Chapter 26

Embedding and machine learning

Machine learning, especially neural network methods, is increasingly important in network analysis. As discussed in Ch. 16, much of the success of modern machine learning is thanks to a neural network's power to obtain *useful representations—embeddings*—of data. In this chapter, we will discuss the theoretical aspects of network embedding methods and graph neural networks.

26.1 Embeddings as representations

Network embedding refers to a variety of methods that represent each node (or edge, or even a whole network) as a point in a space. Formally, this means we seek to learn a function $V \to \mathbb{R}^d$ that maps each node[1] to a d-dimensional vector that meaningfully captures the network's structure. In other words, network embedding methods *embed* a network into a (vector) *space*.

You may be wondering: "But doesn't the adjacency matrix itself represent each node as a vector?" Very clever! Indeed, if we look at each row of the adjacency matrix, the row is a vector with a fixed dimension (N) that captures information about the node's neighbors. Then, why do we apply embedding methods? Don't we already have an "embedding"? Usually, embedding methods aim to find representations that are (1) compact,[2] (2) continuous, and (3) dense.

To understand why we need these three properties, let's think about the adjacency matrix as an embedding. First, assuming that the network is sparse like most real-world networks, each node representation vector would have few nonzero entries; the representation vectors are *sparse*. If we pick two random nodes from a large network, they are likely to share no common neighbors. In other words, they are completely orthogonal from the perspective of adjacency matrix embedding and we don't have *any*

[1] Or each edge $E \to \mathbb{R}^d$ or even network $G \to \mathbb{R}^d$.

[2] We can also think of embedding methods as dimensionality reduction methods, because the compactness leads to a reduced number of dimensions, often significantly reduced.

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

information about their relationship. This would be the case for most pairs of nodes for many real-world networks. Even if they share some common neighbors, it is very likely that the amount of information that can be gleaned from the shared nodes is negligible for most pairs.

Second, the adjacency vectors—assuming a non-weighted network—treats all neighbors exactly the same, which is quite limiting. For instance, suppose i has two neighbors u and v, where i and u share numerous common neighbors and i and v do not share any. It is reasonable to assume that the relationship between i and u is much stronger than that between i and v. Yet, the adjacency embedding—which only considers immediate neighbors—does not tell us anything about such relationships. The two neighbors are identical. A better representation would capture the difference.

Lastly, the adjacency vectors are big, of dimension N. We are not compressing the information in the original adjacency matrix into a compact representation. It not only produces undesirable sparsity and discontinuity, it is also inefficient, and high-dimensional spaces are notoriously poor settings for building predictive models. We need compactness.

What's a simple alternative to using the adjacency matrix? A low-rank approximation, found with matrix factorization:

Laplacian Eigenmaps Although the term "embedding" is popular recently, thanks to the advancement of deep learning and neural network methods, the idea of low-dimensional representation has a long history. One notable method is *Laplacian Eigenmaps* [48], which leverages the spectral properties of networks.

In fact, we have encountered this embedding method before, as it is spectral clustering (Sec. 25.7.2). The graph Laplacian is diagonalized and its k leading eigenvectors are used to represent the nodes in a k-dimensional space (specifically, for a node i, its jth embedding coordinate is the ith element of eigenvector $\mathbf{v}_j$). The only difference from earlier was that spectral clustering limited the dimensionality to the number of clusters being inferred. For a Laplacian eigenmap, we can take the dimensionality as a "hyperparameter" of the method.

Of course, many other approaches exist, such as non-negative matrix factorization (Sec. 16.7.1).

What can we do with embedded representations?

Embeddings are helpful for exploratory tasks such as visualization, but one major reason we seek embedded versions of nodes in a network is that these representations can help with subsequent ***machine learning tasks***. We may want to train a node classifier, for instance, and a classifier that uses node vectors instead of the network structure may be easier to set up, less costly to train, and may work better at predicting node classes. Node classes can be considered a form of attribute (Ch. 9) and we can even train a model to impute missing attributes by learning a combined representation of nodes and attributes. Another example is link prediction, which is essentially a node-pair classification task. And we can even in principle learn to generate synthetic networks by embedding the entire network and then learning a "decoder" mechanism that can

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

translate an embedding back to a network structure. In general, dense vectors may be more "learnable" than sparse network structure. Logistic regression (Sec. 16.2.1) with the embedding vectors as features is often used.

Transductive and inductive learning

One important distinction we should mention involves the kinds of predictions we make. Will our model make predictions within a network it has already seen, or will it be expected to accommodate an unseen network? For the former, as an example, suppose we have our network G and a $N \times p$ node attribute matrix $\mathbf{X}$. One of our attributes, say the last one, is incomplete for some nodes: X_{ip} is missing for some nodes $i \in V_{\text{missing}}$. Can we train a model to predict X_{ip} for those nodes? Our model is able to see all of G, all of $\mathbf{X}$ for nodes $i \in V \setminus V_{\text{missing}}$, and the non-missing columns of $\mathbf{X}$ for nodes $i \in V_{\text{missing}}$. We can use all this information to supervise the training and validation of our model, then apply it to the same network to impute X_{ip}.

This imputation problem is an example of *transductive learning* (we also can call it a *semi-supervised* problem). Because the model will be trained on the same network it will make its predictions on, we can't expect it to generalize to an entirely different network. Doing that is called *inductive learning*, which is a more difficult problem. Many other problems can be either transductive or inductive depending on the setting, such as link prediction in a seen or unseen network.

26.2 Language models and word2vec—embeddings come of age

Often but not always when we now say "embedding," researchers now refer to neural network methods. To explain common embedding methods, it is useful to first visit the idea of *language models* and *word embedding* methods, upon which many network embedding methods build.

The *word2vec* method [307] is one of the early neural network methods that heralded the recent boom of deep learning. Word2vec and related models build on the classical idea of "language models." To understand where the idea came from, let's imagine a basic approach to obtain word representations. Let us simply assume that the properties of any given word can be represented by a bunch of numbers—a vector. Each element of this vector answers a particular question about the property of the word. Say, the first element is about whether the word is a verb or not. The second element quantifies how concrete (vs. abstract) the word is. We can imagine preparing a bunch of questions and building a vector for each word. Once we build these vectors—representations—for every word, we can estimate the similarity between words based on the similarity (e.g., cosine similarity) of the vectors and do all sorts of other tasks. But the list of questions can be arbitrary and it will not be trivial to build these vectors. The question is: can we build a *meaningful* vector of a word so that it captures the *meaning* of every word coherently? But then, how do we know the *meaning* of a word and how can we represent it quantitatively?

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

Language models assume that the meaning of a word comes from the *contexts* in which that word appears. This idea can be traced to Ferdinand de Saussure, who lays an important foundation for modern linguistics. Saussure argued [123] that the meaning of a "sign" converges onto the same concept through the averaging of the speakers who use the sign to communicate:

> Among all the individuals that are linked together by speech, some sort of average will be set up: all will reproduce—not exactly of course, but approximately—the same signs united with the same concepts.

This idea was later more formalized into the *distributional hypothesis* by Zellig S. Harris, who argued [202], as summarized by Pantel [357]: "*words that occur in the same contexts tend to have similar meaning*." John R. Firth succinctly captures this idea by saying [159],

> You shall know a word by the company it keeps.

Therefore, we can study large corpora of natural language to understand the meaning of those words.

Language models In other words, the idea is that we can figure out the meaning of a word by examining the other words around it. Let's look at these three sentences:

> "The quick brown _____ jumps over the lazy dog."
> "He is cunning as a _____."
> "The _____ was already in the hen house."

Just by examining the words around the blank, we can see that the word that can fill the blank should represent something brown, quick, cunning, something that can jump and likes to go into hen houses. And you may be already thinking about a *fox*.

Let's approach it more formally. A language model is a statistical model of natural language, in a sense answering the question, "can we distinguish a *natural* document from gibberish?" Mathematically, this question can be written as: can we accurately estimate the probability of an observed sequence of words ($\Pr(\{w_1, w_2, \ldots, w_T\})$)? If we can assign high probability to *actual* sentences and low probability to *random* sequences of words, then we have a *good* language model. Note that this joint probability can be broken down into a product of conditional probabilities using the chain rule:

$$\Pr(w_1, w_2, \ldots, w_T) = \Pr(w_T \mid w_1, w_2, \ldots, w_{T-1}) \cdots \Pr(w_3 \mid w_1, w_2) \times \Pr(w_2 \mid w_1) \Pr(w_1). \quad (26.1)$$

In other words, if we can accurately estimate $\Pr(\text{target word} \mid \text{context words})$, we can construct a good language model. And this conditional probability captures the conceptual idea of language models—given the context (previous words), can we predict the word that comes next?[3]

[3] Also note that we can rearrange context and target words: "what would be the missing word in the middle, given the words around it?"

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

***N*-gram model** However, estimating the conditional probabilities directly from data is not feasible due to the exploding number of word combinations. A common simplification to address this issue is the *n-gram* model, where we only consider the previous n words, not all previous words. Namely, we assume that

$$\Pr(w_t \mid w_1, \ldots, w_{t-1}) \simeq \Pr(w_t \mid w_{t-n}, \ldots, w_{t-1}). \tag{26.2}$$

If we set n small enough, we can find most n-gram combinations and count their actual occurrences. Then we can estimate the conditional probabilities by using our counts. This is the n-gram language model.

Word2vec takes a different approach. Instead of estimating the conditional probabilities directly, we assume that there exists a meaningful vector representation for every word that allows us to estimate the conditional probabilities without counting the actual occurrences.

Imagine every word has two vector representations that capture their meaning very well. For convenience, we call them *query* and *key* vectors[4] and denote as $\mathbf{q}_i$ and $\mathbf{k}_i$ (for word i). Now we assume that

$$\Pr(w_t \mid w_{t-n}, \ldots, w_{t-1}) \simeq f(\mathbf{k}_{w_t}, \mathbf{q}_{w_{t-n}}, \ldots, \mathbf{q}_{w_{t-1}}). \tag{26.3}$$

In other words, instead of directly estimating $\Pr(w_t \mid w_{t-n}, \ldots, w_{t-1})$ from the data, we imagine a function f that can estimate the conditional probability based on the *representations of the words (vectors)*. A database is a nice analogy. The *query* to the database is the sequence of n-gram context words. When the query matches the *key* of a target word, we return this target word. Unlike a real database, where we identify a single perfect and unique match, everything here is probabilistic. Our training goal becomes learning the vectors from data (the corpus) to make this work.

Skip-gram model The word2vec method suggested a further simplification to the n-gram model. It asks, why don't we just decompose the n-gram conditional probability into a product of 1-gram conditional probabilities (and also consider both preceding and following contexts)? This gives

$$\Pr(w_t \mid w_{t-l}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+l}) \simeq \prod_{t-l \le i \le t+l} \Pr(w_t \mid w_i). \tag{26.4}$$

This is called the "skip-gram" model, where we *skip* all the other words but one from the n-gram formulation. Then, we can focus on a much simpler function with just two vectors: $f(\mathbf{k}_{w_i}, \mathbf{q}_{w_j})$. Specifically, the word2vec model proposes to use the *softmax* function of the dot product $\mathbf{k} \cdot \mathbf{q} = \mathbf{k}^\mathsf{T}\mathbf{q}$ between the two vectors:

$$f(\mathbf{k}_{w_t}, \mathbf{q}_{w_c}) = \frac{\exp(\mathbf{k}_{w_t} \cdot \mathbf{q}_{w_c})}{\sum_i \exp(\mathbf{k}_{w_i} \cdot \mathbf{q}_{w_c})}. \tag{26.5}$$

[4] They are analogous to the query and key vectors in the self-attention mechanism used in Transformer models like BERT and GPT [277].

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

Then,

$$\Pr(w_1, \ldots, w_T) \simeq \prod_t \prod_{c \in C_t} \Pr(w_t \mid w_c)$$
$$= \prod_t \prod_{c \in C_t} \frac{\exp(\mathbf{k}_{w_t} \cdot \mathbf{q}_{w_c})}{\sum_i \exp(\mathbf{k}_{w_i} \cdot \mathbf{q}_{w_c})}, \tag{26.6}$$

where C is the set of words that are considered as context words (usually $w_{t-l}, \ldots, w_{t+l}$ not including w_t, with l being the size of the context window). The skip-gram model tries to maximize the log probability $\log \Pr(w_t \mid w_i)$.

Hierarchical softmax and negative sampling Once we can compute $\Pr(w_t \mid w_c)$, we can initialize every word vector randomly and then go through actual natural sentences to generate context–target word pairs. For every context–target word pair from our data, we can estimate $\Pr(w_t \mid w_c)$ and use the backpropagation algorithm to learn the vectors. All standard stuff at this point. Yet, this is computationally challenging because we have to compute $\sum_i \exp(\mathbf{k}_{w_i} \cdot \mathbf{q}_{w_c})$ every time for all possible words. The word2vec model proposes another simplifying innovation here. Instead of directly computing this summation, the authors suggest two methods to drastically reduce the computational complexity: *hierarchical softmax* and *negative sampling* (NS). The hierarchical softmax method is a tree-based data structure that allows us to compute the summation in $\log(n)$ time, where n is the number of words. Although it is a very clever algorithm, it is not as popular as NS. NS is a simple idea that belongs to the class of methods called "*contrastive learning*," which suggests that, instead of computing the probability directly, we can solve *another problem*, that of distinguishing the actual word w_t from a small set of randomly sampled words. If our representation and conditional probability function can distinguish the actual answer very well from random noise, then we can argue that the representation is good (recall the basic idea of language models). Formally, we can write the NS objective as

$$\log \sigma(\mathbf{k}_{w_t} \cdot \mathbf{q}_{w_c}) + \sum_{i=1}^{b} \mathbb{E}_{w_i \sim \Pr_n(w)} \left[\log \sigma(-\mathbf{k}_{w_i} \cdot \mathbf{q}_{w_c}) \right], \tag{26.7}$$

where $\sigma(x) = 1/(1+\mathrm{e}^{-x})$ is the sigmoid function and b is the number of negative pairs we sample from the "noise" or "negative" distribution $\Pr_n$. (Mikolov et al. [307] used $\Pr_n(w) \propto n_w^{3/4}$, where n_w is the (unigram) count of occurrences of word w.) This is called the "*negative sampling*" model or *skip-gram negative sampling*. We seek vectors that align the word and context (first term) and oppose the negative word and context (second term; note the minus sign on the dot product). It allows us to learn by sampling a few random words rather than computing the softmax over all possible words.

26.3 From writing to walking: embedding networks

What does all this have to do with networks? Once word2vec achieved great success at natural language processing tasks, particularly at capturing *analogical relationships*

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

between words, many researchers recognized the generalizability of the word2vec model and the idea of language modeling itself. Note that in the word2vec model, all we need is a fixed vocabulary[5] and lots of example sentences, where a sentence is simply a sequence of "words" in the vocabulary. There isn't, however, any restriction about the nature of the "words." Any set of entities where we can find their natural *sequences* can be considered as "words" and "sentences."

Not surprisingly, networks were one such generalization. Networks consist of nodes and they can be considered as "words." How can we get the natural sequences of nodes? One place to start is a *random walk*. If we perform a random walk on a network, we will have a sequence of nodes that organically captures the structural information of the network. If two nodes are close in a network, they will likely appear nearby in more random walk trajectories than another pair of nodes that are far apart. If two nodes share many neighbors together, they are more likely to co-appear in random walk trajectories than another pair of nodes that do not share any neighbors.

The first model that applies this line of thinking is "DeepWalk" [369]. The idea is exactly as laid out above. We generate many random walks from a network and then feed them as "sentences" to the word2vec model. Soon, another model called "node2vec" [193] was published with a similar idea, but an additional twist of using a biased random walk. Node2vec argued that we can modulate the nature of the random walks to obtain different representations that focus on different aspects of the network structure. For instance, if we want to capture the *community structure* of a network, we can bias the random walk so that it is more likely to stay in the same community (like BFS); if we want to capture the *hierarchical structure* of a network, we can bias the random walk so that it explores the network more (like depth-first search). These two models became foundational for many following models that adopt the paradigm of language models.

There was an interesting, subtle difference in the implementation of DeepWalk and node2vec. As we discussed, the softmax function is difficult to calculate and word2vec proposed using either hierarchical softmax or negative sampling to speed it up. DeepWalk's implementation adopted hierarchical softmax while node2vec used negative sampling. It turned out that this choice was actually an important one that changed how the two behave and perform.

The hidden bias of negative sampling Analysis has shown that negative sampling has an implicit bias [325, 249]. To see this, we need to first look at a similar, yet unbiased model called "*noise contrastive estimation*" (NCE) [197]. NCE is a general contrastive estimator that allows us to estimate a probability model $\Pr_m(x)$ of the following form:

$$\Pr_m(x) = \frac{f(x;\theta)}{\sum_{x' \in \mathcal{X}} f(x';\theta)}, \tag{26.8}$$

where f is a non-negative function of x and θ is a parameter vector. The word2vec model can be considered a special case of NCE where $f(x) = \exp(x)$ and $x = \mathbf{k}_i^\top \mathbf{q}_j$. NCE tries to solve the same logistic regression problem as the negative sampling model, but

[5] How about new words? It is still possible to *inductively* learn the vector representations of new words based on those of existing words.

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

using Bayesian inference. Given one positive example and b randomly sampled negative examples, we take as prior probabilities for the positive and negative samples:

$$\Pr(Y_j = 1) = \frac{1}{b+1}, \;\; \Pr(Y_j = 0) = \frac{b}{b+1}. \tag{26.9}$$

Here, the positive example is sampled from $\Pr_m(j)$ and negative examples are sampled from a noise distribution $p_0(j)$,

$$\Pr(j \mid Y_j = 1) = \Pr_m(\mathbf{q}_i \cdot \mathbf{k}_j), \;\; \Pr(j \mid Y_j = 0) = p_0(j). \tag{26.10}$$

Based on Bayes' rule, the posterior probability of the positive example is

$$\begin{aligned} \Pr_{\text{NCE}}(Y_j = 1 \mid j) &= \frac{\Pr(j \mid Y_j = 1)\Pr(Y_j = 1)}{\sum_{y \in \{0,1\}} \Pr(j \mid Y_j = y)\Pr(Y_j = y)} \\ &= \frac{\Pr_m(\mathbf{q}_i \cdot \mathbf{k}_j)}{\Pr_m(\mathbf{q}_i \cdot \mathbf{k}_j) + b p_0(j)}, \end{aligned} \tag{26.11}$$

which can be written in the form of a sigmoid function:

$$\begin{aligned} \Pr_{\text{NCE}}(Y_j = 1 \mid j) &= \frac{1}{1 + b p_0(j)/\Pr_m(\mathbf{q}_i \cdot \mathbf{k}_j)} \\ &= \frac{1}{1 + \exp\left[-\ln f(\mathbf{q}_i \cdot \mathbf{k}_j) + \ln p_0(j) + c\right]}. \end{aligned} \tag{26.12}$$

The negative sampling estimator $\Pr_{\text{NS}}$ is similar to the NCE estimator, and can be written as

$$\begin{aligned} \Pr_{\text{NS}}(Y_j = 1 \mid j) &= \frac{1}{1 + \exp(-\mathbf{q}_i \cdot \mathbf{k}_j)} \\ &= \frac{1}{1 + \exp\left[-\left(\mathbf{q}_i \cdot \mathbf{k}_j + \ln p_0(j) + c\right) + \ln p_0(j) + c\right]} \\ &= \frac{1}{1 + \exp\left[-\ln f'(\mathbf{q}_i \cdot \mathbf{k}_j) + \ln p_0(j) + c\right]}. \end{aligned} \tag{26.13}$$

By comparing the two estimators, we can see that the negative sampling estimator is a special case of the NCE estimator with $f'(\mathbf{q}_i \cdot \mathbf{k}_j) = \exp(\mathbf{q}_i \cdot \mathbf{k}_j + \ln p_0(j) + c)$. This means that the negative sampling word2vec model is an unbiased estimator for the following probability model:

$$\Pr_{\text{w2v-NS}}(j \mid i) = \frac{p_0(j)\exp(\mathbf{q}_i \cdot \mathbf{k}_j)}{\sum_{j'} p_0(j')\exp(\mathbf{q}_i \cdot \mathbf{k}_{j'})}. \tag{26.14}$$

In other words, the negative sampling model of the word2vec model is a biased estimator of the original word2vec model, but an unbiased estimator of a modified model where the word similarity represents the *deviation* from $p_0(j)$, or the information about the words that is *not* captured by the noise distribution, which is the frequency of each word.

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

The implicit bias of the negative sampling model has a profound impact when applied to random walks on networks. The random walk is naturally biased towards the nodes with higher degrees, because whenever we sample a *neighbor* we preferentially sample a node with higher degree (the friendship paradox; Ch. 21). This degree bias is, however, captured by the noise distribution $p_0(j)$. Because of this, the resulting embedding of the negative sampling model captures the deviation from what is expected based on the degree alone. In other words, the bias of the negative sampling model exactly *negates* the bias of the random walk, producing embedding vectors that are free of the degree bias!

Although DeepWalk and node2vec methods look like more or less the same method when we don't apply any biased random walk for node2vec, the subtle difference in the choice of estimation method in their implementations has a profound impact on the resulting embedding vectors, usually producing much better results for node2vec.

Other embedding methods

There exist a plethora of other methods for graph embedding and it will be impossible to cover all of them in this chapter. Let us mention a few of the most popular methods with unique ideas.

LINE (Large-scale Information Network Embedding) [453], a simpler special case of DeepWalk, aims to encode the "proximity" between nodes into a dense embedding. LINE adopts the ideas of word2vec and seeks to model from random walks the probability for a directed edge $\Pr(j \mid i) = \exp(\mathbf{u}_j \cdot \mathbf{v}_i)/\sum_{s=1}^{N} \exp(\mathbf{u}_s \cdot \mathbf{v}_i)$, where $\mathbf{v}_i$ and $\mathbf{u}_j$ are node vectors and context vectors, respectively. Crucially, the same node gets different vectors depending on whether it is treated as a context for a walk or the target of the walk itself. LINE's overall objective is to learn the vectors which maximize $\sum_{i,j\in E} A_{ij} \log p(j \mid i)$. It also adopts NS, leading to an objective function for each edge i, j that mirrors Eq. (26.7) with $\mathbf{k} \to \mathbf{u}$ and $\mathbf{q} \to \mathbf{v}$. The negative distribution $\Pr_n(v) = k_v^a / \sum_{v'} k_{v'}^a$ [6] accounts for the overall degree distribution of the network.

The ComplEx method [463] shows success at link prediction by using a matrix factorization technique similar to many others (such as Laplacian Eigenmaps), but with the twist of allowing for complex-valued vectors as the representations. Among other motivations, directed networks lead to non-symmetric matrices, a problem we mostly avoided in Ch. 25, and we may encounter complex eigenvalues and eigenvectors. Embracing complex values and using this for link prediction was very successful.

ComplEx was tailored for *knowledge graphs* (Ch. 27) as are many other methods. In a knowledge graph, links are semantic triples ("Rome *IsA* City") that represent factual statements. Nodes are identified by words or phrases, and this brings in many ways to use word embeddings, either as the node embeddings or, more often, as *part* of learning the node embeddings. The TransE [65] method, for example, was designed around translation of words to learn embeddings; the factual relationships should hold across languages. Other knowledge graph methods include DistMult [501], which shares some similarities with TransE, and RESCAL [346], which uses tensor factorization instead of matrix factorization.

[6] The authors followed word2vec and used $a = 3/4$; $a = 1$ is also common.

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

It's common to treat the space we embed in as Euclidean but there is no need to do so. Indeed, *hyperbolic* spaces have properties of interest specifically for modeling networks. A metric defined on a hyperbolic geometry can incorporate heterogeneous degree distributions, transitive closure, and hierarchy more naturally than in Euclidean space [254, 60]. This has led to researchers pursuing embedding methods specific to hyperbolic spaces [347, 98].

Finally, a large class of methods called graph neural networks incorporate embedding as part of their function. We discussed these briefly in Sec. 16.7.3 and we'll return to them in Sec. 26.5.

26.4 Embedding as matrix factorization

Soon after word2vec emerged, researchers seeking to understand what it was calculating showed that it was implicitly performing a matrix factorization [269]. This is powerful to know. Matrix factorization is very useful for data analysis, as we see whenever we use singular value decomposition (SVD), and many other embedding approaches use it explicitly. Indeed, in the context of natural language processing, SVD powers latent semantic analysis (LSA) a classical NLP technique [261]. In a way, it's both surprising (because it looks so different) and not surprising that the more advanced word2vec method does something similar.

Let's derive the factorization occurring in word2vec. Then we'll discuss some network-specific embedding methods in this context.

Negative sampling as implicit matrix factorization

First, we summarize an influential discovery by Levy and Goldberg [269]: the sampling strategy used by word2vec (and adopted for networks by DeepWalk and its descendents) is implicitly factorizing a matrix $\mathbf{M}$.

Recall that a language model seeks to understand the co-occurrence between words $w \in V_W$ and contexts $c \in V_C$, the surrounding words; for a word w_i, the surrounding L-sized context is $w_{i-L}, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{i+L}$. Let D be the multiset of observed word–context pairs and use $\#(w, c)$ to denote the number of occurrences of pair $(w, c) \in D$. Marginalizing gives counts for w and c, $\#(w) = \sum_{c' \in V_C} \#(w, c')$ and $\#(c) = \sum_{w' \in V_W} \#(w', c)$, respectively. Our embedding goal is to find vector representations for words and contexts. Let $\mathbf{w} \in \mathbb{R}^d$ be the vector representing word $w \in V_W$ and likewise $\mathbf{c} \in \mathbb{R}^d$ for context $c \in V_C$, where d is the embedding dimension. (Previously we used $\mathbf{k}$ and $\mathbf{q}$.) Generally only the word vectors are used for subsequent NLP tasks, but both are necessary for optimization.

From Eq. (26.7), the negative sampling objective is

$$\begin{aligned}\ell &= \sum_{w \in V_W} \sum_{c \in V_C} \#(w, c) \Big(\log \sigma(\mathbf{w} \cdot \mathbf{c}) + b\, \mathbb{E}_{c_N \sim \Pr_D} \left[\log \sigma\left(-\mathbf{w} \cdot \mathbf{c}_N\right) \right] \Big) \\ &= \sum_{w \in V_W} \sum_{c \in V_C} \#(w, c) \log \sigma(\mathbf{w} \cdot \mathbf{c}) + \sum_{w \in V_W} \#(w) \left(b\, \mathbb{E}_{c_N \sim \Pr_D} \left[\log \sigma\left(-\mathbf{w} \cdot \mathbf{c}_N\right) \right] \right), \end{aligned} \tag{26.15}$$

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

where b is the number of negative samples and C_N is the sampled context drawn from the empirical distribution $\Pr_D(c) := \#(c)/|D|$.[7] Next, pull the true context out of the NS expectation:

$$\begin{aligned}\mathbb{E}_{c_N \sim \Pr_D}\left[\log \sigma\left(-\mathbf{w}\cdot\mathbf{c}_N\right)\right] &= \sum_{c_N \in V_C} \frac{\#(c_N)}{|D|} \log \sigma\left(-\mathbf{w}\cdot\mathbf{c}_N\right) \\ &= \frac{\#(c)}{|D|} \log \sigma\left(-\mathbf{w}\cdot\mathbf{c}\right) + \sum_{c_N \in V_C \setminus \{c\}} \frac{\#(c_N)}{|D|} \sigma\left(-\mathbf{w}\cdot\mathbf{c}_N\right). \end{aligned} \tag{26.16}$$

For sufficiently large d, we can assume each product $\mathbf{w}\cdot\mathbf{c}$ takes on a value independently of the others, letting us treat ℓ as a function of independent $\mathbf{w}\cdot\mathbf{c}$ terms. Using this and Eq. (26.16), the term specific to the pair (w, c) is

$$\ell(w,c) = \#(w,c)\log\sigma(\mathbf{w}\cdot\mathbf{c}) + b\,\#(w)\frac{\#(c)}{|D|}\log\sigma(-\mathbf{w}\cdot\mathbf{c}). \tag{26.17}$$

Because we seek to optimize this objective, we find the partial derivative with respect to $x := \mathbf{w}\cdot\mathbf{c}$:

$$\frac{\partial \ell}{\partial x} = \#(w,c)\,\sigma(-x) - b\,\#(w)\frac{\#(c)}{|D|}\,\sigma(x). \tag{26.18}$$

Simplifying and setting equal to zero gives

$$e^{2x} - \left(\frac{\#(w,c)}{b\,\#(w)\frac{\#(c)}{|D|}} - 1\right)e^x - \frac{\#(w,c)}{b\,\#(w)\frac{\#(c)}{|D|}} = 0. \tag{26.19}$$

Letting $y := \mathrm{e}^x$, this equation becomes a quadratic of y, which has two solutions. The first, $y = -1$, is invalid (since $\mathrm{e}^x > 0$) while the second (using $\mathbf{w}\cdot\mathbf{c} = \log(y)$) is

$$\mathbf{w}\cdot\mathbf{c} = \log\left(\frac{\#(w,c)}{b\,\#(w)\frac{\#(c)}{|D|}}\right) = \log\left(\frac{\#(w,c)|D|}{\#(w)\#(c)}\right) - \log b. \tag{26.20}$$

We can now see better what is happening. The expression

$$\log\left(\frac{\frac{\#(w,c)}{|D|}}{\frac{\#(w)}{|D|}\frac{\#(c)}{|D|}}\right) = \log\left(\frac{\#(w,c)|D|}{\#(w)\#(c)}\right) \tag{26.21}$$

is the *pointwise mutual information* (PMI) for (w, c), estimated from the corpus D. It tells us how strongly associated w and c are by comparing their joint distribution to the joint distribution if they were independent. And lastly, the matrix being factorized by NS, since this is equal to a dot product for each term, has elements defined by the PMI: $M_{ij} = PMI(w_i, c_j) - \log b$. For $b > 1$, we can think of this as a *shifted* PMI matrix.

[7] The original word2vec and DeepWalk models drew negative contexts from $\propto \#(c)^{3/4}$ instead of $\propto \#(c)$. This difference does not substantially change our results [269] and avoiding the 3/4 exponent also lets us make clear graph-theoretic connections we'll use shortly.

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

However, we assumed every $\mathbf{w}_i \cdot \mathbf{c}_j$ was independent. If this is not true, the loss for a given pair (Eq. (26.17)) depends on the observed counts of the pair together ($\#(w, c)$) versus the expected number of negative samples ($b\,\#(w)\#(c)/|D|$) and we can instead think of NS as performing a *weighted* factorization, where the solution is biased in favor of more frequent pairs, the same bias discussed above.

Having a better understanding of what NS is doing "behind-the-scenes" motivates spectral methods, the original approach to embedding, which can be more computationally efficient than using stochastic gradient descent and more theoretically tractable. Indeed, Levy and Goldberg [269] compare NS to SVD on the shifted PPMI[8] and show that it achieves better optimization of the loss function than NS. That said, NS performed better at subsequent linguistic tasks, such as finding word analogies, probably due to the weighted factorizing, as PMI solutions are known to be over-affected by rare observations. It is good to understand what NS is doing here because it motivates in NLP the search for better weighted factorization methods [269].

Network embedding as factorization

Network embedding methods such as DeepWalk and node2vec follow the spirit of word2vec closely, so it stands to reason that they too are implicitly performing matrix factorization. Indeed, Qiu et al. [382] show that this is exactly the case.

For brevity, we focus our discussion on LINE [453], which is a special case of DeepWalk. As mentioned, LINE follows the word2vec model and uses NS to learn vector representations that help predict node associations during random walks. The objective function they derive is nearly identical to that of word2vec:

$$\ell = \sum_{i \in V} \sum_{j \in V} A_{ij} \left(\log \sigma(\mathbf{v}_i \cdot \mathbf{u}_j) + b \mathbb{E}_{s \sim \Pr_n} \left[\log \sigma(-\mathbf{v}_i \cdot \mathbf{u}_s) \right] \right), \tag{26.22}$$

where, as discussed in Sec. 26.4, for simplicity, we now take $\Pr_n(v) = k_v/2M$. (For a weighted network, k_v and M are the total edge weights for a node v and the entire graph, respectively.) Our previous analysis for word2vec transfers over almost exactly given the similar objective function, meaning that LINE also performs an implicit matrix factorization [382]:

$$\mathbf{v}_i \cdot \mathbf{u}_j = \log \left(\frac{2M A_{ij}}{b k_i k_j} \right) \tag{26.23}$$

or, in matrix form,

$$\mathbf{V}^\intercal \mathbf{U} = \log(2M \mathbf{D}^{-1} \mathbf{A} \mathbf{D}^{-1}) - \log b, \tag{26.24}$$

where $\mathbf{V}$ and $\mathbf{U}$ contain the node and context vectors as columns, $\mathbf{D}$ is the diagonal degree matrix, and the log works element-wise. Notice the resemblance between the matrix $\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1}$ and some of the matrices we encountered in Ch. 25.

LINE is a special case of DeepWalk, and Qiu et al. [382] show that DeepWalk factorizes $\log \left(2M \left(L^{-1} \sum_{r=1}^{L} \left(\mathbf{D}^{-1}\mathbf{A} \right)^r \right) \mathbf{D}^{-1} \right) - \log b$. (We recover LINE when the

[8] The *positive* PMI matrix is defined as $PPMI = \max(PMI, 0)$. This is often used in NLP because the PMI matrix estimated from a corpus will have entries $\log 0 = -\infty$ for any pairs not observed in the corpus. The PPMI ensures the matrix is well-defined and sparse. (Other fixes, such as Dirichlet smoothing, will not ensure sparsity.)

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

context window $L = 1$.) Qiu et al. [382] also derive the implicit factorization for node2vec and use these results to demonstrate a superior method called NetMF that explicitly factorizes these matrices using SVD instead of implicitly using NS.

26.5 Graph neural networks

The 2010s conclusively demonstrated that neural networks had finally fulfilled their promise, addressing and in some cases even solving longstanding problems of computer vision, speech recognition, and natural language processing. And now neural networks are also making inroads with network data (Ch. 16).

We're going to start using "graphs" more consistently when referring to our data to avoid any confusion with the neural networks.

Just as neural network methods have wildly succeeded in NLP, so have neural methods proliferated for graphs. These networks also learn representations—embeddings—but often also seek to incorporate node and link attributes (Ch. 9) into their representations. Using these as features can enable more and better predictive models, including *inductive* models that work on entirely unseen graphs.

The challenge with graphs, unlike other forms of data such as written text or spoken audio, is that we must be *permutation-invariant*: shuffling the neighborhood of a node should change nothing, but shuffling the context of a word should change (or destroy) the meaning of the writing. This need for permutation invariance extends to graph isomorphism (Sec. 12.6), and two isomorphic graphs (including attributes) should ideally lead to the same representations; we return to this point later.

Recall the basic feedforward neural network propagates data through a collection of layers where linear combinations of values at one layer are passed through a nonlinear activation function when arriving at the next layer, that is,

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l-1)}\mathbf{h}^{(l-1)}, \quad \mathbf{h}^{(l)} = \sigma(\mathbf{a}^{(l)}), \tag{26.25}$$

where $\mathbf{W}^{(l-1)}$ represents the matrix of weights connecting layers $l-1$ and l, $\mathbf{h}^{(l)}$ is the vector of activations at layer l, and $\sigma(\cdot)$ is a nonlinear function, often a sigmoid but not always. Bias units for each layer, which act like the intercept in a linear model, are absorbed into the weight matrices. Our input data $\mathbf{x}$ serve as the original activations, $\mathbf{h}^{(0)} = \mathbf{x}$. The parameters $\{\mathbf{W}\}$ of the network are learned with optimization, often by "backpropagating" errors on training data using stochastic gradient descent and possibly various regularization techniques. Again, for practitioners, all standard stuff.

Modern graph neural networks (GNNs) have coalesced around a framework of learning functions that iteratively update a node's representation by aggregating it with the representations of its neighbors in the graph, including possibly higher-order neighbors such as next-nearest neighbors. Let $\mathbf{x}_i$ be the attribute vector for node i and $\mathbf{h}_i^{(l)}$ be the representation (or activation) of i at the lth layer of the network, with $\mathbf{h}_i^{(0)} = \mathbf{x}_i$ (we feed the original features into the neural network). (Note in principle the dimensionality of $\mathbf{h}$ may be different for different layers.) Broadly, a GNN's lth iteration

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

for node i comes from two learned functions:

$$\mathbf{a}_i^{(l)} = \text{AGGREGATE}^{(l)}\left(\left\{\mathbf{h}_j^{(l-1)} \,\middle|\, j \in N_i\right\}\right), \tag{26.26}$$

$$\mathbf{h}_i^{(l)} = \text{COMBINE}^{(l)}\left(\mathbf{h}_i^{(l-1)}, \mathbf{a}_i^{(l)}\right), \tag{26.27}$$

where N_i are the (possibly higher-order) neighbors of i. Choices for AGGREGATE include mean, max, and sum, which can encapsulate many different GNN architectures, while vector concatenation is often used for COMBINE. Lastly, when seeking a representation $\mathbf{h}_G$ for the entire graph, a permutation-invariant READOUT function is applied to the final iteration L:

$$\mathbf{h}_G = \text{READOUT}\left(\left\{\mathbf{h}_i^{(L)} \,\middle|\, i \in V\right\}\right). \tag{26.28}$$

Some authors have also pursued more complex pooling READOUT functions [21] but a simple sum is often used.

Many architectures are described by these functions, including graph convolutional networks (GCNs) that explicitly feed the graph's adjacency matrix into the neural network. For example, Kipf and Welling [242] use the following layer-wise propagation step:

$$\mathbf{H}^{(l)} = \sigma\left(\tilde{\mathbf{D}}^{-1/2}(\mathbf{I}_N + \mathbf{A})\tilde{\mathbf{D}}^{-1/2}\mathbf{H}^{(l-1)}\mathbf{W}^{(l-1)}\right), \tag{26.29}$$

where $\mathbf{H}^{(l)} \in \mathbb{R}^{N\times d}$ are the layer representations, $\mathbf{W}^{(l)}$ are the corresponding network weights, and $\sigma(\cdot)$ is a nonlinear (not necessarily sigmoid) activation function. Here the adjacency matrix has been augmented with self-loops ($\mathbf{I}_N$) which are included in the diagonal rescaling matrices $[\tilde{\mathbf{D}}]_{ii} = k_i + 1$. (Notice the similarities with the Laplacian matrices we saw in Ch. 25.) By "hitting" the output of each layer of the neural network with the adjacency matrix, the network is forced into accounting for the graph structure, a form of *masking*. However, by explicitly including the adjacency matrix, the GCN is unable to accommodate an unseen network structure, unlike follow-up methods like GraphSAGE [200] whose aggregate methods only sample features from neighborhoods (usually first and second neighbors) and thus can handle novel graphs and perform *inductive* learning.

One of the problems faced with NLP methods and the network methods they inspire is learning long-range relationships. The solutions[9] devised for language, the attention mechanism and transformers, have been wildly successful, and have been adopted for graph structured data. *Graph attention networks* [475] (GATs; here we described what is called GATv2 [77]) learn a scoring function $e : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ for every edge i, j which captures how important the features of neighbor j are to node i (we omit the layer index l):

$$e(\mathbf{h}_i, \mathbf{h}_j) = \mathbf{a}^\intercal \text{LeakyReLU}\left(\mathbf{W}\left[\mathbf{h}_i || \mathbf{h}_j\right]\right), \tag{26.30}$$

where $\mathbf{a}$ and $\mathbf{W}$ are learned parameters whose dimensionalities depends on the network architecture, $[\cdot||\cdot]$ denotes concatenation, $\text{LeakyReLU}(y) = y$ if $y > 0$, βy otherwise,[10]

[9] Recurrent neural networks (RNNs) have long struggled with this, with LSTMs being a solution. Attention works even better, allowing networks to learn long-range associations without the long-range computational paths of RNNs, speeding up training and avoiding vanishing and exploding gradients.

[10] A common choice in GAT is $\beta = 0.2$.

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

and bias units are again omitted for brevity. Attention scores come from a softmax of the edge scores,

$$\alpha_{ij} = \frac{\exp(e(\mathbf{h}_i, \mathbf{h}_j))}{\sum_{j' \in N_i} \exp(e(\mathbf{h}_i, \mathbf{h}_{j'}))}, \tag{26.31}$$

and the next layer's representations are computed from an attentive weighted average,

$$\mathbf{h}'_i = \sigma\left(\sum_{j \in N_i} \alpha_{ij} \mathbf{W} \mathbf{h}_j\right). \tag{26.32}$$

For simplicity we described a single attention mechanism, but common practice is to learn K parallel attention "heads" which are then concatenated or averaged. Many other important details such as hyperparameter values and choice of regularizer will depend on the researcher and the task at hand, but GAT networks have proven successful at transductive and inductive prediction tasks.

Expressiveness and isomorphism

As we mentioned, neural networks intended for graph-structured data have to deal with permutation invariance in ways that networks working with images or other data do not—the set of neighboring nodes in a graph is order-invariant, the set of neighboring pixels in an image is not. The *graph isomorphism* (GI) problem, understanding whether two graphs are the same regardless of how we order or label the nodes, is thus highly relevant to learning GNNs: GNNs trained on two apparently different but actually isomorphic graphs should converge on the same learned functions. We often need to ask whether the GNN architecture is expressive enough to accomplish this.

GI is an interesting computational problem. For many years, it had its own location, called "GI," in the hierarchy of complexity classes, not in P and not in NP. Recently, it has been shown to be in quasi-polynomial time, with complexity $2^{O((\log N)^c)}$, worse than polynomial time but not as bad as exponential [18, 19]. An influential algorithm that tests for isomorphism is called the *Weisfeiler–Lehman* (WL) algorithm [487]. Roughly speaking, WL works by "coloring" nodes of the graph, then updating the coloring of a node using the colors of its neighbors. These colors are found with hash functions that ensure different colorings are represented differently.

The color-passing idea of WL is very closely connected to what GNNs do, passing the node representations as messages during learning. Indeed, many researchers have used this connection as the foundation to understand what GNNs can and cannot do [293, 320, 497, 32]. For example, many of the GNNs we have encountered above cannot distinguish graphs that WL can distinguish, meaning they are less expressive or else powerful. Work continues [32] to improve upon this.

Open problems and the future

GNNs struggle in several key areas. One is computational complexity. Often the methods require significant training time and memory. Many methods are currently limited to small networks or small subgraphs of larger networks. As an extreme example, the

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

Graphormer, a recently introduced, successful application of the Transformer natural language network to GNNs, is intended for networks of at most only dozens of nodes. GNNs currently struggle with larger and more complex graphs and attributes, especially heterogeneous attributes.

Because GNNs work by passing updates between nodes, they suffer when it comes to long-range signals. Skip connections, which allow for information to bypass some layer updates, can be a remedy, but the problem remains substantial. Bottlenecks (cf. Sec. 25.5) often prevent appropriate learning [12]. This leads to "over-smoothing" where the network fails to retain differences in representations between obviously different nodes—as we move outward in a graph, the exponentially growing number of nodes lead to information that must be squashed by the network into a representation of fixed dimensionality. This remains a fundamental challenge. Natural language networks struggled with similar problems for decades, before recent progress led away from recurrent networks to attention and Transformer models. While the problems with graphs are different, the hope is that, like with natural language, the problems can be overcome.

Lastly, robustness and interpretability are concerns for users of GNNs just as with other neural network models. Adversarial data and even noise (Ch. 10) can be a particular problem for GNNs [119, 514], leading us to worry about our robustness. Likewise, in practice, GNNs can be sensitive to choices of architecture and hyperparameter settings. Even design choices, like how to incorporate graph features such as centrality measures or shortest path lengths into the GNN, are often approached in an ad hoc manner. And these models are far over to one side of the interpretability–flexibility curve (Chs. 3 and 16): like many complex machine learning methods, they are black boxes that we can struggle to understand.

26.6 Summary

Embedding network nodes and edges is big business. Embeddings should be compact, continuous and dense, and these properties allow for novel ways to work with network data. We've already encountered ways to embed networks, such as the spectral methods of Ch. 25 but more and better embedding techniques continue to be introduced.

Machine learning and embedding are closely aligned. Translating network elements to embedding vectors and sending those vectors as features to a subsequent predictive model often leads to a simpler, more performant model than designing a model that works directly with the network. Embeddings help with network learning tasks, from node classification to link prediction. We can even embed entire networks and then use models to summarize and compare networks.

But the relationship is also a two-way street. Not only does machine learning benefit from embeddings, but embeddings benefit from machine learning. Inspired by the incredible recent progress with natural language data, embeddings *created* by predictive models are becoming increasingly important and useful. Often these embeddings are produced by neural networks of various flavors.

Neural networks are a major area of machine learning and graph neural networks are currently a very fast moving area of research. Many types exist and many more

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

are sure to come. Currently, they work well at many tasks but significant challenges remain. Mathematically, our understanding of these neural networks is still nascent. Computationally, they are very costly and often struggle to scale up to very large networks. Yet neural networks have shown great success in many previously intractable settings, and it's likely they'll make their mark on graphs as well.

Bibliographic remarks

Although dense vector representations have long been pursued in NLP [261, 51, 357] and network analysis (Ch. 25), word2vec [307] has had a large influence on natural language processing. DeepWalk [369] brought this idea directly to networks (graphs), leading to an explosion of interest in learned embeddings; see Goyal and Ferrara [188] and Xu [498] for recent reviews.

The connections between random walk embeddings, pioneered by Perozzi et al. [369], and matrix factorization were noted by Qiu et al. [382] following the word2vec analysis of Levy and Goldberg [269]. Kojaku et al. [249] noted the mechanism that negative sampling introduced (and effectively removed) which led to a general way (residual2vec) to remove the bias from graph embeddings.

Graph neural networks, proposed by Scarselli et al. [415], are now a very large and fast-evolving area. See Zhou et al. [508] and Chami et al. [99] for recent overviews. Grohe and Schweitzer [192] give a recent review of the graph isomorphism, one of the tools we use to understand the representational power of different graph machine learning methods.

Exercises

26.1 (**Focal network**) Use a Laplacian Eigenmap to embed the Zachary Karate Club. In two dimensions, visualize the embedding in a meaningful manner and interpret what you see in terms of the club's known community structure.

26.2 How might graph neural networks help with some of the problems discussed in Sec. 9.5?

26.3 What are AGGREGATE$(\cdot)$ and COMBINE$(\cdot)$ for the basic GCN (Eq. (26.29))?

26.4 (**Focal network**) Implement the basic GCN (Eq. (26.29)), making some choices for hyperparameters and other details as necessary. Apply to the Malawi Sociometer Network and use its embeddings as input to a logistic regression link prediction method. Validate your predictive model as per Ex. 16.5.

26.5 (**Focal network**) Same as Ex. 26.4 but try changing the propagation mechanism to use the graph Laplacian or other variation. Can you improve your predictions?

26.6 *The devil is in the details*. You may have noticed that our treatments of different graph neural networks omitted many important, practical details such as hyperparameter values, fitting methods, choice of regularization technique (if any), validation procedures, and more.

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press

(a) Find the original paper introducing one of the specific GNNs we described, such as GAT, and identify all the specific implementation details employed in the paper. Use the paper's references if necessary. Describe all the details.

(b) Having determined all this information, are you able to reproduce exactly the original study? If not, what is missing?

26.7 (**Focal network**) Gather GO terms (Sec. 9.1) as node attributes for HuRI. What opportunities are there to predict new or missing GO terms? If the attributes support it, build a GNN-based node classifier to predict GO terms. Validate using held-out attributes or other means and interpret your performance.

https://doi.org/10.1017/9781009212601.031 Published online by Cambridge University Press