Word embeddings

Some slides & images taken from various presentations available on the Web

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

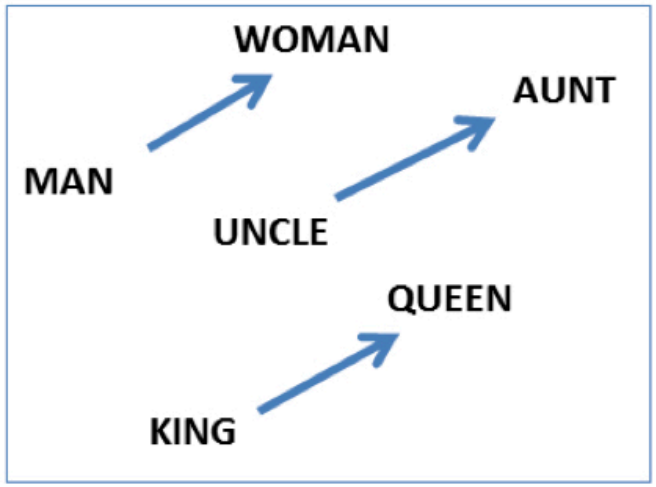
- A learned representation for text where
  - Every word represented by a low-dimensional dense vector
  - Words that have the same meaning, have a similar representation

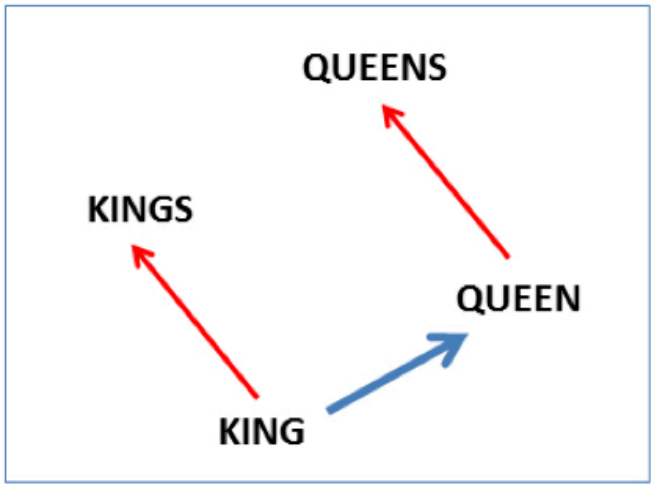
- Representations of similar words (e.g., ‘motel’ and ‘hotel’) would be similar (will have similar values in every dimension)
  - E.g., motel = [1.3, -1.4, 4.1] and hotel = [1.2, -1.5, 4.6]

- Typical number of dimensions: 300 -- 1000

Embeddings capture relational meaning of words

vector('king') - vector('man') + vector('woman') ≈ 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

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...

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

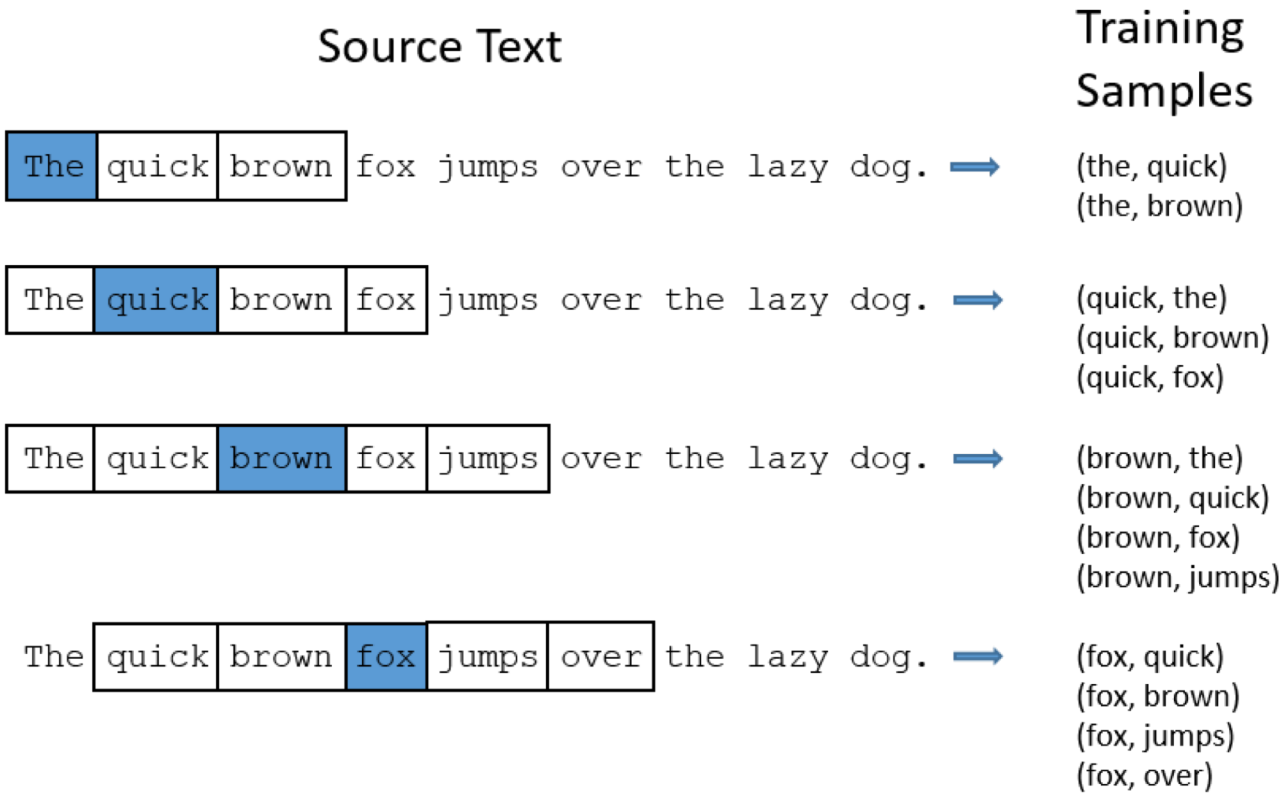
We will focus on the Skip-Gram model

Main idea of Word2Vec

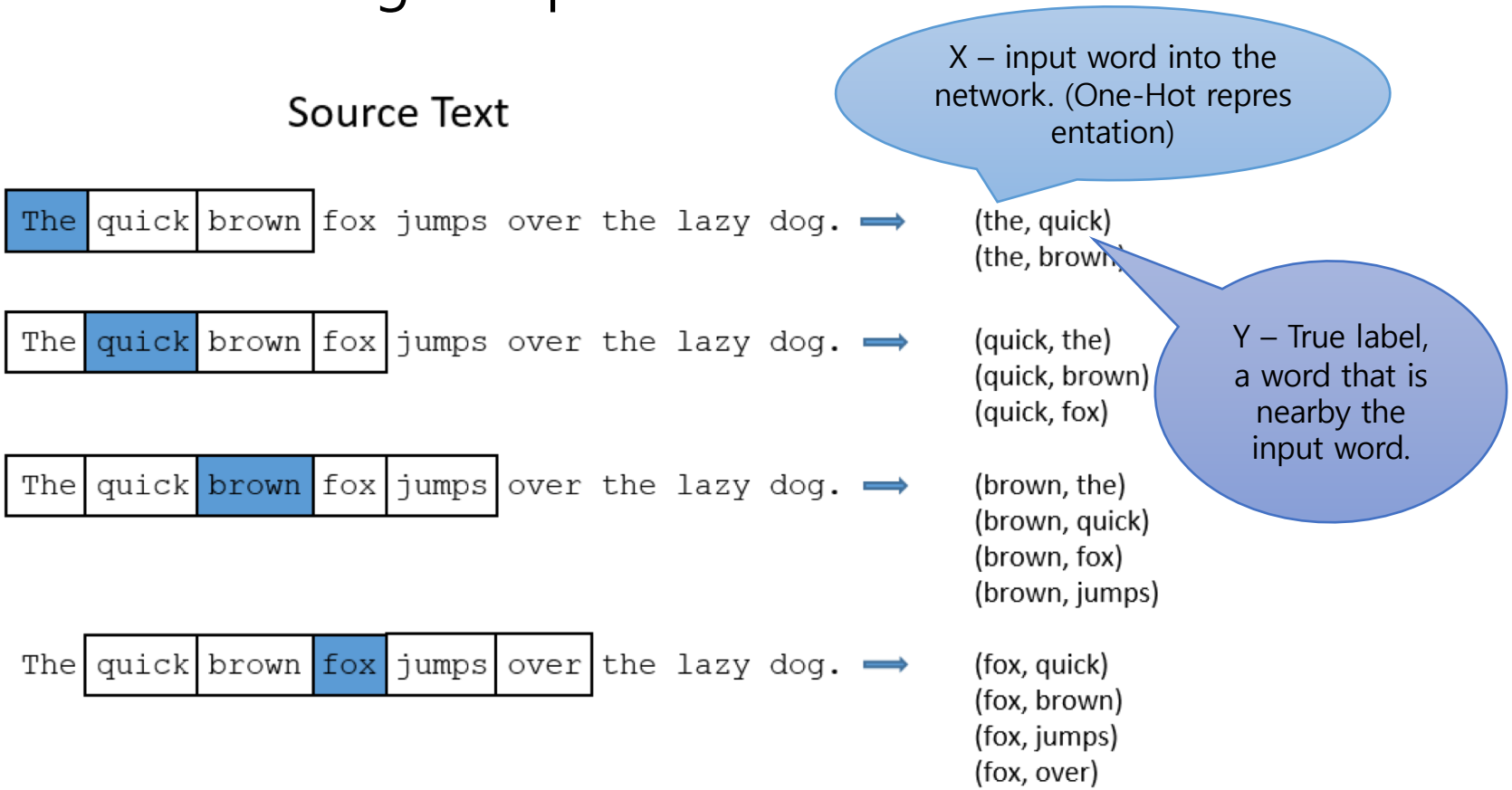
- Consider a local window of a target word

- Build a NN to predict neighbors of a target word

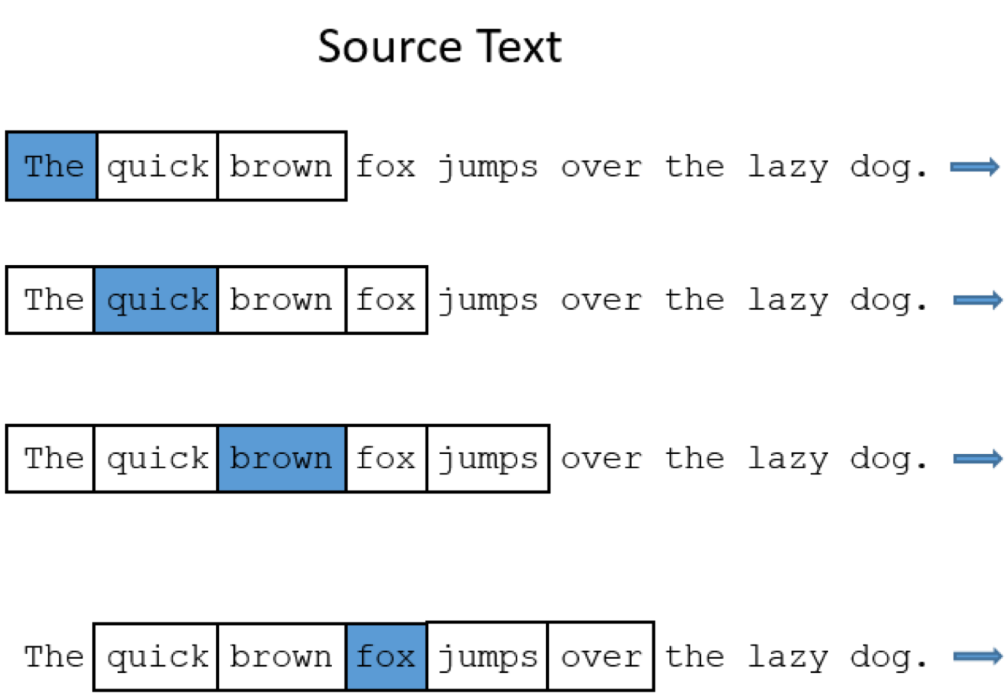
Collection of Training samples with a window of size 2



Collection of Training samples with a window of size 2



Collection of Training samples with a window of size 2



Possible variation:

For input 'brown', consider the label as the sum of all nearby context words 'the', 'quick', 'fox', 'jumps'

Key idea: Use running text as implicitly supervised training data

Advantage: Huge training data with minimal effort

This idea has helped development of many modern pre-trained models including LLMs

Using the samples to train a NN

- Build the vocabulary V from training corpus (assume 10K distinct words)
- Represent each word *w* as a one-hot vector v_w (of 10K dimensions)

- Given a word *w* (target word, whose one-hot vector v_w is input into the N N), look at the words nearby and pick one at random → very large set of word-pairs

- Output of the NN is a single vector (of 10K dimensions) containing the probability for every word in V of being the "nearby word" that we chose
- Train the NN to predict this probability correctly by feeding it word-pairs from our training corpus

Effect of this training

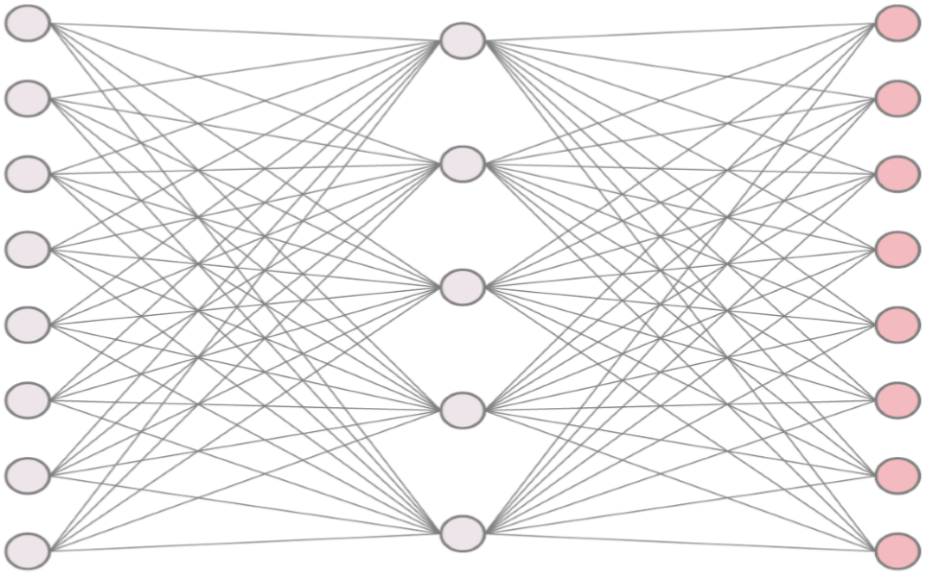
- The NN is going to learn the statistics from the number of times each word-pair shows up
- E.g., the NN is probably going to get many more training samples of (“Soviet”, “Union”) than of (“Soviet”, “Sasquatch”)

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

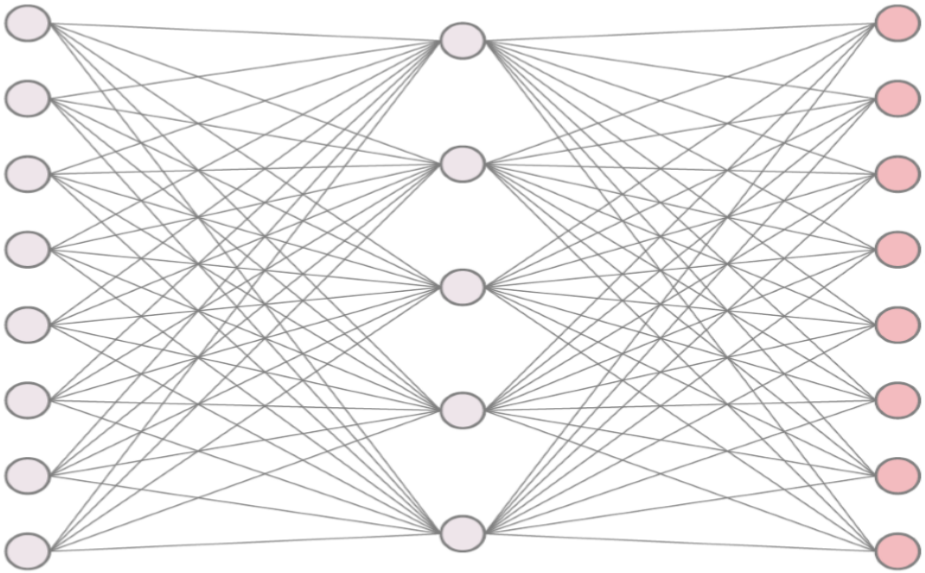


V x N weights

N x V weights

Skip-gram Architecture

- No activation function for hidden layer neurons (i.e., linear neurons)
- Output layer neurons use softmax activation (to ensure that the output values constitute a probability distribution; details to be explained soon)



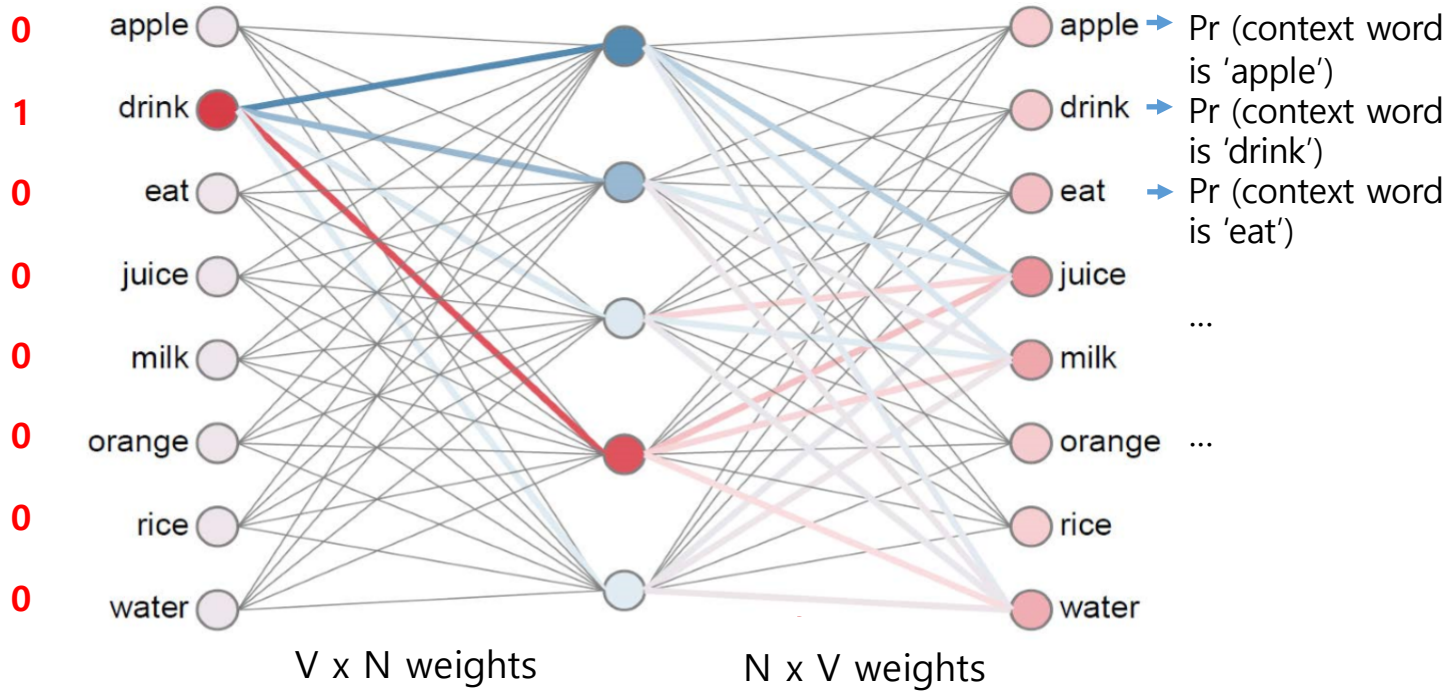
V x N weights

N x V weights

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 (Output is a conditional probability vector)

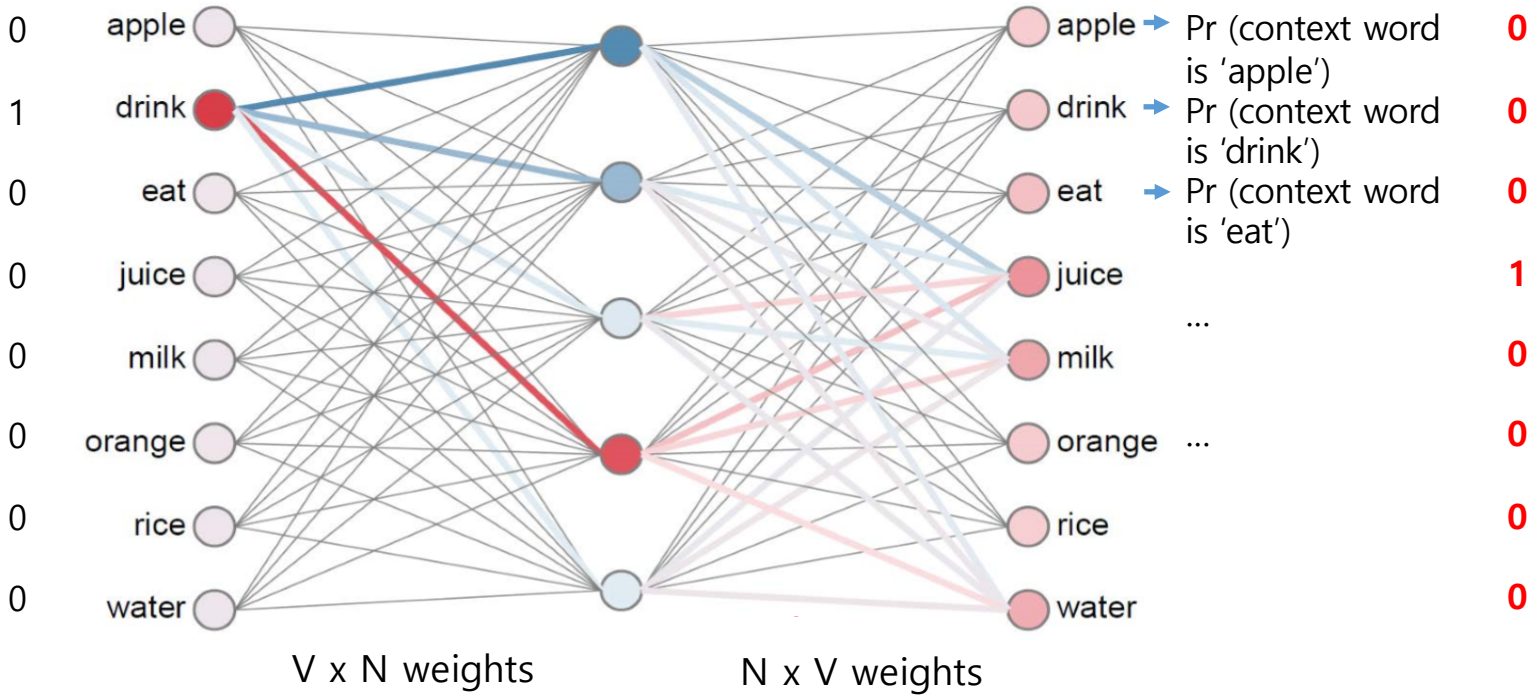
Assume a training sample (drink, juice)



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

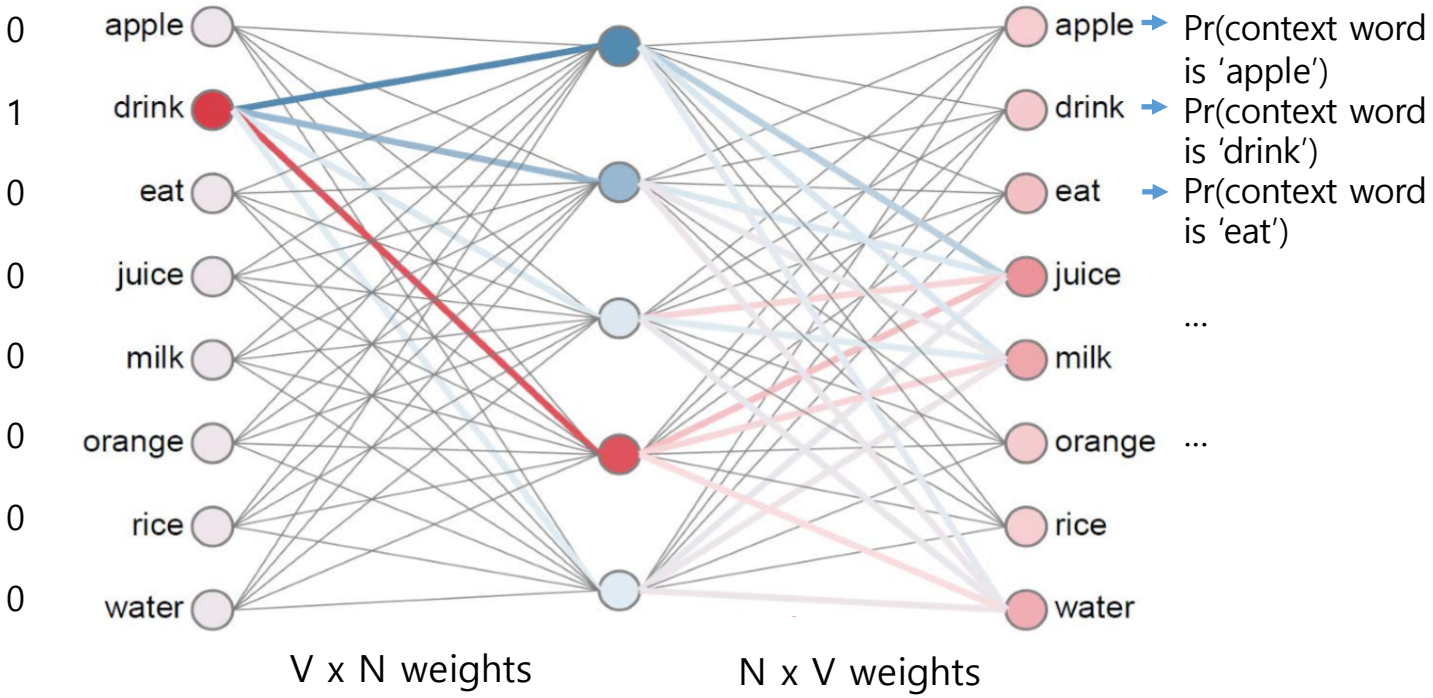
Assume a training sample (drink, juice)



The input layer

- Input: one-hot vector of the target word ('drink' in our example)
- Number of neurons = V (vocabulary size)

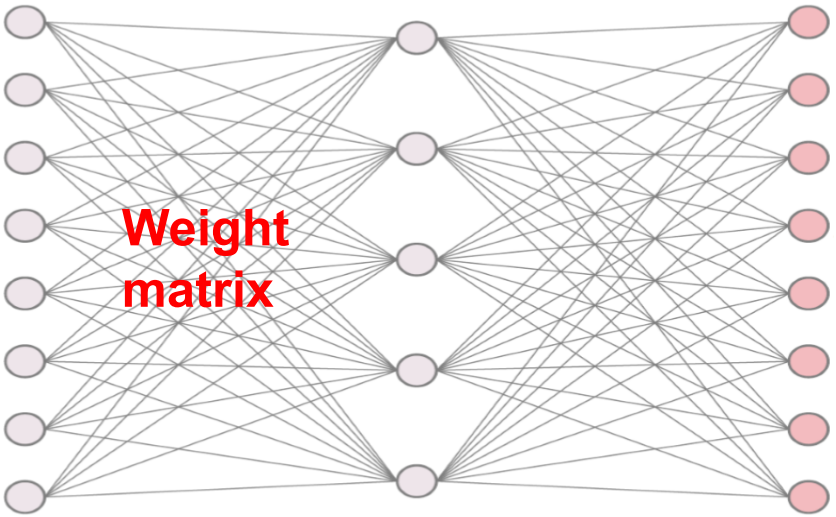
Assume a training sample (drink, juice)



The hidden layer

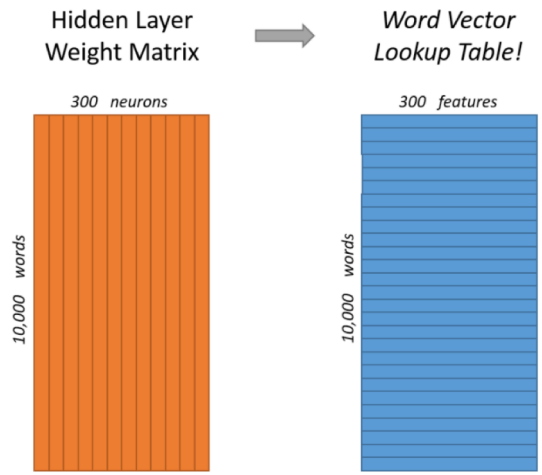
- Assume vocabulary size V = 10,000
- Number of neurons in hidden layer = dimension of the word embeddings = 300 (assume)

- We can represent the hidden layer by a weight matrix with 10,000 rows (one for every word in V) 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 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)

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

The output layer

- What gets fed to the output layer? The 1x300 word embedding of the target / input 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 should add up to 1
- Softmax function
  - Given a set of numbers (may be negative, more than 1.0, …)
  - Convert them to probabilities

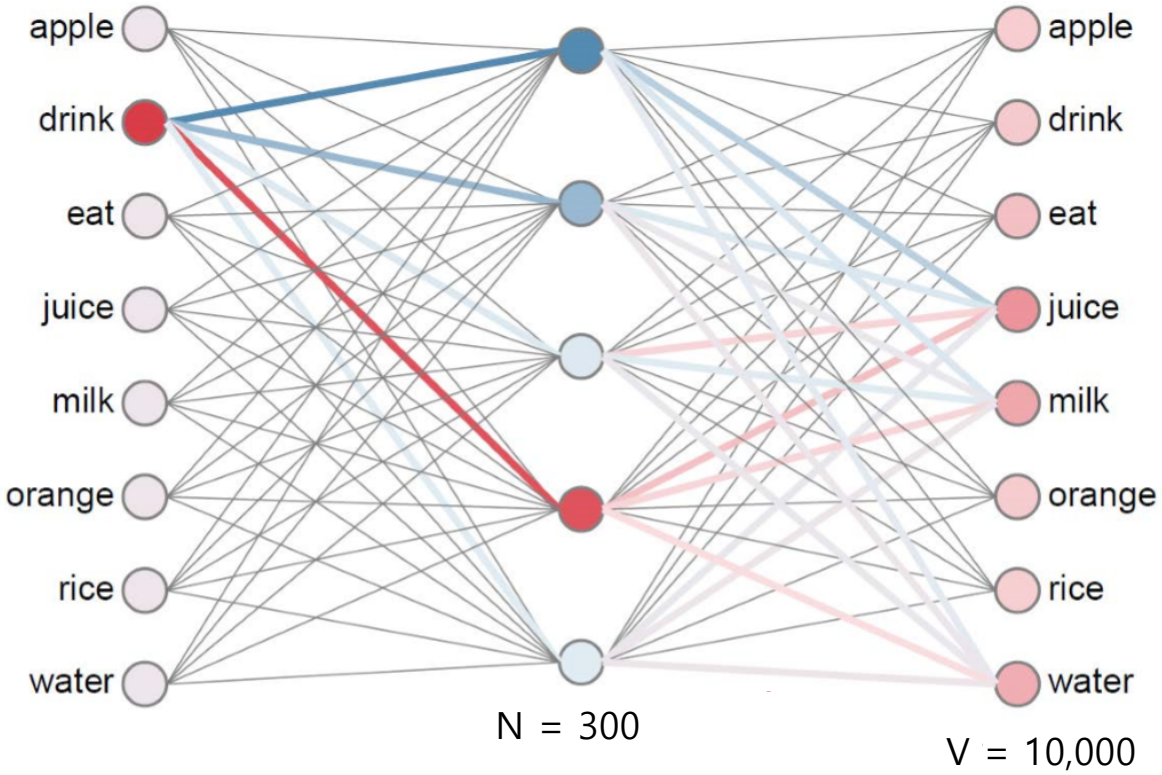
$$[a \;\; b \;\; c] \rightarrow \left[\frac{\exp(a)}{\sum_{i=a,b,c} \exp(i)} \;\; \frac{\exp(b)}{\sum_{i=a,b,c} \exp(i)} \;\; \frac{\exp(c)}{\sum_{i=a,b,c} \exp(i)} \right]$$

Softmax

$$softmax(z = \begin{bmatrix} z_1 & \cdots & z_n \end{bmatrix}^T) = \begin{bmatrix} \frac{exp(z_1)}{\sum_{i=1}^{n} exp(z_i)} & \cdots & \frac{exp(z_n)}{\sum_{i=1}^{n} exp(z_i)} \end{bmatrix}^T$$

The output layer

Each output neuron has an associated 300-dimension weight vector



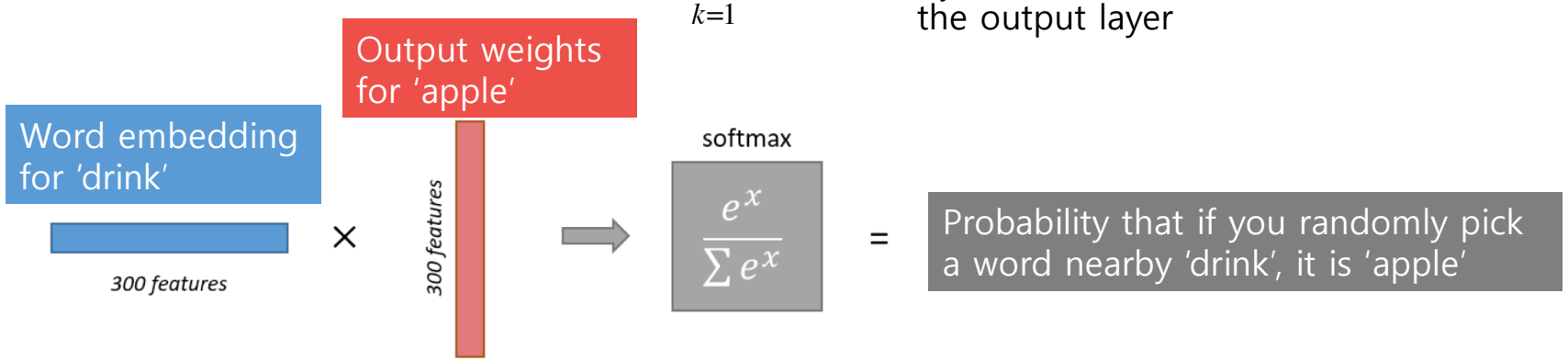
The output layer

- 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)

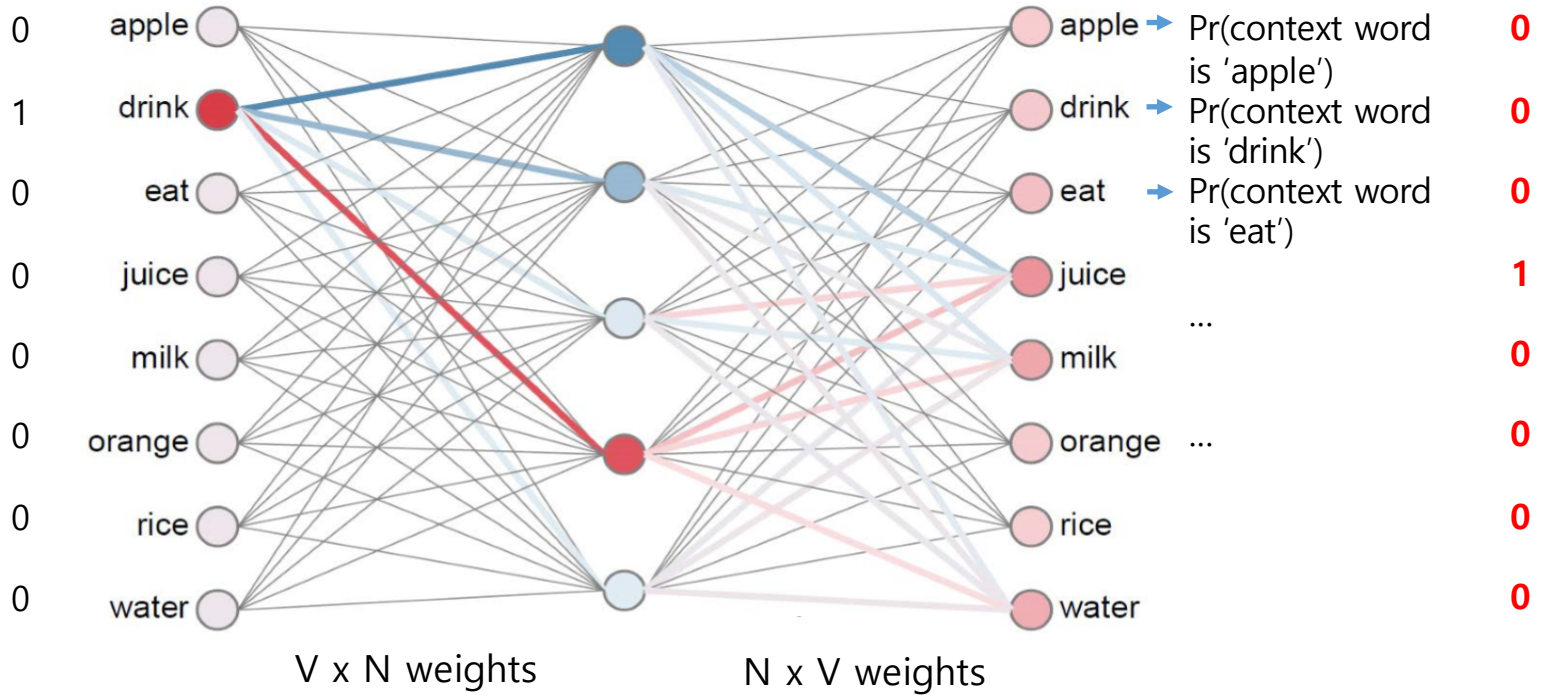
w_j : weight vector of the j-th neuron in the output layer



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
- Cross entropy loss function
- Repeat over many, many pairs … adjust weights to make the positive pairs more likely

Assume a training sample (drink, juice)

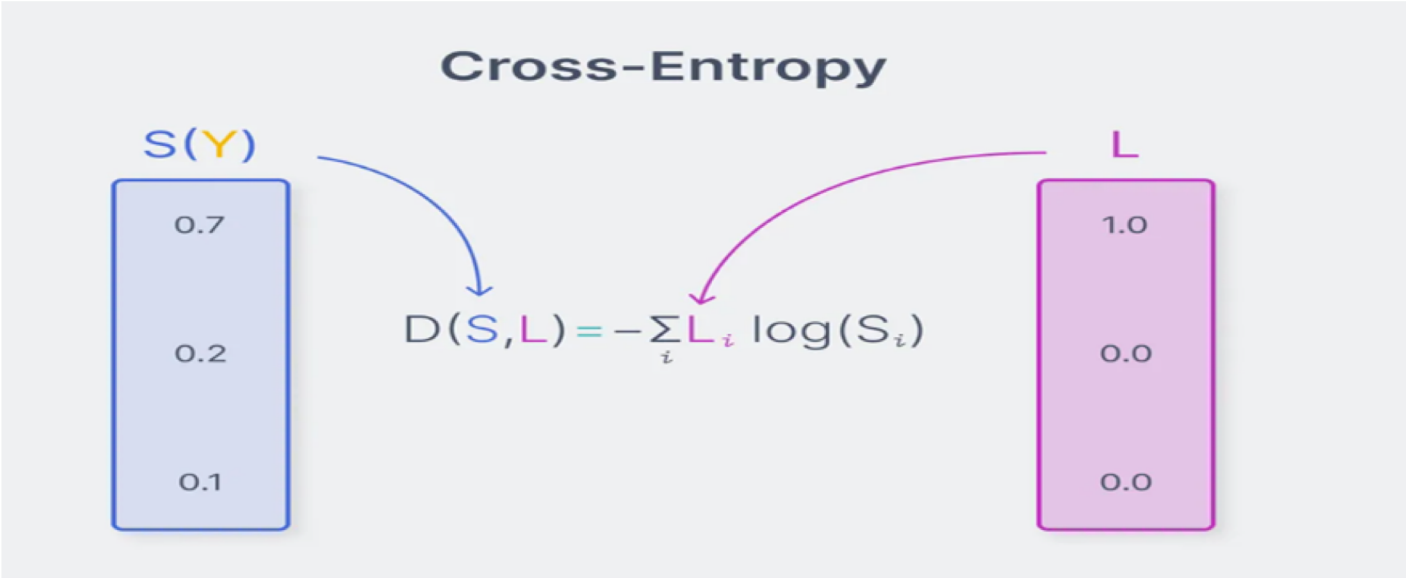


The loss function: Cross Entropy

- Cross entropy is a loss function used to measure the distance betw een two probability distributions
- Returns zero if two distributions match exactly, and higher values if the two diverge

- For classification tasks, we use cross entropy to compare:
  - the probability distribution produced by the neural network (output of softma x)
  - the "target" probability distribution where the probability of the correct class i s 1.0 and everything else is 0.0

The loss function: Cross Entropy



Model's predicted probability distribution

True probability distribution (one-hot)

Some points 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

Some points to note

- No activation function for hidden layer neurons, but output neurons use softmax
- Once training is completed, the weights will be fixed (to the values that have been learned)
- When evaluating / applying the trained network on an input word, the output vector will be a word vector (i.e., a vector of floating point values, *not* a one-hot vector).

Word2Vec (skip-gram) Objective Function

c – central word vector
w_j – context word vector
m – context size

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

Cross-entropy loss (averaged over the corpus)

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

Probability that $\mathbf{w_j}$ is in the context of c (softmax of their vector product)

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

Optimizations for Word2vec

Problem with Word2vec network

- The skip-gram NN contains a huge number of weights
  - With 300 features and a vocabulary of 10,000 words, 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 meaning than the individual words "Boston" and "Globe"
- So treat "Boston Globe", wherever it occurs in the text, as a single word with one word vector representation

(2) Subsampling frequent words to decrease nos. 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
  - Ideally, each training sample tweaks all the weights in the NN
  - Negative sampling: each training sample will modify only a small fraction of the weights

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 1, and *all* the other 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
- 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 corresponding to 6 output neurons (instead of 3 million weights)

The original Word2vec paper says – for smaller datasets, 5-20 negative samples work well. For large datasets, 2-5 negative samples may be sufficient.

How to select the negative samples?

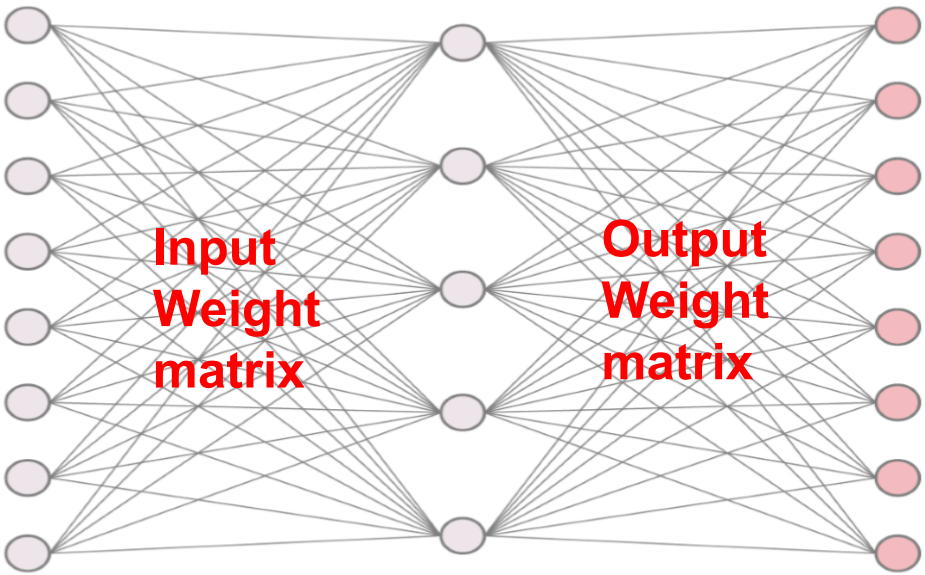
- The “negative samples” (the words that we’ll train to output 0) are chosen using a “unigram distribution”
  - The probability for selecting a word as a negative sample is related to its frequency, with more frequent words being more likely to be selected as negative samples.
  - Each word is given 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:

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

- 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.

Improvements by using two vectors

For each word w in the vocabulary, Word2vec actually learns 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



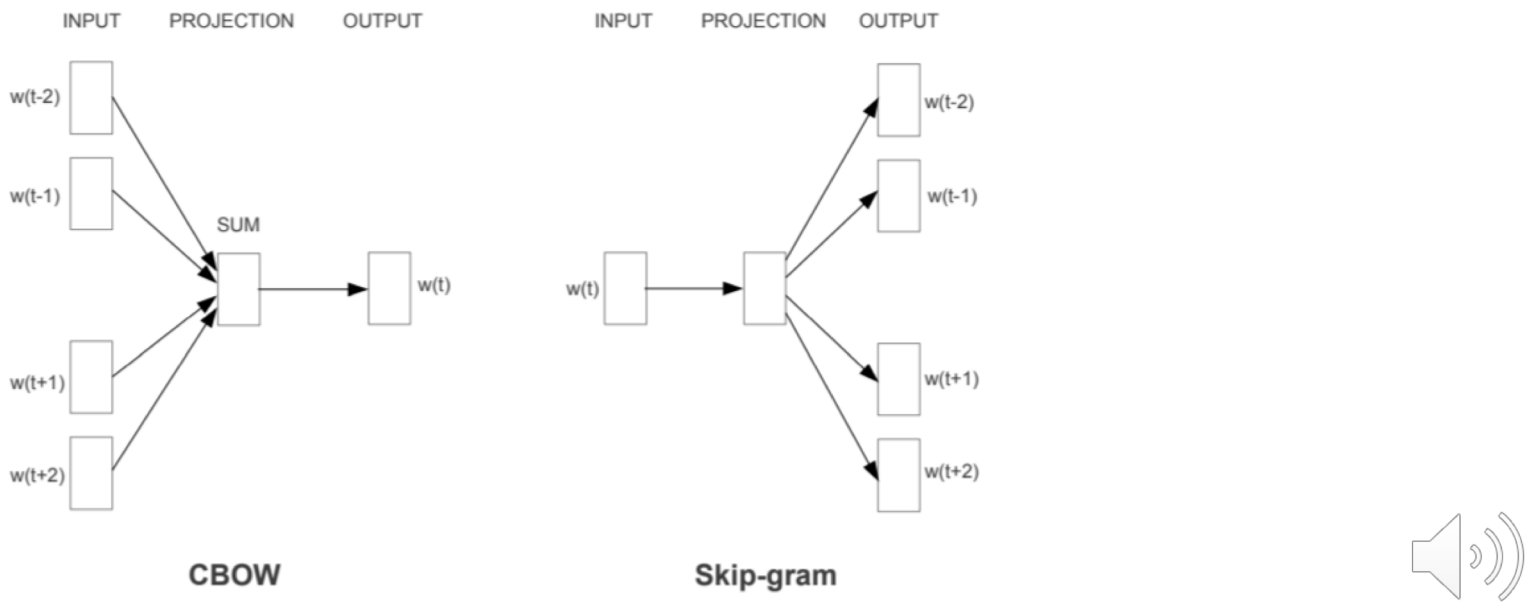
For each word, 2 vectors learned:
- Centre word vector
- Context word vector

Centre word vectors generally used

Subsequent research: it is usually better to consider the average of the two vectors as the word repres entation

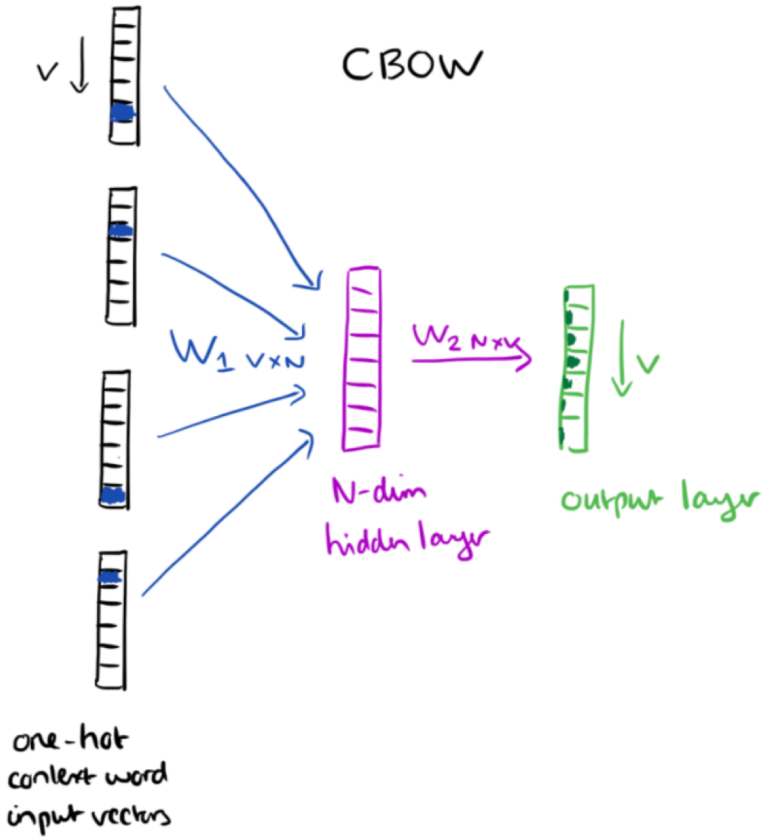
A variation of Word2vec - CBoW

- Continuous bag of words – CBoW
- Skip-gram - From a central target word, predict nearby context word(s)
- CBoW - From the context words, predict the central target word



CBoW

The context words form the input layer. Each word is encoded in one-hot form.
A single hidden and output layer.



Training objective: maximize the conditional probability of observing the actual output word given the input context words

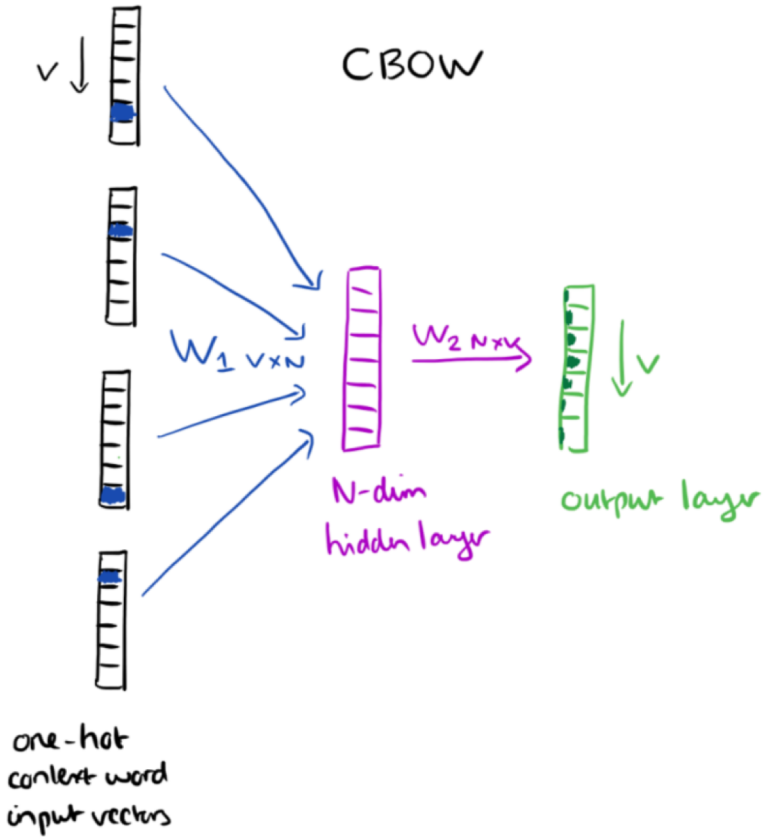
Since input vectors are one-hot, multiplying an input vector by W_1 amounts to simply selecting a row from W_1

Given C input word vectors, the activation function for the hidden layer amounts to summing the corresponding 'hot' rows in W_1 and dividing by C to take the average

CBoW

The context words form the input layer. Each word is encoded in one-hot form.

A single hidden and output layer.



From the hidden layer to the output layer, the second weight matrix W_2 can be used to compute a score for each word in the vocabulary, and softmax can be used to obtain the posterior distribution of words.

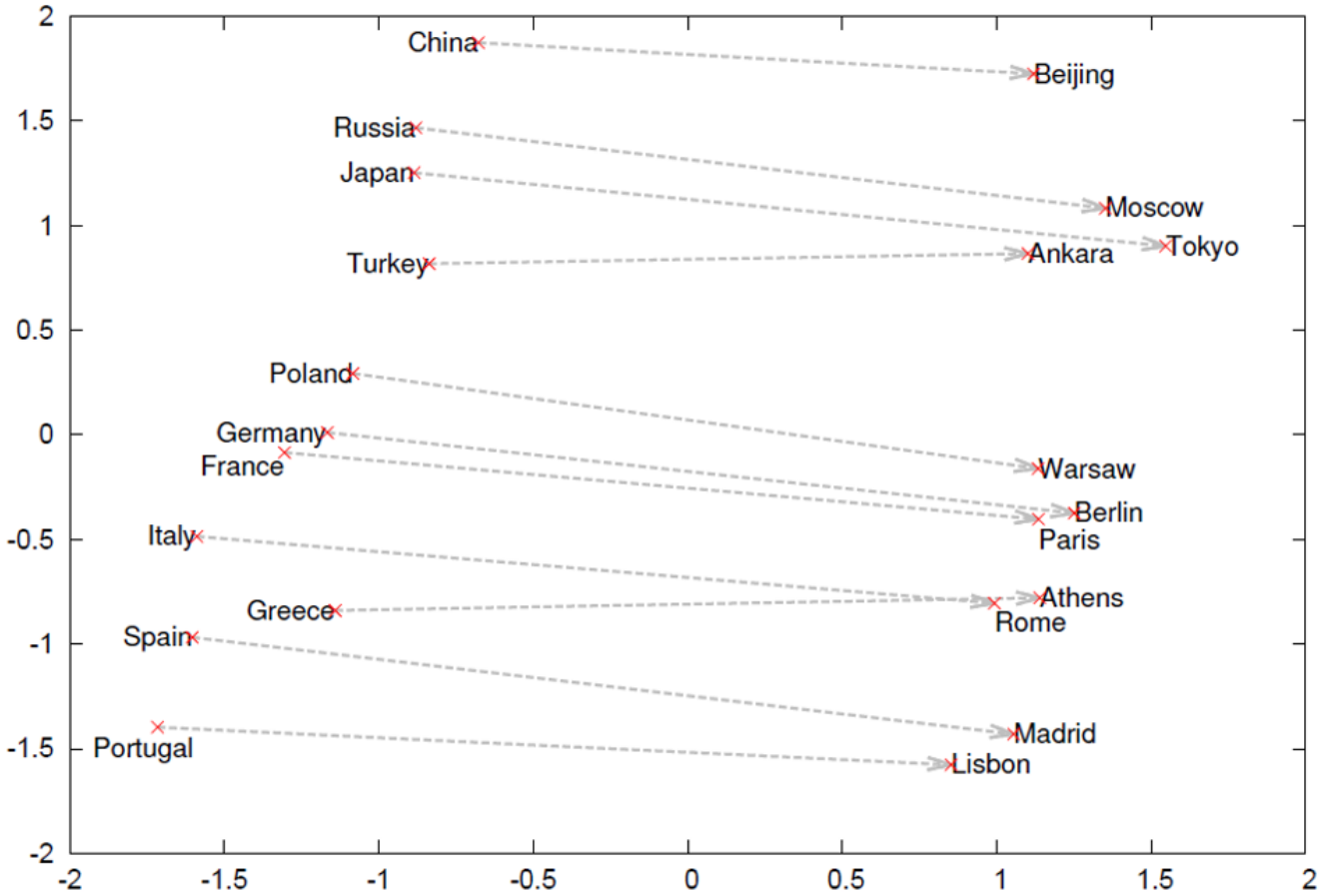
Summary till now

- How to learn word2vec embeddings?
  - Start with V random 300-dimensional vectors as initial weights
  - Take a text corpus and consider pairs of words that co-occur (within a context window) as positive samples
  - Consider some pairs of words that don't co-occur as negative samples
  - Train the classifier to distinguish these by slowly adjusting all the weights (embeddings) to improve the classifier performance
- Throw away the classifier code and keep the embeddings

- Official implementation: https://code.google.com/archive/p/word2vec/

- Paper: Mikolov et al. Efficient Estimation of Word Representations in Vector Space. ICLR 2013 Workshop. https://arxiv.org/abs/1301.3781 [40K citations]

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)

A criticism of Word2vec embeddings: Bias

- Ask “Paris : France :: Tokyo : x”
  - x = Japan
- Ask “father : doctor :: mother : x”
  - x = nurse
- Ask “man : computer programmer :: woman : x”
  - x = homemaker

- "Man is to computer programmer as woman is to homemaker? debiasing word embeddings." In *Advances in Neural Information Processing Systems*, 2016.

Another type of word embeddings

GloVe (Global Vectors for Word Representation)
https://nlp.stanford.edu/projects/glove/

Motivation of GloVe

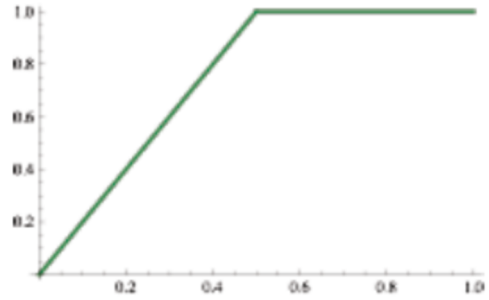
- Word2vec captures word co-occurrences one (sliding) window at a time
- What if we capture the frequency of all word co-occurrences (in a matrix) in one go? → computationally simpler and efficient

- Word2vec only uses local statistics (window-based)

- GloVe – hybrid of window-based and count-based models – combines local context information of words and global statistics (global co-occurrence of words)

GloVe objective function

$$\hat{J} = \sum_{i,j} f(X_{ij})(w_i^T \tilde{w}_j - \log X_{ij})^2$$

$f \sim$ 

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

X_{ij} = number of times that word j appears in the (global) context of word i.

The term $f(X_{ij})$ allows us to weight lower some very frequent co-occurrences and cap the importance of very frequent words.

Advantages of GloVe – faster training, scalable to huge corpora,
better performance with smaller vectors and even with smaller corpus

Evaluation of word embeddings

- Quality of word embeddings depends on many factors
  - Training data size
  - Number of dimensions in the embeddings
  - Exact details of the learning algorithm used

- How to evaluate the quality of word embeddings?
  - **Intrinsic evaluation** – judge the quality of embeddings directly
    - Ask humans to judge relatedness/similarity (S_h) between many pairs of words
    - Compute cosine similarity between the corresponding word vectors learned (S_m)
    - Compute correlation between S_h and S_m
  - **Extrinsic evaluation** – apply the embeddings for some task (e.g., text classification) and evaluate the performance on that task.