Word Vectors

Marta R. Costa-jussà
Universitat Politecnica de Catalunya, Barcelona

Based on slides by
Christopher Manning, Stanford University, adapted from CS224n slides: Lecture 1
and illustrations from Jay Alammar, The Illustrated Word2Vec

Towards an efficient representation of words

```
            Rome   Paris                   word V
Rome   = [1,  0,  0,  0,  0,  0,  …,  0]
Paris  = [0,  1,  0,  0,  0,  0,  …,  0]
Italy  = [0,  0,  1,  0,  0,  0,  …,  0]
France = [0,  0,  0,  1,  0,  0,  …,  0]
```

Towards an efficient representation of words

```
            Rome  Paris                       word V
Rome   = [1,    0,    0, 0, 0, 0, …,  0]
Paris  = [0,    1,    0, 0, 0, 0, …,  0]
Italy  = [0,    0,    1, 0, 0, 0, …,  0]
France = [0,    0,    0, 1, 0, 0, …,  0]
```

```
          Rome  Paris              word V
doc_1 = [32,  14,  1,  0, …,   6]
doc_2 = [ 2,  12,  0, 28, …,  12]
…       …
doc_N = [13,   0,  6,  2, …,   0]
```

Towards an efficient representation of words

```
           Paris
      Rome                           word V
Rome   = [1, 0, 0, 0, 0, 0, …, 0]
Paris  = [0, 1, 0, 0, 0, 0, …, 0]
Italy  = [0, 0, 1, 0, 0, 0, …, 0]
France = [0, 0, 0, 1, 0, 0, …, 0]
```

```
        Rome  Paris            word V
doc_1 = [32, 14, 1,  0, …,  6]
doc_2 = [ 2, 12, 0, 28, …, 12]
…       …
doc_N = [13,  0, 6,  2, …, 0]
```

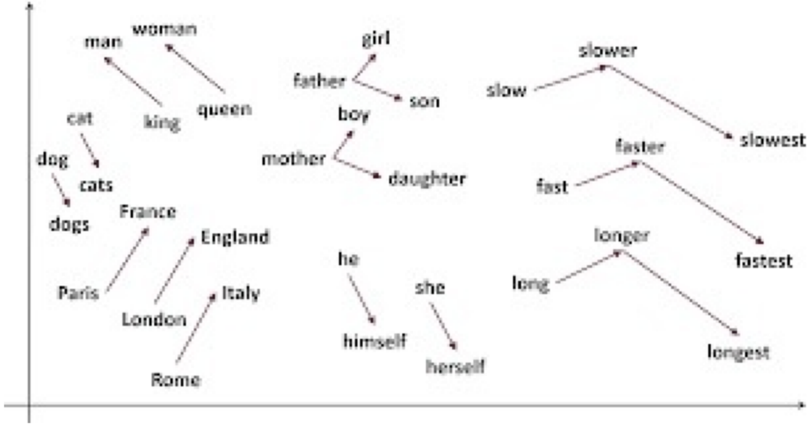
Rome = [0.91, 0.83, 0.17, …, 0.41]

Paris = [0.92, 0.82, 0.17, …, 0.98]

Italy = [0.32, 0.77, 0.67, …, 0.42]

France = [0.33, 0.78, 0.66, …, 0.97]

Towards an efficient representation of words



Rome = [0.91, 0.83, 0.17, …, 0.41]

Paris = [0.92, 0.82, 0.17, …, 0.98]

Italy = [0.32, 0.77, 0.67, …, 0.42]

France = [0.33, 0.78, 0.66, …, 0.97]

What to read

- *Distributed Representations of Words and Phrases and their Compositionality [pdf]*
- *Efficient Estimation of Word Representations in Vector Space [pdf]*
- *A Neural Probabilistic Language Model [pdf]*
- *Speech and Language Processing by Dan Jurafsky and James H. Martin is a leading resource for NLP. Word2vec is tackled in Chapter 6.*
- *Neural Network Methods in Natural Language Processing by Yoav Goldberg is a great read for neural NLP topics.*
- *Chris McCormick has written some great blog posts about Word2vec. He also just released The Inner Workings of word2vec, an E-book focused on the internals of word2vec.*
- *Want to read the code? Here are two options:*
    - Gensim’s python implementation of word2vec
    - Mikolov’s original implementation in C – better yet, this version with detailed comments from Chris McCormick.
- *Evaluating distributional models of compositional semantics*
- *On word embeddings, part 2*
- *Dune*

- *WE and NLP: (Levy and Goldberg, 2014, NIPS)*

Outline

- *Word Embeddings: word2vec*
- *Beyond Word2vec: Glove and Word Senses*
- *Gender Bias in Word Embeddings*

A Word embedding is a numerical representation of a word

- *Word embeddings allow for arithmetic operations on a text*
  - Example: time + flies
- *Word embeddings have been refered also as:*
  - Semantic Representation of Words
  - Word Vector Representation

Vector representation of flies and time

flies = (0.101159, 0.550446, 0.543801, -0.973852, -0.680835, 0.417193, -0.247181, 0.209725, -1.136055, -0.059531, -0.401640, 0.171540, 0.925121, -0.143815, 0.781714, -1.482425, 0.347008, -0.112342, 0.442418, -1.020457, -0.071752, 1.873548, -0.222886, -0.729569, -0.830224, -0.868407, 0.203496, 0.469911, -0.191363, 0.565102, 0.687738, 0.480823, 0.842358, -0.173656, -0.265585, 0.685740, 0.488047, -0.359772, -0.576064, -0.802884, 0.081554, 0.046882, -0.861532, -0.461855, 0.613098, -1.534642, -0.884534, 0.207728, 1.396512, -0.242900, -0.383959, 0.570844, -0.703350, -1.368813, -1.008194, 1.534660, 0.171693, 0.640925, -0.233116, 0.324685, 0.483171, 0.337947, -0.963290, -0.400558, 0.830977, 0.913474, 0.251693, -0.589420, -0.299622, 1.047515, -0.266679, -1.247186, 1.087610, -0.549028, 1.600710, -1.538516, -1.703301, -1.393499, -0.894448, 0.717204, 0.105767, -0.189234, -0.615609, -0.658315, 0.051877, 0.014180, -0.791282, 0.150424, 1.343751, -0.464859, 0.871426, 1.542864, -1.202150, -0.767113, -1.734738, 0.073633, -1.012583, 0.747787, 0.476070, -0.454807, 0.642685, -0.854152, -0.071798, 0.233724, 0.712329, -0.097752, -0.531132, 0.323271, -0.447342, 0.657913, 1.199492, -0.107360, -0.154234, -1.131168, 1.354793, 1.721385, -0.240023, 0.655765, -0.217006, -0.801722, 0.553369, 0.213377, 0.323267, -1.516051, 2.106244, -0.134282, 0.742155, 0.426344, 0.197991, -0.806768, 0.372546, -0.160200, -1.552847, -0.286178, -0.707796, 0.527352, -0.259658, 0.230387, 0.105294, -0.194481, 0.301772, -1.022163, 0.557191, 1.096709, 0.058422, -1.036384, 0.353412, -0.623097, -0.689515, 0.091472, 0.783885, 0.184088, -0.367950, 0.952462, 0.183704, 0.677562, 0.293917, -0.214309, -0.487794, 0.934296, 0.311513, 0.286514, -0.085511, 0.777691, 1.232603, -0.309367, -0.225086, 0.005091, -0.099195, -0.293117, 1.305563, 0.595816, 0.950316, 0.568706, -0.561446, 0.911634, -0.383941, 0.758054, -0.197820, 0.506777, -0.290767, -0.356727, 1.229474, -0.156489, -0.782741, -0.210163, -0.029169, 0.602664, 0.418375, 0.148975, -0.761796, 1.322690, -0.173410, 0.204111, -1.344531, 1.081905, -0.660543, -0.225615, -0.444753, -0.929671, 0.054136, 0.052031, -0.164926, 0.159312, -1.316333, 0.837011, -1.290353, 0.958403, 1.247478, 0.442009, 0.455497, -1.856268, -0.358823, -0.230839, -0.206271, 0.227012, -0.454163, 0.747798, -1.252855, 1.436849, -0.427915, -0.810428, -0.628144, -0.288458, 0.087355, 0.356739, 0.153036, 0.516594, -0.504978, 0.814432, 1.052940, 1.094526, -0.219595, 0.722178, 0.267325, -0.087458, -1.270262, -0.039461, 0.991926, -0.112005, -0.009605, 0.149920, 0.164717, 0.280475, 0.966384, 0.327598, 0.189590, -0.208946, 0.838261, 0.051847, -0.277932, -0.788527, -0.768702, -1.688721, 0.388215, 0.170153, -0.555723, -0.529565, -0.528982, -0.659930, 0.588041, -0.368195, -0.850188, -0.004996, 0.925476, 1.046587, -0.731761, 0.519435, 0.193188, -0.709557, 0.123329, -0.454316, 1.885830, -0.201841, -0.728933, -0.953455, -0.205837, -0.724068, 0.120158, 1.765389, -0.192159, 1.062490, -0.002634, 0.125790, -0.846565, 0.548899, -1.062821, -2.146826, 0.134681, 0.570950, 0.851783, 0.436544, 0.688986, 1.229008, 1.435449, 0.118766, -0.132411, 2.527890, 0.778142, 0.269093)

time = (1.844012, 0.590383, 1.003636, -0.577031, 1.515419, 1.097797, 1.812856, 0.933615, -2.396581, -0.931116, -0.719396, -0.376134, -1.204231, 0.045771, -0.287482, 1.084627, 4.399265, 1.516829, -0.838133, -1.881685, 0.108117, 2.345857, -1.292667, -2.286168, 3.419926, 4.260052, -1.016988, 3.140229, -3.161504, -0.800707, -1.433775, 2.290546, 1.932333, 0.714649, -3.033084, -0.958289, -1.704687, -1.597345, 1.525060, 3.337017, -2.787743, 1.479353, 3.452092, -3.242210, 0.532302, -0.551804, 2.344314, -0.919049, -1.872516, 0.080137, 1.208913, -2.136555, -2.218254, 0.206410, 0.133225, -1.521032, 1.735609, 2.885288, -2.048691, 2.375038, 0.316599, -0.254595, 2.159168, 1.118603, -0.775468, 0.933521, -0.351797, 2.193516, 2.499064, 2.818742, -0.213898, 0.446962, 1.767461, 1.342941, 1.117215, -0.042004, 4.199081, 3.041796, -1.770649, -0.528354, -2.067354, 0.283046, -0.099049, -0.105402, 2.823484, -2.583724, -2.906962, 0.592174, -3.029664, -0.170582, 0.406366, 1.963008, -3.229250, -3.499467, -0.136623, -1.551140, 0.348241, -1.597526, 0.703598, 3.122618, 0.466473, -0.113320, -2.119155, 1.092863, -0.908410, 0.253259, -1.082862, 4.408773, 2.419691, 2.343239, 0.703793, 1.270707, 0.410221, -1.293057, -0.799147, 2.214563, -0.212623, 1.206766, -0.731273, 2.308388, -1.029362, -2.080709, 0.749148, -1.412619, 1.073051, -2.498955, -0.520858, 1.391912, -1.181121, 1.523457, -1.245448, -0.290742, -2.589719, -0.366162, 3.586508, 0.908829, -1.125176, -0.937035, -1.163619, 1.759209, 3.678231, 0.019263, -0.395732, 1.142848, -0.500150, -3.005232, 2.287069, -0.524648, -0.944902, 0.038368, -1.093538, -0.697787, 0.767664, 2.399855, 2.425945, 1.563581, -1.086811, 0.372100, 1.400303, -2.278863, 0.643208, -0.459837, 1.756295, 2.057359, 3.140241, -1.740582, 1.386243, -1.822378, 1.528883, -1.984250, 1.214508, -1.336822, -0.321478, -0.162113, 0.272326, -2.673072, 0.612675, -0.657483, -0.557969, -3.358420, -2.559981, -1.683046, -1.314229, -2.425110, -2.506184, -1.606668, 1.332781, -2.760878, -2.400824, -1.830618, -2.406664,-1.169146, -1.838281, 0.588559, 2.285466, -0.401462, 1.632473, -0.510084, -2.072332, -2.627897, 2.531830, -2.524195, 2.035469, 1.906113, -1.257332, -4.039220, -0.467614, -2.275054, -3.409202, -0.014383, 0.445576, 1.461529, -1.318478, 0.061049, 0.280523, 2.173227, -0.027133, 2.791830, -0.728346, -1.804815, 1.245291, 0.970318, 2.646388, 0.246842, -1.823608, 1.888760, 0.265116, -2.027269, -0.089802, 0.389976, -0.654499, 2.565478, -2.647825, 2.658914, 1.385568, 2.306623, 0.476923, -0.869644, -0.170338, 0.495097, -2.604649, 0.610231, 0.739677, 0.322778, -2.042915, -1.353154, 0.177016, 1.840185, -0.271689, -0.401560, -0.421108, -0.185526, 1.041765, -4.599578, -0.829409, 0.076258, -0.503421, 1.891007, -0.931777, 0.434825, -0.467926, -1.417658, -0.320597, -4.084039, -3.899607, 0.977403, 0.774670, 3.269479, -1.031264, -0.433907, -2.30576 0, 0.811788, 2.347483, -1.254061, -0.861366, 0.080974, -3.666142, -0.363376, -2.384475, -4.290071, -0.924723, 1.257435, 1.223927, 0.276726, 1.541471, 1.274240, 1.883040, -1.987514, -0.809325, 1.252716, 1.812783, -0.511801, -1.657522, 1.196169, 0.804855, -1.861488, -2.113367, 0.429888, -0.920844, 0.377247)

- *How can we obtain those numbers?*
- *What’s word2vec?*
- *Is it the only way to obtain those numbers?*
- *Do the vectors (and components!) have any semantic meaning?*
- *Are we crazy by summing or multiplying words?*

Distributional Hypothesis Contextuality

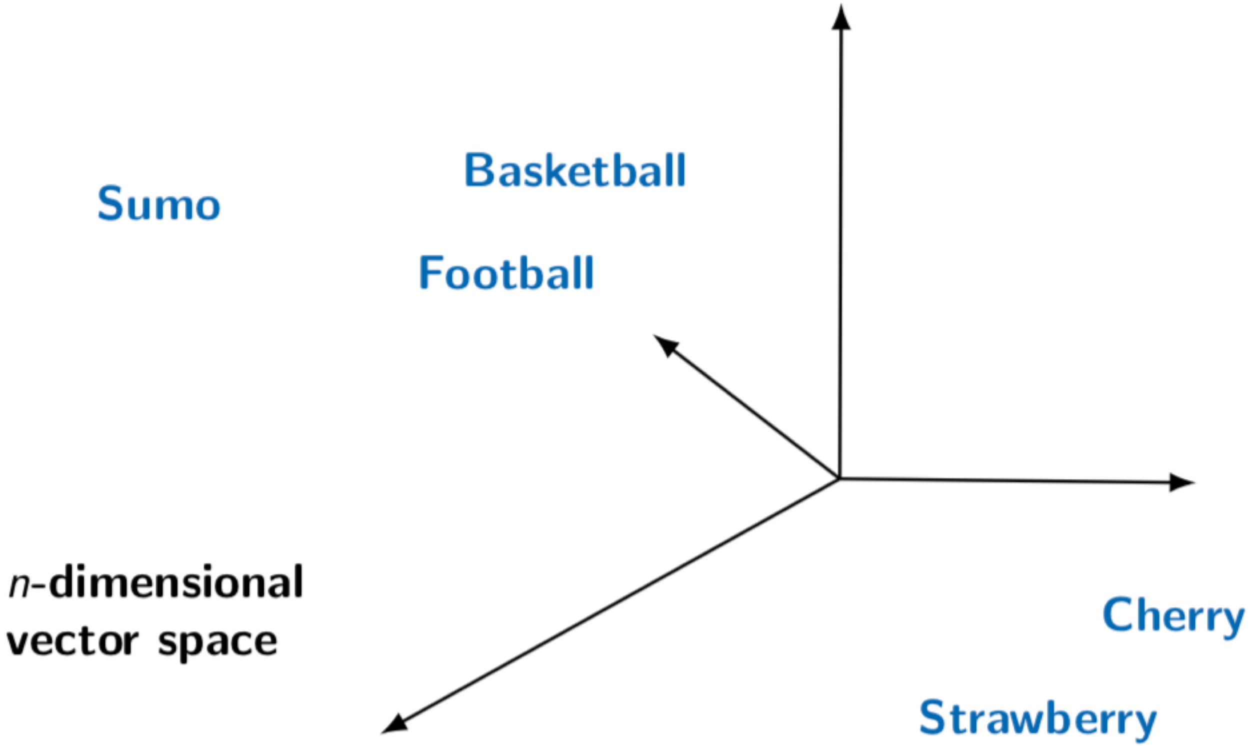
- *Never ask for the meaning of a word in isolation, but only in the context of a sentence*
  (Frege, 1884)

- *For a large class of cases... the meaning of a word is its use in the language*
  (Wittgenstein, 1953)

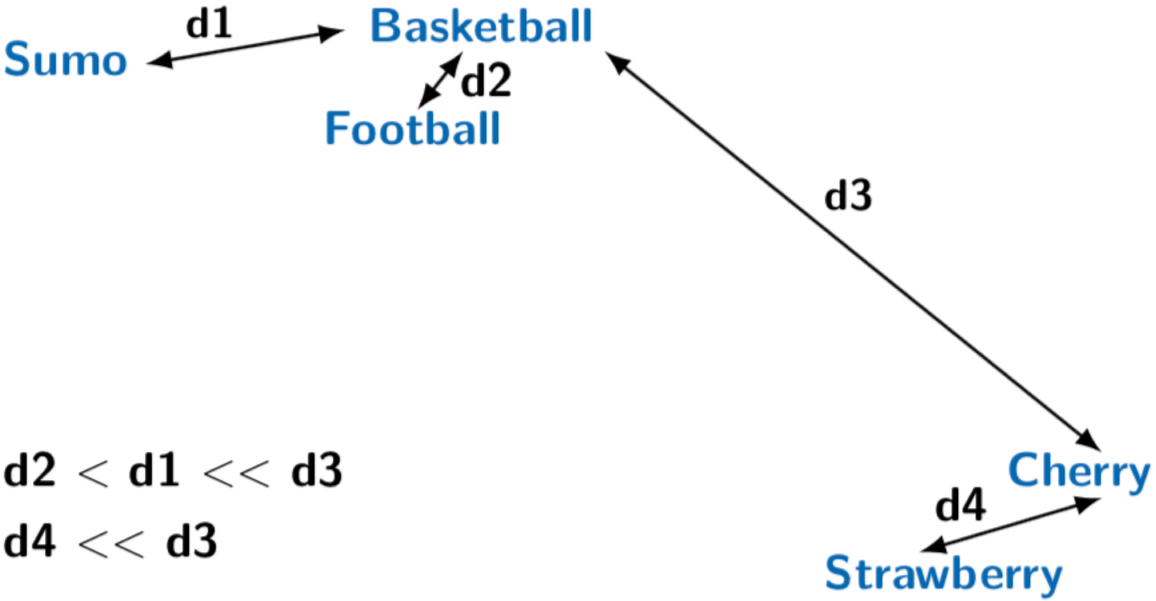
- *You shall know a word by the company it keeps (Firth, 1957)*

- *Words that occur in similar contexts tend to have similar meaning (Harris, 1954)*

Word Vector Space



Similar Meanings…



$$d2 < d1 << d3$$

$$d4 << d3$$

Background: One-hot, frequency-based, words-embeddings

- *One-hot representation*
- *Term frequency or TF-IDF methods*
- *Words embeddings*

Count-based

One-hot vectors

Two for tea and tea for two
Tea for me and tea for you
You for me and you for me

Two = [1,0,0,0]

tea=[0,1,0,0]

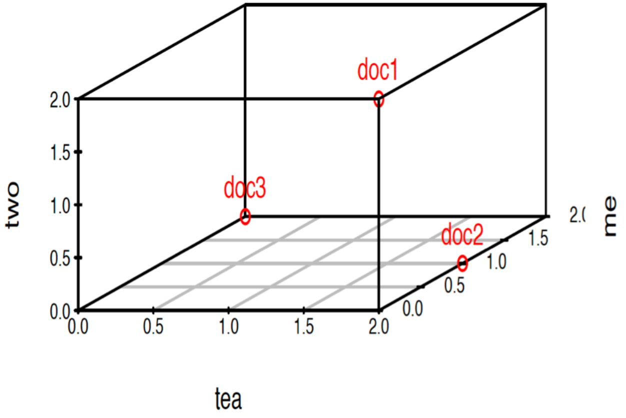
me=[0,0,1,0]

you=[0,0,0,1]

Vector Space Model: Term-document matrix

doc1	Two for tea and tea for two
doc2	Tea for me and tea for you
doc3	You for me and me for you

	two	tea	me	you
doc1	2	2	0	0
doc2	0	2	1	1
doc3	0	0	2	2



Term Frequency-Inverse Document Frecuency

Term Frequency

How frequently a term occurs in a document *d* normalised to account for *d* length

$$\mathrm{TF}(t,d) = \frac{\text{Number of times term } t \text{ appears in a document } d}{\text{Total number of terms in } d}$$

Inverse Document Frequency

Measures how important a term is (low weight for stop words)

$$\mathrm{IDF}(t, D) = \log_e\left(\frac{\text{Total number of documents } D}{\text{Number of documents with term } t \text{ in it}}\right)$$

$$\text{TF-IDF}(t, d, D) = \mathrm{TF}(t, d) \times \mathrm{IDF}(t, D)$$

Problems with simple co-occurrence vectors

Increase in size with vocabulary

Very high dimensional: requires a lot of storage

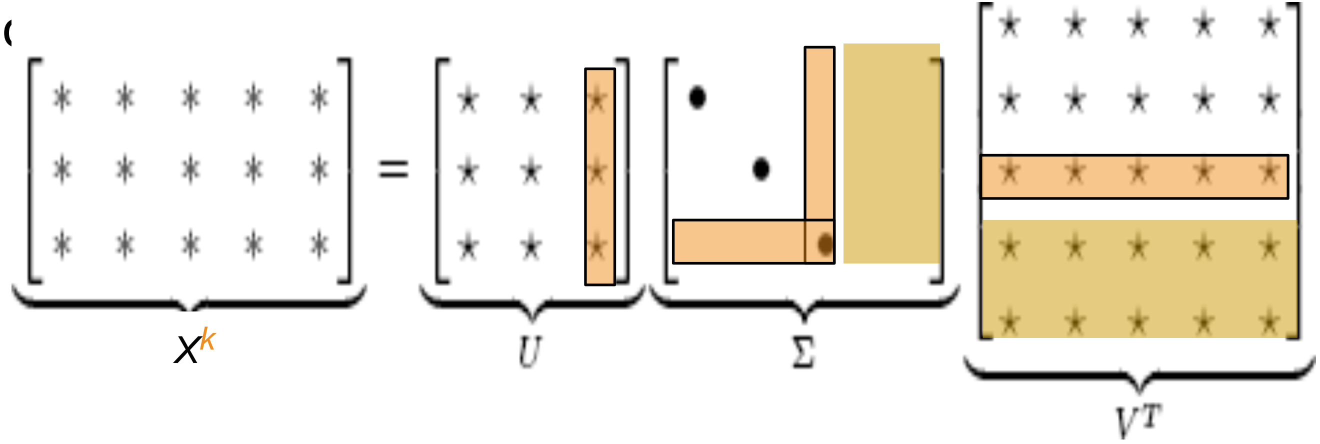
Solution: Low dimensional vectors

- Idea: store "most" of the important information in a fixed, small number of dimensions: a dense vector
- Usually 25–1000 dimensions
- How to reduce the dimensionality?

Method: Dimensionality Reduction on X (HW1)

Singular Value Decomposition of co-occurrence matrix X

Factorizes X into $U\Sigma V^{\mathsf{T}}$, where U and V are



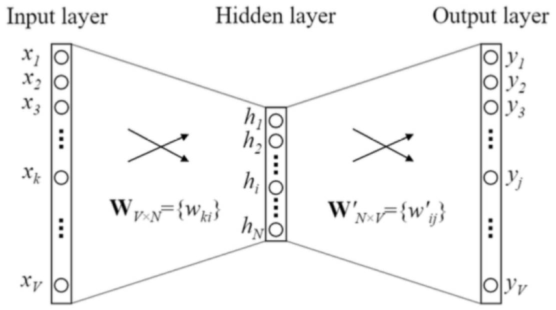
Retain only k singular values, in order to generalize.
X^k is the best rank k approximation to X , in terms of least squares. Classic linear algebra result. Expensive to compute for large matrices.

word2vec

1. *king - man* + *woman* = *queen*
2. Huge splash in NLP world
3. Learns from raw text
4. Pretty simple algorithm

Word Embeddings use simple feed-forward network

- *No deep learning at all!*
- *A hidden layer in a NN interprets the input in his own way to optimise his work in the concrete task*
- *The size of the hidden layer gives you the dimension of the word embeddings*



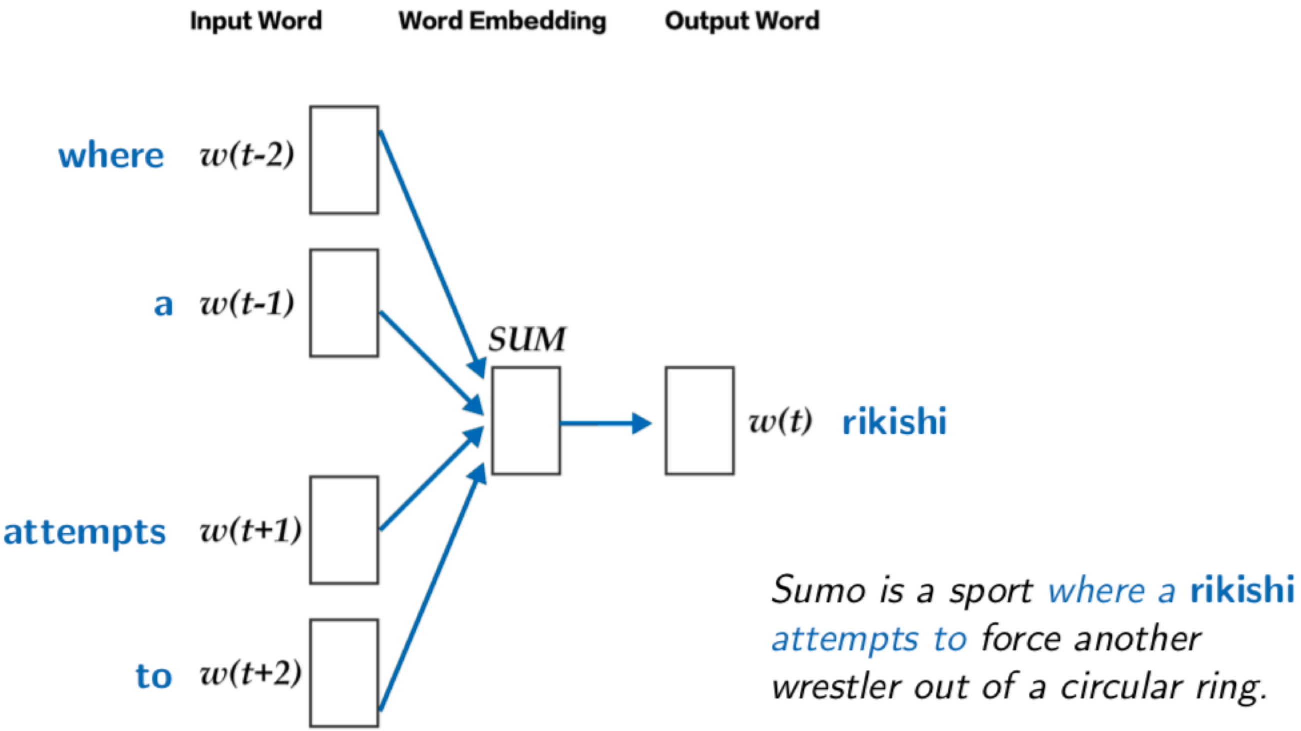
word2vec

1. Set up an objective function
2. Randomly initialize vectors
3. Do gradient descent

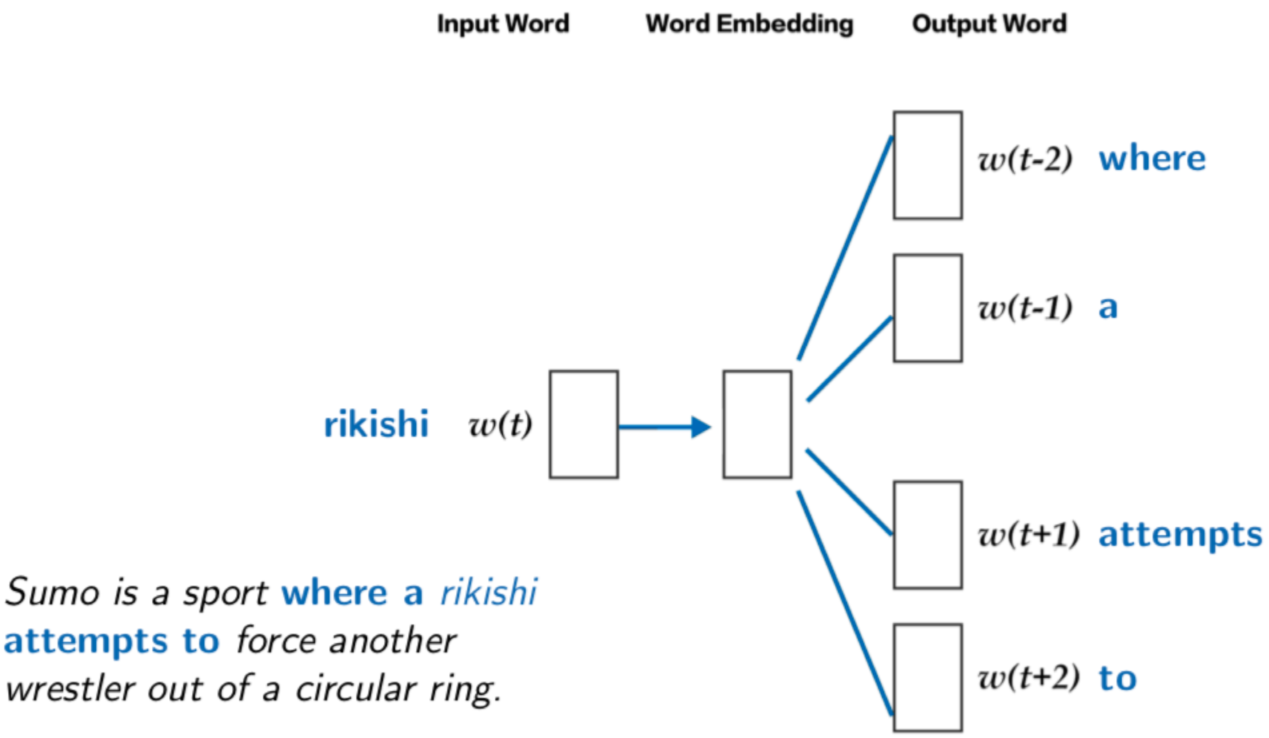
Word Embeddings learned by a neural network in two tasks/objectives:

1. predict the probability of a word given a context (CBoW)
2. predict the context given a word (skip-gram)

Continuous Bag of Words, CBoW



Skip-Gram Model



SkipGram

Guess the context
given the word

v_{IN}

"The fox jumped **over** the lazy dog"

v_{OUT} v_{OUT} v_{OUT} v_{OUT} v_{OUT} v_{OUT}

Better at syntax.
(this is the one we went over)

CBOW

Guess the word
given the context

v_{OUT}

"**The fox jumped** over **the lazy dog**"

v_{IN} v_{IN} v_{IN} v_{IN} v_{IN} v_{IN}

~20x faster.
(this is the alternative.)

Observations (Tensorflow Tutorial)

- *CBoW*
  - Smoothes over a lot of the distributional information by treating an entire context as one observation. This turns out to be a useful thing for smaller datasets
- *Skip-gram*
  - Treats each context-target pair as a new observation, and this tends to do better when we have larger datasets

"The fox jumped **over** the lazy dog"

word2vec: learn word vector from it's surrounding context

Maximize the likelihood of seeing the words given the word **over**.

$P(the|over)$
$P(fox|over)$
$P(jumped|over)$
$P(the|over)$
$P(lazy|over)$
$P(dog|over)$

…instead of maximizing the likelihood of co-occurrence counts.

Word2vec: objective function

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size *m*, given center word w : $P(v_{OUT}|v_{IN})$

$$Likelihood = L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \ne 0}} P(v_{out}|v_{in};\theta)$$

Word2vec: objective function

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size *m*, given center word w_t: $P(v_{OUT}|v_{IN})$

$$Likelihood = L(\theta) = \underbrace{\prod_{t=1}^{T}}_{\text{Loop 1}} \underbrace{\prod_{\substack{-m \leq j \leq m \\ j \neq 0}}}_{\text{Loop 2}} P(v_{out}|v_{in};\theta)$$

Twist: we have *two* vectors for every word.
Should depend on whether it's the input or the output.

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

"The fox jumped **over** the lazy dog"

$\uparrow$

v_{IN}

Loop 1: for the word ‘over’ iteration on loop 2: window around ‘over’

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

“The fox jumped **over** the lazy dog”

v_{OUT} v_{IN}

Loop 1: for the word 'over' iteration on loop 2: window around 'over'

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

"The fox jumped **over** the lazy dog"

v_{OUT} v_{IN}

Loop 1: for the word ‘over’ iteration on loop 2: window around ‘over’

A *context* window around every input word.

$$P(v_{OUT}|v_{IN})$$

“The fox jumped **over** the lazy dog”

v_{OUT} (jumped) v_{IN} (over)

Loop 1: for the word 'over' iteration on loop 2: window around 'over'

A *context* window around every input word.

$$P(v_{OUT} | v_{IN})$$

"The fox jumped **over** the lazy dog"

v_{IN} v_{OUT}

Loop 1: for the word 'over' iteration on loop 2: window around 'over'

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

"The fox jumped **over** the lazy dog"

v_{IN} v_{OUT}

Loop 1: for the word 'over' iteration on loop 2: window around 'over'

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

"The fox jumped **over** the lazy dog"

v_{IN}

v_{OUT}

Once loop 2 is finished for the word ‘over’ we move loop 1 into the following word

Loop 1: for the word ‘the’ iteration on loop 2: window around ‘the’

A *context* window around every input word.

$$P(v_{OUT}|v_{IN})$$

“The fox jumped over **the** lazy dog”

v_{IN}

Loop 1: for the word ‘the’ iteration on loop 2: window around ‘the’

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

“The fox jumped over **the** lazy dog”

$\uparrow$

v_{IN}

Loop 1: for the word 'the' iteration on loop 2: window around 'the'

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

"The fox jumped over **the** lazy dog"

v_{OUT}

v_{IN}

Loop 1: for the word ‘the’ iteration on loop 2: window around ‘the’

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

“The fox jumped over **the** lazy dog”

v_{OUT} v_{IN}

Loop 1: for the word ‘the’ iteration on loop 2: window around ‘the’

A *context* window around every input word.

$$P(v_{OUT}|v_{IN})$$

“The fox jumped over **the** lazy dog”

v_{OUT} (under "jumped")

v_{IN} (under "the")

Loop 1: for the word 'the' iteration on loop 2: window around 'the'

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

"The fox jumped over **the** lazy dog"

v_{OUT} v_{IN}

Loop 1: for the word ‘the’ iteration on loop 2: window around ‘the’

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

“The fox jumped over **the** lazy dog”

v_{IN} v_{OUT}

Loop 1: for the word 'the' iteration on loop 2: window around 'the'

A *context* window around every input word.

$P(v_{OUT}|v_{IN})$

"The fox jumped over **the** lazy dog"

v_{IN} $\quad$ v_{OUT}

Word2vec: objective function

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size *m*, given center word w: $P(v_{OUT}|v_{IN})$

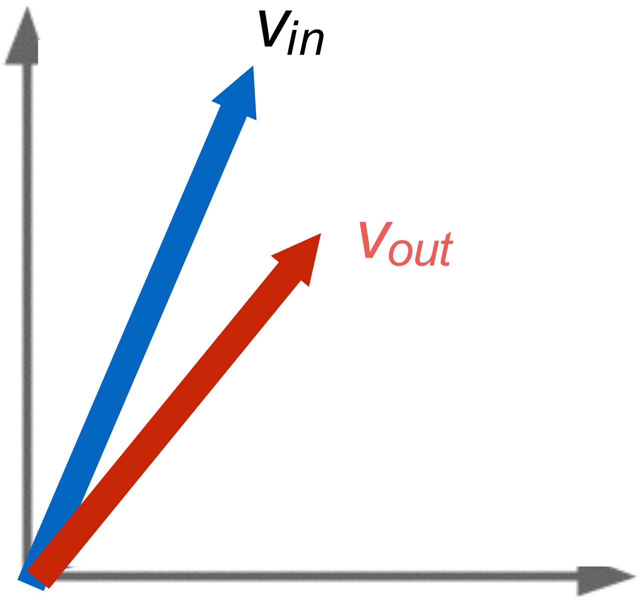
$$Likelihood = L(\theta) = \underbrace{\prod_{t=1}^{T}}_{\text{Loop 1}} \underbrace{\prod_{\substack{-m \le j \le m \\ j \ne 0}}}_{\text{Loop 2}} P(v_{out}|v_{in};\theta)$$

How should we define $P(v_{OUT}|v_{IN})$?

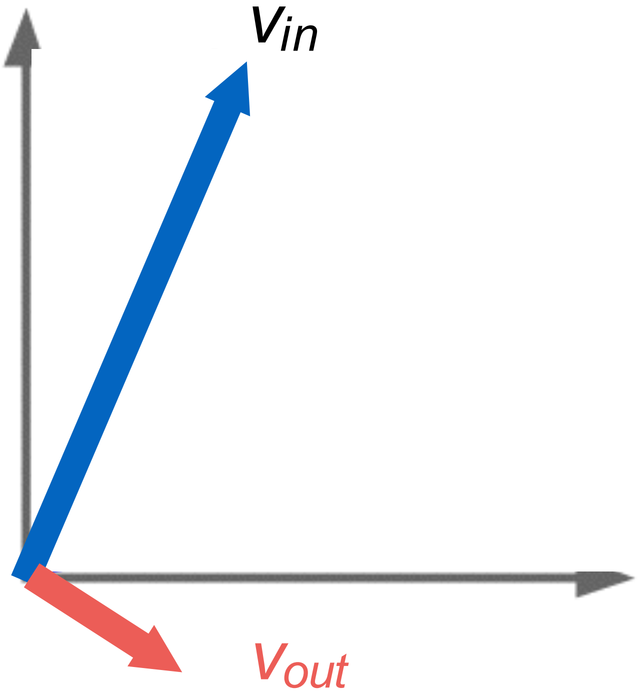
Measure loss between v_{IN} and v_{OUT}?

$$P(v_{out}|v_{in};\theta)$$

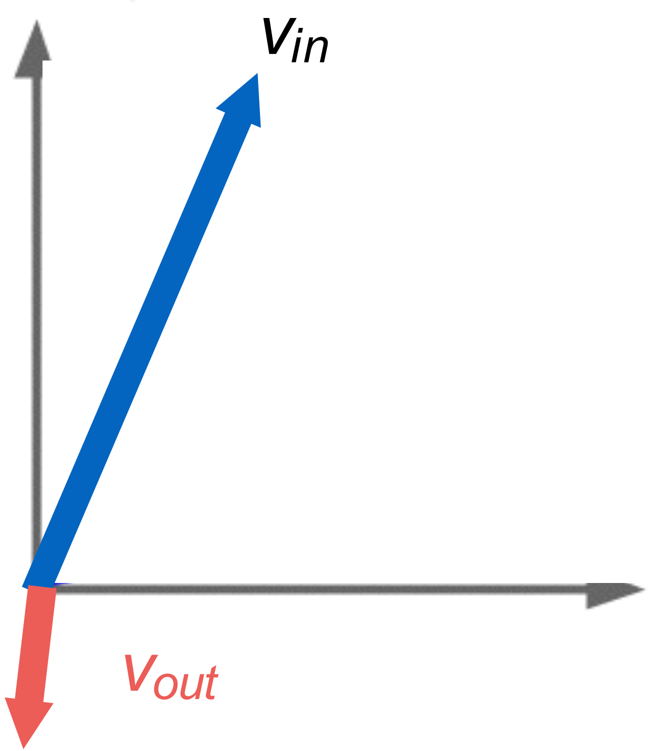
$$V_{in} \cdot V_{out}$$



$$V_{in} \cdot V_{out} \sim 1$$



$$v_{in} \cdot v_{out} \sim 0$$



$v_{in} \cdot v_{out} \sim -1$

But we'd like to measure a probability.

$$v_{in} \cdot v_{out} \in [-1,1]$$

But we'd like to measure a probability.

Dot product compares similarity of vout and vin
Larger dot product = larger probability

Exponentiation makes anything positive

$$\frac{\exp(v_{in} \cdot v_{out})}{\sum_{k \in V} \exp(v_{in} \cdot v_k)} = P(v_{out} | v_{in})$$

Normalize over entire vocabulary
to give probability distribution

But we'd like to measure a probability.

$$\frac{exp(v_{in} \cdot v_{out})}{\sum_{k \in V} exp(v_{in} \cdot v_k)} = P(v_{out}|v_{in})$$

- This is an example of the **softmax function** $\mathbb{R}^n \rightarrow \mathbb{R}^n$

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i$$

- The softmax function maps arbitrary values x_i to a probability distribution p_i
  - "max" because amplifies probability of largest x_i
  - "soft" because still assigns some probability to smaller x_i
  - Frequently used in Deep Learning

Summary of the process

Thou shalt not make a machine in the likeness of a human mind

Input
Features

not

thou

Untrained model
Task: are the two words neighbours?

1) Look up embeddings

V_{IN}

V_{OUT}

2) Calculate prediction

3) Project to output vocabulary

$$P(V_{OUT}|V_{IN})$$

$$softmax(v_{in} \cdot v_{out})$$

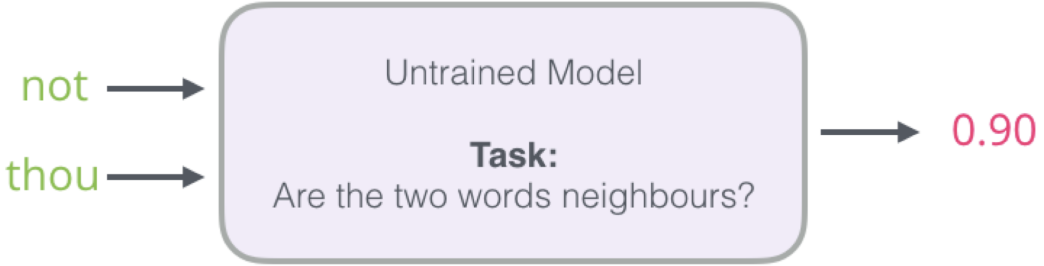
Output
Prediction

input word	output word	target
not	thou	1
not	shalt	1
not	make	1
not	a	1
make	shalt	1
make	not	1
make	a	1
make	machine	1

Let’s glance at how we use it to train a basic model that predicts if two words appear together in the same context.

We start with the first sample in our dataset. We grab the feature and feed to the untrained model asking it to predict if the words are in the same context or not (1 or 0)

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness



Preliminary steps: Negative examples

This can now be computed at blazing speed – processing millions of examples in minutes. But there's one loophole we need to close. If all of our examples are positive (target: 1), we open ourself to the possibility of a smartass model that always returns 1 – achieving 100% accuracy, but learning nothing and generating garbage embeddings.

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine

input word	output word	target
not	thou	1
not	shalt	1
not	make	1
not	a	1
make	shalt	1
make	not	1
make	a	1
make	machine	1

Preliminary steps: Negative examples

*For each sample in our dataset, we add **negative examples**. Those have the same input word, and a 0 label.*

input word	output word	target
not	thou	1
not		0
not		0
not	shalt	1
not	make	1

Negative examples

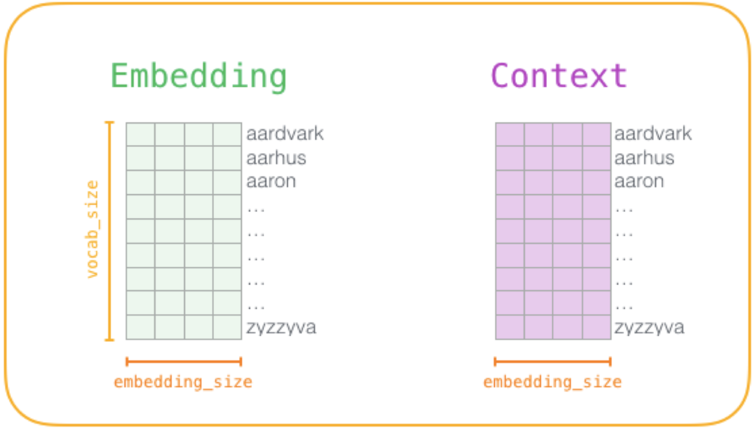
We are contrasting the actual signal (positive examples of neighboring words) with noise (randomly selected words that are not neighbors). This leads to a great tradeoff of computational and statistical efficiency.

Preliminary steps: pre-process the text

Now that we've established the two central ideas of skipgram and negative sampling, one last preliminary step is we ***pre-process the text*** *we're training the model against. In this step, we determine the* ***size of our vocabulary*** *(we'll call this vocab_size, think of it as, say, 10,000) and which words belong to it.*

Now that we've established the two central ideas of skipgram and negative sampling and pre-process, we can proceed to look closer at the actual word2vec training process.

*At the start of the training phase, we create **two matrices** – an Embedding matrix and a Context matrix. These two matrices have an **embedding for each word** in our vocabulary (So vocab_size is one of their dimensions). The seconddimension is how long we want embedding to be (**embedding_size** – 300 is a common value*



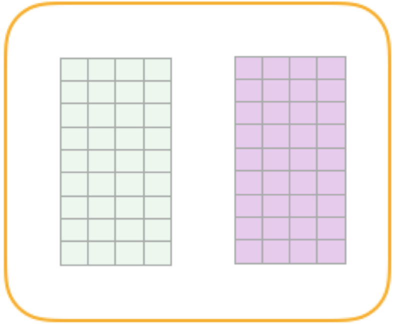
Training process: matrix initialization

*1. At the start of the training process, we **initialize** these matrices with **random values**. Then we start the training process. In each training step, **we take one positive example and its associated negative examples**. Let's take our first group:*

dataset

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	mango	0
not	finglonger	0
not	make	1
not	plumbus	0
...	...	...

model



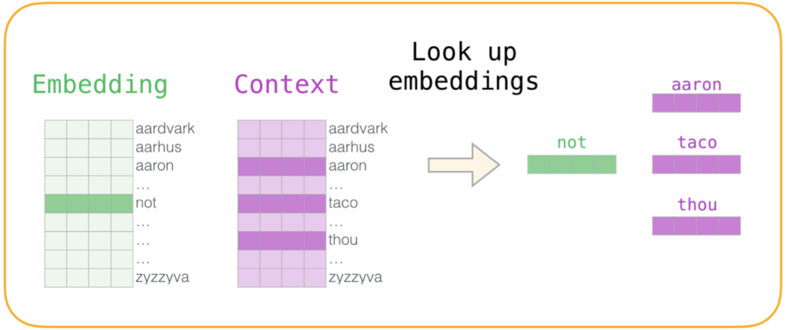
Training process

2. *Now we have four words:*

- the input word *not*
- the output/context words (1-Word window):

  thou (the actual neighbor), aaron, and taco (the negative examples).

We proceed to ***look up their embeddings*** *– for the input word, we look in the Embedding matrix. For the context words, we look in the Context matrix (even though both matrices have an embedding for every word in our vocabulary)..*



*3. Then, we take the **dot product** of the input embedding with each of the context embeddings. In each case, that would result in a number, that number indicates the similarity of the input and context embeddings*

*4. Now we need a way to **turn these scores into something that looks like probabilities** – we need them to all be positive and have values between zero and one. This is a great task for [sigmoid](), the [logistic operation](). And we can now treat the output of the sigmoid operations as the model's output for these examples.*

input word	output word	target	input • output	sigmoid()
not	thou	1	0.2	0.55
not	aaron	0	-1.11	0.25
not	taco	0	0.74	0.68

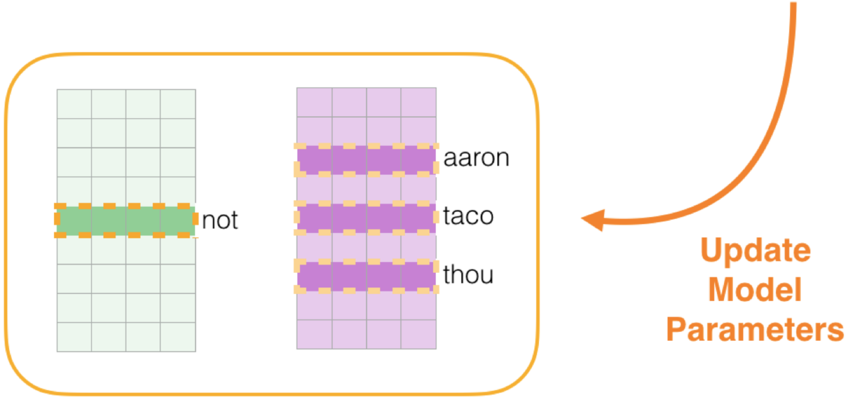
*5. Now that the untrained model has made a prediction, and seeing as though we have an actual target label to compare against, let's calculate **how much error** is in the model's prediction. To do that, we just subtract the sigmoid scores from the target labels.*

input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68

Training process

6. Here comes the "learning" part of "machine learning". We can now use this error score to ***adjust the embeddings*** *of not, thou, aaron, and taco so that the next time we make this calculation, the result would be closer to the target scores*

input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68



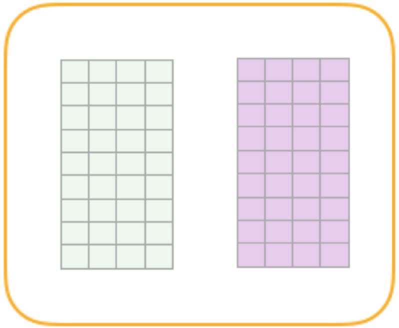
Training process

7. This concludes the training step. We emerge from it with slightly better embeddings for the words involved in this step (not, thou, aaron, and taco). We now proceed ***to our next step*** *(the next positive sample and its associated negative samples) and do the same process again.*

dataset

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	mango	0
not	finglonger	0
not	make	1
not	plumbus	0
...	...	...

model



*8. The embeddings **continue to be improved while we cycle through our entire dataset** for a number of times. We can then stop the training process, discard the Context matrix, and use the Embeddings matrix as our pre-trained embeddings for the next task.*

Gradient Descent

We go through gradients for each center vector Vin in a window. We also need gradients for outside vectors Vout

But Corpus may have 40B tokens and Windows you would wait a very long time before making a single update!

Stochastic Gradient Descent

We will update parameters after each samples of corpus sentences (what is called batches) → Stochastic gradient descent (SGD)
and update weights after each one

- *Word Embedding Visual Inspector, wevi*

  https://ronxin.github.io/wevi/

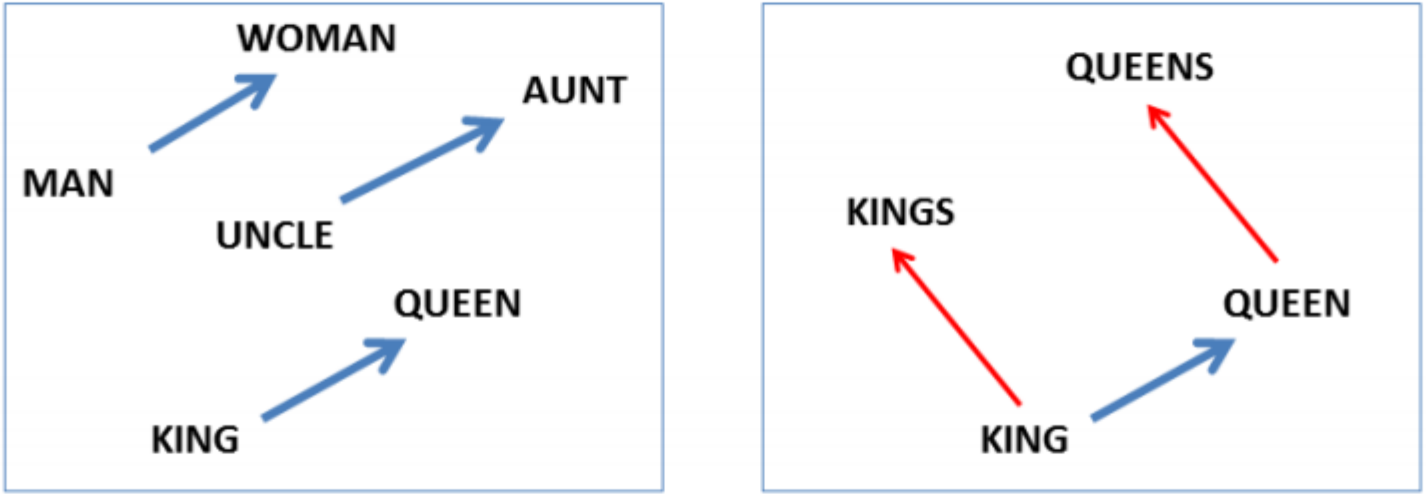
- *Gensim*

  [*http://web.stanford.edu/class/cs224n/materials/Gensim%20word%20vector%20visualization.html*](http://web.stanford.edu/class/cs224n/materials/Gensim%20word%20vector%20visualization.html)

- *Embedding Projector*

  http://projector.tensorflow.org/

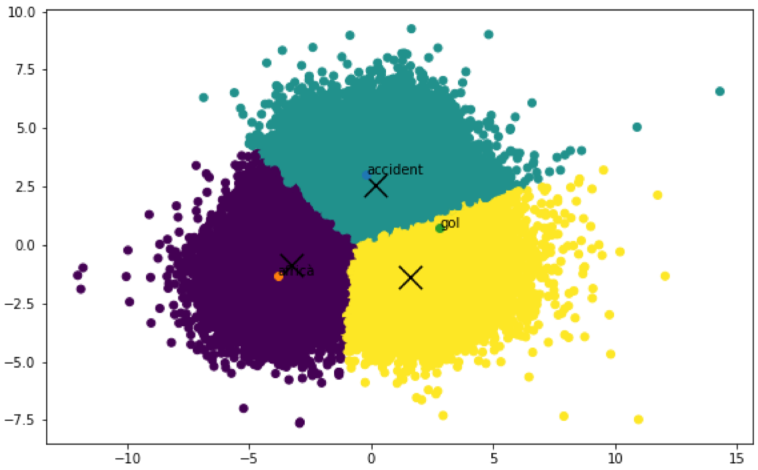
- *King-Man + Woman = Queen*



(Mikolov et al., NAACL HLT, 2013)

Word2vec in Vikipedia

‘dimecres’ + (‘dimarts’ – ‘dilluns’) = ‘dijous’
‘tres’ + (‘dos’ – ‘un’) = ‘quatre’
‘tres’ + (‘2’ – ‘dos’) = ‘3’
‘viu’ + (‘coneixia’ – ‘coneix’) = ‘vivia’
‘la’ + (‘els’ – ‘el’) = ‘les’
‘Polònia’ + (‘francès’ – ‘França’) = ‘polonès’



GloVe and Words Senses

Frequency based vs. direct prediction

- *LSA, HAL (Lund & Burgess),*
- *COALS, Hellinger-PCA (Rohde et al, Lebret & Collobert)*

- *Fast training*
- *Efficient usage of statistics*
- *Primarily used to capture word similarity*
- *Disproportionate importance given to large counts*

- *Skip-gram/CBOW (Mikolov et al)*
- *NNLM, HLBL, RNN (Bengio et al; Collobert & Weston; Huang et al; Mnih & Hinton)*

- *Scales with corpus size*
- *Inefficient usage of statistics*
- *Generate improved performance on other tasks*
- *Can capture complex patterns beyond word similarity*

Combines the advantages of the two major model families in the literature: global matrix factorization and local context window methods

The model efficiently leverages statistical information by training only on the nonzero elements in a word-word co-occurrence matrix rather than on the entire sparse matrix or on individual context windows in a large corpus

Ratios of co-occurrence probabilities can encode meaning components

	x = solid	x = gas	x = water	x = random
$P(x \mid \text{ice})$	large	small	large	small
$P(x \mid \text{steam})$	small	large	large	small
$\frac{P(x \mid \text{ice})}{P(x \mid \text{steam})}$	large	small	~1	~1

	x = solid	x = gas	x = water	x = fashion
$P(x \mid \text{ice})$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(x \mid \text{steam})$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$\frac{P(x \mid \text{ice})}{P(x \mid \text{steam})}$	8.9	8.5×10^{-2}	1.36	0.96

Q: How can we capture ratios of co-occurrence probabilities as linear meaning components in a word vector space?

A: Log-bilinear model: $$w_i \cdot w_j = \log P(i|j)$$

with vector differences $$w_x \cdot (w_a - w_b) = \log \frac{P(x|a)}{P(x|b)}$$

GloVe does this by setting a function that represents ratios of co-occurrence probabilities rather than the probabilities themselves

$$w_i \cdot w_j = \log P(i|j)$$

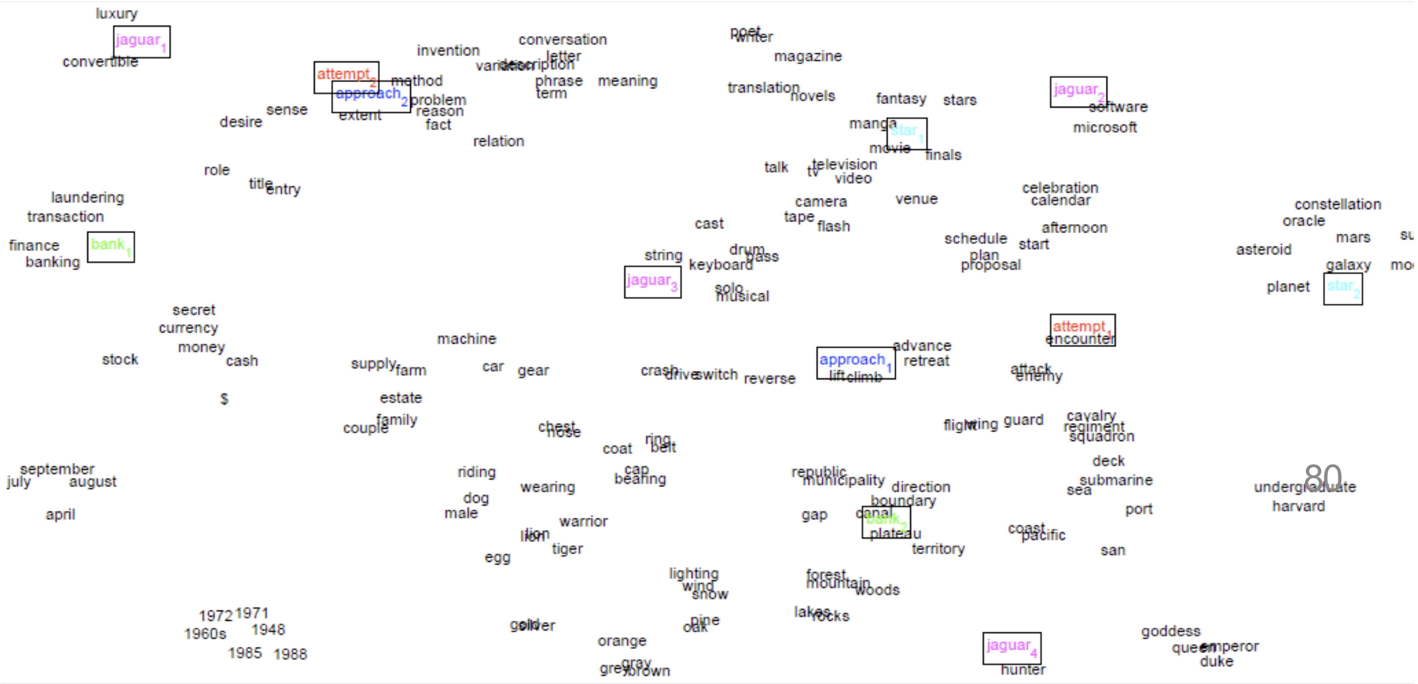
$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

- Fast training
- Scalable to huge corpora
- Good performance even with small corpus and small vectors

- *Most words have lots of meanings!*
  - Especially common words
  - Especially words that have existed for a long time

Improving Word Representations Via Global Context And Multiple Word Prototypes (Huang et al. 2012)

- Idea: Cluster word windows around words, retrain with each word assigned to multiple different clusters $bank_1$, $bank_2$, etc



Linear Algebraic Structure of Word Senses, with application to polysemy

(Arora, …,Ma, …,TACL2018)

- Different senses of a word reside in a linear superposition (weighted sum) in standard word embeddings like word2vec
  - $v_{\text{pike}} = \alpha_1 v_{\text{pike}_1} + \alpha_2 v_{\text{pike}_2} + \alpha_3 v_{\text{pike}_3}$
  - Where: $\alpha_1 = \frac{f_1}{f_1+f_2+f_3}$, etc., for frequency *f*
  - Surprising result:
    - Because of ideas from *sparse coding* you can actually separate out the senses (providing they are relatively common)

tie				
trousers	season	scoreline	wires	operatic
blouse	teams	goalless	cables	soprano
waistcoat	winning	equaliser	wiring	mezzo
skirt	league	clinching	electrical	contralto
sleeved	finished	scoreless	wire	baritone
pants	championship	replay	cable	coloratura

Not so nice…

Man is to computer programmer as woman is to ….

homemaker

Gender bias in words embeddings

A man *and* ***his son*** *are in* ***a terrible accident*** *and are rushed to the hospital in critical care.*

The ***doctor*** *looks at the boy and exclaims "I* ***can't operate*** *on this boy, he's my* ***son****!"*

How could this be?

“Doctor” vs “Female doctor”

Related Work: Word Embeddings encode bias

[credits to Hila Gonen]

[Caliskan et al. 2017] replicate a spectrum of biases from using word embeddings, showing text corpora contain several types of biases:

- morally neutral as toward insects or flowers
- problematic as toward race or gender ,
- reflecting the distribution of gender with respect to careers or first names

Concepts 1	Concepts 2	Attributes 1	Attributes 2
Flowers: buttercup, daisy, lily	**Insects**: ant, caterpillar, flea	**Pleasant**: freedom, health, love	**Unpleasant**: abuse, crash, filth
European American names: Brad, Brendan	**African American names**: Darnell, Lakisha	**Pleasant**: joy, love, peace	**Unpleasant**: agony, terrible
Male attributes: male, man, boy	**Female attributes**: female, woman, girl	**Math words**: math, algebra, geometry	**Arts Words**: poetry, art, dance

Techniques to Debias Word Embeddings

(1) Debias **After** Training [Bolukbasi et al. 2016] ---> Debias WE

Define a gender direction

Define inherently neutral words (nurse as opposed to mother)

Zero the projection of all neutral words on the gender direction

Remove that direction from words

(2) Debias **During** Training [Zhao et al. 2018] ---> GN-Glove

Train word embeddings using GloVe (Pennington et al., 2014)

Alter the loss to encourage the gender information to concentrate in the last coordinate (use two groups of male/female seed words, and encourage words from different groups to differ in their last coordinate)

To ignore gender information –simply remove the last coordinate

Three experiments *were carried out in our evaluation:*

1. *Detecting the gender space and the Direct bias*
2. *Male and female biased words clustering*
3. *Classification approach of biased words*

Our comparison is based on pre-trained sets of all these options. For experiments, we use the English-German news corpus from WMT18

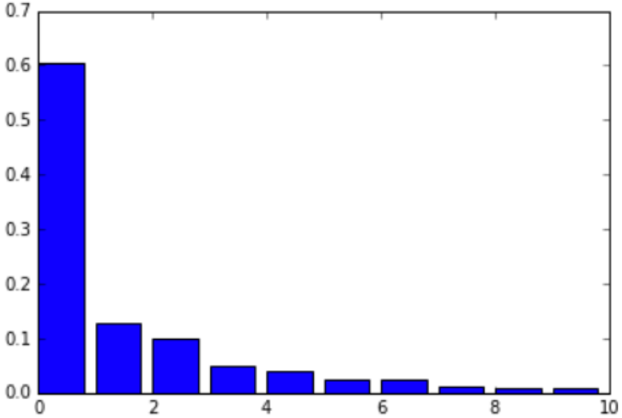
Lists for Definitional, Biased and Professional Terms

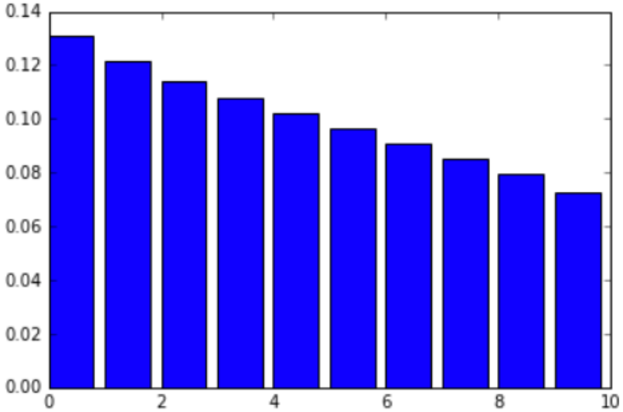
- ***Definitional List*** *10 pairs (e.g. he-she, man-woman, boy-girl)*
- ***Biased List****, which contains of 1000 words, 500 female biased and 500 male biased. (e.g. diet for female and hero for male)*
- ***Extended Biased List****, extended version of Biased List. (5000 words, 2500 female biased and 2500 male biased)*
- ***Professional List*** *319 tokens (e.g. accountant, surgeon)*

1. *Randomly sampling sentences that contain words from the Definitional List, swap the definitional word with its pair-wise equivalent from the opposite gender.*
2. *Get word embeddings for the word and its swapped equivalence, compute their difference.*
3. *On the set of difference vectors, we compute their principal components to verify the presence of bias.*
4. *Repeat for an equivalent list of random words (skipping the swapping).*

1. Gender Space and Direct Bias

Percentage of variance in PCA: definitional vs random





(Left) Percentage of variance explained in the PCA of definitional vector differences.
(Right) The corresponding percentages for random vectors

***Direct Bias** is a measure of how close a certain set of words are to the gender vector.*
Computed on list of professions.

$$\frac{1}{|N|}\sum_{w \epsilon N} |cos(\vec{w}, g)|$$

	Direct Bias
WE	0.08

k-means

Generate 2 clusters of the embeddings of tokens from the ***Biased list*** *(e.g. diet for female and hero for male)*

	Accuracy
WE	99.9%

Debias WE	92.5%
GN-WE	85.6%

SVM

*Classify **Extended Biased List** into words associated between male and female*

1000 for training, 4000 for testing

	Accuracy
WE	98.25%
Debias WE	88.88%
GN-WE	98,65%

Conclusions. Is Debiasing What We Want?

Word Embeddings exhibit Gender Biases

Difficult to scale to different forms of bias

Is debiasing even (always) desirable?

- ML is about learning biases. Removing attributes removes information.
- Gender information in NLP systems becomes harmful when the use of the system has a negative impact on people’s lives.

Gender bias is a social phenomenon that can’t be solved with mathematical methods alone. Collaborate with social sciences/sociolinguistics.

Arguments for Doing Research in Gender Bias

Unconscious bias can be harmful

Debiasing computer systems may help in debiasing society

Gender bias causes NLP systems to make errors. You should care about this even if accuracy is all you care about.