

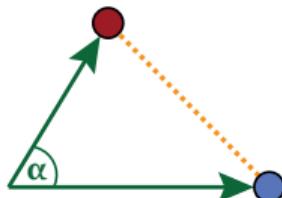
## 10. Graph Representation Learning

## 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

- ▶ many machine learning techniques assume **Euclidean feature spaces**, e.g.  $x_i \in \mathbb{R}^d$
- ▶  $d$ -dimensional Euclidean space is **metric space** with **Euclidean distance** metric
- ▶ Euclidean vector space =  $d$ -dimensional **inner product space** over  $\mathbb{R}$



$$\|\vec{y} - \vec{x}\| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

$$\vec{x} \cdot \vec{y} = x_1 y_1 + x_2 y_2 = \|\vec{x}\| \|\vec{y}\| \cos \alpha$$



Euclid of Alexandria as depicted in the fresco  
“The School of Athens”  
born ca. 325 BC

image credit: public domain

## Notes:

- You are probably familiar with basic machine learning algorithms like SVM, logistic regression,  $k$ -nearest-neighbor classification, or  $k$ -means clustering. Such general-purpose techniques often assume a Euclidean feature space, e.g.  $d$ -dimensional feature vectors  $\mathbb{R}^d$ . Such a Euclidean space has special properties that are used by many machine learning techniques.
- A  $d$ -dimensional Euclidean space with the Euclidean distance metric

$$dist(\vec{x}, \vec{y}) := \|\vec{y} - \vec{x}\| = \sqrt{\sum_{i=1}^d (x_i - y_i)^2}$$

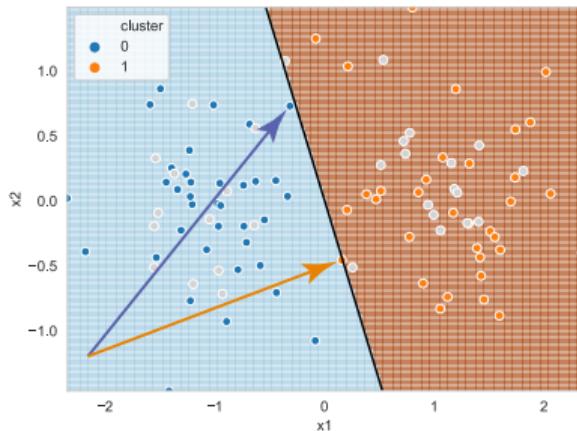
is a prototypical example of a **metric space**, where the Euclidean distance fulfills the three axioms of a metric (identity of indiscernibles, symmetric, triangle inequality). Many machine learning techniques that calculate distances between data points implicitly use the resulting properties of a metric space.

- Using the dot product

$$\vec{x} \cdot \vec{y} = x_1 y_1 + x_2 y_2 = \|\vec{x}\| \|\vec{y}\| \cos \alpha$$

the Euclidean space fulfills the properties of an **inner product space**. The dot product can be viewed as a measure of “similarity” between data points, where the similarity depends both on the length of those vectors and the angle  $\alpha$  between them.

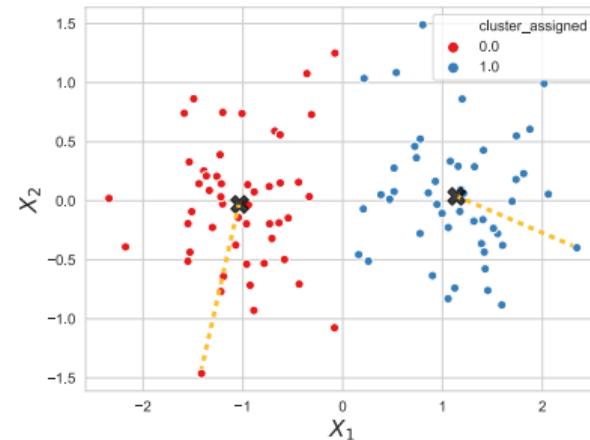
# Machine Learning in Euclidean data



## example method: support vector machine (SVM)

find  $d = 1$ -dim. hyperplane that separates classes such that margin of decision boundary is maximized → BSc Lecture: AKIDS1

**dot product** between  $x_i \in \mathbb{R}^2$ , i.e. we use property of **inner product space**



## example method: *k*-means clustering

assign points to  $k$  clusters, such that squared distance of points to closest cluster center is minimized → BSc Lecture: AKIDS1

**distance** between  $x_i \in \mathbb{R}^2$ , i.e. we use property of **metric space**

## Notes:

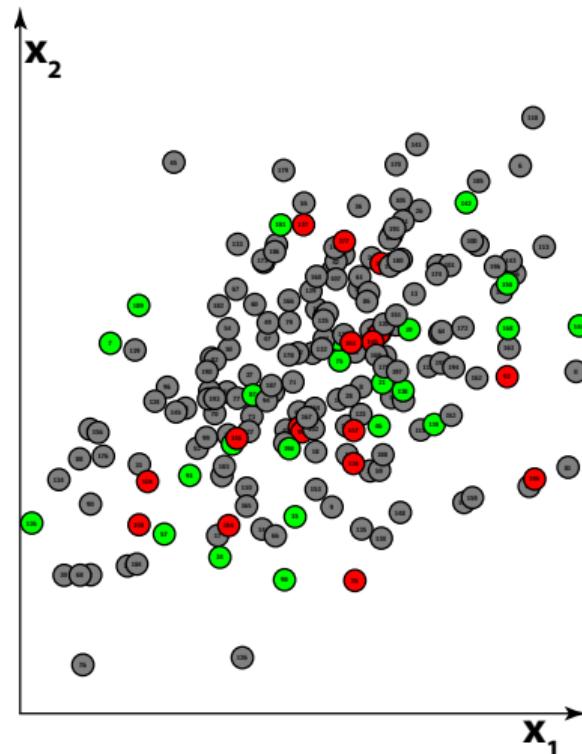
- Let us have a look at the two exemplary applications of machine learning from before.
- $k$ -means clustering seeks to minimize the squared distance between the cluster centers and the data points assigned to the closest cluster center. Due to the use of a distance measure, this algorithm implicitly assume that the feature space fulfills **metric** properties.
- Other machine learning techniques are based on “similarities” between feature vectors that can be calculated based on a dot product between vectors (e.g. in Euclidean space this yields the cosine of the angle between vectors, which is zero if they are orthogonal). This dot product is directly used by a support vector machine (which is named after support vectors that are orthogonal to the decision boundary). With this, we use the fact that our feature space is an **inner product space**
- Note that the property of an inner product space is more general than a metric space. We can use vector norms to obtain a metric space, but we can define inner product spaces that are non-Euclidean. Consider, for instance support vector machines. Using the kernel trick, we can define an inner product that implicitly maps features into a (high-dimensional) inner product space. Consider, e.g., the radial basis function kernel that helps us to fit linear decision boundaries in a high-dimensional inner product space, which correspond to a complex non-linear decision boundaries in our original Euclidean space.

# A more complex example

- ▶ consider synthetic data  $x_i \in \mathbb{R}^2$  with classes  $y_i \in \mathbb{C} = \{0, 1\}$
- ▶ split data in **training and test set**
- ▶ train **support vector machine** with **non-linear RBF kernel**

## example: kernel SVM

- ▶ classification accuracy  $\approx 0.57$
- ▶ kernel SVM in Euclidean feature space unable to classify data
- ▶ problem: data often exhibits **complex relationships** not captured by Euclidean features



correctly and wrongly classified data points using **SVM with radial kernel** (accuracy = 0.57)

## Notes:

- Despite the remarkable successes of machine learning in recent years, there are application domains where general purpose machine learning techniques reach their limits. This is due to the fact that most standard machine learning techniques assume data in a Euclidean domain. For many real-world data sets, this is not sufficient to capture the complex patterns that enable us to address supervised or unsupervised learning tasks.
- To give a concrete example, consider the synthetic data set shown on the right, which contains two (Euclidean) features  $X_1$  and  $X_2$  and binary classes 0 and 1.
- Those classes are clearly not separable by a simple linear model, which is why we adopt a kernel SVM with a non-linear radial basis function kernel. We train the SVM in a training set and then apply it to a test set. We find that the classification accuracy in this example is very low, approx. 0.57. Since we have balanced classes, this is not much better than a random guess.
- You could now think that this bad performance is simply due to the absence of a pattern that can be learned by the model. Such a pattern exists in this data set, but we cannot capture this in a simple two-dimensional Euclidean space.
- The point is that the observations in this data set, like many other real-world data, exhibits complex relationships that are not captured by the Euclidean features.

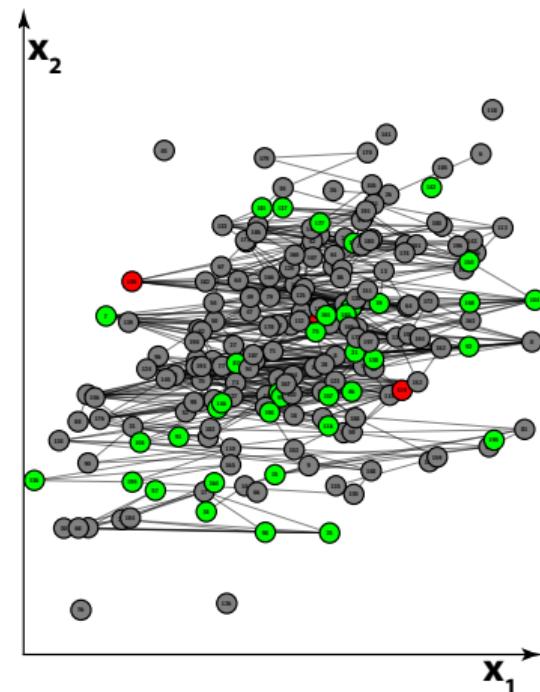
# Learning in graph-structured data?

## definition

- ▶ **graph or network**  $G = (V, E)$  consists of a set of nodes (vertices)  $V$  and links (edges)  $E \subseteq V \times V$
- ▶ nodes can have features/classes, e.g.  $x : V \rightarrow \mathbb{R}^d$  and  $y : V \rightarrow \mathbb{C}$
- ▶ how can we apply machine learning to such **graph-structured data?**

## exemplary approaches

1. apply statistical learning to **ensembles of networks** → statistical network analysis
2. map graph to **vectorial feature space**  
→ graph embedding techniques
3. machine learning for graphs with **additional** vectorial features  
→ graph neural networks



correctly and **wrongly** classified nodes using **graph neural network** (accuracy = 0.93)

## Notes:

- We often have additional information on relationships that can not be represented in a (low-dimensional) Euclidean space. In our example, additional relationships exist between data points, which can be represented as links in a graph.
- These relationships contain a pattern that we can use for machine learning. To highlight those patterns, we can assign positions to nodes based on a **force-directed layout algorithm** (rather than assigning positions based on Euclidean features of nodes). This shows that the graph topology constitutes an independent dimension of information on our problem.
- Note that graphs are one of the most expressive data structures. Interestingly, most machine learning problems can be viewed as special cases of graph-structured problems (e.g. features in a Euclidean space, image data) but not the other way round. There are different approaches to leverage this (additional) graph structure:
- First, we can model patterns based on **statistical models for graphs** like those discussed in our course *Statistical Network Analysis*. This approach yields statistical learning techniques that we discuss in a first chapter of the course.
- Second, we could embed the nodes in the graph into a (Euclidean) feature space and then apply standard machine learning algorithms. This embedding should preserve the information/patterns contained in the topology. We will discuss such **graph embedding** techniques in a second chapter.
- A third approach uses new techniques that leverage both (vectorial) features of nodes/links as well as the graph topology. **Graph neural networks** are one important way to achieve this, which we will discuss in a final chapter of the course.

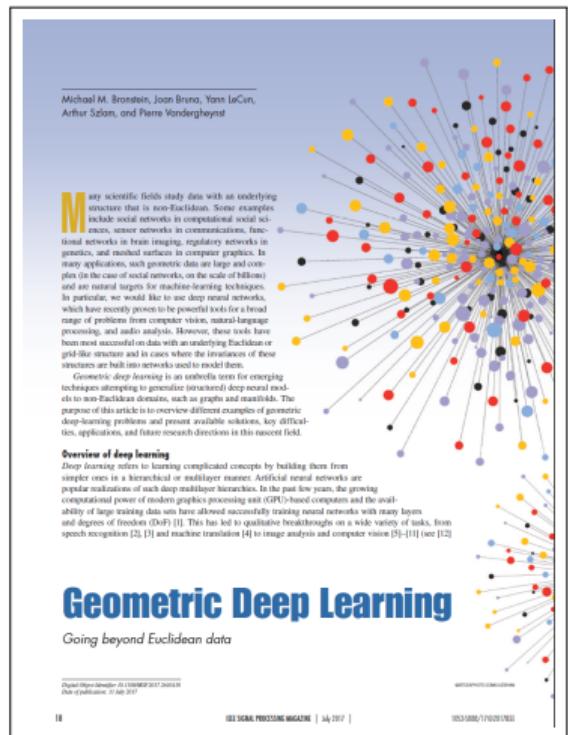
# Geometric Machine Learning

- ▶ graph learning = special class of **geometric machine learning techniques** for non-Euclidean domains

“Geometric deep learning is an umbrella term for emerging techniques attempting to generalize (structured) deep neural network models to non-Euclidean domains, such as graphs and manifolds.” → Michael M Bronstein et al., 2017

## geometric learning domains

- ▶ 3D molecular structures
- ▶ two-dimensional image/video data
- ▶ tree structures (e.g. knowledge graphs/phylogenetics)
- ▶ non-Euclidean manifolds (e.g. surface of 3D shape)



Michael M. Bronstein, Joan Bruna, Yann LeCun,  
Arthur Szlam, and Pierre Vandergheynst

**M**any scientific fields study data with an underlying structure that is non-Euclidean. Some examples are social networks in communication, protein structures, sensor networks in communications, functional networks in brain imaging, regulatory networks in genetics, and meshed surfaces in computer graphics. In many applications, such problems data are large and complex enough that traditional machine learning methods can no longer be applied. This motivates a need for new and more natural inpers for machine learning techniques. In particular, we would like to use deep neural networks, which have recently proven to be powerful tools for a broad range of problems from computer vision, natural-language processing, and audio analysis. However, these networks have been mainly designed for data with an underlying Euclidean or grid-like structure and in cases where the irregularities of these structures are built into networks used to model them.

Geometric deep learning is an umbrella term for emerging techniques attempting to generalize (structured) deep neural networks to non-Euclidean domains, such as graphs and manifolds. The purpose of this article is to overview different examples of geometric deep-learning problems and present available solutions, key difficulties, applications, and future research directions in this nascent field.

**Overview of deep learning**

Deep learning refers to learning complicated concepts by building them from simpler ones in a hierarchical or multi-layer manner. Artificial neural networks are popular realizations of such deep multilayer hierarchies. In the past few years, the growing computational power of modern graphics processing unit (GPU)-based computers and the availability of large amounts of training data have led to significant breakthroughs in learning and degrees of freedom (DoFs) [1]. This has led to qualitative breakthroughs on a wide variety of tasks, from speech recognition [2], [3] and machine translation [4] to image analysis and computer vision [5]–[11] (see [12]

## Geometric Deep Learning

Going beyond Euclidean data

Digital Object Identifier 10.1109/MSP.2017.2640510  
Date of publication: 31 July 2017

II IEEE SIGNAL PROCESSING MAGAZINE | July 2017 | 1053-5886/17/070201-03

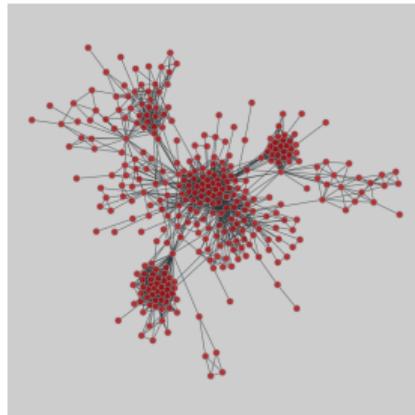
→ MM Bronstein et al. IEEE Sig. Proc. Mag., 2017

## Notes:

- Machine learning techniques for graph-structured data are a special class of **geometric machine learning techniques** for non-Euclidean domains, i.e. for data that exhibits structures that cannot be represented in a simple vector space.
- Data with graph structures are only one example for non-Euclidean data. Other examples include three-dimensional molecular structures, where atoms can exhibit complex properties, bindings, and relationships based on the laws of physics. Network models of molecular are one increasingly popular approach to address machine learning problems in such data.
- Image or video data are another example for data that require specialized machine learning methods that account for the special structure of pixel data arranged in a lattice. Note that we can again think of an image as a lattice network of pixels, where adjacent pixel coordinates (e.g. at Hamming distance one) are connected by an edge.
- We also often have data that can naturally be arranged in a hierarchy or tree structure. Consider, e.g. certain types of knowledge graphs or trees formed by genetic inheritance in phylogenetics. We note that such trees are naturally represented as a special type of graphs that exhibit no cycles.
- Finally, an important example for non-Euclidean data are all sorts of manifolds, e.g. surfaces of higher-dimensional objects. Such manifolds locally resemble a Euclidean space, but have a more complex topology. Consider, e.g. a machine learning problem that can be mapped to the surface of the 3D rabbit above. Interestingly, we can again map such problems to a graph topology, which underpins the fact that they provide us with a very general approach to study problems in non-Euclidean domains.

# Example 1: Social Networks

consider an **online social network**, where nodes represent **users** and links represent **friend/follower relationships**



## question

can we automatically recommend interesting accounts to (new) users?

## graph learning task

we can cast this as an **link prediction** problem  
→ week 07/08

image credit: image from [https://networks.skewed.de/net/facebook\\_friends](https://networks.skewed.de/net/facebook_friends) based on data from BF Maier and D Brockmann

## Notes:

- Let us now study motivating examples for applications of machine learning in data with an (additional) graph structure. Those examples highlight the rich interdisciplinary applications of graph learning. They also exemplify different supervised and unsupervised graph learning tasks that we address in more detail in the coming weeks.
- As a first example, consider online social networks like, e.g. Instagram, Twitter, or TikTok. Each node in the network is a user account, and (directed or undirected) links represent relationships between two accounts, e.g. one user following another user or two users being friends with each other. The example network depicted above shows an undirected network of friend relationships in a sample of Facebook users.
- Consider the case where a new user joins the online social network, initially adding links to some of his/her acquaintances to the profile. Can we use the network of existing relationships to automatically recommend accounts that could be of interest for the new user?
- This question can be cast as a **link prediction problem**, i.e. given a set of links we predict which unobserved links are likely to exist in the network. We cover different techniques to address this in week seven and eight of our course.
- On the one hand, such techniques add value both for the new user and existing users as we help to discover accounts they may wish to interact with. On the other hand, this can have negative implications for privacy, i.e. platforms can learn about relationships that users did not wish to disclose or they can create so-called shadow profile, i.e. they can collect information on users that do not even use the platform.

## Example 2: eCommerce

consider a **product-product co-purchasing network**, where nodes represent **products** and links represent **products frequently bought together**



### question

can we automate the assignment of **products to categories**?

### graph learning task

we can cast this as an **node classification** problem

→ week 10/11

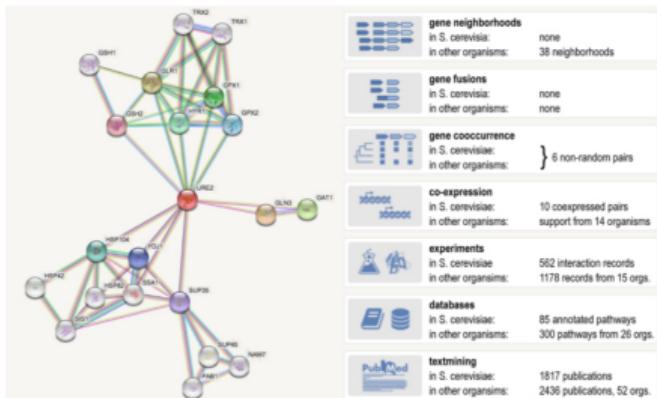
image credit: T-SNE visualization of network from → W Hu et al., NeurIPS 2020 based on data from → WL Chiang et al., SIGKDD 2019

## Notes:

- As a second example, consider a network where nodes represent products offered by an eCommerce platform. Those products that are frequently purchased together are connected by a (weighted) link. The resulting graph structure is likely to capture thematic relationships between products, i.e. we can hypothesize that groups of products that are often bought together have something in common. The visualization shown above shows an Amazon co-purchasing network from the Open Graph Benchmark data set (see references below).
- Can we use such data to automatically augment the product catalogue of the platform, e.g. by automatically assigning products to product categories?
- This can be phrased as a node classification problem, i.e. we seek to assign nodes to a discrete set of classes. In week 10 and 11 of our course, we will cover techniques that can be used to address this task.
- Such techniques can help providers of eCommerce platforms to maintain and augment their product catalogue. It helps to limit the amount of boring manual categorization work and can assist us in finding inconsistencies in the catalogue.

# Example 3: Protein-protein networks

consider a **protein-protein association network**, where nodes represent **proteins in a cell** and links represent **functional associations**



## question

can we identify groups of proteins that jointly perform a **biological function**?

## graph learning task

we can cast this as a **community detection** problem  
→ week 02/04/05

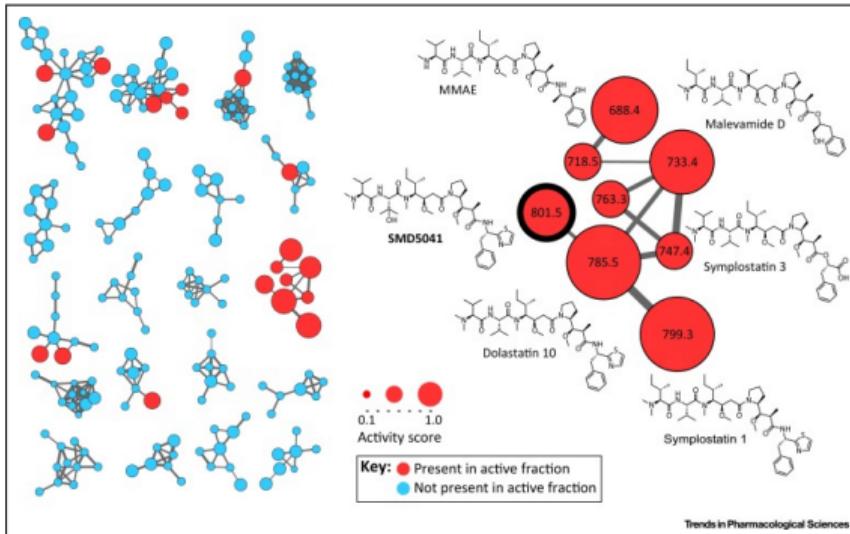
image credit: image from → D Szklarczyk, Nucl. Acids. Res., 2019

## Notes:

- Leaving the online world, consider a network of protein-protein interactions within a (human) cell. Here nodes represent proteins and a link between two proteins captures that they are engaged in a specific biological function. Such functional associations between proteins can be inferred based on various measurements in the lab, e.g. through electron microscopy, spectroscopy, fluorescence methods, etc. The network depicted above shows possible functional associations between the yeast protein URE2 (in the center) and ten other proteins based on the STRING database. This database covers different selectable sources for functional associations (including gene co-expression or text mining on scientific articles).
- Can we use the resulting network to identify groups of proteins that jointly perform specific biological functions?
- We can cast this question as a **cluster or community detection** problem, i.e. we seek to identify groups of nodes in the network that are more strongly connected (or – in a network-sense – “similar”) to each other than to nodes belonging to other groups. We have already covered some basic community detection techniques in our course *Statistical Network Analysis*. We will repeat one of those approaches in week two. In week four and five, we will introduce more advanced techniques, which directly build upon concepts introduced in this course.
- Methods to detect clusters in protein-protein interaction networks can help us to identify targets for therapeutic interventions, better understand gene functions, or identify genes associated with certain diseases.

# Example 4: Molecular networks

consider **molecular networks**, where nodes represent **atoms** and links represent **chemical bonds**



## question

can we use network structure and node/link features to automatically **discover new drugs**?

## graph learning task

we can cast this as a **graph classification** problem  
→ week 12/13

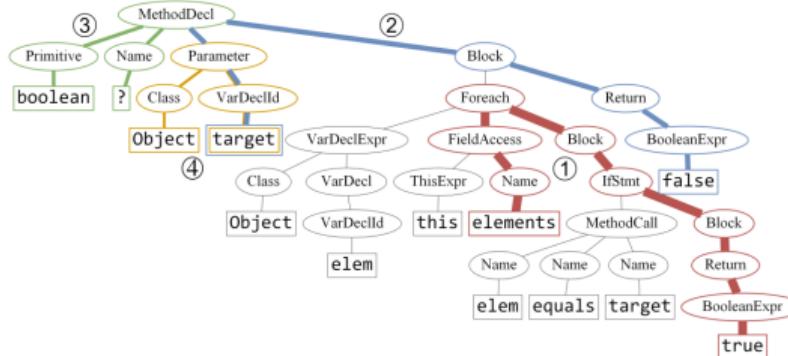
image credit: figure from → RA Quinn et al., Trends in Pharmacological Sciences, 2017

## Notes:

- For the examples considered so far, we were interested in either the nodes or links within a single graph that represented our data set. But what if we have data where a graph is only one data point? What if we have a collection of graphs and want to answer a question about one specific graph? Consider a data set of chemical compounds, where each chemical compound is represented as a molecular graph. Within such a graph, the nodes represent atoms and links represent chemical bonds between atoms. Both nodes and links can have additional features (e.g. the type of atom or properties of the chemical bond). In addition, we can have graph-level properties that relate to the chemical compound as a whole, e.g. the toxicity of the substance, therapeutic effects, or its solubility in water.
- Given a set of molecular networks representing different chemical compounds, where some of those chemical compounds have known properties, can we use the topology and node/link properties of molecular networks to discover new promising drugs?
- This can be cast as a **graph classification problem**, where we wish to assign classes to each instance of a molecular network. Hence, different from the examples before, here the whole graph represents one observation in our data set, rather than the collection of nodes or edges.
- Network-based methods to predict properties of chemical compounds have developed into an important approach in computer-assisted drug discovery and precision medicine. See the reference above for more details.

# Example 5: Code networks

consider **abstract syntax trees**, where nodes represent **programming language constructs** and links represent the **syntactic structure of code snippets**



## question

can we infer a **latent semantic space** that captures the meaning of code snippets?

## graph learning task

we can cast this as a **graph embedding** problem

→ week 12/13

image credit: figure from → U Alon, M Zilberstein, O Levy, E Yahav, POPL 2019

## Notes:

- With our final example of a graph-level learning task, we return to the domain of computer science. For this, consider an abstract syntax tree, i.e. an acyclic network where nodes are programming language constructs (e.g. variables, operators, control structures) and links represent the syntactic structure of a code snippet (e.g. one source code file, a class, or a method). Each code snippet is represented by a network, and nodes and links may have additional features. The example above shows an abstract syntax tree of a single Java method declaration (see reference for mode details).
- Can we use such a graph representation of a code snippet to position the code in a latent vector space? And can we use the resulting latent space to assist software developers, e.g. by automatically suggesting meaningful method names, discover code clones, or help in refactoring code?
- We can cast this problem as a **graph-level embedding** problem, e.g. rather than using the topology of links to embed nodes in a vector space, here we use a collection of graphs to embed each graph into a latent space.
- This example is one instance of **machine learning applications in software engineering**, which are becoming increasingly sophisticated and important in practice. You will find that some of those applications have already been implemented in modern IDEs (integrated development environments) like Eclipse or Visual Studio Code.

# A Graph Learning taxonomy

	(semi-)supervised	unsupervised
node level	node classification → example 2	node embedding community detection → example 3
edge level	link prediction link classification → example 1	link prediction network reconstruction → example 1
graph level	graph classification graph regression → example 4	graph clustering graph embedding → example 5

## Notes:

- The eight examples introduced on the previous slides highlight the interdisciplinary relevance and appeal of graph learning beyond computer science. They further showcase different types of graph learning problems, which we can roughly categorize as depicted above.
- We have seen examples for learning problems at the level of nodes, links, or whole graphs. We have further seen examples both for **(semi-)supervised and unsupervised machine learning in graphs**. Embedding and clustering tasks are examples for unsupervised problems, as we seek to model patterns in data without ground truth labels. Classification tasks are a common example for a supervised problem, where ground truth labels (either on nodes, links or graphs) are available. Note that we can also have semi-supervised problems, where labels may only be available for a (small) subset of the training examples. Finally, note that some problems, like link prediction, can be addressed both in a supervised or unsupervised fashion.
- A note on graph regression problems: While we have not considered an example for this, we can consider the supervised learning problem of predicting a numerical quantity that is associated with a given graph (i.e. similar to linear regression, where a graph is considered the independent variable).
- Moreover (and not depicted in the necessarily incomplete taxonomy above) in some of the examples we have additional features at the level of nodes, links or graphs, which we can use to improve the performance of our algorithms.

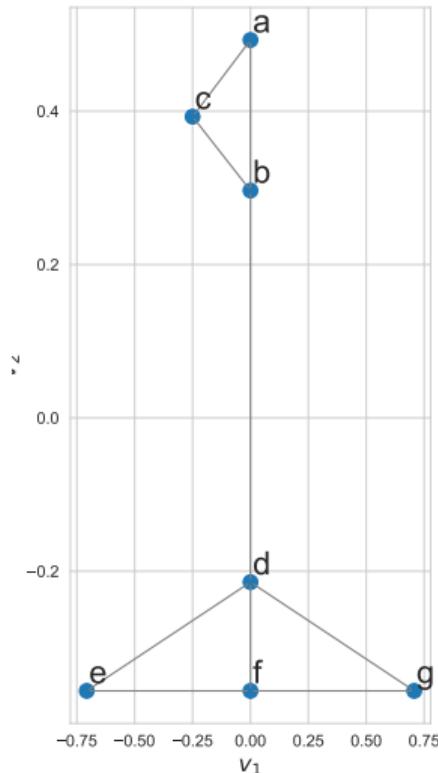
# Representation Learning for Graphs

- ▶ nodes are  **$n$ -dimensional objects** in a discrete non-Euclidean space ( $n = |V|$ )
- ▶ how can we map nodes  $v$  to features  $x_v \in \mathbb{R}^d$ ?
- ▶ we could use **node-level positional measures** (e.g. node centralities)
  1. independent feature dimensions?
  2.  $\text{dist}(x_u, x_v)$  vs.  $\text{dist}(x_v, x_w)$ ?
  3. which information on the topology can we recover?

## node embedding

find **low-dimensional Euclidean representation** of nodes that **best preserves information in graph**, i.e. map nodes  $v$  to positions  $x_v \in \mathbb{R}^d$  such that distances between nodes are correlated with their “similarity” in the graph

- ▶ node embedding is **foundation to apply machine learning to graphs**



## Notes:

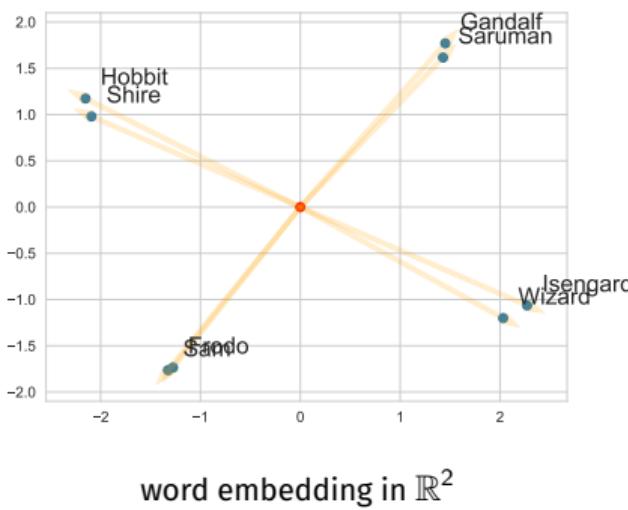
- While the Fruchterman-Reingold algorithm works well for link prediction in a random geometric graph where nodes were assigned to positions in  $\mathbb{R}^2$ , we want to address this in a more principled way. Graphs are discrete, high-dimensional and non-Euclidean objects, while standard ML Techniques require discrete, low-dimensional, and Euclidean data. In particular, we can consider each node as an  $n$ -dimensional object that is “defined” by its relationship to all  $n$  other nodes (incl. self-loops).
- How can we map such high-dimensional objects to a low-dimensional Euclidean feature space? In principle, we could use any collection of  $d$  node-level measures like, e.g., node degrees, node centrality scores, local clustering coefficients, etc. to map nodes to a multi-dimensional feature space. This approach was, in fact, used by many early works applying machine learning to graphs, however it has several issues. First, the resulting feature dimensions are not necessarily independent or, in a geoemtric interpretation, we obtain coordinates where the dimensions are not necessarily orthogonal (consider, e.g., correlations between degrees and centrality measures). Moreover, it is not clear whether the resulting space fulfils the metric properties, which are implicitly used by many machine learning techniques. And finally, depending on the choice of measures, it is not clear which information about the original graph we can recover.
- We are thus interested in methods to find low-dimensional Euclidean representations of graphs, which we collectively refer to as **graph representation learning** or **node embedding**. Our goal is to find representations of nodes in a low-dimensional vector space that retain a maximum of information on the underlying graph topology.

# Word embeddings in NLP

- ▶ in **natural language processing** we are interested in **word embeddings**
- ▶ for text with vocabulary  $V$  find **embedding of words in vector space**, i.e.  $f : V \rightarrow \mathbb{R}^d$
- ▶ 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
$$\text{sim}(v, w) \approx \vec{v} \cdot \vec{w}$$
- ▶ basis for text completion, semantic analysis, sentiment analysis, homonym detection, ...

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

sample vocabulary



## 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 important 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 Wizard 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-occurrence 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  $\Theta$  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)  $\Theta$ .
- 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  $\sigma$  to normalize dot products  $\vec{c} \cdot \vec{w}$  to probabilities**, i.e. for given vector representations we can write

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

- ▶ 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.

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

- ▶ 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.

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

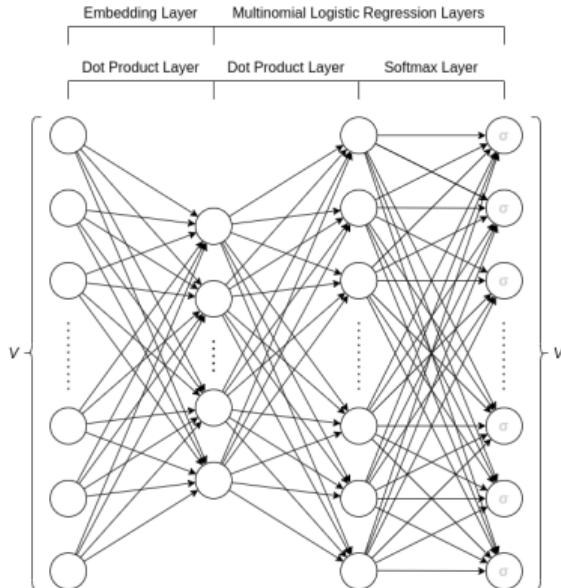
- ▶ add **second layer with  $|V|$  perceptrons** with weights  $\beta^2 \in \mathbb{R}^{d \times |V|}$  and **softmax activation function  $\sigma$**  with output

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

## 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 simple perceptron model, 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**

---

**Authors:**

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 published results of the N-gram language modeling task [1]. We observe large improvements in accuracy of 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 most NLP models trained on huge amounts of data empirically perform reasonably well on tens of data. An example is the n-gram language model used for statistical language modeling - today, it is possible to train N-grams on virtually all available data (billions of words) [3].

However, the simple techniques are at their limits in many tasks. For example, the amount of relevant (i.e. local) data available for such methods is limited - the performance is easily dominated by the lack 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 performance improvement.

With progress of machine learning techniques in recent years, it has become possible to train more complex models on much larger data sets, 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, 2, 13].

---

**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  $\beta_1$  and  $\beta_2$  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  $\beta^1, \beta^2$  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  $P \in \mathbb{R}^{|V| \times |V|}$  of conditional probabilities** that can be expressed as

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

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

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

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

## Notes:

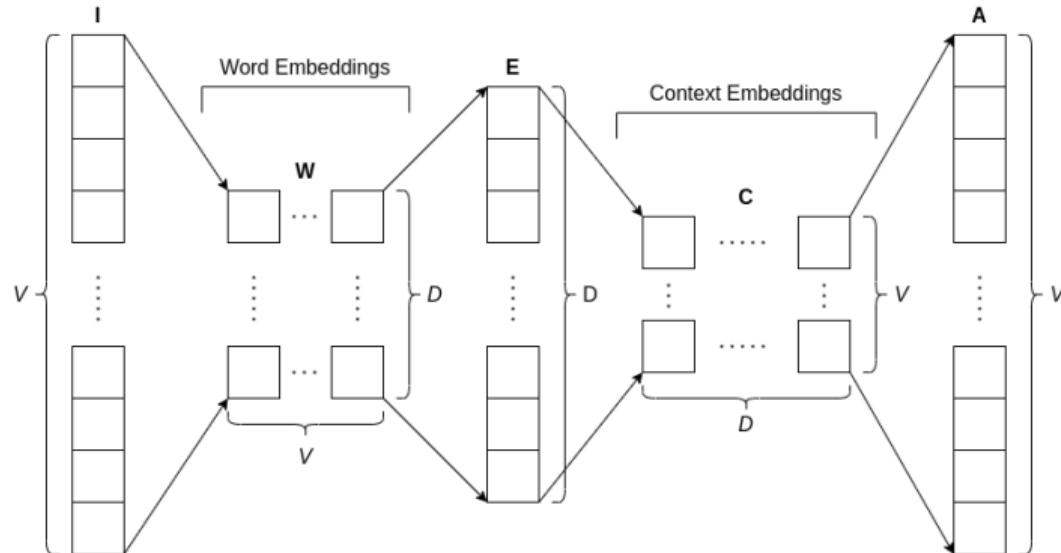
- To justify the interpretation of weights as vector representations of center and context words we can expand the expression for the outputs  $y_j$  of our neural network as

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

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  $(\beta^1)_i \in \mathbb{R}^d$  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:

- 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

```
1 import torch
2 import torch.nn.functional as F
3 from torch.nn import Parameter
4 from torch import autograd
5 from torch import optim
6 import numpy as np
7 import matplotlib.pyplot as plt
8 import pandas as pd
9 import seaborn as sns
10 from collections import Counter
11
12 plt.style.use('default')
13 sns.set_style("whitegrid")
14
15 from sklearn.manifold import TSNE
✓ 25s
```

c:\Users\ingos\Anaconda3\lib\site-packages\statsmodels\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

```
1 text = [
2     'Gandalf is a Wizard',
3     'Frodo is a Hobbit',
4     'Sam is a Hobbit',
5     'Saruman is a Wizard',
6     'Sam lives in the Shire',
7     'Frodo lives in the Shire',
8     'Saruman lives in Isengard',
9     'Gandalf is imprisoned in Isengard'
10 ]
✓ 0.3s
```

Python

Python

## practice session

see notebook 10-01 in gitlab repository at

→ [https://gitlab.informatik.uni-wuerzburg.de/ml4nets\\_notebooks/2024\\_sose\\_ml4nets\\_notebooks](https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks)

## Notes:

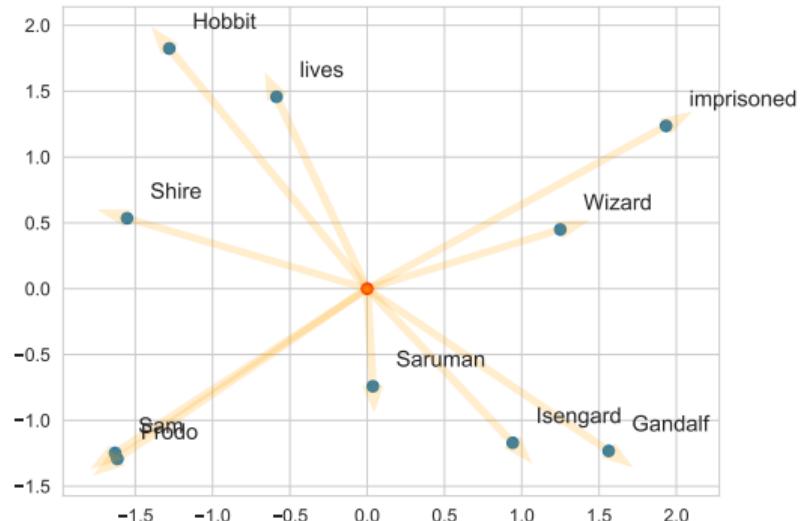
- In the first practice session, we use our newly acquired knowledge to implement the Skipgram model from scratch in pytorch. We use the model to calculate word embeddings for a small toy example text corpus.

# Example: word2vec

## example

- ▶ word2vec embedding using Skipgram model with  $d = 5$  hidden dimensions
- ▶ window size  $s = 2$ , stop words {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.
```



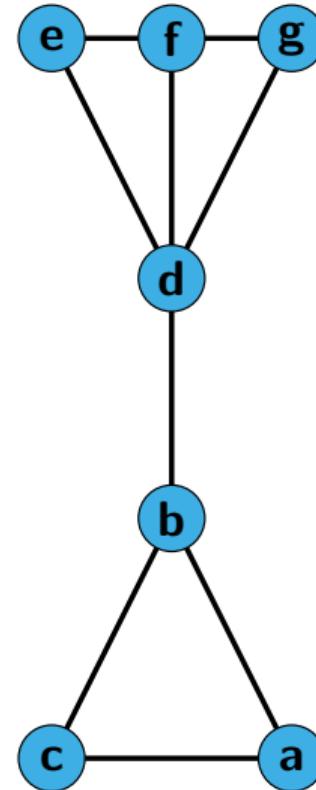
word embedding in  $\mathbb{R}^2$  (dimensions reduced via truncated SVD)

## 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 graph representation learning?

- ▶ can we apply the **neural matrix factorization** idea behind word2vec to calculate node embeddings?
- ▶ network analogy of **distributional hypothesis** in linguistics?
- ▶ for network  $G = (V, E)$  consider **walk**  $v_0, v_1, \dots, v_l$  of **length  $l$** , i.e.  $v_i \in V$  and  $(v_i, v_{i+1}) \in E$  for  $i = 0, \dots, l - 1$
- ▶ 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:

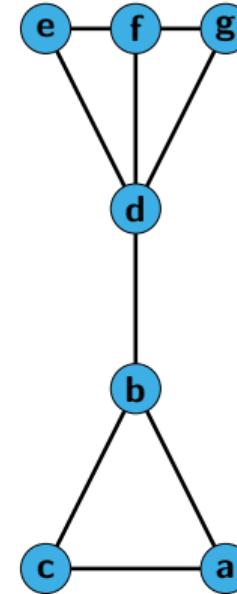
- In word2vec, words with similar context are identified as similar and get closer vector representations. Can we use the same approach to calculate vector representations of nodes in complex networks? To make this work, we first need a network analogy of the distributional hypothesis in linguistics, 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.

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

- ▶ 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

## practice session

see notebook 10-02 in gitlab repository at

→ [https://gitlab.informatik.uni-wuerzburg.de/ml4nets\\_notebooks/2024\\_sose\\_ml4nets\\_notebooks](https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks)

## 10-02 Neural Graph Embedding with DeepWalk

July 13, 2022

```
1 import torch
2 import torch.nn.functional as F
3 from torch.nn import Parameter
4 from torch import autograd
5 from torch import optim
6 import numpy as np
7 import matplotlib.pyplot as plt
8 import pandas as pd
9 import seaborn as sns
10
11 plt.style.use('default')
12 sns.set_style("whitegrid")
13
14 from sklearn.manifold import TSNE
15
16 import pathpy as pp
```

Python

```
1 n = pp.Network(directed=False)
2 n.add_edge('a', 'b')
3 n.add_edge('b', 'c')
4 n.add_edge('a', 'c')
5 n.add_edge('b', 'd')
6 n.add_edge('d', 'f')
7 n.add_edge('d', 'g')
8 n.add_edge('d', 'e')
9 n.add_edge('e', 'f')
10 n.add_edge('f', 'g')
11 n.plot(edge_color='gray')
```

Python

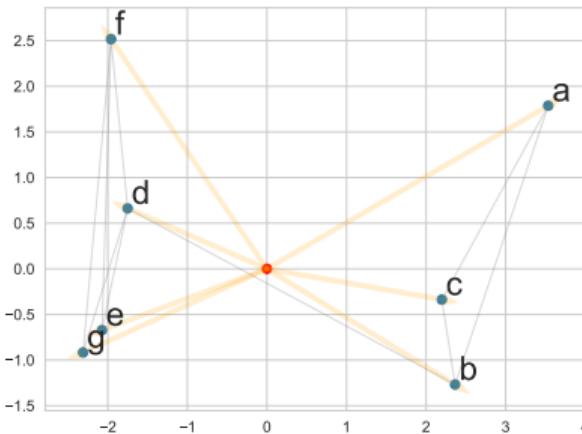
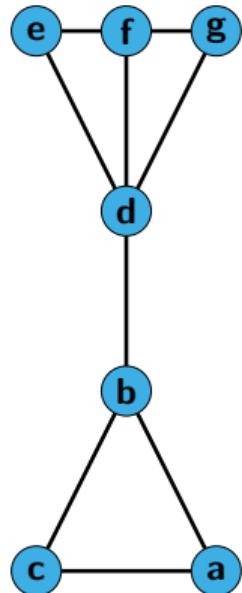
## Notes:

- 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

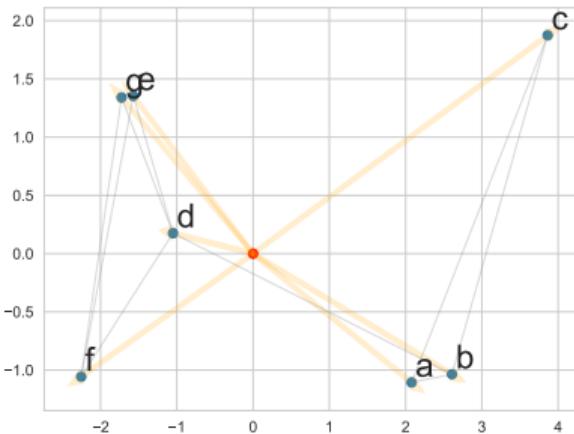
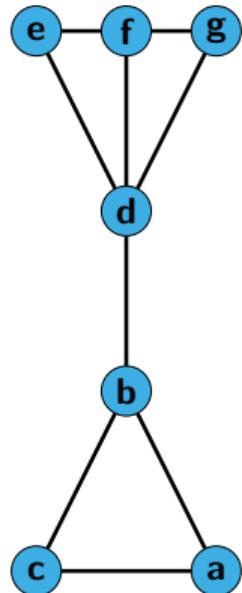
## Notes:

- 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  $s = 3$



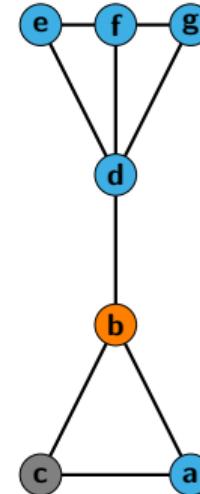
**two-dimensional projection of DeepWalk embedding** generated by truncated singular value decomposition

## 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 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

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

## 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-exploitation 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**
- for random 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  $p = q = 1$



Aditya Grover



Jure Leskovec

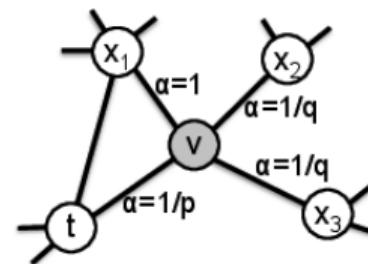


illustration of transition probabilities for random walker that just moved from node  $t$  to  $v$

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  $q \gg 1$  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

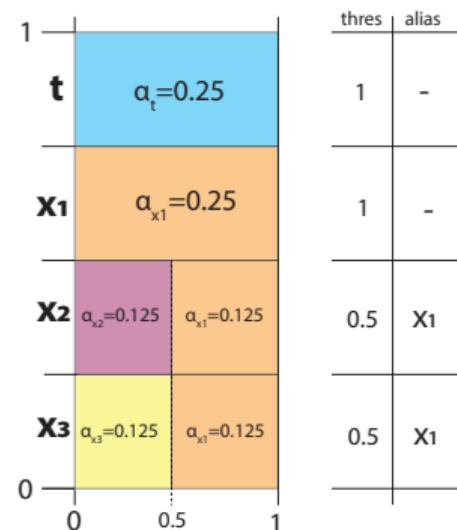
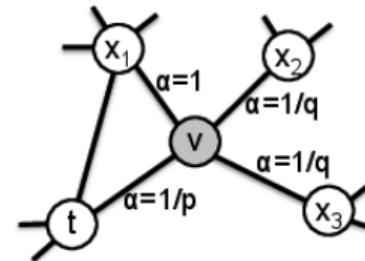
**alias sampling** → AJ Walker, 1977

initialization: generate threshold/alias table in  $O(m \log m)$

sampling:  $i = \text{randint}(m)$

```
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 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  $\alpha_i$  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

10-03 - node2vec

July 13th, 2022

```
1 import torch
2 import torch.nn.functional as F
3 from torch import Parameter
4 from torch import autograd
5 from torch import optim
6 import numpy as np
7 import matplotlib.pyplot as plt
8 import pandas as pd
9 import seaborn as sns
10 from sklearn.decomposition import TruncatedSVD
11
12 plt.style.use('default')
13 sns.set_style("whitegrid")
14
15
16 import pathpy as pp
✓ 0.8s
```

Python

```
1 n = pp.Network(directed=False)
2 n.add_edge('a', 'b')
3 n.add_edge('b', 'c')
4 n.add_edge('a', 'c')
5 n.add_edge('b', 'd')
6 n.add_edge('c', 'd')
7 n.add_edge('d', 'e')
8 n.add_edge('e', 'a')
9 n.add_edge('e', 'f')
10 n.add_edge('f', 'g')
11 n.plot(edge_color='gray')
```

✓ 0.6s

Python

SVG | PNG | Reset | Download

practice session

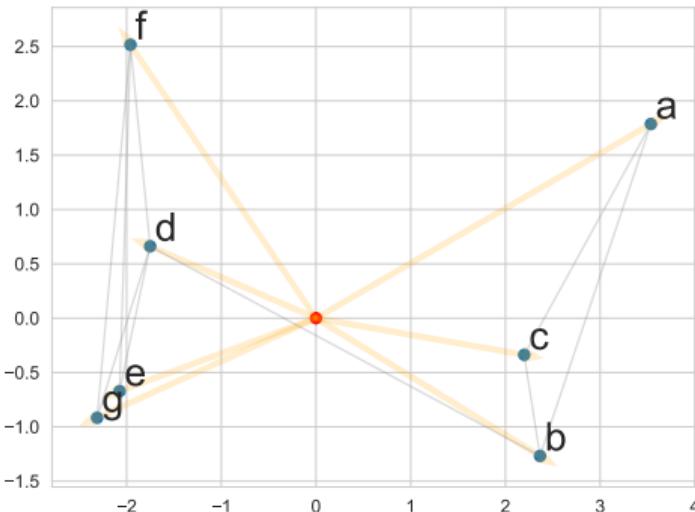
see notebook 10-03 in gitlab repository at

→ [https://gitlab.informatik.uni-wuerzburg.de/ml4nets\\_notebooks/2024\\_sose\\_ml4nets\\_notebooks](https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks)

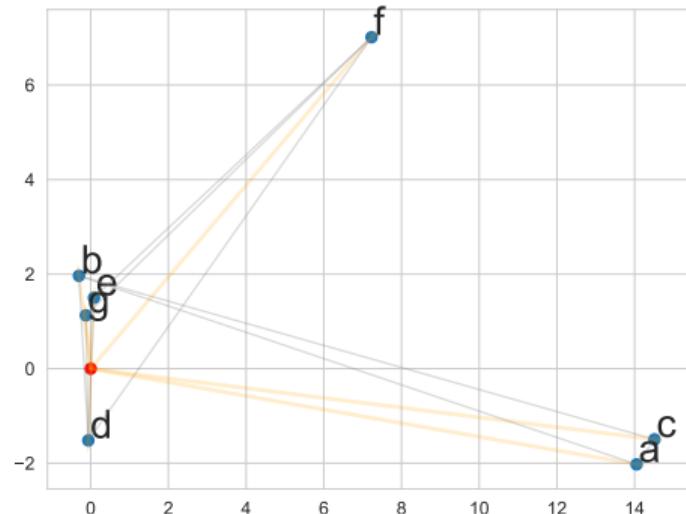
## Notes:

- 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  
 $s = 2$

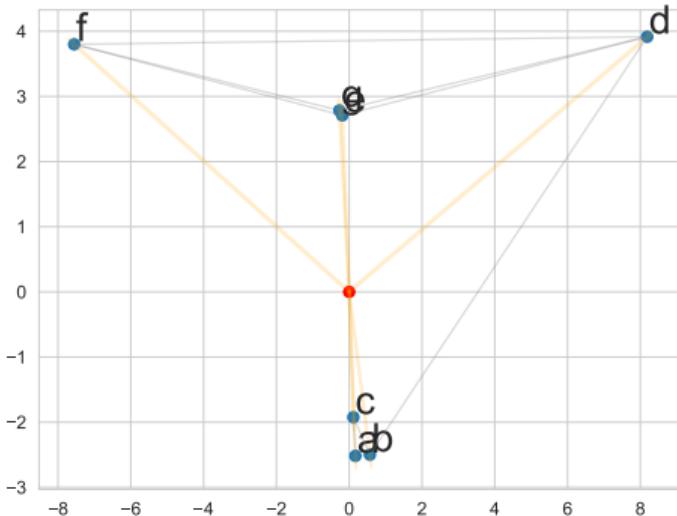


node2vec embedding for  $p = 100$  and  $q = 0.01$   
**low return probability, strong exploration bias**  
walk length  $l = 20$ , window size  $s = 2$

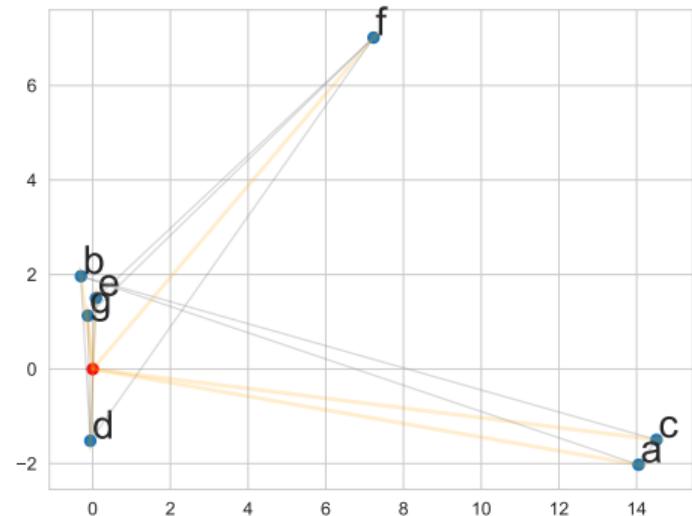
## Notes:

- 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 node2vec 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  $s = 2$



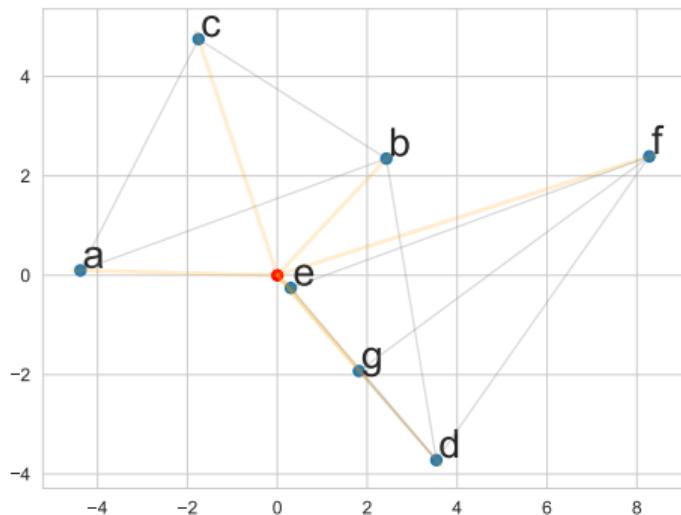
node2vec embedding for  $p = 100$  and  $q = 0.01$   
**low return probability, strong exploration bias**  
 walk length  $l = 20$ , window size  $s = 2$

## 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 $q$

2/2

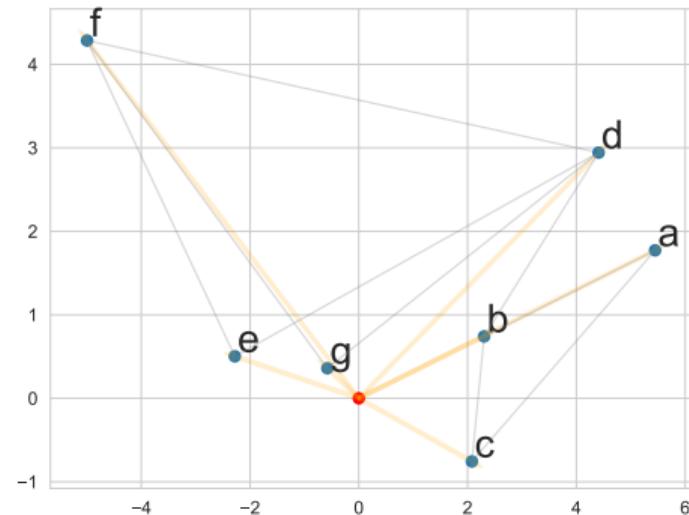


node2vec embedding for  $p = 0.01$  and  $q = 100$

**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  $s = 2$

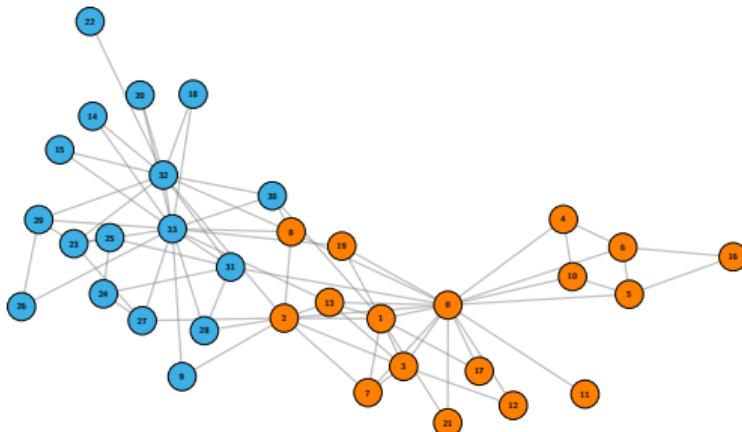
## Notes:

- 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).

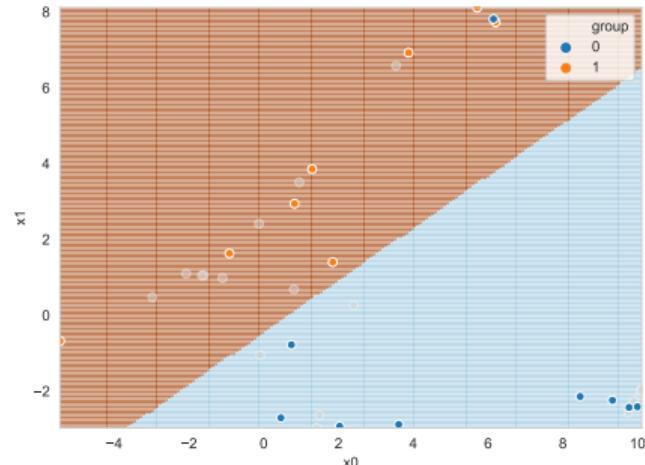
# Node classification with DeepWalk

## 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  $\approx 0.94$

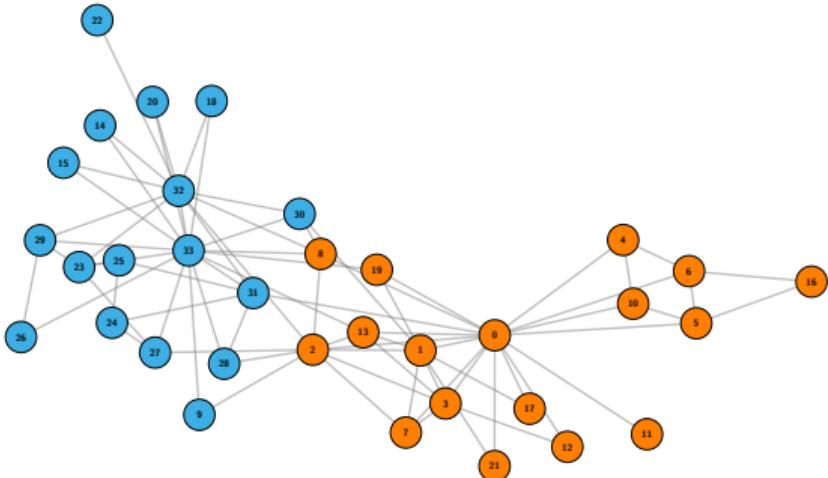
## Notes:

- 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.

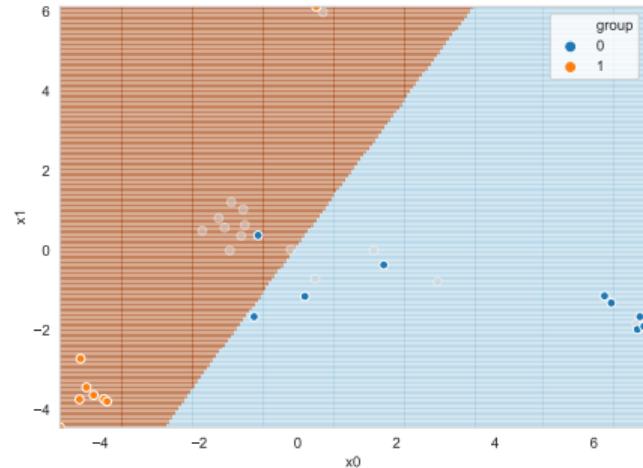
# Node classification with node2vec

## 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  $\approx 0.94$

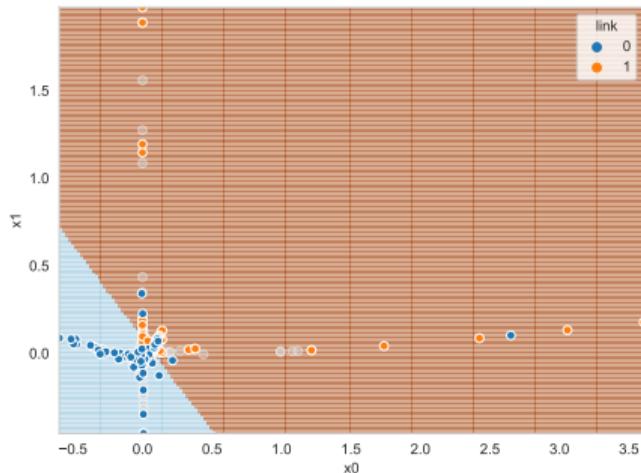
## Notes:

- 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.

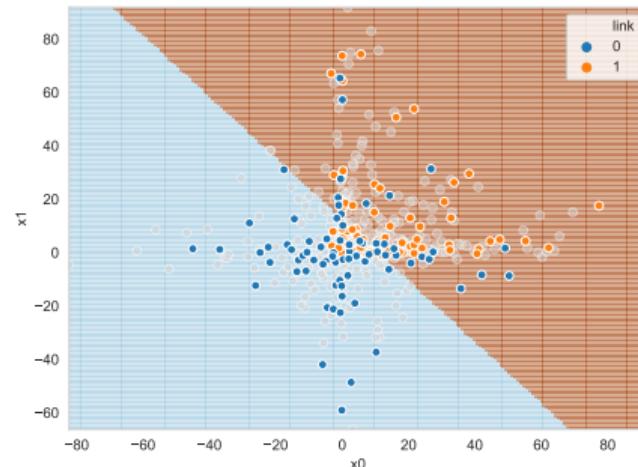
# Link prediction with DeepWalk

## 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:**



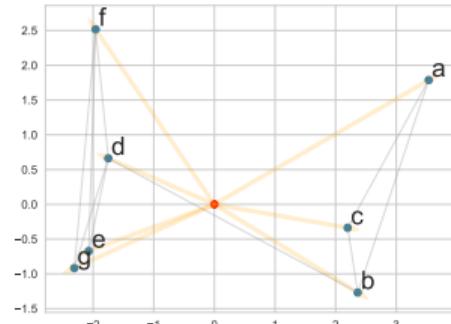
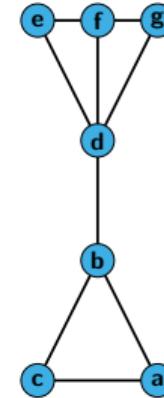
## Notes:

- 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 that 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 remind us of the loss functions that are implicitly minimized by matrix factorization technique, e.g. the minimization of euclidean distance between connected nodes, which is minimized by Laplacian eigenmaps.
- 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 Laplacian embedding. This can be seen as a possible explanation for the results on slide 38, 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 ~ neural matrix factorization

→ 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.
- 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 8th** (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

$$L = \frac{1}{T} \sum_{t=1}^T \sum_{-k \leq j \leq k, j \neq 0} \log q(w_{t+j} | w_t)$$

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

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

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

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

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  $D_{KL}(p \| q)$  of  $p$  from  $q$ .

$$H(p, q) = H(p) + D_{KL}(p \| q)$$

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.

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

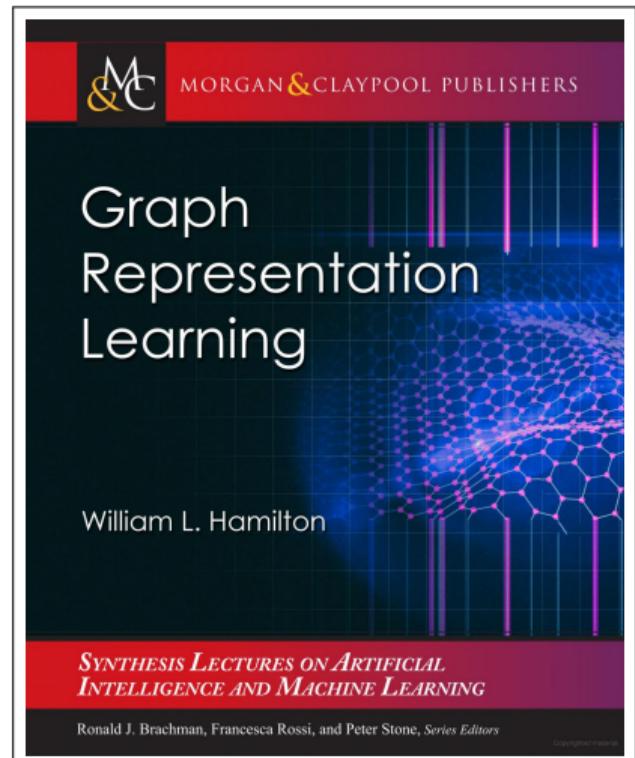
1. What is the distributional hypothesis in linguistics and how we can use it to embed words in a vector space?
2. Explain how we can use neural networks to find approximate factorizations of matrices.
3. Explain whether the power law distribution of degrees is likely to affect node embeddings generated 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 matrix factorization techniques like Laplacian Eigenmaps?

## **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



## **Notes:**