Temporal Graph Embedding

Justin Mücke
justin.muecke@uni-ulm.com
University Ulm
Ulm, Germany

ABSTRACT

This paper discusses different methods for temporal graph embedding. It first gives a basic description on graphs itself and then shows basic methods for embedding static graphs, like word2vec, random walks, DeepWalk, node2vec, structural deep network embedding and graph2vec. Then it is discussed how these static methods can be modified to apply them onto temporal graphs. For this purpose, there are shown two different methods: tNodeEmbed, which is based on node2vec, and tdGraphEmbed, which is based on random walks. Furthermore, there are a few examples on how to use such embedded graphs.

KEYWORDS

Temporal Graph, Embedding

ACM Reference Format:
Justin Mücke. 2021. Temporal Graph Embedding. In *University Ulm: Seminar Data Science*. ACM, New York, NY, USA, 4 pages.

1 INTRODUCTION

With the surge of social networks like facebook, twitter and instagram, huge and quickly changing social networks became a part of the everyday life of most people. The goal of this paper is to show, how those networks can be analyzed and which predictions can be made from the data that is available. This analysis can be done using so-called *temporal graph embedding*. Here, a graph gets transformed into either a vector field, with one vector describing one node out of the graph, or the graph at a time t in its evolution. With these vectors, it is easier to train neural networks, as graphs are too complex of a structure for them. These can then be, in turn used to perform tasks like: node prediction, where the focus lies on prediction if an edge will form between two nodes and node classification, with which nodes that share attributes can be labeled. As there are different methods on how to achieve such an embedding of graphs into sets of vectors, this paper focuses on giving a quick overview over the basic approaches with non-temporal graphs, to lay down a groundwork. These methods can be used in a modified way to also work with temporal graphs. How two methods for temporal graphs work is shown in section 3 of this paper. To finish of, in section 4, different applications for embedded temporal graphs are shown.

Seminar Data Science, 20/21, Ulm, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

2 PRELIMINARY

To start this paper out, let's first create a level playing field for all readers by laying out what exactly a Graph is, how it can change over time and what is meant when we talk about embedding.

2.1 Graphs

A graph is a mathematical construct which is used in a variety of tasks. It is often used to model relationships between entities, thus making it possible to operate on such structures to, for example, analyze them.

Mathematically it is consisting of two sets $G = (V, E)$, where

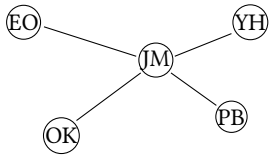

Figure 1: Part of the follower-relationship-graph of personal Instagram account.

V consists of all the vertices of the Graph, and E of all the edges, represented through a tuple of two vertices. Our graph above would thus look like following:
$G_i = (\{E, O, Y, P, J\}, \{(J, E), (J, O), (J, P), (J, Y)\})$.

2.2 Temporal Graphs

A temporal graph, on the other hand, is a graph that changes its structure over time. This happens when either one of our sets (V, E) changes. Our graph G is then represented through $G = \{g_1, g_2, \ldots, g_t\}$ where g_i is the static graph after time $i * \Delta t$. [1]

2.3 Embedding

Embedding is a process in which the graph $G = (V_g, E_g)$ is transformed into a set of vectors V. These capture the graph topology, vertex-to-vertex relationship as well as other relevant information. This way, it is more accessible for analyzing the graph and comparing it to others. Overall we can divide graph embedding techniques into two categories; vertex embedding and graph embedding.
When using vertex embedding, there has to be one vector v for each node in G, so that $|V| = |V_g|$. This is used to make prediction on a node level.
When using graph embedding, there is one vector representing the whole graph. This method is most useful to analyze the data on a graph level e.g. comparing two graph structures with each other.

In order to achieve this transformation there are several methods available.

2.3.1 Word2Vec. The first method is word2vec which is the foundation for many other methods. It takes a text $T = (w_1, w_2, \ldots, w_n)$ as input and returns a set of vectors$V$, where v_i is a distribution describing how likely it is for each word to be a direct neighbor to it. The neighbors of a word w_c are defined as $\{w_i | i \in (c - \epsilon, c + \epsilon)\}$ where ϵ describes a predefined window size as seen in figure 2. With this, the whole text is represented solely through vectors, where similar words have similar vectors. One way to achieve this, is to train a neural network to create the distributions. This neural network consists of one input-, one hidden- and one output layer. Each word in our text is assigned an ID through a one-hot coded vector, so that each word w_i can be represented like $(0_1, 0_2, \ldots, 0_{i-1}, 1_i, 0_{i+1}, \ldots 0_n)$. It then computes how likely it is for each word to be its neighbor and transforms it into a distribution using a softmax-function. [2] [3]

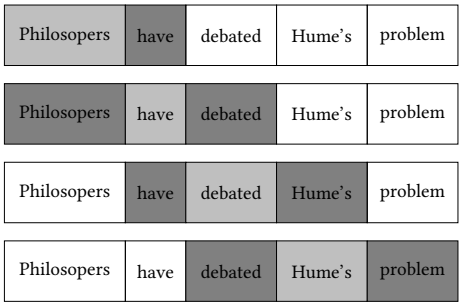

Figure 2: Light-gray word, with dark-gray neighbor and $\epsilon = 1$.

2.3.2 DeepWalk. This method is a continuation of the word2vec approach, but now the input is not a text, but a graph $G = (V, E)$ It consists of three steps. The first being the sampling, where random walks are performed from each node. Hereby, a random walk is a path in the graph from a starting point v_i of defined length λ. The following nodes are generated by:

$$P(v_i = x | v_{i-1} = y) = \begin{cases} \frac{\pi_{xy}}{Z} & \text{if}(x, y) \in E \\ 0 & \text{otherwise} \end{cases}$$

where π_{xy} is the probability between nodes x and y with$Z$ as the normalizing constant.
It is sufficient to perform $32 - 64$ random walks per Node. The resulting paths are of the same structure as sentences in a text, so the word2vec method can now be used on it. The result is a set of vectors V where each vector v_i is a distribution of the probability that two nodes are next to each other. [3]

2.3.3 Node2Vec. This is an optimized version of DeepWalk. Again, there is the sampling phase in the beginning. The difference in the methods is, that in node2vec the random walks are now biased and not completely random. An order for the random walk is defined through two parameters p and q. To evaluate which node to visit next the walk looks at the transition probability π_{xy} on the edge (x, y). This probability is now defined as $\pi_{xy} = \alpha_{pq}(x, y) \cdot w_{xy}$ where w_{xy} is the weight assigned to the edge (x, y) and:

$$\alpha_{pq}(xy) = \begin{cases} \frac{1}{p} & \text{if } d_{xy} = 0 \\ 1 & \text{if } d_{xy} = 1 \\ \frac{1}{q} & \text{if } d_{xy} = 2 \end{cases}$$

with d_{xy} being the shortest path between nodes x and y. Hereby, our parameter p defines how likely it is to revisit a node of the walk. Setting p to a higher value encourages the walk to go deeper into the graph, whereas a small value ensures the walk to stay local around the starting point. In contrast, q describes the preference of nearer or further nodes. For $q > 1$ the walk favors closer nodes, for $q < 1$ nodes that are further away. [4]

2.3.4 Structural Deep Network Embedding. In contrast to the methods used before, SDNE does not use random walks. It aims to preserve local pairwise similarity which characterizes the local structure, and as well as the global network structure. To achieve this, two autoencoder neural networks are used. These get an adjacency vector as input and construct node adjacency vectors as output. Then the distance between the two outputs is computed and added to the loss function of the network. The total loss function is then calculated through summation of the distance loss plus the losses of the two encoders. At the end remains a collection of adjacency vectors which describe the graph structure. [3]

2.3.5 Graph2Vec. Now not the nodes are represented by vectors, but the whole graph. For that, there are once again three steps. In the first step the algorithm creates sub-graphs for each node and encodes them once again in a one-hot code. We then use these sub-graphs to train the network used in word2vec to maximize the probability that a predicted sup-graph exists in the input graph. The embedding is then the result of the network. [3]

3 EMBEDDING OF TEMPORAL GRAPHS

Let $G = (V, E)$ be a temporal graph with $G_t = \{G_{t_1}, \ldots, G_{t_T}\}$ as its evolution over the time steps T. The goal is now, to create a vector space which can be used to analyze the graph to find anomalies in it, compare it to other graphs, classify nodes or predict if certain links are going to exist. Depending on which outcome is wanted, different methods have to be used to optimize it. For predictions on the node level like node classification and link prediction a node-level algorithm should be used. On the other hand, a graph-level algorithm should be used to compare graphs with each other and to find anomalies, as shown in 3.3.

3.1 tNodeEmbed

As aforementioned, a node-level algorithm should be used to perform e.g. link predictions and node classification. One proposed algorithm of that form is *tNodeEmbed*. Here, the embedding can be split into three parts:

3.1.1 Initialization. To start out, the algorithm initializes a representative vector $Q_t \in \mathbb{R}^{T \times d}$ for each node v in all graphs $G_{t_1}, \ldots, G_{t_T}$, where d is the embedding size, and T the number of time steps. These vectors are created using the node2vec algorithm discussed in 2.3.3.

3.1.2 Node alignment. A downside to node2vec is that, when running on a graph, it aims to minimize the word embedding distances

but not consistency over multiple trainings. The resulting embedding could be in a two-dimensional space for example. It lays in a x-y-plane. Now imagine a rotation around the unused z-axis. The resulting graphs have the same structure but different orientation in the plane, thus not having an alignment, as seen in figure 3.

Similarly, when embedding two graphs G_{t_i} and G_{t_j} it is not guaranteed that, even if the graphs are identical, their axes align. To adjust those embeddings, tNodeEmbed uses an orthogonal transformation between embeddings at two time points t_i and t_j[5]. Used in this algorithm, it takes the matrix $Q_t \in \mathbb{R}^{d\times|V|}$ of node embeddings at time t. Then the matrices get aligned iteratively starting from the earliest timestep. For the alignment an orthogonal matrix R between Q_t and Q_{t+1} is needed to result in the final embedding $Q'_t = RQ_t$. The matrix are is approximated by :

$$R_{t+1} = \arg\min_{Rs.t.R^TR=I} \|RQ_{t+1} - Q_t\|$$

where $R_{t+1} \in \mathbb{R}^{d\times d}$ is the transformation which fits the time steps the most.

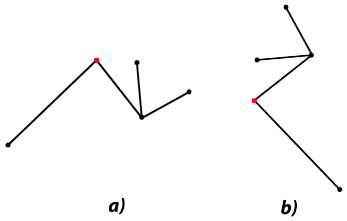

Figure 3: a) graph described by first embedding. b) graph described by second embedding. The red node do not align even though the graph structure is the same.

3.1.3 Finalization. The algorithm works in such a way, that the only difference in the algorithm, between the two outcomes, is the loss function. For the node classification task it uses a categorical cross-entropy loss:

$$L_{task} = -\sum_{v\in V} \log Pr(class(v)|f_T(v))$$

For the link prediction task it considers a binary classification loss:

$$L_{task} = -\sum_{v_1,v_2\in V} \log Pr((v_1,v_2) \in E|g(f_T(v_1), f_T(v_2)))$$

The Algorithm now wants to learn a function F_T so that $f_T(v) = F_T(v, G_1, \ldots, G_T)$ that optimizes for L_Task. The function is defined recursively as:

$$f_{t+1} = \sigma(Af_t(v) + BQ_tR_tv) \quad \text{and } f_0(v) = \vec{0}$$

with A, B, R_t and Q_t being matrices, v again being a one-hot vector representing a node and σ being an activation funcion. The final temporal embedding is now defined as the minimization of the loss-function:

$$L = \min_{A,B,Q_1,\ldots Q_T,R2,\ldots,RT} L_{task}.$$

Now, it is only a Question on how to define A, B and σ. Due to the steps beforehand, each node is associated with a matrix $X^{(v)} \in \mathbb{R}^{T\times d}$ consisting of all embeddings T, which are of size$d$, thus the whole graph can be seen as $G_X = X^{(v_1)}, \ldots, X^{(v_{|V|})}$. To perform the wanted graph-prediction tasks, the matrices representing each node now need to be reduced into a single vector, so it can be used as input for a classifier, as in the static embeddings. For this it is proposed to use recurrent neural networks with long short term memory. [6]

3.2 tdGraphEmbed

The goal of this algorithm is to create a mapping function that embeds a graph G_t into a $d-$dimensional space $\mathbb{R}^d$ with $d \in \mathbb{Z}$. This embedding captures not only the nodes' evolution over time, but also the graphs structure. Additionally, graphs with a similar structure have an embedding close to each other. To achieve this, the algorithm follows these steps:

3.2.1 Random Walks. The algorithm uses random walks W, as described in 2.3.2, to sample the graphs. Using these random walks, the algorithm estimates the likelihood of the graph at time t being completely described through them. This likelihood is defined as:

$$Pr(G_t|W_{v_1}, W_{v_2}, \ldots, W_{v_k}))$$

with $v_1, v_2, \ldots, v_k \; inV_t$. Such random walks are initiated γ times for each node in $V_t \in G_t$. Those random walks are then concatenated forming a document describing the graph.

3.2.2 Context Nodes. After all the documents are formed, the algorithm predicts the next node in a random walk. The nodes in a neighborhood, later referred to as context nodes, are defined as:

$$N_s(v_i^t) = \{v_{i-\omega}^t, \ldots, v_{i+\omega}^t\}.$$

Using these and the graph G_t the algorithm maximizes the equation:

$$\log p(v_i^t|N_s(v_i^t), G_t)$$

The goal is to learn a $d-$dimensional representation ϕ where $\phi(v_i^t)$ is the mapping of node $v_i \in V_t$ and $\phi(G_t)$ the mapping of the graph. With this mapping and a soft-max function the equation above can be caluclated with:

$$p(V_i^t|N_s(v_i^t), G_t) = \frac{e^{\phi(v_i^t)\cdot h}}{\sum_{j\in V_t} e^{\phi(v_j^t)\cdot h}}$$

$$\text{where } h = \sum_{c\in N_s(v_i^t)} \phi(v_c^t) + \phi(G_t)$$

where G_t is the global context shared between all embeddings. This is combined with the local context of the Nodes.

3.2.3 Optimization. With these equations the output calculation for each random walk W can be simplified to:

$$\max_{\phi} \sum_{t\in T} \sum_{W\in G_t} \sum_{v_i^t} -log \sum_{j\in V_t} e^{\phi(v_j^t)\cdot h} \quad + \phi(v_i^t \cdot h).$$

The calculating neural network is trained using gradient descent [7]. For each step, a fixed-length context is sampled from a walk W, with which the error gradient is calculated and used to update the parameters of the model. The last equation gets computationally expensive for large graphs, thus a negative sampling strategy is used [8]. To enhance the efficiency even more, the algorithm only considers nodes that are active in each time step instead of all nodes in V. Active nodes are those, who change one or more property over a time step. Even though, the optimizations influence the computation heavily, the embedding ϕ is still accurate. [9]

4 APPLICATION

Nowadays, graphs are used for a variety of applications. With it comes the need to analyze such graphs. They model e.g. Protein to Protein interactions in biology, social networks in social sciences and even so-called word co-occurence in linguistics. To understand and analyze those structures better, scientists have aimed to model such graphs and the interaction between different entities they contain. Those analytical tasks can be divided into four main groups: node classification, link prediction, clustering and visualization [10].

4.1 Node classification

A lot of the information contained in graphs can be put into different categories e.g. gender, age, demographic and political beliefs. Those categories are called labels and can be applied to nodes in a graph representing e.g. users of a social network. Such labels can then be used to recommend new connections between individuals, movies and music based on the interests and personalize advertisements.

4.2 Link prediction

Every relation in a graph describing a social network, can be described through an edge or link. So, whenever one entity engages another, the graph changes. The link prediction problem defines the question, how likely it is, given a snapshot of such a social network, that members engage each other in the near future [11]. Everybody has an intuitive feeling for who would follow whom. Due to the fact that they may have the same friends, live in the same city or have the same hobbies. Link prediction is the attempt to make these intuitive thoughts more precise and have an underlying method on how to do so.

4.3 Clustering

With increasing graph size, increases the probability that two nodes share a lot of labels. Clustering aims to find such nodes and generate a sub-graph out of them. Using this, e.g. users of an online social network with the same interests can be quickly found as they are members of the same sub-graph. It also enables to structure the network, as one can label whole sub-graphs instead of individual nodes. [12]

4.4 Similarity

Another use for embeddings is answering the question of how alike two graphs are. The more similar the topologies of the two graphs are, the closer the embeddings are to one another. This can be used on whole graphs or to see if a graph contains multiple similar sub-graphs. If a sub-graph existed in another part of the graph, this information can be used to predict its evolution, as the earlier formed sub-graph can be used as reference point [9].

4.5 Anomaly

Using the evolution of temporal graphs, repeating trends can be found. With the repeating patterns it is also possible to find anomalies in the graph which to not adhere to the pattern. This can be used to analyze how big of an impact e.g. political decisions can have on social constructs [9].

5 CONCLUSION

REFERENCES

[1] G. B. Mertzios, H. Molter, R. Niedermeier, V. Zamaraev, and P. Zschoche, "Computing maximum matchings in temporal graphs," *CoRR*, vol. abs/1905.05304, 2019.

[2] B. Gao and L. Pavel, "On the properties of the softmax function with application in game theory and reinforcement learning," 2018.

[3] P. Godec, "Graph embeddings — the summary - towards data science," *Towards Data Science*, 31.12.2018.

[4] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks."

[5] P. H. Schönemann, "A generalized solution of the orthogonal procrustes problem," *Psychometrika*, vol. 31, no. 1, pp. 1–10, 1966.

[6] U. Singer, I. Guy, and K. Radinsky, "Node embedding over temporal graphs," *IJCAI*, pp. 4605–4612, 2019.

[7] S. Ruder, "An overview of gradient descent optimization algorithms."

[8] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method."

[9] M. Beladev, L. Rokach, G. Katz, I. Guy, and K. Radinsky, "tdgraphembed: Temporal dynamic graph-level embedding," ACM, 2020.

[10] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowledge-Based Systems*, vol. 151, pp. 78–94, 2018.

[11] D. Liben-Nowell and J. Kleinberg, "The link prediction problem for social networks," in *CIKM 2003* (D. Kraft, O. Frieder, J. Hammer, S. Qureshi, and L. Seligman, eds.), (New York, N.Y.), p. 556, Association for Computing Machinery, op. 2003.

[12] C. Ding, X. He, H. Zha, M. Gu, and H. D. Simon, "A min-max cut algorithm for graph partitioning and data clustering," in *Proceedings* (N. Cercone, ed.), (Los Alamitos, Calif), pp. 107–114, IEEE Computer Society, 2001.