University of
Cambridge

Mathematics Tripos

Part III Essay

Walking Deeper on Dynamic Graphs

December 18, 2019

Written by
Joshua Snyder

Contents

1 Introduction

The motivation of this essay and the methods it presents comes from the desire to understand networks of people and the relationships between them. From a mathematical perspective, the natural way to go about studying social networks is to let nodes of a graph G represent the people and edges their relationships. From this, we can induce some understanding of the neighbourhoods that a network consists of. This task of understanding the communities that exist in a social network is a complex one; even with a relatively small number of people it is not a task for which humans perform very well and the task rapidly becomes difficult as the size of the network increases.
Network embeddings are a useful way to characterise the complex structures that can exist in these graphs. The idea behind them is to represent each member of a network with a point in space, where the distance according to some metric (often simply Euclidean) between any two points corresponds to the similarity of the members in the network. These points in space are called vertex representations. The difficulty with this approach is defining a suitable similarity function $f(p1, p2)$ that gives a good understanding of the network structure. This ambiguity of similarity has led to a diverse range of algorithms for calculating network embeddings.

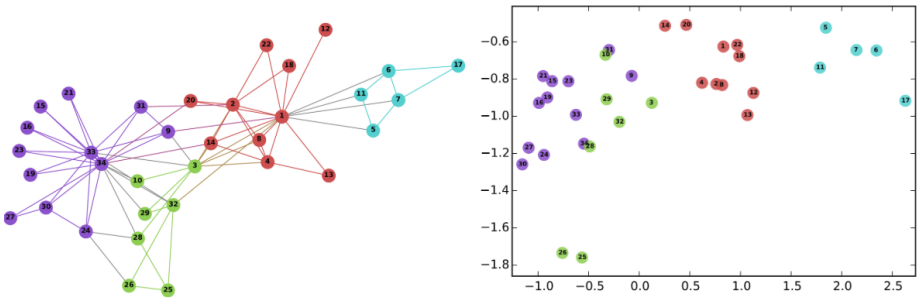

Figure 1: Two dimensional network embeddings calculated using DeepWalk on Zachary's Karate network[1]. There is a strong correspondence between community structure in the original graph and in the embedding space. Vertex colours are representing a modularity-based clustering of the input graph. In practive, network embeddings have dimensions in the tens to low-hundreds and t-SNE is often used to visualise these embeddings in 2D space.[2]

The problem of analysing social structures is made more difficult still by the fact that such networks often change over time. As new connections are formed and others die away, the network moves and over a long time period, it can change significantly. Calculating the vertex representations for a large social graph, where the size of the network is often in the millions or billions, takes a lot of time. It is therefore computationally infeasible to recalculate these representations at every time step and so a more intelligent approach to this problem is required.
This essay looks at DeepWalk and how it can be modified for dealing with dynamic graphs. DeepWalk is an algorithm that calculates vertex representations for large social graphs by using techniques adapted from natural language pro-

cessing. When proposed in 2014, the algorithm was a first of its kind and has since sparked a large amount of future research, with new algorithms improving upon it using similar methods. The core idea behind DeepWalk is to model random walks on a graph similar to sentences in a large corpus of text, this idea is explored further throughout the essay.

1.1 Summary

The section on DeepWalk explores the fundamental connection between DeepWalk and matrix factorisation as presented in the paper by Qiu et al.[3]. In this section, we demonstrate the conditions under which DeepWalk is factorising an appropriate matrix that stores some measure of similarity between nodes in the graph. In the section "Why DeepWalk sucks" we look at some of the criticism that followed the original DeepWalk paper and how these criticisms were addressed by future authors, the section is indicative of the shortcomings of DeepWalk but is by no means exhaustive.
After this, the section "Dynamic DeepWalking" focusses on applying a variant of the DeepWalk algorithm, proposed by Sajjad et al. [4], for dynamic graphs. This is especially important in the context of social graphs, which usually contain temporal information about user interactions. Learning social representations for dynamic graphs can allow for a better understanding of how communities move and change over time.
Finally, to exhibit Dynamic DeepWalking in practice, the final section of this essay applies the Unbiased Update algorithm introduced by Sajjad et al.[4] to data from users on the social network Gab to look at how user interaction on the network varies with time.

1.2 Preliminary Notation

I have endeavoured to use similar notation to the original DeepWalk paper[5] to ease cross-referencing. However given that a number of other papers are touched upon throughout the essay, there are some pieces of notation that have been changed. If reading through the essay systematically, the reader should not find difficulty following the notation.

> **Definition** ((Partially Labelled) Network Graph)**.** Let $G = (V, E)$ be an undirected graph (representing a network). V represents the members of the network, commonly referred to as the *nodes* or *vertices* and $E \subset V \times V$ represents their connections, usually referred to as *edges*.
> Some of the nodes and edges have labels and $G_L = (V, E, X, Y)$ represents the partially labelled network. The node features are represented by $X \in \mathbb{R}^{|V| \times S}$ where S is the size of the feature space for each attribute vector. The node labels are represented by $Y \in \mathbb{R}^{|V| \times |\mathcal{Y}|}$, where $\mathcal{Y}$ is the set of labels.

Some of the nodes are labelled with $y \in \mathcal{Y}$ and the end task is to predict the labels of the other nodes. This is called a relational classification problem. The aim of the DeepWalk algorithm is to learn embeddings $W \in \mathbb{R}^{|V| \times d}$ where d is a small number of latent dimensions ($d \ll |V|$) such that each of the rows in W corresponds to a d dimensional embedding of a node in V.
To avoid confusion, it is important to distinguish that the goal of DeepWalk is

not to classify the nodes, but to generate useful embeddings. These embeddings are then treated as additional features and are combined with the features X to be inputted into a classification algorithm.

2 DeepWalk

This section gives an outline of the social representation learning algorithm DeepWalk, first introduced in the seminal paper DeepWalk: Online Learning of Social Representations by B. Perozzi et. al. [5]. The method proposed in this paper not only demonstrated performance improvements from previous methodologies but also motivated an entirely different approach. At the time of the paper being written, significant advancements were being made in natural language processing (NLP) and the idea of word embeddings was becoming popular through an embedding algorithm known as Word2Vec [6, 7]. DeepWalk implements this algorithm but replaces the idea of the context of a word in a sentence with the context of a node in a random walk on a graph. This is the crucial concept of DeepWalk from which the remaining details of the algorithm naturally follow.
The original paper on DeepWalk is lacking in a mathematical foundation and so in this section we will model the algorithm mathematically. It is suggested that the reader is familiar with the concepts outlined in the paper by Perozzi et. al. prior to reading this (more) mathematical exposition.

Algorithm 1 DeepWalk

Input: network $G(V, E)$
window size k
embedding size d
walks per vertex γ
walk length l
Output: matrix of vertex representations $W \in \mathbb{R}^{|V| \times d}$

Initialization: Sample W from $\mathcal{U}^{|V| \times d}$
for $i = 0$ to γ **do**
  $\mathcal{O} = \text{Shuffle}(V)$
  for each $v \in \mathcal{O}$ **do**
    $\mathcal{W}_v = \text{RandomWalk}(G, v, l)$
    $\text{Skipgram}(W, \mathcal{W}_v, k)$
  end for
end for

2.1 Introduction to DeepWalk

To understand DeepWalk mathematically, we first need to understand what the SkipGram model is doing, since this is the model underpinning DeepWalk. SkipGram was prposed by Mikolev et al.[6] as an efficient way to learn word embeddings. The idea behind SkipGram is to learn embeddings which are good at predicting nearby words in sentences. For a sentence $w_1, ..., w_N$, the nearby words from w_i, known as context words, are defined as the set of words within distance k, $\{w_{i-k}, w_{i-k+1}, \ldots, w_{i+k}\} \setminus w_i$, where k is the size of the window. SkipGram minimises the following objective function

$$\mathcal{L} = -\sum_{c \in C(w)} \log \mathbb{P}(c|w)$$

where $\mathbb{P}(c|w)$ is modelled by a softmax function

$$\mathbb{P}(u|v) = \frac{\exp(\vec{w} \cdot \vec{c})}{\sum_{c \in \mathcal{F}} \exp(\vec{w} \cdot \vec{c})}$$

where $\mathcal{F}$ is the vocabulary and $\vec{w}$ represents the embedding of the word w and $\vec{c}$ the distributed representation of c when it serves as a context word.
*** Can I be clearer about what this distributed representation is? ***
Perrozi et al. took this model from NLP and applied it to graphs by observing that nodes in the graph can be thought of as words in an antificial language. Firstly, to find the context for each node, DeepWalk generates random walks which are analagous to sentences in a language. The context nodes are the set of nodes within a window of size k, from which input-context pairs (v, c) are constructed and added to a corpus $\mathcal{D}$, which is a multi-set. After these context nodes have been found the same optimization function as in SkipGram is used to learn embeddings which maximise the chance of predicting context nodes.
Algorithm 1 shows how DeepWalk is applied. γ represents the number of random walks that are started from each node and at the start of each pass the nodes are shuffled so that they are traversed in a random order.[1] For each node $v \in V$ a random walk $\mathcal{W}_v$ of length l is generated and used to update the network embeddings using the SkipGram algorithm applied to the random walk.
From the perspective of machine learning, SkipGram trains a neural network to do a "fake" task. It trains on this task, and then the corresponding weights learned are used as embeddings. The task is as follows:
Given an input vertex v somewhere in a random walk $\mathcal{W}$. Pick a nearby vertex at random. The task of the neural network is to predict the probability that each vertex in V will be this randomly chosen vertex. Therefore, verticies far away on the graph that correspond to unfamiliar nodes are unlikely to co-occur on the same random walk and will be assigned a low probability. Conversely, nearby and well connected vertices are likely to co-occur on a random walk with input v and thus will be assigned higher probabilities. This allows us to train a network with weights that represent the connectedness between nodes on the graph. The neural network is trained by feeding it pairs of nodes (v, c) where v represents the input node and c is a context node, which lies within distance k of the vertex v.

To formalise this, each of the nodes $v \in V$ are represented by a one-hot encoding vector $e_v \in \mathbb{R}^{|V|}$ allowing us to feed e_v into the neural network. When e_v is fed into the network, a single linear hidden layer with d neurons is used, where d is the desired dimension of the latent representations, which is then passed to a softmax classifier for output. The output of the network is a vector $o \in \mathbb{R}^{|V|}$ containing the estimated probabilities that a randomly selected nearby word is that vocabulary word.

The idea behind having a linear hidden layer, which does not use an activation function, is to use the resulting weight matrix $W \in \mathbb{R}^{|V| \times d}$ as the embedding vectors for the nodes in the graph. This is intuitive as the hidden layer acts as a bottleneck that tries to represent as much information as possible to distinguish

[1] This helps to prevent the algorithm from staying in a local minima when stochastic gradient descent is applied since on each iteration, as the vertices are shuffled, the shape of the loss surface changes.

the nodes, but is only allowed d neurons to do so. Since $d \ll |V|$ there is a low risk of overfitting.

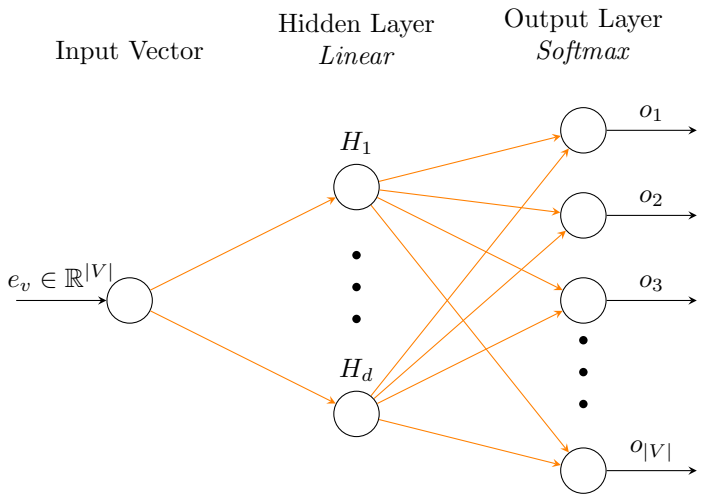

Figure 2: The structure of the neural network that is trained to obtain the weight matrix W used for the network embeddings in the SkipGram algorithm.

The algorithm used in DeepWalk varies slightly from the SkipGram algorithm discussed. Calculating the normalization factor in the Softmax layer requires a computational complexity of $O(|V|)$, which is intractable since this is often in the millions or billions. In the original DeepWalk paper this is reduced by using Hierarchical Softmax to approximate the softmax probabilities, requiring a complexity of only $O(log|V|)$. In particular, a Huffman coding is used to reduce the access time of frequent elements in the tree, as suggested by Mikolov et al. in the original Word2Vec papers.[6, 7]
In later adaptations of DeepWalk, SkipGram with Negative Sampling (SGNS) is used instead of Hierarchical Softmax. Negative Sampling updates only a sample of the output vectors per iteration. In the remainder of this essay, when referring to DeepWalk, it will be implicitly assumed that Negative Sampling is used as appose to Hierarchical Softmax. This is because SGNS has been found to be more efficient and therefore has been adopted by much of the further literature. This convention will also serve us well when we look at applying DeepWalk to dynamic graphs since here Negative Sampling is also applied.
*** Can give a summary of NS here, if so just summarise what is said in Blog Post 2 on SkipGram ***
*** Has NS been shown, or just found empirically, to be more efficient then Hierarchical Softmax? Check node2vec paper for information on this. ***

2.2 SkipGram as matrix factorisation

In this section we will exhibit a proof that SGNS is equivalent to factorising a certain matrix M into two smaller matricies W and C where the rows in W correspond to the learned embedding of each vertex in the graph. This result

was first proved by Levy and Goldberg[8] in the context of word embeddings.

2.2.1 Objective of SGNS

Given an arbitrary input-context pair (v, c) the objective is to determine if the pair comes from the random-walk corpus $\mathcal{D}$.
Let $P(\mathcal{D} = 1|v, c)$ denote the probability that (v, c) comes from a random walk on the graph and $P(\mathcal{D} = 0|v, c)$ the probability it does not. Then the distribution is modelled by a sigmoid function

$$P(D = 1|v, c) = \sigma(\vec{v} \cdot \vec{c}) = \frac{1}{1 + e^{-\vec{v} \cdot \vec{c}}}$$

where $\vec{v}$ and $\vec{c}$ are d-dimensional vectors to be learned. SGNS attempts to maximise $P(\mathcal{D} = 1|v, c)$ for observed pairs (v, c) whilst simultaneously maximising $P(\mathcal{D} = 0|v, c)$ for randomly sampled negative examples.
It assumes that randomly selecting a context c for a given node v is likely to result in a "negative sample", an unobserved pair (v, c). In the context of social networks, this assumption is reasonable since social networks are almost always sparse (The number of edges is usually $O(|V|)$). However in a different context, if the network is dense, then this may be an unreasonable assumption.
According to this assumption, the objective function of SGNS for a single observation (v, c) is:

$$\phi(v, c) = \log \sigma(\vec{v} \cdot \vec{c}) + b \cdot \mathbb{E}_{c_N \sim P_D} \log \sigma(-\vec{v} \cdot \vec{c})$$

where the minus sign comes from the fact that $1 - \sigma(x) = \sigma(-x)$, b is the number of negative samples and c_N is the sampled context node, drawn according to $P_D(c) = \frac{\#(c)}{|\mathcal{D}|}$ which is known as the unigram distribution.

*** The objective function and this bit generally needs more explanation, read Mikolev et al. to get a better grasp. It is not clear here. ***

Notation. $\#(v, c)$, $\#(v)$ and $\#(c)$ denote the number of times vertex-context pair (v, c), vertex v and context c appear in the generated random-walk corpus $\mathcal{D}$ respectively.

This objective function is trained using stochastic gradient descent with updates after each observed pair in the random-walk corpus $\mathcal{D}$. The resulting global objective becomes

$$\mathcal{L} = \sum_{v \in V} \sum_{c \in V} \phi(v, c) \tag{1}$$

2.2.2 Finding the similarity function learned by SkipGram

If we let W be the matrix with rows v_i (The matrix W is used to highlight that it is the weight matrix in the neural network presented in [Figure XX]) and C the matrix with columns c_i then SGNS can be interpreted as factorising a matrix $M = WC^T$. An entry in the matrix M_{ij} corresponds to the dot product $\vec{v_i} \cdot \vec{c_j}$. Therefore SGNS is factorising a matrix in which each row corresponds to an input node $v_i \in |V|$ and each column to a context node $c_j \in |V|$ and the

value of M_{ij} expresses the strength of association between the input-context pair (v_i, c_j) using some similarity function $s(v_i, c_j)$. What is the similarity function learned by SkipGram?

Theorem 2.1 (Levy, Goldberg (2014))**.** *SkipGram with Negative Sampling (SGNS) is implicitly factorising a matrix $M = WC^T$ with*

$$M_{ij} = \log \frac{\#(v_i, c_j)|\mathcal{D}|}{\#(v_i)\#(c_j)} - \log b$$

where $W, C \in \mathbb{R}^{|V| \times d}$, and b is the number of negative samples.

Proof. Firstly, for sufficiently large dimensionality d (so as to allow for a perfect reconstruction of M), each of the products $v_i \cdot c_j$ can be assumed to take their values independently of the others. *** WHY. EXPLAIN THIS. Why not sufficiently large value of $|\mathcal{D}|$, why does d matter here? ***

Due to this independence, the objective function $\mathcal{L}$ can be maximised with respect to each pair $v \cdot c$ individually.

The expectation term can be written explicitly as

$$\begin{aligned}\mathbb{E}_{\tilde{c} \sim P_{\mathcal{D}}}[\log \sigma(-\vec{v} \cdot \vec{\tilde{c}})] &= \sum_{\tilde{c} \in V} \frac{\#(\tilde{c})}{|\mathcal{D}|} \log \sigma(-\vec{v} \cdot \vec{\tilde{c}}) \\ &= \frac{\#(c)}{|\mathcal{D}|} \log \sigma(-\vec{v} \cdot \vec{c}) + \sum_{\tilde{c} \in V \setminus \{c\}} \frac{\#(\tilde{c})}{|\mathcal{D}|} \log \sigma(-\vec{v} \cdot \vec{\tilde{c}})\end{aligned}$$

and the objective function $\mathcal{L}$ can be expressed as

$$\mathcal{L} = \sum_{v \in V} \sum_{c \in V} \#(v, c) \log \sigma(\vec{v} \cdot \vec{c}) + \sum_{v \in V} \#(v) \left(b \cdot \mathbb{E}_{\tilde{c} \sim P_{\mathcal{D}}}[\log \sigma(-\vec{v} \cdot \vec{\tilde{c}})] \right)$$

where the second term comes from the fact that $\#(v) = \sum_{c \in V} \#(v, c)$ by definition. Combining these equations gives that the local objective for an input-context pair is

$$\phi(v, c) = \#(v, c) \log \sigma(\vec{v} \cdot \vec{c}) + b \cdot \#(v) \cdot \frac{\#(c)}{|\mathcal{D}|} \log -\sigma(-\vec{v} \cdot \vec{c})$$

To simplify the notation let $x = \vec{v} \cdot \vec{c}$ and, since we are assuming each of the $\vec{v_i} \cdot \vec{c_j}$ to take their values independently, we take the partial derivatve with respect to x and optimise the local objective:

$$\frac{\partial \phi}{\partial x} = \#(v, c) \cdot \log \sigma(-x) - b \cdot \#(v) \cdot \frac{\#(c)}{|\mathcal{D}|} \cdot \sigma(x)$$

Where the derivatives are since $\sigma'(x) = \sigma(-x)$. Setting the derivative to zero and multipying through by $\frac{-e^x}{\sigma(x)\sigma(-x)}$ gives:

$$\frac{b \cdot \#(v) \cdot \#(c)}{|\mathcal{D}|} e^{2x} + \left(\frac{b \cdot \#(v) \cdot \#(c)}{|\mathcal{D}|} - \#(v, c) \right) e^x - \#(v, c) = 0$$

This is a quadratic equation in e^x with two solutions. The first solution, $e^x = -1$ is infeasable since $x \in \mathbb{R}$ and so the appropriate solution is

$$e^x = \frac{\#(v, c) \cdot |\mathcal{D}|}{\#(v)\#(c)} \cdot \frac{1}{b}$$

Substituting $x = \vec{v} \cdot \vec{c}$ back into the equation and taking logs gives

$$M_{ij} = \vec{v} \cdot \vec{c} = log\left(\frac{\#(v,c) \cdot |\mathcal{D}|}{\#(v) \cdot \#(c)}\right) - \log b$$

□

Most interestingly, the resulting expression for the similarity function s is the pointwise mutual information (PMI) shifted by a factor of $\log b$. PMI was first introduced as a measure of association between words in 1990 by Church and Hanks [9] and became widely adopted for NLP tasks.
*** Would make sense to go into this more ***
There is an equivalent theorem for SkipGram with Softmax that was proved by Yang et al.[10] and was later used in their development of text-associated DeepWalk (TADW)[11] which encorporates text features of the verticies in a social graph.

Theorem 2.2 (Yang et al. (2015)). *SkipGram with Softmax is implicitly factorising the matrix $M = WC^T$ with*

$$M_{ij} = \log \frac{\#(v_i, v_j)}{\#(v_i)}$$

The proof of this follows very similarly to the previous theorem and will not be shown here since we will only be concerned with SGNS.

2.3 DeepWalk as matrix factorisation

We continue to look at DeepWalk in the context of matrix factorisation. Much of the proceeding analysis was exhibited in a recent paper by Qiu et al.[3] published in 2018 building upon work by Yang et al.[11] from 2015. The aim of the former paper was to lay the foundationds for, and unify, the SkipGram based network embedding methods. First we give some preliminary definitions.

Definition (Adjacency Matrix (A)). The adjacency matrix $A \in \mathbb{R}^{|V| \times |V|}$ for a graph G is the matrix with $A_{ij} = 1$ if $(v_i, v_j) \in E(G)$ and $A_{ij} = 0$ otherwise.

Definition (Degree Matrix (D)). The degree matrix $D \in \mathbb{R}^{|V| \times |V|}$ for a graph G is a diagonal matrix with $D_{ii} = d_i = degree(v_i)$ for $v_i \in V$ and $D_{ij} = 0$ otherwise.

Definition (Transition Matrix (P)). The transition matrix $P \in \mathbb{R}^{|V| \times |V|}$ for a graph G is the matrix $P = D^{-1}A$. It has entries $P_{ij} = \frac{1}{d_i}$ if $(v_i, v_j) \in E(G)$ and $P_{ij} = 0$ otherwise. It is the transition matrix corresponding to a simple random walk on the graph G.

In their paper, Qiu et al. gave a theoretical understanding of the DeepWalk algorithm by proving the following theorem:

Theorem 2.3 (DeepWalk as implicit matrix factorisation)**.** *As $l \to \infty$, DeepWalk is equivalent to factorising*

$$\log\left(\frac{2|E|}{k}\left(\sum_{r=1}^{k} P^r\right)D^{-1}\right) - \log b$$

where b is the negative sampling rate.

The theorem assumes that the graph is undirected and that it is connected so that P is irreducible. It is also assumed that the graph is non-bipartite to ensure that the random walk converges to it's invariant distribution. In the application to social network graphs, this assumption will almost certainly hold as the existence of an odd cycle is expected (Social networks usually contain a large amount of triangles, representing mutual friends or connections). It is possible however that the graph is not connected, in this case a dummy node can be introduced which contains edges to all nodes which will not affect the community structure, provided the graph sub-communities are sufficiently dense. What follows in the remainder of this section is a careful outline of the proof. *** If want to include details on bipartite asssumption, see the folder for DeepWalk for more information. ***
Firstly, $\pi_i = \frac{d_i}{2|E|}$ satisfies the detailed balance equations:

$$\pi_i P_{ij} = d_i \cdot \frac{1}{d_i} = d_j \cdot \frac{1}{d_j} = \pi_j P_{ji}$$

and $\sum_{v_i \in V} \pi_i = 1$.Thus π defines a distribution which is invariant for a simple random walk on the graph. Since the state space is finite and by assumption, P is irreducible, a random walk on G defines an irreducible Markov Chain X with transition matrix P. Therefore π is unique by the following theorem

Theorem 2.4. *Consider an irreducible Markov chain. Then*

(i) There exists an invariant distribution if and only if some state is positive recurrent.

(ii) If there is an invariant distribution π, then every state is positive recurrent, and

$$\pi_i = \frac{1}{\mu_i}$$

for $i \in S$, where μ_i is the mean recurrence time of i. In particular, π is unique.

This is a standard proof in any course on Markov Chains and so the proof is omitted here. See Page 31 of [12] for details of a proof. As well as this, since P is irreducible and thus positive recurrent since the graph is finite, under the further assumption that a simple random walk on the graph is aperiodic the distribution of the chain tends towards π as $l \to \infty$.
To proceed with the first lemma it is useful to partition the random-walk corpus as follows:

Definition. For $r = 1, \dots, k$, we define the following

$$\mathcal{D}_{\overrightarrow{r}} = \{(v, c) : (v, c) \in \mathcal{D}, v = v_j, c = v_{j+r}\}$$

$$\mathcal{D}_{\overleftarrow{r}} = \{(v, c) : (v, c) \in \mathcal{D}, v = v_{j+r}, c = v_j\}$$

*** Need to give a proper definition to clarify what I mean here *** Thus $\mathcal{D}_{\overrightarrow{r}}/\mathcal{D}_{\overleftarrow{r}}$ are sub-multisets of $\mathcal{D}$ such that the context c is r steps after or before the vertex v in random walks respectively.

As an extension of previous definitions, we let $\#(v, c)_{\overrightarrow{r}}$ and $\#(v, c)_{\overleftarrow{r}}$ denote the number of times that an input-context pair (v, c) appears in $\mathcal{D}_{\overrightarrow{r}}$ and $\mathcal{D}_{\overleftarrow{r}}$ respectively. Then the following lemma holds

Lemma 2.5. *As $l \to \infty$, we have*

$$\frac{\#(v, c)_{\overrightarrow{r}}}{|\mathcal{D}_{\overrightarrow{r}}|} \xrightarrow{p} \pi_v(P^r)_{v,c} \text{ and } \frac{\#(v, c)_{\overleftarrow{r}}}{|\mathcal{D}_{\overleftarrow{r}}|} \xrightarrow{p} \pi_v(P^r)_{v,c}$$

Proof. *** A proof of this can be found in Paper 2 but the proof is not nice, perhaps I can come up with an original proof of this using convergence of the random walk to it's invariant distribution.

Note that by reversibility the two statements should be provably equivalent and I have used detailed balance here to prove that the two limits given in Paper 2 are the same.

I will leave this to come back to for a nicer proof. ***

*** Note that Ref 3 gives info about as the length of the RWs becomes long, the singleton distribution of vertices will tend to the invariant distribution, referencing Modern Graph Theory, Bollobas *** □

From this we can show that

Lemma 2.6. *As $l \to \infty$, we have*

$$\frac{\#(v, c)}{|\mathcal{D}|} \xrightarrow{p} \frac{1}{k} \sum_{r=1}^{k} \pi_v(P^r)_{v,c}$$

Proof.

$$\frac{\#(v, c)}{|\mathcal{D}|} = \frac{\sum_{r=1}^{k}(\#(v, c)_{\overrightarrow{r}} + \#(v, c)_{\overleftarrow{r}})}{\sum_{r=1}^{k}(|\mathcal{D}_{\overrightarrow{r}}| + |\mathcal{D}_{\overleftarrow{r}}|)} = \frac{1}{2k} \sum_{r=1}^{k} \left(\frac{\#(v, c)}{|\mathcal{D}_{\overrightarrow{r}}|} + \frac{\#(v, c)}{|\mathcal{D}_{\overleftarrow{r}}|} \right)$$

$$\xrightarrow{p} \frac{1}{k} \sum_{r=1}^{k} \frac{d_v}{2|E|} (P^r)_{v,c}$$

where the second equality uses the fact that $|\mathcal{D}_{\overrightarrow{r}}| = |\mathcal{D}_{\overleftarrow{r}}| = \frac{|\mathcal{D}|}{2k}$ and the convergence comes from applying [Theorem XX], together with the continuous mapping theorem. □

This gives everything we need to prove the main theorem of this section, that allows us to understand the matrix that DeepWalk is implicitely factorising.

Theorem 2.3 (DeepWalk as implicit matrix factorisation)**.** *As $l \to \infty$, DeepWalk is equivalent to factorising*

$$\log\left(\frac{2|E|}{k}\left(\sum_{r=1}^{k} P^r\right)D^{-1}\right) - \log b$$

where b is the negative sampling rate.

Proof. Firstly, by summing the result of [Theorem XX] we get

$$\begin{aligned}\frac{\#(v)}{|\mathcal{D}|} &= \sum_{c\in V} \#(v,c) \\ &\xrightarrow{p} \sum_{c\in V} \frac{1}{k}\sum_{r=1}^{k} \pi_v (P^r)_{v,c} \\ \frac{\pi_v}{k}\sum_{r=1}^{k}\sum_{c\in V}(P^r)_{v,c} &= \frac{\pi_v}{k}\sum_{r=1}^{k} 1 = \pi_v\end{aligned}$$

since P^r is stochastic for any r.
Similarly, using the fact that π is in detailed balance with P, and thus the result of [Theorem XX] can be rewritten as

$$\frac{\#(v,c)}{|\mathcal{D}|} \xrightarrow{p} \frac{1}{k}\sum_{r=1}^{k} \pi_c (P^r)_{c,v}$$

it can be shown that $\frac{\#(c)}{|\mathcal{D}|} \xrightarrow{p} \pi_c$.
Using this and by applying the continuous mapping theorem

$$\begin{aligned}\frac{\#(v,c)\cdot|\mathcal{D}|}{\#(v)\cdot\#(c)} = \frac{\frac{\#(v,c)}{|\mathcal{D}|}}{\frac{\#(v)}{|\mathcal{D}|}\cdot\frac{\#(c)}{|\mathcal{D}|}} &\xrightarrow{p} \frac{\frac{1}{k}\sum_{r=1}^{k}\frac{d_v}{2|E|}(P^r)_{v,c}}{\frac{d_v}{2|E|}\cdot\frac{d_c}{2|E|}} \\ &= \frac{2|E|}{k}\sum_{r=1}^{k}(P^r)_{v,c}\frac{1}{d_c} = \frac{2|E|}{k}\left(\sum_{r=1}^{k}(P^r D^{-1})_{v,c}\right)\end{aligned}$$

where the last equality follows since D is diagonal.
From this, applying [Theorem XX] gives that as $l \to \infty$ DeepWalk is equivalent to factorising

$$\log\left(\frac{2|E|}{k}\left(\sum_{r=1}^{k} P^r\right)D^{-1}\right) - \log b$$

□

This proof gives a better understanding of what the DeepWalk algorithm is doing. There are many other good questions to ask about DeepWalk, for example whether the algorithm is gaurunteed to converge under stochastic gradient descent, but these questions will not explored in this essay.

*** Here can talk about how long it takes to tend towards this. Important point, how large does l need to be for this to be meaningful? ***

3 Deep Drawbacks

Whilst the original DeepWalk paper made a huge impact on the future development of social representation learning on graphs, the algorithm itself has a number of drawbacks that have been pointed out in subsequent papers. In this short section, I will detail some of the issues and criticism of the implementation of DeepWalk.

Firstly, as has briefly been mentioned, in the original paper Hierarchical Softmax was used to estimate the probabilities in the softmax layer of SkipGram. This is important, since to calculate the partition function (denominator of the softmax probability) would normally have complexity $O(|V|)$. Using Hierarchical softmax however, the nodes are asssigned to the leaves of a binary tree. This reduces the computational complexity to $O(\log |V|)$.

Negative Sampling on the other hand is developed ...

*** Talk about Node2Vec and improvements to DeepWalk etc. Talk about various shortcomings, just look at papers that came afterwards that shred DeepWalk. Point to the algorithms that improved it.

Plan: Pick 2 or 3 algorithms, state what they improved, link to the papers. ***

*** node2vec ***

3.1 Capturing Structural Similarity

Another drawback of both DeepWalk and node2vec is that both of these algorithms do well at capturing neighbourhouds in the graph, but they do not perform well at capturing structural similarities amongst nodes. Applying DeepWalk to two nodes that have a similar structural role in the graph but are in far away locations will result in embeddings that are distant from eachother. An alternative approach that performs significantly better at this task was suggested by Ribeiro et al. in their paper on struc2vec[]. Unlike DeepWalk, struc2vec does not take distance between nodes into account and instead builds a context graph whhere nodes are close together if they are structurally similar and then SkipGram is applied in a simmilar mannner to DeepWalk, but this time on the context graph instead of the original graph.
The approach of struc2vec is clearly to recognise a different kind of network structure to the one DeepWalk was created to recognise. struc2vec is more applicable to classification tasks where the labels depend heavily on the structural role of the nodes. As an example, struc2vec would be more suitable for the task of finding social group leaders within a large network graph. In such a task it is not the communities that are important to identify, but the roles people play within their communities. For this reason, it is important to identify the suitability of a relational classification problem to the application of DeepWalk.

4 Dynamic DeepWalking

In this section we transition away from the static implementation of DeepWalk towards network embeddings for Dynamic datasets. In particular, the focus of the section is to introduce an adaptation of DeepWalk to dynamic datasets as proposed in the paper published by Sajjad et al.[4] in early 2019.
As stated in the motivation for the paper, most of the recently developed representation learning methods can only be applied to static graphs while many real-world graphs are constantly changing over time. Therefore, to apply these methods the graph must be reanalysed at regular snapshots in time. This is very inneficient. Sajjad et al. proposed a modified version of the DeepWalk algorithm that is better suited to dynamic graphs and utilises the fact that the graph structure is unlikely to change significantly to develop a much more efficient way of analyzing social representation on dynamic graphs. The computational complexity of the algorithm depends on the graph density and number of edges added per epoch, however in the case of social networks these are both low and thus the algorithm is well suited to this application.
Previous algorithms to be used on dynamic graphs have not used what was learned about the graph in previous snapshots to efficiently calculate the node representations in the next snapshot of time. The problem of doing so can be split into two main tasks:

1. Generate a random walks corpus for a new snapshot of the graph based on a corpus from an older snapshot. In particular, we seek to find an efficent update algorithm that generates a set of random walks on the new snapshot from random walks on the old one, that is statistically indistinguishable from generating a set of random walks from scratch on the new snapshot of the graph.

2. Update the vertex representations incrementally so that they do not need to be generated from scratch every time step.

In what follows both of these tasks are explored. Firstly it is necessary to introduce notation appropriate for dynamic graphs.

> **Definition.** A dynamic network graph is a series of network graphs $\mathcal{G}^t = \{\mathcal{V}^t, \mathcal{E}^t\}$ with $\mathcal{V}^t = \{v_1^t, \ldots, v_{n(t)}^t\}$ and $\mathcal{E}^t = \{e_1^t, \ldots, e_{m(t)}^t\}$, with t a discrete series of times.

A dynamic network graph is a series of updates to network graphs that consist of adding/removing vertices and/or edges to the graph.

> **Definition** (Deleted and added vertex sets)**.** The sets of vertices and edges of the graph that are deleted from time t to $t+1$ are denoted by $D_{\mathcal{V}}^{t+1}$ and $D_{\mathcal{E}}^{t+1}$ respectively. The sets of added vertices and edges are denoted $A_{\mathcal{V}}^{t+1}$ and $A_{\mathcal{E}}^{t+1}$.
> The sets of all vertices contained in the added and deleted edges are denoted $\mathcal{V}(A_{\mathcal{V}}^{t+1})$ and $\mathcal{V}(D_{\mathcal{E}}^{t+1})$ respectively.

Then to analyse the effect of changes on the random walks we define the following terms:

- *Affected vertices*: All vertices that are in an added or removed edge which have not been deleted.

$$\mathcal{V}^{t+1}_{\text{affected}} = \mathcal{V}(A^{t+1}_{\mathcal{E}}) \cup \mathcal{V}(D^{t+1}_{\mathcal{E}}) \setminus D^{t+1}_{\mathcal{V}}$$

- *Affected walks*: All random walks i the corpus of random walks W^t that contain at least one affected vertex.

Importantly all unaffected walks in the corpus W^t remain valid random walks on the graph $\mathcal{G}_{t+1}$

4.1 Updating the Random Walk Corpus

As a preliminary solution to the problem of generating new random walks for the random walk corpus an algorithm called Naïve Update Algorithm 2 is introduced. This algorithm generates random walks of length l for each of the affected vertices. Once this is complete, it updates the random walks W^t by replacing the old walks with the corresponding re-generated walks and adds the random walks initiated from new vertices.

However, Sajjad et al. show by way of example that the Naïve update algo-

Algorithm 2 Naïve Update

1: **procedure** NaïveUpdate($\mathcal{G}^{t+1}, W^t, \mathcal{V}^1_{\text{affected}}, r, l$)
2: ▷ Initialise r new walks from each of the affected vertices in $\mathcal{V}^1_{\text{affected}}$
3: $W \leftarrow initWalks(\mathrm{V}^1_{\text{affected}}, r)$
4: ▷ *** WHAT DOES THIS ACTUALLY DO? ***
5: $W^{t+1} \leftarrow randomwalk(\mathrm{W}, \mathrm{l}\ \mathrm{G}^{t+1})$
6: ▷ Replace the old walks and add new ones
7: $W^{t+1} \leftarrow update(\mathrm{W}^t, W^{t+1})$
8: **return** W^{t+1}
9: **end procedure**

rithm has biased empirical transition probabilities and that the resulting corpus of random walks is not statistically indistinguishable from generating a new random walk corpus at time $t + 1$. To solve this problem a more sophisticated algorithm must be introduced.

4.1.1 Unbiased Update

*** In the procedures I need to be clear about what initWalks and randomwalk are doing ***

The motivation behind the unbiased update algorithm is the following: Consider a random walk that has arrived at a vertex v at any point along the walk, if this vertex is not in the set $\mathcal{V}^{t+1}_{\text{affected}}$ then there is no change to it's neighbours and thus the choice for the next vertex in the random walk remains the same. However if $v \in \mathcal{V}^{t+1}_{\text{affected}}$ then the neighbours of the vertex have changed which makes the random walk biased from this vertex onwards since it evolves with incorrect transition probabilities. Therefore only the affected walks need to be updated to retain a statistically satisfactory set of random walks at time $t+1$.

Algorithm 3 Unbiased Update

procedure UNBIASEDUPDATE($\mathcal{G}^{t+1}, W^t, \mathcal{V}^1_{\text{affected}}, r, l$)
  ▷ Partition the affected vertices into new and existing ones
  $\mathcal{V}_n \leftarrow newVertices(\mathrm{V}^1_{\text{affected}})$
  $V_e \leftarrow existingVertices(\mathrm{V}^1_{\text{affected}})$
  ▷ Filter the existing random walk corpus to get only the walks that contain affected vertices (in $\mathcal{V}_e$)
  $W_{\text{affected}} \leftarrow filter(\mathrm{W}^t, \mathcal{V}_e)$
  ▷ Trim the affected walks to the first affected vertex
  $W_e \leftarrow trim(\mathrm{W}_{\text{affected}}, \mathcal{V}_e)$
  $\triangleright \leftarrow Initialise walks from each of the new vertices in \mathrm{V}_n$
  $W_n \leftarrow initWalks(\mathrm{V}_n, r)$
  ▷ Take the union of the new walks to create the updated corpus
  $W \leftarrow \mathrm{W}_e \cup W_n$
  $W^{t+1} \leftarrow randomwalk(\mathrm{W}, \mathrm{l}, \mathrm{G}^{t+1})$
  $W^{t+1} \leftarrow update(\mathrm{W}^t, W^{t+1})$
  return W^{t+1}
end procedure

The Unbiased Update algorithm updates only those walks that contain vertices in $\mathcal{V}^{t+1}_{\text{affected}}$ and does so by re-sampling the affected walks from the first affected vertex in the walk. Searching for these affected vertices is computationally expensive so another algorithm, Fast Update, is also suggested that generates the affected walks from the begginning instead of trimming them. Whilst computationally more efficient, this generates a biased random walk corpus. This is because by applying the Fast Update routine, we have effectively generated random walks on the graph $\mathcal{G}^{t+1}$ and then regenerated all walks that hit an affected vertex, results in a corpus that is biased towards walks that do not visit the affected vertices.
In practice, when implementing the Unbiased Update algorithm the random seed that is used to generate each of the walks can be stored and then used to regenerate the random walks up to their first affected vertex by reusing the seed. Therefore the Unbiased Update algorithm can be implemented with the same complexity as the Fast Update algorithm.
*** Can I give a proof that the Unbiased Update algorithm does indeed generate an unbiased corpus? They have given an explanation but it is not very mathematical. ***
*** Can include the complexity analysis if need be but it is not particularly informative ***

4.2 Updating the Vertex Representations Efficiently

Once the updated random walk corpus W^{t+1} has been calculated, the vertex representations must be updated. In the DeepWalk algorithm, SkipGram is used to optimize the objective function $\mathcal{L}$ using SGNS. The algorithm starts by initialising the vertex representations randomly and uses stochastic gradient descent to optimize the objective function over the input-context pairs in $\mathcal{D}_{t+1}$. However, initialising the representations randomly in the dynamic case is inef-

ficient since between each epoch the optimal vertex representations are similar. It is therefore much more efficient to initialise the vertex representations at time $t+1$ as the corresponding representations at time t with any new vertices being randomly initiated, before performing stochastic gradient descent to minimize the SGNS objective function over the new random walks corpus W^{t+1}. The idea behind this is similar to using pretrained weights in a transfer learning problem instead of initialising weights randomly and results in stochastic gradient descent converging at a much faster rate.
A potential downside of this is that if the objective function is not convex and stochastic gradient descent gets stuck in a local minima whilst training then it is more likely to stay there.

5 Discussion of an application

The aim of this section is to bring the theory discussed on Dynamic DeepWalking into practice.

6 Conclusion

See the esssay descriptors. It is required to have a conclusion and suggested to reccomend further areas of research!

References

[1] W. W. Zachary, "An information flow model for conflict and fission in small groups," *Journal of Anthropological Research*, vol. 33, no. 4, pp. 452–473, 1977.

[2] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," 2008.

[3] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang, "Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec," *CoRR*, vol. abs/1710.02971, 2017.

[4] H. P. Sajjad, A. Docherty, and Y. Tyshetskiy, "Efficient representation learning using random walks for dynamic graphs," *CoRR*, vol. abs/1901.01346, 2019.

[5] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 03 2014.

[6] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[7] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013.

[8] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," *Advances in Neural Information Processing Systems*, vol. 3, pp. 2177–2185, 01 2014.

[9] K. W. Church and P. Hanks, "Word association norms, mutual information, and lexicography," *Comput. Linguist.*, vol. 16, pp. 22–29, Mar. 1990.

[10] C. Yang and Z. Liu, "Comprehend deepwalk as matrix factorization," *CoRR*, vol. abs/1501.00358, 2015.

[11] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information," in *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pp. 2111–2117, AAAI Press, 2015.

[12] D. Chua, "Part ib markov chains."