Word2vec

LING 570

Fei Xia

Many methods for learning word embeddings

- Static word embedding:
  - word2vec (2013)
  - GloVe (2014)
  - fastText (2016)

- Contextualized word embedding:
  - BERT, etc.

word2vec papers

1) Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In Proceedings of Workshop at ICLR, 2013.

2) Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of NIPS, 2013.

3) Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic Regularities in Continuous Space Word Representations. In Proceedings of NAACL HLT, 2013.

- A good blog that explains the algorithm:
  http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

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

Many implementations

- In C:
  - https://code.google.com/p/word2vec/ (the original one)
  - https://github.com/dav/word2vec (on patas dropbox/17-18/570/hw11)

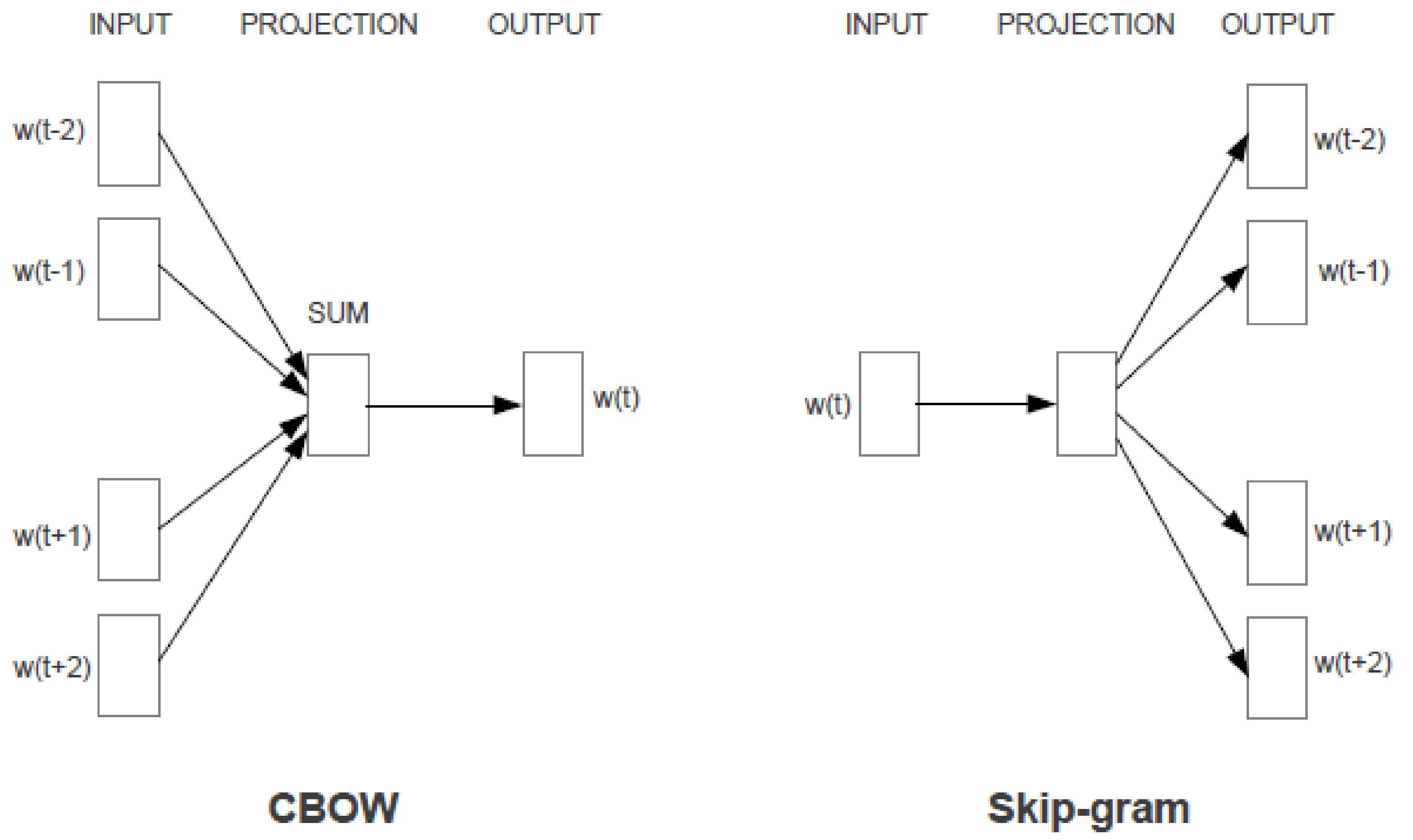
- In Java:
  - http://deeplearning4j.org/word2vec.html
  - https://github.com/medallia/Word2VecJava

- In python:
  - https://rare-technologies.com/deep-learning-with-word2vec-and-gensim/ (a python library)

Intuition behind word2vec

- To train a simple neural network with a single hidden layer.
  - But we are not going to use the NN for the task we trained it on.
  - All we care about are the weights of the hidden layers.
  - Training: use stochastic gradient descent and backpropagation
- Two models:
  - Continuous bag-of-word (CBOW): predict current word using the neighbor words
  - Continuous skip-gram model: predict neighbor words using the current word

Two word2vec models (Mikolov et al., ICLR 2013)

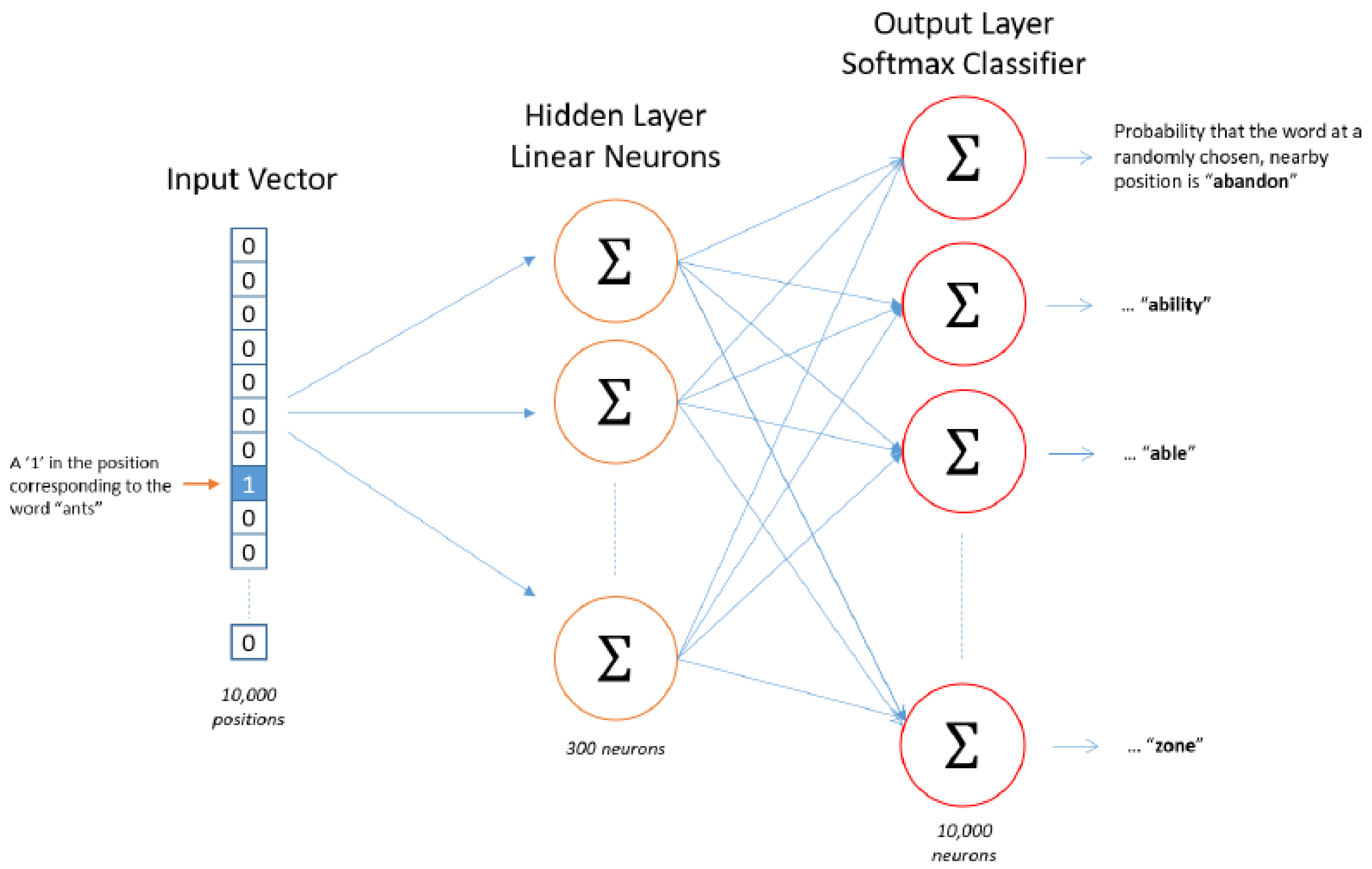


Each word is fed into a model as a one-hot vector

The fake task for skip-gram

- Task: Given a specific word w1 (aka the input word) in the middle of a sentence, pick a nearby word at random, what's the probability that w2 is chosen?
  - The input is a word
  - The output is a probability distribution
  - "nearby": window size

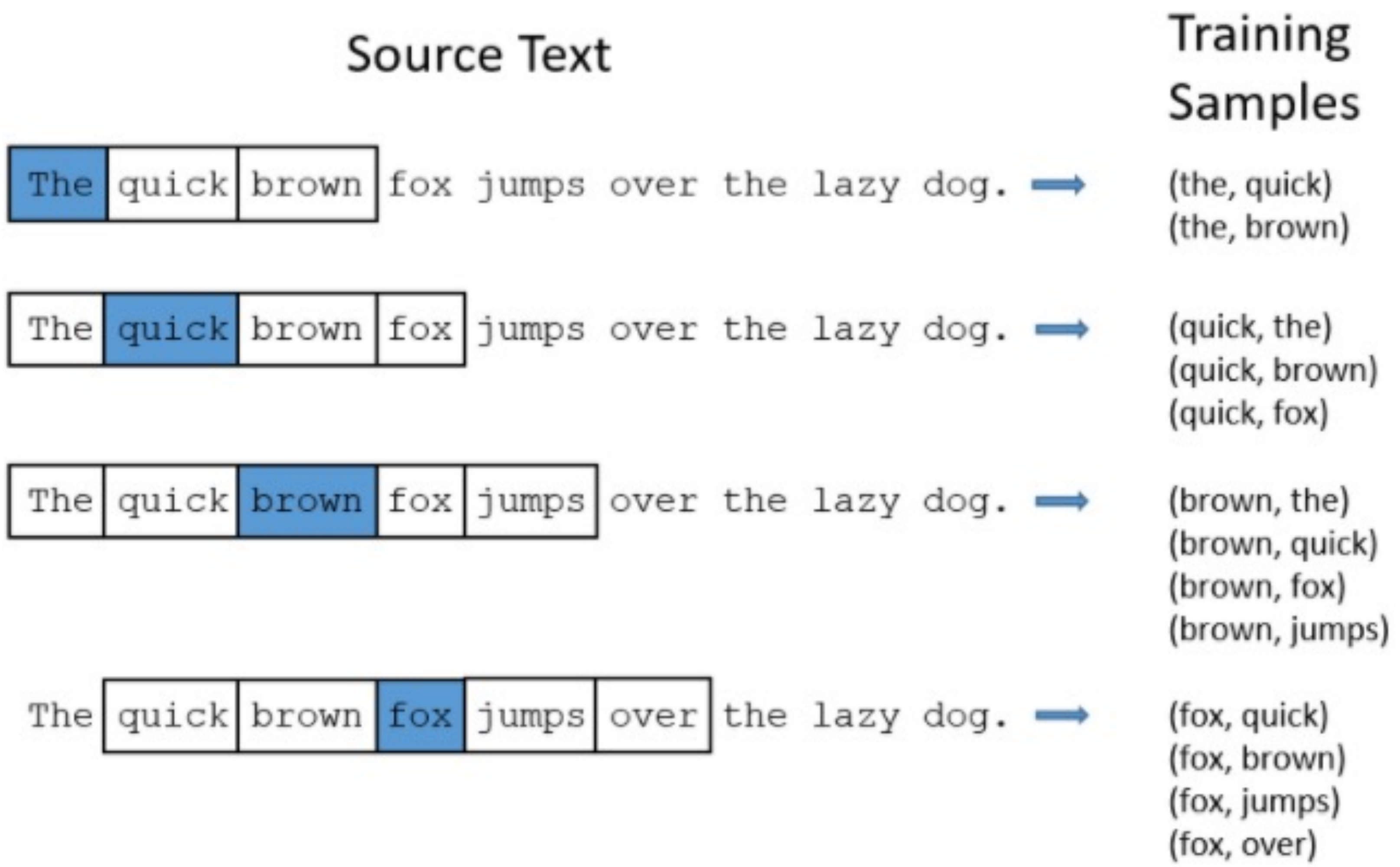
- Training data: a large corpus of text (e.g., 100B words)

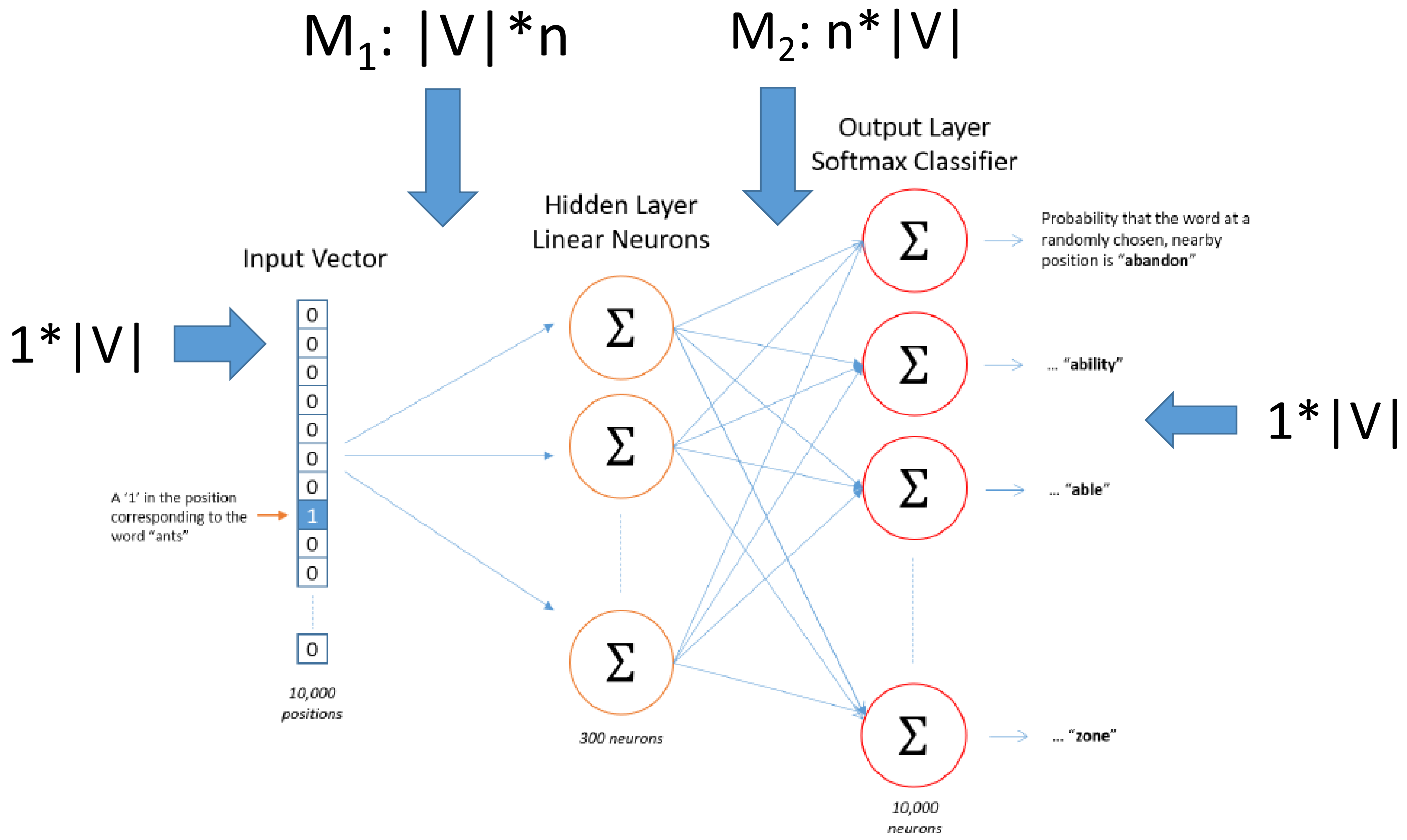

Input Vector
Hidden Layer
Linear Neurons
Output Layer
Softmax Classifier
0
0
0
0
0
0
0
1
0
0
0
A '1' in the position corresponding to the word "ants"
10,000 positions
Σ
300 neurons
10,000 neurons
Probability that the word at a randomly chosen, nearby position is "abandon"
... "ability"
... "able"
... "zone"

The NN

- Input layer:
  - |V| neurons: one for each word in the vocabulary
  - Each input word is represented as a one-hot vector: one dimension is 1, the rest are all zeros.
- Hidden layer:
  - # of neurons = # of dimensions in word embeddings
  - Use linear neuron (“no activation function”): i.e., y = z
- Output layer:
  - |V| neurons: the output of each neuron is [0,1]
  - Output neurons use softmax: so the output vector is a probability distribution

Training examples




M_1: |V|*n
M_2: n*|V|
Output Layer
Softmax Classifier
Hidden Layer
Linear Neurons
Input Vector
1*|V|
0
0
0
0
0
0
0
1
0
0
0
A '1' in the position corresponding to the word "ants"
10,000 positions
300 neurons
Probability that the word at a randomly chosen, nearby position is "abandon"
... "ability"
... "able"
... "zone"
1*|V|
10,000 neurons

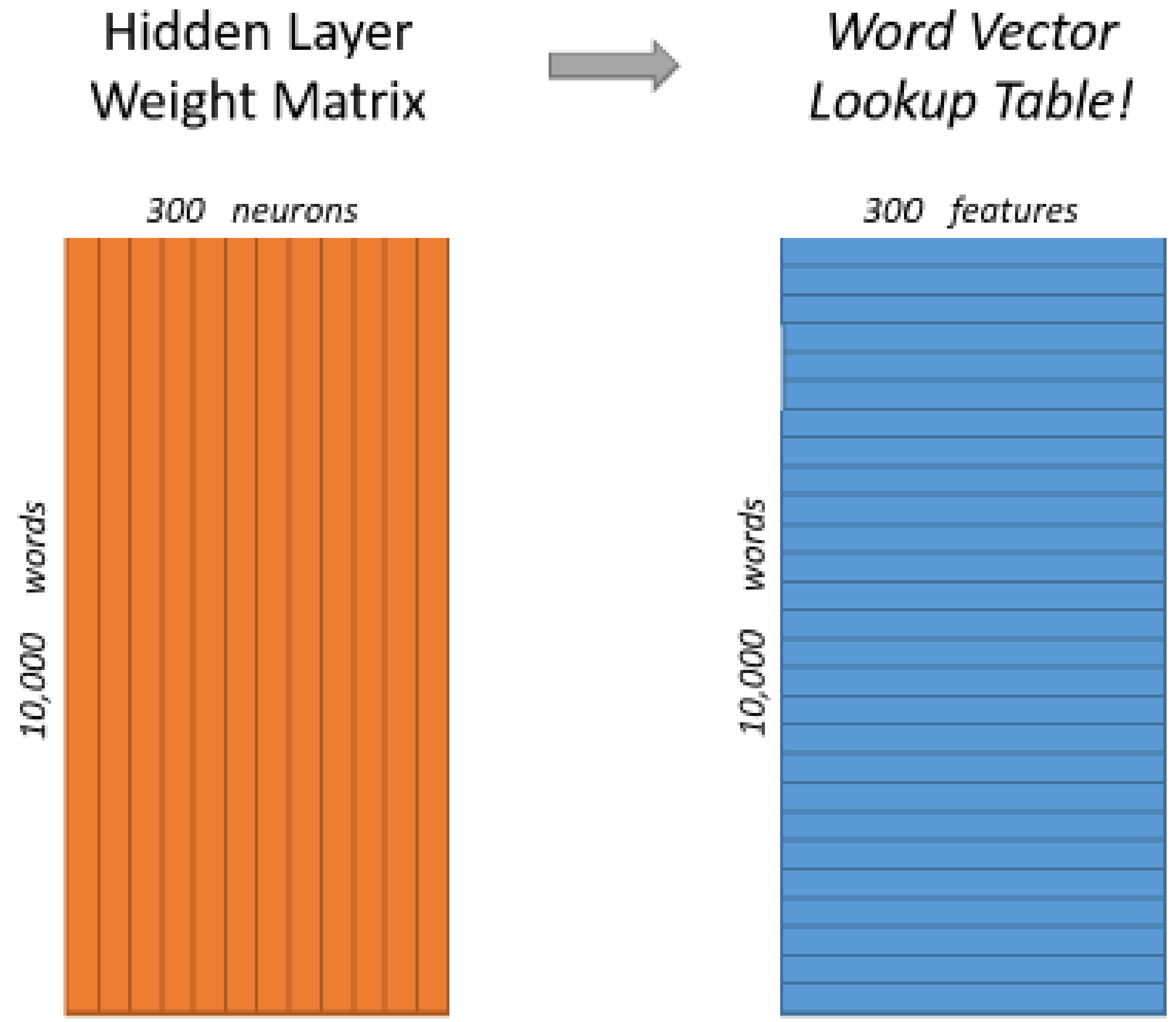
Input one-hot vector $1*|V|$

M1 $|V|*n$

Output of the hidden layer $1*n$

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = \begin{bmatrix} 10 & 12 & 19 \end{bmatrix}$$

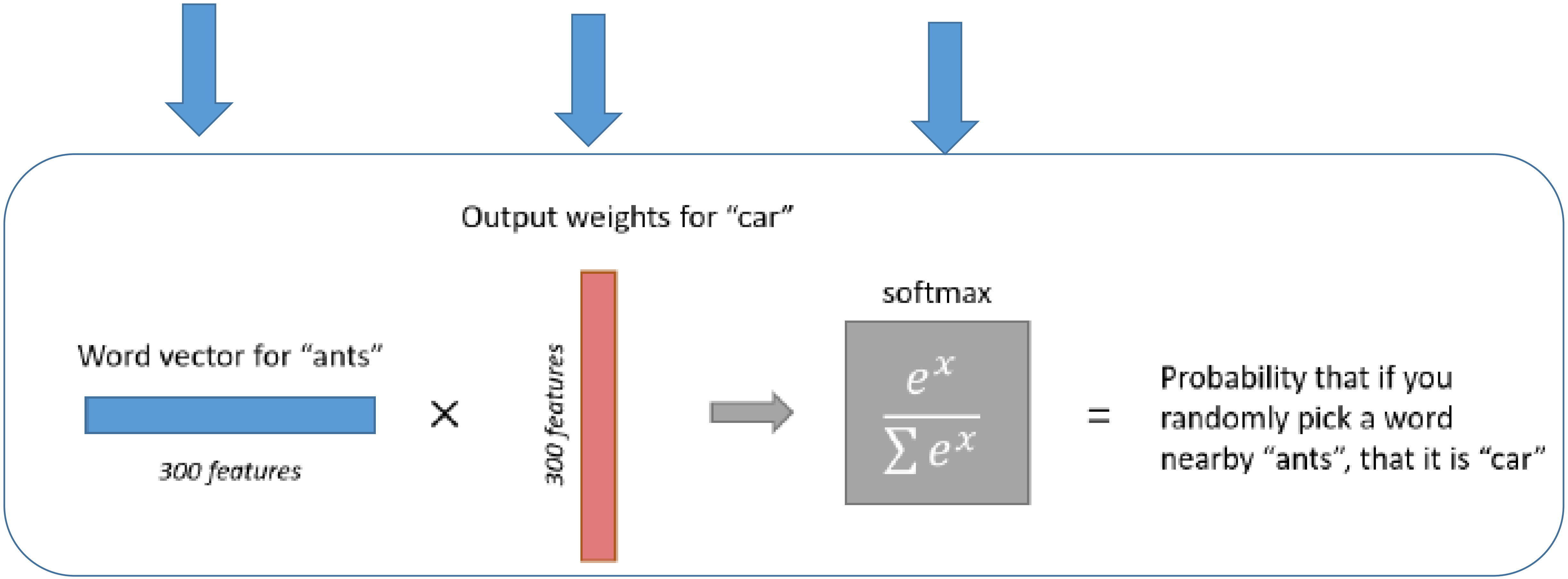
The output of the hidden layer is just the word vector (or word embedding) of the input word.


Hidden Layer Weight Matrix
Word Vector Lookup Table!
300 neurons
10,000 words
300 features
10,000 words

Output of the hidden layer for Input word “ants” in M_1

The column for “car” in M_2

The output layer after softmax



Time complexity for training

Time complexity is O(E*T*Q):

- E is the number of the training epochs
- T is the number of word tokens in the training set
- Q is determined by the model architecture (e.g., sizes of the two matrices)

Big model:

- The number of parameters is 2 * Vocabulary size * embedding size

Tricks for Skip-Gram training (Mikolov et al., 2013-NIPS)

- Tricks:
  - Subsampling frequent words
  - Negative sampling
  - Dealing with phrases
  - Hierarchical softmax: to speed up training

- The first three tricks are explained at the blog

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

Frequent words in the training data

In the sentence: “The fox jumps over the fence”

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”.

Subsampling frequent words

- For each word we encounter in our training text, there is a chance that we will effectively delete it from the text. The probability that we cut the word is positively correlated with the word’s frequency.

- 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.

Negative sampling

- Need to adjust all the weights in M2 for every training example.

  ➔ The number of weights in M2 being updated is Vocabulary size * embedding size.

- Negative sampling: for each positive example (w1, w2), randomly choose k (w1, w2') negative examples.

  ➔ The number of weights in M2 being updated is (k+1) * embedding size.

- Selecting negative samples:

$$P(w_i) = \frac{f(w_i)}{\sum_{j=0}^{n} \left(f(w_j)\right)}$$

Increase the prob for less frequent words:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n} \left(f(w_j)^{3/4}\right)}$$

Summary

- There are many ways to learn word embeddings.

- Word2vec is one of the earliest and most well-known methods:
  - Creating “fake” tasks and use the weights from the models
  - There are two models: CBOW and Skip-gram
  - For both, use feed-forward NNs with linear neurons (to make learning faster)

- There are many implementations:
  - Speed is a big issue.
  - Many tricks to make training faster: e.g., https://rare-technologies.com/word2vec-in-python-part-two-optimizing/