CS 60050
Machine Learning

Word Embeddings
An Application of Neural Networks

Some slides taken from various presentations available on the Web

How to represent a word

- Vocabulary: set of distinct words in a text collection
- Consider a vocabulary of size V
- How to represent each word?
- Simple representation:
  - One-hot representation: a vector of size V, with one 1 and rest zeroes
  - Words can be arranged in some order, e.g., alphabetically

dog	1	[1 0 0 0 0 0 0 0 0 0]
cat	2	[0 1 0 0 0 0 0 0 0 0]
person	3	[0 0 1 0 0 0 0 0 0 0]

Problem of One-Hot representation

- High dimensionality

  E.g.) For Google News, V = 13M words

- Very sparse: Only 1 non-zero value
  - Many operations are difficult on such sparse vectors

- Shallow representation

  motel = [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0] AND

  hotel = [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0] = 0
  - Distance between any two words always the same
  - One-hot representations do not capture any semantics

Word embeddings

- A learned representation for text where
  - Every word represented by a low-dimensional vector
  - Words that have the same meaning, have a similar representation

- Representations of similar words (e.g., ‘motel’ and ‘hotel’) would be similar (will have similar values in every dimension)

  motel = [1.3, -1.4] and hotel = [1.2, -1.5]

- Typical number of dimensions: 300

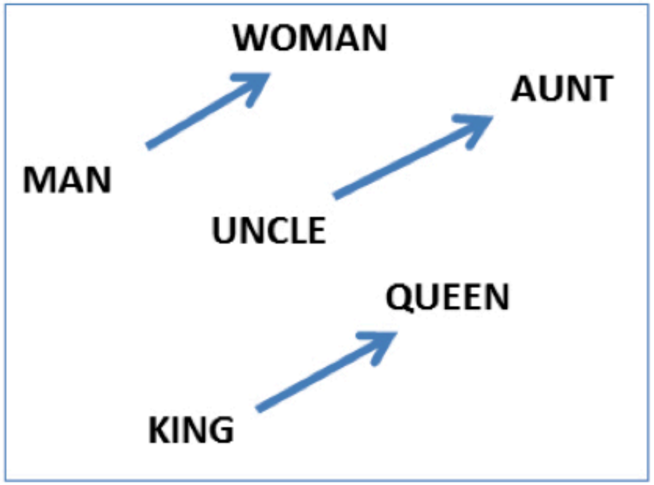
Word2vec

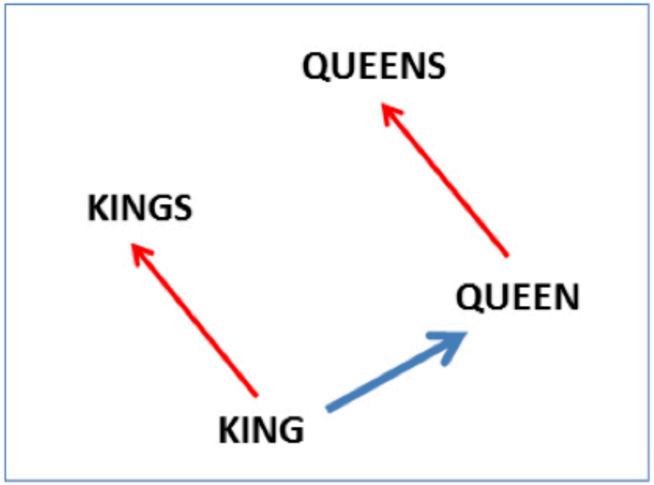
- word2vec is **not** a single algorithm
- A **software package** for representing words as vectors
  - Two distinct models
    - Continuous bag of words (CBoW)
    - **Skip-Gram**
  - Various training methods
    - Negative Sampling
    - Hierarchical Softmax
  - A rich preprocessing pipeline
    - Dynamic Context Windows
    - Subsampling
    - Deleting Rare Words

We will focus on the Skip-Gram model

Embeddings capture relational meaning of words

vector('king') - vector('man') + vector('woman') $\approx$ vector('queen')



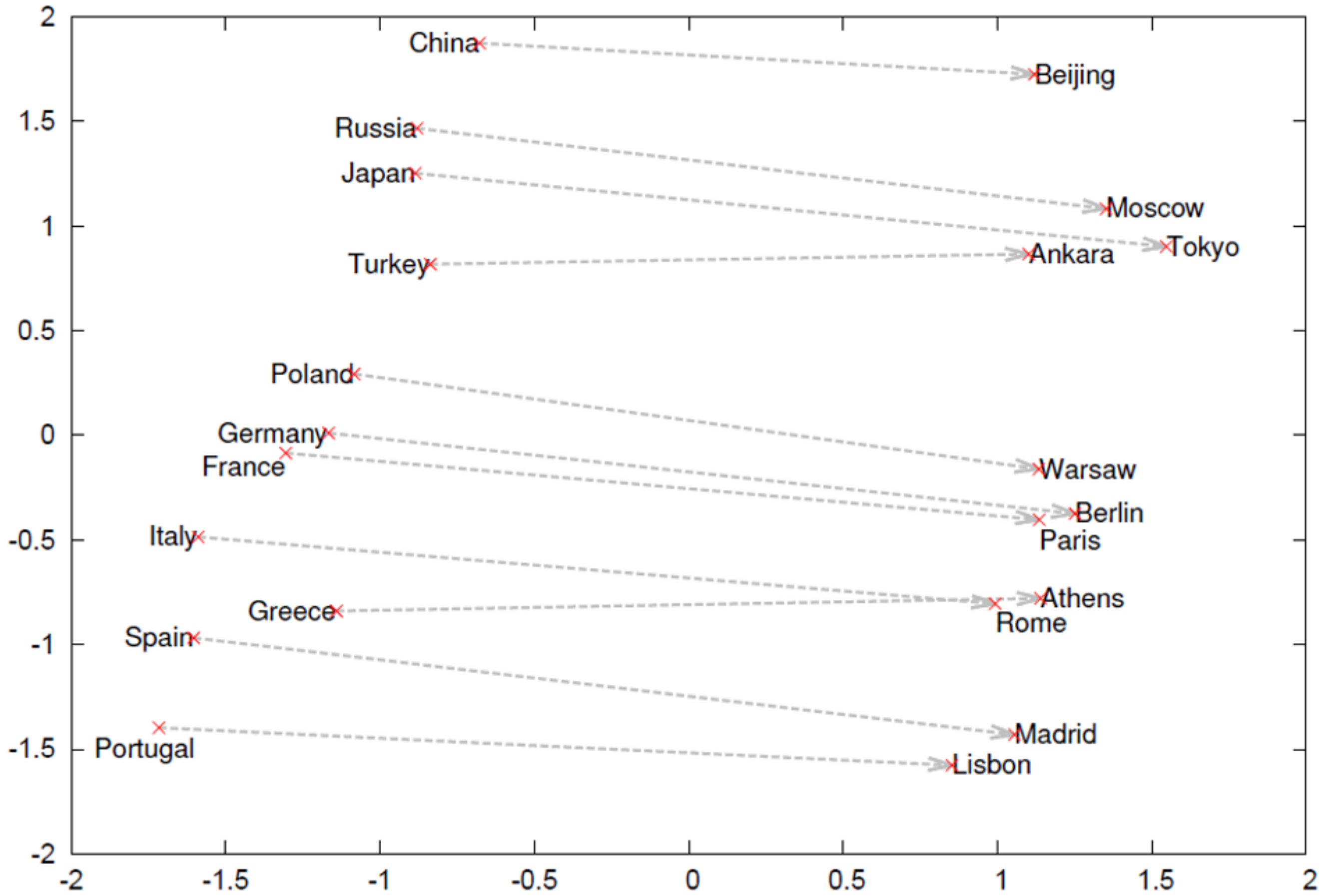


What we are seeing:

A projection of the 300-dimensional vector space into 2 dimensions (e.g., using PCA)

Slide by Dan Jurafsky

Embeddings capture relational meaning of words



vector('Paris') – vector('France') + vector('Italy') ≈ vector('Rome')

What we are seeing:

A projection of the 300-dimensional vector space into 2 dimensions
(e.g., using PCA)

How to learn such word embeddings?

How to learn such word embeddings ?

- Use context information !!
- Context of a word w ~ other words that frequently appear nearby w

...he curtains open and the moon shining in on the barely...

...ars and the cold , close moon " . And neither of the w...

...rough the night with the moon shining so brightly , it...

...made in the light of the moon . It all boils down , wr...

...surely under a crescent moon , thrilled by ice-white...

...sun , the seasons of the moon ? Home , alone , Jay pla...

...m is dazzling snow , the moon has risen full and cold...

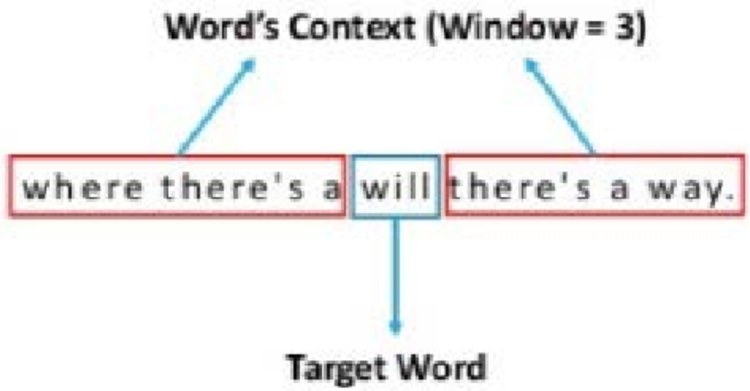
...un and the temple of the moon , driving out of the hug...

...in the dark and now the moon rises , full and amber a...

...bird on the shape of the moon over the trees in front...

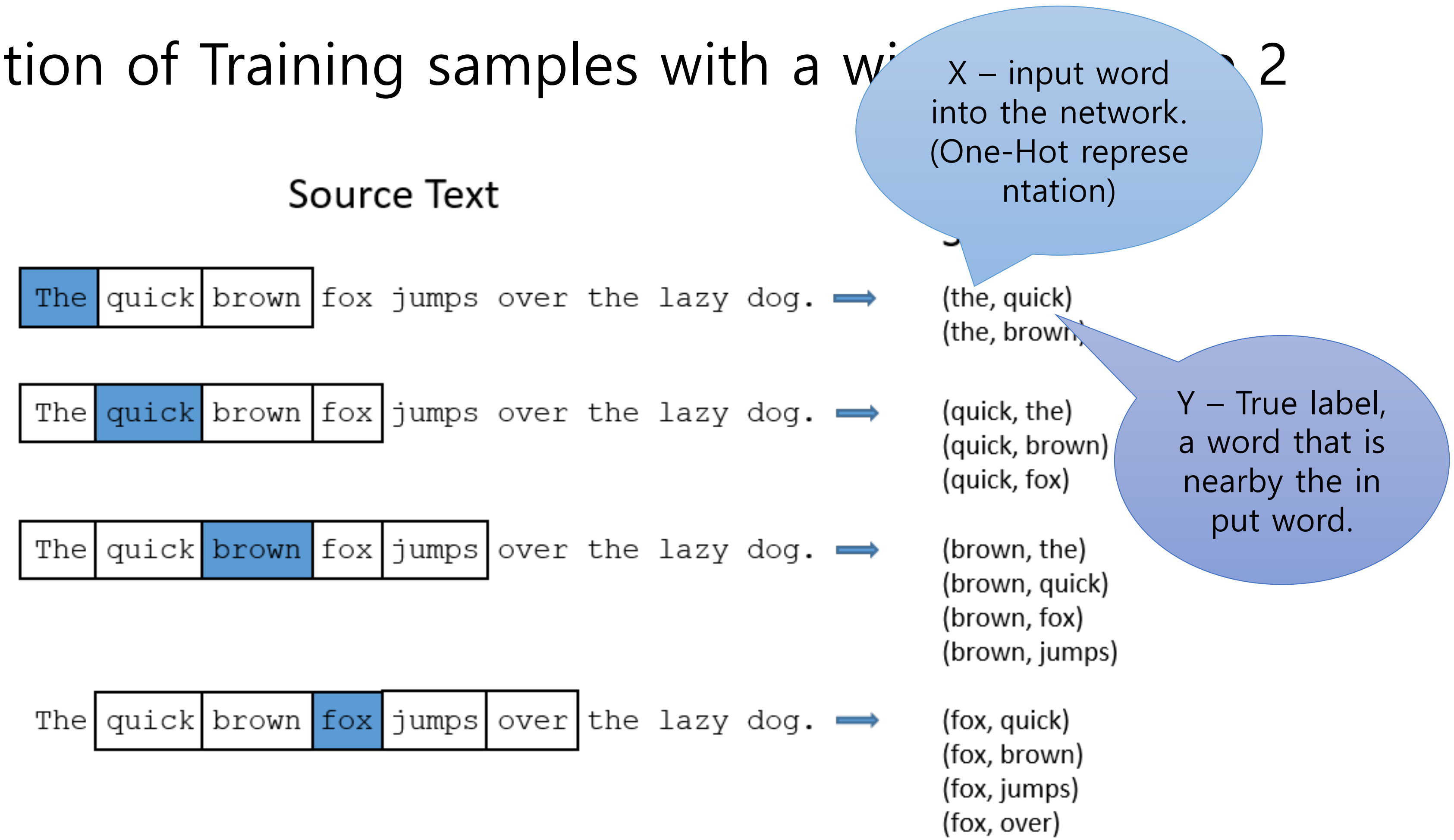
Main idea of Word2Vec

- Consider a local window of a target word



- Build a NN to predict neighbors of a target word

Collection of Training samples with a wi 2



Key insight: Use running text as implicitly supervised training data

Using the samples to train a NN

- From the training documents we first build the vocabulary (assume 10,000 unique words)
- Represent each word as a one-hot vector (of 10,000 dimensions)

- Given a word in the middle of a sentence (target word, that is input into the network), look at the words nearby and pick one at random

- The output of the network is a single vector (of 10,000 dimensions) containing, the probability for every word in our vocabulary of being the “nearby word” that we chose
- We will train the NN to predict this probability correctly by feeding it word pairs, found in our training documents

Effect of this training

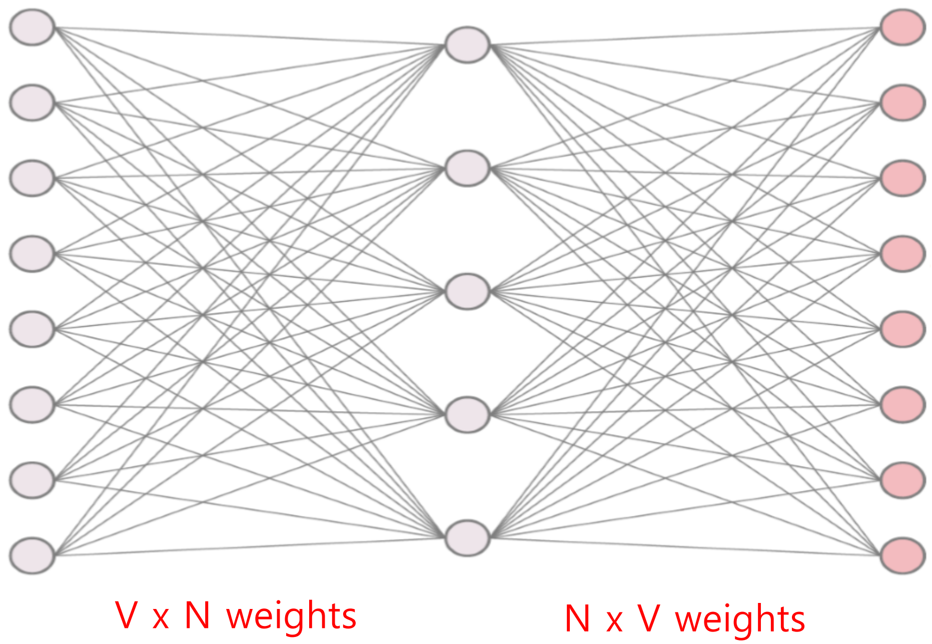
- The network is going to learn the statistics from the number of times each word-pair shows up
- E.g., the network is probably going to get many more training samples of (“Soviet”, “Union”) than of (“Soviet”, “Sasquatch”)

- When the training is finished:
  - if you give it the word “Soviet” as input, then it will output a much highe r probability for “Union” or “Russia” than it will for “Sasquatch”
  - The vectors for “Soviet” and “Russia” will be much more similar, than th e vectors for “Soviet” and “Sasquatch”

The word2vec skip-gram neural network

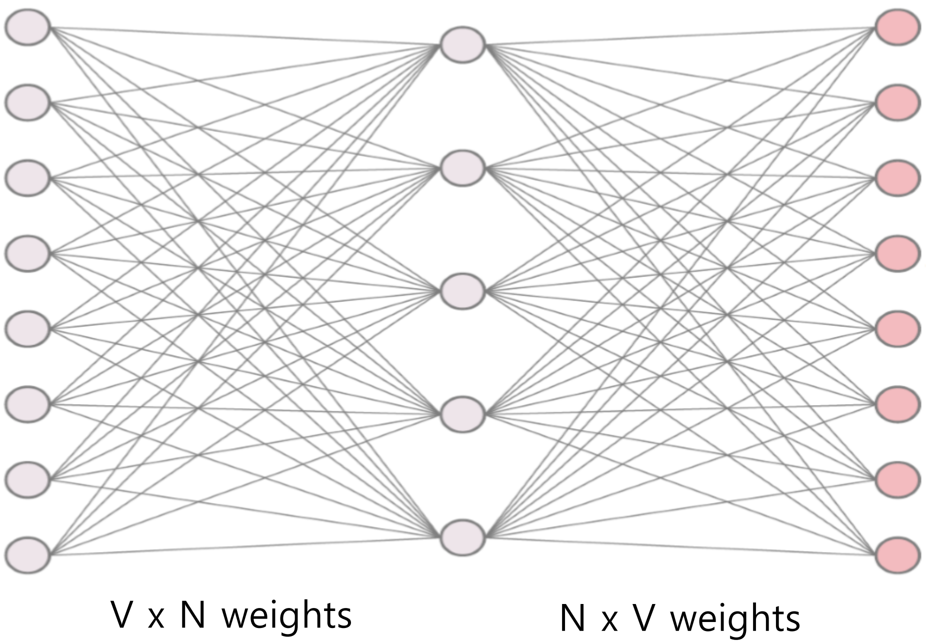
Skip-gram Architecture

- V (= vocabulary size) neurons in input layer
- V neurons in output layer
- One hidden layer containing N neurons (dimensions in the word embeddings that will be learned)
- Fully connected architecture



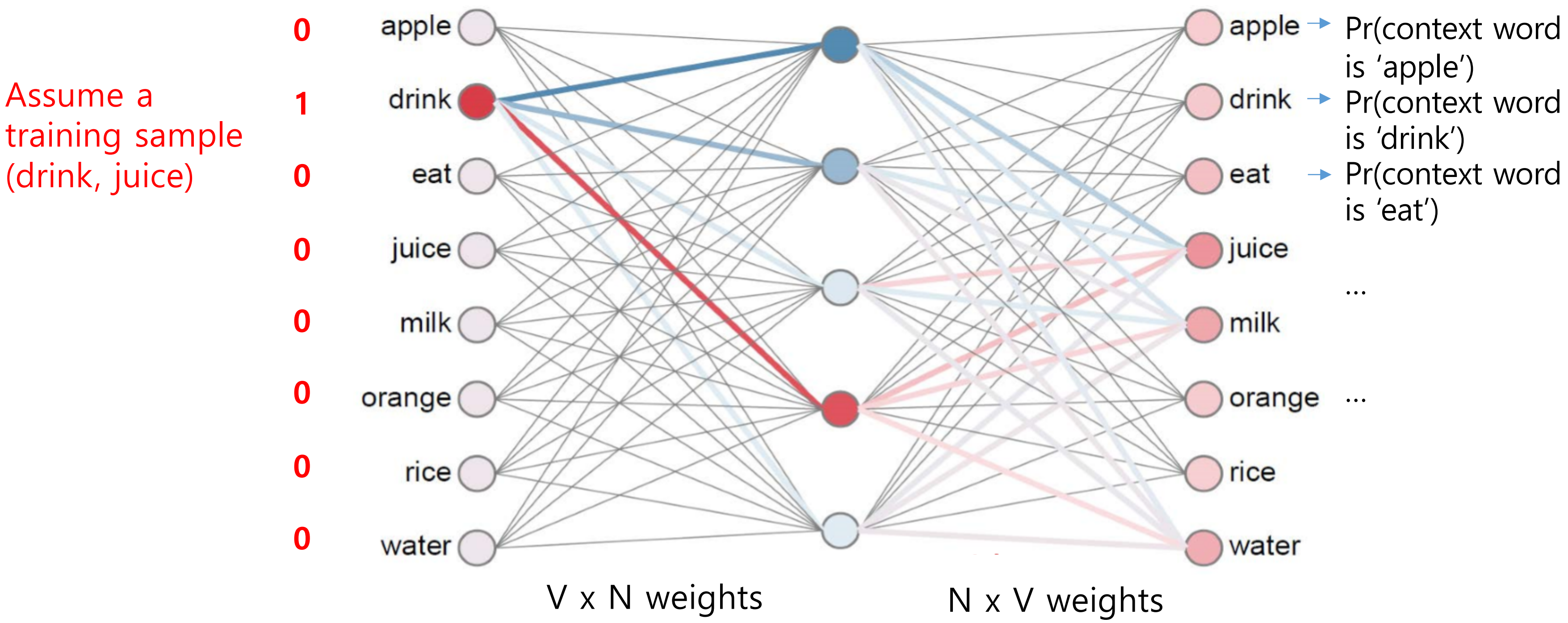
Skip-gram Architecture

- No activation function for hidden layer neurons (i.e., linear neurons)
- Output layer neurons use softmax activation (to be explained soon)



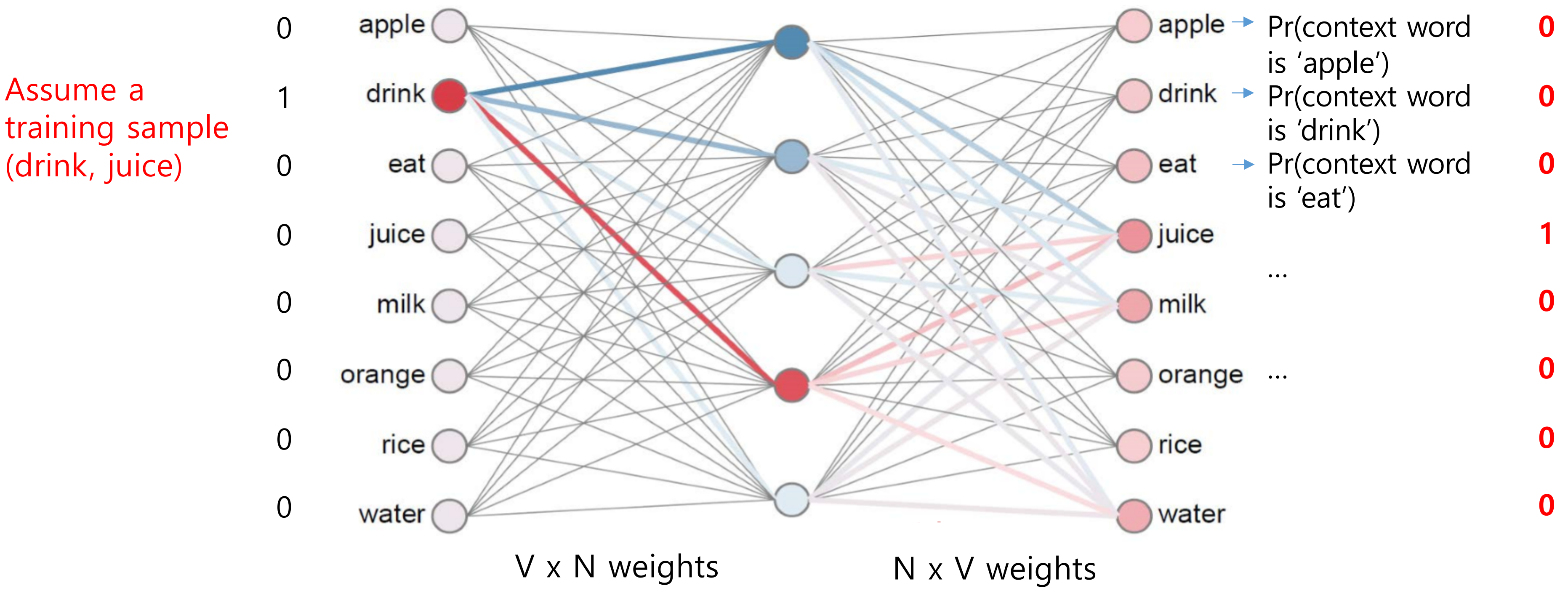
Skip-gram Architecture

- Given a one-hot vector of a target word as input
- Weights initialized randomly
- Network outputs probabilities of all words in the vocabulary being the context word



Skip-gram Architecture

- We know the actual context word (from the text)
- Adjust weights through backpropagation, to maximize the probability of the context word
- Repeat over many, many pairs … adjust weights to make the positive pairs more likely

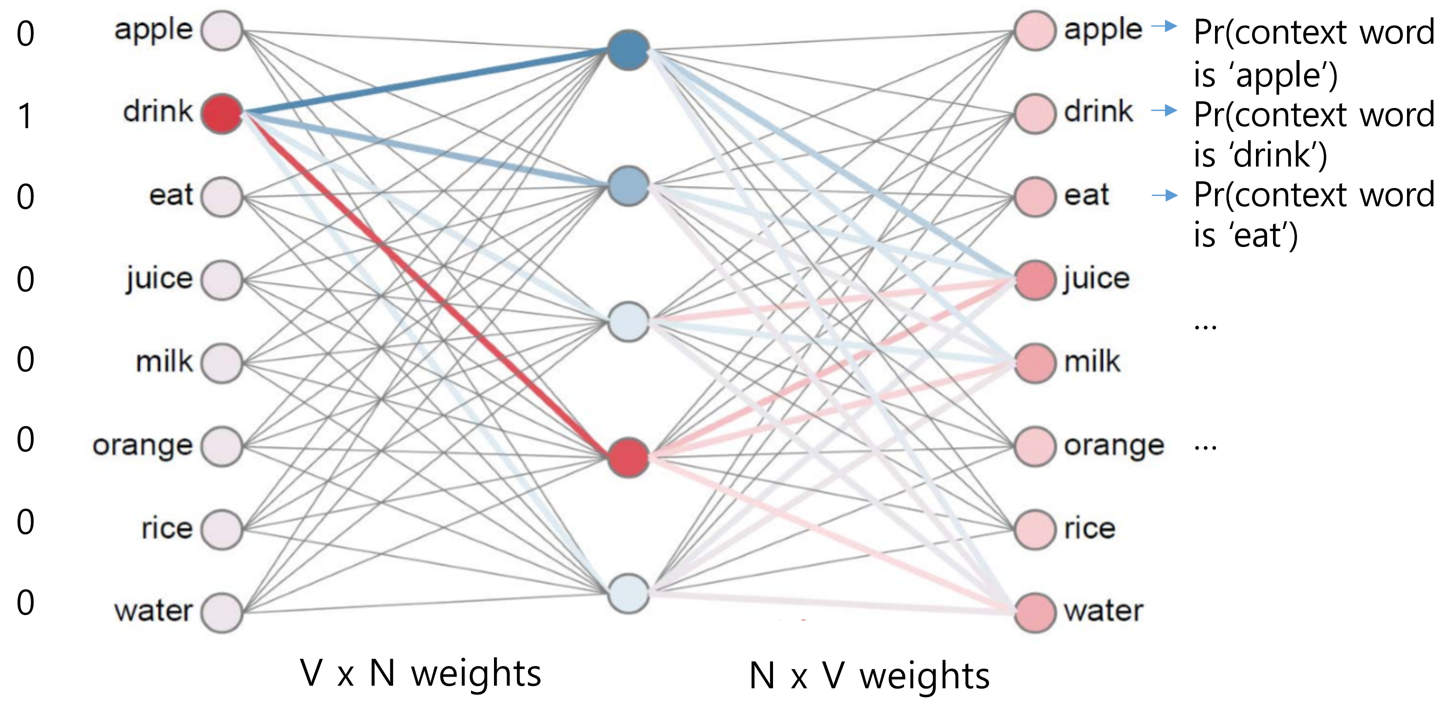


The input layer

The skip-gram neural network

- Input: one-hot vector of the target word ('drink' in our example)

Assume a training sample (drink, juice)

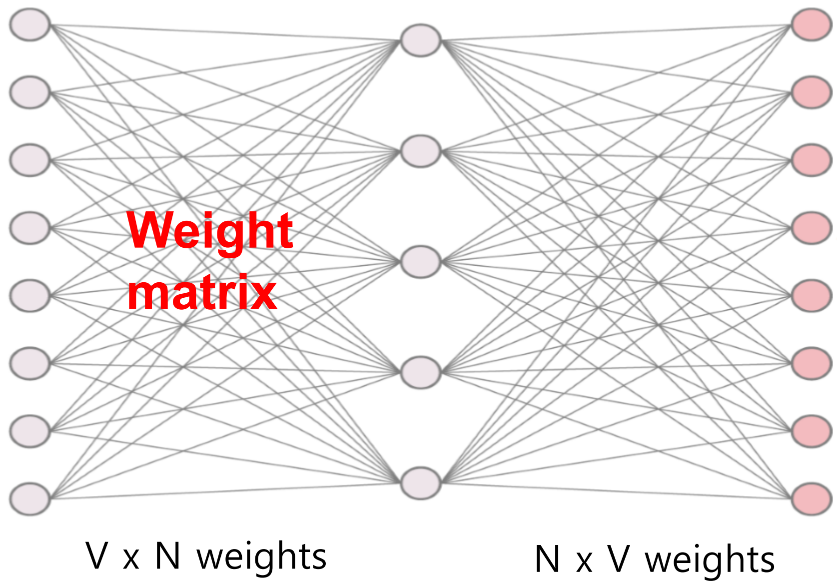


The hidden layer

The hidden layer

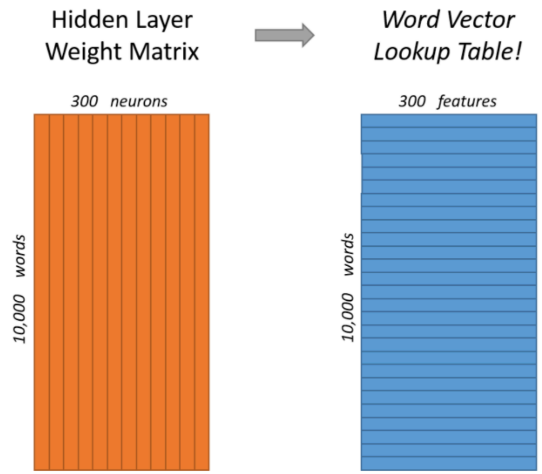
- Assume vocabulary size V = 10,000
- Number of neurons in hidden layer = dimension of the word embeddings = 300 (assume)

- We can represent all the hidden layer by a weight matrix with 10,000 rows (one for every word in our vocabulary) and 300 columns (one for each neuron in the hidden layer)



The hidden layer

- If you look at the rows of this weight matrix, these are actually what will be our word vectors:



- So the end goal is just to learn this hidden layer weight matrix

The hidden layer

- Multiplying a 1 x 10,000 one-hot vector by a 10,000 x 300 matrix, will effectively just select the matrix row corresponding to the "1"
- Hence the hidden layer of this model is really just operating as a lookup table
- The output of the hidden layer is just the word vector/embedding for the input word (e.g., 'drink' in the previous example)

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

The output layer

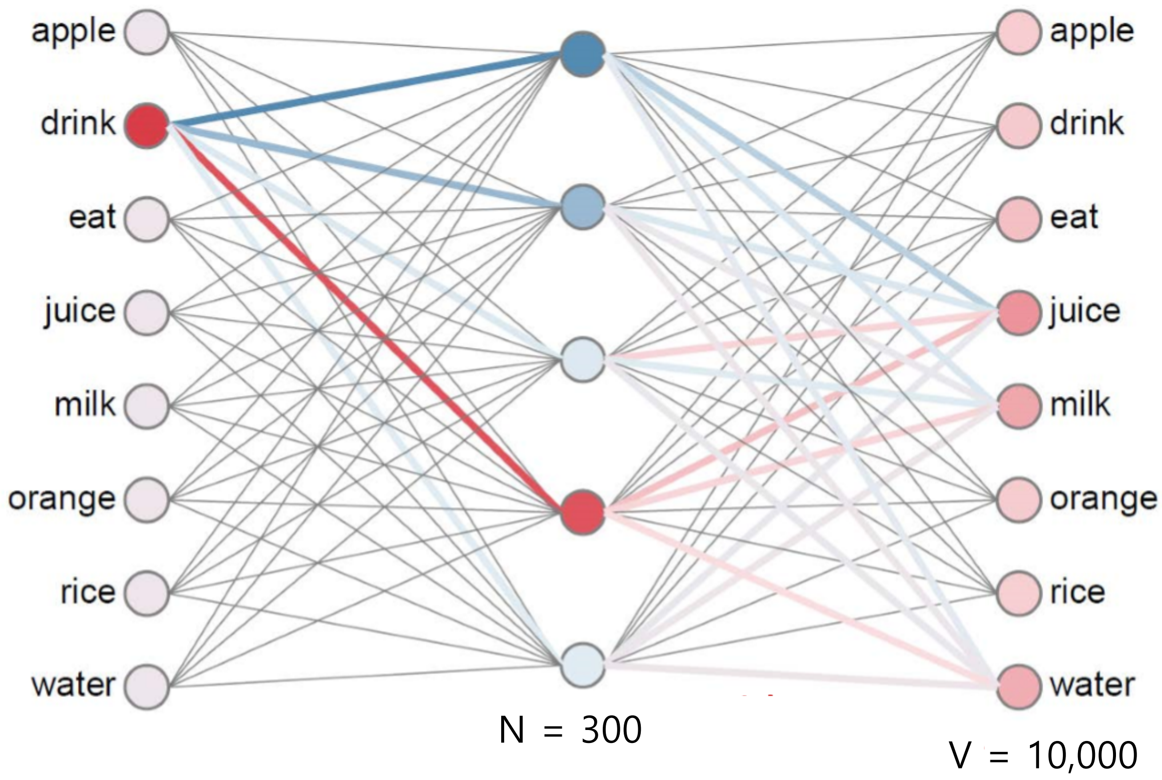
The output layer

- What gets fed to the output layer? The 1x300 word embedding of the target word (e.g., 'drink')
- Each output neuron (one per word in our vocabulary), will produce an output between 0 and 1; sum of all outputs add up to 1
- Softmax function
  - Given a set of numbers (may be negative, more than 1.0, …)
  - Convert them to probabilities

$$a \quad b \quad c \quad \rightarrow \quad \exp(a) \quad \exp(b) \quad \exp(c) \quad \rightarrow \quad \frac{\exp(a)}{\sum_{i=a,b,c} \exp(i)} \quad \frac{\exp(b)}{\sum_{i=a,b,c} \exp(i)} \quad \frac{\exp(c)}{\sum_{i=a,b,c} \exp(i)}$$

The output layer

Each output neuron has an associated
300-dimension weight vector



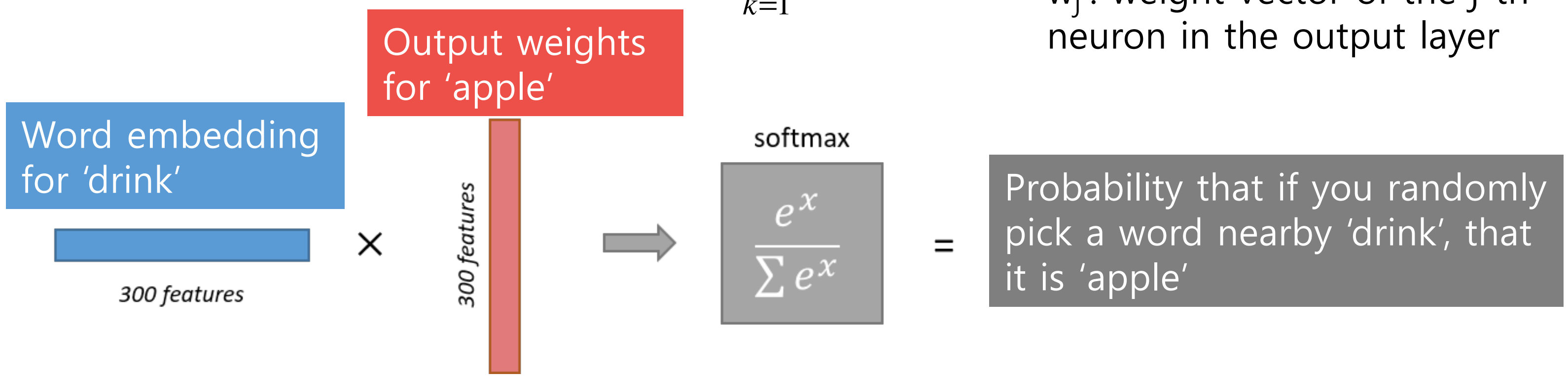
The output layer

- Each output neuron has a weight vector which it multiplies against the word embedding coming from the hidden layer
- Then it applies the function exp(x) to the result. Finally, divide this result by the sum of the results from all 10,000 output nodes

$$out_j = \frac{e^{x^T w_j}}{\sum_{k=1}^{10,000} e^{x^T w_k}}$$

x : word embedding of the input target word (coming from hidden layer)

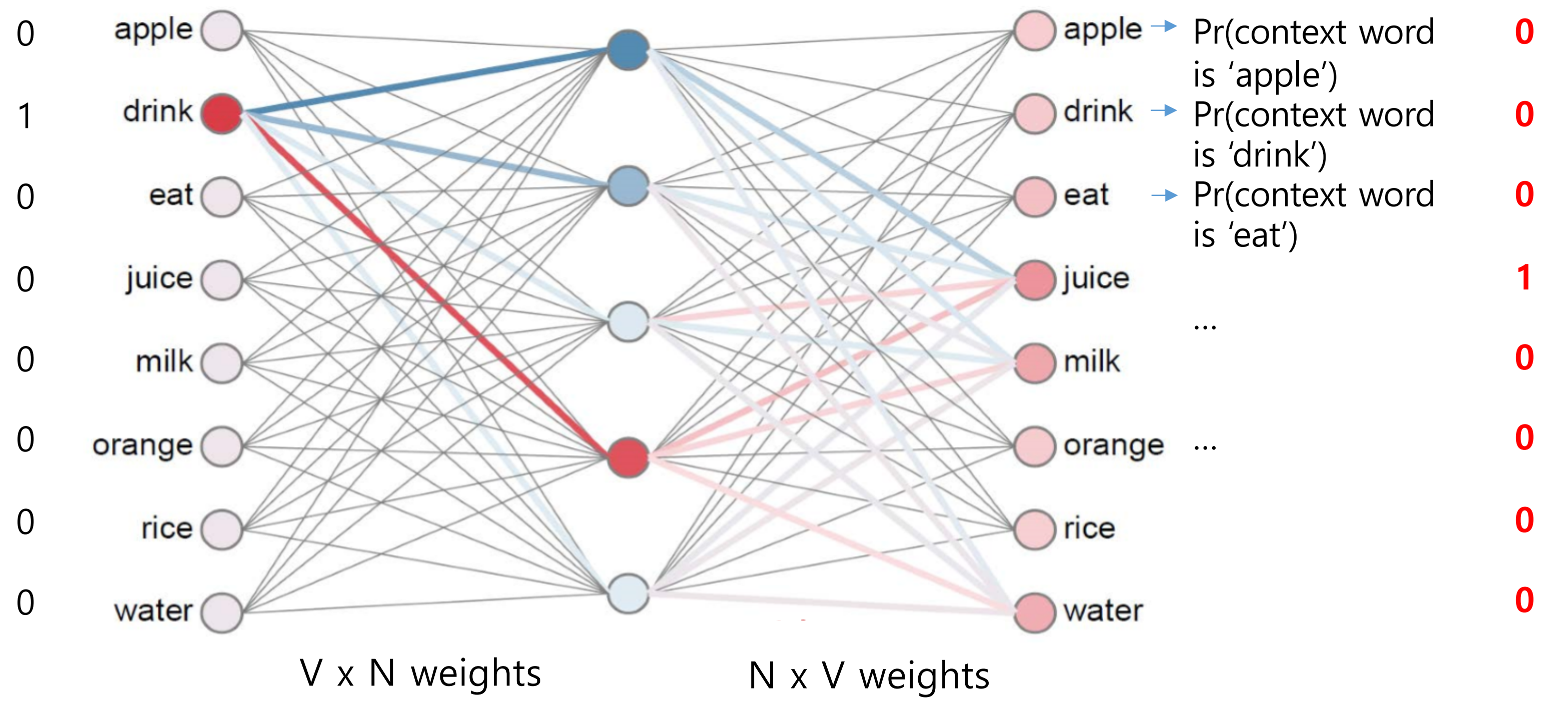
w_j : weight vector of the j-th neuron in the output layer



Using the output probabilities

- We know the actual context word (from the text)
- Adjust weights through backpropagation, to maximize the probability of the context word
- Repeat over many, many pairs … adjust weights to make the positive pairs more likely

Assume a training sample (drink, juice)



A point to note

- The task of predicting context words for a target word
- … is actually a dummy task, in which we are not interested

- Our interest is in learning the word representations

- Basically, we will just discard the output layer, and use the learned word embeddings

Word2Vec Objective/Cost Function

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T}\sum_{-m\le j\le m, j\ne 0} \log p(w_j|c)$$

$$p(w_j|c) = \frac{\exp\left(w_j^T c\right)}{\sum_{i=1}^{N}\exp(w_i^T c)}$$

Task: understand that the neural network we discussed is actually minimizing this cost function

Some optimizations

Problem with Word2vec network

- The skip-gram NN contains a huge number of weights
  - For our example with 300 features and a vocab of 10,000 words, that's 3 million weights in the hidden layer and output layer each!

- Running gradient descent on such a large network will be slow

- Also, need a huge amount of training data in order to tune that many weights and avoid over-fitting.

- The designers of Word2vec proposed 3 optimizations to make the training more efficient

Optimizations

(1) Treating common word-pairs or phrases as single words

- A word pair like "**Boston Globe**" (a newspaper) has a much different mea ning than the individual words "Boston" and "Globe"
- So treat "Boston Globe", wherever it occurs in the text, as a single word with its own word vector representation

(2) Subsampling frequent words to decrease the number of training samples

- "the" will appear in the context window of many different words
- Sub-sample words whose frequency in the corpus exceeds a threshold, e.g., do not include all training samples having 'the'

Optimizations

(3) Modifying the optimization objective with a technique called "Negative Sampling", which causes each training sample to update only a small percentage of the model's weights

- The idea
  - Training a NN means taking a training example and adjusting all weights slightly so that it predicts that training sample more accurately
  - Each training sample will tweak all of the weights in the NN
  - Negative sampling: have each training sample only modify a small fraction of the weights, rather than all of them

Negative sampling for Skip-gram

- When training the NN on the word pair ("drink", "juice"), the correct output of the network is a one-hot vector:
  - The output neuron corresponding to "juice" should output a 1, and *all* of the other thousands of output neurons should output a 0
  - A "negative" word is one for which network should output 0 and "positive" word is one for which network should output 1

- With negative sampling, randomly select just a small number of "negative" words (let's say 5) for which to update the weights
- We will also update the weights for our "positive" word ("juice")
- So just update the weights for the positive word, plus the weights for 5 other words that we want to output 0. That's updating a total of 1,800 weights corr esponding to 6 output neurons (instead of 3 million weights)

How to select the negative samples?

- The "negative samples" (the 5 words that we'll train to output 0) are chosen using a "unigram distribution"
  - The probability for selecting a word as a negative sample is related to its frequency, with more frequent words being more likely to be selected as negative samples.
  - Each word is given a weight equal to its **frequency** (word count) raised to the 3/4 power. Probability for a selecting word w_i is its weight divided by the sum of weights for all words:

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

- The decision to raise the frequency to the 3/4 power appears to be empirical. In the original paper, the authors say it outperformed other functions.
- The power makes less frequent words be sampled more often.

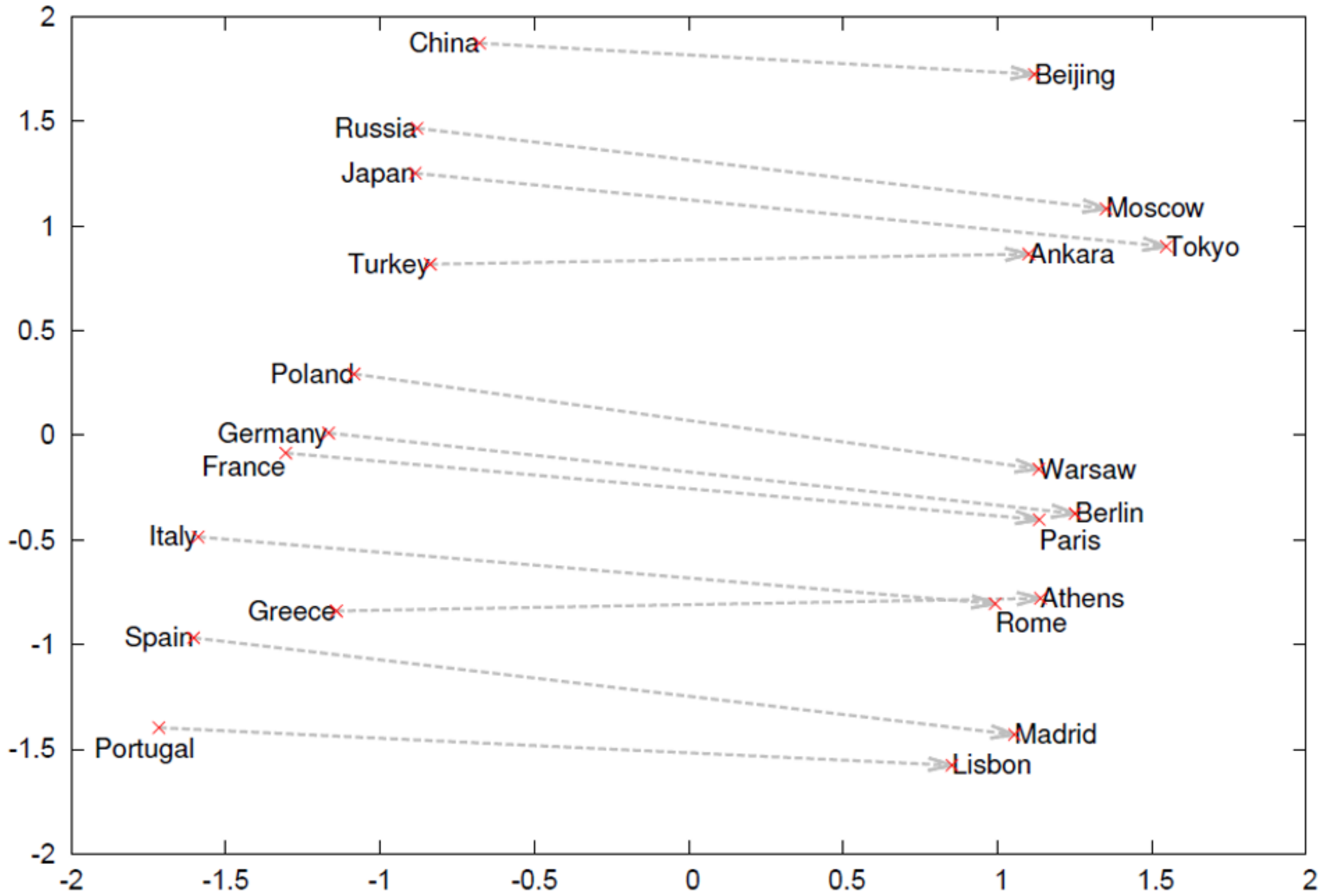
Another architecture of Word2vec

- Continuous bag of words – CBoW

- Skip-gram
  - From a central target word, predict a nearby context word
- CBoW
  - From the context words, predict the central target word

- See from papers (given on course website)

Many applications of Word2vec embeddings in NLP, IR



vector('Paris') – vector('France') + vector('Italy') ≈ vector('Rome')

What we are seeing:

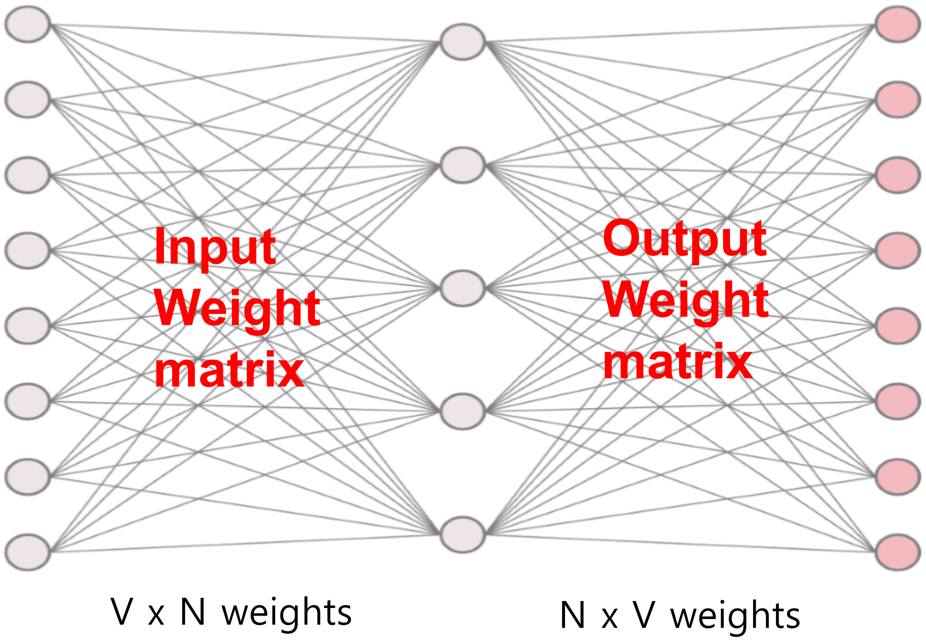
A projection of the 300-dimensional vector space into 2 dimensions (e.g., using PCA)

Improvements by using two vectors

For each word w, we actually get two vectors, each of N dimensions

One vector $v_{in}(w)$ from the input weight matrix (that we discussed); another vector $v_{out}(w)$ from the transpose of the output weight matrix

Subsequent research showed that it is usually better to consider the average of the two vectors as the representation of w



Thank you

Questions can be mailed to Dr. S. Ghosh (saptarshi@cse.iitkgp.ac.in)