

Machine Learning 1 – Fundamentals

Natural Language Processing and Sequence Models

Prof. Dr. J. M. Zöllner, M.Sc. Marcus Fechner, M.Sc. Nikolai Polley



Outline

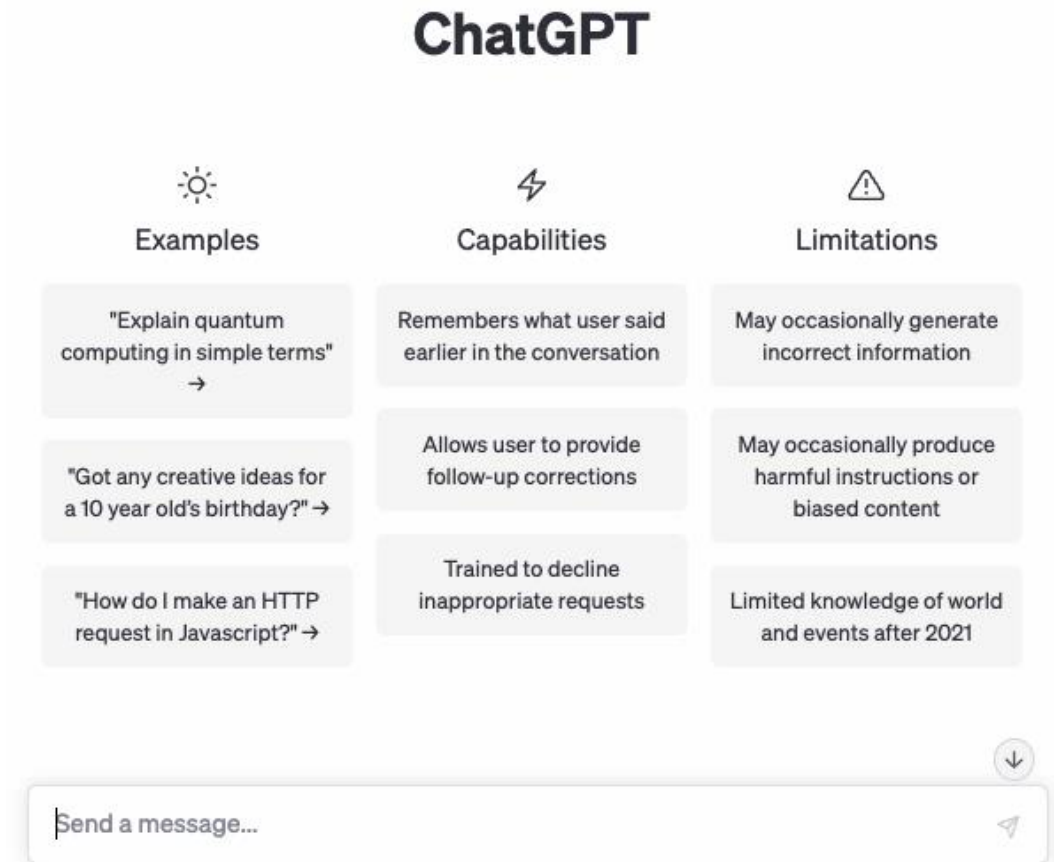
- Introduction
- Text Normalization (Preprocessing)
- Text Representation
- Traditional Language Models
- Neural Language Models
- Decoding
- Literature

What is Natural Language Processing?

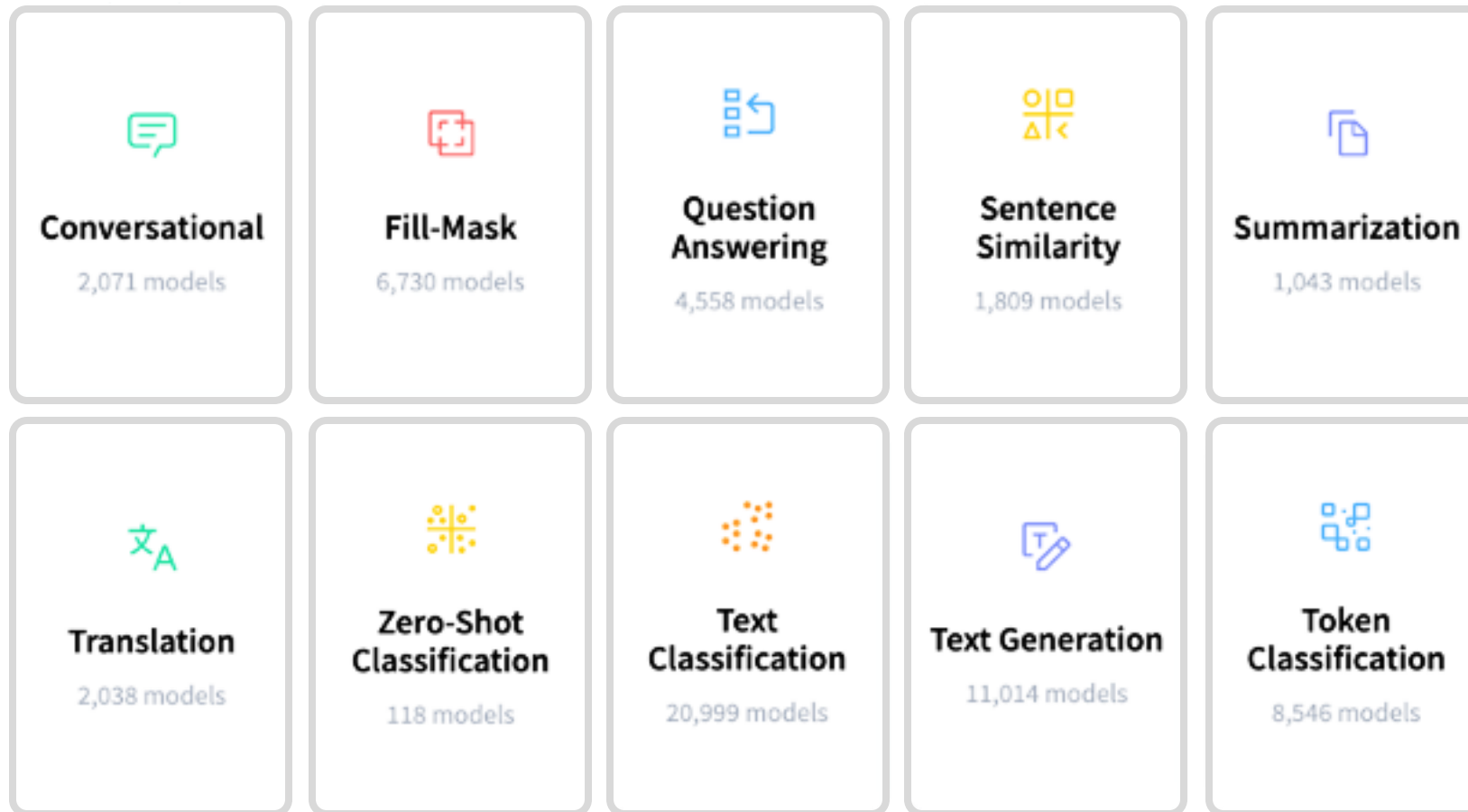
- NLP is a field of study that focuses on the **interaction between computers and humans in natural language**
- **Goal:** Develop algorithms and models that enable computers to understand, interpret and generate natural language in a way that is meaningful and useful
- **Challenges:**
 - **Ambiguity:** Language is often ambiguous, and understanding meaning requires context.
 - **Polysemy:** A word or phrase can have different meanings depending on the context.
 - **Syntax and grammar:** Parsing and understanding sentence structure.
 - **Cultural differences:** Language variations across regions and cultures.
 - ...

Text Understanding – Large Language Models

- **Question:** How can we train capable general purpose chatbots?
- **ChatGPT / Bing-Chat / Bard....:**
 - Chatbots with a wide variety of use cases
 - Machine learning with neural network architectures

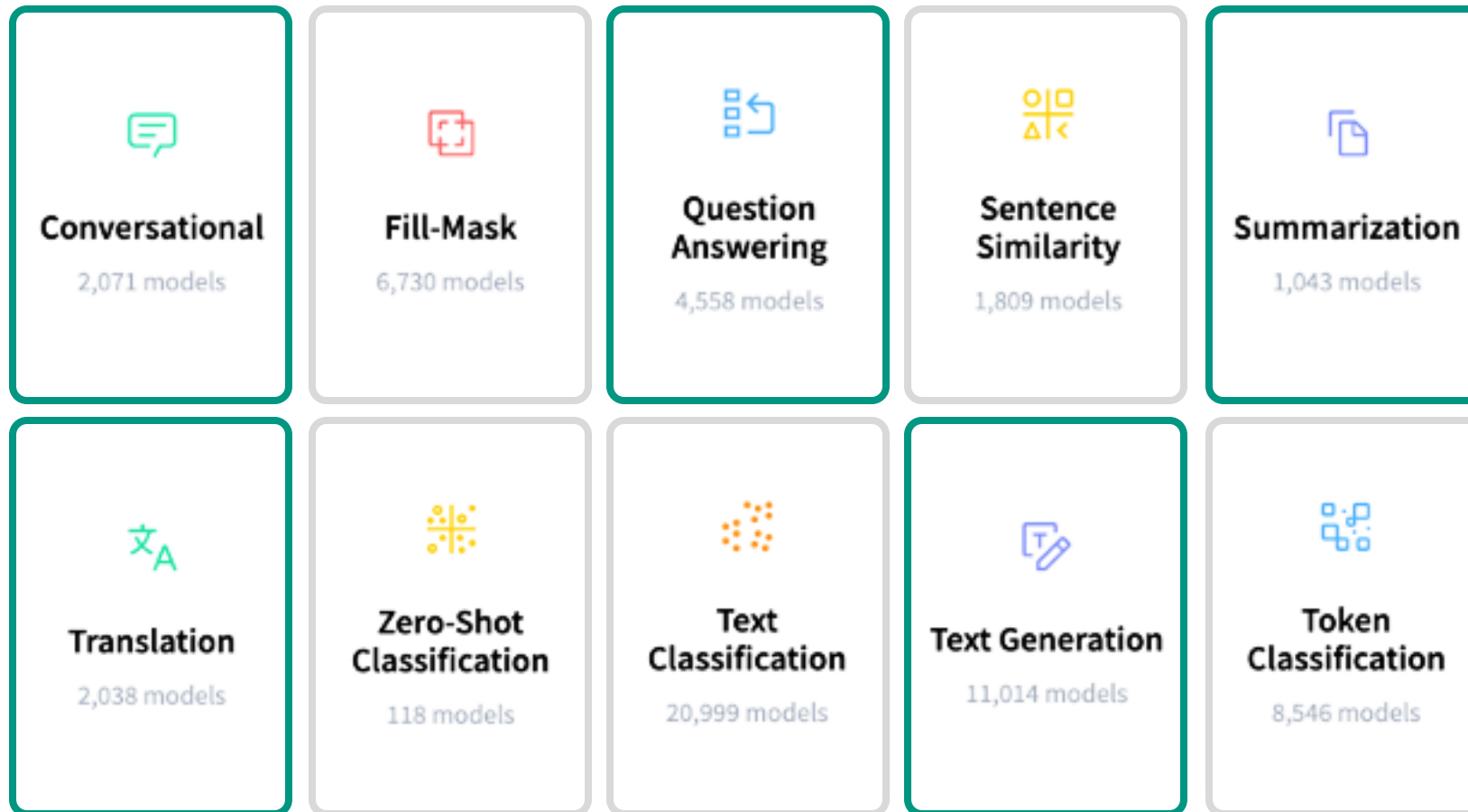


Tasks and Applications



[[Hugging Face NLP Tasks](#)]

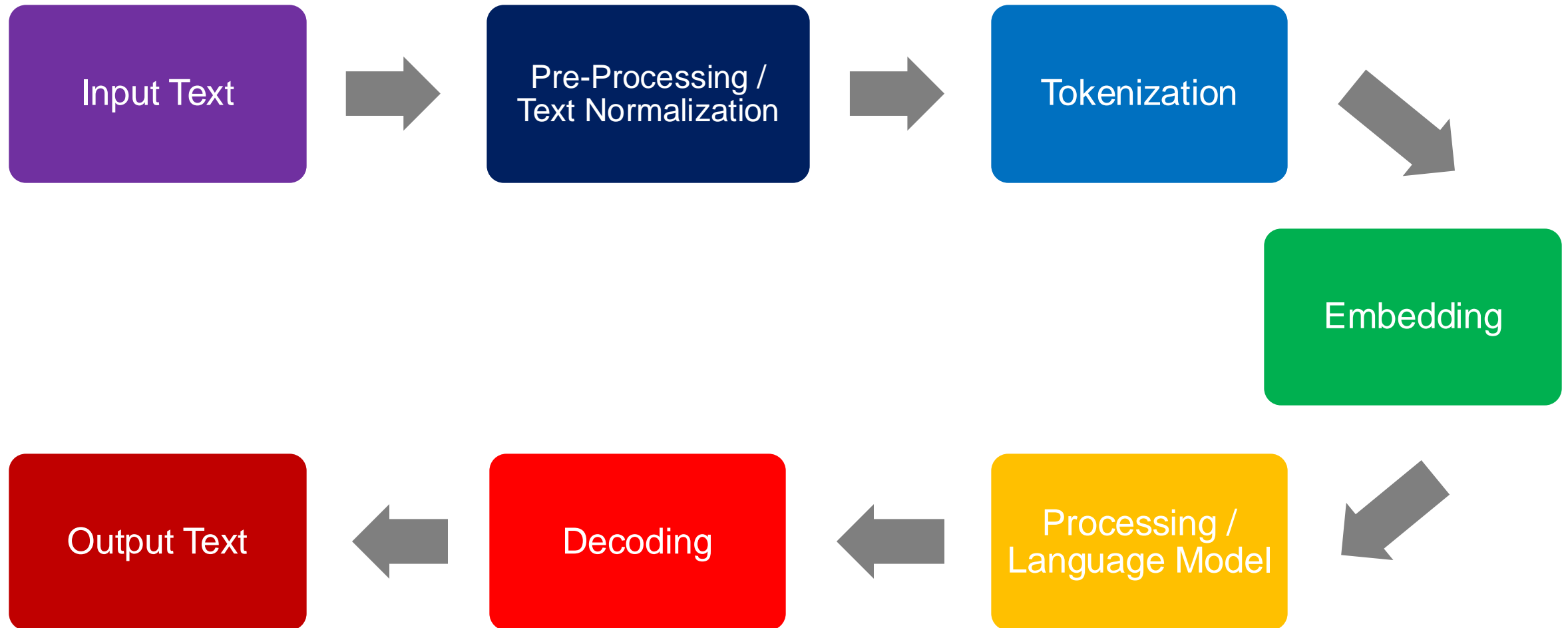
Tasks and Applications



Focus of this lecture

[Hugging Face NLP Tasks]

NLP Pipeline (Generative Models)



Outline

- Introduction
- Text Normalization (Preprocessing)
- Text Representation
- Traditional Language Models
- Neural Language Models
- Decoding
- Literature

Text Normalization

- Text normalization is the process of converting text into a convenient, standardized form that can be easily processed and understood by machines.
- **Goals**
 - Standardization
 - Consistency
 - Simplification
 - Compression

Text Normalization Methods

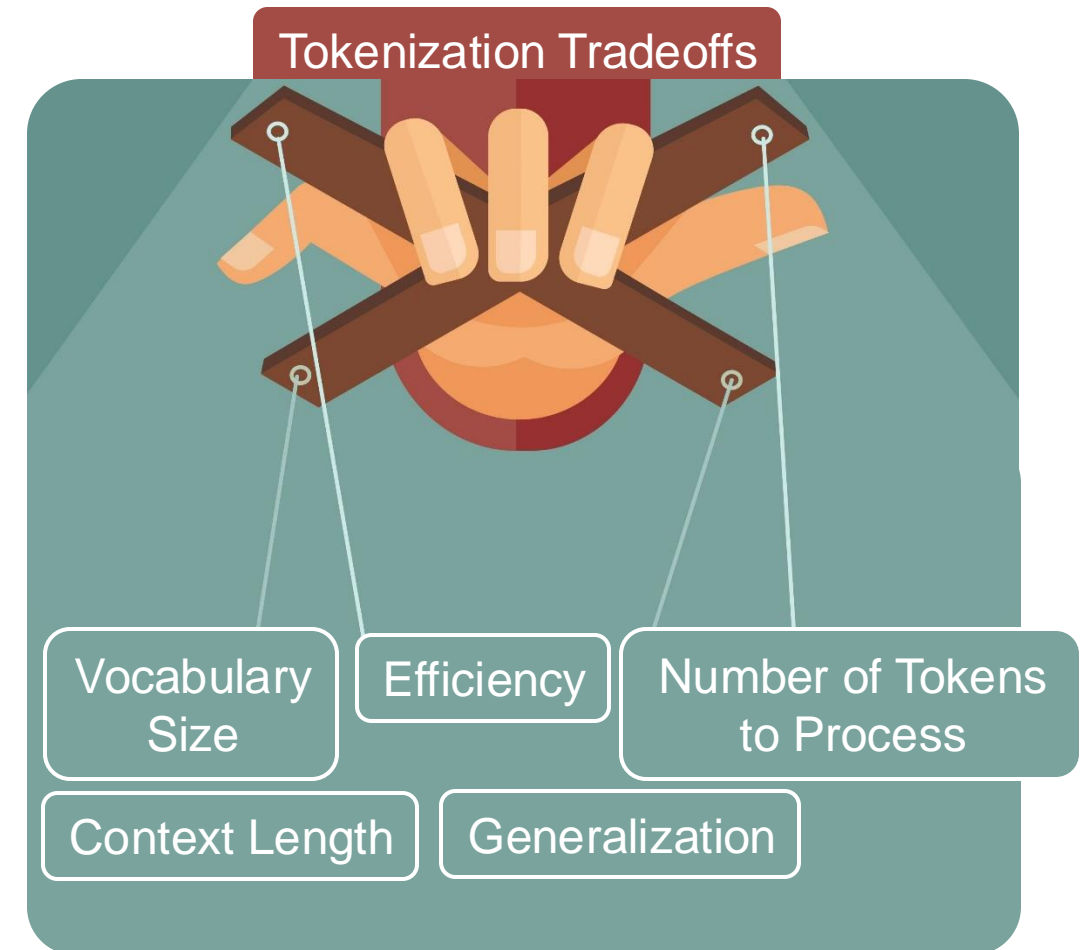
- **Lowercasing:** Converting all text to lowercase to reduce variations due to capitalization.
 - „*I Am a TeAchEr*“ → „*i am a teacher*“
- **Stemming:** Strip affixes/suffixes from the word. (simple form of lemmatization)
 - “books” → “book”, “looked” → “look”, “denied” → “deni”, “flies”, → “fli”
- **Lemmatization:** Converting words to their lemma, their shared root (dictionary headword form).
 - “am”, “are”, “is” → “be”, “car”, “cars”, “car’s”, “cars” → “car”
- **Removing stop words:** Eliminate words that do not carry much meaning, such as „the“, „a“, „and“, etc.
 - „*The quick brown fox jumps over the lazy dog.*“ → „*quick brown fox jumps lazy dog.*“

Tokenization

- Tokenization **breaks down a continuous stream of text into smaller units**, such as characters, words, phrases, or sentences, **called tokens**.
- Analogous to a segmentation task
- **Goal**: Divide the text into **meaningful units** that capture the **fundamental semantic** and **syntactic elements** of the language.
- Vocabulary $V = \{w_1, \dots, w_{|V|}\}$: Set of unique tokens in the training corpus
- Language models process text on token level
- Example
 - Sentence: *“The students are closely following the lecture”*
 - Token output: [*“The”, “students”, “are”, “closely”, “following”, “the”, “lecture”*]
 - Token-ID output: [3244, 2255, 3245, 1234, 7753, 3244, 4446]

Tokenization Methods

- Sentence tokenization
- Word tokenization
 - Space-based tokenization
- Subword tokenization
 - Byte-Pair Encoding tokenization
 - Unigram tokenization
 - WordPiece tokenization
- Character tokenization



Whitespace Tokenization

- Splits the text based on whitespace characters
 - Spaces, tabs, newlines
- Example
 - Input: *“The students are closely following the lecture”*
 - Token output: [*“The”, “students”, “are”, “closely”, “following”, “the”, “lecture”*]
- **Problem:** Out-of-vocabulary words – Words that are not part of the vocabulary generated during training.
- Basis for further processing and analysis

Byte-Pair Encoding Tokenization

- **Idea:** Instead of using rule-based tokenization, use the data to tell us how to tokenize
- **Subwords:** Tokens that can be smaller than a word. Can be arbitrary substring or meaning-bearing units (e.g. morphemes, e.g. –bar, -los)
- **Advantage:** Can deal with unknown words
 - Train corpus: *low*, *new*, *newer*
 - Test: *lower*
- **Components**
 - **Token learner:** Training corpus \rightarrow Vocabulary $V = \{w_1, \dots, w_{|V|}\}$ (Set of tokens)
 - **Token segmenter:** Test sentence \rightarrow Sequence of tokens

Byte-Pair Encoding Token Learner

- Create initial vocabulary V as a set of all individual characters in the corpus
- Repeat k times
 - Choose the two symbols that are most frequently adjacent in the training corpus (say “A”, “B”)
 - Add a new merged symbol “AB” to the vocabulary
 - Replace every adjacent “A” “B” in the corpus with the new symbol “AB”
- **Result:** Original set of characters + k new symbols
- **Remark:** BPE is run inside words (not merging across word boundaries), therefore end-of-word symbol “_” needed

Byte Pair Encoding Data Compression Example

aaabdaaabc

aaabdaaabc Replace Z = aa

ZabdZabc Replace Y = ab

ZYdZYac Replace X = ZY

XdXac Final compressed string

Byte pair	Replacement
X	ZY
ab	Y
aa	Z

Byte-Pair Encoding Token Learner

Byte-Pair Encoding Token Learner Algorithm

```
def byte_pair_encoding(corpus: str, number_of_merges: int) -> set:
    vocabulary = all_unique_characters(corpus) # initial set of tokens is characters
    for _ in range(number_of_merges): # merge tokens number_of_merges times
        t_l, t_r = most_frequent_pair_of_adjacent_tokens(corpus)
        t_new = t_l + t_r # make new token by concatenating
        vocabulary = vocabulary + t_new # update the vocabulary
        # Replace each occurrence of t_l, t_r in corpus with t_new and update the corpus
        corpus = replace_occurences(t_l, t_r, t_new)
    return vocabulary
```


Byte-Pair Encoding Token Learner Example

■ Extremely challenging corpus:

*“low low low low low lowest lowest newer newer
newer newer newer newer wider wider wider new new”*

Step 0: Initialization

Corpus

```

5 l o w _
2 l o w e s t _
6 n e w e r _
3 w i d e r _
2 n e w _
  
```

Vocabulary

```

_, d, e, i, l,
n, o, r, s, t, w
  
```

Step 1: Merge ,e' ,r' → ,er'

Corpus

```

5 l o w _
2 l o w e s t _
6 n e w e r _
3 w i d e r _
2 n e w _
  
```

Vocabulary

```

_, d, e, i, l,
n, o, r, s, t,
w, er
  
```

Byte-Pair Encoding Token Learner

Step 2: Merge ,er' → ,er_'

Corpus

5 l o w _
2 l o w e s t _
6 n e w e r _
3 w i d e r _
2 n e w _

Vocabulary

_, d, e, i, l,
n, o, r, s, t,
w, er, er_

Step 3: Merge ,n' ,e' → ,ne'

Corpus

5 l o w _
2 l o w e s t _
6 n e w e r _
3 w i d e r _
2 n e w _

Vocabulary

_, d, e, i, l,
n, o, r, s, t,
w, er, er_, ne

Step	Merge	Vocabulary
4.	(ne, w)	_, ... , ne, new
5.	(l, o)	_, ... , ne, new, lo
6.	(lo, w)	_, ... , ne, new, lo, low
7.	(new, er_)	_, ... , ne, new, lo, low, newer_
8.	(low, _)	_, ... , ne, new, lo, low, newer_, low_

Byte-Pair Encoding Token Segmenter

- Split up each word into characters
- **Encoding:** Apply each learned merch from the training data
 - Greedily
 - In the order we learned them

- **Example:** „l o w e r _“

1. l o w e r _
2. l o w e r _
3. l o w e r _
4. l o w e r _

Token Ids: 17 13

- **Vocabulary**

_, d, e, i, l, n, o, r, s, t,
w, er, er_, ne, new, lo, low,
newer_, low_

- **Token Ids**

1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17,
18, 19

Byte-Pair Encoding Token Properties

- Usually include frequent words and subwords
 - Often morphemes like –est or –er
- Words that appear often in the corpus will be compressed into a single token
- Rule of thumb
 - **Frequent words:** Word → Token
 - **Rare words:** Word → Multiple tokens

GPT-3 Codex

English sentences most often contain one token per word.
Deutsche Sätze normalerweise mehr.
🚀 同学们好
width -->widht

Clear

Show example

Tokens	Characters
45	118

```
[15823, 13439, 749, 1690, 3994, 530, 11241, 583, 1573, 13, 220, 198,
5005, 30433, 311, 11033, 83, 2736, 3487, 263, 732, 786, 502, 11840, 13,
198, 8582, 248, 222, 10263, 238, 234, 27764, 99, 20015, 105, 25001, 121,
198, 10394, 14610, 9214, 4352, 220, 628]
```




TEXT

TOKEN IDS

Outline

- Introduction
- Text Normalization (Preprocessing)
- Text Representation
- Traditional Language Models
- Neural Language Models
- Decoding
- Literature

Text Representation

- **Practical considerations:** Need to convert the input strings to a machine understandable representation
- **Requirement:** Representation should encode 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.
- Most common linguistic way of thinking about meaning
 - signifier (symbol) \Leftrightarrow signified (idea or thing)
 - E.g. tree \Leftrightarrow {, , , ...}

Covered Methods

- Naïve approach
 - One-Hot encoding
- Static embeddings
 - Word2Vec

Local Word Representation - One-Hot Encoding

- Traditional NLP: Treat words as discrete symbols w
 - “hotel”, “conference”, “motel”
- Can be implemented as one-hot vectors over the vocabulary V :
 - “hotel” $w_3 = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
 - “motel” $w_6 = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$
- Vector dimension = number of words w in vocabulary V (e.g. $|V| \approx 50.000+$)

Limitations of Local Word Representations

■ No natural notion of similarity

- E.g. web search for „*Karlsruhe motel*“ and „*Karlsruhe hotel*“ should yield similar results

- But: “hotel” $\mathbf{w}_3 = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$

- “motel” $\mathbf{w}_6 = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$

- These two vectors are orthogonal ($\mathbf{w}_4 \cdot \mathbf{w}_2 = 0$)!

- **Sparse** (most entries are 0s), high-dimensional vectors

- **No generalization** possible

- **Idea:** Learn to encode similarity in the vectors themselves

Representing Words by Their Context

- **Distributional hypothesis:** A word's meaning is given by the words that frequently appear close to it
 - „*You shall know a word by the company it keeps*“ (J. R. Firth 1957: 11)
- **Example:**
 - *“I love to pet my dog”*
 - *“I love to pet my cat”*
- **Context:** Is the set of words that appear nearby (within a fixed-size window) a word w .
- We use the different contexts of w to build up a representation of w

Word Vectors

■ Word vectors:

- Numerical representations of words in a continuous vector space
- Computational models of word meaning
- We will build a dense vector for each word, chosen so that it is **similar to vectors of words that appear in similar contexts**, measuring similarity as the vector dot (scalar) product
- **Idea:** Represent a word as a point in a multidimensional semantic space that is derived from the distribution of word neighbors
- **Note:** Word vectors are also called (word) **embeddings**, because the discrete word is embedded into continuous vector space.

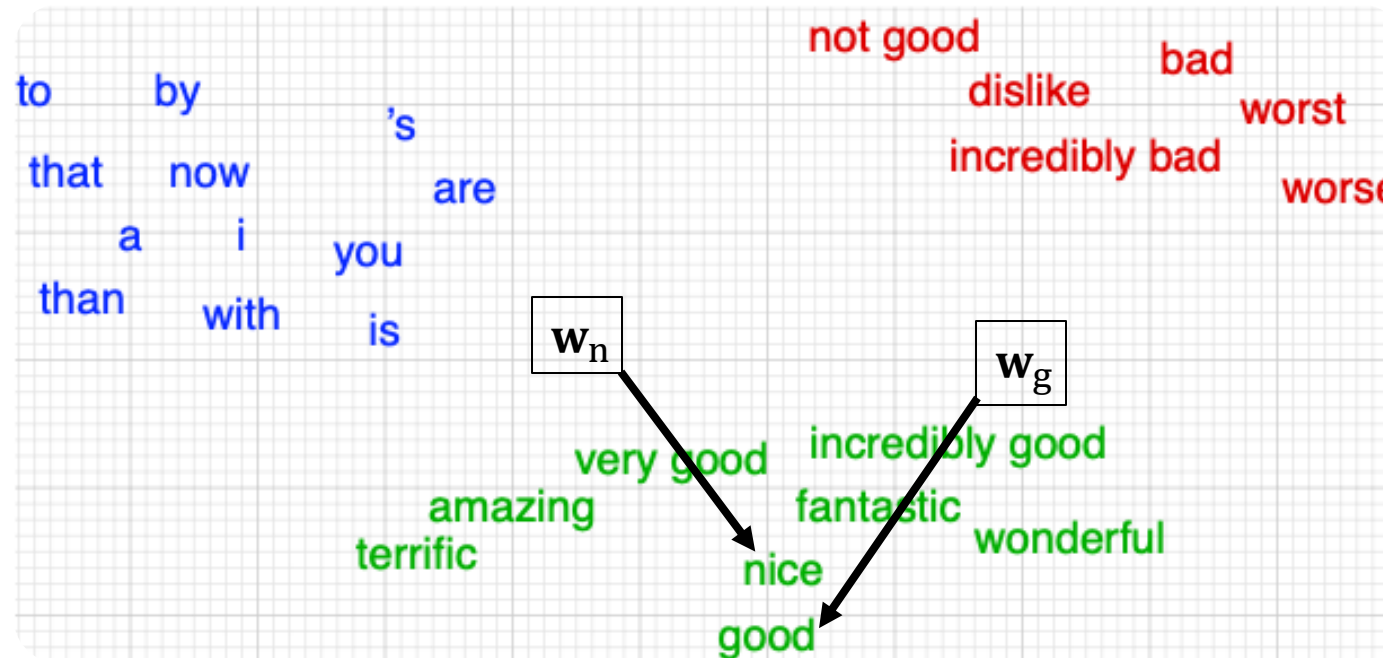
Word Vectors - Visualization

■ “nice”

$$\mathbf{w}_n = [0.286, 0.792, -0.177, -0.107, 0.109, -0.542, 0.349, 0.271, 0.487]$$

■ “good”

$$\mathbf{w}_g = [0.284, 0.790, -0.170, -0.108, 0.108, -0.532, 0.342, 0.201, 0.467]$$

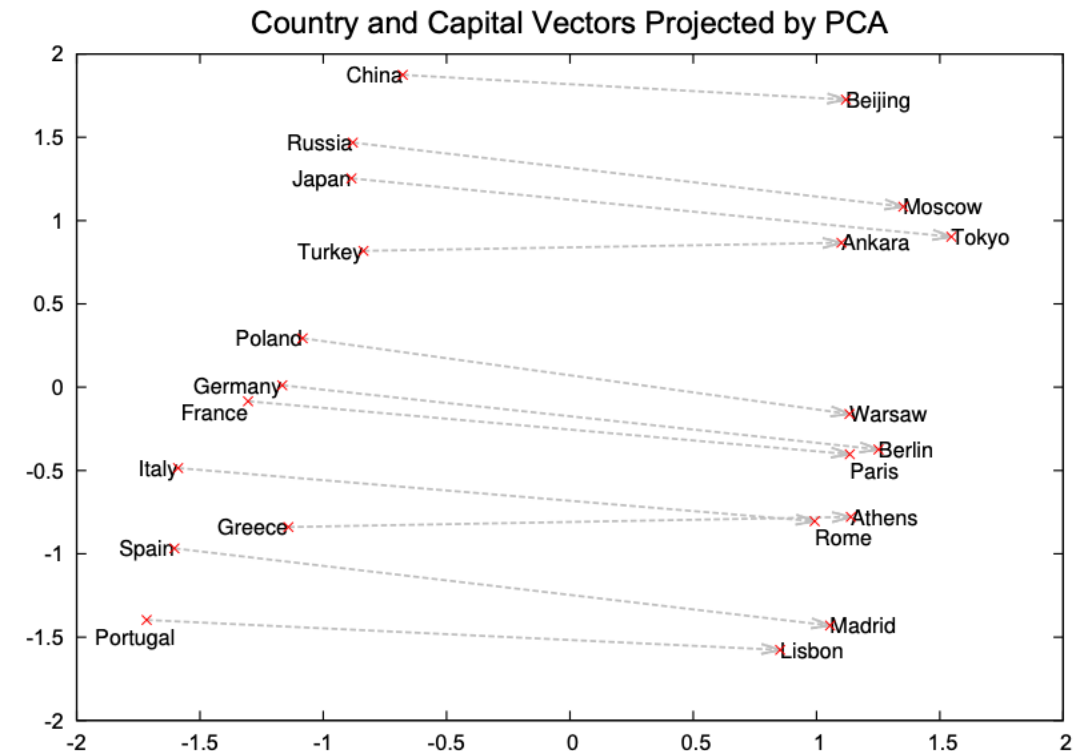


■ **Note:** 2-dimensional (t-SNE) projection of some 60-dimensional embeddings (Li et al. 2015)

Word Vectors – Intuition

$$\mathbf{w}_{\text{king}} - \mathbf{w}_{\text{man}} + \mathbf{w}_{\text{woman}} \approx \mathbf{w}_{\text{queen}}$$

$$\mathbf{w}_{\text{Paris}} - \mathbf{w}_{\text{France}} + \mathbf{w}_{\text{Italy}} \approx \mathbf{w}_{\text{Rome}}$$



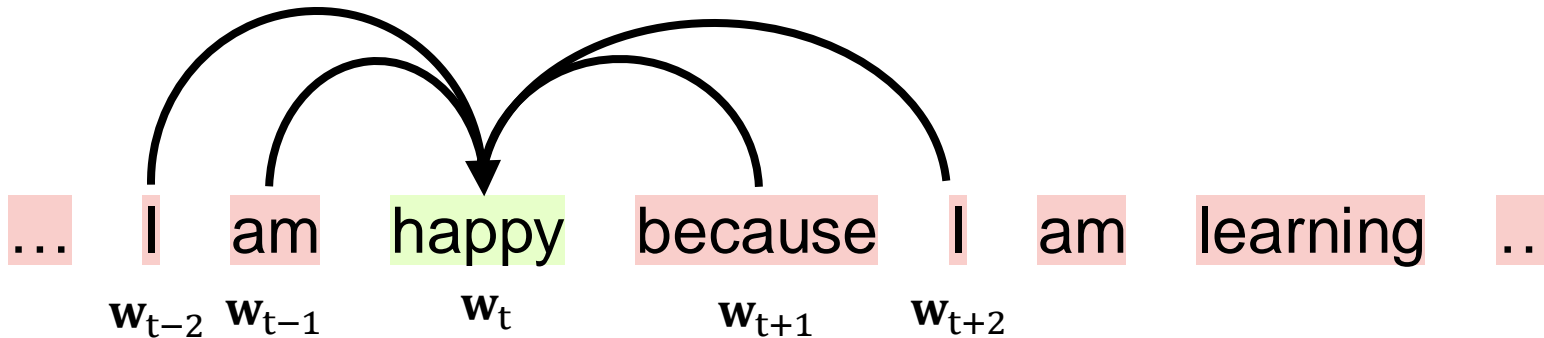
(Mikolov et al. 2013)

Word2Vec (Mikolov et al. 2013)

- Popular method for learning dense word embeddings that captures semantic and syntactic relationships
- Train a **shallow neural network with a single hidden layer** to predict:
 - **Continuous bag-of-words (CBOW)**: Center word given context words
 - **Continuous skip-gram**: Context words given center word
- **By-product of the learning task** are the word embeddings, which are the columns/rows of the learned layer weights W_1/W_2

Word2Vec – CBOW

- CBOW: Learn to predict center word \mathbf{w}_t , ...
 - ... given context words $\mathbf{w}_{t-C}, \dots, \mathbf{w}_{t-1}, \mathbf{w}_{t+1}, \dots, \mathbf{w}_{t+C}$
 - Context half-size C is a hyperparameter
 - Here $C = 2$



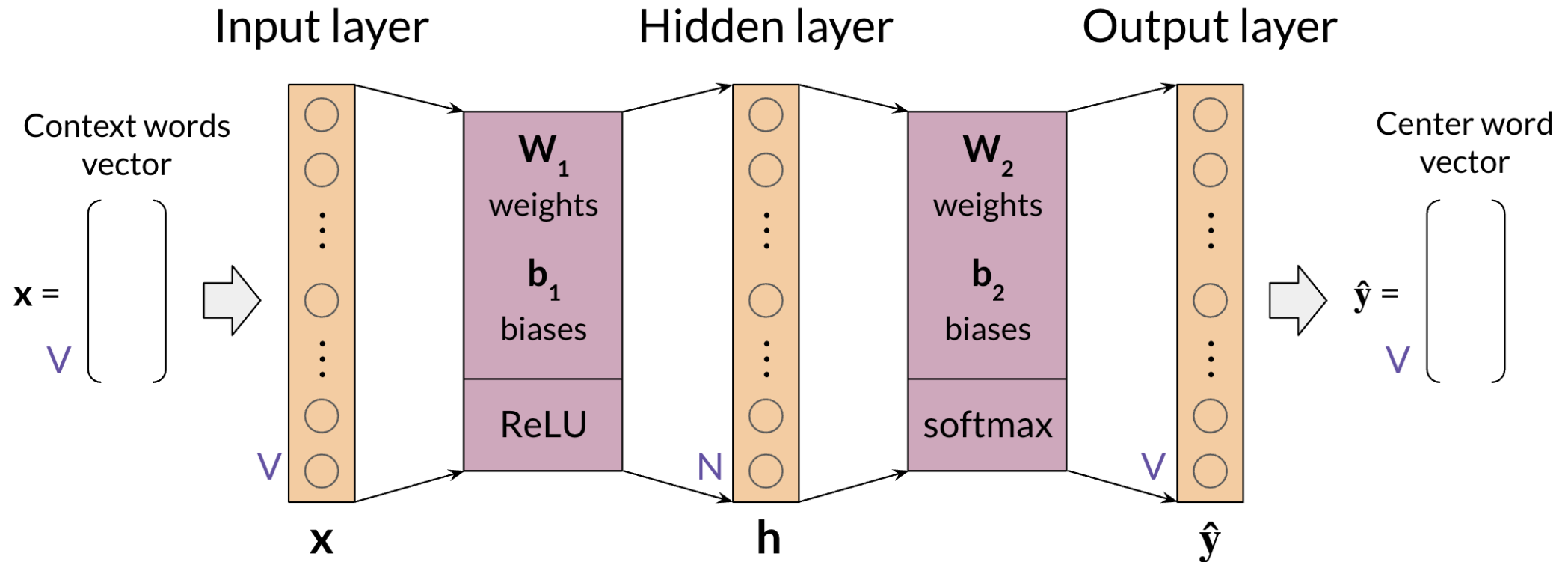
Word2Vec – CBOW

- To transform the **context words vector** into a single vector, we take the average of the one-hot encoded word vectors.
- **Center word vector** w_t is one-hot encoded

$$\begin{array}{c}
 \left(\begin{array}{c} \text{I} \\ \text{am} \\ \text{because} \\ \text{happy} \\ \text{I} \\ \text{learning} \end{array} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{array}{c} \text{am} \\ \text{because} \\ \text{happy} \\ \text{I} \\ \text{learning} \end{array} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{array}{c} \text{because} \\ \text{am} \\ \text{happy} \\ \text{I} \\ \text{learning} \end{array} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{array}{c} \text{I} \\ \text{am} \\ \text{because} \\ \text{happy} \\ \text{I} \\ \text{learning} \end{array} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \right) / 4 = \begin{pmatrix} 0.25 \\ 0.25 \\ 0 \\ 0.5 \\ 0 \end{pmatrix} \\
 \mathbf{w}_{t-2} \quad \mathbf{w}_{t-1} \quad \mathbf{w}_{t+1} \quad \mathbf{w}_{t+2} \quad \bar{\mathbf{w}} \\
 \text{Context words vector}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{happy} \\
 \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \\
 \mathbf{w}_t \\
 \text{Center word vector}
 \end{array}$$

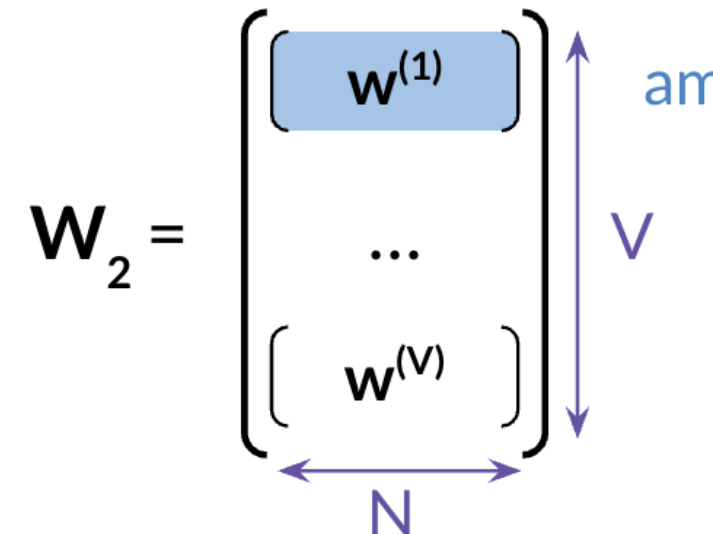
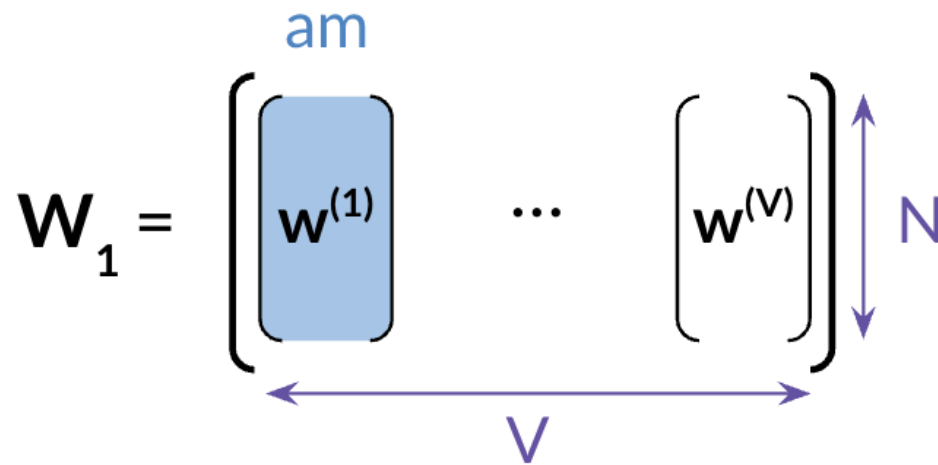
Word2Vec – CBOW Architecture

- **Embedding size** (size of hidden layer) is hyperparameter N
- Trained with cross-entropy loss as classification task



Word2Vec – CBOW Vectors

- After training the model, there are 3 options to obtain the word vectors.
 - Columns of W_1
 - Rows of W_2
 - Average of W_1 and W_2 : $W_3 = 0,5(W_1 + W_2^T)$



Further Embedding Methods

- Naïve approach
 - One-Hot encoding
- Static embeddings
 - Word2Vec
 - GloVe
- Dynamic/Contextual embeddings
 - ELMo
 - BERT
 - ...

“I saw a bat in the cave.”

VS.

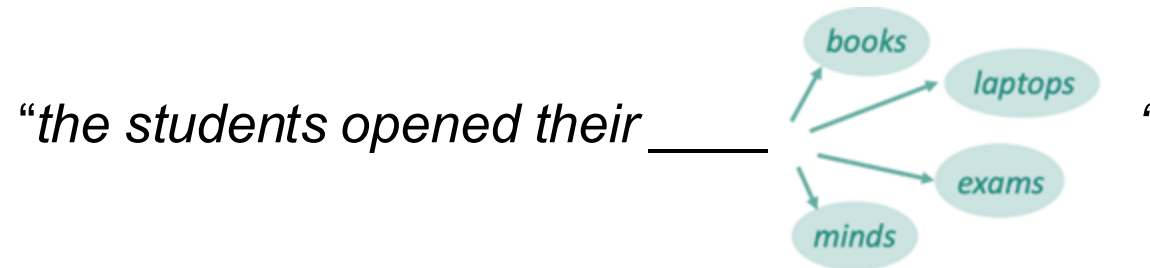
“He hit a home run with his baseball bat.”

Outline

- Introduction
- Text Normalization (Preprocessing)
- Text Representation
- Traditional Language Models
- Neural Language Models
- Decoding
- Literature

Language Modeling

- Language modeling is the task of predicting the next token



- **Formally:** Given a sequence of tokens x_1, x_2, \dots, x_{t-1} , compute the conditional probability distribution of the next token x_t .

$$p(x_t | x_{t-1}, \dots, x_1)$$

where x_t can be any token in the vocabulary $V = \{w_1, \dots, w_{|V|}\}$

- A system that does that is called a **language model**
- The history (conditioning variables/tokens) is called context in NLP

Language Modeling

- You can also think of a language model as a system that assigns a probability to a piece of text
- For example, if we have some text, then the probability of this text (according to the language model) is:

$$p(\text{the, students, opened, their}) = p(\text{the})p(\text{students}|\text{the})p(\text{opened}|\text{students, the}) \dots$$

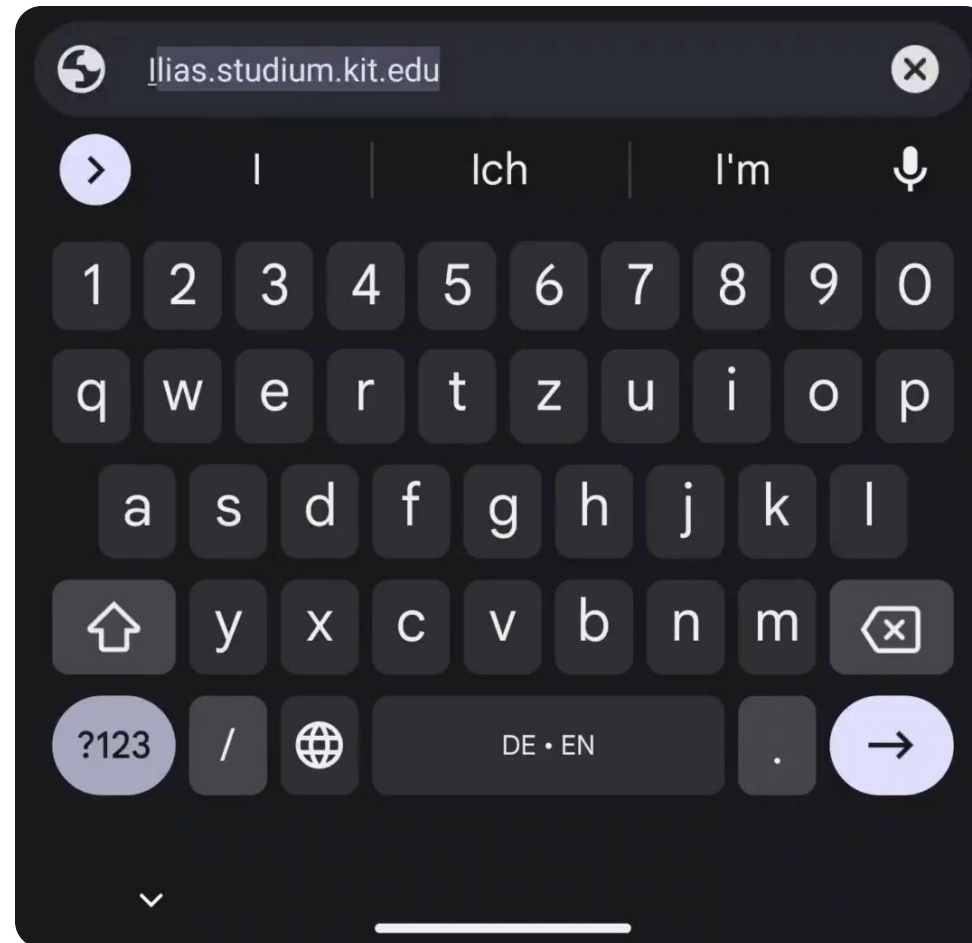
$$p(\mathbf{x}_1, \dots, \mathbf{x}_t) = p(\mathbf{x}_1) \cdot p(\mathbf{x}_2 | \mathbf{x}_1) \cdots p(\mathbf{x}_t | \mathbf{x}_{t-1}, \dots, \mathbf{x}_1)$$

$$= \prod_{t=1}^T \underbrace{p(\mathbf{x}_t | \mathbf{x}_{t-1}, \dots, \mathbf{x}_1)}$$

This is what our LM provides

Note: We use the chain rule of probability to decompose the sequence

You Use Language Models Every Day!



n-gram Language Model

- Before deep learning, n-gram language models were the model of choice!
- **Definition:** An n-gram is a chunk of n consecutive tokens
 - Example: *“the students opened their”*
 - **Unigram**, $n = 1$: *“the”, “students”, “opened”, “their”*
 - **bigram**, $n = 2$: *“the students”, “students opened”, “opened their”*
 - **trigram**, $n = 3$: *“the students opened”, “students opened their”*
 - **four-gram**, $n = 4$: *“the students opened their”*
- **Idea:** Collect statistics about how frequent different n-grams are and use these to predict next word.

n-gram Properties

- n-gram models approximate the history context by the last n-1 tokens
 - They make a Markov assumption (the model is memoryless)

Markov assumption

Preceding $n - 1$ words

$$p(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \dots, \mathbf{x}_1) = p(\mathbf{x}_t \mid \overbrace{\mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-n+1}}^{\text{Preceding } n-1 \text{ words}})$$

$$p(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-n+1}) = \frac{p(\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-n+1})}{p(\mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-n+1})} \quad \begin{array}{l} \text{Prob. of n-gram} \\ \text{Prob. of (n-1)-gram} \end{array}$$

- Each conditional probability $p(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-n+1})$ is represented by an entry in a probability table

n-gram Training

- **Question:** How do we get those n-gram and (n-1)-gram probabilities?
- **Answer:** By counting them in some large corpus of text

$$p(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-n+1}) = \frac{p(\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-n+1})}{p(\mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-n+1})} \approx \frac{\text{count}(\mathbf{x}_t, \mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-n+1})}{\text{count}(\mathbf{x}_{t-1}, \dots, \mathbf{x}_{t-n+1})}$$

- This is a statistical approximation!

n-gram Training

■ Example: Learning a 4-gram language model

“~~as the proctor started the clock, the~~ students opened their ”

└────────────────────────────────┘
└────────────────────────────────┘
└──┘

discard
condition / context
 w

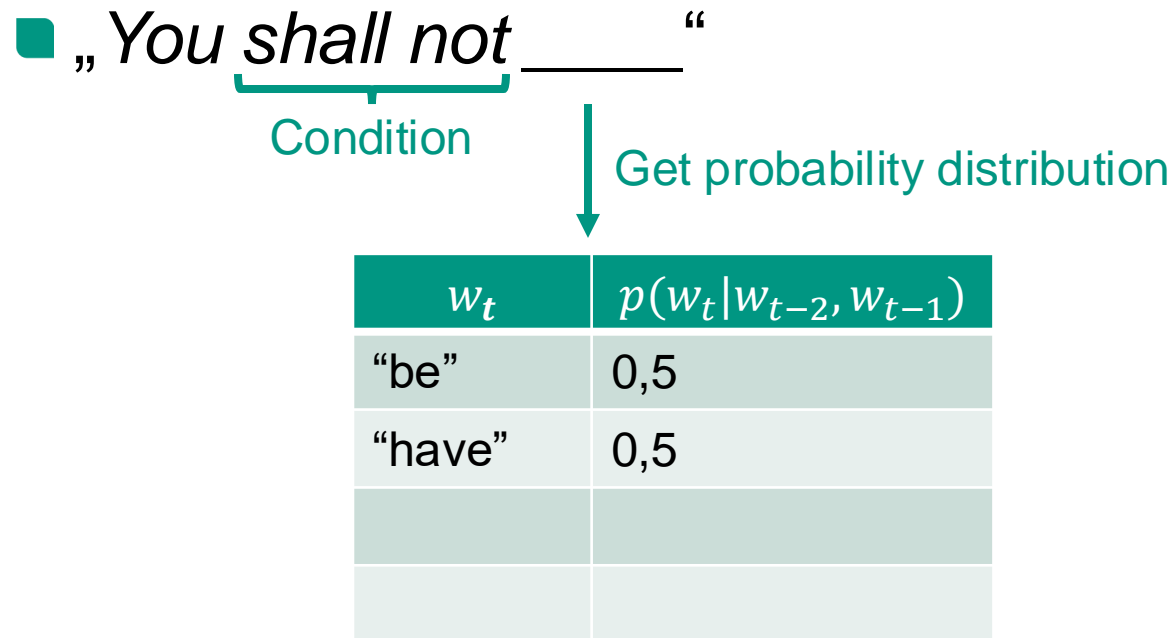
$$p(w \mid \text{students opened their}) = \frac{\text{count}(\text{students opened their } w)}{\text{count}(\text{students opened their})}$$

■ Suppose that for a given corpus:

- “students opened their” occurred 1000 times
- “students opened their books” occurred 400 times
 - $p(\text{books} \mid \text{students opened their}) = 0,4$
- “students opened their exams” occurred 100 times
 - $p(\text{exams} \mid \text{students opened their}) = 0,1$

Should we have discarded the “proctor” context?

Generating Text with Trigram Models



Note: Trained on Reuters dataset.
Business and financial news corpus.

Generating Text with Trigram Models

■ „You shall not be _____“
Condition ↓ Get probability distribution

w_t	$p(w_t w_{t-2}, w_{t-1})$
“able”	0.055
“reached”	0.044
“allowed”	0.037
“a”	0.037

Note: Trained on Reuters dataset.
Business and financial news corpus.

Generating Text with Trigram Models

■ „*You shall not be able* _____“

Condition Get probability distribution

w_t	$p(w_t w_{t-2}, w_{t-1})$
“to”	0.99
“plant”	0.01

Note: Trained on Reuters dataset.
Business and financial news corpus.

Generating Text with Trigram Models

■ Sample text:

„today the price of gold per ton, while production of shoe lasts and shoe industry, the bank intervened just after it considered and rejected an imf demand to rebuild depleted european stocks, sept 30 end primary 76 cts a share.“

- Surprisingly grammatical, but incoherent!
- **We need to consider more than three words at a time** if we want to model language well!

n-gram Problems and Limitations

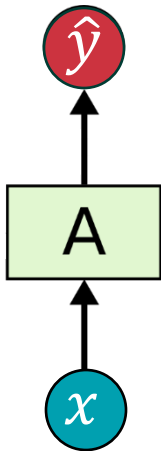
- Text is **incoherent** for small n
- Increasing n worsens sparsity problem and increases model size
- The **probability table comprises $|V|^n$ entries**, where $|V| \approx 30.000$
- **Huge memory and corpora** (training sets) **are needed** (many long sentences are very rare)
- **Limited history context** and **parameters grow exponentially**

Outline

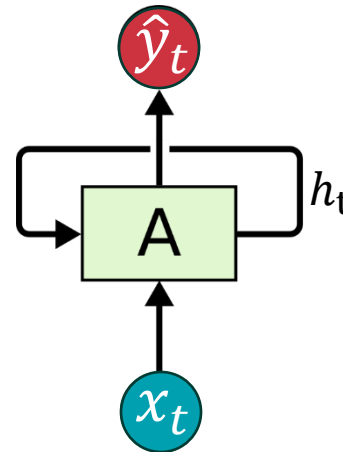
- Introduction
- Text Normalization (Preprocessing)
- Text Representation
- Traditional Language Models
- Neural Language Models
- Decoding
- Literature

Feedforward vs. Recurrent Neural Networks (RNNs)

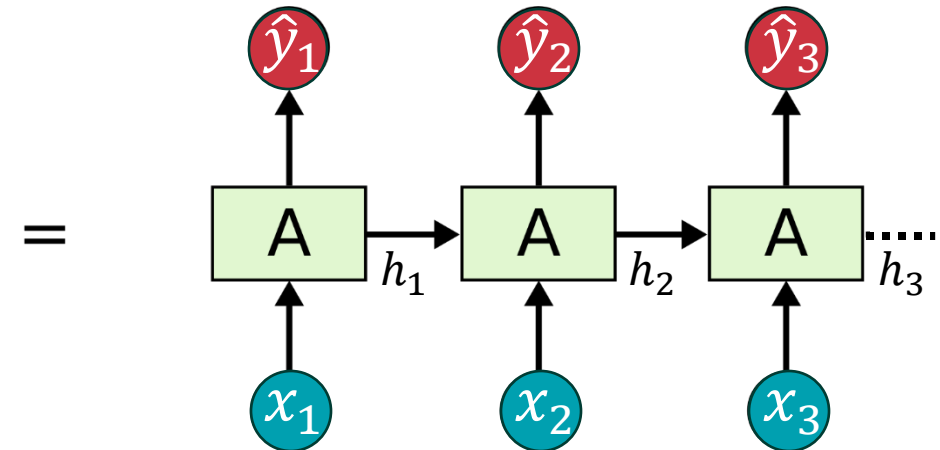
Feedforward
Neural Network



Recurrent
Neural Network



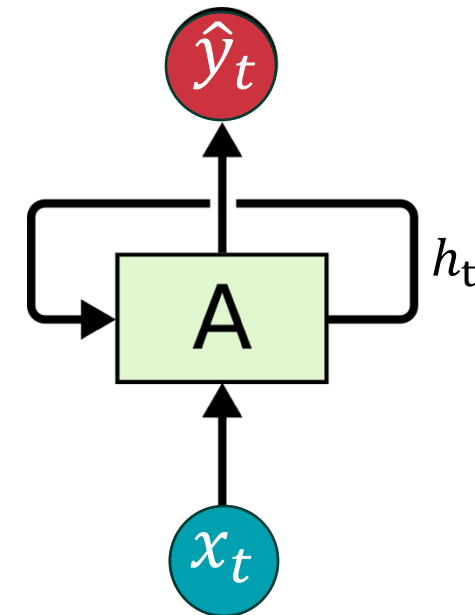
RNN unrolled
over time



- **Feedforward neural networks: Stateless**, only the current input x_t is considered for the output \hat{y}_t
- **Recurrent neural networks: Stateful**, recurrent connection (loop) allows to consider not only the current input x_t , but also a hidden state h_t , that summarizes past inputs x_t , for the output \hat{y}_t

Recurrent Neural Networks (RNNs)

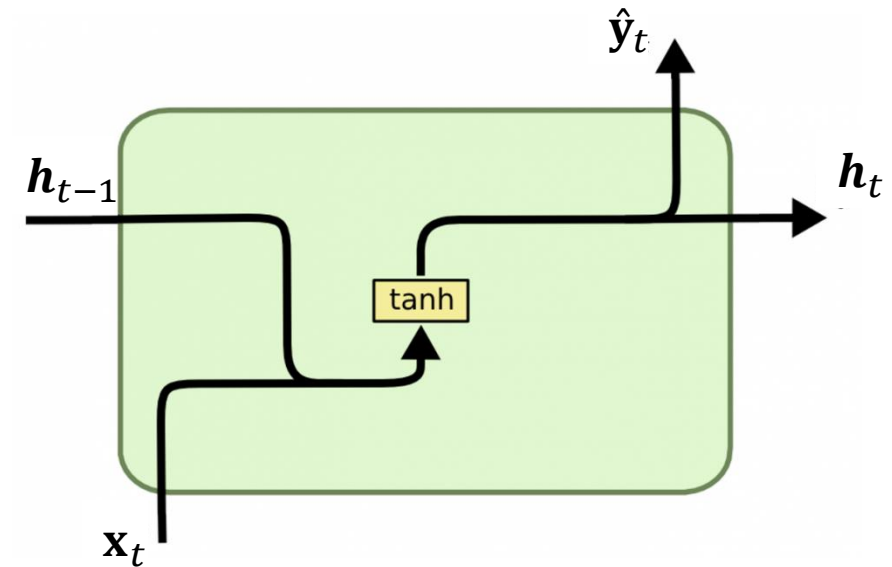
- **Idea:** Update hidden state \mathbf{h}_t based on input \mathbf{x}_t and previous hidden state \mathbf{h}_{t-1} using the same shared parameters at each time step
- **Hidden state \mathbf{h}_t :** Internal representation of information within a NN, here the “memory”
 - **Infinite memory:** \mathbf{h}_t is a function of all previous inputs
 - Computation for step t can (in theory) use information from many steps back
- Allows for processing sequences of **variable length**, not only fixed-sized vectors
- Same parameters are applied on each step
- Parameters don't grow with sequence length



- **General equation:**
$$\mathbf{h}_t = f_h(\mathbf{h}_{t-1}, \mathbf{x}_t)$$
$$\mathbf{y}_t = f_y(\mathbf{h}_t)$$

Recurrent Neural Network (RNN)

- **Hidden state \mathbf{h}_t**
 - Linear combination of current input \mathbf{x}_t and previous hidden state \mathbf{h}_{t-1}
 - Initial hidden state \mathbf{h}_0
- **Output $\hat{\mathbf{y}}_t$**
 - Prediction based on current hidden state \mathbf{h}_t
- Activation function $\tanh()$ compresses data in range $[-1,1]$
- Can process any length input



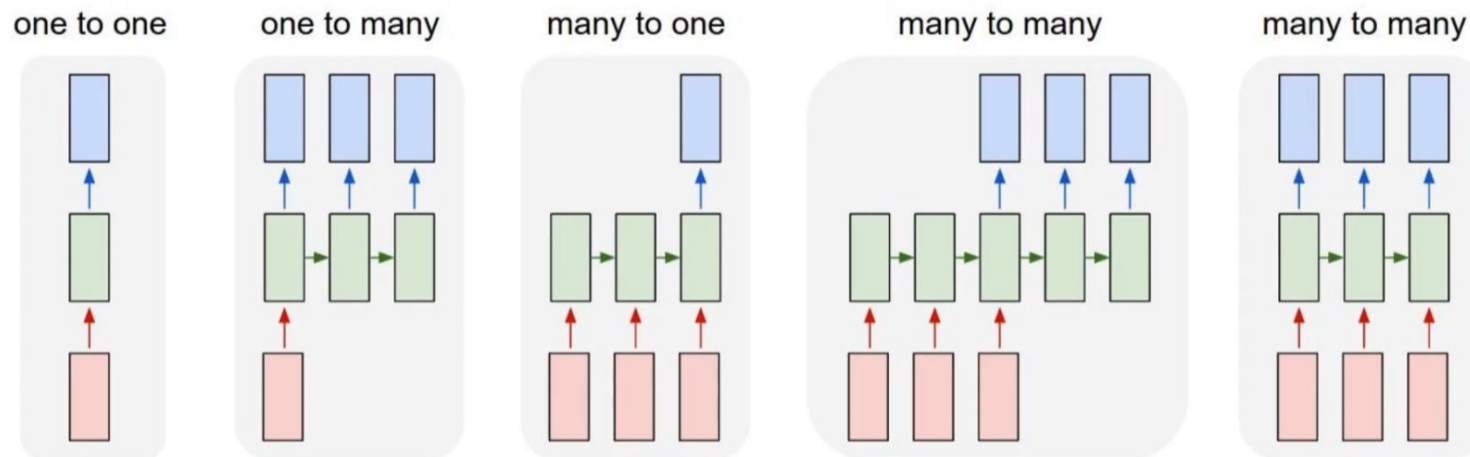
Equations:

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}_h)$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y)$$

Mapping Types

- **Previously:** Conventional feed forward networks (e.g. MLP, CNN, etc.)
 - **One to One:** E.g. image classification (image to class)
- **Now:** RNNs allow for processing variable length inputs and outputs
 - **One to many:** E.g. image captioning (image to text)
 - **Many to one:** E.g. sentiment analysis (text to sentiment)
 - **Many to many:** E.g. machine translation (sentence to sentence)



Training of RNN Language Models

- Given a big corpus of text, we sample sequences x_1, \dots, x_T of size T .
- In each step t , feed token x_t into LM, calculate output distribution \hat{y}_t and calculate loss $\mathcal{L}_t(\theta)$

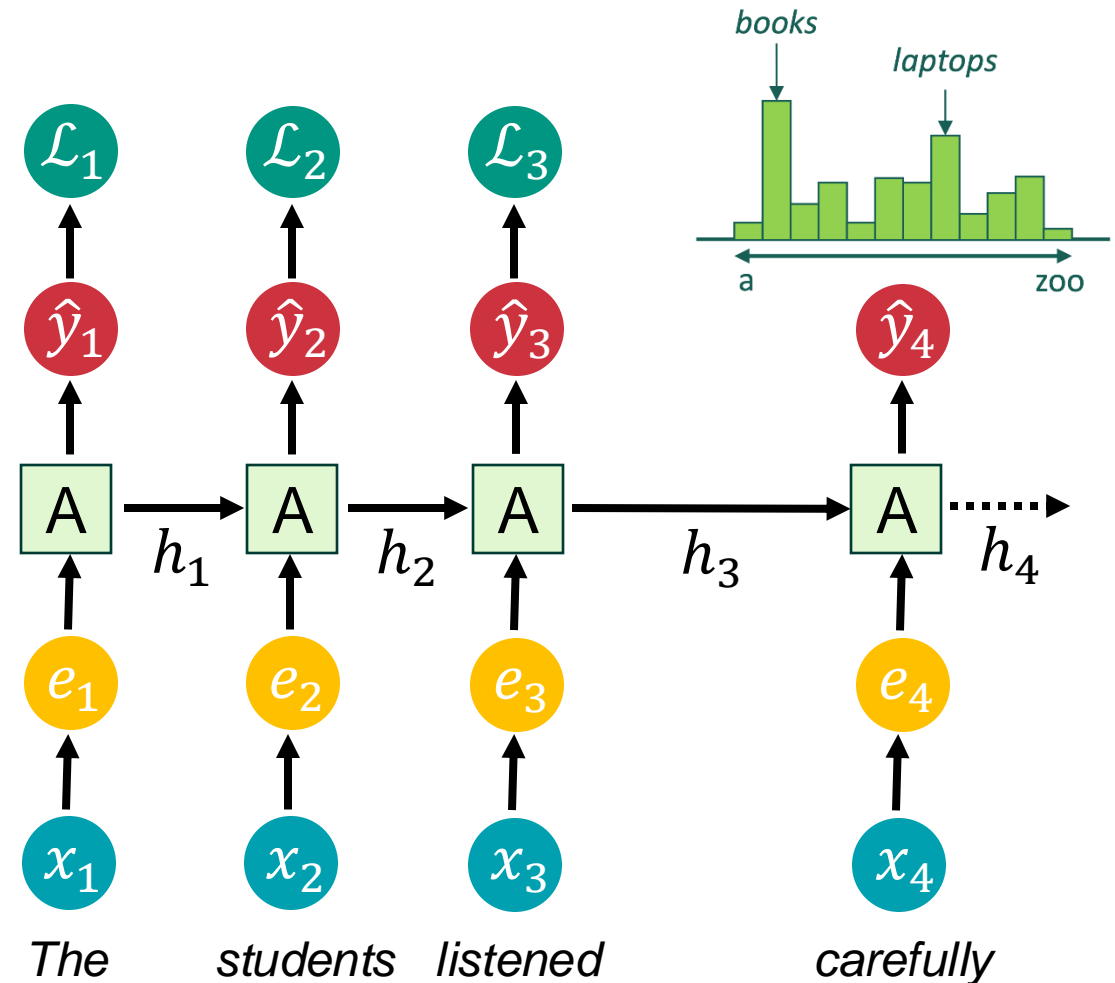
- Cross-entropy loss function:

$$\mathcal{L}_t(\theta) = \text{CE}(y_t, \hat{y}_t) = -\log p(\hat{y}_t = x_{t+1} | x_{1:t})$$

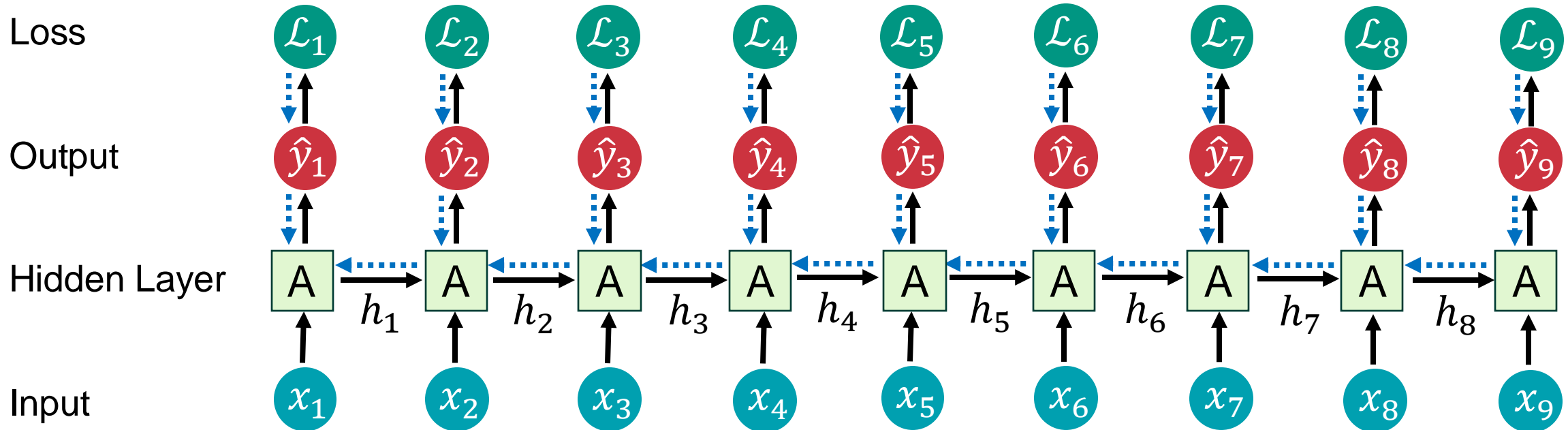
- Loss over one sequence:

$$\mathcal{L}(\theta) = \frac{1}{T} \sum_{t=1}^T \mathcal{L}_t(\theta)$$

- Teacher forcing: Model is fed the correct output from the previous timestep during training, rather than its own generated output.
 - Leads to faster convergence of training

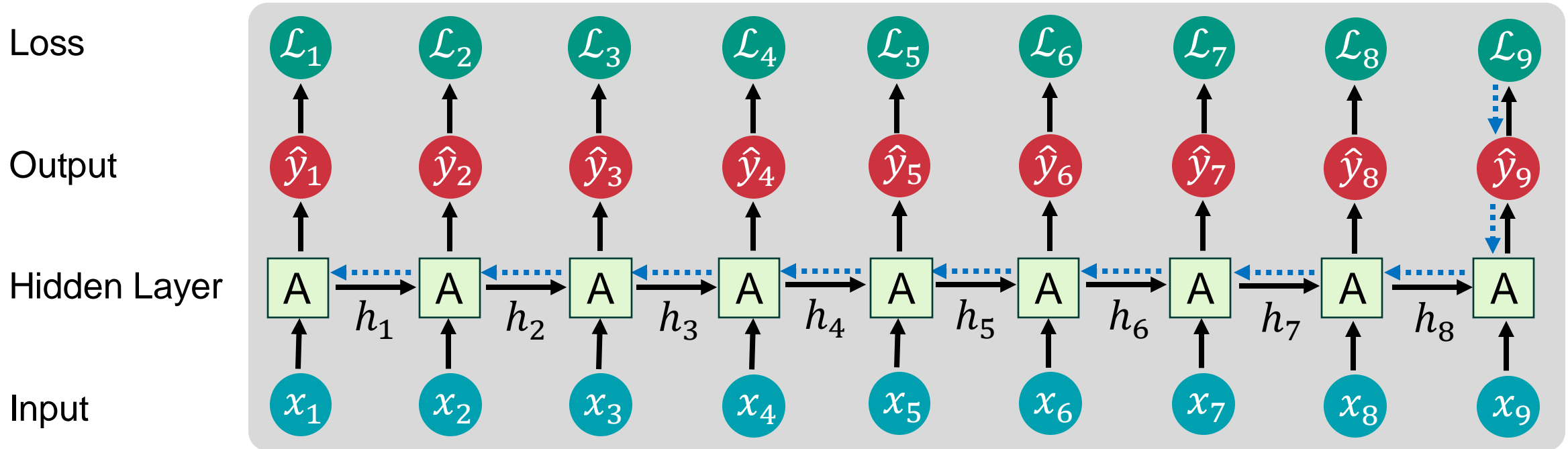


Backpropagation Through Time



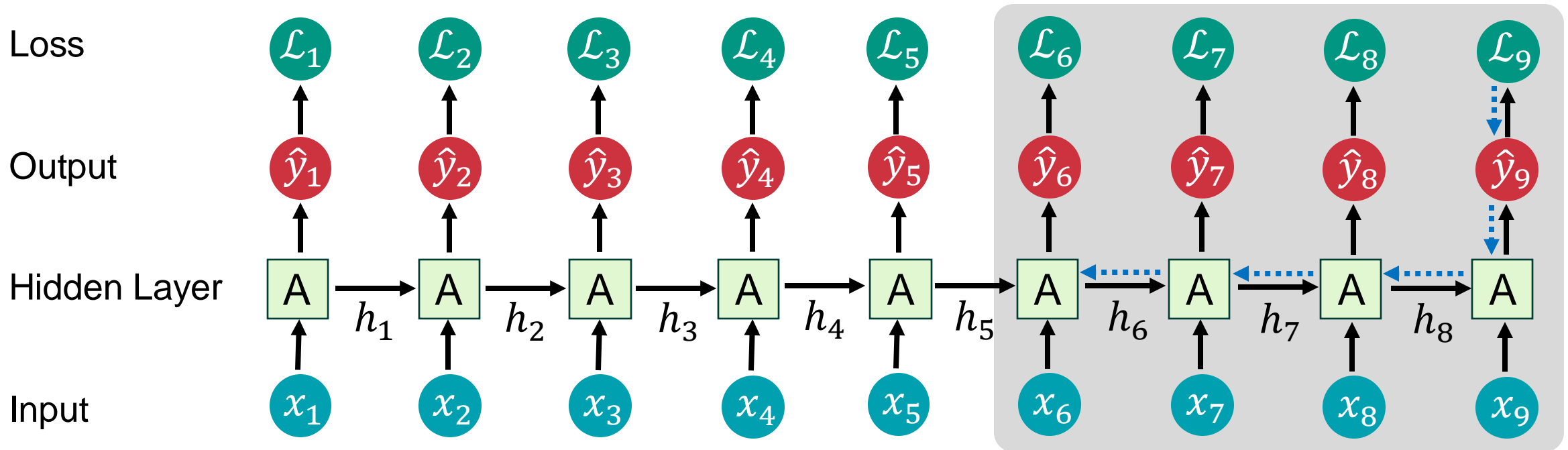
- To train RNNs, we backpropagate gradients through time
- As all hidden RNN cells share the same parameters, gradients get accumulated over time
- However, this is very quickly intractable (memory) for large sequences

Truncated Backpropagation Through Time



- **Idea:** Propagate gradient only back for k timesteps instead of all the way to the beginning
- But carry hidden state forward in time forever
- In practice $k \sim 20$ timesteps

Truncated Backpropagation Through Time



- **Idea:** Propagate gradient only back for k timesteps instead of all the way to the beginning
- But carry hidden state forward in time forever
- In practice $k \sim 20$ timesteps

Vanishing / Exploding Gradients

- Vanilla RNNs tend to produce vanishing or exploding gradients
- Let's consider an RNN with only scalar variables and a linear activation function

$$h_t = w_h h_{t-1} + w_x x_t + b_h$$

- Consider gradient of the loss \mathcal{L}_i on step i , with respect to the hidden state h_j on some previous step j . Let $l = i - j$

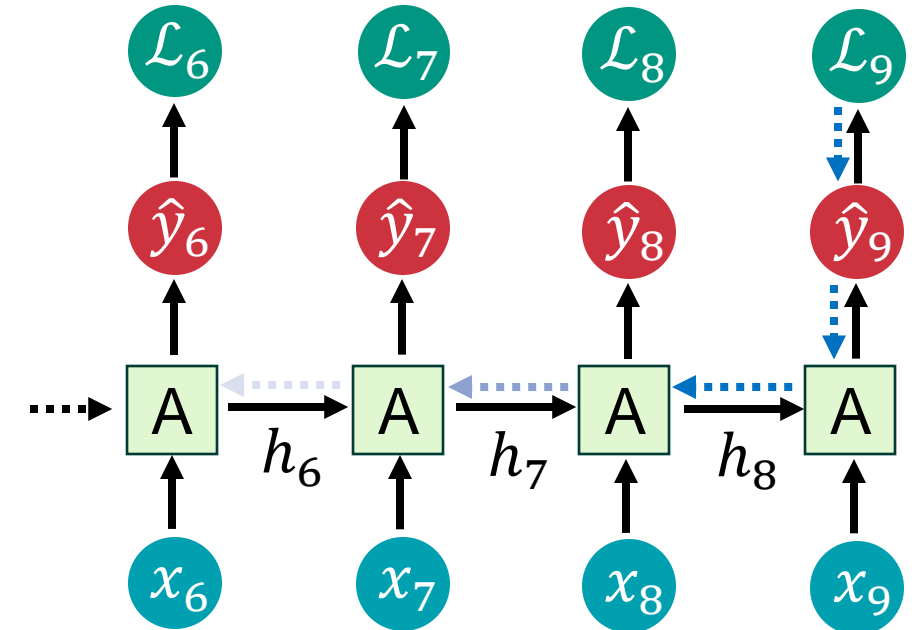
$$\frac{\partial \mathcal{L}_i}{\partial h_j} = \frac{\partial \mathcal{L}_i}{\partial h_i} \prod_{j < t \leq i} \frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \mathcal{L}_i}{\partial h_i} w_h^l$$

← Becomes exponentially problematic

Vanishing / Exploding Gradients

$$\frac{\partial \mathcal{L}_i}{\partial h_j} = \frac{\partial \mathcal{L}_i}{\partial h_i} \prod_{j < t \leq i} \frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \mathcal{L}_i}{\partial h_i} w_h^l$$

- For $w_h^l > 1$: **Exploding gradient** (becomes very large)
 - Matrix: Eigenvalues of W_h greater than 1
 - Causes numerical instability
- For $w_h^l < 1$: **Vanishing gradient** (becomes very small)
 - Matrix: Eigenvalues of W_h less than 1
 - Causes model weights only **updated with respect to near effects**, not long-term effects



Gradient Clipping

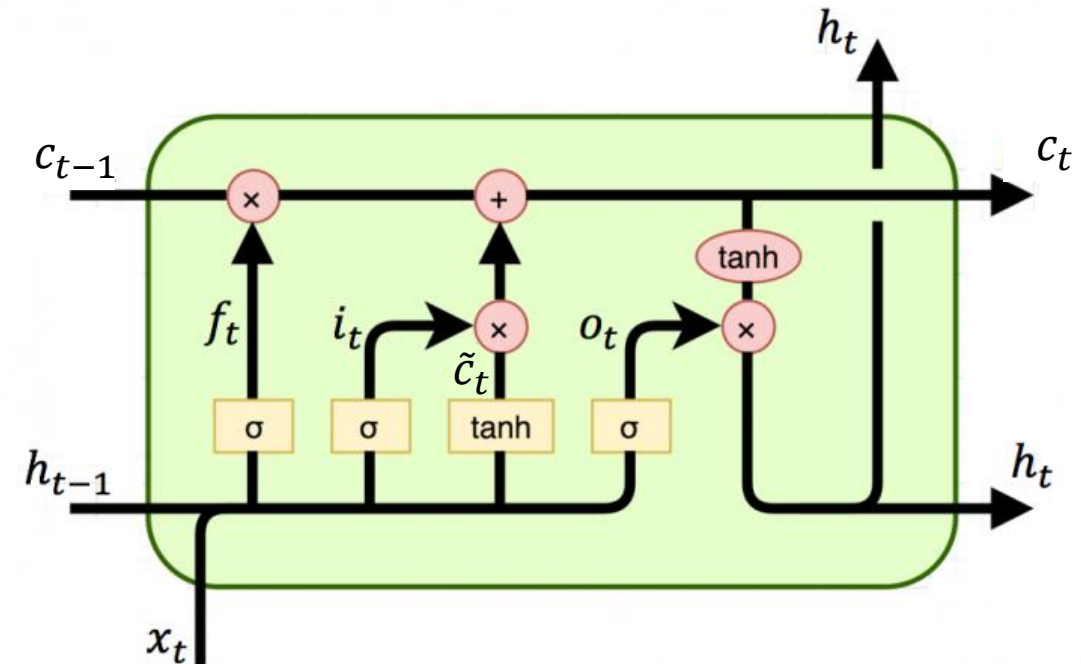
- **Gradient clipping:** If the norm of the gradient $\frac{\partial \mathcal{L}}{\partial \theta}$ exceeds some predefined threshold τ , clip the norm.

$$\hat{\mathbf{g}} = \begin{cases} \frac{\partial \mathcal{L}}{\partial \theta}, & \left\| \frac{\partial \mathcal{L}}{\partial \theta} \right\| < \tau \\ \frac{\tau}{\left\| \frac{\partial \mathcal{L}}{\partial \theta} \right\|} \frac{\partial \mathcal{L}}{\partial \theta}, & \left\| \frac{\partial \mathcal{L}}{\partial \theta} \right\| \geq \tau \end{cases}$$

- **Intuition:** Take a step in the same direction, but with smaller step size
- Solves exploding, but not vanishing gradient!

Long Short-Term Memory RNNs

- A solution to the problem of vanishing gradients in RNNs
- Introduces a **cell state** and a **hidden state**
 - The cell stores long-term information
 - Cell is conceptually like RAM in a computer
- Introduces **gates for filtering information**
 - Control which information is read, erased, written to/from the cell
 - Gates can be open (1) or closed (0) or in-between
 - Values depend on the current context



Long Short-Term Memory RNNs (LSTMs)

- **Forget gate** f_t : Controls what of the previous cell content is forgotten
- **Input gate** i_t : Controls which part of the new candidate cell content is written to cell
- **Output gate** o_t : Controls which part of the cell is output to the hidden state
- **Candidate cell content** \tilde{c}_t : Potential new cell content that can be written to cell
- **Current cell state** c_t
- **Current hidden state** h_t

■ Gates

Sigmoid function: All gates are between 0 and 1

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$

■ Cell and hidden state

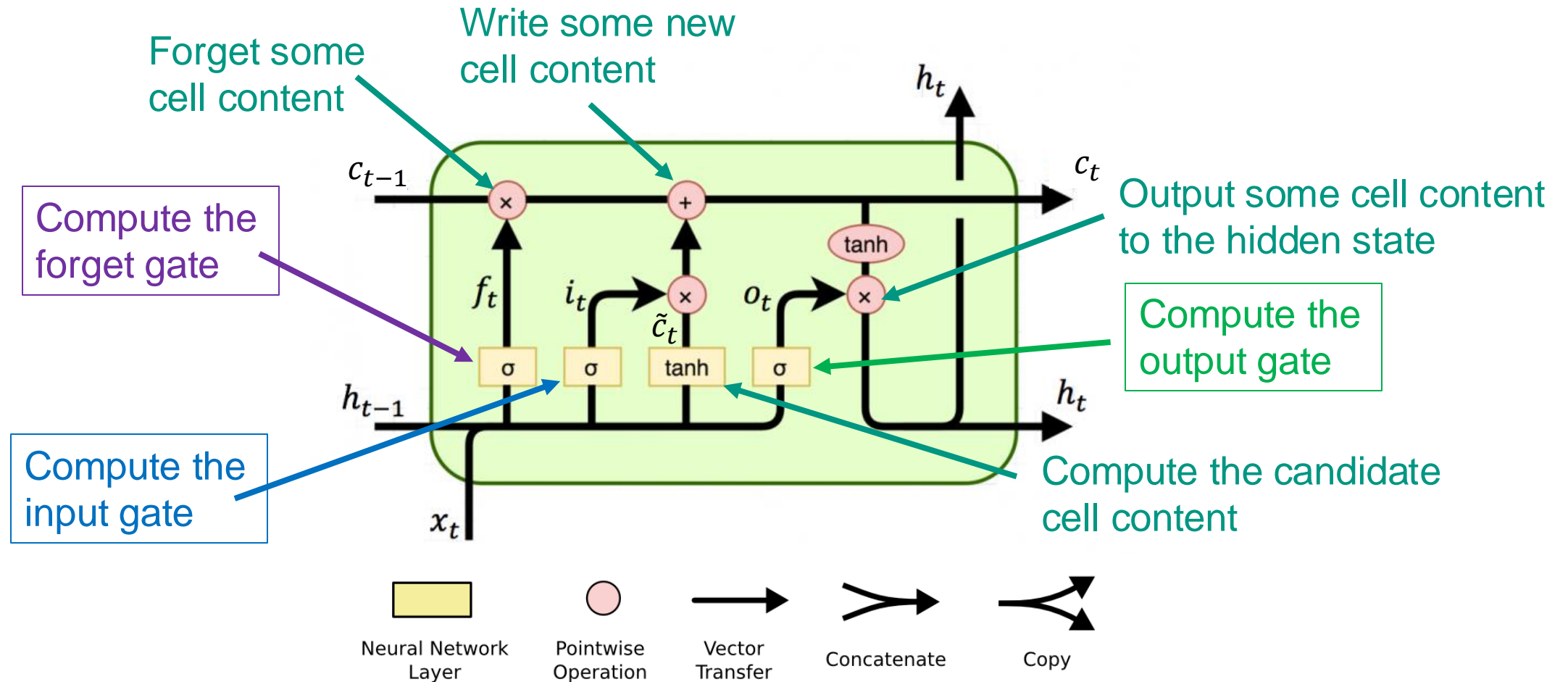
$$\tilde{c}_t = \tanh(W_c h_{t-1} + U_c x_t + b_c)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

$$h_t = o_t \odot \tanh(c_t)$$

Hadamard product:
Element-wise multiplication

Long Short-Term Memory RNNs (LSTMs)

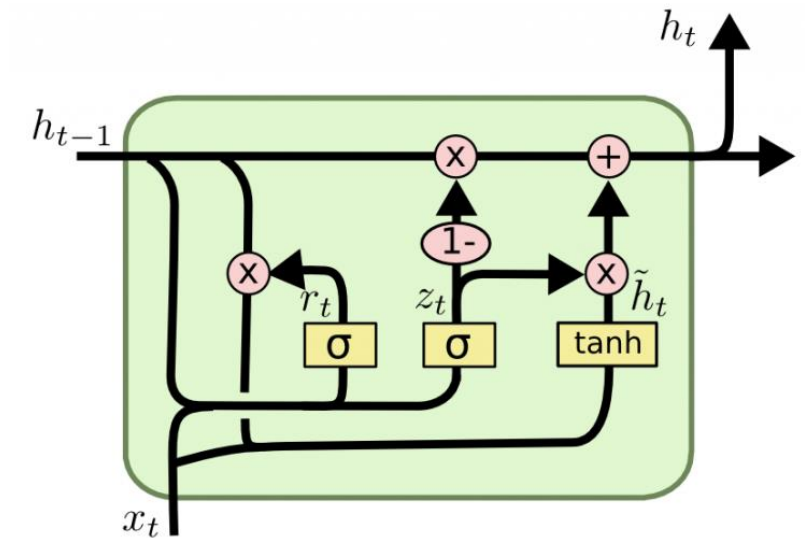
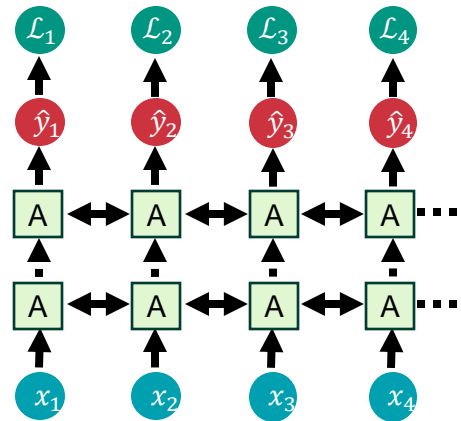
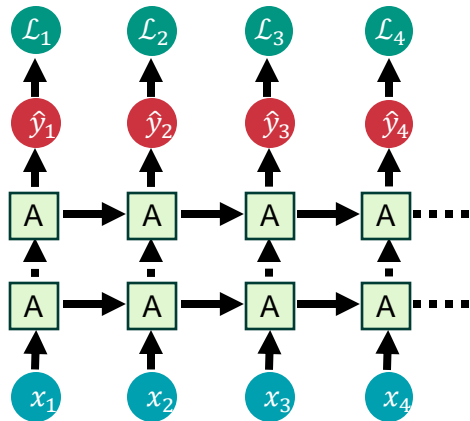


Long Short-Term Memory RNNs (LSTMs)

- The **LSTM architecture** makes it much easier for an RNN to preserve **information** over many timesteps
- **Theoretical example:**
 - Set forget gate to 1
 - Set input gate to 0
 - Cell information is preserved indefinitely
- **Practical results:**
 - RNN: Preserves information **over 7 timesteps**
 - LSTM: Preserves information **over 100 timesteps**

Other RNN Flavors

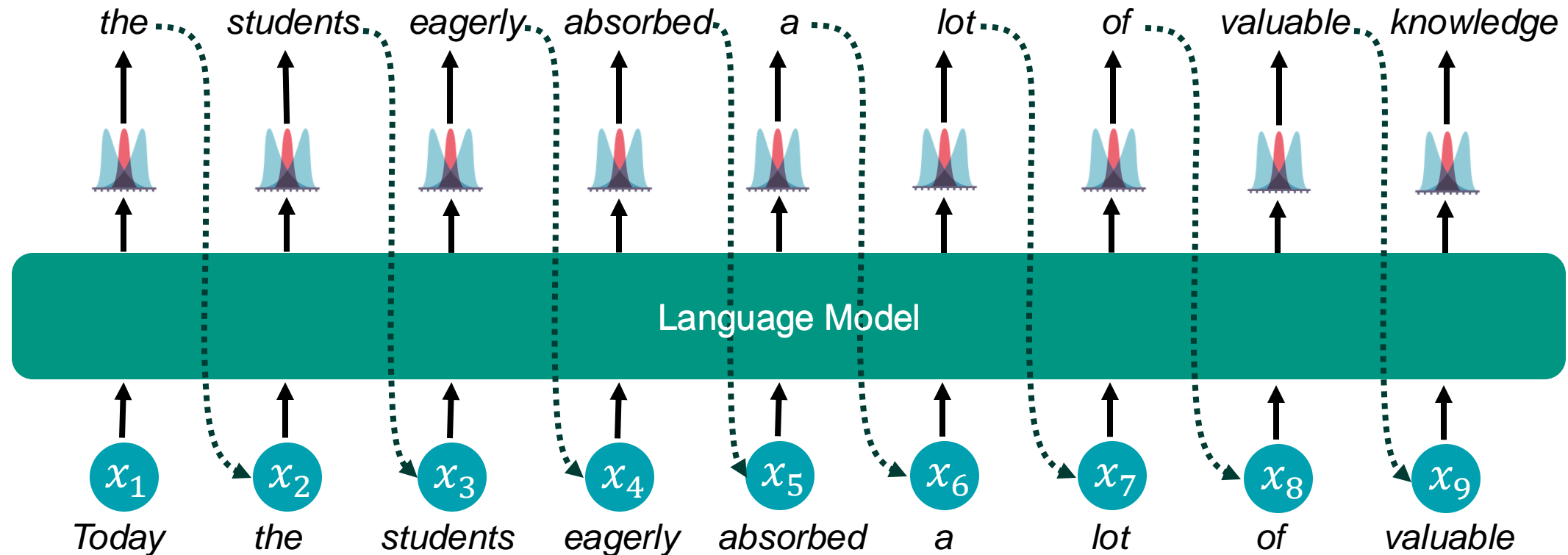
- Multi-Layer/Stacked RNNs
- Bidirectional RNNs
- Simplified LSTM architectures
 - Gated Recurrent Unit (GRU)
 - Update Gate RNN (UGRNN)



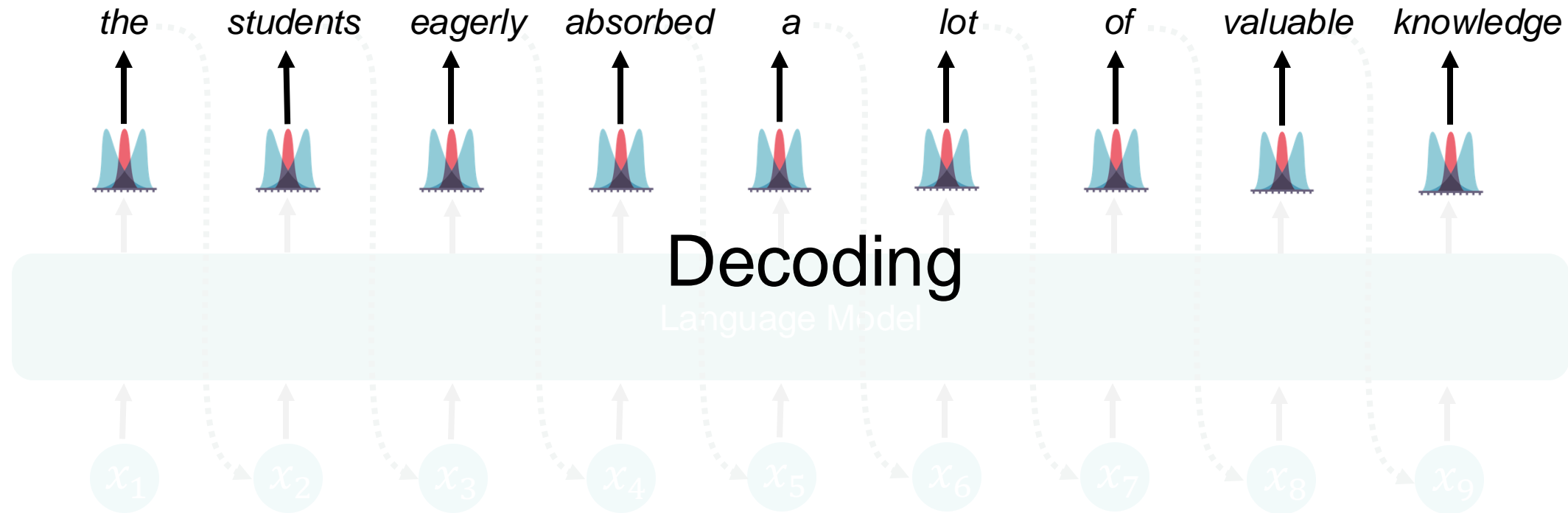
Outline

- Introduction
- Text Normalization (Preprocessing)
- Text Representation
- Traditional Language Models
- Neural Language Models
- Decoding
- Literature

Recap Autoregressive Text Generation



Recap Text Generation



Decoding

- Describes the **process of generating human-readable text from a machine-readable representation** (e.g. a sequence of symbols or numerical values)
- Decoding algorithm defines a function $g()$, which selects a token from the distribution p

$$\hat{x}_t = g(p(x_t|x_{1:t-1}))$$

Decoding algorithm

Greedy Decoding

- Selects token with highest probability in $p(\mathbf{x}_t | \mathbf{x}_{1:t-1})$ for each step

$$\hat{\mathbf{x}}_t = \operatorname{argmax}_{\mathbf{w} \in V} p(\mathbf{x}_t = \mathbf{w} | \mathbf{x}_{1:t-1})$$

- **Problem:** Token is only locally optimal, but might not be the best choice in the long term (globally).

- We have no way of undoing our choice!

- Example: Translate “*Es ist, was es ist*”

- *It* _____

- *It is* _____

- *It is* **a** _____

- How to fix this?

Exhaustive Search Decoding

- **Goal:** Find a sequence \mathbf{x} (length T) that maximizes

$$\begin{aligned} p(\mathbf{x}_1, \dots, \mathbf{x}_T) &= p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_1, \mathbf{x}_2) \dots p(\mathbf{x}_T|\mathbf{x}_1, \dots, \mathbf{x}_{T-1}) \\ &= \prod_{t=1}^T p(\mathbf{x}_t|\mathbf{x}_1, \dots, \mathbf{x}_{t-1}) \end{aligned}$$

- **Naïve idea:** Try computing all possible sequences $\mathbf{x}_1, \dots, \mathbf{x}_T$ and take the one that maximizes $p(\mathbf{x}_1, \dots, \mathbf{x}_T)$
- **Problem:** On each step t of the decoding, we need to keep track of $|V|^t$ partial translations, where $|V|^t$ is the vocabulary size
 - $O(V^T)$ complexity is far too expensive!

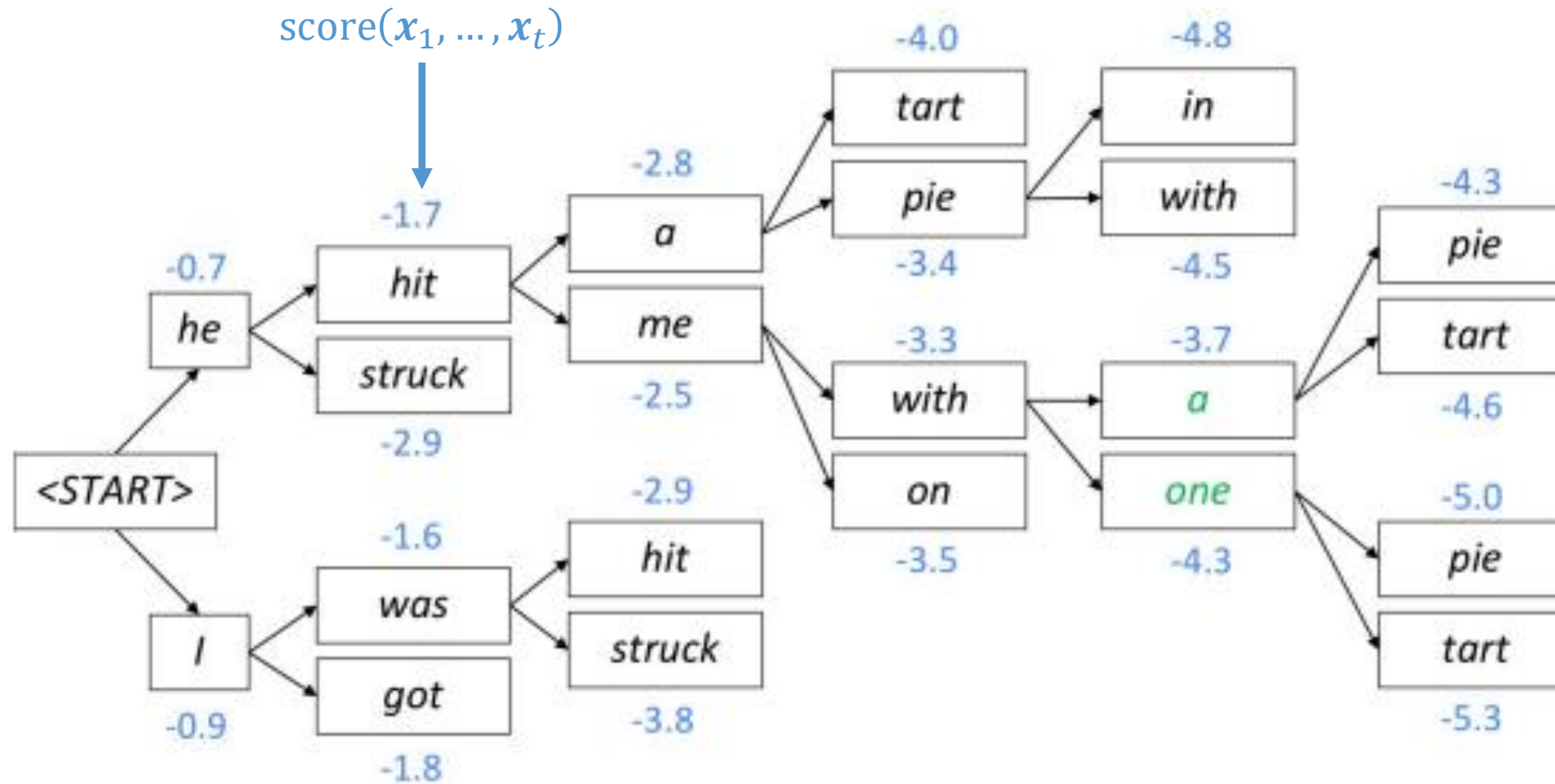
Beam Search Decoding

- **Core idea:** On each step of decoding, keep track of the k most probable (highest score) partial translations (which we call hypotheses)
- **Beam size:** Hyperparameter k (in practice around 5 to 10 for NMT)
- The score of a hypothesis $\mathbf{x}_1, \dots, \mathbf{x}_T$ is its log probability

$$\text{score}(\mathbf{x}_1, \dots, \mathbf{x}_T) = \log p(\mathbf{x}_1, \dots, \mathbf{x}_T) = \sum_{t=1}^T \log p(\mathbf{x}_t | \mathbf{x}_1, \dots, \mathbf{x}_{t-1})$$

- Beam search is only a heuristic method → Not guaranteed to find optimal solution!
- Much more efficient than exhaustive search

Beam Search Decoding Example



Diverse Requirements

- Depending on the task, text generation has diverse requirements on the output space.



- **Non-open-ended generation:** Output is mostly determined by input
- **Open-ended generation:** Output distribution still has high freedom
- **Conclusion:** Different tasks require different decoding/training approaches!

Neural Text Degeneration

- Decoding with the goal to get the most likely string (greedy, beam search) leads to repetition in open-ended text generation!

Context: In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Beam Search, $b=100$:

"The unicorns were very intelligent, and they **were very intelligent**," said Dr. David S. Siegel, a professor of anthropology at the University of California, Berkeley. "**They were very intelligent, and they were very intelligent, and they were very intelligent.**"

↑
Repetition

Pure Sampling:

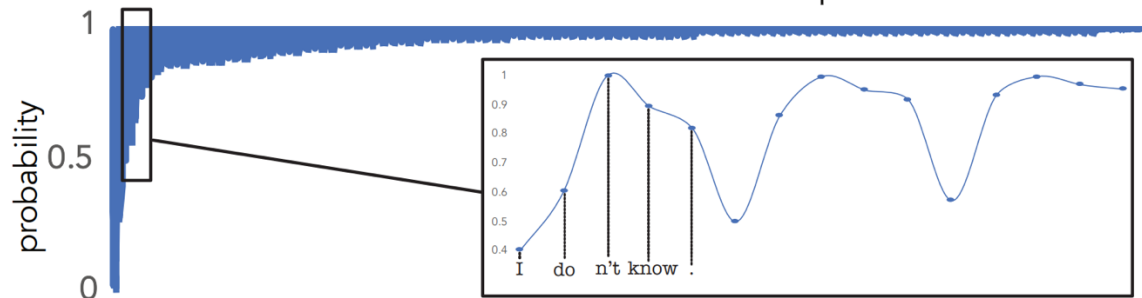
The researchers credited these Mages with building Fabian leather armor. Representing an increasing **global problem for consumers**, the Fabians encouraged different types of leather to be made by **various models of the m animal. A bearded, 350-pound maurice** was engaged in strenuous natural exercise, which was covered by waterproof clothing.

↑
Incoherent gibberish

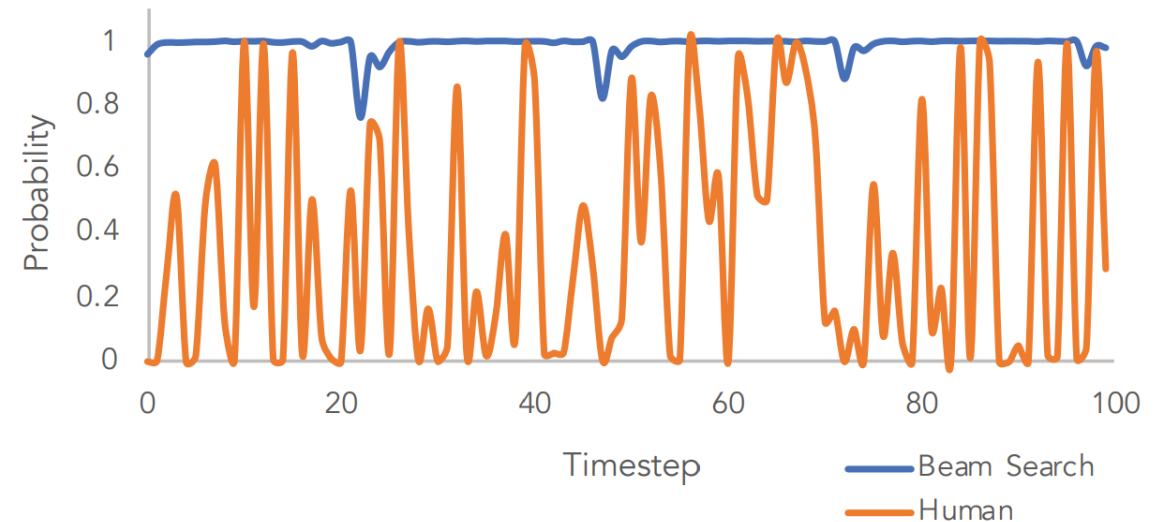
(Holtzman et al., 2020)

Neural Text Degeneration

- Token probability of a repeated phrase increases with each repetition, creating a positive feedback loop.
- **Example:** „I don't know“ repeated 200 times
- **Observation:** Model becomes more and more confident



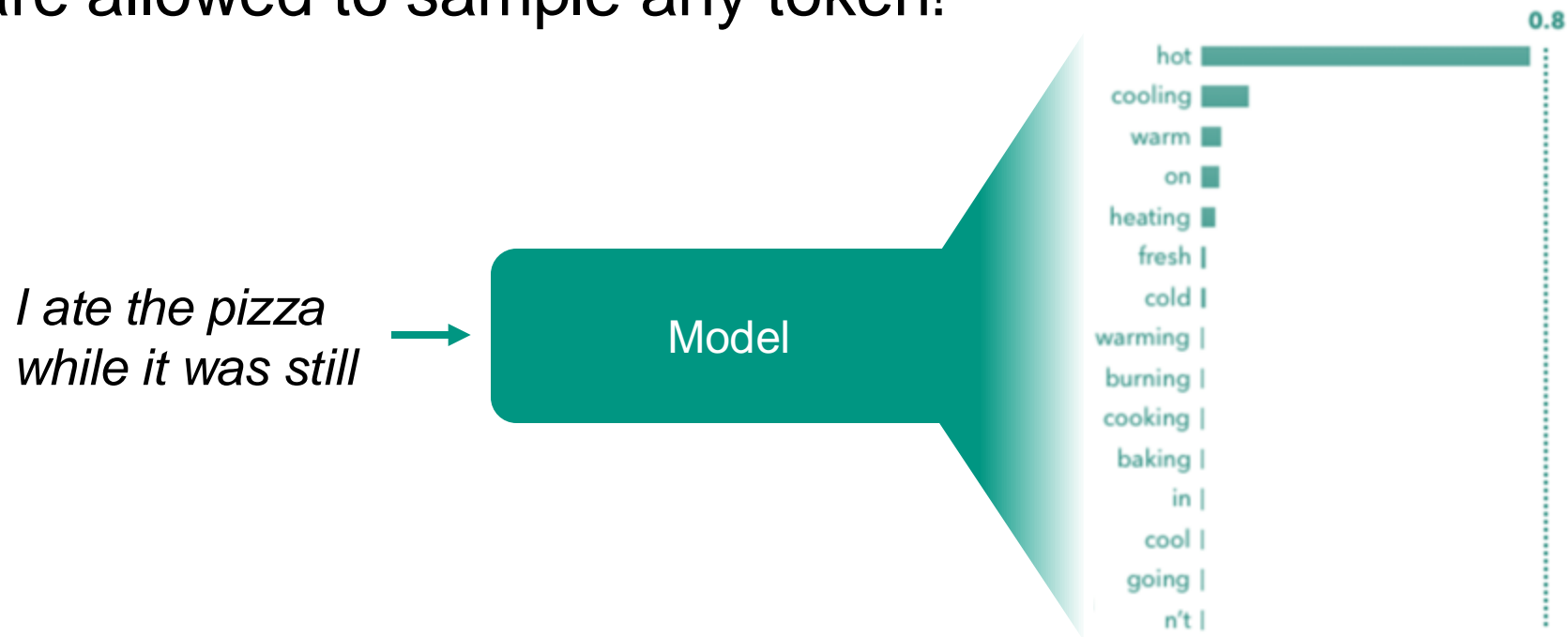
- **Natural language doesn't maximize probability**
- **Example:** Token probability of text generated by beam search and humans
- Human text is characterized by much more variability!



[\(Holtzman et al., 2020\)](#)

Sampling

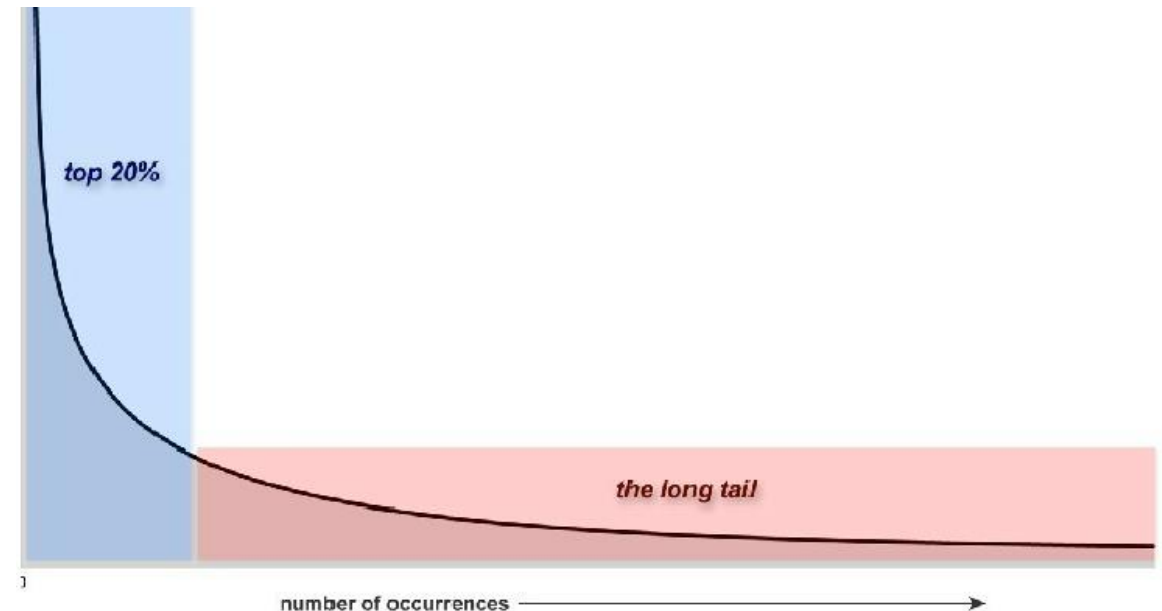
- Sample tokens according to the distribution
$$\hat{x}_t \sim p(x_t = w | \mathbf{x}_{1:t-1})$$
- You are allowed to sample any token!



(Holtzman et al., 2020)

Problems of Naïve Sampling

- Vanilla sampling makes every token in the vocabulary an option
- Token probability distributions are “heavy tailed”
 - Most probability mass is over a limited set of tokens
 - But the tail is very long (remember $V \sim 30.000$) and aggregates to a considerable probability mass
- Many tokens with low probability are really wrong, given the current context
- But low probability tokens as a group have a high chance of being selected.
- **Solution:** Top-k, top-p sampling



Top-k Decoding

- Only sample from the top k tokens in the probability distribution
 - Truncate distribution
- Common values are $k \approx 50$, but it is a hyperparameter
 - Big k leads to **more diverse**, but **risky** outputs
 - Small k leads to **less diverse**, but **safe** outputs

*I ate the pizza
while it was still*



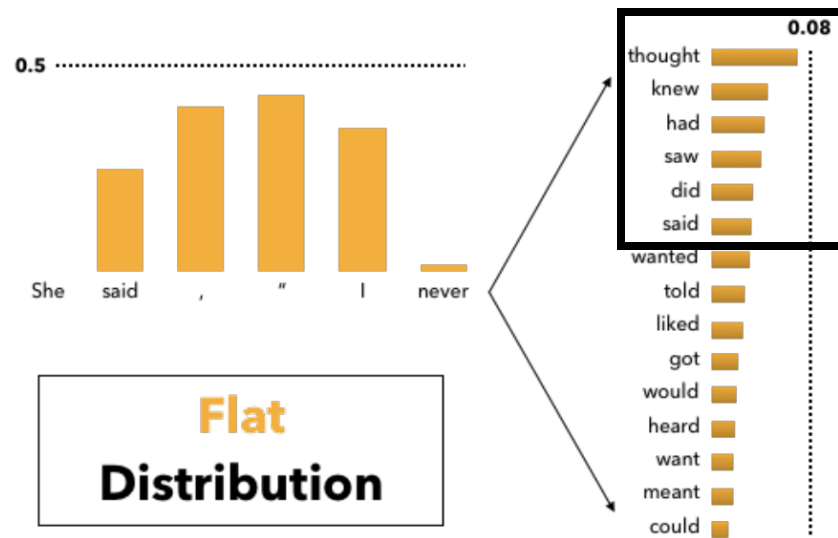
Model



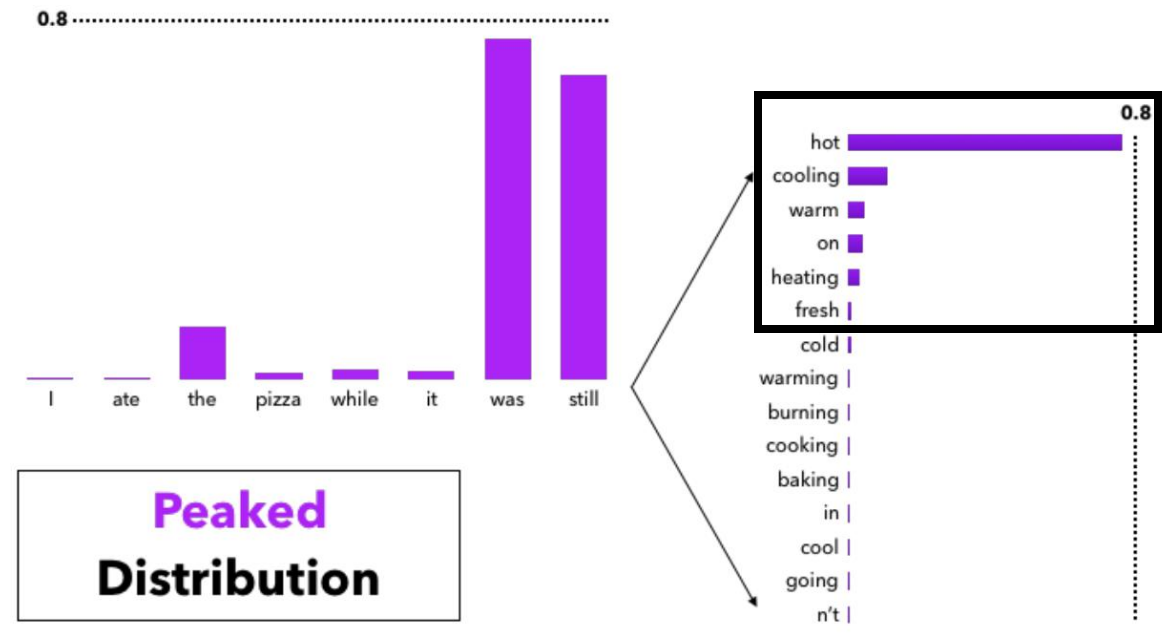
(Fan et al., 2018; Holtzman et al., 2018)

Top-k Decoding Issues

Cut off too quickly



Cut off too slowly



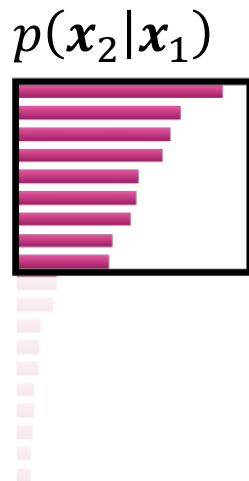
(Holtzman et al., 2020)

Top-p (Nucleus) Decoding

- **Idea:** Don't sample from a fixed amount of tokens, but a fixed amount of probability mass!
→ Sample from all tokens in the top p cumulative probability mass

$$\sum_{w \in V(p)} p(x_t = w | x_{1:t-1}) \geq p$$

- **Result:** Varies k depending on the uniformity of p



([Holtzman et al., 2020](#))

Outline

- Introduction
- Text Normalization (Preprocessing)
- Text Representation
- Traditional Language Models
- Neural Language Models
- Decoding
- Literature

Literature

- [Speech and Language Processing \(3rd ed. Draft\) - Dan Jurafsky, James H. Martin](#)
- [CS224N: Natural Language Processing with Deep Learning](#)
- [Deep Learning – Ian Goodfellow](#)