Chapter 2

Neighborhood reconstruction methods

This part of the book is concerned with methods for learning *node embeddings*. The goal of these methods is to encode nodes as low-dimensional vectors that summarize their graph position and the structure of their local graph neighborhood. In other words, we want to project nodes into a latent space, where geometric relations in this latent space correspond to relationships (*e.g.*, edges) in the original graph [Hoff et al., 2002]. Figure 2.1 visualizes an example embedding of the famous Zachary Karate Club social network [Perozzi et al., 2014], where two dimensional node embeddings capture the community structure implicit in the social network.

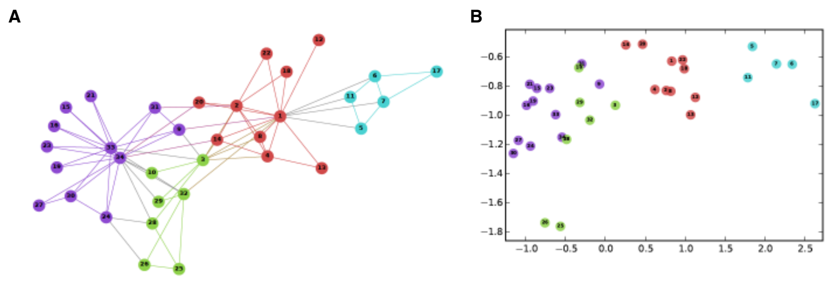

Figure 2.1: **A**, Visualization of the Zachary Karate Club network, where nodes are colored according to the different underlying communities. **B**, Two-dimensional visualization of node embeddings generated from this graph using the DeepWalk method (Section 2.1) [Perozzi et al., 2014]. The distances between nodes in the embedding space reflect similarity in the original graph. Image from from [Perozzi et al., 2014, Perozzi, 2016].

In this chapter we will provide an overview of node embedding methods for simple and weighted graphs. Chapter 3 will provide an overview of analogous embedding approaches for multi-relational graphs.

An encoder-decoder perspective Recent years have seen a surge of research on node embeddings, leading to a complicated diversity of notations, motivations, and conceptual models. Thus, to organize our discussion we introduce the notion of an *encoder-decoder* framework over graphs. In this framework, we organize the various methods around two key mapping functions: an *encoder*, which maps each node to a low-dimensional vector, or embedding, and a *decoder*, which reconstructs information about a node's neighborhood from the learned embeddings (Figure 2.2).

Formally, the *encoder* is a function,

$$\textsc{enc} : \mathcal{V} \to \mathbb{R}^d, \tag{2.1}$$

that maps nodes to vector embeddings $\mathbf{z}_v \in \mathbb{R}^d$ (where $\mathbf{z}_v$ corresponds to the embedding for node $v \in \mathcal{V}$). The *decoder* is a function that accepts a set of node embeddings and decodes user-specified graph statistics from these embeddings. For example, the decoder might measure the overlap in neighborhood between two nodes. In principle, many decoders are possible; however, the vast majority of works use a basic *pairwise decoder*,

$$\textsc{dec} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+, \tag{2.2}$$

that maps pairs of node embeddings to a real-valued node similarity measure, which quantifies the similarity of the two nodes in the original graph.

When we apply the pairwise decoder to a pair of embeddings ($\mathbf{z}_u$,$\mathbf{z}_v$) we get a *reconstruction* of the similarity between u and v in the original graph, and the goal is optimize the encoder and decoder mappings to minimize the error, or loss, in this reconstruction so that:

$$\textsc{dec}(\textsc{enc}(u), \textsc{enc}(v)) = \textsc{dec}(\mathbf{z}_u, \mathbf{z}_v) \approx \mathbf{S}[u, v], \tag{2.3}$$

where $\mathbf{S}[u, v]$ is a user-defined, graph-based similarity measure between nodes, defined over the graph $\mathcal{G}$. In other words, we want to optimize our encoder-decoder model so that we can reconstruct neighborhood information from the original graph. For example, one might set $\mathbf{S}[u, v] \triangleq \mathbf{A}[u, v]$ and define nodes to have a similarity of 1 if they are adjacent and 0 otherwise Ahmed et al. [2013], or one might define $\mathbf{S}[u, v]$ according to one of the more complex neighborhood overlap statistics discussed in Chapter 1. In practice, most approaches realize the reconstruction objective (Equation 2.3) by minimizing an empirical loss $\mathcal{L}$ over a set of training node pairs $\mathcal{D}$:

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} \ell\left(\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v), \mathbf{S}[u, v]\right), \tag{2.4}$$

where $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a user-specified loss function, which measures the discrepancy between the decoded (*i.e.*, estimated) similarity values $\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v)$

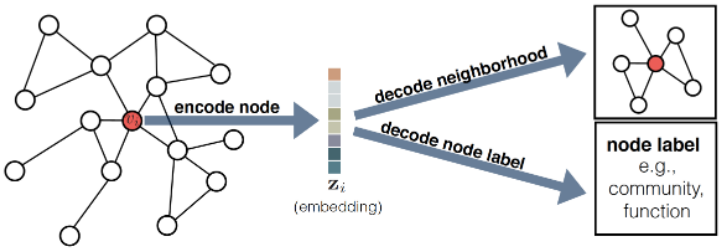

Figure 2.2: Overview of the encoder-decoder approach. First the encoder maps the node, u, to a low-dimensional vector embedding, $\mathbf{z}_u$, based on the node's position in the graph, its local neighborhood structure, and/or its attributes. Next, the decoder extracts user-specified information from the low-dimensional embedding; this might be information about u's local graph neighborhood (*e.g.*, the identity of its neighbors) or a classification label associated with u (*e.g.*, a community label). By jointly optimizing the encoder and decoder, the system learns to compress information about graph structure into the low-dimensional embedding space.

and the true values $\mathbf{S}[u, v]$. Once we have optimized the encoder-decoder system, we can use the trained encoder to generate embeddings for nodes, which can then be used as a feature inputs for downstream machine learning tasks, such as node classification, link prediction, and clustering.

Adopting this encoder-decoder view, we organize our discussion along the following four methodological components:

1. A **pairwise similarity function (or matrix)** $\mathbf{S}[u, v] : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}^+$, defined over the graph $\mathcal{G}$, which measures the similarity between nodes.

2. An **encoder function**, ENC, that generates the node embeddings. This function contains a number of trainable parameters that are optimized during the training phase.

3. A **decoder function**, DEC, which reconstructs pairwise similarity values from the generated embeddings. This function usually contains no trainable parameters.

4. A **loss function**, ℓ, which determines how the quality of the pairwise reconstructions is evaluated in order to train the model, *i.e.*, how $\text{DEC}(\mathbf{z}_u, \mathbf{z}_u)$ is compared to the true $\mathbf{S}[u, v]$ values.

As we will show, the primary methodological distinctions between the various node embedding approaches are in how they define these four components.

Table 2.1: A summary of some well-known shallow embedding embedding algorithms. Note that the decoders and similarity functions for the random-walk based methods are asymmetric, with the similarity function $p_{\mathcal{G}}(v|u)$ corresponding to the probability of visiting v on a fixed-length random walk starting from u.

Method	Decoder	Similarity measure	Loss function (ℓ)
Lap. Eigenmaps	$\|\mathbf{z}_u - \mathbf{z}_v\|_2^2$	general	$\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]$
Graph Fact.	$\mathbf{z}_u^\top \mathbf{z}_v$	$\mathbf{A}[u, v]$	$\|\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u, v]\|_2^2$
GraRep	$\mathbf{z}_u^\top \mathbf{z}_v$	$\mathbf{A}[u, v], ..., \mathbf{A}^k[u, v]$	$\|\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u, v]\|_2^2$
HOPE	$\mathbf{z}_u^\top \mathbf{z}_v$	general	$\|\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u, v]\|_2^2$
DeepWalk	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v\|u)$	$-\mathbf{S}[u, v] \log(\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v))$
node2vec	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v\|u)$ (biased)	$-\mathbf{S}[u, v] \log(\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v))$

2.1 Shallow embedding approaches

The majority of node embedding algorithms rely on what we call *shallow embedding*. For these shallow embedding approaches, the encoder function—which maps nodes to vector embeddings—is simply an "embedding lookup":

$$\textsc{enc}(v) = \mathbf{Z}[v] \tag{2.5}$$

where $\mathbf{Z} \in \mathbb{R}^{|\mathcal{V}| \times d}$ is a matrix containing the embedding vectors for all nodes and $\mathbf{Z}[v]$ denotes the row of $\mathbf{Z}$ corresponding to node v. The set of trainable parameters for shallow embedding methods is simply $\Theta_{\textsc{enc}} = \{\mathbf{Z}\}$, *i.e.*, the embedding matrix $\mathbf{Z}$ is optimized directly.

Table 2.1 summarizes some well-known shallow embedding methods within the encoder-decoder framework. Table 2.1 highlights how these methods can be succinctly described according to (i) their decoder function, (ii) their graph-based similarity measure, and (iii) their loss function. The following two sections describe these methods in more detail, distinguishing between matrix factorization-based approaches (Section 2.1) and more recent approaches based on random walks (Section 2.1).

Factorization-based approaches

Early methods for learning representations for nodes largely focused on matrix-factorization approaches, which are directly inspired by classic techniques for dimensionality reduction [Belkin and Niyogi, 2002, Kruskal, 1964].

Laplacian eigenmaps One of the earliest, and most well-known instances, is the Laplacian eigenmaps (LE) technique [Belkin and Niyogi, 2002], which we can view within the encoder-decoder framework as a shallow embedding approach in which the decoder is defined as

$$\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v) = \|\mathbf{z}_u - \mathbf{z}_v\|_2^2$$

and where the loss function weights pairs of nodes according to their similarity in the graph:

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u,v]. \tag{2.6}$$

The optimal solution to Equation 2.6 is identical to the solution for spectral clustering (Section 1.3.3). If we assume the embeddings $\mathbf{z}_u$ are d-dimensional, then the optimal solution to Equation 2.6 is given by the d smallest eigenvectors of the Laplacian (excluding the eigenvector of all ones).

Inner-product methods Following on the Laplacian eigenmaps technique, there are a large number of recent embedding methodologies based on a pairwise, inner-product decoder:

$$\text{DEC}(\mathbf{z}_u, \mathbf{z}_v) = \mathbf{z}_u^\top \mathbf{z}_v, \tag{2.7}$$

where the strength of the relationship between two nodes is proportional to the dot product of their embeddings. In these approaches the dot product can be viewed as a measure of neighborhood overlap between nodes.

The Graph Factorization (GF) algorithm[1] [Ahmed et al., 2013], GraRep [Cao et al., 2015], and HOPE [Ou et al., 2016] all fall firmly within this class. In particular, all three of these methods use an inner-product decoder, a mean-squared-error (MSE) loss,

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} \|\text{DEC}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u,v]\|_2^2, \tag{2.8}$$

and they differ primarily in the node similarity measure used, *i.e.* how they define $\mathbf{S}[u,v]$. The Graph Factorization algorithm defines node similarity directly based on the adjacency matrix (*i.e.*, $\mathbf{S}[u,v] \triangleq \mathbf{A}[u,v]$); GraRep considers various powers of the adjacency matrix (*e.g.*, $\mathbf{S}[u,v] \triangleq \mathbf{A}^2[u,v]$) in order to capture higher-order node similarity; and the HOPE algorithm supports general neighborhood overlap measures (*e.g.*, any neighborhood overlap measure from Chapter 1).

We refer to these methods as matrix-factorization approaches because, averaging over all nodes, they optimize loss functions (roughly) of the form:

$$\mathcal{L} \approx \|\mathbf{Z}\mathbf{Z}^\top - \mathbf{S}\|_2^2, \tag{2.9}$$

where $\mathbf{S}$ is the matrix containing the pairwise node-node similarity measures and $\mathbf{Z}$ is the matrix of node embeddings. Intuitively, the goal of these methods is simply to learn embeddings for each node such that the inner product between the learned embedding vectors approximates some deterministic measure of node similarity.

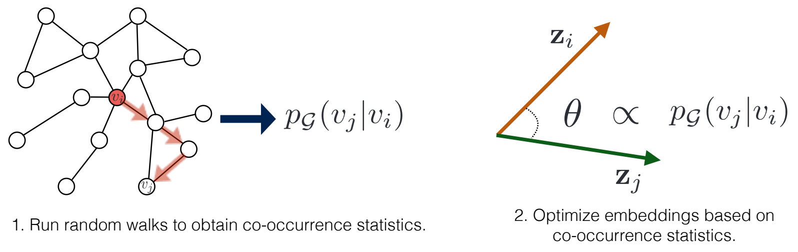
$p_{\mathcal{G}}(v_j|v_i)$
1. Run random walks to obtain co-occurrence statistics.
$\mathbf{z}_i$
$\theta \propto p_{\mathcal{G}}(v_j|v_i)$
$\mathbf{z}_j$
2. Optimize embeddings based on co-occurrence statistics.

Figure 2.3: The random-walk based methods sample a large number of fixed-length random walks starting from each node, u. The embedding vectors are then optimized so that the dot-product, or angle, between two embeddings, $\mathbf{z}_u$ and $\mathbf{z}_v$, is (roughly) proportional to the probability of visiting v on a fixed-length random walk starting from u.

Random walk approaches

Many recent successful methods learn node embeddings based on random walk statistics. Their key innovation is optimizing the node embeddings so that nodes have similar embeddings if they tend to co-occur on short random walks over the graph (Figure 2.3). Thus, instead of using a deterministic measure of node similarity, like the methods of Section 2.1, these random walk methods employ a flexible, stochastic measure of node similarity, which has led to superior performance in a number of settings [Goyal and Ferrara, 2017].

DeepWalk and node2vec Like the matrix factorization approaches described above, DeepWalk and node2vec rely on shallow embedding and use a decoder based on the inner product. However, instead of trying to decode a deterministic node similarity measure, these approaches optimize embeddings to encode the statistics of random walks. The basic idea behind these approaches is to learn embeddings so that (roughly):

$$\begin{aligned}\text{DEC}(\mathbf{z}_u, \mathbf{z}_v) &\triangleq \frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{v_k \in \mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}} \\ &\approx p_{\mathcal{G},T}(v|u),\end{aligned} \tag{2.10}$$

where $p_{\mathcal{G},T}(v|u)$ is the probability of visiting v on a length-T random walk starting at u, with T usually defined to be in the range $T \in \{2, ..., 10\}$. Note that unlike the similarity measures in Section 2.1, $p_{\mathcal{G},T}(v|u)$ is both stochastic and asymmetric.

More formally, these approaches attempt to minimize the following cross-

[1]Of course, Ahmed et al. [Ahmed et al., 2013] were not the first researchers to propose factorizing an adjacency matrix, but they were the first to present a scalable $O(|\mathcal{E}|)$ algorithm for the purpose of generating node embeddings.

entropy loss:

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} -\log(\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v)), \tag{2.11}$$

where the training set $\mathcal{D}$ is generated by sampling random walks starting from each node (*i.e.*, where N pairs for each node u are sampled from the distribution $(u, v) \sim p_{\mathcal{G},T}(v|v)$). However, naively evaluating the loss in Equation (2.11) is prohibitively expensive—in particular, $O(|\mathcal{D}||\mathcal{V}|)$—since evaluating the denominator of Equation (2.10) has time complexity $O(|\mathcal{V}|)$. Thus, DeepWalk and node2vec use different optimizations and approximations to compute the loss in Equation (2.11). DeepWalk employs a "hierarchical softmax" technique to compute the normalizing factor, using a binary-tree structure to accelerate the computation [Perozzi et al., 2014]. In contrast, node2vec approximates Equation (2.11) using "negative sampling": instead of normalizing over the full vertex set, node2vec approximates the normalizing factor using a set of random "negative samples":

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} -\log(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)) - \gamma\mathbb{E}_{v_n\sim P_n(\mathcal{V})}[\log(-\sigma(\mathbf{z}_u^\top \mathbf{z}_{v_n}))], \tag{2.12}$$

where σ denotes the logistic function, $P_n(\mathcal{V})$ denotes a distribution over the set of nodes $\mathcal{V}$ and $\gamma > 0$ is a hyperparameter. In practice $P_n(\mathcal{V})$ is often defined to be a uniform distribution. In other cases it is defined so that a node's sampling probability is a sub-linear function of its degree. The expectation in Equation 2.12 is evaluated via Monte-Carlo sampling by drawing K nodes from the distribution $P_n(\mathcal{V})$.[2]

Another key distinction between node2vec and DeepWalk is that node2vec allows for a flexible definition of random walks, whereas DeepWalk uses simple unbiased random walks over the graph. By introducing these hyperparameters, node2vec is able to smoothly interpolate between walks that are more akin to breadth-first or depth-first search, which led to increased performance on some benchmarks Grover and Leskovec [2016].

Large-scale information network embeddings (LINE) Another highly successful node embedding approach, which is not based random walks but is contemporaneous and often compared with DeepWalk and node2vec, is the LINE method [Tang et al., 2015]. LINE combines two encoder-decoder objectives that optimize "first-order" and "second-order" node similarity, respectively. The first-order objective uses a decoder based on the sigmoid function,

$$\textsc{dec}(\mathbf{z}_u, \mathbf{z}_v) = \frac{1}{1 + e^{-\mathbf{z}_u^\top \mathbf{z}_v}}, \tag{2.13}$$

and an adjacency-based similarity measure (*i.e.*, $\mathbf{S}[u, v] = \mathbf{A}[u, v]$). The second-order encoder-decoder objective is similar but considers two-hop adjacency neighborhoods and uses an encoder identical to Equation (2.10). Both the first-order

[2]It is also standard practice to set $\gamma = K$.

and second-order objectives are optimized using loss functions derived from the KL-divergence metric. Thus, LINE is conceptually related to node2vec and DeepWalk in that it uses a probabilistic decoder and loss, but it explicitly factorizes first- and second-order similarities, instead of combining them in fixed-length random walks.

Additional variants of the random-walk idea There have also been a number of further extensions of the random walk idea. For example, Perozzi et al. [2016] extend the DeepWalk algorithm to learn embeddings using random walks that "skip" or "hop" over multiple nodes at each step, resulting in a similarity measure similar to GraRep [Cao et al., 2015], while Chamberlain et al. [2017] modify the inner-product decoder of node2vec to use a hyperbolic, rather than Euclidean, distance measure.

Connections between random walk methods and matrix factorization Recent work has found that random walk methods are actually closely related to matrix factorization approaches Qiu et al. [2018]. Suppose we define the following matrix of node-node similarity values:

$$\mathbf{S}_{\mathrm{DW}} = \log\left(\frac{\mathrm{vol}(\mathcal{V})}{T}\left(\sum_{t=1}^{T}\mathbf{P}^{t}\right)\mathbf{D}^{-1}\right) - \log(b), \tag{2.14}$$

where b is a constant and $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$. In this case Qiu et al. [2018] show that the embeddings $\mathbf{Z}$ learned by DeepWalk satisfy:

$$\mathbf{Z}^{\top}\mathbf{Z} \approx \mathbf{S}_{\mathrm{DW}}. \tag{2.15}$$

Interestingly, we can also decompose the interior part Equation 2.14 as

$$\left(\sum_{t=1}^{T}\mathbf{P}^{t}\right)\mathbf{D}^{-1} = \mathbf{D}^{-\frac{1}{2}}\left(\mathbf{U}\left(\sum_{t=1}^{T}\Lambda^{t}\right)\mathbf{U}^{\top}\right)\mathbf{D}^{-\frac{1}{2}}, \tag{2.16}$$

where $\mathbf{U}\Lambda\mathbf{U}^{\top} = \mathbf{L}_{\mathrm{sym}}$ is the eigendecomposition of the symmetric normalized Laplacian. This reveals that the embeddings learned by DeepWalk are in fact closely related to the spectral clustering embeddings discussed in Part I of this book. The key difference is that the DeepWalk embeddings control the influence of different eigenvalues through T, i.e., the length of the random walk. Qiu et al. [2018] derive similar connections to matrix factorization for node2vec and discuss other related factorization-based approaches inspired by this connection.

2.2 Limitations and generalized encoder-decoders

So far all of the node embedding methods we have reviewed have been shallow embedding methods, where the encoder is a simply an embedding lookup (Equation 2.5). However, these shallow embedding approaches train unique embedding vectors for each node independently, which leads to a number of drawbacks:

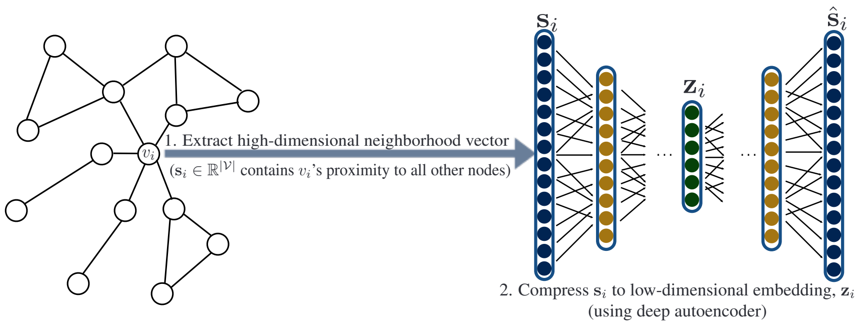
$\mathbf{s}_i$
$\hat{\mathbf{s}}_i$
$\mathbf{z}_i$
v_i
1. Extract high-dimensional neighborhood vector
($\mathbf{s}_i \in \mathbb{R}^{|\mathcal{V}|}$ contains v_i's proximity to all other nodes)
2. Compress $\mathbf{s}_i$ to low-dimensional embedding, $\mathbf{z}_i$
(using deep autoencoder)

Figure 2.4: To generate an embedding for a node, u, the neighborhood autoencoder approaches first extract a high-dimensional neighborhood vector $\mathbf{s}_i \in \mathbb{R}^{|\mathcal{V}|}$, which summarizes u's similarity to all other nodes in the graph. The $\mathbf{s}_i$ vector is then fed through a deep autoencoder to reduce its dimensionality, producing the low-dimensional $\mathbf{z}_u$ embedding.

1. No parameters are shared between nodes in the encoder (*i.e.*, the encoder is simply an embedding lookup based on arbitrary node ids). This can be statistically inefficient, since parameter sharing can act as a powerful form of regularization, and it is also computationally inefficient, since it means that the number of parameters in shallow embedding methods necessarily grows as $O(|\mathcal{V}|)$.

2. Shallow embedding also fails to leverage node attributes during encoding. In many large graphs nodes have attribute information (*e.g.*, user profiles on a social network) that is often highly informative with respect to the node's position and role in the graph.

3. Shallow embedding methods are inherently *transductive* [Hamilton et al., 2017], *i.e.*, they can only generate embeddings for nodes that were present during the training phase, and they cannot generate embeddings for previously unseen nodes unless additional rounds of optimization are performed to optimize the embeddings for these nodes. This is highly problematic for evolving graphs, massive graphs that cannot be fully stored in memory, or domains that require generalizing to new graphs after training.

To alleviate these limitations we can use more complex encoders that depend more generally on the structure and attributes of the graph. We close this chapter with a brief discussion of one approach to generalize the encoder-decoder paradigm using so-called*neighborhood autoencoders*. However, we will defer a discussion of the other major paradigm for generalized encoder-decoders, *graph neural networks (GNNs)*, since Part II of this book will discuss GNNs in detail.

Neighborhood autoencoder methods

Deep Neural Graph Representations (DNGR) Cao et al. [2016] and Structural Deep Network Embeddings (SDNE) Wang et al. [2016] address the first limitation of shallow embeddings outlined above: unlike the shallow embedding methods, they directly incorporate graph structure into the encoder algorithm using deep neural networks. The basic idea behind these approaches is that they use autoencoders—a well known approach for deep learning Hinton and Salakhutdinov [2006]—in order to compress information about a node's local neighborhood (Figure 2.4). DNGR and SDNE also differ from the previously reviewed approaches in that they use a *unary decoder* instead of a pairwise one.

In these approaches, each node, v, is associated with a neighborhood vector, $\mathbf{s}_v \in \mathbb{R}^{|\mathcal{V}|}$, which corresponds to v's row in the matrix $\mathbf{S}$ (recall that $\mathbf{S}$ contains the pairwise node similarities). The $\mathbf{s}_v$ vector contains v's similarity with all other nodes in the graph and functions as a high-dimensional vector representation of v's neighborhood. The autoencoder objective for DNGR and SDNE is to embed nodes using the $\mathbf{s}_v$ vectors such that the $\mathbf{s}_v$ vectors can then be reconstructed from these embeddings:

$$\textsc{dec}(\textsc{enc}(\mathbf{s}_v)) = \textsc{dec}(\mathbf{z}_v) \approx \mathbf{s}_v \tag{2.17}$$

In other words, the loss for these methods takes the following form:

$$\mathcal{L} = \sum_{u \in \mathcal{V}} \|\textsc{dec}(\mathbf{z}_v) - \mathbf{s}_v\|_2^2. \tag{2.18}$$

As with the pairwise decoder, we have that the dimension of the $\mathbf{z}_v$ embeddings is much smaller than $|\mathcal{V}|$ (the dimension of the $\mathbf{s}_v$ vectors), so the goal is to compress the node's neighborhood information into a low-dimensional vector. For both SDNE and DNGR, the encoder and decoder functions consist of multiple stacked neural network layers: each layer of the encoder reduces the dimensionality of its input, and each layer of the decoder increases the dimensionality of its input (Figure 2.4; see Hinton and Salakhutdinov [2006] for an overview of deep autoencoders).

SDNE and DNGR differ in the similarity functions they use to construct the neighborhood vectors $\mathbf{s}_v$ and also in the exact details of how the autoencoder is optimized. DNGR defines $\mathbf{s}_v$ according to the pointwise mutual information of two nodes co-occurring on random walks, similar to DeepWalk and node2vec. SDNE simply sets $\mathbf{s}_v \triangleq \mathbf{A}_v$, *i.e.*, equal to v's adjacency vector. SDNE also combines the autoencoder objective (Equation 2.17) with the Laplacian eigenmaps objective (Equation 2.6) Wang et al. [2016].

Note that the encoder in Equation (2.17) depends on the input $\mathbf{s}_v$ vector, which contains information about v's local graph neighborhood. This dependency allows SDNE and DNGR to incorporate structural information about a node's local neighborhood directly into the encoder as a form of regularization, which is not possible for the shallow embedding approaches (since their encoder depends only on the node id). However, despite this improvement, the autoencoder approaches still suffer from some serious limitations. Most prominently, the input dimension to the autoencoder is fixed at $|\mathcal{V}|$, which can be extremely costly and even intractable for graphs with millions of nodes. In addition, the structure and size of the autoencoder is fixed, so SDNE and DNGR are strictly transductive and cannot cope with evolving graphs, nor can they generalize across graphs. The graph neural network (GNN) paradigm introduced in Part II of this book will alleviate many of these shortcomings.