Distributed Representation

SLP 6.8. Word2vec

https://web.stanford.edu/~jurafsky/slp3/6.pdf

NLP in Action 6. Reasoning with word vectors

Outline

- Sparse vs. Dense representations
- Wor2vec
- Skip-Gram algorithm
- Reasoning with word vectors

Why dense vectors?

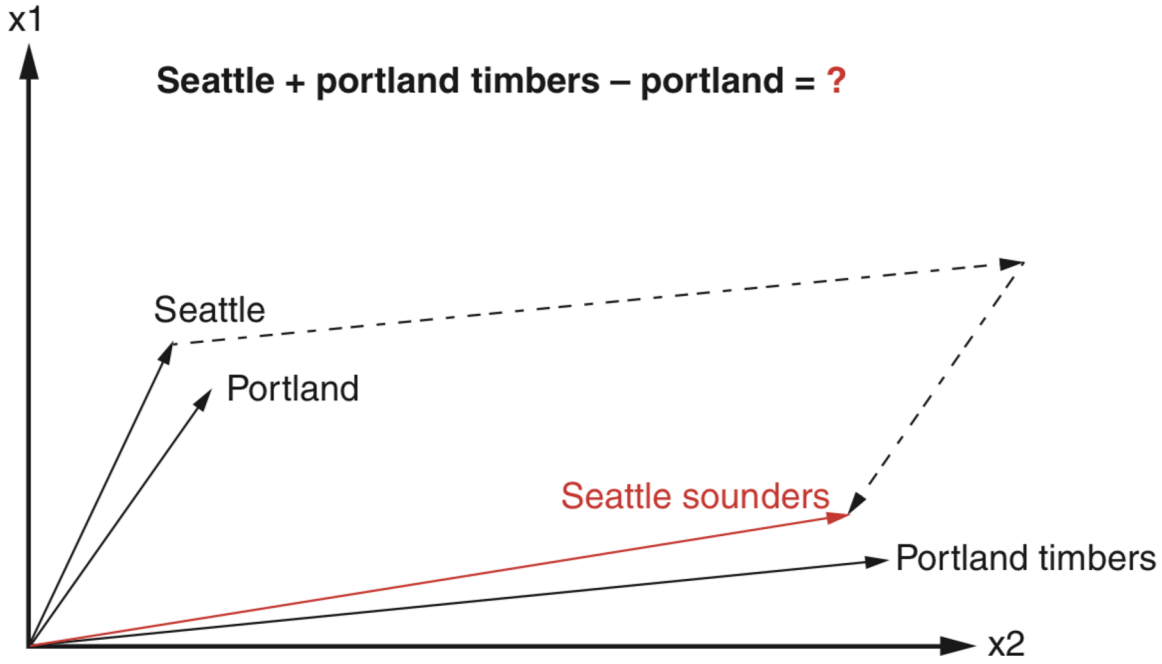
- Tf-idf and one-hot encodings are
  - long (length |V|= 20,000 to 50,000)
  - sparse (most elements are zero)
- Dense vectors
  - Are easier to use short vectors as features in machine learning
  - Generalize better than storing explicit counts
  - May do better at capturing synonymy
- In practice, dense vectors work better

Word Vectors (word embeddings)

- Word vectors are numerical vector representations of word semantics, or meaning, including literal and implied meaning
- Word vectors can capture the connotation of words, like “peopleness,” “animalness,” “placeness,” “thingness,” and even “conceptness.”
- They combine all that into a dense vector of floating point values, which enables queries and logical reasoning
    - Semantic Queries
        - “She invented something to do with physics in Europe in the early 20th century.”
        - >> answer_vector = wv['woman'] + wv['Europe'] + wv[physics'] +wv['scientist'] - wv['male'] - wv['man']
    - Analogy questions
        - Marie Curie is to science as who is to music?
        - MARIE CURIE : SCIENCE :: ? : MUSIC
        - >> wv['Marie_Curie'] - wv['science'] + wv['music']

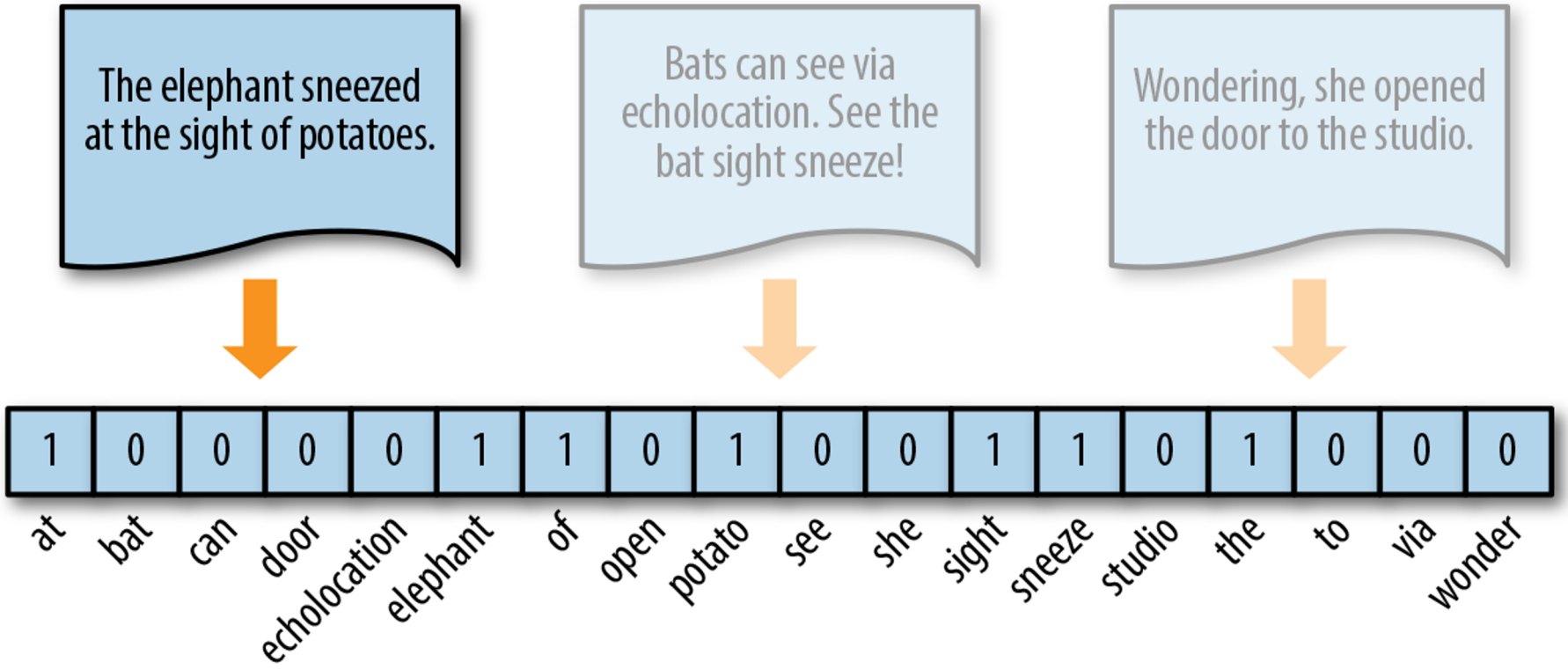
Reasoning with Word Vectors

Portland Timbers + Seattle - Portland = ?



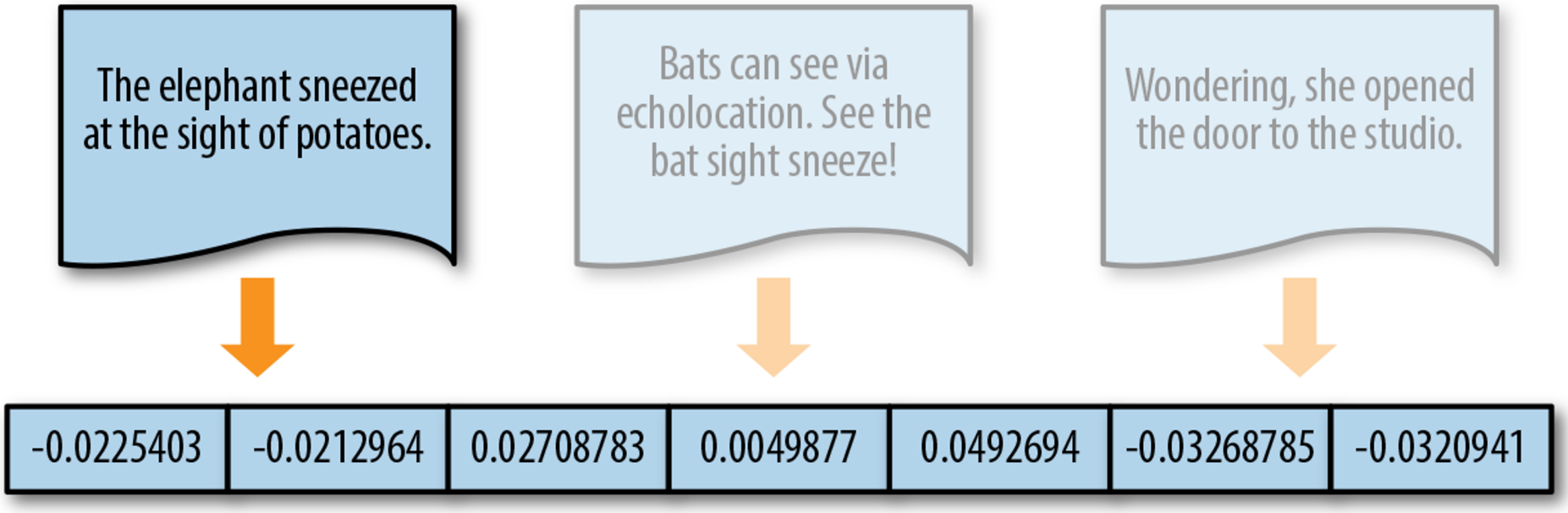
$$\begin{bmatrix} 0.0168 \\ 0.007 \\ 0.247 \\ \dots \end{bmatrix} + \begin{bmatrix} 0.093 \\ -0.028 \\ -0.214 \\ \dots \end{bmatrix} - \begin{bmatrix} 0.104 \\ 0.0883 \\ -0.318 \\ \dots \end{bmatrix} = \begin{bmatrix} 0.006 \\ -0.109 \\ 0.352 \\ \dots \end{bmatrix}$$

One-hot encoding



출처: Applied Text Analysis with Python - Bengfort, Bilbro, Ojeda (O'REilly)

Dense vector



출처: Applied Text Analysis with Python - Bengfort, Bilbro, Ojeda (O'REilly)

Word2vec (Mikolov, 2013)

- **Idea: predict rather than count**
- The idea comes from **neural language modeling**
  - Bengio et al. (2003)
  - Collobert et al. (2011)
- Mikolov and his teammates released the software for creating these word vectors and called it Word2vec
- Popular embedding method
- Very fast to train

Thomas Mikolov, Ilya Sutskever, Kai Chen, Gregory Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 2013.

Use text as implicitly supervised training data!

- Instead of counting how often each word occurs near "apricot"
- Train a classifier on a **binary prediction task**:
  - Is *w* likely to show up near "apricot"?
- A word ***s*** near *apricot*
  - acts as gold ‘correct answer’ to the question
  - “Is word *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**
- No need for hand-labeled supervision

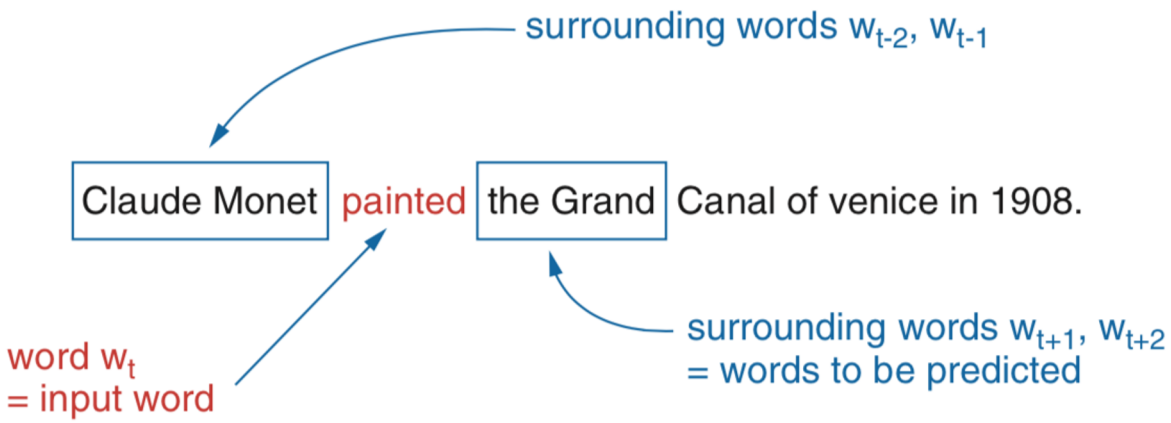
How to compute Word2vec representations

- The **skip-gram** approach predicts the context of words (output words) from a word of interest (the input word)
    - skip-gram with negative sampling (SGNS)
- The **continuous bag-of-words (CBOW)** approach predicts the target word (the output word) from the nearby words (input words)

Skip-gram

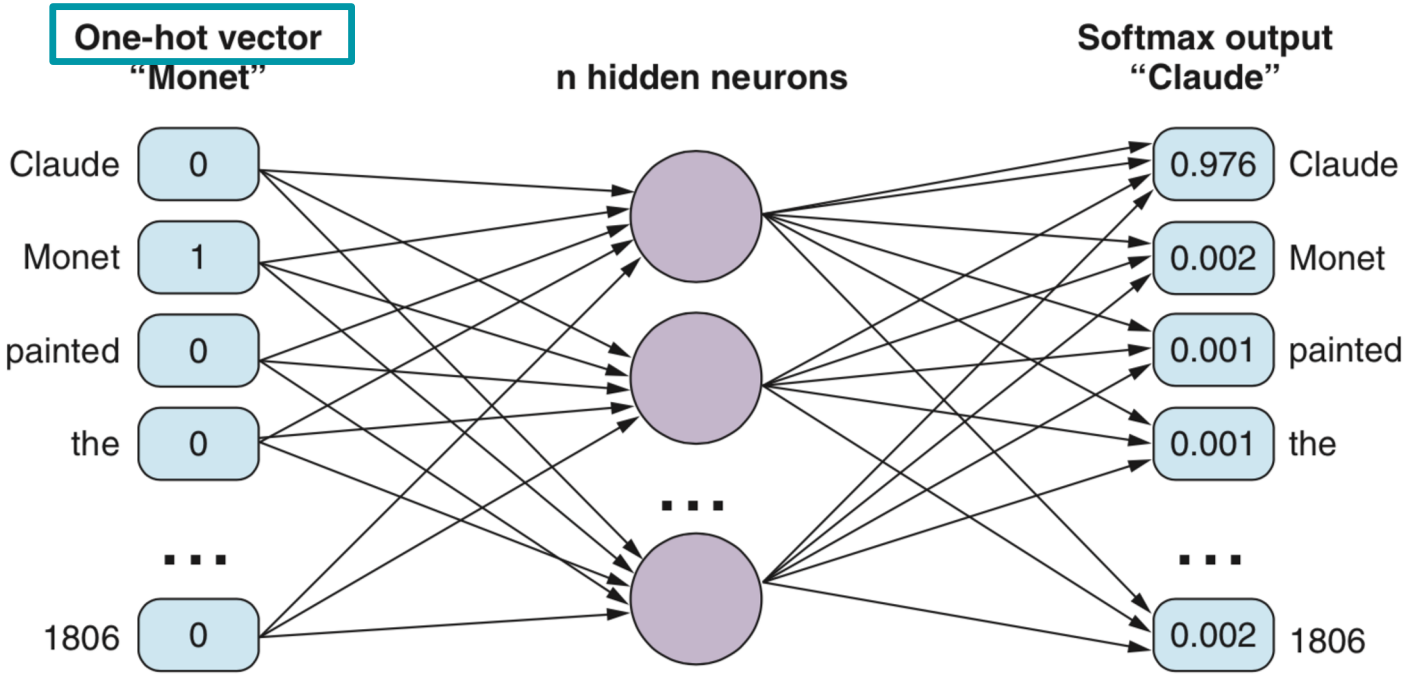
Skip-gram

- Skip-grams are n-grams that contain gaps because you skip over intervening tokens
  - E.g., predicting “Claude” from the input token “painted,” and you skip over the token “Monet.”



Skip-gram

- Skip-grams are n-grams that contain gaps because you skip over intervening tokens
  - E.g., predicting “Claude” from the input token “painted,” and you skip over the token “Monet.”
  - Input and output layers contain M neurons (M = number of words in the vocabulary)
  - N is the number of vector dimensions used to represent a word

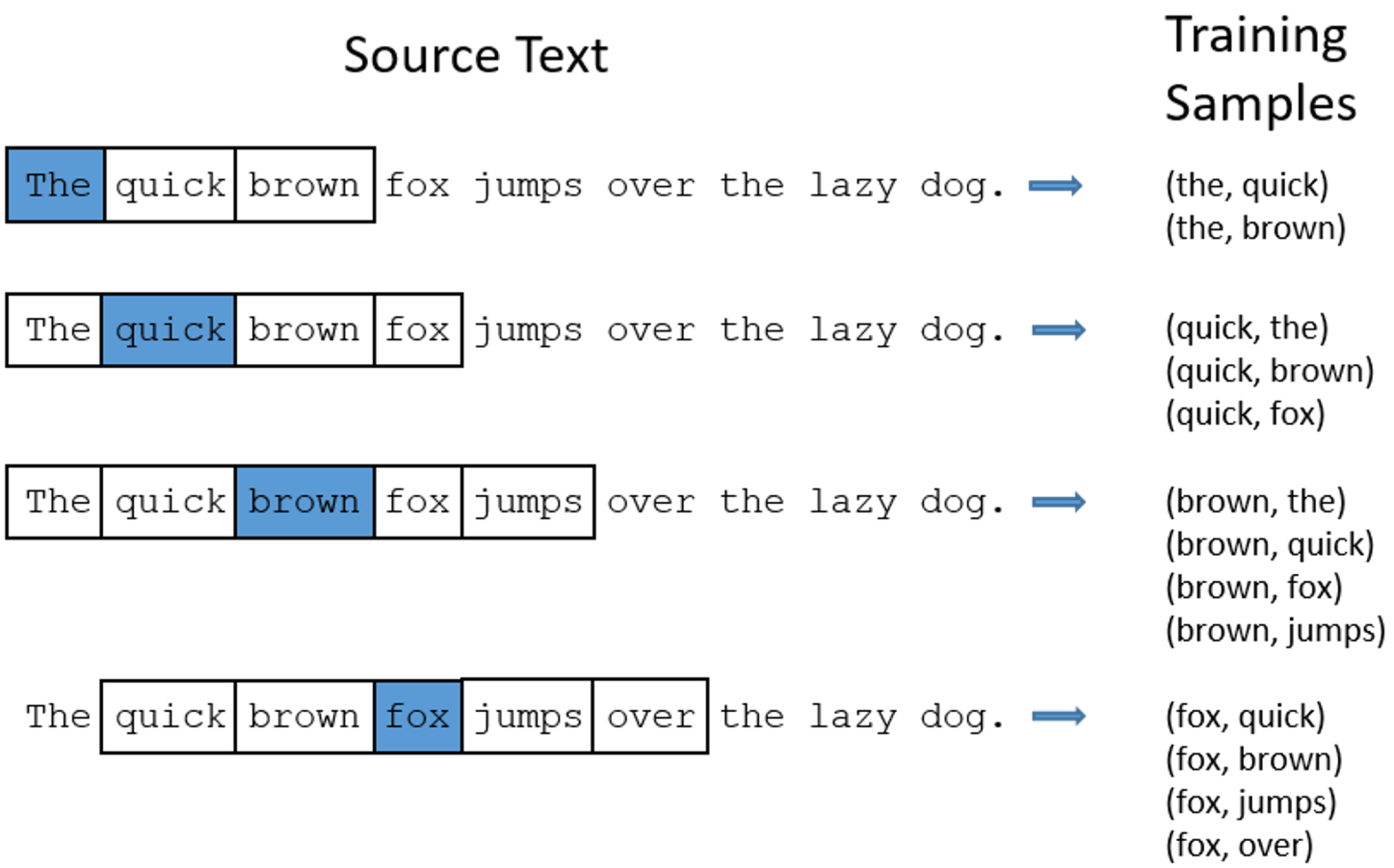


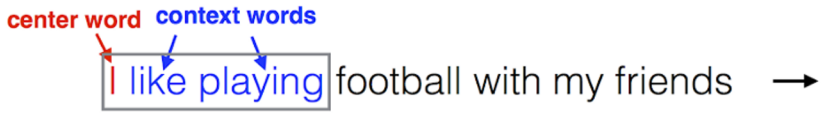
Built the Training set

>>> sentence = "Claude Monet painted the Grand Canal of Venice in 1806."

Input word w_t	Expected output w_{t-2}	Expected output w_{t-1}	Expected output w_{t+1}	Expected output w_{t+2}
Claude			Monet	painted
Monet		Claude	painted	the
painted	Claude	Monet	the	Grand
the	Monet	painted	Grand	Canal
Grand	painted	the	Canal	of
Canal	the	Grand	of	Venice
of	Grand	Canal	Venice	in
Venice	Canal	of	in	1908
in	of	Venice	1908	
1908	Venice	in		

Example


Source Text
Training Samples
The quick brown fox jumps over the lazy dog.
(the, quick)
(the, brown)
The quick brown fox jumps over the lazy dog.
(quick, the)
(quick, brown)
(quick, fox)
The quick brown fox jumps over the lazy dog.
(brown, the)
(brown, quick)
(brown, fox)
(brown, jumps)
The quick brown fox jumps over the lazy dog.
(fox, quick)
(fox, brown)
(fox, jumps)
(fox, over)



Center Word	Context Words
[1, 0, 0, 0, 0, 0, 0]	[0, 1, 0, 0, 0, 0, 0] [0, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0]	[1, 0, 0, 0, 0, 0, 0] [0, 0, 1, 0, 0, 0, 0] [0, 0, 0, 1, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0]	[1, 0, 0, 0, 0, 0, 0] [0, 1, 0, 0, 0, 0, 0] [0, 0, 0, 1, 0, 0, 0] [0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0]	[0, 1, 0, 0, 0, 0, 0] [0, 0, 1, 0, 0, 0, 0] [0, 0, 0, 0, 1, 0, 0] [0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0]	[0, 0, 1, 0, 0, 0, 0] [0, 0, 0, 1, 0, 0, 0] [0, 0, 0, 0, 0, 1, 0] [0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0]	[0, 0, 0, 1, 0, 0, 0] [0, 0, 0, 0, 1, 0, 0] [0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 1]	[0, 0, 0, 0, 1, 0, 0] [0, 0, 0, 0, 0, 1, 0]

Build the Train data: tokenization and vocabulary

```
corpus = [
    'he is a king',
    'she is a queen',
    'he is a man',
    'she is a woman',
    'warsaw is poland capital',
    'berlin is germany capital',
    'paris is france capital',
]

def tokenize_corpus(corpus):
    tokens = [x.split() for x in corpus]
    return tokens
tokenized_corpus = tokenize_corpus(corpus)
```

Word tokenization

```
[['he', 'is', 'a', 'king'],
 ['she', 'is', 'a', 'queen'],
 ['he', 'is', 'a', 'man'],
 ['she', 'is', 'a', 'woman'],
 ['warsaw', 'is', 'poland', 'capital'],
 ['berlin', 'is', 'germany', 'capital'],
 ['paris', 'is', 'france', 'capital']]
```

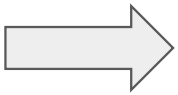
```
vocabulary = []
for sentence in tokenized_corpus:
    for token in sentence:
        if token not in vocabulary:
            vocabulary.append(token)

word2idx = {w: idx for (idx, w) in enumerate(vocabulary)}
idx2word = {idx: w for (idx, w) in enumerate(vocabulary)}

vocabulary_size = len(vocabulary)
```

Create vocabulary

```
0: 'he',
1: 'is',
2: 'a',
3: 'king',
4: 'she',
5: 'queen',
6: 'man',
7: 'woman',
8: 'warsaw',
9: 'poland',
10: 'capital',
11: 'berlin',
12: 'germany',
13: 'paris',
14: 'france'
```

Build the Train data: generate pairs center, context

he is
he a
is he
is a
is king
a he
a is
a king

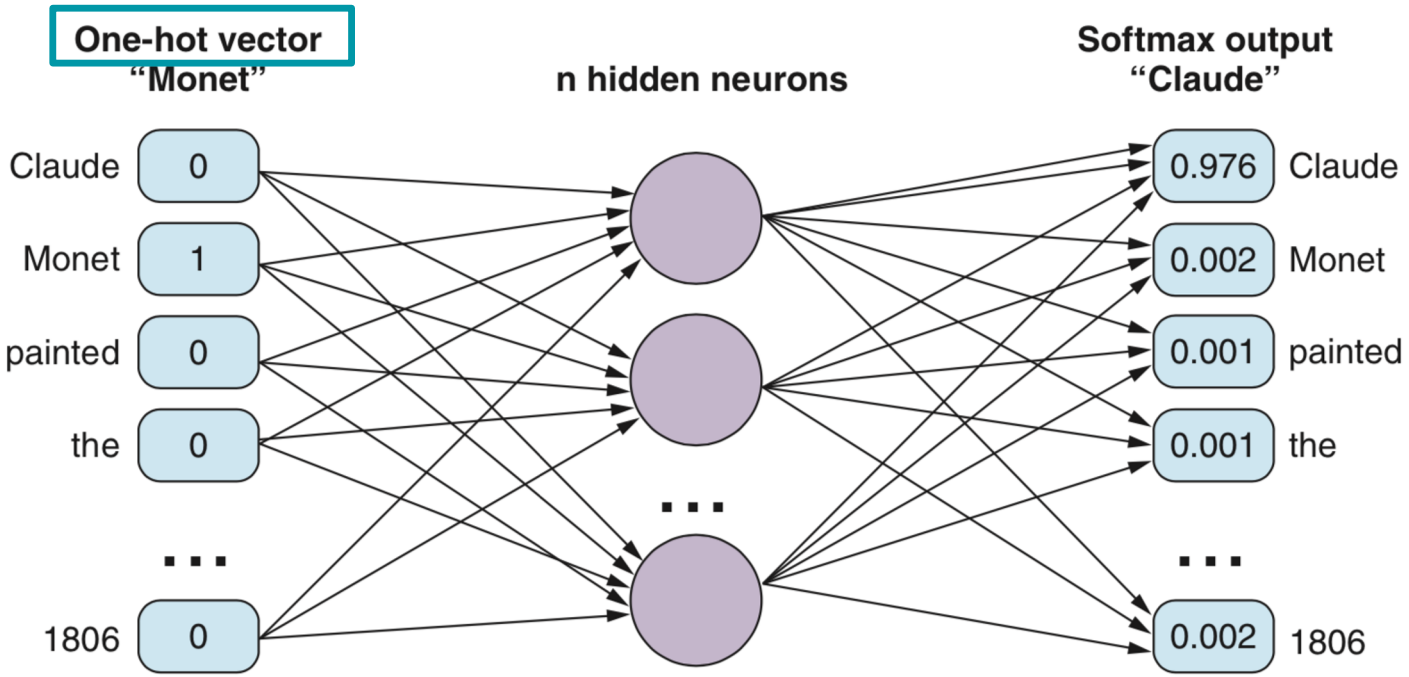


array([[0, 1], [0, 2], …])

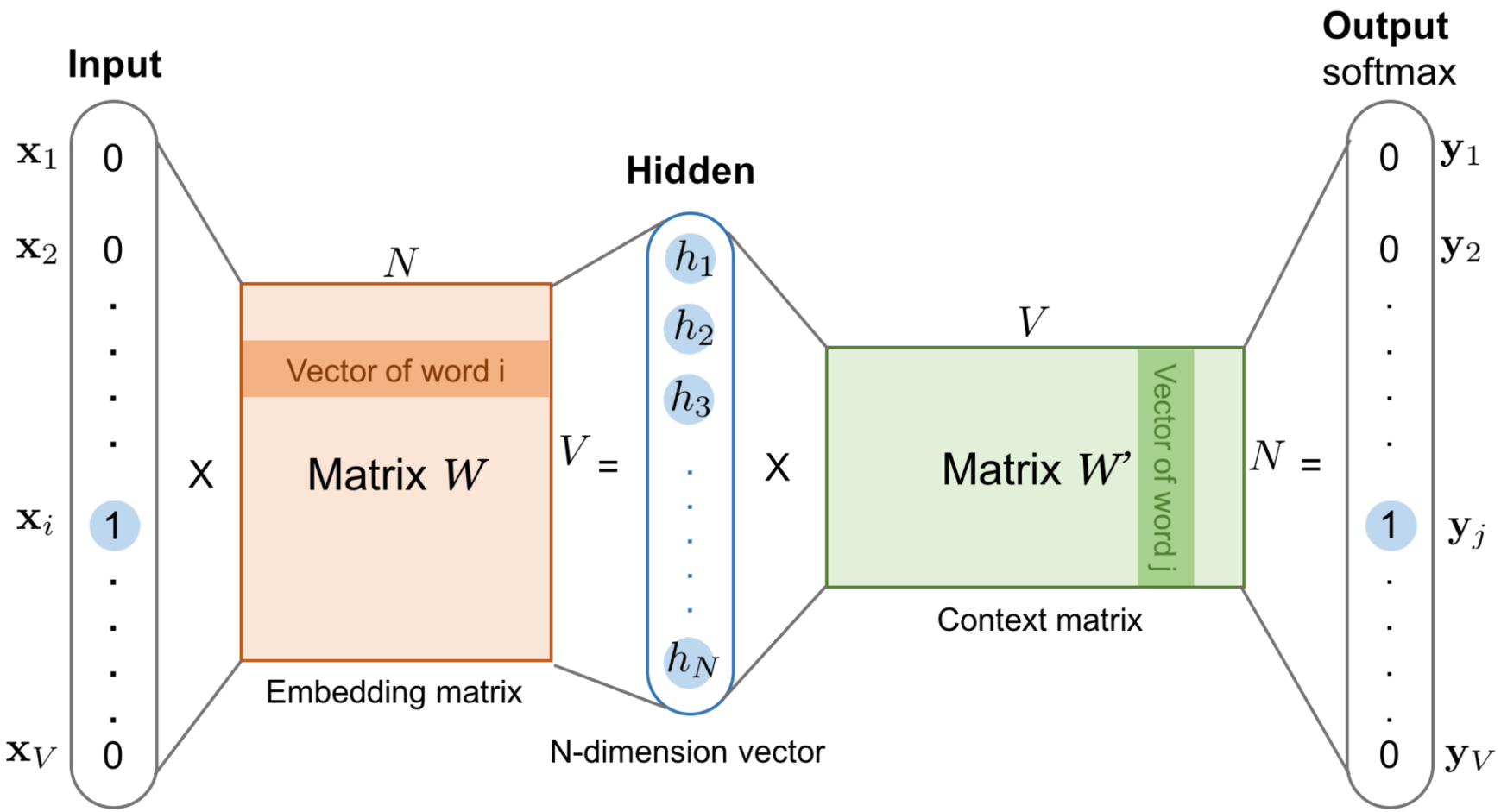
```
window_size = 2
idx_pairs = []

for sentence in tokenized_corpus: # for each sentence
    indices = [word2idx[word] for word in sentence]

    for center_word_pos in range(len(indices)):    # for each word as center word
        for w in range(-window_size, window_size + 1):        # for each window position
            context_word_pos = center_word_pos + w

            if context_word_pos < 0 or context_word_pos >= len(indices) or center_word_pos == context_word_pos:
                # check index is not out of sentence
                continue
            context_word_idx = indices[context_word_pos]
            idx_pairs.append((indices[center_word_pos], context_word_idx))
```

Skip-gram

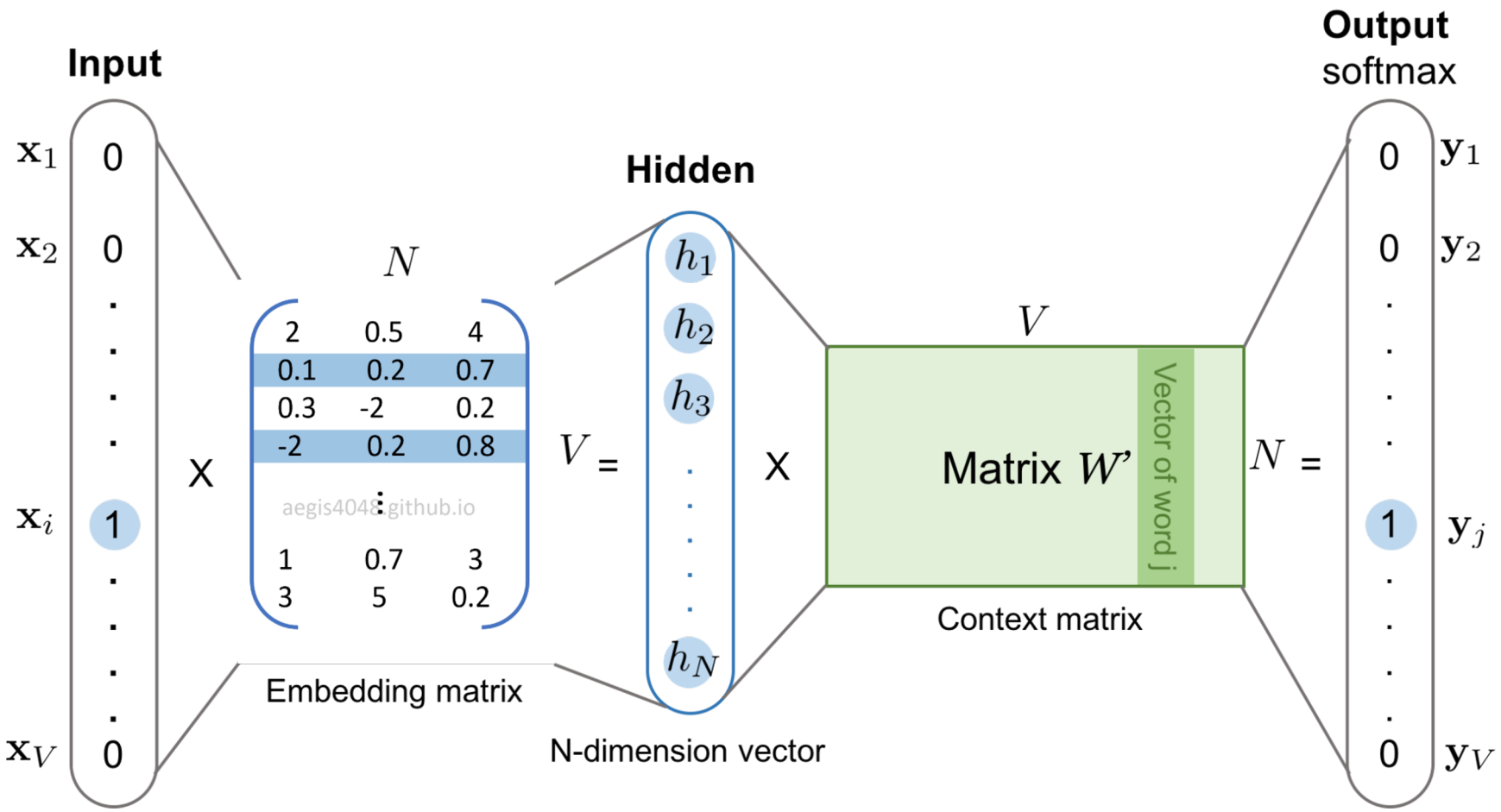
- Skip-grams are n-grams that contain gaps because you skip over intervening tokens
  - e.g., predicting “Claude” from the input token “painted,” and you skip over the token “Monet.”



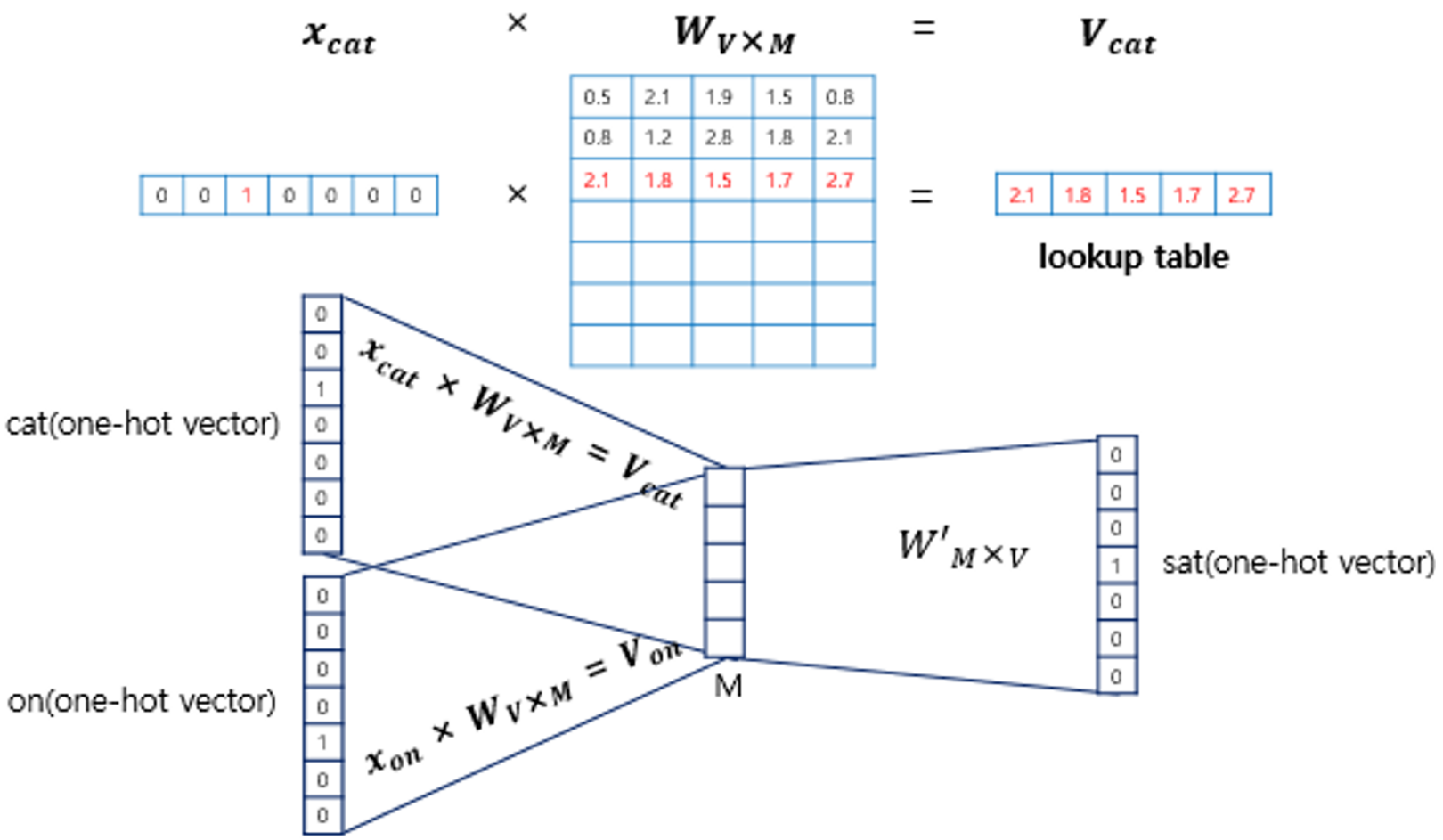
Word Vectors



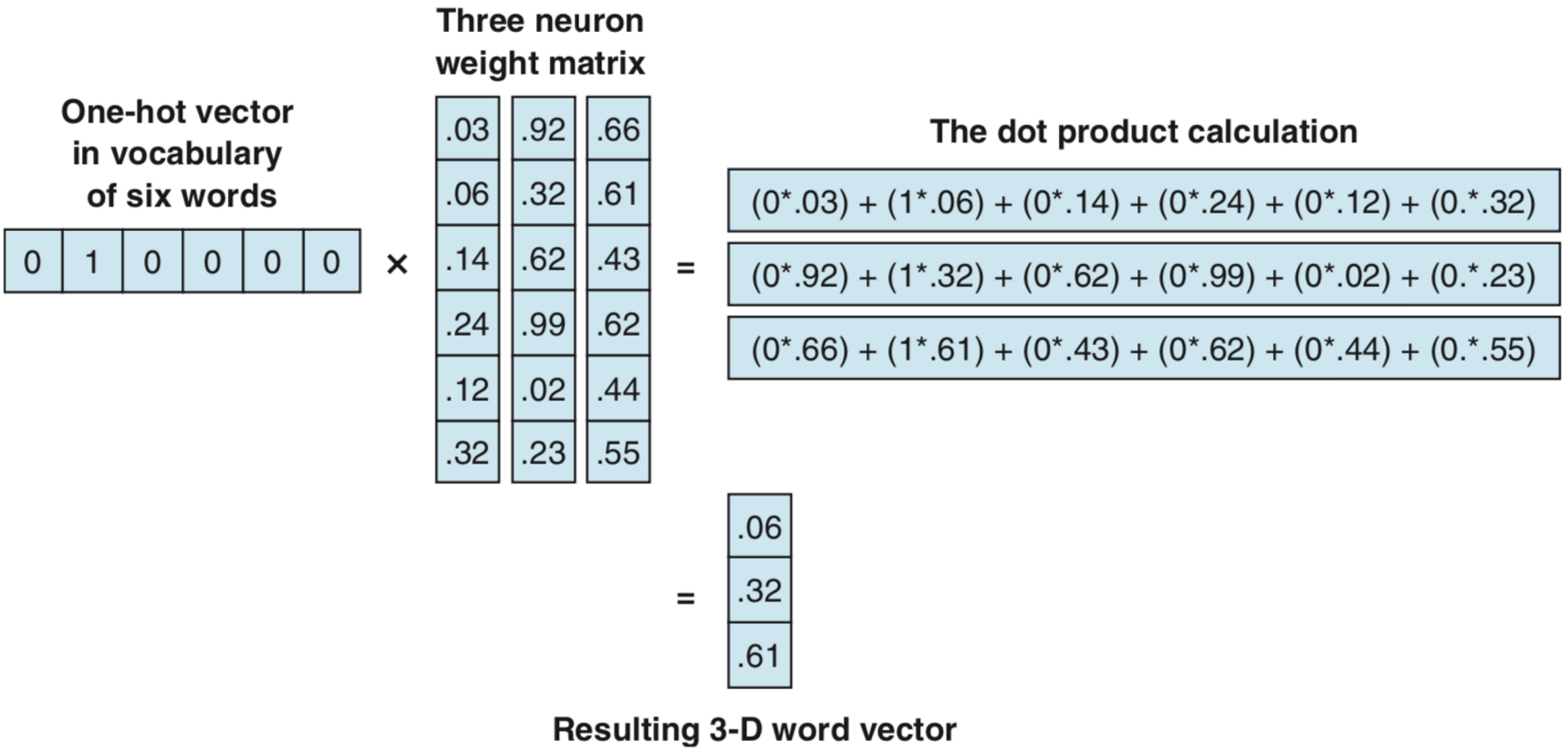
Word Vectors



Conversion of one-hot vector to word vector



Conversion of one-hot vector to word vector



Build Word2vec using Gensim

https://wikidocs.net/50739

tokenized data

sg = 0 means CBOW, 1 means Skip-gram

```
from gensim.models import Word2Vec

model = Word2Vec(sentences=result, size=100, window=5, min_count=5, workers=4, sg=0)

model_result = model.wv.most_similar("man")
print(model_result)

# [('woman', 0.842622697353363), ('guy', 0.8178728818893433), ('boy', 0.7774451375007629), ('lady', 0.7767927646636963), ('girl', 0.7583760023117065), ('gentleman', 0.7437191009521484), ('soldier', 0.7413754463195801), ('poet', 0.7060446739196777), ('kid', 0.6925194263458252), ('friend', 0.6572611331939697)]
```

Use Pre-trained Google Word2vec model

- Load the Word2vec model from Google
- Download the google vector file at https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit

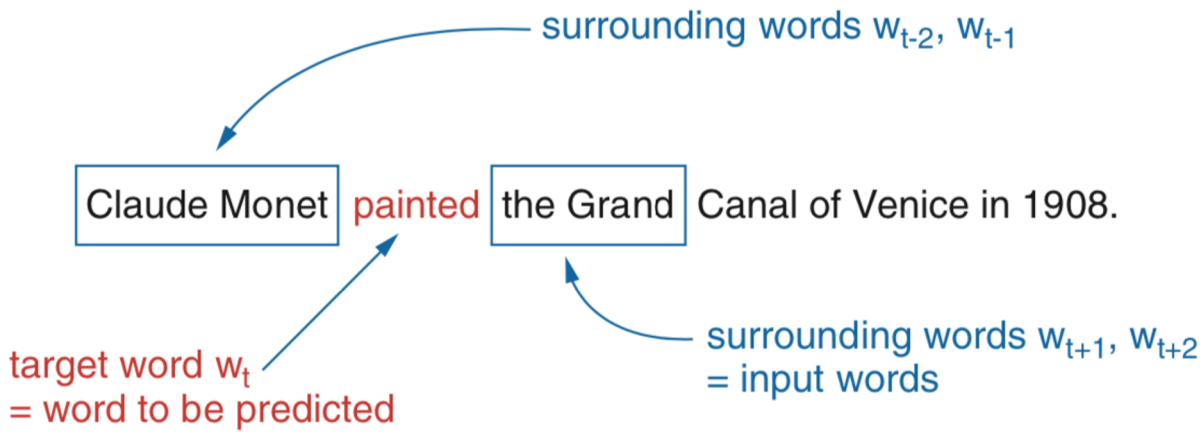
```
from gensim.models import KeyedVectors

filename = 'GoogleNews-vectors-negative300.bin'
# load the google word2vec model
model = KeyedVectors.load_word2vec_format(filename, binary=True)
# calculate: (king - man) + woman = ?
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result)

[Run]
[('queen', 0.7118192315101624)]
```

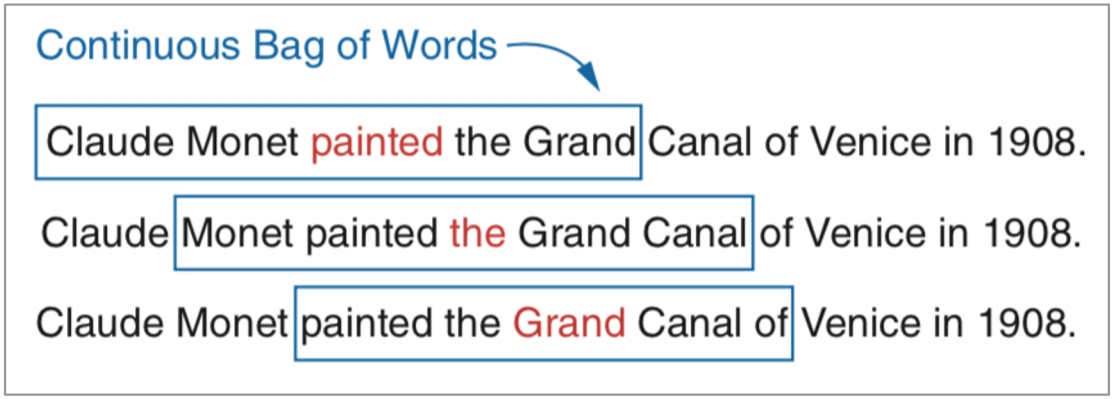
https://machinelearningmastery.com/develop-word-embeddings-python-gensim/

CBOW

Continuous Bag-of-Words (CBOW)

- Predict the center word based on the surrounding words



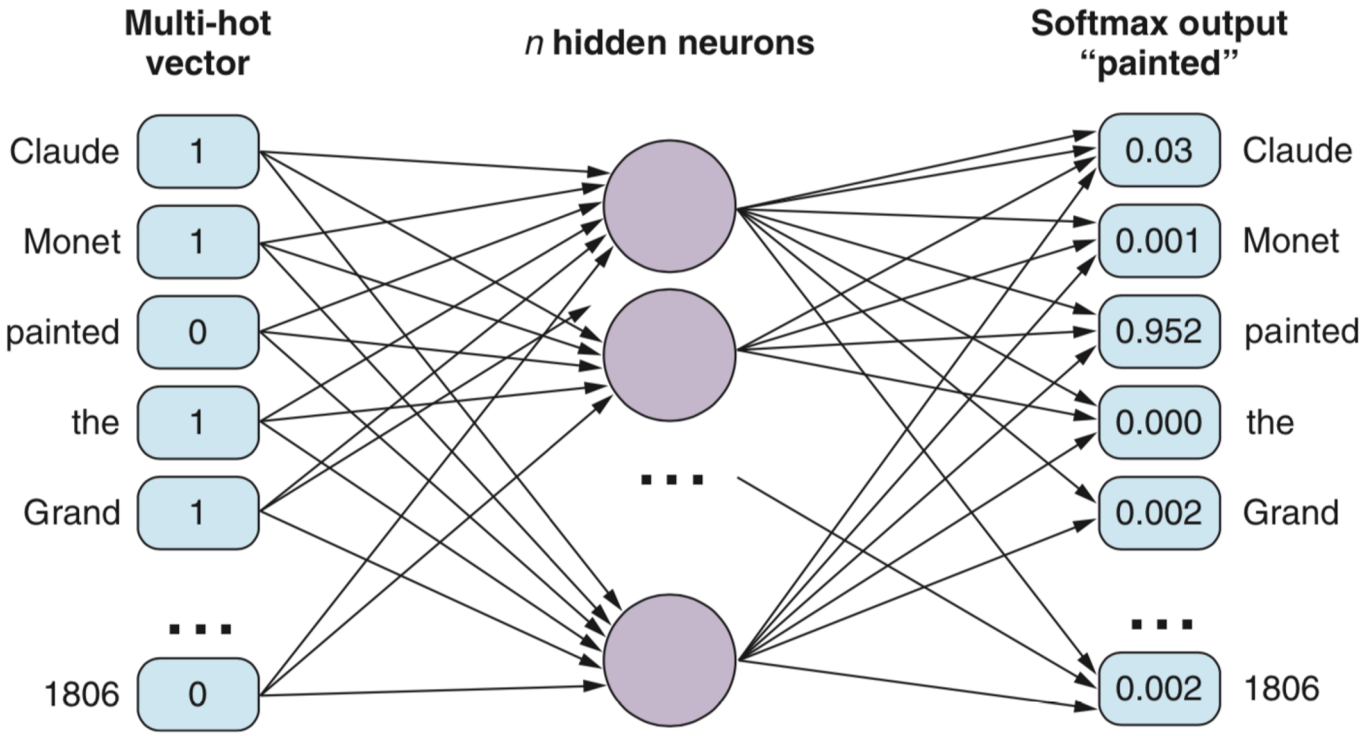


Building the training set

>>> sentence = "Claude Monet painted the Grand Canal of Venice in 1806."

Input word w_{t-2}	Input word w_{t-1}	Input word w_{t+1}	Input word w_{t+2}	Expected output w_t
		Monet	painted	Claude
	Claude	painted	the	Monet
Claude	Monet	the	Grand	painted
Monet	painted	Grand	Canal	the
painted	the	Canal	of	Grand
the	Grand	of	Venice	Canal
Grand	Canal	Venice	in	of
Canal	of	in	1908	Venice
of	Venice	1908		in
Venice	in			1908

CBOW network



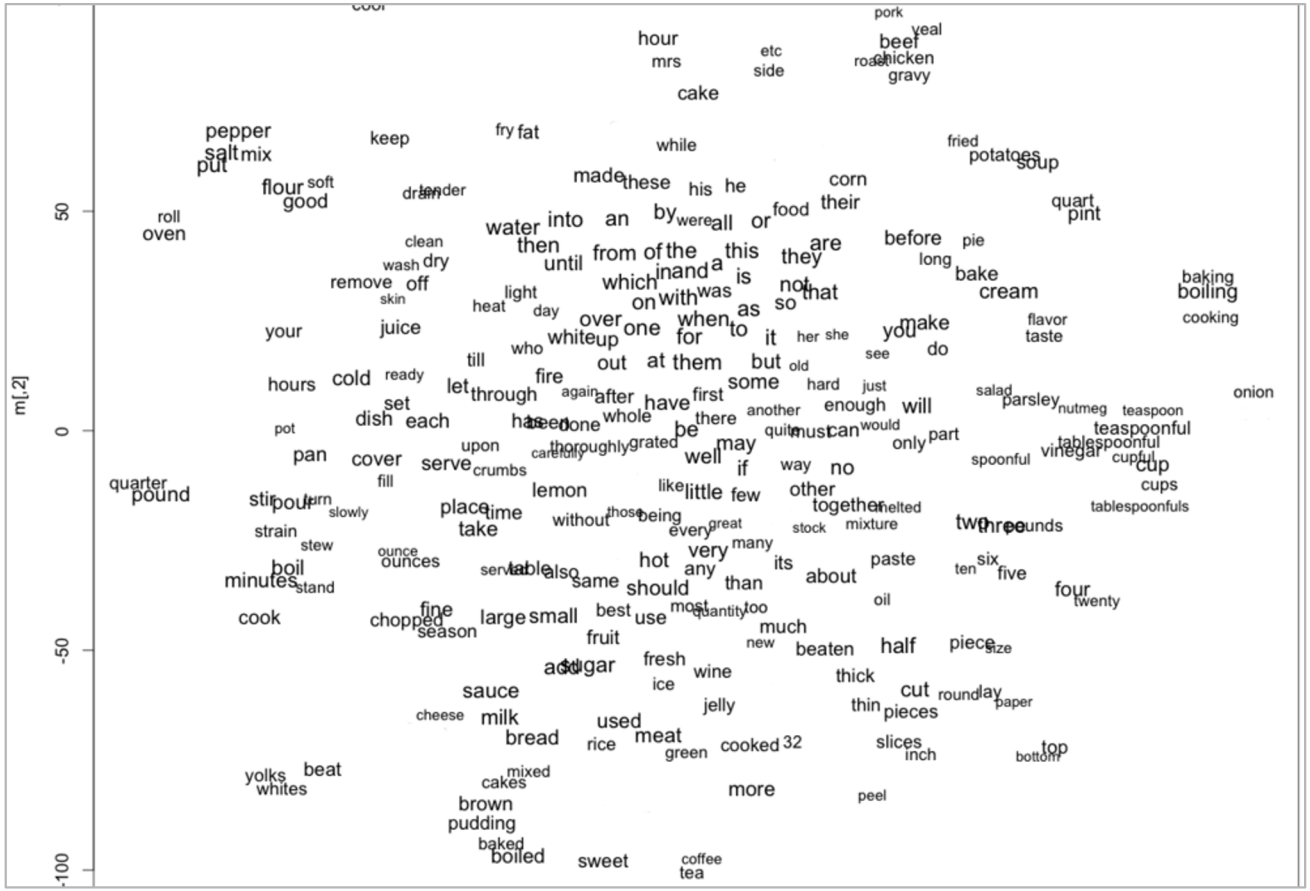
When to use which approach

- The skip-gram approach works well with small corpora and rare terms. With the skip-gram approach, you'll have more examples due to the network structure
- But the continuous bag-of-words approach shows higher accuracies for frequent words and is much faster to train

Reasoning with Word2vec

Vector Space Model

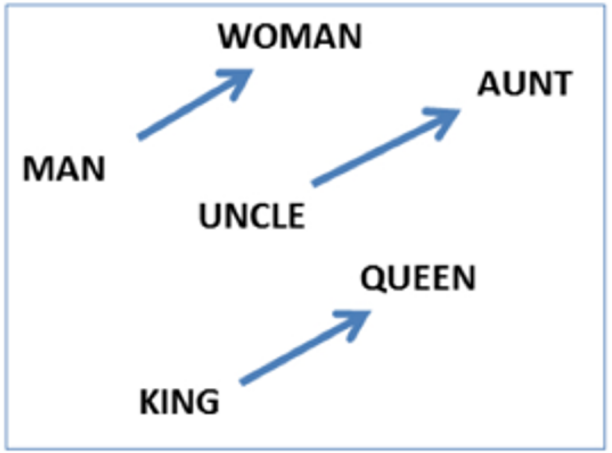
- Represents words in a continuous vector space
- Words that are semantically similar are mapped close to each other
- Words that appear in the same context share semantic meaning
- Have been used since 1990's for distributional semantics, LSA or LDA

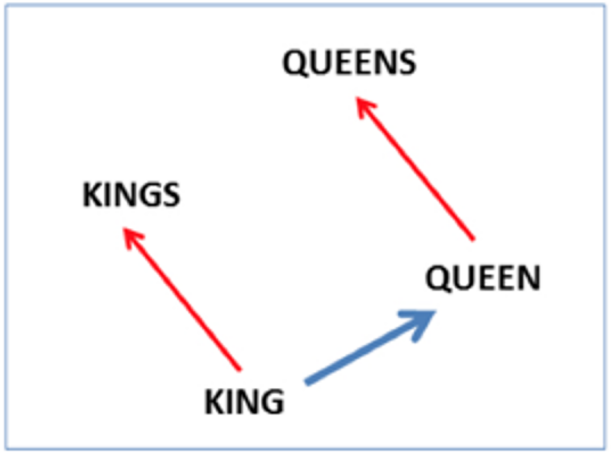


https://www.adityathakker.com/introduction-to-word2vec-how-it-works/

Analogy: Embeddings capture relational meaning!

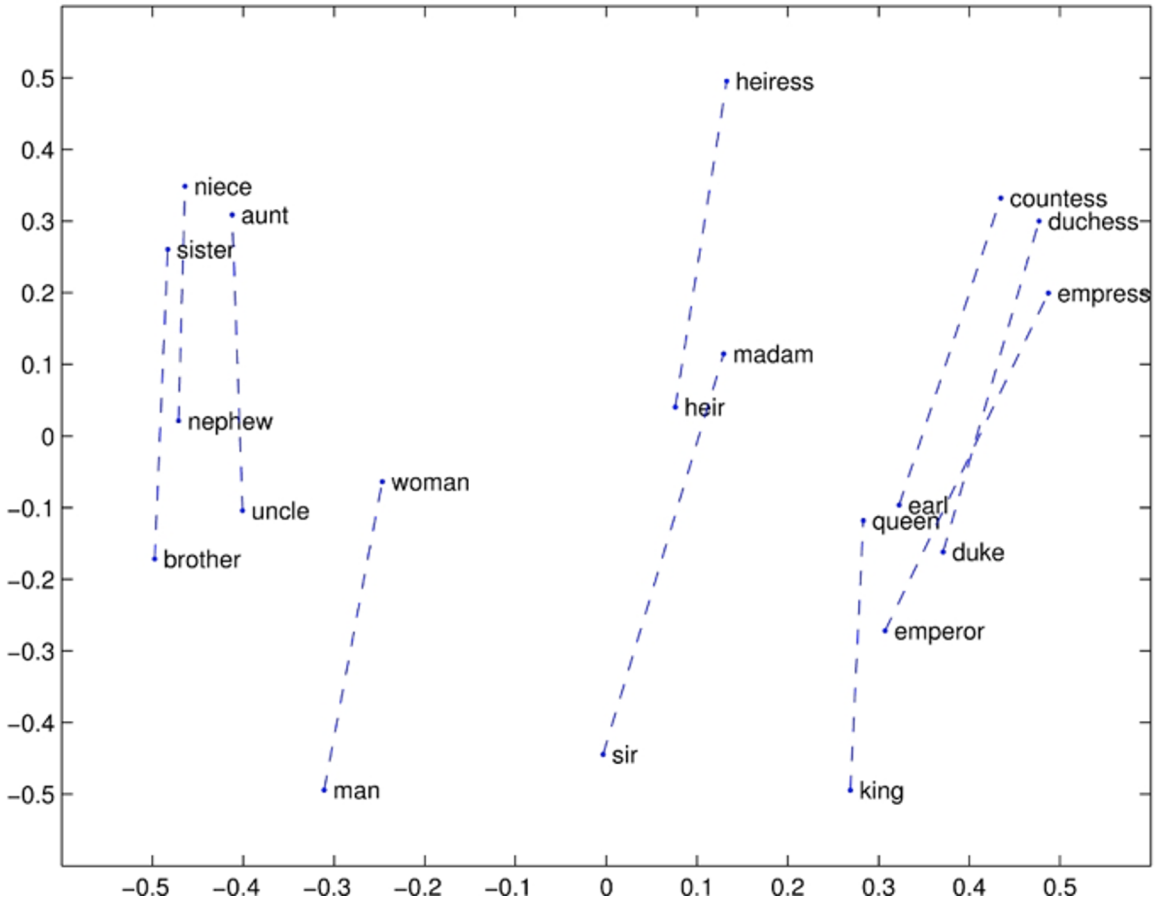
vector(‘king’) - vector(‘man’) + vector(‘woman’) ≈ vector(‘queen’)
vector(‘Paris’) - vector(‘France’) + vector(‘Italy’) ≈ vector(‘Rome’)



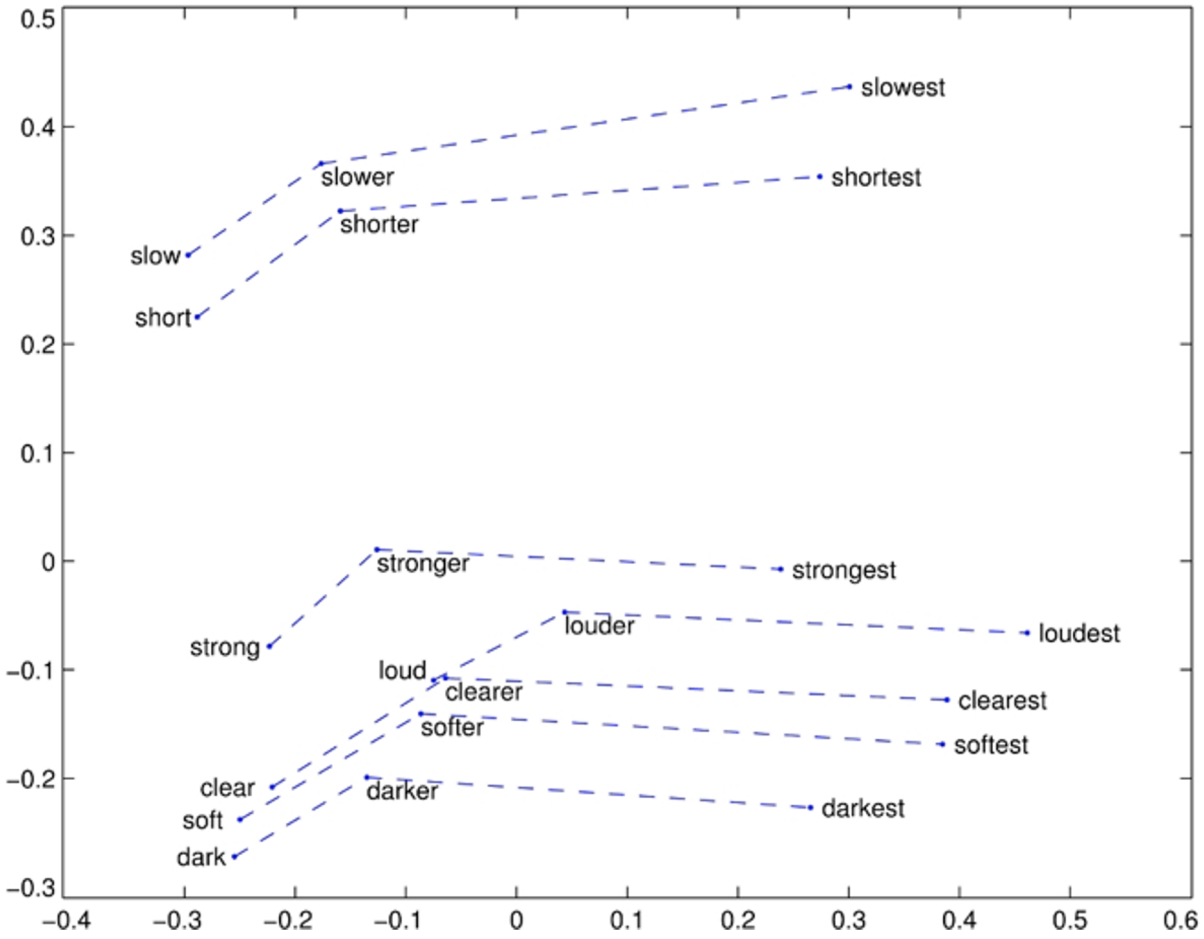


- Several factors influence the quality of the word vectors: amount and quality of the training data, size of the vectors, training algorithm
- The quality of the vectors is crucial for any application
- 한국어: http://w.elnn.kr/search/

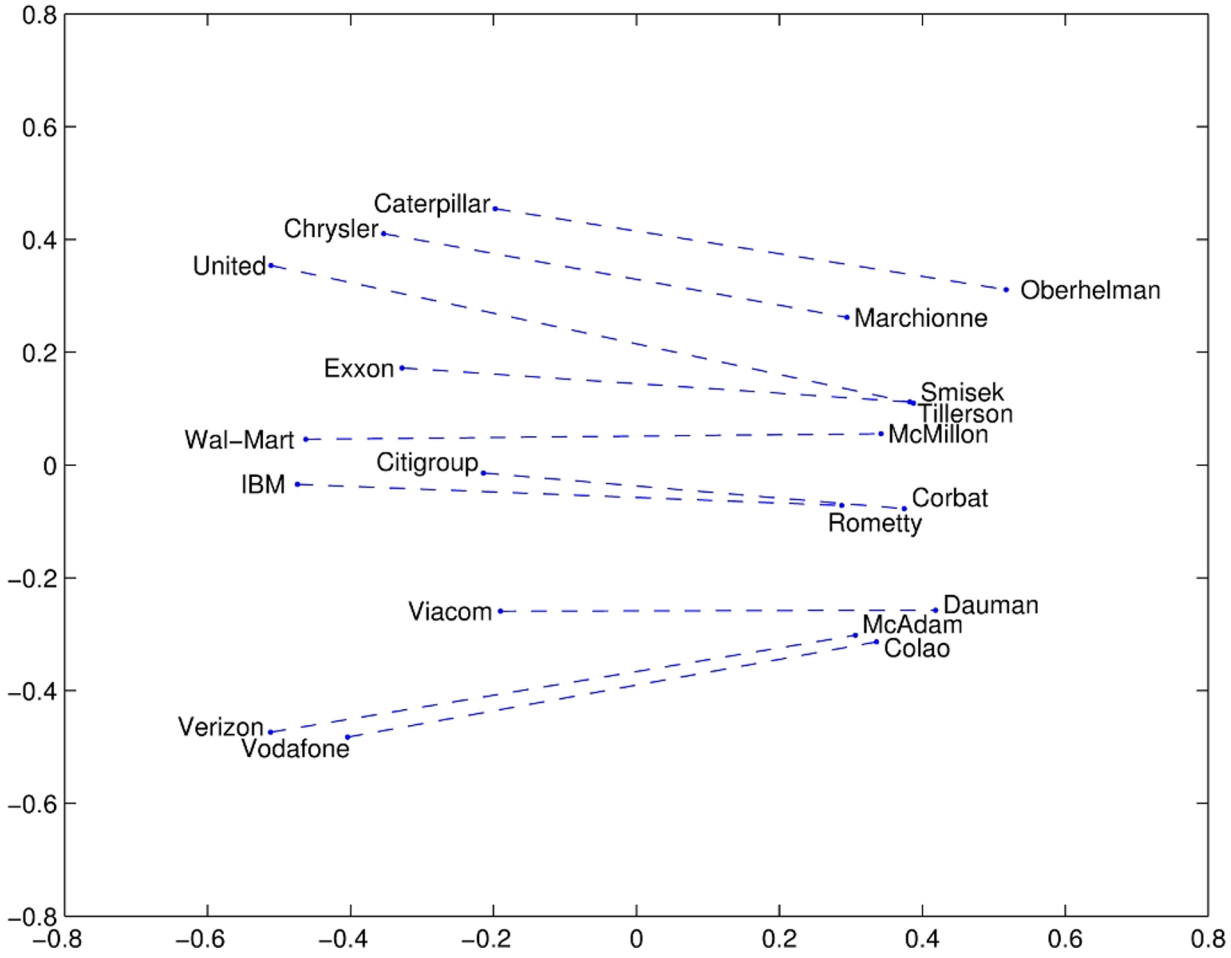
Analogy: gender



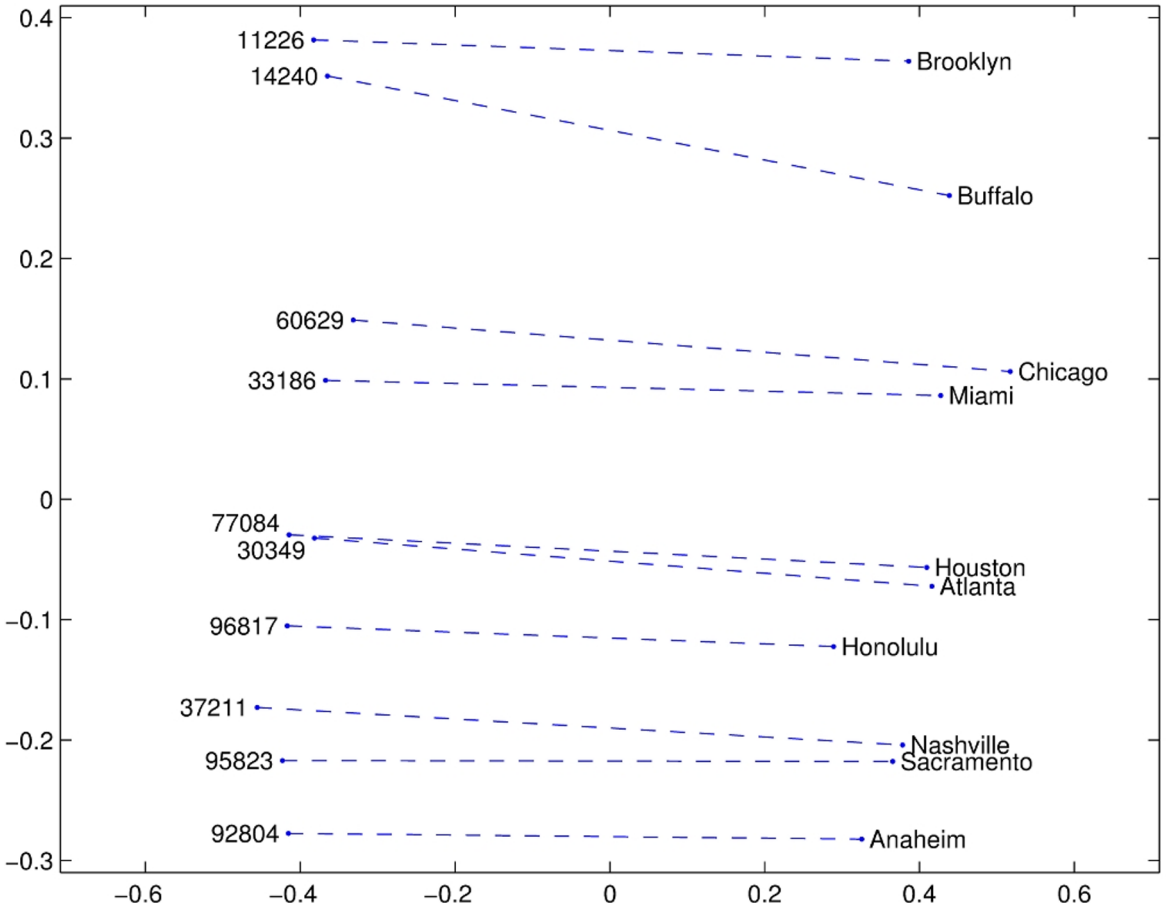
Analogy: comparative



Analogy: company-CEO



Analogy: zip code



Similarity in Word2vec

Words that appear in the same context share semantic meaning

	Most Similar Words for	
	Static Channel	Non-static Channel
bad	*good* *terrible* *horrible* *lousy*	*terrible* *horrible* *lousy* *stupid*
good	*great* *bad* *terrific* *decent*	*nice* *decent* *solid* *terrific*
n't	*os* *ca* *ireland* *wo*	*not* *never* *nothing* *neither*
!	*2,500* *entire* *jez* *changer*	*2,500* *lush* *beautiful* *terrific*
,	*decasia* *abysmally* *demise* *valiant*	*but* *dragon* *a* *and*

Convolutional Neural Networks for Sentence Classification, Yoon Kim, 2014 EMNLP

Doc2vec Train using Gensim

Reimplements word2vec, starting with the hierarchical softmax skip-gram model, which reports the best accuracy

Result

[TaggedDocument(words=['This', 'is', 'the', 'first', 'document', '.'], tags=['d0']), TaggedDocument(words=['This', 'document', 'is', 'the', 'second', 'document', '.'], tags=['d1']), TaggedDocument(words=['And', 'this', 'is', 'the', 'third', 'one', '.'], tags=['d2']), TaggedDocument(words=['Is', 'this', 'the', 'first', 'document', '?'], tags=['d3'])]

[-0.08623783 0.01574823 -0.07002052 -0.03521339 0.03276849]

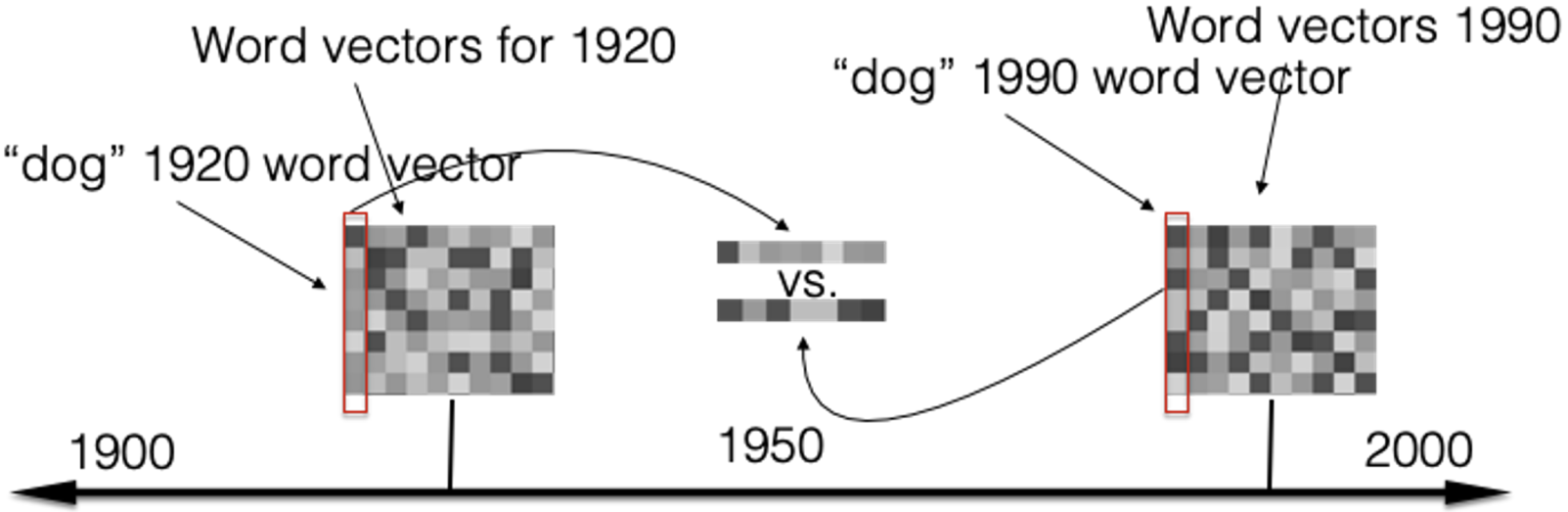
```
from gensim.models.doc2vec import TaggedDocument, Doc2Vec
from nltk.tokenize import word_tokenize
corpus = [     'This is the first document.',
                                    'This document is the second document.',
                                    'And this is the third one.',
                                    'Is this the first document?' ]

corpus = [list(word_tokenize(doc)) for doc in corpus]
corpus = [  TaggedDocument(words, ['d{}'.format(idx)])
                                    for idx, words in enumerate(corpus) ]
print(corpus)
model = Doc2Vec(corpus, vector_size=5, min_count=0)
print(model.docvecs[0])
```

“Distributed Representations of Sentences and Documents” (https://arxiv.org/pdf/ 1405.4053v2.pdf

More issues with Word2vec

Embeddings can help study word history!

Train embeddings on old books to study changes in word meaning!!



Will Hamilton

Project 300 dimensions down into 2

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

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 reflect cultural bias (Caliskan et al., 2017)

- Psychological findings on US participants:
  - African-American names are associated with unpleasant words (more than European-American names)
  - Male names associated more with math, female names with arts
  - Old people's names with unpleasant words, young people with pleasant words.
- Caliskan et al. replication with embeddings:
  - African-American names (Leroy, Shaniqua) had a higher GloVe cosine with unpleasant words (abuse, stink, ugly)
  - European American names (Brad, Greg, Courtney) had a higher cosine with pleasant words (love, peace, miracle)
- Embeddings reflect and replicate all sorts of pernicious biases

Other Embedding Models

GloVe (Global Vectors for Word Representation) Stanford 2014

- LSA는 DTM이나 TF-IDF 행렬과 같이 각 문서에서의 각 단어의 빈도수를 카운트 한 행렬이라는 전체적인 통계 정보를 입력으로 받아 차원을 축소(Truncated SVD)하여 잠재된 의미를 끌어내는 방법. 반면, Word2Vec는 실제값과 예측값에 대한 오차를 손실 함수를 통해 줄여나가며 학습하는 예측 기반의 방법
- LSA는 카운트 기반으로 코퍼스의 전체적인 통계 정보를 고려하기는 하지만, 왕:남자 = 여왕:? (정답은 여자)와 같은 단어 의미의 유추 작업(Analogy task)에는 성능이 떨어집니다. Word2Vec는 예측 기반으로 단어 간 유추 작업에는 LSA보다 뛰어나지만, 임베딩 벡터가 윈도우 크기 내에서만 주변 단어를 고려하기 때문에 코퍼스의 전체적인 통계 정보를 반영하지 못합니다
- GloVe는 이러한 기존 방법론들의 각각의 한계를 지적하며, LSA의 메커니즘이었던 카운트 기반의 방법과 Word2Vec의 메커니즘이었던 예측 기반의 방법론 두 가지를 모두 사용
- **임베딩 된 중심 단어와 주변 단어 벡터의 내적이 전체 코퍼스에서의 동시 등장 확률이 되도록 만드는 것**
- https://wikidocs.net/22885

GloVe

- https://nlp.stanford.edu/projects/glove/
- Trained on the non-zero entries of a global word-word co-occurrence matrix, which tabulates how frequently words co-occur with one another in a given corpus.
- ice co-occurs more frequently with solid than it does with gas, whereas steam co-occurs more frequently with gas than it does with solid. Both words co-occur with their shared property water frequently, and both co-occur with the unrelated word fashion infrequently.
- The training objective is to learn word vectors such that their dot product equals the logarithm of the words' probability of co-occurrence.

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

GloVe

- The training objective is to learn word vectors such that their dot product equals the logarithm of the words' probability of co-occurrence.

$$dot\ product(w_i\ \tilde{w}_k) \approx\ log\ P(k \mid i) = log\ P_{ik}$$

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

$$Loss\ function = \sum_{m,n=1}^{V} f(X_{mn})(w_m^T \tilde{w}_n + b_m + \tilde{b}_n - logX_{mn})^2$$

fastText (Facebook, 2017)

- fastText predicts the surrounding **n-character grams** rather than just the surrounding words
  - For example, the word “whisper” would generate the following 2- and 3-character grams:
  - wh, whi, hi, his, is, isp, sp, spe, pe, per, er
- trains a vector representation for every n-character gram, which includes words, misspelled words, partial words, and even single characters.
- It handles **rare words** and **OOV** much better than the original Word2vec approach
- Facebook published pretrained fastText models for 294 languages
- Enriching Word Vectors with Subword Information,” Bojanowski et al.: https://arxiv.org/pdf/1607.04606.pdf

```
>>> from gensim.models.fasttext import FastText
>>> ft_model = FastText.load_fasttext_format(\
...     model_file=MODEL_PATH)
>>> ft_model.most_similar('soccer')
```

Distributed representations (Word embeddings)

- Based on the distributional hypothesis: words with similar meanings tend to occur in similar context
- Words from the vocabulary are mapped to vectors of real numbers
- Embedding from a space with one dimension per word to a continuous vector space with a much lower dimension
- Used as the underlying input representation
- Word vectors capture the characteristics of the neighbors of a word
- Typically pre-trained
- Software for training and using word embeddings: Word2vec, GloVe, fastText, Elmo, BERT
- Advantage of distributional vectors
    - Capture similarity between words
    - Fast and efficient in computing core NLP tasks

Word vectors are biased!

- Word vectors learn word relationships based on the training corpus.
- If your corpus is about finance then your “bank” word vector will be mainly about businesses that hold deposits. If your corpus is about geology, then your “bank” word vector will be trained on associations with rivers and streams. And if you corpus is mostly about a matriarchal society with women bankers and men washing clothes in the river, then your word vectors would take on that gender bias.
- The following example shows the gender bias of a word model trained on Google News articles. If you calculate the distance between “man” and “nurse” and compare that to the distance between “woman” and “nurse,” you’ll be able to see the bias:
  - >>> word_model.distance('man', 'nurse')
  - 0.7453
  - >>> word_model.distance('woman', 'nurse')
  - 0.5586
- Identifying and compensating for biases like this is a challenge for any NLP practitioner that trains her models on documents written in a biased world.

Summary

Concepts or word senses

- Have a complex many-to-many association with words (multiple senses)
- Have relations with each other: Synonymy, Antonymy, Superordinate
- But are hard to define formally

Embeddings = vector models of meaning

- More fine-grained than just a string or index
- 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

Summary of NLP

- Tf-idf
- Classification: machine learning, deep learning
- word2vec