A Degeneracy Framework for Scalable Graph Autoencoders

Guillaume Salha\textsuperscript{1,2}, Romain Hennequin\textsuperscript{1}, Viet Anh Tran\textsuperscript{1} and Michalis Vaziriannis\textsuperscript{2}

\textsuperscript{1}Deezer Research & Development, Paris, France
\textsuperscript{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 to tackle 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. [Gorë et al., 2005; Scarselli et al., 2009] firstly introduced graph neural networks 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 to extend 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 networks 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. 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 on 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 $G = (V, E)$ with $|V| = n$ nodes and $|E| = m$ edges, without self-loops. We denote by $A$ the adjacency matrix of $G$, weighted or not. Nodes can possibly have features vectors of size $d$, summarized 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 to transpose 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 $n \times m$ matrix $Z$ of all latent space vectors $z_i$, where $m$ 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 $O(n^2)$ 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 by stochastic gradient descent, where $\|\cdot\|_F$ denotes the Frobenius matrix norm.

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 $D^{-1/2}(A + f)D^{-1/2}$ is its symmetric normalization. Weights matrices $W^{(l)}$, of potentially different dimensions, are trained by backpropagation. Implementing GCN encoders is mainly driven by complexity purposes. Indeed, the cost of computing each hidden layer is linear w.r.t. $n$ [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 $i \in V$, 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 $q(z_i|X,A) = \mathcal{N}(z_i|\mu_i, \sigma(z_i)^2)$. Parameters of Gaussian distributions are learned using two-layer GCN. Therefore, $\mu$, the matrix of mean vectors $\mu_i$, 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 to reconstruct $A$, but not $X$, 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:

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

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 on 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 [Liu et al., 2018; Ma et al., 2018; Jin 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 $O(n^3)$ 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 $O(f n^2)$ 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). Moreover, [Samanta et al., 2018] introduced NeVAE, a more flexible model for graph generation, but applications are focused on molecular graphs. 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 $G$ 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 $G$ 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.

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 $G$, is the largest subgraph of $G$ 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 $\delta^*(G)$ of a graph is the maximum $k$ for which the $k$-core is not empty. Nodes from each core $k$, denoted $C_k \subseteq V$, form a nested chain i.e. $C_k \subseteq C_{k+1} \subseteq \ldots \subseteq C_{\delta^*(G)} = V$. Figure 1 illustrates an example of core decomposition.

In step 2, we therefore train an autoencoder, either only on the $\delta^*(G)$-degenerate version of $G$, or on a larger $k$-degenerate subgraph, i.e. for a $k < \delta^*(G)$. 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 $G$ 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

![Figure 1: A graph $G$ of degeneracy 3 and its cores. Some nodes are labeled for the purpose of section 3.3.](image-url)
have been proven to be very useful tools to extract representa-
tive subgraphs over the past years, including for node clustering [Giat-
sidis 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 in-
herent performance/speed trade-off, as illustrated in section 4. Besides, on large graphs, training AE/VAE is usually im-
possible on lowest cores due to overly large memory re-
motives. 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 sub-
graphs. 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 \( V = C_0 = C_1 \), or \( C_0 \approx C_1 \), in all our graphs). Setting \( k = 2 \), i.e. removing leaves from the graph, empirically ap-
pears as a good option, preserving performances w.r.t. mod-
els trained on \( G \) 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 vec-
ors \( z_i \) of dimension \( f \) for each node \( i \) of the \( k \)-core. Step 3 is the inference of such representation for the remaining nodes of \( G \) 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 embed-
ded 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 \( z_E = \frac{1}{2} (z_B + z_C + z_D) \) (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 im-
acts 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 \( V_1 \) denote the set of nodes whose latent vectors are computed, \( V_2 \) the set of nodes connected to \( V_1 \) and without latent vectors, \( A_1 \) the \( |V_1| \times |V_2| \) adjacency matrix linking \( V_1 \) and \( V_2 \)'s nodes, and \( A_2 \) the \( |V_2| \times |V_2| \) adjacency matrix of \( V_2 \) nodes. We normalize \( A_1 \) and \( A_2 \) by the total degree in \( V_1 \cup V_2 \), i.e. we divide rows by row sums of the \( A_1^T \) matrix row-concatenating \( A_1^T \) and \( A_2 \). We denote by \( 'A_1 \) and \( 'A_2 \) these normalized versions. We al-
ready learned the \( |V_1| \times f \) latent representations matrix \( Z_1 \) for nodes in \( V_1 \). To implement our strategy, we want to de-
rive a \( |V_2| \times f \) representation matrix \( Z_2 \) for nodes in \( V_2 \), verifying \( Z_2 = 'A_1 Z_1 + 'A_2 Z_2 \). The solution of this sys-
tem is \( Z^* = (I - 'A_2)^{-1} 'A_1 Z_1 \), which exists since \( (I - 'A_2) \)
is strictly diagonally dominant are therefore invertible from

\[ \text{Lemma 1.} \quad Z(t) = Z^* \] for the Frobenius norm. Then, exponentially fast, \[ \lim_{t \to +\infty} \| Z(t) - Z^* \|_F = 0 \]

**Proof.** We have \( Z(t) - Z^* = \left[ 'A_1 Z_1 + 'A_2 Z(t-1) \right] - \left[ 'A_2 Z^* + (I - 'A_2) Z^* \right] = 'A_1 Z_1 + 'A_2 Z(t-1) - 'A_2 Z^* - (I - 'A_2)(I - 'A_2)^{-1} 'A_1 Z_1 = 'A_2(2^{-1} Z(t-1) - Z^*). \) So, \( Z(t) - Z^* = 'A_2(2^{-1} Z(t-1) - Z^*) \). Then, as a consequence of Cauchy-Schwarz inequality:

\[ \| Z(t) - Z^* \|_F = \| 'A_2(2^{-1} Z(t-1) - Z^*) \|_F \leq \| 'A_2 \|_F \| Z(t-1) - Z^* \|_F \]

Futhermore, \( 'A_2 = PDP^{-D} \), with \( A_2 = PDP^{-D} \) the eigendecomposition of symmetric matrix \( A_2 \). For diagonal matrix \( D \) we have \( \| D^{[i]} \|_F = \sqrt{\sum_{i=1}^{N} |\lambda_i|^{2}} \leq \sqrt{\| D \|_F (\max_i |\lambda_i|)^{1}} \)

with \( \lambda_i \) the \( i \)-th eigenvalue of \( 'A_2 \). Since \( 'A_2 \) has non-
negative entries, we derive from Perron–Frobenius theorem (see [Lovász, 2007]) that \( \max_i |\lambda_i| \) is bounded above by the maximum degree in \( A_2 \)'s graph. By definition, each node in \( V_2 \) has at least one connection to \( V_1 \); moreover rows of \( 'A_2 \) are nor-
malized by row sums of \( (A_1^T \) so the maximum degree in \( A_2 \)'s graph is strictly lower than 1. We conclude with a) and b) that \( 0 \leq |\lambda_i| < 1 \) for all \( i \in \{1, ..., |V_2|\} \), so \( 0 \leq \max_i |\lambda_i| < 1 \). This result implies that \( \| D \|_F \to 0 \) exponentially fast, and so does \( \| 'A_2 \|_F \leq \| P \|_F \| D \|_F \| P^{-D} \|_F \)
then \( \| Z(t) - Z^* \|_F \).

\[ \square \]

Our propagation process is summarized in Algorithm 2. If

some nodes are unreachable by such process because \( G \) is is
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.

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\(^1\): 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 $\hat{A}_{ij} = \sigma(z^T \hat{z})$. Validation and test sets gather 10% and 5% of edges (respectively 3% and 2%), 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 on 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 methods. 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 $\hat{G}$’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 on all possible subgraphs from 2-core to $\delta^*(\hat{G})$-core and on entire graphs, which is still tractable. Table 1 reports mean AUC and AP and their standard errors on 100

| Model | Size of input | Mean Perf. on Test Set (in %) | Mean Running Times (in sec.) |
|-------|--------------|------------------------------|-------------------------------|
|       | $k$-core     | AUC                          | AP                           |
|       |              | 10%-core dec. | Model train | Propagation | Total | Speed gain |
| Vgae  | -            | 83.02 ± 0.13 | 87.55 ± 0.18 | -           | 710.54 | 710.54 | - |
| on 2-core | 9.277 ± 25  | 83.97 ± 0.39 | 85.80 ± 0.49 | 1.35 | 159.15 | 0.31 | 61.81 | ×4.42 |
| on 3-core | 5.551 ± 19  | 83.92 ± 0.44 | 85.49 ± 0.71 | 1.35 | 60.12 | 0.34 | 61.81 | ×11.50 |
| on 4-core | 3.269 ± 30  | 82.40 ± 0.66 | 83.39 ± 0.75 | 1.35 | 22.14 | 0.36 | 23.85 | ×29.79 |
| on 5-core | 1.843 ± 25  | 78.31 ± 1.48 | 79.21 ± 1.64 | 1.35 | 7.71 | 0.36 | 9.42 | ×75.43 |
| ...    | ...          | ...                        | ...                         | ... | ... | ... | ... |
| on 8-core | 414 ± 89    | 67.21 ± 1.65 | 67.65 ± 2.00 | 1.35 | 1.55 | 0.38 | 3.28 | ×216.63 |
| on 9-core | 149 ± 93    | 61.92 ± 2.88 | 63.97 ± 2.86 | 1.35 | 1.14 | 0.38 | 2.87 | ×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

\(^1\)http://snap.stanford.edu/data/index.html
Table 2: Link Prediction on Google graph (\(n = 875K, m = 4.3M\)) using our framework on 17-core (\(|C_{17}\| = 23,787 \pm 208\)) on graph AE/VAE variants.

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

Table 3: Node Clustering on Patent graph (\(n = 2.7M, m = 13.9M\)) using our framework on 15-core (\(|C_{15}\| = 35,432\)) on graph AE/VAE variants.

| Model (using framework, \(k=15\)) | Perf. on Test Set (in %) | Total run. time |
|-------------------------------------|-------------------------|-----------------|
|                                     | Normalized MI          |                |
| 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           | 50min          |
| 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                           | 25.30 ± 1.22           | 1h38           |
| node2vec on \(G\) (best baseline)  | 24.10 ± 1.64           | 7h15           |

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 \(G\). 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 on the 2-core subgraph, 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 \(G\), i.e. without using our framework, is impossible on these graphs 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.** On both tasks, we note that adversarial training from ARGA/ARGV A 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 nonetheless better to clusters 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 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 to obtain theoretical guarantees on \(k\)-core approximations, and extend 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 efficiently 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 Matjaž 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] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. 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] D. Chakrabarti and C. 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] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. NIPS, 2016.

[Giatsidis et al., 2014] C. Giatsidis, F. Malliaros, D. Thilikos, and M. Vazirgiannis. Corecluster: A degeneracy based graph clustering framework. AAAI, 2014.

[Gori et al., 2005] M. Gori, G. Monfardini, and F. 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.

[Jin et al., 2018] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. ICML, 2018.

[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.

[Liu et al., 2018] Q. Liu, M. Allamanis, M. Brockschmidt, and A. L. Gaunt. Constrained graph variational autoencoders for molecule design. NeurIPS, 2018.

[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] F. Malliaros, C. Giatsidis, A. Papadopoulos, and M. 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] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Parallel Distributed Processing, Vol 1, 1986.

[Samanta et al., 2018] B. Samanta, A. De, G. Jana, P. K. Chattaraj, N. Ganguly, and M. Gomez-Rodriguez. Nevae: A deep generative model for molecular graphs. 2018.

[Scarselli et al., 2009] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. Neural Networks, 20(1):61–80, 2009.

[Simonovsky and Komodakis, 2018] M. Simonovsky and N. Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. 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] M. Tschannen, O. Bachem, and M. 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] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. arXiv preprint arXiv:1901.00596, 2019.

[Xu et al., 2019] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? ICLR, 2019.
**Supplementary Material**

This supplementary material provides additional details and more complete tables related to the experimental part of the *A Degeneracy Framework for Scalable Graph Autoencoders* paper. More precisely:

- Annex 1 describes our five datasets and their $k$-core decompositions.
- Annex 2 reports complete tables and experimental settings for the *link prediction* task.
- Annex 3 reports complete tables and experimental settings for the *node clustering* task.

**Annex 1 - Datasets**

**Medium-size graphs**

For comparison purposes, we ran experiments on three medium-size graphs used in [Kipf and Welling, 2016b], i.e. the Cora ($n = 2,708$ and $m = 5,429$), Citeseer ($n = 3,327$ and $m = 4,732$) and Pubmed ($n = 19,717$ and $m = 44,338$) citation networks. In these graphs, nodes are documents and edges are citation links. As [Kipf and Welling, 2016b], we ignored edges’ directions in experiments, i.e. we considered undirected versions of these graphs. Documents/nodes have sparse bag-of-words feature vectors, of sizes 3703, 1433 and 500 respectively. Each document also has a class label corresponding to its topic: in Cora (resp. in Citeseer, in Pubmed), nodes are clustered in 6 classes (resp. 7 classes, 3 classes) that we used as ground-truth communities for the *node clustering* task. Classes are roughly balanced. Data were collected from [Kipf and Welling, 2016b]’s GitHub repository for GAE².

**Large graphs**

We also provided experiments on two publicly available large graphs from Stanford’s SNAP website. The first one is the Google web graph³ ($n = 875,713$ and $m = 4,322,051$), whose nodes are web pages and directed edges represent hyperlinks between these pages. Data do not include ground-truth communities. The second one is the US Patent citation network⁴ ($n = 2,745,762$ and $m = 13,965,410$), originally released by the National Bureau of Economic Research (NBER) representing citations between patents. Nodes have classes corresponding to 6 patent categories; we removed nodes without classes from the Patent original graph. For both graphs, we once again ignored edges’ directions.

**$k$-core decompositions**

Tables 4 to 8 detail the entire $k$-core decomposition of each graph. We used the Python implementation provided in networkx library. For Citeseer, the 1-core is smaller than the 0-core because this graph includes isolated nodes. Figure 2 to 6 illustrate the evolution of the number of nodes in $k$-cores induced by increasing $k$. We note that, for Google (resp. for Patent), it was intractable to train autoencoders on 0 to 15-cores (resp. on 0 to 13-cores) due to memory errors. Therefore, in our experiments we trained models on 16 to 20-cores (resp. 14 to 18-cores).

| $k$ | Number of nodes in $k$-core | Number of edges in $k$-core |
|-----|----------------------------|-----------------------------|
| 0   | 2,708                      | 5,278                       |
| 1   | 2,708                      | 5,278                       |
| 2   | 2,136                      | 4,768                       |
| 3   | 1,257                      | 3,198                       |
| 4 ($\delta^*(\mathcal{G})$) | 174                       | 482                         |

| $k$ | Number of nodes in $k$-core | Number of edges in $k$-core |
|-----|----------------------------|-----------------------------|
| 0   | 3,327                      | 4,552                       |
| 1   | 3,279                      | 4,552                       |
| 2   | 1,601                      | 3,213                       |
| 3   | 564                        | 1,587                       |
| 4   | 203                        | 765                         |
| 5   | 70                         | 319                         |
| 6   | 28                         | 132                         |
| 7 ($\delta^*(\mathcal{G})$) | 18                       | 86                          |

³https://github.com/tkipf/gae
⁴http://snap.stanford.edu/data/web-Google.html
⁵http://snap.stanford.edu/data/cit-Patents.html
Table 6: $k$-core decomposition of Pubmed graph

| $k$ | Number of nodes in $k$-core | Number of edges in $k$-core |
|-----|----------------------------|----------------------------|
| 0   | 19,717                     | 44,324                     |
| 1   | 19,717                     | 44,324                     |
| 2   | 10,404                     | 35,011                     |
| 3   | 6,468                      | 27,439                     |
| 4   | 4,201                      | 21,040                     |
| 5   | 2,630                      | 15,309                     |
| 6   | 1,569                      | 10,486                     |
| 7   | 937                        | 7,021                      |
| 8   | 690                        | 5,429                      |
| 9   | 460                        | 3,686                      |
| 10 ($\delta^*(\mathcal{G})$) | 137                        | 1,104                      |

Table 7: $k$-core decomposition of Google graph

| $k$ | Number of nodes in $k$-core | Number of edges in $k$-core |
|-----|----------------------------|----------------------------|
| 0   | 875,173                    | 4,322,051                  |
| 1   | 875,173                    | 4,322,051                  |
| 2   | 711,870                    | 4,160,100                  |
| 3   | 581,712                    | 3,915,291                  |
| 4   | 492,655                    | 3,668,104                  |
| 5   | 424,155                    | 3,416,251                  |
| 6   | 367,361                    | 3,158,776                  |
| 7   | 319,194                    | 2,902,138                  |
| ... | ...                        | ...                        |
| 16  | 53,459                     | 676,076                    |
| 17  | 40,488                     | 519,077                    |
| 18  | 29,554                     | 384,478                    |
| 19  | 19,989                     | 263,900                    |
| 20  | 11,073                     | 154,000                    |
| ... | ...                        | ...                        |
| 43  | 103                        | 2,513                      |
| 44 ($\delta^*(\mathcal{G})$) | 48                        | 1,121                      |

Table 8: $k$-core decomposition of Patent graph

| $k$ | Number of nodes in $k$-core | Number of edges in $k$-core |
|-----|----------------------------|----------------------------|
| 0   | 2,745,762                  | 13,965,410                 |
| 1   | 2,745,762                  | 13,965,410                 |
| 2   | 2,539,676                  | 13,762,533                 |
| 3   | 2,299,008                  | 13,295,322                 |
| 4   | 2,011,518                  | 12,468,383                 |
| 5   | 1,671,474                  | 11,179,712                 |
| 6   | 1,284,078                  | 9,363,176                  |
| 7   | 888,944                    | 7,164,181                  |
| ... | ...                        | ...                        |
| 14  | 46,685                     | 717,597                    |
| 15  | 35,432                     | 576,755                    |
| 16  | 28,153                     | 480,436                    |
| 17  | 22,455                     | 400,463                    |
| ... | ...                        | ...                        |
| 63  | 106                        | 4,045                      |
| 64 ($\delta^*(\mathcal{G})$) | 106                        | 4,045                      |

Annex 2 - Link Prediction

In this Annex 2, we provide more complete tables for the link prediction task. Conclusions from experiments are discussed in main paper; here, we focus on completeness and details regarding experimental settings, to ensure reproducibility.

Medium-size graphs

For medium-size graphs, we apply our framework on all possible subgraphs from 2-core to $\delta^*(\mathcal{G})$-core and on entire graphs for comparison. Tables 9 to 11 report mean AUC and AP and their standard errors on 100 runs (train incomplete graphs and masked edges are different for each run) along with mean running times, for the VGAE model. Sizes of $k$-cores vary over runs due to the edge masking process. We obtained comparable performance/speed trade-offs for graph AE/VAE variants: for the sake of brevity, we therefore only report results on 2-core for these models.
Large graphs

For Google and Patent graphs, comparison with full models on $G$ is impossible due to overly large memory requirements. As a consequence, we apply our framework on the five largest $k$-cores (in terms of number of nodes) that were tractable using our machines. Tables 12 and 13 report mean AUC and AP and their standard errors on 10 runs (train incomplete graphs and masked edges are different for each run) along with mean running times, for the VGAE model. For other models, we only report results on second largest cores for the sake of brevity. We chose second largest cores (17-core for Google, 14-core for Patent), instead of largest cores (16-core for Google, 13-core for Patent) to lower running times.

Graph AE/VAE training

All graph AE/VAE models were trained on 200 epochs to return 16-dim embeddings, except for Patent (500 epochs, 32-dim). We included a 32-dim hidden layer in GCN encoders (two for the DeepGAE and DeepVGAE models), used Adam optimizer, trained models without dropout and with a learning rate of 0.01. We performed full-batch gradient descent and used the reparameterization trick [Kingma and Welling, 2013]. We resorted to Tensorflow public implementations of models (see corresponding references). Overall, our setting is quite similar to [Kipf and Welling, 2016b] and we indeed managed to reproduce their scores when training GAE and VGAE on entire graphs, with however larger standard errors. This difference comes from the fact that we used 100 different train/test splits, while they launched all runs on fixed dataset splits (randomness therefore only comes from initialization).

Impact of number of iterations $t$

We illustrate the impact of the number of iterations $t$ during propagation on performances in Figure 7. We display the evolution of mean AUC on three different graphs and cores w.r.t. the value of $t$. For $t > 5$ (resp. $t > 10$) in medium-size graphs (resp. large graphs), we achieve stable scores. We specify that the number of iterations has a negligible impact on running time. In our experiments, we therefore set $t = 10$ for all models leveraging our degeneracy framework.

Baselines

For Deepwalk [Perozzi et al., 2014], we trained models from 10 random walks of length 80 per node with a window size of 5, on a single epoch for each graph. We used similar hyper-parameters for node2vec [Grover and Leskovec, 2016], setting $p = q = 1$, and LINE [Tang et al., 2015] enforcing second-order proximity. We directly used public implementations provided by the authors. Due to unstable and underperforming results with 16-dim embeddings, we had to increase dimensions, up to 64, to compete with autoencoders. For the spectral embedding baseline, we also computed embeddings from 64 Laplacian eigenvectors.

In our experiments, we noticed some slight differences w.r.t. [Kipf and Welling, 2016b] regarding baselines, that we explain by our modifications in train/test splits and by different hyperparameters. However, these slight variations do not impact the conclusions of our experiments nor of theirs. We specify that the spectral embedding baseline is not scalable, due to the required eigendecomposition of the Laplacian matrix. Moreover, we chose not to report results for Deepwalk on large graphs due to too large training times ($> 20$h) on our machines. This does not question the scalability of this method (which is quite close to node2vec) but it suggests possible improvements in existing implementation.

Annex 3 - Node Clustering

Last, we provide more complete tables for the node clustering task. As before, we focus on completeness of results and implementation details, and refer to the main paper for interpretations. We evaluate the quality of node clustering from latent representations, running $k$-means in embeddings and reporting normalized Mutual Information (MI) scores. We used scikit-learn’s implementation with $k$-means++ initialization.

We do not report results for Google graph, due to the lack of ground-truth communities. Also, we obtained very low score on Citeseer graph with all methods, which suggests that node features are more useful than the graph structure to explain labels. As a consequence, we also omit this graph and focus on Cora, Pubmed and Patent in Tables 14 to 16. Tables are constructed in a similar fashion w.r.t. Annex 2. Graph AE/VAE models and baselines were trained with identical hyperparameters w.r.t. link prediction task. Contrary to Annex 2, we did not report spectral clustering results because graphs are not connected. Nevertheless, we compared to Louvain’s method [Blondel et al., 2008], a popular scalable algorithm to cluster nodes by maximizing modularity, using Python implementation provided in python-louvain’s library.
Table 9: Link Prediction on Cora \((n = 2, 708, m = 5, 429)\), using VGAE on all cores, graph AE/VAE variants on 2-core, and baselines

| Model | Size of input \(k\)-core | Mean Perf. on Test Set (in \%) | Mean Running Times (in sec.) |
|-------|--------------------------|-------------------------------|-------------------------------|
|       |                          | AUC                           | AP                           | \(k\)-core dec | Model train | Propagation | Total   |
| VGAE on \(G\) | - | 84.07 ± 1.22 | 87.83 ± 0.95 | - | 15.34 | - | 15.34 |
| on 2-core | 1,890 ± 16 | 85.24 ± 1.12 | 87.37 ± 1.13 | 0.16 | 8.00 | 0.10 | 8.26 |
| on 3-core | 862 ± 26 | 84.53 ± 1.33 | 85.04 ± 1.87 | 0.16 | 2.82 | 0.11 | 3.99 |
| on 4-core | 45 ± 13 | 72.33 ± 4.67 | 71.98 ± 4.97 | 0.16 | 0.98 | 0.12 | 1.26 |
| GAE on 2-core | 1,890 ± 16 | 85.17 ± 1.02 | 87.26 ± 1.12 | 0.16 | 8.05 | 0.10 | 8.31 |
| DeepGAE on 2-core | 1,890 ± 16 | 86.25 ± 0.81 | 87.92 ± 0.78 | 0.16 | 8.24 | 0.10 | 8.50 |
| DeepVGAE on 2-core | 1,890 ± 16 | 86.16 ± 0.95 | 87.71 ± 0.98 | 0.16 | 8.20 | 0.10 | 8.46 |
| Graphite on 2-core | 1,890 ± 16 | 86.85 ± 0.82 | 88.13 ± 0.84 | 0.16 | 9.41 | 0.10 | 9.67 |
| Var-Graphite on 2-core | 1,890 ± 16 | 86.39 ± 0.84 | 88.40 ± 0.80 | 0.16 | 9.35 | 0.10 | 9.61 |
| ARGA on 2-core | 1,890 ± 16 | 85.82 ± 0.88 | 88.22 ± 0.70 | 0.16 | 7.99 | 0.10 | 8.25 |
| ARVGA on 2-core | 1,890 ± 16 | 85.74 ± 0.74 | 88.51 ± 0.74 | 0.16 | 7.98 | 0.10 | 8.24 |
| ChebVGAE on 2-core | 1,890 ± 16 | 86.15 ± 0.54 | 88.10 ± 0.39 | 0.16 | 15.78 | 0.10 | 16.04 |
| ChebVGAE on 2-core | 1,890 ± 16 | 86.30 ± 0.49 | 88.29 ± 0.50 | 0.16 | 15.65 | 0.10 | 15.91 |
| GAE with node features on 2-core | 1,890 ± 16 | 88.10 ± 0.87 | 89.36 ± 0.88 | 0.16 | 8.66 | 0.10 | 8.92 |
| VGAE with node features on 2-core | 1,890 ± 16 | 87.97 ± 0.99 | 89.53 ± 0.96 | 0.16 | 8.60 | 0.10 | 8.86 |
| DeepWalk | - | 83.02 ± 1.21 | 84.41 ± 1.23 | - | 38.50 | - | 38.50 |
| LINE | - | 83.49 ± 1.31 | 84.42 ± 1.39 | - | 13.55 | - | 13.55 |
| node2vec | - | 83.52 ± 1.47 | 84.60 ± 1.23 | - | 8.42 | - | 8.42 |
| Spectral | - | 86.53 ± 1.02 | 87.41 ± 1.12 | - | 2.78 | - | 2.78 |

Table 10: Link Prediction on Citeseer \((n = 3, 327, m = 4, 732)\), using VGAE on all cores*, graph AE/VAE variants on 2-core, and baselines.

*6-core and 7-core are not reported due to their frequent vanishing after edge masking
| Model                | Size of input $k$-core | Mean Perf. on Test Set (in %) | Mean Running Times (in sec.) |
|----------------------|------------------------|-------------------------------|-----------------------------|
|                      |                        | $\text{AU/C}$ | $\text{AP}$ | $k$-core dec. | Model train | Propagation | Total      |
| GAE on 2-core        | 9.277 ± 25             | 83.90 ± 0.39 | 85.80 ± 0.49 | 1.35         | 159.15      | 0.31        | 160.81     |
| DeepGAE on 2-core    | 9.277 ± 25             | 83.90 ± 0.39 | 85.80 ± 0.49 | 1.35         | 159.15      | 0.31        | 160.81     |
| Graphite on 2-core   | 9.277 ± 25             | 83.90 ± 0.39 | 85.80 ± 0.49 | 1.35         | 159.15      | 0.31        | 160.81     |
| Var-Graphite on 2-core| 9.277 ± 25             | 83.90 ± 0.39 | 85.80 ± 0.49 | 1.35         | 159.15      | 0.31        | 160.81     |
| ARGA on 2-core       | 9.277 ± 25             | 83.90 ± 0.39 | 85.80 ± 0.49 | 1.35         | 159.15      | 0.31        | 160.81     |
| ChebGAE on 2-core    | 9.277 ± 25             | 83.90 ± 0.39 | 85.80 ± 0.49 | 1.35         | 159.15      | 0.31        | 160.81     |

Table 11: Link Prediction on Pubmed ($n = 19,717, m = 44,338$), using VGAE on all cores*, graph AE/VAE variants on 2-core, and baselines. * 10-core is not reported due to its frequent vanishing after edge masking.

| Model                | Size of input $k$-core | Mean Perf. on Test Set (in %) | Mean Running Times (in sec.) |
|----------------------|------------------------|-------------------------------|-----------------------------|
|                      |                        | $\text{AU/C}$ | $\text{AP}$ | $k$-core dec. | Model train | Propagation | Total      |
| GAE with node features on 2-core | 9.277 ± 25 | 84.94 ± 0.54 | 85.83 ± 0.58 | 1.35         | 168.62      | 0.31        | 170.28     |
| VGAE with node features on 2-core | 9.277 ± 25 | 85.81 ± 0.68 | 88.01 ± 0.53 | 1.35         | 165.10      | 0.31        | 167.65     |
| 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             | -                      | 81.26 ± 0.26 | 85.55 ± 0.26 | -            | 48.91        | -           | 48.91      |
| Spectral             | -                      | 83.14 ± 0.42 | 86.55 ± 0.41 | -            | 31.71        | -           | 31.71      |

Table 12: Link Prediction on Google ($n = 875,713, m = 4,322,051$), using VGAE on 16 to 20 cores, graph AE/VAE variants on 17-core, and baselines.
| Model                  | Size of input | Mean Perf. on Test Set (in %) | Mean Running Times (in sec.) |
|------------------------|---------------|-------------------------------|-------------------------------|
|                        | k-core        | AUC                           | k-core dec. | Model train | Propagation | Total           |
| VGAE on 14-core        | 38,408 ± 147  | **88.48 ± 0.35**              | **88.81 ± 0.32**              | 507.08      | 3,024.31    | 122.29          | 3,653.68 (16min) |
| on 15-core             | 29,191 ± 243  | 88.16 ± 0.50                  | 88.37 ± 0.57                  | 507.08      | 1,656.46    | 123.47          | 2,878.01 (3min)  |
| on 16-core             | 23,132 ± 48   | 87.85 ± 0.47                  | 88.02 ± 0.48                  | 507.08      | 949.09      | 124.26          | 1,579.43 (20min) |
| on 17-core             | 18,066 ± 143  | 87.34 ± 0.56                  | 87.64 ± 0.47                  | 507.08      | 574.25      | 126.55          | 1,207.88 (20min) |
| on 18-core             | 13,972 ± 86   | 87.27 ± 0.55                  | 87.78 ± 0.51                  | 507.08      | **351.73**  | 127.01          | **985.82** (16min) |
| GAE on 15-core         | 29,191 ± 243  | 87.59 ± 0.29                  | 87.30 ± 0.28                  | 507.08      | 1,880.11    | 123.47          | 2,510.66 (42min) |
| DeepGAE on 15-core     | 29,191 ± 243  | 87.71 ± 0.31                  | 87.64 ± 0.19                  | 507.08      | 2,032.15    | 123.47          | 2,662.70 (44min) |
| DeepVGAE on 15-core    | 29,191 ± 243  | 87.03 ± 0.54                  | 87.20 ± 0.44                  | 507.08      | 1,927.33    | 123.47          | 2,557.88 (43min) |
| Graphite on 15-core    | 29,191 ± 243  | 85.19 ± 0.38                  | 86.01 ± 0.31                  | 507.08      | 1,989.72    | 123.47          | 2,620.27 (44min) |
| Var-Graphite on 15-core| 29,191 ± 243  | 85.37 ± 0.30                  | 86.07 ± 0.24                  | 507.08      | 1,916.79    | 123.47          | 2,547.34 (42min) |
| ARGA on 15-core        | 29,191 ± 243  | **89.22 ± 0.10**              | **89.40 ± 0.11**              | 507.08      | 2,028.46    | 123.47          | 2,659.01 (44min) |
| ARVGA on 15-core       | 29,191 ± 243  | 87.18 ± 0.17                  | 87.39 ± 0.33                  | 507.08      | 1,915.53    | 123.47          | 2,546.08 (42min) |
| ChebGAE on 15-core     | 29,191 ± 243  | 88.53 ± 0.20                  | 88.91 ± 0.20                  | 507.08      | 3,391.01    | 123.47          | 4,021.56 (1h07)  |
| ChebVGAE on 15-core    | 29,191 ± 243  | 87.75 ± 0.19                  | 89.07 ± 0.24                  | 507.08      | 3,230.52    | 123.47          | 3,861.07 (1h04)  |
| LINE                   | -             | 90.07 ± 0.41                  | 94.52 ± 0.49                  | -           | 33,063.80   | -               | 33,063.80 (9h11) |
| node2vec               | -             | **95.04 ± 0.25**              | **96.01 ± 0.19**              | -           | 26,126.01   | -               | 26,126.01 (7h15) |

Table 13: Link Prediction on Patent ($n = 2,745,762$, $m = 13,965,410$), using VGAE on 14 to 18 cores, graph AE/VAE variants on 15-core, and baselines

| Model                  | Size of input | Mean Perf. on Test Set (in %) | Mean Running Times (in sec.) |
|------------------------|---------------|-------------------------------|-------------------------------|
|                        | k-core        | MI                            | k-core dec. | Model train | Propagation | Total           |
| VGAE on $G$            |               | 29.52 ± 2.61                  | -           | 15.34       | -           | 15.34          |
| on 2-core              | 2,136         | 34.08 ± 2.55                  | 0.16        | 9.94        | 0.10        | 10.20          |
| on 3-core              | 1,257         | **36.29 ± 2.52**              | 0.16        | 4.43        | 0.11        | 4.70           |
| on 4-core              | 174           | 35.93 ± 1.88                  | 0.16        | **1.16**    | 0.12        | **1.44**       |
| VGAE with node features on $G$ |               | **47.25 ± 1.80**              | -           | 15.89       | -           | 15.89          |
| on 2-core              | 2,136         | 45.08 ± 1.91                  | 0.16        | 10.42       | 0.10        | 10.68          |
| on 3-core              | 1,257         | 40.96 ± 2.06                  | 0.16        | 4.75        | 0.11        | 5.02           |
| on 4-core              | 174           | 38.11 ± 1.23                  | 0.16        | **1.22**    | 0.12        | **1.42**       |
| GAE on 2-core          | 2,136         | 34.91 ± 2.51                  | 0.16        | 10.02       | 0.10        | 10.28          |
| DeepGAE on 2-core      | 2,136         | 35.30 ± 2.52                  | 0.16        | 10.12       | 0.10        | 10.38          |
| DeepVGAE on 2-core     | 2,136         | 34.49 ± 2.85                  | 0.16        | 10.09       | 0.10        | 10.35          |
| Graphite on 2-core     | 2,136         | 33.91 ± 2.17                  | 0.16        | 10.97       | 0.10        | 11.23          |
| Var-Graphite on 2-core | 2,136         | 33.89 ± 2.13                  | 0.16        | 10.91       | 0.10        | 11.17          |
| ARGA on 2-core         | 2,136         | 34.73 ± 2.84                  | 0.16        | 9.99        | 0.10        | 10.25          |
| ARVGA on 2-core        | 2,136         | 33.36 ± 2.53                  | 0.16        | 9.97        | 0.10        | 10.23          |
| ChebGAE on 2-core      | 2,136         | **36.52 ± 2.05**              | 0.16        | 19.22       | 0.10        | 19.48          |
| ChebVGAE on 2-core     | 2,136         | 37.83 ± 2.11                  | 0.16        | 20.13       | 0.10        | 20.28          |
| DeepWalk               |               | 40.37 ± 1.51                  | -           | 38.50       | -           | 38.50          |
| LINE                   | -             | 39.75 ± 1.24                  | -           | 11.55       | -           | 11.55          |
| Louvain                | -             | **46.76 ± 0.82**              | -           | **1.83**    | -           | **1.83**       |
| node2vec               | -             | 43.45 ± 1.32                  | -           | 8.42        | -           | 8.42           |

Table 14: Node Clustering on Cora ($n = 2,708$, $m = 4,292$), using VGAE on all cores, graph AE/VAE variants on 2-core, and baselines
| Model                      | Size of input $k$-core | Mean Perf. on Test Set (in %) $\mu$ | Mean Running Times (in sec.) $k$-core dec. | Model train | Propagation | Total |
|----------------------------|------------------------|-------------------------------------|-------------------------------------------|-------------|-------------|-------|
| DeepGraphite on 2-core     | 10, 404                | 25.19 ± 1.59                       | 1.35                                      | 102.37      | 0.38        | 204.75 |
| DeepVGAE on 2-core         | 10, 404                | 23.6 ± 1.83                        | 1.35                                      | 202.37      | 0.38        | 204.75 |
| Graphite on 2-core         | 10, 404                | 20.17 ± 1.73                       | 1.35                                      | 38.59       | 0.38        | 38.59  |
| Louvain                   | 10, 404                | 18.15 ± 2.04                       | 1.35                                      | 17.79       | 0.38        | 17.79  |
| ChebGraphite on 2-core     | 10, 404                | 11.67 ± 0.71                       | 1.35                                      | 48.91       | 0.38        | 48.91  |
| ChebVGAE on 2-core         | 10, 404                | 29.57 ± 0.22                       | -                                         |             |             |       |
| LINE                      | -                      | 27.23 ± 0.32                       | -                                         | 342.25      | -           | 342.25 |
| node2vec                  | -                      | 26.26 ± 0.28                       | -                                         | 63.52       | -           | 63.52  |

Table 15: Node Clustering on Pubmed ($n = 19,717$, $m = 44,338$), using VGAE on all cores, Graph AE/VAE variants on 2-core, and baselines.

| Model                      | Size of input $k$-core | Mean Perf. on Test Set (in %) $\mu$ | Mean Running Times (in sec.) $k$-core dec. | Model train | Propagation | Total |
|----------------------------|------------------------|-------------------------------------|-------------------------------------------|-------------|-------------|-------|
| VGAE on 14-core            | 46, 685                | 25.22 ± 1.51                       | 507.08                                    | 6,390.37    | 120.80      | 7,018.25 (1h57) |
| on 15-core                 | 35, 432                | 24.53 ± 1.62                       | 507.08                                    | 2,589.95    | 123.95      | 3,200.98 (54min) |
| on 16-core                 | 28, 153                | 24.16 ± 1.96                       | 507.08                                    | 1,567.78    | 123.14      | 2,100.00 (37min) |
| on 17-core                 | 22, 455                | 24.14 ± 2.01                       | 507.08                                    | 898.27      | 124.02      | 1,520.32 (25min) |
| on 18-core                 | 17, 799                | 22.54 ± 1.98                       | 507.08                                    | 551.83      | 126.67      | 1,185.58 (20min) |
| GAE on 15-core             | 35, 432                | 23.76 ± 2.25                       | 507.08                                    | 2,750.09    | 123.95      | 3,381.13 (56min) |
| DeepGAE on 15-core         | 35, 432                | 24.27 ± 1.10                       | 507.08                                    | 3,007.31    | 123.95      | 3,638.34 (1h01) |
| DeepVGAE on 15-core        | 35, 432                | 24.54 ± 1.23                       | 507.08                                    | 2,844.16    | 123.95      | 3,675.10 (58min) |
| Graphite on 15-core        | 35, 432                | 24.22 ± 1.45                       | 507.08                                    | 2,899.87    | 123.95      | 3,530.90 (50min) |
| Var-Graphite on 15-core    | 35, 432                | 24.25 ± 1.51                       | 507.08                                    | 2,869.92    | 123.95      | 3,500.95 (58min) |
| ARGA on 15-core            | 35, 432                | 24.26 ± 1.18                       | 507.08                                    | 3,013.28    | 123.95      | 3,644.31 (1h01) |
| ARVGA on 15-core           | 35, 432                | 24.76 ± 1.32                       | 507.08                                    | 2,862.54    | 123.95      | 3,493.57 (58min) |
| ChebGAE on 15-core         | 35, 432                | 25.23 ± 1.21                       | 507.08                                    | 5,412.12    | 123.95      | 6,043.15 (1h41) |
| ChebVGAE on 15-core        | 35, 432                | 25.30 ± 1.22                       | 507.08                                    | 5,289.91    | 123.95      | 5,920.94 (1h38) |
| LINE                      | -                      | 23.19 ± 1.82                       | -                                         | 33,063.80   | -           | 33,063.80 (9h11) |
| Louvain                   | -                      | 11.90 ± 1.79                       | -                                         | 13,634.16   | -           | 13,634.16 (3h47) |
| node2vec                  | -                      | 24.10 ± 1.64                       | -                                         | 26,126.01   | -           | 26,126.01 (7h15) |

Table 16: Node Clustering on Patent ($n = 2,745,762$, $m = 13,965,410$), using VGAE on 14 to 18 cores, graph AE/VAE variants on 15-core, and baselines.