



Natural Language Processing

Text Pre-Processing and Exploring Data

There are various methods and techniques to preprocess the text data along with exploratory data analysis.

We are going to learn the following :

- Lowercasing
- Punctuation removal
- Stop words removal
- Text standardization
- Tokenization
- Stemming
- Lemmatization
- Exploratory data analysis
- End-to-end processing pipeline

Lemmatization and Stemming Difference

Word	Lemmatization	Stemming
was	be	wa
studies	study	studi
studying	study	study

Stemming vs Lemmatization

change
changing
changes
changed
changer




Diagram illustrating stemming: five arrows point from the words 'change', 'changing', 'changes', 'changed', and 'changer' to the stem 'chang'.

chang

change
changing
changes
changed
changer

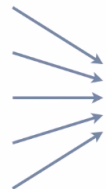


Diagram illustrating lemmatization: five arrows point from the words 'change', 'changing', 'changes', 'changed', and 'changer' to the lemma 'change'.

change

POS Tagging

POS tagging marks words in the corpus to a corresponding part of a speech tag based on its context and definition.



POS tags used in NLTK
<ul style="list-style-type: none">• NNP proper noun, singular 'Alice'• NN noun, singular 'desk'• RB adverb very, silently,• VBD verb, past tense took• JJ adjective 'large'• VBZ verb, 3rd person sing. present takes

POS Tagging (Contd.)

```
stop_words = set(stopwords.words('english'))
txt = '''
    Text mining, also referred to as text data mining,
    roughly equivalent to text analytics, is the process of deriving high-quality information from text.
    High-quality information is typically derived through the devising of patterns and trends through means
    such as statistical pattern learning.
'''
tokenized = sent_tokenize(txt)
for i in tokenized:
    wordsList = nltk.word_tokenize(i)
    wordsList = [w for w in wordsList if not w in stop_words]
    tagged = nltk.pos_tag(wordsList)

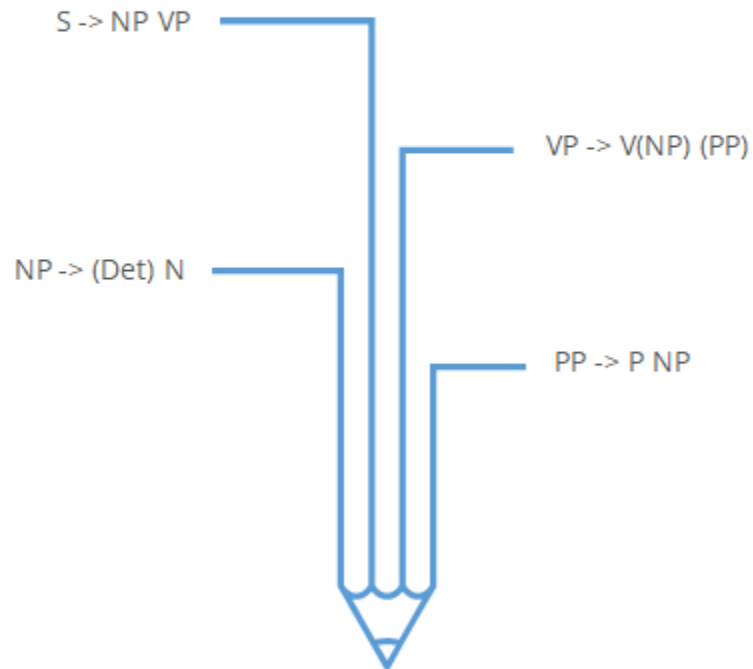
print(tagged)
```

```
[('Text', 'NNP'), ('mining', 'NN'), (',', ','), ('also', 'RB'), ('referred', 'VBD'), ('text', 'JJ'), ('data', 'NN'), ('mining', 'NN'), (',', ','), ('roughly', 'RB'), ('equivalent', 'JJ'), ('text', 'NN'), ('analytics', 'NNS'), (',', ','), ('process', 'N'), ('deriving', 'VBG'), ('high-quality', 'NN'), ('information', 'NN'), ('text', 'NN'), (',', ',')]
[('High-quality', 'NNP'), ('information', 'NN'), ('typically', 'RB'), ('derived', 'VBD'), ('devising', 'VBG'), ('patterns', 'NN'), ('trends', 'NNS'), ('means', 'VBZ'), ('statistical', 'JJ'), ('pattern', 'NN'), ('learning', 'NN'), (',', ',')]
```

POS Tags are useful for lemmatization, in building NERs and extracting relations between words

Phrase Structure Rules

Phrase structure rules are rewrite rules that generate phrase structure.



Chunking and Chunk Parsing



Chunking

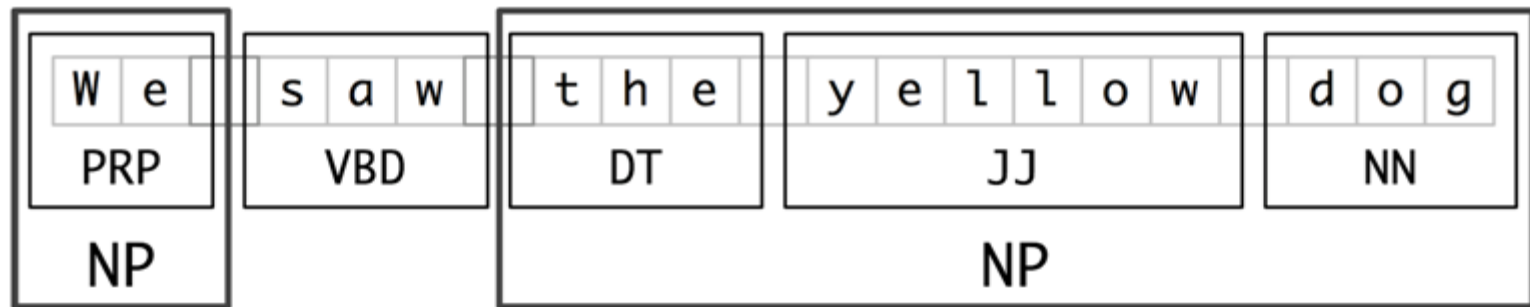
- Segmentation: identifying tokens
- Labeling: identifying the correct tag



Chunk Parsing

- Segmentation: identifying strings of tokens
- Labeling: identifying the correct chunk type

Chunking: An Example



Source: nltk.org

You can see here: Yellow(adjective), dog(noun), and the determiner are chunked together into a noun phrase (NP)

Chunking Using Python

Let's consider the sentence below:

```
sent = "The little mouse ate the fresh cheese"
```

Code

```
sent_tokens = nlk.pos_tag(word_tokenize(sent))  
sent_tokens
```

```
[('The', 'DT'),  
( 'little', 'JJ'),  
( 'mouse', 'NN'),  
( 'ate', 'VB'),  
( 'the', 'DT'),  
( 'fresh', 'JJ'),  
( 'cheeze', 'NN')]
```

NP Chunk and Parser

You will now create grammar from a noun phrase and will mention the tags you want in your chunk phrase within {}. Here you have created a regular expression matching the string.

Code

```
grammar_np = r"NP: {<DT>?<JJ>*<NN>}"
```

You will now have to parse the chunk. Therefore, you will create a chunk parser and pass your noun phrase string to it.

Code

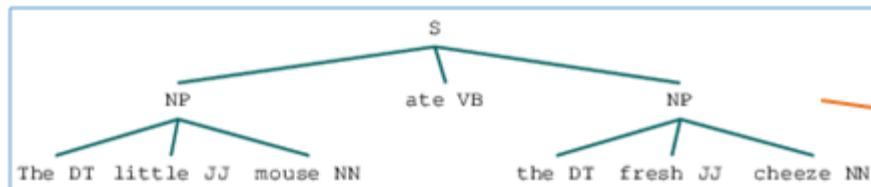
```
chunk_parser = nltk.RegexpParser(grammar_np)
```

NP Chunk and Parser (Contd.)

The parser is now ready. You will use the parse () within your chunk parser to parse your sentence.

Code

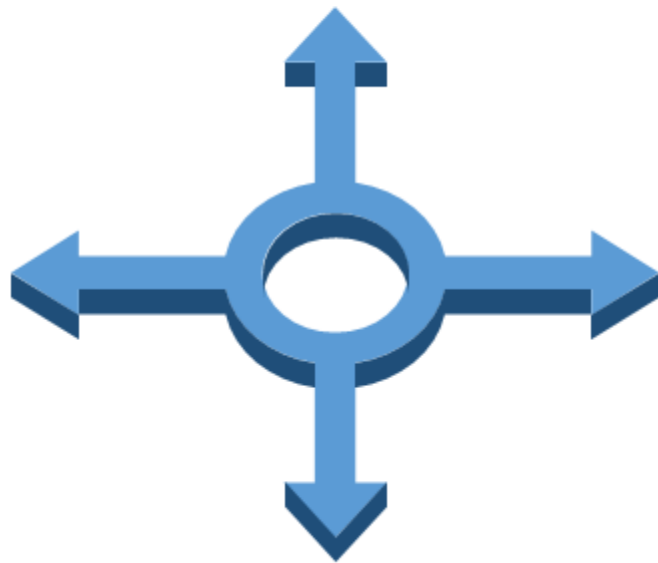
```
chunk_result = chunk_parser.parse(sent_tokens)
chunk_result
```



The tokens that matched our regular expressions are chunked together into noun phrases(NP)

Chinking

Chinking is the process of removing a sequence of tokens from a chunk



If the sequence of tokens spans an entire chunk, then the whole chunk is removed

If the sequence of tokens appears in the middle of the chunk, these tokens are removed, leaving two chunks where there was only one before

If the sequence is at the beginning or end of the chunk,

Create Chink Grammar

Consider you create a chinking grammar string containing three things:

- Chunk name
- The regular expression sequence of a chunk
- The regular expression sequence of your chink

Code

```
chink_grammar = r"""  
    chk_name: #chunk name  
  
    {<PRP>?<VB|VBD|VBZ|VBG>*<RB|RBR>?} #chunk regex sequence  
  
    }<RB>+{ #chink regex sequence - adverb  
    """
```

Inside chinking block with } {, you have created one or more adverbs

Create Chink Parser

You will now create a parser from `nltk.RegexpParser` and pass the `chink grammar` to it.

Code

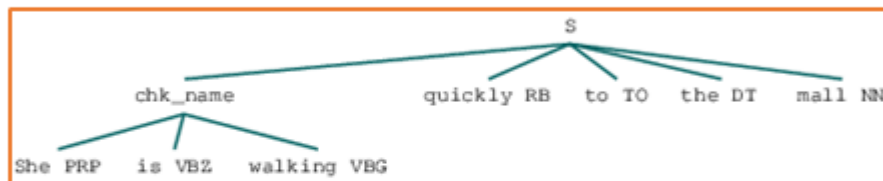
```
chink_parser = nltk.RegexpParser(chink_grammar)
```

Now, use the new chink parser to parse the tokens (`sent3`) and run the results.

Code

```
chink_parser.parse(sent_tokens3)
```

Create Chink Parser (Contd.)



While comparing the syntax tree of chink parser with that of the original chunk, you can see that the token quickly (adverb) is chinked out of the chunk

Why text doesn't work?

Text doesn't work because raw text cannot be understood by the computer . Text should be converted to numbers.

Until now, we discussed about reducing the number of words etc., but we should find some way or the other to convert the text into numbers and also the numbers should be meaningful and should represent the text

But HOW ?? ----- ☐ Feature Engineering



Bag of Words

It a count of all the words in the vocabulary in that specific document .

In other words ,

The **bag-of-words** (BOW) model is a representation that turns arbitrary text into **fixed-length vectors** by counting how many times each word appears.

Bag of Words is a text feature engineering technique which converts text into numerical form for model building.

It is applied to the text after all the preprocessing i.e after removing all the unwanted text from the data like punctuations , stops words etc.

How does Bag of Words work ?

Suppose we have three sentences :

Sentence 1 : the cat sat

Sentence 2 : the cat sat in the hat

Sentence 3 : the cat with the hat

After removing stop words we are left with :

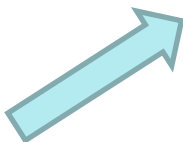
Sentence 1 : cat sat

Sentence 2 : cat sat hat


Sentence 3 : cat hat

When we convert frequency table to vectors it is known as **BOW**

The f1,f2,f3 features will be independent features for the model building.



Words	Frequency
cat	3
sat	2
hat	2



	f1	f2	f3
	cat	sat	hat
Sent 1	1	1	0
Sent 2	1	1	1
Sent 3	1	0	1



Disadvantage of Bag of Words

In this technique, the words are given equal importance and does not have any semantic difference .
Like in our case , in **Sentence 1** cat and sat are represented by 1 .

We can not classify which word is important than the other in this technique so to tackle this problem we have another feature engineering technique which is **TF-IDF**



TF - IDF

TF-IDF stands for
Term Frequency – Inverse Document Frequency .

This is a technique to quantify a word in documents,
we generally compute a weight to each word which
signifies the importance of the word in the document .

This technique converts text into numerical values ,
giving more importance to some words by multiplying
TF and IDF i.e final result is got by $TF * IDF$

How does TF- IDF work ?

Lets First understand TF i.e. **Term Frequency** .

TF = No. of repetition of words in the document / No. of words in the document

So our , TF table will look like this for our example :

Sentence 1 : cat sat

Sentence 2 : cat sat hat

Sentence 3 : cat hat



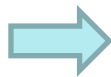
	f1	f2	f3
	cat	sat	hat
Sent 1	1/2	1/2	0
Sent 2	1/3	1/3	1/3
Sent 3	1/2	0	1/2

How does TF- IDF work ?

Now understand IDF i.e. **Inverse Document Frequency**.

IDF = $\log(\text{No. of documents} / \text{No. of documents containing words})$

So our , IDF table will look like this for our example :



Sentence 1 : cat sat

Sentence 2 : cat sat hat

Sentence 3 : cat hat

Here , no. of documents means no.of sentences.

Words	IDF
cat	$\log(3/3) = 0$
sat	$\log(3/2)$
hat	$\log(3/2)$

How does TF- IDF work ?

Finally ,

TF * IDF table in our case is :

Here we can see that the word sat is given the most importance in Sentence 1 out of other words .

Similarly in Sentence 2 and 3 as well .



	f1	f2	f3
	cat	sat	hat
Sent 1	$\frac{1}{2} * 0 = 0$	$\frac{1}{2} * \log(3/2)$	$0 * \log(3/2) = 0$
Sent 2	$\frac{1}{3} * 0 = 0$	$\frac{1}{3} * \log(3/2)$	$\frac{1}{3} * \log(3/2)$
Sent 3	$\frac{1}{2} * 0 = 0$	$0 * \log(3/2) = 0$	$\frac{1}{2} * \log(3/2)$


Limitations of Bag-of-Words

The bag-of-words model is very simple to understand and implement and offers a lot of flexibility for customization on your specific text data.

It has been used with great success on prediction problems like language modeling and documentation classification.

Nevertheless, it suffers from some shortcomings, such as:

- **Vocabulary:** The vocabulary requires careful design, most specifically in order to manage the size, which impacts the sparsity of the document representations.
- **Sparsity:** Sparse representations are harder to model both for computational reasons (space and time complexity) and also for information reasons, where the challenge is for the models to harness so little information in such a large representational space.
- **Meaning:** Discarding word order ignores the context, and in turn meaning of words in the document (semantics). Context and meaning can offer a lot to the model, that if modeled could tell the difference between the same words differently arranged (“this is interesting” vs “is this interesting”), synonyms (“old bike” vs “used bike”), and much more.




Word Embeddings

The size of the matrix generated in Bag of Words and TF-IDF is huge and sparse in nature . It becomes difficult for the model to generalize and to find the similarities and context between the words.

For overcoming this problem , **Word embeddings** concept is used .

Word embeddings are basically a form of word representation which give us a way to use an efficient, dense representation in which similar words have a similar encoding.



Word Embedding technique – word2vec

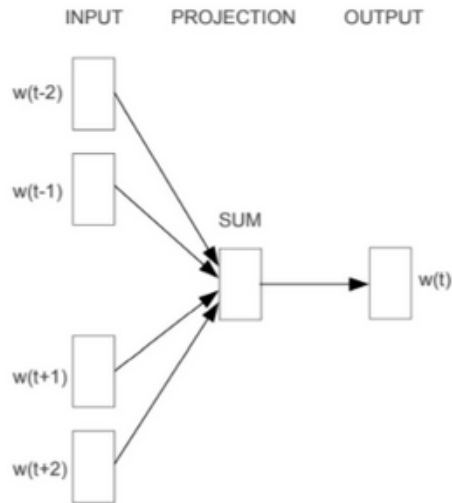
word2vec is the deep learning Google framework to train word embeddings. It will use all the words of the whole corpus and predict the nearby words. It will create a vector for all the words present in the corpus in a way so that the context is captured.

There are mainly 2 types of word2vec Model.

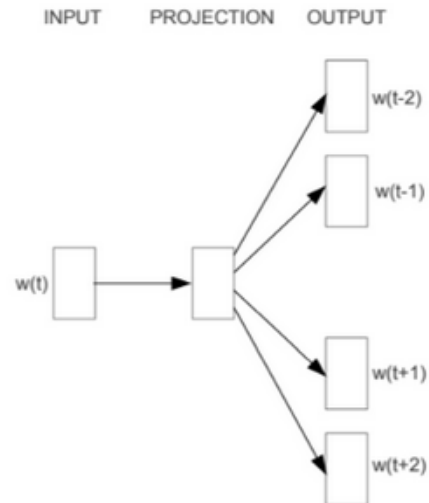
- Skip-Gram
- Continuous Bag of Words (CBOW)

Continuous Bag of Words Model (CBOW) and Skip-gram

Both are architectures to learn the underlying word representations for each word by using neural networks.



CBOW



Skip-gram



In the **CBOW** model, the distributed representations of context (or surrounding words) are combined to **predict the word in the middle**.

While in the **Skip-gram** model, the distributed representation of the input word is used to **predict the context**.

Continuous Bag of Words (CBOW)

CBOW predicts target words (e.g. mat) from the surrounding context words ('the cat sits on the').


In other words , CBOW predicts the probability of a word to occur given the words surrounding it.

Suppose we have a sentence , **The cat sits on the mat.**

Firstly we will convert all words into one hot encoding and then predict the target word using the surrounded words . For ex in the case we use **The cat sits on** to predict the word **mat** .

It is not necessary to use all the words for prediction , the number of words used for prediction is called window size .

Example of CBOW

Surrounded Words	Target word	Window size
The cat sits on	mat	4
cat sits on	mat	3
sits on mat	cat	3
sits mat	cat 	2

Statistically, it has the effect that CBOW smoothes over a lot of the distributional information (by treating an entire context as one observation). For the most part, this turns out to be a useful thing for smaller datasets.

Skip-gram

Skip-gram predicts surrounding context words from the target words (inverse of CBOW).

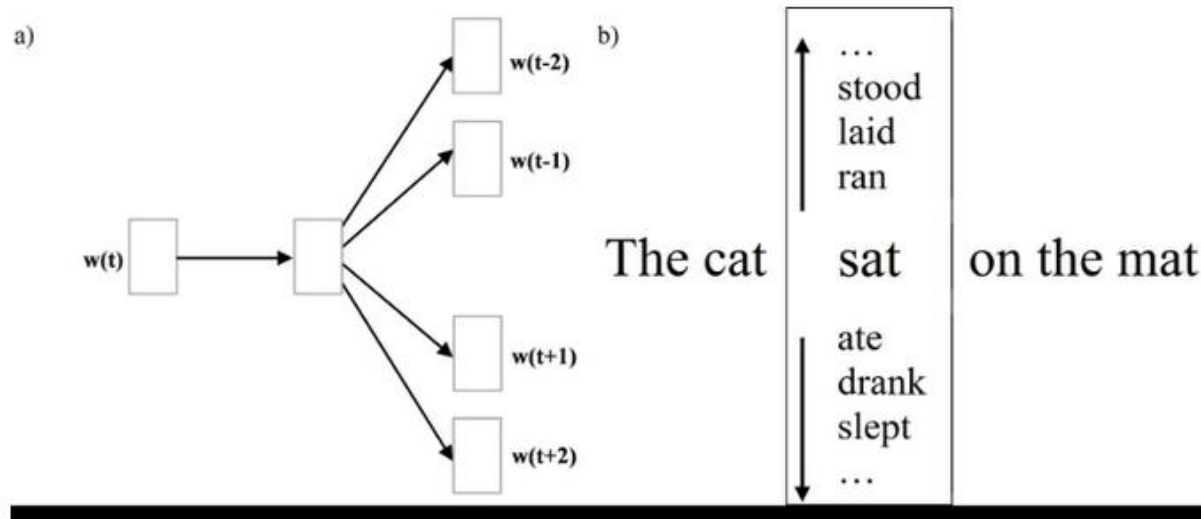
The working of the skip-gram model is quite similar to the CBOW but there is just a difference in the architecture

Let us take a small sentence and understand how it actually works. Each sentence will generate a target word and context, which are the words nearby.

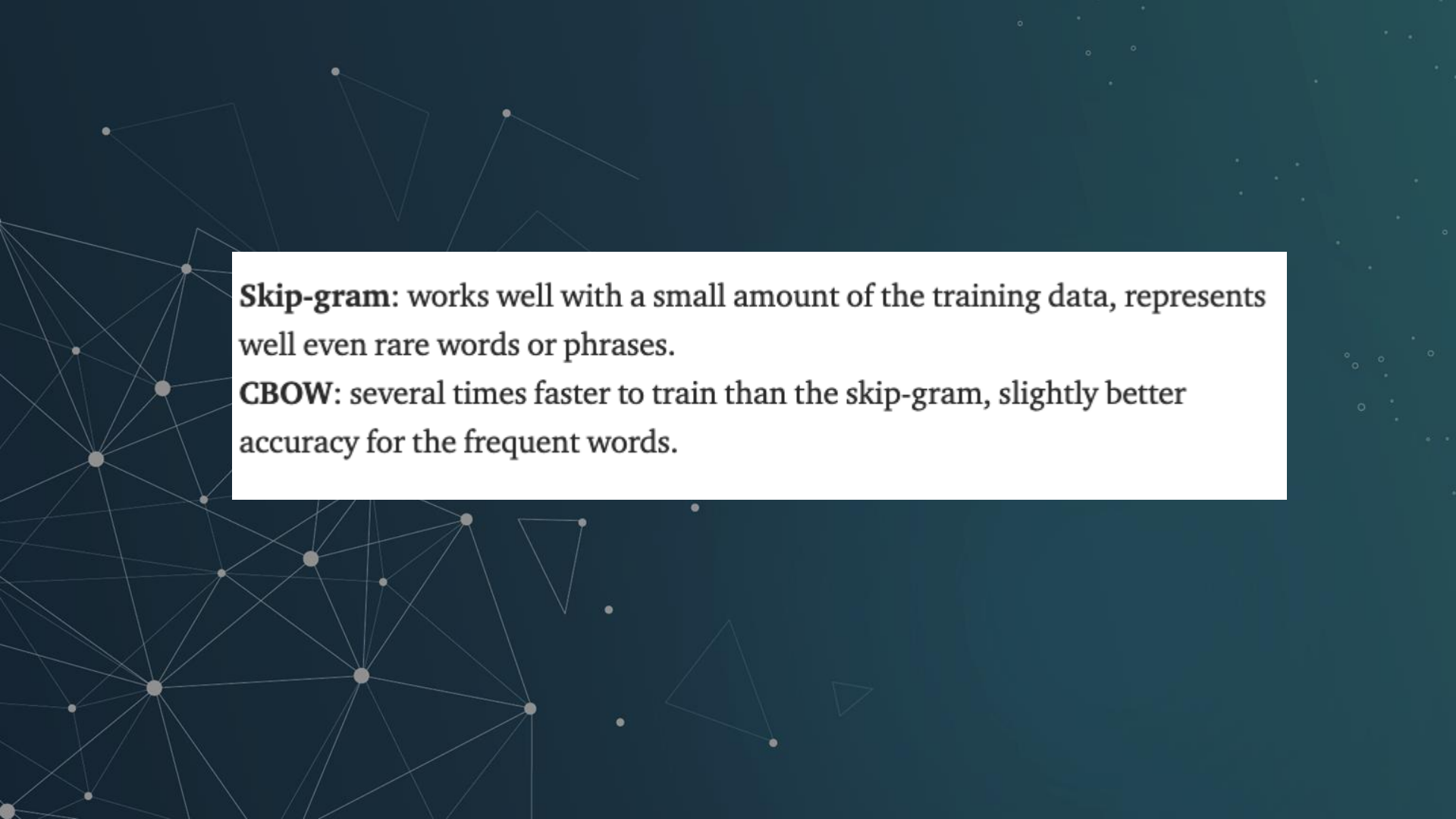
Text = "I love NLP and I will learn NLP in 2 months"

	Target word	Context
I love NLP	I	love, NLP
I love NLP and	love	love, NLP, and
I love NLP and I will learn	NLP	I, love, and, I
...
in 2 months	month	in, 2

Skip-gram is used to predict the context word for a given target word. It's reverse of CBOW algorithm. Here, target word is input while context words are output. As there is more than one context word to be predicted which makes this problem difficult.



skip-gram example

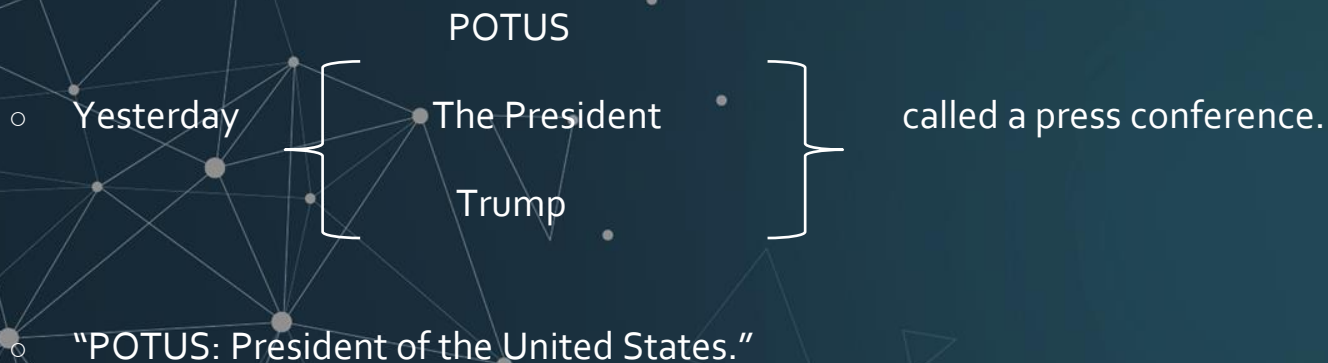


Skip-gram: works well with a small amount of the training data, represents well even rare words or phrases.

CBOW: several times faster to train than the skip-gram, slightly better accuracy for the frequent words.

Vector Embedding of Words

- A word is represented as a **vector**.
- Word embeddings depend on a notion of **word similarity**.
 - Similarity is computed using cosine.
- A very useful definition is paradigmatic similarity:
 - **Similar words** occur in **similar contexts**. They are **exchangeable**.



Vector Embedding of Words

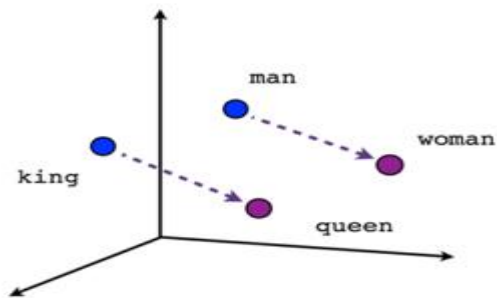
Traditional Method - Bag of Words Model

- Either uses one hot encoding.
 - Each word in the vocabulary is represented by one bit position in a HUGE vector.
 - For example, if we have a vocabulary of 10000 words, and "Hello" is the 4th word in the dictionary, it would be represented by: 0 0 0 1 0 0 0 0 0
- Or uses document representation.
 - Each word in the vocabulary is represented by its presence in documents.
 - For example, if we have a corpus of 1M documents, and "Hello" is in 1th, 3th and 5th documents **only**, it would be represented by: 1 0 1 0 1 0 0 0 0
- Context information is not utilized.

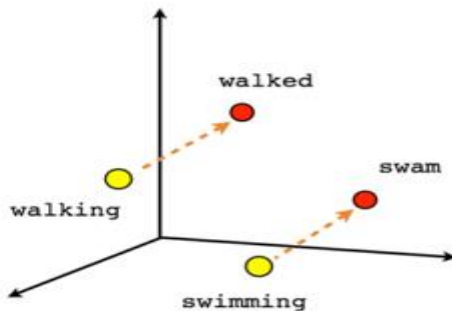
Word Embeddings

- Stores each word in as a point in space, where it is represented by a dense vector of fixed number of dimensions (generally 300) .
- Unsupervised, built just by reading huge corpus.
- For example, "Hello" might be represented as : [0.4, -0.11, 0.55, 0.3 . . . 0.1, 0.02].
- Dimensions are basically projections along different axes, more of a mathematical concept.

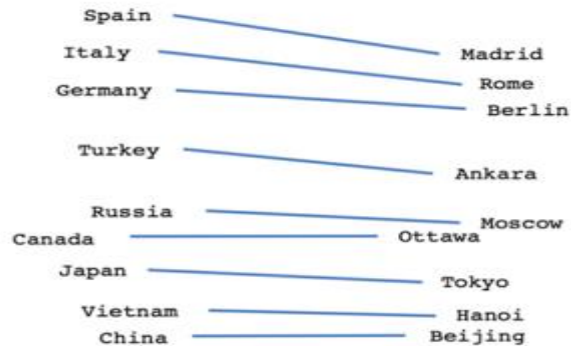
Example



Male-Female



Verb tense



Country-Capital

\bullet $\text{vector}[\text{Queen}] \approx \text{vector}[\text{King}] - \text{vector}[\text{Man}] + \text{vector}[\text{Woman}]$

\bullet $\text{vector}[\text{Paris}] \approx \text{vector}[\text{France}] - \text{vector}[\text{Italy}] + \text{vector}[\text{Rome}]$

Applications of Word Vectors

- Word Similarity
- Machine Translation
- Part-of-Speech and Named Entity Recognition
- Relation Extraction
- Sentiment Analysis
- Co-reference Resolution

- Chaining entity mentions across multiple documents - can we find and unify the multiple contexts in which mentions occurs?

Clustering

- Words in the same class naturally occur in similar contexts, and this feature vector can directly be used with any conventional clustering algorithms (K-Means, agglomerative, etc). Human doesn't have to waste time hand-picking useful word features to cluster on.

- Semantic Analysis of Documents

- Build word distributions for various topics, etc.

Vector Embedding of Words

- Three main methods described in the talk :
 - Latent Semantic Analysis/Indexing (1988)
 - Term weighting-based model
 - Consider occurrences of terms at document level.
 - Word2Vec (2013)
 - Prediction-based model.
 - Consider occurrences of terms at context level.
 - GloVe (2014)
 - Count-based model.
 - Consider occurrences of terms at context level.
 - ELMo (2018)
 - Language model-based.
 - A different embedding for each word for each task.

Word2Vec

word2Vec: Local contexts

- Instead of entire documents, *Word2Vec* uses words k positions away from each center word.
 - These words are called **context words**.
- Example for $k=3$:
 - "It was a bright cold **day** in April, and the clocks were striking".
 - **Center word: red (also called focus word).**
 - **Context words: blue (also called target words).**
- Word2Vec considers all words as center words, and all their context words.

Word2Vec: Data generation (window size = 2)

- Example: d1 = "king brave man". d2 = "queen beautiful women"

word	Word one hot encoding	neighbor	Neighbor one hot encoding
king	[1,0,0,0,0,0]	brave	[0,1,0,0,0,0]
king	[1,0,0,0,0,0]	man	[0,0,1,0,0,0]
brave	[0,1,0,0,0,0]	king	[1,0,0,0,0,0]
brave	[0,1,0,0,0,0]	man	[0,0,1,0,0,0]
man	[0,0,1,0,0,0]	king	[1,0,0,0,0,0]
man	[0,0,1,0,0,0]	brave	[0,1,0,0,0,0]
queen	[0,0,0,1,0,0]	beautiful	[0,0,0,0,1,0]
queen	[0,0,0,1,0,0]	women	[0,0,0,0,0,1]
beautiful	[0,0,0,0,1,0]	queen	[0,0,0,1,0,0]
beautiful	[0,0,0,0,1,0]	women	[0,0,0,0,0,1]
woman	[0,0,0,0,0,1]	queen	[0,0,0,1,0,0]
woman	[0,0,0,0,0,1]	beautiful	[0,0,0,0,1,0]

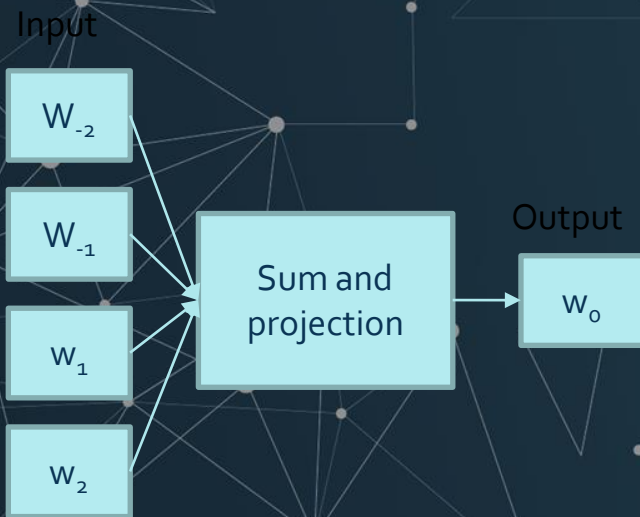
Word2Vec: Data generation (window size = 2)

- Example: d1 = "king brave man". d2 = "queen beautiful women"

word	Word one hot encoding	neighbor	Neighbor one hot encoding
king	[1,0,0,0,0,0]	brave	[0,1,1,0,0,0]
		man	
brave	[0,1,0,0,0,0]	king	[1,0,1,0,0,0]
		man	
man	[0,0,1,0,0,0]	king	[1,1,0,0,0,0]
		brave	
queen	[0,0,0,1,0,0]	beautiful	[0,0,0,0,1,1]
		women	
beautiful	[0,0,0,0,1,0]	queen	[0,0,0,1,0,1]
		women	
woman	[0,0,0,0,0,1]	queen	[0,0,0,1,1,0]
		beautiful	

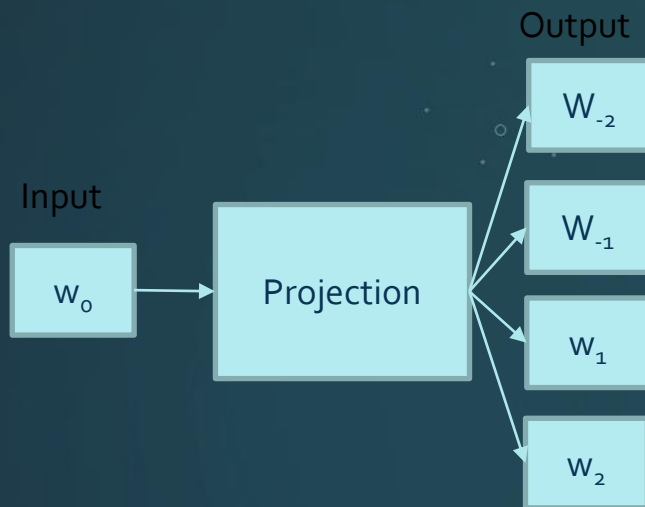
Word2Vec: main context representation models

Continuous Bag of Words (CBOW)



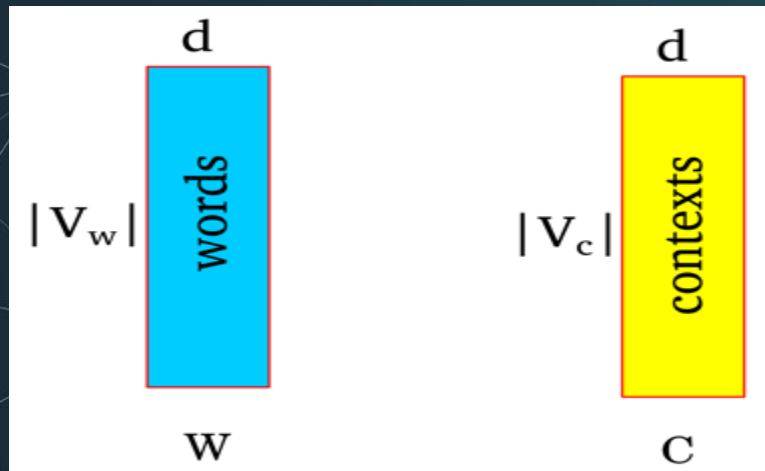
- Word2Vec is a predictive model.
- Will focus on Skip-Ngram model

Skip-Ngram



How does word2Vec work?

- Represent each word as a d dimensional vector.
- Represent each context as a d dimensional vector.
- Initialize all vectors to random weights.
- Arrange vectors in two matrices, W and C .

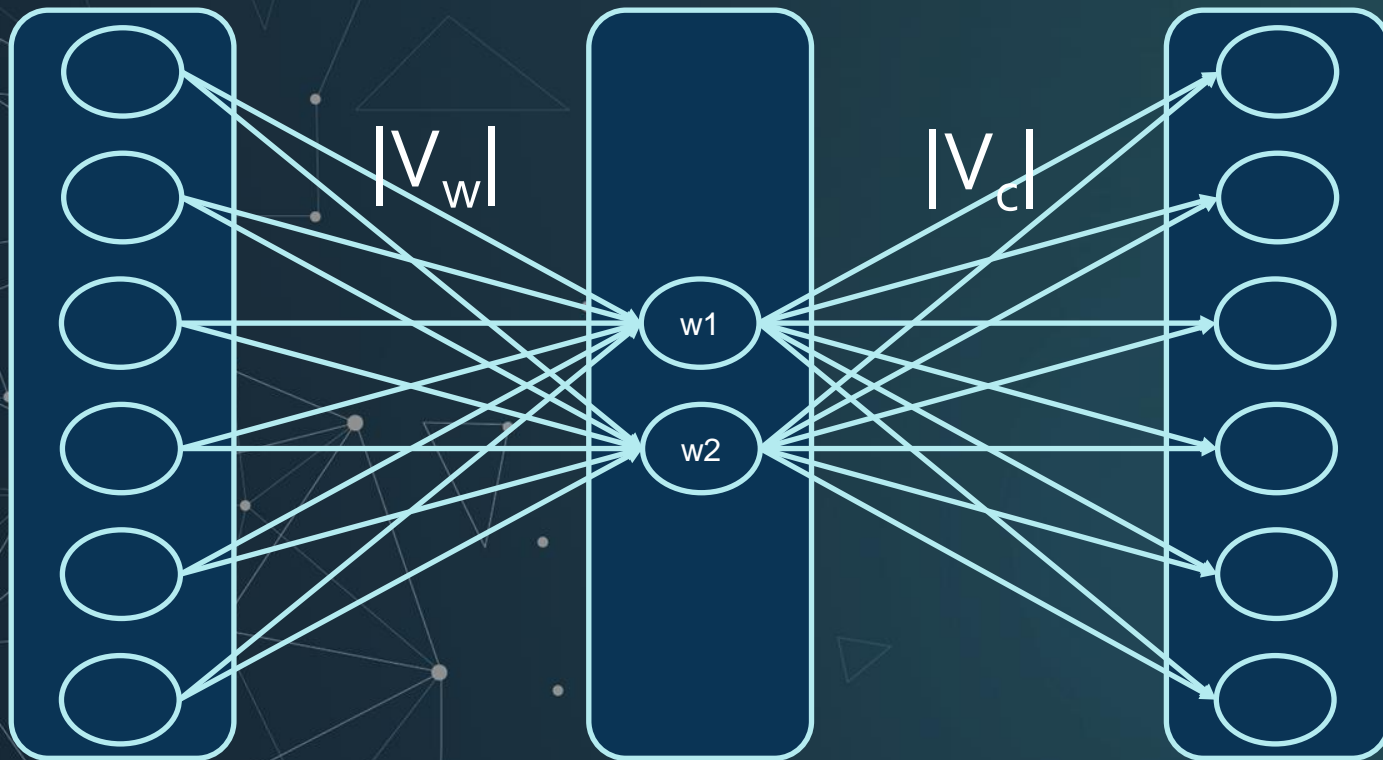


Word2Vec : Neural Network representation

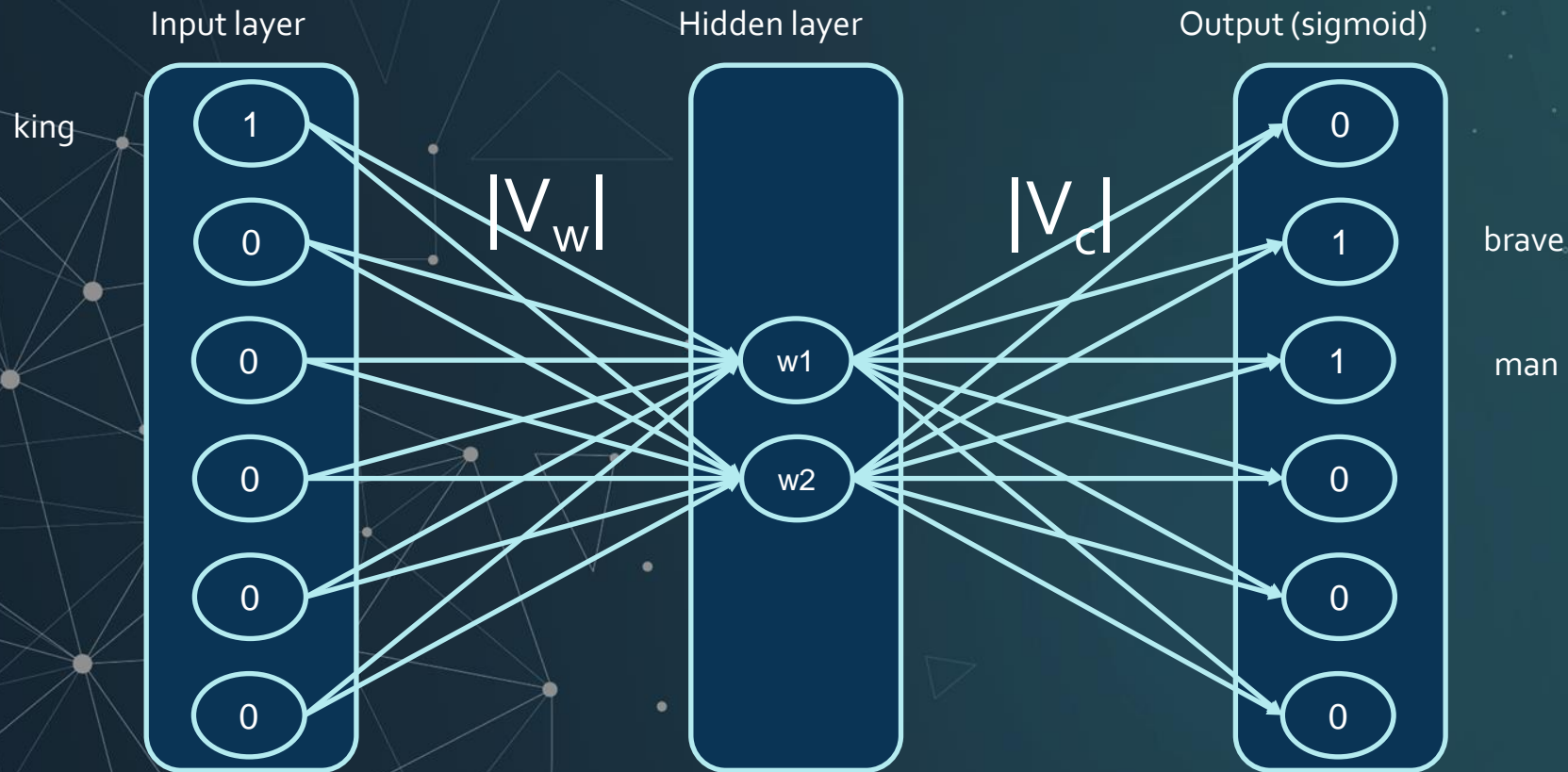
Input layer

Hidden layer

Output (sigmoid)



Word2Vec : Neural Network representation

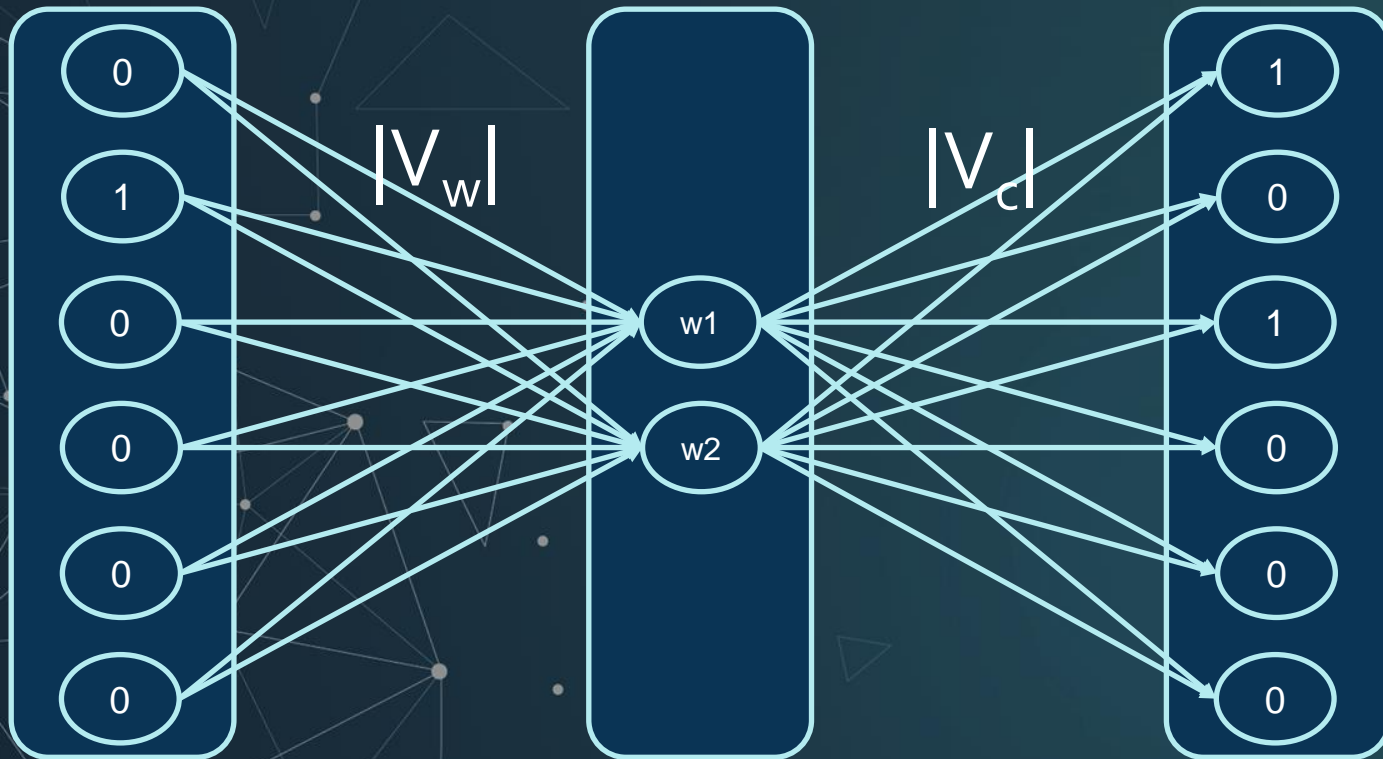


Word2Vec : Neural Network representation

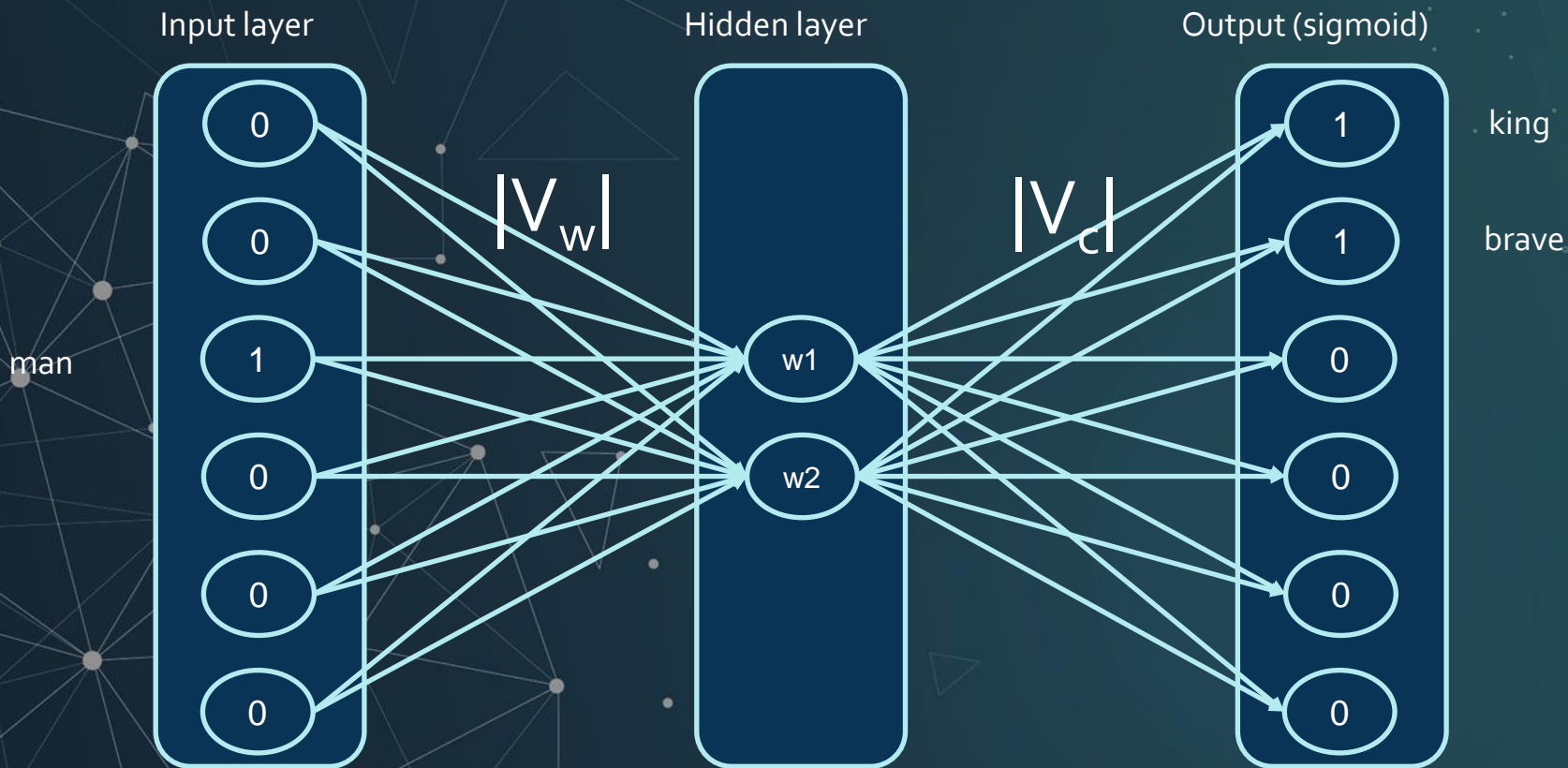
Input layer

Hidden layer

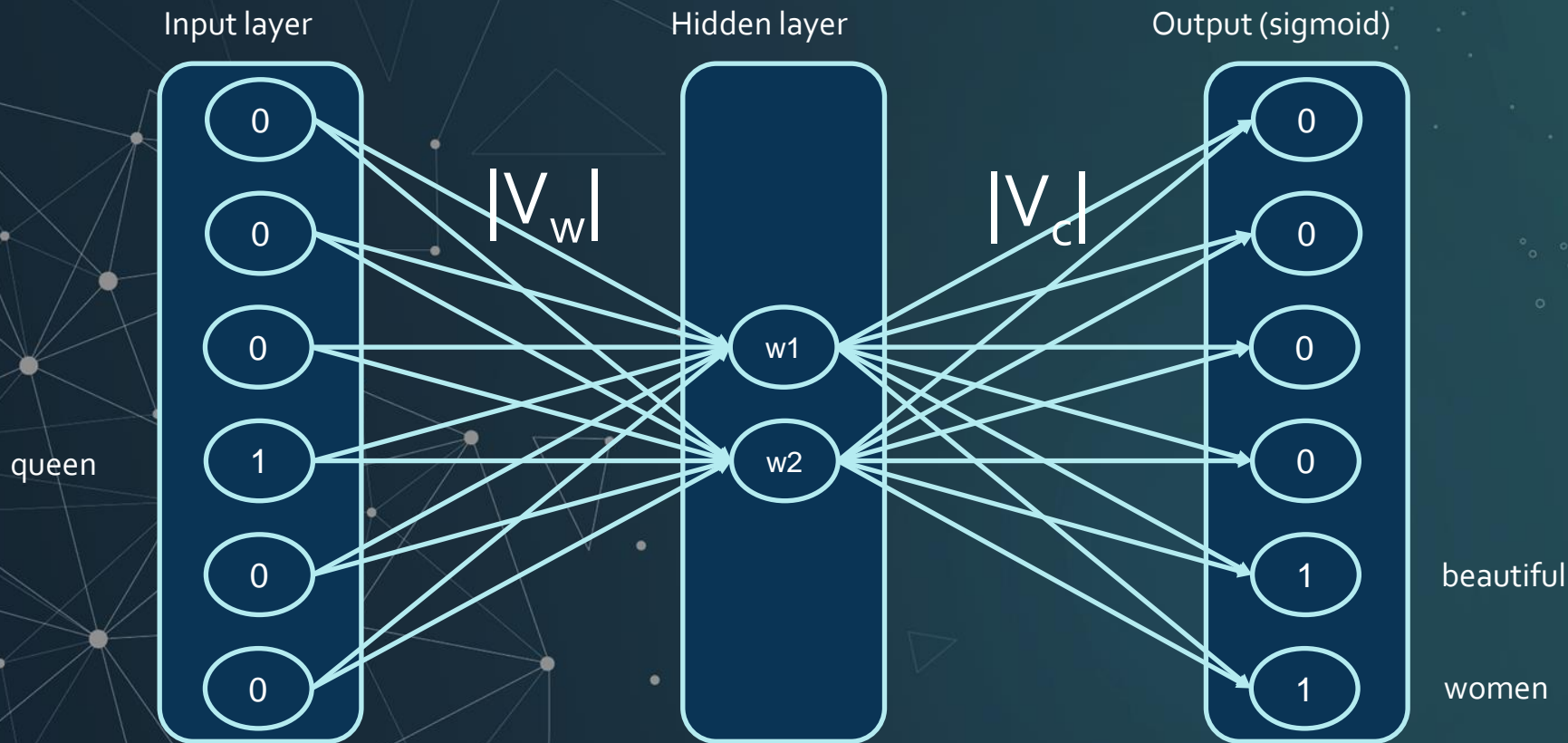
Output (sigmoid)



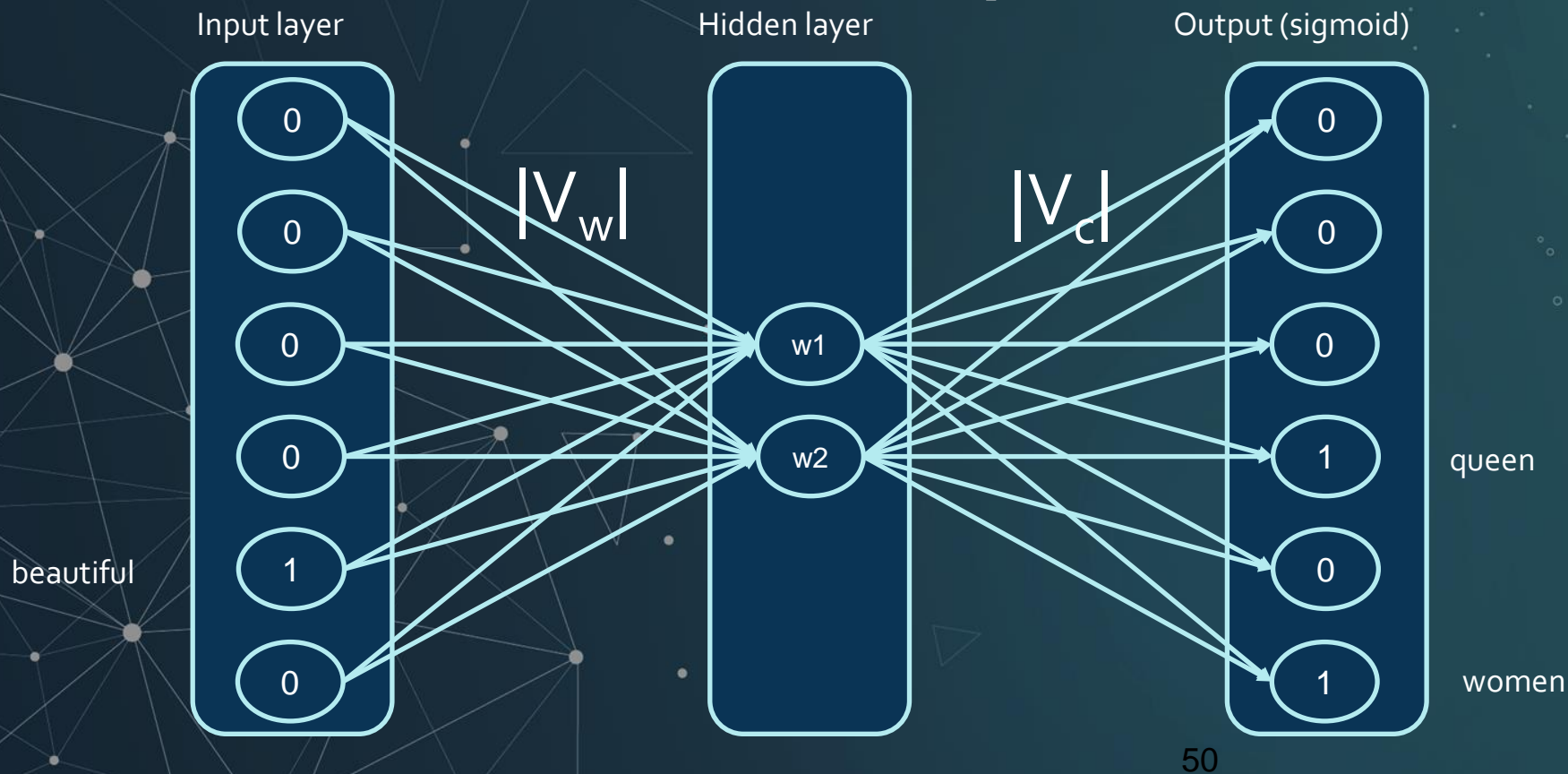
Word2Vec : Neural Network representation



Word2Vec : Neural Network representation



Word2Vec : Neural Network representation

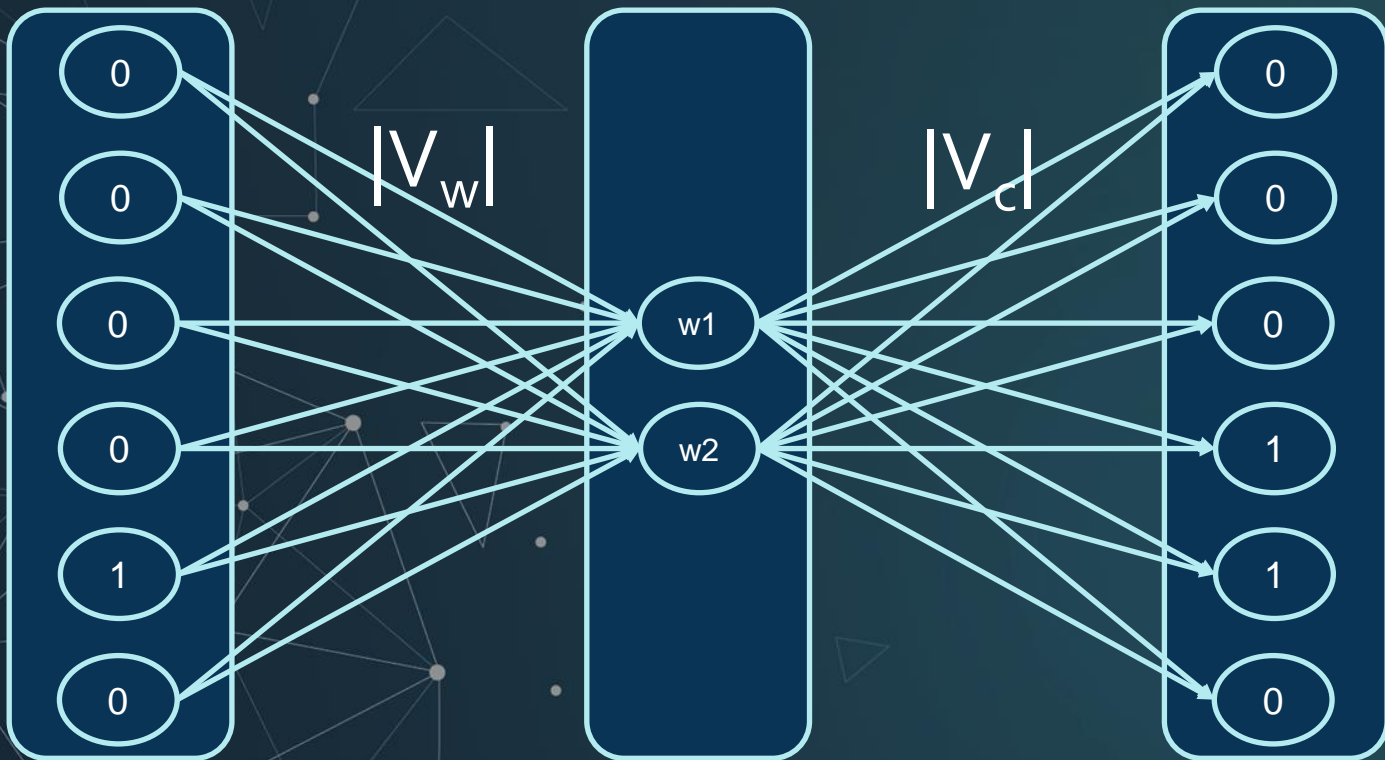


Word2Vec : Neural Network representation

Input layer

Hidden layer

Output (sigmoid)



Skip-Ngram: Example

- While more text:
 - Extract a word window:

A	springer	is	[a	cow	or	heifer	close	to	calving]
				c_1	c_2	c_3	w	c_4	c_5	c_6	

- Try setting the vector values such that:

- $\sigma(w \cdot c_1) + \sigma(w \cdot c_2) + \sigma(w \cdot c_3) + \sigma(w \cdot c_4) + \sigma(w \cdot c_5) + \sigma(w \cdot c_6)$ is high!

- Create

[a	cow	or	comet	close	to	calving]
	c_1	c_2	c_3	w'	c_4	c_5	c_6	

- Try setting the vector values such that:

- $\sigma(w' \cdot c_1) + \sigma(w' \cdot c_2) + \sigma(w' \cdot c_3) + \sigma(w' \cdot c_4) + \sigma(w' \cdot c_5) + \sigma(w' \cdot c_6)$ is low!

Skip-Ngram: How to select negative samples?

- Can sample using frequency.
 - Problem: will sample a lot of stop-words.
- Mikolov et al. proposed to sample using:

$$p(w_i) = \frac{f(w_i)^{3/4}}{\sum_j f(w_j)^{3/4}}$$

- Not theoretically justified, but works well in practice!

Relations Learned by Word2Vec

- A relation is defined by the vector displacement in the first column. For each start word in the other column, the closest displaced word is shown.

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

GloVe: Global Vectors for Word Representation

GloVe: Global Vectors for Word Representation

- While word2Vec is a predictive model — learning vectors to improve the predictive ability, **GloVe is a count-based model.**
- Count-based models learn vectors by doing dimensionality reduction on a **co-occurrence counts matrix**.
 - Factorize this matrix to yield a lower-dimensional matrix of words and features, where each row yields a vector representation for each word.
 - The counts matrix is preprocessed by normalizing the counts and log-smoothing them.

Difference to other methods

	Source	Nearest Neighbors
GloVe	<u>play</u>	playing, game, games, played, players, plays, player, Play, football, multiplayer
biLM	Chico Ruiz made a spectacular play on Alusik 's grounder { . . . }	Kieffer , the only junior in the group , was commended for his ability to hit in the clutch , as well as his all-round excellent play .
	Olivia De Havilland signed to do a Broadway play for Garson { . . . }	{ . . . } they were actors who had been handed fat roles in a successful play , and had talent enough to fill the roles competently , with nice understatement .