AASD 4004
Machine Learning - II

Applied AI Solutions Developer Program

Module 2
Feature Extraction in Text

Vejey Gandyer

Agenda

- Feature Extraction in Text
- Why Feature Extraction
- Bag of Words
- TF-IDF
- N-Grams
- One-hot encoding
- Word2Vec *
- GloVe *

* Will be seen in detail in DL-1

Feature Extraction

What is it?

Feature Extraction

Document - refers to a single piece of text information. This could be a text message, tweet, email, book, lyrics to a song. This is equivalent to one row or observation.

Corpus - a collection of documents. This would be equivalent to a whole data set of rows/observations.

Token - this is a word, phrase, or symbols derived from a document through the process of tokenization. This will happen behind the scenes so we won't need to worry too much about it and for our purposes it essentially means a word. For example the document *'How are you'* would have tokens of *'How'*, *'are'*, and *'you'*

Feature Extraction

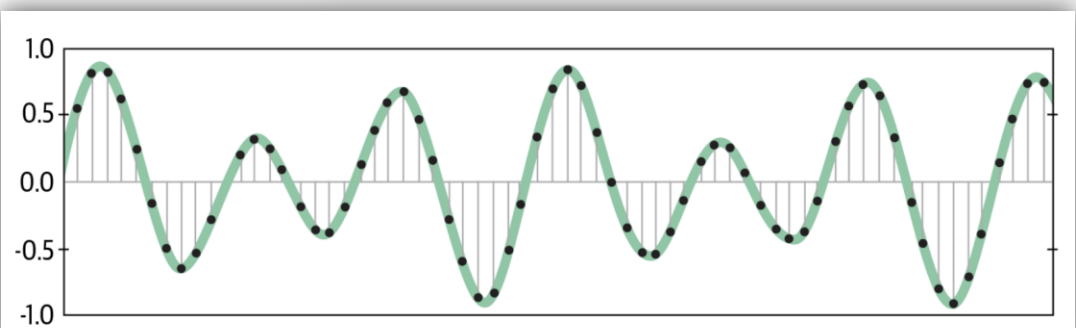
Why

Feature Extraction

What We See

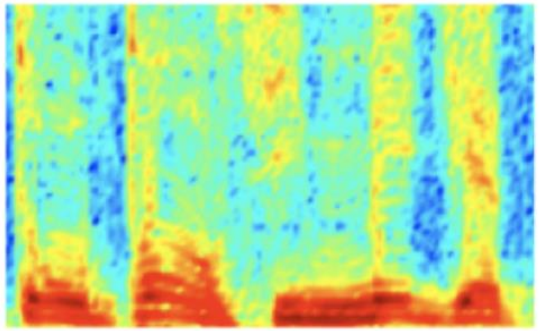
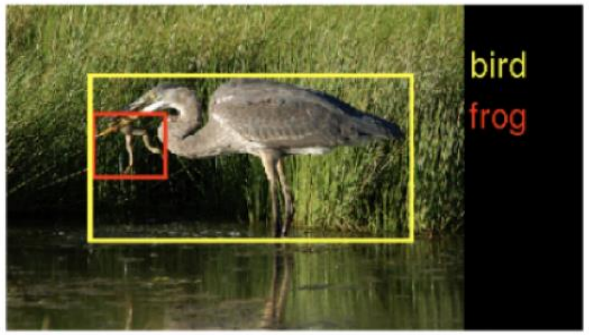
```
08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48
```

What Computers See




```
[-1274, -1252, -1160, -986, -792, -692, -614, -429, -286, -134, -57, -41,
-169, -456, -450, -541, -761, -1067, -1231, -1047, -952, -645, -489, -448,
-397, -212, 193, 114, -17, -110, 128, 261, 198, 390, 461, 772, 948, 1451,
1974, 2624, 3793, 4968, 5939, 6057, 6581, 7302, 7640, 7223, 6119, 5461,
4820, 4353, 3611, 2740, 2004, 1349, 1178, 1085, 901, 301, -262, -499,
-488, -707, -1406, -1997, -2377, -2494, -2605, -2675, -2627, -2500, -2148,
-1648, -970, -364, 13, 260, 494, 788, 1011, 938, 717, 507, 323, 324, 325,
350, 103, -113, 64, 176, 93, -249, -461, -606, -909, -1159, -1307, -1544]
```

Feature Extraction

AUDIO	IMAGES	TEXT
		
Audio Spectrogram	Image pixels	Word, context, or document vectors
DENSE	DENSE	SPARSE

Bag of Words

Bag of Words

Suppose we have a corpus with three sentences:

- "I like to play football"
- "Did you go outside to play tennis"
- "John and I play tennis”

Goal: Convert text to numbers

Bag of Words

1. Tokenize the sentences into words
2. Create Dictionary of Word Frequency
3. Bag of Words Model

BoW Step 1: Tokenization

Sentence 1	Sentence 2	Sentence 3
I	Did	John
like	you	and
to	go	I
play	outside	play
football	to	tennis
	play	
	tennis	

BoW Step 2: Dictionary of Word frequency

Word	Frequency
I	2
like	1
to	2
play	3
football	1
Did	1
you	1
go	1
outside	1
tennis	2
John	1
and	1

BoW Step 3: Bag of Words Model

	Play	Tennis	To	I	Football	Did	You	go
Sentence 1	1	0	1	1	1	0	0	0
Sentence 2	1	1	1	0	0	1	1	1
Sentence 3	1	1	0	1	0	0	0	0

Bag of Words

Let’s Create a Bag Of Words Model

BagOfWords.ipynb

Problems in Bag Of Words

"I like to play football"

"Did you go outside to play tennis"

"John and I play tennis"

	Play	Tennis	To	I	Football	Did	You	go
Sentence 1	1	0	1	1	1	0	0	0
Sentence 2	1	1	1	0	0	1	1	1
Sentence 3	1	1	0	1	0	0	0	0

Assigns equal importance to all words
Play = 3 (all sentences) COMMON
Football = 1 (RARE)

TF-IDF

TF-IDF

1. Tokenize the sentences into words
2. Create a Dictionary of word frequency
3. Sort the dictionary with word frequency
4. Compute Term frequency
5. Compute Inverse Document frequency
6. Compute TF-IDF values

TF-IDF Step 1: Tokenization

Sentence 1	Sentence 2	Sentence 3
I	Did	John
like	you	and
to	go	I
play	outside	play
football	to	tennis
	play	
	tennis	

TF-IDF Step 2: Dictionary of word frequency

Word	Frequency
I	2
like	1
to	2
play	3
football	1
Did	1
you	1
go	1
outside	1
tennis	2
John	1
and	1

TF-IDF Step 3: Sort word frequency

Word	Frequency
play	3
tennis	2
to	2
I	2
football	1
Did	1
you	1
go	1
outside	1
like	1
John	1
and	1

TF-IDF Step 4: Term frequency

Sentence 1
I
like
to
play
football

$$TF = \frac{\text{(Frequency of the word in the sentence)}}{\text{(Total number of words in the sentence)}}$$

play
Term Frequency = 1 / 5 = 0.20

football
Term Frequency = 1 / 5 = 0.20

TF-IDF Step 5: Inverse Term frequency

$$IDF = \frac{\text{(Total number of sentences (documents))}}{\text{(Number of sentences (documents) containing the word)}}$$

Word	Frequency	IDF
play	3	3/3 = 1
tennis	2	3/2 = 1.5
to	2	3/2 = 1.5
I	2	3/2 = 1.5
football	1	3/1 = 3
Did	1	3/1 = 3
you	1	3/1 = 3
go	1	3/1 = 3

TF-IDF Step 6: TF-IDF Values

```
TF-IDF = Term Frequency * Inverse Document Frequency
```

TF-IDF = Term Frequency * Inverse Document Frequency

Example: play

Term Frequency = 1 / 5 = 0.20

Inverse Document Frequency = 3 / 3 = 1

TF-IDF = 0.20 * 1 = **0.20**

TF-IDF

Word	Sentence 1	Sentence 2	Sentence 3
play	0.20 x 1 = 0.20	0.14 x 1 = 0.14	0.20 x 1 = 0.20
tennis	0 x 1.5 = 0	0.14 x 1.5 = 0.21	0.20 x 1.5 = 0.30
to	0.20 x 1.5 = 0.30	0.14 x 1.5 = 0.21	0 x 1.5 = 0
I	0.20 x 1.5 = 0.30	0 x 1.5 = 0	0.20 x 1.5 = 0.30
football	0.20 x 3 = 0.6	0 x 3 = 0	0 x 3 = 0

Problems with TF-IDF

- Problems with TF-IDF & BOW
  - Words are treated individually
  - No context information is retained
- Solution: **N-Gram**

N-Grams

Character

N-Grams

"A contiguous sequence of N items from a given sample of text or speech"

Markov Chains - sequence of states

Example: 2 states X and Y

In a Markov chain, you can stay at one state or move to another state at any given time XXYX

In an N-Grams model, an item in a sequence can be treated as a Markov state

Character N-Grams

Suppose we have a corpus with one sentence:

"Football is a very famous game"

2-Grams:

fo, oo, ot, tb, ba, al, ll, l , i, is, s , a, v, ve, er, ry, y , f, fa, am, mo, ou, us

3-Grams:

foo, oot, otb, tba, bal, all, ll , l i, is, is , s a, a , a v, ve, ver, ery, …

Character N-Grams Model

Step 1: Import all the necessary libraries

Step 2: Extract the contents of wikipedia article of interest

Step 3: Clean the extracted text

Step 4: Build the N-Grams model

Step 5: Generate sequence

Step 1: Import libraries

```
import nltk
import numpy as np
import random
import string

import bs4 as bs
import urllib.request
import re
```

Step 2: Scrape wiki article

```
raw_html = urllib.request.urlopen('https://en.wikipedia.org/wiki/Natural_language_processing')
raw_html = raw_html.read()

article_html = bs.BeautifulSoup(raw_html, 'lxml')

article_paragraphs = article_html.find_all('p')

article_text = ''

for para in article_paragraphs:
    article_text += para.text
```

Step 3: Clean text

```
article_text = re.sub(r'[^A-Za-z. ]', '', article_text)
```

Step 4: N-Grams Model

```
ngrams = {}
chars = 3

for i in range(len(article_text)-chars):
    seq = article_text[i:i+chars]
    print(seq)
    if seq not in ngrams.keys():
        ngrams[seq] = []
    ngrams[seq].append(article_text[i+chars])
```

Step 5: Generate Sequence

```
curr_sequence = article_text[0:chars]
output = curr_sequence
for i in range(200):
    if curr_sequence not in ngrams.keys():
        break
    possible_chars = ngrams[curr_sequence]
    next_char = possible_chars[random.randrange(len(possible_chars))]
    output += next_char
    curr_sequence = output[len(output)-chars:len(output)]
```

Character N-Gram Sequence Generator

```
Natural for language proped.  Duringfor a starties show to a complexistical gradigm common ablem.[2] infor used involution mached be featural languisticalled responding Machine tree or develow.
Thes impl
```

N-Grams

Word

Word N-Gram

Suppose we have a corpus with three sentences:

- "I like to play football"
- "Did you go outside to play tennis"
- "John and I play tennis”

2-Grams:

I like, like to, to play, play football

Did you, you go, go outside, outside to, to play, play tennis

John and, and I, I play, play tennis

3-Grams:

I like to, like to play, to play football

Did you go, you go outside, go outside to, outside to play, to play tennis

John and I, and I play, I play tennis

Word N-Grams Model

Step 1: Import all the necessary libraries

Step 2: Extract the contents of wikipedia article of interest

Step 3: Clean the extracted text

Step 4: Build the N-Grams model

Step 5: Generate sequence

Step 1: Import libraries

```
import nltk
import numpy as np
import random
import string

import bs4 as bs
import urllib.request
import re
```

Step 2: Scrape wiki article

```
raw_html = urllib.request.urlopen('https://en.wikipedia.org/wiki/Natural_language_processing')
raw_html = raw_html.read()

article_html = bs.BeautifulSoup(raw_html, 'lxml')

article_paragraphs = article_html.find_all('p')

article_text = ''

for para in article_paragraphs:
    article_text += para.text
```

Step 3: Clean text

```
article_text = re.sub(r'[^A-Za-z. ]', '', article_text)
```

Step 4: N-Grams Model

```
ngrams = {}
words = 3

words_tokens = nltk.word_tokenize(article_text)
for i in range(len(words_tokens)-words):
    seq = ' '.join(words_tokens[i:i+words])
    print(seq)
    if  seq not in ngrams.keys():
        ngrams[seq] = []
    ngrams[seq].append(words_tokens[i+words])
```

Step 5: Generate Sequence

```
curr_sequence = ' '.join(words_tokens[0:words])
output = curr_sequence
for i in range(50):
    if curr_sequence not in ngrams.keys():
        break
    possible_words = ngrams[curr_sequence]
    next_word = possible_words[random.randrange(len(possible_words))]
    output += ' ' + next_word
    seq_words = nltk.word_tokenize(output)
    curr_sequence = ' '.join(seq_words[len(seq_words)-words:len(seq_words)])
```

Word N-Gram Sequence Generator

Character-Gram Generator

```
Natural for language proped.  Duringfor a starties show to a complexistical gradigm common ablem.[2] infor used involution mached be featural languisticalled responding Machine tree or develow.
Thes impl
```

Word-Gram Generator

```
Natural language processing ( NLP ) generally started in the 1950s , although work can be found from earlier periods . In 1950 , Alan Turing published an article titled `` Computing Machinery and Intelligence ''
```

```
Natural language processing tasks are closely intertwined , they are frequently subdivided into categories for convenience . A coarse division is given below . The Georgetown experiment in 1954 involved fully automatic translation of more than sixty Russian sentences into English . The Georgetown experiment in 1954 involved fully automatic translation of
```

One-hot encoding

One-hot encoding

```
def get_onehot_vector(somestring):
    onehot_encoded = []
    for word in somestring.split():
                temp = [0]*len(vocab)
                if word in vocab:
                            temp[vocab[word]-1] = 1
                onehot_encoded.append(temp)
    return onehot_encoded
```

CountVectorizer

```
from sklearn.feature_extraction.text import CountVectorizer
# create CountVectorizer object
vectorizer = CountVectorizer()
corpus = [
          'Text of first document.',
          'Text of the second document made longer.',
          'Number three.',
          'This is number four.',
]
# learn the vocabulary and store CountVectorizer sparse matrix in X
X = vectorizer.fit_transform(corpus)
# columns of X correspond to the result of this method
vectorizer.get_feature_names() == (
    ['document', 'first', 'four', 'is', 'longer',
     'made', 'number', 'of', 'second', 'text',
     'the', 'this', 'three'])
# retrieving the matrix in the numpy form
X.toarray()
# transforming a new document according to learn vocabulary
vectorizer.transform(['A new document.']).toarray()
```

Word2Vec

Word2Vec

Words that appear in same context share semantic meaning

1. **Count-based method** (Latent Semantic Analysis)

Count-based methods compute the statistics of how often some word co-occurs with its neighbour words in a large text corpus, and then map these count-statistics down to a small, dense vector for each word.

2. **Predictive method** (Neural Probabilistic Language Model)

Predictive models directly try to predict a word from its neighbours in terms of learned small, dense embedding vectors.

Word2Vec —> Predictive Method

Word2Vec

```
from gensim.models import Word2Vec, KeyedVectors
pretrainedpath = "NLPBookTut/GoogleNews-vectors-negative300.bin"
w2v_model = KeyedVectors.load_word2vec_format(pretrainedpath, binary=True)
print('done loading Word2Vec')
print(len(w2v_model.vocab)) #Number of words in the vocabulary.
print(w2v_model.most_similar['beautiful'])
W2v_model['beautiful']
```

```
[('gorgeous', 0.8353004455566406),
 ('lovely', 0.810693621635437),
 ('stunningly_beautiful', 0.7329413890838623),
 ('breathtakingly_beautiful', 0.7231341004371643),
 ('wonderful', 0.6854087114334106),
 ('fabulous', 0.6700063943862915),
 ('loveliest', 0.661257624626159 7),
 ('prettiest', 0.6595001816749573),
 ('beatiful', 0.6593326330184937),
 ('magnificent', 0.659140229225158 7)]
```

Word2Vec

Similar words tend to occur together and will have similar context

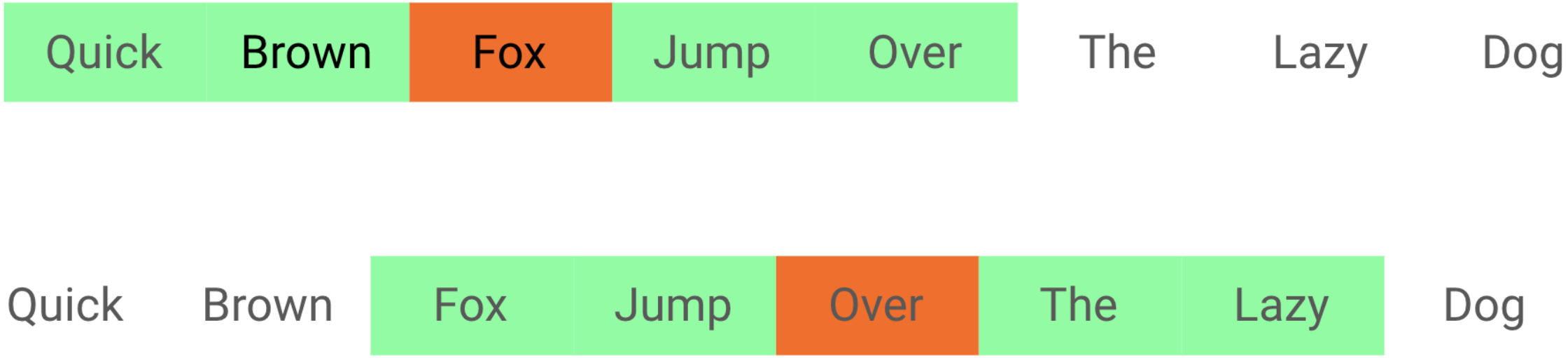
Eg: – Apple is a fruit. Mango is a fruit.

Apple and mango tend to have a similar context i.e fruit.

Word2Vec

Co-occurrence – For a given corpus, the co-occurrence of a pair of words say w1 and w2 is the number of times they have appeared together in a Context Window.
Context Window – Context window is specified by a number and the direction. So what does a context window of 2 (around) means?

Context Window



Context Window

Corpus: He is not lazy. He is intelligent. He is smart.

	He	is	not	lazy	intelligent	smart
He	0	4	2	1	2	1
is	4	0	1	2	2	1
not	2	1	0	1	0	0
lazy	1	2	1	0	0	0
intelligent	2	2	0	0	0	0
smart	1	1	0	0	0	0

Context Window

Corpus: He is not lazy. He is intelligent. He is smart.

He	is	not	lazy	He	is	intelligent	He	is	smart
He	is	not	lazy	He	is	intelligent	He	is	smart
He	is	not	lazy	He	is	intelligent	He	is	smart
He	is	not	lazy	He	is	intelligent	He	is	smart

Factorization

Co-occurrence matrix is decomposed using techniques like PCA, SVD etc. into factors and combination of these factors forms the word vector representation.

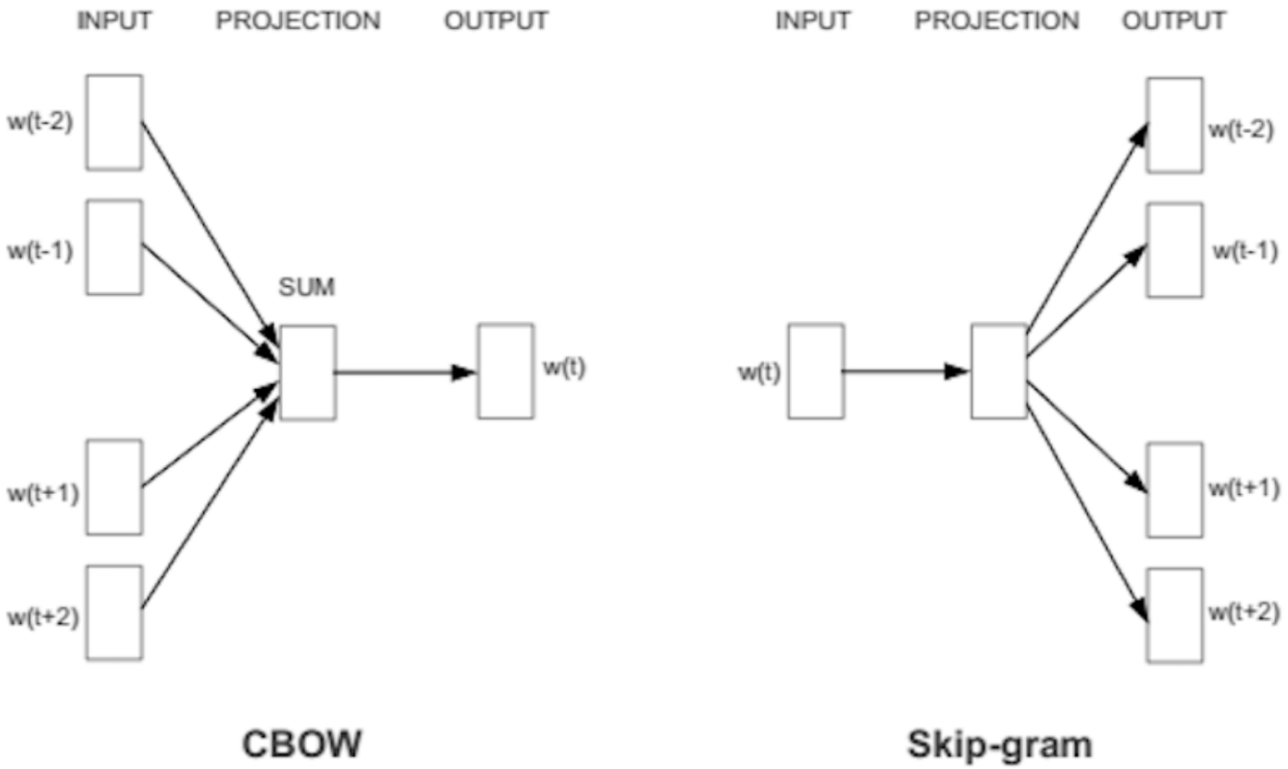
PCA decomposes Co-Occurrence matrix into three matrices, U,S and V where U and V are both orthogonal matrices. What is of importance is that dot product of U and S gives the word vector representation and V gives the word context representation.

$$\underset{m \times n}{\hat{X}} \approx \underset{m \times r}{U} \underset{r \times r}{S} \underset{r \times n}{V^{\mathsf{T}}}$$

Word2Vec

Word2vec is to group the vectors of similar words together in vectorspace.
Word2vec creates vectors that are distributed numerical representations of word features, features such as the contex of individual words.

Word2vec is similar to an autoencoder, encoding each word in a vector, but rather than training against the input words through reconstruction, as a [restricted Boltzmann machine] does, word2vec trains words against other words that neighbor them in the input corpus.



Word2Vec

CBoW

CBoW

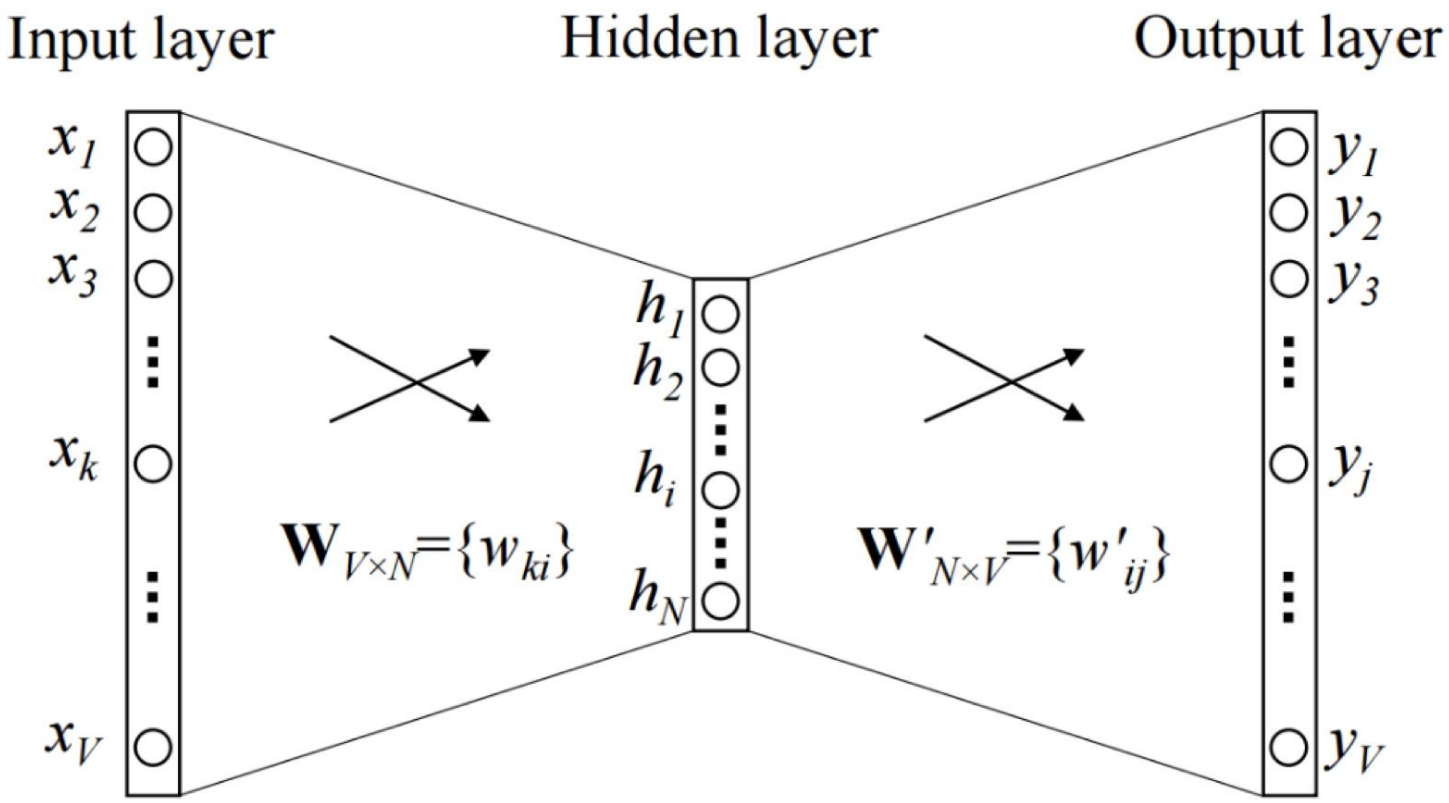
- Predict Probability of a word given its context
- P(word | context)

Corpus = Hey, this is sample corpus using only one context word.

Input	Output			Hey	This	is	sample	corpus	using	only	one	context	word
Hey	this		Datapoint 1	1	0	0	0	0	0	0	0	0	0
this	hey		Datapoint 2	0	1	0	0	0	0	0	0	0	0
is	this		Datapoint 3	0	0	1	0	0	0	0	0	0	0
is	sample		Datapoint 4	0	0	1	0	0	0	0	0	0	0
sample	is		Datapoint 5	0	0	0	1	0	0	0	0	0	0
sample	corpus		Datapoint 6	0	0	0	1	0	0	0	0	0	0
corpus	sample		Datapoint 7	0	0	0	0	1	0	0	0	0	0
corpus	using		Datapoint 8	0	0	0	0	1	0	0	0	0	0
using	corpus		Datapoint 9	0	0	0	0	0	1	0	0	0	0
using	only		Datapoint 10	0	0	0	0	0	1	0	0	0	0
only	using		Datapoint 11	0	0	0	0	0	0	1	0	0	0
only	one		Datapoint 12	0	0	0	0	0	0	1	0	0	0
one	only		Datapoint 13	0	0	0	0	0	0	0	1	0	0
one	context		Datapoint 14	0	0	0	0	0	0	0	1	0	0
context	one		Datapoint 15	0	0	0	0	0	0	0	0	1	0
context	word		Datapoint 16	0	0	0	0	0	0	0	0	1	0
word	context		Datapoint 17	0	0	0	0	0	0	0	0	0	1

Hey	this	is	sample	corpus	using	only	one	context	word
0	0	0	1	0	0	0	0	0	0

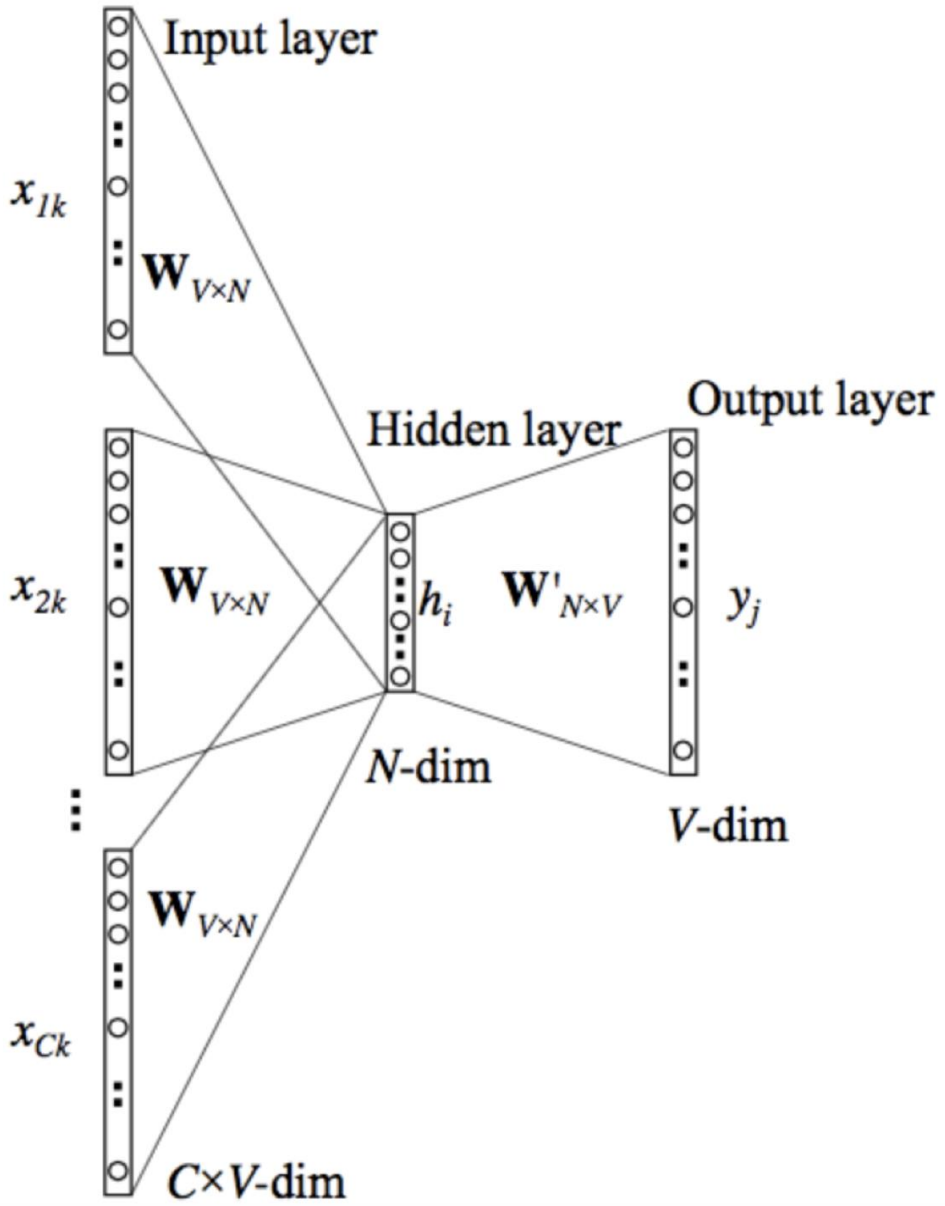
CBoW


Input layer
Hidden layer
Output layer
x_1
x_2
x_3
x_k
x_V
h_1
h_2
h_i
h_N
y_1
y_2
y_3
y_j
y_V
$\mathbf{W}_{V\times N}=\{w_{ki}\}$
$\mathbf{W}'_{N\times V}=\{w'_{ij}\}$

CBoW

1. The input layer and the target, both are one- hot encoded of size [1 X V]. Here V=10
2. There are two sets of weights. one is between the input and the hidden layer and second between hidden and output layer. Input-Hidden layer matrix size =[V X N] , hidden-Output layer matrix size =[N X V] : N is the number of dimensions we choose to represent our word in. It is arbitary and a hyper-parameter for a Neural Network. Also, N is the number of neurons in the hidden layer. Here, N=4.
3. There is a no activation function between any layers.
4. The input is multiplied by the input-hidden weights and called hidden activation. It is simply the corresponding row in the input-hidden matrix copied.
5. The hidden input gets multiplied by hidden- output weights and output is calculated.
6. Error between output and target is calculated and propagated back to re-adjust weights.
7. The weight between the hidden layer and the output layer is taken as the word vector representation of the word.

CBoW


Input layer
Hidden layer
Output layer
x_{1k}
x_{2k}
x_{Ck}
$\mathbf{W}_{V\times N}$
$\mathbf{W}_{V\times N}$
$\mathbf{W}_{V\times N}$
h_i
$\mathbf{W}'_{N\times V}$
y_j
N-dim
V-dim
$C\times V$-dim

CBoW

Context

C1	this	0	1	0	0	0	0	0	0	0	0
C2	corpus	0	0	0	0	1	0	0	0	0	0
C3	context	0	0	0	0	0	0	0	0	1	0

Input-Hidden Weight

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32
33	34	35	36
37	38	39	40

Hidden Activation

5	6	7	8
17	18	19	20
33	34	35	36

Average hidden Activation

18.33333333	19.33333333	20.33333333	21.33333333

So, the input layer will have 3 [1 X V] Vectors in the input as shown above and 1 [1 X V] in the output layer. Rest of the architecture is same as for a 1-context CBOW.

The steps remain the same, only the calculation of hidden activation changes. Instead of just copying the corresponding rows of the input-hidden weight matrix to the hidden layer, an average is taken over all the corresponding rows of the matrix. The average vector calculated becomes the hidden activation. So, if we have three context words for a single target word, we will have three initial hidden activations which are then averaged element-wise to obtain the final activation.

CBoW

Advantages

Being probabilistic is nature, it is supposed to perform superior to deterministic methods.

It is low on memory. It does not need to have huge RAM requirements like that of co-occurrence matrix where it needs to store three huge matrices.

Disadvantages

CBOW takes the average of the context of a word (as seen above in calculation of hidden activation). For example, Apple can be both a fruit and a company but CBOW takes an average of both the contexts and places it in between a cluster for fruits and companies.

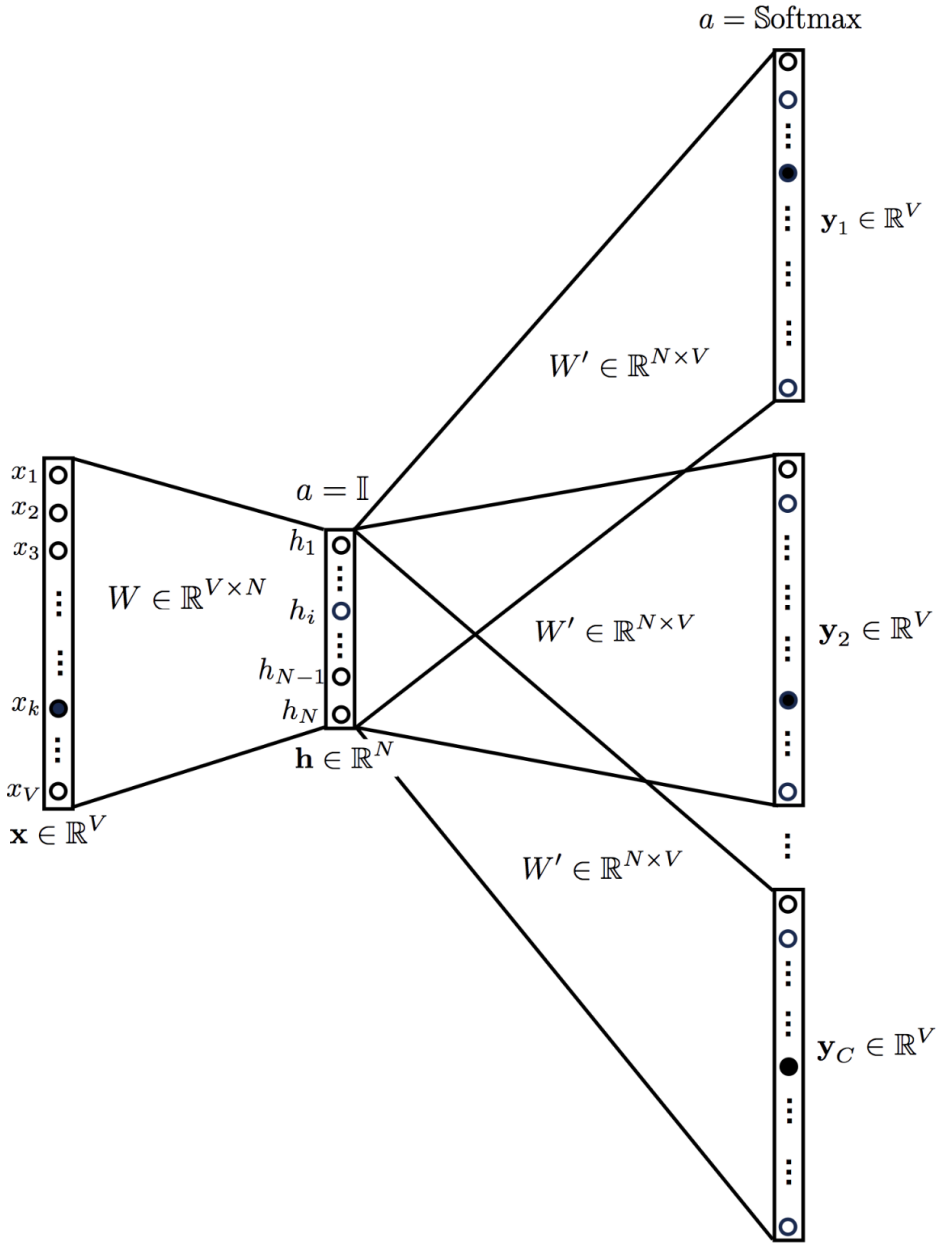
Training a CBOW from scratch can take forever if not properly optimized.

Word2Vec

SkipGram

SkipGram

Predicts the context given a word
P(context | word)



SkipGram

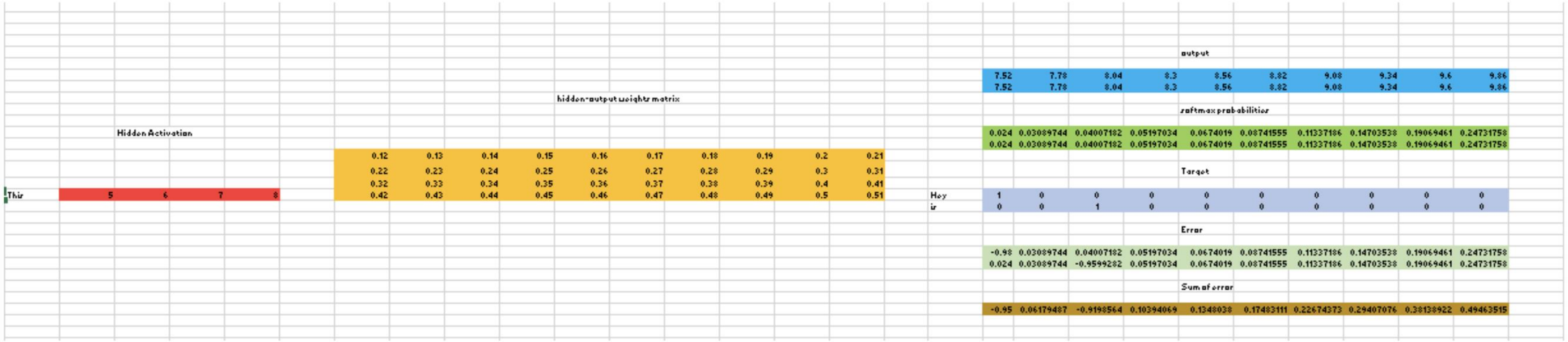
Hey, this is sample corpus using only one context word.

Input	Output(Context1)	Output(Context2)
Hey	this	<padding>
this	Hey	is
is	this	sample
sample	is	corpus
corpus	sample	corpus
using	corpus	only
only	using	one
one	only	context
context	one	word
word	context	<padding>

SkipGram

The input vector for skip-gram is going to be similar to a 1-context CBOW model.
The difference will be in the target variable. Since we have defined a context window of 1 on both the sides, there will be “two” one hot encoded target variables and “two” corresponding outputs.
Two separate errors are calculated with respect to the two target variables and the two error vectors obtained are added element-wise to obtain a final error vector which is propagated back to update the weights.
The weights between the input and the hidden layer are taken as the word vector representation after training.
The loss function is of same type as of the CBOW model.

SkipGram



Input layer size – [1 X V]
Input hidden weight matrix size – [V X N]
Number of neurons in hidden layer – N
Hidden-Output weight matrix size – [N X V]
Output layer size – C [1 X V]
C - Context words = 2

SkipGram

1. The row in red is the hidden activation corresponding to the input one-hot encoded vector. It is basically corresponding row of input-hidden matrix copied.
2. The yellow matrix is the weight between the hidden layer and the output layer.
3. The blue matrix is obtained by the matrix multiplication of hidden activation and the hidden output weights. There will be two rows calculated for two target(context) words.
4. Each row of the blue matrix is converted into its softmax probabilities individually as shown in the green box.
5. The grey matrix contains the one hot encoded vectors of the two context words(target).
6. Error is calculated by substracting the first row of the grey matrix(target) from the first row of the green matrix(output) element-wise. This is repeated for the next row. Therefore, for n target context words, we will have n error vectors.
7. Element-wise sum is taken over all the error vectors to obtain a final error vector.
8. This error vector is propagated back to update the weights.

SkipGram

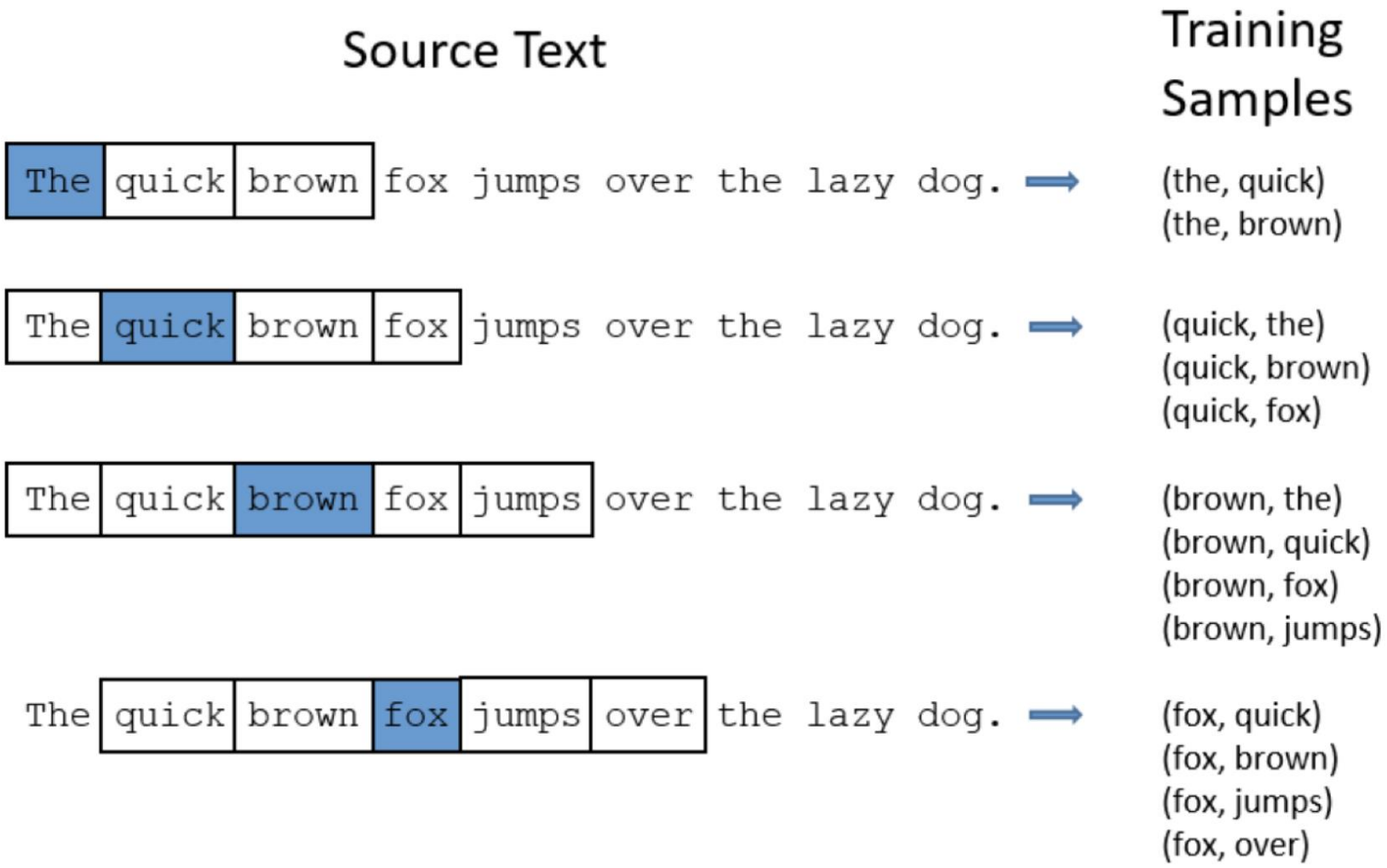
Advantages

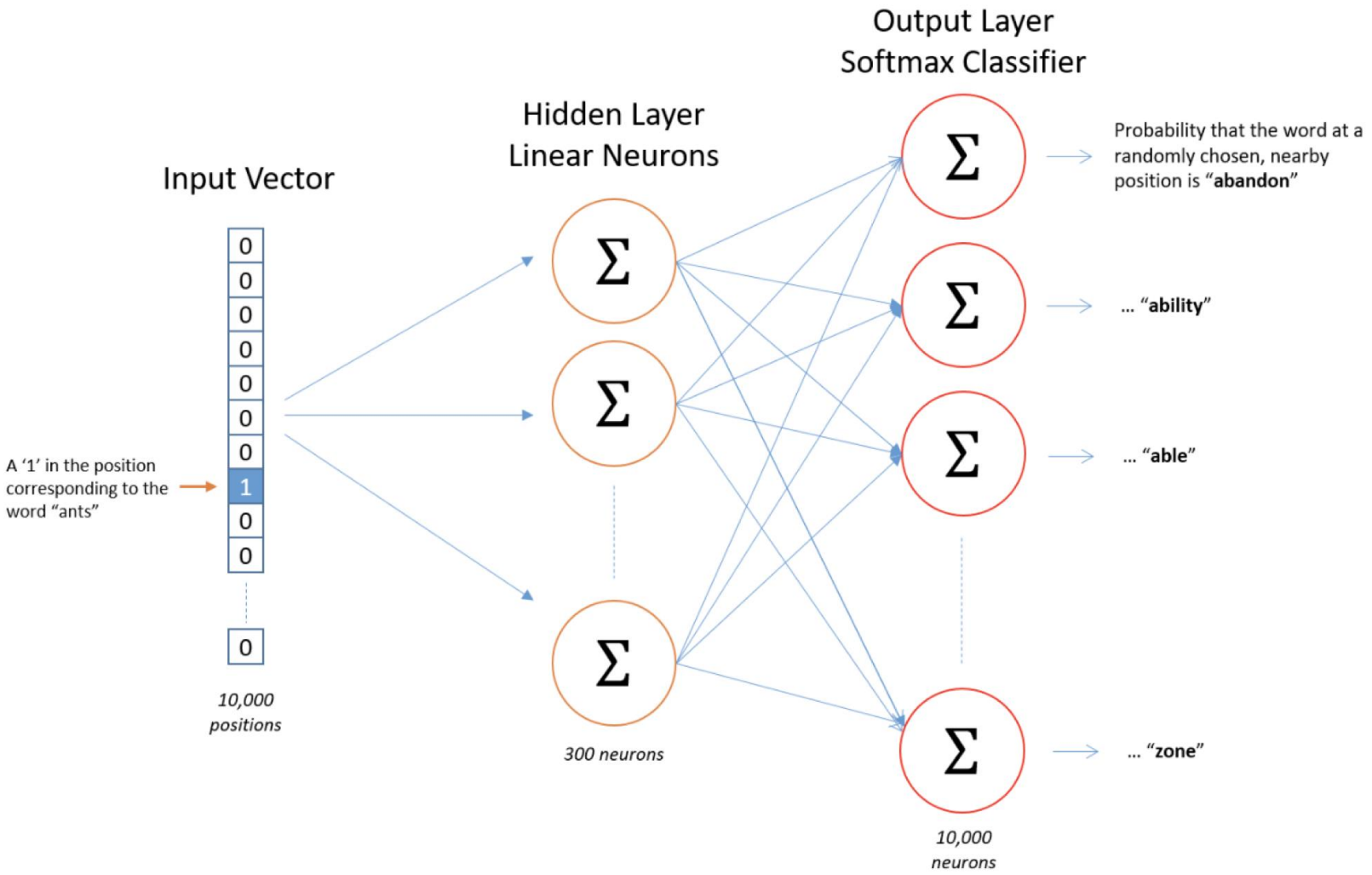
Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit.

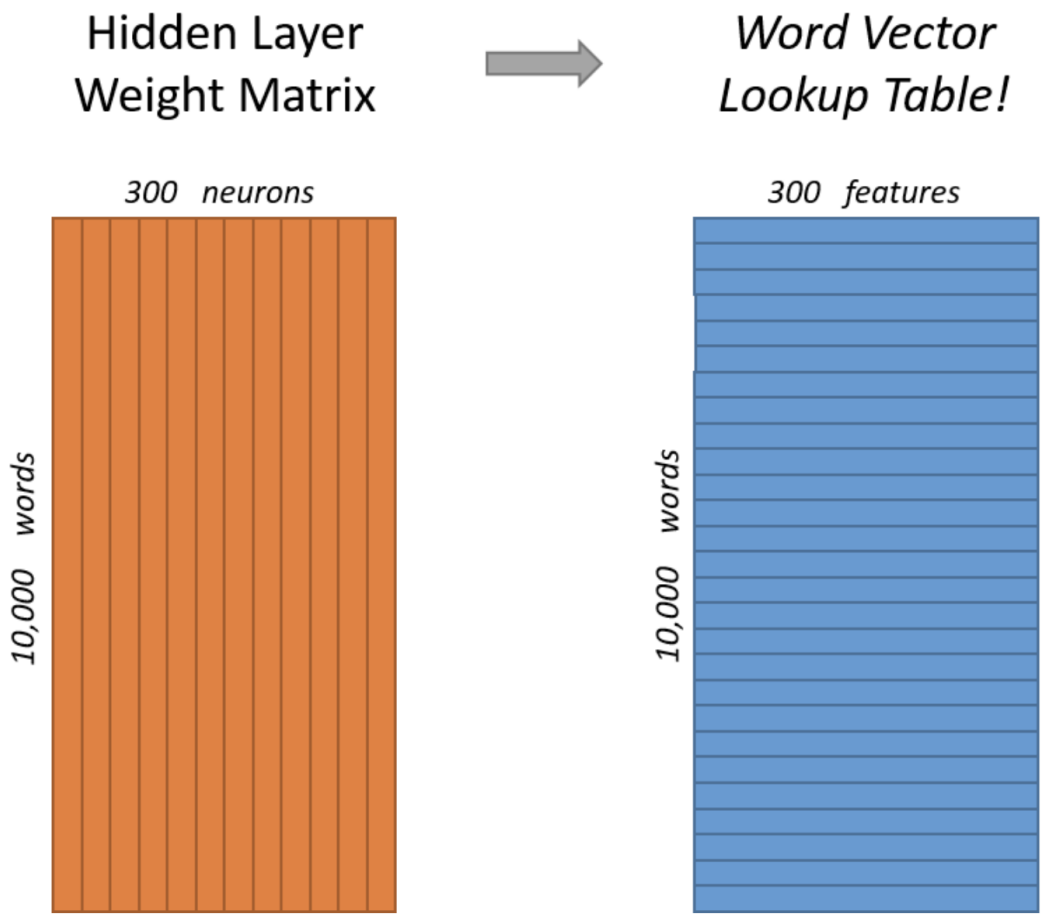
http://bit.ly/wevi-online

Word2Vec Steps Summary

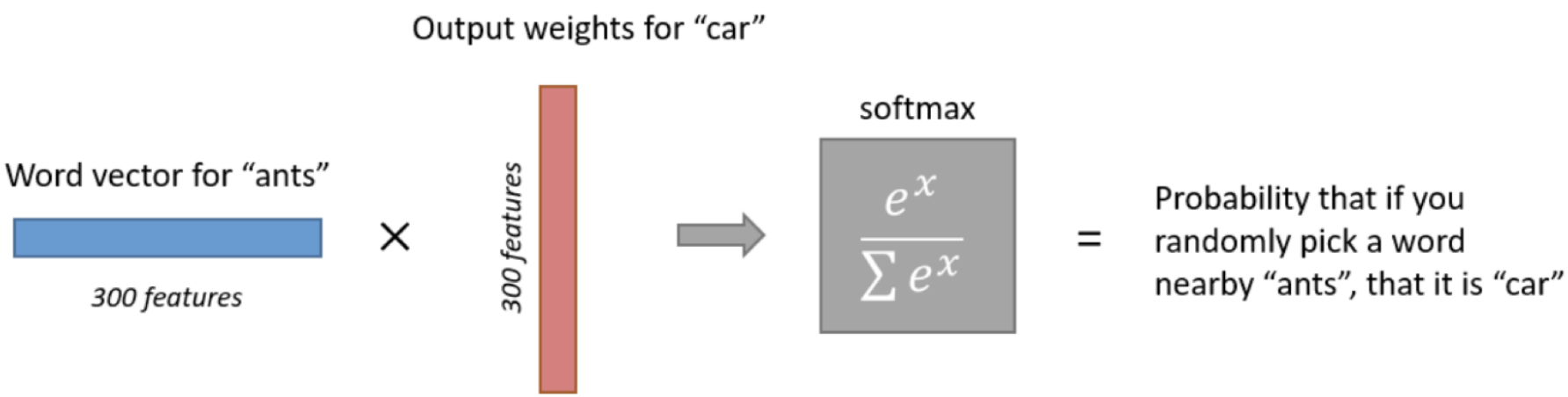
1. Take a 3 layer neural network. (1 input layer + 1 hidden layer + 1 output layer)
2. Feed it a word and train it to predict its neighbouring word.
3. Remove the last (output layer) and keep the input and hidden layer.
4. Now, input a word from within the vocabulary. The output given at the hidden layer is the '*word embedding*' of the input word.


Source Text
Training Samples
The quick brown fox jumps over the lazy dog.
(the, quick)
(the, brown)
The quick brown fox jumps over the lazy dog.
(quick, the)
(quick, brown)
(quick, fox)
The quick brown fox jumps over the lazy dog.
(brown, the)
(brown, quick)
(brown, fox)
(brown, jumps)
The quick brown fox jumps over the lazy dog.
(fox, quick)
(fox, brown)
(fox, jumps)
(fox, over)

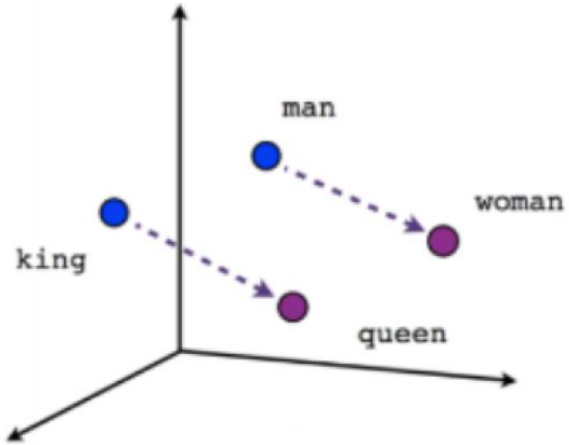

Output Layer
Softmax Classifier
Hidden Layer
Linear Neurons
Input Vector
Probability that the word at a randomly chosen, nearby position is "abandon"
... "ability"
... "able"
... "zone"
A '1' in the position corresponding to the word "ants"
0
0
0
0
0
0
0
1
0
0
0
10,000 positions
300 neurons
10,000 neurons


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

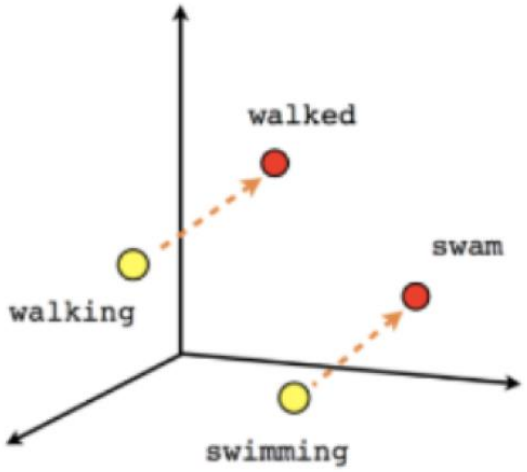
$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$



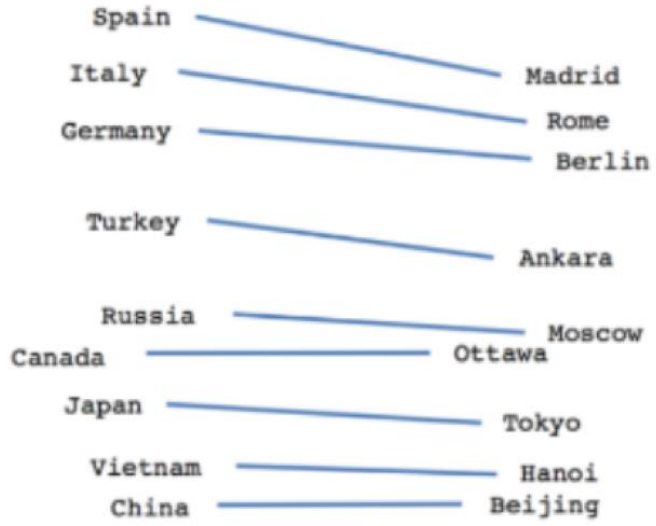
Word2Vec Relationships



Male-Female



Verb tense



Country-Capital

Usecases of Word Embeddings

1. Finding the degree of similarity between two words.

```
model.similarity('woman','man')
```

```
0.73723527
```

2. Finding odd one out.

```
model.doesnt_match('breakfast cereal dinner lunch';.split())
```

```
'cereal'
```

3. Amazing things like woman+king-man =queen

```
model.most_similar(positive=['woman','king'],negative=['man'],topn=1)
```

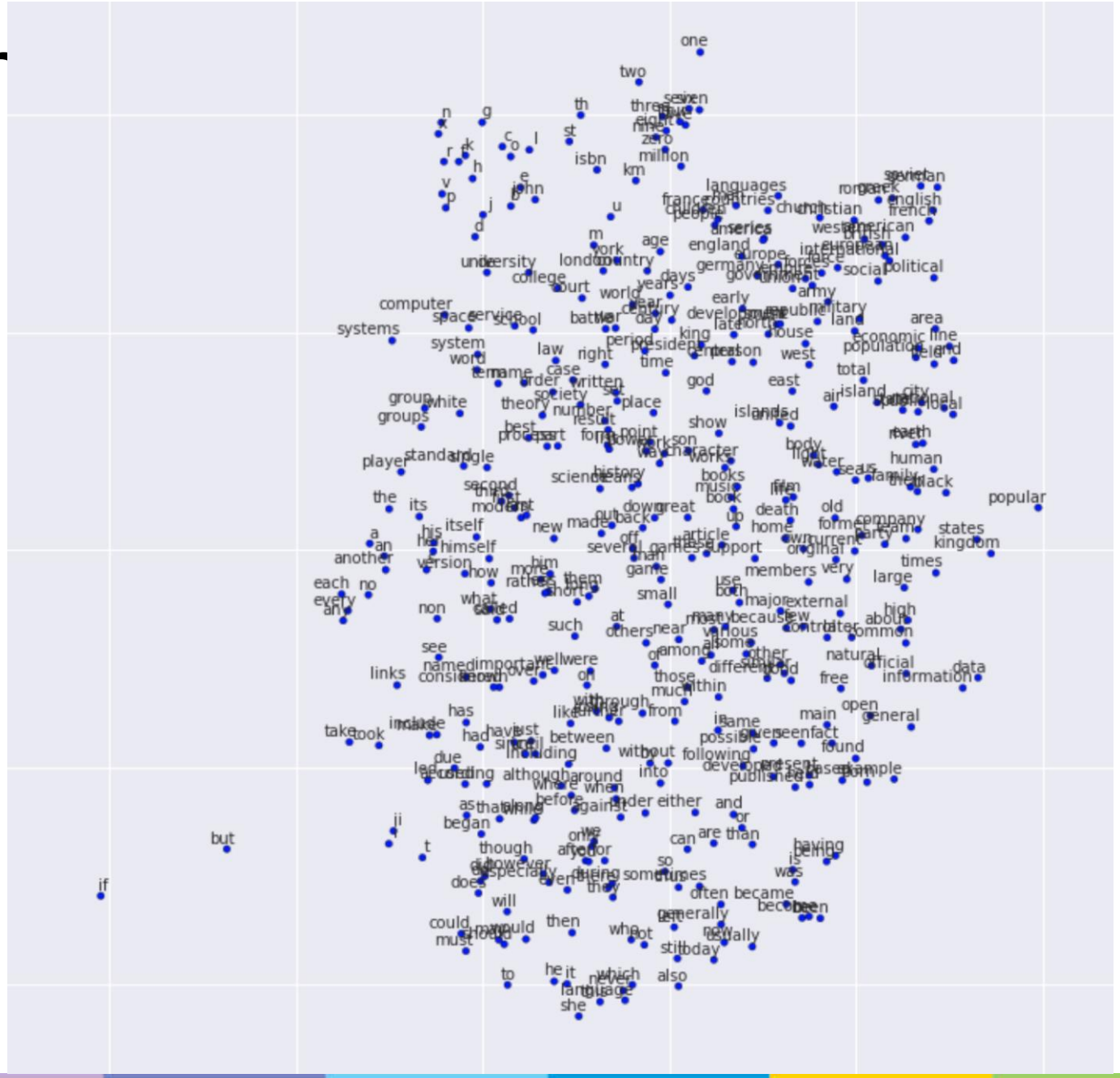
```
queen: 0.508
```

4. Probability of a text under the model

```
model.score(['The fox jumped over the lazy dog'.split()])
```

```
0.21
```

Visualizatior



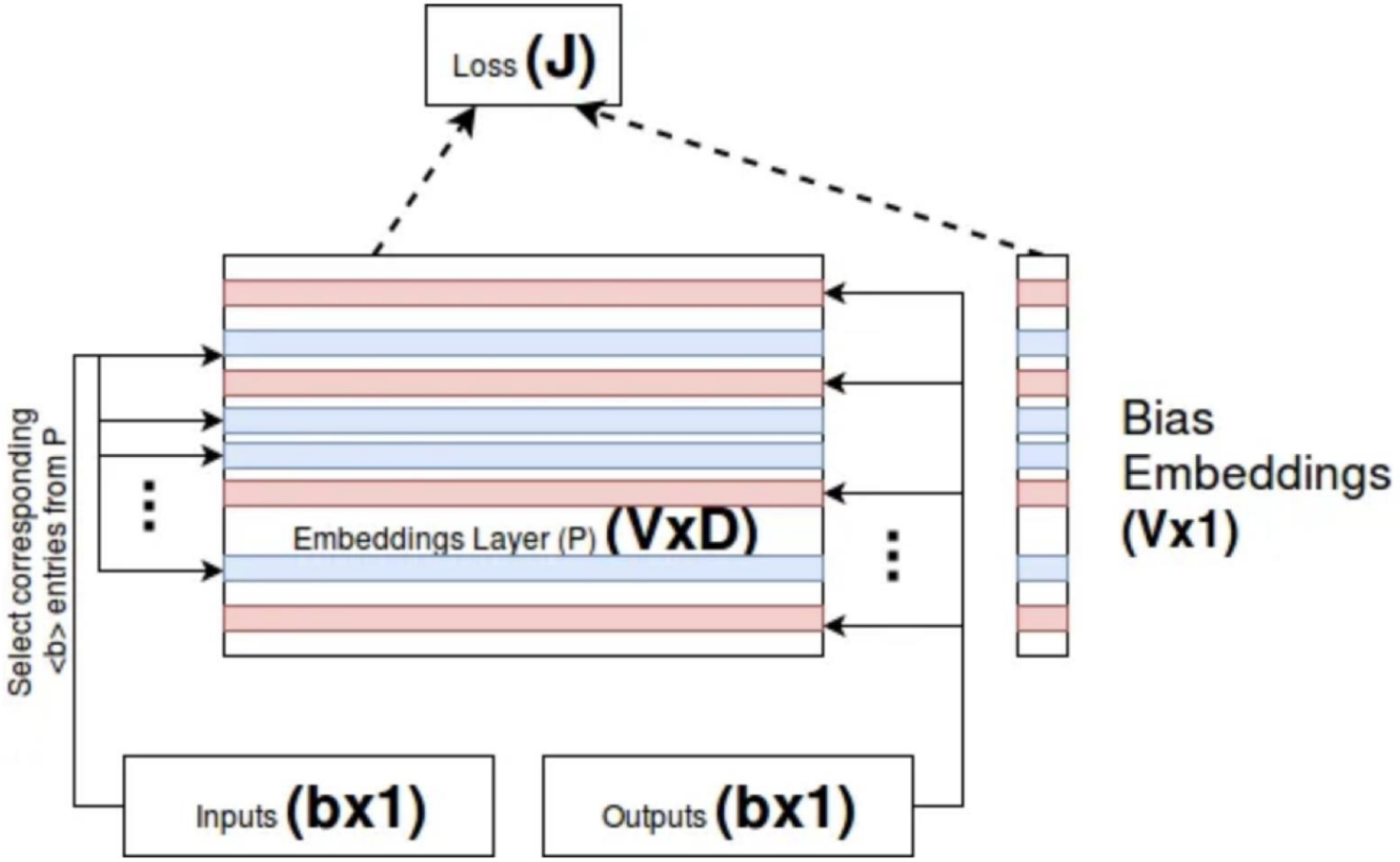
GloVe

Glove

Idea: Ratios between probabilities of words appearing next to each other carry more information that individual probabilities

1. $P_{ice,solid}/P_{steam,solid}$ will be very high
2. $P_{ice,gas}/P_{steam,gas}$ is very low
3. $P_{ice,water}/P_{steam,water}$ will be higher than $P_{ice,fashion}/P_{steam,fashion}$

Glove



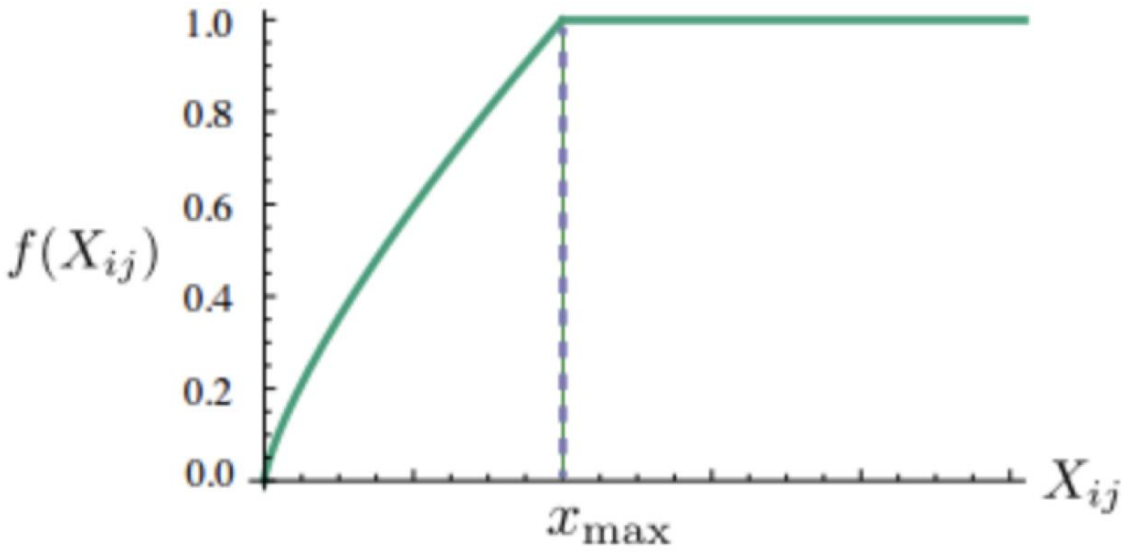
GloVe: High-level Architecture

Glove

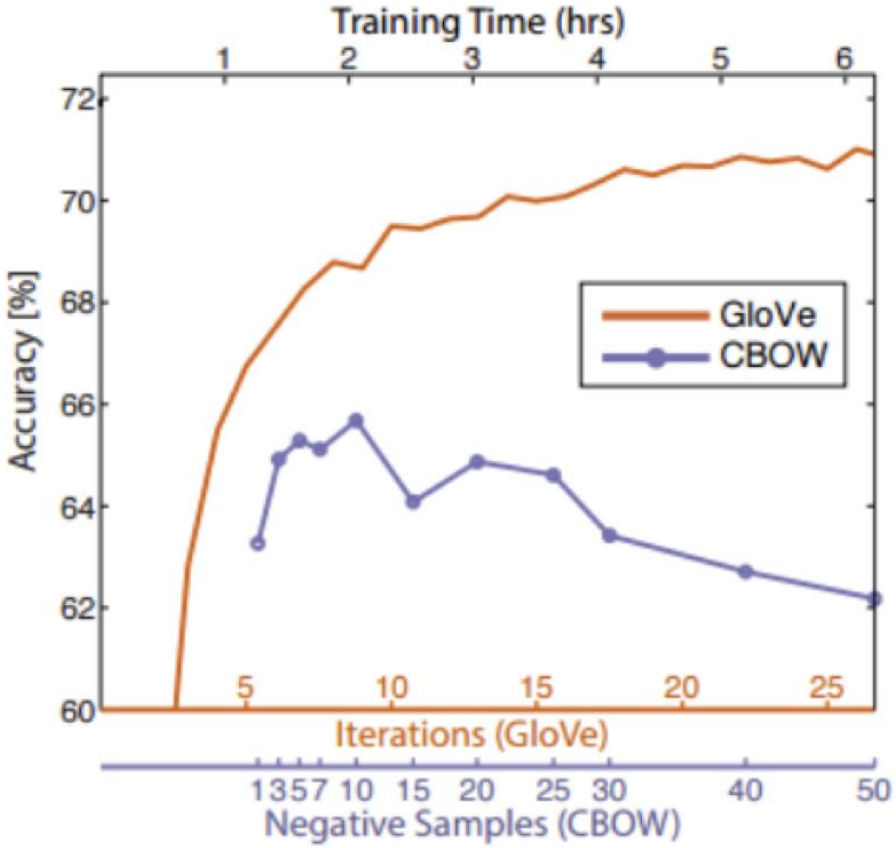
Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Glove

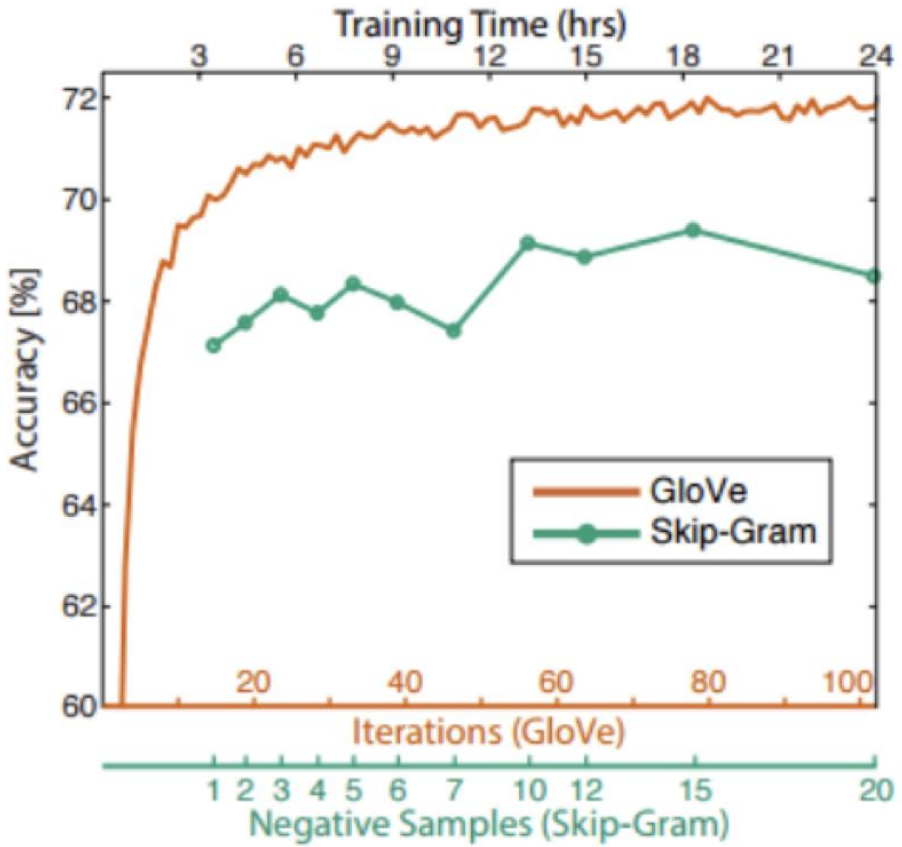
$$f(x) = \begin{cases} (x/x_{\max})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases}$$



Glove Vs Word2Vec



(a) GloVe vs CBOW



(b) GloVe vs Skip-Gram

Further Reading

Practical Natural Language Processing
Soumya Vajjala, Anuj Gupta, Harshit Surana, Bodhisattwa Majumder