Word embeddings for IR

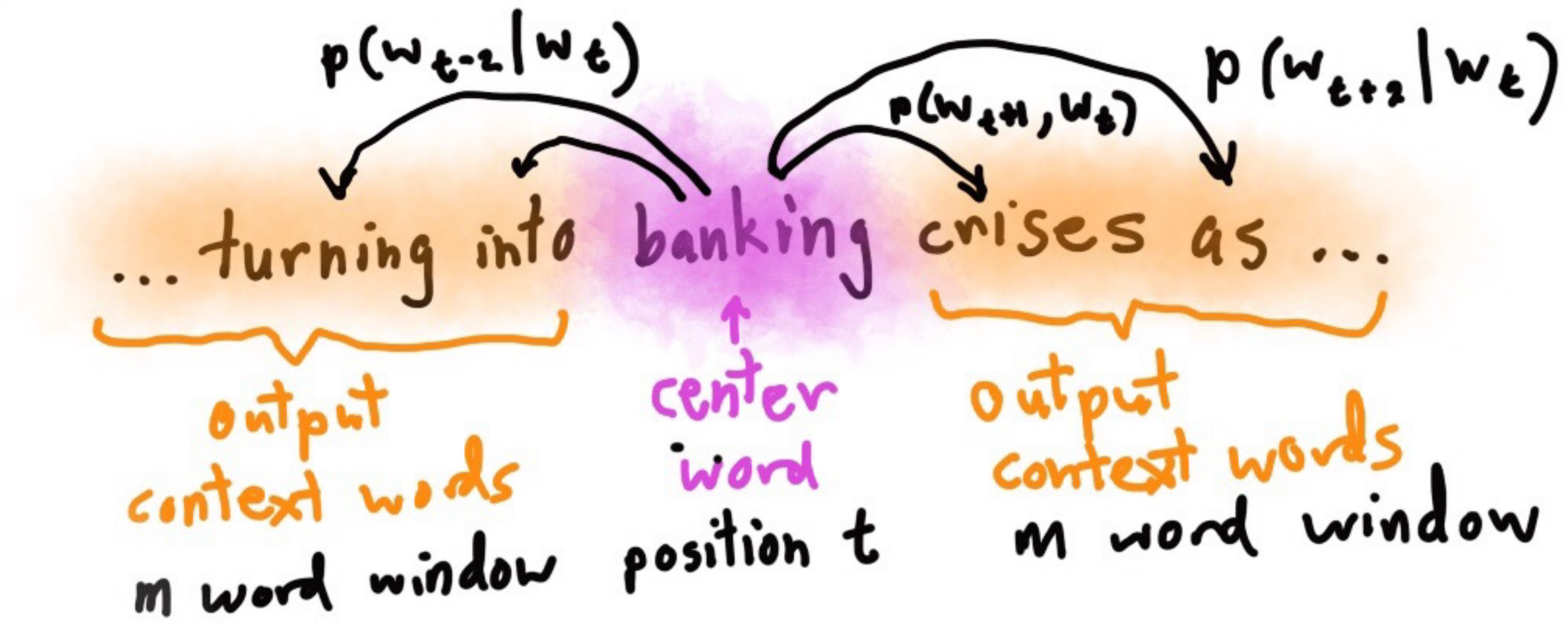
An alternative way to go beyond keyword matching

Word Embedding approach: main ideas

- Represent each word with a low-dimensional vector (like for LSI)
- **Word similarity = vector similarity (two words with similar vectors, are similar)**
- Key idea: learn to predict surrounding words in the context of every word, or, learn to predict a word from its surrounding context
- Faster and, wrt SVD, can easily incorporate a new sentence/document or add a new word to the vocabulary

$$\textit{linguistics} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

Key idea: semantic similarity among words depends on similarity among **word contexts** in documents



Co-occurrences are considered in a left-right context

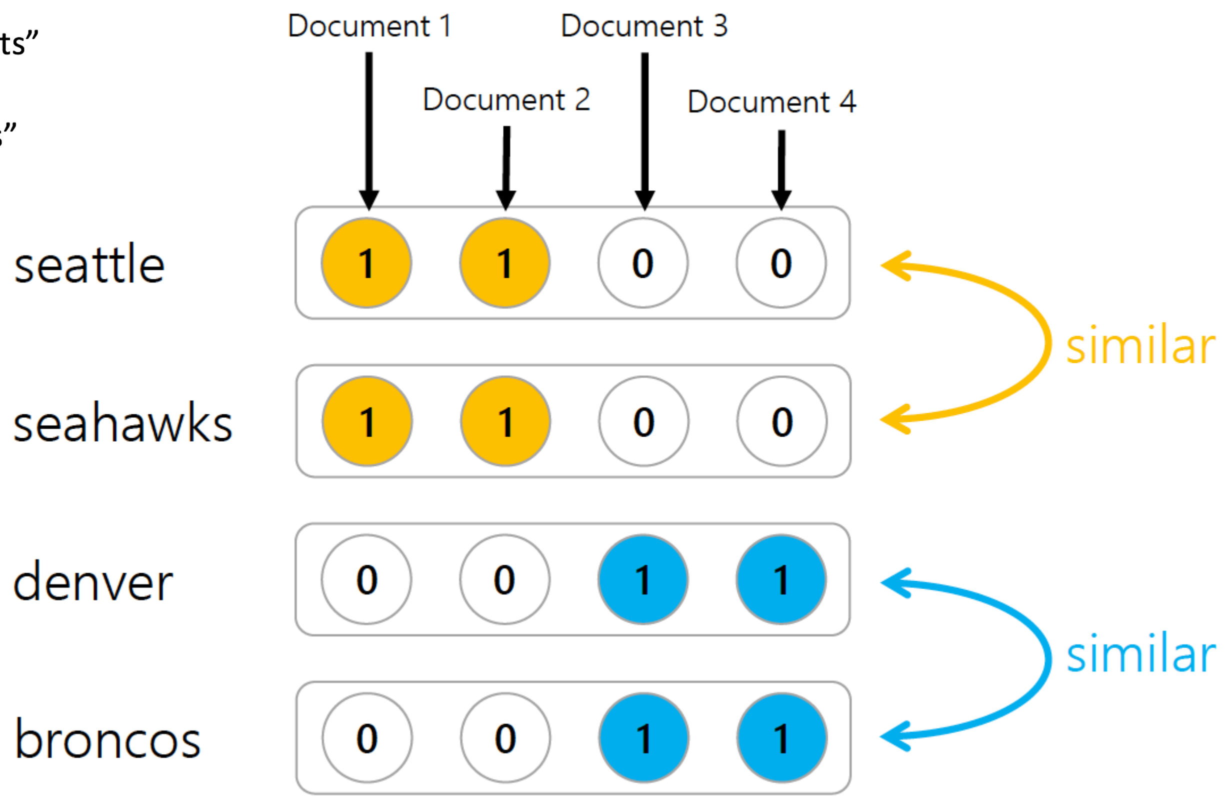
Let’s consider the following example…

- We have four (tiny) documents:

  Document 1 : “seattle seahawks jerseys”
  Document 2 : “seattle seahawks highlights”
  Document 3 : “denver broncos jerseys”
  Document 4 : “denver broncos highlights”

Basic difference with previous methods (e.g. LSI with SVD)

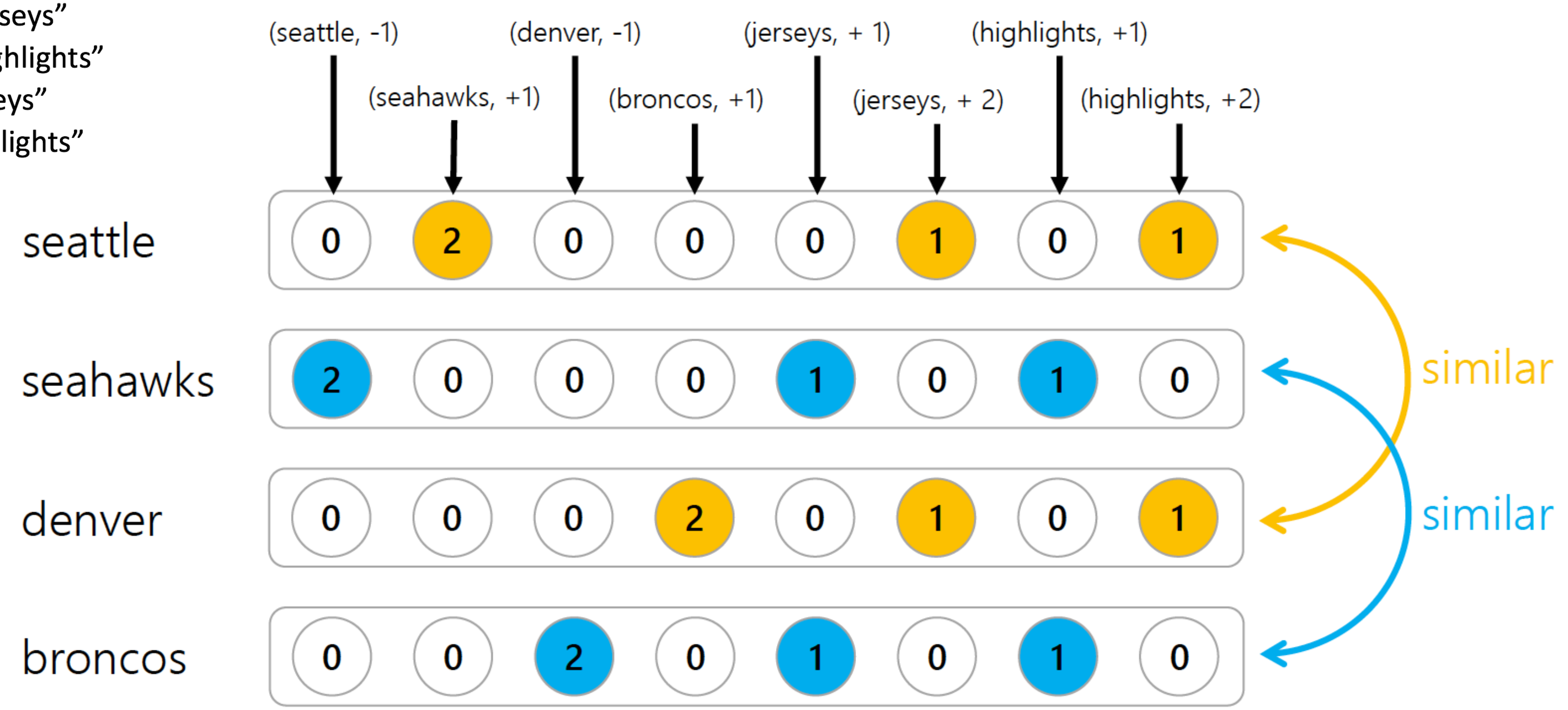
Document 1 : “seattle seahawks jerseys”
Document 2 : “seattle seahawks highlights”
Document 3 : “denver broncos jerseys”
Document 4 : “denver broncos highlights”



SVD would group words based on co-occurrences in documents

If we use context vectors:

Document 1 : “seattle seahawks jerseys”
Document 2 : “seattle seahawks highlights”
Document 3 : “denver broncos jerseys”
Document 4 : “denver broncos highlights”



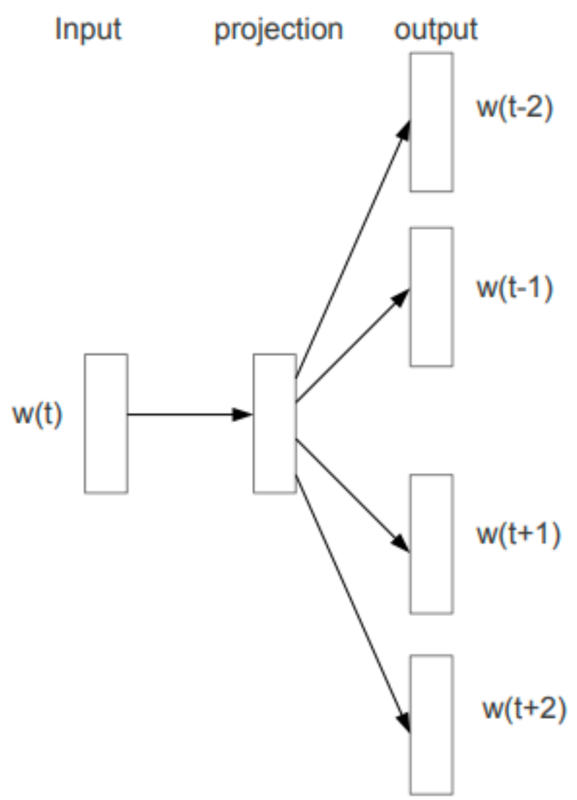
Every position in the vector is a tuple <word, distance from “center” word> an tells us how many times we see that word in the left(right) context of a word (e.g. *seahawks* is found 2 times in position +1 to the right of *seattle*) ➔ $p(w_{t\pm i}/w_t)$

Embeddings

- These “context vectors” are very high dimensional (thousands, or even millions) and very sparse.
- But there are techniques to learn **lower-dimensional dense vectors** for words using the same intuitions.
- These dense vectors are called **embeddings**.
- Rather than using matrix factorization techniques (such as SVD) we use *deep neural methods*.
- The objective is to represent each word with a dense vector, **such that similar words have similar vectors**
- **We can, as for LSI, consider the dimensions of this dense space as “concepts” or “semantic domains”**

Word Embeddings – Skip Grams Model

- Objective: Given a specific word in the middle of a sentence (the input word **w_t**) (e.g., *broncos*) look at the words *nearby* and pick one at random. The neural network should tell us the **probability for every word in our vocabulary of being the “nearby word” that we chose**.
- "nearby” means that there is a "window size" parameter *m* to the algorithm. A typical window size might be 5, meaning 5 words behind and 5 words ahead (10 in total).
- Our examples hereafter will be with smaller m (1 or 2)
- Note that the system **only predicts «nearby-ness» not the exact position**!



Training phase

- The original Skip-gram's objective is to **maximise** $\boldsymbol{P(w_c|w_t)}$ — the probability of w_c being predicted as w_t's context for all training pairs (e.g., *denver* being nearby *broncos*) . If we define the set of all training pairs as D we can formulate this objective as maximising the following expression:

$$\sum\nolimits_{(w_c, wt) \in D} \log(P(w_c|w_t))$$

- To calculate $P(w_c|w_t)$ we will need a means to quantify the «closeness» of the target-word w_t and the context-word w_c.
- In Skip-gram this closeness is computed using the dot product between the *input-embedding* of the target and the *output-embedding* of the context: $\boldsymbol{u_{ct} = e_t \cdot o_c}$ where e_t is a dense vector or «embedding» of w_t and o_c is the dense vector or « embedding» of w_c.
- The idea is then that words that occur in similar contexts (<u>but do not necessarily co-occur</u>) should have have similar input embeddings and words that **tend to co-occur** in same contexts should have similar input and output embeddings.

Training phase

The similarity function (dot product) is turned into a probability using the SOFTMAX function, so objective is to learn all u_c, u_k such as to maximize:

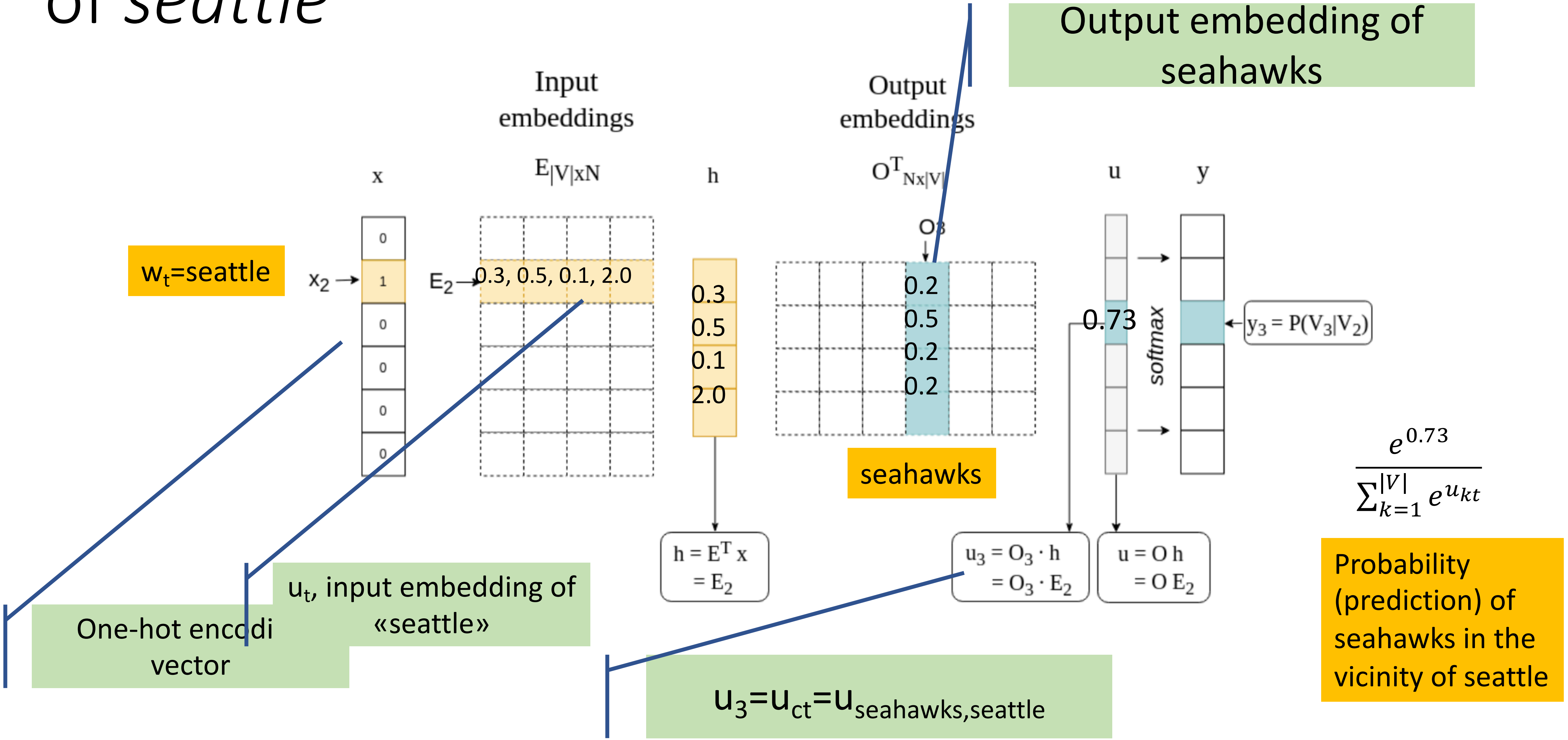
$$\sum_{(w_c,w_t)\in D} \log(P(w_c|w_t)) = \sum_{(w_c,w_t)\in D} \log\left(\frac{e^{u_{ct}}}{\sum_{k=1}^{|V|} e^{u_{kt}}}\right)$$

If |V| is the dimension of the vocabulary, and N is the dimension of embedding vector (an hyperparameter), then our task si to learn two matrixes, E |V|xN and O Nx|V| where V is the dimension of vocabulary and N the dimension of the dense embedding space (N<<V)

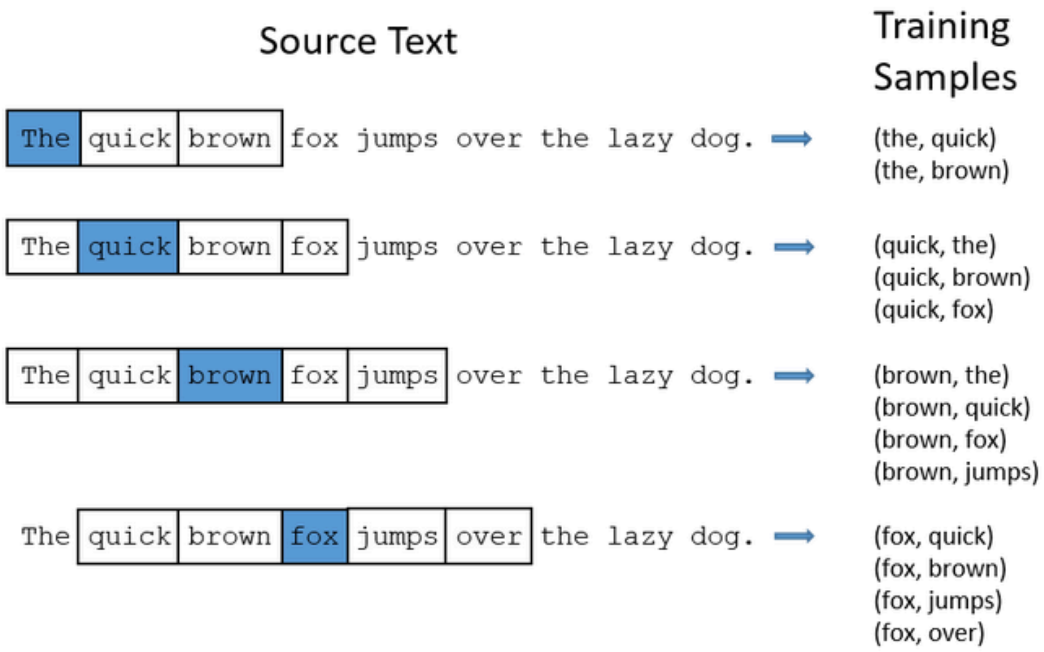
E is the input embegging matrix that projects a word onto a N-dimensional dense space.

O is the output embedding matrix where each row is the embedding of context words

Example: predicting *seahawks* in the vicinity of *seattle*



Matrixes E and O are initially unknown – how do we learn these numbers?



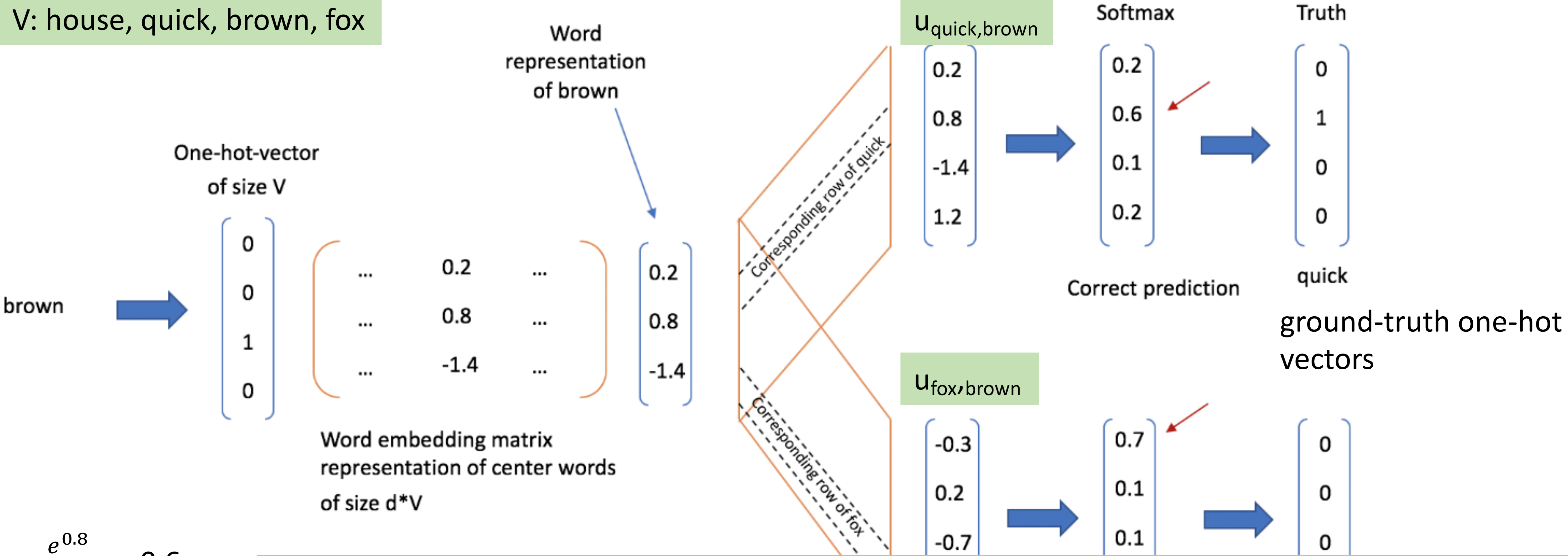
First, a training set of document is scanned and word pairs are extracted.

Each word pair represents a tuple
w_c, w_t
Tuples are evidences of contexts

In this example the blue word is the w_t and the other words are the w_c

Suppose we want to learn predicting P(quick/brown) and P(fox/brown)

V: house, quick, brown, fox



$$\frac{e^{0.8}}{\sum_{k=1}^{|V|} e^{u_{kt}}}=0.6$$

Probability of quick in the vicinity of brown

Error (softmax-true) is used to adjust values in E and O with the objective of «reducing» the gap between predicted and true value (backpropagation algorithm, see neural networks)

Negative sampling (1)

- The original softmax objective of Skip-gram is highly computationally expensive, as it requires scanning through the output-embeddings of *all* words in the vocabulary in order to calculate the sum from the denominator. And this must be repeated for any input pair, and for many epochs
- And typically such vocabularies contain hundreds of thousands of words. Because of this inefficiency most implementations use an alternative, negative-sampling objective, which rephrases the problem as a set of independent binary classification tasks.

Negative sampling (2)

- Instead of defining the complete probability distribution over words, the model learns to differentiate between the **correct** training pairs retrieved from the corpus and a set of incorrect, randomly generated pairs.
- For each correct pair the model draws *m* negative ones — with *m* being a hyperparameter.
- All negative samples have the same w_t (e.g., *seattle*) as the original training word, but their context words w_c are drawn at random from an arbitrary noisy distribution.
- For the training pair (*seattle, seahawks*) the incorrect ones could be (*seattle, logarithm*) or (*seattle, monkey*). The new objective of the model is to maximise the probability of the correct samples coming from the corpus and minimise the corpus probability for the negative samples, such as (*seattle, logarithm*) .

Negative sampling (2)

- Let's set D to be the set of all correct pairs and D' to denote a set of all negatively sampled |D| × *m* pairs. We will also define P(C = 1|w_t, w_c) to be the probability of (w_t, w_c) being a correct pair, originating from the corpus.
- Given this setting, the negative-sampling objective is defined as maximising:

$$\sum_{w_t w_c \in D} \log P(C = 1|w_t, w_c) - \sum_{w_t w_c \in D'} \log (1 - P(C = 1|w_t, w_c))$$

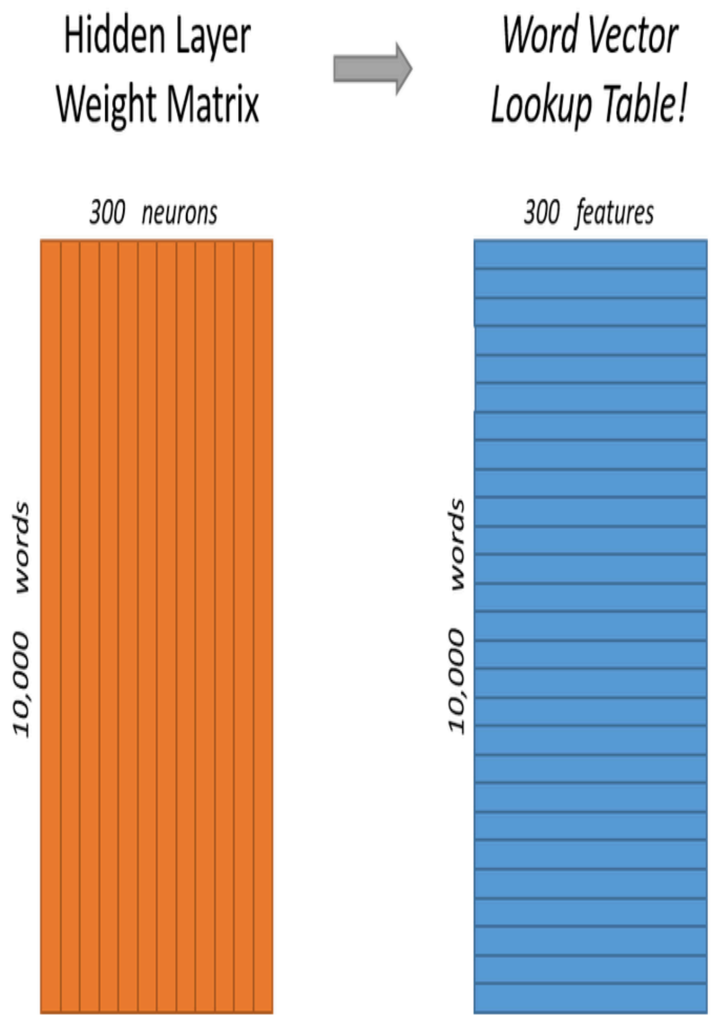
where:

$$P(C = 1|w_t, w_c) = \sigma(u_c) = \frac{1}{1+e^{-u_c}}$$ (u_c output embedding of w_c)

Word embedding hyperparameters

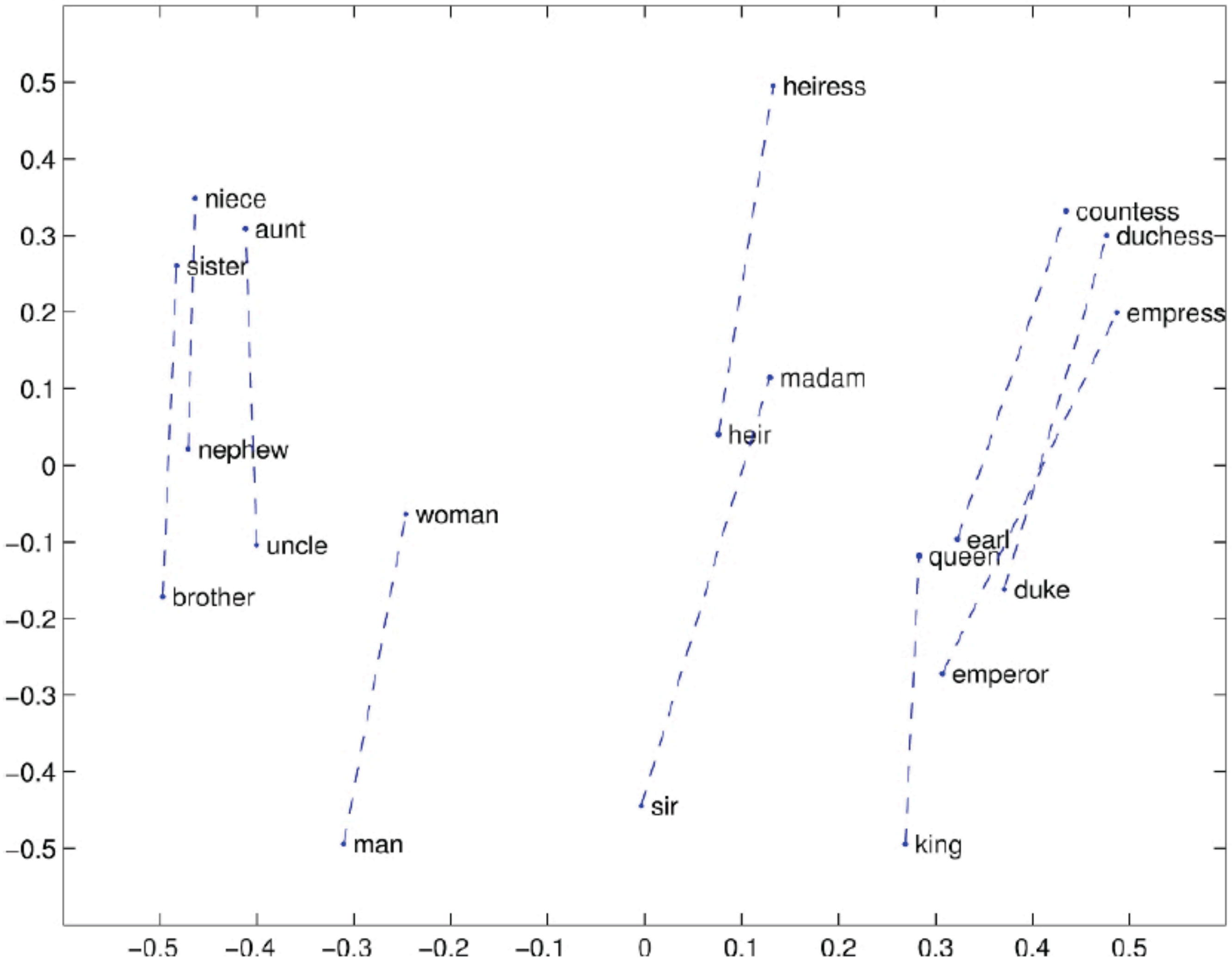
- *|V| dimension of vocabulary*
- *N dimension of embeddibg vectors*
- *m dimension of context for extracting word pairs*

Matrixes D and O

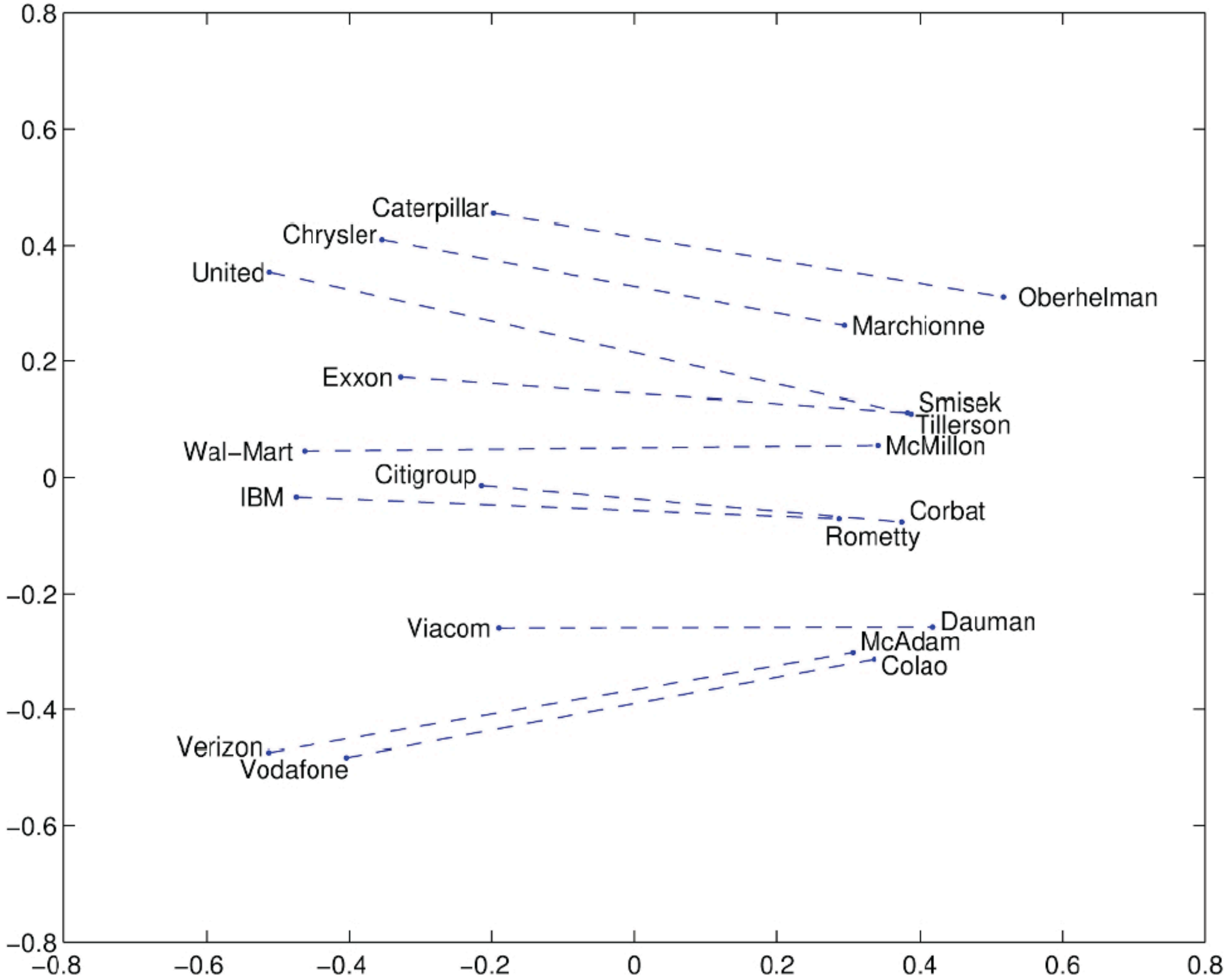


- **Several implementations: word2vect and Glove among the most well known**
- Google word2vect original paper has N=300 and |V|=10,000
- The matrix D is what we are really interested in: **the embedding matrix**.
- It has the property that **words with similar embedding vectors are similar.**

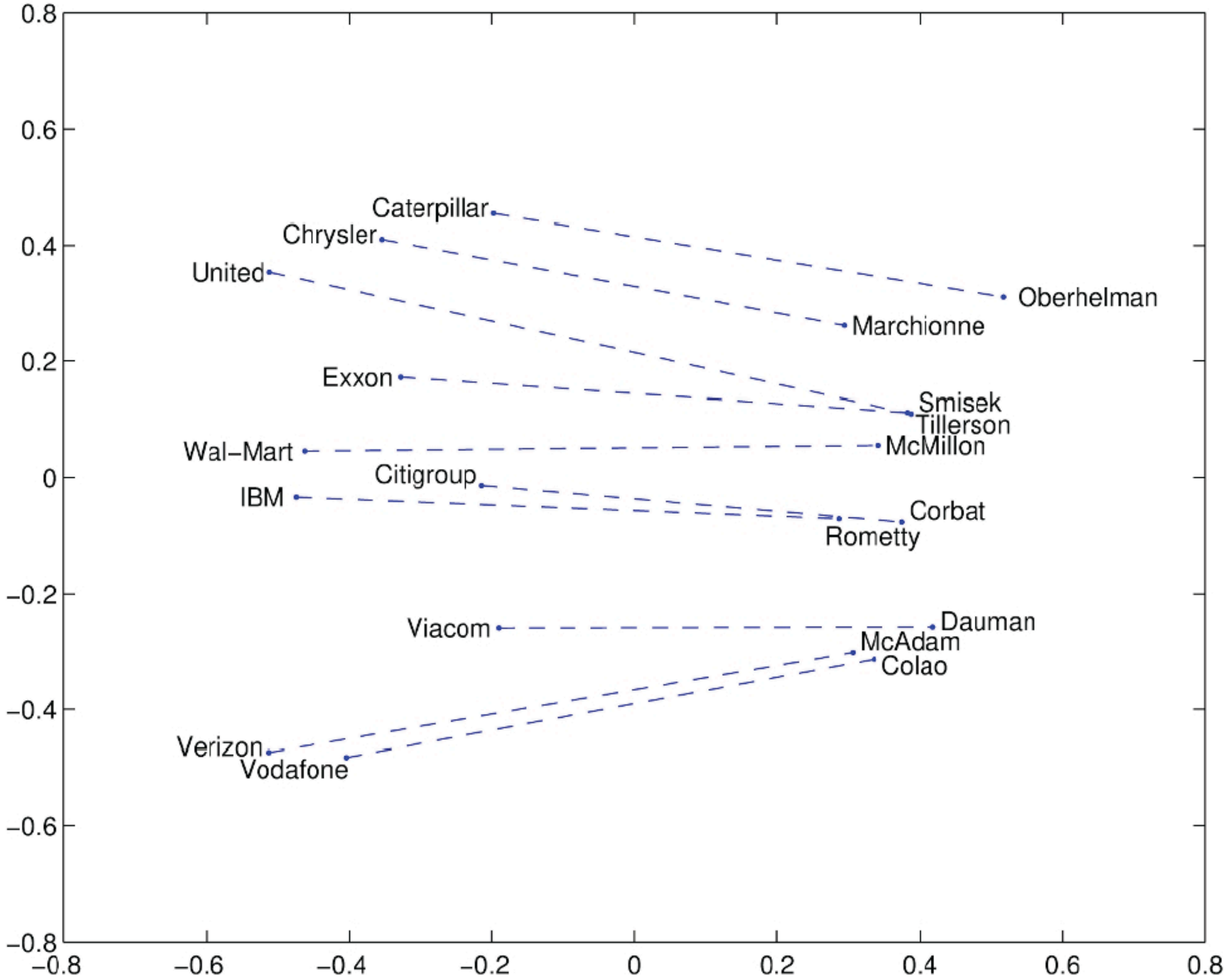
GloVe Visualizations



Glove Visualizations: Company - CEO



Glove Visualizations: Company - CEO



Applications of Word Embeddings to IR

- Word embeddings are the “hot new” technology for document ranking
- Lots of applications wherever knowing word contexts or similarity helps predicting users’ interests:
    - **Synonym handling in search**
    - **Query expansion**
    - **Document “aboutness”**
    - Machine translation
    - Sentiment analysis
    - ….

Applications of Word Embeddings to IR: Google RankBrain

- Google's RankBrain – almost nothing is publicly known
  - Bloomberg article by Jack Clark (Oct 26, 2015):
  - http://www.bloomberg.com/news/articles/2015-10-26/google-turning-itslucrative-web-search-over-to-ai-machines
  - A result re-ranking system

Weakness of Word Embedding

- Very vulnerable, and not a robust concept
- Can take a long time to train (despite negative sampling and other "tricks")
- Non-uniform results
- Hard to understand and visualize
- Emerging technique, yet not sufficiently robust and well understood
- Important: it learns the **same embedding** for different senses e.g. "*bank account*" and "*bank of the river.*"

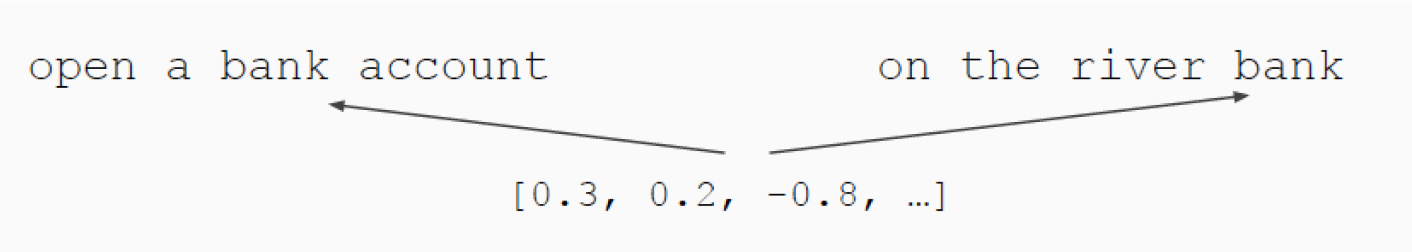
New trend: bidirectional encoders

- BERT Bidirectional Encoder Representations from Transformers https://arxiv.org/pdf/1810.04805.pdf

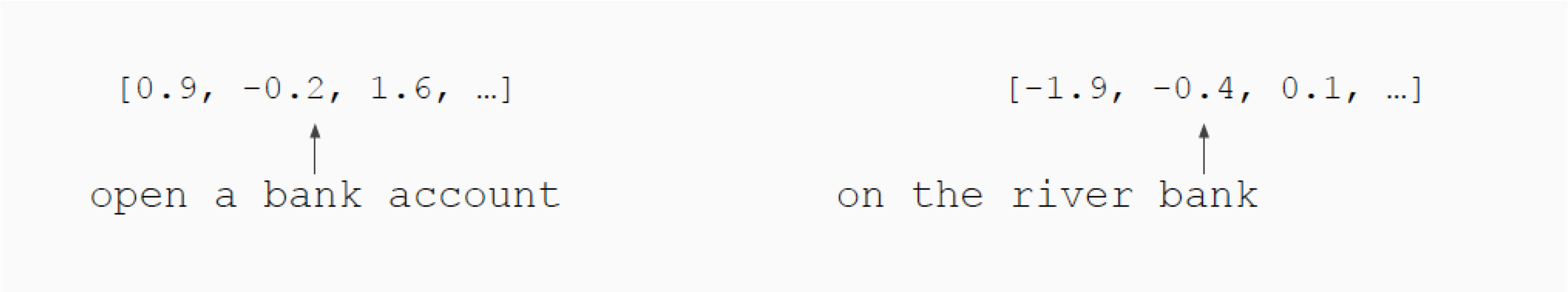
- BERT is Google latest search algorithm based on deep neural networks
- It has been proved to improve:
  - Named entity identification
  - Next sentence prediction (conversational analysis)
  - Co-reference (pronouns)
  - Question answering
  - Summarization
  - Ambiguity

Basic idea

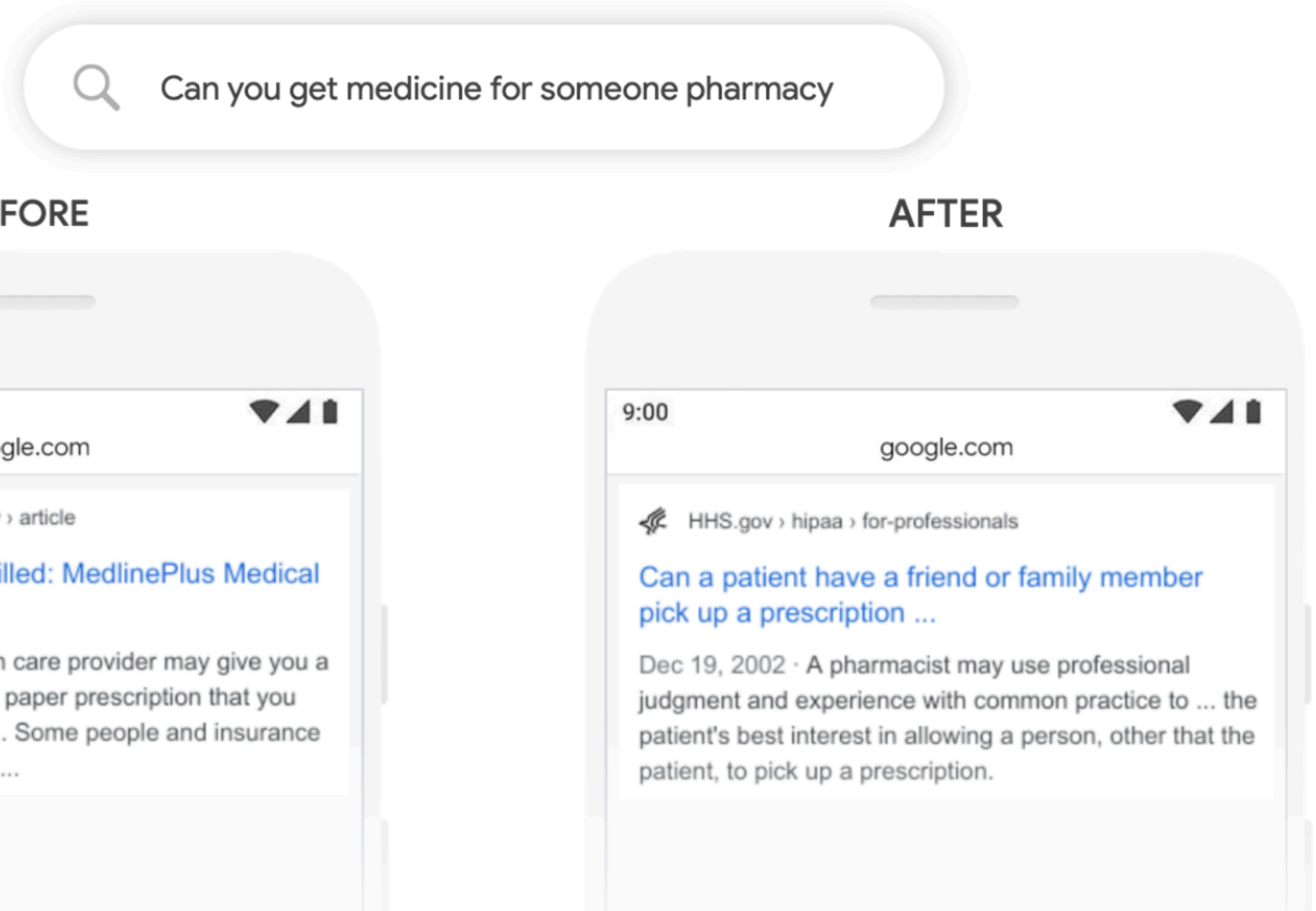
- Word embeddings are context independent



- BERT is context-aware (train on contextual representations)



Better understanding of language nuances



+1 on final grade for presenting BERT next week (20 minutes max, max 2 presentations)

lots of BERT-based papers since mid-2019, just read the original paper and the necessary ML background (LSTM, Transformers)