# Capturing and evaluating higher order relations in word embeddings using tensor factorization

A thesis submitted by

Eric Bailey

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Tufts University

May 2017



Adviser: Dr. Shuchin Aeron

**Abstract**

In Natural Language Processing, most popular word embeddings involve low-rank factorization of a word co-occurrence based matrix. We aim to generalize this trend by studying word embeddings given by low-rank factorization of word co-occurrence based higher-order arrays, or *tensors*. We present four novel word embeddings based on tensor factorization and show they outperform popular state-of-the-art baselines on a number of recent benchmarks, encoding useful properties in a new way. To create one of our word embeddings, we present a novel joint symmetric tensor factorization problem related to the idea of *coupled* tensor factorization. We also modify a recent embedding evaluation technique known as Outlier Detection to measure the degree to which an embedding captures $N^{th}$ order information, showing that tensor embeddings (naturally) outperform popular pairwise embeddings at this task. Suggested applications of tensor factorization-based word embeddings are given, and all source code and pre-trained vectors are publicly available online.

# Contents

# List of Tables





# List of Figures

# Chapter 1

# Introduction

## 1.1 Word embeddings

Word embeddings are an important part of NLP, and can be used in almost every NLP task. They have been shown to improve the performance of many tasks, from language modelling [Bengio et al., 2003] to Machine Translation [Bahdanau et al., 2014] to sentiment analysis [Kim, 2014]. Their broad applicability to almost every problem in NLP and ease of training makes them a hot area of study, especially since [Mikolov et al., 2013b] recently showed great results on quickly extracting information on massive amounts of data with their `word2vec` model.



The word embedding problem is to learn a function $\eta$ mapping words in a language to $\mathbb{R}^k$ ($k \approx$ 100-300 in most applications) that encodes meaningful semantic and/or syntactic information[1]. For example, typically words mean similar things if and only if their corresponding vectors appear nearby in the vector space. So for most modern word embeddings, $\eta(\text{car}) \approx \eta(\text{automobile})$, since the words mean similar things.

Models often encode many different types of semantic information. For instance, in the `word2vec` model [Mikolov et al., 2013b], we can answer analogy queries of the form $a : b :: c : ?$ using simple vector arithmetic: the answer to bed

[1] In NLP, *Semantic information* is the information related to intrinsic meaning. For example, the fact that *cat* and *dog* mean similar things (but not the same thing) is semantic information. *Syntactic information* is the information about the set of rules that govern the structure of sentences in a language. For example, the fact that *dog* is a noun (and that verbs can have subjects/objects that are nouns) is syntactic information.

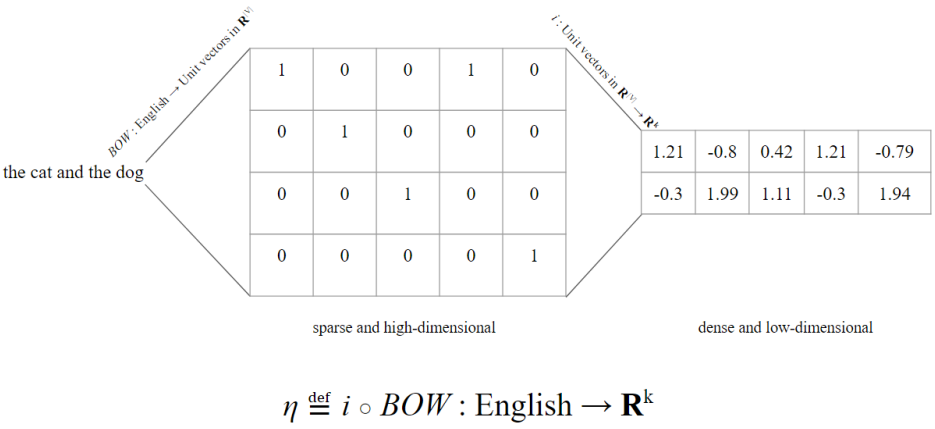


**Figure 1.1:** Toy example encoding using BoW with $|V| = 4$ and $k = 2$

: sleep :: chair : $x$ is given by the vector closest to $\eta(\text{sleep}) - \eta(\text{bed}) + \eta(\text{chair})$ ($\approx \eta(\text{sit})$). However, different word embeddings encode different types of information, or may encode such information in a nonlinear way [Jastrzebski et al., 2017].

$\eta$ is typically learned by first creating a finite vocabulary $V$ and then mapping words to a one-of-$|V|$ encoding (also known as a Bag of Words encoding). Then, the index of each word is used to index into a $|V| \times k$ matrix, where the $i^{th}$ row of the matrix holds the vector corresponding to the $i^{th}$ word in the vocabulary. So the word embedding problem can be (and often is) solved by finding a $|V| \times k$ matrix with rows that encode meaningful information about the corresponding vocabulary word. $|V|$ typically ranges from around 20,000 to 300,000 in production-ready word embeddings. This process is visualized in Figure 1.1.

Word embeddings are quite advantageous for many machine learning algorithms, which require a fixed-length vector representation of the input such as Logistic Regression or simple Artificial Neural Networks. For example, sentiment analysis of a full sentence can easily be solved by first mapping each word into is fixed-length vector representation and then summing each of the word vectors in the sentences elementwise. Then the entire sentence has a vector representation in $\mathbb{R}^k$ which can be fed into any off-the-shelf classification algorithm.

Word embedding can be applied to other areas of computer science – graph embeddings can be given via word embedding techniques [Grover and Leskovec, 2016]. Other fields of CS have also used ideas from word embeddings; for example, computational biology (dna2vec) [Ng, 2017], 3D Graphics (shape2vec) [Tasse and Dodgson, 2016], and Item Recommendation (item2vec) [Barkan and Koenigstein, 2016] all have contributions using word embeddings[2]. Thus, even incremental improvements on word embedding architectures may lead to further improvements across many areas.

Almost all of the recent word embeddings rely on the distributional hypothesis [Harris, 1954], which states that a word's meaning can be inferred from the words that surround it. For example, one can infer that the $x$ in "he ate two full servings of $x$ for breakfast" is something that is supposed to be eaten for breakfast. For instance, "cereal" and "granola" can reasonably be substituted for $x$, and we see that they mean similar things.

In this work we work with the distributional hypothesis, but work with higher-order co-occurrence information. Next, we will outline our main contributions.



## 1.2 Summary of main contributions

Our main contributions are fourfold:

1. **Four novel tensor-based word embeddings.** We apply the theory of tensor factorization to training generic word embeddings, studying many different types of tensor factorization that utilize different aspects of tensor structure – supersymmetry, sparsity, and nonnegativity. We call our new embeddings **CP, CP-S, CP-SN, JCP-S**. We motivate intuition behind some new properties these embeddings encode in Section 1.4 and show that they actually encode such properties in Section 4.4. We show experimentally that sparse embeddings can encode *more* (or encode information more readily) than dense embeddings. This suggests that sparse embeddings (which tend

[2]A list of many of the "*2vec"s can be found online: `https://gist.github.com/nzw0301/333afc00bd508501268fa7bf40cafe4e`

to be overlooked in the deep learning/NLP communities) should be considered and further studied.

Suggested applications for our embeddings are outlined in Section 4.6.

2. **A novel joint tensor factorization technique for simultaneously decomposing $N$ supersymmetric tensors.** Joint Tensor Factorization (or Coupled Tensor Factorization [Acar et al., 2011, Naskovska and Haardt, 2016]) is a problem in which multiple tensors are being factorized sharing at least one factor matrix. We introduce and utilize a new joint symmetric tensor factorization problem in order to create a novel word embedding, which captures multiple different orders of relations between words. To the best of our knowledge, the special case of joint supersymmetric tensor decomposition has not been considered so far in the literature.

3. **New embedding evaluation metric to measure degree of capturing $N^{th}$-order information.** We modify the idea of Outlier Detection for evaluating word embeddings [Camacho-Collados and Navigli, 2016] to produce an $N$-way analog of this metric we call Outlier Detection ($N$) (OD$N$). This metric evaluates to what degree $N$-way information is captured by a given word embedding. We demonstrate that our third order models outperform state-of-the-art baselines (including `word2vec` [Mikolov et al., 2013b]) on OD2 and OD3.

4. **General-purpose TensorFlow framework for computing online CP decomposition.** While implementing our embeddings in TensorFlow, we found it useful to create a general framework for computing online CP Decomposition. To the best of our knowledge, such a library does not yet exist for TensorFlow, so we release it to the general public included in our word embedding code[3]. We have implemented asymmetric, symmetric, sparse, and joint online CP decomposition, including both GPU and CPU support.

[3] `https://github.com/popcorncolonel/tensor_decomp_embedding/blob/master/tensor_decomp.py`

## 1.3 Mathematical Background

In this section we describe the mathematical notation used in this work and outline the mathematical background required to understand our methodology.

### 1.3.1 Pointwise Mutual Information

Pointwise mutual information (PMI) is a measure very useful in NLP. We discuss its utility in Section 1.4 and we give some of its past uses in NLP in Section 2.3.

PMI is an extension of the notion of mutual information between two random variables [Cover and Thomas, 2006]. For two random variable $X, Y$, the Mutual Information between $X$ and $Y$ is given by:

$$I(X;Y) = \mathbb{E}\left[\log \frac{p(X,Y)}{p(X)p(Y)}\right]$$

where $\mathbb{E}$ is the expectation of a random variable. PMI is defined mathematically as

$$PMI(x,y) = \log \frac{p(x,y)}{p(x)p(y)}$$

PMI measures the degree of correlation between its arguments – it directly compares the *statistical dependence* between $x$ and $y$ (given by the numerator $p(x,y)$) vs. the *statistical independence* of $x$ and $y$ (given by the denominator $p(x)p(y)$).

Often in literature working with PMI, it is useful to observe the *positive* PMI (PPMI), defined as

$$PPMI(x,y) := \max(0, PMI(x,y))$$

[Bullinaria and Levy, 2007, Levy and Goldberg, 2014, Van de Cruys, 2009], as negative PMI values have little grounded interpretation.

Given an indexed finite set $V$, it is often useful to construct a $|V| \times |V|$ PPMI matrix $\mathbf{M}$ where $m_{ij} = PPMI(V_i, V_j)$. Many word embedding models (and more generally, problems in NLP) involve factorizing this PPMI matrix. For

example, [Levy and Goldberg, 2014] not only showed that `word2vec` *implicitly* factors a PMI-related matrix, but also *explicitly* factorized a shifted version of this PPMI matrix to produce a word embedding – see Section 1.4 for a more explicit formulation.

PMI can be generalized to $N$ variables. There are multiple different ways to formulate $N$-way PPMI [Van de Cruys, 2011]. One is given (on three variables) by:

$$\begin{aligned} PMI_1(x,y,z) &= \log \frac{p(x,y)}{p(x)p(y)} - \log \frac{p(z)p(x,y,z)}{p(x,z)p(y,z)} \\ &= \log \frac{p(x,y)p(x,z)p(y,z)}{p(x,y,z)p(x)p(y)p(z)} \end{aligned}$$

And another given by:

$$PMI_2(x_1^N) = \log \frac{p(x_1,\ldots,x_n)}{p(x_1)\cdots p(x_N)}$$

At least based on notational similarity, we believe $\text{PMI}_2$ to be the more natural generalization of PMI for two variables. Also, it has been shown (albeit preliminarily) that $\text{PMI}_2$ performs better at extracting certain types of salient information than $\text{PMI}_1$ [Van de Cruys, 2011] based on their experiments on extracting subject-verb-object triples in NLP, so in this work we choose to use this version.

These different forms of PMI can be considered in a graph-based ideological framework: nodes are the variables and the edges are the strength of association between the variables. Under this framework, $\text{PMI}_1$ compares the strength of all pairwise connections in the system ($p(x_i, x_j)$) to the other connected components ($p(x_i), p(x_1, x_2, x_3)$), whereas $\text{PMI}_2$ compares the strength of the clique formed by the nodes compared (how related all $n$ words are - $p(x_1, \ldots, x_N)$) to the disconnected graph ($p(x_1)\cdots p(x_N)$). Using this idea, perhaps $\text{PMI}_1$ may be better at capturing pairwise relations between words than $\text{PMI}_2$, but we leave different formulations of PMI to future research.

For intuition about some of the information PPMI encodes, we present a

list of some of the most interesting triples with high PPMI's we found in Table 1.1. As we can see, word neighbor information (*las, vegas, poker*), phrase-based information (*teenage, mutant, ninja*), and relational syntax information (i.e., subject-verb-object triples: *enzyme, catalyzes, reaction*) are captured.

| **High PPMI Triples** |
|---|
| (*las, vegas, poker*) |
| (*buick, cadillac, chevrolet*) |
| (*jesus, christ, saints*) |
| (*teenage, mutant, ninja*) |
| (*enzyme, catalyzes, reaction*) |
| (*sonic, hedgehog, knuckles*) |
| (*dorsal, fin, spine*) |

**Table 1.1:** Meaningful triples with some of the the highest PPMIs.



In the same way one can construct a PPMI matrix, we study $N$-way PPMI *tensors* $\mathcal{M}$, where $m_{ijk} = PPMI(w_i, w_j, w_k)$.

### 1.3.2 Tensors

#### Basic tensor notation

A tensor for our purpose is simply an $N$-way array [Kolda and Bader, 2009]. For example, vectors can be seen as tensors of order 1, and matrices can be seen as tensors of order 2. Following the standard notation in the literature, throughout this work we will write scalars in lowercase italics $\alpha$, vectors in lowercase bold letters $\mathbf{v}$, matrices with uppercase bold letters $\mathbf{M}$, and tensors (of order $N > 2$) with Euler script notation $\mathcal{X}$.

The mode-$n$ product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1,\ldots,I_N}$ and a matrix $\mathbf{M} \in \mathbb{R}^{J \times I_n}$ is of size $I_1, \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N$ and is given by

$$(\mathcal{X} \times_n \mathbf{M})_{i_1 \ldots i_{n-1} j i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 \ldots i_N} m_{j i_n}$$

In other words, each mode-$n$ fiber ($\in \mathbb{R}^{I_n}$) of $\mathcal{X}$ is left-multiplied by $\mathbf{M}$ ($\in \mathbb{R}^{J \times I_n}$).

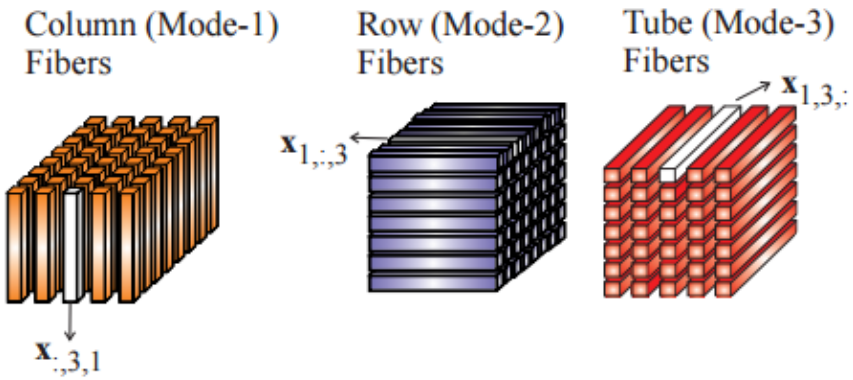


**Figure 1.2:** Mode-$n$ fibers for a third order tensor ($n = 1, 2, 3$) [Cichocki et al., 2009]

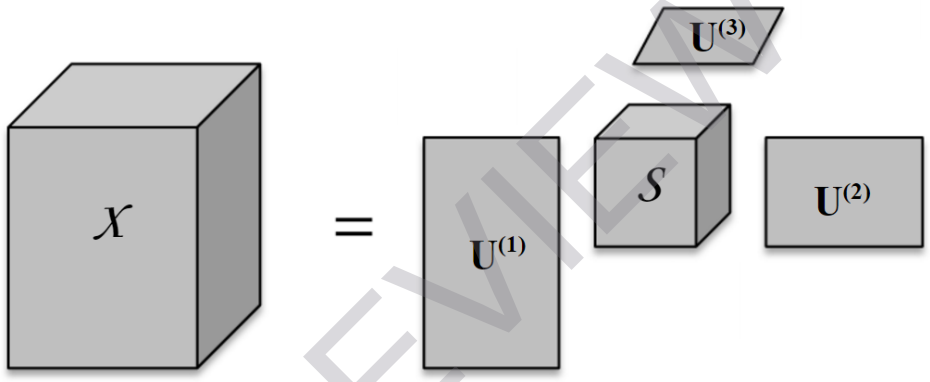


**Figure 1.3:** Tucker decomposition of a third order tensor



### Tensor factorization

Now we will describe the most common tensor factorization techniques. For ease of notation, we will discuss factorization of a 3-way tensor, but it will be obvious how to generalize to the case of an $N$-way tensor.

Assume $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$. The most common tensor factorizations algorithms decompose $\boldsymbol{\mathcal{X}}$ into a small core tensor $\boldsymbol{\mathcal{S}} \in \mathbb{R}^{I' \times J' \times K'}$ where $I' \ll I, J' \ll J, K' \ll K$, and 3 factor matrices $\mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \mathbf{U}^{(3)} \in \mathbb{R}^{x' \times x}$ (for $x \in \{I, J, K\}$) such that

$$\boldsymbol{\mathcal{X}} \approx \boldsymbol{\mathcal{S}} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \times_3 \mathbf{U}^{(3)}$$

Such a decomposition is called the *Tucker decomposition*. This is visualized in Figure 1.3.