

Chapter 1

Word Representations

These slides are mostly from the Stanford University course cs224n (R. Socher, <http://web.stanford.edu/class/cs224n>).

Content of this Chapter

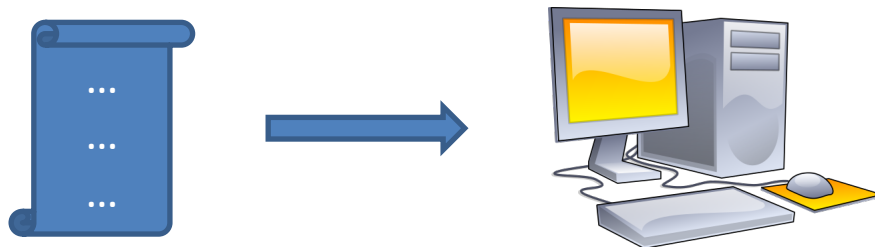
1. Modelling Text
2. Word Meaning
3. Word2Vec
4. Short Introduction to GloVe and FastText
5. Pre-Processing

1.1 Modelling Text

- Our ultimate goal
- Ways of representing documents/sentences/...
 - Bag of Words
 - Sequence of words
 - Document Embeddings

Modelling Documents

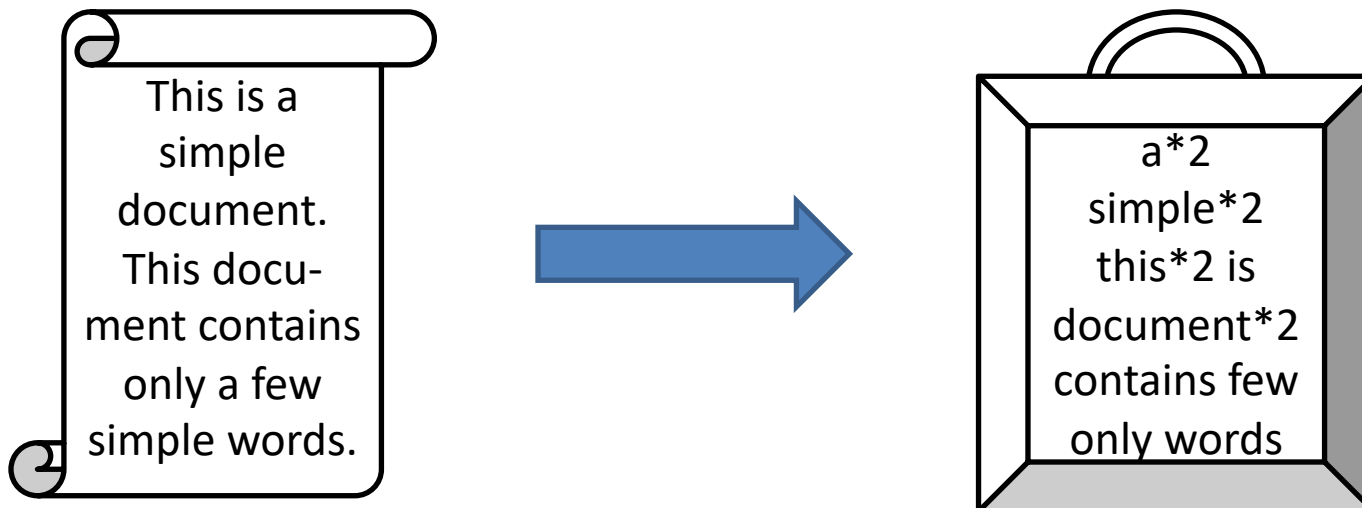
- Basic task in all NLP applications
- „Documents“ can be all kinds of texts
 - Sentences
 - Paragraphs
 - Tweets
 - Books
 - ...
- Need to represent these in a computer-readable way!



Modelling Documents – Bag of Words

- A document consists of words
→ Use these to represent the document!
- Very easy approach:
Use only the words, discard their order
- Known as **Bag of Words**

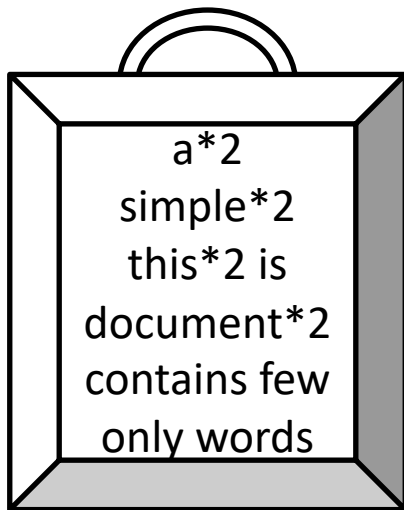
Known from
IR/Text
Mining



Modelling Documents – Bag of Words

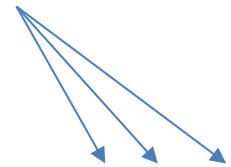
- Convert the „bag“ to a vector
- Represent every word by an integer
 - $a \rightarrow 1$, simple $\rightarrow 2$, this $\rightarrow 3$, ...
- 1. position in the vector: Count of „a“ in the document
- 2. position in the vector: Count of „simple“ in the document

Known from
IR/Text
Mining



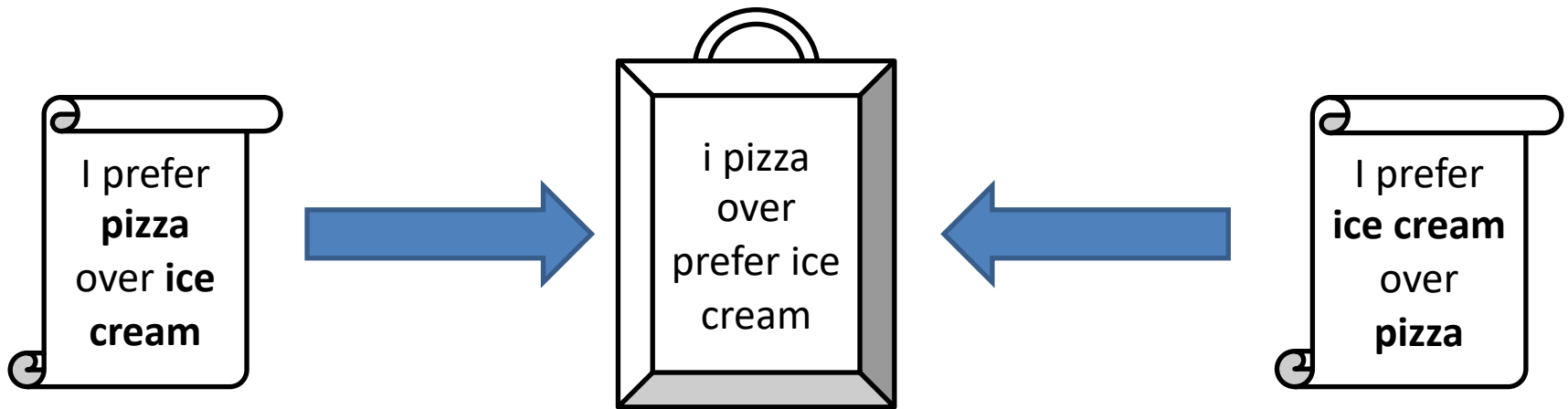
$d = [2, 2, 2, 1, 2, 1, 1, 1, 1, 0, 0, \dots]$

Other words that appear
in other documents (all 0)



Modelling Documents – Bag of Words

- Works rather well...
- ...but loses a lot of information!



Modelling Documents – Sequence of Words

- We need to consider word order!
→ A document d is an ordered sequence of words
- $d = (w_1, w_2, \dots, w_n)$
- Now we need to find a good representation for words

1.2 Representing Words

- What is the meaning of a word?
- How do we represent this meaning?
- What are the limitations of classical representations?

The meaning of a word

- Definition: **meaning** (Webster dictionary)
 - the idea that is represented by a word, phrase, etc.
 - the idea that a person wants to express by using words, signs, etc.
 - the idea that is expressed in a work of writing, art, etc.

Representing Words — The traditional way

Representing Words – The traditional way

- In traditional NLP, we regard words as discrete symbols: **fish**, **shark**, **moose**
- Words are represented as *one-hot-vectors*

Fish = [0 0 0 0 0 0 1 0 0 0]

Shark = [0 0 0 1 0 0 0 0 0 0]

Moose = [0 0 0 0 0 0 0 1 0 0]



As many entries as words in the vocabulary
e.g. 500 000

Similar to
before!*

- * Summing up these vectors yields a bag of words representation

Representing Words – Problems of the traditional way

- Some words are more similar than others!
- For example:
 - A fish is much more similar to a shark than to a moose.
 - But:
 - $\text{sim}(\text{fish}, \text{shark}) = \cos([0,0,0,0,0,0,1,0,0,0], [0,0,0,1,0,0,0,0,0,0]) = 0$
 - $\text{sim}(\text{fish}, \text{moose}) = \cos([0,0,0,0,0,0,1,0,0,0], [0,0,0,0,0,0,1,0,0,0]) = 0$
 - The vectors do not reflect this!
- For one-hot vectors, there is no notion of word similarity

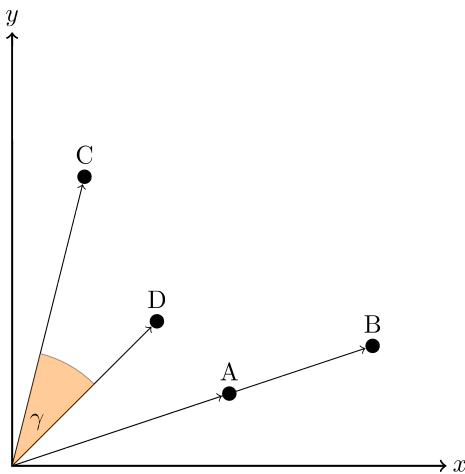
[See next slide](#)

Representing Words – Similarity

- We used *cos* on the previous slide to determine similarity
- Why?

→ **Cosine-Similarity** is a popular measure for word similarity

$$sim_{cos}(x, y) = \frac{\sum_{i=1}^N x_i \cdot y_i}{\sqrt{\sum_{i=1}^N x_i^2} \cdot \sqrt{\sum_{i=1}^N y_i^2}}$$



Improving Word Representations

- Ways to model word similarity
 - Using external sources (WordNet, ...)
 - Using the context!

Representing Words – WordNet

- A resource containing lists of **synonym sets** and **hypernyms**

e.g. synonym sets containing “good”:

```
from nltk.corpus import wordnet as wn
for synset in wn.synsets("good"):
    print("(%s)" % synset.pos(),
          print ", ".join([l.name() for l in synset.lemmas()]])
```

```
(adj) full, good
(adj) estimable, good, honorable, respectable
(adj) beneficial, good
(adj) good, just, upright
(adj) adept, expert, good, practiced,
proficient, skillful
(adj) dear, good, near
(adj) good, right, ripe
...
(adv) well, good
(adv) thoroughly, soundly, good
(n) good, goodness
(n) commodity, trade good, good
```

e.g. hypernyms of “panda”:

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(panda.closure(hyper))
```

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```


Representing Words – WordNet's Problems

- Missing nuances
 - „proficient“ is listed as a synonym for „good“
 - But a „proficient book“ does not make sense, while a „good book“ certainly does
- Incompleteness
 - Any manually crafted list will always be incomplete
 - Missing neologisms, slang words, ...
- Possible bias
 - Manually crafted resources often contain some form of bias from their creators
- How to use it?
 - Does not directly provide vector representation

Representing Words – The Modern Way™


“You shall know a word by the company it keeps”

(J. R. Firth 1957: 11)

Representing Words – The Modern Way™

- Core idea: **A word's meaning is given by the words that frequently appear close-by**
 - When a word w appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).
 - Use the many contexts of w to build up a representation of w

...government debt problems turning into	banking	crises as happened in 2009...
...saying that Europe needs unified	banking	regulation to replace the hodgepodge...
...India has just given its	banking	system a shot in the ...


 These **context words** will represent **banking**

Representing Words – The Modern Way™

- A word is now represented by a **dense** vector
- Words with similar meaning will have similar vectors

$$\textit{linguistics} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

1.3 Word2Vec

- A framework to calculate word embeddings on very large corpora

Representing Words – Word2Vec

- **Word2Vec** (Mikolov et al. 2013) is a framework for learning word vectors.
- Idea:
 - We have a large corpus of text
 - Every word in the corpus' fixed vocabulary should be represented by a vector
 - Go through each position w in the text, which has a center word w_t and context ("outside") words $w_{t\pm j}$ ($j > 0$)
 - Use the similarity of the word vectors for w_t and $w_{t\pm j}$ to calculate the probability of $w_{t\pm j}$ given w_t (or vice versa)
 - Keep adjusting the word vectors to maximise this probability

Representing Words – Word2Vec

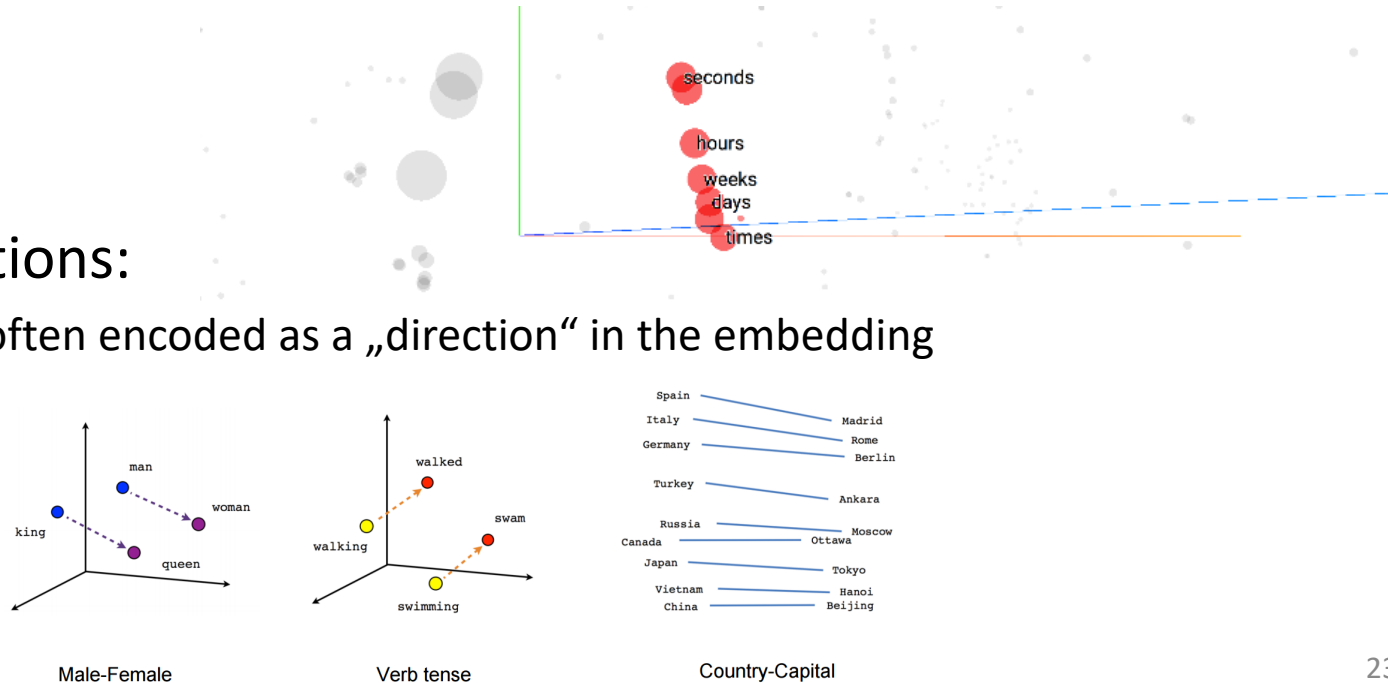
- Embeddings created by Word2Vec represent both **semantic similarity** and **semantic relations**!

- Semantic similarity:

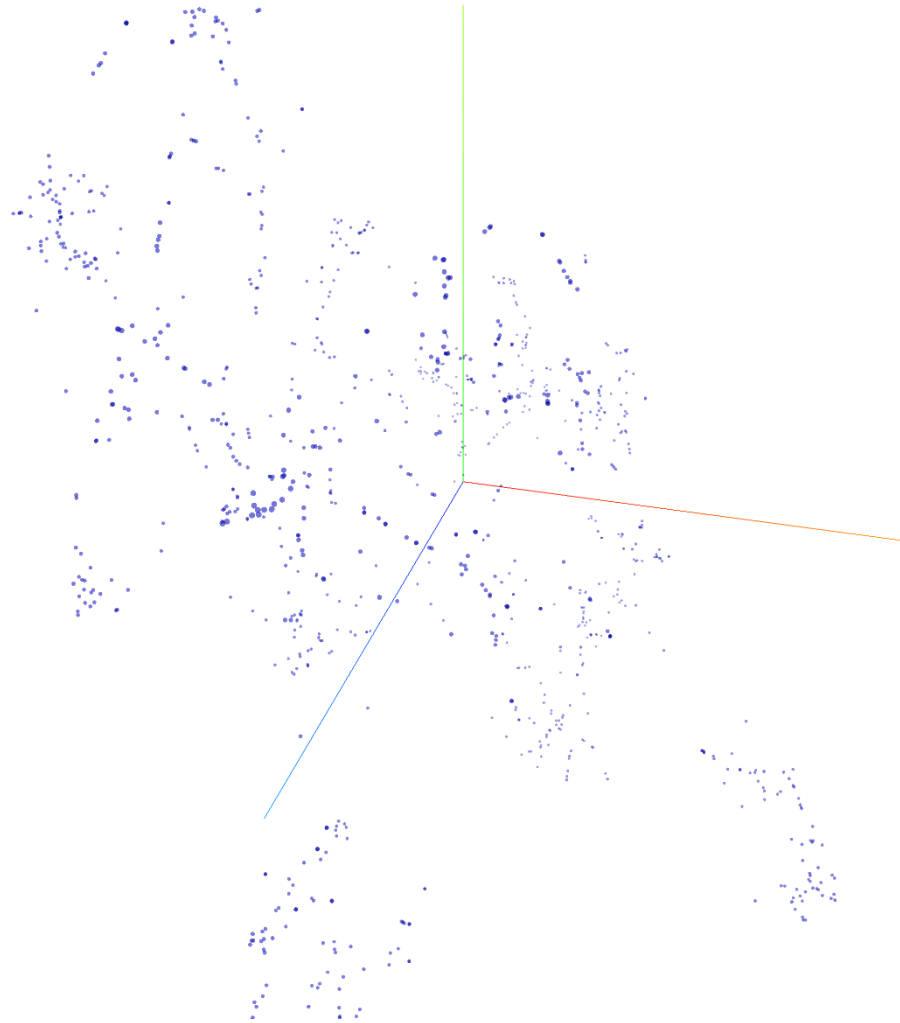
- Words with similar embeddings have similar meaning

- Semantic relations:

- Relations are often encoded as a „direction“ in the embedding



Representing Words – Demo

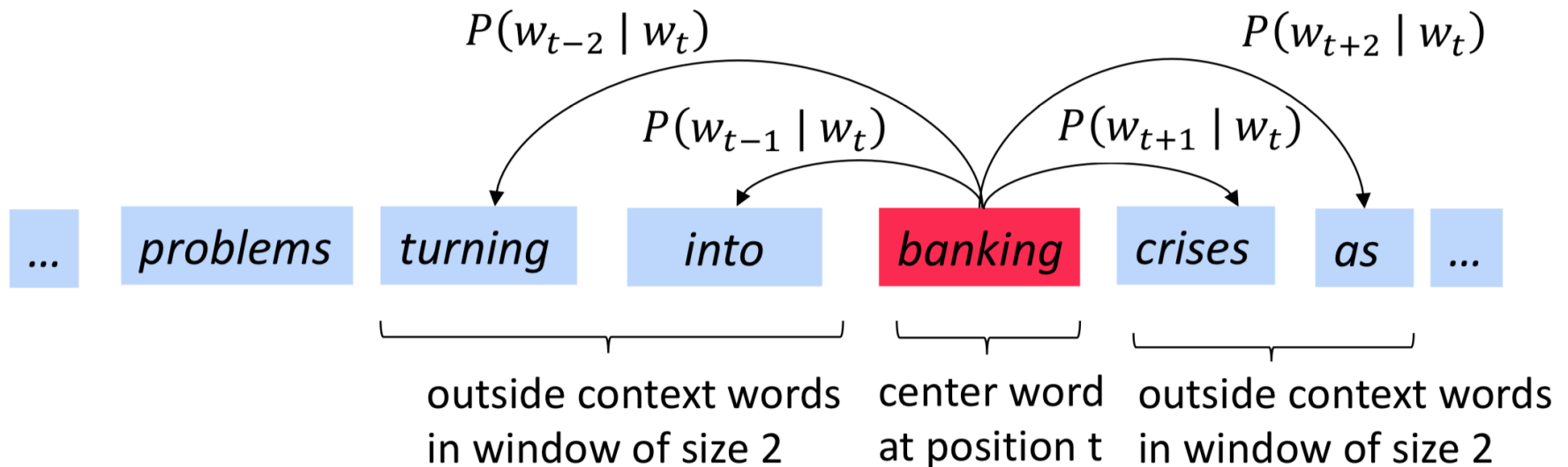


Representing Words – Word2Vec

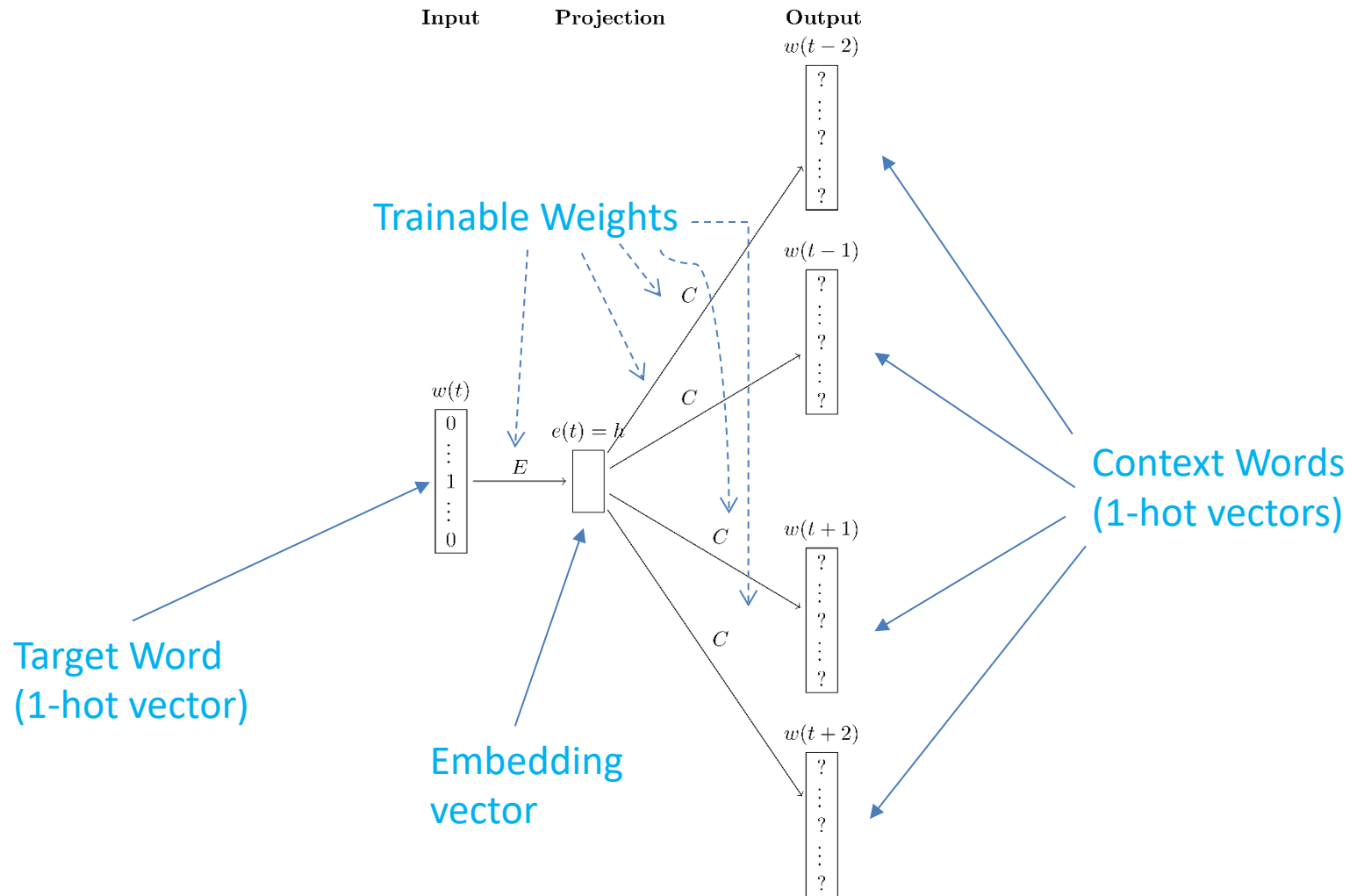
- Two different models exist:
 - CBOW
 - Predict the **center word** from the **context words**
 - Skip-gram
 - Predict the **context words** from the **center word**
- Both work similarly well
- We focus on skip-gram here

Representing Words – Word2Vec

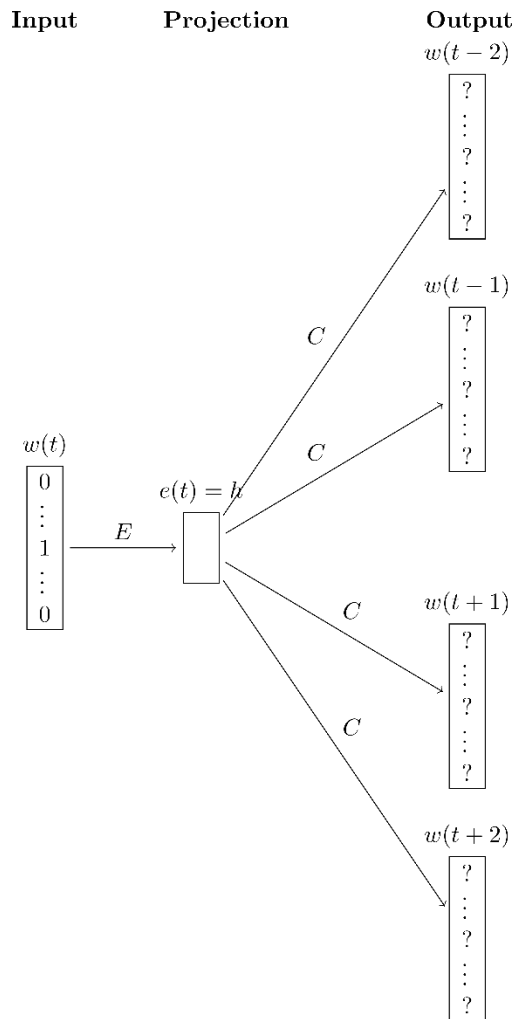
- Example windows and process for computing $P(w_{t+j} | w_t)$



Word2Vec as a Neural Network (Skip-gram)



Word2Vec as a Neural Network



Example:

- Let „fish“ be the word with index 23

23rd position

$$\rightarrow w_{fish} = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \quad \left. \vphantom{\begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}} \right\} \text{length } d$$

$$\rightarrow e_{fish} = E \cdot w_{fish} = \text{the 23rd line of } E$$

$$\rightarrow out = e_{fish} \cdot C$$

Multiplication with 1-hot vector = line selection!

Note: out is a vector of dimension d again!
 Interpretation: unnormalized probabilities for each word in the vocabulary to appear as a context word of w_{fish} .

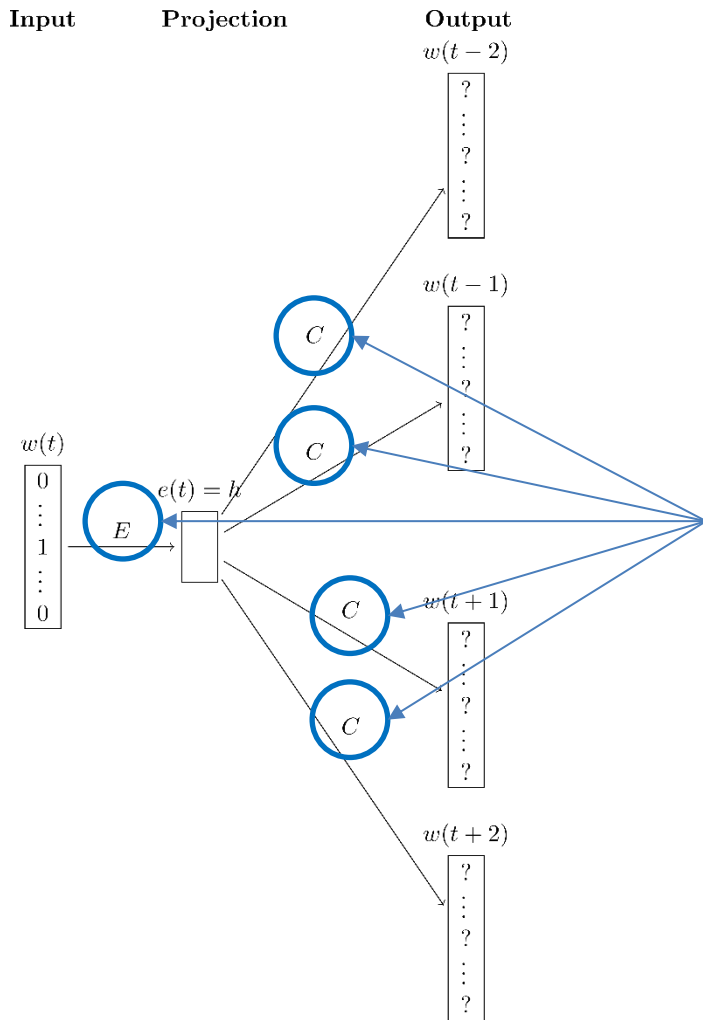
Probabilities are normalised by a **softmax**.

The Softmax Function

- The softmax is a very frequent function in Deep Learning
- It maps arbitrary values x_i to a probability distribution p_i
 → Transforms the output of a neural network into class probabilities!
- Generally it has this form:

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^N \exp(x_j)} = p_i$$

Word2Vec as a Neural Network



How do we train these weights?

Word2Vec as an Equation

- Translate the network to an equation with parameters that we can optimise!
- Two sets of parameters/embeddings:
 - e_x are the vectors for center words, stored in E
 - c_x are the vectors for context words, stored in C
- For a center word w_t and a context word w_{t+j} , the probability of w_{t+j} being in the context of w_t is:

$$P(w_{t+j}|w_t) = \frac{\exp(e_t \cdot c_{t+j}^T)}{\sum_{w \in V} \exp(e_t \cdot c_w^T)}$$

This is basically the network
— just a vector multiplication

Everything else is a **softmax** function

Word2Vec – Objective Function

- How do we optimise the weights?
→ Define a quality function for the current network weights θ
- How well do the weights explain the word occurrences in a corpus?
→ Enter Likelihood!

$$L(\theta) = \prod_{t=1}^T \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta)$$

Diagram illustrating the components of the Word2Vec Likelihood function $L(\theta)$:

- All trainable weights, i.e. E and C from the last slides** (points to θ)
- Product over all words in the corpus** (points to the outer product $\prod_{t=1}^T$)
- Product over the word's context** (points to the inner product $\prod_{\substack{-m \leq j \leq m \\ j \neq 0}}$)
- Probability of the context word given the center word and weights θ** (points to $P(w_{t+j} | w_t; \theta)$)

Word2Vec – Objective Function

$$L(\theta) = \prod_{t=1}^T \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta)$$

- We prefer minimising over maximising
- We also prefer not to multiply probabilities
→ Use the negative log-likelihood!

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^T \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j} | w_t; \theta)$$

Word2Vec – Objective Function

All words in the corpus

All context words of the current center word

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^T \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log \frac{\exp(e_t \cdot c_{t+j}^T)}{\sum_{w \in V} \exp(e_t \cdot c_w^T)}$$

probability of a context word given a center word

→ Optimise this with **Gradient Descent!**

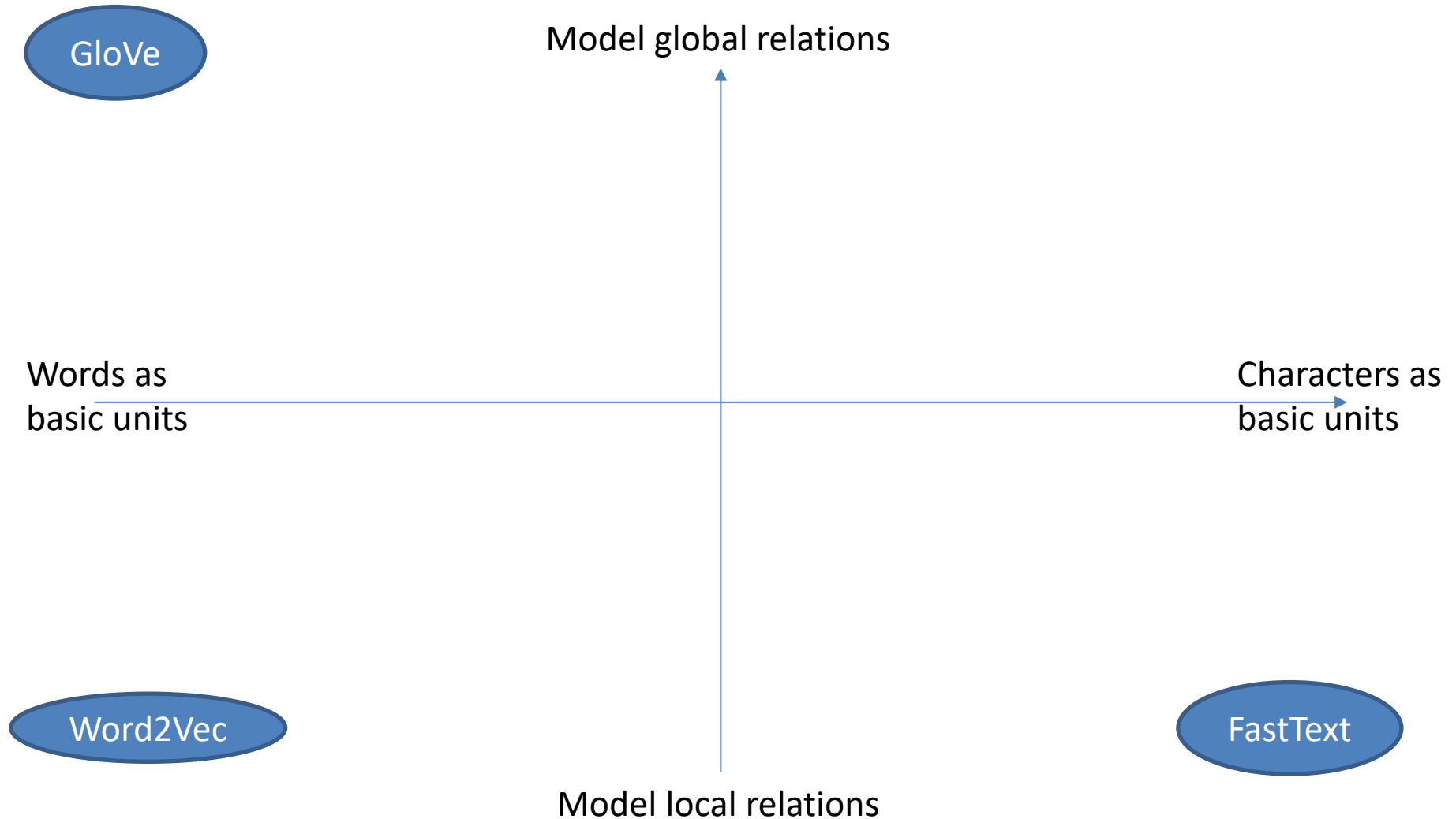
Alternative Word Embedding Models

- Two choices in „model design“ made for Word2Vec:
 - Optimise for local contexts/specific word occurrences
 - Treat words as elementary units
- Not all word embedding models take these choices!
- In the following section:
Two models that take the different road on one of these choices

1.4 Short Introduction to GloVe and FastText

- What is GloVe/FastText?
- What are the differences to Word2Vec?

Word2Vec, GloVe, FastText in the „Design Choice Space“



GloVe – Counting Global Co-Occurrences

- GloVe = **G**lobal **V**ectors
- Another method for creating word embeddings from a corpus
- Calculated over the global word co-occurrence matrix
- Results in embeddings of quality similar to Word2Vec

GloVe – Differences to Word2Vec

- Word2Vec generates embeddings from local neighbourhoods: predict neighbouring words from a target word
- GloVe generates word embeddings w and context embeddings \tilde{w} from **ratios of co-occurrences**

$$w_i^T \tilde{w}_k = \log(X_{ik}) - \log(X_i)$$

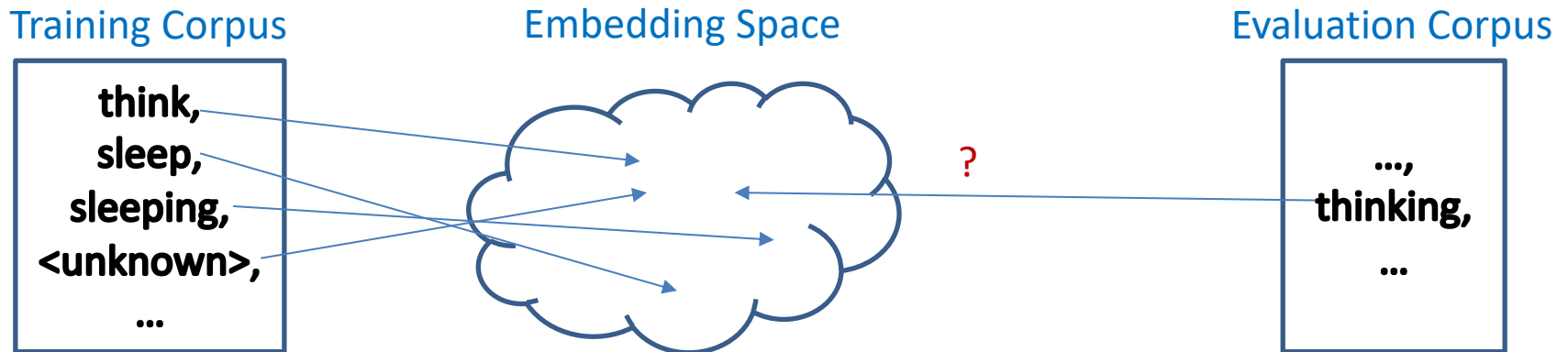
$$X_{ij} \quad \text{“frequency of word } j \text{ occurring in context of } i\text{”}$$

$$X_i = \sum_k X_{ik} \quad \text{“frequency of any word in context of } i\text{”}$$

- Find a set of vectors that represents this ratio well (using gradient descent)!

Dealing with Unknown Words

- Drawback of both Word2Vec and GloVe:
Previously unseen words cannot be embedded!
- Example for unknown words:



→ No information about „thinking“!

Dealing with unknown words – FastText

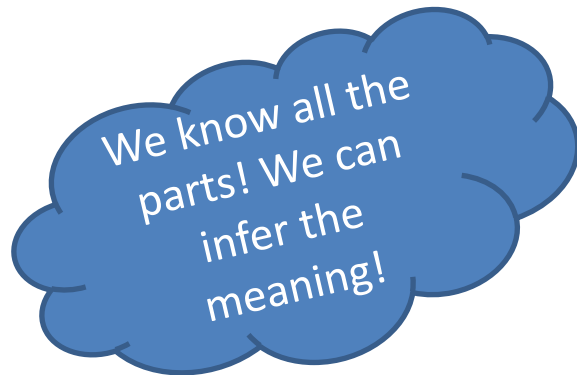
- A library for word embeddings taking into account sub-word information
- Modifies Word2Vec:
 - Word2Vec/skip-gram predicts context words given a **center word**
 - FastText predicts context words given **center n-grams**
- Intuition:
Unknown words can be modelled by their n-grams!

Dealing with unknown words – FastText

Training Corpus

think = <th + thi + hin + ink + nk>
sleep = <sl + sle + lee + eep + ep>
sleeping = <sl + ... + ing + ng>

Embedding Space



...,
thinking = <th + thi + hin + ink + ... + ing + ng>,
...

Evaluation Corpus

Optimisation Goal

- Represent each word as a sum of character n-gram vectors z :

$$G_{\text{"thinking"}} = \{\langle \text{th}, \text{thi}, \dots, \text{ing}, \text{ng} \rangle, \dots\}, \quad e_w = \sum_{g \in G_w} z_g$$

All character n-grams of length 3 to 6

- Tune the embedding vectors so that

$$P(w_{t+j} | w_t) = \frac{\exp(\sum_{g \in G_t} z_g \cdot c_{t+j}^T)}{\sum_{w \in V} \exp(\sum_{g \in G_t} z_g \cdot c_t^T)}$$

Comparing Word2Vec, GloVe and FastText

- Word2Vec:

$$P(w_{t+j}|w_t) = \frac{\exp(e_t \cdot c_{t+j}^T)}{\sum_{w \in V} \exp(e_t \cdot c_w^T)}$$

- FastText:

$$P(w_{t+j}|w_t) = \frac{\exp(\sum_{g \in G_t} z_g \cdot c_{t+j}^T)}{\sum_{w \in V} \exp(\sum_{g \in G_t} z_g \cdot c_w^T)}$$

- GloVe:

$$w_i^T \widetilde{w}_k = \log(X_{ik}) - \log(X_i)$$

1.5 Pre-Processing

- Tokenization
- Lower Casing
- Stemming
- Lemmatization

Text Pre-Processing

- Up to now we have based our embeddings on single words
 - We will refer to these minimal inputs to our models as *tokens*
- Creating those tokens looks trivial at first glance, e.g.:

“Sentences are fun, said the student after he’d
worked out his sentence.”

Text Pre-Processing

- We could just separate the sentence at all the spaces

[Sentences] [are] [fun,] [said] [the] [student]
[after] [he'd] [worked] [out] [his] [sentence.]

This looks problematic

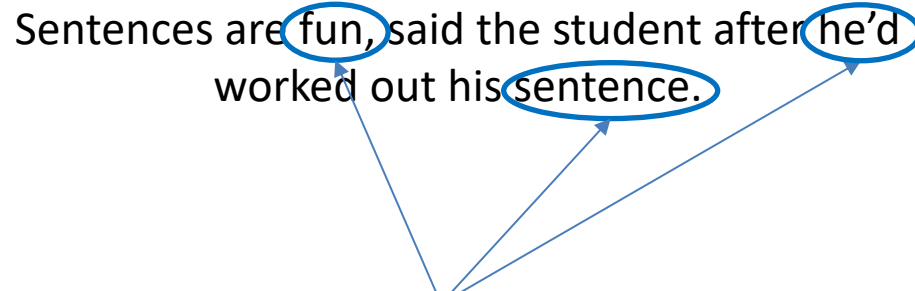
```
sentence.split(" ")
```

- On closer examination, several problems appear to exist
 1. Punctuation: *fun,* vs. *fun*
 2. Cases: *Sentence* vs. *sentence*
 3. Form: *work* vs. *worked*
- The number of embeddings needed increases dramatically

Text Pre-Processing – Tokenization

1. Punctuation
2. Cases
3. Form

Sentences are fun, said the student after he'd
worked out his sentence.

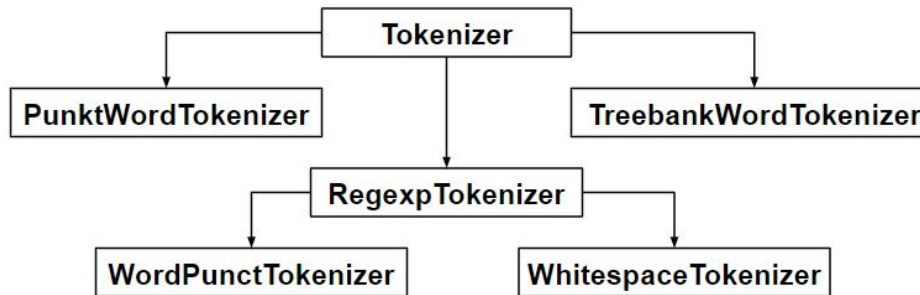


- Considering only spaces for tokenization is not enough
 - Interpunction
 - Abbreviations
 - Contractions
 - ...

Text Pre-Processing – Tokenization

- There are several more sophisticated methods

1. Punctuation
2. Cases
3. Form



- ... like using regular expressions or lexical databases
- Modern libraries will do the work for you

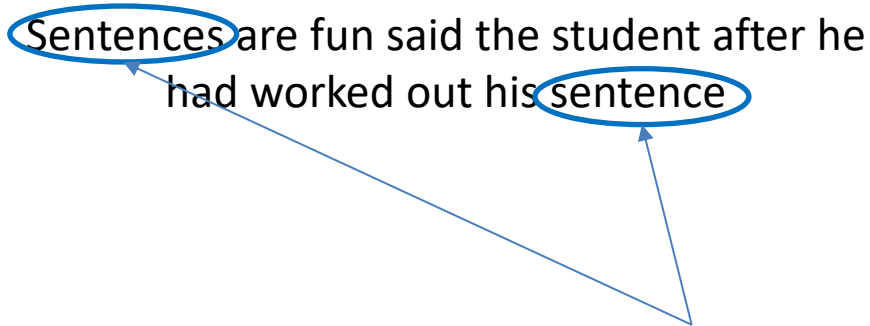
spaCy



Text Pre-Processing – Lower Casing

1. Punctuation ✓
2. Cases
3. Form

Sentences are fun said the student after he
had worked out his sentence

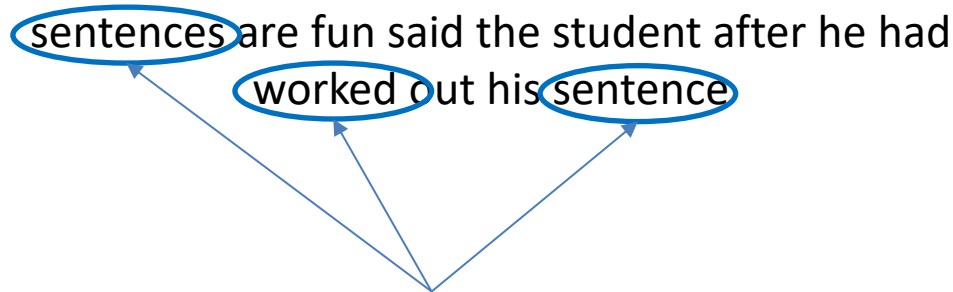


- Do we need different embeddings for upper and lower case?
→ Let's just lower-case everything

```
sentence.lower()
```

Text Pre-Processing – Stemming

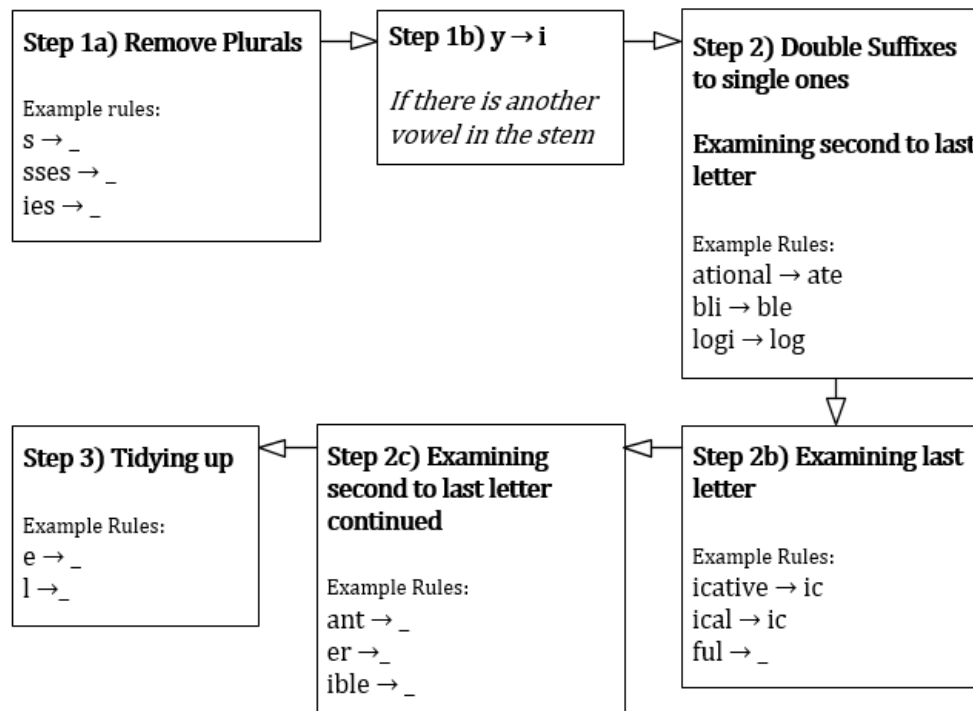
1. Punctuation ✓
2. Cases ✓
3. Form



- The many different word forms still inflate our embeddings
 - Number (Singular / Plural)
 - Case (“Kasus”)
 - Declination
 - Conjugation
 - ...

Text Pre-Processing – Stemming

- The reduction of a word to its stem is called *Stemming*
- Usually rule based, Porter Stemmer best known
 - Fairly efficient and easy to implement
 - But: Artificial word stems possible (as long as it is consistent)



Text Pre-Processing – Lemmatization

- Lemmatization also aims to reduce inflectional forms
- “[...] the canonical form, dictionary form, or citation form”¹
 - Resulting in real existing words
 - More sophisticated, but also more complex
- Usually solved with big databases, e.g. WordNet

Stemming

sentence^{ces} → sentence^c
had → had

```
from nltk.stem import
PorterStemmer
ps = PorterStemmer()
for word in sentence:
    print(ps.stem(word))
```

Lemmatization

sentences → sentence
had → have

```
from nltk.stem import
WordNetLemmatizer
lm = WordNetLemmatizer()
for word in sentence:
    print(lm.lemmatize(word))
```

¹ [https://en.wikipedia.org/wiki/Lemma_\(morphology\)](https://en.wikipedia.org/wiki/Lemma_(morphology))

Text Pre-Processing – The Result

- Stemmed Version

- | | |
|----|---------------|
| 1. | Punctuation ✓ |
| 2. | Cases ✓ |
| 3. | Form ✓ |

sentenc are fun said the student after he had
work out his sentenc

- Lemmatized Version

sentences are fun say the student after he
have work out his sentence

Depending on the method



Text Pre-Processing – Outlook

- Modern systems tend to approach NLP end-to-end, i.e. not using any kind of pre-processing
- Tokenization still remains relevant
 - Word Piece Tokenization, as used in BERT
 - Byte-level Byte-Pair-Encoding, as used in GPT-2

[Sen] [##ten] [##ces] [are] [fun] [,] [said] [the] [student] [after] [he] [']
[d] [worked] [out] [his] [sentence] [.]

[Sent] [ences] [Ġare] [Ġfun] [,] [Ġsaid] [Ġthe] [Ġstudent] [Ġafter] [Ġhe]
[ĠâĠ] [Ġ,] [d] [Ġworked] [Ġout] [Ġhis] [Ġsentence] [.]

*We'll revisit
this later*



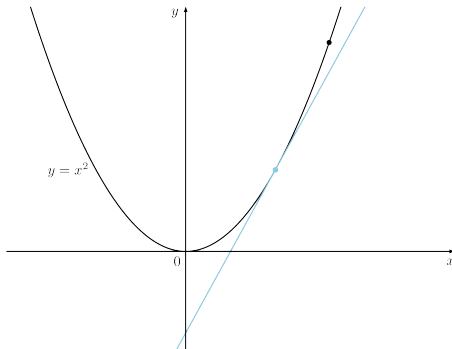
**HUGGING
FACE**

Next Lecture: Gradient Descent

Gradient Descent

Finding the gradient

Backpropagation



„Descending“ along
the gradient

Different optimisation algorithms

