

# Semantics with Dense Vectors

Reference:

- D. Jurafsky and J. Martin, "Speech and Language Processing"

# Semantics with Dense Vectors

- We saw how to represent a word as a sparse vector with dimensions corresponding to the words in the vocabulary, and whose values were some function of the count of the word co-occurring with each neighboring word.
- Each word is represented with a vector that is both **long** (length  $|V|$ , with vocabularies of 20,000 to 50,000) and **sparse**, with most elements of the vector for each word equal to zero.

# Semantics with Dense Vectors

- Now we turn to an alternative family of methods of representing a word.
- We use the vectors that are **short** (of length perhaps 50 – 1000) and **dense** (most values are non-zero).
- Short vectors have a number of potential advantages.
- First, they are easier to include as features in machine learning systems.

# Semantics with Dense Vectors

- For example, if we use 100-dimensional word embeddings as features:
- A classifier can just learn 100 weights to represent a function of word meaning, instead of having to learn tens of thousands of weights for each of the sparse dimensions.
- Because they contain fewer parameters than sparse vectors of explicit counts, dense vectors may generalize better and help avoid overfitting.

# Semantics with Dense Vectors

- Dense vectors may do a better job of capturing synonymy than sparse vectors.
- For example, *car* and *automobile* are synonyms.
- In a typical sparse vectors representation, the *car* dimension and the *automobile* dimension are distinct dimensions.
- Because the relationship between these two dimensions is not modeled, sparse vectors may fail to capture the similarity between a word with *car* as a neighbor and a word with *automobile* as a neighbor.

# Dense Vectors via SVD

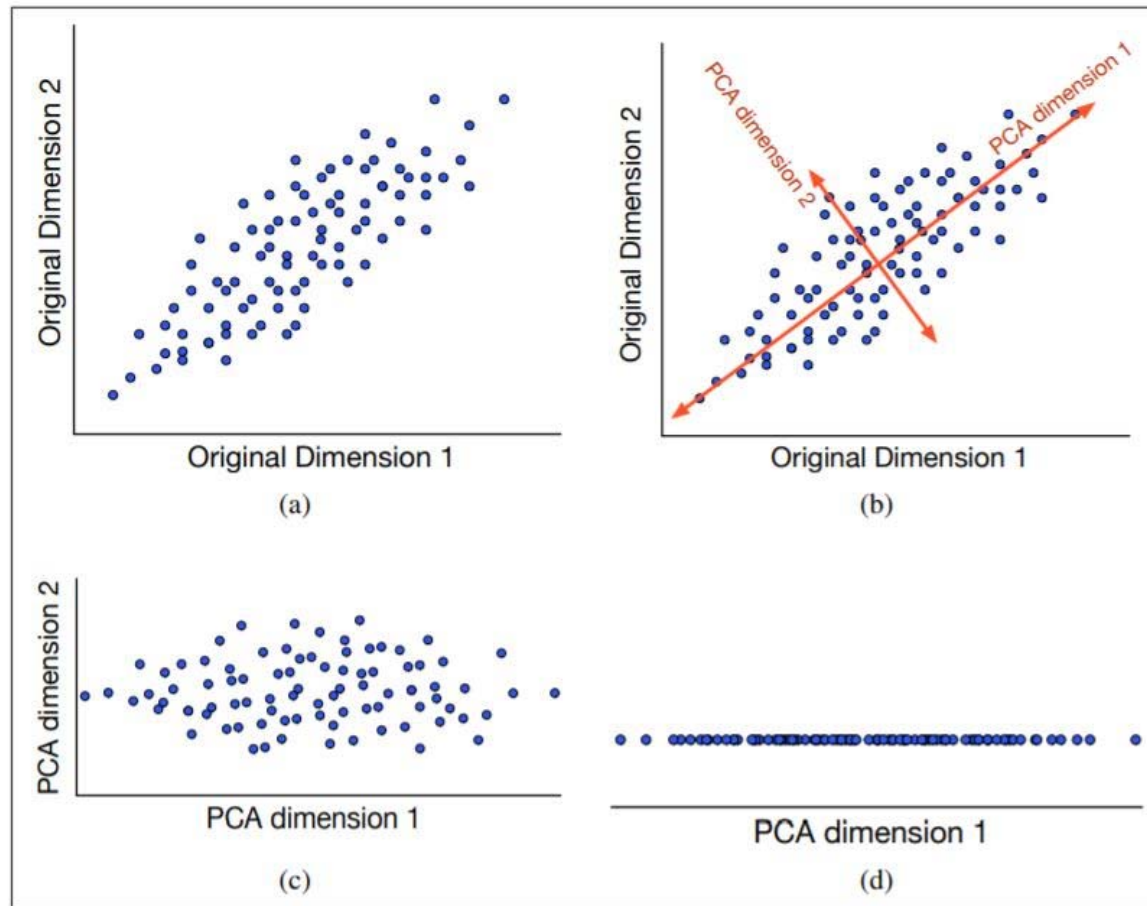
- We begin with a classic method for generating dense vectors: **singular value decomposition**, or **SVD**
- It is first applied to the task of generating embeddings from term-document matrices by Deerwester et al. (1988) in a model called **Latent Semantic Indexing** or **Latent Semantic Analysis (LSA)**
- Singular Value Decomposition (SVD) is a method for finding the most important dimensions of a data set, those dimensions along which data varies the most.

# Dense Vectors via SVD

- In general, dimensionality reduction methods first rotate the axes of the original dataset into a new space.
- The new space is chosen so that the highest order dimension captures the most variance in the dataset.
- The next dimension captures the next most variance, and so on.

# Dense Vectors via SVD

- The following figure shows a visualization.





# Dense Vectors via SVD

- A set of points (vectors) in two dimensions is rotated so that the first new dimension captures the most variation in the data.
- In this new space, we can represent data with a smaller number of dimensions (for example using one dimension instead of two) and still capture much of the variation in the original data.

# Latent Semantic Analysis

- LSA is a particular application of SVD to a  $|V| \times c$  term-document matrix  $X$  representing  $|V|$  words and their co-occurrence with  $c$  documents or contexts.
- SVD factorizes any such rectangular  $|V| \times c$  matrix  $X$  into the product of three matrices  $W$ ,  $\Sigma$ , and  $Y^T$ , i.e.  $W\Sigma Y^T$
- In the  $|V| \times m$  matrix  $W$ , each of the  $w$  rows still represents a word, but columns do not.

# Latent Semantic Analysis

- Each column now represents one of  $m$  dimensions in a latent space.
- The  $m$  column vectors are orthogonal to each other.
- The columns are ordered by the amount of variance in the original dataset each accounts for.
- The number of such dimensions  $m$  is the **rank** of  $X$  (the rank of a matrix is the number of linearly independent rows).

# Latent Semantic Analysis

- $\Sigma$  is a diagonal  $m \times m$  matrix, with **singular values** along the diagonal, expressing the importance of each dimension.
- The  $m \times c$  matrix  $Y^T$ , denoted as  $C$ , still represents documents or contexts, but each row now represents one of the new latent dimensions and the  $m$  row vectors are orthogonal to each other.

# Latent Semantic Analysis

- By using only the first  $k$  dimensions, of  $W$ ,  $\Sigma$ , and  $C$  instead of all  $m$  dimensions, the product of these 3 matrices becomes a least-squares approximation to the original  $X$ .
- Since the first dimensions encode the most variance, one way to view the reconstruction is thus as modeling the most important information in the original dataset.

# Latent Semantic Analysis

- SVD applied to co-occurrence matrix  $X$ :

$$\begin{bmatrix} X \\ |V| \times c \end{bmatrix} = \begin{bmatrix} W \\ |V| \times m \end{bmatrix} \begin{bmatrix} \begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_m \end{bmatrix} \\ m \times m \end{bmatrix} \begin{bmatrix} C \\ m \times c \end{bmatrix}$$

# Latent Semantic Analysis

- Taking only the top  $k$ ,  $k \leq m$  dimensions after the SVD is applied to the co-occurrence matrix  $X$ :

$$\begin{bmatrix} X \\ |V| \times c \end{bmatrix} = \begin{bmatrix} W_k \\ |V| \times k \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_k \end{bmatrix} \begin{bmatrix} C \\ k \times c \end{bmatrix}$$

SVD factors a matrix into a product of three matrices,  $W$ ,  $\Sigma$ , and  $C$ . Taking the first  $k$  dimensions gives a  $|V| \times k$  matrix  $W_k$  that has one  $k$ -dimensioned row per word that can be used as an embedding.

# Latent Semantic Analysis

- Using only the top  $k$  dimensions (corresponding to the  $k$  most important singular values) leads to a reduced  $|V| \times k$  matrix  $W_k$ , with one  $k$ -dimensioned row per word.
- This row now acts as a dense  $k$ -dimensional vector (embedding) representing that word, substituting for the very high-dimensional rows of the original  $X$ .
- LSA embeddings generally set  $k=300$ , so these embeddings are relatively short by comparison to other dense embeddings.



# Latent Semantic Analysis

- Instead of PPMI or tf-idf weighting on the original term-document matrix, LSA implementations generally use a particular weighting of each co-occurrence cell that multiplies two weights called the **local** and **global** weights for each cell  $(i, j)$  – *term  $i$  in document  $j$* .

# Latent Semantic Analysis

- The local weight of each term  $i$  is its log frequency:

$$\log f(i, j) + 1$$

- The global weight of term  $i$  is a version of its entropy:

$$1 + \frac{\sum_j p(i, j) \log p(i, j)}{\log D}$$

- $D$  is the number of documents.

# SVD applied to word-context matrices

- Rather than applying SVD to the term-document matrix, an alternative that is widely practiced is to apply SVD to the word-word or word-context matrix.
- In this version the context dimensions are words rather than documents.

# SVD applied to word-context matrices

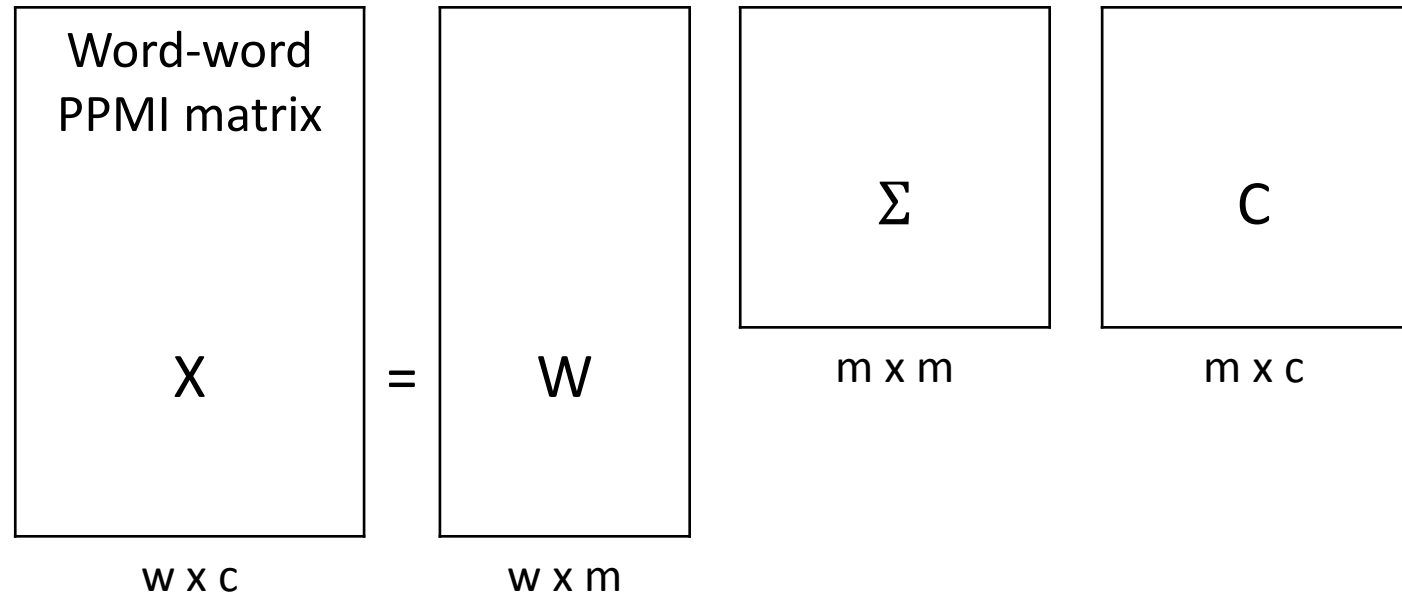
- The mathematics is identical to what is described in LSA.
- SVD factorizes the word-context matrix  $X$  into three matrices  $W$ ,  $\Sigma$ , and  $C^T$ .
- The only difference is that we are starting from a PPMI-weighted word-word matrix, instead of a term-document matrix.

# SVD applied to word-context matrices

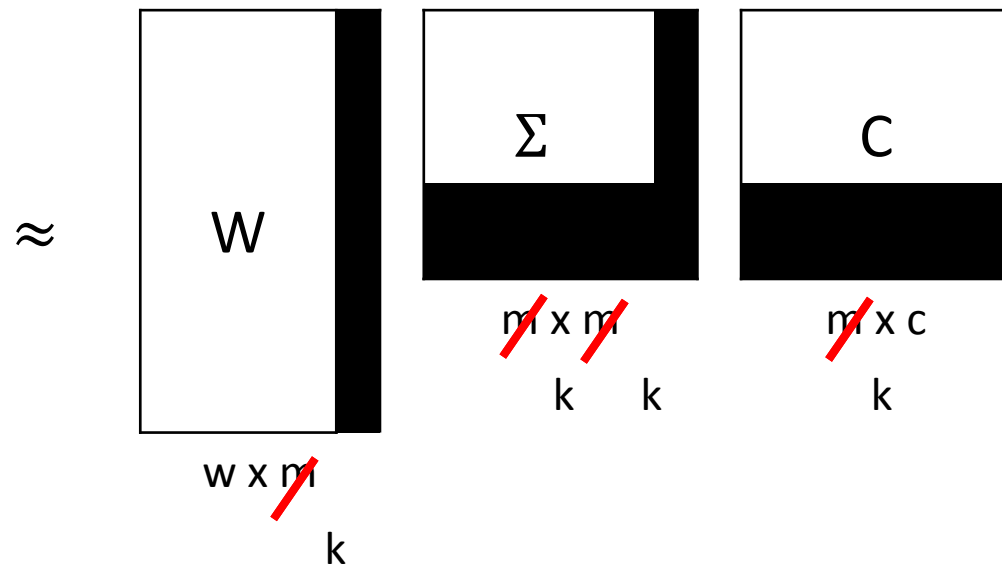
- Once again only the top  $k$  dimensions are retained (corresponding to the  $k$  most important singular values), leading to a reduced  $|V| \times k$  matrix  $W_k$ , with one  $k$ -dimensioned row per word.
- Just as with LSA, this row acts as a dense  $k$ -dimensional vector (embedding) representing that word.
- The other matrices ( $\Sigma$  and  $C$ ) are simply thrown away.

# SVD applied to word-context matrices

## 1) SVD

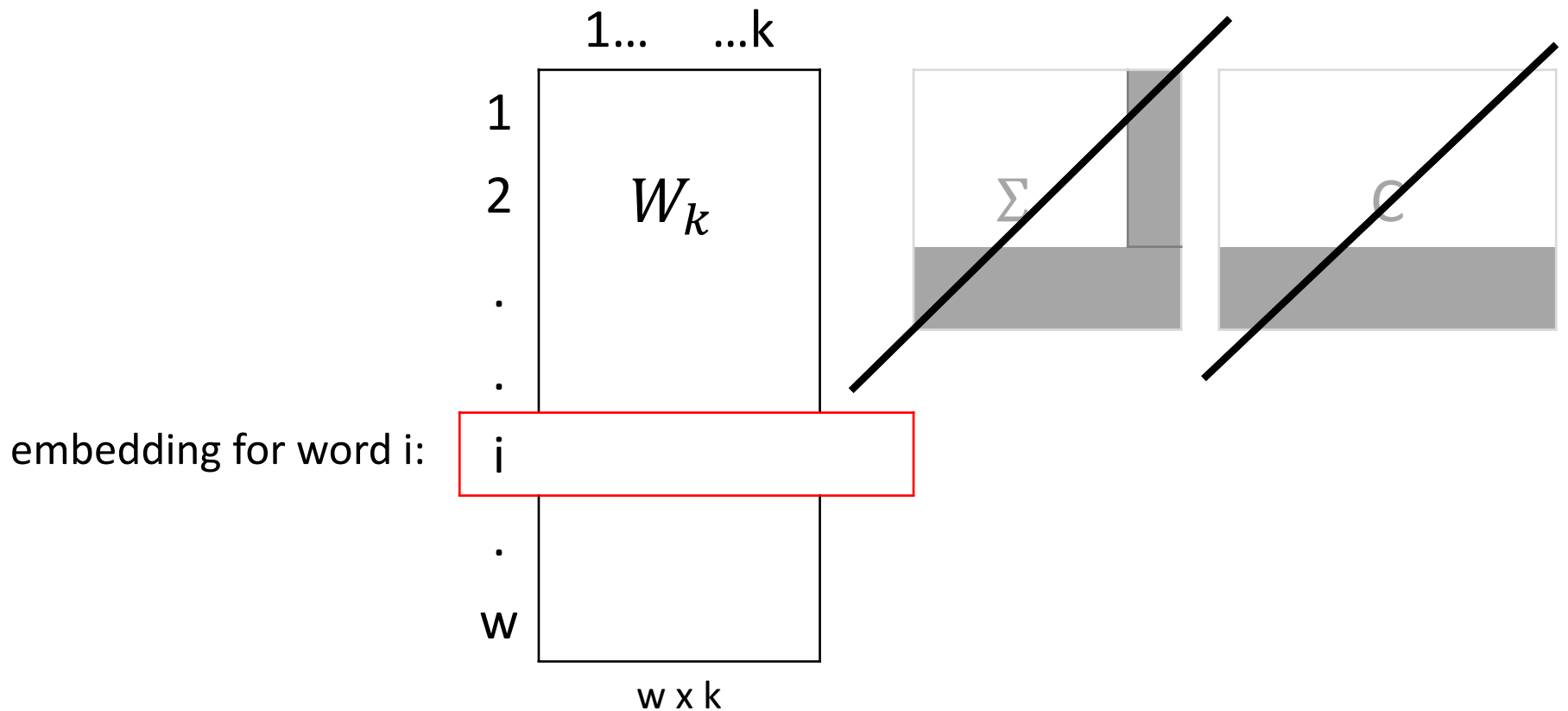


## 2) Truncation:



# SVD applied to word-context matrices

## 3) Embeddings:



# SVD applied to word-context matrices

- This use of just the top dimensions, whether for a term-document matrix like LSA, or for a term-term matrix, is called **truncated SVD**.
- Truncated SVD is parameterized by  $k$ , the number of dimensions in the representation for each word, typically ranging from 500 to 5,000.
- Thus SVD run on term-context matrices tends to use many more dimensions than the 300-dimensional embeddings produced by LSA.



# SVD applied to word-context matrices

- This difference presumably has something to do with the difference in granularity.
- LSA counts for words are much coarser-grained, counting the co-occurrences in an entire document, while word-context PPMI matrices count words in a small window.

# Embeddings from prediction: Skip-gram and CBOW

- A second method for generating dense embeddings draws its inspiration from the neural network models used for language modeling.
- The idea is: Given a word, we predict context words.
- The intuition is that words with similar meanings often occur near each other in texts.

# Embeddings from prediction: Skip-gram and CBOW

- We learn an embedding by starting with a random vector and then iteratively shifting a word's embeddings to be more like the embeddings of neighboring words, and less like the embeddings of words that don't occur nearby.

# Embeddings from prediction: Skip-gram and CBOW

- The most popular family of methods is referred to as **word2vec**, after the software package that implements two methods for generating dense embeddings: **skip-gram** and **CBOW (continuous bag of words)**.

# Embeddings from prediction: Skip-gram and CBOW

- The word2vec models learn embeddings by training a network to predict neighboring words.
- The prediction task is not the main goal.
- Words that are semantically similar often occur near each other in text, and so embeddings that are good at predicting neighboring words are also good at representing similarity.
- The advantage of the word2vec methods is that they are fast, efficient to train, and easily available online with code and pretrained embeddings.

# Embeddings from prediction: Skip-gram

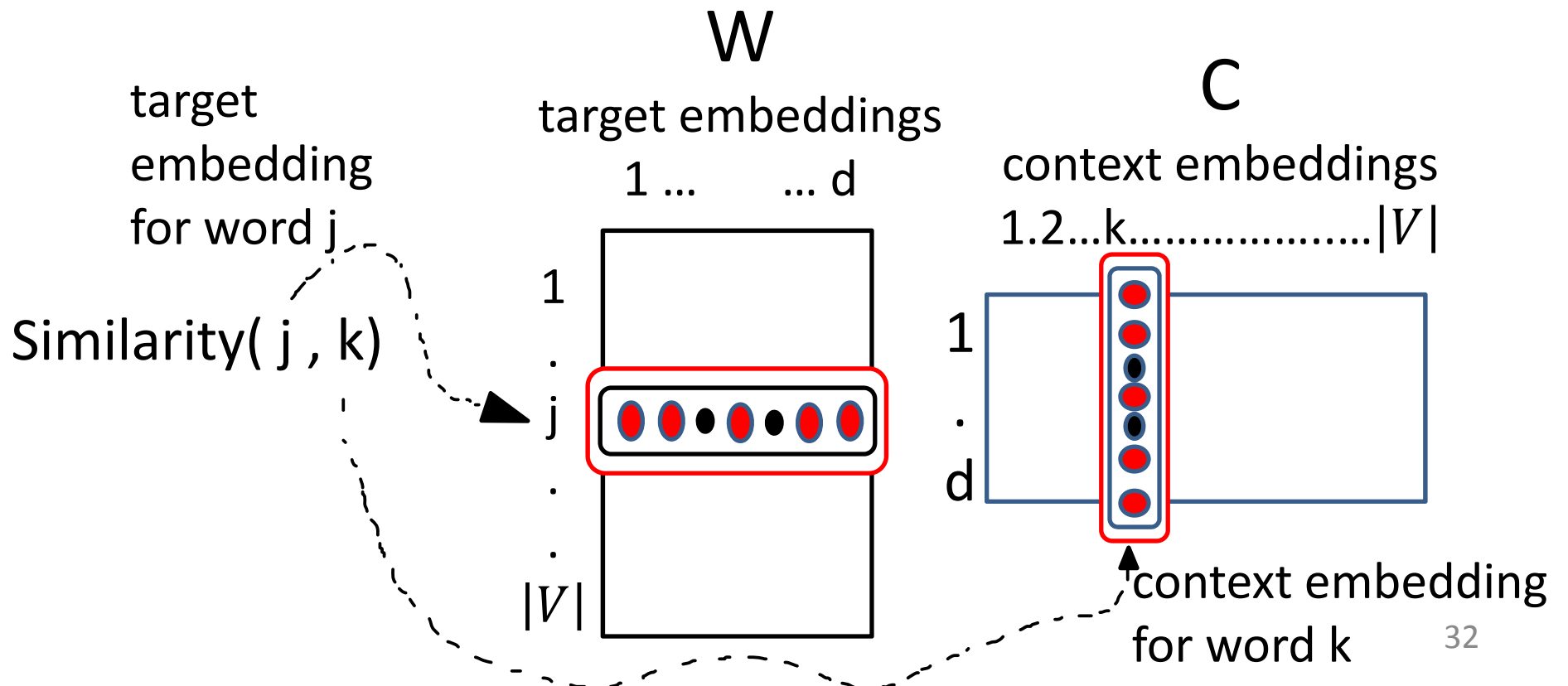
- We'll begin with the skip-gram model.
- Like the SVD model, the skip-gram model actually learns two separate embeddings for each word  $w$ : the **word embedding**  $v$  and the **context embedding**  $c$ .
- These embeddings are encoded in two matrices, the **word matrix**  $W$  and the **context matrix**  $C$ .

# Embeddings from prediction: Skip-gram

- We'll discuss how  $W$  and  $C$  are learned, but let's first see how they are used.
- Each row  $i$  of the word matrix  $W$  is the  $1 \times d$  vector embedding  $v_i$  for word  $i$  in the vocabulary.
- Each column  $i$  of the context matrix  $C$  is a  $d \times 1$  vector embedding  $c_i$  for word  $i$  in the vocabulary.
- In principle, the word matrix and the context matrix could use different vocabularies  $V_w$  and  $V_c$ .
  - We'll simplify by assuming the two matrices share the same vocabulary, which we'll just call  $V$ .

# Embeddings from prediction: Skip-gram

- The following figure shows the intuition that the similarity function requires selecting out a target vector  $v_j$  from  $W$ , and a context vector  $c_k$  from  $C$ .





# Embeddings from prediction: Skip-gram

- Let's consider the prediction task.
- We are walking through a corpus of length  $T$  and currently pointing at the  $t$  th word  $w^{(t)}$ , whose index in the vocabulary is  $j$ , so we'll call it  $w_j$  ( $1 < j < |V|$ ).
- The skip-gram model predicts each neighboring word in a context window of  $2L$  words from the current word.

# Embeddings from prediction: Skip-gram

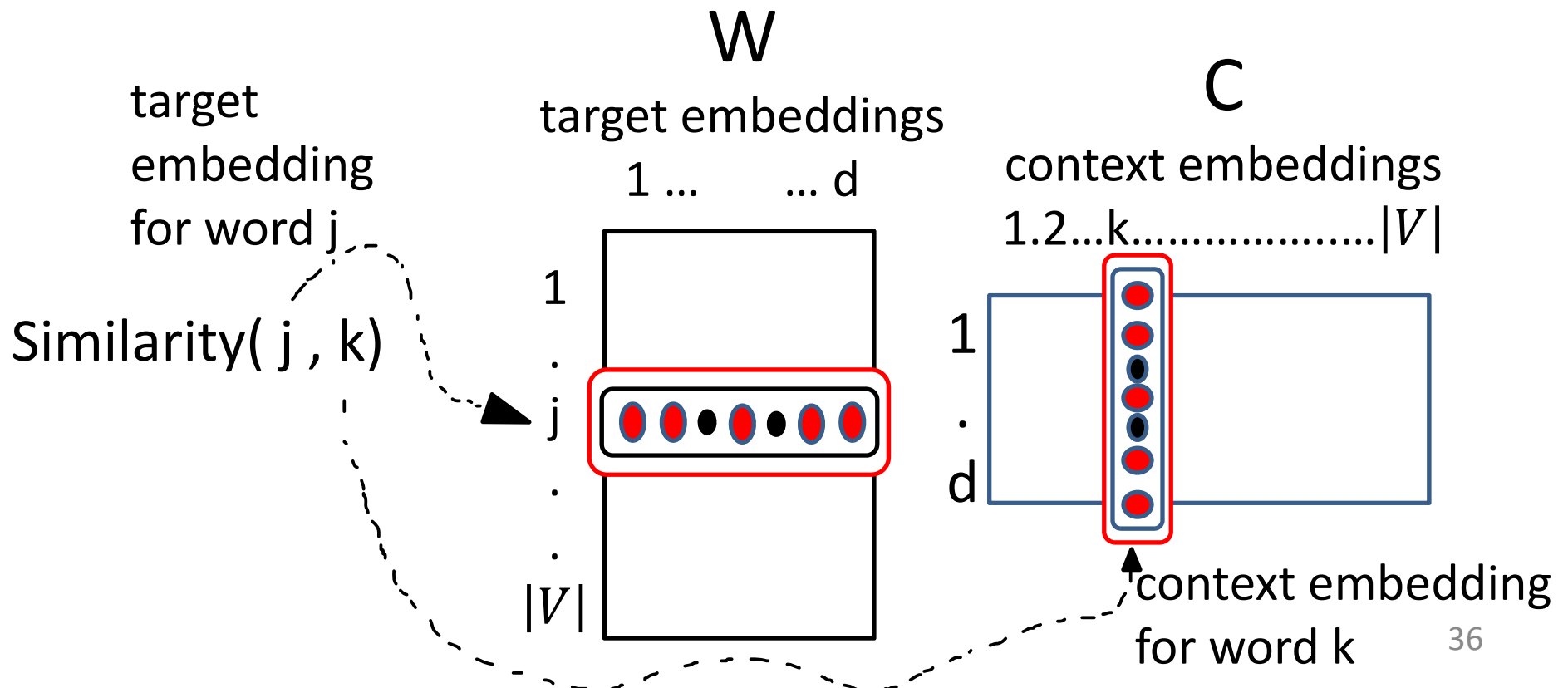
- So for a context window  $L = 2$  the context is  $[w^{t-2}, w^{t-1}, w^{t+1}, w^{t+2}]$  and we are predicting each of these from word  $w_j$ .
- But let's simplify for a moment and imagine just predicting one of the  $2L$  context words, for example  $w^{(t+1)}$ , whose index in the vocabulary is  $k$  ( $1 < k < |V|$ ).
- Hence our task is to compute  $P(w_k | w_j)$ .

# Embeddings from prediction: Skip-gram

- The heart of the skip-gram computation of the probability  $p(w_k | w_j)$  is computing the dot product between the vectors for  $w_j$  and  $w_k$ , namely, the *target vector* for  $w_j$  and the *context vector* for  $w_k$
- For simplicity, we'll represent this dot product as  $c_k \cdot v_j$  (precisely it is  $c_k^T v_j$ ).  
where  $c_k$  is the context vector of word  $k$  and  $v_j$  is the target vector for word  $j$ .

# Embeddings from prediction: Skip-gram

- The following figure shows the intuition that the similarity function requires selecting out a target vector  $v_j$  from  $W$ , and a context vector  $c_k$  from  $C$ .



# Embeddings from prediction: Skip-gram

- The higher the dot product between two vectors, the more similar they are.
- That was the intuition of using the cosine as a similarity metric, since cosine is just a normalized dot product.

# Embeddings from prediction: Skip-gram

- Of course, the dot product  $c_k \cdot v_j$  is not a probability, it's just a number ranging from  $-\infty$  to  $\infty$ .
- We can use the *softmax* function to normalize the dot product into probabilities.
- Computing this denominator requires computing the dot product between each other word  $w$  in the vocabulary with the target word  $w_i$ :

$$p(w_k | w_j) = \frac{\exp(c_k \cdot v_j)}{\sum_{i \in |V|} \exp(c_i \cdot v_j)}$$

# Embeddings from prediction: Skip-gram

- In summary, the skip-gram computes the probability  $p(w_k | w_j)$  by taking the dot product between the word vector for  $j$  ( $v_j$ ) and the context vector for  $k$  ( $c_k$ ), and turning this dot product  $v_j \cdot c_k$  into a probability by passing it through a softmax function.

# Embeddings from prediction: Skip-gram

- This version of the algorithm, however, has a problem: the time it takes to compute the denominator.
- For each word  $w^t$ , the denominator requires computing the dot product with all other words.
- In practice, we generally solve this by using an approximation of the denominator.



# Embeddings from prediction: CBOW

- The CBOW (**continuous bag of words**) model is roughly the mirror image of the skip-gram model.
- Like skip-grams, it is based on a predictive model, but this time predicting the current word  $w_t$  from the context window of  $2L$  words around it.

e.g. for  $L = 2$  the context is  $[w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}]$

# Embeddings from prediction: CBOW

- While CBOW and skip-gram are similar algorithms and produce similar embeddings, they do have slightly different behavior.
- Often one of them will turn out to be the better choice for a particular task.

# Learning word and context embeddings

- We already mentioned the intuition for learning the word embedding matrix  $W$  and the context embedding matrix  $C$ :
- Iteratively make the embeddings for a word more like the embeddings of its neighbors and less like the embedding of other words.

# Learning word and context embeddings

- In this version of the prediction algorithm, the probability of a word is computed by normalizing the dot-product between a word and each context word by the dot products for all words.
- This probability is optimized when a word's vector is closest to the words that occur near it (the numerator), and further from every other word (the denominator).
- Such a version of the algorithm is very expensive.
- We need to compute a whole lot of dot products to make the denominator.

# Learning word and context embeddings

- Instead, the most commonly used version of skip-gram, *skip-gram with negative sampling*, approximates this full denominator.
- We will describe a brief sketch of how this works.

# Learning word and context embeddings

- In the training phase, the algorithm walks through the corpus.
- At each target word choosing the surrounding context words as positive examples.
- For each positive example also choosing  $k$  **noise** samples or **negative samples**: non-neighbor words.
- The goal will be to move the embeddings toward the neighbor words and away from the noise words.

# Learning word and context embeddings

- For example, in walking through the example text below we come to the word *apricot*, and let  $L = 2$  so we have 4 context words  $c_1$  through  $c_4$ :

lemon, a  $\left[ \begin{array}{cccc} \textit{tablespoon} & \textit{of} & \textit{apricot} & \textit{preserves} & \textit{or} \end{array} \right]$  jam  
 $c_1 \quad c_2 \quad w \quad c_3 \quad c_4$

- The goal is to learn an embedding whose dot product with each context word is high.

# Learning word and context embeddings

- In practice skip-gram uses a sigmoid function  $\sigma$  of the dot product, where

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

- For the above example, we want  $\sigma(c1 \cdot w) + \sigma(c2 \cdot w) + \sigma(c3 \cdot w) + \sigma(c4 \cdot w)$  to be high.



# Learning word and context embeddings

- In addition, for each context word the algorithm choose  $k$  random noise words according to their unigram frequency.
- If we let  $k = 2$ , for each target/context pair, we'll have 2 noise words for each of the 4 context words:

*cement metaphysical dear coaxial*  
*n1 n2 n3 n4*

*apricot attendant whence forever puddle*  
*n5 n6 n7 n8*

# Learning word and context embeddings

- We'd like these noise words  $n$  to have a low dot-product with our target embedding  $w$ .
- In other words we want  $\sigma(n1 \cdot w) + \sigma(n2 \cdot w) + \dots + \sigma(n8 \cdot w)$  to be low.

# Learning word and context embeddings

- More formally, the learning objective for one word/context pair  $(w, c)$  is

$$\log \sigma(c \cdot w) + \sum_{i=1}^k \mathbb{E}_{w_i \sim p(w)} [\log \sigma(-w_i \cdot w)]$$

- We want to maximize the dot product of the word with the actual context words, and minimize the dot products of the word with the  $k$  negative sampled non-neighbor words.
- The noise words  $w_i$  are sampled from the vocabulary  $V$  according to their weighted unigram probability.
- In practice rather than  $p(w)$  it is common to use the weighting  $p^{\frac{3}{4}}(w)$ .

# Learning word and context embeddings

- The learning algorithm starts with randomly initialized  $W$  and  $C$  matrices.
- Then, the learning algorithm walks through the training corpus moving  $W$  and  $C$  so as to maximize the objective just previously mentioned.
- An algorithm like stochastic gradient descent is used to iteratively shift each value so as to maximize the objective, using error backpropagation to propagate the gradient back through the network.

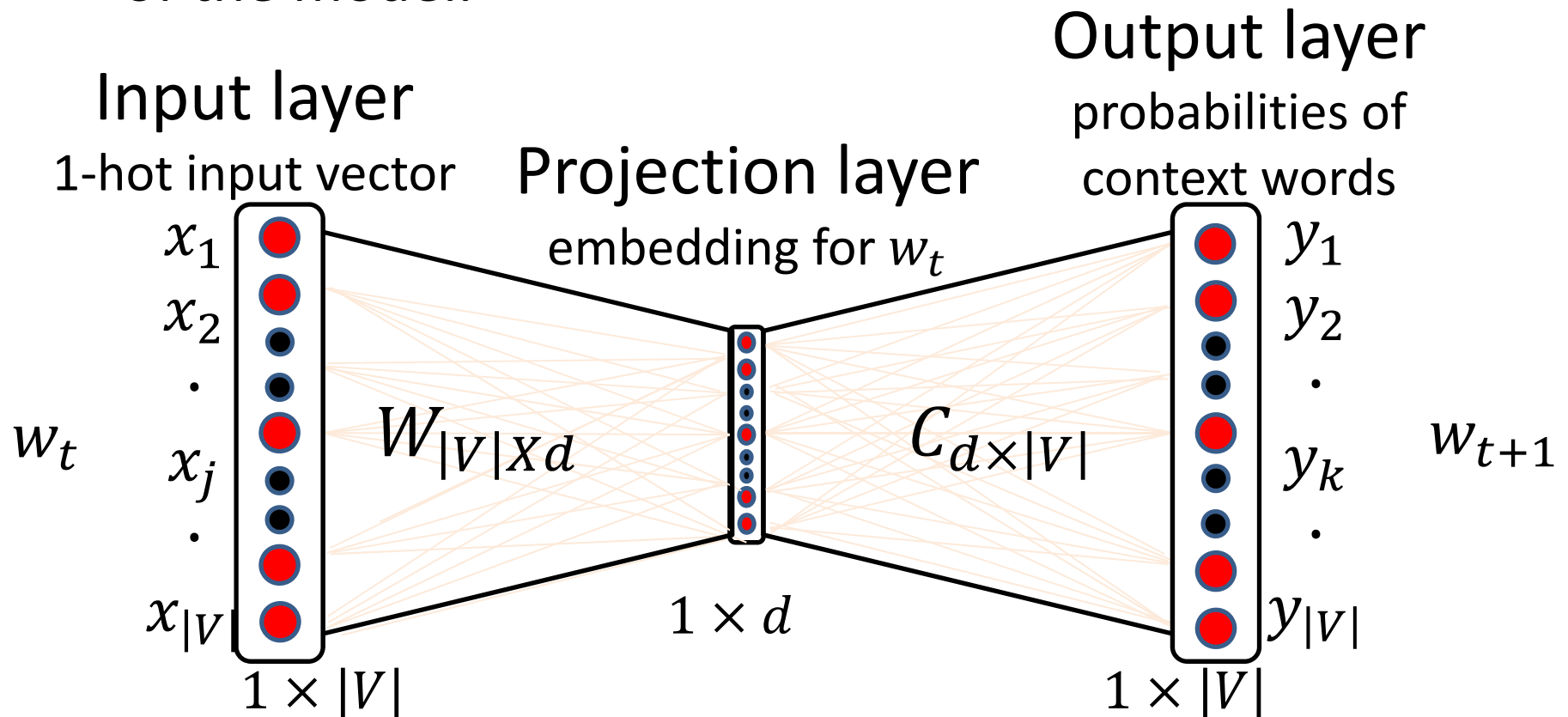
# Learning word and context embeddings

- In summary, the above learning objective is not the same as the  $p(w_k | w_j)$ .
- Nonetheless, although negative sampling is a different objective than the probability objective, and so the resulting dot products will not produce optimal predictions of upcoming words, it seems to produce good embeddings, and that's the goal we care about.

# Learning word and context embeddings

## Visualizing the Network

- The following figure shows a simplified visualization of the model.



The skip-gram model viewed as a network.

# Learning word and context embeddings

## Visualizing the Network

- Using error backpropagation requires that we envision the selection of the two vectors from the  $W$  and  $C$  matrices as a network that we can propagate backwards across.
- We've simplified to predict a single context word rather than 2L context words, and simplified to show the softmax over the entire vocabulary rather than just the  $k$  noise words.

# Learning word and context embeddings

- It's worth taking a moment to envision how the network is computing the same probability as the dot product version we described above.
- In the above network, we begin with an input vector  $x$ , which is a **one-hot** vector for the current word  $w_j$ .



# Learning word and context embeddings

- A one-hot vector is just a vector that has one element equal to 1, and all the other elements are set to zero.
- Thus in a one hot presentation for the word  $w_j, x_j = 1$ , and  $x_i = 0 \forall i \neq j$ , as shown in the following:

$w_0$	$w_1$									$w_j$										$w_{ V }$
0	0	0	0	0	...	0	0	0	0	1	0	0	0	0	0	...	0	0	0	0

A one-hot vector, with the dimension corresponding to word  $w_j$  set to 1.

# Learning word and context embeddings

- We then predict the probability of each of the  $2L$  output words – that means the one output word  $w_{t+1}$  – in 3 steps.

# Learning word and context embeddings

## 1. Select the embedding from $W$ .

- $x$  is multiplied by  $W$ , the input matrix, to give the hidden or **projection layer**.
- Since each row of the input matrix  $W$  is just an embedding for word  $w_t$ , and the input is a one-hot column vector for  $w_j$ , the projection layer for input  $x$  will be  $h = W * w_j = v_j$ , the input embedding for  $w_j$ .

# Learning word and context embeddings

## 2. Compute the dot product $c_k \cdot v_j$ .

- For each of the  $2L$  context words we now multiply the projection vector  $h$  by the context matrix  $C$ .
- The resulting for each context word,  $o = Ch$ , is a  $1 \times |V|$  dimensional output vector giving a score for each of the  $|V|$  vocabulary words.
- In doing so, the element  $o_k$  was computed by multiplying  $h$  by the *output embedding* for word  $w_k$ :  
$$w_k: o_k = c_k \cdot h = c_k \cdot v_j.$$

# Learning word and context embeddings

## 3. Normalize the dot products into probabilities.

- For each context word we normalize this vector of dot product scores, turning each score element  $o_k$  into a probability by using softmax function:

$$p(w_k | w_j) = y_k = \frac{\exp(c_k \cdot v_j)}{\sum_{i \in |V|} \exp(c_i \cdot v_j)}$$

# Properties of embeddings

- The following table shows the words/phrases that are most similar to some sample words using the phrase-based version of the skip-gram algorithm.

---

<b>target:</b>	Redmond	Havel	ninjutsu	graffiti	capitulate
	Redmond Wash.	Vaclav Havel	ninja	spray paint	capitulation
	Redmond Washington	president Vaclav Havel	martial arts	graffiti	capitulated
	Microsoft	Velvet Revolution	swordsmanship	taggers	capitulating

---

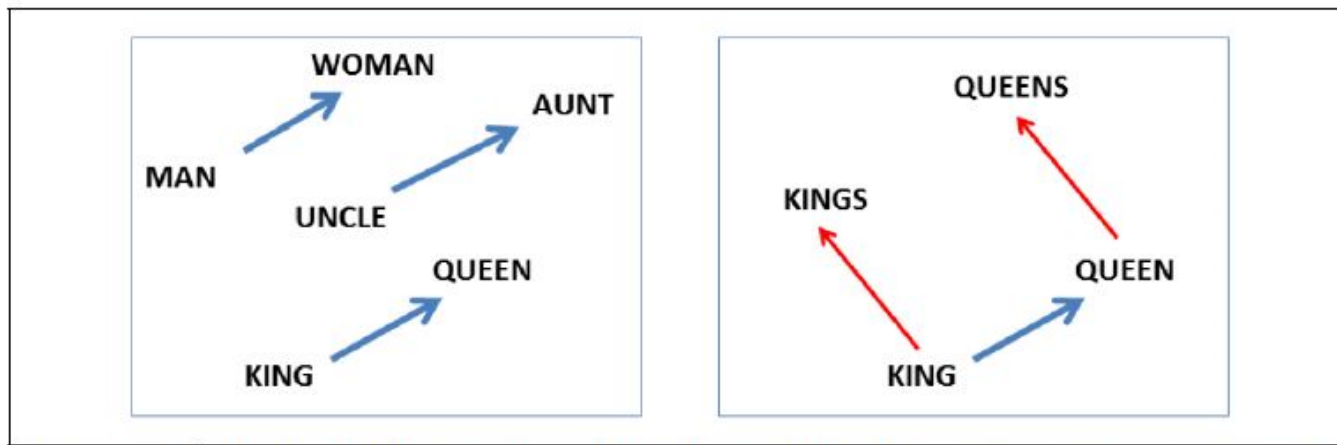
Examples of the closest tokens to some target words using a phrase-based extension of the skip-gram algorithm.

# Properties of embeddings

- One semantic property of various kinds of embeddings that may play in their usefulness is their ability to capture relational meanings.
- Mikolov et al. demonstrates that the *offsets* between vector embeddings can capture some relations between words.

# Properties of embeddings

- For example, the result of the expression  $\text{vector}('king') - \text{vector}('man') + \text{vector}('woman')$  is a vector close to  $\text{vector}('queen')$ .
- The left panel in the following figure visualizes this by projecting a representation down into 2 dimensions.





# Properties of embeddings

- Similarly, they found that the expression  $\text{vector}(\textit{Paris}) - \text{vector}(\textit{France}) + \text{vector}(\textit{Italy})$  results in a vector that is very close to  $\text{vector}(\textit{Rome})$ .
- Levy and Goldberg show that various other kinds of embeddings also seem to have this property.