Dan Jurafsky and James Martin

Speech and Language Processing

Chapter 6:
Vector Semantics, Part II

Distributional similarity based representations

You can get a lot of value by representing a word by means of its neighbors

"You shall know a word by the company it keeps"

(J. R. Firth 1957: 11)

One of the most successful ideas of modern statistical NLP

…government debt problems turning into	***banking***	*crises as happened in 2009…*
…saying that Europe needs unified	***banking***	*regulation to replace the hodgepodge…*
…India has just given its	***banking***	*system a shot in the arm…*

Solution: Low dimensional vectors

The number of topics that people talk about is small (in some sense)

- Clothes, movies, politics, …

- 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?
  - Go from big, sparse co-occurrence count vector to low dimensional "word embedding"

Idea: Directly learn low-dimensional word vectors based on ability to predict

- Old idea: Learning representations by back-propagating errors. (Rumelhart et al., 1986)
- A neural probabilistic language model (Bengio et al., 2003)
- NLP (almost) from Scratch (Collobert & Weston, 2008)
- A recent, even simpler and faster model: word2vec (Mikolov et al. 2013) → intro now
- The GloVe model from Stanford (Pennington, Socher, and Manning 2014) connects back to matrix factorization
- Per-token representations: Deep contextual word representations: ELMo, ULMfit, **BERT**

Non-linear and slow

Fast bilinear models

Current state of the art

Tf-idf and PPMI are sparse representations

tf-idf and PPMI vectors are

- **long** (length |V|= 20,000 to 50,000)
- **sparse** (most elements are zero)

Alternative: dense vectors

vectors which are

- **short** (length 50-1000)
- **dense** (most elements are non-zero)

Sparse versus dense vectors

Why dense vectors?

- Short vectors may be easier to use as **features** in machine learning (less weights to tune)
- Dense vectors may **generalize** better than storing explicit counts
- They may do better at capturing synonymy:
  - *car* and *automobile* are synonyms; but are distinct dimensions
    - a word with *car* as a neighbor and a word with *automobile* as a neighbor should be similar, but aren't
- **In practice, they work better**

Dense embeddings you can download!

Word2vec (Mikolov et al.)
https://code.google.com/archive/p/word2vec/

Fasttext http://www.fasttext.cc/

Glove (Pennington, Socher, Manning)
http://nlp.stanford.edu/projects/glove/

Word2vec

Popular embedding method

Very fast to train

Code available on the web

Idea: **predict** rather than **count**

Word2vec

- Instead of **counting** how often each word *w* occurs near "*apricot*"
- Train a classifier on a binary **prediction** task:
  - Is *w* likely to show up near "*apricot*"?

- We don’t actually care about this task
  - But we'll take the learned classifier weights as the word embeddings

Brilliant insight: Use running text as implicitly supervised training data!

- A word *s* near *apricot*
  - Acts as gold 'correct answer' to the question
  - "Is word *w* likely to show up near *apricot*?"
- No need for hand-labeled supervision
- The idea comes from **neural language modeling**
  - Bengio et al. (2003)
  - Collobert et al. (2011)

BENGIO, YOSHUA, ET AL. "A NEURAL PROBABILISTIC LANGUAGE MODEL." JOURNAL OF MACHINE LEARNING RESEARCH 3.FEB (2003): 1137-1155.

Word2vec is a family of algorithms

[Mikolov et al. 2013]

Predict between every word and its context words!

Two algorithms

1. **Skip-grams (SG)**

   Predict context words given target (position independent)

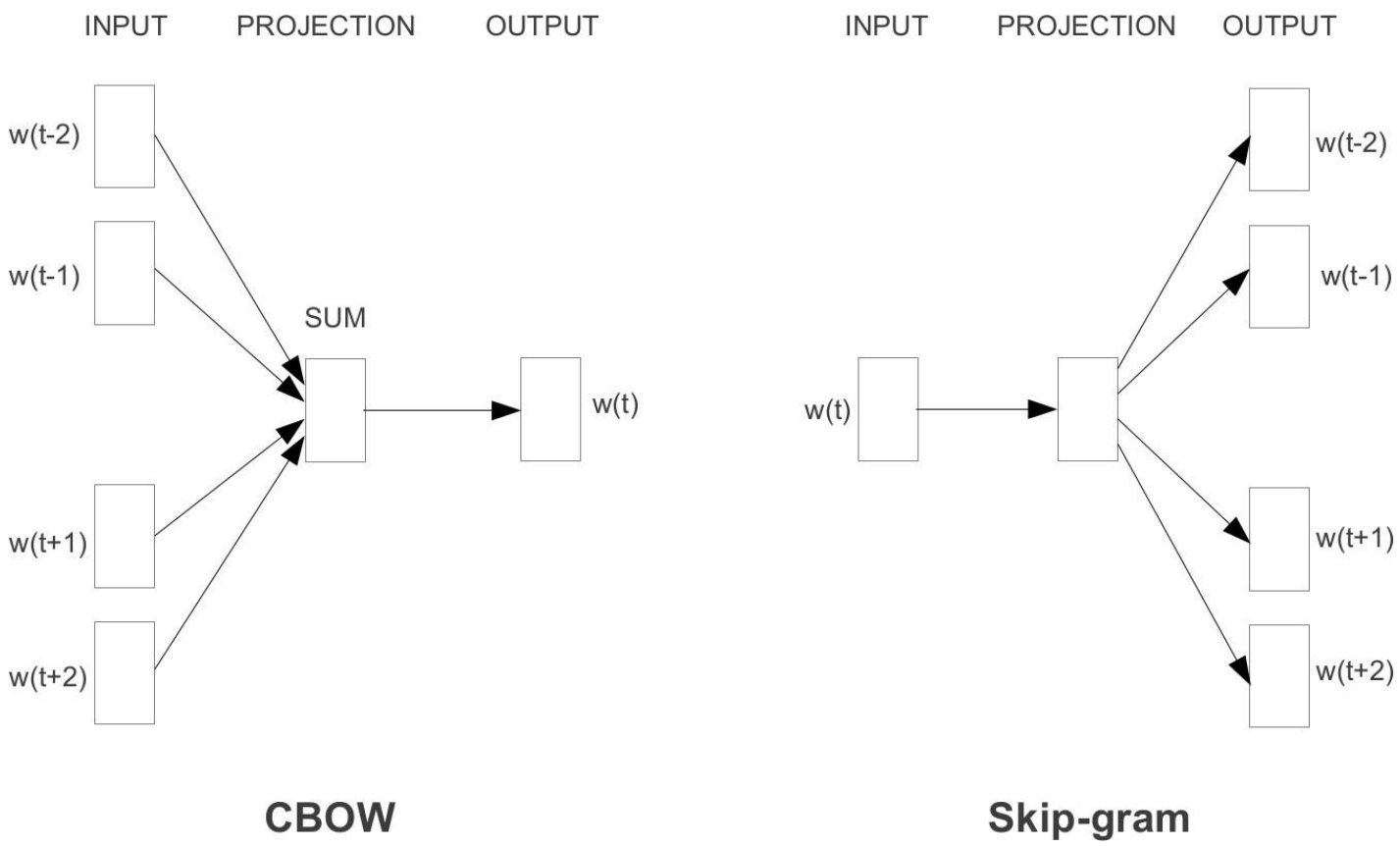
2. Continuous Bag of Words (CBOW)

   Predict target word from bag-of-words context

Two (moderately efficient) training methods

1. Hierarchical softmax
2. **Negative sampling**
3. Naïve softmax

Word2vec CBOW (Continuous Bag Of Words) and Skip-gram network architectures



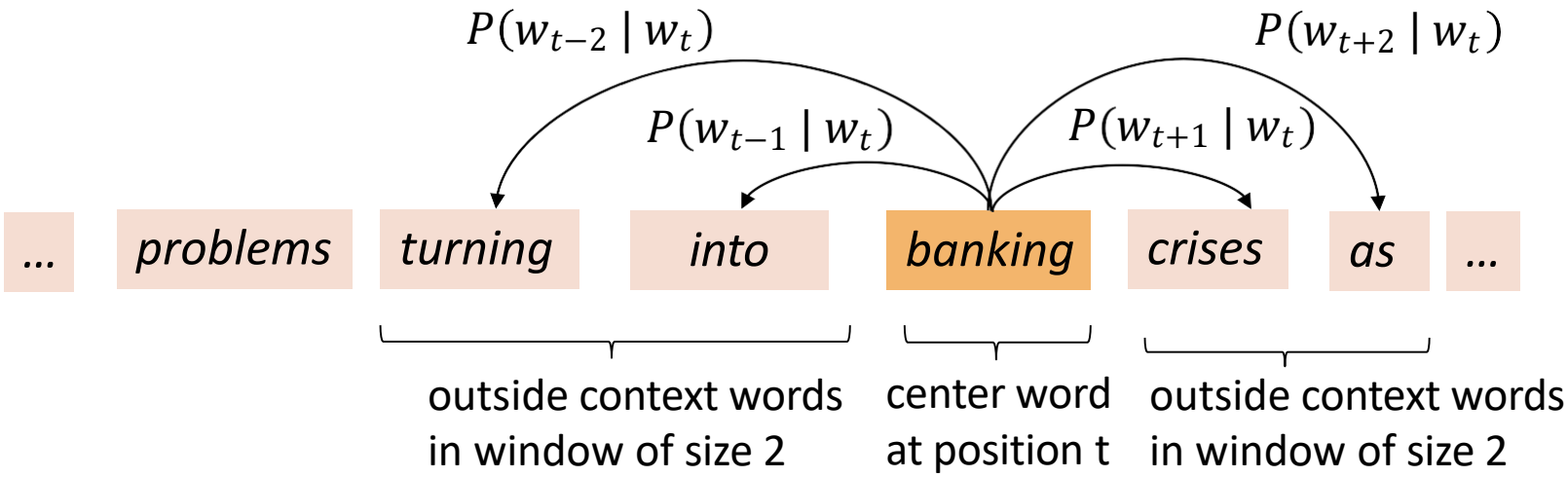
Word2Vec: Skip-Gram Task

Word2vec provides a variety of options. Let's do

- "skip-gram with negative sampling" (SGNS)

Word2Vec Skip-gram Overview

Example windows and process for computing $P\left(w_{t+j} \mid w_t\right)$



WordToVec algorithm

1. Treat the target word and a neighboring context word as positive examples.
2. Use other words not in context as negative samples
3. Use a model to train a classifier to distinguish those two cases
4. Use the learned weights (parameters) of classifier as the embeddings

Skip-Gram Training Data

Training sentence:

... lemon, a tablespoon of **apricot** jam a pinch ...

c1 c2 target c3 c4

Asssume context words are those in +/- 2 word window

WordToVec Goal

Given a tuple (t,c) = target, context

- (*apricot*, *jam*)
- (*apricot*, *aardvark*)

Return probability that c is a real context word:

$P(+|t,c)$

$P(-|t,c) = 1-P(+|t,c)$

How to compute p(+|t,c)?

Intuition:

- Words are likely to appear near similar words
- Model similarity with dot-product!
- Similarity(t,c) $\propto t \cdot c$

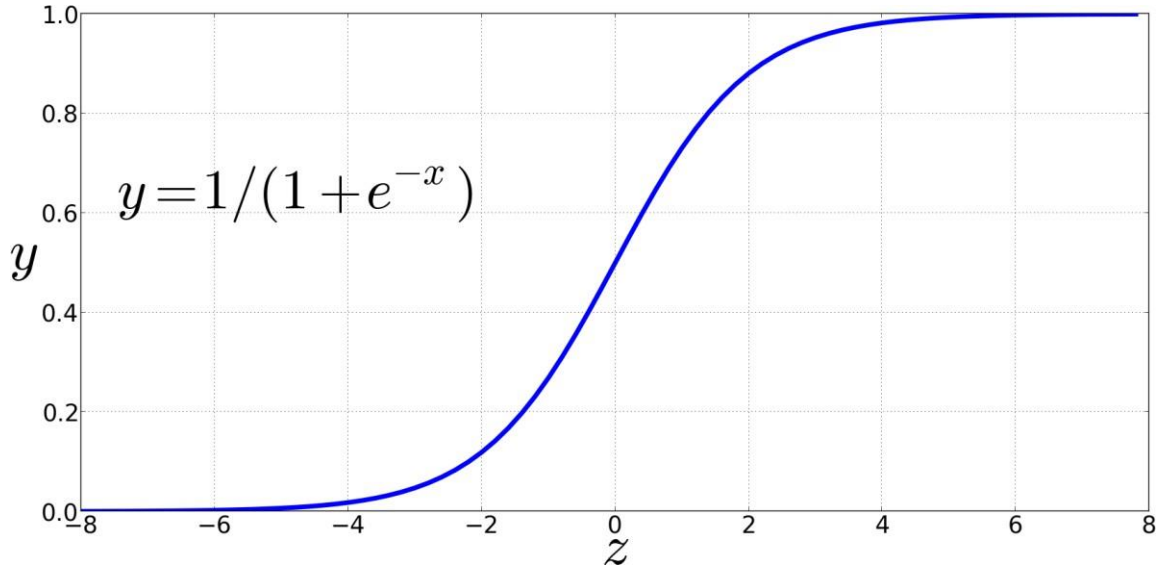
Problem:

- *Dot product is not a probability!*
  - *(Neither is cosine)*

Turning dot product into a probability

The sigmoid lies between 0 and 1:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



Turning dot product into a probability

$$P(+|t,c) = \frac{1}{1+e^{-t\cdot c}}$$

$$P(-|t,c) = 1 - P(+|t,c)$$

$$= \frac{e^{-t\cdot c}}{1+e^{-t\cdot c}}$$

For all the context words:

Assume all context words are independent

$$P(+|t, c_{1:k}) = \prod_{i=1}^{\kappa} \frac{1}{1 + e^{-t \cdot c_i}}$$

$$\log P(+|t, c_{1:k}) = \sum_{i=1}^{k} \log \frac{1}{1 + e^{-t \cdot c_i}}$$

Objective Criteria

We want to maximize…

$$\sum_{(t,c)\in+} logP(+|t,c) + \sum_{(t,c)\in-} logP(-|t,c)$$

Maximize the + label for the pairs from the positive training data, and minimize the – label for the pairs sample from the negative data.

Loss Function: Binary Cross-Entropy / Log Loss

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

1. **When $y_i = 1$** (Positive Sample):
   - The second term $(1 - y_i) \log(1 - p(y_i))$ becomes 0 because $1 - y_i = 0$.
   - The equation simplifies to:
     $$-\log(p(y_i))$$
     - This term encourages the model to predict $p(y_i)$ as close to 1 as possible for positive samples. The closer $p(y_i)$ is to 1, the smaller (better) the loss becomes.
2. **When $y_i = 0$** (Negative Sample):
   - The first term $y_i \log(p(y_i))$ becomes 0 because $y_i = 0$.
   - The equation simplifies to:
     $$-\log(1 - p(y_i))$$
     - This term encourages the model to predict $p(y_i)$ as close to 0 as possible for negative samples. The closer $1 - p(y_i)$ is to 1, the smaller (better) the loss becomes.

Setup

Let's represent words as vectors of some length (say 300), randomly initialized.

So we start with 300 * V random parameters

Over the entire training set, we'd like to adjust those word vectors such that we

- Maximize the similarity of the target word, context word pairs (t,c) drawn from the positive data
- Minimize the similarity of the (t,c) pairs drawn from the negative data.

Learning the classifier

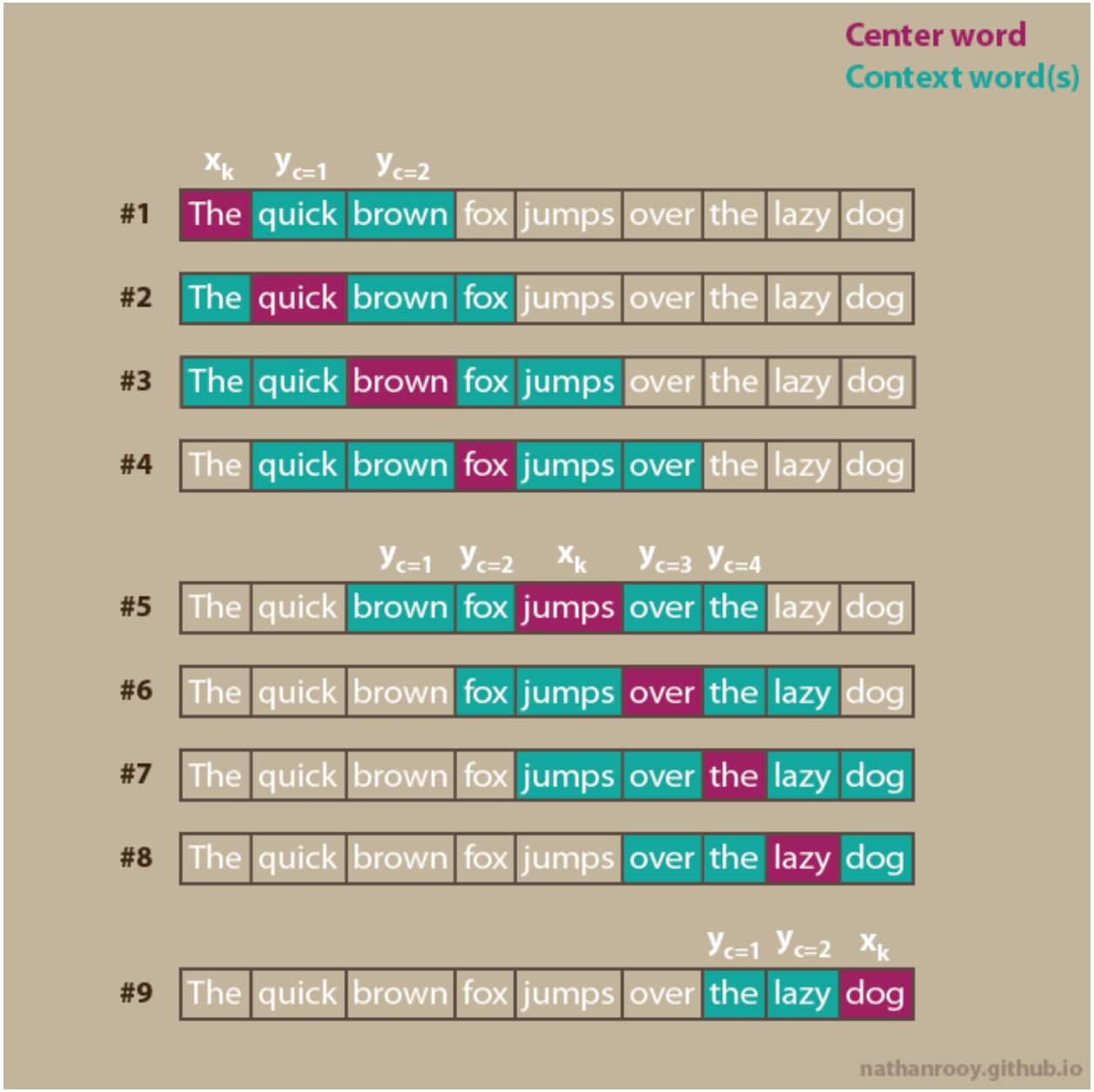
Iterative process.

We'll start with 0 or random weights

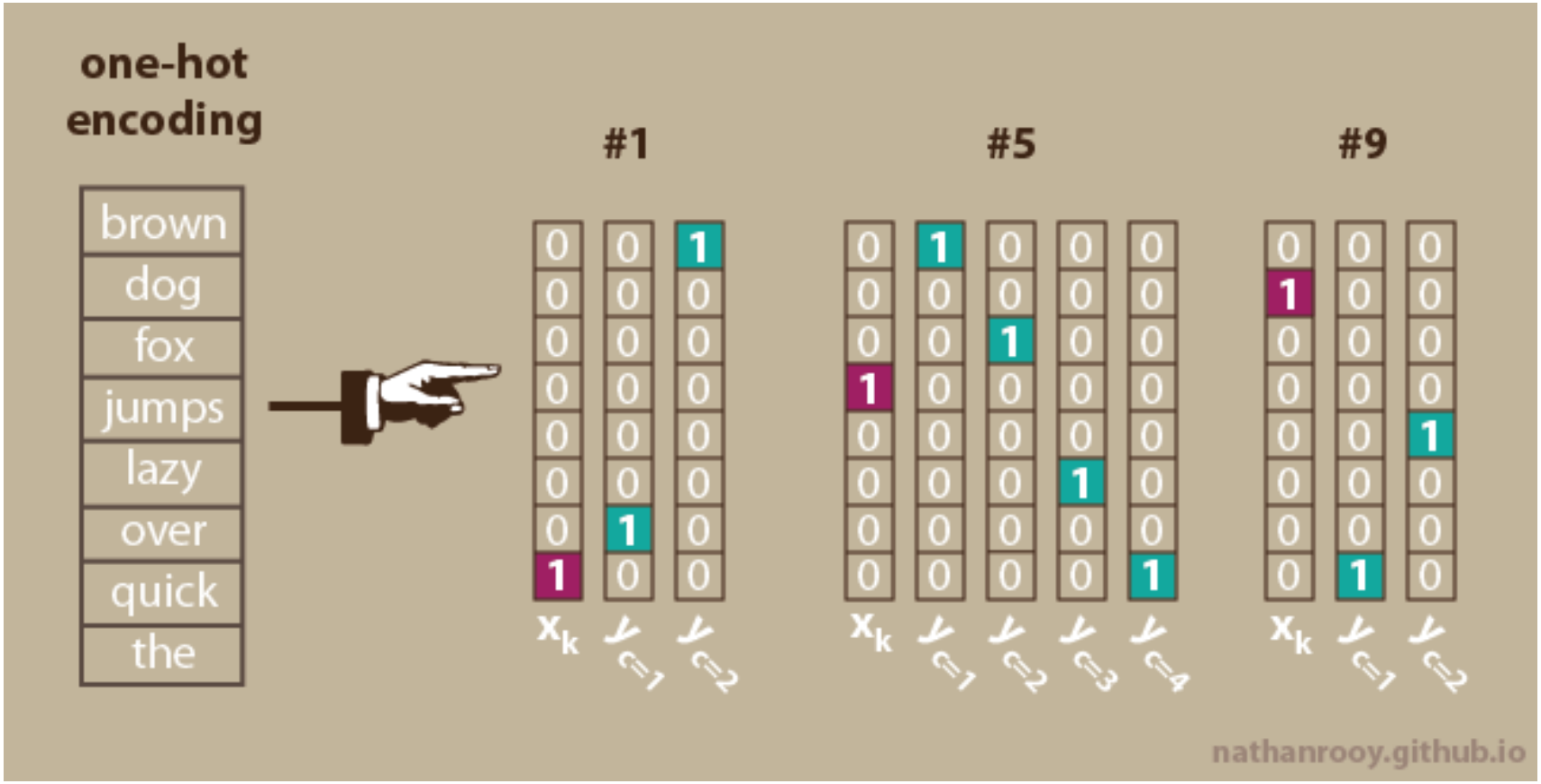
Then adjust the word weights to

- make the positive pairs more likely
- and the negative pairs less likely

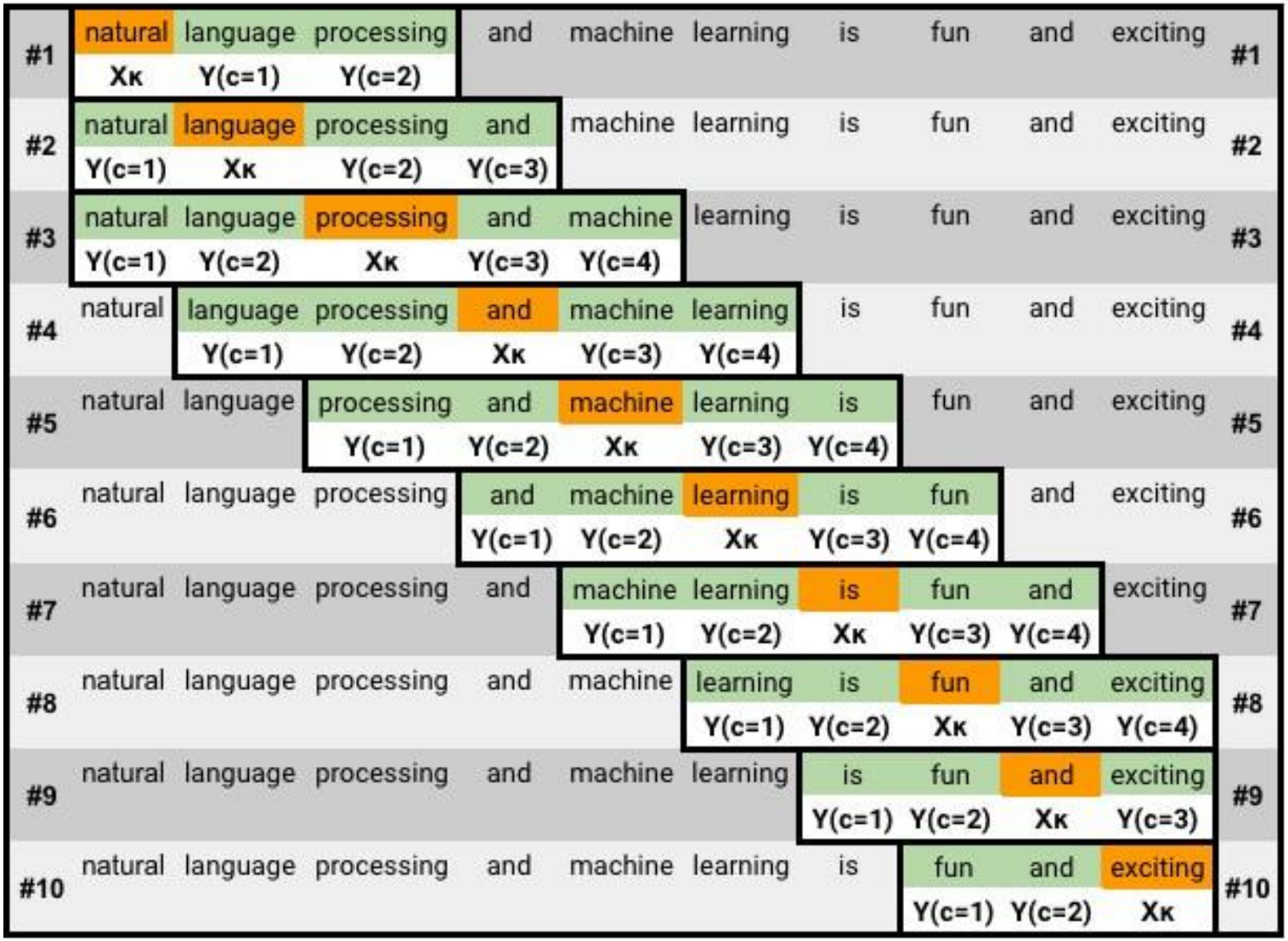
over the entire training set:



HTTPS://NATHANROOY.GITHUB.IO/POSTS/2018-03-22/WORD2VEC-FROM-SCRATCH-WITH-PYTHON-AND-NUMPY/?SOURCE=POST_PAGE-----13445EEBD281



HTTPS://NATHANROOY.GITHUB.IO/POSTS/2018-03-22/WORD2VEC-FROM-SCRATCH-WITH-PYTHON-AND-NUMPY/?SOURCE=POST_PAGE-----13445EEBD281



HTTPS://TOWARDSDATASCIENCE.COM/AN-IMPLEMENTATION-GUIDE-TO-WORD2VEC-USING-NUMPY-AND-GOOGLE-SHEETS-13445EEBD281

#	Token	#1			#2				#3					#4					#5				
0	natural	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	language	0	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
2	processing	0	0	1	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0
3	and	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0
4	machine	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0
5	learning	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
6	is	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
7	fun	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	exciting	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		Xk	Y(c=1)	Y(c=2)	Xk	Y(c=1)	Y(c=2)	Y(c=3)	Xk	Y(c=1)	Y(c=2)	Y(c=3)	Y(c=4)	Xk	Y(c=1)	Y(c=2)	Y(c=3)	Y(c=4)	Xk	Y(c=1)	Y(c=2)	Y(c=3)	Y(c=4)

#	Token	#6					#7					#8					#9				#10		
0	natural	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	language	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	processing	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	and	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1
4	machine	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	learning	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
6	is	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
7	fun	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	1	0
8	exciting	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	1	0	0
		Xk	Y(c=1)	Y(c=2)	(c=3	Y(c=4)	Xk	Y(c=1)	(c=2	Y(c=3)	Y(c=4)	Xk	Y(c=1)	(c=2	Y(c=3)	Y(c=4)	Xk	Y(c=1)	(c=2	Y(c=3)	Xk	Y(c=1)	Y(c=2)

derekchia.com

HTTPS://TOWARDSDATASCIENCE.COM/AN-IMPLEMENTATION-GUIDE-TO-WORD2VEC-USING-NUMPY-AND-GOOGLE-SHEETS-13445EEBD281

first and last element in the first training window

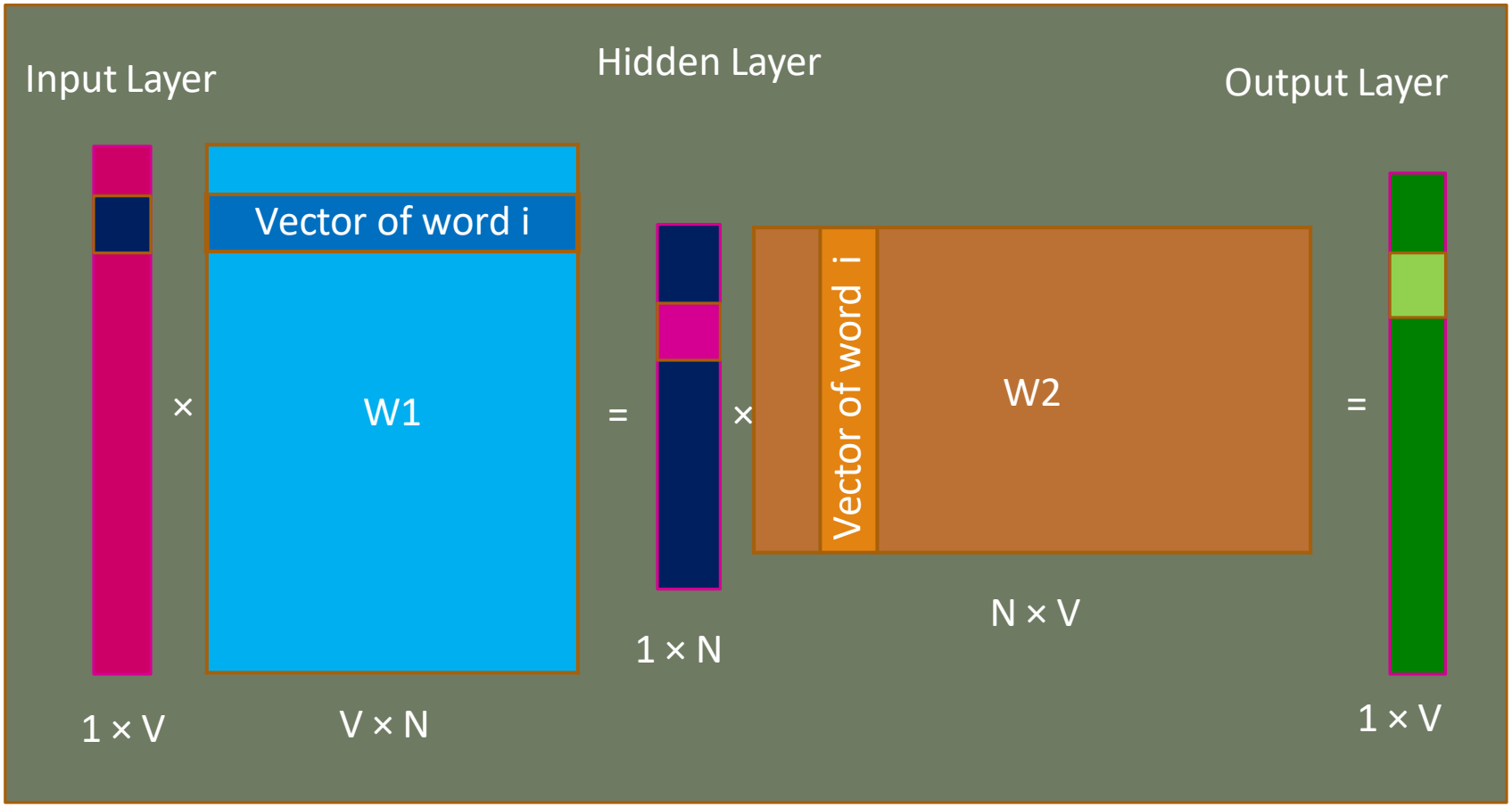
*# 1 [Target (**natural**)], [Context (*language, processing*)]*
*[list(**[1, 0, 0, 0, 0, 0, 0, 0, 0]**)*
list([[0, 1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0, 0]*])]*

first and last element in the last training window

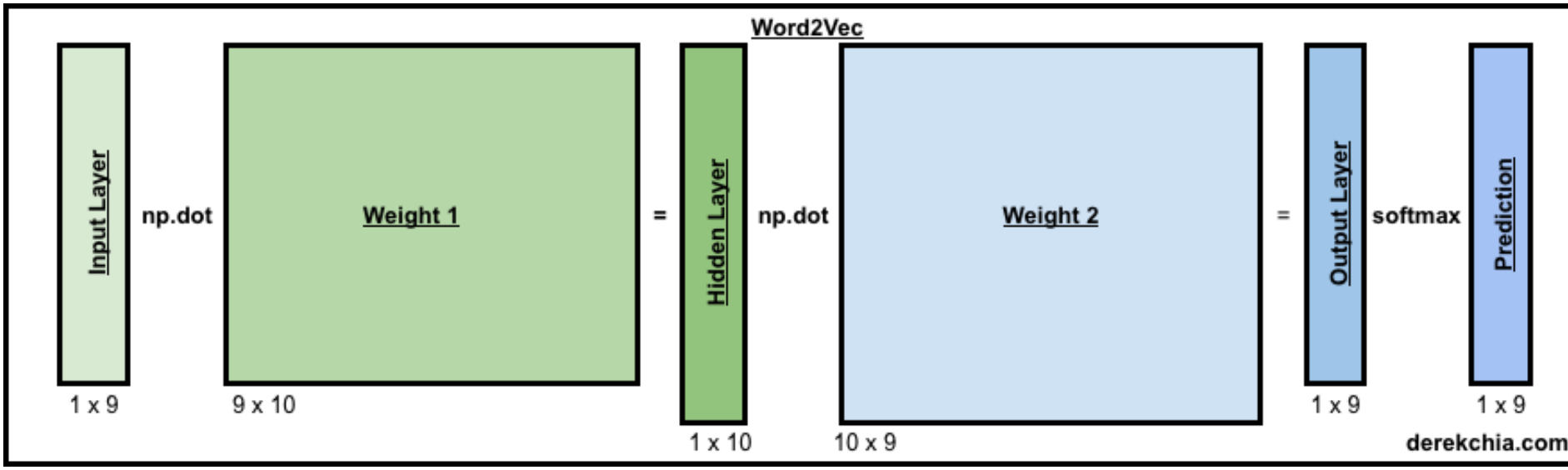
*#10 [Target (**exciting**)], [Context (*fun, and*)]*
*[list(**[0, 0, 0, 0, 0, 0, 0, 0, 1]**)*
list([[0, 0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 1, 0, 0, 0, 0, 0]*])]*

HTTPS://TOWARDSDATASCIENCE.COM/AN-IMPLEMENTATION-GUIDE-TO-WORD2VEC-USING-NUMPY-AND-GOOGLE-SHEETS-13445EEBD281

N = Dimensions, V = Vocabulary, unique words

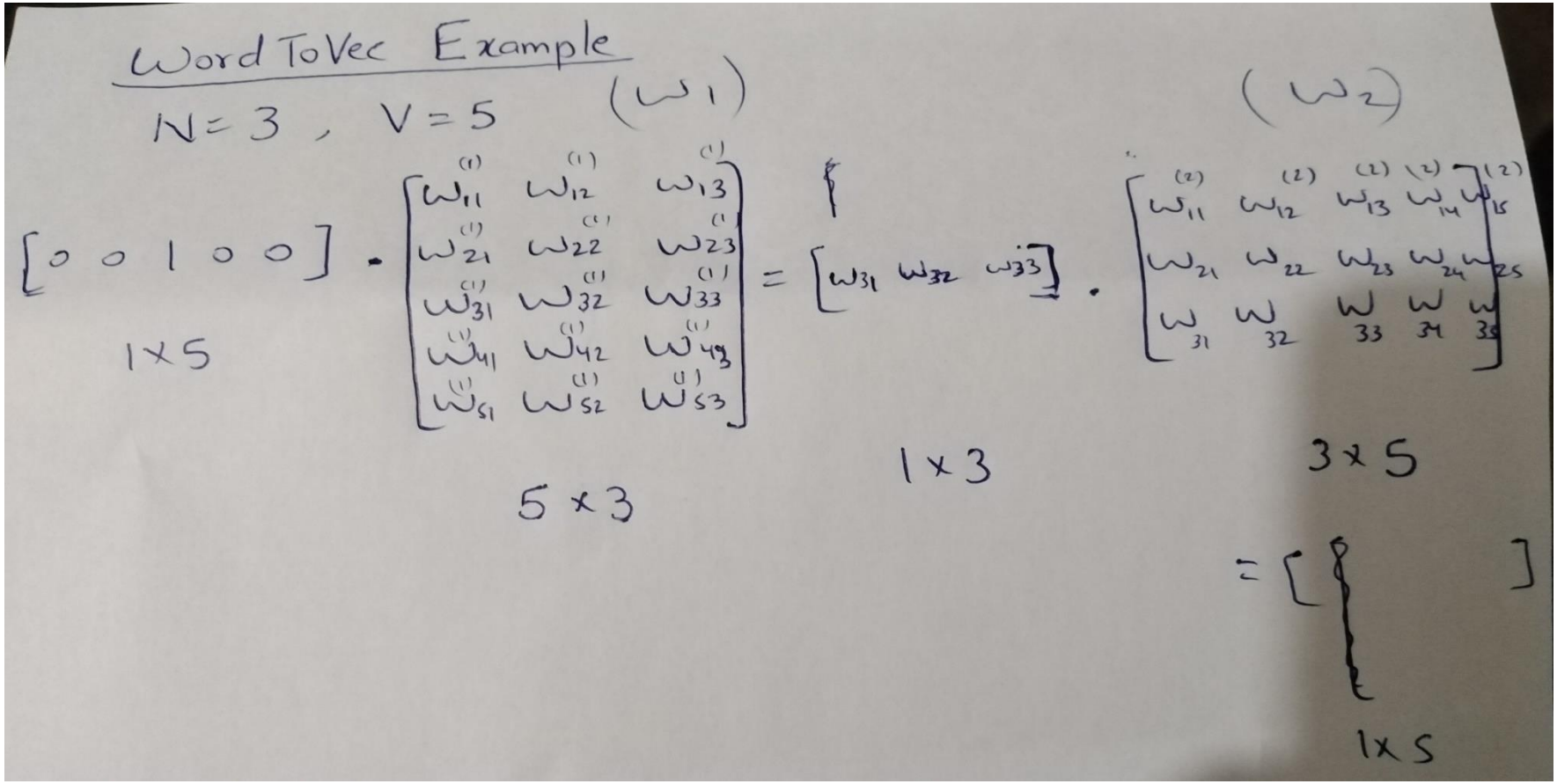


Word2Vec — skip-gram network architecture

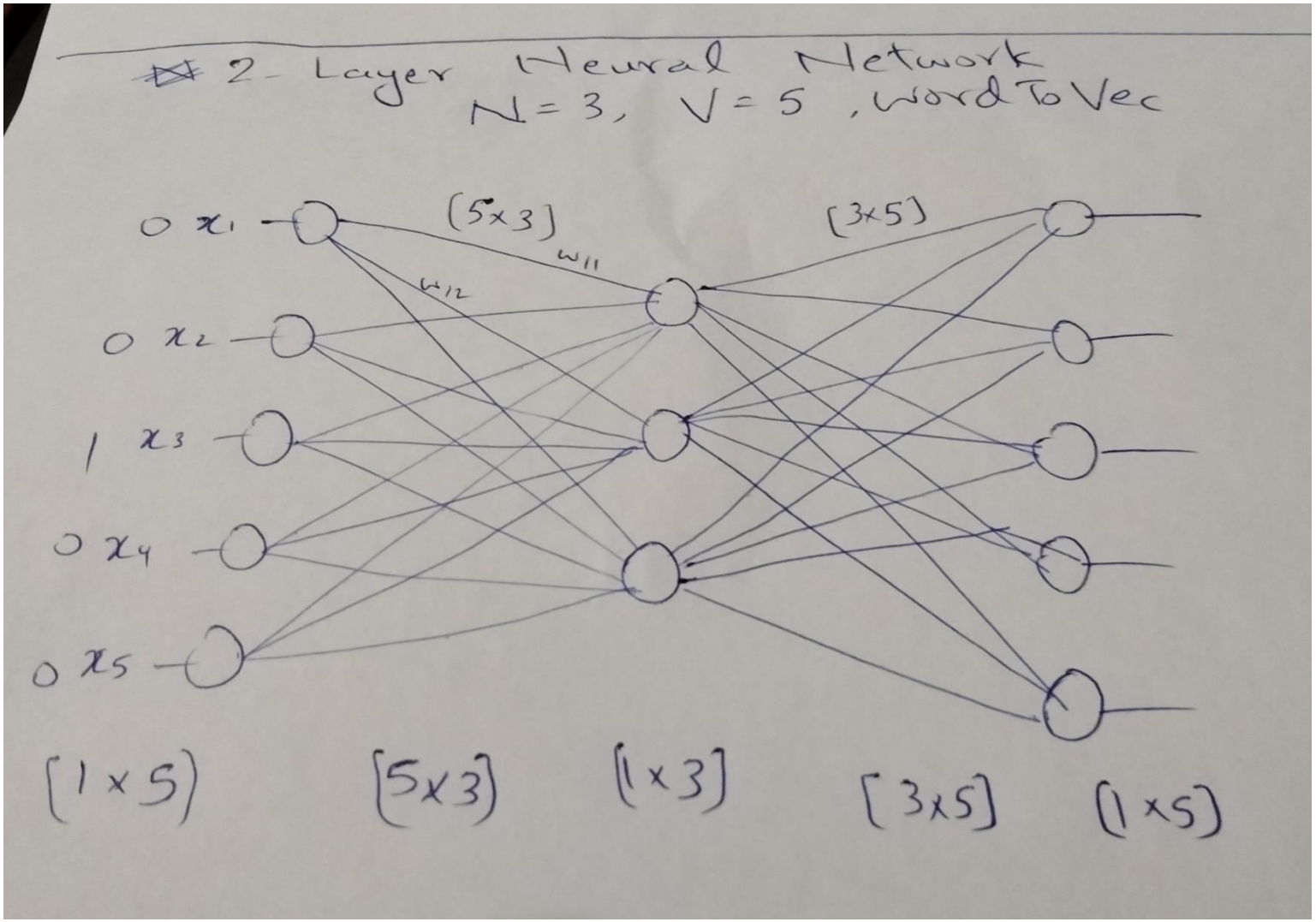


HTTPS://TOWARDSDATASCIENCE.COM/AN-IMPLEMENTATION-GUIDE-TO-WORD2VEC-USING-NUMPY-AND-GOOGLE-SHEETS-13445EEBD281

WordToVec Example



WordToVec Example 2 layer neural network



Word To Vec Example

$N = 3, \quad V = 5$

(W_1) (W_2)

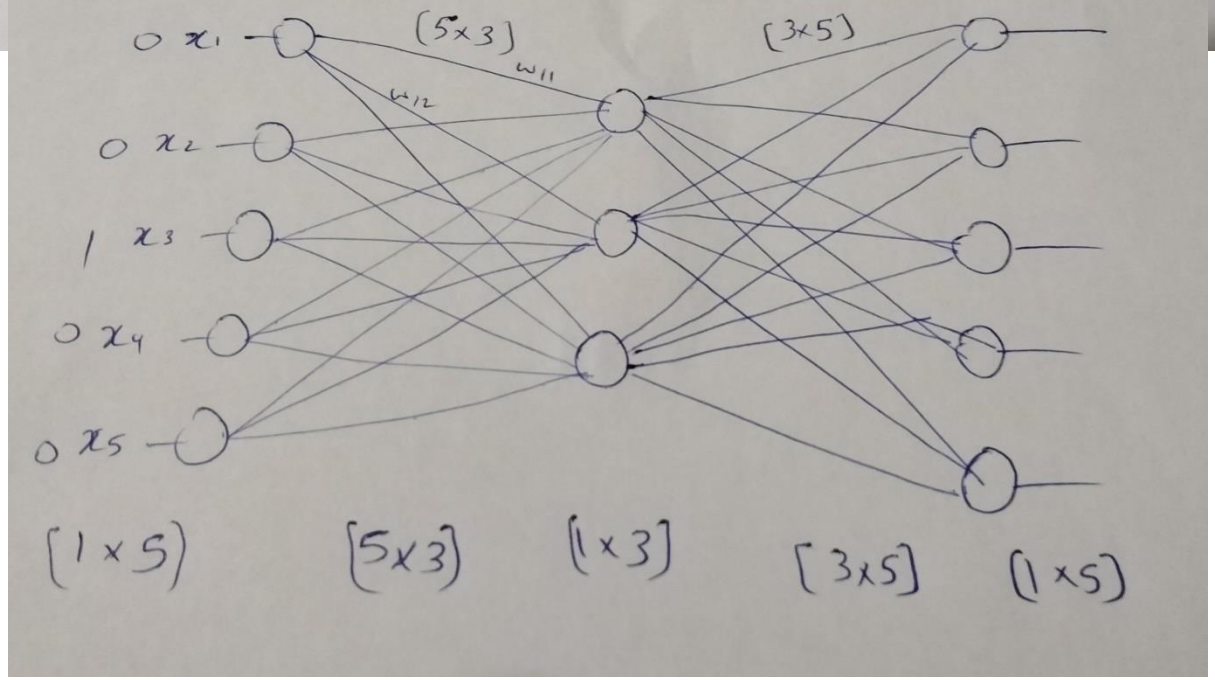
$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \\ w_{41}^{(1)} & w_{42}^{(1)} & w_{43}^{(1)} \\ w_{51}^{(1)} & w_{52}^{(1)} & w_{53}^{(1)} \end{bmatrix} = \begin{bmatrix} w_{31} & w_{32} & w_{33} \end{bmatrix} \cdot \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} & w_{14}^{(2)} & w_{15}^{(2)} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \end{bmatrix}$$

$1 \times 5 \qquad 5 \times 3 \qquad 1 \times 3 \qquad 3 \times 5$

$$= \begin{bmatrix} \{ & & & & \} \end{bmatrix}$$

1×5

2- Layer Neural Network
$N = 3, \quad V = 5$, Word To Vec



4. Skip-gram Model Architecture derekchia.com

Forward Pass for #1

Parameters - Embedding size 10

Random initialisation (Range) -1 to 1

Calculate Hidden Layer

#	Token	Input - w_t
0	natural	1
1	language	0
2	processing	0
3	and	0
4	machine	0
5	learning	0
6	is	0
7	fun	0
8	exciting	0

1 x 9

np.dot

Weight 1 - W1

0.236	**-0.962**	**0.686**	**0.785**	**-0.454**	**-0.833**	**-0.744**	**0.677**	**-0.427**	**-0.066**
-0.907	0.894	0.225	0.673	-0.579	-0.428	0.685	0.973	-0.070	-0.811
-0.576	0.658	-0.582	-0.112	0.662	0.051	-0.401	-0.921	-0.158	0.529
0.517	0.436	0.092	-0.835	-0.444	-0.905	0.879	0.303	0.332	-0.275
0.859	-0.890	0.651	0.185	-0.511	-0.456	0.377	-0.274	0.182	-0.237
0.368	-0.867	-0.301	-0.222	0.630	0.808	0.088	-0.902	-0.450	-0.408
0.728	0.277	0.439	0.138	-0.943	-0.409	0.687	-0.215	-0.807	0.612
0.593	-0.699	0.020	0.142	-0.638	-0.633	0.344	0.868	0.913	0.429
0.447	-0.810	-0.061	-0.495	0.794	-0.064	-0.817	-0.408	-0.286	0.149

9 x 10

=

Hidden Layer - h

0.236
-0.962
0.686
0.785
-0.454
-0.833
-0.744
0.677
-0.427
-0.066

1 x 10

Calculate y_pred

Hidden Layer - h

0.236
-0.962
0.686
0.785
-0.454
-0.833
-0.744
0.677
-0.427
-0.066

1 x 10

np.dot

Weight 2 - W2

-0.868	-0.406	-0.288	-0.016	-0.560	0.179	0.099	0.438	-0.551
-0.395	0.890	0.685	-0.329	0.218	-0.852	-0.919	0.665	0.968
-0.128	0.685	-0.828	0.709	-0.420	0.057	-0.212	0.728	-0.690
0.881	0.238	0.018	0.622	0.936	-0.442	0.936	0.586	-0.020
-0.478	0.240	0.820	-0.731	0.260	-0.989	-0.626	0.796	-0.599
0.679	0.721	-0.111	0.083	-0.738	0.227	0.560	0.929	0.017
-0.690	0.907	0.464	-0.022	-0.005	-0.004	-0.425	0.299	0.757
-0.054	0.397	-0.017	-0.563	-0.551	0.465	-0.596	-0.413	-0.395
-0.838	0.053	-0.160	-0.164	-0.671	0.140	-0.149	0.708	0.425
0.096	-0.995	-0.313	0.881	-0.402	-0.631	-0.660	0.184	0.487

10 x 9

=

Output Layer

1.258
-1.369
-1.828
1.196
0.545
1.113
1.333
-1.528
-2.335

1 x 9

Softmax - y_pred

0.218
0.016
0.010
0.205
0.107
0.189
0.235
0.013
0.006

1 x 9

HTTPS://TOWARDSDATASCIENCE.COM/AN-IMPLEMENTATION-GUIDE-TO-WORD2VEC-USING-NUMPY-AND-GOOGLE-SHEETS-13445EEBD281

Read from following links for WordToVec example

https://towardsdatascience.com/an-implementation-guide-to-word2vec-using-numpy-and-google-sheets-13445eebd281

https://nathanrooy.github.io/posts/2018-03-22/word2vec-from-scratch-with-python-and-numpy/?source=post_page-----13445eebd281

Word2vec is a family of algorithms

[Mikolov et al. 2013]

Predict between every word and its context words!

Two algorithms

1. **Skip-grams (SG)**

   Predict context words given target (position independent)

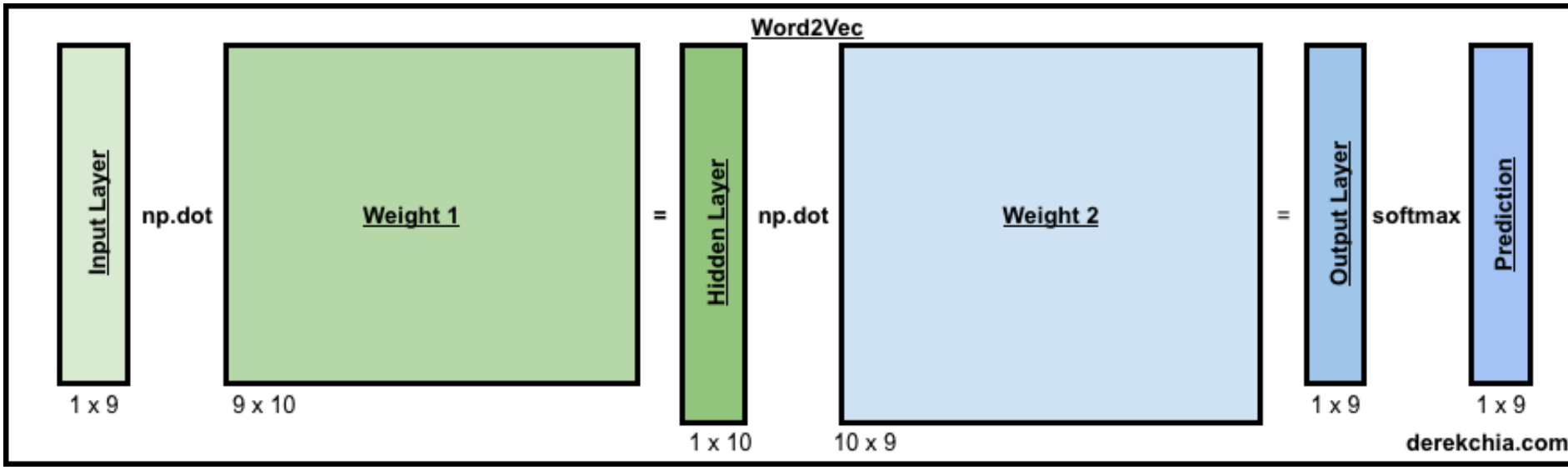
2. Continuous Bag of Words (CBOW)

   Predict target word from bag-of-words context

Two (moderately efficient) training methods

1. Hierarchical softmax
2. **Negative sampling**
3. Naïve softmax

Motivation for Negative Sampling (weight 2 matrix is updated for every example)



This means that instead of updating the entire matrix (which has a size of **hidden layer x vocabulary size**), the model only updates the weights associated with the context word and the few negative samples.

HTTPS://TOWARDSDATASCIENCE.COM/AN-IMPLEMENTATION-GUIDE-TO-WORD2VEC-USING-NUMPY-AND-GOOGLE-SHEETS-13445EEBD281

Skip-gram algorithm

1. Treat the target word and a neighboring context word as positive examples.
2. Randomly sample other words in the lexicon to get negative samples
3. Use a model to train a classifier to distinguish those two cases
4. Use the weights as the embeddings

Skip-Gram Training Data with Negative Sampling

Training sentence:

... lemon, a tablespoon of **apricot** jam a pinch ...

c1 c2 t c3 c4

Training data: input/output pairs centering on *apricot*

Asssume a +/- 2 word window

Skip-Gram Training with Negative Sampling

Training sentence:

... lemon, a tablespoon of **apricot** jam a pinch ...

c1 c2 t c3 c4

positive examples +	
t	c
apricot	tablespoon
apricot	of
apricot	preserves
apricot	or

- For each positive example, we'll create *k* negative examples.
- Any random word that isn't *t*

Skip-Gram Training with Negative Sampling

Training sentence:

... lemon, a tablespoon of **apricot** jam a pinch ...

c1 c2 t c3 c4

positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	preserves
apricot	or

negative examples - k=2

t	c	t	c
apricot	aardvark	apricot	twelve
apricot	puddle	apricot	hello
apricot	where	apricot	dear
apricot	coaxial	apricot	forever

Total words = dog, cat, lion, table, chair, dog , dog, cat , cat , cat

Vocabulary = dog , cat , lion, table, chair

Cat = 4/10 = 0.4

Dog = 0.3

Table = 0.1

Chair = 0.1

Lion =0.1

Choosing noise words (negative samples)

Could pick w according to their unigram frequency P(w)

More common to use $p_\alpha(w)$

$$P_\alpha(w) = \frac{count(w)^\alpha}{\sum_w count(w)^\alpha}$$

α= ¾ works well because it gives rare noise words slightly higher probability

To show this, imagine two events p(a)=.99 and p(b) = .01:

$$P_\alpha(a) = \frac{.99^{.75}}{.99^{.75} + .01^{.75}} = .97$$

$$P_\alpha(b) = \frac{.01^{.75}}{.99^{.75} + .01^{.75}} = .03$$

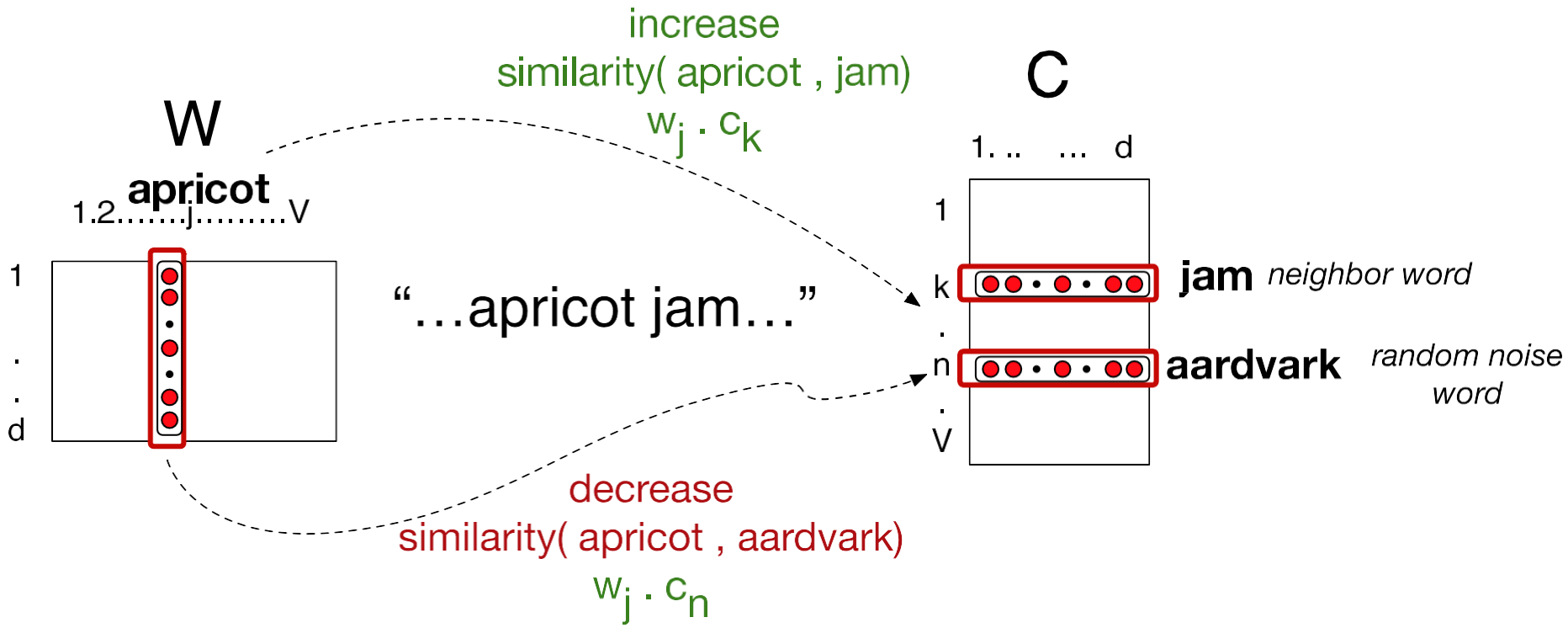
Loss Function

We want to maximize…

$$\sum_{(t,c)\in+} logP(+|t,c) + \sum_{(t,c)\in-} logP(-|t,c)$$

Maximize the + label for the pairs from the positive training data, and the – label for the pairs sample from the negative data.

The number of negative examples is much smaller using negative sampling


increase
similarity(apricot , jam)
$w_j \cdot c_k$
W
apricot
1.2.....j.....V
1
.
.
d
"…apricot jam…"
C
1. .. ... d
1
k
.
n
.
V
jam
neighbor word
aardvark
random noise word
decrease
similarity(apricot , aardvark)
$w_j \cdot c_n$

Two sets of embeddings

SGNS learns two sets of embeddings

Target embeddings matrix W (weight 1)

Context embedding matrix C (weight 2)

It's common to just add them together, representing word *i* as the vector $w_i + c_i$

Summary: How to learn word2vec (skip-gram) embeddings

Start with V random 300-dimensional vectors as initial embeddings

Use logistic regression, the second most basic classifier used in machine learning after naïve bayes

- Take a corpus and take pairs of words that co-occur as positive examples
- Take pairs of words that don't co-occur, as negative examples
- Train the classifier to distinguish these by slowly adjusting all the embeddings to improve the classifier performance
- Throw away the classifier code and keep the embeddings.

Properties of embeddings

Similarity depends on window size C

(**small window size:** words that are syntactically similar and have same part of speech**,**

Large window size: words related but do not have same Part of speech (semantics)

1. **Syntactic** similarity: Words with similar grammatical structures, word order, or linguistic patterns.
2. **Semantic** similarity: Words with similar meanings, concepts, or contexts.

C = ±2 The nearest words to *Hogwarts (name of school)* were names of other fictional schools*:*

- *Sunnydale (name of fictional school)*
- *Evernight (name of fictional school)*

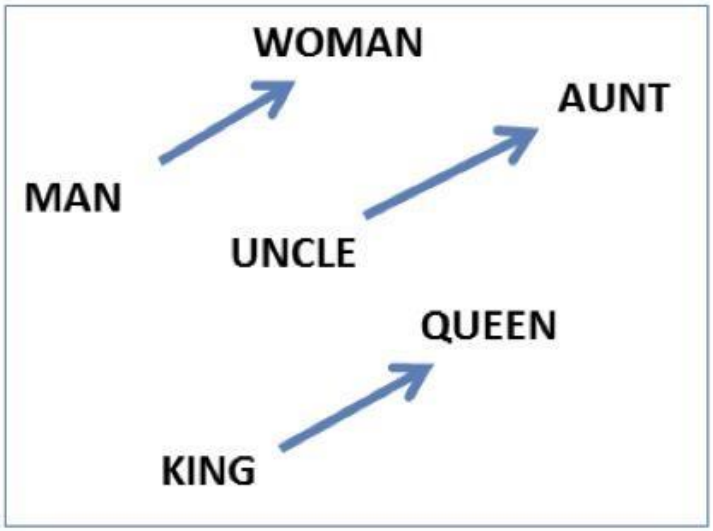
C = ±5 The nearest words to *Hogwarts* were other words topically related to the *Harry Potter* series *:*

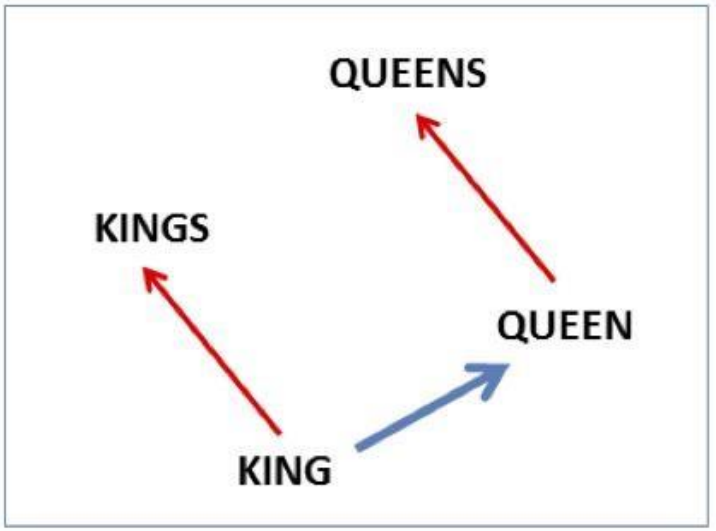
- *Dumbledore*
- *Malfoy*
- *halfblood*

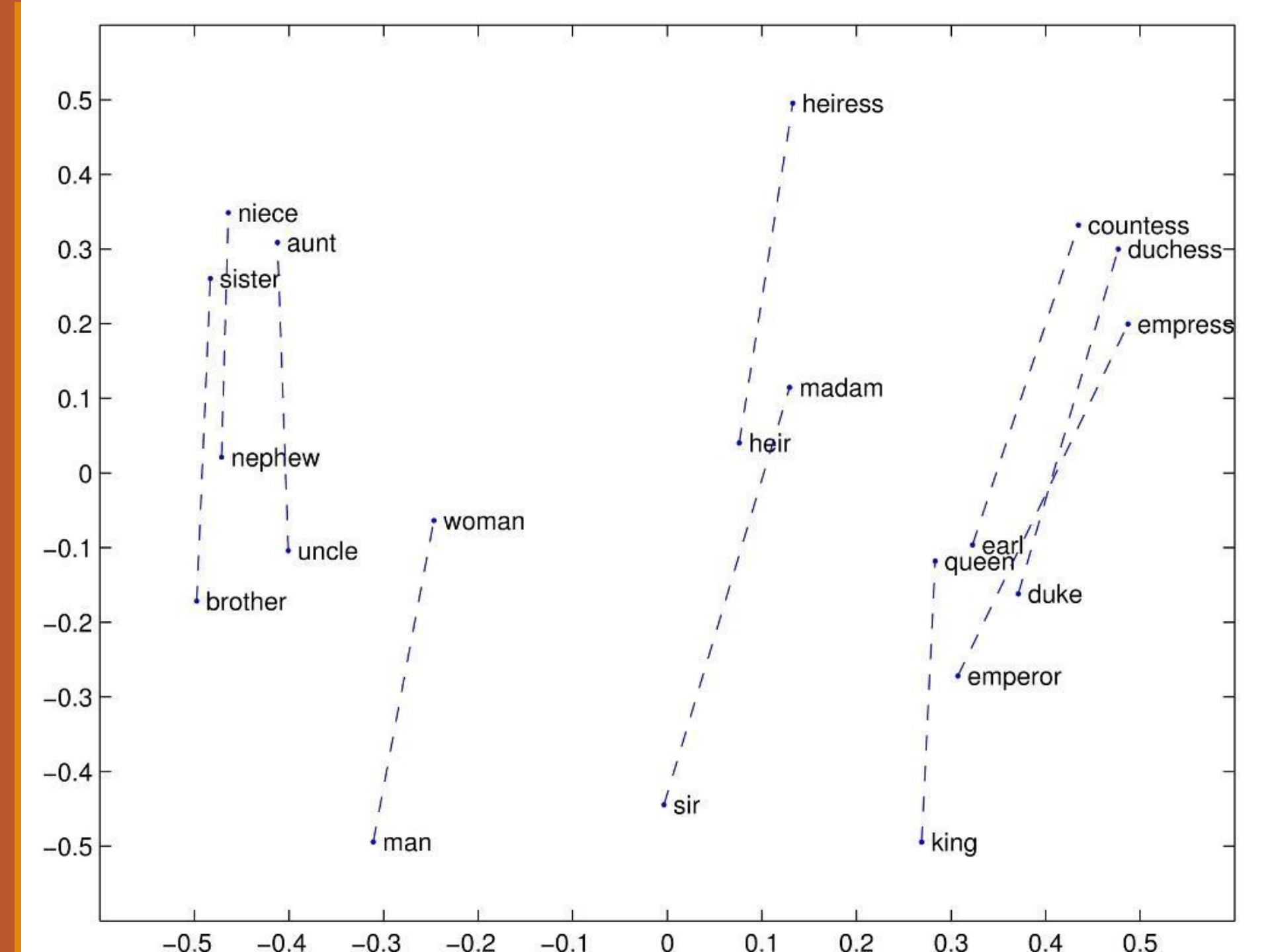
Analogy: Embeddings capture relational meaning!

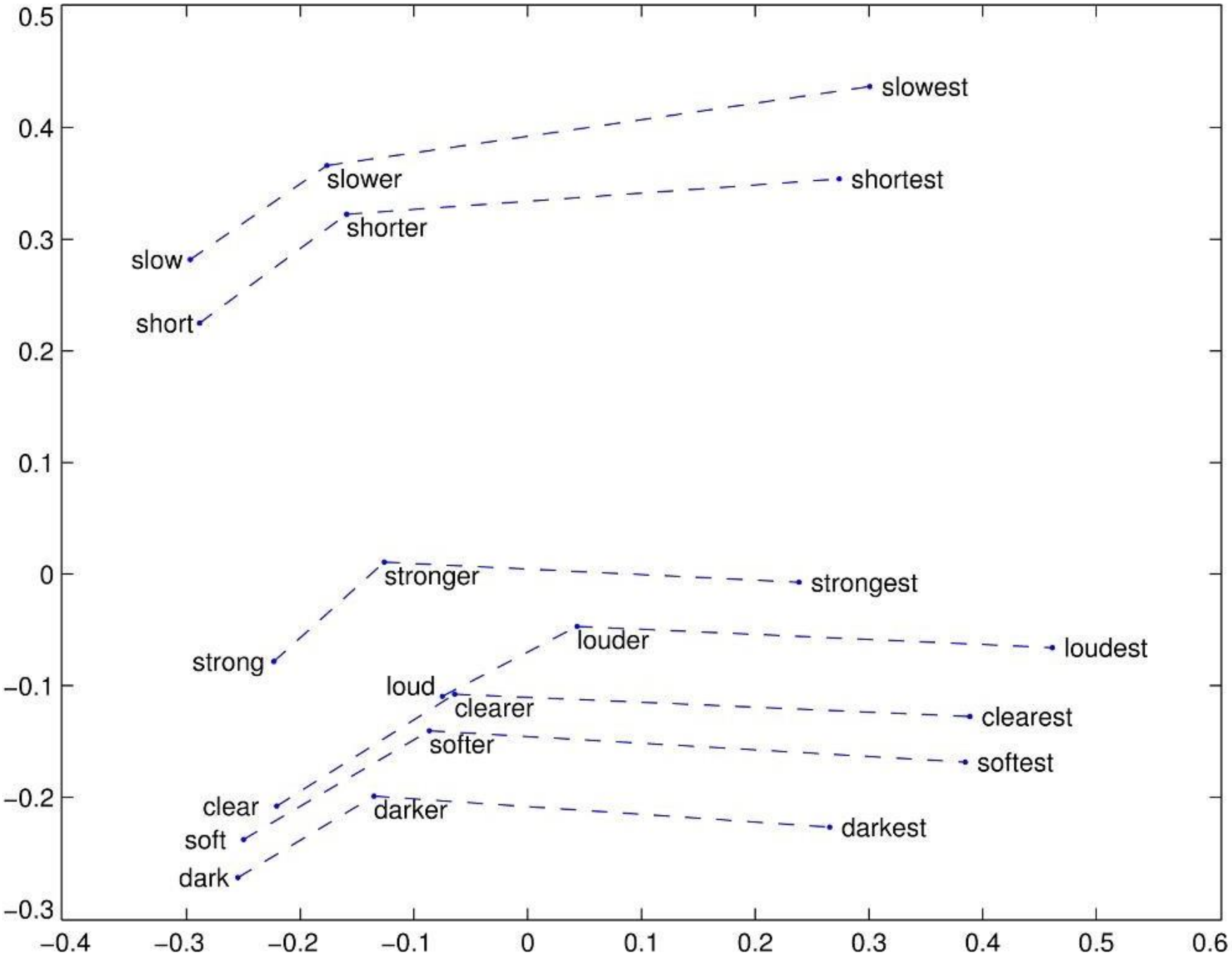
vector(*'king'*) - vector(*'man'*) + vector(*'woman'*) ≈ vector('queen')

vector(*'Paris'*) - vector(*'France'*) + vector(*'Italy'*) ≈ vector('Rome')



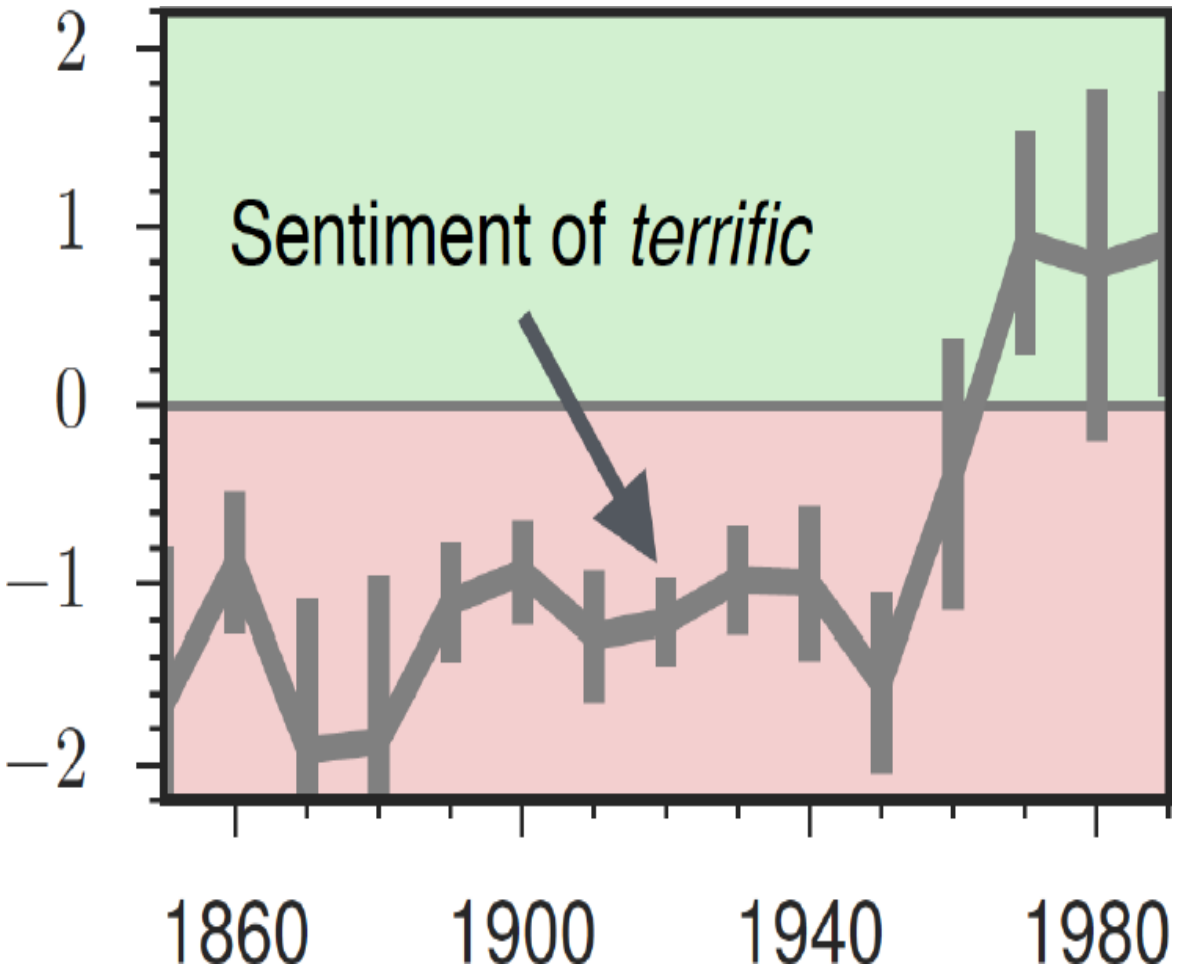



heiress
niece
aunt
sister
countess
duchess
empress
madam
heir
nephew
woman
uncle
earl
queen
duke
brother
emperor
sir
man
king
0.5
0.4
0.3
0.2
0.1
0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.5
−0.4
−0.3
−0.2
−0.1
0
0.1
0.2
0.3
0.4
0.5


slowest
slower
shortest
shorter
slow
short
stronger
strongest
louder
loudest
strong
loud
clearer
clearest
softer
softest
clear
darker
darkest
soft
dark
0.5
0.4
0.3
0.2
0.1
0
−0.1
−0.2
−0.3
−0.4
−0.3
−0.2
−0.1
0
0.1
0.2
0.3
0.4
0.5
0.6

The evolution of sentiment words

Negative words change faster than positive words



Embeddings reflect cultural bias

Bolukbasi, Tolga, Kai-Wei Chang, James Y. Zou, Venkatesh Saligrama, and Adam T. Kalai. "Man is to computer programmer as woman is to homemaker? debiasing word embeddings." In *Advances in Neural Information Processing Systems*, pp. 4349-4357. 2016.

Ask “Paris : France :: Tokyo : x”

- x = Japan

Ask “father : doctor :: mother : x”

- x = nurse

Ask “man : computer programmer :: woman : x”

- x = homemaker

Embeddings as a window onto history

Garg, Nikhil, Schiebinger, Londa, Jurafsky, Dan, and Zou, James (2018). Word embeddings quantify 100 years of gender and ethnic stereotypes. *Proceedings of the National Academy of Sciences*, *115*(16), E3635–E3644

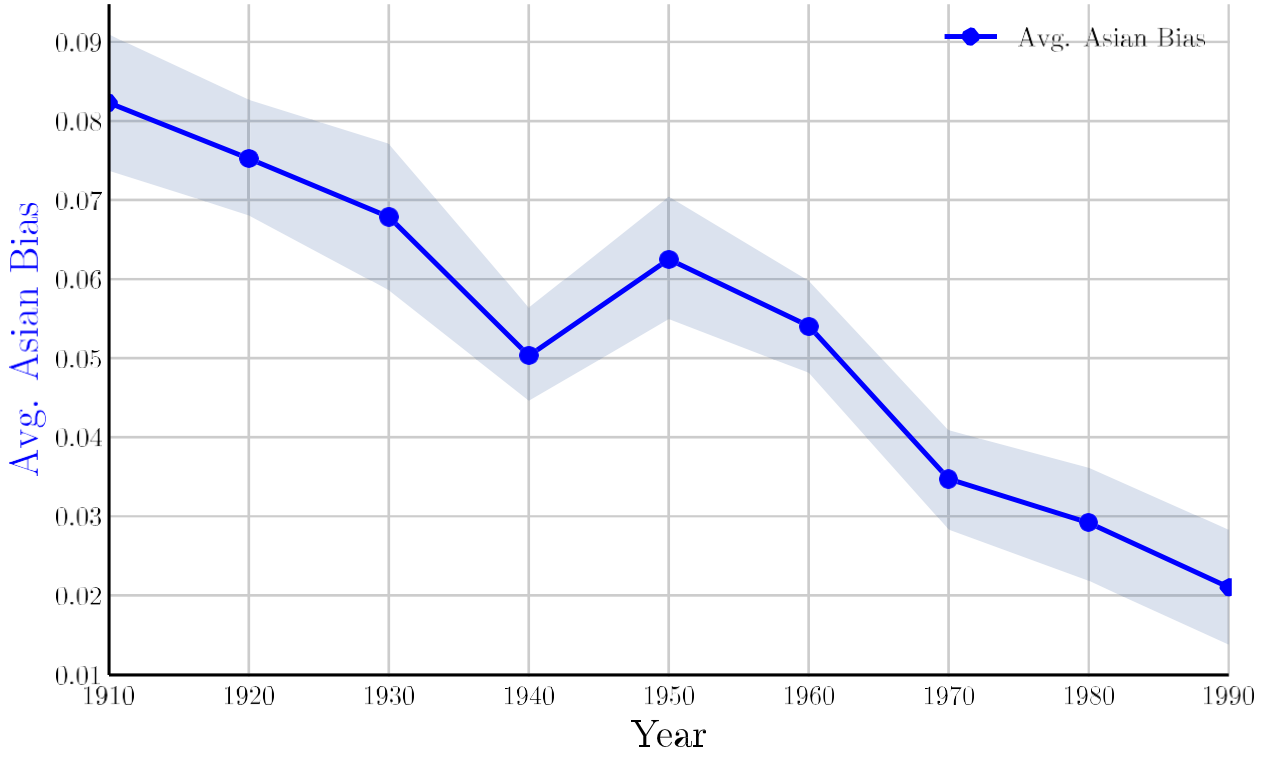
The cosine similarity of embeddings for decade X for occupations (like teacher) to male vs female names

- Is correlated with the actual percentage of women teachers in decade X

Change in linguistic framing 1910-1990

Garg, Nikhil, Schiebinger, Londa, Jurafsky, Dan, and Zou, James (2018). Word embeddings quantify 100 years of gender and ethnic stereotypes. *Proceedings of the National Academy of Sciences*, *115*(16), E3635–E3644

Change in association of Chinese names with adjectives framed as "othering" (*barbaric*, *monstrous*, *bizarre*)



Changes in framing: adjectives associated with Chinese

Garg, Nikhil, Schiebinger, Londa, Jurafsky, Dan, and Zou, James (2018). Word embeddings quantify 100 years of gender and ethnic stereotypes. *Proceedings of the National Academy of Sciences*, *115*(16), E3635–E3644

1910	1950	1990
Irresponsible	Disorganized	Inhibited
Envious	Outrageous	Passive
Barbaric	Pompous	Dissolute
Aggressive	Unstable	Haughty
Transparent	Effeminate	Complacent
Monstrous	Unprincipled	Forceful
Hateful	Venomous	Fixed
Cruel	Disobedient	Active
Greedy	Predatory	Sensitive
Bizarre	Boisterous	Hearty

Conclusion

Embeddings = vector models of meaning

- More fine-grained than just a string or index
- Especially good at modeling similarity/analogy
  - Just download them and use cosines!!
- Can use sparse models (tf-idf) or dense models (word2vec, GLoVE)
- Useful in practice but know they encode cultural stereotypes
- **W2v gives static embeddings**

- Which one to chose?
  - Skipgram: for large corpus, higher dimensions though performs slower, good for infrequent words
  - CBOW: faster for small corpus, overfitting problem
- Increasing training dataset
- Increasing vector dimensions
  - The size of the vector is typically set to be between 50 and 1000 dimensions, with 300 being a common default choice.
- Increasing window size

Limitations of w2v

1. Word2Vec relies on local information about words, i.e. the context of a word relies only on its neighbors.
2. The obtained word embedding is a byproduct of training a neural network, hence the linear relationships between feature vectors are a black box (kind of).
3. Word2Vec cannot understand out-of-vocabulary (OOV) words, i.e. words not present in training data. You could assign a UNK token which is used for all OOV words or you could use other models that are robust to OOV words.
4. By assigning a distinct vector to each word, Word2Vec ignores the **morphology** of words. For example, *eat*, *eats*, and *eaten* are considered independently different words by Word2Vec, but they come from the same root: *eat*, which might contain useful information.

https://levelup.gitconnected.com/glove-and-fasttext-clearly-explained-extracting-features-from-text-data-1d227ab017b2

Readings

1. Dan Jurafsky and James Martin, Speech and Language Processing
   Chapter 6: Vector Semantics
   chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://web.stanford.edu/~jurafsky/slp3/6.pdf

2. https://towardsdatascience.com/an-implementation-guide-to-word2vec-using-numpy-and-google-sheets-13445eebd281