest possible subgraphs. In practice, these subgraphs are significantly smaller than the original ones (at least 95% of nodes are removed). Moreover, when running experiments on medium-size graphs where all cores are tractable, we plainly avoid choosing $k < 2$ (since $\mathcal{V} = \mathcal{C}_0 = \mathcal{C}_1$, or $\mathcal{C}_0 \approx \mathcal{C}_1$, in all our graphs). Setting $k = 2$, i.e. removing *leaves* from the graph, empirically appears as a good option, preserving performances w.r.t. models trained on $\mathcal{G}$ while significantly reducing running times by pruning up to 50% of nodes in our graphs.

3.3 Propagation of Latent Representations

From steps 1 and 2, we computed latent representation vectors z_i of dimension f for each node i of the k-core. Step 3 is the inference of such representation for the remaining nodes of $\mathcal{G}$ in a scalable way. Nodes are assumed featureless so the only information we leverage comes from the graph structure. Our strategy starts by assigning representations to nodes directly connected to the k-core. We average the values of their embedded neighbors *and* of the nodes being embedded at the same step of the process. For instance, in the graph of Figure 1, to compute z_D and z_E we would solve the system $z_D = \frac{1}{2}(z_A + z_E)$ and $z_E = \frac{1}{3}(z_B + z_C + z_D)$ (or a weighted mean, if edges are weighted). Then, we repeat this process on the neighbors of these newly embedded nodes, and so on until no new node is reachable. Taking into account the fact that nodes D and E are themselves connected is important. Indeed, node A from the maximal core is also a second-order neighbor of E; exploiting such proximity when computing z_E empirically improves performance, as it also strongly impacts all the following nodes whose latent vectors will then be derived from z_E (in Figure 1, nodes F, G and H).

More generally, let $\mathcal{V}_1$ denote the set of nodes whose latent vectors are computed, $\mathcal{V}_2$ the set of nodes connected to $\mathcal{V}_1$ and without latent vectors, A_1 the $|\mathcal{V}_1| \times |\mathcal{V}_2|$ adjacency matrix linking $\mathcal{V}_1$ and $\mathcal{V}_2$'s nodes, and A_2 the $|\mathcal{V}_2| \times |\mathcal{V}_2|$ adjacency matrix of $\mathcal{V}_2$'s nodes. We normalize A_1 and A_2 by the total degree in $\mathcal{V}_1 \cup \mathcal{V}_2$, i.e. we divide rows by row sums of the $(A_1^T | A_2)$ matrix row-concatenating A_1^T and A_2. We denote by $\tilde{A}_1$ and $\tilde{A}_2$ these normalized versions. We already learned the $|\mathcal{V}_1| \times f$ latent representations matrix Z_1 for nodes in $\mathcal{V}_1$. To implement our strategy, we want to derive a $|\mathcal{V}_2| \times f$ representation matrix Z_2 for nodes in $\mathcal{V}_2$, verifying $Z_2 = \tilde{A}_1 Z_1 + \tilde{A}_2 Z_2$. The solution of this system is $Z^* = (I - \tilde{A}_2)^{-1} \tilde{A}_1 Z_1$, which exists since $(I - \tilde{A}_2)$ is strictly diagonally dominant are therefore invertible from Levy-Desplanques theorem. Unfortunately, the exact computation of Z^* has a cubic complexity. We approximate it by randomly initializing Z_2 with values in $[-1, 1]$ and iterating $Z_2 = \tilde{A}_1 Z_1 + \tilde{A}_2 Z_2$ until convergence to a fixed point, which is guaranteed to happen exponentially fast as stated below.

Algorithm 2 Propagation of Latent Representations

Input: Graph $\mathcal{G}$, list of embedded nodes $\mathcal{V}_1$, $|\mathcal{V}_1| \times f$ latent matrix Z_1 (already learned), number of iterations t
Output: Latent representations of each node in $\mathcal{G}$

$\mathcal{V}_2 =$ set of not-embedded nodes reachable from $\mathcal{V}_1$
while $|\mathcal{V}_2| > 0$ **do**
  $A_1 = |\mathcal{V}_1| \times |\mathcal{V}_2|$ adj. matrix linking $\mathcal{V}_1$ and $\mathcal{V}_2$ nodes
  $A_2 = |\mathcal{V}_2| \times |\mathcal{V}_2|$ adj. matrix of $\mathcal{V}_2$ nodes
  $\tilde{A}_1, \tilde{A}_2 =$ normalized A_1, A_2 by row sum of $(A_1^T | A_2)$
  Randomly initialize $|\mathcal{V}_2| \times f$ matrix Z_2 *(rows of Z_2 will be latent representation vectors of $\mathcal{V}_2$'s nodes)*
  for $i = 1$ **to** t **do**
    $Z_2 = \tilde{A}_1 Z_1 + \tilde{A}_2 Z_2$
  end for
  $\mathcal{V}_1 = \mathcal{V}_2$
  $\mathcal{V}_2 =$ set of not-embedded nodes reachable from $\mathcal{V}_1$
end while
Assign random vectors to remaining unreachable nodes

Theorem 1. *Let $Z^{(t)}$ the $|\mathcal{V}_2| \times f$ matrix obtained from iterating $Z^{(t)} = \tilde{A}_1 Z_1 + \tilde{A}_2 Z^{(t-1)}$ t times starting from $Z^{(0)}$. Let $\| \cdot \|_F$ the Frobenius norm. Then, exponentially fast,*

$$\|Z^{(t)} - Z^*\|_F \xrightarrow[t \to +\infty]{} 0$$

Proof. We have $Z^{(t)} - Z^* = [\tilde{A}_1 Z_1 + \tilde{A}_2 Z^{(t-1)}] - [\tilde{A}_2 Z^* + (I - \tilde{A}_2) Z^*] = \tilde{A}_1 Z_1 + \tilde{A}_2 Z^{(t-1)} - \tilde{A}_2 Z^* - (I - \tilde{A}_2)(I - \tilde{A}_2)^{-1} \tilde{A}_1 Z_1 = \tilde{A}_2 (Z^{(t-1)} - Z^*)$. So, $Z^{(t)} - Z^* = \tilde{A}_2^t (Z^{(0)} - Z^*)$. Then, as a consequence of Cauchy-Schwarz inequality: $\|Z^{(t)} - Z^*\|_F = \|\tilde{A}_2^t (Z^{(0)} - Z^*)\|_F \leq \|\tilde{A}_2^t\|_F \|Z^{(0)} - Z^*\|_F$. Futhermore, $\tilde{A}_2^t = P D^t P^{-1}$, with $\tilde{A}_2 = P D P^{-1}$ the eigen-decomposition of symmetric matrix $\tilde{A}_2$. For diagonal matrix D^t we have $\|D^t\|_F = \sqrt{\sum_{i=1}^{|\mathcal{V}_2|} |\lambda_i^t|^2} \leq \sqrt{|\mathcal{V}_2|} (\max_i |\lambda_i|)^t$ with λ_i the i-th eigenvalue of $\tilde{A}_2$. Since $\tilde{A}_2$ has non-negative entries, we derive from Perron–Frobenius theorem (see [Lovász, 2007]) that a) maximum absolute value among all eigenvalues of $\tilde{A}_2$ is reached by a nonnegative real eigenvalue, and b) that $\max_i \lambda_i$ is bounded above by the maximum degree in $\tilde{A}_2$'s graph. By definition, each node in $\mathcal{V}_2$ has at least one connection to $\mathcal{V}_1$; moreover rows of $\tilde{A}_2$ are normalized by row sums of $(A_1^T | A_2)$, so the maximum degree in $\tilde{A}_2$'s graph is strictly lower than 1. We conclude with a) and b) that $0 \leq |\lambda_i| < 1$ for all $i \in \{1, ..., |\mathcal{V}_2|\}$, so $0 \leq \max_i |\lambda_i| < 1$. This result implies that $\|D^t\|_F \to_t 0$ exponentially fast, and so does $\|\tilde{A}_2^t\|_F \leq \|P\|_F \|D^t\|_F \|P^{-1}\|_F$, then $\|Z^{(t)} - Z^*\|_F$. □

Our propagation process is summarized in Algorithm 2. If some nodes are unreachable by such process because $\mathcal{G}$ is

Model	Size of input k-core	Mean Perf. on Test Set (in %) AUC	AP	Mean Running Times (in sec.) k-core dec.	Model train	Propagation	Total	Speed gain
VGAE on $\mathcal{G}$	-	83.02 ± 0.13	$\mathbf{87.55 \pm 0.18}$	-	710.54	-	710.54	-
on 2-core	$9,277 \pm 25$	$\mathbf{83.97 \pm 0.39}$	85.80 ± 0.49	1.35	159.15	0.31	160.81	$\times 4.42$
on 3-core	$5,551 \pm 19$	$\mathbf{83.92 \pm 0.44}$	85.49 ± 0.71	1.35	60.12	0.34	61.81	$\times 11.50$
on 4-core	$3,269 \pm 30$	82.40 ± 0.66	83.39 ± 0.75	1.35	22.14	0.36	23.85	$\times 29.79$
on 5-core	$1,843 \pm 25$	78.31 ± 1.48	79.21 ± 1.64	1.35	7.71	0.36	9.42	$\times 75.43$
...	...	...	...	...	...	...	...	...
on 8-core	414 ± 89	67.27 ± 1.65	67.65 ± 2.00	1.35	1.55	0.38	**3.28**	$\times$**216.63**
on 9-core	149 ± 93	61.92 ± 2.88	63.97 ± 2.86	1.35	1.14	0.38	**2.87**	$\times$**247.57**
DeepWalk	-	81.04 ± 0.45	84.04 ± 0.51	-	342.25	-	342.25	-
LINE	-	81.21 ± 0.31	84.60 ± 0.37	-	63.52	-	63.52	-
node2vec	-	81.25 ± 0.26	85.55 ± 0.26	-	48.91	-	48.91	-
Spectral	-	83.14 ± 0.42	86.55 ± 0.41	-	31.71	-	31.71	-

Table 1: Link Prediction on Pubmed graph ($n = 19,717$, $m = 44,338$), using VGAE model, its k-core variants, and baselines

not connected, then we eventually assign them random latent vectors. Using sparse representations for $\tilde{A}_1$ and $\tilde{A}_2$, memory requirement is $O(m + nf)$, and the computational complexity of each evaluation of line 7 also increases linearly w.r.t. the number of edges m in the graph. Moreover, in practice t is small: we set $t = 10$ in our experiments (we illustrate the impact of t in Annex 2). The number of iterations in the while loop of line 2 corresponds to the size of the longest shortest-path connecting a node to the k-core, a number bounded above by the diameter of the graph which increases at a $O(\log(n))$ speed in most real-world graphs [Chakrabarti and Faloutsos, 2006]. In next section, we empirically check our claim that both steps 1 and 3 run linearly and therefore scale to large graphs with millions of nodes.

4 Empirical Analysis

In this section, we empirically evaluate our framework. Although all main results are presented here, we report additional and more complete tables in supplementary material[1].

4.1 Experimental Setting

Datasets. We provide experiments on the three medium-size graphs used in [Kipf and Welling, 2016b]: Cora ($n = 2,708$ and $m = 5,429$), Citeseer ($n = 3,327$ and $m = 4,732$) and Pubmed ($n = 19,717$ and $m = 44,338$), and on two large graphs from Stanford's SNAP project: the Google web graph ($n = 875,713$ and $m = 4,322,051$) and the US Patent citation networks ($n = 2,745,762$ and $m = 13,965,410$). Details, statistics and full k-core decompositions of these graphs are reported in Annex 1. Cora, Citeseer and Pubmed's nodes have bag-of-words features. Graphs are unweighted and we ignore edges' potential directions.

Tasks. We consider two learning tasks. The first one, as in [Kipf and Welling, 2016b], is a *link prediction* task. We train models on incomplete versions of graphs where some edges were randomly removed. We create validation and test sets from removed edges and from the same number of randomly sampled pairs of unconnected nodes, and check the model's ability to classify edges (i.e. the true $A_{ij} = 1$) from non-edges ($A_{ij} = 0$) via the reconstructed value $\hat{A}_{ij} = \sigma(z_i^T z_j)$. Validation and test sets gather 5% and 10% of edges (respectively 2% and 3%), for medium-size (resp. large-size) graphs. The incomplete train adjacency matrix is used when running Algorithm 2. Validation set is only used for model tuning. We compare performances using *Area Under the Receiver Operating Characteristic (ROC) Curve* (AUC) and *Average Precision* (AP) scores. The second task is *node clustering* from latent representations z_i. More precisely, we run k-means in embedding spaces, compare clusters to ground-truth communities and report normalized *Mutual Information* (MI) scores.

Models. We apply our degeneracy framework to ten graph autoencoders: the seminal two-layer GAE and VGAE models [Kipf and Welling, 2016b], two deeper variants of GAE/VGAE with two GCN hidden layers, Graphite and Variational Graphite [Grover *et al.*, 2018], [Pan *et al.*, 2018]'s adversarially regularized models (denoted ARGA and ARVGA), ChebAE and ChebVAE i.e. two variants of GAE/VGAE with ChebNets [Defferrard *et al.*, 2016] of order 3 instead of GCN. We omit models designed for small molecular data. All models are trained on 200 epochs to return 16-dim embeddings (32-dim for Patent) to reproduce [Kipf and Welling, 2016b]'s results. We also compare to DeepWalk [Perozzi *et al.*, 2014], LINE [Tang *et al.*, 2015] and node2vec [Grover and Leskovec, 2016] node embeddings. We focus on these methods because they directly claim scalability. For each model, hyperparameters were tuned on AUC scores using validation set (see Annex 2 for details). We also implemented a spectral decomposition baseline (embedding axis are first eigenvectors of $\mathcal{G}$'s Laplacian matrix) and, for node clustering, Louvain's method [Blondel *et al.*, 2008]. We used Python and especially the Tensorflow library, training models on a NVIDIA GTX 1080 GPU and running other operations on a double Intel Xeon Gold 6134 CPU.

4.2 Results

Medium-Size Graphs. For Cora, Citeseer and Pubmed, we apply our framework to all possible subgraphs from 2-core to $\delta^*(\mathcal{G})$-core and on entire graphs, which is still tractable. Table 1 reports mean AUC and AP and their standard errors on 100

[1]Supplementary material in: https://arxiv.org/abs/1902.08813

Model (using framework, k=17)	Perf. on Test Set (in %) AUC	AP	Total run. time
GAE	94.02 ± 0.20	94.31 ± 0.21	23min
VGAE	93.22 ± 0.40	93.20 ± 0.45	**22 min**
DeepGAE	93.74 ± 0.17	92.94 ± 0.33	24min
DeepVGAE	93.12 ± 0.29	92.71 ± 0.29	24min
Graphite	93.29 ± 0.33	93.11 ± 0.42	23min
Var-Graphite	93.13 ± 0.35	92.90 ± 0.39	**22 min**
ARGA	93.82 ± 0.17	94.17 ± 0.18	23min
ARVGA	93.00 ± 0.17	93.38 ± 0.19	23min
ChebGAE	$\mathbf{95.24 \pm 0.26}$	$\mathbf{96.94 \pm 0.27}$	41min
ChebVGAE	95.03 ± 0.25	96.58 ± 0.21	40min
node2vec on $\mathcal{G}$ *(best baseline)*	94.89 ± 0.63	96.82 ± 0.72	4h06

Table 2: Link Prediction on Google graph ($n = 875K, m = 4,3M$) using our framework on 17-core ($|\mathcal{C}_{17}| = 23,787 \pm 208$) on graph AE/VAE variants.

Model (using framework, k=15)	Perf. on Test Set (in %) Normalized MI	Total run. time
GAE	23.76 ± 2.25	56min
VGAE	24.53 ± 1.51	**54min**
DeepGAE	24.27 ± 1.10	1h01
DeepVGAE	24.54 ± 1.23	58min
Graphite	24.22 ± 1.45	59min
Var-Graphite	24.25 ± 1.51	58min
ARGA	24.26 ± 1.18	1h01
ARVGA	24.76 ± 1.32	58min
ChebGAE	25.23 ± 1.21	1h41
ChebVGAE	$\mathbf{25.30 \pm 1.22}$	1h38
node2vec on $\mathcal{G}$ *(best baseline)*	24.10 ± 1.64	7h15

Table 3: Node Clustering on Patent graph ($n = 2,7M$, $m = 13,9M$) using our framework on 15-core ($|\mathcal{C}_{14}| = 35,432$) on graph AE/VAE variants.

runs (train incomplete graphs and masked edges are different for each run) along with mean running times, for *link prediction* task with VGAE on Pubmed. Sizes of k-cores vary over runs due to the edge masking process in *link prediction*; this phenomenon does not occur for *node clustering* task. Overall, our framework significantly improves running times w.r.t. training VGAE on $\mathcal{G}$. Running time decreases when k increases (up to $\times 247.57$ speed gain in Table 1), which was expected since the k-core is smaller. We observe this improvement on all other datasets, on both tasks, and for GAE and all GAE/VGAE variants (see Annex 2 and 3). Also, for low cores, especially for the 2-core subgraphs, performances are consistently competitive w.r.t. models trained on entire graphs, and sometimes better both for *link prediction* (e.g. $+0.95$ point in AUC for 2-core in Table 1) and *node clustering*. It highlights the relevance of our propagation process, and the fact that training models on smaller graphs is easier. Choosing higher cores leads to even faster training, at the price of decreasing performance scores.

Large Graphs. Table 2 details *link prediction* results on Google from 17-core and for all autoencoders variants. Also, in Table 3 we display *node clustering* results on Patent, whose ground-truth clusters are six roughly balanced patent categories, reporting performances from all autoencoders variants trained on 15-core. Core numbers were selected according to section 3's tractability criterion. Scores are averaged over 10 runs. Overall, we reach similar conclusions w.r.t. medium-size graphs, both in terms of good performance and of scalability. However, comparison with full models on $\mathcal{G}$, i.e. without using our framework, is impossible on these graphs due due to overly large memory requirements. We therefore compare performances on several computationally tractable cores (see Annex 2 and 3 for complete tables), illustrating once again the inherent performance/speed trade-off when choosing k and validating previous insights.

Graph AE/VAE Variants. For both tasks, we note that adversarial training from ARGA/ARGVA and Graphite's decoding tend to slightly improve predictions, as well as ChebNet-based models that often stand out in terms of AUC, AP and MI (e.g. a top 95.24 AUC for ChebGAE in Table 2). It indicates the relevance of replacing GCN by more complex encoders, which is facilitated by our framework.

Baselines. Our core variants are competitive w.r.t. baselines. They are significantly faster on large graphs while achieving comparable or outperforming performances in most experiments, which emphasizes the interest of scaling graph AE and VAE. Futhermore, we specify that 64 dimensions were needed to reach stable performing results on baselines, against 16 for autoencoders. This suggests that graph autoencoders are more suitable to encode information in low dimensional embeddings. On the other hand, baselines, notably Louvain and node2vec, are better to cluster nodes in Cora and Pubmed ($+10$ points in MI for Louvain on Cora) which questions the global ability of existing graph AE/VAE to identify clusters in a robust way.

Extensions and Openings. Based on this last finding, future works on graph VAE will investigate alternative prior distributions designed to detect communities in graphs. Moreover, while this paper mainly considered featureless nodes, we note that our method easily extends to attributed graphs, since we can add node features from the k-core subgraph as input of GAE/VGAE models. In this direction, we also report experiments on GAE and VGAE *with node features* (when available) for both tasks in supplementary materials, significantly improving scores (e.g. from 85.24 to 88.10 AUC for 2-core GAE on Cora). However, node features are not included in step 3's propagation : future works will study more efficient features integrations. Last, we also aim at obtaining theoretical guarantees on k-core approximations, and at extending existing approaches to directed graphs.

5 Conclusion

We introduced a degeneracy-based framework to easily scale graph (variational) autoencoders, and provided experimental evidences of its ability to effectively process large graphs. Our work confirms the representational power of these models, and identifies several directions that, in future research, should lead towards their improvement.

References

[Baldi, 2012] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. *ICML workshop on unsupervised and transfer learning*, 2012.

[Batagelj and Zaversnik, 2003] Vladimir Batagelj and Matjaz Zaversnik. An o(m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[Berg *et al.*, 2018] Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph convolutional matrix completion. *KDD Deep Learning day*, 2018.

[Blondel *et al.*, 2008] Vincent Blondel, Jean-Loup Guillaume, and Etienne. Lefebvre. Fast unfolding of communities in large networks. *J. Stat. Mech*, 2008.

[Bruna *et al.*, 2014] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. Spectral networks and locally connected networks on graphs. *ICLR*, 2014.

[Chakrabarti and Faloutsos, 2006] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM computing surveys*, 38(1):2, 2006.

[Chen *et al.*, 2018] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *ICLR*, 2018.

[Defferrard *et al.*, 2016] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *NIPS*, 2016.

[Giatsidis *et al.*, 2014] Christos Giatsidis, Fragkiskos Malliaros, Dimitrios Thilikos, and Michalis Vazirgiannis. Corecluster: A degeneracy based graph clustering framework. *AAAI*, 2014.

[Gori *et al.*, 2005] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. *IJCNN*, 2005.

[Grover and Leskovec, 2016] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *SIGKDD*, 2016.

[Grover *et al.*, 2018] Aditya Grover, Aaron Zweig, and Stefano Ermon. Graphite: Iterative generative modeling of graphs. *arXiv preprint arXiv:1803.10459*, 2018.

[Hamilton *et al.*, 2017] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *NIPS*, 2017.

[Kingma and Welling, 2013] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *ICLR*, 2013.

[Kipf and Welling, 2016a] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *ICLR*, 2016.

[Kipf and Welling, 2016b] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *NIPS Workshop on Bayesian Deep Learning*, 2016.

[Lovász, 2007] László Lovász. Eigenvalues of graphs. *Technical report, Eotvos Lorand University*, 2007.

[Ma *et al.*, 2018] Tengfei Ma, Jie Chen, and Cao Xiao. Constrained generation of semantically valid graphs via regularizing variational autoencoders. *NeurIPS*, 2018.

[Malliaros *et al.*, 2019] Fragkiskos Malliaros, Christos Giatsidis, Apostolos Papadopoulos, and Michalis Vazirgiannis. The core decomposition of networks: Theory, algorithms and applications. 2019.

[Nikolentzos *et al.*, 2018] Giannis Nikolentzos, Polykarpos Meladianos, Stratis Limnios, and Michalis Vazirgiannis. A degeneracy framework for graph similarity. *IJCAI*, 2018.

[Pan *et al.*, 2018] Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, Lina Yao, and Chengqi Zhang. Adversarially regularized graph autoencoder. *IJCAI*, 2018.

[Perozzi *et al.*, 2014] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *SIGKDD*, 2014.

[Rumelhart *et al.*, 1986] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. *Parallel Distributed Processing, Vol 1*, 1986.

[Scarselli *et al.*, 2009] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *Neural Networks*, 20(1):61–80, 2009.

[Simonovsky and Komodakis, 2018] Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using vae. *ICANN*, 2018.

[Taheri *et al.*, 2018] Aynaz Taheri, Kevin Gimpel, and Tanya Berger-Wolf. Learning graph representations with recurrent neural network autoencoders. *KDD DL Day*, 2018.

[Tang *et al.*, 2015] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. *WWW*, 2015.

[Tixier *et al.*, 2016] Antoine Tixier, Fragkiskos Malliaros, and Michalis Vazirgiannis. A graph degeneracy-based approach to keyword extraction. *EMNLP*, 2016.

[Tran, 2018] Phi Vu Tran. Learning to make predictions on graphs with autoencoders. *DSAA*, 2018.

[Tschannen *et al.*, 2018] Michael Tschannen, Olivier Bachem, and Mario Lucic. Recent advances in autoencoder-based representation learning. *NeurIPS Bayesian DL workshop*, 2018.

[Wang *et al.*, 2016] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. *SIGKDD*, 2016.

[Wang *et al.*, 2017] Chun Wang, Shirui Pan, Guodong Long, Xingquan Zhu, and Jing Jiang. Mgae: Marginalized graph autoencoder for graph clustering. *CIKM*, 2017.

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

[Xu *et al.*, 2019] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *ICLR*, 2019.