Unsupervised Word Embeddings

Slides are adopted from Stanford: NLP with DL by C. Manning

1. How do we represent the meaning of a word?

Definition: **meaning** (Webster dictionary)

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.
- the idea that is expressed in a work of writing, art, etc.

Commonest linguistic way of thinking of meaning:

$\boxed{\text{signifier (symbol)} \Leftrightarrow \text{signified (idea or thing)}}$

= denotational semantics

How do we have usable meaning in a computer?

Common solution: Use e.g. WordNet, a thesaurus containing lists of **synonym sets** and **hypernyms** (“is a” relationships).

e.g. synonym sets containing “good”:

```
from nltk.corpus import wordnet as wn
poses = { 'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}
for synset in wn.synsets("good"):
    print("{}: {}".format(poses[synset.pos()],
          ", ".join([l.name() for l in synset.lemmas()])))
```

```
noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good
adj: good
adj (sat): estimable, good, honorable, respectable
adj (sat): beneficial, good
adj (sat): good
adj (sat): good, just, upright
…
adverb: well, good
adverb: thoroughly, soundly, good
```

e.g. hypernyms of “panda”:

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(panda.closure(hyper))
```

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

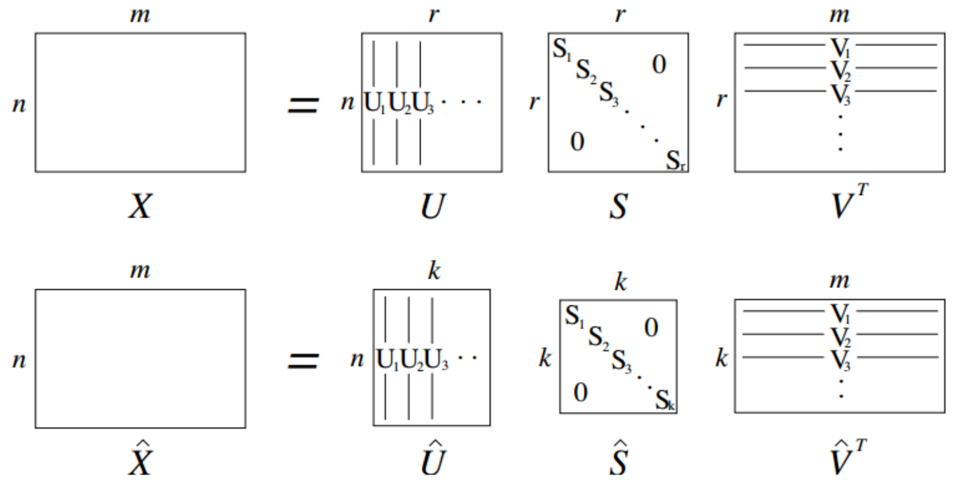
To compare pieces of text

- We need effective representation of:
    - Words
    - Sentences
    - Text
- **Approach 1:** Use existing thesauri or ontologies like *WordNet*
    - Drawbacks:
        - Manual
        - Not context specific
- **Approach 2:** Use co-occurrences for word similarity
    - Drawbacks:
        - Quadratic space needed
        - Relative position and order of words not considered

Approach 3: low dimensional vectors

- Store only “important” information in fixed, low dimensional vector.

- Singular Value Decomposition (SVD) on co-occurrence matrix
  - $\hat{X}$ is the best rank k approximation to X , in terms of least squares
  - Motel = [0.286, 0.792, -0.177, -0.107, 0.109, -0.542, 0.349, 0.271]



Problems with SVD

- Computational cost scales quadratically for n x m matrix: O(mn^2) flops (when n<m)

- Hard to incorporate new words or documents

- Does not consider order of words

Problems with resources like WordNet

- Great as a resource but missing nuance
  - e.g. "proficient" is listed as a synonym for "good". This is only correct in some contexts.
- Missing new meanings of words
  - e.g., wicked, badass, nifty, wizard, genius, ninja, bombest
  - Impossible to keep up-to-date!
- Subjective
- Requires human labor to create and adapt
- Can't compute accurate word similarity →

Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols:
hotel, conference, motel – a localist representation

Means one 1, the rest 0s

Words can be represented by one-hot vectors:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

Vector dimension = number of words in vocabulary (e.g., 500,000)

Problem with words as discrete symbols

Example: in web search, if user searches for "Seattle motel", we would like to match documents containing "Seattle hotel".

But:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

These two vectors are orthogonal.

There is no natural notion of **similarity** for one-hot vectors!

Solution:

- Could try to rely on WordNet's list of synonyms to get similarity?
  - But it is well-known to fail badly: incompleteness, etc.
- **Instead: learn to encode similarity in the vectors themselves**

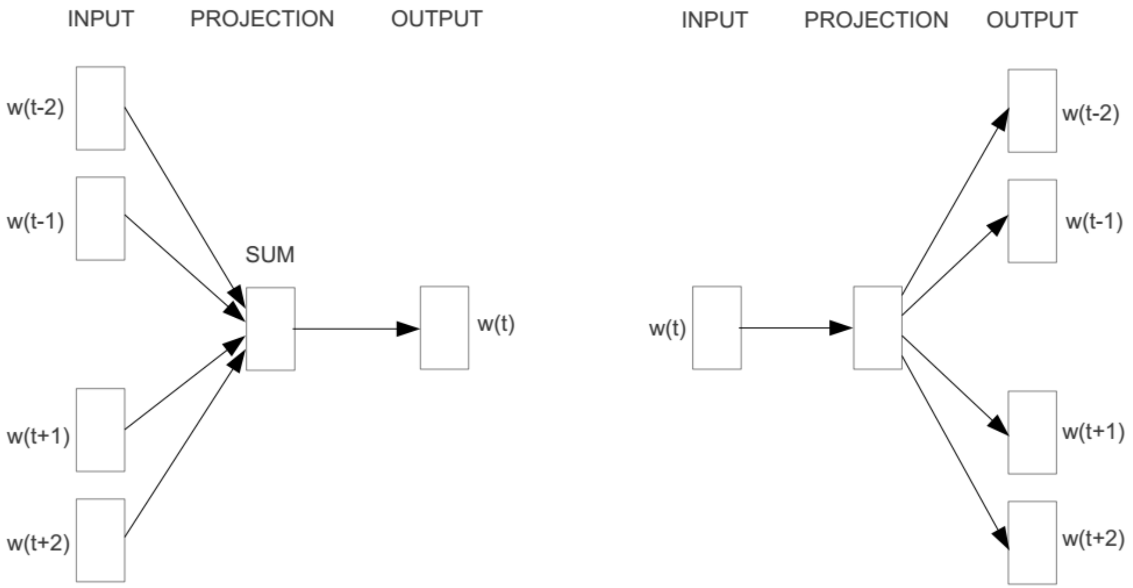
Representing words by their context

- Distributional semantics: **A word's meaning is given by the words that frequently appear close-by**
  - *"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!
- When a word *w* appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).
- Use the many contexts of *w* to build up a representation of *w*

…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…*

Represent the meaning of **word** – word2vec

- 2 basic neural network models:
  - **Continuous Bag of Word** (CBOW): use a window of word to predict the middle word
  - **Skip-gram** (SG): use a word to predict the surrounding ones in window.



CBOW

Skip-gram

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
- Big idea: **self-supervision**:
    - A word c that occurs near apricot in the corpus cats as the gold "correct answer" for supervised learning
    - No need for human labels
    - Bengio et al. (2003); Collobert et al. (2011)

Approach: predict if candidate word *c* is a "neighbor"

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

Skip-Gram Training Data

Assume a +/- 2 word window, given training sentence:

…lemon, a [tablespoon of apricot jam, a] pinch…

c1 c2 [target] c3 c4

Skip-Gram Classifier

(assuming a +/- 2 word window)

…lemon, a [tablespoon of apricot jam, a] pinch…

c1 c2 [target] c3 c4

- Goal: train a classifier that is given a candidate (**w**ord, **c**ontext) pair

  (apricot, jam)

  (apricot, aardvark)

  …
- And assigns each pair a probability:

  $P(+|w, c)$

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

Similarity is computed using dot product

- Remember: two vectors are similar if they have a high dot product
  - Cosine is just a normalized dot product
- Similarity(w,c) $\propto w \cdot c$
- We'll need to normalize to get a probability

  (cosine isn't a probability either)

Turning dot products into probabilities

- Sim(w,c) ≈ w $\cdot$ c
- To turn this into a probability
- We'll use the sigmoid from logistic regression:

$$P(+|w,c) = \sigma(c\cdot w) = \frac{1}{1+\exp(-c\cdot w)}$$

$$\begin{aligned}P(-|w,c) &= 1-P(+|w,c)\\ &= \sigma(-c\cdot w) = \frac{1}{1+\exp(c\cdot w)}\end{aligned}$$

How Skip-Gram Classifier computes $P(+|w, c)$

$$P(+|w,c) \;=\; \sigma(c\cdot w) = \frac{1}{1+\exp(-c\cdot w)}$$

- This is for one context word, but we have lots of context words.
- We'll assume independence and just multiply them:

$$P(+|w,c_{1:L}) \;=\; \prod_{i=1}^{L}\sigma(c_i\cdot w)$$

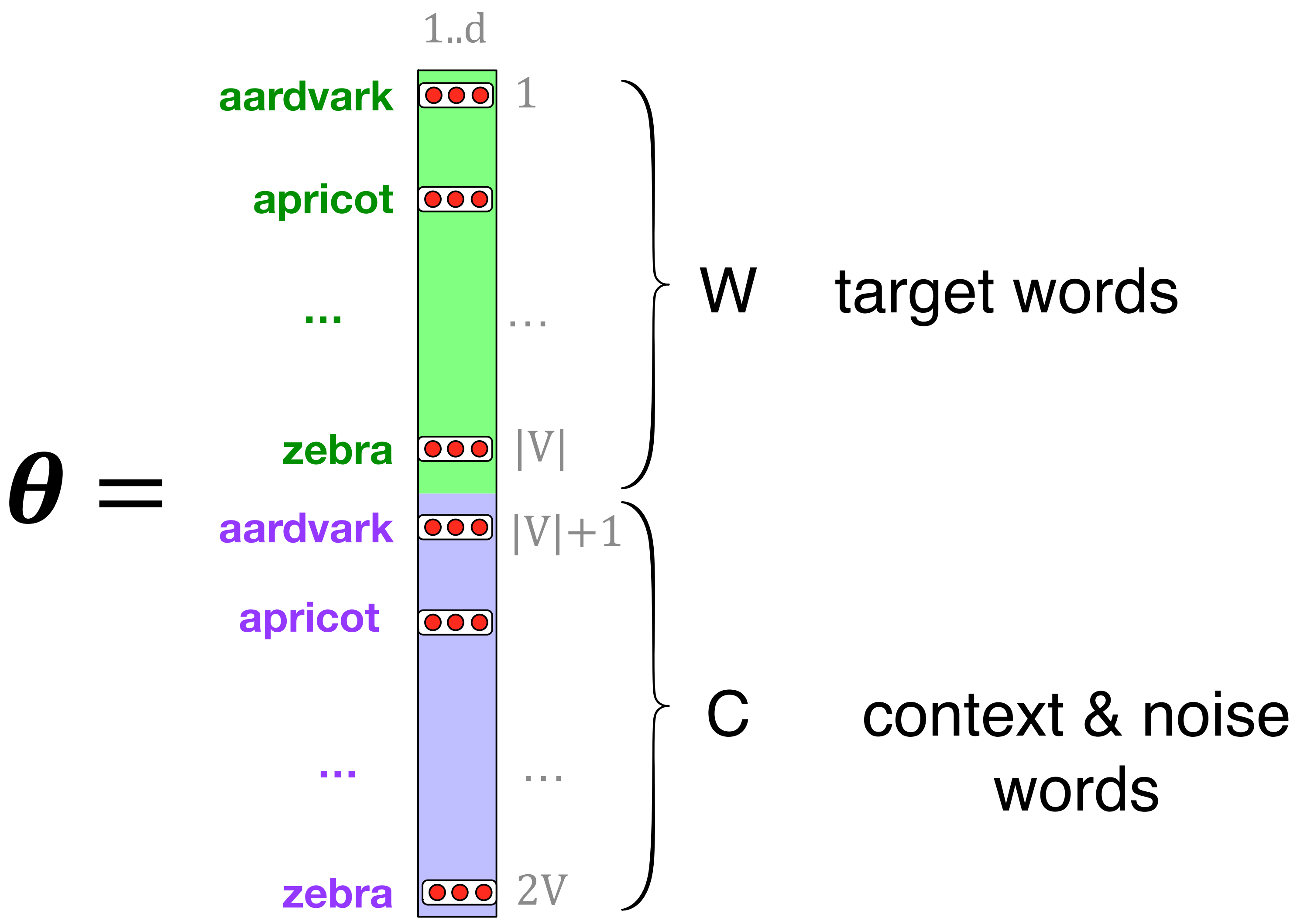
$$\log P(+|w,c_{1:L}) \;=\; \sum_{i=1}^{L}\log\sigma(c_i\cdot w)$$

Skip-gram classifier: summary

- A probabilistic classifier, given
  - a test target word *w*
  - its context window of *L* words $c_{1:L}$
- Estimates probability that w occurs in this window based on similarity of w (embeddings) to $\boldsymbol{c}_{1:L}$ (embeddings).

- To compute this, we just need embeddings for all the words.

These embeddings we'll need: a set for w, a set for c



Word2vec: Learning the embeddings

Skip-Gram Training data

…lemon, a [tablespoon of apricot jam, a] pinch…
c1 c2 [target] c3 c4

positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

Skip-Gram Training data

...lemon, a [tablespoon of apricot jam, a] pinch...

c1 c2 [target] c3 c4

positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

For each positive example we'll grab k negative examples, sampling by frequency

Skip-Gram Training data

…lemon, a [tablespoon of apricot jam, a] pinch…

c1 c2 [target] c3 c4

positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

t	c	t	c
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

Choosing negative examples

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

Setting $\alpha = .75$ gives better performance because it gives rare noise words slightly higher probability: for rare words, $P_\alpha(w) > P(w)$.

$P(a) = .99$

$P(b) = .01$

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

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

Word2vec: how to learn vectors

- Given the set of positive and negative training instances, and an initial set of embedding vectors
- The goal of learning is to adjust those word vectors such that we:
  - **Maximize** the similarity of the target word, context word pairs (w , c_{pos}) drawn from the positive data
  - **Minimize** the similarity of the (w , c_{neg}) pairs drawn from the negative data.

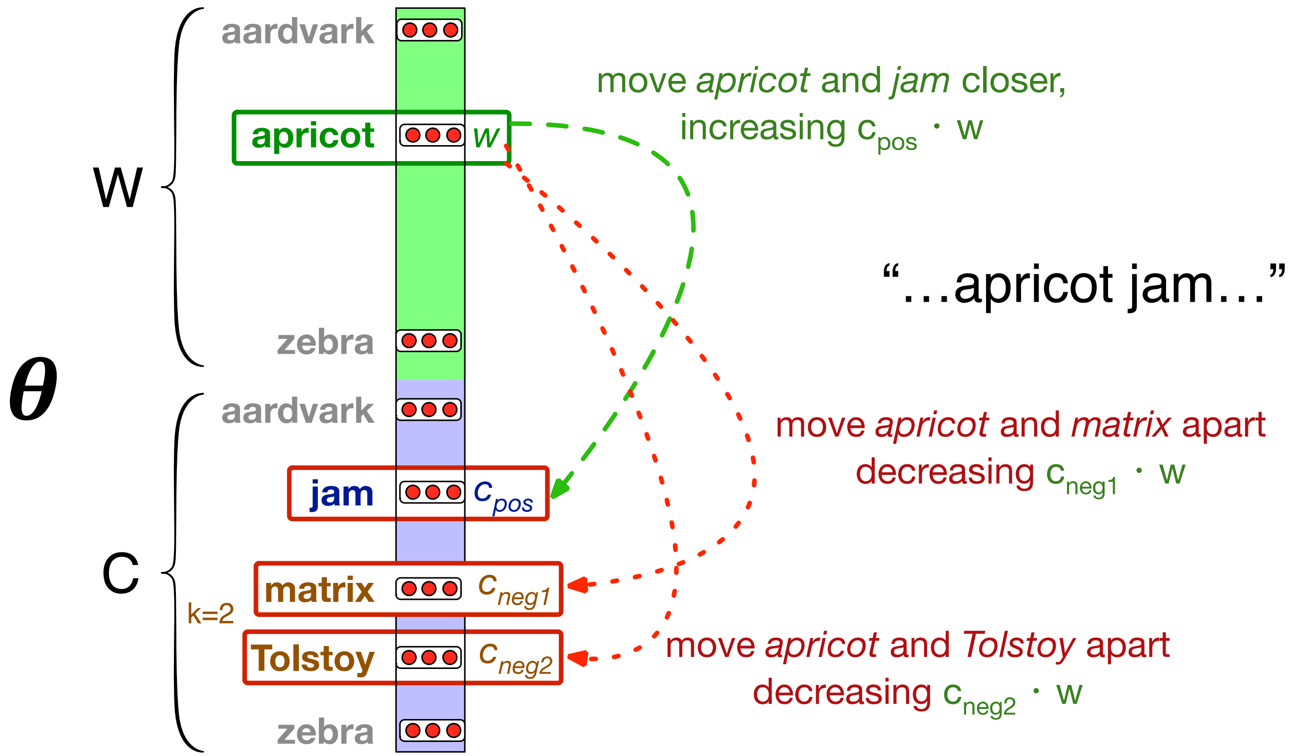
Loss function for one *w with* c_{pos}, c_{neg1} ...c_{negk}

- Maximize the similarity of the target with the actual context words, and minimize the similarity of the target with the *k* negative sampled non-neighbor words.

$$
\begin{aligned}
L_{\mathrm{CE}} &= -\log\left[P(+|w,c_{pos})\prod_{i=1}^{k}P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos})+\sum_{i=1}^{k}\log P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos})+\sum_{i=1}^{k}\log\left(1-P(+|w,c_{neg_i})\right)\right] \\
&= -\left[\log\sigma(c_{pos}\cdot w)+\sum_{i=1}^{k}\log\sigma(-c_{neg_i}\cdot w)\right]
\end{aligned}
$$

Learning the classifier

- How to learn?
  - Stochastic gradient descent!
- We'll adjust the word weights to
  - make the positive pairs more likely
  - and the negative pairs less likely,
  - over the entire training set.



Reminder: gradient descent

- At each step
  - Direction: We move in the reverse direction from the gradient of the loss function
  - Magnitude: we move the value of this gradient $\frac{d}{dw}L(f(x;w),y)$ weighted by a **learning rate** η
  - Higher learning rate means move *w* faster

$$w^{t+1} = w^{t} - h\frac{d}{dw}L(f(x;w),y)$$

The derivatives of the loss function

$$L_{\mathrm{CE}} = -\left[\log\sigma(c_{pos}\cdot w)+\sum_{i=1}^{k}\log\sigma(-c_{neg_i}\cdot w)\right]$$

$$\frac{\partial L_{CE}}{\partial c_{pos}} = [\sigma(c_{pos}\cdot w)-1]w$$

$$\frac{\partial L_{CE}}{\partial c_{neg}} = [\sigma(c_{neg}\cdot w)]w$$

$$\frac{\partial L_{CE}}{\partial w} = [\sigma(c_{pos}\cdot w)-1]c_{pos}+\sum_{i=1}^{k}[\sigma(c_{\mathrm{neg}_i}\cdot w)]c_{neg_i}$$

Update equation in SGD

Start with randomly initialized C and W matrices, then incrementally do updates

$$
\begin{aligned}
c_{pos}^{t+1} &= c_{pos}^{t} - \eta[\sigma(c_{pos}^{t} \cdot w^{t}) - 1]w^{t} \\
c_{neg}^{t+1} &= c_{neg}^{t} - \eta[\sigma(c_{neg}^{t} \cdot w^{t})]w^{t} \\
w^{t+1} &= w^{t} - \eta\left[[\sigma(c_{pos} \cdot w^{t}) - 1]c_{pos} + \sum_{i=1}^{k}[\sigma(c_{\mathrm{neg}_i} \cdot w^{t})]c_{neg_i}\right]
\end{aligned}
$$

Two sets of embeddings

- Skig-gram learns two sets of embeddings

  Target embeddings matrix W

  Context embedding matrix C

- 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 d-dimensional vectors as initial embeddings
- Train a classifier based on embedding similarity
  - 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.

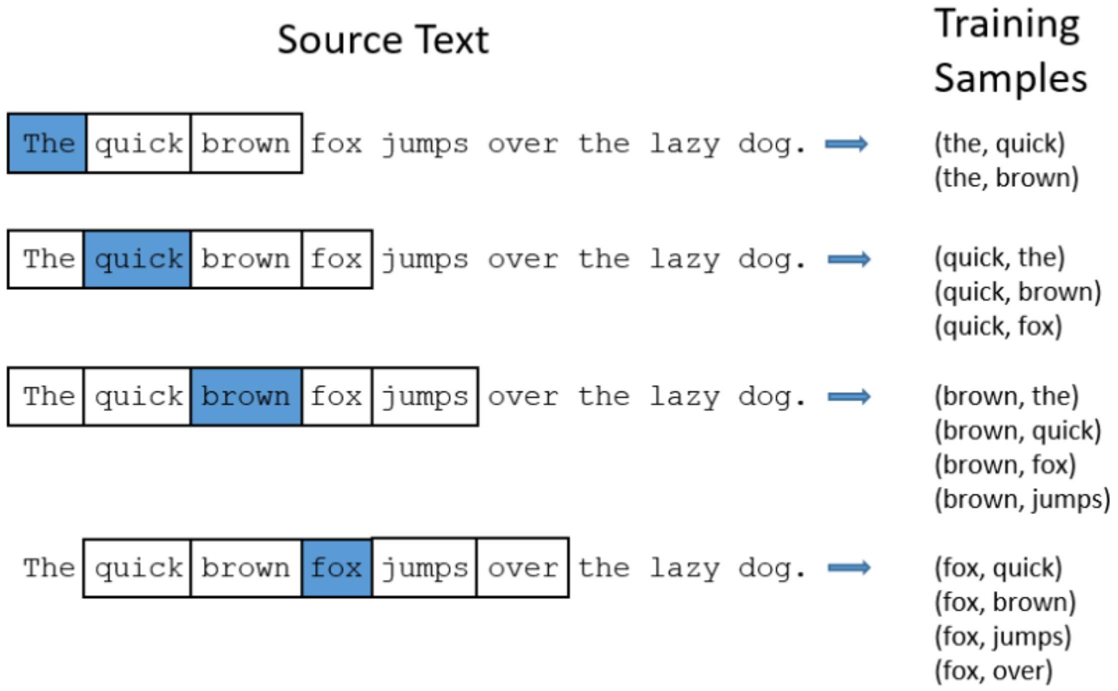
Some Tricks

- **Subsampling Frequent Words**

There are two “problems” with common words like “the”:

1.When looking at word pairs, (“fox”, “the”) doesn’t tell us much about the meaning of “fox”. “the” appears in the context of pretty much every word.

2.We will have many more samples of (“the”, …) than we need to learn a good vector for “the”.



$P(w_i)$ is the probability of *keeping* the word:

$$P(w_i) = (\sqrt{\frac{z(w_i)}{0.001}} + 1) \cdot \frac{0.001}{z(w_i)}$$

http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/

Some Tricks

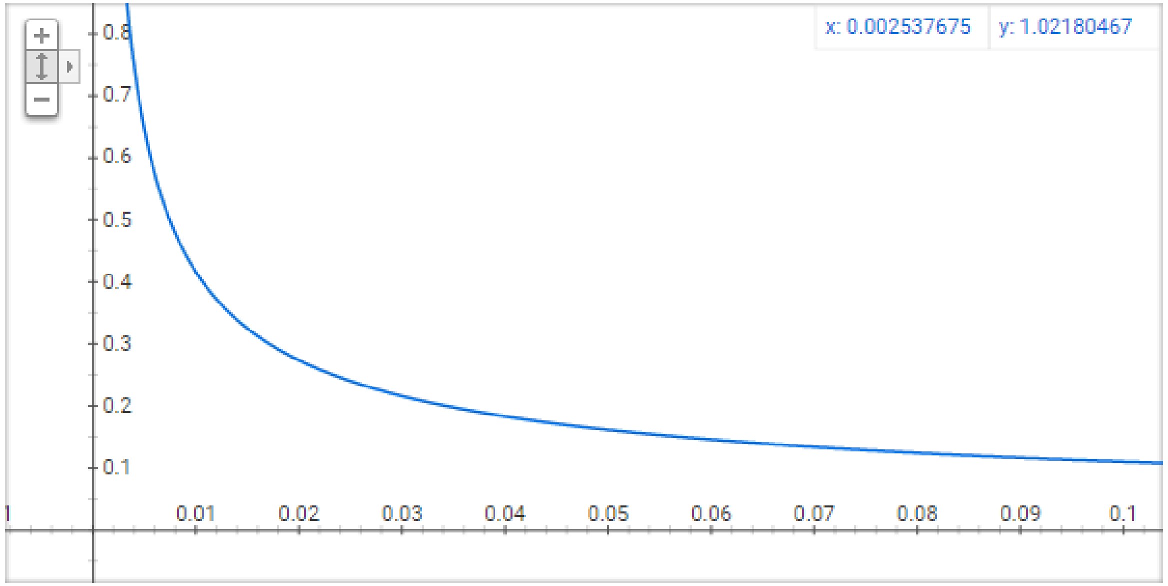
- **Subsampling Frequent Words**

There are two “problems” with common words like

1.When looking at word pairs, (“fox”, “the”) doesn’t
about the meaning of “fox”. “the” appears in the co
pretty much every word.

2.We will have many more samples of (“the”, …) th
to learn a good vector for “the”.

Graph for (sqrt(x/0.001)+1)*0.001/x



Training Samples

(the, quick)
(the, brown)

(quick, the)
(quick, brown)
(quick, fox)

(brown, the)
(brown, quick)
(brown, fox)
(brown, jumps)

(fox, quick)
(fox, brown)
(fox, jumps)
(fox, over)

$P(w_i)$ is the probability of *keeping* the word:

$$P(w_i) = (\sqrt{\frac{z(w_i)}{0.001}} + 1) \cdot \frac{0.001}{z(w_i)}$$

http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/

Subsampling Frequent Words

- If we have a window size of 10, and we remove a specific instance of “the” from our text:
  - As we train on the remaining words, “the” will not appear in any of their context windows.
  - We’ll have 10 fewer training samples where “the” is the input word.

Here are some interesting points in this function (again this is using the default sample value of 0.001).

- $P(w_i) = 1.0$ (100% chance of being kept) when $z(w_i) <= 0.0026$.
  - This means that only words which represent less than 0.26% of the total words will be subsampled.
- $P(w_i) = 0.5$ (50% chance of being kept) when $z(w_i) = 0.00746$.
- $P(w_i) = 0.033$ (3.3% chance of being kept) when $z(w_i) = 1.0$.
  - That is, if the corpus consisted entirely of word w_i, which of course is ridiculous.

Some interesting results

Word Analogies

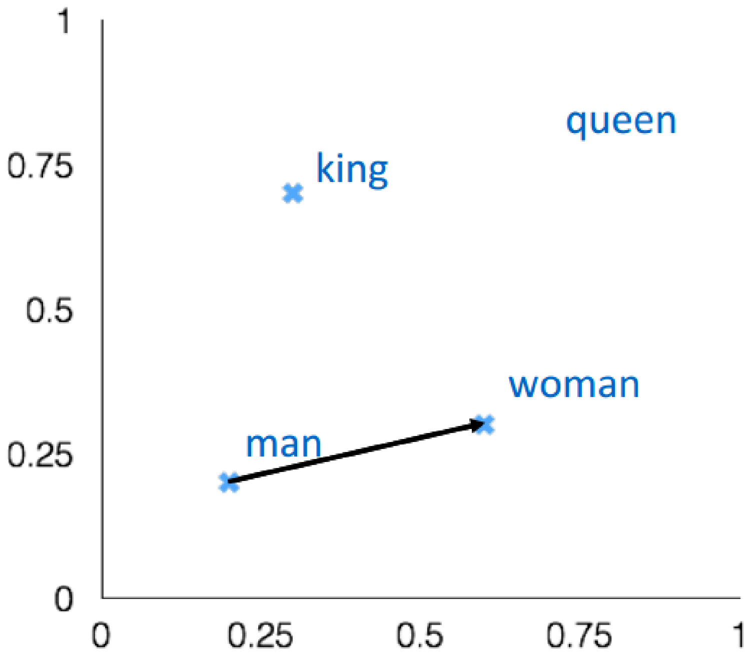
Test for linear relationships, examined by Mikolov et al. (2014)

a:b :: c:?

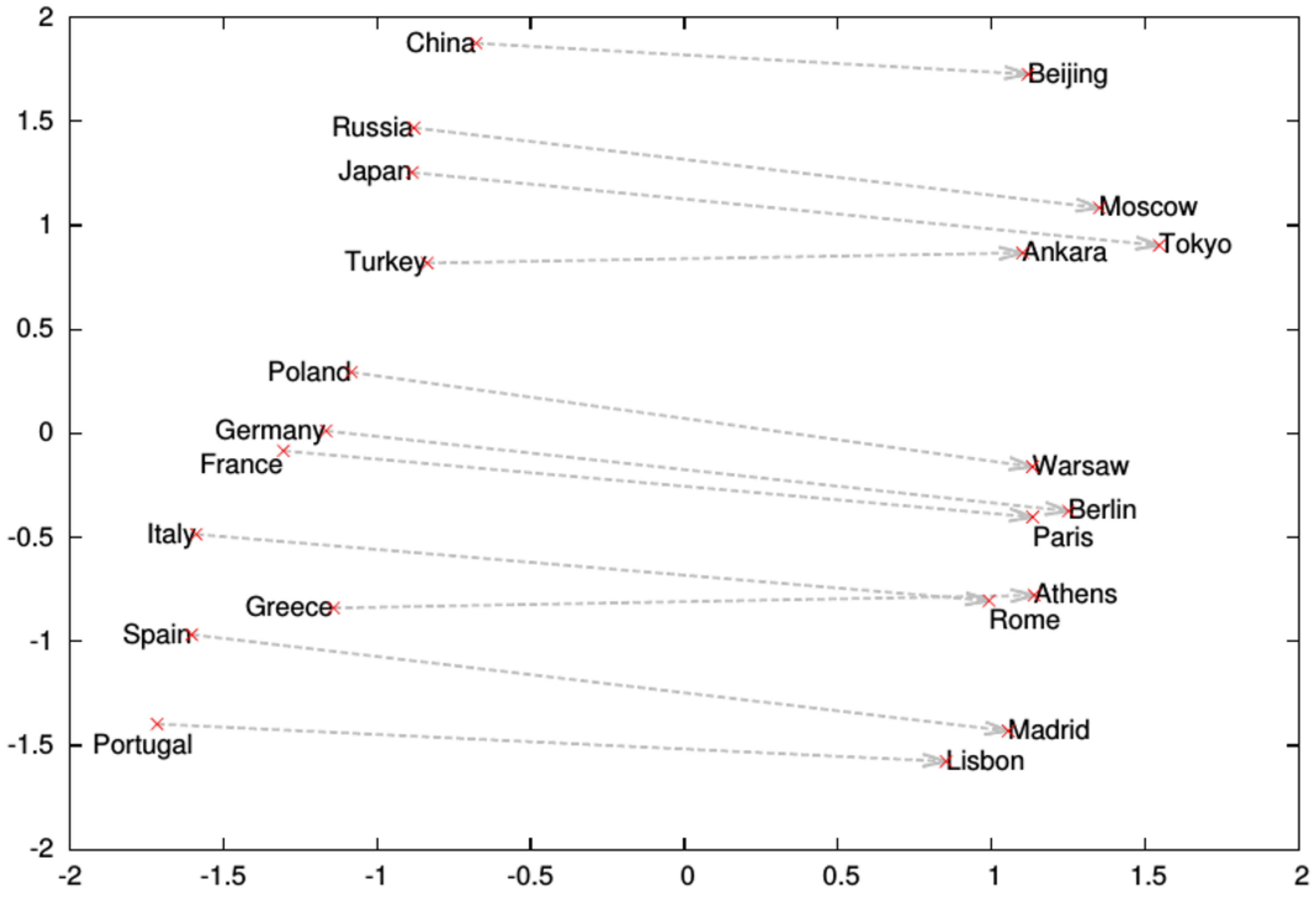
$$d = \arg\max_x \frac{(w_b - w_a + w_c)^T w_x}{||w_b - w_a + w_c||}$$

man:woman :: king:?

+	king	[0.30 0.70]
-	man	[0.20 0.20]
+	woman	[0.60 0.30]
	queen	[0.70 0.80]



Word analogies



Why not capture the co-occurrence matrix directly

With a co-occurrence matrix *X*

- 2 options: windows vs. full document
- Window: Similar to word2vec, use window around each word → captures both syntactic (POS) and semantic information
- Word-document co-occurrence matrix will give general topics (all sports terms will have similar entries) leading to "Latent Semantic Analysis"

Window based co-occurrence matrix

- Example corpus:
  - I like deep learning.
  - I like NLP.
  - I enjoy flying.

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Problems with simple co-occurrence matrix

Increase in size with vocabulary

Very high dimensional: requires a lot of storage

Subsequent classification models have sparsity issues

→ Models are less robust

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, similar to word2vec

- How to reduce the dimensionality?

Count based prediction: learn vectors by doing dimensionality reduction on a co-occurrence counts matrix.

1. Construct a large matrix of co-occurrence information
2. Factorize this matrix to yield a lower-dimensional matrix of words and features, where each row yields a vector representation for each word

Count 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

GloVe (Pennington et al, 2014)

Crucial insight: 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
$\dfrac{P(x \mid \text{ice})}{P(x \mid \text{steam})}$	large	small	~1	~1

GloVe (Pennington et al, 2014)

Crucial insight: Ratios of co-occurrence probabilities can encode meaning components

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

Encoding meaning in vector differences

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

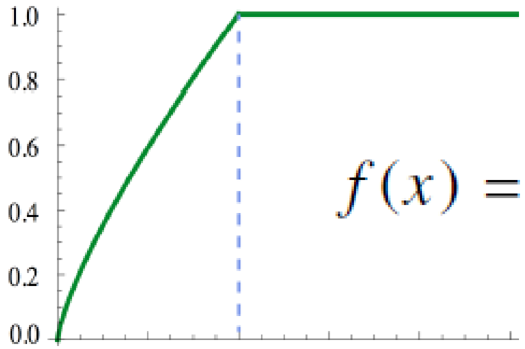
Glove: Objective Function

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

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

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

$f \sim$



$$f(x) = \begin{cases} (x/x_{\max})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases}$$

Details about GloVe

Original paper: https://nlp.stanford.edu/pubs/glove.pdf

Blogs with easy explanations:

- https://medium.com/sciforce/word-vectors-in-natural-language-processing-global-vectors-glove-51339db89639
- https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/?fbclid=IwAR3-pws3-K-Snfk6aqbixdxS8zFf-uuPDJ_0ipb94kWeygrdCSEqE9HWmNs
- https://towardsdatascience.com/light-on-math-ml-intuitive-guide-to-understanding-glove-embeddings-b13b4f19c010

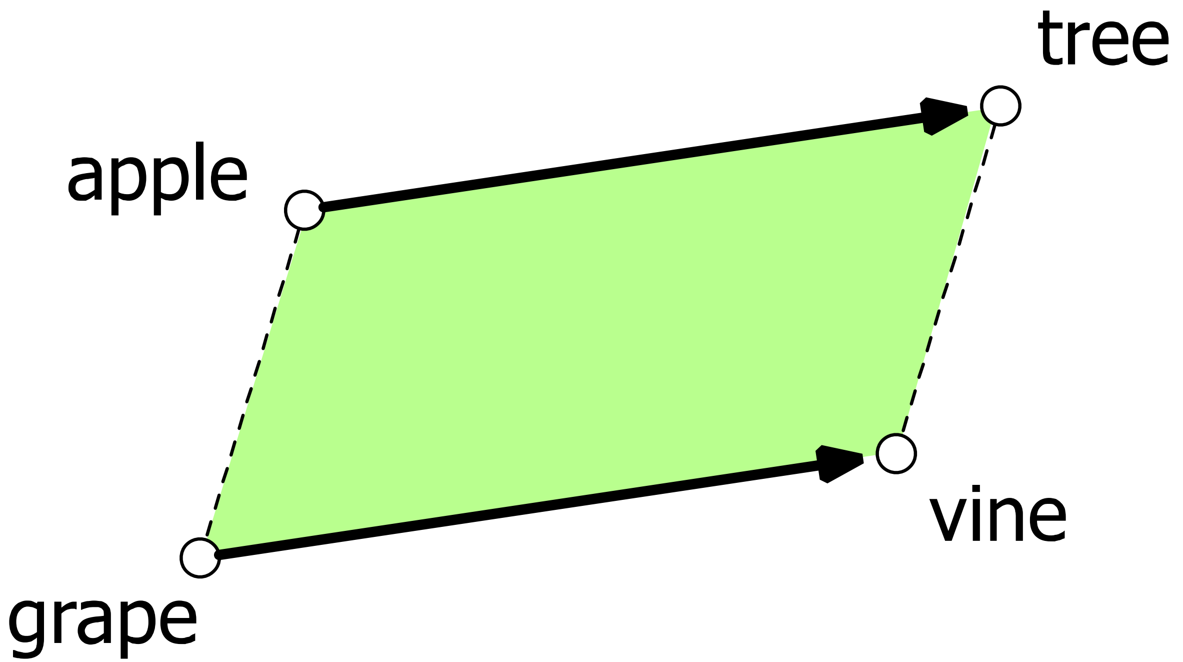
Properties of Embeddings

The kinds of neighbors depend on window size

- **Small windows** (C= +/- 2) : nearest words are syntactically similar words in same taxonomy
    - *Hogwarts* nearest neighbors are other fictional schools
        - *Sunnydale, Evernight, Blandings*
- **Large windows** (C= +/- 5) : nearest words are related words in same semantic field
    - *Hogwarts* nearest neighbors are Harry Potter world:
        - *Dumbledore, half-blood, Malfoy*

Analogical relations

- The classic parallelogram model of analogical reasoning (Rumelhart and Abrahamson 1973)
- To solve: *"apple is to tree as grape is to _____"*
- *Add* $\overrightarrow{\textit{tree}} - \overrightarrow{\textit{apple}}$ *to* $\overrightarrow{\textit{grape}}$ *to get* $\overrightarrow{\textit{vine}}$

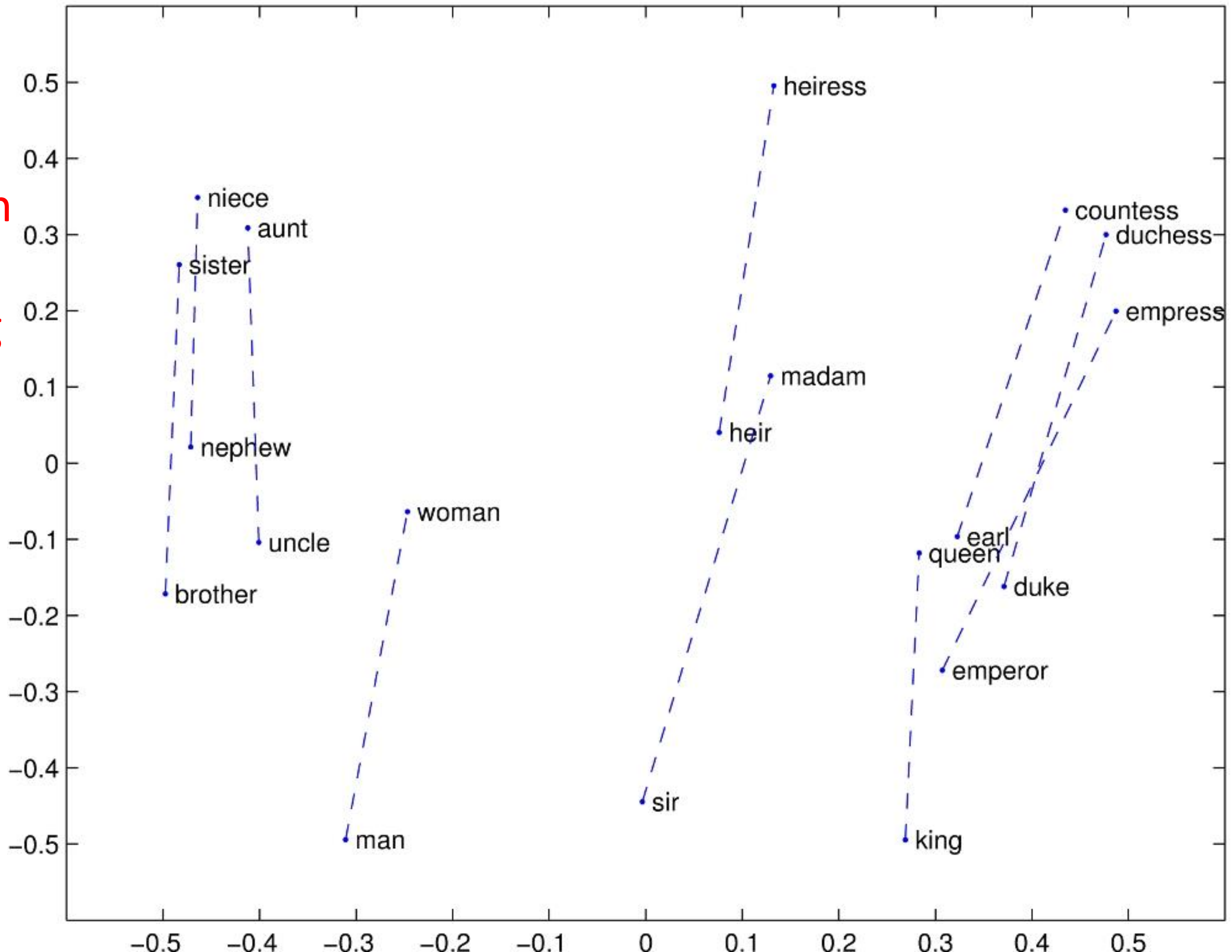


Analogical relations via parallelogram

- The parallelogram method can solve analogies with both sparse and dense embeddings (Turney and Littman 2005, Mikolov et al. 2013b)
- $\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ is close to $\overrightarrow{\text{queen}}$
- $\overrightarrow{\text{Paris}} - \overrightarrow{\text{France}} + \overrightarrow{\text{Italy}}$ is close to $\overrightarrow{\text{Rome}}$
- For a problem a:a*::b:b*, the parallelogram method is:

$$\hat{b}^* = \underset{x}{\operatorname{argmax}}\ \operatorname{distance}(x, a^* - a + b)$$

Structure in GloVE Embedding space



Caveats with the parallelogram method

- It only seems to work for frequent words, small distances and certain relations (relating countries to capitals, or parts of speech), but not others. (Linzen 2016, Gladkova et al. 2016, Ethayarajh et al. 2019a)

- Understanding analogy is an open area of research (Peterson et al. 2020)

Embeddings as a window onto historical semantics

Train embeddings on different decades of historical text to see meanings shift

~30 million books, 1850-1990, Google Books data

William L. Hamilton, Jure Leskovec, and Dan Jurafsky. 2016. Diachronic Word Embeddings Reveal Statistical Laws of Semantic Change. Proceedings of ACL.

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 *NeurIPS*, 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

Algorithms that use embeddings as part of e.g., hiring searches for programmers, might lead to bias in hiring

Historical embedding as a tool to study cultural biases

Garg, N., Schiebinger, L., Jurafsky, D., and Zou, J. (2018). Word embeddings quantify 100 years of gender and ethnic stereotypes. Proceedings of the National Academy of Sciences 115(16), E3635–E3644.

- Compute a **gender or ethnic bias** for each adjective: e.g., how much closer the adjective is to "woman" synonyms than "man" synonyms, or names of particular ethnicities
    - Embeddings for **competence** adjective (*smart, wise, brilliant, resourceful, thoughtful, logical)* are biased toward men, a bias slowly decreasing 1960-1990
    - Embeddings for **dehumanizing** adjectives (barbaric, monstrous, bizarre) were biased toward Asians in the 1930s, bias decreasing over the 20th century.
- These match the results of old surveys done in the 1930s

Appendix

Word2Vec

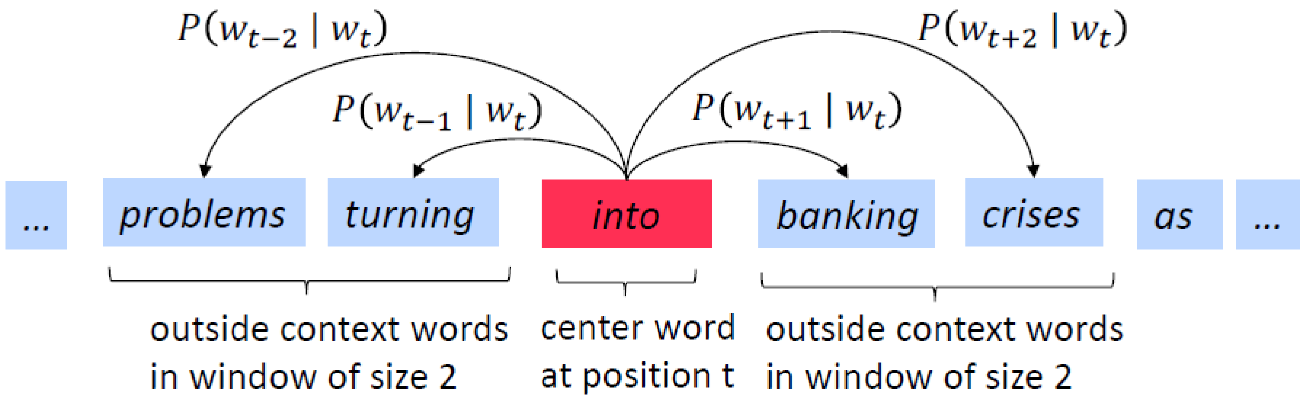
Word2vec (Mikolov et al. 2013) is a framework for learning word vectors

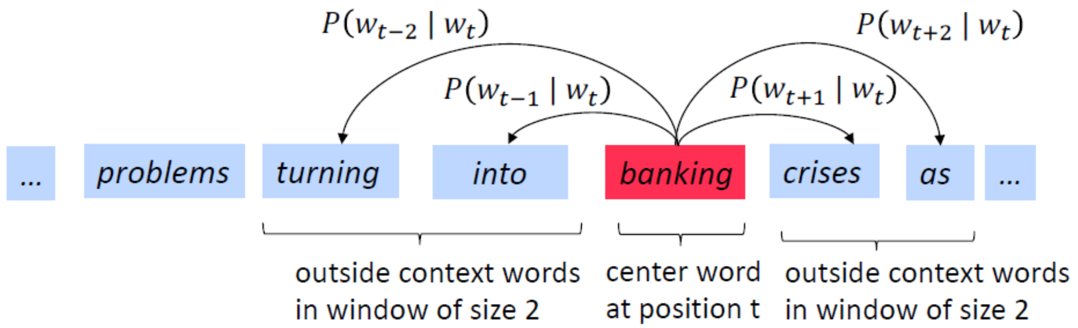
Idea:

- We have a large corpus of text
- Every word in a fixed vocabulary is represented by a vector
- Go through each position *t* in the text, which has a center word *c* and context ("outside") words *o*
- Use the similarity of the word vectors for *c* and *o* to calculate the probability of *o* given *c* (or vice versa)
- Keep adjusting the word vectors to maximize this probability

Word2Vec

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





Word2Vec: Objective Function

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size *m*, given center word w_j.

Likelihood = $L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$

θ is all variables to be optimized

sometimes called *cost* or *loss* function

The objective function $J(\theta)$ is the (average) negative log likelihood:

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j} \mid w_t; \theta)$$

Word2Vec: Objective Function

- We want to minimize the objective function:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m\leq j\leq m\\ j\neq 0}} \log P\left(w_{t+j} \mid w_t;\theta\right)$$

- Question: How to calculate $P\left(w_{t+j} \mid w_t;\theta\right)$?
- Answer: We will *use two* vectors per word *w*:
  - v_w when *w* is a center word
  - u_w when *w* is a context word
- Then for a center word *c* and a context word *o*:

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w\in V}\exp(u_w^T v_c)}$$

Word2Vec: Prediction Function

Exponentiation makes anything positive

$$P(o|c) = \frac{\exp(\boxed{u_o^T v_c})}{\boxed{\sum_{w \in V} \exp(u_w^T v_c)}}$$

Dot product compares similarity of *o* and *c*.
$u^T v = u.v = \sum_{i=1}^{n} u_i v_i$
Larger dot product = larger probability

Normalize over entire vocabulary
to give probability distribution

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

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

Gradient calculation

- Go to the while board and do some math

To train the model, compute all vector gradients

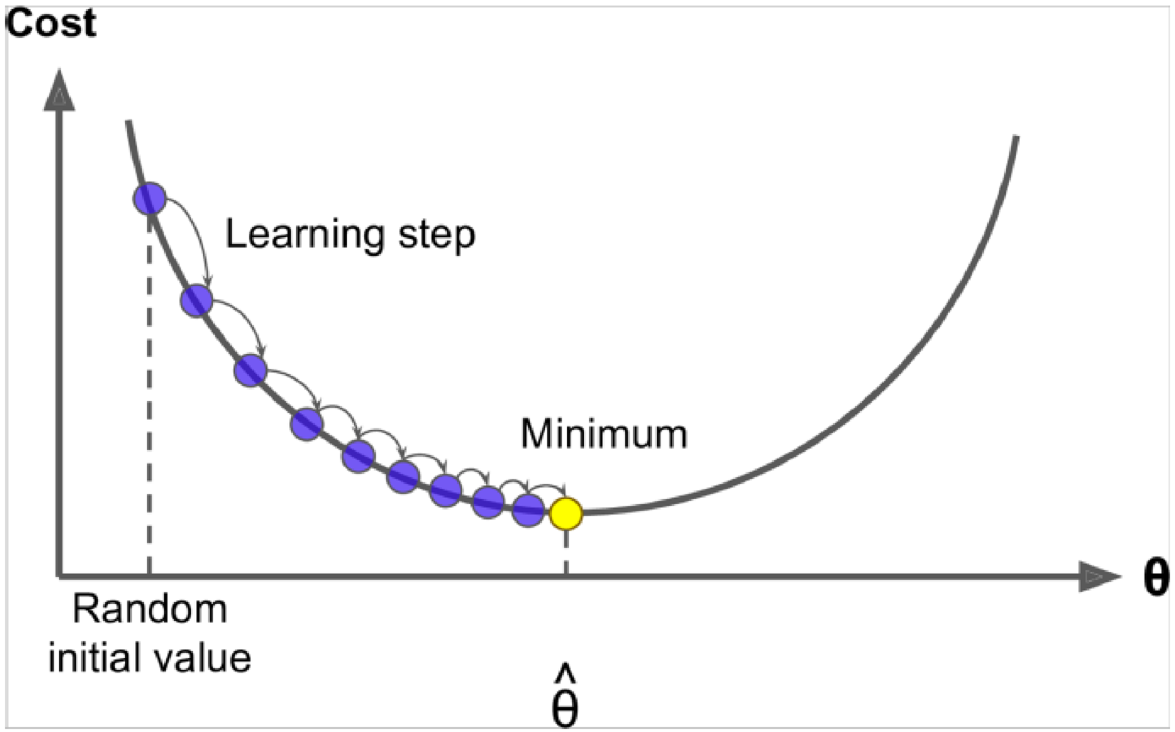
- Recall: θ represents all model parameters, in one long vector
- In our case with *d*-dimensional vectors and *V*-many words:

$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \vdots \\ v_{zebra} \\ u_{aardvark} \\ u_a \\ \vdots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$

- Remember: every word has two vectors
- We optimize these parameters by walking down the gradient

Optimization

- We have a cost function $J(\theta)$ we want to minimize
- **Gradient Descent** is an algorithm to minimize $J(\theta)$
- Idea: for current value of θ, calculate gradient of $J(\theta)$, then take small step in direction of negative gradient. Repeat.



Note: Our objectives may not be convex like this :(

Gradient Descent

- Update equation (in matrix notation):

$$\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J(\theta)$$

α = *step size* or *learning rate*

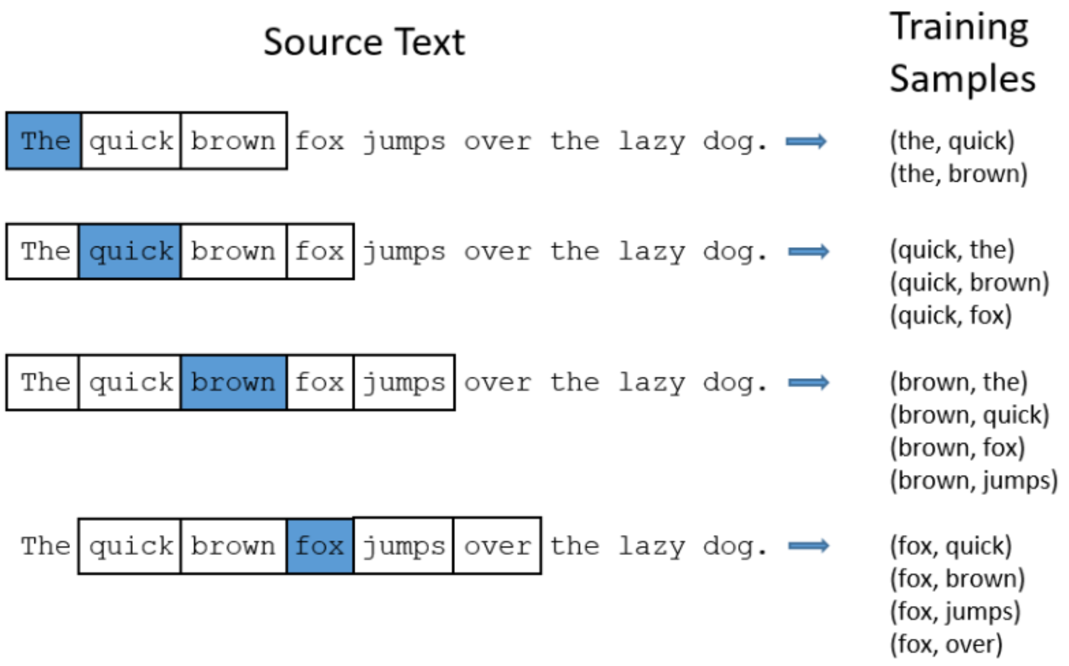
- Update equation (for single parameter):

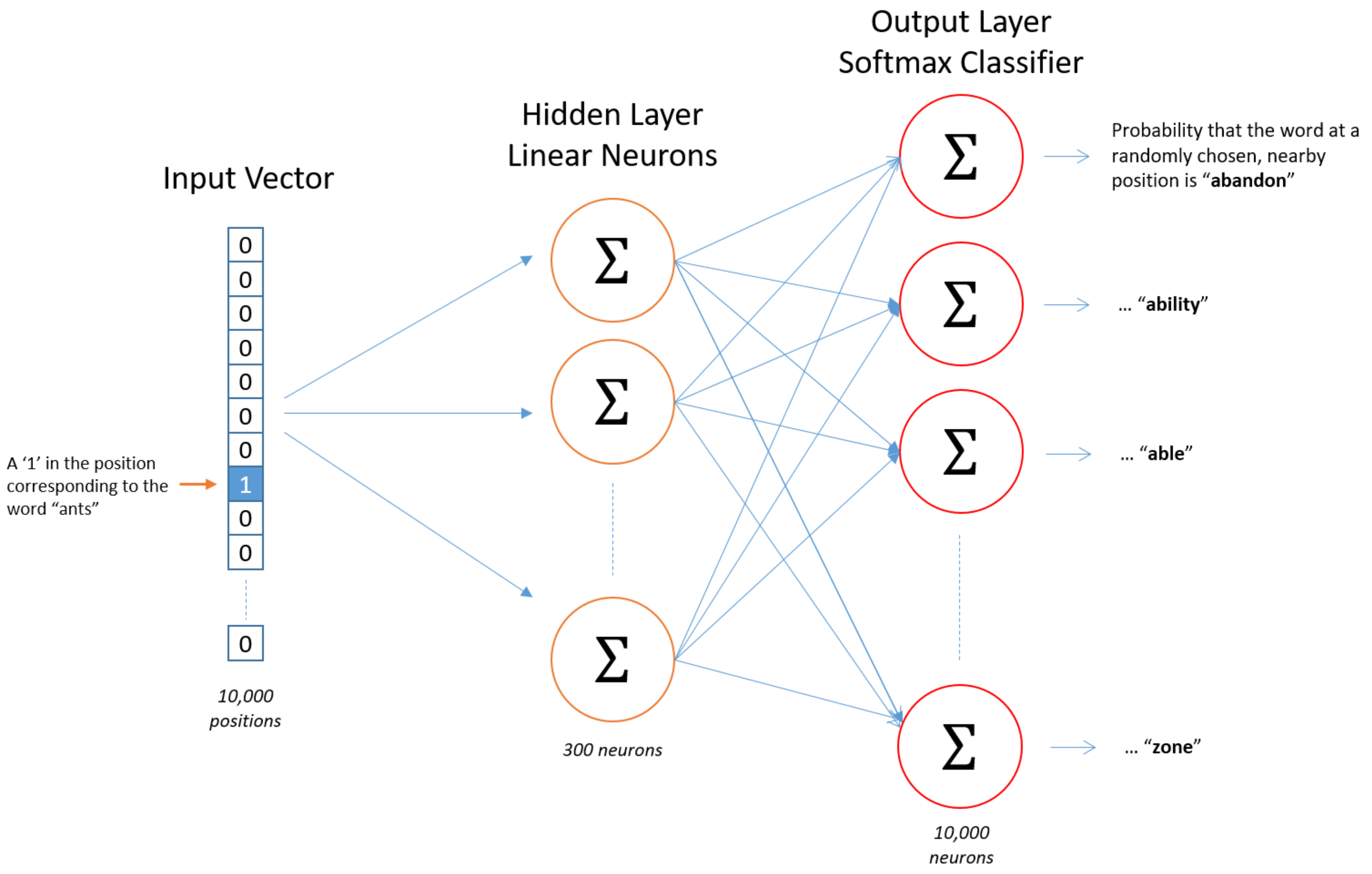
$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial}{\partial \theta_j^{old}} J(\theta)$$

Word2Vec

The output probabilities are going to relate to how likely it is find each vocabulary word nearby our input word.

For example, if you gave the trained network the input word “Soviet”, the output probabilities are going to be much higher for words like “Union” and “Russia” than for unrelated words like “watermelon” and “kangaroo”.





https://towardsdatascience.com/word2vec-skip-gram-model-part-1-intuition-78614e4d6e0b