Machine Learning for Complex Networks

Prof. Dr. Ingo Scholtes

Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany

ingo.scholtes@uni-wuerzburg.de

Lecture 10
Neural Graph Representation Learning

July 3, 2024

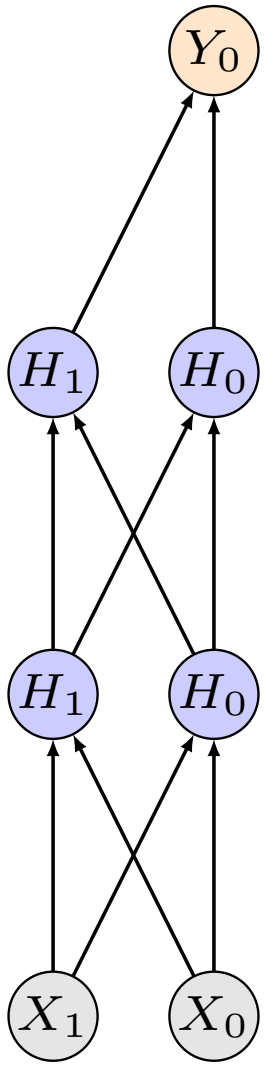
Notes:

- **Lecture L10:** Neural Graph Representation Learning 03.07.2024
- **Educational objective:** Addressing the problem of word embeddings in natural language processing, we introduce the SkipGram model. We then show how we can apply the SkipGram model to random walk sequences to learn low-dimensional representations of nodes in complex networks.
  - Neural Networks and Word Embedding
  - Node Embeddings and DeepWalk
  - Biased Random Walks and node2vec

- **Exercise sheet 09:** node2vec 10.07.2024

Motivation

- we have considered **matrix factorization techniques** to generate embeddings of nodes in Euclidean space → Lecture L08
- we introduced **neural networks** and used them for supervised graph learning → Lecture 09
- neural networks allow to **approximate arbitrary functions** based on loss function and gradient-based optimization
- how can we apply (deep) neural networks to graph-structured data?
- can we use **neural networks to learn graph representations**?



Notes:

- In the last two weeks, we introduced important techniques that facilitate supervised and unsupervised machine learning in complex networks. In lecture 08, we introduced graph representation learning, i.e. the task of embedding nodes in a (compared to the number of nodes) low-dimensional Euclidean space. Those node embeddings can then be used as node features, i.e. we can apply them in downstream graph learning tasks.
- In Lecture 08, we covered matrix factorization techniques for graph representation learning. We discussed the eigendecomposition of the adjacency matrix (which can be viewed as an application of PCA to graphs), Laplacian eigenmaps (which minimize the Euclidean distance between connected node pairs) and singular value decomposition (which generates the best low-rank approximation of the adjacency matrix w.r.t. the Frobenius norm).
- In Lecture 09, we explored how we can supervised classification techniques to those node representations, showing that we can use them to address node classification and link prediction. We further introduced neural networks, which can be thought of as powerful method to approximate functions based on a given loss function and gradient descent optimization.
- Considering the fact that matrix factorization can be used to find low-dimensional vector representations of nodes that minimize a loss function, it is natural to ask whether we can address the same issue in a heuristic fashion based on neural networks. This would yield neural graph representation learning, with potentially better performance than matrix factorization techniques.

From node to word embeddings

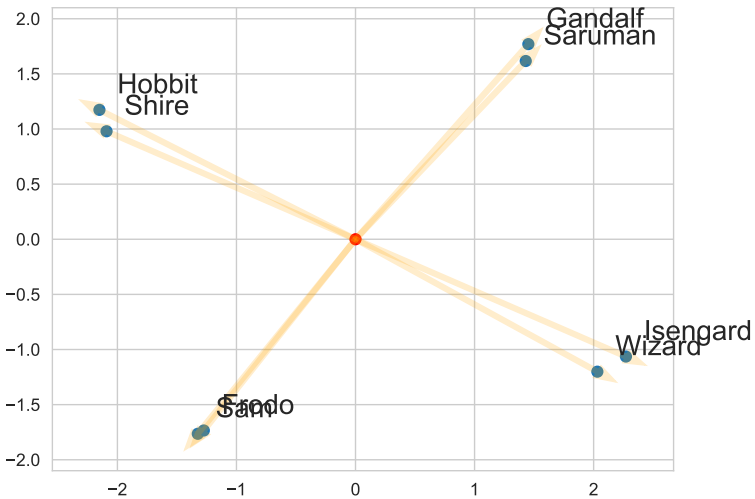
- in **natural language processing** we are interested in **word embeddings**
- for text with vocabulary V find **embedding of words in vector space**, i.e. $f : V \to \mathbb{R}^d$
- goal: vector space embedding should capture **semantic similarities**
- for words $v, w \in V$ with embeddings $\vec{v} := f(v)$ and $\vec{w} := f(w)$ we want

$$sim(v, w) \approx \vec{v} \cdot \vec{w}$$

- basis for text completion, semantic analysis, sentiment analysis, homonym detection, ...

Frodo, Sam, Gandalf, Saruman, Hobbit, Wizard, Shire, Isengard

sample vocabulary



word embedding in $\mathbb{R}^2$

Notes:

- Before we consider the problem of **graph representation learning**, let us take a quick excursion into natural language processing, where representation learning is an importance challenge as well. In NLP, we are interested in **word embeddings**, i.e. for a vocabulary V consisting of words that occur in a text corpus, we are interested in a mapping from V to a low-dimensional Euclidean space $\mathbb{R}^d$. The goal of this embedding is to capture **semantic similarities**, i.e. we want the vector representation of words that are semantically related to be similar, e.g. in terms of the dot product or cosine similarity.
- An exemplary embedding of words in a small text corpus is shown on the left. We can use such an embedding for various tasks in NLP, e.g. for text completion, semantic analysis, sentiment analysis or homonym detection. In the example above, we could use the vector representations of words to guess that Hobbits live in the shire, that Frodo and Sam are similar characters, that a Wizards resides in Isengard, etc.
- But how can we generate such a word embedding?

Distributional semantics

- we can use **distributional properties of text corpora** to learn semantic relationships between words

distributional hypothesis

"You shall know a word by the company it keeps!" → JR Firth, 1957

- we can study **co-ocurrence of words in text** by constructing `(word, context)` pairs
- for given **window size** s and center word w, consider all pairs (w, c) such that c occurs within distance at most s of w

example ($s = 2$, ignoring "a", "in", "is", "the")

(lives, Shire), (Shire, lives), (Gandalf, Wizard), (Wizard, Gandalf), (Frodo, Hobbit), (Hobbit, Frodo), (Sam, Hobbit), (Hobbit, Sam), (Saruman, Wizard), (Wizard, Saruman), (Sam, lives), (Sam, Shire), (lives, Sam), (Shire, Sam) …

Gandalf is a Wizard,
Frodo is a Hobbit,
Sam is a Hobbit.
Saruman is a Wizard.
Sam lives in the Shire.
Frodo lives in the Shire.
Saruman lives in Isengard.
Gandalf is imprisoned in Isengard.

sample text corpus

Notes:

- One popular approach to generate word embeddings, which is facilitated by the availability of large corpora of texts, is based on the so-called **distributional hypothesis**, which was formulated in the 1957 summary of state-of-the-art linguistics. It refers to the fact that we can assess the meaning of a word based on those words that occur in close proximity in a text corpus.
- This is the basis for a simple approach to learn semantic similarities between words: We consider word-context pairs (w, c), which are constructed based on the sequence of words occurring in sentences. We call w the center words, and we add a pair (w, c) whenever the so-called *context*-word appears at a distance smaller than a given value s, which is the window size of our analysis.
- In this analysis, it is common to exclude frequently occurring stop-words, which can appear in many different contexts and which are thus not informative. For the example above, we excluded the stop words “a”, “in”, “is” and “the”, which leaves us with a vocabulary of ten words. By sliding a window of size $s = 2$, we obtain the word-context pairs shown in the example above.

The SkipGram model

- consider a model that predicts "context words" c based on "center" word w, i.e. for model params Θ we consider **probability of context word c conditional on center word w**

$$P(c|w;\Theta)$$

- for set $C = \{(c_i, w_i)\}$ of **observed word-context pairs** we can fit model parameters by maximizing likelihood

$$L(\Theta) = \prod_{(c,w)\in C} P(c|w;\Theta) \prod_{(c,w)\notin C} (1 - P(c|w;\Theta))$$

- using a logarithmic transformation, we can instead **minimize loss function**

$$L(\Theta) = -\sum_{(c,w)\in C} \log P(c|w;\Theta)$$

Notes:

- Using the word-context pairs as training data, let us now consider a statistical classifier that predicts a context word c based on a given center word w, i.e. we are interested in the probability of context words c conditional on a given center word c. Let us further assume that those probabilities depend on a parameter (vector) Θ.
- We could now use the word-context pairs c_i, w_i observed in a given text corpus to fit the parameters of our statistical model, i.e. we can use the training data to find the most plausible model parameters, e.g. in terms of the likelihood. For our word-context pairs, the likelihood of the model parameters can simply be calculated as a product of the conditional probabilities over all observed word-context pairs and the inverted probabilities of those word-context pairs that we did not observe in the corpus.
- We note that this can be simplified, since by maximizing the first term $\prod_{(c,w)\in C} P(c|w;\Theta)$ we implicitly maximize

$$L(\Theta) = \prod_{(c,w)\in C} P(c|w;\Theta) \prod_{(c,w)\notin C} (1 - P(c|w;\Theta))$$

  This is due to the fact that the probabilities must sum to one.
- If we apply a logarithmic transformation and take the negative sum of the log-probabilities, we obtain a loss function whose minimization corresponds to the maximization of the likelihood of our model.

Vector representations

- assume that each word $w \in V$ is represented by vector $\vec{w} \in \mathbb{R}^d$
- considering the **distributional hypothesis** we are interested in vector representations such that

$$P(c|w) \sim \vec{c} \cdot \vec{w} \in \mathbb{R}$$

- for given center word w we use **softmax function σ to normalize dot products $\vec{c} \cdot \vec{w}$ to probabilities**, i.e. for given vector representations we define

$$P(c|w) = \frac{e^{\vec{c} \cdot \vec{w}}}{\sum_{x \in V} e^{\vec{x} \cdot \vec{w}}}$$

- for given center word w, softmax function yields **categorical distribution over all possible context words c**

Notes:

- Rather than in conditional probabilities, we are actually interested in a vector representation of words. Let us assume that the words w and c in our word-context pairs (w, c) are represented by vectors $\vec{w}$ and $\vec{c}$ respectively. Considering the distributional hypothesis that "semantically similar" words frequently occur in close proximity, we are interested in a vector representation such that higher probabilities $P(c|w)$ are associated with a larger dot product between vectors $\vec{c}$ and $\vec{w}$.
- We can approach the problem of finding optimal vector space representations of nodes based on the dot product between vectors representing the word-context pairs. To apply the loss function from the previous slide, we need probabilities. Similar to logistic regression, we can use a generalization of the logistic function to the multinomial case (i.e. more than two classes). With the so-called softmax function, we can map the dot product between vectors to values between 0 and 1, i.e. for each center word w we obtain a categorical probability distribution over all context words c_i.
- With this approach, we can use the vector positions of nodes in $\mathbb{R}^d$ as parameters of our model. As we shall see on the next slide, we can use a feed-forward neural network to find optimal parameters, i.e. vectors that "explain" the observed conditional probabilities in our word-context pairs.

A Neural Network approach

- we can use a feed-forward neural network to **find vector representations of words that minimize loss function**
- consider **one-hot encoding** of k-th word x in vocabulary, i.e.

$$\vec{x} = (\underbrace{0, \ldots, 0}_{k-1 \text{ times}}, 1, 0 \ldots, 0)$$

- define **neural network** with input $\vec{x} \in \mathbb{N}^{|V|}$ being the **one-hot encoding of center word** x
- add **hidden layer** with d perceptrons h_j and weights $\beta^1 \in \mathbb{R}^{|V| \times d}$, i.e.

$$h_j = \sum_{i=1}^{|V|} \beta^1_{i,j} x_i =: (\vec{\beta}^1)_j \cdot \vec{x}^T$$

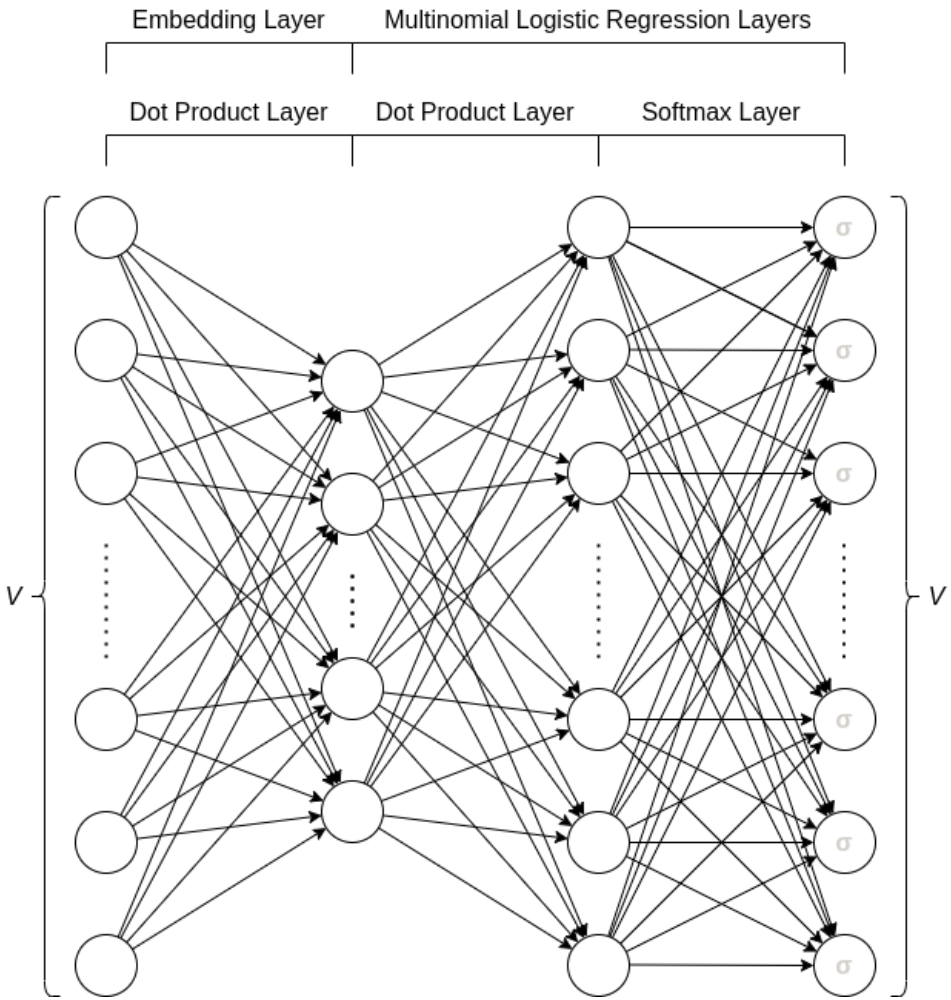
- add **second layer with $|V|$ perceptrons** with weights $\beta^2 \in \mathbb{R}^{d \times |V|}$ and **softmax activation function** σ with output

$$y_i = \sigma(\vec{\beta}^2_i \cdot \vec{h}^T) =: P(c_i|x)$$

Notes:

- We can use a neural network to find model parameters, i.e. vector representations of nodes, that minimize the loss function. This neural network accepts a center word as input and outputs the probabilities of all context words, which are computed based on the softmax transformation of the dot product between the given center word and all possible context words. Since we have a categorical input, we can use one-hot encoding to encode the input to our neural network, i.e. we use a vector $\mathbb{N}^{|V|}$ and encode the k-th word in our vocabulary V by a vector in which the k-th element is one and all other elements are zero.
- We thus have a neural network with $|V|$ inputs. To find an embedding of words in $\mathbb{R}^d$ we add a fully connected hidden layer with d perceptrons, where each perceptron receives each of the $|V|$ inputs. Referring to the description of the perceptron model in Lecture 09, the output of each perceptron is computed as linear combination of its parameters with all inputs. For each hidden neuron h_j, we can thus express the output as dot product between the vector of $|V|$ inputs and the parameter vector $(\beta^1)_j$. We further note that there are d weight parameters associated with each of the $|V|$ inputs, one for each of the d neurons in the hidden layer.
- In a standard perceptron model, we would add an activation function to transform the output of the dot product to a probability. Here we instead add a second linear layer of $|V|$ perceptrons, feeding the scalar output (i.e. the dot product) of each neuron in the first hidden layer as input to each of the $|V|$ neurons in the second layer. We finally apply a softmax transformation to the output of each neuron in the second layer, which ensure that we can interpret the outputs as probabilities.

Skipgram model and `word2vec`



Efficient Estimation of Word Representations in Vector Space

Tomas Mikolov
Google Inc., Mountain View, CA
tmikolov@google.com

Kai Chen
Google Inc., Mountain View, CA
kaichen@google.com

Greg Corrado
Google Inc., Mountain View, CA
gcorrado@google.com

Jeffrey Dean
Google Inc., Mountain View, CA
jeff@google.com

Abstract

We propose two novel model architectures for computing continuous vector representations of words from very large data sets. The quality of these representations is measured in a word similarity task, and the results are compared to the previously best performing techniques based on different types of neural networks. We observe large improvements in accuracy at much lower computational cost, i.e. it takes less than a day to learn high quality word vectors from a 1.6 billion words data set. Furthermore, we show that these vectors provide state-of-the-art performance on our test set for measuring syntactic and semantic word similarities.

1 Introduction

Many current NLP systems and techniques treat words as atomic units - there is no notion of similarity between words, as these are represented as indices in a vocabulary. This choice has several good reasons - simplicity, robustness and the observation that simple models trained on huge amounts of data outperform complex systems trained on less data. An example is the popular N-gram model used for statistical language modeling - today, it is possible to train N-grams on virtually all available data (trillions of words [3]).

However, the simple techniques are at their limits in many tasks. For example, the amount of relevant in-domain data for automatic speech recognition is limited - the performance is usually dominated by the size of high quality transcribed speech data (often just millions of words). In machine translation, the existing corpora for many languages contain only a few billions of words or less. Thus, there are situations where simple scaling up of the basic techniques will not result in any significant progress, and we have to focus on more advanced techniques.

With progress of machine learning techniques in recent years, it has become possible to train more complex models on much larger data set, and they typically outperform the simple models. Probably the most successful concept is to use distributed representations of words [10]. For example, neural network based language models significantly outperform N-gram models [1, 27, 17].

1.1 Goals of the Paper

The main goal of this paper is to introduce techniques that can be used for learning high-quality word vectors from huge data sets with billions of words, and with millions of words in the vocabulary. As far as we know, none of the previously proposed architectures has been successfully trained on more

→ T Mikolov, K Chen, G Corrado, J Dean, 2013

we can use observed word-context pairs to train model and **interpret weight parameters as vector representations of nodes**

Notes:

- A conceptual illustration of the resulting neural network architecture, the so-called SkipGram model, is shown in the figure above. The network has $|V|$ inputs accepting one-hot encodings of center words, and $|V|$ outputs, which are the probabilities of all $|V|$ context words given the input word. The SkipGram model is one of the architectures used in the popular neural word embedding algorithm word2vec, which was first proposed (and patented) by Google in 2013.
- We note that each node in the input layer (representing one center word w based on the one-hot encoding) is associated with d weight parameters, one for each perceptron in the first hidden layer of our neural network. Those weight parameters can be interpreted as $\vec{w} \in \mathbb{R}^d$, i.e. a vector space representation of the input words w. Similarly, each of neuron in the second linear layer (which is associated with one context word c due to the one-hot-encoding) has d weight parameters, which can be interpreted as $\vec{c} \in \mathbb{R}^d$, i.e. a vector representation of the context word c.
- The key idea behind word2vec is to use observed word-context pairs to train a feed-forward neural network, and then use the learned weights as vector space representation of the words. But how can we train such a model?

Training the Skipgram model

1. for given word-context pair (w, c) **pass one-hot encoding $\vec{x}$ of center word w as input**
2. use neural network to **compute output $\vec{y} \in \mathbb{R}^{|V|}$ containing conditional probabilities $P(c_i|w)$ of all context words c_i**
3. compute one-hot encoding $\vec{\hat{y}}$ of observed context word c and use it as **ground truth** for output $\vec{y}$ generated by neural network
4. compute **cross entropy loss function**

$$L = -\sum_{i=1}^{|V|} \hat{y}_i \log y_i$$

5. use **backpropagation to compute gradients of loss function**
6. use **gradient descent** to find optimal weights β_1 and β_2 for both layers

Notes:

- How can we train this neural network? For each word-context pair (w, c) we pass the one-hot encoding of the center word w as input to the network. Using the current weight parameters (which can be initialized randomly), we then compute all $|V|$ outputs y_i, which are the conditional probabilities of context words c_i given center word w.
- We now use the observed context word c as "ground truth" for this specific word-context pair, i.e. we compute the one-hot encoding of the context word c and compare it with the output of the neural network. Ideally (only considering a single word-context pair) the network would yield a probability of one for the observed context word c and zero for all other context words, i.e. the output would perfectly match the one-hot encoding of c.
- Consider the definition of the cross entropy loss function, which we compute for each observed word-context pair. Thanks to the one-hot encoding of $\hat{y}_i$ we only consider the negative log probabilities of those words c_i that actually appear as context words. By minimizing this loss function, we thus maximize the conditional probability of our observations!
- To optimize the weight parameters β^1, β^2 of the two linear layers of our neural network, we can compute the cross entropy loss function based on the one-hot encoding of c. We then use backpropagation to compute the gradients of the loss function, which enables us to update all weight parameters based on (stochastic) gradient descent.

Matrix Factorization Perspective

- each of the d perceptrons h_i in **hidden layer** has $|V|$ weight parameters, i.e. $\beta^1 \in \mathbb{R}^{|V| \times d}$
- each of the $|V|$ perceptrons in **output layer** has d weight parameters, i.e. $\beta^2 \in \mathbb{R}^{d \times |V|}$
- Skipgram model generates **matrix $\mathbf{P} \in \mathbb{R}^{|V| \times V}$ of conditional probabilities** that can be expressed as

$$\mathbf{P} = \sigma(\mathbf{I} \cdot \beta^1 \cdot \beta^2)$$

where $\mathbf{I} \in \mathbb{R}^{|V| \times |V|}$ is identity matrix containing one-hot encoded input words and P_{ij} is probability of context word j given center word i

- for center word i output y_j of neural network is given as → exercise sheet

$$P_{ij} = \sigma\left(\vec{\beta}_j^2 \cdot (\vec{\beta}^1)_i^T\right)$$

i.e. $(\vec{\beta}^1)_i^T \in \mathbb{R}^d$ **and** $\vec{\beta}_i^2 \in \mathbb{R}^d$ **are vector representations of center word** i **and context word** j

Notes:

- To justify the interpretation of weights as vector representations of center and context words we can expand the expression for the outputs y_j of our neural network as

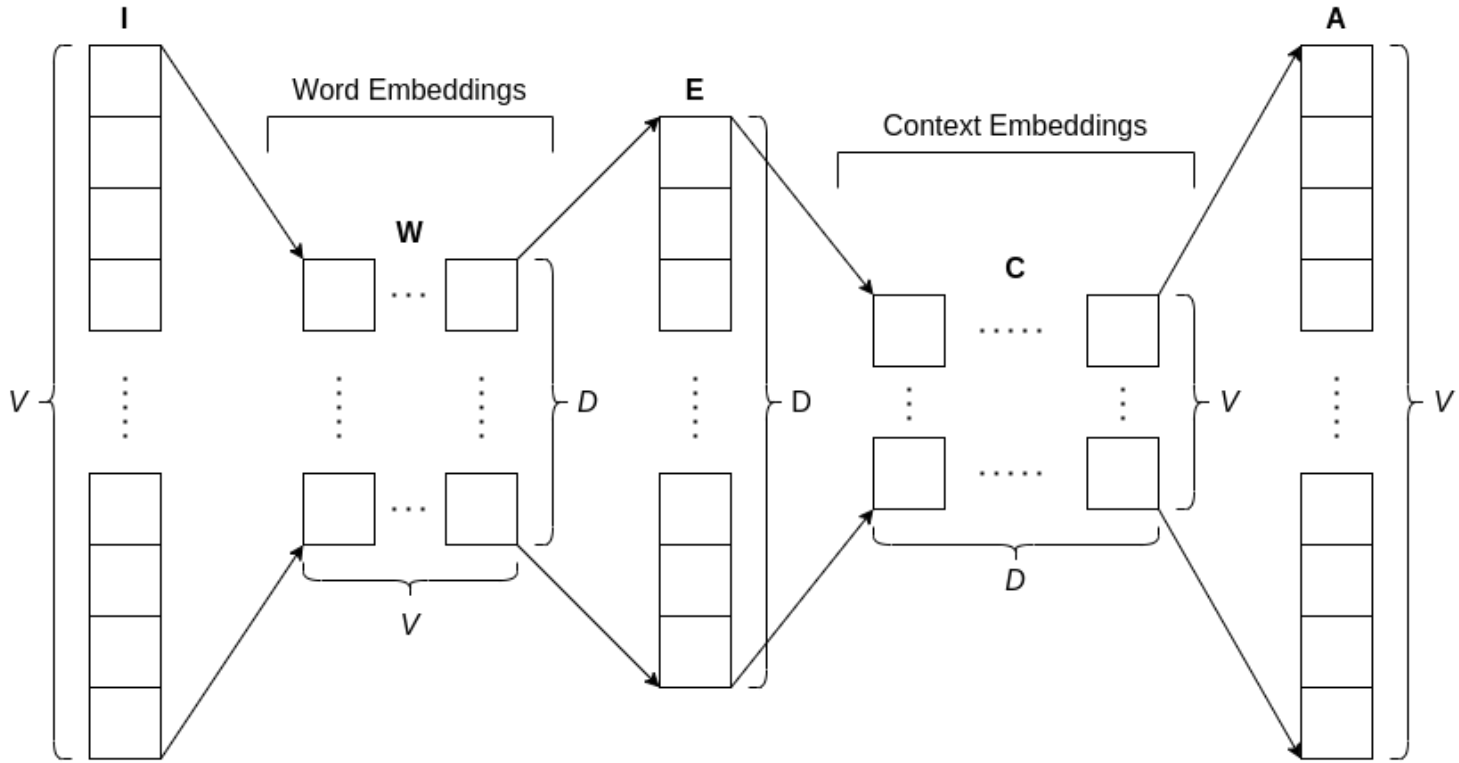
$$y_j = P(c_j|x) = \sigma\left(\vec{\beta}_j^2 \cdot \vec{h}^T\right) = \sigma\left(\sum_{i=1}^{d} \beta_{i,j}^2 \cdot h_i\right) = \sigma\left(\sum_{i=1}^{d} \beta_{i,j}^2 \sum_{k=1}^{|V|} \beta_{k,j}^1 \cdot x_k\right)$$

$$= \sigma\left(\sum_{i=1}^{d} \beta_{i,j}^2 \beta_{s,i}^1\right) = \sigma\left(\vec{\beta}_j^2 \cdot (\vec{\beta}_s^1)^T\right)$$

  where s is the index of the one in the one-hot-encoding $\vec{x}$ of the center word

- We can also take a matrix factorization perspective on the SkipGram architecture. For each of the d perceptrons in the first (hidden) layer we have $|V|$ weight parameters, i.e we can arrange weights in a matrix $\beta^1 \in \mathbb{R}^{|V| \times d}$. Similarly, for each of the $|V|$ perceptrons in the second linear layer, we have $|d|$ weight parameters, i.e. we can arrange in a matrix $\beta^2 \in \mathbb{R}^{d \times |V|}$.
- Each perceptron simply calculates a dot product between the inputs and the weights, i.e. we can express the output of a two-layer network as product of matrices $\mathbf{I} \cdot \beta^1 \cdot \beta^2$, where $\mathbf{I}$ is the identity matrix (which contains one column for each of the $|V|$ one-hot encoded center words).
- We can view the weight parameters of the layers as factorization of a matrix of conditional probabilities for all pairs of center and context words, with an additional application of the non-linear softmax function to normalize the output.

Neural Matrix Factorization



deep representation learning

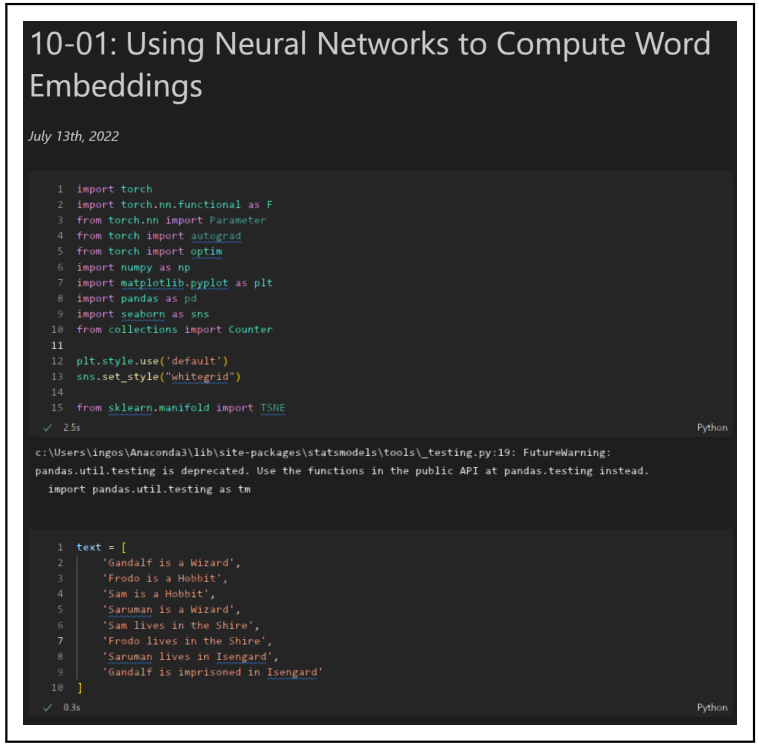
- neural network **approximately factorizes matrix** such that loss function is minimized
- i-th column $(\beta^1)_i^T \in \mathbb{R}^d$ contains **vector representation of input word** encoded in i-th component of one-hot vector
- i-th row $(\beta^2)_i \in \mathbb{R}^d$ contains **vector representation of context word** encoded in i-th component of one-hot vector
- use of deep neural networks to learn low dimensional embeddings is called **deep representation learning**

Notes:

- Similar to the matrix factorization techniques covered in Lecture 08, which were related to different loss functions, this neural network approach to matrix factorization depends on the loss function that is used to calculate the gradients and thus find the optimal weight parameters.
- Depending on the number of layers of the neural network, we call this approach (deep) neural representation learning.

Practice session

- we implement `word2vec` **from scratch** in `pytorch`
- we use it to **calculate word embeddings** for a simple text corpus


10-01: Using Neural Networks to Compute Word Embeddings

July 13th, 2022

```
import torch
import torch.nn.functional as F
from torch.nn import Parameter
from torch import autograd
from torch import optim
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from collections import Counter

plt.style.use('default')
sns.set_style("whitegrid")

from sklearn.manifold import TSNE
```

✓ 2.5s Python

```
c:\Users\ingos\Anaconda3\lib\site-packages\statsmodels\tools\_testing.py:19: FutureWarning:
pandas.util.testing is deprecated. Use the functions in the public API at pandas.testing instead.
  import pandas.util.testing as tm
```

```
text = [
    'Gandalf is a Wizard',
    'Frodo is a Hobbit',
    'Sam is a Hobbit',
    'Saruman is a Wizard',
    'Sam lives in the Shire',
    'Frodo lives in the Shire',
    'Saruman lives in Isengard',
    'Gandalf is imprisoned in Isengard'
]
```

✓ 0.3s Python

practice session

see notebook 10-01 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

Notes:

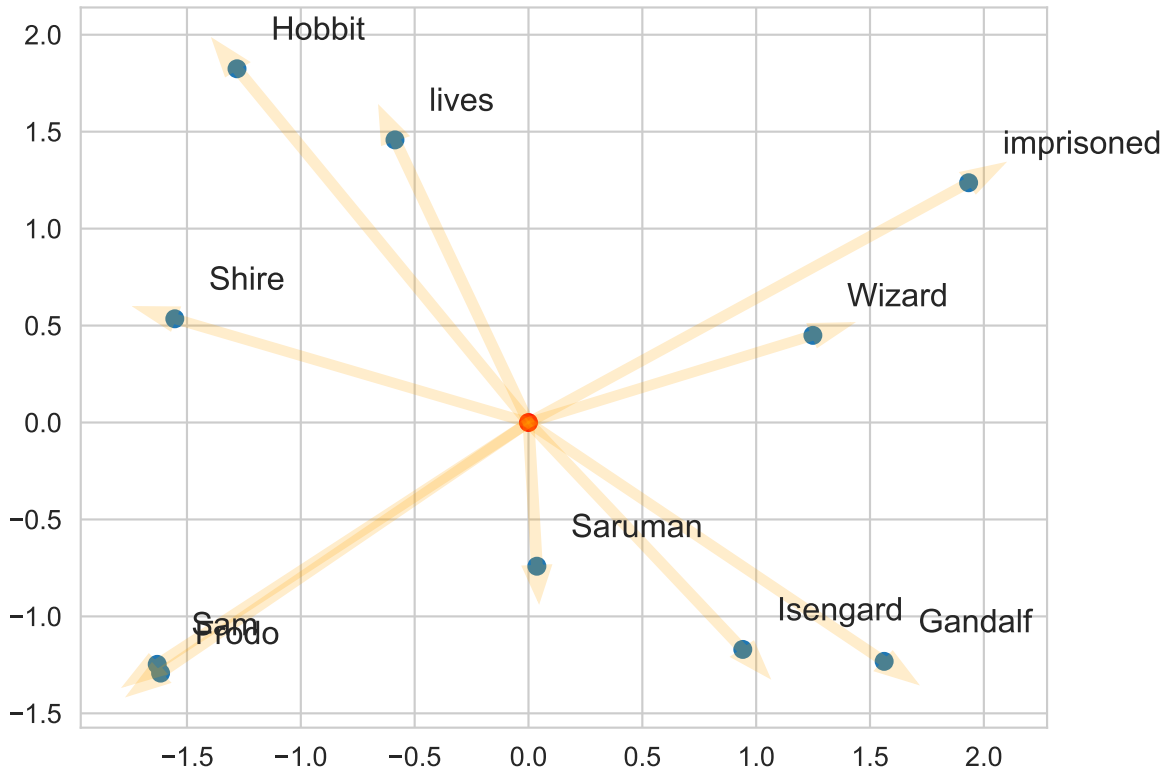
- In the first practice session, we use our newly acquired knowledge to implement the Skipgram model from scratch in pytorch. We use the model to calculate word embeddings for a small toy example text corpus.

Example: word2vec

example

- word2vec embedding using Skipgram model with $d = 5$ hidden dimensions
- window size $s = 2$, stop words $\{\text{is, a, the, in}\}$

Gandalf is a Wizard.
Frodo is a Hobbit.
Sam is a Hobbit.
Saruman is a Wizard.
Sam lives in the Shire.
Frodo lives in the Shire.
Saruman lives in Isengard.
Gandalf is imprisoned in Isengard.



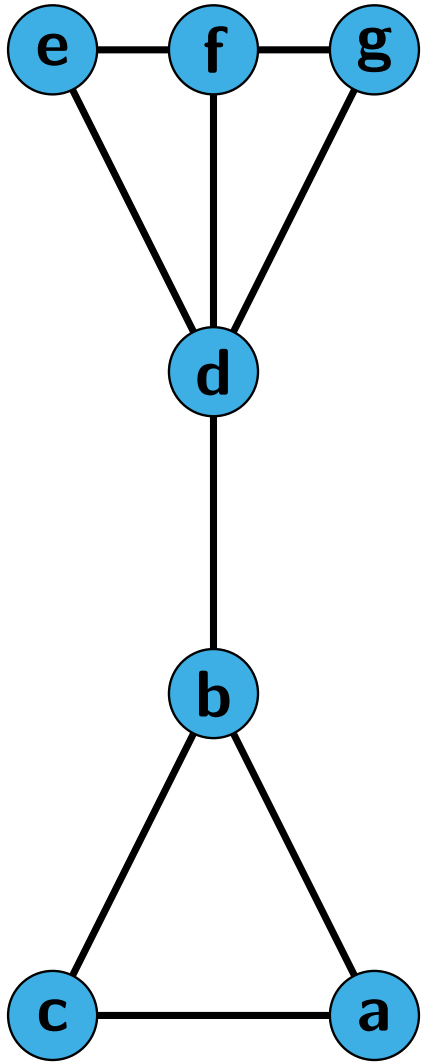
word embedding in $\mathbb{R}^2$ (dimensions reduced via truncated SVD)

Notes:

- The figure above shows an exemplary embedding obtained by applying the SkipGram neural network model to the simple text corpus shown on the left. We used four stop words and a window-size of two to generate word-context pairs. We then used those word-context pairs to train the SkipGram network and use the weights of the first linear layer as word embeddings.
- We find that semantically related words are represented by vectors that are similar with respect to the dot product, i.e. their angle.

From word to node embeddings?

- can we apply the **neural matrix factorization** idea behind `word2vec` to calculate graph embeddings?
- network analogy of **distributional hypothesis** in linguistics?
- for network $G = (V, E)$ consider **walk** $v_0, v_1, \ldots v_l$ **of length** l, i.e. $v_i \in V$ and $(v_i, v_{i+1}) \in E$ for $i = 0, \ldots, l-1$
- we expect that **"similar nodes" frequently co-occur** in sequence of traversed nodes
- both **distance and cluster structure** influence co-occurrence of nodes on walks



Notes:

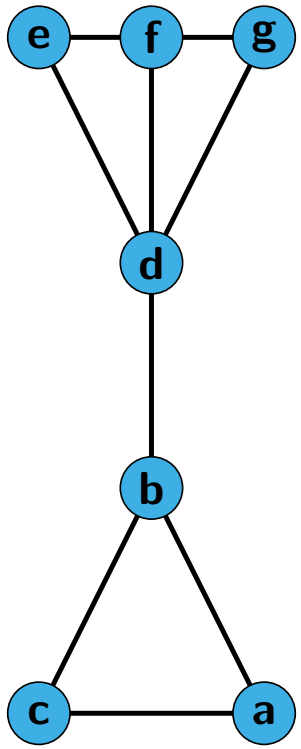
- In word2vec, words with similar context are identified as similar and get closer vector representations. Can we use the same approach to calculate vector representations of nodes in complex networks? To make this word, we first need a network analogy of the distributional hypothesis in linguistics, i.e. we need sequential data on nodes where nodes that are “similar” (in a topological sense) frequently occur in close proximity.
- A simple approach to generate sequences of nodes is a random walk model. I.e. starting at a random node, we randomly walk through the network for l steps. The sequence of traversed nodes can now be seen in analogy to a sentence consisting of l words. Like in linguistics, we expect nodes that are topologically similar to occur in close proximity. First of all, nodes at closer distance are expected to occur within shorter distance in the walk sequences. Moreover, we expect nodes in the same cluster to frequently occur in close proximity, because random walks are likely to be trapped within densely connected clusters of nodes.

DeepWalk

- consider **corpus of n randomly generated walks** of length l, i.e.

$$\{(v_0^1, \ldots, v_l^1), (v_0^2, \ldots, v_l^2), (v_0^3, \ldots, v_l^3), \ldots\}$$

- treating each walk as sentence, we can **compute node-context pairs** and use them to train SkipGram model
- use weight parameters of first hidden layer as **embedding of nodes in $\mathbb{R}^d$**



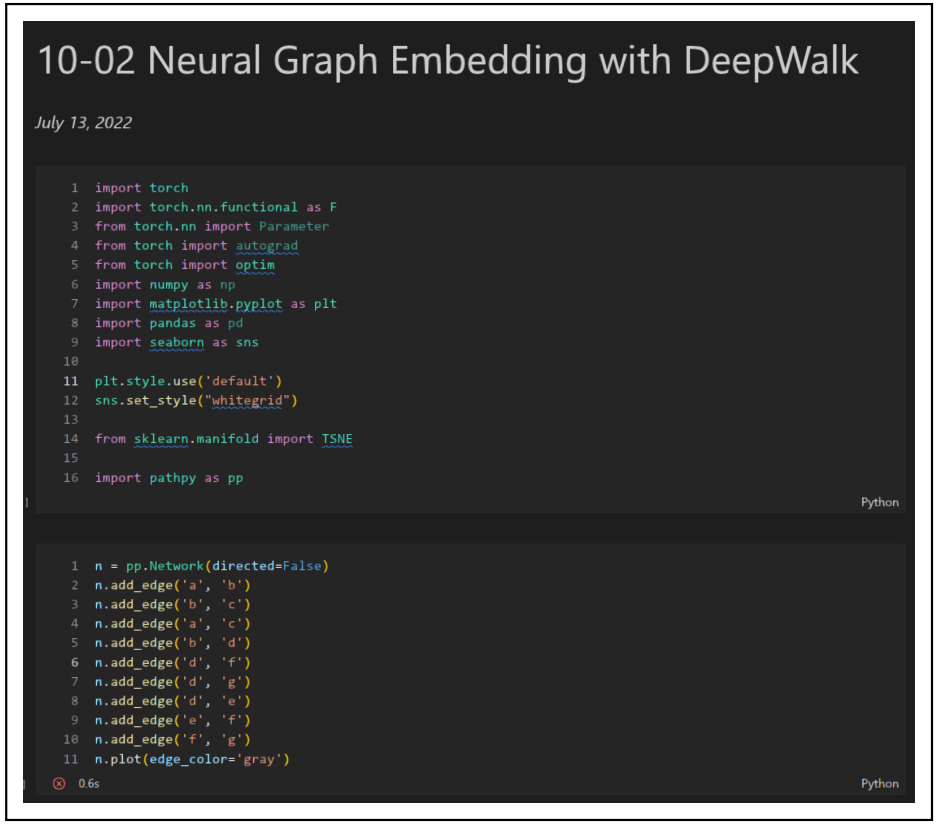
(a, b, c, a, c)
(e, f, g, e, d)
(b, d, e, d, f) ...

Notes:

- This analogy yields a simple yet powerful approach to neural graph representation learning. We generate a corpus of n random walk sequences, where each walk has length l. We can treat each walk in analogy to a sentence in a text corpus. While there are different ways to generate walks, it is common to start a number of relatively short walks in each node of the network. Alternatively, we could also use a smaller number of longer walks.
- We now simply treat the collection of random walks as text corpus, i.e. we generate node-context pairs based on a given window size s and train a SkipGram model on the resulting node pairs. We can now use the weight parameters of the first hidden layer to obtain an embedding of nodes in $\mathbb{R}^d$.

Practice Session

- we implement **graph representation learning with DeepWalk** in `pytorch`
- we use **DeepWalk** to learn node embeddings for empirical networks
- we compare **DeepWalk embeddings** to matrix factorization techniques



10-02 Neural Graph Embedding with DeepWalk

July 13, 2022

```python
import torch
import torch.nn.functional as F
from torch.nn import Parameter
from torch import autograd
from torch import optim
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

plt.style.use('default')
sns.set_style("whitegrid")

from sklearn.manifold import TSNE

import pathpy as pp
```

Python

```python
n = pp.Network(directed=False)
n.add_edge('a', 'b')
n.add_edge('b', 'c')
n.add_edge('a', 'c')
n.add_edge('b', 'd')
n.add_edge('d', 'f')
n.add_edge('d', 'g')
n.add_edge('d', 'e')
n.add_edge('e', 'f')
n.add_edge('f', 'g')
n.plot(edge_color='gray')
```

0.6s Python

practice session

see notebook 10-02 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

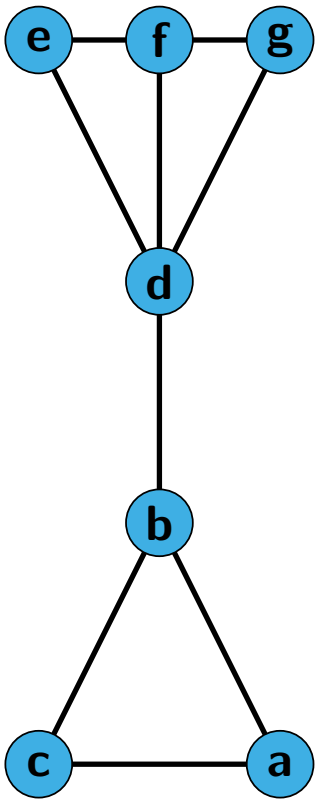
Notes:

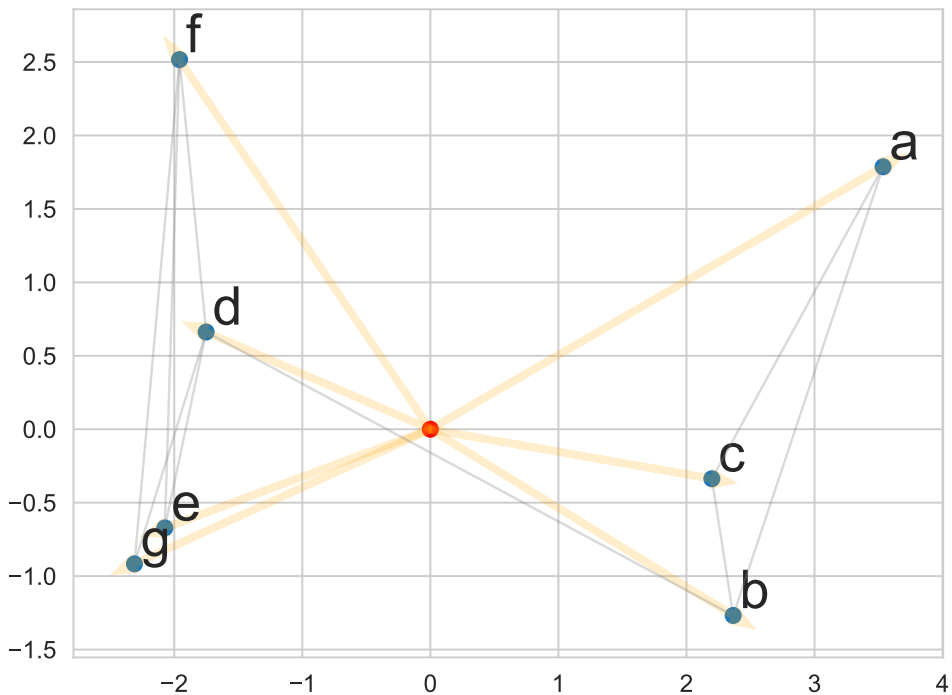
- Building on our from-scratch implementation of the SkipGram model, in the second practice session we implement Neural Graph Embedding with DeepWalk in pytorch.
- We apply DeepWalk to our toy example and compare its output to matrix factorization techniques.

Example: window size $s = 2$

example

DeepWalk embedding with $d = 5$ hidden dimensions using $n = 100$ walks of length $l = 10$ and window size $s = 2$





two-dimensional projection of DeepWalk embedding

generated by truncated singular value decomposition

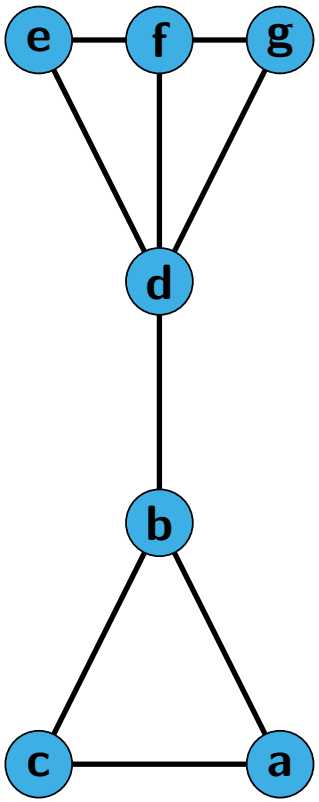
Notes:

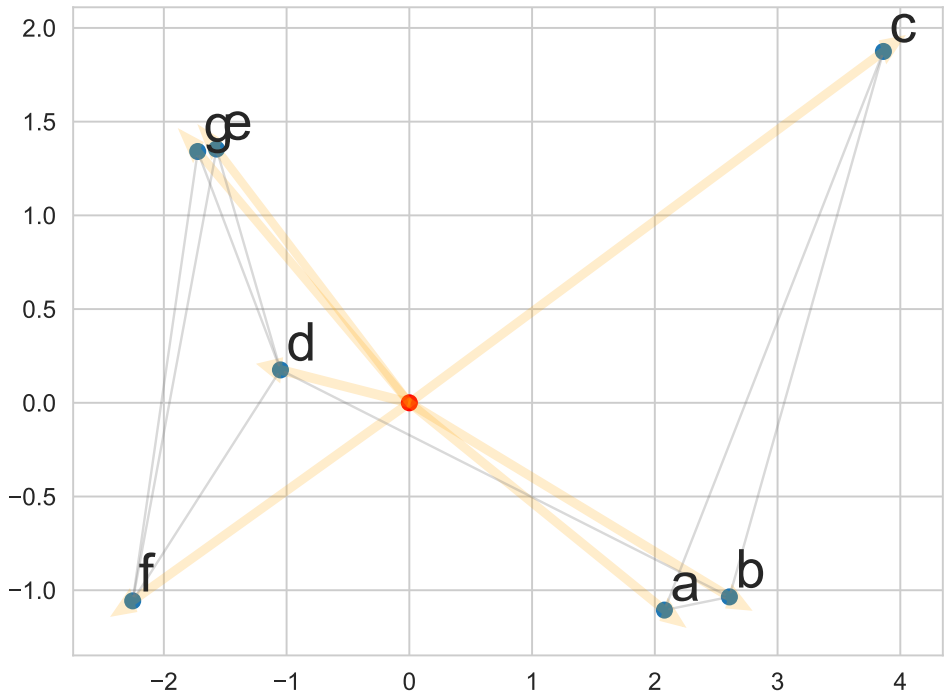
- The figure above shows the neural graph representation obtained using DeepWalk. Here we used 100 random walks started in randomly chosen nodes, each walk having length $l = 10$. We used a window size of $s = 2$ to generate word-context pairs. We further used five hidden dimensions and used a truncated singular value decomposition to visualize the embedding in two dimensions.

Example: window size $s = 3$

example

DeepWalk embedding with $d = 5$ hidden dimensions using $n = 100$ walks of length $l = 10$ and window size $s = 3$





two-dimensional projection of DeepWalk embedding

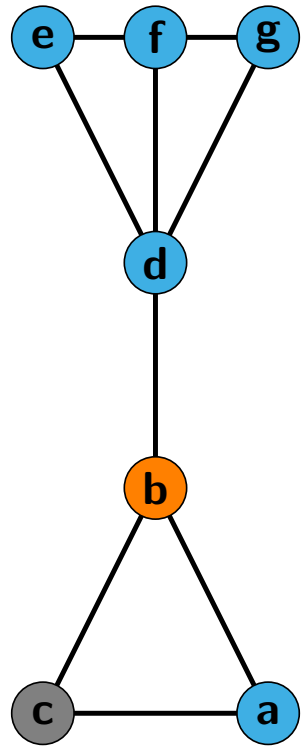
generated by truncated singular value decomposition

Notes:

- The window size influences the node-context pairs, and thus the notion of similarity that we use to generate the node embedding. If we use a window size of three, we get a different embedding of nodes in our toy example network. Here, a node-context pair is generated if two nodes occur at distance up to three in the walk sequences. This somewhat reduces the influence of the “local” neighbourhood of a node on its embedding.

Limitations of DeepWalk

- DeepWalk uses random walks where **neighbor of current node v is chosen uniformly at random**
- for low-degree nodes, we have high **backtracking probability**, i.e. we likely return to node that we came from
- in networks with **strong clustering**, walks likely "exploit" clusters rather than "exploring" network
- how could we influence the **exploration-exploitation** behaviour of random walker in a network?



example

- $P(b|b \to a) = \frac{1}{2}$
- $P(d|c \to b) = \frac{1}{3}$
- $P(\{a, c\}|c \to b) = \frac{2}{3}$

Notes:

- The question how local and non-local topological features of a network influence the resulting node embeddings leads us to a discussion of potential limitations of DeepWalk. We first note that, at least in an unweighted network, we generate node sequences based on a uniform random walk, i.e. the next node in the random walk is chosen uniformly at random from all the neighbours of the current node. This choice of a maximum entropy random walk has a couple of interesting implications regarding the representation of topological features in the resulting embedding.
- We first notice that for nodes with low degree, we have a large probability to return to the node that we came from. This is an implication of the Markovian nature of the random walk, which forgets where it came from as soon as it enters a node.
- We further note that for networks with strong clustering, walks are likely to repeatedly visit nodes within the local cluster rather than moving to nodes that are farther away. We actually used this “trapping” behaviour of random walks to detect communities using InfoMap. Here, this could be seen as limiting the explorative behaviour of random walks, i.e. it implies that the notion of similarity between nodes is strongly based on the cluster structure and less on the distance between nodes.

Biased random walks and node2vec

- introduce params p, q that guide **exploration-eploitation behavior**
  - $p \in (0, \infty)$ controls **backtracking**, i.e. prob. to return to recently visited node
  - $q \in (0, \infty)$ controls prob. to **move to node at larger distance**

Aditya Grover

Jure Leskovec

- for walker entering node v from t define **search bias** $\alpha_{pq}(t, x)$ for neighbors x as

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} \text{ if } dist(t,x) = 0 \\ 1 \text{ if } dist(t,x) = 1 \\ \frac{1}{q} \text{ if } dist(t,x) = 2 \end{cases}$$

- $q \ll 1$: resembles **depth-first search**
- $q \gg 1$: resembles **breadth-first search**
- DeepWalk = node2vec for $p = q = 1$

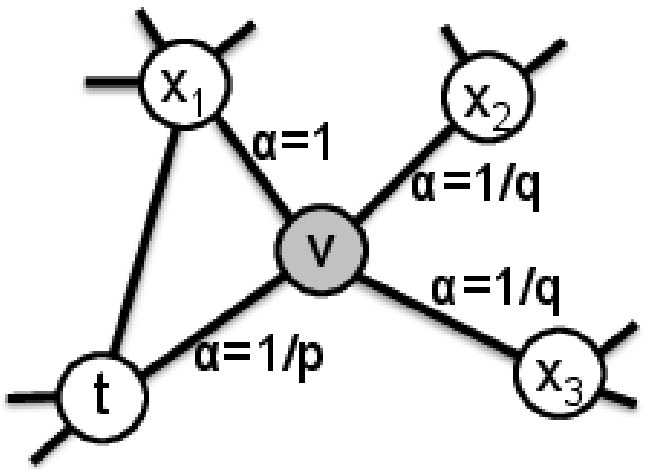


illustration of transition probabilities for random walker that just moved from node t to v

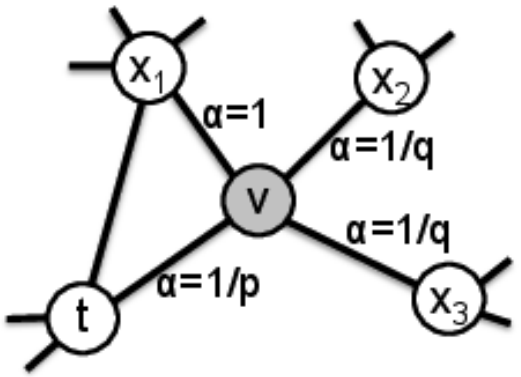
image from → A Grover, J Leskovec, 2016

Notes:

- To address this issue, the neural representation learning technique node2vec introduces two parameters p and q that control the backtracking and exploration-exploitation behaviour of the random walk model that is used to generate node sequences, and thus the node-context pairs that are fed to the SkipGram model (i.e. the underlying neural network model is exactly the same).
- node2vec replaces the uniform random walk in DeepWalk with a biased second-order random walk, i.e. it includes one step memory to tune the transition probabilities.
- This allows us to influence the probability to return , which is controlled via a parameter p. Larger values of p decrease the backtracking probability.
- We can further influence the probability with which a random walk moves to a node that is connected to the previously visited node. This probability is controlled via the parameter q, where small values of q reduce the probability to visit a node that is a direct neighbor of the previously visited node, while large values of q increase this probability.
- For $p = q = 1$ we recover the uniform random walk used in DeepWalk, i.e. we can see DeepWalk as a special case of node2vec. For parameters $q \ll 1$ the random walks resemble a depth-first exploration of the network, i.e. we avoid neighbours of previously visited nodes, while for $q \gg$ the walks resemble a breadth-first search exploration. Taken together, the parameters p, q influence the node sequences generated by the random walk and thus **how local and non-local topological features are expressed in node-context pairs**.

Efficient sampling of biased random walks

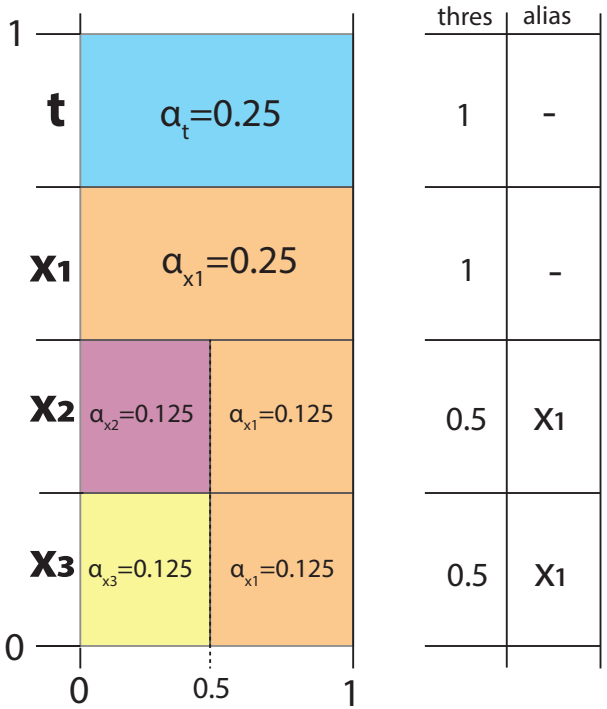
- to generate sequences, we start r random walks of length l for each node
- network with n nodes: $n \cdot r \cdot l$ randomly sampled neighbors
- we need $O(1)$ method to **non-uniformly sample from discrete set** with m elements



alias sampling → AJ Walker, 1977

initialization: generate threshold/alias table in $O(m \log m)$
sampling: i = randint(m)

```
if random() < thres[i]:
    return x[i]
else:
    return alias[i]
```



- use sampled walks to **generate node-context pairs**
- apply Skipgram to **generate embeddings**

Notes:

- To implement node2vec, we must efficiently sample walk sequences from a biased random walk model, i.e. different from DeepWalk we cannot choose neighbors uniformly at random (assuming an unweighted network). What is the computational complexity of the neighbor sampling used in random walk simulations. For a maximum-entropy, i.e. unbiased random walk we just need to draw a neighbor uniformly at random, which can be done in $O(1)$.
- We need a non-uniform sampling method with $O(1)$ complexity, which rules out methods that traverse all outcomes whenever we generate a sample. We can use the alias method to sample from a discrete distribution in $O(1)$. The details of this clever algorithm can be found in the reference above. It works as follows: For each outcome we define an interval from 0 to 1, which we use to sample based on a function that returns uniform random values from $(0, 1)$. For a set of options i with weights α_i a naive approach would be to first draw option i uniformly at random and then draw a random number $p \in (0, 1)$ uniformly. If $p < \frac{\alpha_i}{\max_j(\alpha_j)}$ we return outcome i, otherwise we repeat the procedure. While this will sample outcomes with desired probabilities, it has the problem that we may need to repeat the sampling to obtain a single sample. To avoid this, we can rearrange the sampling intervals such that we have at most two intervals associated with each outcome. For each outcome i we store a threshold value for p below which we return i, and above which we return a so-called alias.
- The alias method allows us to construct an alias table for each node in the network once, and then sample neighbors in the random walk simulation in $O(1)$. Once we have sampled our random walks, we generate node-context pairs and use them to train the SkipGram model, which yields the embeddings.

Practice Session

- we implement `node2vec` **in** `pytorch`
- we use `node2vec` to calculate node embeddings in empirical networks
- we explore **how hyperparameters** p, q **influence embeddings**

practice session

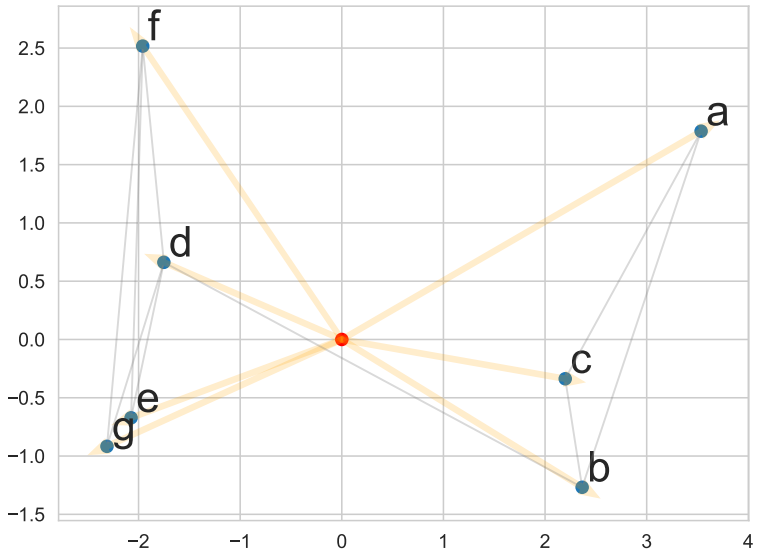
see notebook 10-03 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

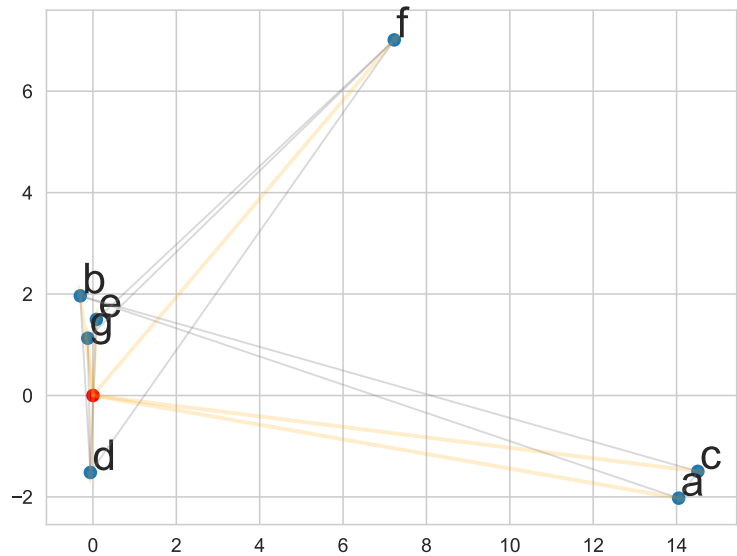
Notes:

- In the third practice session, we implement the random walk model that is at the heart of node2vec. We implement this based on a second-order graph model and use the SkipGram model to calculate node embeddings.
- We apply node2vec to empirical networks and explore how the two additional hyperparameters p, q influence the embeddings.

Example: DeepWalk vs. node2vec



DeepWalk embedding, walk length $l = 20$, window size $s = 2$



node2vec embedding for $p = 100$ and $q = 0.01$

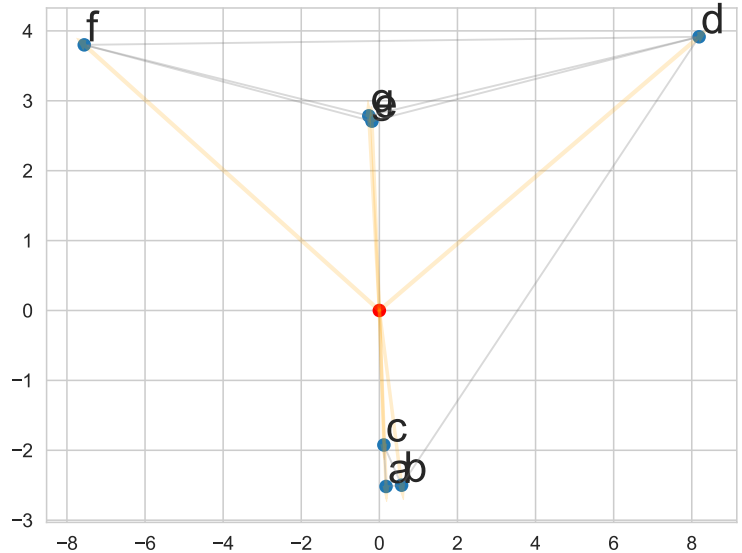
low return probability, strong exploration bias

walk length $l = 20$, window size $s = 2$

Notes:

- The figures above show a comparison of the embeddings generated for our toy example network by DeepWalk and node2vec. For $p = q = 1$ the embedding generated by node2ve is identical to that generated by DeepWalk. If we reduce the return probability and add a strong exploration bias, the node2vec embedding differs considerably from the DeepWalk embedding. In the example above, we find that nodes which are member of the same densely connected cluster are placed further apart, i.e. due to the strong exploration bias those nodes are considered less similar. The fact that the nodes b, e, g are considered similar in the generated embedding is likely due to the fact that - due to the low return probability and the strong exploration bias - a random walk starting in node b is virtually forced to move to nodes e, f or g after two steps (which is the window size used above). Similarly, all walks starting in e, f or g are forced to move to node b after two steps.

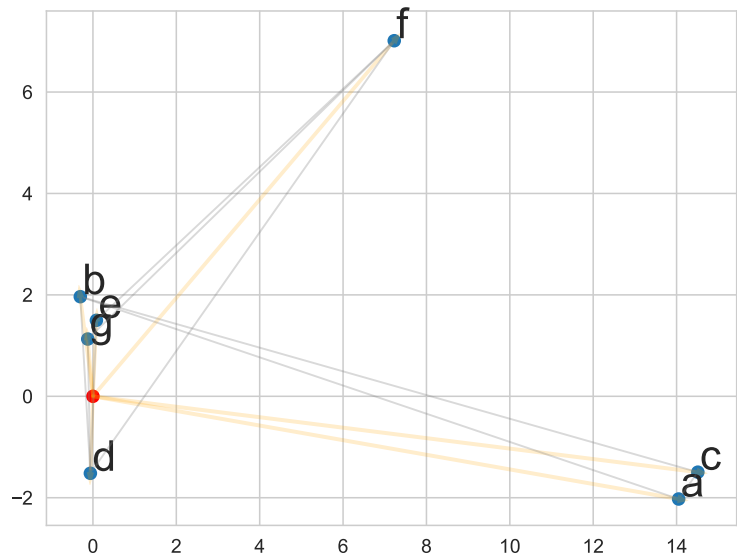
node2vec: Influence of p and q



node2vec embedding for $p = 100$ and $q = 100$

low return probability, strong exploitation bias

walk length $l = 20$, window size $s = 2$



node2vec embedding for $p = 100$ and $q = 0.01$

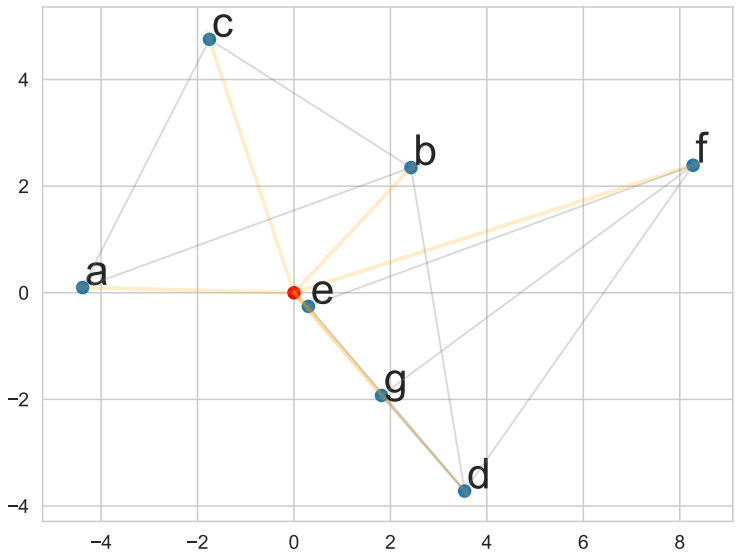
low return probability, strong exploration bias

walk length $l = 20$, window size $s = 2$

Notes:

- Above we show two more node2vec embeddings that we created for our toy example network using different values of p and q. For both cases, we used a large value of p (i.e. a low return probability). The embedding on the left was generated with a large value of q, which translates to a strong exploitation bias, i.e. walks are likely to remain in the local neighborhood. We find that nodes a, b, c are considered highly similar. Similarly, nodes e, g are considered highly similar, while nodes d and f are less similar to each other (and to e and g).
- The embedding on the right is the same as the one on the previous slide.

node2vec: Influence of p and q

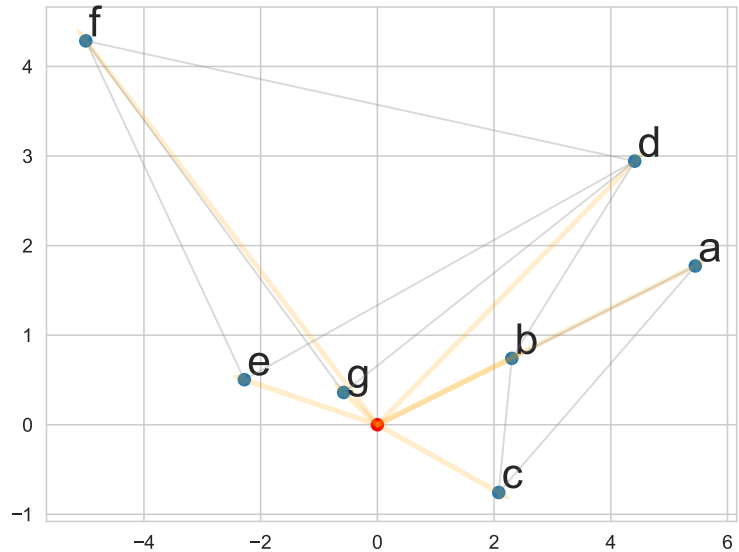


node2vec embedding for $p = 0.01$ and $q = 100$

high return probability

strong exploitation bias

walk length $l = 20$, window size $s = 2$



node2vec embedding for $p = 0.01$ and $q = 0.01$

high return probability

strong exploration bias

walk length $l = 20$, window size $s = 2$

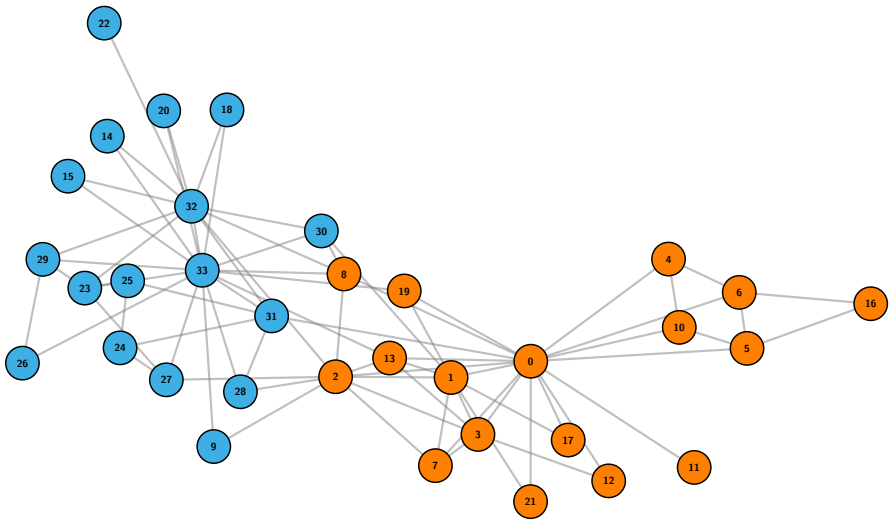
Notes:

- For the final two examples, we use a small value of p (i.e. a high return probability) and different values of q. We find that node vectors are generally less likely to be in close proximity, which is likely due to the fact that we have many more repetitions in the node sequences (i.e. due to the high return probability).

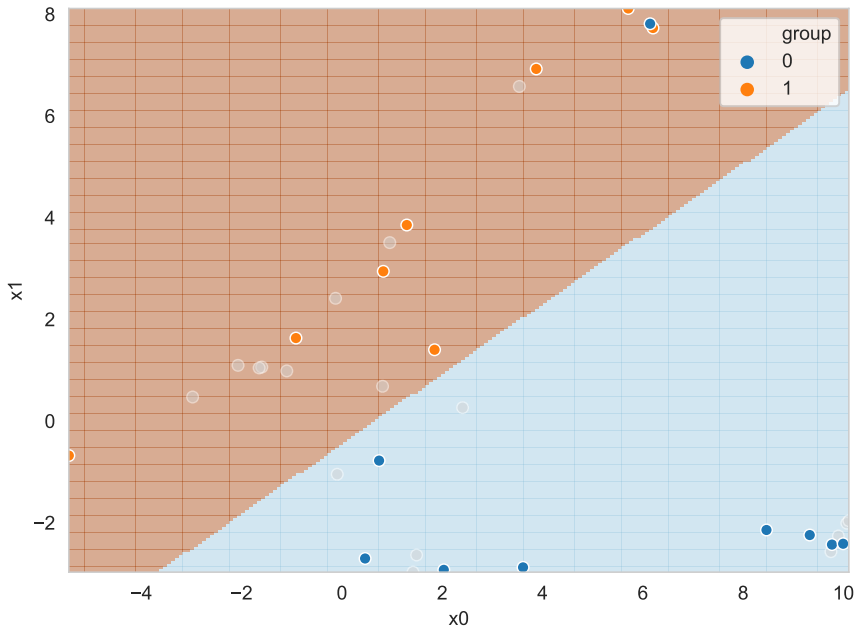
Node classification with DeepWalk

example

node classification in Karate club network with $n = 34$ nodes and $m = 77$ links, where ground truth node classes are given by groups



Karate club network with ground truth node classes



decision boundary of logistic regression for DeepWalk embedding (reduced to $\mathbb{R}^2$ via truncated SVD)
accuracy ≈ 0.94

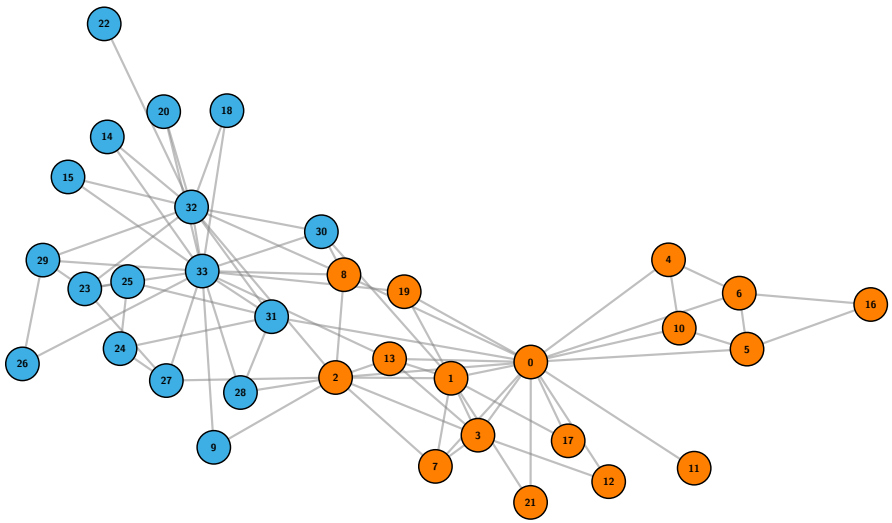
Notes:

- We conclude this lecture with an application of neural graph representation learning in node classification and link prediction. We first use DeepWalk to generate a node embedding and then use this embedding to train a logistic regression classifier. For the Karate club network, we observe an accuracy of 0.94 in the test set.

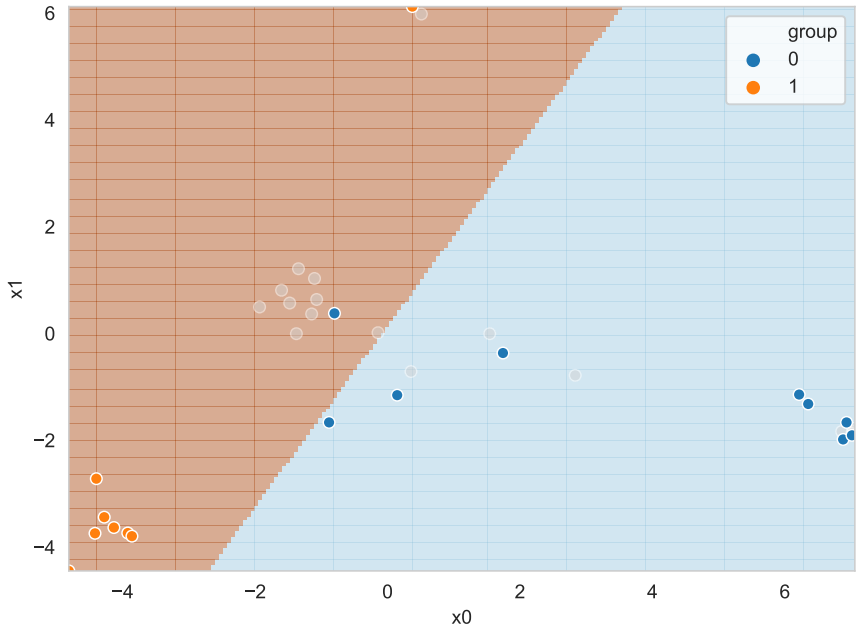
Node classification with node2vec

example

node classification in Karate club network with $n = 34$ nodes and $m = 77$ links, where ground truth node classes are given by groups



Karate club network with ground truth node classes



decision boundary of logistic regression for node2vec embedding ($p = 100$, $q = 0.5$) (reduced to $\mathbb{R}^2$ via truncated SVD) accuracy ≈ 0.94

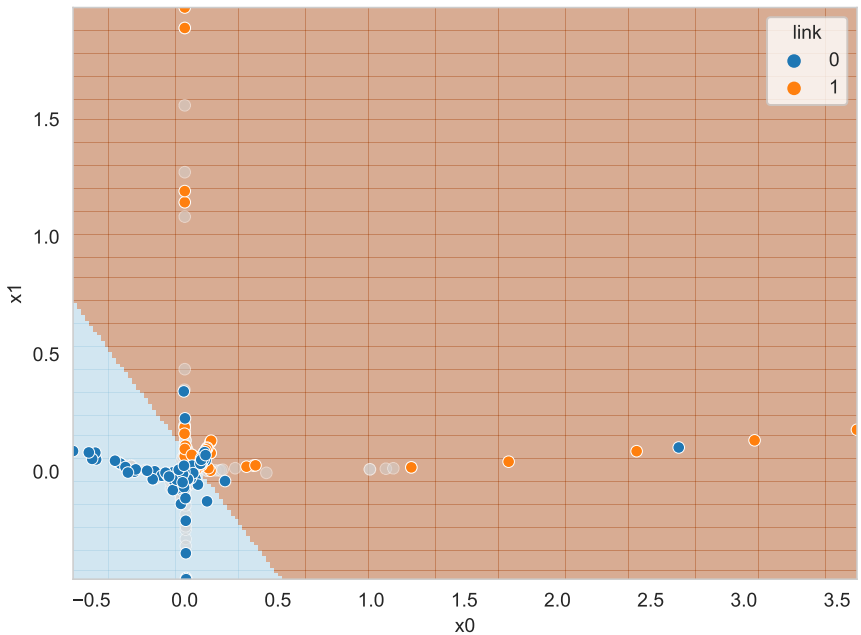
Notes:

- For node2vec we observe a similar result if we use hyperparameters $q = 100$ and $q = 0.5$. However, you will find that the choice of p and q can strongly influence the accuracy of the classification, i.e. we can get results that are worse than for the DeepWalk embedding. On the one hand, this need to choose suitable values for the hyperparameters can be problematic in real graph learning tasks. On the other hands, assuming we have sufficient amounts of data, we can also tune these hyperparameters to our specific network, which can both increase the performance of our classifier but also introduces an additional risk of overfitting.

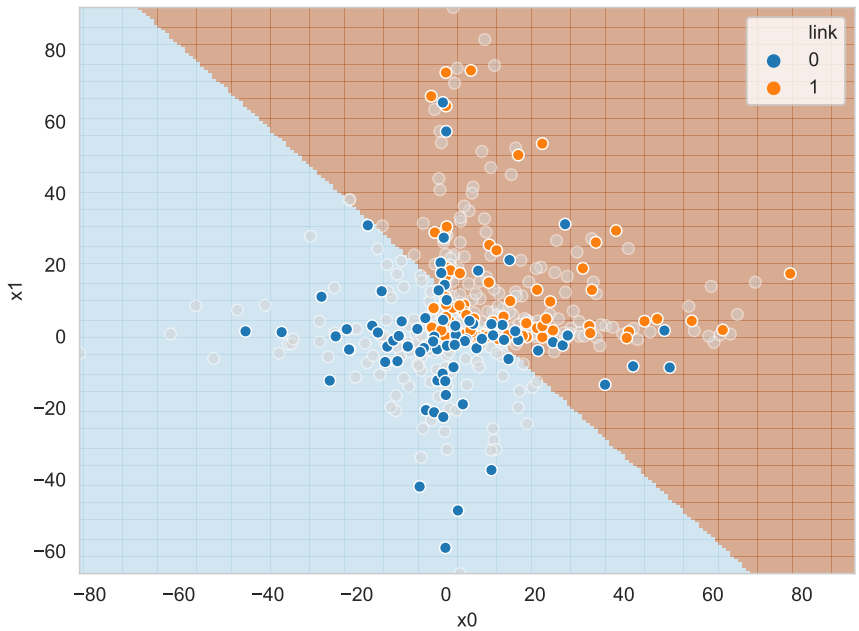
Link prediction with DeepWalk

empirical example

- network of homework cooperations between students at Ben-Gurion University
- $n = 185$ nodes, $m = 311$ links
- E_{train} = 90 % of links



decision boundary of logistic regression using Hadamard product of **Laplacian embedding** in balanced test set (accuracy = 0.76)



decision boundary of logistic regression using Hadamard product of **DeepWalk embedding** in balanced test set (accuracy = 0.72)

Notes:

- The example above shows that neural graph representations do not necessarily lead to better performance than the graph factorization techniques introduced in Lecture 08.

Neural Graph Representation Learning

- large number of **neural graph representation learning** methods proposed in past years

exemplary methods

- DeepWalk → B Perozzi, R Al-Rfou, KDD 2014
- LINE → J Tang et al., WWW 2015
- PTE → J Tang et al., KDD 2015
- node2vec → A Grover, J Leskovec, KDD 2016
- SDNE → D Wang, P Cui, W Zhu, KDD 2016
- GraphSAGE → W Hamilton, Z Ying, J Leskovec, NeurIPS 2017

- some methods **implicitly factorize matrix representations** of networks

- embeddings generated by DeepWalk are closely related to **Laplacian Eigenmaps**

→ W Hamilton, 2020

Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec

Jiezhong Qiu[†*], Yuxiao Dong[‡], Hao Ma[‡], Jian Li[♯], Kuansan Wang[‡], and Jie Tang[†]

[†]Department of Computer Science and Technology, Tsinghua University
[‡]Microsoft Research, Redmond
[♯]Institute for Interdisciplinary Information Sciences, Tsinghua University
qiujz16@mails.tsinghua.edu.cn,{yuxdong,haoma,kuansanw}@microsoft.com,{lijian83,jietang}@tsinghua.edu.cn

ABSTRACT

Since the invention of word2vec [28, 29], the skip-gram model has significantly advanced the research of network embedding, such as the recent emergence of the DeepWalk, LINE, PTE, and node2vec approaches. In this work, we show that all of the aforementioned models with negative sampling can be unified into the matrix factorization framework with closed forms. Our analysis and proofs reveal that: (1) DeepWalk [31] empirically produces a low-rank transformation of a network's normalized Laplacian matrix; (2) LINE [37], in theory, is a special case of DeepWalk when the size of vertices' context is set to one; (3) As an extension of LINE, PTE [36] can be viewed as the joint factorization of multiple networks' Laplacians; (4) node2vec [16] is factorizing a matrix related to the stationary distribution and transition probability tensor of a 2nd-order random walk. We further provide the theoretical connections between skip-gram based network embedding algorithms and the theory of graph Laplacian. Finally, we present the NetMF method[1] as well as its approximation algorithm for computing network embedding. Our method offers significant improvements over DeepWalk and LINE for conventional network mining tasks. This work lays the theoretical foundation for skip-gram based network embedding methods, leading to a better understanding of latent network representation learning.

ACM Reference Format:
Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2018. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec . In *Proceedings of WSDM'18.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3159652.3159706

1 INTRODUCTION

The conventional paradigm of mining and learning with networks usually starts from the explicit exploration of their structural properties [13, 32]. But many of such properties, such as betweenness centrality, triangle count, and modularity, require extensive domain knowledge and expensive computation to handcraft. In light of these issues, as well as the opportunities offered by the recent emergence of representation learning [2], learning latent representations for networks, a.k.a., network embedding, has been extensively studied in order to automatically discover and map a network's structural properties into a latent space.

Formally, the problem of network embedding is often formalized as follows: Given an undirected and weighted graph $G = (V, E, A)$ with V as the node set, E as the edge set and A as the adjacency matrix, the goal is to learn a function $V \to \mathbb{R}^d$ that maps each vertex to a d-dimensional ($d \ll |V|$) vector that captures its structural properties. The output representations can be used as the input of mining and learning algorithms for a variety of network science tasks, such as label classification and community detection.

The attempt to address this problem can date back to spectral graph theory [11] and social dimension learning [38]. Its very recent advances have been largely influenced by the skip-gram model originally proposed for word embedding [28, 29], whose input is a text corpus composed of sentences in natural language and output is the latent vector representation for each word in the corpus. Notably, inspired by this setting, DeepWalk [31] pioneers network embedding by considering the vertex paths traversed by random walks over networks as the sentences and leveraging skip-gram for learning latent vertex representations. With the advent of

Table 1: The matrices that are implicitly approximated and factorized by DeepWalk, LINE, PTE, and node2vec.

Algorithm	Matrix
DeepWalk	$\log\left(\mathrm{vol}(G)\left(\frac{1}{T}\sum_{r=1}^{T}(D^{-1}A)^r\right)D^{-1}\right) - \log b$
LINE	$\log(\mathrm{vol}(G)D^{-1}AD^{-1}) - \log b$
PTE	$\log\left(\begin{pmatrix}\alpha\,\mathrm{vol}(G_{ww})(D_{\mathrm{row}}^{ww})^{-1}A_{ww}(D_{\mathrm{col}}^{ww})^{-1}\\ \beta\,\mathrm{vol}(G_{dw})(D_{\mathrm{row}}^{dw})^{-1}A_{dw}(D_{\mathrm{col}}^{dw})^{-1}\\ \gamma\,\mathrm{vol}(G_{tw})(D_{\mathrm{row}}^{tw})^{-1}A_{tw}(D_{\mathrm{col}}^{tw})^{-1}\end{pmatrix}\right) - \log b$
node2vec	$\log\left(\frac{\frac{1}{2T}\sum_{r=1}^{T}\left(\sum_{u}X_{w,u}\underline{P}_{w,u}^{r}+\sum_{u}X_{c,u}\underline{P}_{w,c,u}^{r}\right)}{\left(\sum_{u}X_{w,u}\right)\left(\sum_{u}X_{c,u}\right)}\right) - \log b$

Notations in DeepWalk and LINE are introduced below. See detailed notations for PTE and node2vec in Section 2.
$\boldsymbol{A}: \boldsymbol{A} \in \mathbb{R}_{+}^{|V|\times|V|}$ is G's adjacency matrix with $\boldsymbol{A}_{i,j}$ as the edge weight between vertices i and j;
D_{col}: $D_{\mathrm{col}} = \mathrm{diag}(\boldsymbol{A}^\top \boldsymbol{e})$ is the diagonal matrix with column sum of $\boldsymbol{A}$;
D_{row}: $D_{\mathrm{row}} = \mathrm{diag}(\boldsymbol{A}\boldsymbol{e})$ is the diagonal matrix with row sum of $\boldsymbol{A}$;
$\boldsymbol{D}$: For undirected graphs ($\boldsymbol{A}^\top = \boldsymbol{A}$), $D_{\mathrm{col}} = D_{\mathrm{row}}$. For brevity, $\boldsymbol{D}$ represents both D_{col} & D_{row}.
$D = \mathrm{diag}(d_1, \cdots, d_{|V|})$, where d_i represents generalized degree of vertex i;
$\mathrm{vol}(G)$: $\mathrm{vol}(G) = \sum_i \sum_j A_{i,j} = \sum_i d_i$ is the volume of a weighted graph G;
T & b: The context window size and the number of negative sampling in skip-gram, respectively.

[*]This work was partially done when Jiezhong was an intern at Microsoft Research.
[1]Code available at github.com/xptree/NetMF.

WSDM'18, , February 5–9, 2018, Marina Del Rey, CA, USA

ACM ISBN 978-1-4503-5581-0/18/02…$15.00
https://doi.org/10.1145/3159652.3159706

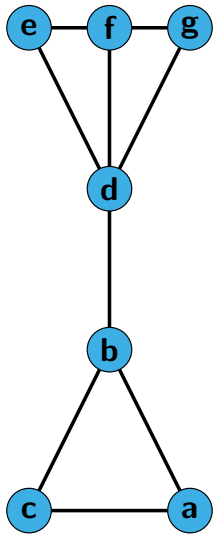
→ J Qiu et al., 2018

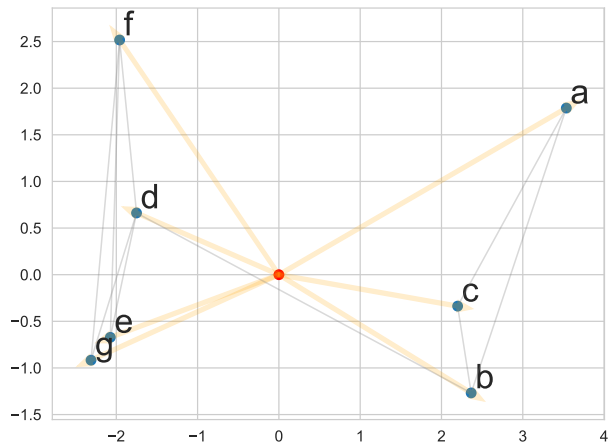
Notes:

- We close this lecture with a general comment about neural graph representation learning techniques. Following the success of DeepWalk and node2vec, a large number of neural graph representation learning techniques are being proposed in recent years. In today's lecture, we have seen why the weight parameters of a multi-layer perceptron model can be viewed as a low-dimensional factorization of an input matrix, subject to a loss function.
- This should remind us of the loss functions that are implicitly minimized by the matrix factorization techniques introduced in Lecture 08, e.g. the minimization of euclidean distance between connected nodes, which is minimized in the Laplacian eigenmap approach.
- Interestingly, in a recent work it has been shown that a several popular neural graph representation learning techniques actually factorize different matrix representations of a graph. Moreover, it can be shown that the specific embedding generated by DeepWalk is actually closely related to the Laplacian embedding introduced in lecture L08. This can be seen as a possible explanation for the results on slide 27, where we found that a DeepWalk embedding does not perform better than a simple Laplacian embedding. An advantage of the neural network approach to matrix factorization is the ability to scale it to networks millions of nodes, by means of using modern GPU acceleration.

Conclusion

- we introduced **neural representation learning** techniques
- **SkipGram model** developed in the context of **natural language processing**
- perspective 1: **multinomial logistic regression** to predict context word given one-hot encoding of center word
- perspective 2: **approximate neural matrix factorization** minimizing cross entropy loss
- we can apply SkipGram model to **sequences of nodes generated by (biased) random walks**
- DeepWalk and node2vec $\sim$ neural matrix factorization $\rightarrow$ J Qiu et al, 2018





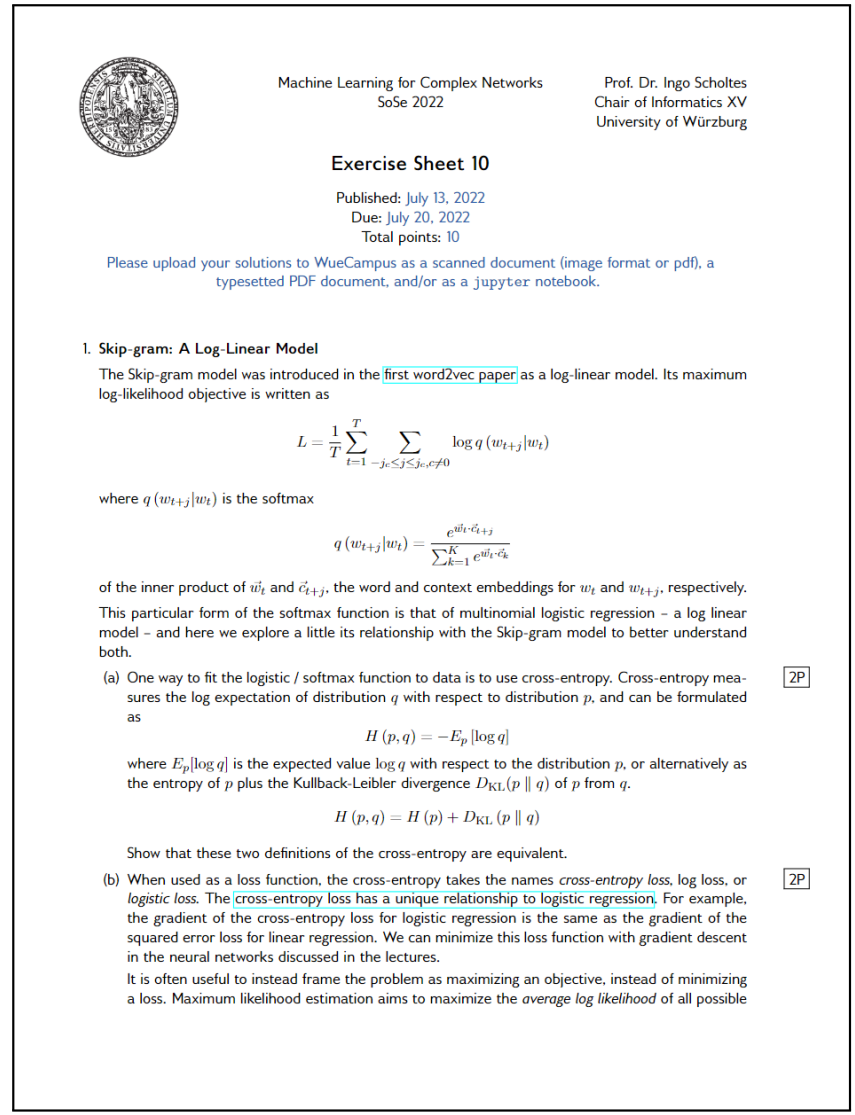
DeepWalk embedding of example network

Notes:

- In summary, we introduced neural graph representation learning techniques, which were one of the first applications of neural networks to graph learning. We presented two walk-based embedding techniques, DeepWalk and node2vec, which are based on the SkipGram model that was originally developed in the context of natural language processing.
- We can actually take two perspectives on this technique. The first perspective is based on the fact that we are using gradient descent to fit a log-linear model for the probabilities of context words, i.e. we effectively use an approach that is very similar to multinomial logistic regression. The second perspective is based on the fact that we can see the two-layer neural network architecture of SkipGram as an implicit matrix factorization that minimizes the cross entropy loss function. In this sense, neural graph representation learning is closely related to the embedding techniques introduced in Lecture 08, with the difference that we cannot compute the factorization based on an eigenvector calculation.
- We have seen that we can naturally apply the SkipGram model to sequences of nodes generated by a random walk model. The proposed method DeepWalk was one of the first **neural graph representation learning techniques**. We have further discussed that we can tune this random walk model to obtain embeddings that can be controlled based on additional hyperparameters that guide the exploration-exploitation behavior of random walks.

Exercise sheet

- **exercise sheet** will be released today
  - explore neural matrix factorization
  - apply deep graph representation learning
- solutions are due **July 10th** (via Moodle)
- present your solution to earn bonus points



Machine Learning for Complex Networks
SoSe 2022

Prof. Dr. Ingo Scholtes
Chair of Informatics XV
University of Würzburg

Exercise Sheet 10

Published: July 13, 2022
Due: July 20, 2022
Total points: 10

Please upload your solutions to WueCampus as a scanned document (image format or pdf), a typesetted PDF document, and/or as a `jupyter` notebook.

1. **Skip-gram: A Log-Linear Model**

The Skip-gram model was introduced in the first word2vec paper as a log-linear model. Its maximum log-likelihood objective is written as

$$L = \frac{1}{T}\sum_{t=1}^{T}\sum_{-j_c \le j \le j_c, c \neq 0} \log q\left(w_{t+j}|w_t\right)$$

where $q\left(w_{t+j}|w_t\right)$ is the softmax

$$q\left(w_{t+j}|w_t\right) = \frac{e^{\vec{w}_t \cdot \vec{c}_{t+j}}}{\sum_{k=1}^{K} e^{\vec{w}_t \cdot \vec{c}_k}}$$

of the inner product of $\vec{w}_t$ and $\vec{c}_{t+j}$, the word and context embeddings for w_t and w_{t+j}, respectively.

This particular form of the softmax function is that of multinomial logistic regression – a log linear model – and here we explore a little its relationship with the Skip-gram model to better understand both.

(a) One way to fit the logistic / softmax function to data is to use cross-entropy. Cross-entropy measures the log expectation of distribution q with respect to distribution p, and can be formulated as 2P

$$H\left(p, q\right) = -E_p\left[\log q\right]$$

where $E_p[\log q]$ is the expected value $\log q$ with respect to the distribution p, or alternatively as the entropy of p plus the Kullback-Leibler divergence $D_{\mathrm{KL}}(p \parallel q)$ of p from q.

$$H\left(p, q\right) = H\left(p\right) + D_{\mathrm{KL}}\left(p \parallel q\right)$$

Show that these two definitions of the cross-entropy are equivalent.

(b) When used as a loss function, the cross-entropy takes the names *cross-entropy loss*, log loss, or *logistic loss*. The cross-entropy loss has a unique relationship to logistic regression. For example, the gradient of the cross-entropy loss for logistic regression is the same as the gradient of the squared error loss for linear regression. We can minimize this loss function with gradient descent in the neural networks discussed in the lectures. 2P

It is often useful to instead frame the problem as maximizing an objective, instead of minimizing a loss. Maximum likelihood estimation aims to maximize the *average log likelihood* of all possible

Notes:

Self-Study Questions

1. What is the distributional hypothesis in linguistics and how we can use to embed words in a vector space?
2. Explain how we can use neural networks to find approximate factorizations of matrices.
3. Explain whether the power law distribution of degrees is likely to affect node embeddings generated by DeepWalk.
4. Investigate the concept of negative sampling in the context of word2vec and explain how it is used to speed up the training process.
5. Consider the activation and loss function used in the SkipGram model of word2vec. Explain how the SkipGram model is related to maximum likelihood estimation of a multinomial logistic regression model.
6. What is the difference between DeepWalk and node2vec? Explain the influence of the parameters p and q in node2vec.
7. What is alias sampling and how can we use it to efficiently sample large numbers of biased random walks.
8. How are DeepWalk and node2vec related to the representation learning techniques introduced in Lecture 08.

Notes:

References

reading list

- JR Firth: **A synopsis of linguistic theory 1930-1955**, Studies in Linguistic Analysis: 1–32, 1957
- AJ Walker: **An efficient method for generating discrete random variables with general distributions**, Electronics Letters 10, 1977
- MD Vose: **A Linear Algorithm for Generating Random Numbers With a Given Distribution**, IEEE Trans. on Software Engineering, 1991
- T Mikolov, K Chen, G Corrado, J Dean: **Efficient Estimation of Word Representations in Vector Space**, arXiv 1301.3781, 2013
- T Mikolov, I Sutskever, K Chen, G Corrado, J Dean : **Distributed Representations of Words and Phrases and their Compositionality**, NeurIPS 2013
- B Perozzi, R Al-Rfou, S Skiena: **DeepWalk: Online Learning of Social Representations**, KDD'14, 2014
- A Grover, J Leskovec: **node2vec: Scalable Feature Learning for Networks**, KDD'16, 2016
- WL Hamilton, R Ying, J Leskovec: **Representation Learning on Graphs: Methods and Applications**, Bulletin of the IEEE CS Technical Committee on Data Engineering, 2017
- J Qiu et al. **Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec**, Proc. of WSDM, 2018
- WL Hamilton: **Graph Representation Learning**, Synthesis Lectures on AI and ML, 2020

DeepWalk: Online Learning of Social Representations

Bryan Perozzi
Stony Brook University
Department of Computer Science

Rami Al-Rfou
Stony Brook University
Department of Computer Science

Steven Skiena
Stony Brook University
Department of Computer Science

{bperozzi, ralrfou, skiena}@cs.stonybrook.edu

ABSTRACT

We present DeepWalk, a novel approach for learning latent representations of vertices in a network. These latent representations encode social relations in a continuous vector space, which is easily exploited by statistical models. DeepWalk generalizes recent advancements in language modeling and unsupervised feature learning (or *deep learning*) from sequences of words to graphs.

DeepWalk uses local information obtained from truncated random walks to *learn* latent representations by treating walks as the equivalent of sentences. We demonstrate DeepWalk's latent representations on several multi-label network classification tasks for social networks such as BlogCatalog, Flickr, and YouTube. Our results show that DeepWalk outperforms challenging baselines which are allowed a global view of the network, especially in the presence of missing information. DeepWalk's representations can provide F_1 scores up to 10% higher than competing methods when labeled data is sparse. In some experiments, DeepWalk's representations are able to outperform all baseline methods while using 60% less training data.

DeepWalk is also scalable. It is an online learning algorithm which builds useful incremental results, and is trivially parallelizable. These qualities make it suitable for a broad class of real world applications such as network classification, and anomaly detection.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications - Data Mining; I.2.6 [**Artificial Intelligence**]: Learning; I.5.1 [**Pattern Recognition**]: Model - Statistical

1. INTRODUCTION

The sparsity of a network representation is both a strength and a weakness. Sparsity enables the design of efficient discrete algorithms, but can make it harder to generalize in statistical learning. Machine learning applications in networks (such as network classification [15, 37], content recommendation [11], anomaly detection [5], and missing link prediction [22]) must be able to deal with this sparsity in order to survive.

In this paper we introduce *deep learning* (unsupervised feature learning) [2] techniques, which have proven successful in natural language processing, into network analysis for the first time. We develop an algorithm (DeepWalk) that learns *social representations* of a graph's vertices, by modeling a stream of short random walks. Social representations are latent features of the vertices that capture neighborhood similarity and community membership. These latent representations encode social relations in a continuous vector space with a relatively small number of dimensions. DeepWalk generalizes neural language models to process a special language composed of a set of randomly-generated walks. These neural language models have been used to capture the semantic and syntactic structure of human language [6], and even logical analogies [28].

DeepWalk takes a graph as input and produces a latent representation as an output. The result of applying our method to the well-studied Karate network is shown in Figure 1. The graph, as typically presented by force-directed layouts, is shown in Figure 1a. Figure 1b shows the output of our method with 2 latent dimensions. Beyond the striking similarity, we note that linearly separable portions of (1b) correspond to clusters found through modularity maximization in the input graph (1a) (shown as vertex colors).

To demonstrate DeepWalk's potential in real world sce-

(a) Input: Karate Graph (b) Output: Representation

Figure 1: Our proposed method *learns* a latent space representation of social interactions in $\mathbb{R}^d$. The learned representation encodes community structure so it can be easily exploited by standard classification methods. Here, our method is used on Zachary's Karate network [44] to generate a latent representation in $\mathbb{R}^2$. Note the correspondence between community structure in the input graph and the embedding. Vertex colors represent a modularity-based clustering of the input graph.

Notes: