University of
Cambridge

Mathematics Tripos

Part III Essay

Walking Deeper on Dynamic Graphs

May 2, 2020

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 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 network is a complex one; even with a relatively small number of people it is not a task for which humans perform 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 $f(p_1, p_2)$ 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.

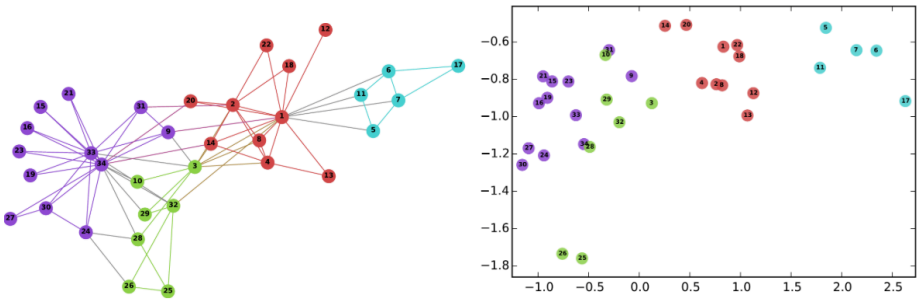

Figure 1: Two-dimensional network embeddings calculated using DeepWalk (2014, [**?**]) on Zachary's Karate network (1977, [**?**]). There is a strong correspondence between community structure in the original graph and in the embedding space. Vertex colours are representing a modularity-based clustering of the input graph. In practice, network embeddings have dimensions in the tens to low-hundreds and t-SNE (2008, [**?**]) is often used to visualise these embeddings in 2D space.

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 evolves and over a long time, 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 (2014, [**?**]) and how it can be modified for dealing with dynamic graphs. DeepWalk is an algorithm that calculates vertex

representations for large graphs by using techniques adapted from natural language processing. 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 is helpful since random walks store information about the similarity of nodes in a graph, but in a linear data type which makes it much easier to handle directly then the complex graph structures that are usually observed. There are many applications of DeepWalk to problems in which community structure is important for classification. Figure 1 presents the results of applying DeepWalk to a small network of people and t-SNE helps to visualise the groups that DeepWalk separates in the embedding space.

In practice, DeepWalk can be applied to any network; in recommender systems, previous interactions between users and a website are used to form a graph of interactions from which similar content can be determined to make better suggestions to a user (2020, [**?**]). Another example is for recommending relevant scientific research to a given paper (2016, [**?**]) or even suggesting possible collaborators for scientific research projects (2019, [**?**]).

1.1 Essay Outline

Section 2 explores the fundamental connection between DeepWalk and matrix factorisation as presented in the paper by Qiu et al. (2018, [**?**]). 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. This helps to give a mathematical understanding of DeepWalk that is not seen in the original paper.

Section 3 focuses on applying a variant of the DeepWalk algorithm, proposed by Sajjad et al. (2019, [**?**]) 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. In this section, the task of efficiently generating an unbiased corpus of random walks that is representative of the dynamic graph is discussed.

In Section 4 we look at some of the criticism that followed the original DeepWalk paper and how these criticisms were addressed in future research. Along with other issues, the major flaw of orthogonal invariance of network embeddings is highlighted along with the problems that this poses for using DeepWalk with large dynamic graphs and suggestions for how it can be resolved. As well as this, subsequent algorithms node2vec and struc2vec are briefly discussed to highlight some of the network structure that is not captured well by DeepWalk. Finally, we explore the lack of rigour in the selection of embedding dimensionality and consider the damage this brings to the effectiveness of generated embeddings in the dynamic setting.

In the conclusion the essay is wrapped up by suggesting areas that could be explored for further research.

1.2 Preliminaries

I have endeavoured to use similar notation to the original DeepWalk paper to ease cross-referencing. However given that several other papers are touched upon throughout the essay, there are some pieces of notation that have been changed.

> **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 DeepWalk algorithm aims 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 Understanding 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. (2014, [**?**]). 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 (2013, [**?**, **?**]). 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 proposed by Mikolev et al. (2013, [**?**]) 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 $w_1, ..., w_l$ the nearby words from w_i, known as context words, are defined as the set of words within distance k, $C(w_i) = \{w_{i-k}, w_{i-k+1}, \dots, w_{i+k}\} \setminus w_i$, where k is the size of the window. SkipGram minimises the following objective function

$$\mathcal{L} = -\sum_{r=1}^{l} \sum_{c \in C(w)} \log \mathbb{P}(c|w)$$

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

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

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.

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 artificial language. Firstly, to find the context for each node, DeepWalk generates random walks which are analogous 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 a different perspective, 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, vertices 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 the nodes, but is only allowed d neurons to do so. Since $d \ll |V|$ there is a low risk of overfitting.

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

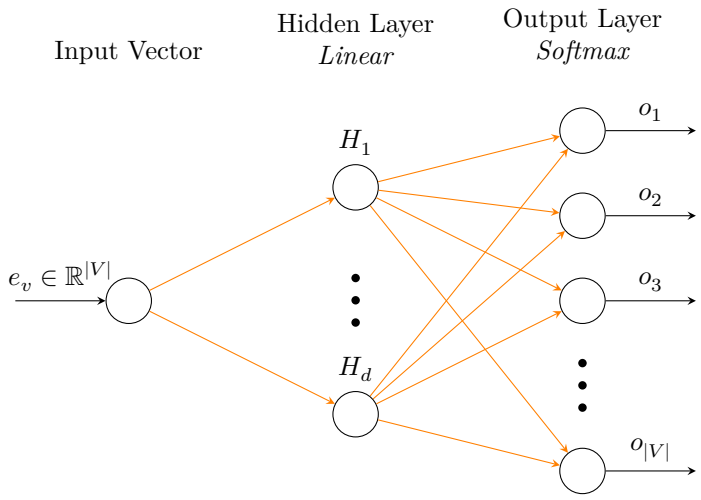

Figure 2: The structure of the neural network that is trained to obtain the weight matrix W used for the network embeddings in the SkipGram algorithm.

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, 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 (2013, [**?**, **?**]).

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 empirically to be more efficient and therefore has been adopted by much of the further literature (2013, [**?**]). This convention will also serve us well when we look at applying DeepWalk to dynamic graphs since here Negative Sampling is also applied.

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 matrices 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 (2014, [**?**]) in the context of word embeddings.

2.2.1 Objective of SGNS

Given an arbitrary input-context pair (v, c) the objective of SGNS is to determine if the pair comes from the random walk corpus $\mathcal{D}$, which is a multiset containing vertex-context pairs generated from the random walks.
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 $O(|V|)$ and not $O(|V|^2)$). 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:

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

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

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

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

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 2) and C the matrix with rows 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 $f(v_i, c_j)$. 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[1]), each of the products $v_i \cdot c_j$ can be assumed to take their values independently of the others.
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

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

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

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

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

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

[1] If d is not sufficiently large then the product of W and C^T might not have enough free parameters to encode the similarity function at each vertex-context pair.

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 by Church and Hanks (1990, [**?**]) and became widely adopted for NLP tasks.
There is an equivalent theorem for SkipGram with Softmax that was proved by Yang et al. (2015, [**?**]) and was later used in their development of text-associated DeepWalk (2015, [**?**]) 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. (2018, [**?**]) building upon work by Yang et al. (2015, [**?**]). The former paper aimed to lay the foundations for, and unify, the SkipGram based network embedding methods which had arisen in the years since DeepWalk in 2014. 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*

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

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

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 its 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 number 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.

Firstly we note that $\pi_i = \frac{d_i}{2|E|}$ 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[1]. 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:

[1] See Page 31 of [**?**] for details of a proof.

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

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

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

where the subscripts denote the positioning of a vertex in a random walk.

As an extension of previous definitions, we let $\#(v, c)_{\vec{r}}$ and $\#(v, c)_{\bar{r}}$ denote the number of times that an input-context pair (v, c) appears in $\mathcal{D}_{\vec{r}}$ and $\mathcal{D}_{\bar{r}}$ respectively. We turn to the proof of a useful proposition before using this notation for the next Lemma.

Proposition 2.5. *Let $Y_1, Y_2, \dots$ be a sequence of random variables with finite expectation $\mathbb{E}[Y_j] \to \mu$ as $j \to \infty$ and bounded variance $Var(Y_j) < K$. Suppose also that the covariances $Cov(Y_i, Y_j) \to 0$ as $|i - j| \to \infty$. Then*

$$\frac{1}{n} \sum_{i=1}^{n} Y_i \xrightarrow{p} \mu \text{ as } n \to \infty$$

Proof. Let $S_n = \sum_{i=1}^{n} Y_i$. We show that $Var\left(\frac{S_n}{n}\right) \to 0$ and apply Chebyshev's inequality as in the proof of the Weak Law of Large Numbers.

$$Var\left(\frac{S_n}{n}\right) = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} Cov(Y_i, Y_j) \leq \frac{K}{n} + \frac{1}{n^2} \sum_{i \neq j} Cov(Y_i, Y_j)$$

The first term clearly tends to 0 so we focus on the second. Suppose that $|i - j| > N(\delta)$ implies that $Cov(Y_i, Y_j) < \delta$. For $n > N = N(\delta)$ we have

$$\begin{aligned} \frac{1}{n^2} \sum_{i \neq j} Cov(Y_i, Y_j) &= \frac{1}{n^2} \sum_{1 \leq |i-j| \leq N}^{n} Cov(Y_i, Y_j) + \frac{1}{n^2} \sum_{|i-j| > N}^{n} Cov(Y_i, Y_j) \\ &\leq \frac{1}{n^2} \left(nNK + n^2\delta\right) = \frac{NK}{n} + \delta \end{aligned}$$

where the inequality comes from applying Cauchy-Schwarz together with the bounded variance assumption for the first term and using $n > N$ for the second. For any $\epsilon > 0$, we can pick δ sufficiently small and N_1 such that for all $n > N_1 \gg N(\delta)$ we have $Var\left(\frac{S_n}{n}\right) < \epsilon$ and so $Var\left(\frac{S_n}{n}\right) \to 0$.
Finally, applying Chebyshev's inequality with the triangle inequality yields

$$\begin{aligned} \mathbb{P}\left(\left|\frac{S_n}{n} - \mu\right| > \epsilon\right) &\leq \mathbb{P}\left(\left|\frac{S_n}{n} - \mathbb{E}\left[\frac{S_n}{n}\right]\right| > \frac{\epsilon}{2}\right) + \mathbb{P}\left(\left|\mathbb{E}\left[\frac{S_n}{n}\right] - \mu\right| > \frac{\epsilon}{2}\right) \\ &\leq \frac{4}{\epsilon^2} Var\left(\frac{S_n}{n}\right) + \mathbb{P}\left(\left|\mathbb{E}\left[\frac{S_n}{n}\right] - \mu\right| > \frac{\epsilon}{2}\right) \to 0 \text{ as } n \to \infty \end{aligned}$$

which gives the desired result. □

We can use Proposition 2.5 to understand the limiting distribution of vertex-context pairs in $\mathcal{D}$.

Lemma 2.6 (Limiting Distribution of vertex-context pairs in $\mathcal{D}$)**.** *As $l \to \infty$, we have*

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

Proof. We proceed with a single markov chain X_n which represents a single random walk with initial distribution λ. We define

$$Y_j = \mathbb{1}(X_j = v, X_{j+r} = c) \text{ and } Q = P^r$$

.
Then we have that

$$\mathbb{E}[Y_j] = \mathbb{P}(X_j = v, X_{j+r} = c) = (P^j\lambda)_v Q_{vc} \to \pi_v Q_{vc} \text{ as } j \to \infty$$

by convergence of the markov chain to it's invariant distribution[1]. For $i+r < j$:

$$\mathbb{E}[Y_i Y_j] = \mathbb{P}(X_i = v, X_{i+r} = c, X_j = v, X_{j+r} = c) = (P^i\lambda)_v Q_{vc} (P^{j-i-r})_{cv} Q_{vc}$$

which gives that

$$Cov(Y_i, Y_j) = (P^i\lambda)_v Q_{vc}^2 \left[\underbrace{(P^{j-i-r})_{cv}}_{\to \pi_v} - \underbrace{(P^j\lambda)_v}_{\to \pi_v} \right] \to 0 \text{ as } j - i \to \infty$$

Since the Y_j are indicator functions their variances are bounded above by 1. Applying Proposition 2.5 gives

$$\frac{\#(v,c)_{\vec{r}}}{|\mathcal{D}_{\vec{r}}|} = \frac{1}{l-r} \sum_{j=0}^{l-r-1} Y_j \xrightarrow{p} \pi_v Q_{vc}$$

The second limit can be obtained from the first by applying the detailed balance equations:

$$\begin{aligned} \frac{\#(v,c)_{\bar{r}}}{|\mathcal{D}_{\bar{r}}|} &= \frac{\#(c,v)_{\vec{r}}}{|\mathcal{D}_{\bar{r}}|} \\ &= \frac{\#(c,v)_{\vec{r}}}{|\mathcal{D}_{\vec{r}}|} \xrightarrow{p} \pi_c(P^r)_{c,v} = \pi_v(P^r)_{v,c} \end{aligned}$$

where the second equality follows since $|\mathcal{D}_{\bar{r}}| = |\mathcal{D}_{\vec{r}}|$.
Since this holds for a random walk initialised at any starting distribution, it holds for any number of random walks initialized from the vertices in any order since the corresponding value of $\frac{\#(v,c)_{\bar{r}}}{|\mathcal{D}_{\bar{r}}|}$ is just this result averaged over the number of random walks. □

From this lemma, we can go on to show that

[1] This follows since the chain is irreducible, positive recurrent and aperiodic by assumption. Convergence of a markov chain to it's invariant distribution is standard, but a proof may be found on Page 34 of [**?**].

Lemma 2.7. *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.

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

where the second equality uses the fact that $|\mathcal{D}_{\vec{r}}| = |\mathcal{D}_{\overleftarrow{r}}| = \frac{|\mathcal{D}|}{2k}$ and the convergence comes from applying Lemma 2.6, 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*

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

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

Proof. Firstly, by summing the result of Lemma 2.7 we get

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

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 Lemma 2.7 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

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

where the last equality follows since D is diagonal.
From this, applying Theorem 2.1 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 as the lengths of the random walks becomes increasingly large. There are many other good questions to ask about DeepWalk, including matters of convergence with stochastic gradient descent, but these questions will not explored in this essay.

3 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. (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 inefficient. 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; 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 $\mathcal{W}^{t+1}$ for a new snapshot of the graph based on a corpus from an older snapshot $\mathcal{W}^t$. In particular, we seek to find an efficient 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 $\mathcal{V}^t = \{v_1^t, ..., v_{n(t)}^t\}$ and $\mathcal{E}^t = \{e_1^t, ..., e_{m(t)}^t\}$, 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 in the corpus of random walks $\mathcal{W}^t$ that contain at least one affected vertex.

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

3.1 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 $\mathcal{W}^t$ by replacing the old walks with the corresponding re-generated walks and adds the random walks initiated from new vertices.

Algorithm 2 Naïve Update

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

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

3.1.1 Unbiased Update

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 $t + 1$.

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

Algorithm 3 Unbiased Update

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

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 beginning 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, resulting 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.

3.2 Updating the Vertex Representations Efficiently

Once the updated random walk corpus $\mathcal{W}^{t+1}$ 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 inefficient since between each epoch the optimal vertex representations can be similar, provided only a small portion of the graph has changed. 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 $\mathcal{W}^{t+1}$. The idea behind this is similar to using pre-trained weights in a transfer learning problem in-

stead 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 minimum whilst training then it is more likely to stay there. To reduce the likelihood of getting stuck in a local minimum, at each epoch a small random subset of embedding coordinates could be chosen to be randomly initialised.

4 Drawbacks and Opportunities of DeepWalk

Whilst the original DeepWalk paper made a huge impact on the future development of social representation learning on graphs, the algorithm itself has many drawbacks that have been pointed out in subsequent papers. In this section, I will detail some of the issues and criticism of the implementation of DeepWalk in both static and dynamic applications and some algorithms that were subsequently proposed to tackle these issues.

4.1 Exploring Local and Global Properties of Graphs

There are two extreme strategies to search a graph starting from a vertex. The first is Breadth-First Search (BFS) which samples the immediate neighbours of the node and helps to understand local structure within the nodes close proximity. The other is Depth-First Search (DFS) which samples nodes by going as far away as possible from the starting vertex. BFS focuses on understanding local structure and DFS on global structure. By using unbiased random walks to try and get a fair coverage of the graph, DeepWalk does not allow for orientating the focus to a depth-first or breadth-first approach which is useful for many applications. A different algorithm, node2vec, was proposed by Grover and Leskovec (2016, [**?**]) and aims to combine the properties of BFS and DFS through introducing biased random walks. New parameters p, q are introduced which control the transition probabilities of the random walk on the graph. The unnormalised transition probability from y to z given that the walk transitioned to y from node x is:

$$p_x(y,z) = \begin{cases} \frac{1}{p} & \text{if } z = x \\ 1 & \text{if } d(x,z) = 1 \\ \frac{1}{q} & \text{if } d(x,z) = 2 \end{cases}$$

Varying p and q affects the random walks likelihood to travel far away versus exploring locally. Therefore by controlling these parameters a balance between BFS and DFS can be achieved. By biasing the random walks in a creative way and learning good parameters for p, q appropriately for each graph (e.g using a grid search) node2vec shows superior performance for learning vertex representations used in several upstream classification tasks, including on the BlogCatalog and Wikipedia datasets used in the original DeepWalk paper.

Since DeepWalk uses unbiased random walks, it corresponds to using node2vec with $p = q = 1$. Whilst node2vec brings more flexibility, it still suffers from the orthogonal invariance problem discussed in Section 4.3. As well as this, it is not always clear if the values of p and q give a meaningful interpretation of the importance of BFS and DFS in the exploration of a given network or just increase the dimension of the hyperparameter space.

4.2 Capturing Structural Similarity

Another drawback of DeepWalk is that it does well at capturing neighbourhoods in the graph, but does not perform well at capturing structural similarities amongst nodes. That is to say that DeepWalk performs well at identifying node similarity via homophily but not by structural equivalence. Applying

DeepWalk to two nodes that have a similar structural role in the graph but are in faraway locations will result in embeddings that are distant from each other. An alternative approach that performs significantly better than both DeepWalk and node2vec at this task was suggested by Ribeiro et al. in their paper on struc2vec (2017, [**?**]). Unlike DeepWalk, struc2vec does not take the distance between nodes into account and instead builds a context graph where nodes are close together if they are structurally similar and then SkipGram is applied similarly 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 recognise, 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.

4.3 Orthogonal Invariance

Perhaps the largest criticism that can be given to DeepWalk is inherited from it's constructing part: SkipGram. As we have seen in Theorem 2.1, SkipGram is implicitly factorising a matrix $M = WC^T$ and the embeddings W are used. We saw that by construction DeepWalk performs a similar factorisation and so do many algorithms that share similarities to DeepWalk. The learned embeddings are then combined with the features attached to each vertex and are often fed into a neural network for an upstream classification task. However, the factorisation of M is not invariant under orthogonal transformation: Let $Q \in \mathbb{R}^{d\times d}$ be an orthogonal matrix, then $M = WC^T = WQQ^TC = (WQ)(CQ)^T$. Therefore embeddings can undergo arbitrary orthogonal transformations without changing the resulting matrix M.

In static applications, this is an issue because the loss function is many-to-one with respect to the embeddings. The result of this is that when generating two sets of embeddings using DeepWalk, one set often appears very different from the other, but is in fact similar and has undergone some composition of rotation and reflection. After generating some embeddings $W_1, \dots, W_n$, the best embedding W_i is selected based on its performance on the training data in the upstream classification task. However W_i is unlikely to have captured the graph structure better than any of the other embeddings, it is instead likely that this process overfits the embeddings to a particular orthogonal transformation that works well for the training set in the upstream classification task.

Orthogonal invariance poses a particular problem in the case of dynamic network embeddings since at each epoch embeddings are altered to reflect updates to the graph. However, when the embeddings are updated, they can also undergo any orthogonal transformation from the previous set of embeddings and maintain the same loss. This is problematic as it makes embeddings very difficult to compare and to use for upstream classification tasks. This problem is especially prevalent when network embeddings are initialized randomly in the Unbiased Update algorithm. Whilst initializing embeddings at their previous values may help with this, over time or with the introduction of new vertices to

the graph structure, orthogonal invariance remains a prevalent issue.
A possible solution to this involves careful thought of the way generated embeddings are used in upstream tasks. If the loss function for the upstream task is dependent on the network embeddings in an orthogonally invariant way then this invariance is not an issue. However this reduces the ability to make comparisons between time epochs, where the potential of orthogonal transformation must be carefully considered.
To give a concrete example of this, if $v^s, v^t \in \mathbb{R}^d$ [Think about notation here!] represent the embeddings of a vertex at times s and t respectively, then if the upstream loss is a function of $\|v^s\| - \|v^t\|$ then orthogonal invariance will not cause issues since $\|Qv\| = \|v\|$ for $Q \in \mathbb{R}^{d\times d}$ orthogonal. Though a much more common loss might be a function of $\|v^s - v^t\|$. Such a loss function would not be appropriate, since $\|Qv^s - v^t\| \neq \|v^s - v^t\|$ for an arbitrary orthogonal matrix Q.

A solution to this is to add a regularisation term to the loss function for SkipGram which penalises future embeddings based on previous ones through a metric that changes with respect to orthogonal transformation. As an example, if W^t and W^{t+1} represent embeddings generated at times t and $t+1$ respectively, then the Euclidean distance $\|W^{t+1} - W^t\|$ could be penalised, where the norm is restricted to vertices in $\mathcal{V}_t \cap \mathcal{V}_{t+1}$. This penalisation would reduce the incentive for orthogonal transformations that change the embeddings across time periods, but its effectiveness would be diminished as the time gap between two embeddings is increased.

4.4 Embedding Space Dimensionality

In recent years there has been a large body of research devoted to algorithms, such as DeepWalk, for generating meaningful embeddings, yet the selection of an appropriate dimensionality for embeddings takes a more arbitrary approach. Based on empirical evidence, a dimensionality in the low hundreds is often suggested (2008, [**?**]). However basing dimensionality on a broad guess is not appropriate, since the embeddings generated by DeepWalk are sensitive to dimensionality.
Since increasing the dimensionality of network embeddings increases the number of model parameters, it also increases the learning time. This means that in practice embedding dimensionality is sometimes reduced based on computational savings rather than hyperparameter selection, which can severely reduce the embeddings ability to distinguish between graphs effectively, especially in the dynamic case where graph updates are small and incremental. Conversely, having too large a dimension increases the likelihood of overfitting, thus increasing the variance of the embeddings for small updates to the graph. This can significantly worsen the ability of network embeddings to give a faithful representation of the graph structure and reduces the performance of upstream classification tasks.

For applying DeepWalk to large dynamic social network graphs this presents a significant stumbling block since the dimensionality of the embeddings must be large enough to capture incremental differences to the graph resulting from changes to a tiny percentage of the graph's nodes and edges, yet the embedding dimensionality must also not be too large to keep its variance low enough for embeddings to give a meaningful representation. Consequently, very careful

consideration of embedding dimensionality is suggested when applying DeepWalk to large social network graphs.

5 Conclusion

We end by commenting briefly on our journey of understanding more about DeepWalk and suggesting future areas of research.

In Section 2 several theoretical results are given which show that DeepWalk amounts to an implicit matrix factorisation as the length of the random walks sampled tends to infinity. As well as explaining the related results of papers by Levy and Goldberg (2014, [**?**]) and Qui et al. (2018, [**?**]), a full proof of Lemma 2.6 was given without assuming that the random walks are initiated in the invariant distribution which is not the case in the original DeepWalk paper. Whilst these results shed light on the core ideas used by DeepWalk to generate embeddings, no thought is given here to the applicability of these results for smaller random walk lengths, which were used in the original DeepWalk paper ($l = 40$) as well as being widely adopted in future literature. Therefore, by consideration of the mixing times of random walks on large sparse graphs, an analysis of how long random walks need to be before these results are meaningful would be a helpful area for further research, especially if it was coupled with a trade-off between random walk lengths and the number of random walks (i.e the performance trade-off of l vs γ). An understanding of how we can reduce l and increase γ to maintain performance would be especially useful in applying DeepWalk with parallel computing since generating the random walk corpus is trivially parallel with respect to γ.

[Include something about Section 3]

In Section 4 we looked at some of the problems that DeepWalk faces and how these may be solved. The most notable issue was the orthogonal invariance of the loss function with respect to the generated embeddings, resulting in arbitrary orthogonal transformations of embeddings going unnoticed between time epochs for dynamic applications. This also makes embeddings generated by DeepWalk difficult to use in static applications since there is no clear metric of the distance between graph embeddings that deals appropriately with orthogonal invariance, which would be helpful to understand graph similarity. This is explored further by Yin et al. (2018, [**?**]) who introduce the Pairwise Inner Product (PIP) loss.

We also looked at alternatives to DeepWalk, such as node2vec, which improve on DeepWalk by introducing more flexibility. Despite improved performance on many tasks, it is not clear whether these results lend themselves to a better interpretation of the graph; by increasing the hyperparameter space of DeepWalk, node2vec allows for a larger search space of network embeddings and so it will naturally yield better results. It would be useful to look at how meaningful these performance gains are. Also of interest would be a study of applying node2vec in the dynamic setting and how an appropriate tuning of p_t and q_t could lead to a balance between breadth and depth-first search over time.

Lastly, it was noted that DeepWalk, as well as other proposed algorithms, are suitable for generating network embeddings only for suitable upstream tasks. Understanding how a variety of these algorithms could be combined in an ensemble would be useful for applications when the upstream classification task is not yet known. For example, if a social network does not want to store relational information of groups due to privacy concerns or regulation, it may instead store network embeddings from which it can apply upstream classification tasks in the future. For such applications, it would be helpful to have an

algorithm that tries to store as much information as possible for a more general class of relational classification problems.