

Practical Natural Language Processing (Generative AI, LLM)

Using Python

Veerasak Kritsanapraphan, Software Park Thailand, July 2023

About myself?

- Graduated from San Francisco State University in Master of Science in Computer Information System, 1997
- PhD. at Chulalongkorn University, research focus on Data Science, Big Data, Mobile Computing and Internet of Thing (IOT)
- Senior Director at Arise by Infinitas
- Data Science and AI Independent Consultant
- Instructor for Software Park in Data Science Courses
-

Introduce yourself?

- What is your name/position/role in your organization?
- Tell you a bit about yourself?
- What is your passion?
- What is your expectation for this class?



Agenda

Day 1

1) Introduction to Natural Language Processing

- Definition and applications of NLP
- Challenges in NLP

2) Python for Natural Language Processing

- Basics of Python
- Python libraries for NLP (NLTK, Spacy, TextBlob)

3) Introduction to Thai NLP using PyThaiNLP

- Introduction to PyThaiNLP
- Basic Text Processing in Thai Language

4) Textual Sources and Formats

- API
- Social Media
- Web Scraping
- Building your Corpus

Agenda

Day 2

5) Text Processing and Analysis

- Tokenization, N-grams, Scriptio Continua
- Stemming and Lemmatization, Synsets and Hypernyms
- Part of Speech Tagging
- Named Entity Recognition

6) Thai Text Processing and Analysis using PyThaiNLP

- Tokenization, Part of Speech Tagging, and Named Entity Recognition in Thai

7) Word Embeddings and Language Models

- Bag of Words, TF-IDF
- Word2Vec, GloVe
- Understanding and implementing BERT (Bidirectional Encoder Representations from Transformers)

8) Introduction to Transformers Models

- Understanding Transformer Architecture
- Overview of BERT (Bidirectional Encoder Representations from Transformers)

Agenda

Day 3

9) Introduction to Large Language Models

- Understanding LLM and its significance in NLP
- Deep dive into GPT (Generative Pretrained Transformer) and its variants, including GPT3, GPT4 and ChatGPT
- Opensource alternatives of ChatGPT
- Practical Application and Use cases

10) Sentiment Analysis and Text Classification

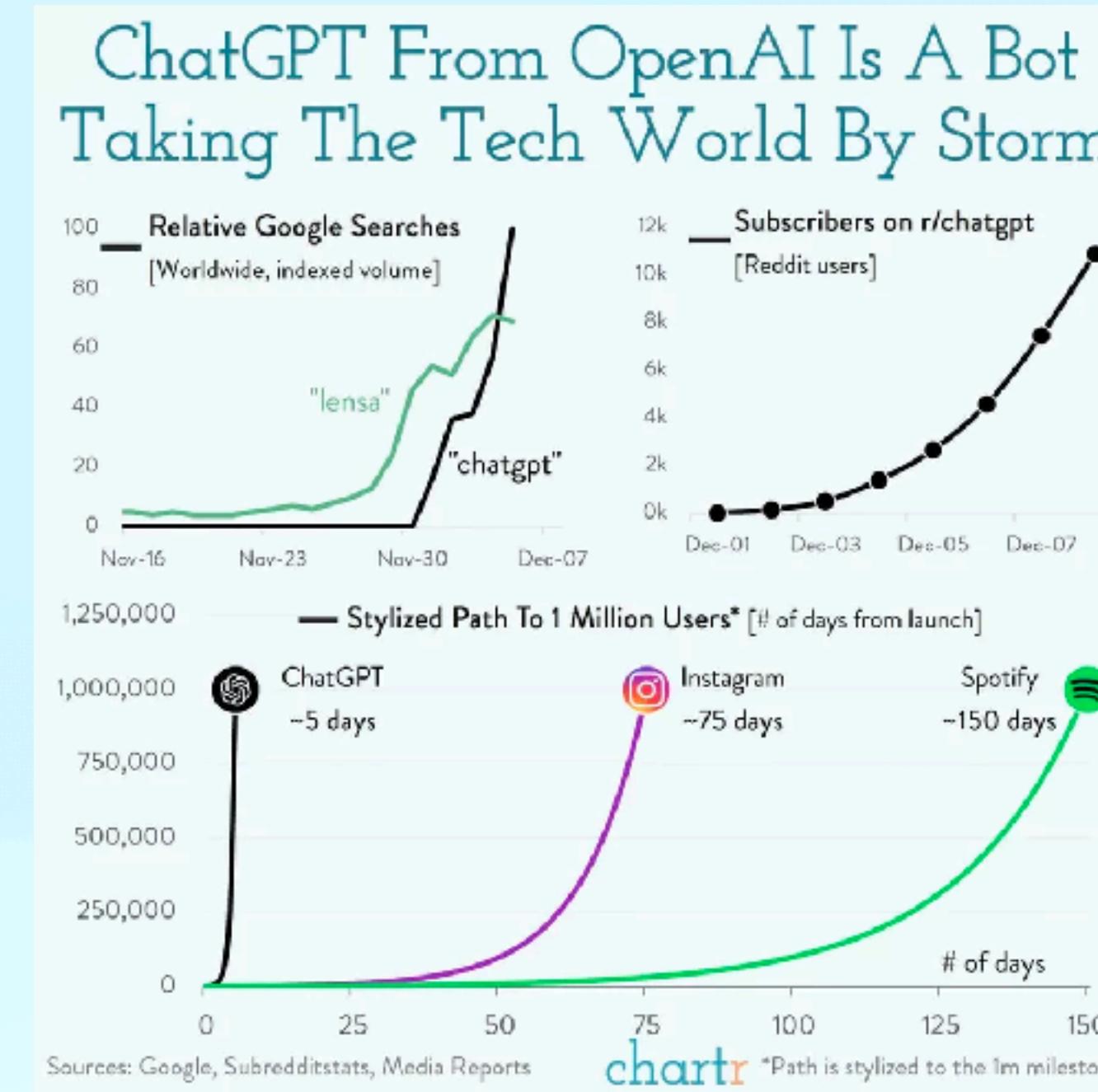
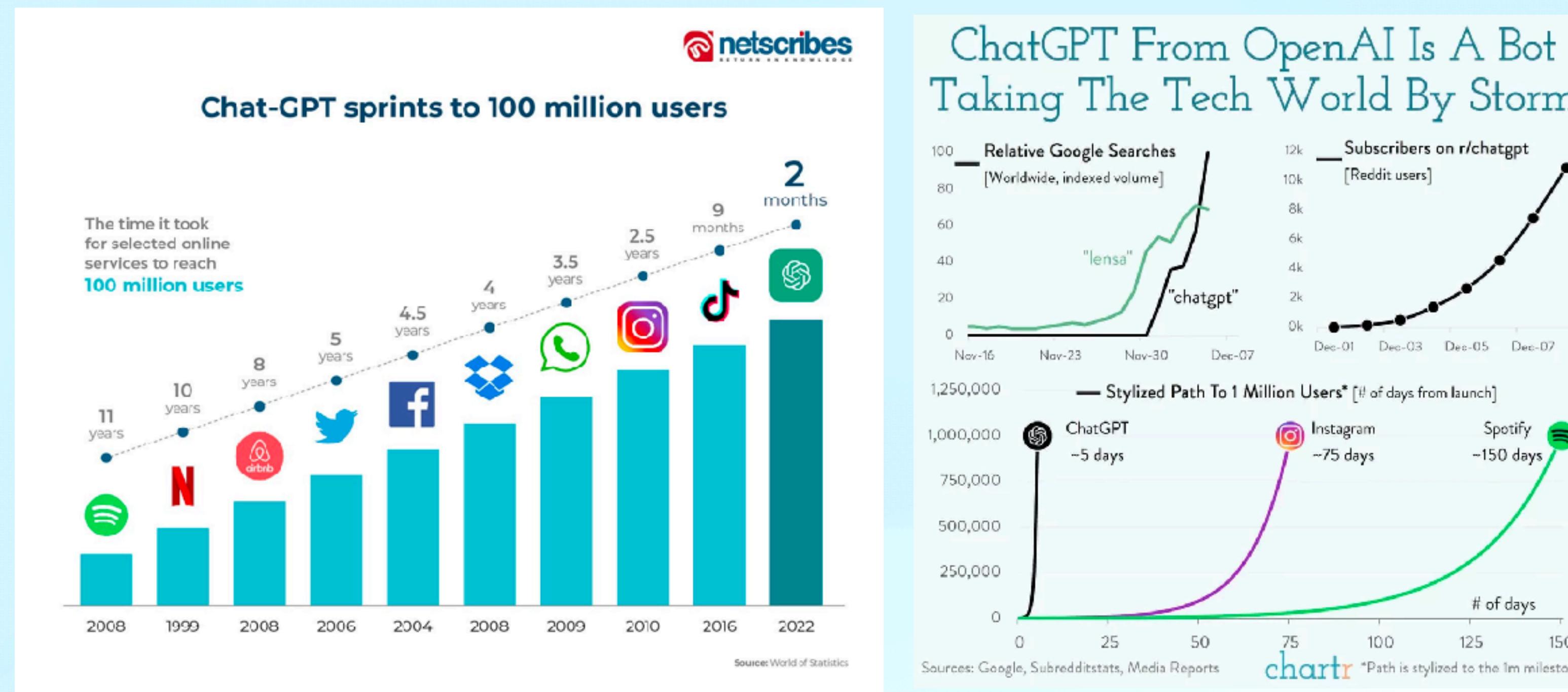
- Understanding sentiment analysis and Text Classification
- Implementing sentiment analysis and Text Classification with Python

11) Introduction to Chatbots

- Understanding chatbots and their working
- Implementing a basic chatbot with Python

12) Real-world NLP Applications and Future Trends

- Overview of real-world use cases of NLP
- Discussing the future of NLP and advancements in the field
- Applications and trends in Thai NLP



TECH

'Godfather of AI' urges governments to stop machine takeover

PUBLISHED : 29 JUN 2023 AT 08:45

WRITER: AFP

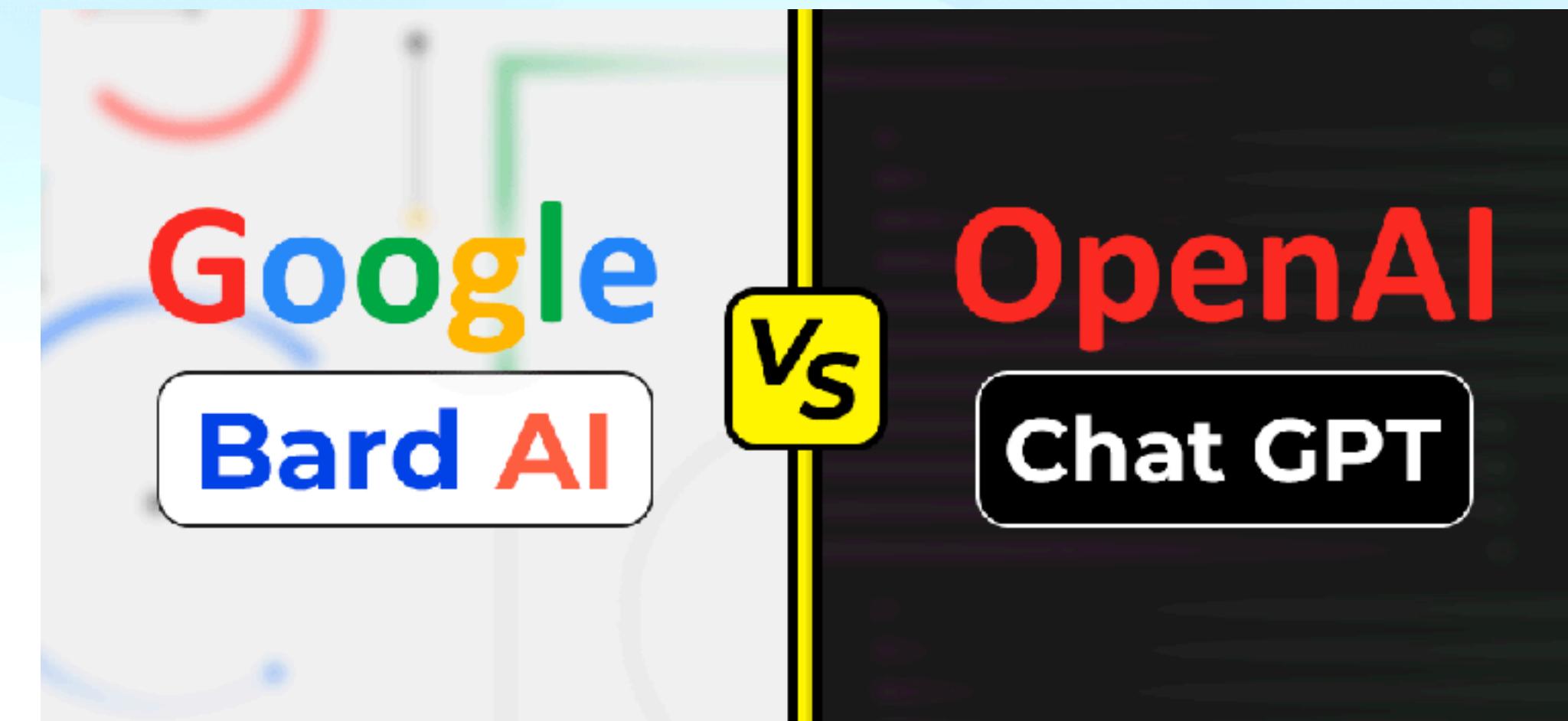


Computer scientist Geoffrey Hinton, known as the 'godfather of AI' speaks during the Collision Tech Conference in Toronto, Canada

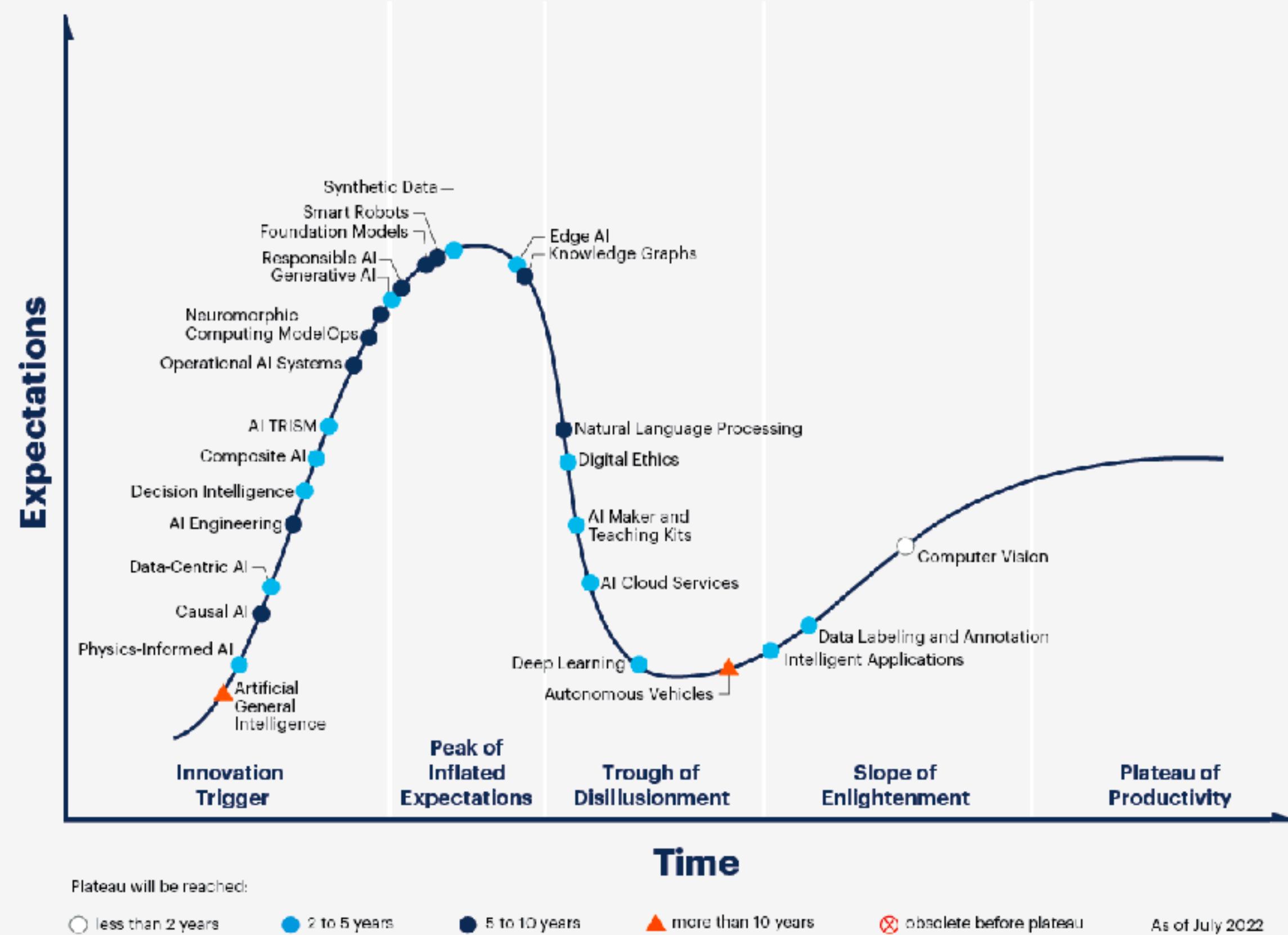
Technology | AI

Apple Tests 'Apple GPT,' Develops Generative AI Tools to Catch OpenAI

- Company builds large language models and internal chatbot
- Executives haven't decided how to release tools to consumers



Hype Cycle for Artificial Intelligence, 2022

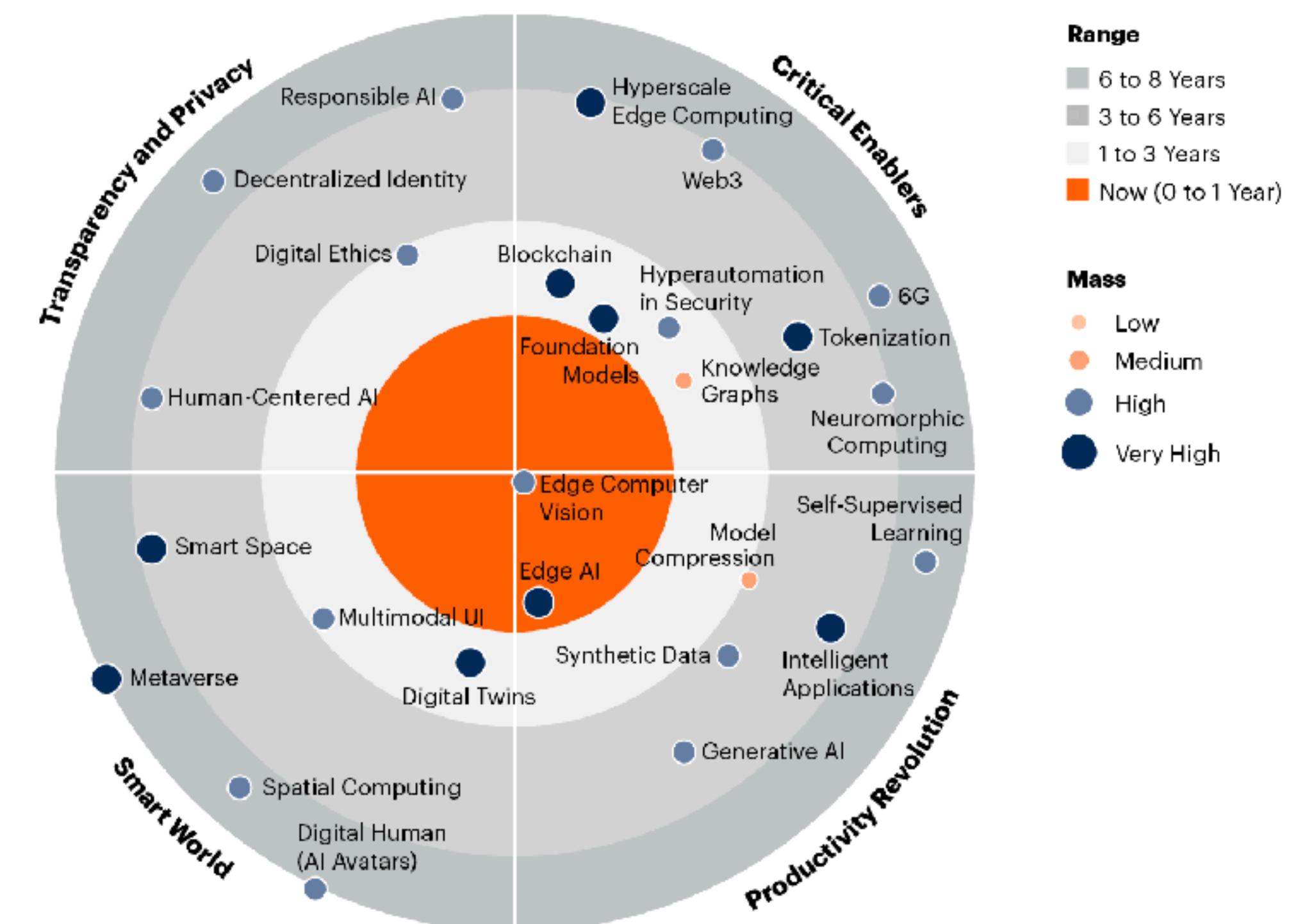


gartner.com

Source: Gartner
© 2022 Gartner, Inc. and/or its affiliates. All rights reserved. Gartner and Hype Cycle are registered trademarks of Gartner, Inc. and its affiliates in the U.S. 1957009

Gartner®

2023 Gartner Emerging Technologies and Trends Impact Radar



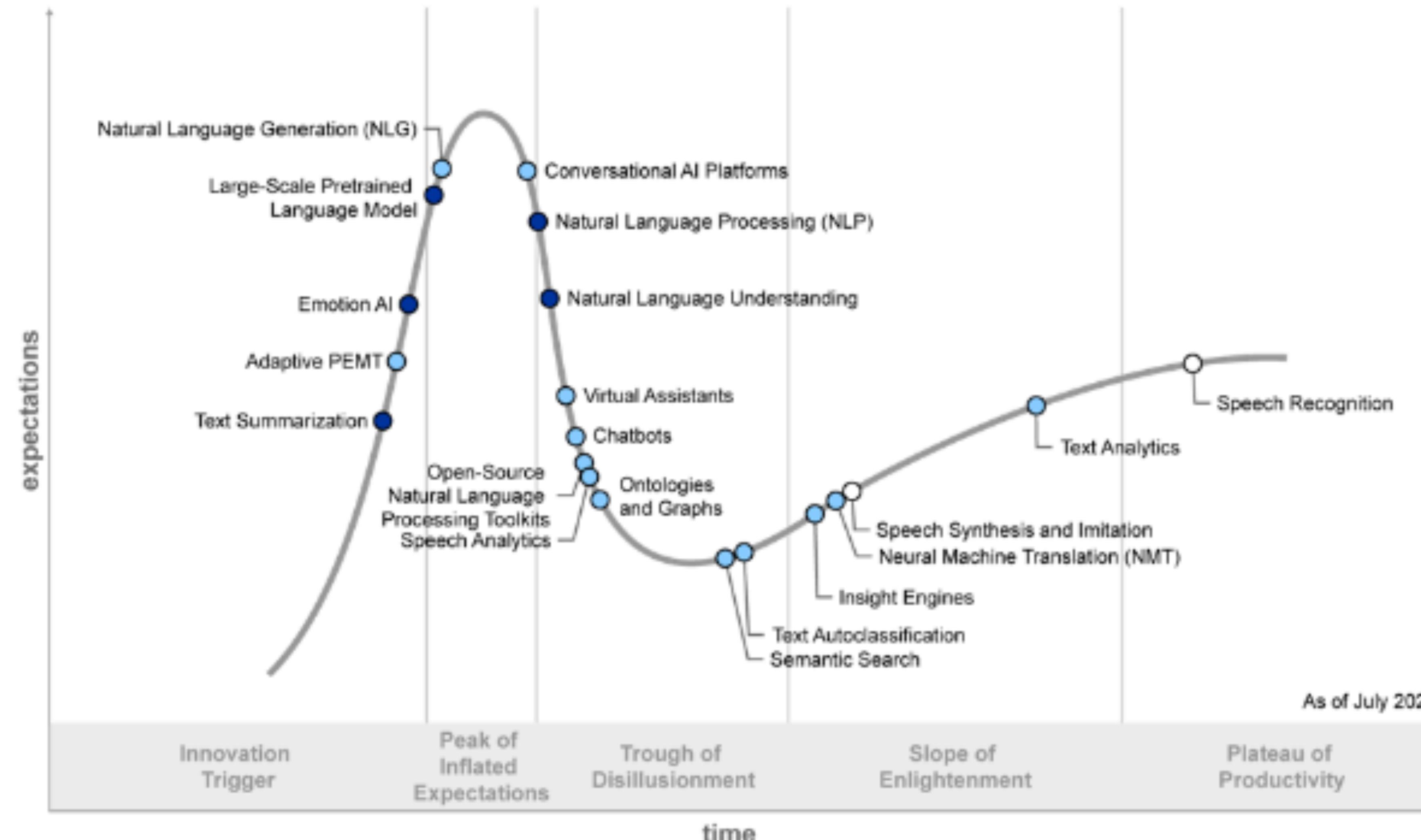
gartner.com

Note: Range measures number of years it will take the technology/trend to cross over from early adopter to early majority adoption. Mass indicates how substantial the impact of the technology or trend will be on existing products and markets.

Source: Gartner
© 2023 Gartner, Inc. All rights reserved. OM_G15_2034284

Gartner®

Hype Cycle for Natural Language Technologies, 2020



Plateau will be reached:

- less than 2 years
- 2 to 5 years
- 5 to 10 years
- ▲ more than 10 years
- ✖ obsolete before plateau

	Old	New
	Clicks, Forms, Buttons	Chat, voice, video
	Guess	Understand
	Static	Dynamic
	Computer	Personal / Human

Natural Language UI - The Next Big Leap in UX?



Pasi Vuorio

Generative AI | Business Oriented Product Architect | Digital Commerce 16 articles

+ Follow

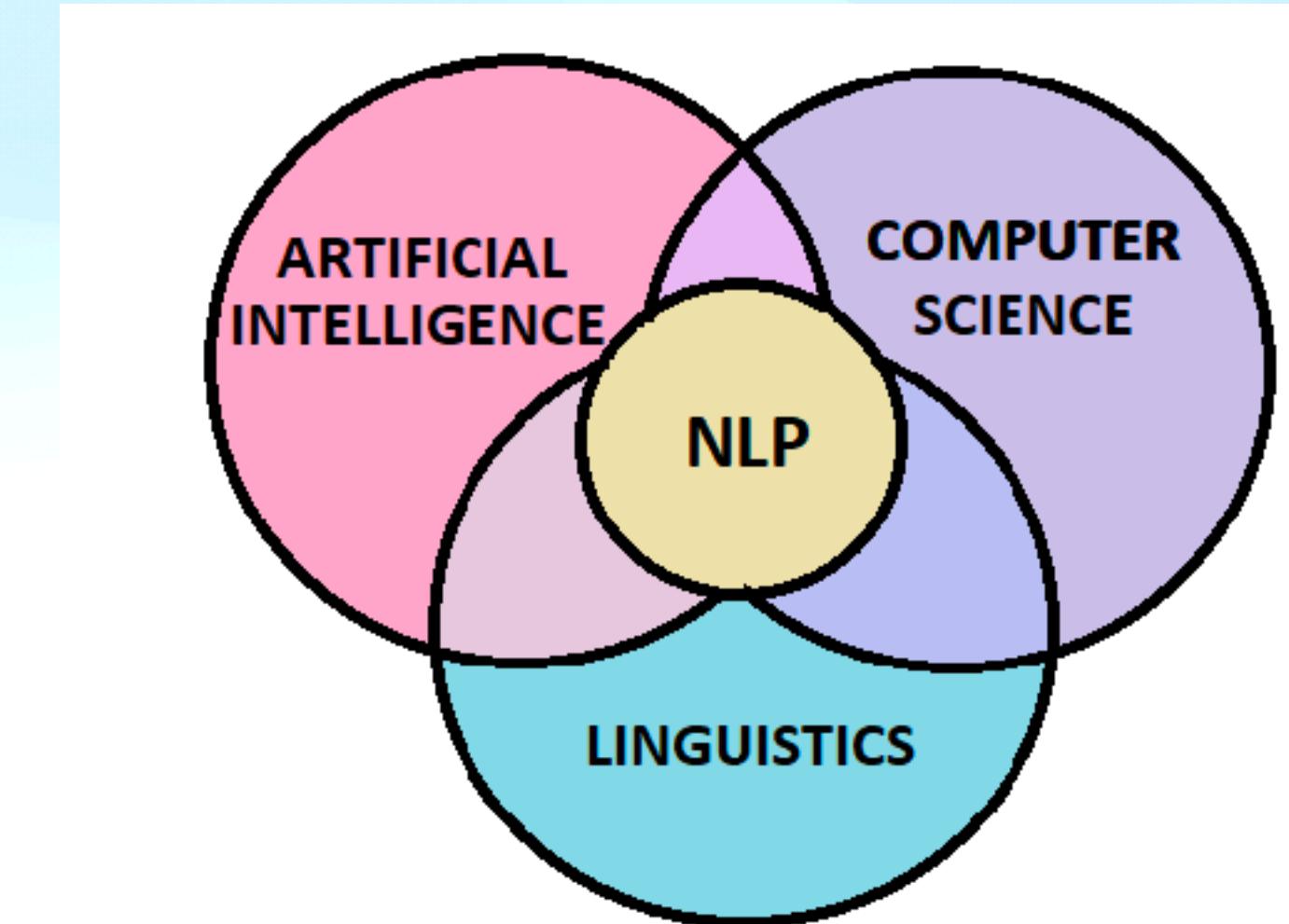
Expert | Tech Leadership | Developer At Heart

What is Natural Language Processing?

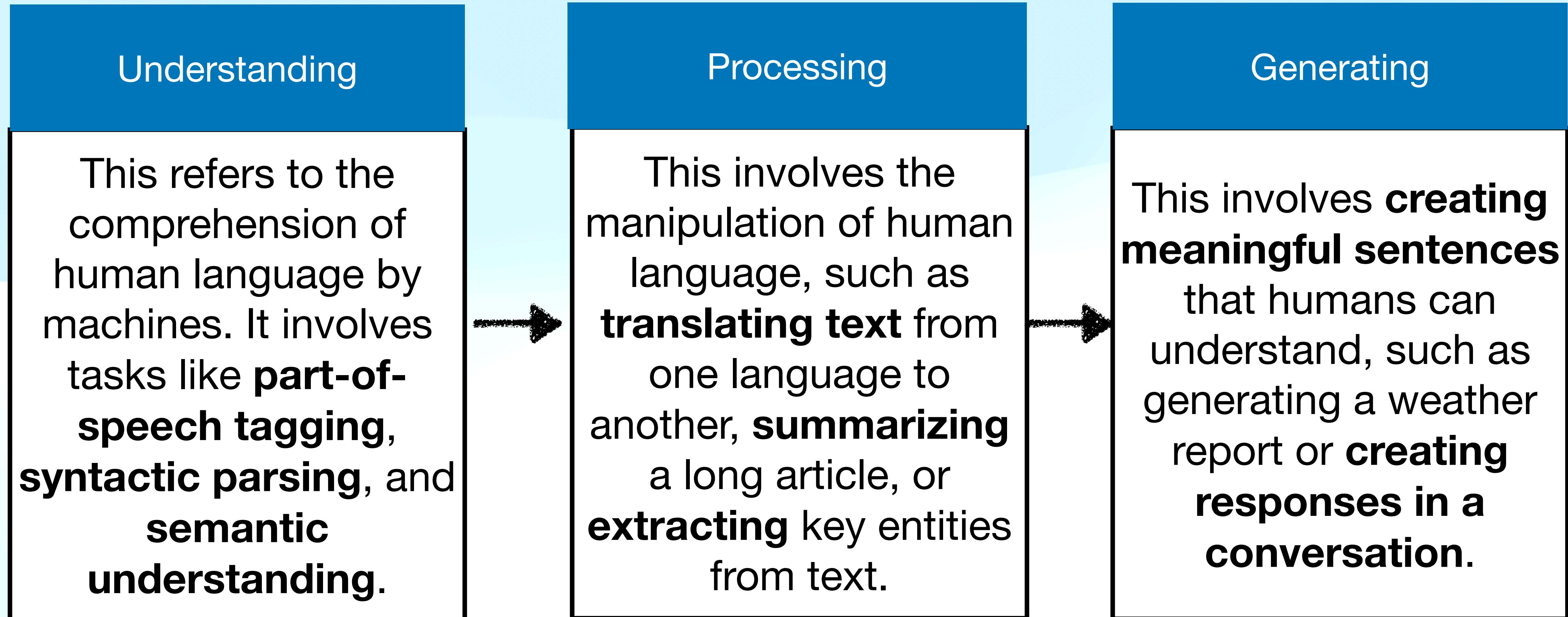
Definition: Natural Language

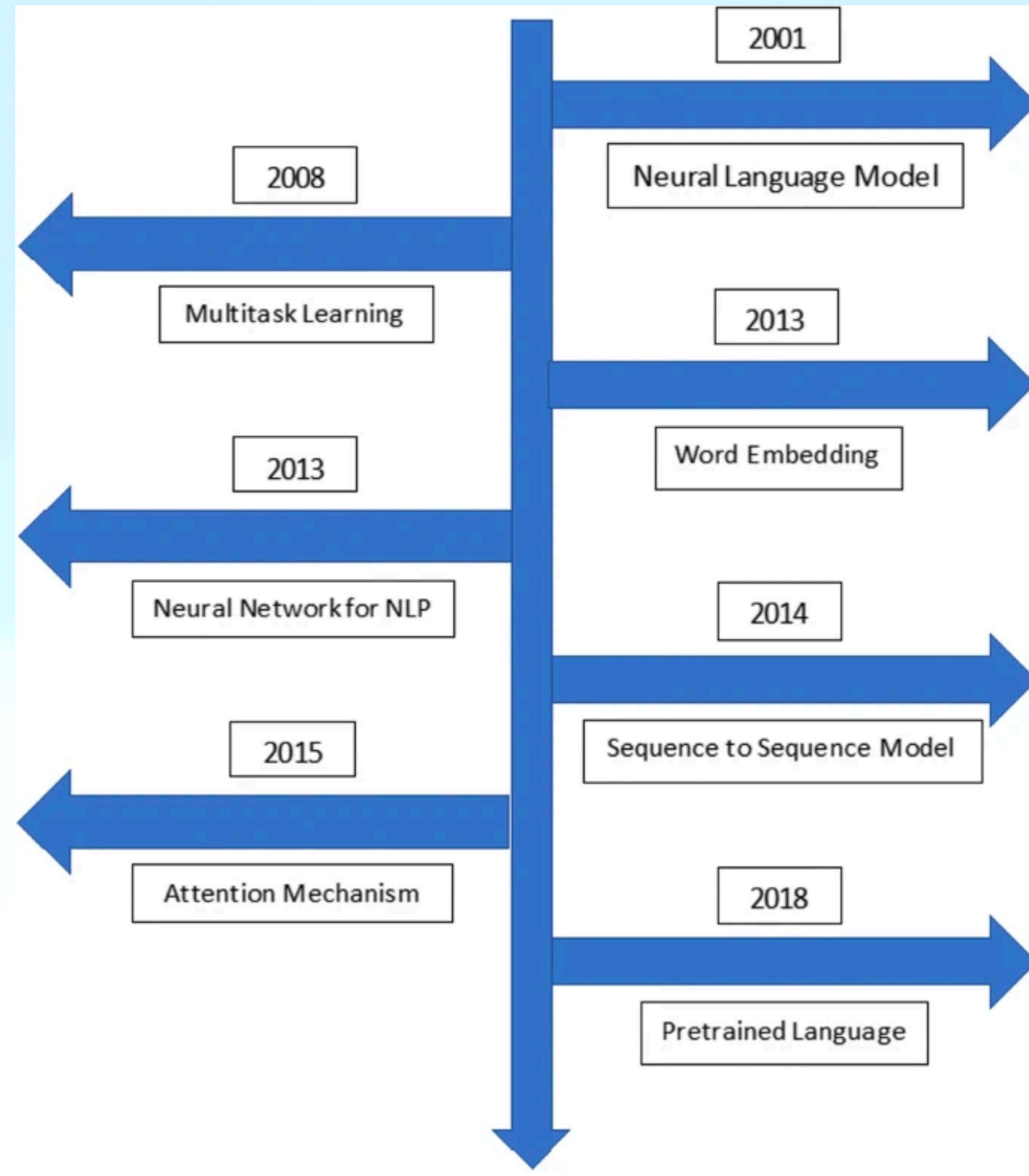
Processing, or NLP, is a field at the intersection of computer science, artificial intelligence, and linguistics.

The goal of NLP is to enable computers to understand, process, and generate human language in a valuable way.



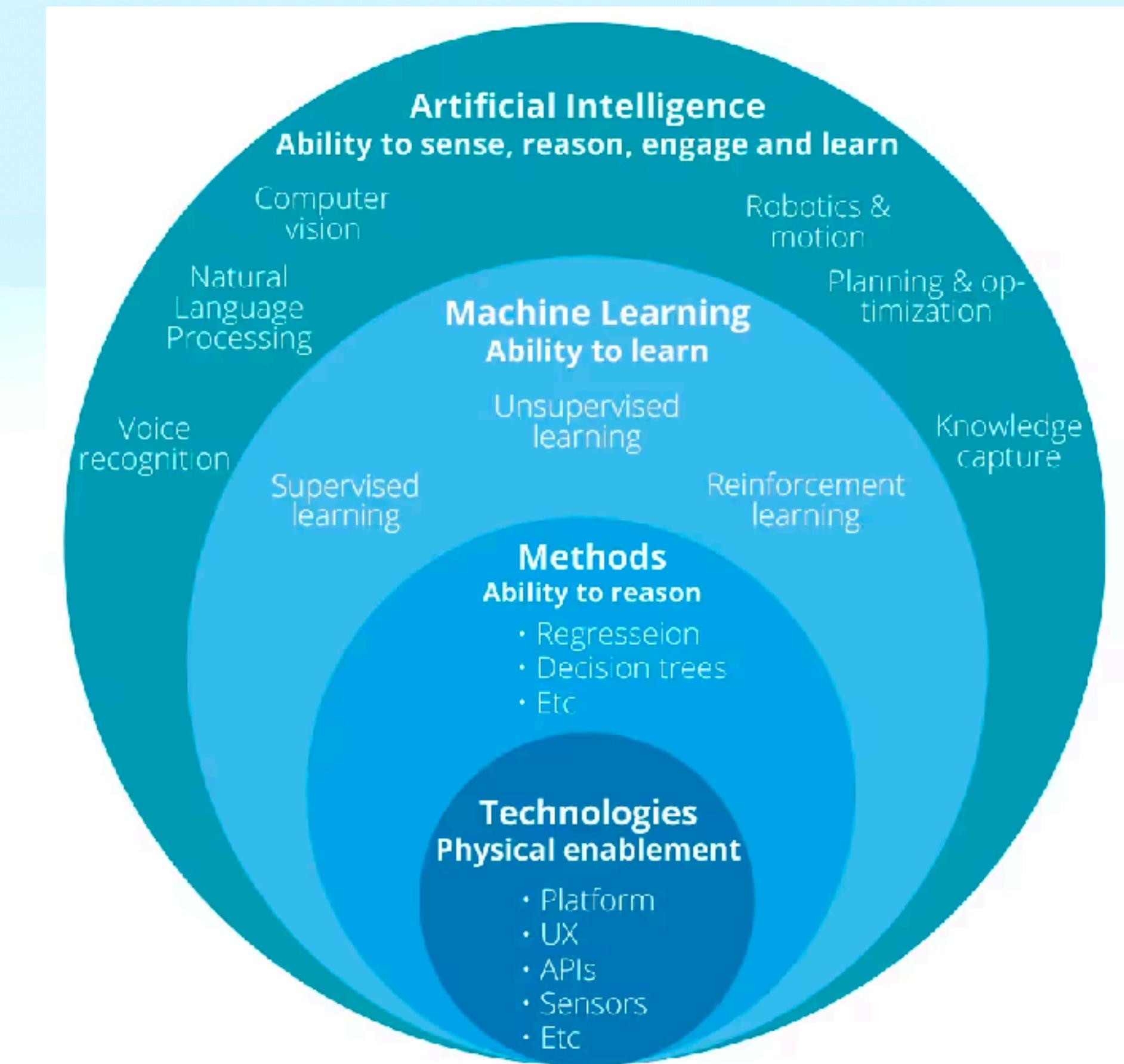
Breakdown of NLP

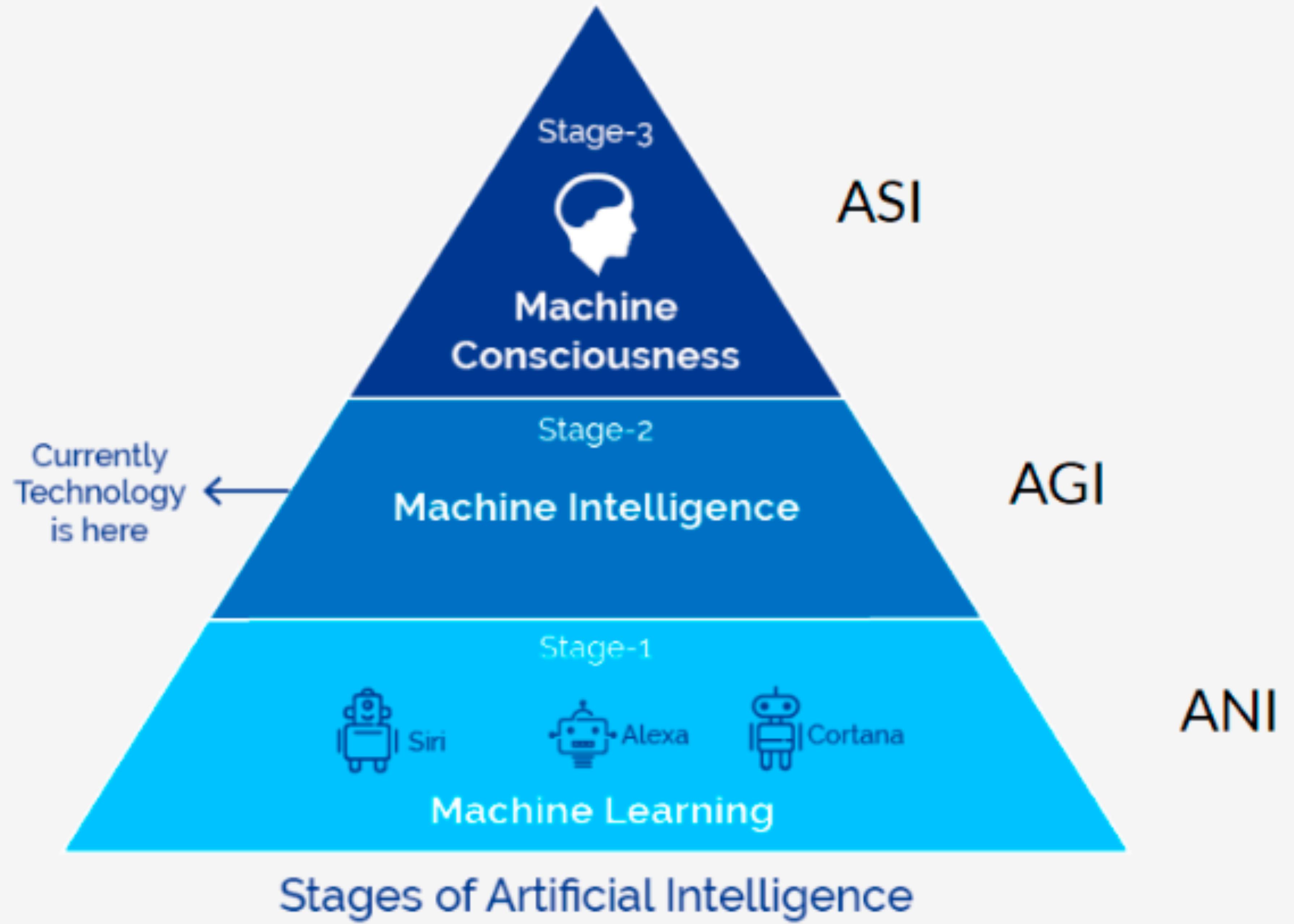




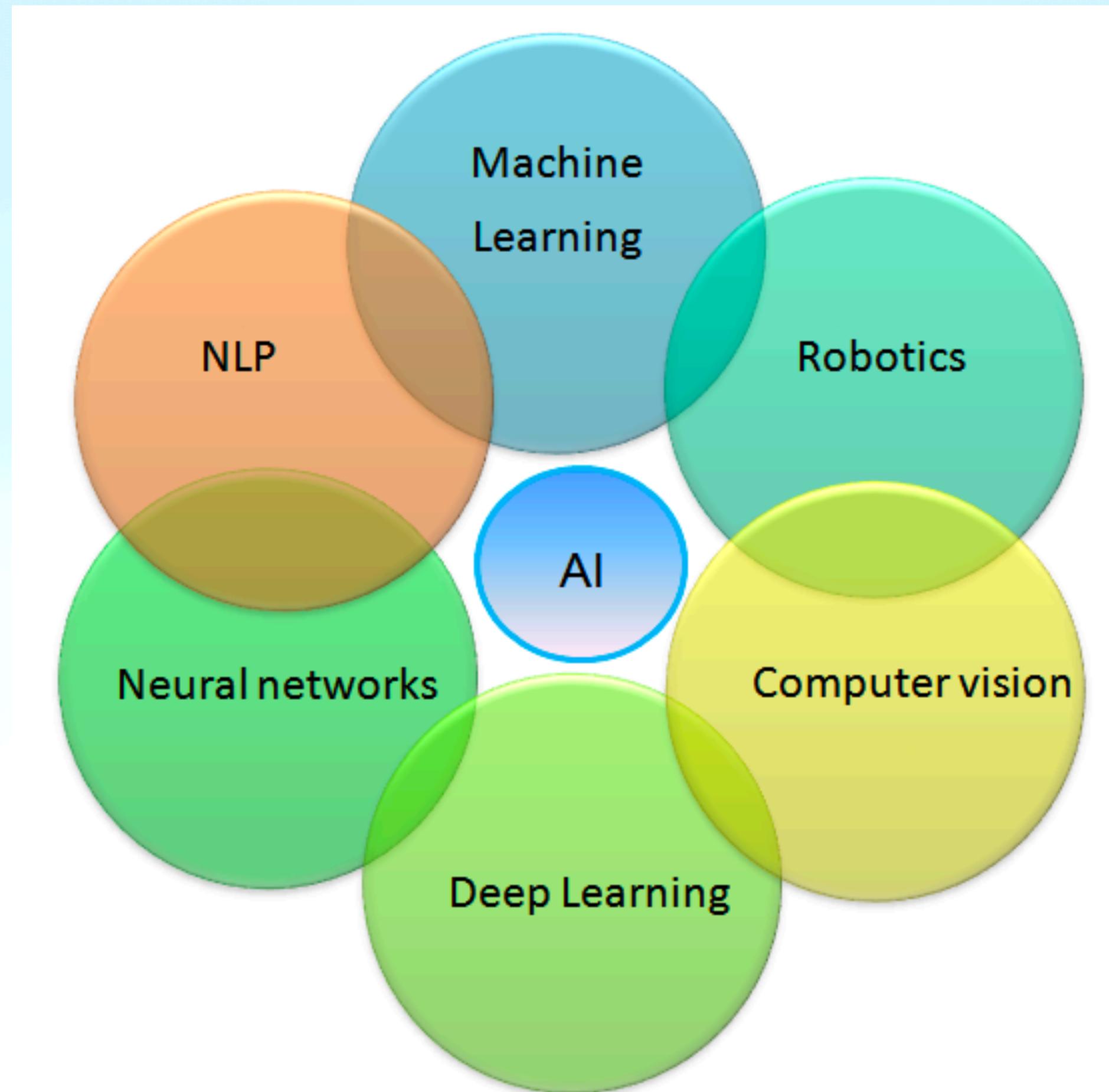
Natural Language Processing in the Field of Artificial Intelligence

- **AI and Its Subfields:**
 - **Definition of AI:** AI as the science of making computers do things that require intelligence when done by humans. It is a broad field aimed at simulating human intelligence processes by machines, especially computer systems.



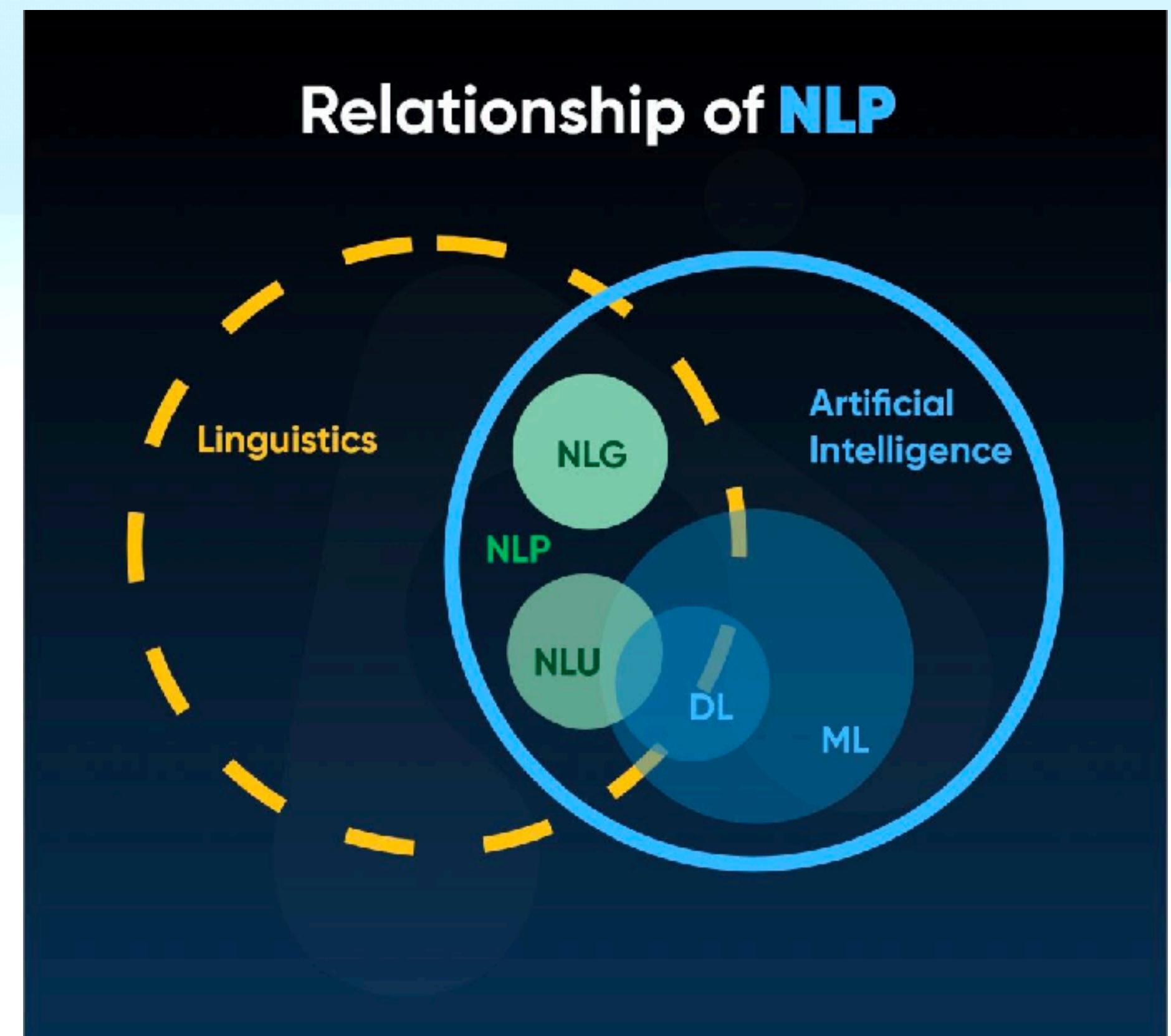


What is AI?



Importance of Natural Language Processing in Artificial Intelligence

- NLP is an integral part of AI. As AI is about **simulating human intelligence, and language is a core part of human intelligence**, understanding and processing language is essential for AI.
- NLP acts as a **bridge between humans and computers**, allowing humans to communicate with machines in their natural language.
- **NLP enables AI** to not just understand human language in a literal sense, but to **comprehend the nuances, context, and subtleties** that come with natural language. This includes things like idioms, metaphors, and cultural references.
- NLP also allows **AI to generate human language in a way that is both meaningful and sounds natural**. This is critical for applications like chatbots or virtual assistants.



Interplay Between Natural Language Processing and Machine Learning

- **Machine Learning** is a subset of AI that uses statistical methods to enable machines to improve with experience. ML is about creating and using algorithms that can learn from and make decisions or predictions based on data.
- NLP tasks are not hardcoded with rules but instead **rely on ML models** that learn from data. This is crucial for handling the complexity and variability of human language.
- **Examples of NLP Tasks Powered by ML:**
 - **Sentiment Analysis:** ML algorithms can be trained on large amounts of data, such as product reviews or social media posts, to learn the linguistic patterns associated with positive, negative, and neutral sentiments. The trained model can then analyze new texts and predict their sentiment.
 - **Language Translation:** ML algorithms, especially deep learning models, have revolutionized machine translation by learning to convert text from one language to another based on large bilingual datasets.
 - **Text Summarization:** ML can be used to automatically generate summaries of long texts, like news articles or scientific papers. Mention both extractive summarization (where key sentences are pulled from the original text) and abstractive summarization (where the model generates new sentences).

Key Tasks in Natural Language Processing

- Natural Language Processing **tasks** are specific computational procedures or methods designed to **handle** and **manipulate human language**. These tasks attempt to capture different levels of linguistic information - from basic linguistic units like words (tokens) to the meanings of sentences and the sentiments expressed in a piece of text.
- Each task is generally defined by its inputs and outputs, as well as the methods used to transform one into the other.

NLP Tasks

1. **Tokenization:** This is the process of splitting text into individual words or tokens. This is often the first step in many NLP pipelines.
2. **Part-of-Speech (POS) Tagging:** This involves marking up the words in a text as corresponding to a particular part of speech, based on both its definition and its context. POS tags are useful in disambiguating the semantic meaning of words.
3. **Named Entity Recognition (NER):** This is the task of identifying and classifying named entities (e.g., people, organizations, locations, dates) in text. NER is often used in information extraction and question answering systems to identify the key components of a text.
4. **Sentiment Analysis:** Also known as opinion mining, this involves determining the sentiment expressed in a piece of text. Sentiments are often classified as positive, negative, or neutral. This task is widely used in customer service and brand monitoring.

NLP Tasks

5. **Text Summarization:** This task involves generating a shorter version of a text that preserves its key information content and overall meaning. There are two main types of summarization: extractive (where key sentences are pulled from the original text) and abstractive (where the system generates new sentences).
6. **Machine Translation:** This involves automatically translating text from one human language to another. This task requires understanding the syntax, semantics, and sometimes even the cultural context of the source language.
7. **Topic Modeling:** This is a type of statistical model for discovering the abstract "topics" that occur in a collection of documents. It's often used in text mining and information retrieval.
8. **Semantic Role Labeling:** This task involves analyzing a sentence to identify its semantic arguments and label them with their roles in the sentence.

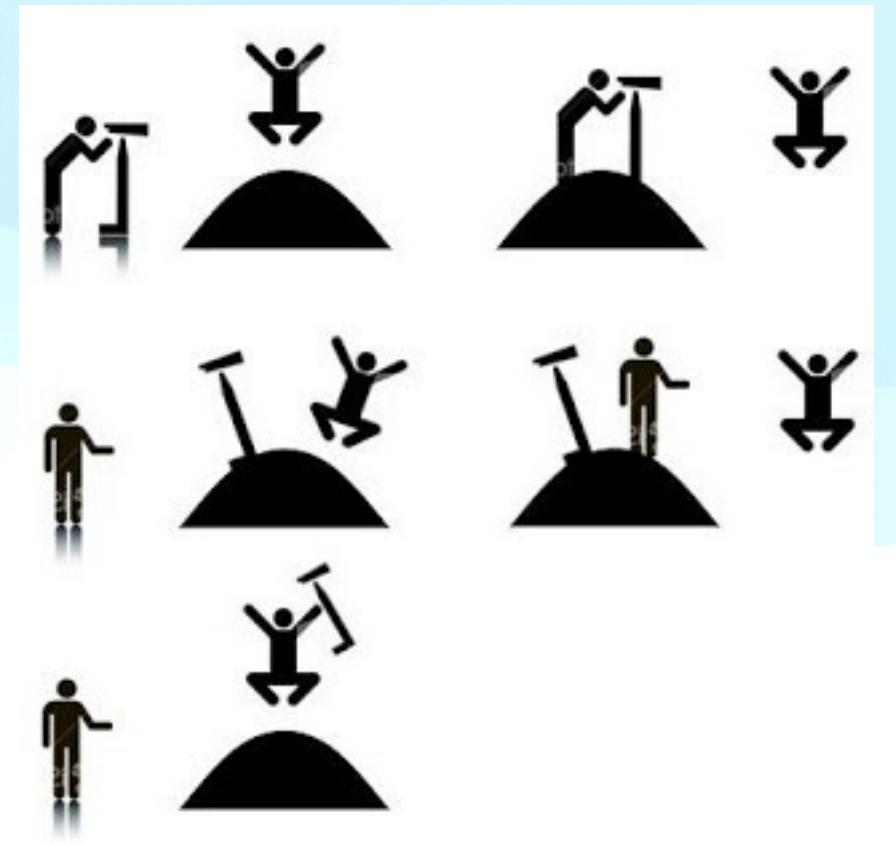
NLP Tasks

9. **Coreference Resolution:** This task involves finding all expressions that refer to the same entity in a text. It's crucial for understanding the context and reducing ambiguity in NLP tasks.
10. **Word Sense Disambiguation:** This task involves assigning the appropriate meaning to a word based on its context. A word may have multiple meanings, so this task helps clarify the correct sense of a word in a given context.
11. **Question Answering:** This involves building a system that can automatically answer questions posed by humans in a natural language. It requires understanding the question, finding the relevant information, and presenting it in a human-readable form.
12. **Conversational AI / Dialog Systems / Chatbots:** These systems are designed to converse with humans in a natural, human-like way. They are used in a wide range of applications, from customer service bots to personal virtual assistants.

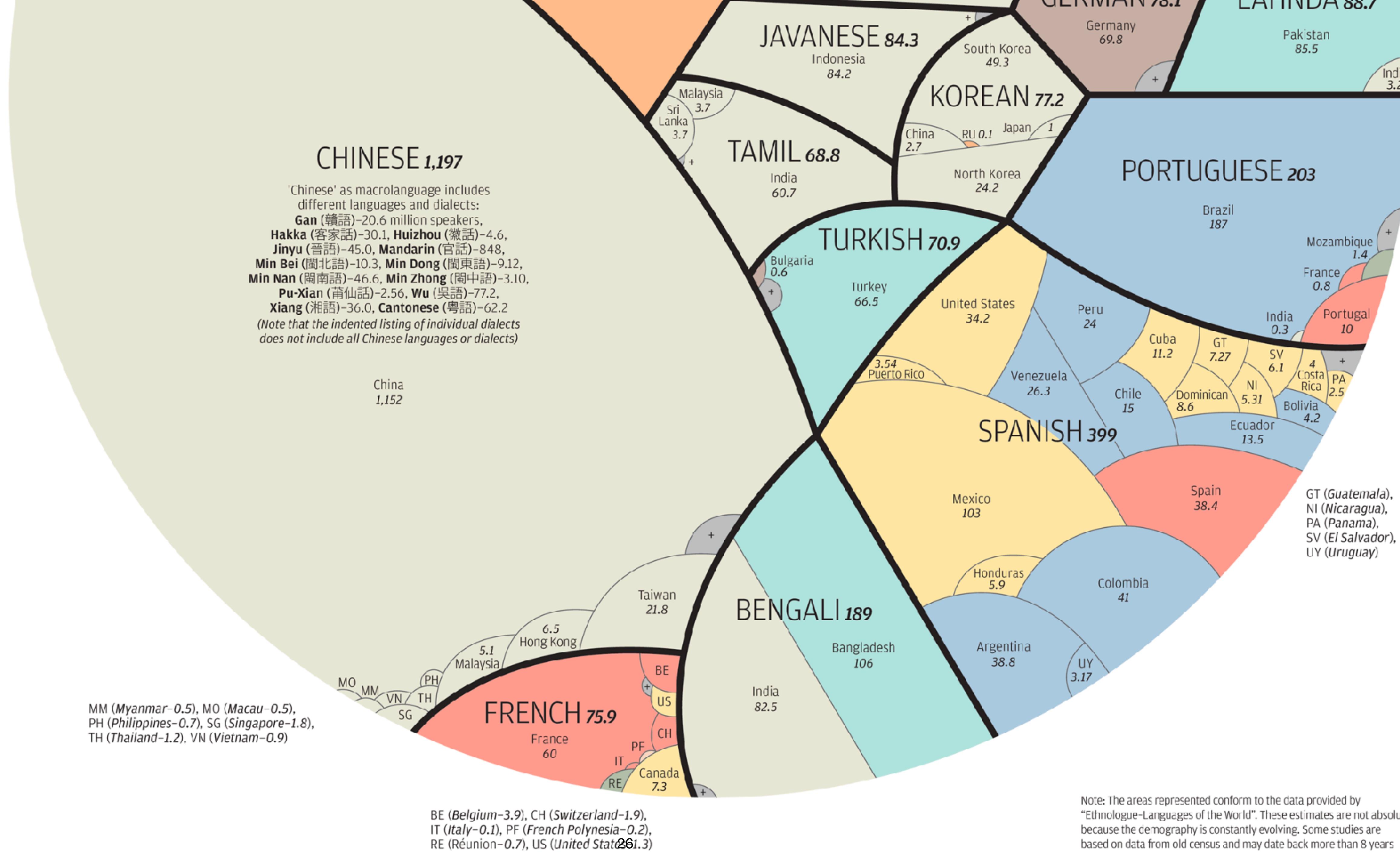
Challenges in NLP

Ambiguity & Diversity

1. **Linguistic Ambiguity:** This refers to the inherent ambiguity present in human language. Words can have **multiple meanings (polysemy)**, and sentences can often be interpreted in multiple ways depending on context (**syntactic or semantic ambiguity**). This makes understanding and interpretation challenging for computational models.



I saw the man with the telescope



Challenges in NLP

Context & Sarcasm, Lack of Structured Data

3. **Context and Sarcasm:** Understanding the context in which a statement is made or detecting sarcasm is difficult for machines. **Sarcasm** and irony often require high-level knowledge of language and the world, which is difficult for a machine to grasp.
4. **Lack of Structured Data:** While there is a lot of text data available, much of it is unstructured, which makes it difficult for machines to process and understand. Creating structured datasets for training NLP models is a time-consuming and challenging task.

Irony and Sarcasm

examQA.com

Sarcasm Example:

The food service of the restaurant was just brilliant. The plate was cold and so was the food.

Not to mention that the food was extremely bland and tasteless.

Module 1 : Python Refresher

Python



Python Syntax: Simplicity and Readability

Indentation: In Python, white spaces (indentation) at the start of a line are important. This is used to determine the grouping of statements or the block of code. Python uses indentation to define scope in the code.

Statement Termination: The end of a line marks the end of a statement. There's no need for a semicolon or other character to signify the end of the line.

Comments: Python interpreter ignores any text after # on a line. This can be used for explaining your code or preventing a piece of code from executing.

```
x = 10
if x > 5:
    print("x is greater than 5") # This line is indented and
else:
    print("x is not greater than 5") # This line is part of the if block
```

```
x = 5
y = 10
```

```
# This is a comment
x = 5 # This is an inline comment
```

Python Data Types: Numerics and Strings

Numerics: Python has two basic number types - integers and floats. Integers are whole numbers and can be both positive and negative. Floats represent real numbers and are written with a decimal point dividing the integer and fractional parts.

Strings: Strings in Python are sequences of character data. Textual data in Python is handled with str objects, or strings.

Variables: You can assign any data type to a variable. Variables are like containers for storing data.

Operators: You can perform various operations with these data types using operators. For example, numeric operators like +, -, *, /, %, and string operators like + (concatenation), * (repetition), and [] (slice).

```
# Define an integer
x = 10
print(x)

# Define a float
y = 5.5
print(y)

# Perform operations
print(x + y)
print(x * y)
print(x / y)

# Define a string
s = "Hello, World!"

# String concatenation
s = s + " How are you?"
print(s)

# String repetition
print(s * 2)

# String slice
print(s[0:5])
```

Python Data Types: Lists and Dictionaries

Lists: Python lists are ordered collections of items (strings, integers, or even other lists). Lists are mutable, meaning you can change their content.

Dictionaries: Python dictionaries are unordered collections of items. Each item of a dictionary has a key/value pair. Dictionaries are optimized for retrieving values when the key is known.

Manipulation: Discuss the various operations that can be done on lists and dictionaries like adding, removing items, retrieving items, etc.

```
# Define a list
my_list = [1, 2, 3, "Hello", 5.5]
print(my_list)

# Access an item
print(my_list[3])

# Change an item
my_list[3] = "World"
print(my_list)

# Add an item
my_list.append("How are you?")
print(my_list)

# Define a dictionary
my_dict = {"name": "John", "age": 25, "city": "New York"}
print(my_dict)

# Access an item
print(my_dict["name"])

# Change an item
my_dict["name"] = "Jane"
print(my_dict)

# Add an item
my_dict["profession"] = "Engineer"
print(my_dict)
```

Python Control Flow: Conditionals

Python uses `if`, `elif` (else if), and `else` statements to control the flow of execution based on certain conditions.

These conditional statements are used to perform different computations or actions depending on whether a condition evaluates to true or false.

```
x = 10

if x < 0:
    print("Negative number")
elif x == 0:
    print("Zero")
else:
    print("Positive number")
```

Python Control Flow: Loops

Python uses for and while loops to repeat a block of code multiple times.

A for loop is used for iterating over a sequence (list, tuple, dictionary, string, or a range of numbers).

A while loop executes as long as a certain condition is true.

```
# Define a list
numbers = [1, 2, 3, 4, 5]

# Use a for loop to iterate over the list
for number in numbers:
    print(number)
```

```
# Initialize a variable
x = 0

# Use a while loop
while x < 5:
    print(x)

    x += 1 # same as x = x + 1
```

Python Functions

- Functions in Python are blocks of reusable code that perform a specific task.
- You can define your own functions using the ***def*** keyword.
- Functions can have parameters that allow them to take inputs, and they can return a value using the ***return*** statement.

```
# Define a function
def greet(name):
    return "Hello, " + name + "!"

# Call the function
greeting = greet("Alice")
print(greeting)
```

Python and Text: String Manipulation

- Python provides a set of operations and methods for manipulating strings.
- This includes concatenation (joining strings), slicing (getting a part of the string), splitting a string into a list of substrings, etc.
- String manipulation is especially important in NLP tasks for preprocessing and cleaning text data.

```
s1 = "Hello, "
s2 = "world!"

# Concatenation
greeting = s1 + s2
print(greeting) # Output: Hello, world!

# Repetition
echo = s1 * 3
print(echo) # Output: Hello, Hello, Hello,
```

```
s = "Natural Language Processing"

# Slicing
print(s[8:16]) # Output: Language

# Splitting
words = s.split()
print(words) # Output: ['Natural', 'Language', 'Processing']
```

Python and Text: Regular Expressions

- Regular Expressions (RegEx) are sequences of characters that define a search pattern in text.
- This search pattern can be used to perform operations like pattern matching, substitution, and splitting.
- Python's re module provides functionalities to work with Regular Expressions.

```
import re

text = "The email addresses are: example1@test.com, example2@test.com"

# Find all email addresses in the text
emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)
print(emails) # Output: ['example1@test.com', 'example2@test.com']
```

Python Data Structures for Text: Lists and Dictionaries

- Lists and dictionaries are two fundamental Python data structures that are commonly used in text processing and NLP tasks.
- Lists in Python are used to store an ordered collection of items, which can be of any type. In the context of text processing, lists of strings are often used to represent a document or a set of tokens.
- Dictionaries in Python are used to store key-value pairs. They are incredibly useful for counting frequencies, such as word frequencies in a document.

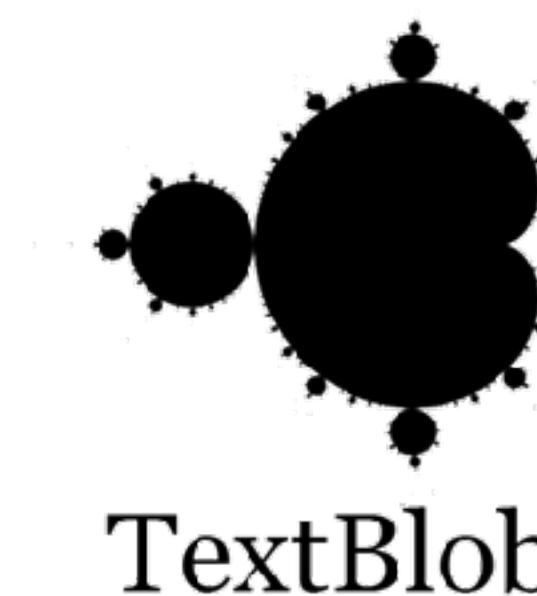
```
document = ["Natural", "Language", "Processing", "is", "exciting."]
print(document)
```

```
word_freq = {"Natural": 1, "Language": 1, "Processing": 1, "is": 1, "exciting": 1}
print(word_freq)
```

Module 2: Python Libraries for NLP

Introduction to Python Libraries for NLP

- Python, as a versatile programming language, offers an array of libraries specifically designed to tackle Natural Language Processing (NLP) tasks. These libraries are pre-built pieces of code containing specific functionality that developers can leverage while designing complex NLP systems.
- The three main libraries often employed in the field of NLP are the **Natural Language Toolkit (NLTK)**, **Spacy**, and **TextBlob**. Each library comes with its unique set of features and capabilities, making them more suited for certain types of tasks than others.

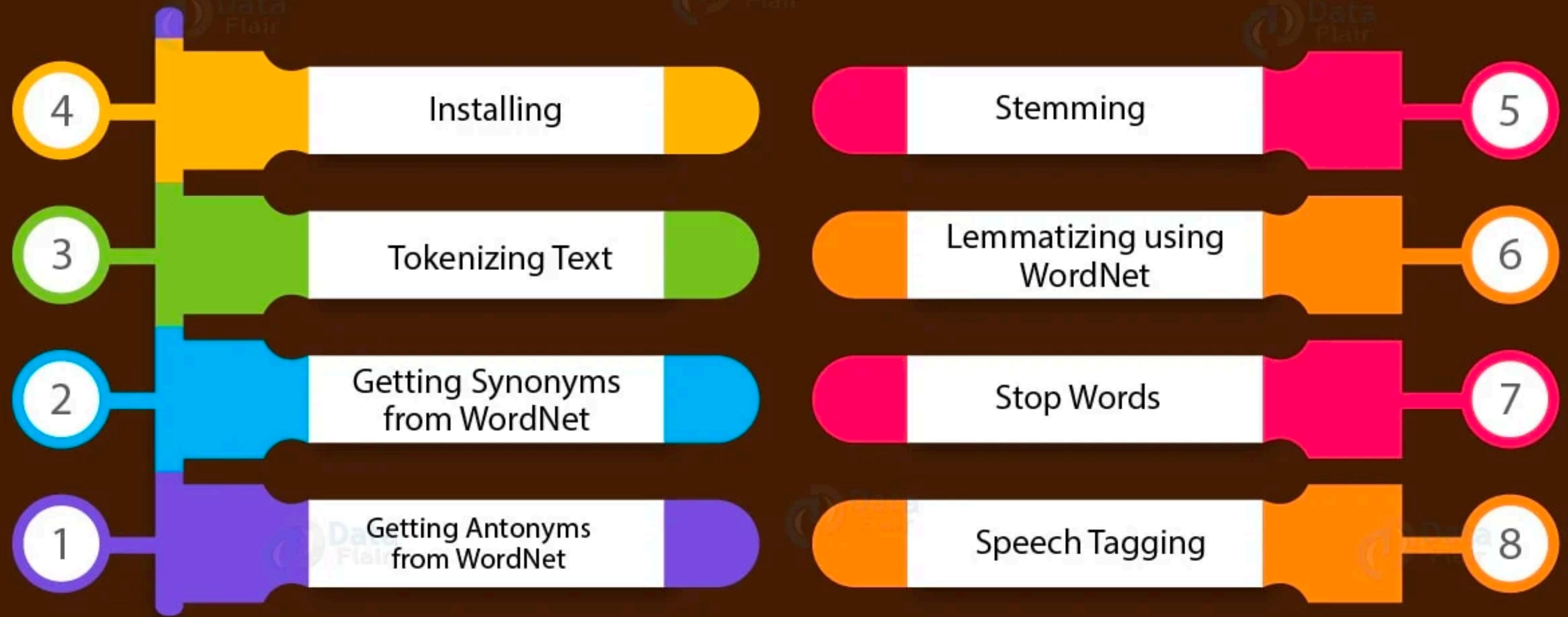


Natural Language Toolkit (NLTK)



- NLTK is one of the most widely-used libraries for academic and research purposes due to its exhaustive list of functionalities. It is like a treasure trove for linguists and has been instrumental in pioneering the field of NLP in Python. NLTK includes support for a wide range of tasks, from tokenization, parsing, and POS tagging, to machine learning, semantic reasoning, and more.
- **Strengths:** NLTK is an excellent tool for teaching and researching linguistics, symbolic and statistical natural language processing, including unsupervised machine learning algorithms. It comes with a host of resources like datasets, lexical databases, grammars, which can be leveraged for various tasks.
- **Weaknesses:** NLTK's functionality can sometimes be overwhelming and unnecessarily complex, especially for beginners. It's also not as fast or optimized as some other libraries like Spacy. Therefore, it's less commonly used in production environments where speed and efficiency are crucial.

NLTK- NLP with Python



Using NLTK for perform tokenization



```
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

sentence = "NLTK is a leading platform for building Python programs to work with human language data."

# Tokenization
tokens = nltk.word_tokenize(sentence)
print(f'Tokens: {tokens}')

# POS Tagging
pos_tags = nltk.pos_tag(tokens)
print(f'POS tags: {pos_tags}')
```

spaCy

spaCy

- Spacy is a free, open-source library for advanced Natural Language Processing in Python. It's specifically designed for **production use** and helps in building applications that can process and 'understand' large volumes of text. It provides **functionalities for almost all NLP tasks** and has in-built support for different languages.
- **Strengths:** Spacy is incredibly **fast and efficient**, designed with the industrial use-case in mind. It excels in large-scale information extraction tasks and has a unified and consistent API. It also **integrates seamlessly with deep learning libraries** like TensorFlow and PyTorch.
- **Weaknesses:** Spacy does not provide functionalities like stemming, but this is by design as it **opts for lemmatization**, considering it a more sophisticated operation. It may not be as beginner-friendly as libraries like NLTK or TextBlob.

spaCy's Sample Codes

The logo for spaCy, featuring the word "spaCy" in a large, blue, sans-serif font.

```
import spacy

nlp = spacy.load('en_core_web_sm')

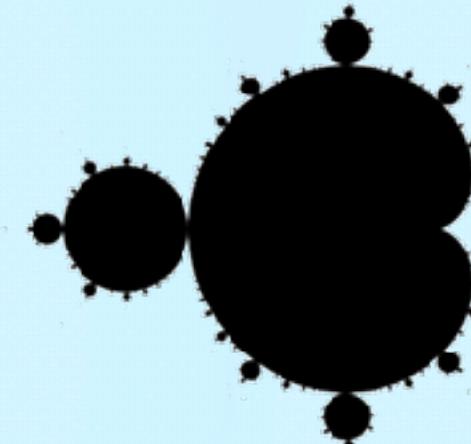
doc = nlp("Spacy is an open-source software library for advanced Natural Language Processing.")

# Tokenization
tokens = [token.text for token in doc]
print(f'Tokens: {tokens}')

# POS Tagging
pos_tags = [(token.text, token.pos_) for token in doc]
print(f'POS tags: {pos_tags}')

# Named Entity Recognition
entities = [(ent.text, ent.label_) for ent in doc.ents]
print(f'Entities: {entities}')
```

TextBlob



TextBlob

- TextBlob is a Python library for processing textual data. It provides a simple API for diving into common natural language processing (NLP) tasks such as **part-of-speech tagging, noun phrase extraction, sentiment analysis, classification, translation**, and more.
- **Strengths:** TextBlob is built on the foundations of **NLTK** and another package called **Pattern**, so it's incredibly user-friendly and intuitive while providing a broad range of capabilities. It's an excellent choice for beginners and for tasks that do not require highly optimized performance.
- **Weaknesses:** TextBlob may not be suitable for tasks that require the handling of large volumes of text or require more advanced functionalities, as it does not provide the same level of performance optimization as Spacy.

TextBlob's Sample Codes

spaCy

```
from textblob import TextBlob  
  
testimonial = TextBlob("TextBlob is a fantastic library for beginners in NLP. It's quite easy to use.")  
  
# Sentiment Analysis  
sentiment = testimonial.sentiment  
print(f'Sentiment: {sentiment}')
```

Workshop # 1 : Python practice with NLTK

- Explore NLTK and SpaCy

Module 3: Introduction to ThaiNLP

Unique Characteristics of Thai Language

- **Lack of Spaces:** Unlike English and many other languages, Thai does not use spaces between words. Instead, spaces are often used to separate sentences or clauses, making word segmentation a unique challenge in Thai NLP.
- **Multi-Tiered Script:** Thai is written with a multi-tiered script, where some characters can be written above or below others, and can change form depending on their position in the word. This poses difficulties for optical character recognition (OCR) and handwriting recognition.

"ชลน่าน" เพย์ผลหารีอ พปชร. ยังไม่หนุนพรครกแก้ม.112 ยังไม่ชวนร่วมรบ . ย้ำไม่คุยกปชป.เหตุยังไม่มี กก.บห. หวังได้เสียง ส.ว.หนุนโหวตนายกฯ บอกมวลชนต้องอยู่ในกรอบกม. บอกป่วนเจอแบงกันหมดไม่อยากเอาผิด

เป็นมนุษย์สุดประเสริฐเลิศคุณค่า กว่าบรรดาฝูงสัตว์เดรัจฉาน จะฝ่าฟันพัฒนาวิชาการ อย่างล้างเผาอย่างที่เป็นภาษาไทย ไม่ถือโทห์กรและเข้าใจดีดีดี ทั้งความยั่งยืนกีฬาอีซูซูกิ ปฏิบัติประพฤติกฎกำหนดใจ พูดจาให้ดีๆ จ้าๆ น่าฟังอย่าง

Unique Characteristics of Thai Language

- **Complex Spelling Rules:** Thai has complex spelling rules with many exceptions, and multiple possible spellings for the same sound, especially for words borrowed from other languages. This complicates tasks like text-to-speech and speech-to-text.

- **Variations in Colloquial Language:** There can be significant variations between formal written Thai and colloquial spoken Thai, including differences in vocabulary, grammar, and usage. This makes it necessary to consider the context and register (formal vs. colloquial) in NLP tasks.

ໄກລ້

initial consonant ກ (g), a vowel ໄ (i),
and a final consonant ລ (l),

the word "to eat" in
formal Thai is "ການ" (tan),
while in colloquial,
everyday conversation,
the word "ກິນ" (kin) might
be used instead

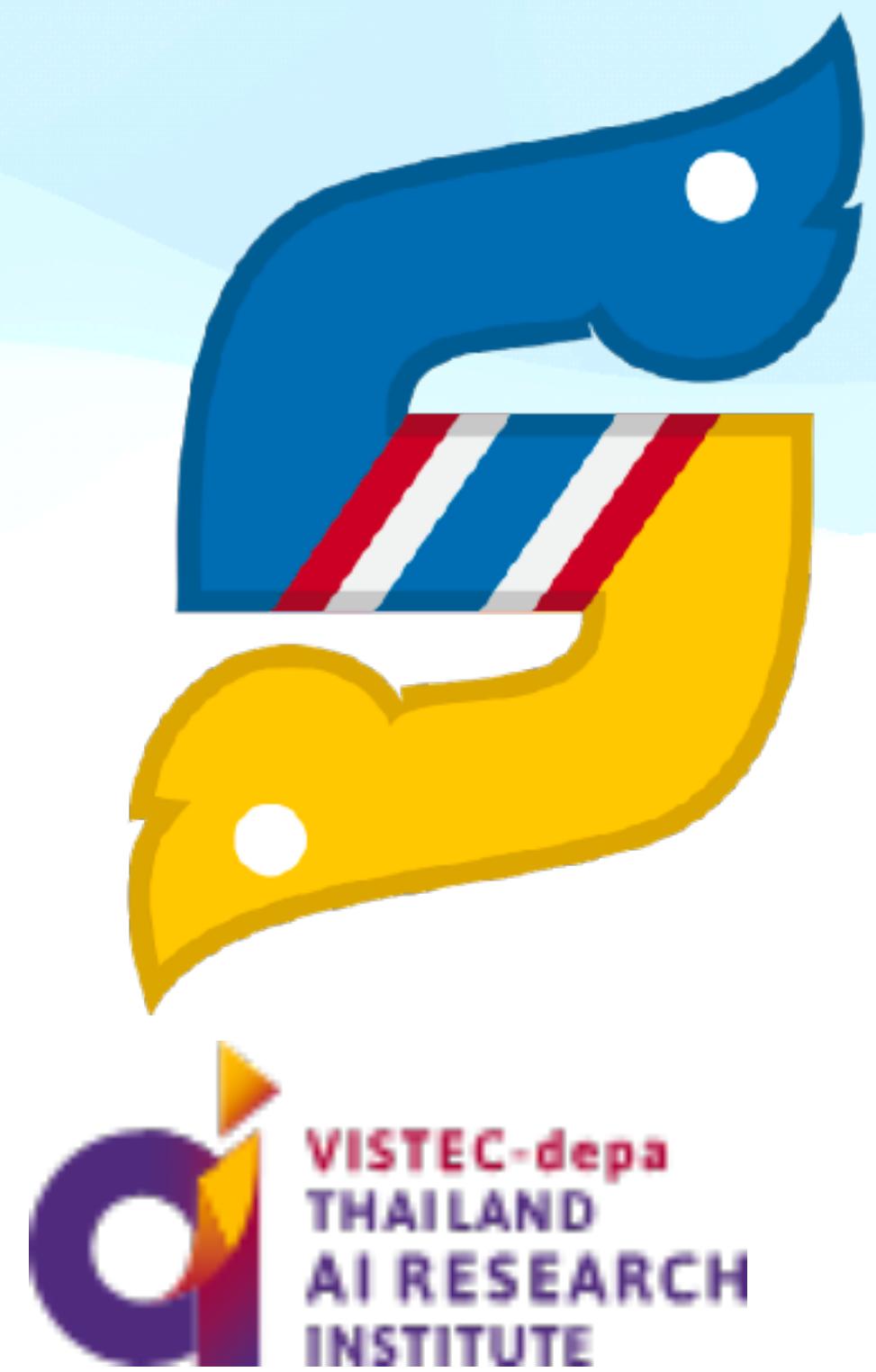
Challenges of Thai NLP

- **Word Segmentation:** Unlike languages such as English where words are separated by spaces, Thai words are not. This makes tokenization a fundamental challenge in Thai NLP. For example, the phrase "ผมรักคุณ" would be tokenized into three words: "ผม" (I), "รัก" (love), "คุณ" (you).
- **Text Normalization:** Thai language includes a wide variety of writing styles and tones (formal, informal, and internet slang), different spellings for the same words, and no standard rules for things like date and time formats. For instance, the word "ເຊື່ອ" (you) might be written in internet slang as "ເຊື່ອຣ໌".
- **Named Entity Recognition (NER):** Identifying and classifying named entities (such as persons, organizations, locations) in Thai text can be challenging due to the lack of capitalization and the fact that names often blend seamlessly into surrounding text. For instance, in the sentence "ຈັນໄປເຖິງວິທີປະເມີນບູນນຸ້ມີ" (I went to Pran Buri), "ປະເມີນບູນນຸ້ມີ" is a named entity (a location).



Introduction to PyThaiNLP

- PyThaiNLP is an open-source natural language processing (NLP) library specifically designed for Thai language. Its goal is to make NLP for the Thai language easily accessible and useful.
- **Addressing Challenges:** PyThaiNLP addresses unique Thai NLP challenges such as word segmentation, text normalization, and named entity recognition, making it a go-to resource for anyone working on Thai NLP.



PyThaiNLP: Features Overview

- **Word Tokenization:** PyThaiNLP provides word tokenization, which is particularly important in Thai language processing as there are no spaces between words.

```
from pythainlp import word_tokenize
text = "สวัสดีครับ"
tokens = word_tokenize(text)
print(tokens)
```

```
['สวัสดี', 'ครับ']
```

PyThaiNLP: Features Overview

- **Part-of-speech Tagging:** PyThaiNLP can assign grammatical information (part-of-speech) to individual words in a sentence.

```
from pythainlp.tag import pos_tag
text = "เขากำลังวิ่งอยู่ที่สนามหญ้า"
pos_tags = pos_tag(word_tokenize(text))
print(pos_tags)
```

```
[('เข้า', 'PPRS'), ('กำลัง', 'XVBM'), ('วิ่ง', 'VACT'), ('อยู่', 'XVAE'), ('ที่', 'RPRE'), ('สนามหญ้า', 'NCMN')]
```

PyThaiNLP: Features Overview

- **Transliteration:** PyThaiNLP provides transliteration from Thai script to Roman script.

```
from pythainlp.transliterate import transliterate  
  
transliterate("แมว")
```

```
Corpus: thai-g2p  
- Downloading: thai-g2p 0.1  
Done.  
'm ε: w +'
```

```
from pythainlp.transliterate import romanize  
  
romanize("แมว")
```

```
'maeo'
```

PyThaiNLP: Features Overview

- **Spell Checking:** PyThaiNLP offers spell checking capabilities, which is useful for processing informal or user-generated text.

```
from pythainlp import spell  
  
spell("เหลี่ยม")  
  
['เหลี่ยม', 'เหลือມ']  
  
from pythainlp import correct  
  
correct("เหลี่ยม")  
  
'เหลี่ยม'
```

PyThaiNLP: Features Overview

- **Text Normalization:** PyThaiNLP helps in text normalization, removes zero-width spaces (ZWSP and ZWNJ), duplicated spaces, repeating vowels, and dangling characters. It also reorder vowels and tone marks during the process of removing repeating vowels.

```
from pythainlp.util import normalize  
  
normalize("ເປັນ") == "ປັນ"
```

True

```
normalize("ເກະວະ")
```

'ກະ'

PyThaiNLP: Word Tokenization

- **Word tokenization** is a foundational step in many NLP tasks, and it's particularly crucial in Thai language processing because Thai sentences don't use spaces to separate words. This means that we can't use a simple space-based method to divide the text into individual words. Instead, we need a library like PyThaiNLP, which understands the Thai language and can accurately segment the text.
- PyThaiNLP uses a dictionary-based method and a maximum matching algorithm for word tokenization. Let's look at an example.

PyThaiNLP: Word Tokenization

```
from pythainlp import word_tokenize  
  
text = "สวัสดีครับผมชื่อเจมส์"  
tokens = word_tokenize(text)  
print(tokens)  
  
['สวัสดี', 'ครับผม', 'ชื่อ', 'เจมส์']
```

PyThaiNLP: Part-of-Speech Tagging

- **Part-of-speech (POS) tagging** is the task of labeling the words in a sentence with their appropriate part of speech, such as noun, verb, adjective, etc. POS tagging is crucial for syntactic and semantic analysis in NLP.
- PyThaiNLP provides the `pos_tag` function for this purpose. It employs the ORCHID Thai part-of-speech tagset, a standard POS tagset in Thai language processing.

PyThaiNLP: Part-of-Speech Tagging

```
from pythainlp import word_tokenize, pos_tag

text = "สวัสดีครับผมชื่อเจมส์"
tokens = word_tokenize(text)
tagged_tokens = pos_tag(tokens)
print(tagged_tokens)

[('สวัสดี', 'NCMN'), ('ครับ', 'NCMN'), ('ผม', 'NCMN'), ('ชื่อ', 'NCMN'), ('เจมส์', 'NCMN')]
```

PyThaiNLP: Part-of-speech Tagging

Abbreviation	Part-of-Speech tag	Examples
NPRP	Proper noun	วันเดียว ๙๕, โตโน่, ใจดี
NCNM	Cardinal number	พี่๔, สี่, สาม, ๑, ๒, ๑๐
NONM	Ordinal number	ที่หนึ่ง, ที่สอง, ที่สาม, ที่๑, ที่๒
NLBL	Label noun	๑, ๒, ๓, ๔, ก, ๖, a, b
NCMN	Common noun	คน, ผู้, อาจารย์, ศาสตราจารย์, ศัลษะ
NTTL	Title noun	ครู, พลเอก
PPRS	Personal pronoun	ฉัน, เขายัง, ยัง
PDMN	Demonstrative pronoun	นี่, นั้น, ที่นั้น, ที่นี่
PNTR	Interrogative pronoun	ใคร, อะไร, อย่างไร
PREL	Relative pronoun	ที่, ซึ่ง, ซัน, ดู
VACT	Active verb	ทำงาน, วันเพลิง, บิน
VSTA	Stative verb	เป็น, ดู, หืด
VATT	Attributive verb	ช้าน, ตี, สวย
XVBM	Pre-verb auxiliary, before negator "ไม"	เกิด, เกิดมา, ที่เกิด
XVAM	Pre-verb auxiliary, after negator "ไม"	คือ, ค่า, ได้
XVMM	Pre-verb, before or after negator "ไม"	ควร, เดชะ, ต้อง
XVBB	Pre-verb auxiliary, in imperative mood	กรุณา, จะ, เชิญ, อ่า, ห้าม
XVAE	Post-verb auxiliary	ไป, มา, ขึ้น
DDAN	Definite determiner, after noun without classifier in between	นี่, นั้น, โน่น, หัวเหด
DDAC	Definite determiner, allowing classifier in between	นี่, นั้น, โน่น, หัว
DDBQ	Definite determiner, between noun and classifier or preceding quantitative expression	น้ำ, น้ำดื่ม, เพื่อ
DDAQ	Definite determiner, following quantitative expression	พอยต์, ร้าน

DIBQ	Indefinite determiner, between noun and classifier or preceding quantitative expression	บาง, ประมาณ, เดือน
DIAQ	Indefinite determiner, following quantitative expression	ก้าว, เดียว
DCNM	Determiner, cardinal number expression	ห้ามคน, เดียว, ๒ ตัว
DONM	Determiner, ordinal number expression	ที่หนึ่ง, ที่สอง, ที่สุดท้าย
ADVN	Adverb with normal form	เบ่ง, เร็ว, ช้า, สนับสนุน
ADVI	Adverb with iterative form	เร็วๆ, เดชะๆ, ช้าๆ
ADVP	Adverb with prefixed form	โดยเร็ว
ADVS	Sentential adverb	โดยปกติ, ธรรมชาติ
CNIT	Unit classifier	ตัว, คน, เสื่อม
CLTV	Collective classifier	ฝูง, กลุ่ม, ผู้, จำนวน, แบบ, รุ่น
CMTR	Measurement classifier	กิโลกรัม, แก้ว, ช้อนใจ
CFQC	Frequency classifier	ครั้ง, เที่ยว
CVBL	Verbal classifier	ผ่าน, มัด
JCRG	Coordinating conjunction	และ, หรือ, แต่
JCMP	Comparative conjunction	ก้าว, เทียบกับ, เท่ากับ
JSBR	Subordinating conjunction	เพราจะ, เมื่อจะ, ที่, แม้ว
RPRE	Preposition	จาก, ด้วย, ของ, ให้, บน
INT	Interjection	โอ้, ใช่, เอ๊, ล้อ
FIXN	Nominal prefix	การท่องเที่ยว, ความสุขสนาน
FIXV	Adverbial prefix	อย่างเร็ว
EAFF	Ending for affirmative sentence	ซึ่ง, ซึ่ง, ค่ะ, ครับ, นะ, ย่า, แล้ว
EITT	Ending for interrogative sentence	หรือ, หรือ, ใหม่, ยัง
NEG	Negator	ไม, ถ้าไม่, ไม่ได้, ไม่ได้, ไม่
PUNC	Punctuation	(,), " ; :

PyThaiNLP: Transliteration

- **Transliteration** is the process of converting a word from one script to another. This is particularly useful when working with a script like Thai that uses a non-Latin alphabet.
- PyThaiNLP provides the transliterate function, which can convert Thai script to Roman letters. This can be very useful for people who are unfamiliar with Thai script but want to pronounce Thai words correctly.

PyThaiNLP: Transliteration

```
from pythainlp.transliterate import romanize

text = "สวัสดีครับผมชื่อเจมส์"
transliterated_text = romanize(text)
print(transliterated_text)

sattikhnappmchuechem
```

```
from pythainlp.transliterate import transliterate

transliteratetext = transliterate("สวัสดีครับผมชื่อเจมส์")
print(transliterated_text)

sattikhnappmchuechem
```

PyThaiNLP: Spell Checking and Text Normalization

- Spell checking and text normalization are critical tasks in NLP that involve the correction of spelling errors and the conversion of text into a standard form, respectively.
- PyThaiNLP provides the correct function for spell checking and the normalize function for text normalization.

PyThaiNLP: Spell Checking and Text Normalization

```
from pythainlp.spell import correct

text = "ເກົ່າ"
corrected_text = correct(text)
print(corrected_text)
```

ເກົ່າ

```
from pythainlp.util import normalize

text = "ເກົ່າ"
normalized_text = normalize(text)
print(normalized_text)
```

ເກົ່າ

PyThaiNLP: Named Entity Recognition

- Named Entity Recognition (NER) is a crucial NLP task that involves identifying and categorizing named entities in text such as names of people, organizations, locations, and dates.
- PyThaiNLP provides the `get_ner` function in its `ner` module for performing named entity recognition in Thai language.

PyThaiNLP: Named Entity Recognition

```
from pythainlp.tag.thainer import ThaiNameTagger

ner = ThaiNameTagger()

text = "บริษัท Apple Inc. สำเร็จอุปกรณ์จากไทยเมื่อวันจันทร์ที่ 7 กุมภาพันธ์ 2023"

entities = ner.get_ner(text)
for entity in entities:
    print(f"Entity: {entity[0]}, Type: {entity[1]}")

Corpus: thainer-1.4
- Downloading: thainer-1.4 1.4
Done.
Entity: บริษัท, Type: NOUN
Entity: , Type: PUNCT
Entity: Apple, Type: PROPN
Entity: , Type: PUNCT
Entity: Inc, Type: NOUN
Entity: ., Type: PUNCT
Entity: , Type: PUNCT
Entity: สำเร็จ, Type: VERB
Entity: อุปกรณ์, Type: NOUN
Entity: จาก, Type: ADP
Entity: ไทย, Type: PROPN
Entity: เมื่อ, Type: SCONJ
Entity: วัน, Type: NOUN
Entity: จันทร์, Type: PROPN
Entity: ที่, Type: SCONJ
Entity: , Type: PUNCT
Entity: 7, Type: NUM
Entity: , Type: PUNCT
Entity: กุมภาพันธ์, Type: PROPN
Entity: , Type: PUNCT
Entity: 2023, Type: NUM
```

Tag

- DATA - date
- TIME - time
- EMAIL - email
- LEN - length
- LOCATION - Location
- ORGANIZATION - Company / Organization
- PERSON - Person name
- PHONE - phone number
- TEMPERATURE - temperature
- URL - URL
- ZIP - Zip code
- MONEY - the amount
- LAW - legislation
- PERCENT - PERCENT

Workshop # 2 : Practice PyThaiNLP

- Practice Page 54-69

Module 4 : Textual Sources

Introduction to Textual Sources and Formats

This module will explore various sources of text data that we can use for Natural Language Processing (NLP) tasks.

Examples of text data sources include APIs, social media, and web scraping.

APIs as Textual Sources

- APIs (Application Programming Interfaces) allow us to access data from various platforms.
- Many APIs return data in JSON format, which is easy to parse in Python.
- Python's requests library can be used to make API calls.

APIs as Textual Sources

```
import requests
import json

headers = {"Authorization": "Bearer sk-gYxhjjf7csEltB4CA6coT3BlbkFJya7NmTHyXXXXXXXXXXXXXX"}

# Make a GET request to the OpenAI API
response = requests.get('https://api.openai.com/v1/models',
                        headers=headers)

# Parse the JSON response
data = response.json()

print(data)

{'object': 'list', 'data': [{}{'id': 'babbage', 'object': 'model', 'created': 1649358449, 'owned_by': 'openai', 'permission': [{}{'id': 'modelperm-49FUp5v084tBB49tC4z8LPH5', 'object': 'model_permission', 'created': 1669085501, 'allow_create_engine': False, 'allow_sampling': True, 'allow_logprobs': True, 'allow_search_indices': False, 'allow_view': True, 'allow_fine_tuning': False, 'organization': '*', 'group': None, 'is_blocking': False}], 'root': 'babbage', 'parent': None}, {'id': 'text-davinci-003', 'object': 'model', 'created': 1669599635, 'owned_by': 'openai-internal', 'permission': [{}{'id': 'modelperm-jepinXYt59ncUQrjQEIUEDyC', 'object': 'model_permission', 'created': 1688551385, 'allow_create_engine': False, 'allow_sampling': True, 'allow_logprobs': True, 'allow_search_indices': False, 'allow_view': True, 'allow_fine_tuning': False, 'organization': '*', 'group': None, 'is_blocking': False}], 'root': 'text-davinci-003', 'parent': None}, {'id': 'davinci', 'object': 'model', 'created': 1649359874, 'owned_by': 'openai', 'permission': [{}{'id': 'modelperm-U6ZwlyAd0LyMk4rcMdz33Yc3', 'object': 'model_permission', 'created': 1669066355, 'allow_create_engine': False, 'allow_sampling': True, 'allow_logprobs': True, 'allow_search_indices': False, 'allow_view': True, 'allow_fine_tuning': False, 'organization': '*', 'group': None, 'is_blocking': False}], 'root': 'davinci', 'parent': None}, {'id': 'text-davinci-edit-001', 'object': 'model', 'created': 1649809179, 'owned_by': 'openai', 'permission': [{}{'id': 'modelperm-otmQSS0hmabtVGH19QB3bct3', 'object': 'model_permission', 'created': 1679934178, 'allow_create_engine': False, 'allow_sampling': True, 'allow_logprobs': True, 'allow_search_indices': False, 'allow_view': True, 'allow_fine_tuning': False, 'organization': '*', 'group': None, 'is_blocking': False}], 'root': 'text-davinci-edit-001', 'parent': None}, {'id': 'babbage-code-search-code', 'object': 'model', 'created': 1651172509, 'owned_by': 'openai-dev', 'permission': [{}{'id': 'modelperm-4qRnA3Hj8HIJbgo0cGbcmErn', 'object': 'model_permission', 'created': 1669085863, 'allow_create_engine': False, 'allow_sampling': True, 'allow_logprobs': True, 'allow_search_indices': True, 'allow_view': True, 'allow_fine_tuning': False, 'organization': '*', 'group': None, 'is_blocking': False}], 'root': 'babbage-code-search-code', 'parent': None}, {'id': 'text-similarity-babbage-001', 'object': 'model', 'created': 1651172505, 'owned_by': 'openai-dev', 'permission': [{}{'id': 'modelperm-48kcCHhfzvnfY840tJf5m8Cz', 'object': 'model_permission', 'created': 1669081947, 'allow_create_engine': False, 'allow_sampling': True, 'allow_logprobs': True, 'allow_search_indices': True, 'allow_view': True, 'allow_fine_tuning': False, 'organization': '*', 'group': None, 'is_blocking': False}], 'root': 'text-similarity-babbage-001', 'parent': None}, {'id': 'code-davinci-edit-001', 'object': 'model', 'created': 1649880484, 'owned_by': 'openai', 'permission': [{}{'id': 'modelperm-FnaEVATvsku8Vvt7AnKMu2TRt', 'object': 'model_permission', 'created': 1670024179, 'allow_create_en'}]}]
```

Social Media as Textual Sources

- Social media platforms like Twitter and Facebook are rich sources of text data.
- We can use libraries like Tweepy for Twitter or Facebook SDK for Facebook to fetch data.

Social Media as Textual Sources

```
import praw

# Setup PRAW with your Reddit API keys
reddit = praw.Reddit(
    client_id="my_client_id",
    client_secret="my_client_secret",
    user_agent="my_user_agent",
)

# Get the top 5 hot posts from the Python subreddit
hot_posts = reddit.subreddit('การเมืองไทย').hot(limit=5)
for post in hot_posts:
    print(post.title)
```



Web Scraping as a Textual Source

- Web scraping allows us to extract data from websites.
- Python libraries like BeautifulSoup and Scrapy are often used for this.

Web Scraping as a Textual Source

```
from bs4 import BeautifulSoup
import requests

# Get the HTML content of a webpage
response = requests.get('https://en.wikipedia.org/wiki/Move_Foward_Party')
soup = BeautifulSoup(response.text, 'html.parser')

# Extract the text from all paragraph tags
paragraphs = soup.find_all('p')
for p in paragraphs:
    print(p.get_text())
```

The Move Forward Party (Thai: พรรคร้าวไก่, RTGS: Phak Kao Klai, pronounced [pʰák kâ:w klāj]) is a social democratic and progressive political party in Thailand. It opposes the remaining influence of the military junta which ruled the country from 2014 to 2019. It was founded in 2014 as the Ruam Pattana Chart Thai Party (Thai: พรรคร่วมพัฒนาชาติไทย) and later changed its name to the Phung Luang Party (Thai: พรรคผืองหลวง), but after the 2019 Thai general election, reverted to its original name. It obtained its current name in 2020 after becoming the de facto successor to the dissolved Future Forward Party.

The party was officially founded on 1 May 2014 as the Ruam Pattana Chart Thai Party. [17]

Building Your Corpus

- A corpus is a large collection of text data that is used in NLP for various tasks like training models or testing performance.
- Python provides several functionalities that can help us build our corpus from various textual sources.

```
import os, os.path

# using the given path
path = os.path.expanduser('~/nltk_data')

# checking
if not os.path.exists(path):
    os.mkdir(path)

print ("Does path exists : ", os.path.exists(path))

import nltk.data
print ("\nDoes path exists in nltk : ",
      path in nltk.data.path)
```

Does path exists : True

Does path exists in nltk : True

```
# loading libraries
import nltk.data

nltk.data.load('corpora/cookbook/word_file.txt', format ='raw')

'''
```

Workshop # 3

1. Use request package to access API "<https://restcountries.com/v3.1/all>" and print in pretty format.
2. Use BeautifulSoup to scrape Pantip forum and print the content.

Module 5: Text Processing and Analysis

Tokenization

Natural Language Processing

['Natural', 'Language', 'Processing']

Tokenization

Why : tokenization is important step as a precursor to virtually all NLP tasks, as it converts unstructured data (text) into a format that can be represented as structured data (tokens).

Types:

1. **Word Tokenization** : The most common form of tokenization is word tokenization, where a piece of text is split into individual words. These individual words act as the base units for further analysis.
2. **Sentence Tokenization** : also known as sentence segmentation, is often applied where the text is split into individual sentences.
3. **Subword Tokenization** : including byte-pair encoding (BPE), unigram language model, and WordPiece. These methods help to deal with out-of-vocabulary (OOV) words in languages with rich morphology or in tasks that require learning meaningful subword units.

Word Tokenization in Python

```
from nltk.tokenize import word_tokenize

text = "This is a sample sentence for tokenization."
tokens = word_tokenize(text)
print(tokens)

# Output: ['This', 'is', 'a', 'sample', 'sentence', 'for', 'tokenization',
['This', 'is', 'a', 'sample', 'sentence', 'for', 'tokenization', '.']
```

```
from pythainlp.tokenize import word_tokenize

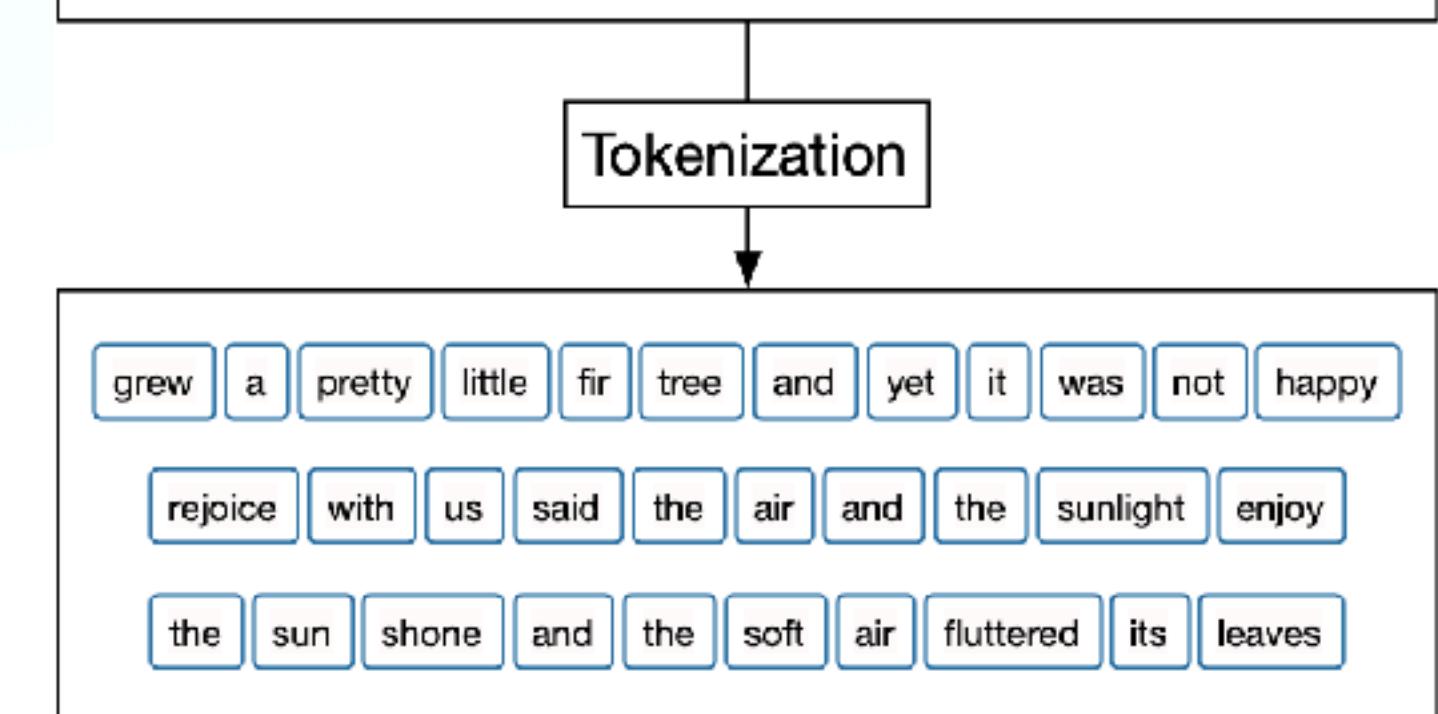
text = "นี่เป็นประโยคตัวอย่างสำหรับการตัดคำ"
tokens = word_tokenize(text)
print(tokens)

['นี่', 'เป็น', 'ประโยค', 'ตัวอย่าง', 'สำหรับ', 'การ', 'ตัด', 'คำ']
```

grew a pretty little fir-tree; and yet it was not happy

"Rejoice with us," said the air and the sunlight. Enjoy

The sun shone, and the soft air fluttered its leaves



Sentence Tokenization in Python

```
from nltk.tokenize import sent_tokenize

text = "This is a sample sentence. Here's another one!"
sentences = sent_tokenize(text)
print(sentences)
```

```
['This is a sample sentence.', "Here's another one!"]
```

```
from pythainlp.tokenize import sent_tokenize

text = "นี่เป็นประโยคตัวอย่าง. นี่คืออีกหนึ่ง!"
sentences = sent_tokenize(text, engine="whitespace+newline")
print(sentences)
# Output: ['นี่เป็นประโยคตัวอย่าง.', 'นี่คืออีกหนึ่ง!']
```

```
['นี่เป็นประโยคตัวอย่าง.', 'นี่คืออีกหนึ่ง!']
```

BPE Algorithm – a Frequency-based Model

- **Byte Pair Encoding** uses the frequency of subword patterns to shortlist them for merging.
- The drawback of using frequency as the driving factor is that you can end up having ambiguous final encodings that might not be useful for the new input text.
- But it still has the scope of improvement in terms of generating unambiguous tokens.

Subword Tokenization using BPE in Python

```
from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import Whitespace
import nltk
from nltk.corpus import gutenberg
nltk.download('gutenberg')
nltk.download('punkt')
plays = ['shakespeare-macbeth.txt', 'shakespeare-hamlet.txt', 'shakespeare-ca']
shakespeare = [" ".join(s) for ply in plays for s in gutenberg.sents(ply)]

# Initialize a tokenizer
tokenizer = Tokenizer(BPE(unk_token="[UNK]"))

# Customize pre-tokenization and decoding
tokenizer.pre_tokenizer = Whitespace()

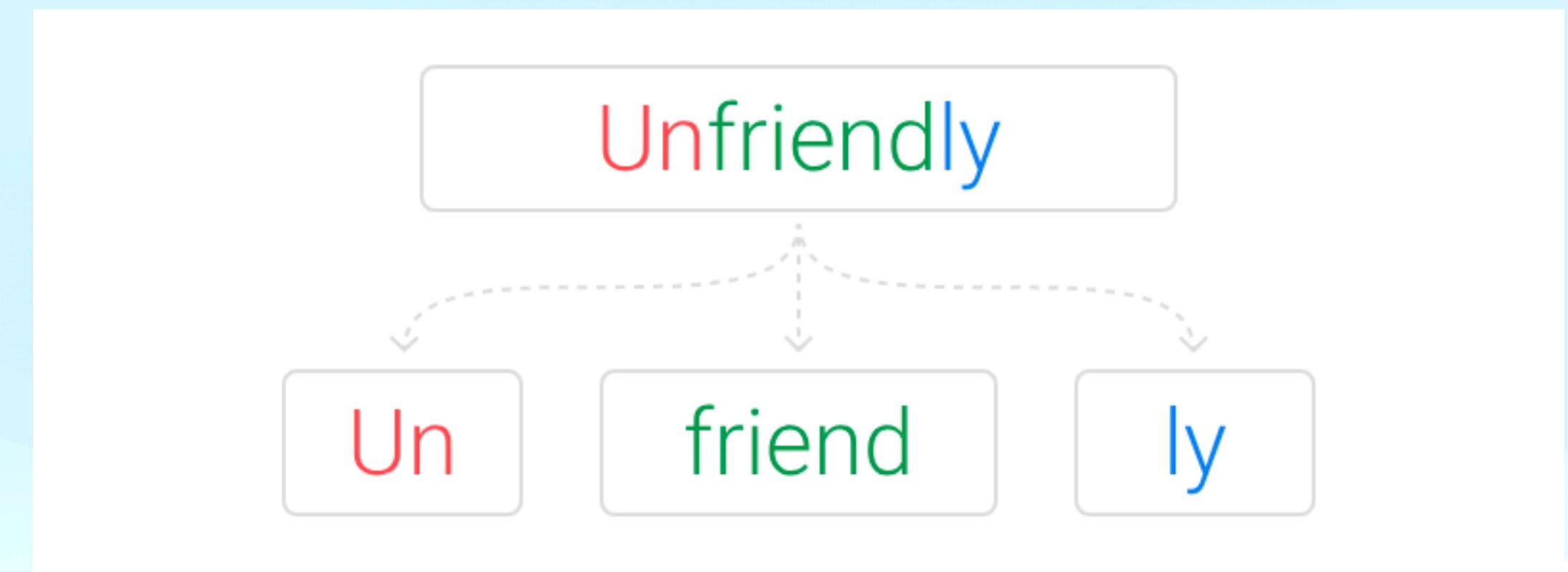
# Initialize a trainer, training from a file
special_tokens=["[UNK]", "[CLS]", "[SEP]", "[PAD]", "[MASK]"]
trainer = BpeTrainer(vocab_size=5000,special_tokens=special_tokens)
# Training the tokenizer
tokenizer.train_from_iterator(shakespeare, trainer)

# Encode a sentence
encoding = tokenizer.encode("This is a sample sentence for tokenization.")

print(encoding.tokens)
# Output will be tokens at the subword level
```

```
[nltk_data] Downloading package gutenberg to
[nltk_data]      /Users/a667207/nltk_data...
[nltk_data]      Package gutenberg is already up-to-date!
[nltk_data] Downloading package punkt to /Users/a667207/nltk_data...
[nltk_data]      Package punkt is already up-to-date!
```

```
['This', 'is', 'a', 's', 'am', 'ple', 'sent', 'ence', 'for', 'to', 'ken',
'iz', 'ation', '.']
```



Subword Tokenization using BPE in Python

```
from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import Whitespace
import nltk
from nltk.corpus import gutenberg
nltk.download('gutenberg')
nltk.download('punkt')
plays = ['shakespeare-macbeth.txt', 'shakespeare-hamlet.txt', 'shakespeare-ca']
shakespeare = [" ".join(s) for ply in plays for s in gutenberg.sents(ply)]

# Initialize a tokenizer
tokenizer = Tokenizer(BPE(unk_token="[UNK]"))

# Customize pre-tokenization and decoding
tokenizer.pre_tokenizer = Whitespace()

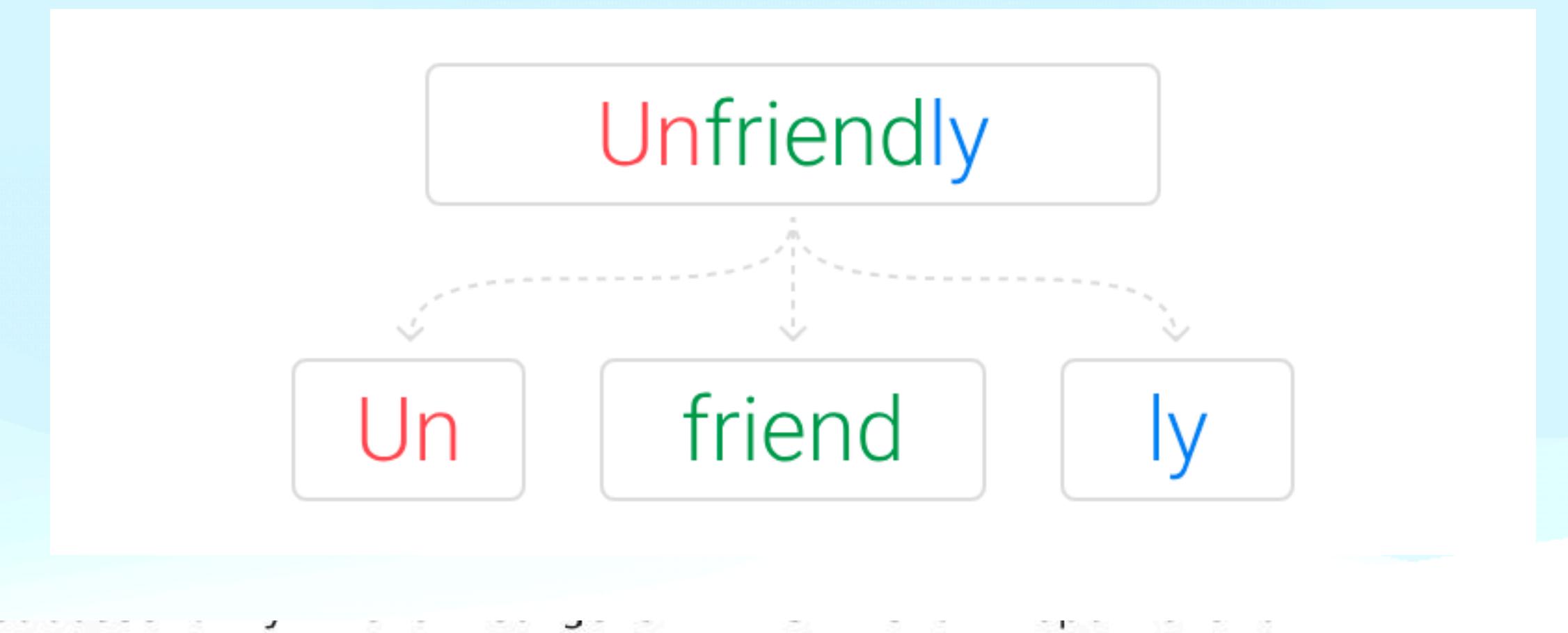
# Initialize a trainer, training from a file
special_tokens=[ "[UNK]", "[CLS]", "[SEP]", "[PAD]", "[MASK]" ]
trainer = BpeTrainer(vocab_size=5000, special_tokens=special_tokens)
# Training the tokenizer
tokenizer.train_from_iterator(shakespeare, trainer)

# Encode a sentence
encoding = tokenizer.encode("This is a sample sentence for tokenization.")

print(encoding.tokens)
# Output will be tokens at the subword level
```

```
[nltk_data] Downloading package gutenberg to
[nltk_data]   /Users/a667207/nltk_data...
[nltk_data] Package gutenberg is already up-to-date!
[nltk_data] Downloading package punkt to /Users/a667207/nltk_data...
[nltk_data] Package punkt is already up-to-date!
```

```
['This', 'is', 'a', 's', 'am', 'ple', 'sent', 'ence', 'for', 'to', 'ken',
'iz', 'ation', '.']
```



```
from pythainlp.tokenize import subword_tokenize

text_1 = "ยุคเริ่มแรกของ ราชวงศ์พมึง"
text_2 = "ความแปลกแยกและพัฒนาการ"

print(subword_tokenize(text_1, engine='wangchanberta'))

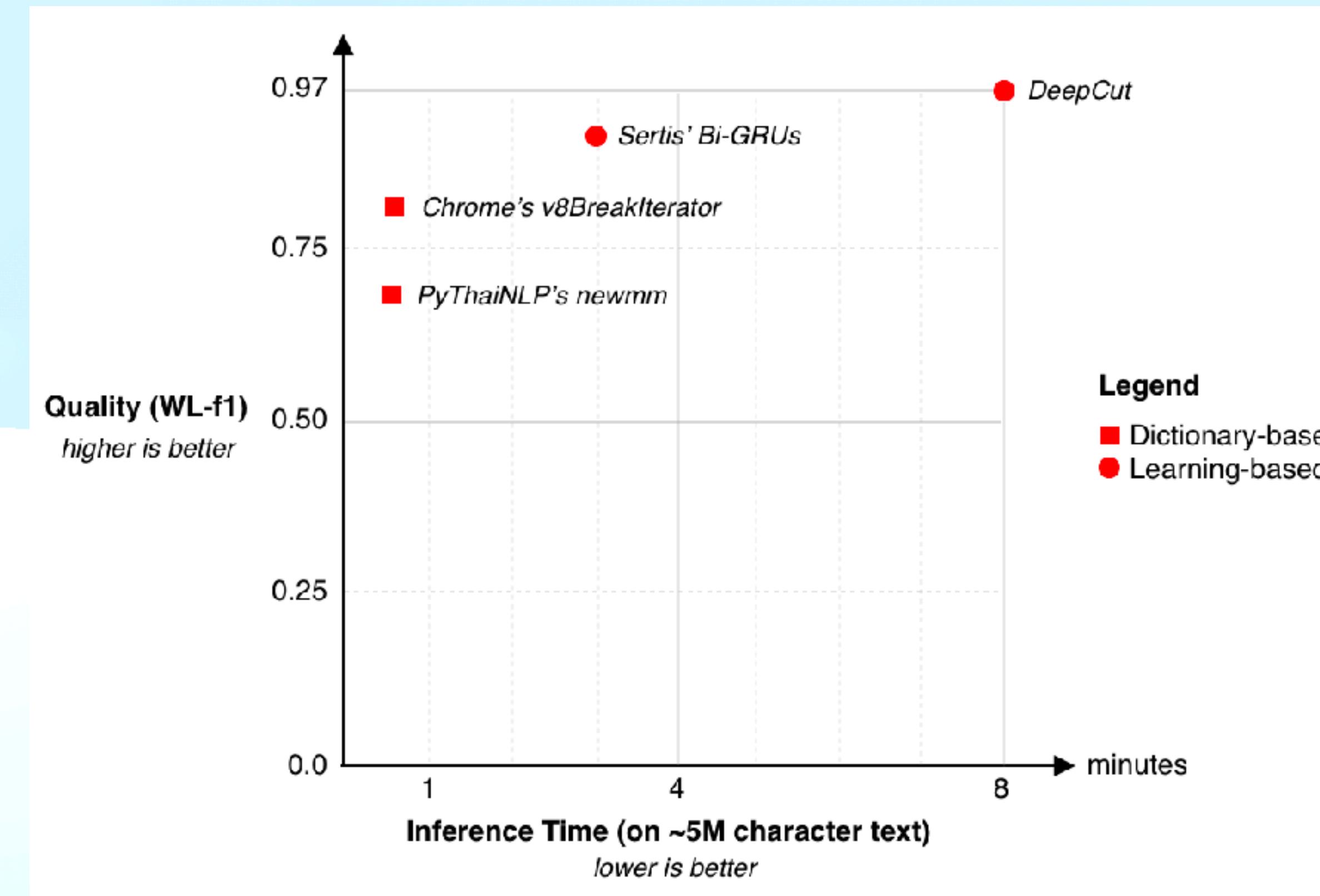
print(subword_tokenize(text_2, engine='wangchanberta'))
```

```
['_', 'ยุค', 'เริ่มแรก', 'ของ', '_', 'ราชวงศ์', 'พมึง']
['_', 'ความ', 'แปลก', 'แยก', 'และ', 'พัฒนาการ']
```

Word Tokenization for Thai

- Research in word tokenization for Thai started around 1990. Over these 20 years, there have been several algorithms being proposed to address the problem. These algorithms can be clustered into two categories, namely
 1. Dictionary-based (Chrome's V8BreakIterator and PyThaiNLP's newmm)
 2. Learning-based (DeepCut and Sertis' Bi-GRUs)

Word Tokenization for Thai



Options for engine

- `attacut` - wrapper for `AttaCut`, learning-based approach
- `deepcut` - wrapper for `DeepCut`, learning-based approach
- `icu` - wrapper for a word tokenizer in `PyICU`, from ICU (International Components for Unicode), dictionary-based
- `longest` - dictionary-based, longest matching
- `mm` - "multi-cut", dictionary-based, maximum matching
- `nercut` - dictionary-based, maximal matching, constrained with Thai Character Cluster (TCC) boundaries, combining tokens that are parts of the same named-entity
- `newmm` (default) - "new multi-cut", dictionary-based, maximum matching, constrained with Thai Character Cluster (TCC) boundaries with improve the TCC rule that used in `newmm`.
- `newmm-safe` - `newmm`, with a mechanism to avoid long processing time for text with continuous ambiguous breaking points
- `nlpO3` - wrapper for a word tokenizer in `nlpO3`, `newmm` adaptation in Rust (2.5x faster)
- `oskut` - wrapper for `OSKut`, Out-of-domain StackEd cut for Word Segmentation
- `sefr_cut` - wrapper for `SEFR CUT`, Stacked Ensemble Filter and Refine for Word Segmentation
- `tltk` - wrapper for `TLTK`,

maximum collocation approach

N-grams

- **An n-gram** is a contiguous sequence of n items from a given sample of text or speech. In the context of NLP, an n-gram can be a sequence of letters, syllables, or words. They have a wide range of applications, like language models, semantic features, spelling correction, machine translation, text mining, etc.
- n-grams are classified into the following types, depending on the value that ‘n’ takes.
 - 1.Unigram
 - 2.Bigram
 - 3.Trigram
 - 4.n-gram

Example of N-grams

- Let's understand n-grams practically with the help of the following sample sentence:

“I reside in Bangkok”.

SL.No.	Type of n-gram	Generated n-grams
1.	Unigram.	[“I”, “reside”, “in”, “Bangkok”]
2.	Bigram.	[“I reside”, “reside in”, “in Bangkok”]
3.	Trigram.	[“I reside in”, “reside in Bangkok”]

Example of N-grams

```
from nltk import ngrams
sentence = 'I reside in Bangkok.'
n = 1
unigrams = ngrams(sentence.split(), n)

for grams in unigrams:
    print(grams)

('I',)
('reside',)
('in',)
('Bangkok.',)
```

Example of N-grams

```
n = 2
bigrams = ngrams(sentence.split(), n)

for grams in bigrams:
    print(grams)
```

```
('I', 'reside')
('reside', 'in')
('in', 'Bangkok.')
```

```
n = 3
trigrams = ngrams(sentence.split(), n)

for grams in trigrams:
    print(grams)
```

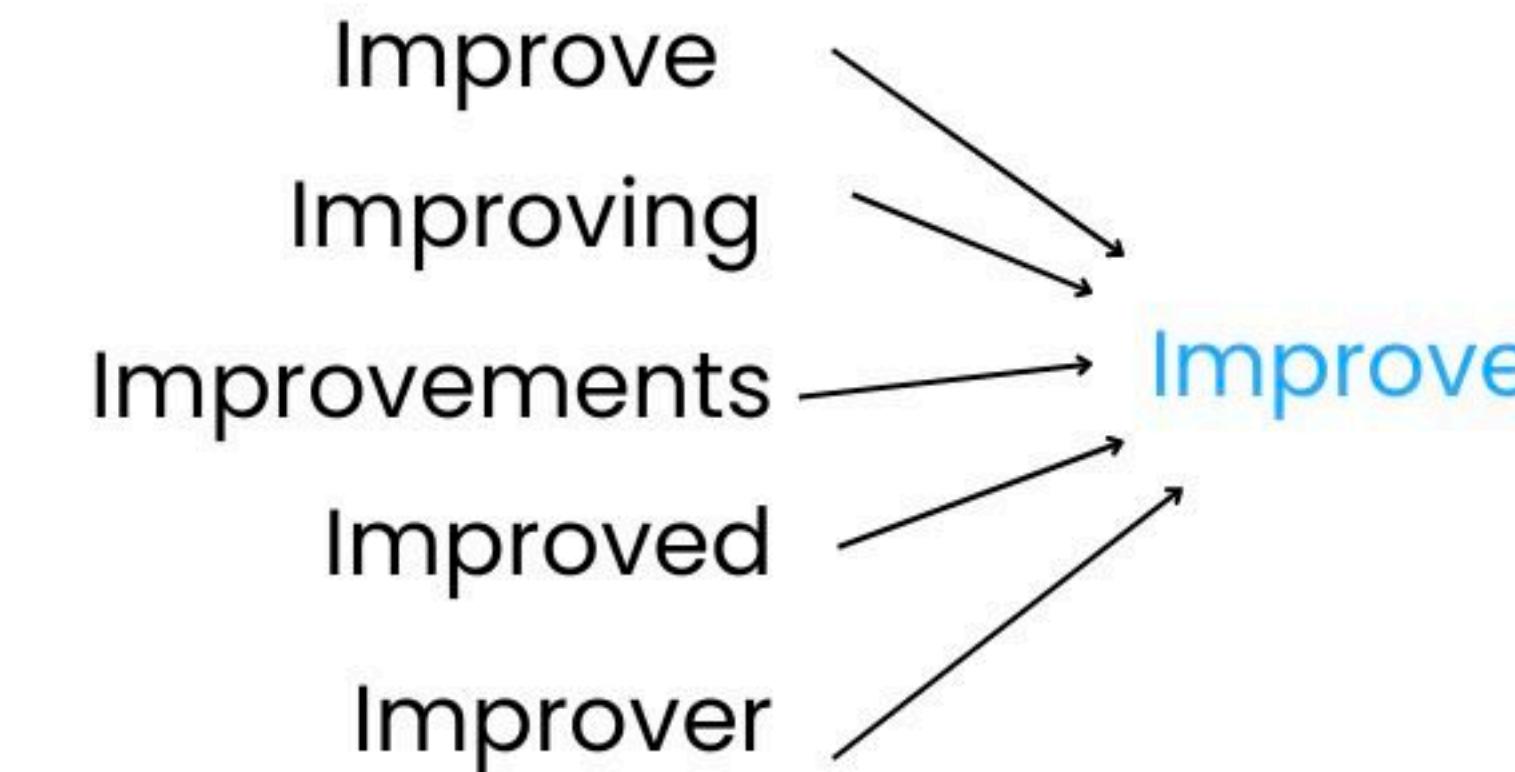
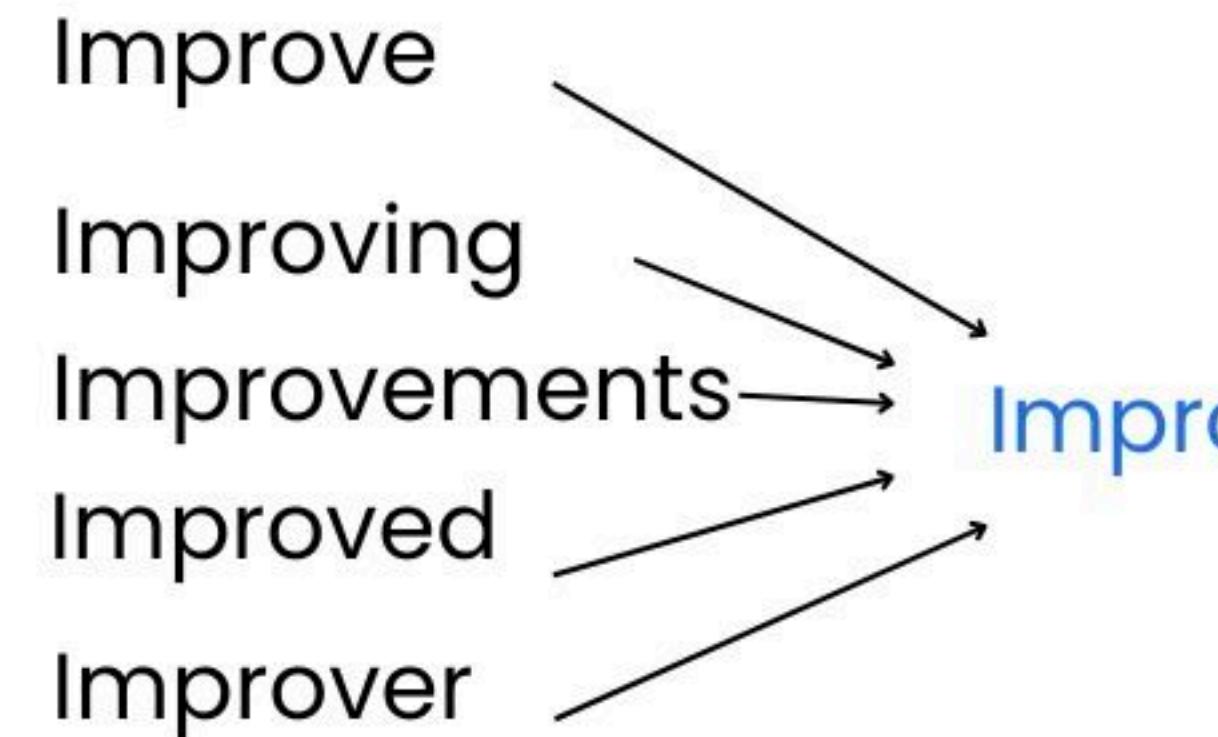
```
('I', 'reside', 'in')
('reside', 'in', 'Bangkok.')
```

Scriptio Continua in Languages

- **Scriptio continua** is a style of writing without spaces or other marks between words or sentences. In the Chinese, Japanese, Korean (CJK languages), Thai and Javanese language, words are often written together without spaces, a form of scriptio continua.
- **Challenge:** This presents a unique challenge for NLP, as word tokenization becomes non-trivial.

Stemming and Lemmatization

- **Stemming and lemmatization** are techniques used to reduce words to their root form. Stemming uses a heuristic process that removes the end of words, while lemmatization takes into account morphological analysis of the words.



Stemming and Lemmatization

```
from nltk.stem import PorterStemmer, WordNetLemmatizer
nltk.download('wordnet')
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

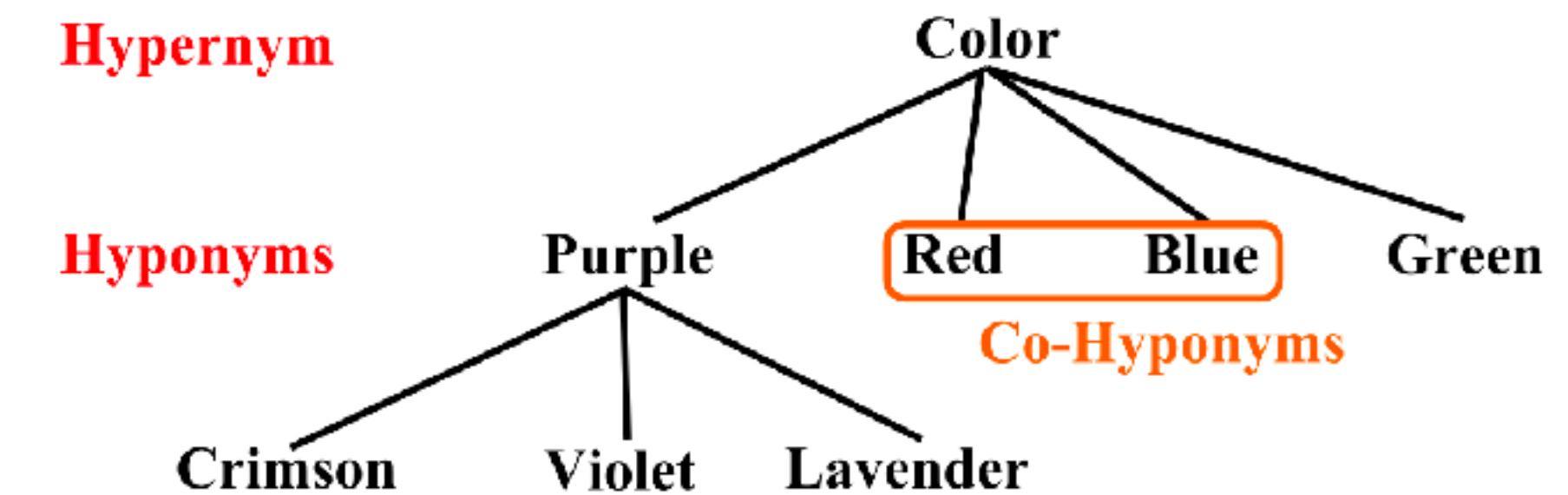
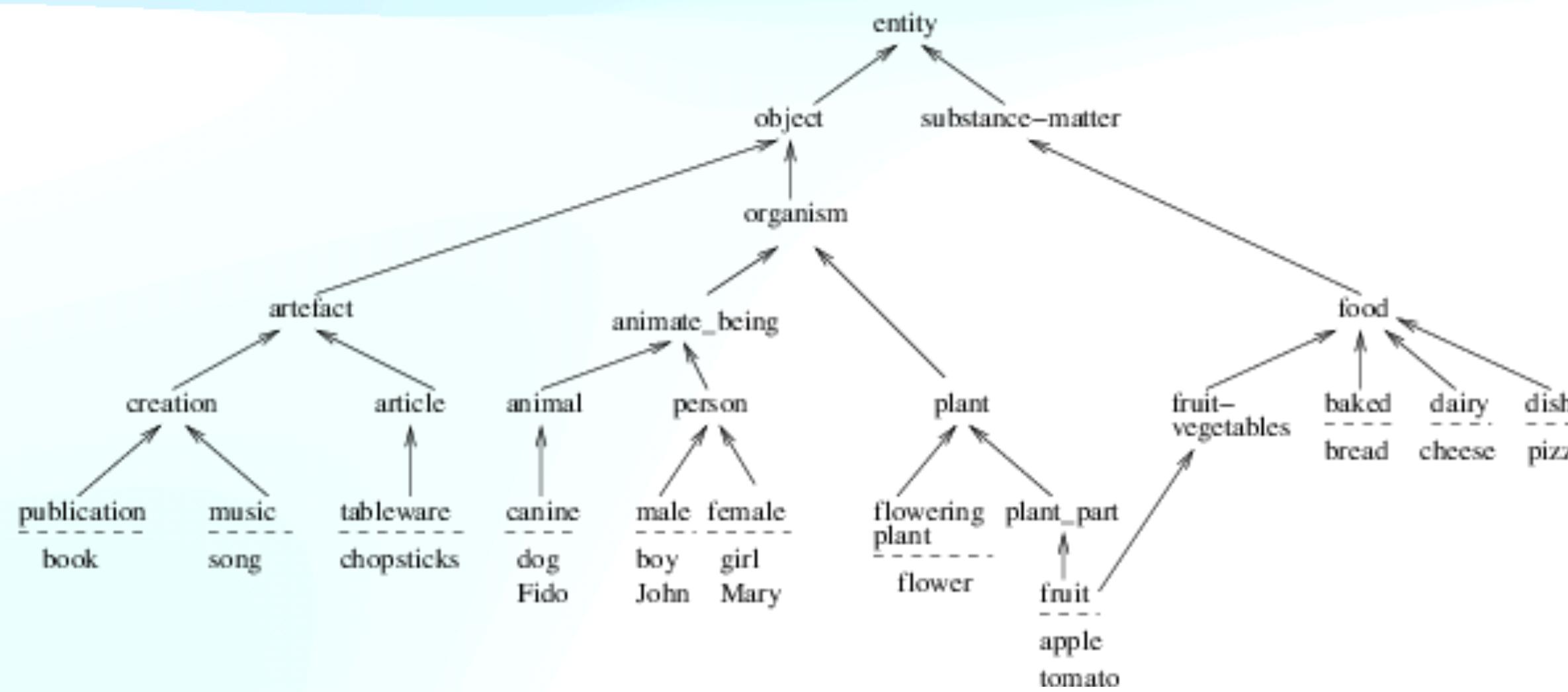
word = 'improvements'
print(stemmer.stem(word)) # Output: 'run'
print(lemmatizer.lemmatize(word, pos='v')) # Output: 'run'
```

```
improv
improvements
```

```
[nltk_data] Downloading package wordnet to /Users/a667207/nltk_data...
[nltk_data] Package wordnet is already up-to-date!
```

Synsets and Hypernyms

- A **synset** is a set of synonyms that share a common meaning. A **hypernym** is a word with a broad meaning constituting a category into which words with more specific meanings fall.



Synsets and Hyponyms

```
from nltk.corpus import wordnet

synsets = wordnet.synsets('Boy')
print(synsets)

hyponyms = synsets[0].hyponyms()
print(hyponyms)
```

```
[Synset('male_child.n.01'), Synset('boy.n.02'), Synset('son.n.01'), Synset
('boy.n.04')]
[Synset('male.n.02')]
```

Workshop # 4

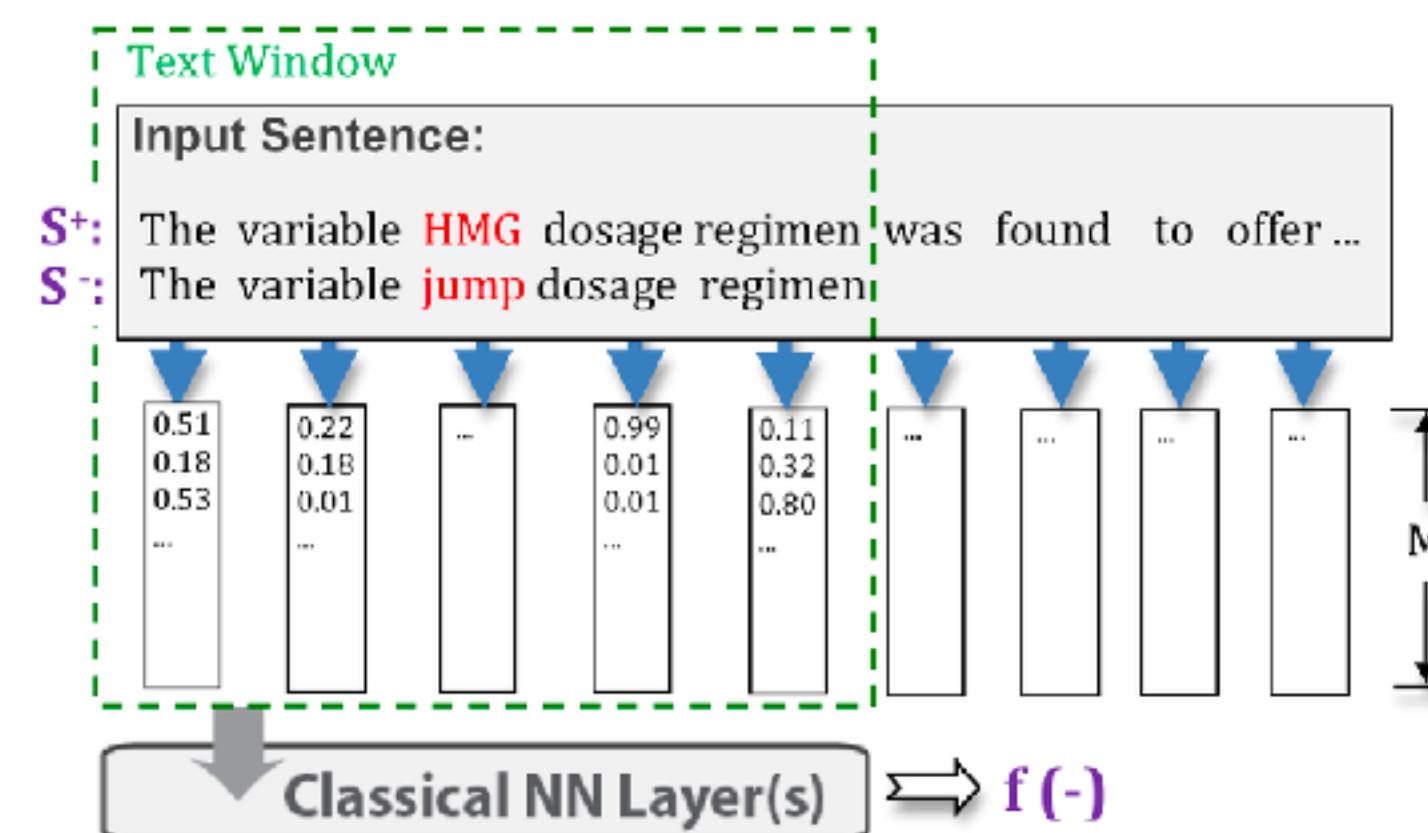
1. Practice text processing using NLTK, PyThaiNLP

https://github.com/vkrit/NLP_using_python

Module 5: Word Embeddings and Language Models

Introduction to Word Embeddings and Language Models

Word embeddings and language models represent the cornerstone of modern NLP. Word embeddings convert words into high-dimensional vectors that capture their semantic meaning. **Language models**, on the other hand, learn the probability distribution of a sequence of words in a language, enabling us to generate coherent sentences or predict the next word in a sentence.



Word Embeddings - Concept

- **Word embeddings transform words into high-dimensional vectors** in a way that preserves their semantic relationships. For example, words with similar meanings are close to each other in the vector space, and relationships between words can be represented by vector arithmetic (e.g., 'king' - 'man' + 'woman' = 'queen').
- Popular algorithms for creating word embeddings include **Word2Vec** and **GloVe**. These algorithms work by learning vectors that are good at predicting a word given its context (Word2Vec) or that capture the global co-occurrence statistics of the corpus (GloVe).

How do we represent 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.

Commonest linguistic way of thinking of meaning:

signifier (symbol) \Leftrightarrow signified (idea or thing)

= denotational semantics

tree $\Leftrightarrow \{ , , , , \dots \}$

How do we have usable meaning in a computer?

- Previously commonest NLP solution: Use, e.g., **WordNet**, a thesaurus containing lists of synonym sets and hypernyms (“is a” relationships)

e.g., synonym sets containing “good”:

```
from nltk.corpus import wordnet as wn
poses = {'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}
for synset in wn.synsets("good"):
    print("{}: {}".format(poses[synset.pos()],
                           ", ".join([l.name() for l in synset.lemmas()])))
```

```
noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good
adj: good
adj (sat): estimable, good, honorable, respectable
adj (sat): beneficial, good
adj (sat): good
adj (sat): good, just, upright
...
adverb: well, good
adverb: thoroughly, soundly, 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(pandaclosure(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')]
```

Problems with resources like WordNet

- A useful resource but missing nuance:
 - e.g., “**proficient**” is listed as a synonym for “**good**”. This is only correct in some contexts
 - Also, WordNet lists offensive synonyms in some synonym sets without any coverage of the connotations or appropriateness of words
- Missing new meanings of words:
 - e.g., **wicked, badass, nifty, wizard, genius, ninja, bombest**
 - Impossible to keep up-to-date!
- Requires human labor to create and adapt
- Can’t be used to accurately compute word similarity

Representing words as discrete symbols

- In traditional NLP, we regard words as discrete symbols: hotel, conference, motel – a localist representation
- Such symbols for words can be represented by **one-hot** (Means one 1, the rest 0s) vectors:
 - motel = [0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 - hotel = [0 0 0 0 0 0 1 0 0 0 0 0 0 0]
- Vector dimension = number of words in vocabulary (e.g., 500,000+)

Problem with words as discrete symbols

- Example: in web search, if a user searches for “**Seattle motel**”, we would like to match documents containing “**Seattle hotel**”
- But:
 - **motel** = [0 0 0 0 0 0 0 0 0 1 0 0 0]
 - **hotel** = [0 0 0 0 0 0 1 0 0 0 0 0 0]
- These two vectors are orthogonal
- There is **no natural notion of similarity for one-hot vectors!**
- **Solution:**
 - Could try to rely on WordNet’s list of synonyms to get similarity?
 - But it is well-known to fail badly: incompleteness, etc.
 - **Instead: learn to encode similarity in the vectors themselves**

Representing words by their context

- Distributional semantics: **A word's meaning is given by the words that frequently appear close-by**
 - “*You shall know a word by the company it keeps*” (J. R. Firth 1957: 11)
 - **One of the most successful ideas of modern statistical NLP!**
- When a word w appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).
- We 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 arm...

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

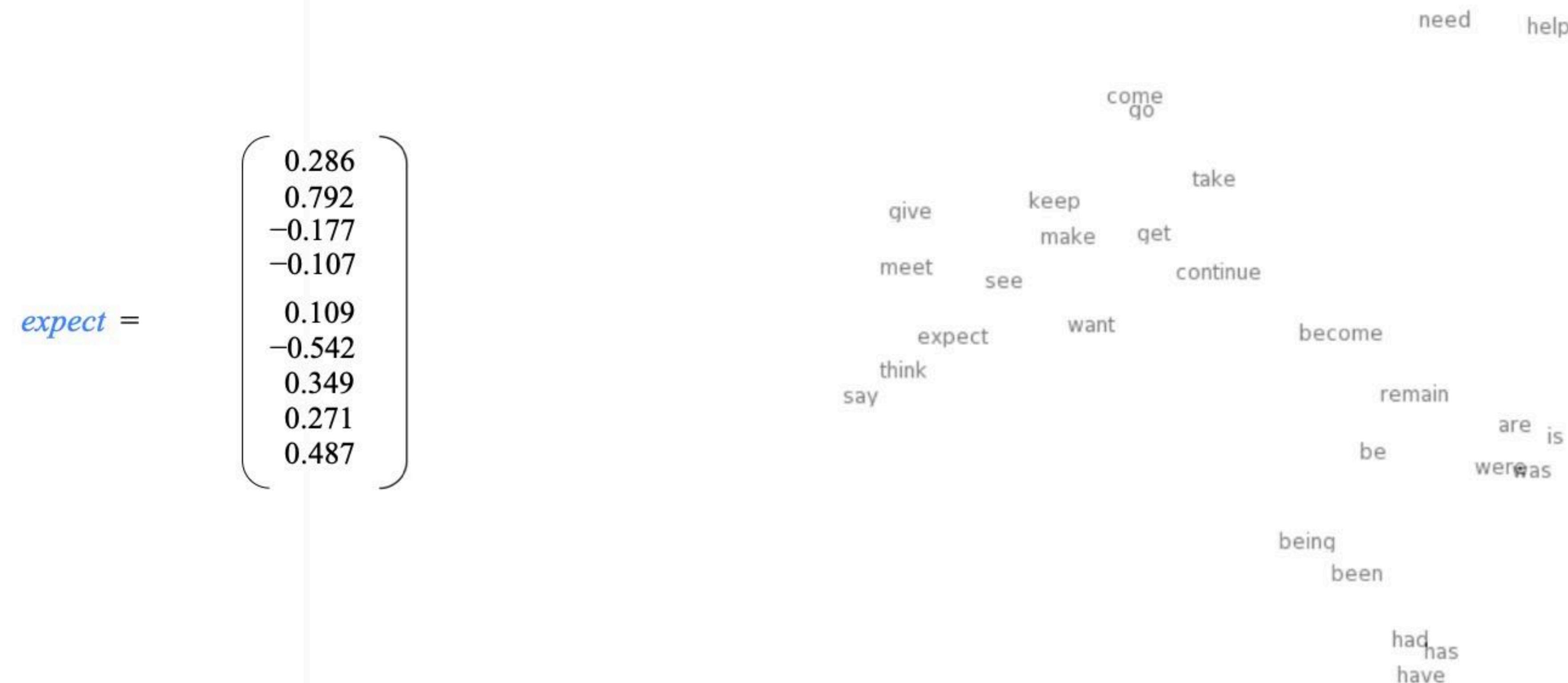
Word vectors

- 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

$$\begin{aligned} \text{banking} &= \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix} \\ \text{monetary} &= \begin{pmatrix} 0.413 \\ 0.582 \\ -0.007 \\ 0.247 \\ 0.216 \\ -0.718 \\ 0.147 \\ 0.051 \end{pmatrix} \end{aligned}$$

Note: **word vectors** are also called **(word) embeddings** or **(neural) word representations**
They are a **distributed** representation

Word meaning as a neural word vector – visualization

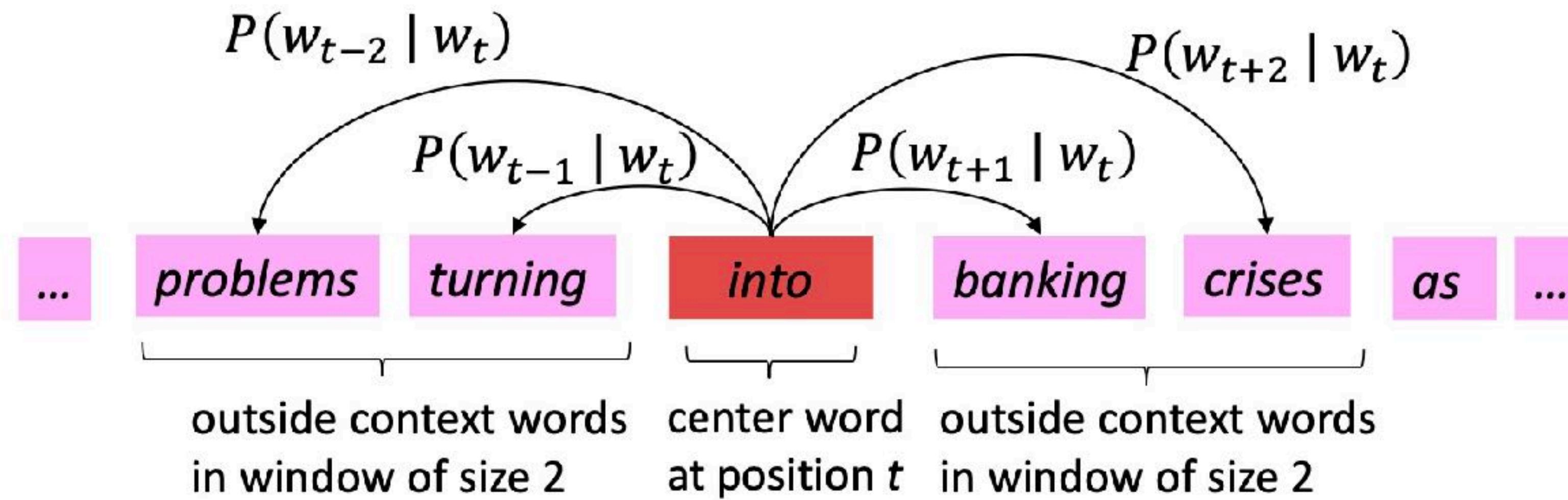


Word2vec: Overview

- **Word2vec** (Mikolov et al. 2013) is a framework for learning word vectors
- **Idea:**
 - We have a large corpus (“body”) of text: a long list of words
 - Every word in a fixed vocabulary is represented by a **vector**
 - Go through each position t in the text, which has a center word c and context (“outside”) words o
 - Use the **similarity of the word vectors** for c and o to **calculate the probability** of o given c (or vice versa)
 - **Keep adjusting the word vectors** to maximize this probability

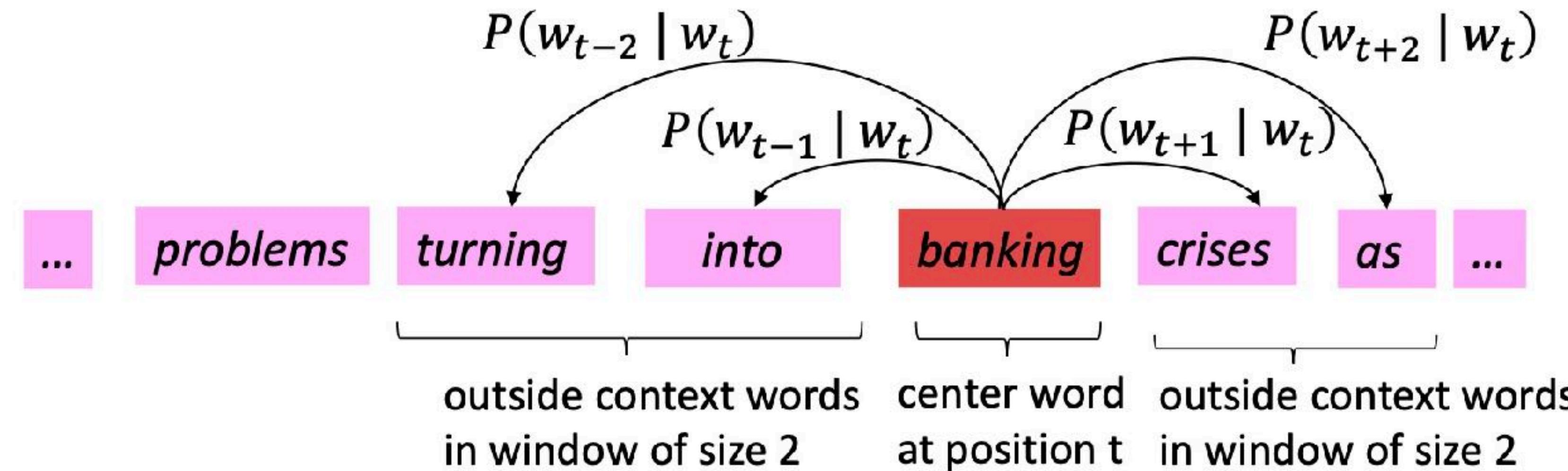
Word2vec: Overview

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



Word2vec: Overview

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



Word2vec: objective function

For each position $t = 1, \dots, T$, predict context words within a window of fixed size m , given center word w_t . Data likelihood:

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

θ is all variables to be optimized

sometimes called a *cost* or *loss* function

The **objective function** $J(\theta)$ is the (average) negative log likelihood:

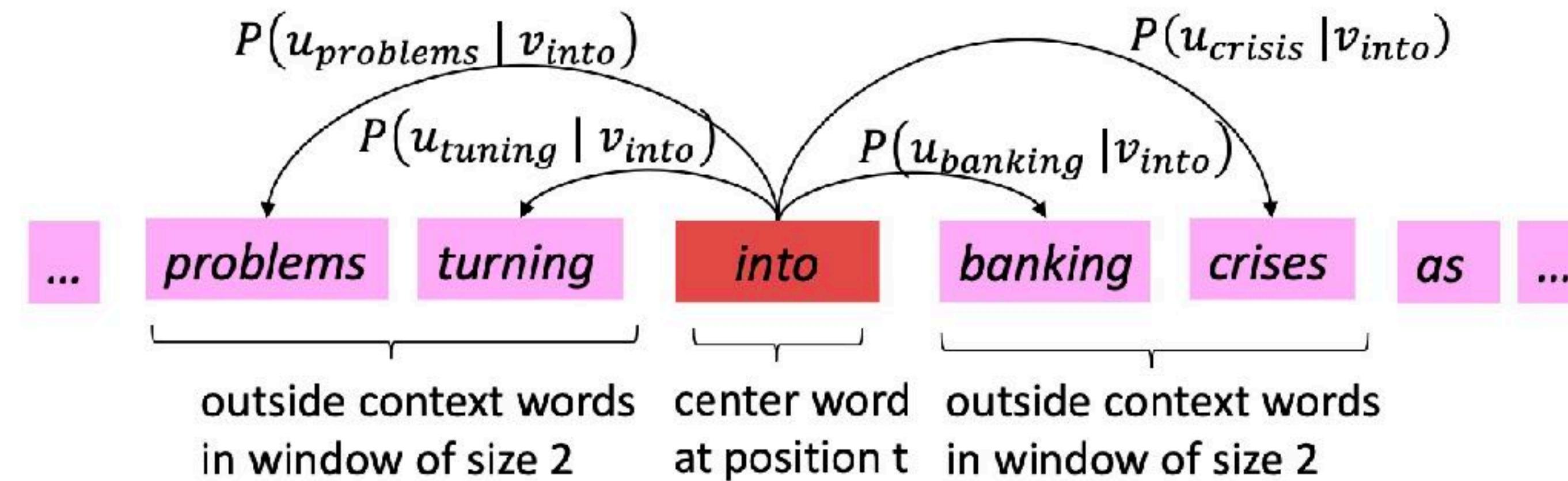
$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^T \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j} | w_t; \theta)$$

Minimizing objective function \Leftrightarrow Maximizing predictive accuracy

Word2vec with Vectors

- Example windows and process for computing $P(w_{t+j} | w_t)$
- $P(u_{problems} | v_{into})$ short for $P(problems | into ; u_{problems}, v_{into}, \theta)$

All words vectors θ
appear in denominator



Word2vec: prediction function

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

② Exponentiation makes anything positive
↓
 $\exp(u_o^T v_c)$

① Dot product compares similarity of o and c .
 $u^T v = u \cdot v = \sum_{i=1}^n u_i v_i$
Larger dot product = larger probability

③ Normalize over entire vocabulary to give probability distribution

- This is an example of the **softmax function** $\mathbb{R}^n \rightarrow (0,1)^n$

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

Open region

- The softmax function maps arbitrary values x_i to a probability distribution p_i
 - “max” because amplifies probability of largest x_i
 - “soft” because still assigns some probability to smaller x_i
 - Frequently used in Deep Learning

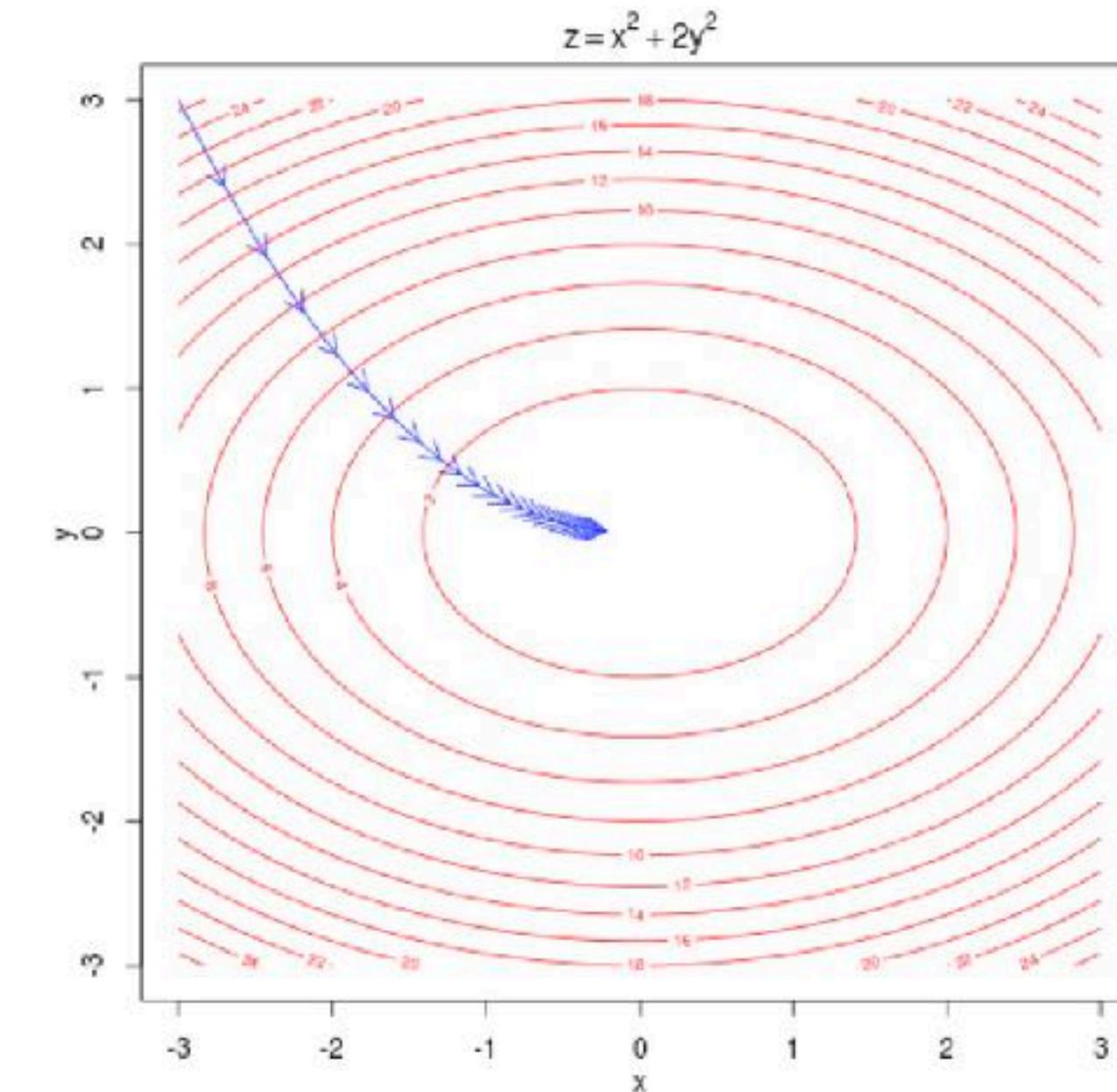
But sort of a weird name because it returns a distribution!

Train the model: Optimize value of parameters to minimize loss

To train a model, we gradually adjust parameters to minimize a loss

- Recall: θ represents **all** the model parameters, in one long vector
- In our case, with d -dimensional vectors and V -many words, we have →
- Remember: every word has two vectors

$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \vdots \\ v_{zebra} \\ u_{aardvark} \\ u_a \\ \vdots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$



- We optimize these parameters by walking down the gradient (see right figure)
- We compute **all** vector gradients!

Word Embeddings using Gensim Library

```
from gensim.models import Word2Vec

# Prepare a sample corpus
corpus = [['king', 'is', 'a', 'man'], ['queen', 'is', 'a', 'woman'], ['man'

# Train a Word2Vec model
model = Word2Vec(corpus, min_count=1)
print(model.wv['king']) # Print vector representation of 'king'

[ 8.13227147e-03 -4.45733406e-03 -1.06835726e-03  1.00636482e-03
 -1.91113955e-04  1.14817743e-03  6.11386076e-03 -2.02715401e-05
 -3.24596534e-03 -1.51072862e-03  5.89729892e-03  1.51410222e-03
 -7.24261976e-04  9.33324732e-03 -4.92128357e-03 -8.38409644e-04
  9.17541143e-03  6.74942741e-03  1.50285603e-03 -8.88256077e-03
  1.14874600e-03 -2.28825561e-03  9.36823711e-03  1.20992784e-03
  1.49006362e-03  2.40640994e-03 -1.83600665e-03 -4.99963388e-03
  2.32429506e-04 -2.01418041e-03  6.60093315e-03  8.94012302e-03
 -6.74754381e-04  2.97701475e-03 -6.10765442e-03  1.69932481e-03
 -6.92623248e-03 -8.69402662e-03 -5.90020278e-03 -8.95647518e-03
  7.27759488e-03 -5.77203138e-03  8.27635173e-03 -7.24354526e-03
  3.42167495e-03  9.67499893e-03 -7.78544787e-03 -9.94505733e-03
 -4.32914635e-03 -2.68313056e-03 -2.71289347e-04 -8.83155130e-03
 -8.61755759e-03  2.80021061e-03 -8.20640661e-03 -9.06933658e-03
 -2.34046578e-03 -8.63180775e-03 -7.05664977e-03 -8.40115082e-03
 -3.01328895e-04 -4.56429832e-03  6.62717456e-03  1.52716041e-03
 -3.34147573e-03  6.10897178e-03 -6.01328490e-03 -4.65616956e-03
 -7.20750913e-03 -4.33658017e-03 -1.80932996e-03  6.48964290e-03
 -2.77039292e-03  4.91896737e-03  6.90444233e-03 -7.46370573e-03
  4.56485013e-03  6.12697843e-03 -2.95447465e-03  6.62502181e-03
  6.12587947e-03 -6.44348515e-03 -6.76455162e-03  2.53895880e-03
 -1.62381888e-03 -6.06512791e-03  9.49920900e-03 -5.13014663e-03
 -6.55409694e-03 -1.19885204e-04 -2.70142802e-03  4.44400299e-04
 -3.53745813e-03 -4.19330609e-04 -7.08615757e-04  8.22820642e-04
  8.19481723e-03 -5.73670724e-03 -1.65952800e-03  5.57160750e-03]
```

Word Embeddings using Gensim Library

See Demo using Jupyter

Workshop # 5 : Word2Vec using Gensim

- Using Reuters corpus from NTLK to train Word2Vec model from Gensim model
- Set size of word vectors to be 100 and ignore words that appear less than 2 times in the corpus
- Explore the model by finding the vector of a word of your choice and find the most similar words to a given word
- Save the trained model for later use.

Bag of Words (BoW)

- The **Bag of Words** (BoW) model is a popular way of representing text data in machine learning. The fundamental idea behind it is to represent text as a bag of its words, disregarding grammar, word order, and even ignoring the frequency of each word.
- **Strengths:** The Bag of Words model is simple and efficient to implement. It's a great place to start with text analysis, particularly with small and medium-sized datasets.
- **Weaknesses:** It disregards the order of words, the context around each word (semantics), and grammar rules. Also, it treats words as equally important by not taking into account the relative importance of words in the text.

Bag of Words (BoW) : Applications

- 1. Information Retrieval:** BoW is widely used in information retrieval systems (like search engines). In this context, each document is represented as a bag of words and relevance of a document to a query is calculated using the degree of similarity between the document and the query vectors.
- 2. Document Classification:** BoW can be used for document classification tasks like spam filtering, sentiment analysis, or topic assignment. In these applications, the frequency of words in the documents is a good indicator of the class to which the document belongs.
- 3. Topic Modeling:** BoW is used in topic modeling where the objective is to identify the main topics that occur in a collection of documents. Latent Dirichlet Allocation (LDA) is a popular technique for topic modeling and it treats each document as a bag of words.
- 4. Text Summarization:** BoW can be used to extract the most frequent words in a document as keywords and use them for summary.
- 5. Feature Extraction:** In Machine Learning, BoW is often used for feature extraction. For instance, in a sentiment analysis task, a BoW can be used to identify the words that often occur in positive reviews and those that occur frequently in negative reviews.

Bag of Words (BoW)

```
from sklearn.feature_extraction.text import CountVectorizer

# Example corpus
corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?']

# Create the BoW model
vectorizer = CountVectorizer()

# Learn a vocabulary dictionary of all tokens in the raw documents
X = vectorizer.fit_transform(corpus)

# Summarize
print(vectorizer.get_feature_names_out())
print(X.toarray())

# Output
# ['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
# [[0 1 1 1 0 0 1 0 1]
# [0 2 0 1 0 1 1 0 1]
# [1 0 0 1 1 0 1 1 1]
# [0 1 1 1 0 0 1 0 1]]
```



```
['and' 'document' 'first' 'is' 'one' 'second' 'the' 'third' 'this']
[[0 1 1 1 0 0 1 0 1]
[0 2 0 1 0 1 1 0 1]
[1 0 0 1 1 0 1 1 1]
[0 1 1 1 0 0 1 0 1]]
```

Term Frequency-Inverse Document Frequency (TF-IDF)

- **TF-IDF stands for Term Frequency-Inverse Document Frequency**, a numerical statistic used to reflect how important a word is to a document in a collection or corpus. It's a popular feature extraction method for text data, often used in information retrieval and text mining.
- The TF-IDF value increases proportionally to the **number of times a word appears in the document**, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general.

Term Frequency-Inverse Document Frequency (TF-IDF)

Strengths:

- TF-IDF can overcome the shortcomings of the Bag of Words (BoW) model by taking into account not just the occurrence of words in a single document (or a piece of text) but in the entire corpus.
- It reduces the impact of common words in determining the importance of a document.

Weaknesses:

- Like BoW, TF-IDF does not consider the order of words and semantics of the word in the document.
- It assumes that the words are independent of each other.

Term Frequency-Inverse Document Frequency (TF-IDF)

```
from sklearn.feature_extraction.text import TfidfVectorizer

# create a corpus of sentences
corpus = [
    'The sky is blue and beautiful.',
    'Love this blue and beautiful sky!',
    'The quick brown fox jumps over the lazy dog.',
    'The brown fox is quick and the blue dog is lazy.',
    'The sky is very blue and the sky is very beautiful today.',
    'The dog is lazy but the brown fox is quick.'
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)

# print the tf-idf values
print(X.toarray())

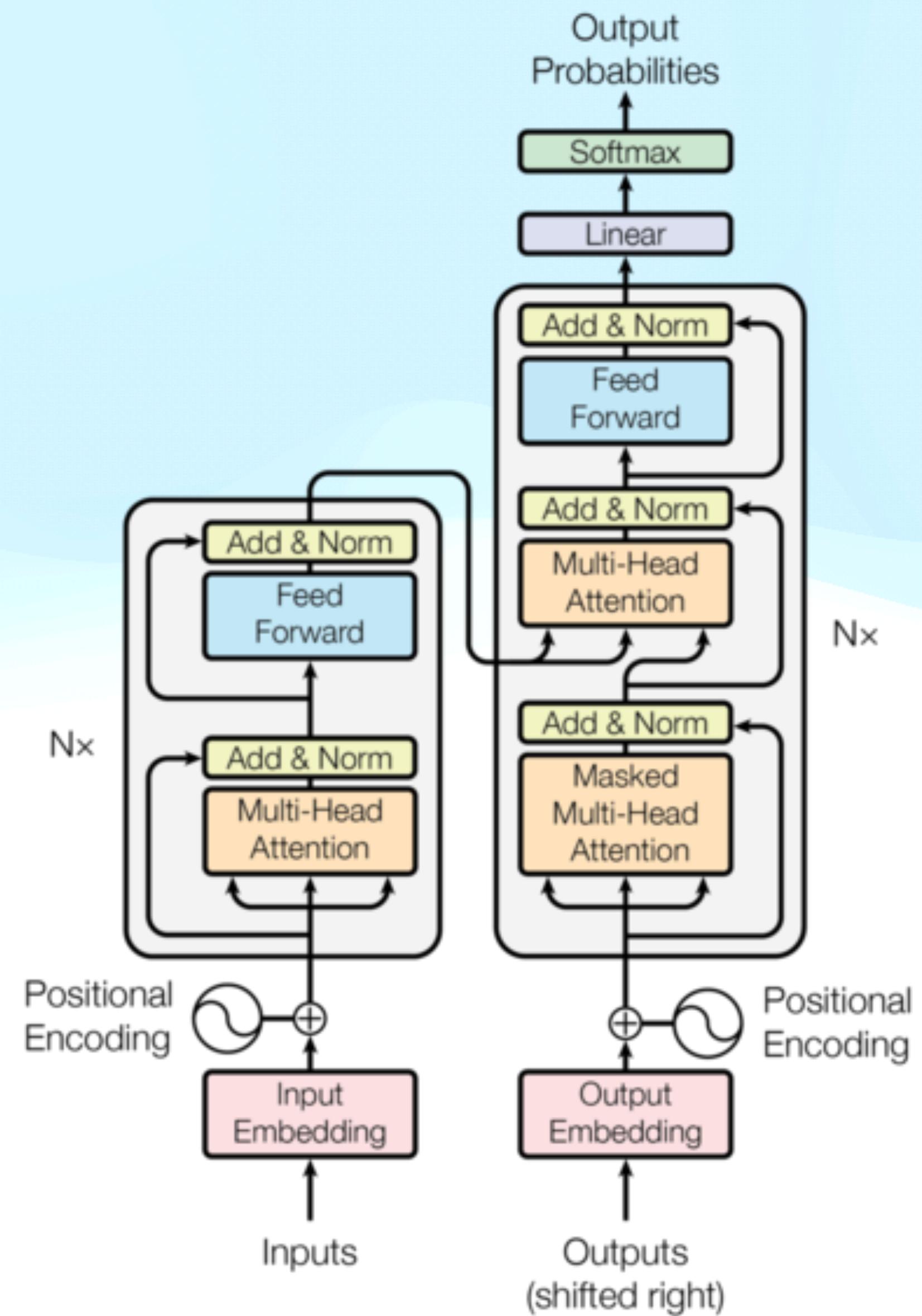
# print the feature names
print(vectorizer.get_feature_names_out())
```



```
[[0.39315931 0.458803 0.39315931 0.          0.          0.
   0.          0.39315931 0.          0.          0.
   0.          0.458803 0.33952451 0.          0.          0.
   0.          0.30999542 0.36175368 0.30999542 0.          0.
   0.          0.          0.          0.52252953 0.
   0.          0.36175368 0.          0.52252953 0.          0.
   0.          0.          0.          0.29665213 0.          0.29665213
   0.29665213 0.          0.42849459 0.29665213 0.          0.42849459
   0.29665213 0.          0.43905847 0.          0.          0.
   0.25164011 0.          0.25164011 0.2936551 0.          0.2936551
   0.2936551 0.50328023 0.          0.2936551 0.          0.
   0.2936551 0.          0.43462273 0.          0.          0.
   0.182579 0.21306323 0.182579 0.          0.          0.
   0.          0.365158 0.          0.          0.          0.
   0.          0.42612646 0.31534314 0.          0.3077559 0.61551179]
   [0.          0.          0.          0.2861327 0.41329997 0.2861327
   0.2861327 0.49038798 0.          0.2861327 0.          0.
   0.2861327 0.          0.42348924 0.          0.          0.        ]]
['and' 'beautiful' 'blue' 'brown' 'but' 'dog' 'fox' 'is' 'jumps' 'lazy'
 'love' 'over' 'quick' 'sky' 'the' 'this' 'today' 'very']
```

Module 6 : Transformers Architecture

Attention and Transformers



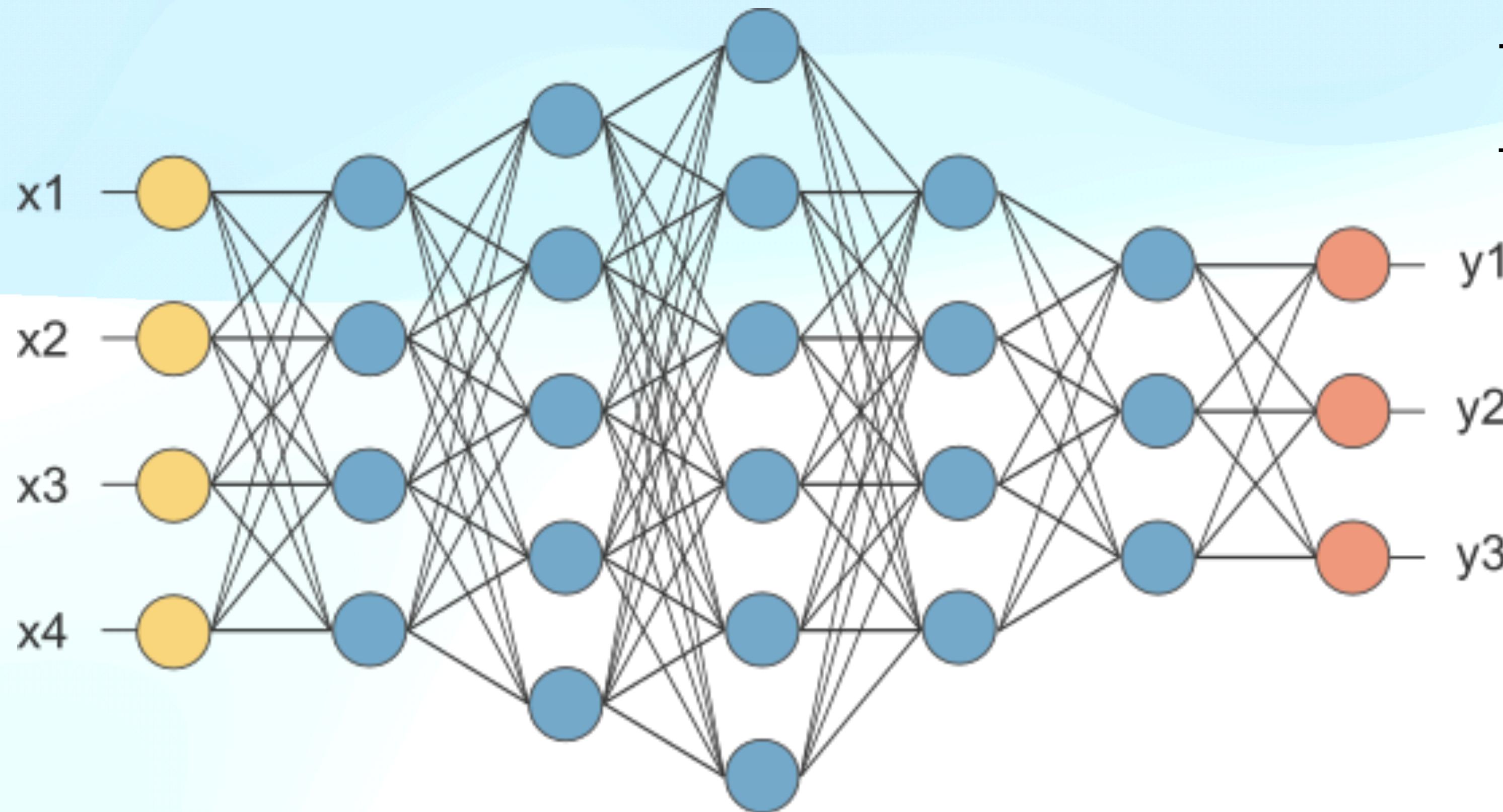
Attention and Transformers

Plan: We trace back history to see how attention and transformers have emerged

1. Basic models, related to transduction models and attention.
2. Encoder-Decoder model, using recurrent networks such as LSTM.
3. Transformer models are general models sufficient for almost all biotech applications (graph models may be treated to be special cases too).
4. For example, DeepMind AlphaFold2 uses depends on a transformer architecture to train an end-to-end model.
5. The transformer model also makes it easy for large scale biological data (pre)training.

Attention and Transformers

1. Fully connected network, feedforward network

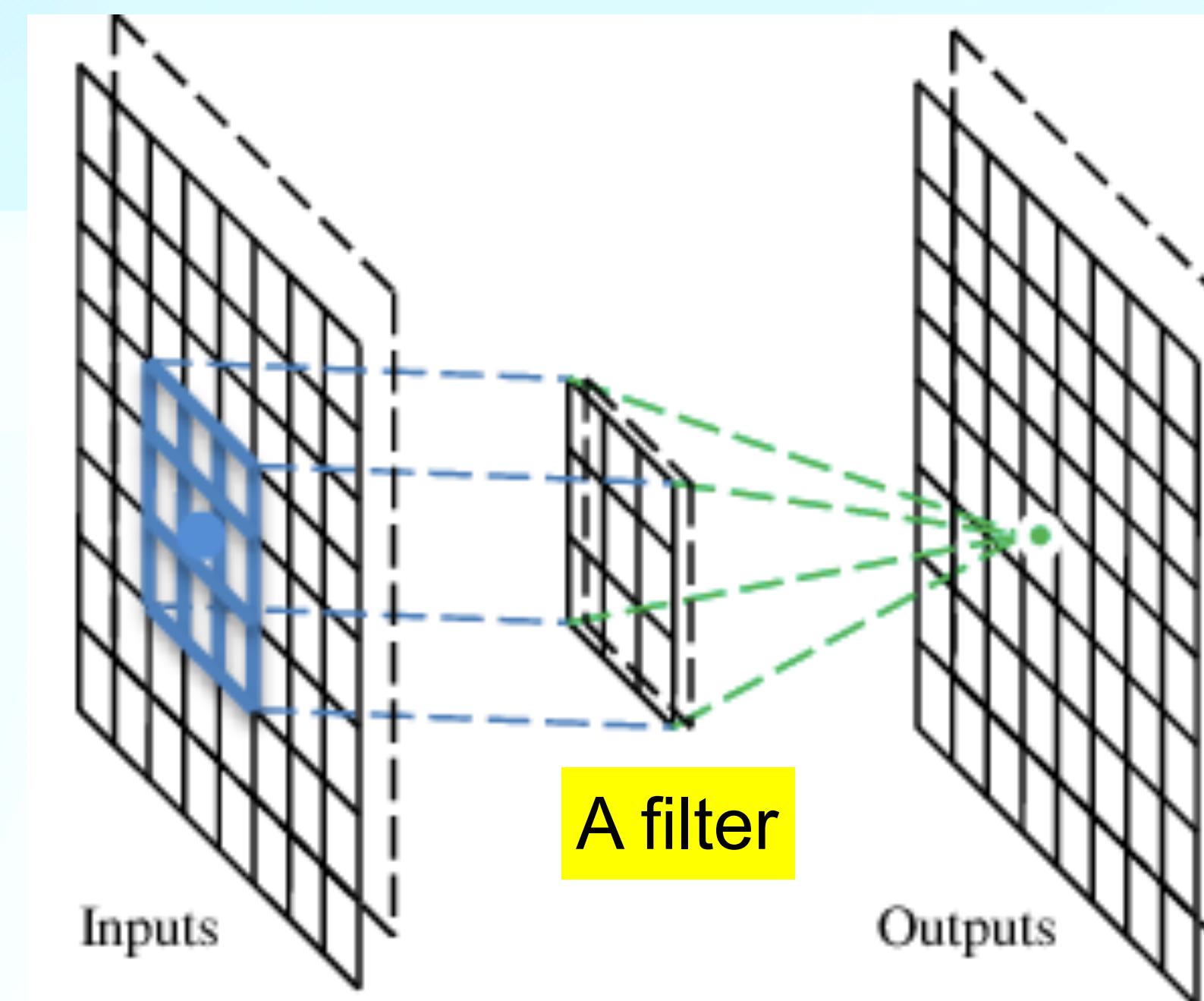


To learn the weights on
the edges

Attention and Transformers

2. CNN

A CNN is a neural network with some convolutional layers (and some other layers). A convolutional layer has a number of filters that do convolutional operation.



Attention and Transformers

Convolutional layer

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

Input

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

:

Each filter detects a small pattern (3 x 3).

Convolution Operation

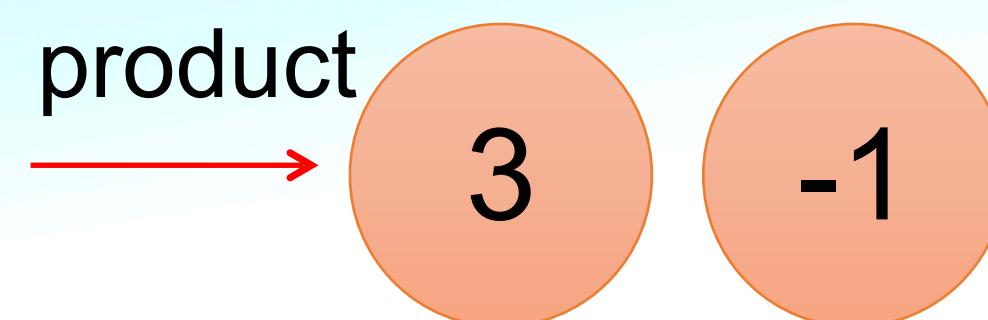
1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

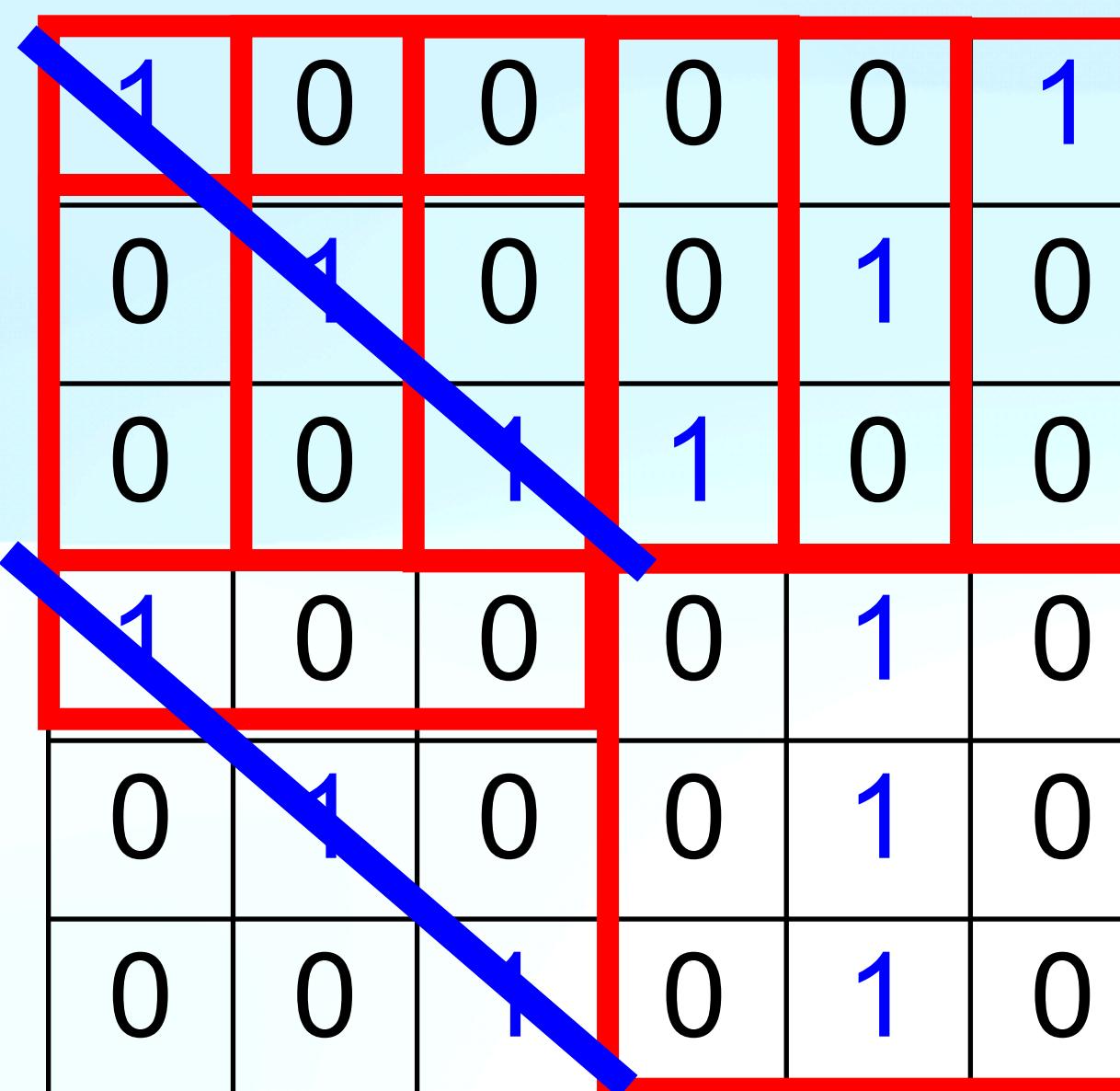
Dot
product



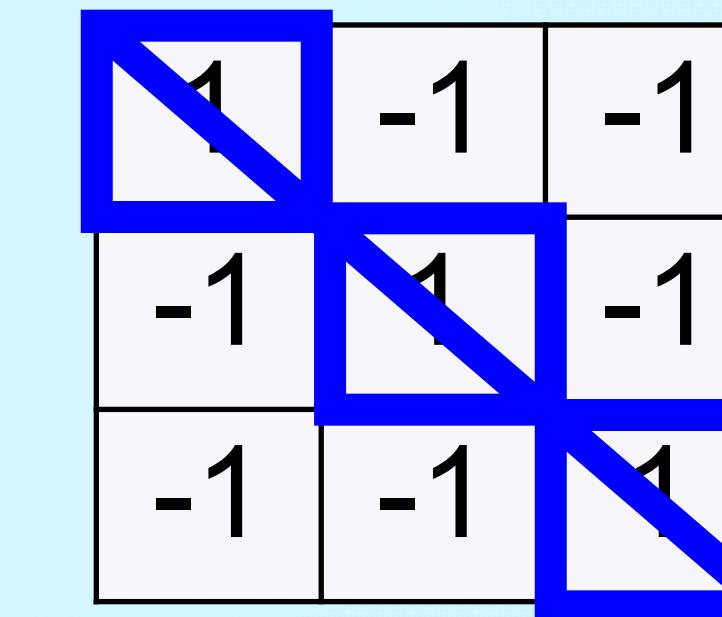
Input

Convolution

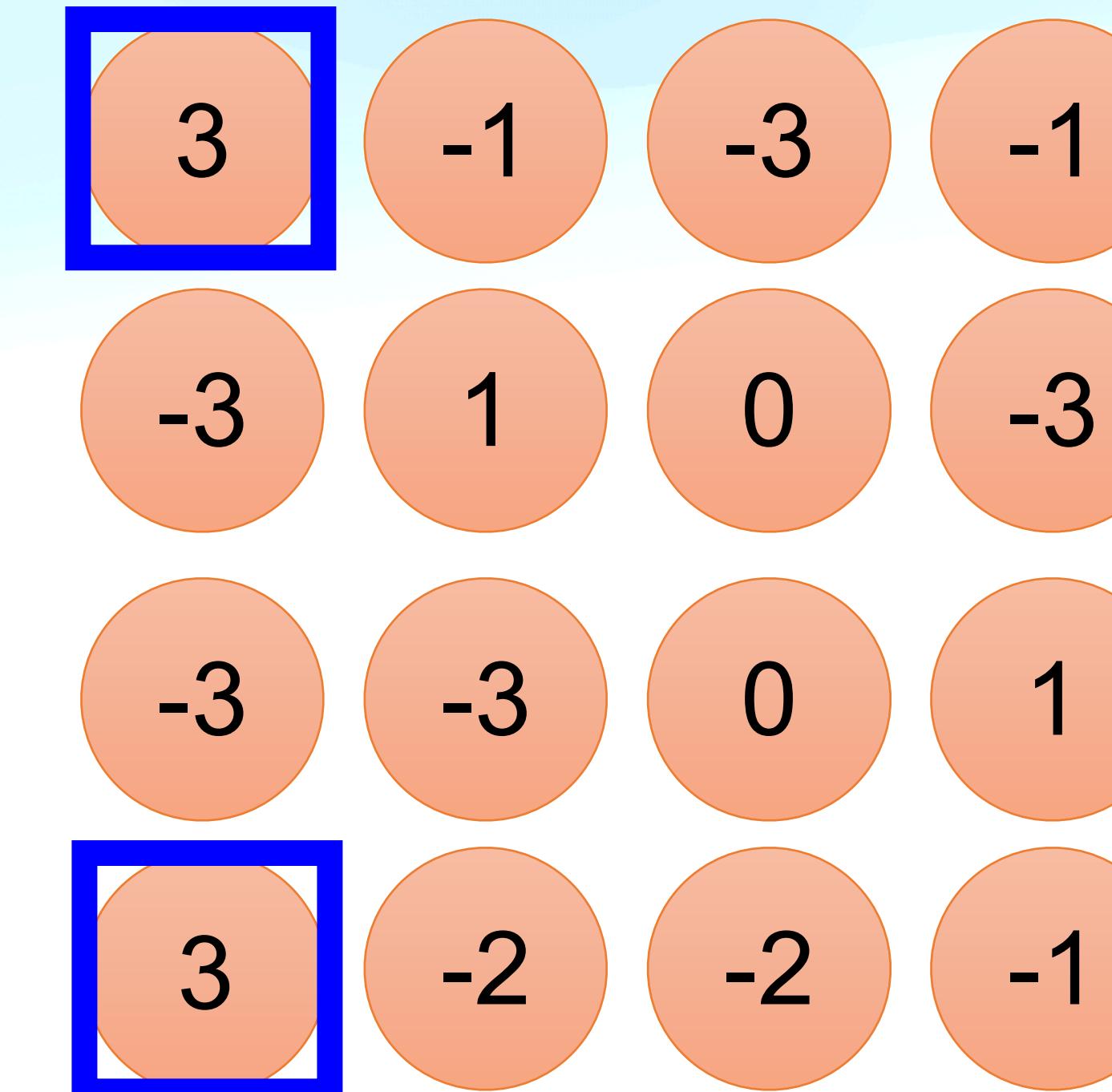
stride=1



Input

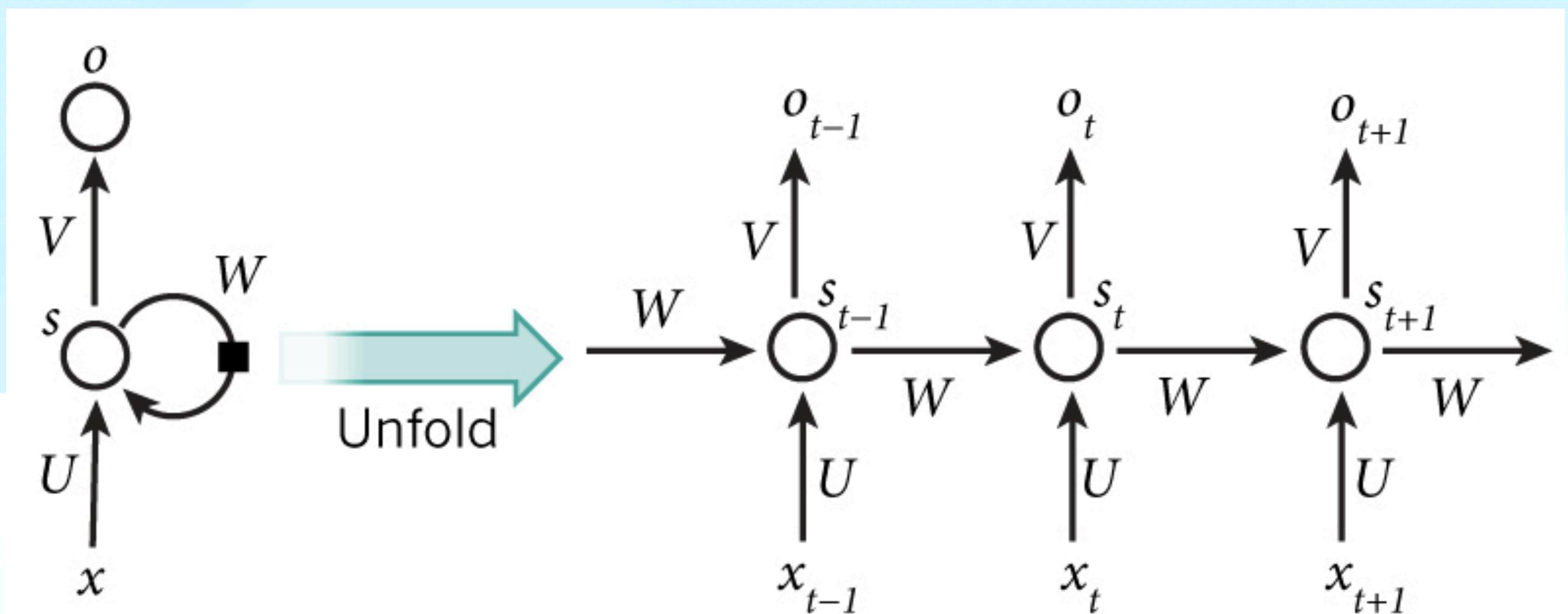


Filter 1



Attention and Transformers

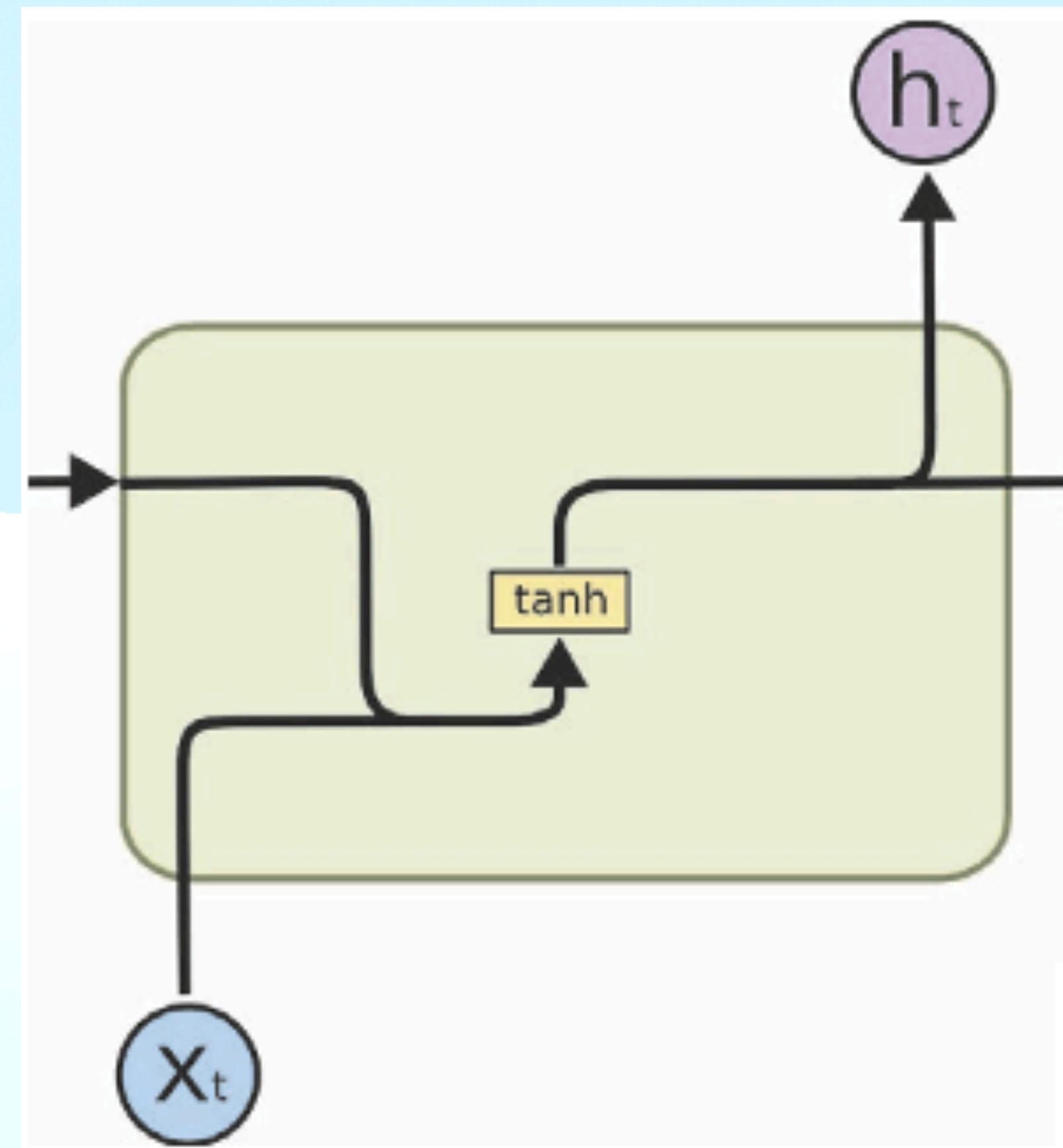
3. RNN



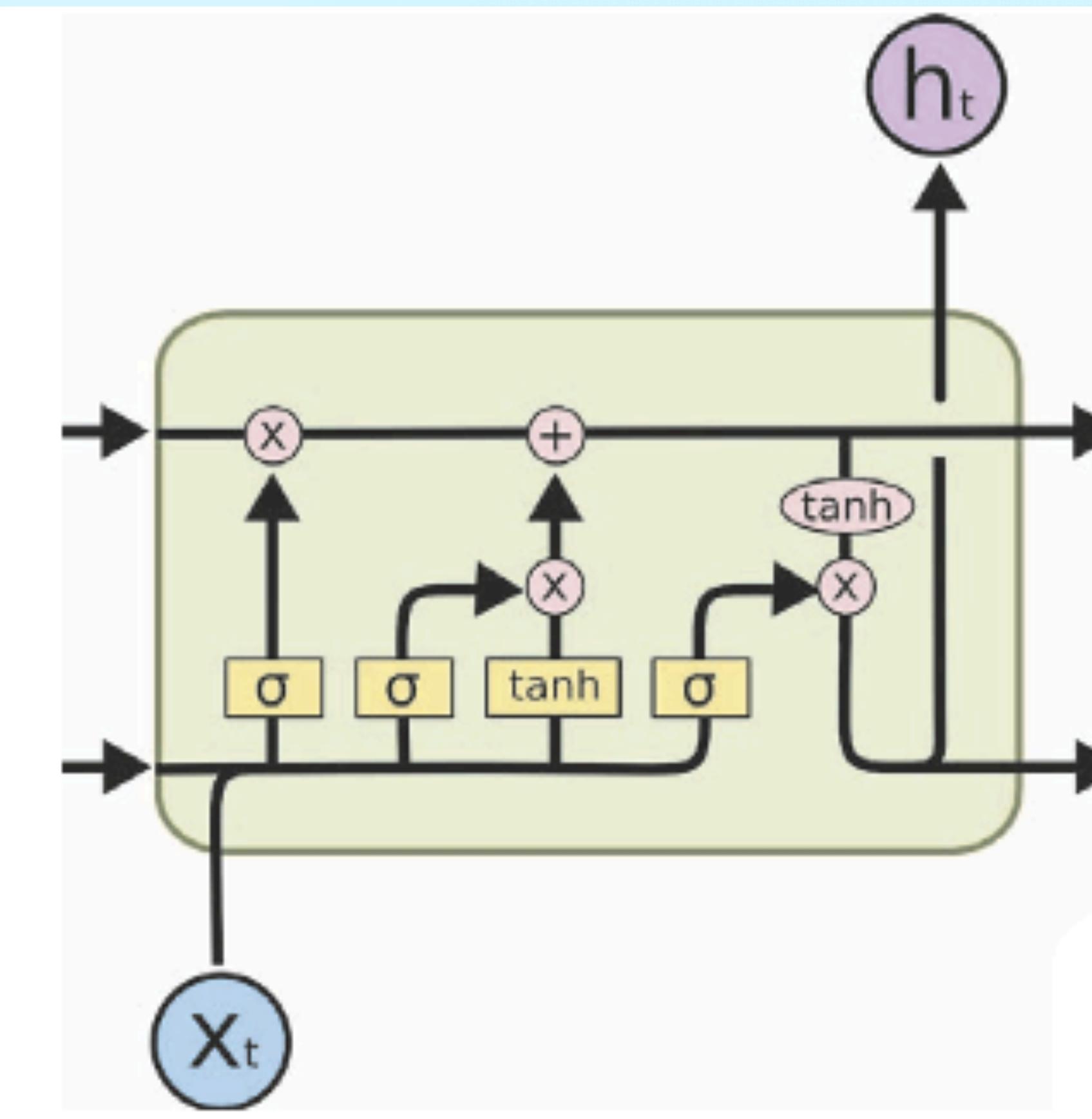
Parameters to be learned:
 U, V, W

Attention and Transformers

Simple RNN vs LSTM



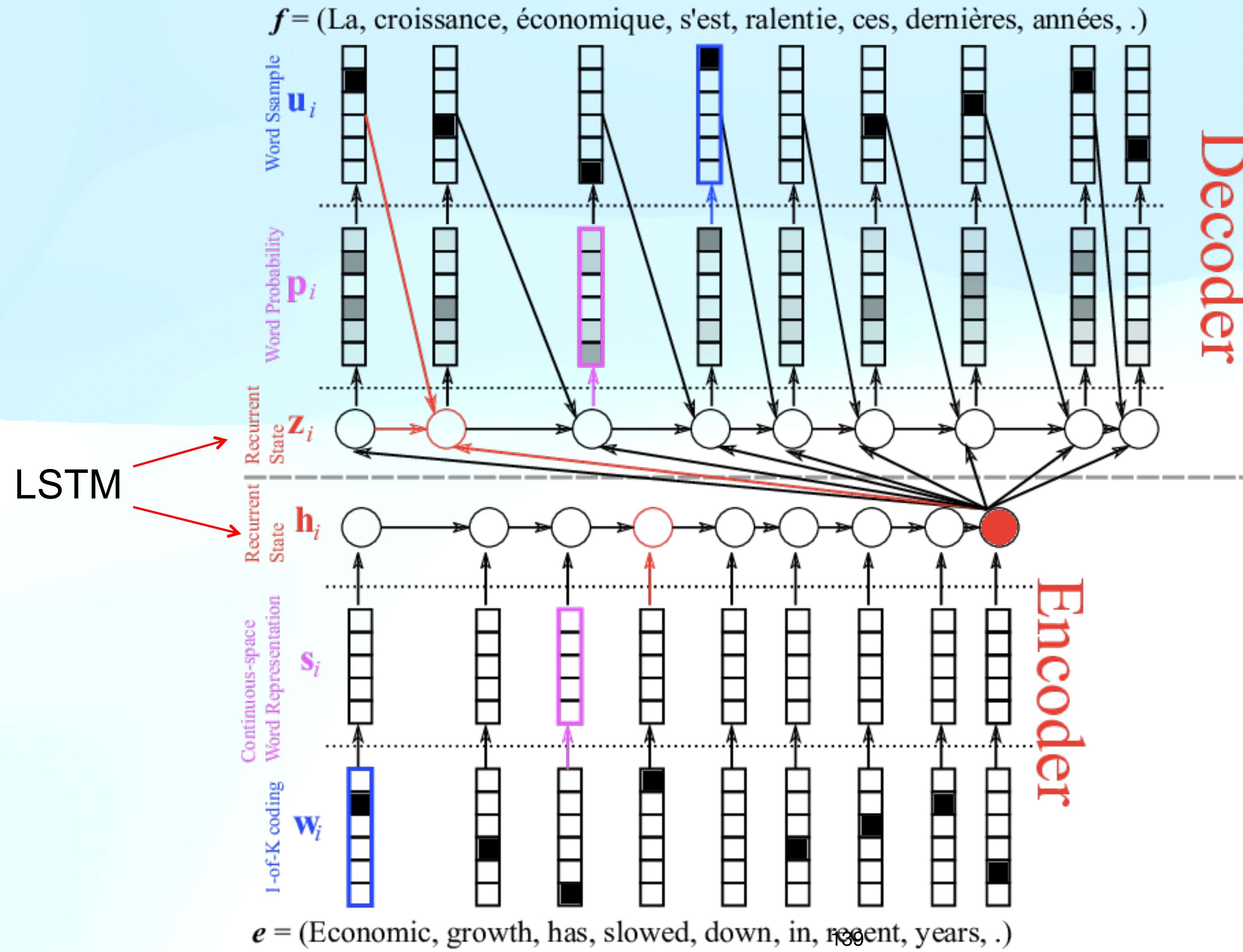
(a) RNN



(b) LSTM

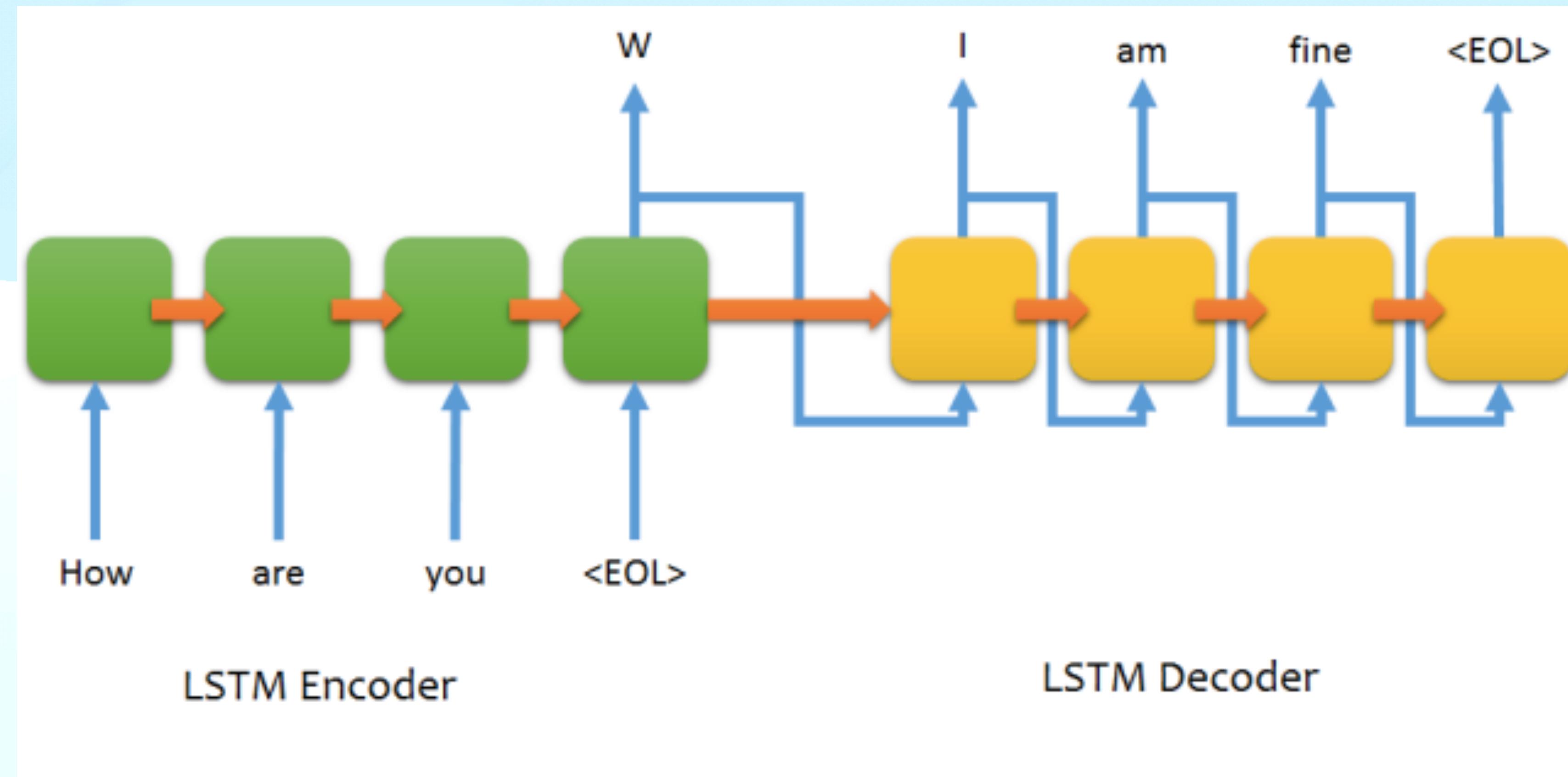
Attention and Transformers

Encoder-Decoder machine translation



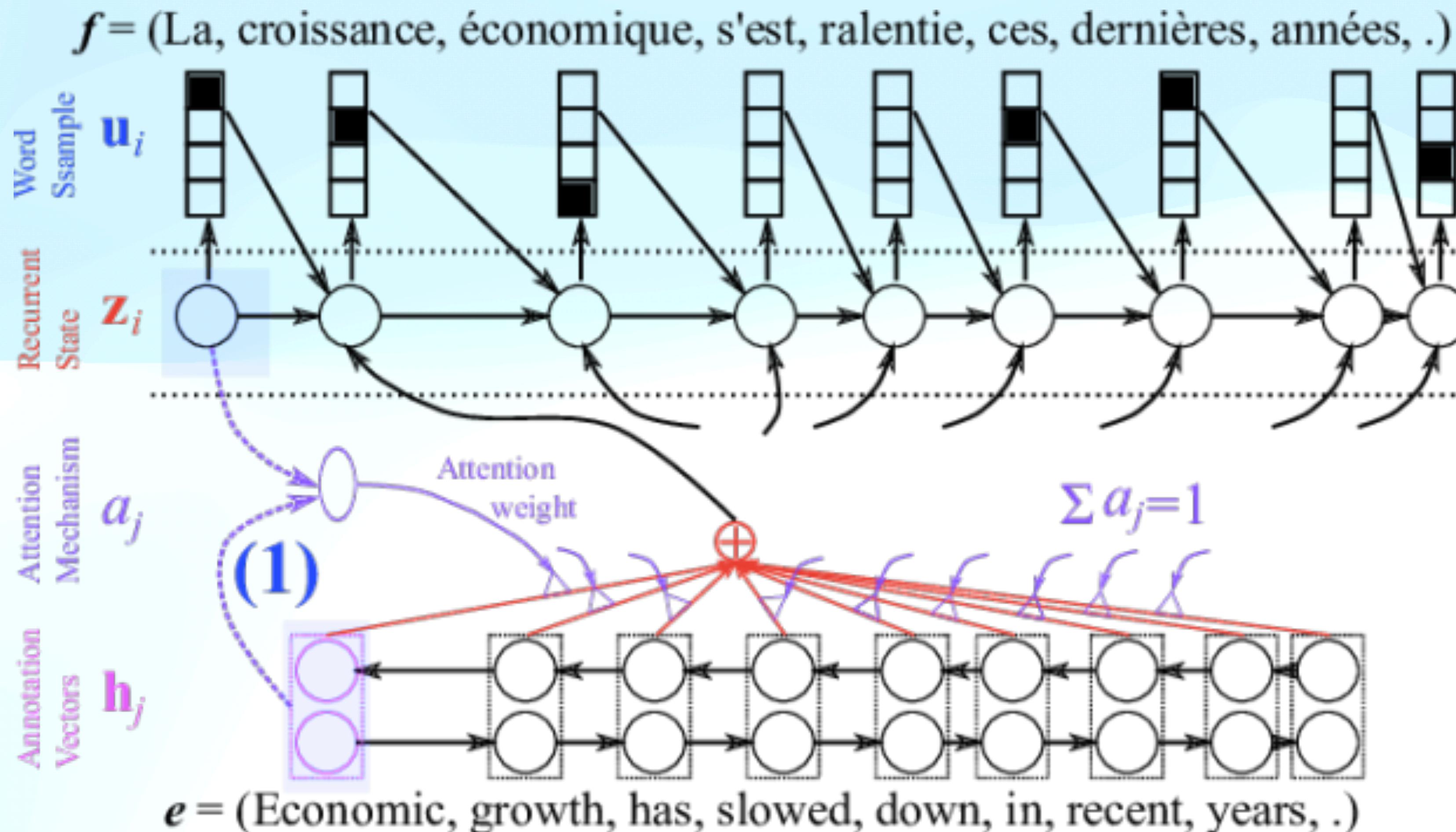
Attention and Transformers

Encoder-Decoder LSTM structure for chatting



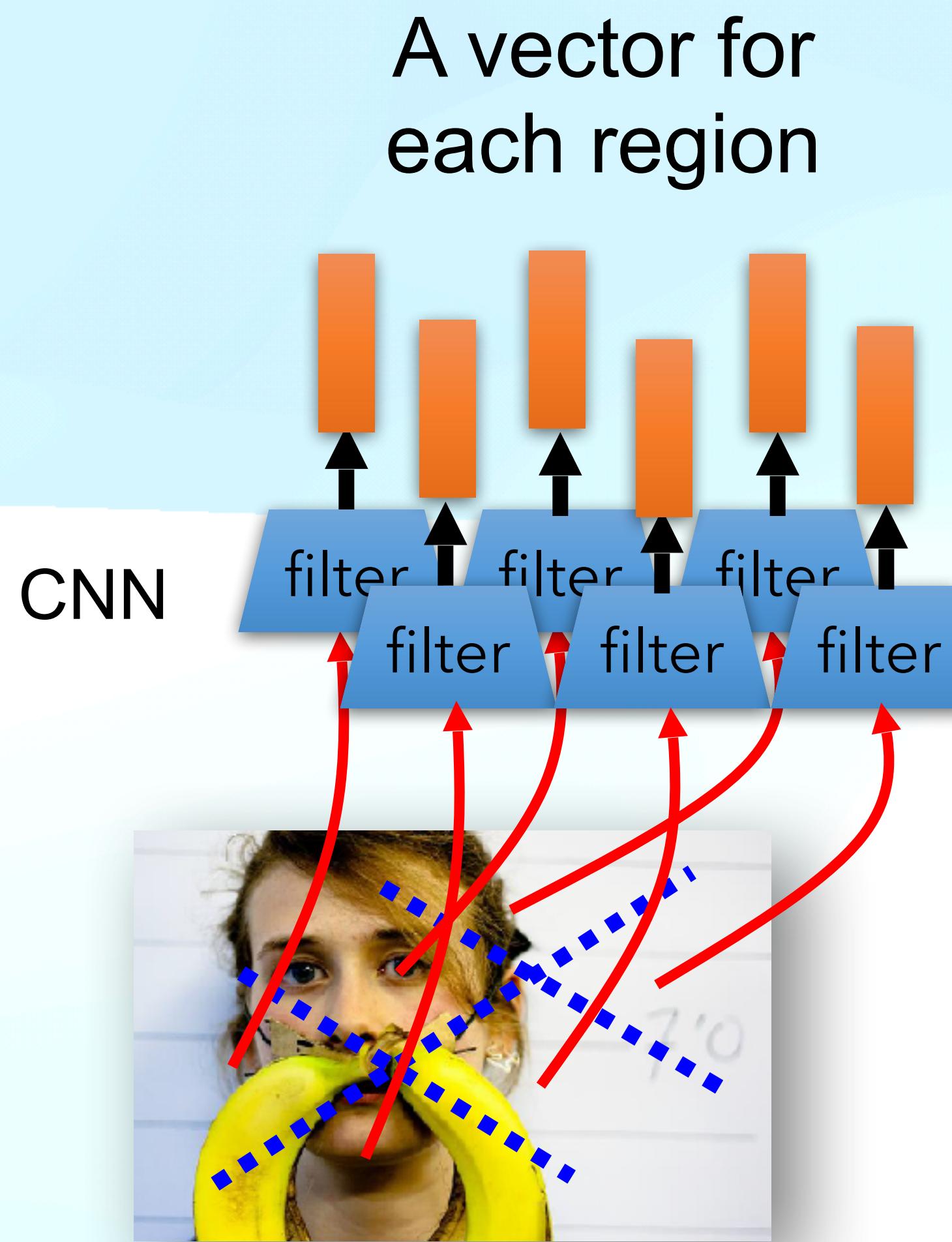
Attention and Transformers

Attention

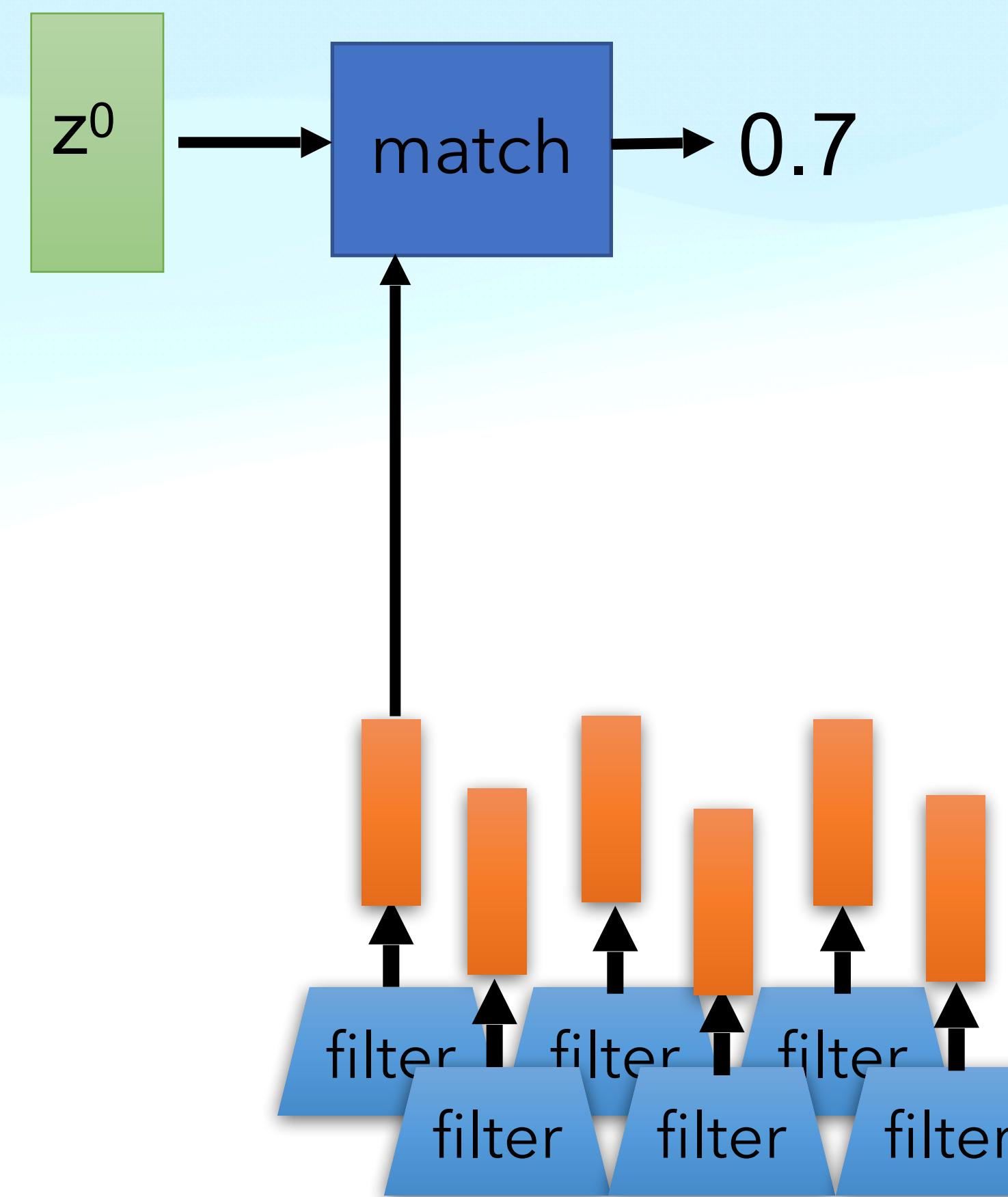


Attention and Transformers

Image caption generation using attention

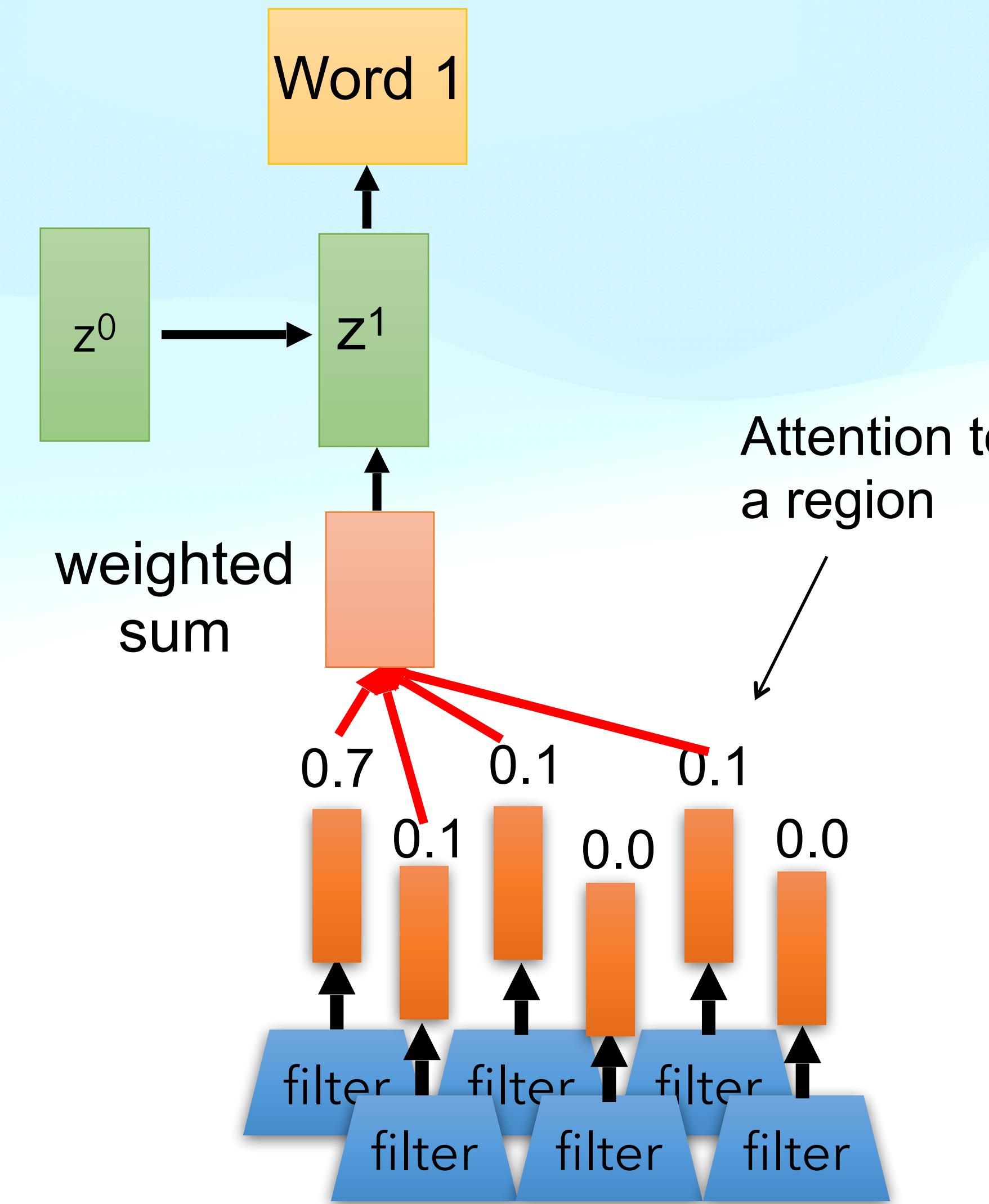
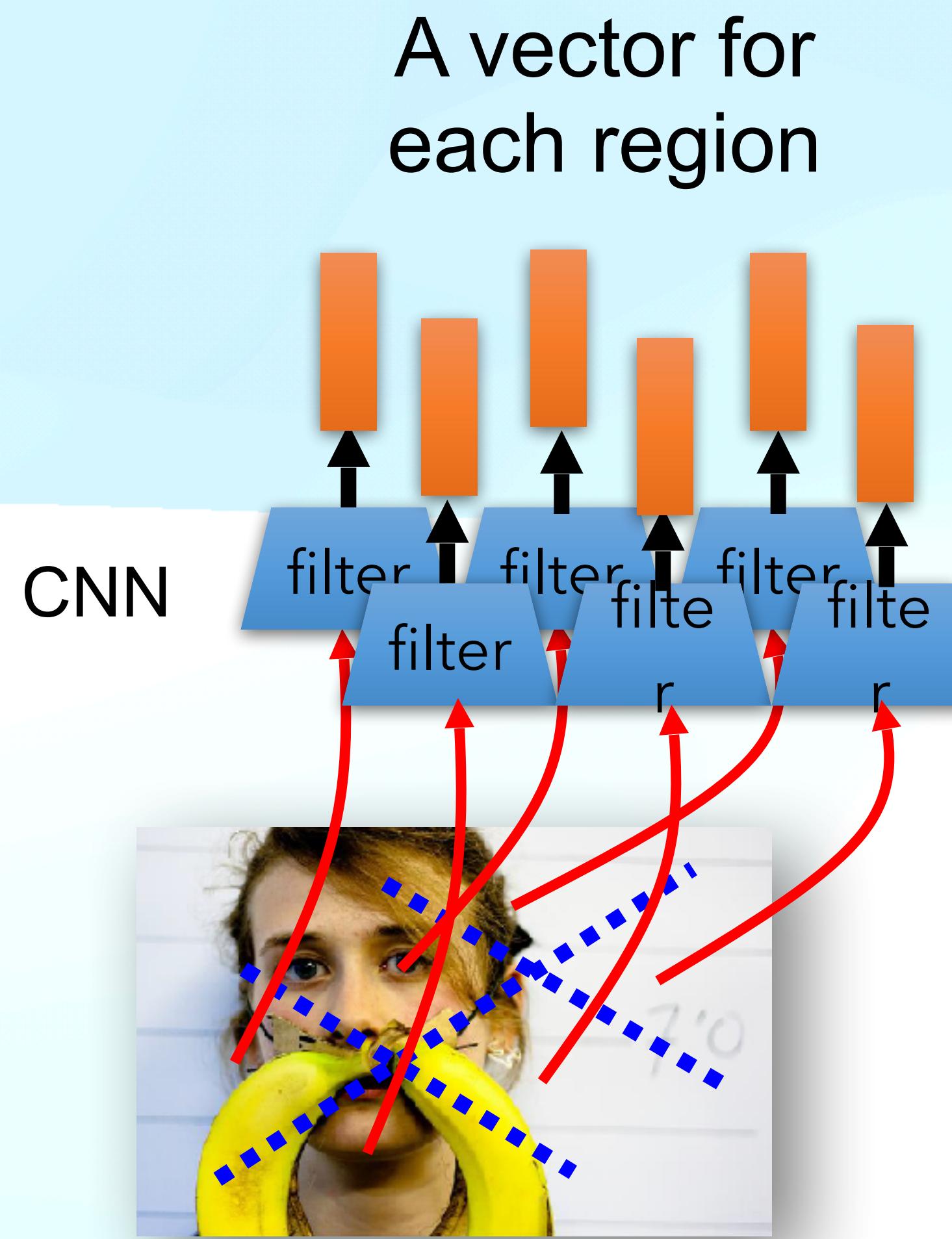


z^0 is initial parameter, it is also learned



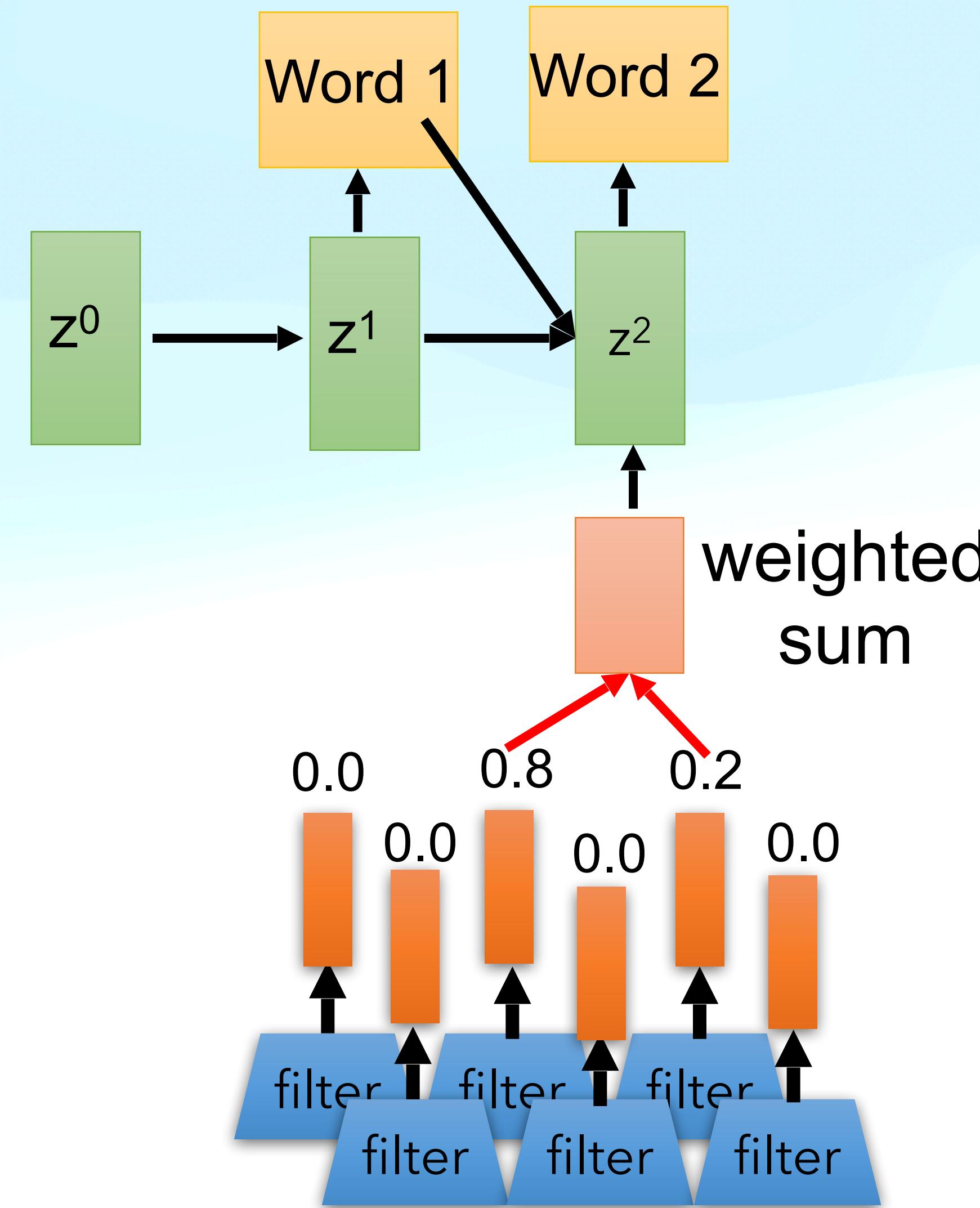
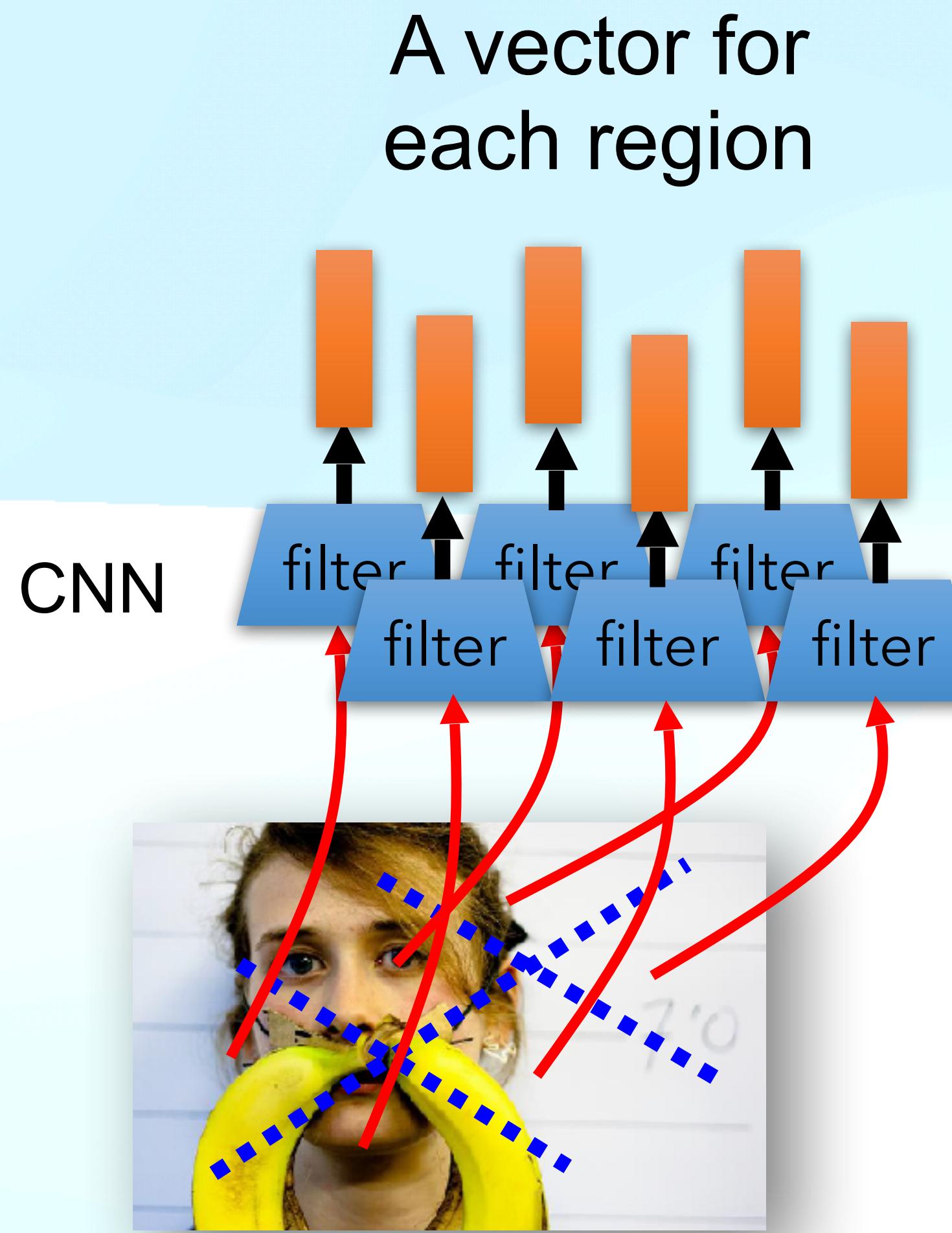
Attention and Transformers

Image caption generation using attention



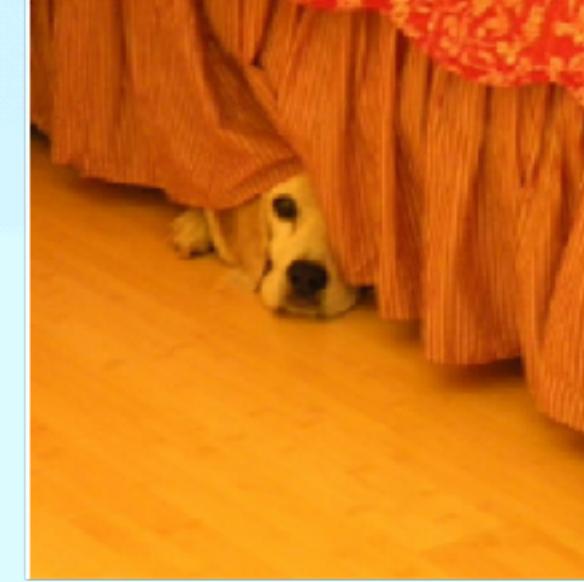
Attention and Transformers

Image caption generation using attention



Attention and Transformers

Image caption generation using attention

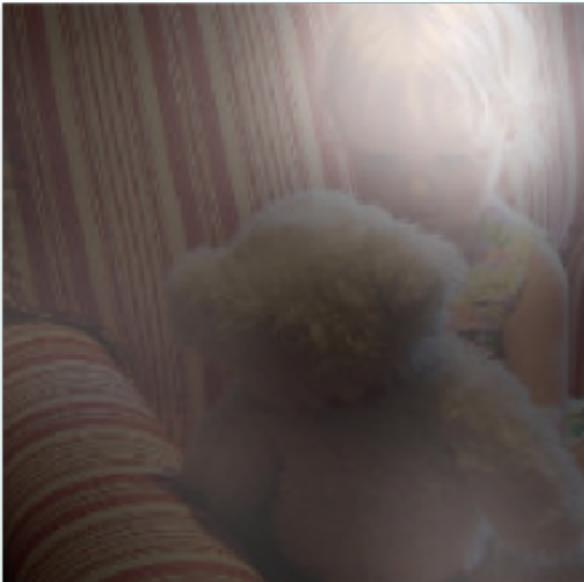


A woman is throwing a frisbee in a park.

A dog is standing on a hardwood floor.



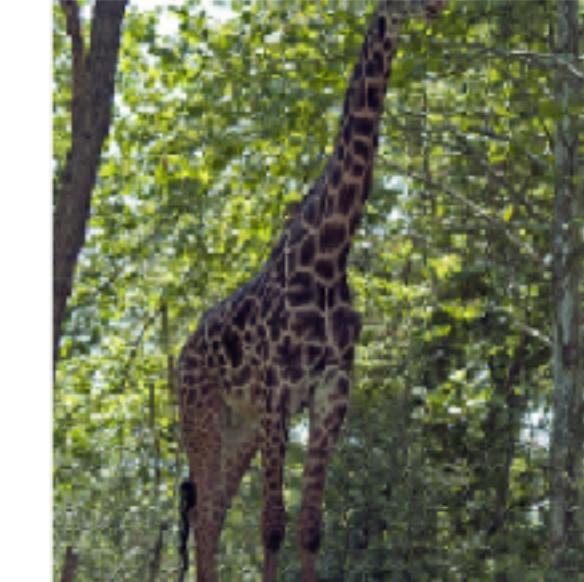
A stop sign is on a road with a mountain in the background.



A little girl sitting on a bed with a teddy bear.



A group of people sitting on a boat in the water.



A giraffe standing in a forest with trees in the background.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, Yoshua Bengio,
"Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML, 2015

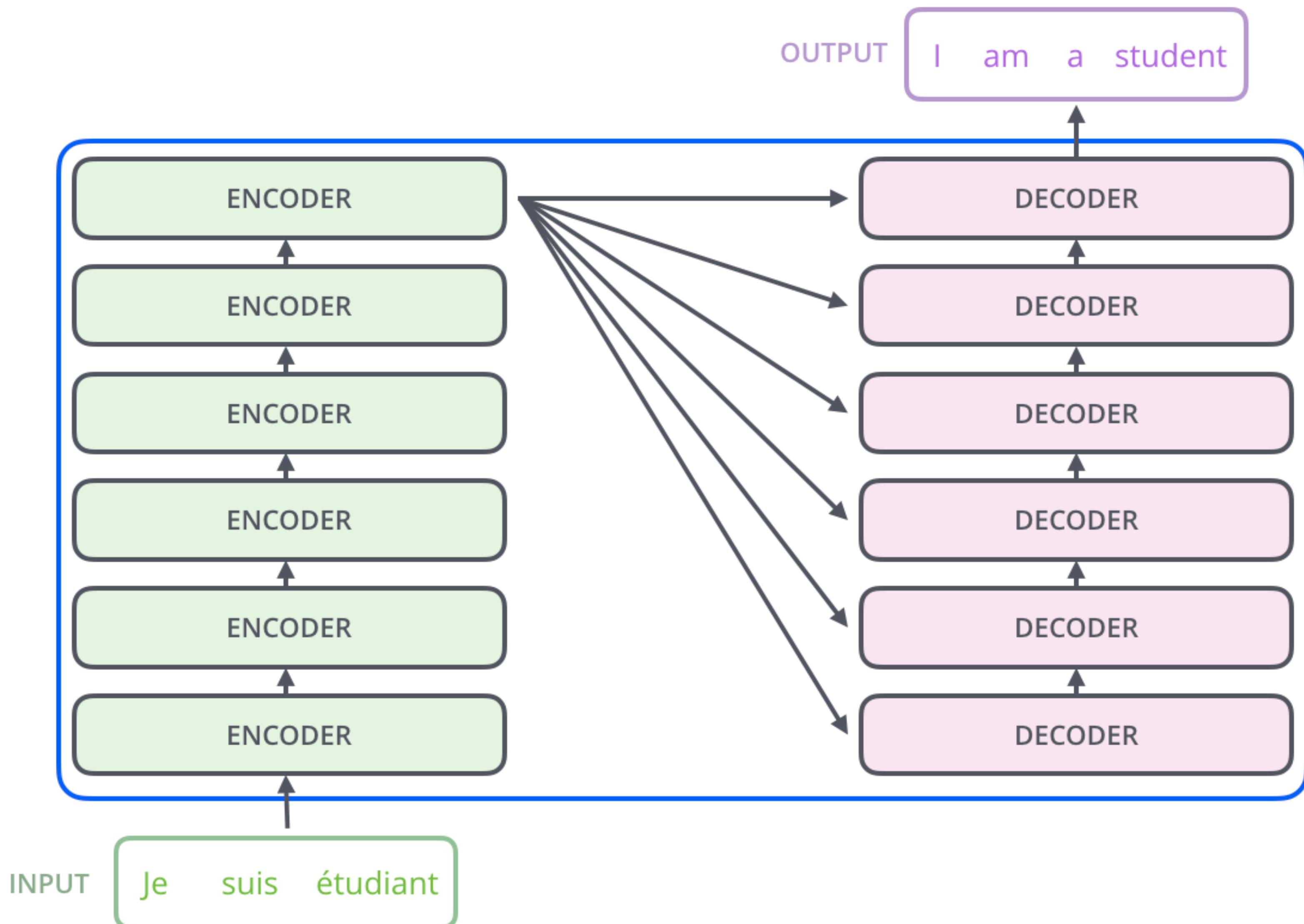
Attention and Transformers

More new ideas:

1. ULM-FiT, pre-training, transfer learning in NLP
2. Recurrent models require linear sequential computation, hard to parallelize.
ELMo, bidirectional LSTM.
3. In order to reduce such sequential computation, several models based on CNN are introduced, such as ConvS2S and ByteNet. Dependency for ConvS2S needs linear depth, and ByteNet logarithmic.
4. The transformer is the first transduction model relying entirely on self-attention to compute the representations of its input and output without using RNN or CNN.

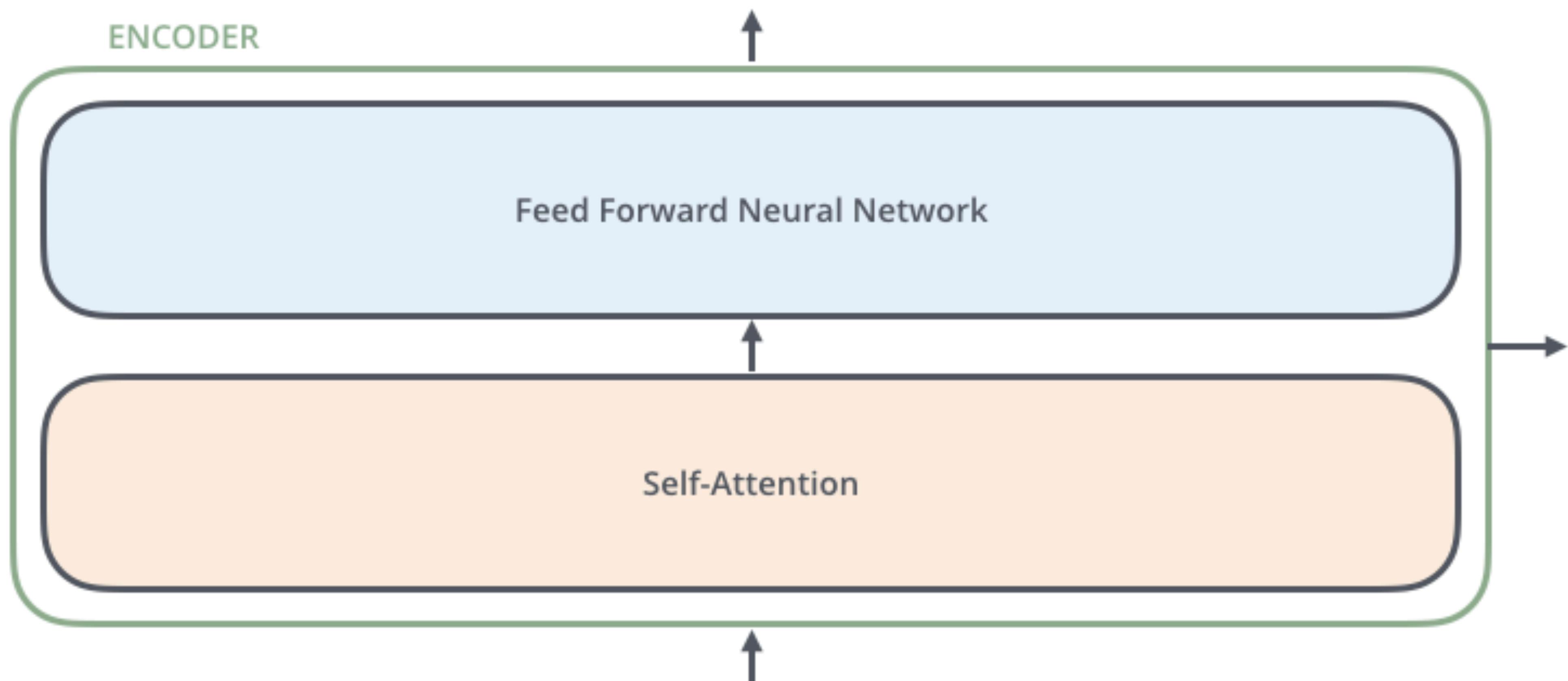
Attention and Transformers

Transformer



Attention and Transformers

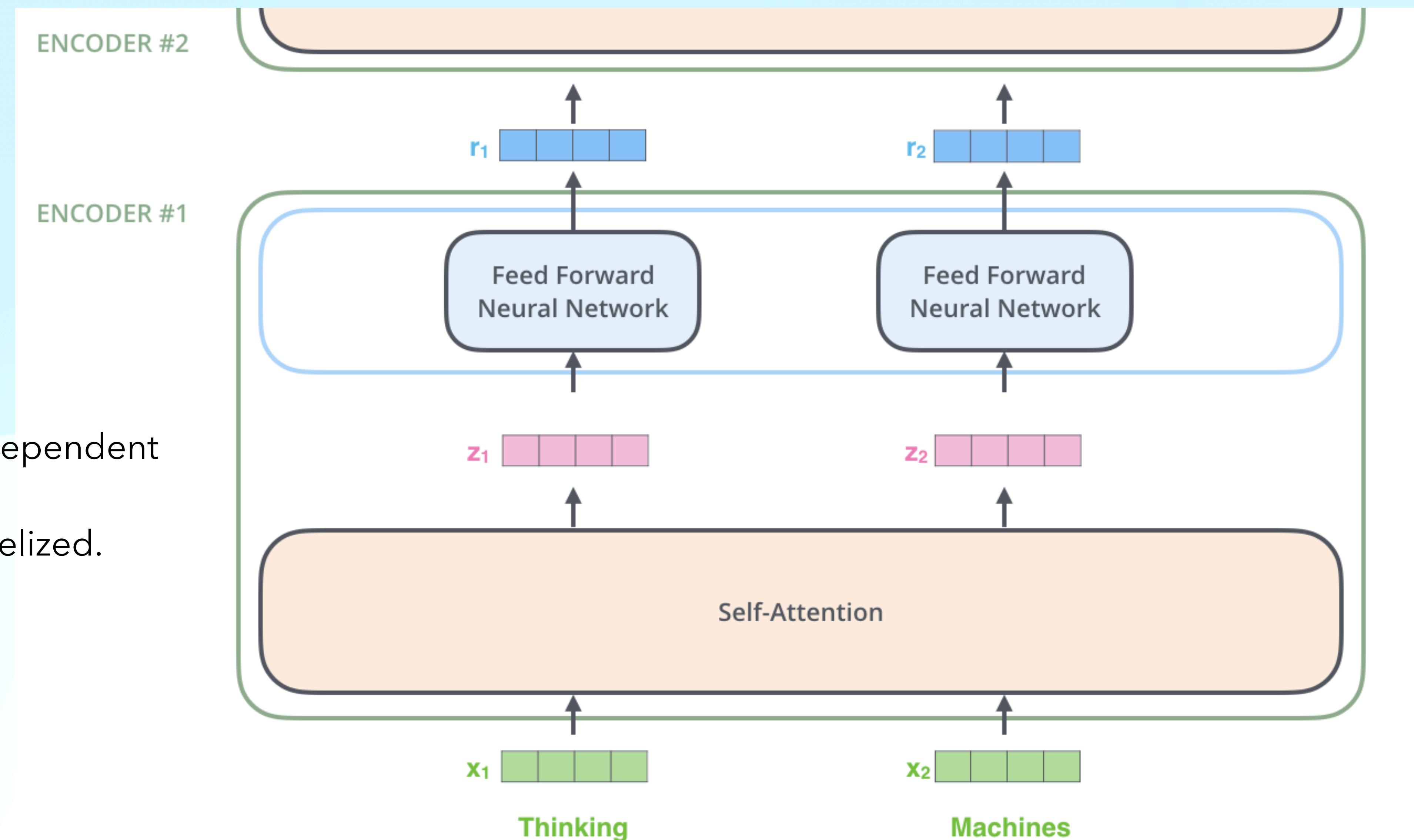
An Encoder Block: same structure, different parameters



Attention and Transformers

Encoder

Note: The ffnn is independent for each word.
Hence can be parallelized.



Attention and Transformers

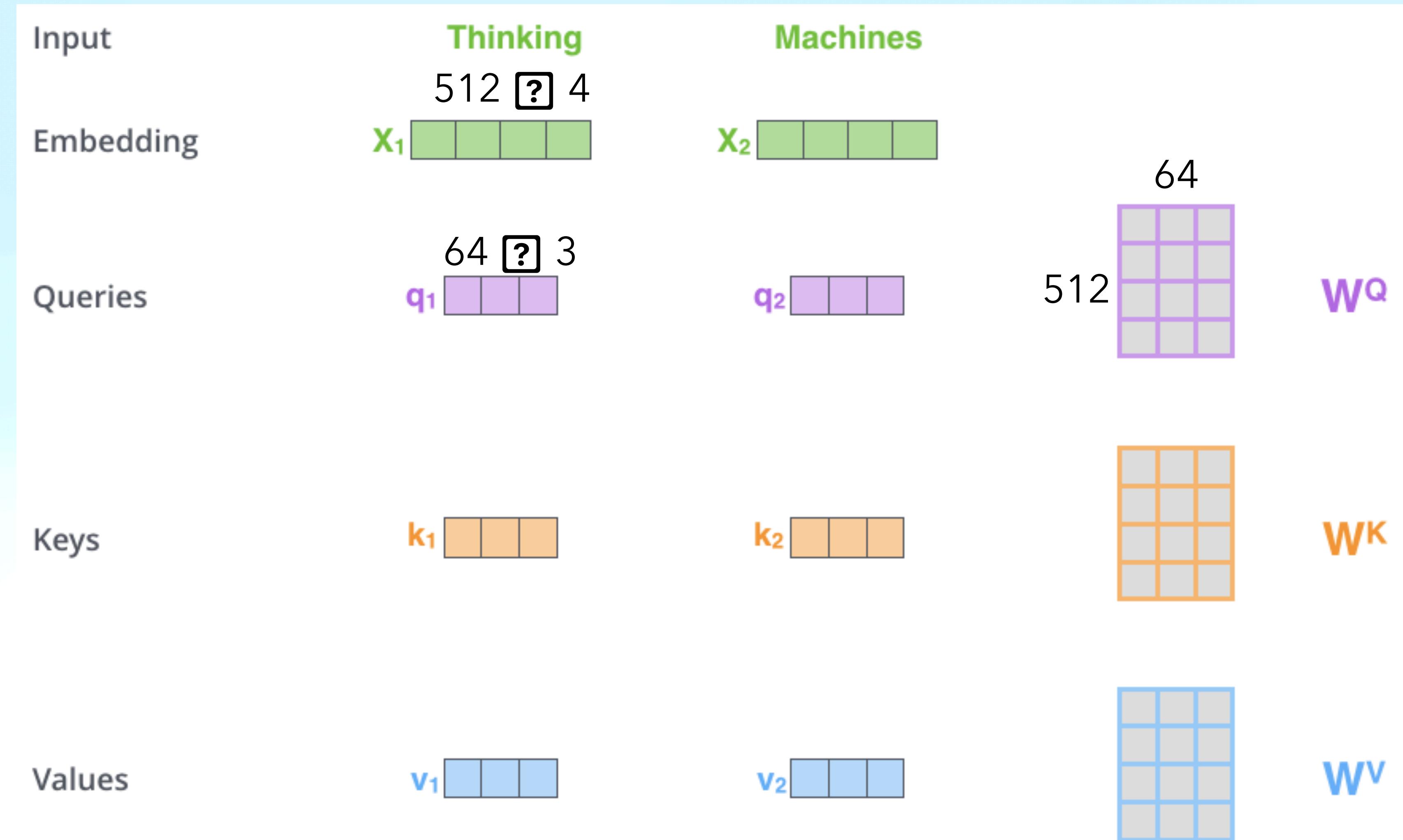
Self Attention

First we create three vectors by multiplying input embedding (1×512) x_i with three matrices (64×512):

$$q_i = x_i W^Q$$

$$K_i = x_i W^K$$

$$V_i = x_i W^V$$



Attention and Transformers

Self Attention

Now we need to calculate a score to determine how much focus to place on other parts of the input.

Input

Embedding

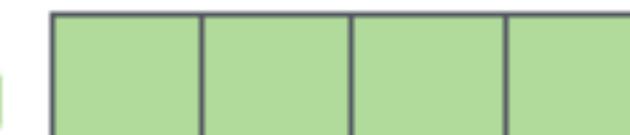
Queries

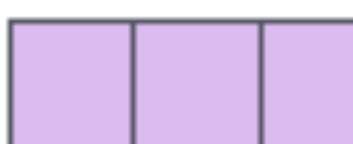
Keys

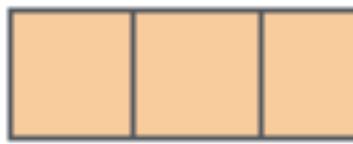
Values

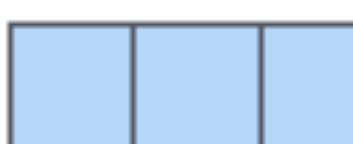
Score

Thinking

x_1 

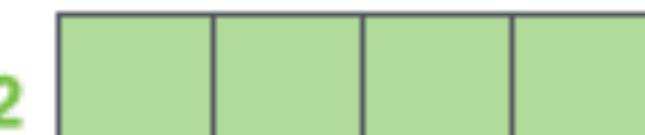
q_1 

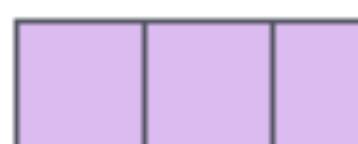
k_1 

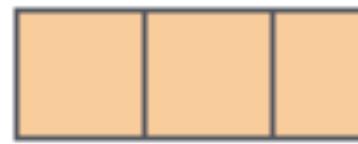
v_1 

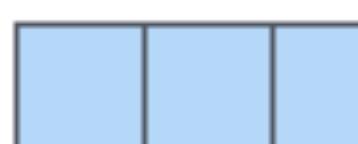
$$q_1 \cdot k_1 = 112$$

Machines

x_2 

q_2 

k_2 

v_2 

$$q_1 \cdot k_2 = 96$$

02 Attention and Transformers

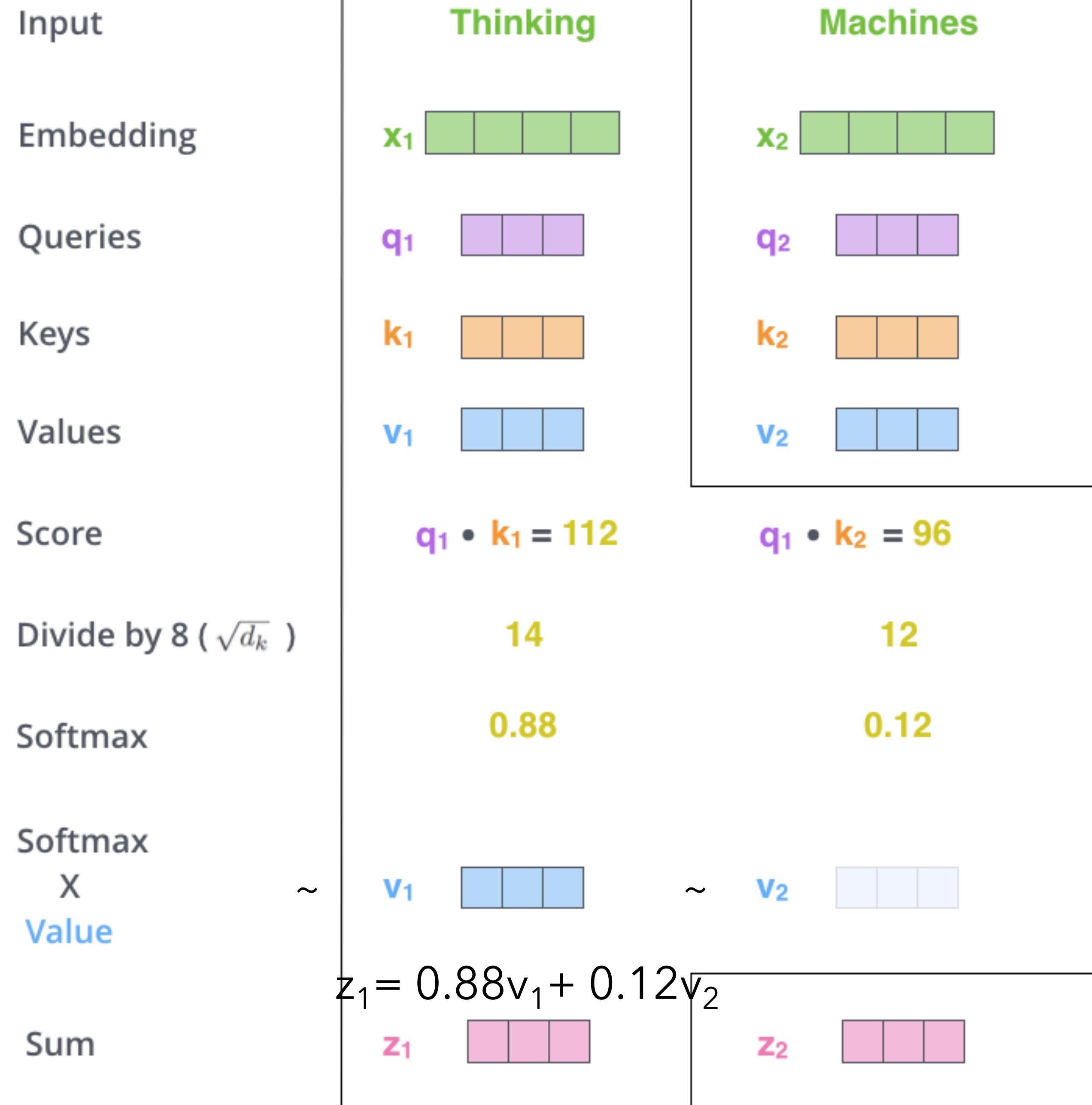
Self Attention

Formula

$$\text{softmax} \left(\frac{\begin{matrix} Q & K^T \\ \begin{matrix} \text{---} \\ \times \end{matrix} & \begin{matrix} \text{---} \\ \sqrt{d_k} \end{matrix} \end{matrix}}{V} \right) = Z$$

64x64 64x512

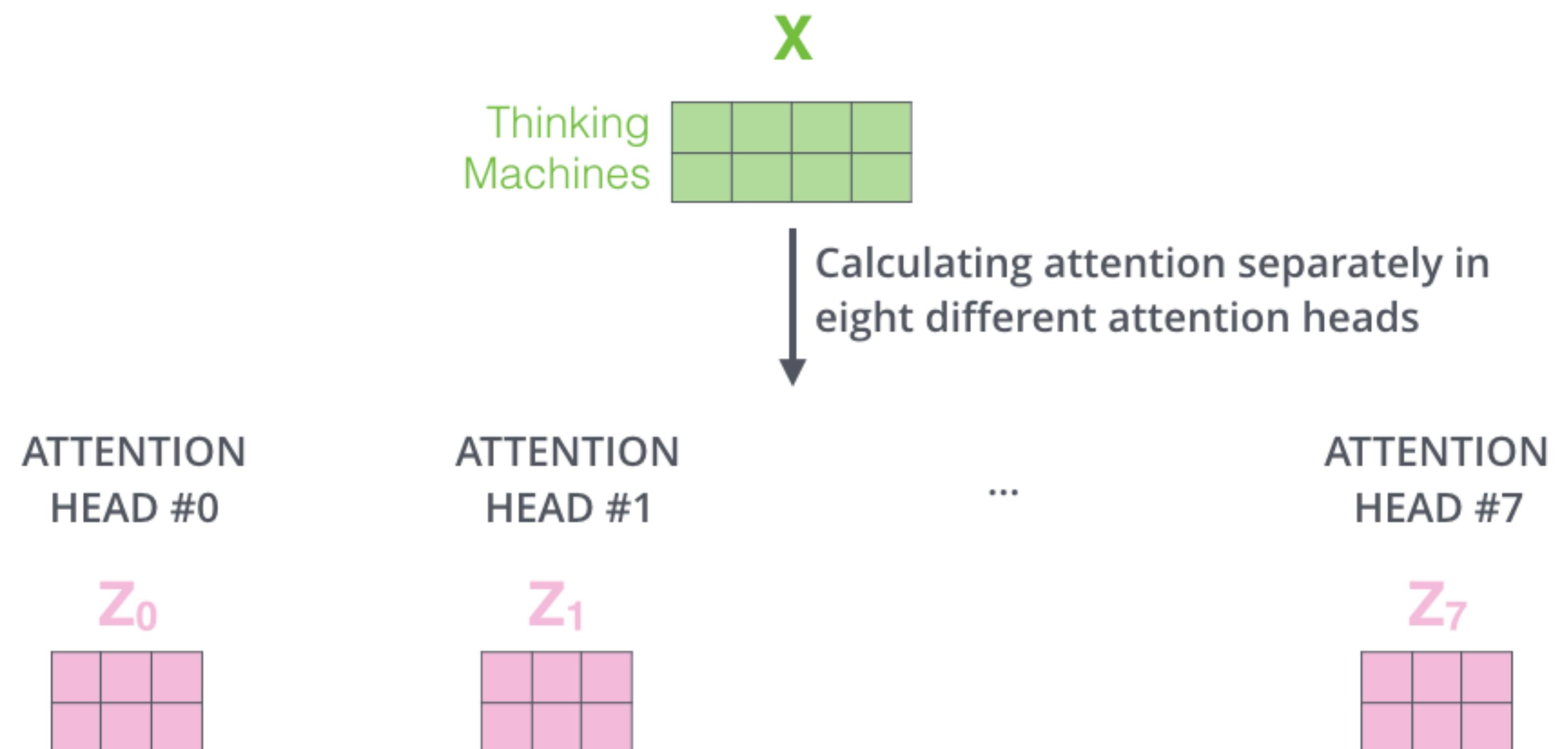
$d_k=64$ is dimension of key vector



Attention and Transformers

Multiple heads

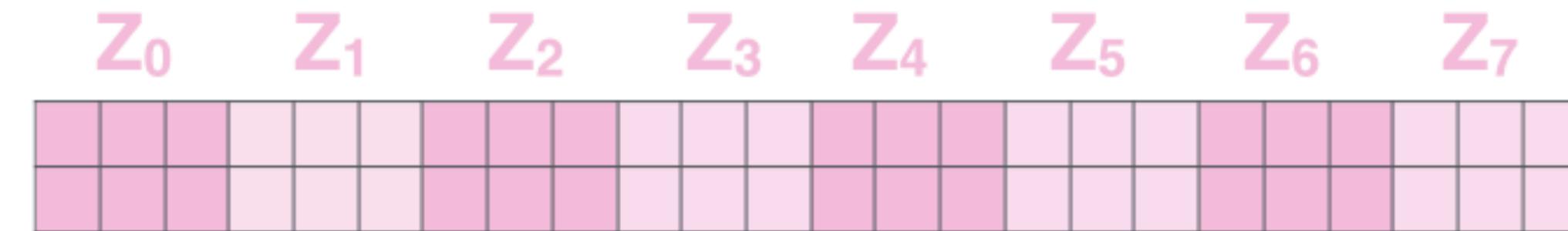
1. It expands the model's ability to focus on different positions.
2. It gives the attention layer multiple "representation subspaces"



Attention and Transformers

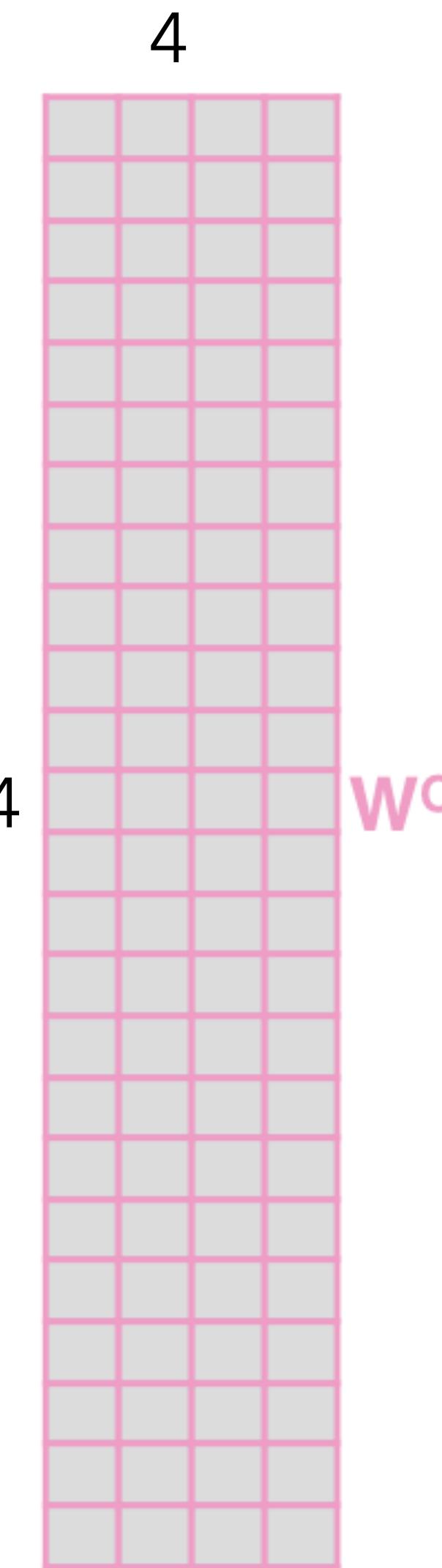
The output
is expecting
only a 2×4
matrix,
hence,

1) Concatenate all the attention heads



2) Multiply with a weight
matrix W^o that was trained
jointly with the model

x



3) The result would be the Z matrix that captures information
from all the attention heads. We can send this forward to the FFNN

$$= \begin{matrix} Z \\ \begin{matrix} \text{---} & \text{---} & \text{---} & \text{---} \end{matrix} \end{matrix}$$

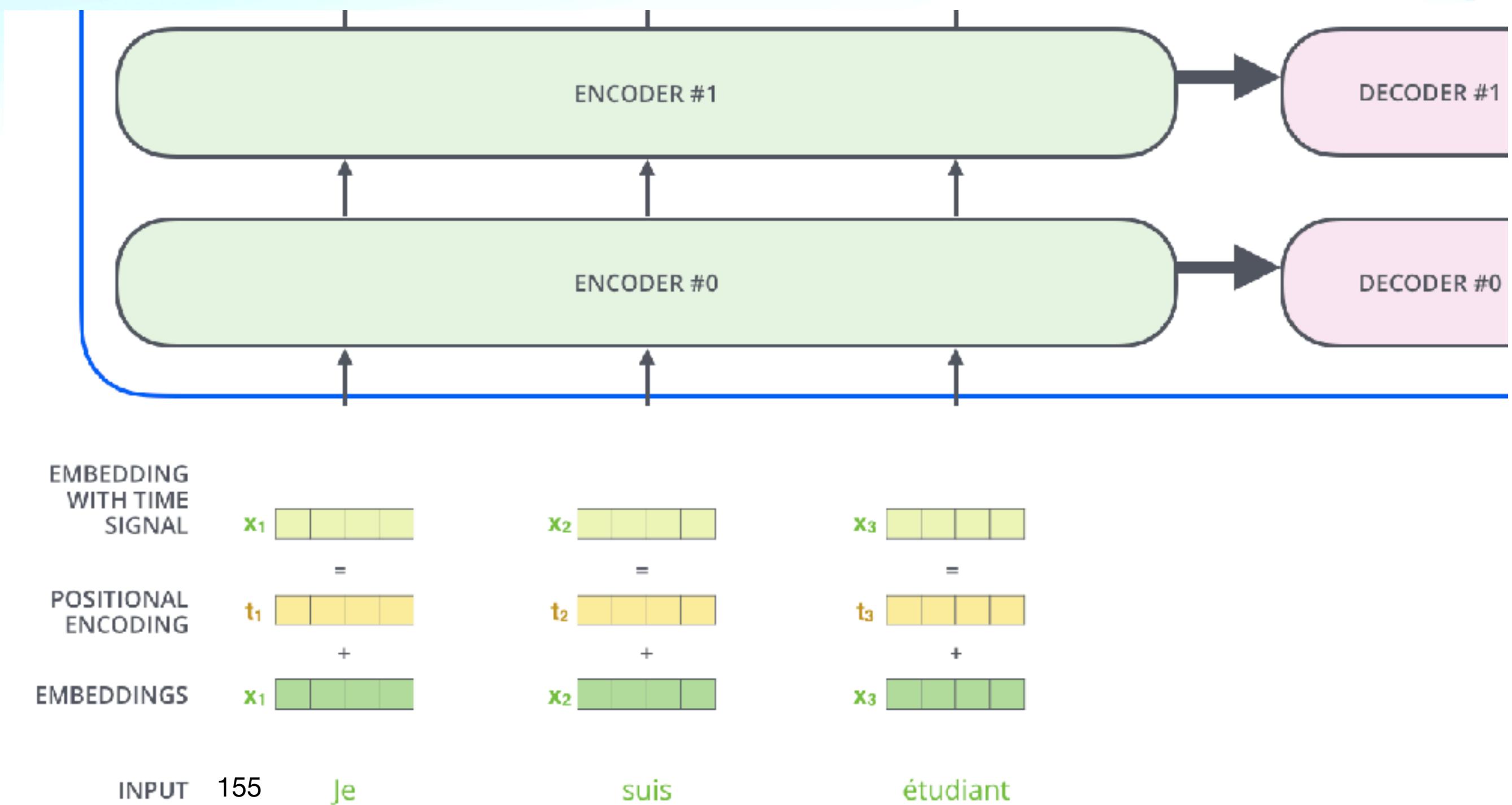
If you want some more intuition on attention: watch <https://www.youtube.com/watch?v=-9vVhYEXeyQ>

Attention and Transformers

Representing the input order (positional encoding)

The transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence. The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.

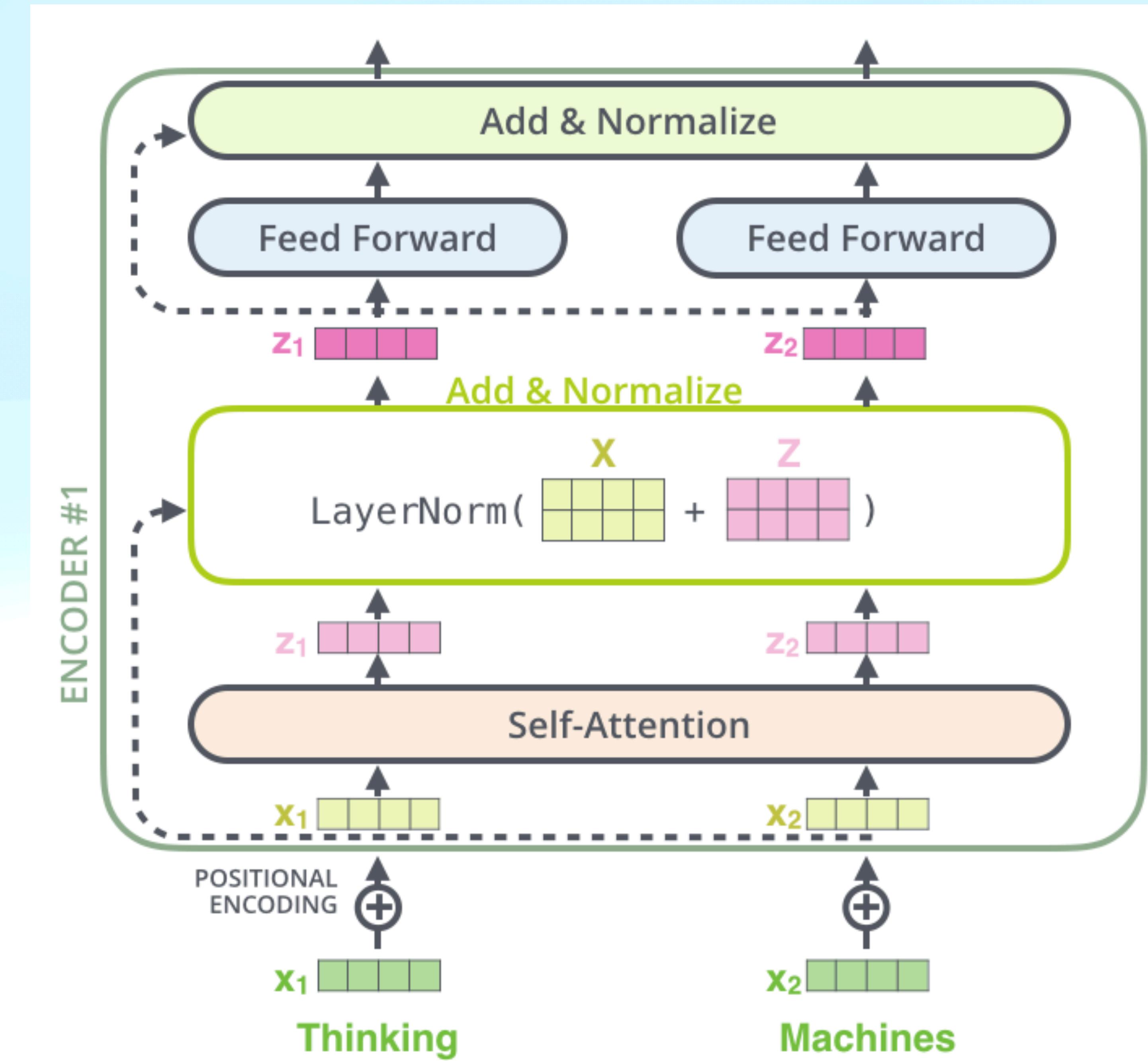
More on positional encoding: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/



Attention and Transformers

Add and Normalize

In order to regulate the computation, this is a normalization layer so that each feature (column) have the same average and deviation.

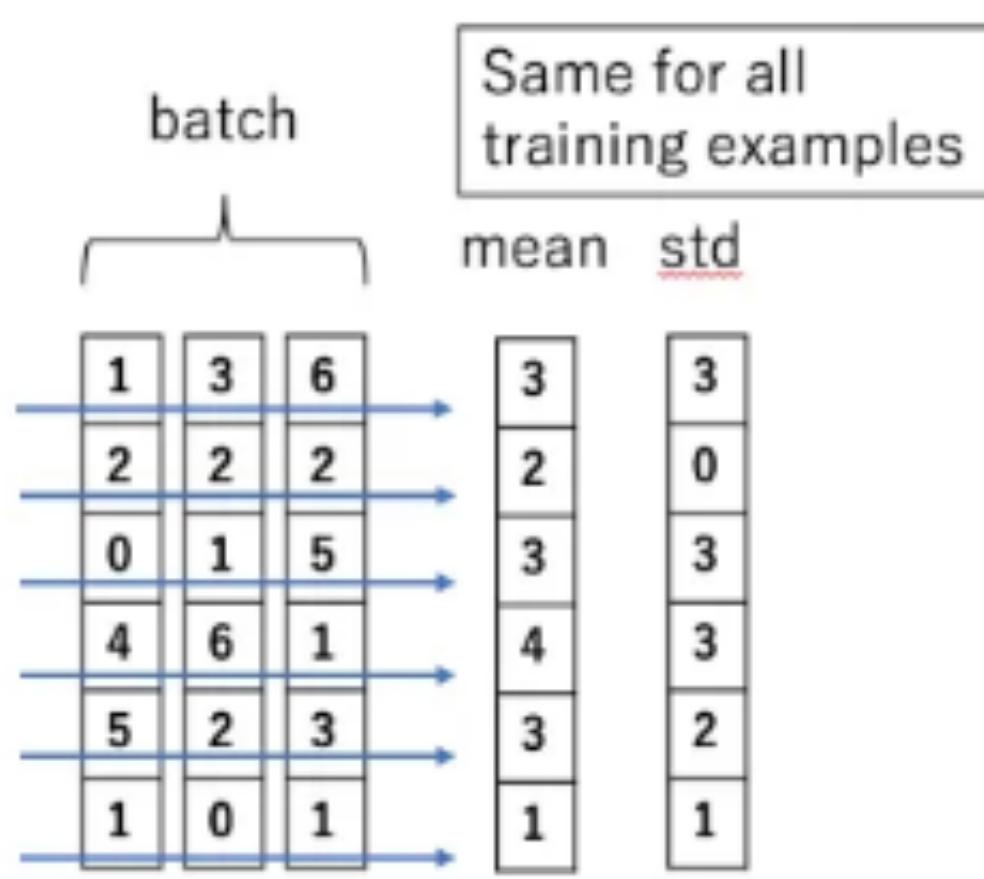


Attention and Transformers

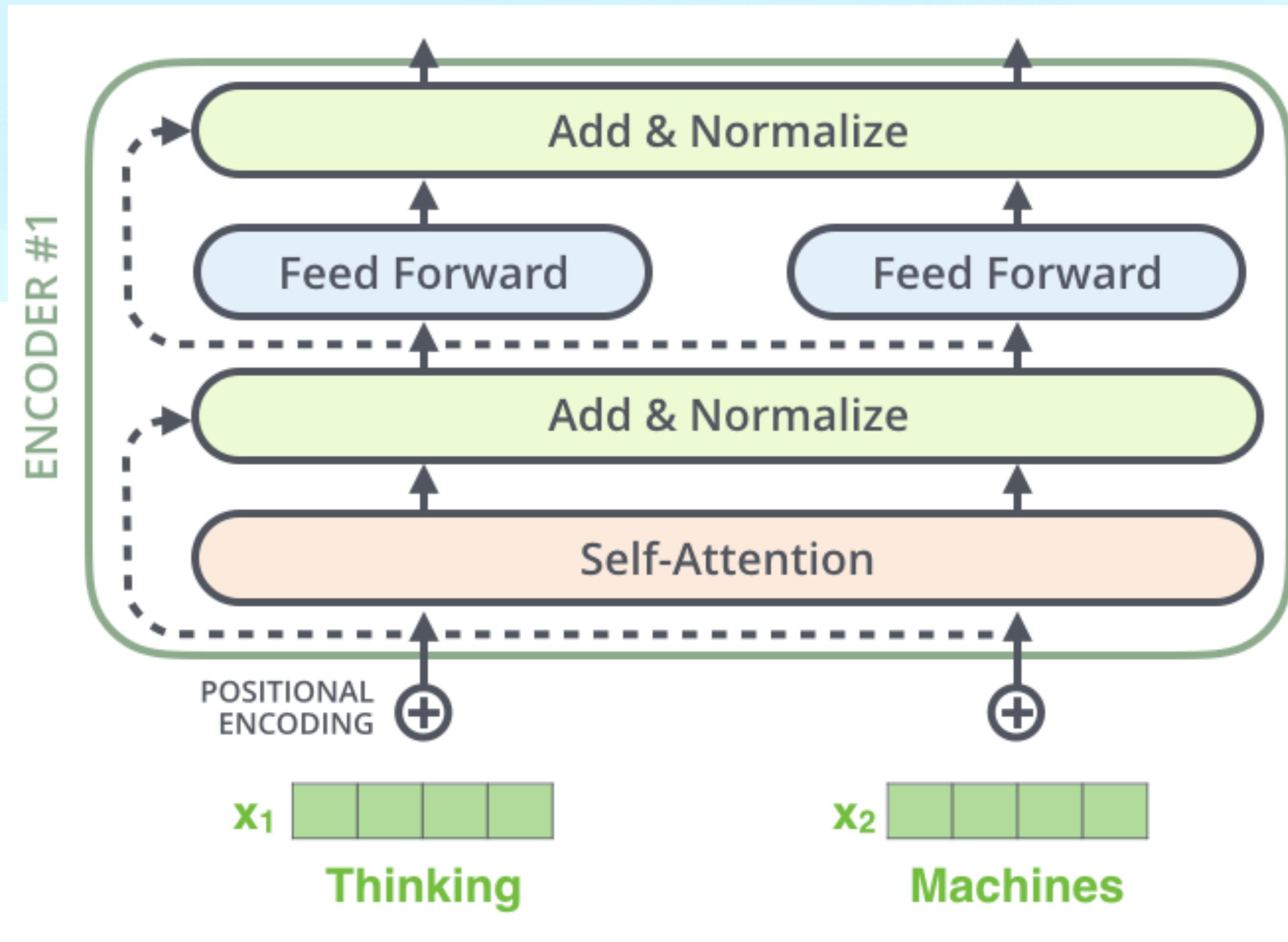
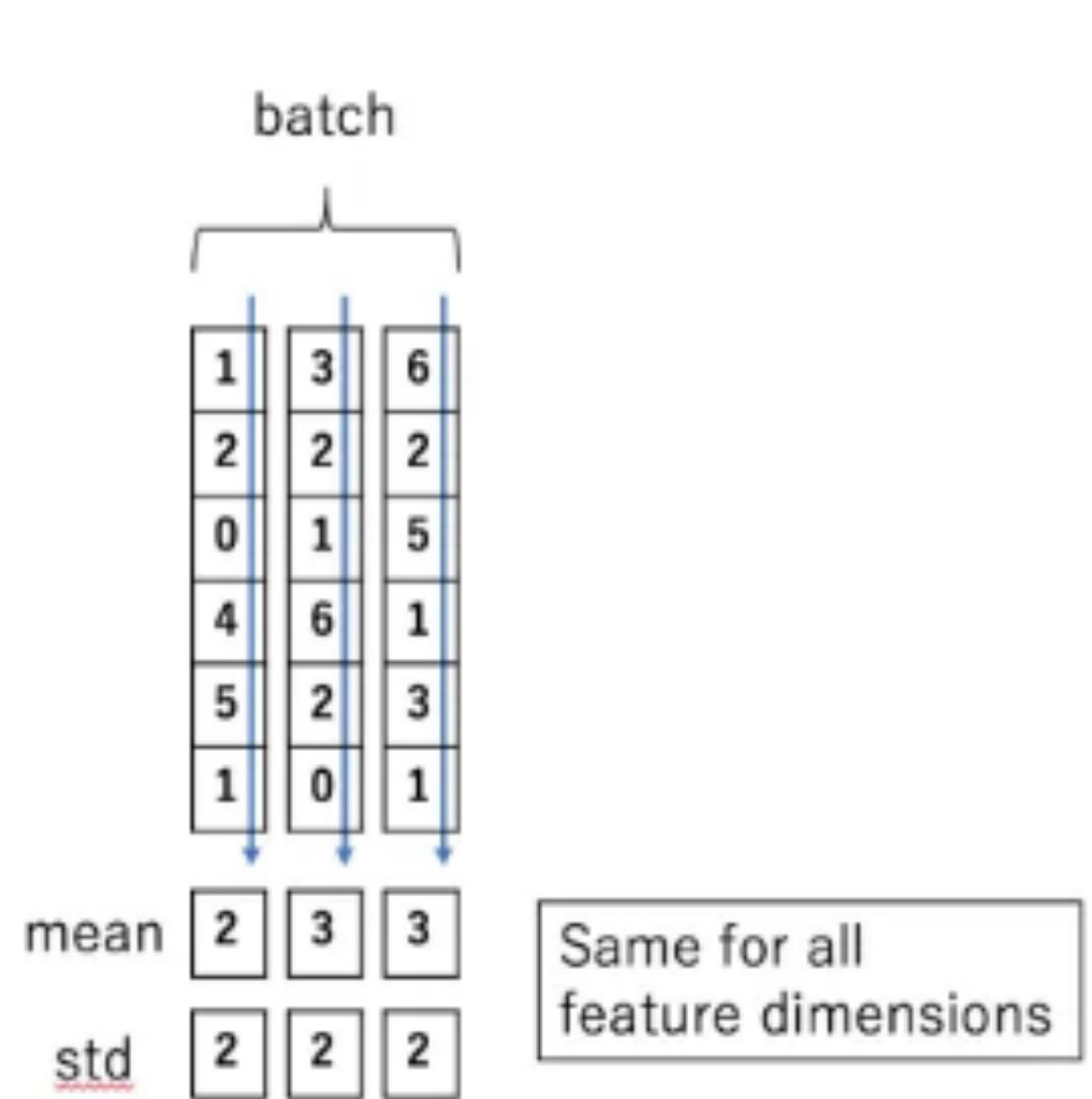
Layer Normalization (Hinton)

Layer normalization normalizes the inputs across the features.

Batch Normalization



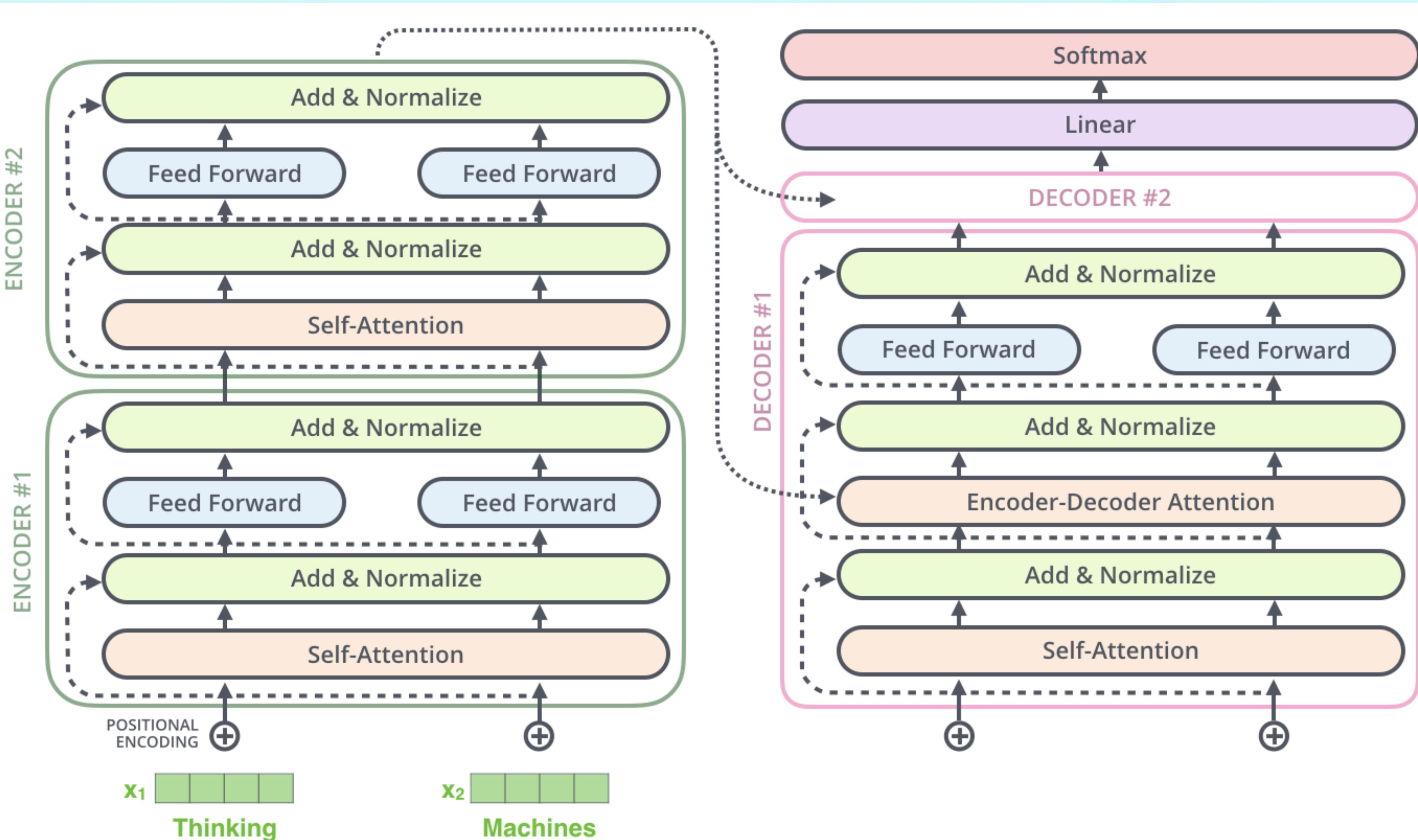
Layer Normalization



Attention and Transformers

The complete transformer

The encoder-decoder attention is just like self attention, except it uses K, V from the top of encoder output, and its own Q



Attention and Transformers

Note: In decoder, the input is “incomplete” when calculating self-attention.

The solution is to set future unknown values with “-inf”.

Attention and Transformers

Decoder's
Output
Linear
Layer

Which word in our vocabulary
is associated with this index?

Get the index of the cell
with the highest value
(**argmax**)

log_probs

logits

Decoder stack output

am

5



Softmax



Linear

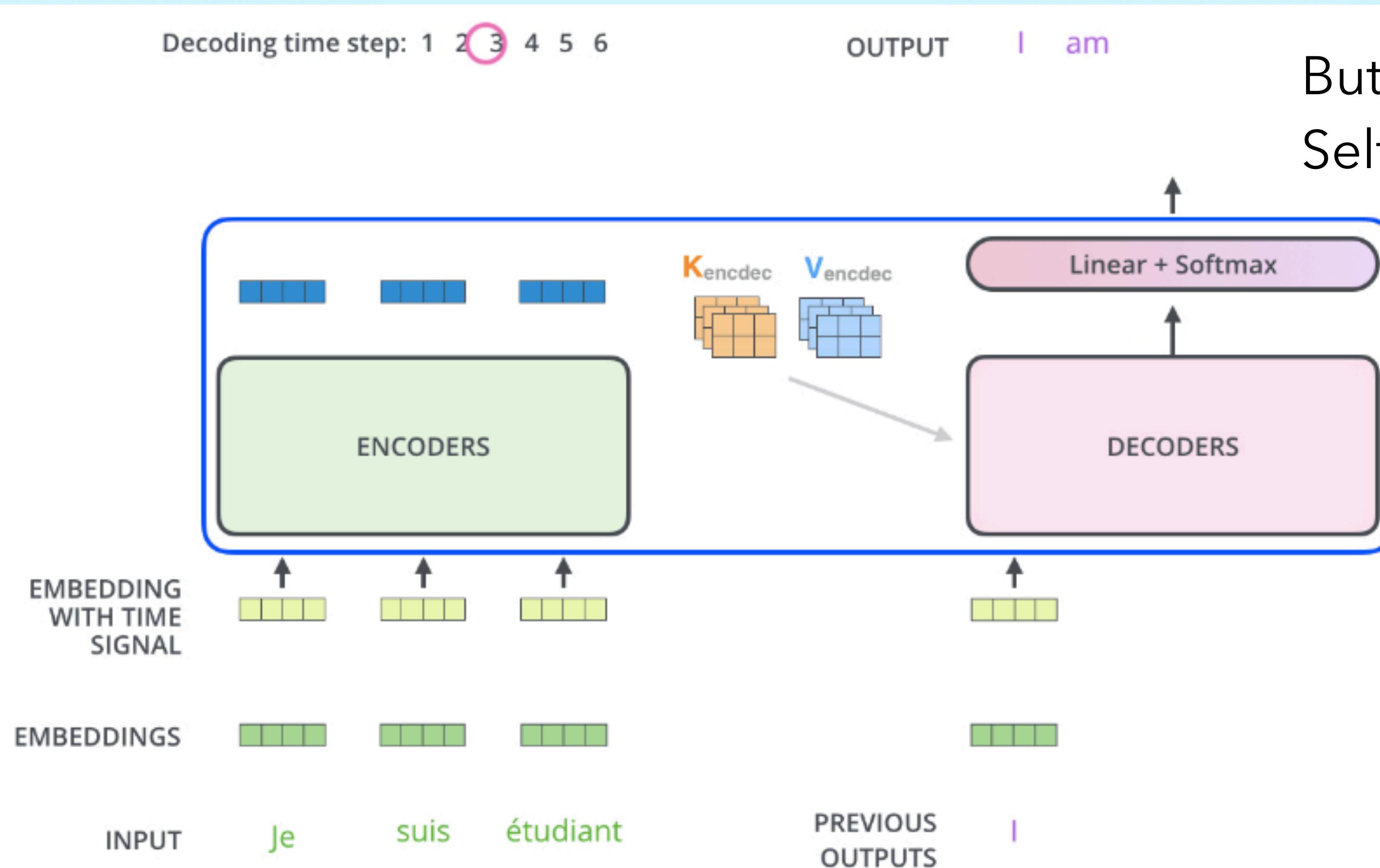


Attention and Transformers

How it works

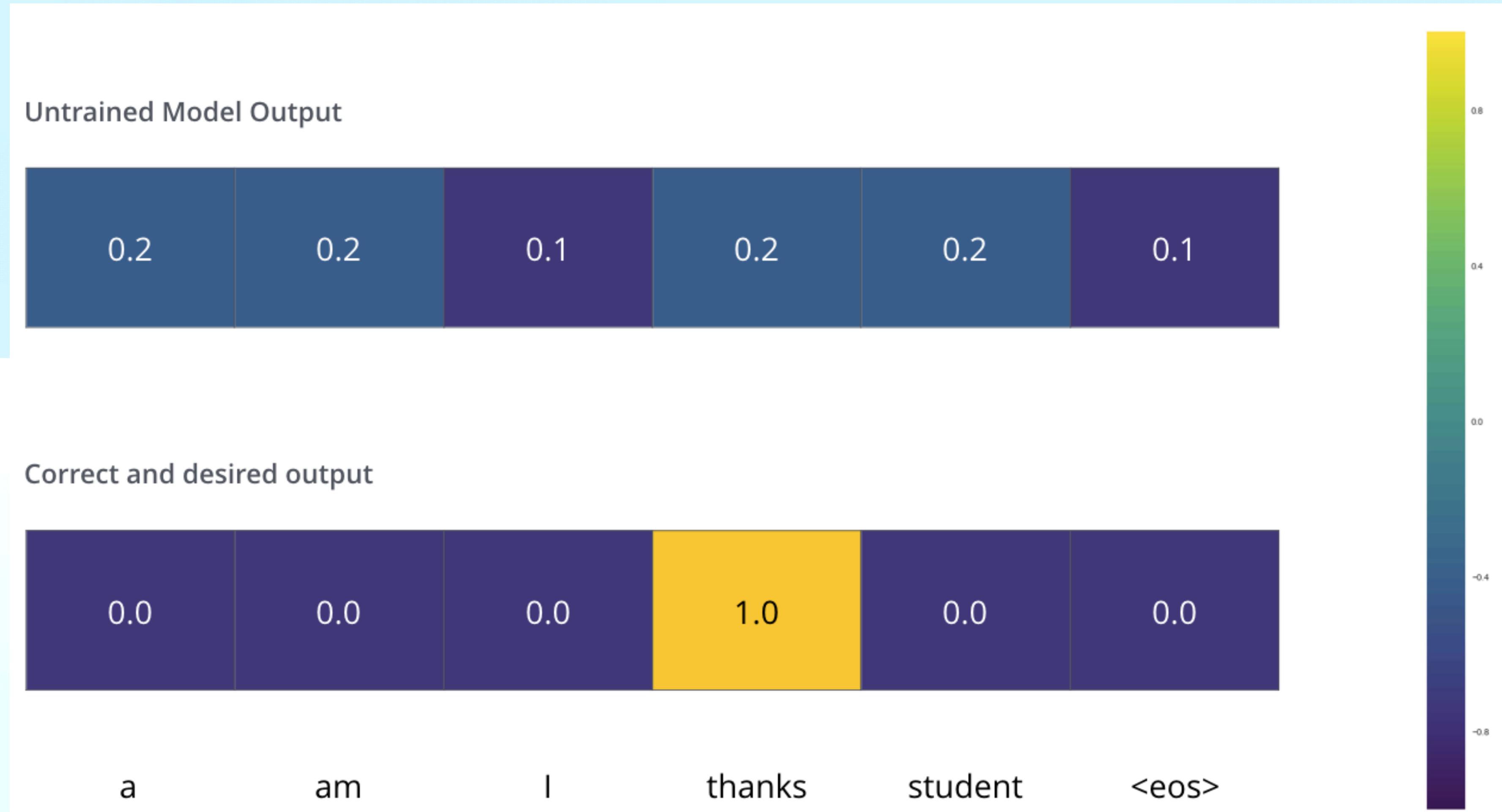
Decoder's
Output
Linear
Layer

But what about
Self-attention?



Attention and Transformers

Training and the Loss Function



We can use cross Entropy.

We can also optimize two words at a time: using BEAM search: keep a few alternatives for the first word.

Attention and Transformers

Cross Entropy and KL (Kullback-Leibler) divergence

- **Entropy:** $E(P) = - \sum_i P(i) \log P(i)$ - expected prefix-free code length (also optimal)
- **Cross Entropy:** $C(P) = - \sum_i P(i) \log Q(i)$ – expected coding

length using optimal code for Q

- **KL divergence:**

$$D_{KL}(P || Q) = \sum_i P(i) \log [P(i)/Q(i)] = \sum_i P(i) [\log P(i) - \log Q(i)], \text{ extra bits to code using } Q \text{ rather than } P$$

- **JSD(P||Q) = $\frac{1}{2} D_{KL}(P||M) + \frac{1}{2} D_{KL}(Q||M)$, M= $\frac{1}{2} (P+Q)$, symmetric KL**

* JSD = Jensen-Shannon Divergency

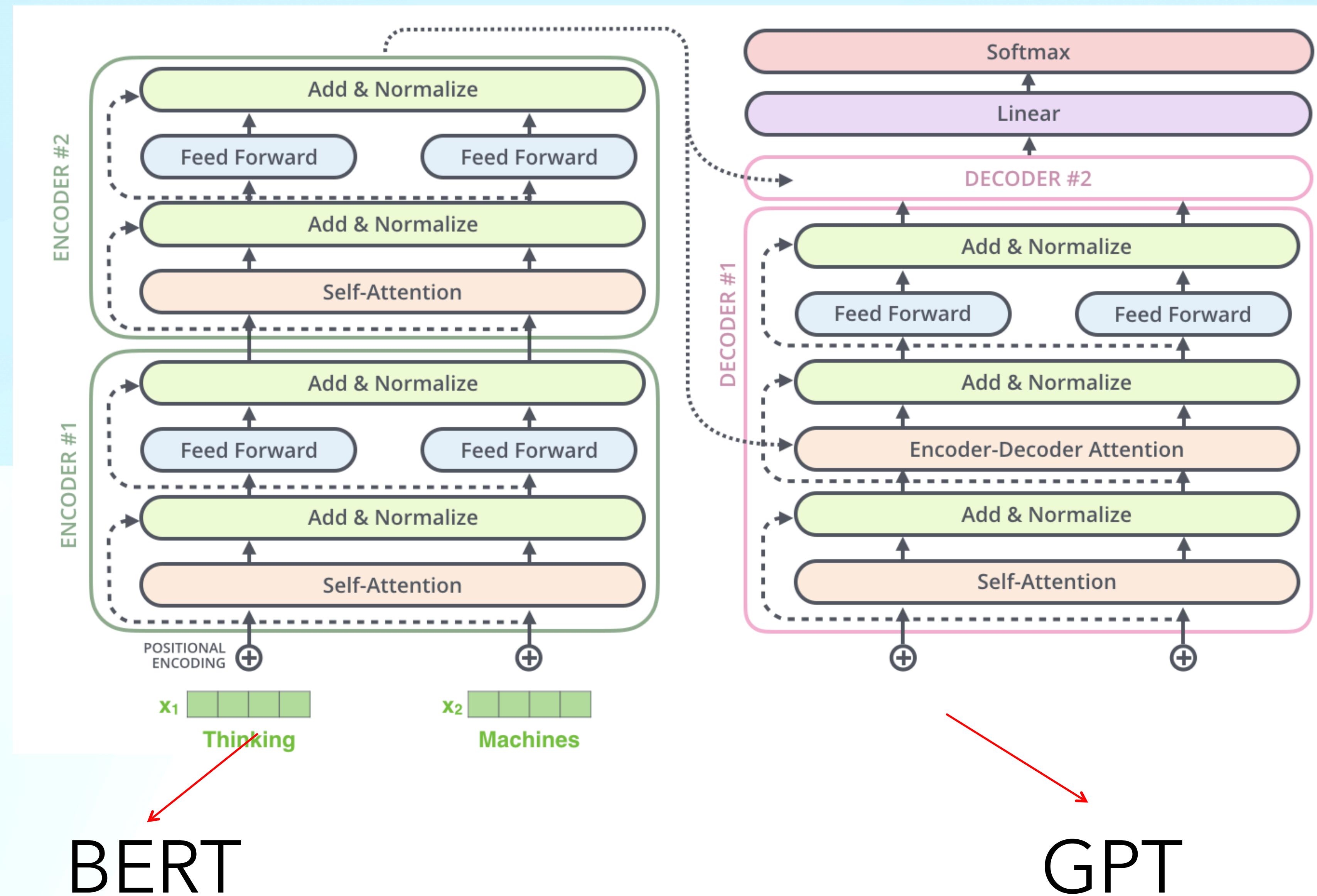
Attention and Transformers

Transformer Results

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1		$3.3 \cdot 10^{18}$
Transformer (big)	28.4	41.8		$2.3 \cdot 10^{19}$

Attention and Transformers



BERT

GPT

Understanding BERT

- **BERT (Bidirectional Encoder Representations from Transformers) is a transformer-based machine learning technique for natural language processing (NLP).** It was developed by Google and introduced in a 2018 paper titled "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding".
- BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. It represents a major leap in the evolution of NLP models because of its superior ability to understand the context of a word in a sentence.

Understanding BERT

- **Applications:**
- **BERT is widely used in many NLP tasks, including:**
 - Text classification
 - Sentiment analysis
 - Named entity recognition
 - Question answering
- **Strengths:**
 - It understands the context of words in sentences very well due to its bidirectional nature.
 - It has state-of-the-art performance in a wide variety of NLP tasks.
- **Weaknesses:**
 - BERT models are typically large and require significant computational resources to train.
 - Fine-tuning BERT models can be a bit challenging and requires a good understanding of the underlying technology.

Sentiment Analysis with BERT

- See Jupyter Notebook

Workshop # 6 : Text Classification using BERT

- **Objective:** Classify movie reviews as either positive or negative using BERT.
- Install the necessary libraries: You'll need the Transformers library from HuggingFace to get the BERT model and tokenizer. Install it using pip.
- Download IMDB dataset from https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz. You should have a folder called 'aclImdb' with 'train' and 'test' subfolders, each containing 'pos' and 'neg' subfolders.
- Load the data: Create a function to load the data from the text files.
- Preprocess the data: Use the **BERT tokenizer** to tokenize the reviews and prepare them for input into the model.
- Create the model: Use the **BertForSequenceClassification** model from the Transformers library.
- Train the model: Train the model on the preprocessed data.
- Evaluate the model: Evaluate the model's performance on the test set.

Workshop # 6 : Text Classification using BERT

- **Code Skeleton should be in the file called “Workshop 6 - BERT”**

HuggingFace Transformer



Introduction to HuggingFace Transformers

- HuggingFace's Transformers is an **open-source library** that offers a comprehensive platform for state-of-the-art Natural Language Processing (NLP) tasks.
- It is built on top of popular machine learning frameworks like **TensorFlow** and **PyTorch**, providing an abstraction layer for handling numerous complex operations involved in building and training NLP models.
- At its core, HuggingFace Transformers provides easy access to a **broad range of pre-trained models**, such as BERT, GPT-2, GPT-3, T5, RoBERTa, and many more. These models have been trained on extensive datasets and are ready to be deployed or further fine-tuned on specific tasks.
- Not just limited to pre-trained models, HuggingFace Transformers also offers **utilities for model fine-tuning**. Fine-tuning involves adjusting the pre-trained model on specific datasets or tasks, helping in achieving high performance with significantly less data and computational resources.
- The library's **high-level APIs are designed to be intuitive and user-friendly**, reducing the barrier of entry for both beginners and experts in machine learning and NLP.

Introduction to HuggingFace Transformers

```
# Importing HuggingFace Transformers library
from transformers import pipeline

# Initializing a text classification pipeline with a pre-trained BERT model
classifier = pipeline("sentiment-analysis", model="bert-base-uncased")

# Using the pipeline to classify a sentence
result = classifier("I love using HuggingFace Transformers")[0]

# Output: {'label': 'POSITIVE', 'score': 0.9998656511306763}
print(result)
```

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.weight', 'classifier.bias']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Downloading

466k/466k [00:00<00:00,

(...)/main/tokenizer.json: 100%

737kB/s]

Xformers is not installed correctly. If you want to use memory_efficient_attention to accelerate training use the following command to install Xformers

pip install xformers.

```
{'label': 'LABEL_0', 'score': 0.540632963180542}
```

HuggingFace: Key Features

- **Pre-trained Models:** HuggingFace provides access to a plethora of **pre-trained models** that have been fine-tuned on vast datasets. They encompass a broad range of architectures including BERT, GPT, GPT-2, Transformer-XL, XLNet, and many more. These models can be used directly for tasks like text classification, named entity recognition, and translation among others.
- **Tokenization:** HuggingFace also **offers tokenization for all models** it provides, ensuring that the input text is appropriately preprocessed and ready to be fed into the model. **The tokenizer takes care of all the necessary steps such as splitting the input into words, subwords, or symbols (known as tokens)**, mapping these tokens to their index in the model vocabulary, and creating the various inputs that the model requires.
- **Model Classes:** HuggingFace organizes models into a set of high-level classes that are easy to understand and work with. Each class corresponds to a **specific model architecture and provides all the necessary methods for working with that model**. For instance, the **BertModel class provides methods for working with the BERT model architecture**.

HuggingFace: Key Features

```
# Importing necessary classes from HuggingFace Transformers
from transformers import BertTokenizer, BertModel

# Initializing the BERT tokenizer and model
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Tokenizing input text
inputs = tokenizer("Hello, HuggingFace!", return_tensors="pt")

# Feeding the inputs to the model
outputs = model(**inputs)

# Output: tensor containing the hidden states of the last layer of the model
print(outputs.last_hidden_state)

tensor([[[-0.2291,  0.0135, -0.1188, ..., -0.2613, -0.0115,  0.6596],
        [-0.0107,  0.0749,  0.6104, ..., -0.1752,  0.3824, -0.0338],
        [-0.7685,  0.8493,  0.3917, ..., -1.2588, -0.3544,  0.0577],
        ...,
        [ 0.2830, -0.0994,  1.1351, ..., -0.4314,  0.6670, -0.0023],
        [-0.5194, -0.1928, -0.3060, ...,  0.6108, -0.3512,  0.1036],
        [ 0.7359,  0.3968, -0.0619, ...,  0.4613, -0.4899, -0.3350]]],  
grad_fn=<NativeLayerNormBackward0>)
```

HuggingFace: Popular Models

- **BERT (Bidirectional Encoder Representations from Transformers):** BERT is one of the most influential NLP models to date. It is pre-trained on a large corpus of text and then fine-tuned for specific tasks. It is unique in its use of bidirectional transformers, allowing it to understand the context of a word based on all of its surroundings (left and right of the word).
- **GPT-2 (Generative Pretrained Transformer 2):** GPT-2, developed by OpenAI, is renowned for its impressive capabilities in text generation. GPT-2 is trained on a diverse range of internet text and can generate coherent and contextually relevant sentences by predicting the next word in a given piece of text.
- **RoBERTa (Robustly Optimized BERT Approach):** RoBERTa is a variant of BERT that uses a different training approach and more training data. It was shown to outperform BERT on several benchmarks.
- **DistilBERT:** DistilBERT is a smaller, faster, and lighter version of BERT. It retains 95% of BERT's performance while being 60% smaller and 60% faster.
- **T5 (Text-to-Text Transfer Transformer):** T5 is a versatile transformer-based model that can be fine-tuned for various NLP tasks by simply changing the task-specific prefix in the input text.
- Support over 400+ models in Thai languages.

HuggingFace: Popular Models

```
# Importing necessary classes from HuggingFace Transformers
from transformers import BertModel, GPT2Model, RobertaModel, DistilBertModel

# Initializing the models
bert = BertModel.from_pretrained('bert-base-uncased')
gpt2 = GPT2Model.from_pretrained('gpt2')
roberta = RobertaModel.from_pretrained('roberta-base')
distilbert = DistilBertModel.from_pretrained('distilbert-base-uncased')
t5 = T5Model.from_pretrained('t5-base')
```

HuggingFace: Pipeline API

- **Introduction:** HuggingFace provides a high-level pipeline API that allows us to make use of pre-trained models in a very straightforward way. It supports a wide range of tasks like Text Generation, Sentiment Analysis, Name Entity Recognition, Translation, and more.
- **Ease of Use:** The pipeline API takes care of all the preprocessing and postprocessing required for the input and output of the models, making it easier for developers to use these state-of-the-art models.
- **Customizable:** Though simple to use, the pipeline is also highly customizable. You can easily switch between different models and tokenizers suited to your specific task.
- **Model Agnostic:** The pipeline function is model-agnostic, meaning you can use it with any model as long as the model has been fine-tuned on a similar task.

HuggingFace: Pipeline API

```
# Importing HuggingFace Transformers library
from transformers import pipeline

# Initializing a text classification pipeline with a pre-trained BERT model
classifier = pipeline("sentiment-analysis", model="bert-base-uncased")

# Using the pipeline to classify a sentence
result = classifier("I love using HuggingFace Transformers")[0]

# Output: {'label': 'POSITIVE', 'score': 0.9998656511306763}
print(result)
```

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Downloading  466k/466k [00:00<00:00,
(...)main/tokenizer.json: 100% 737kB/s]

Xformers is not installed correctly. If you want to use memory_efficient_attention to accelerate training use the following command to install Xformers

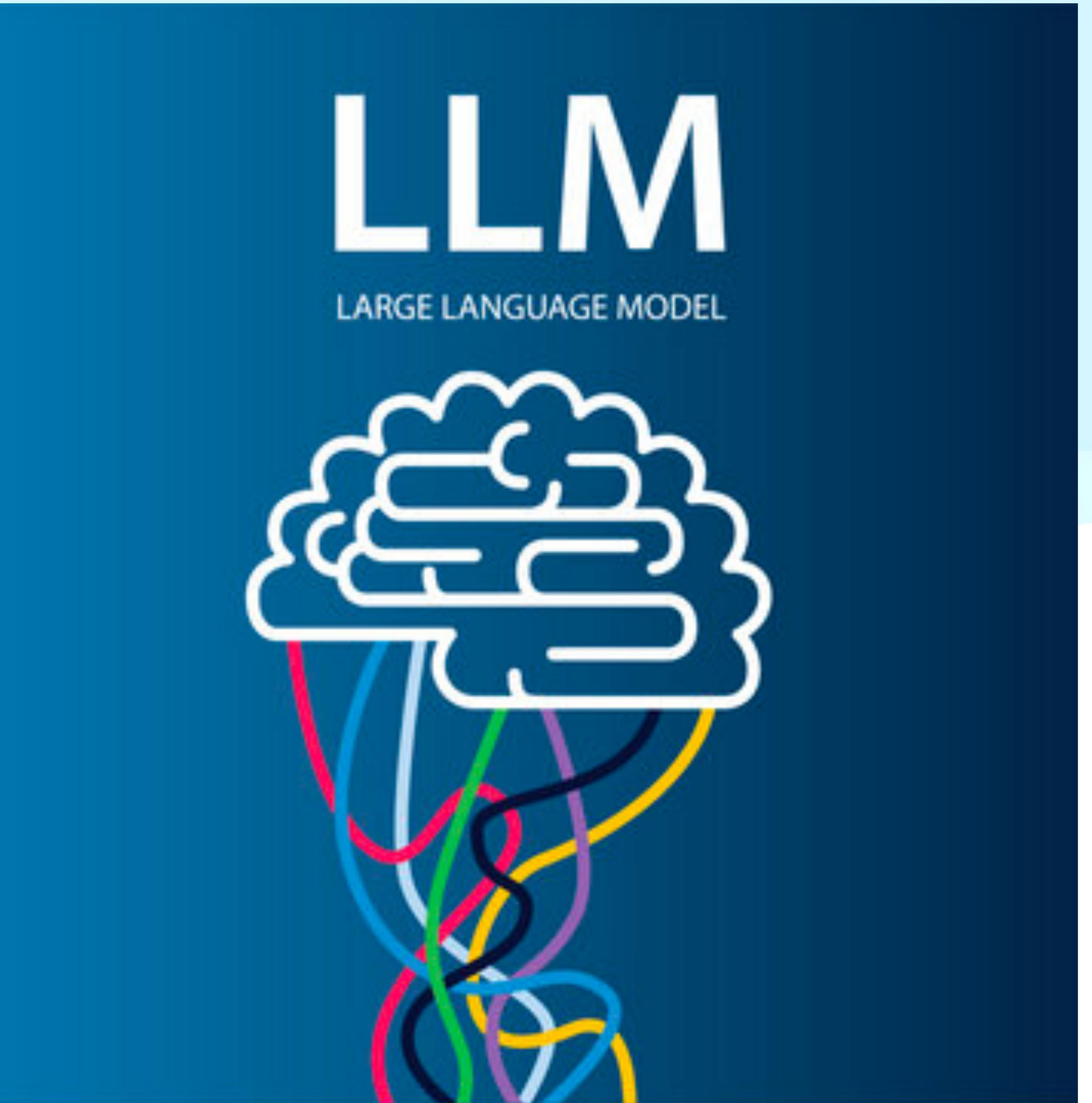
pip install xformers.

```
{'label': 'LABEL_0', 'score': 0.540632963180542}
```

HuggingFace: Community and Model Hub

- **Community:** HuggingFace is open-source and boasts a large community of researchers, developers, and data scientists actively contributing to its development. This makes it one of the fastest-evolving libraries for NLP tasks.
- **Model Hub:** The HuggingFace Model Hub is a platform where users can share and download pre-trained models. It currently hosts thousands of models based on different architectures (BERT, GPT-2, T5, etc.) and fine-tuned for various tasks.
- **Diverse Models:** The Model Hub hosts models trained on diverse languages and tasks, making it a versatile tool for various NLP applications. Users can use these models for their specific tasks without needing to train a model from scratch.
- **Collaborative Environment:** HuggingFace provides an environment that encourages sharing and collaboration. This fosters continual learning, progress, and innovation within the field of NLP.

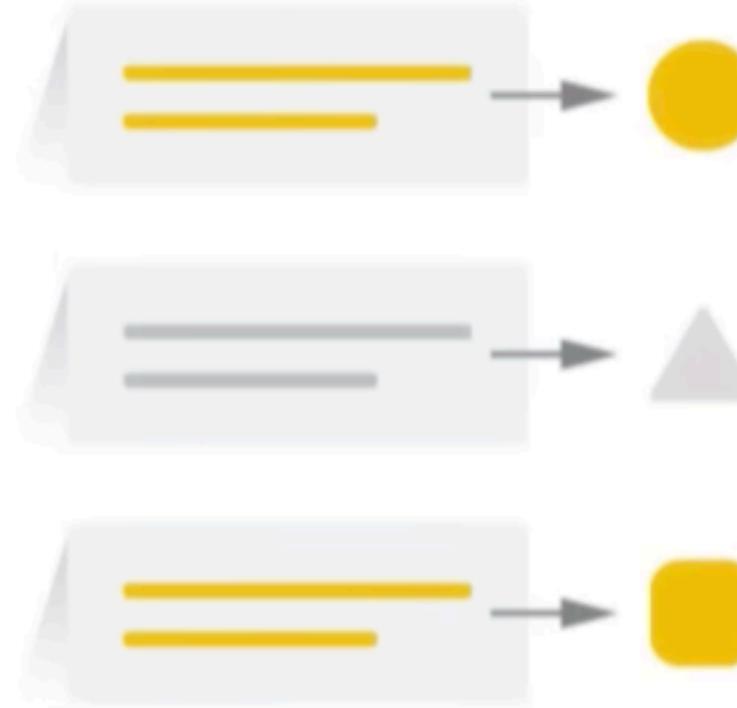
What are Large Language Model (LLM)?



Large, general-purpose language models
can be pre-trained and then fine-tuned for
specific purposes

Large language models are trained to solve common language problems, like

.....



Text
classification



Question
answering



Document
summarization

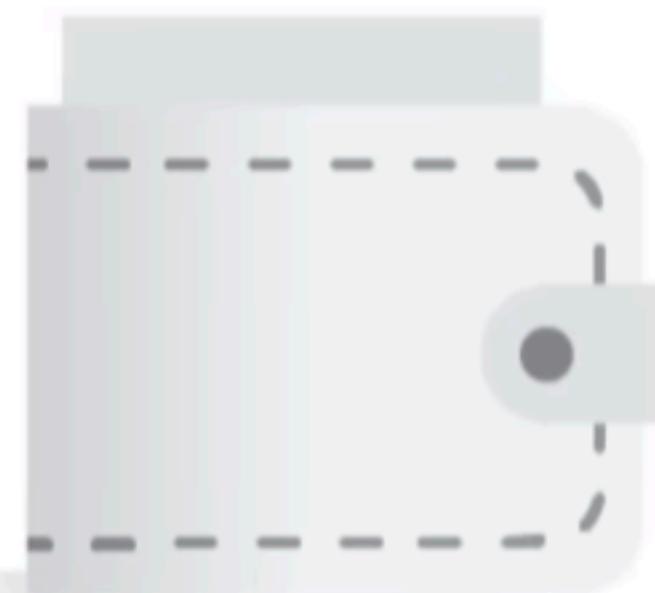


Text
generation

Then fine tune it



Retail



Finance

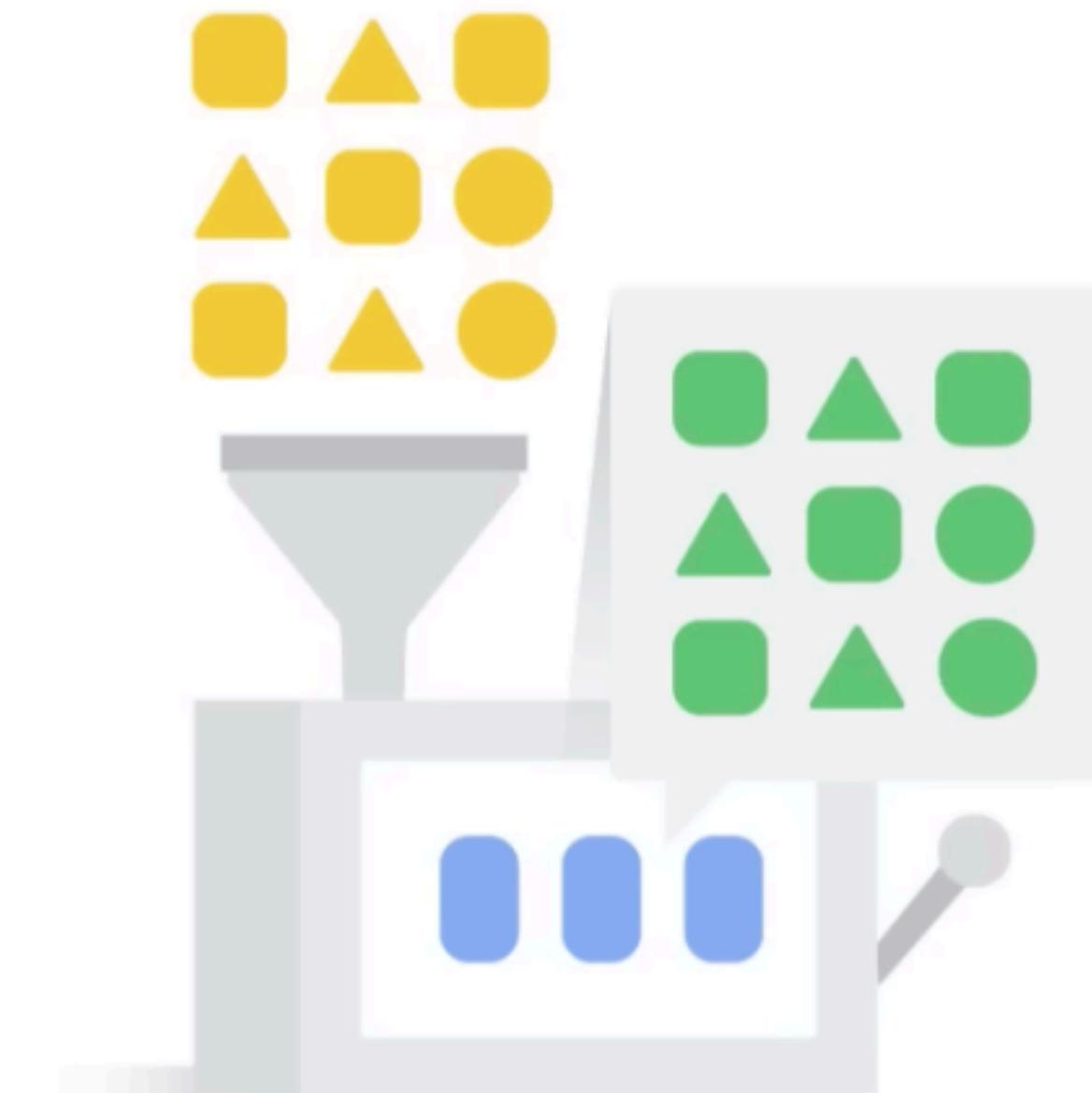


Entertainment

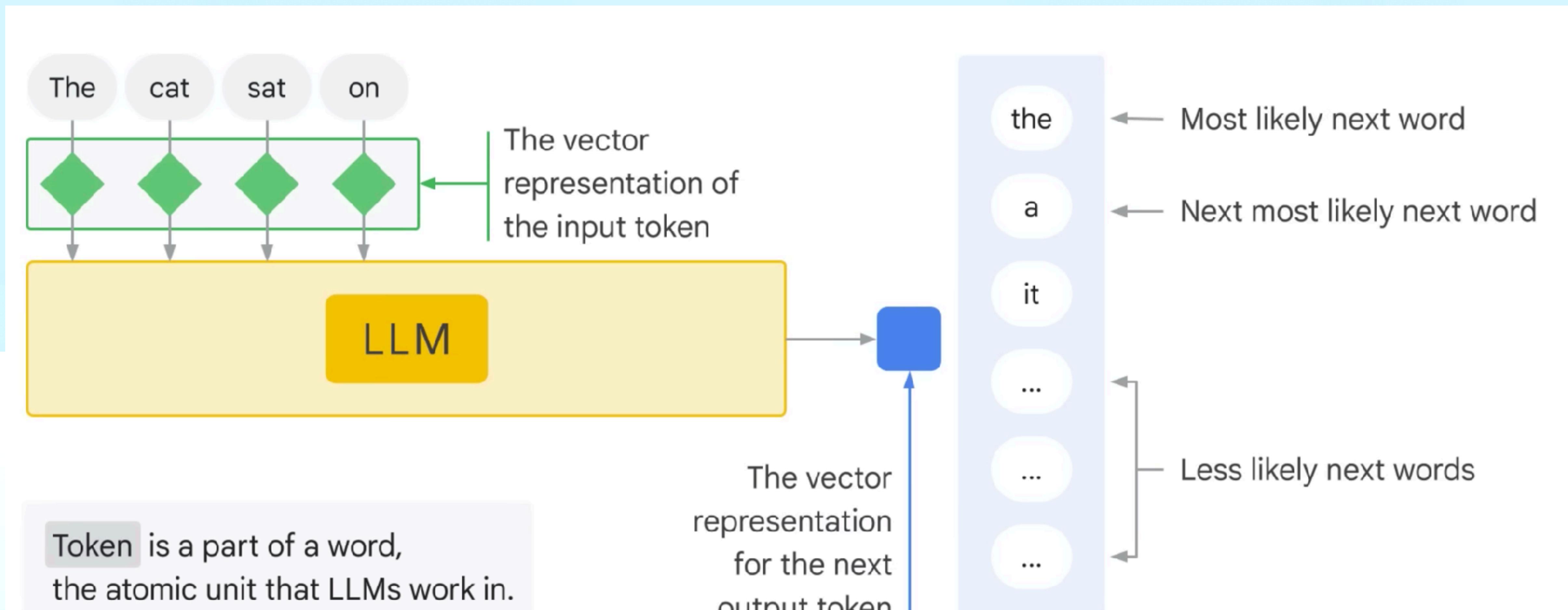
Trained with
a relatively small
size of field
datasets

Characteristics of LLM

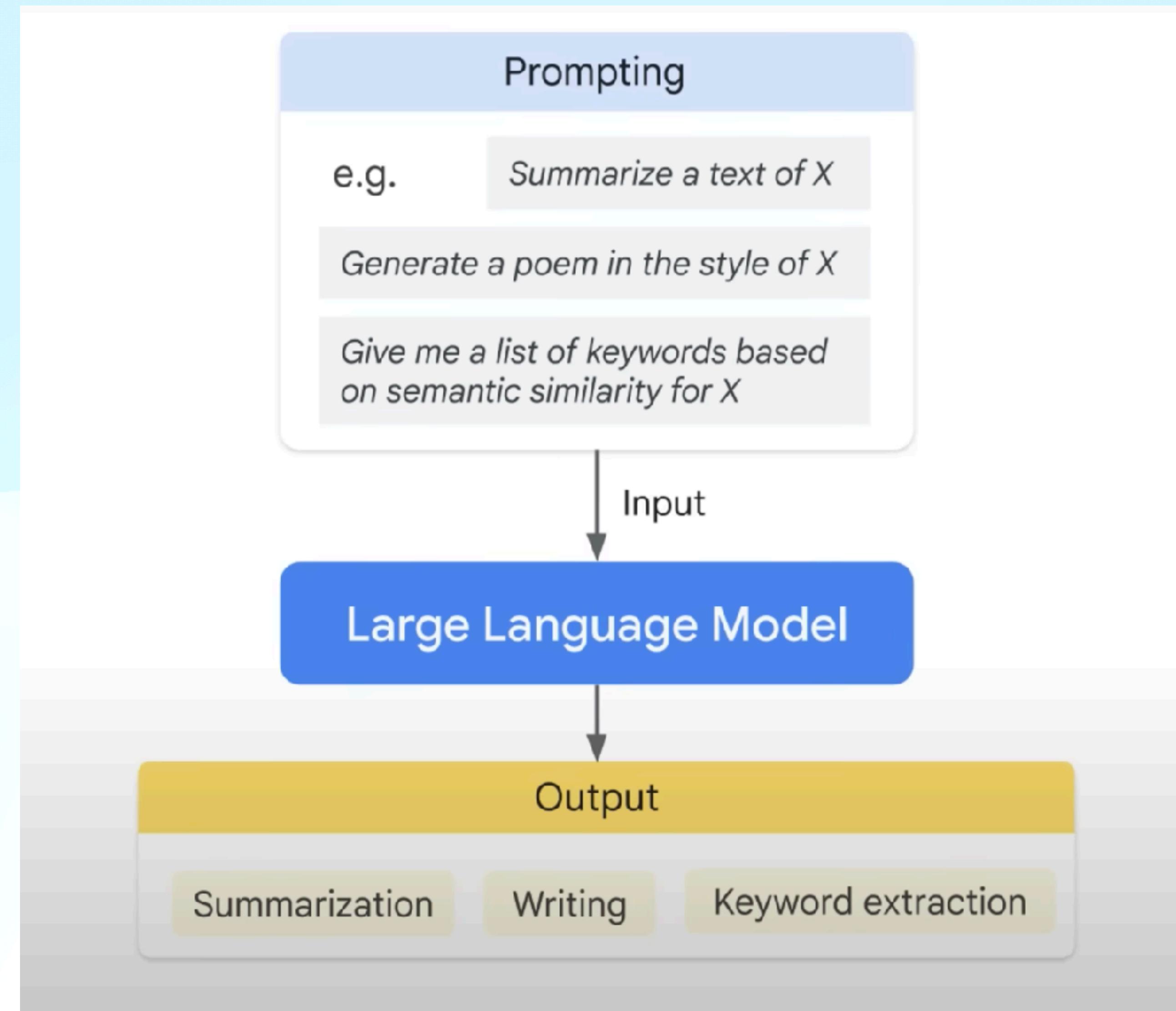
- Large
 - Large training dataset
 - Large number of parameters
- General purpose
 - Commonality of human languages
 - Resource restriction
- Pre-trained and fine-tuned



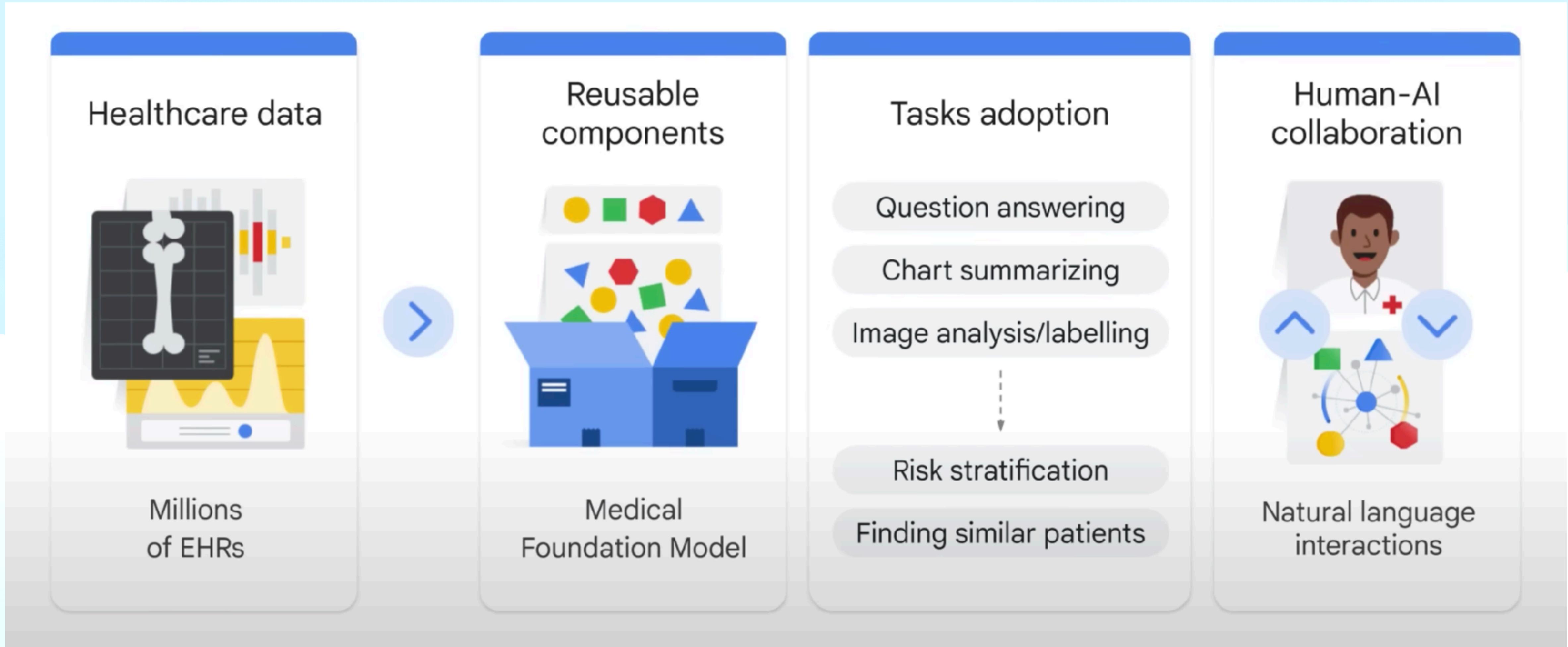
How LLM works?



Interact with LLM using Prompts

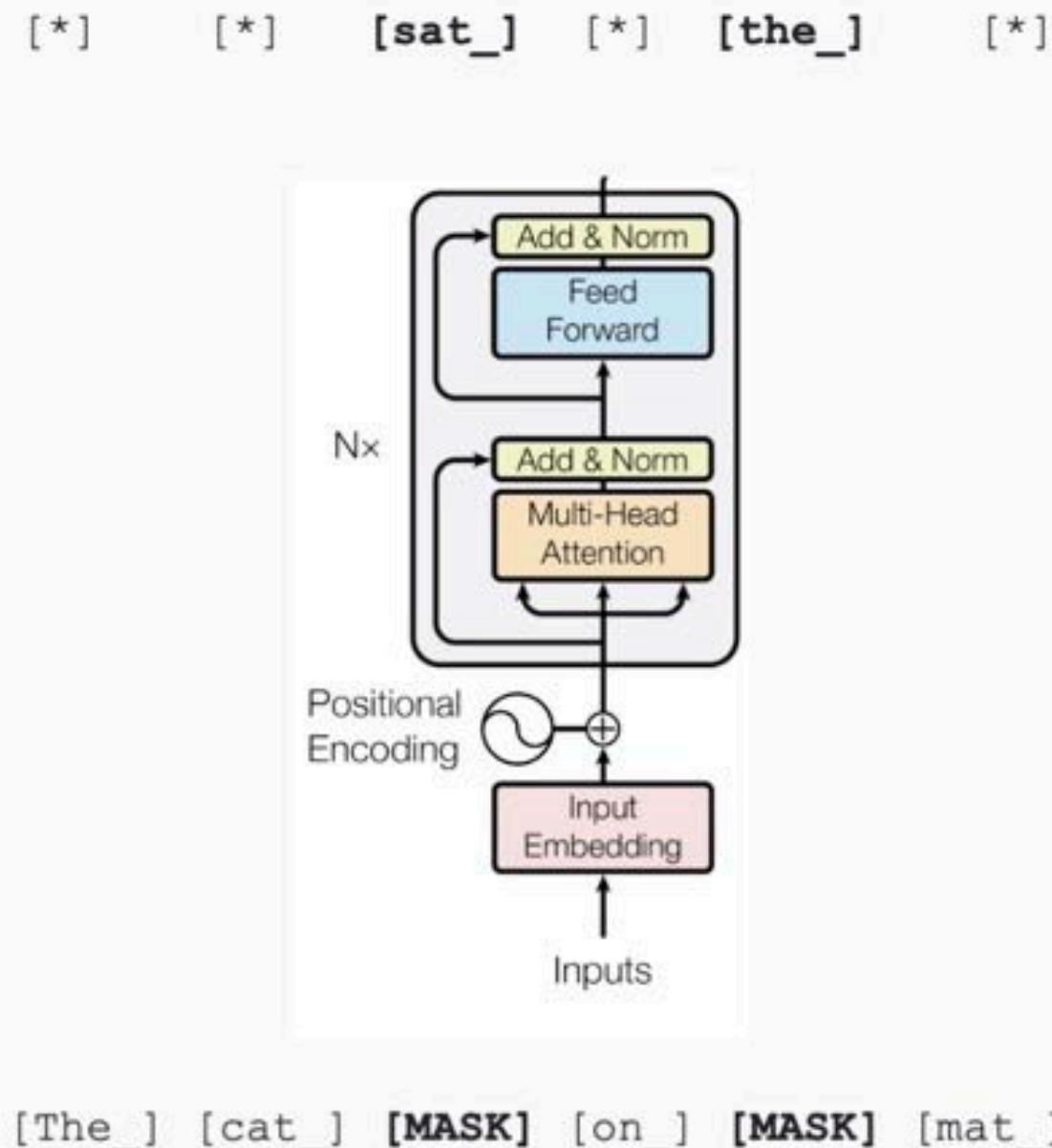


Fine-tune it



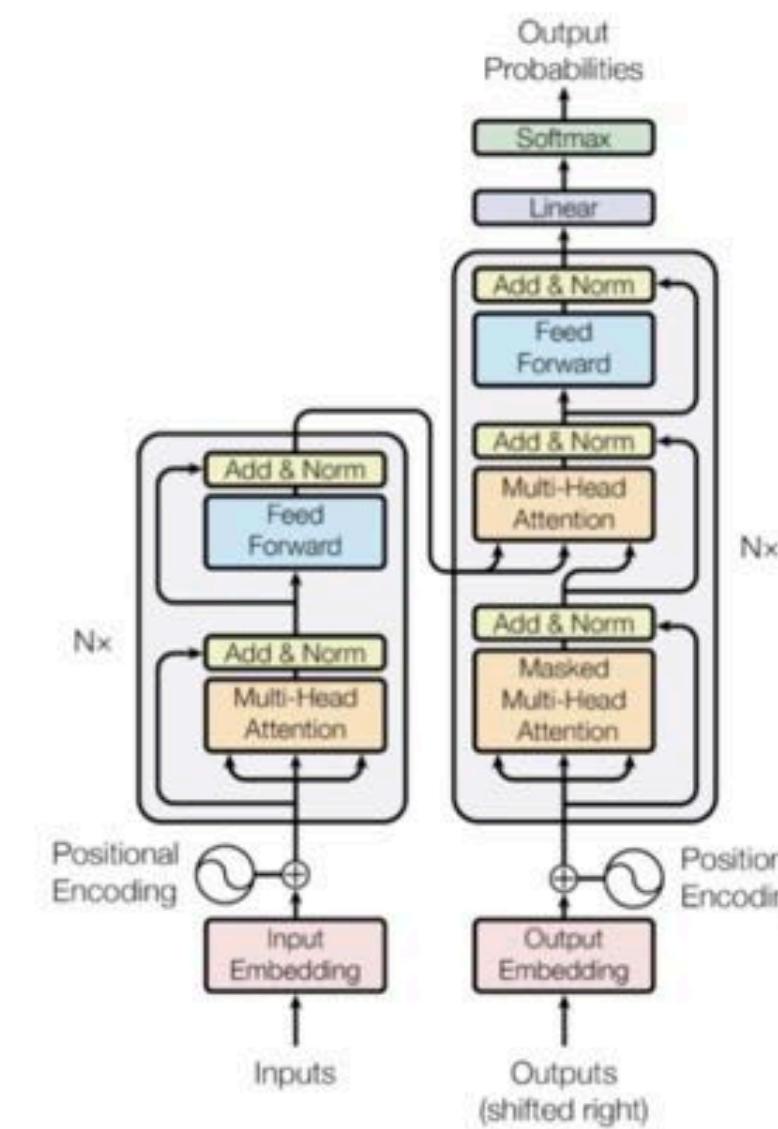
Three Easy Models

BERT

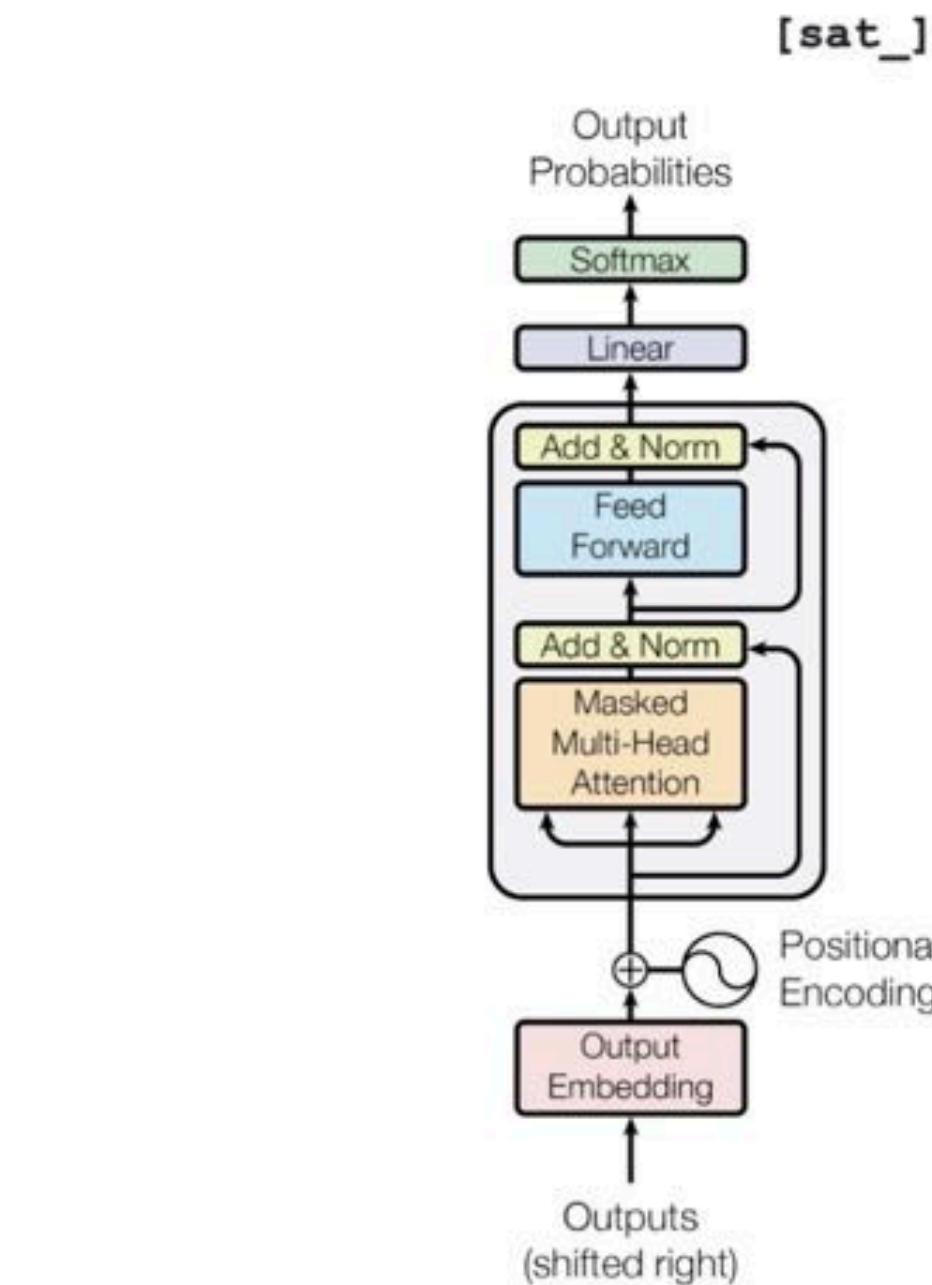


T5

Das ist gut.
A storm in Attala caused 6 victims.
This is not toxic.

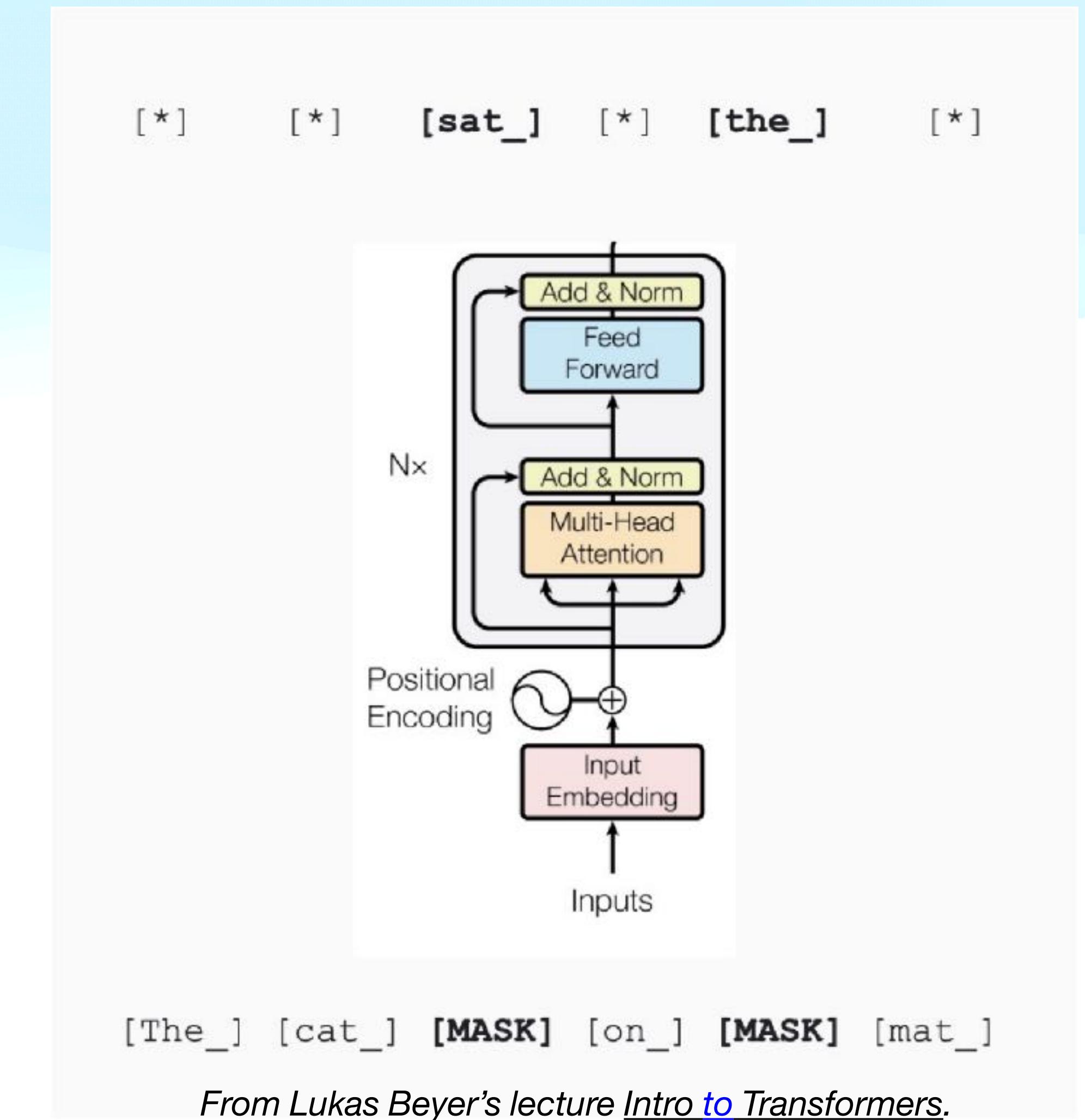


GPT



BERT (2019)

- *Bidirectional* Encoder Representations from Transformers
- Encoder-only (no attention masking)
- 110M params
- 15% of all words masked out
- Was great, now dated



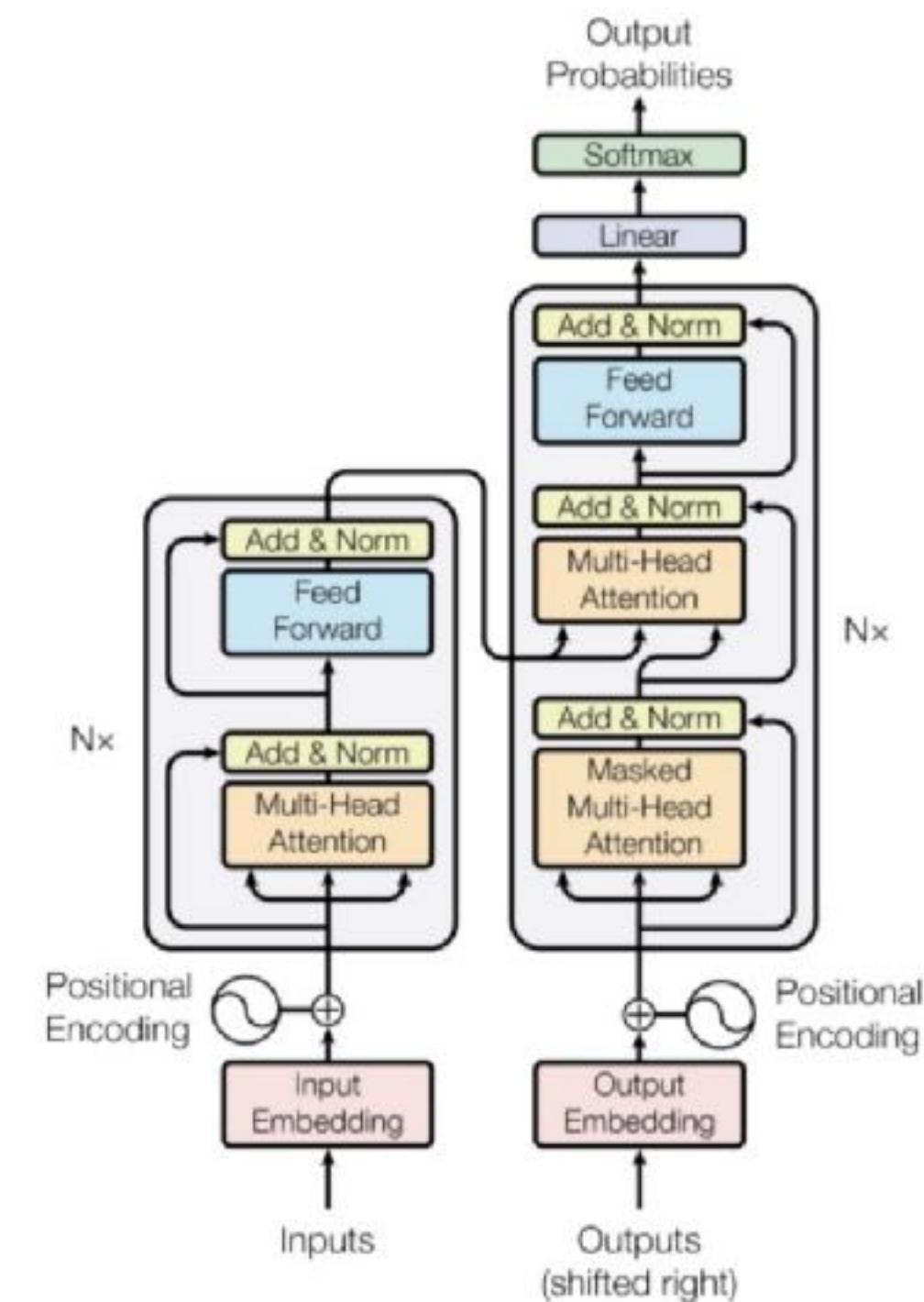
T5: Text-to-Text Transfer Transformer (2020)

- Input and output are both text strings
- Encoder-Decoder architecture
- 11B parameters
- Still could be a good choice for fine-tuning!

Das ist gut.

A storm in Attala caused 6 victims.

This is not toxic.



Translate EN-DE: This is good.

Summarize: state authorities dispatched...

Is this toxic: You look beautiful today!

T5 Training Data

- Unsupervised pre-training on Colossal Clean Crawled Corpus (C4)
 - Start with Common Crawl (over 50TB of compressed data, 10B+ web pages)
 - Filtered down to ~800GB, or ~160B tokens
- Also trained on academic supervised tasks
- We discarded any page with fewer than 5 sentences and only retained lines that contained at least 3 words.
- We removed any page that contained any word on the “List of Dirty, Naughty, Obscene or Otherwise Bad Words”.⁶
- Some pages inadvertently contained code. Since the curly bracket “{” appears in many programming languages (such as Javascript, widely used on the web) but not in natural text, we removed any pages that contained a curly bracket.
- To deduplicate the data set, we discarded all but one of any three-sentence span occurring more than once in the data set.

<https://paperswithcode.com/dataset/c4>

- Sentence acceptability judgment
 - CoLA [Warstadt et al., 2018](#)
- Sentiment analysis
 - SST-2 [Socher et al., 2013](#)
- Paraphrasing/sentence similarity
 - MRPC [Dolan and Brockett, 2005](#)
 - STS-B [Cer et al., 2017](#)
 - QQP [Jyer et al., 2017](#)
- Natural language inference
 - MNLI [Williams et al., 2017](#)
 - QNLI [Rajpurkar et al., 2016](#)
 - RTE [Dagan et al., 2005](#)
 - CB [De Marneff et al., 2019](#)
- Sentence completion
 - COPA [Roemmele et al., 2011](#)
- Word sense disambiguation
 - WIC [Pilehvar and Camacho-Collados, 2018](#)
- Question answering
 - MultiRC [Khashabi et al., 2018](#)
 - ReCoRD [Zhang et al., 2018](#)
 - BoolQ [Clark et al., 2019](#)

Here come the GPTs



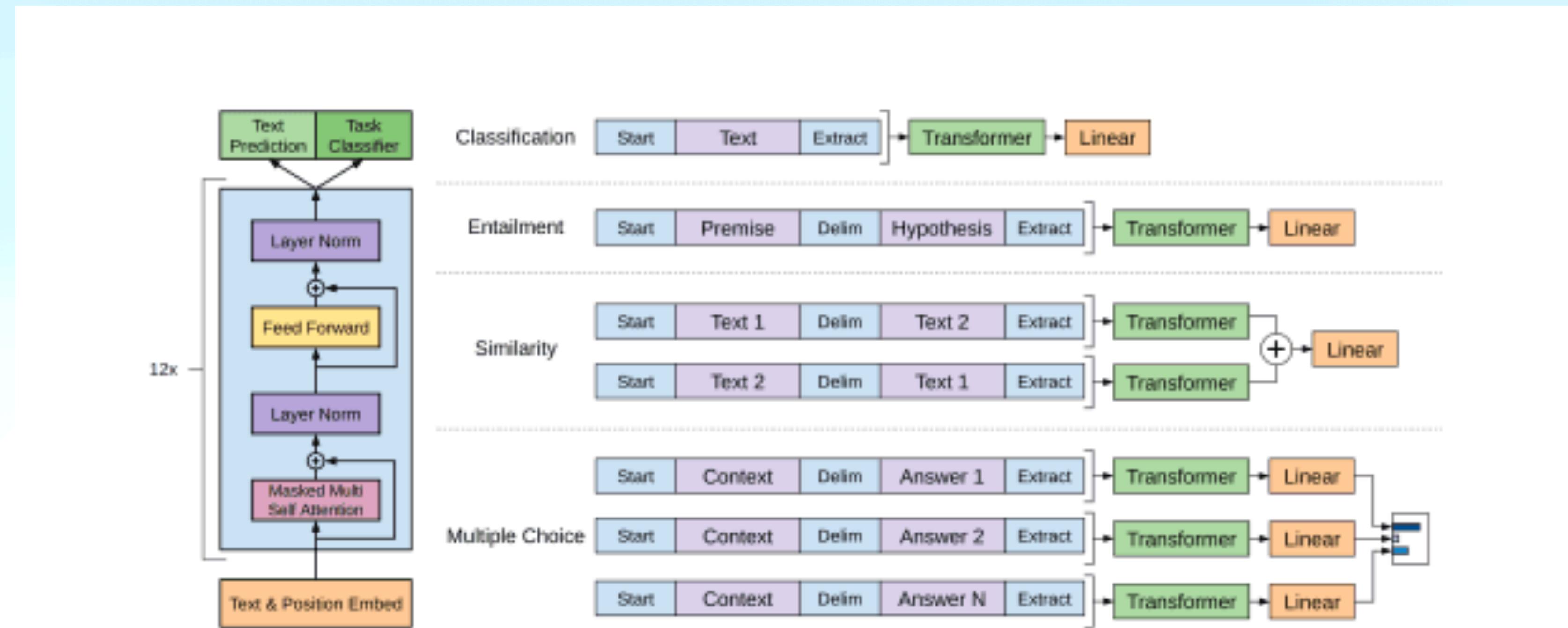
What are Generative Pre-trained Transformers?

- **Generative Pre-trained Transformers** (GPT) are a type of deep learning model used to generate human-like text.
- Common uses include
 - answering questions
 - summarizing text
 - translating text to other languages
 - generating code
 - generating blog posts, stories, conversations, and other content types.
- There are **endless applications for GPT models**, and you can even fine-tune them on specific data to create even better results. By using transformers, you will be saving costs on computing, time, and other resources.



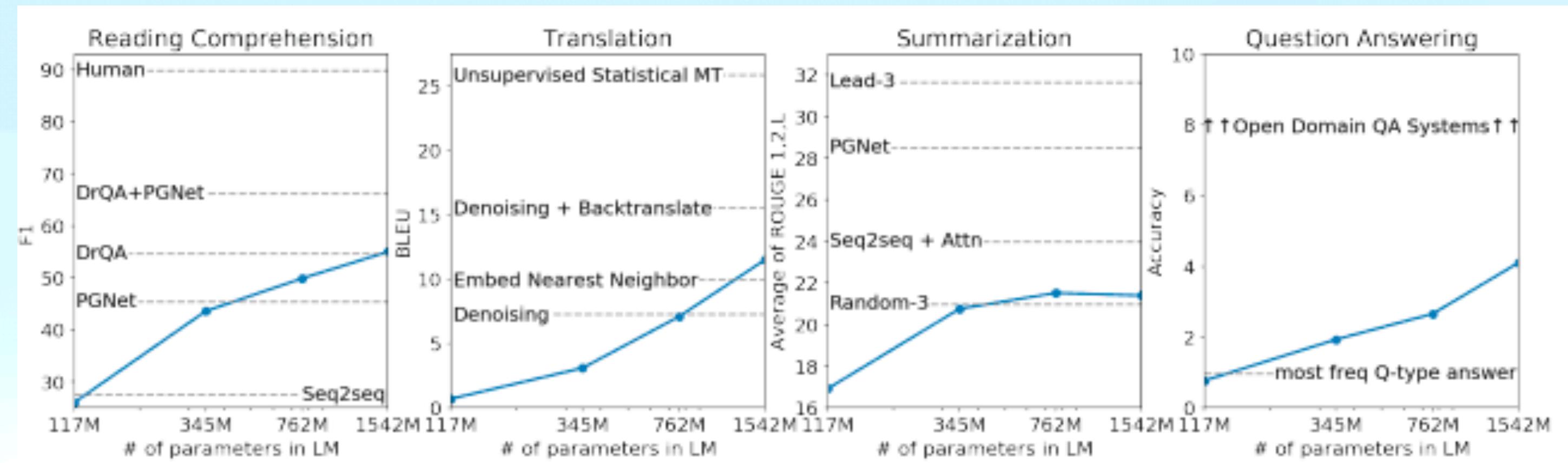
GPT-1

In 2018, OpenAI published a paper (*Improving Language Understanding by Generative Pre-Training*) about using natural language understanding using their GPT-1 language model. This model was a proof-of-concept and was not released publicly.



GPT-2

- The following year, OpenAI published another paper (*Language Models are Unsupervised Multitask Learners*) about their latest model, GPT-2.
- This time, the model was made available to the machine learning community and found some adoption for text generation tasks.
- GPT-2 could often generate a couple of sentences before breaking down. This was state-of-the-art in 2019.



Model performance on various tasks | GPT-2 paper

GPT-3

- In 2020, OpenAI published another paper (**Language Models are Few-Shot Learners**) about their GPT-3 model. The model had **100 times more parameters than GPT-2** and was trained on an even larger text dataset, resulting in better model performance.
- The model continued to be improved with various iterations known as the GPT-3.5 series, including the **conversation-focused ChatGPT**.
- This version took the world by storm after surprising the world with its ability to generate pages of human-like text.
- ChatGPT became the fastest-growing web application ever, reaching 100 million users in just two months.

Setting	NaturalQS	WebQS	TriviaQA
RAG (Fine-tuned, Open-Domain) [LPP ⁺ 20]	44.5	45.5	68.0
T5-11B+SSM (Fine-tuned, Closed-Book) [RRS20]	36.6	44.7	60.5
T5-11B (Fine-tuned, Closed-Book)	34.5	37.4	50.1
GPT-3 Zero-Shot	14.6	14.4	64.3
GPT-3 One-Shot	23.0	25.3	68.0
GPT-3 Few-Shot	29.9	41.5	71.2

Results on three Open-Domain QA tasks | GPT-3 paper

GPT-4

GPT-4 has been developed to improve model "alignment" - the ability to follow user intentions while also making it more truthful and generating **less offensive or dangerous output**.

GPT-4

Performance improvements

- GPT-4 improves on GPT-3.5 models regarding the **factual correctness** of answers. The number of "**hallucinations**," where the model makes factual or reasoning errors, is lower, with GPT-4 scoring **40% higher** than GPT-3.5 on OpenAI's internal factual performance benchmark.
- It also improves "**steerability**," which is the ability to change its behavior according to user requests. For example, you can command it to write in a different style or tone or voice.
- A further improvement is in the **model's adherence to guardrails. If you ask it to do something illegal or unsavory**, it is better at refusing the request.

Using Visual Inputs in GPT-4

- One major change is that **GPT-4 can use image inputs** (research preview only; not yet available to the public) and text. Users can specify any vision or language task by entering interspersed text and images.
- Examples showcased highlight **GPT-4 correctly interpreting complex imagery such as charts, memes, and screenshots from academic papers.**

User What is funny about this image? Describe it panel by panel.



Source: [hmmm \(Reddit\)](#)

GPT-4 The image shows a package for a "Lightning Cable" adapter with three panels.

Panel 1: A smartphone with a VGA connector (a large, blue, 15-pin connector typically used for computer monitors) plugged into its charging port.

Panel 2: The package for the "Lightning Cable" adapter with a picture of a VGA connector on it.

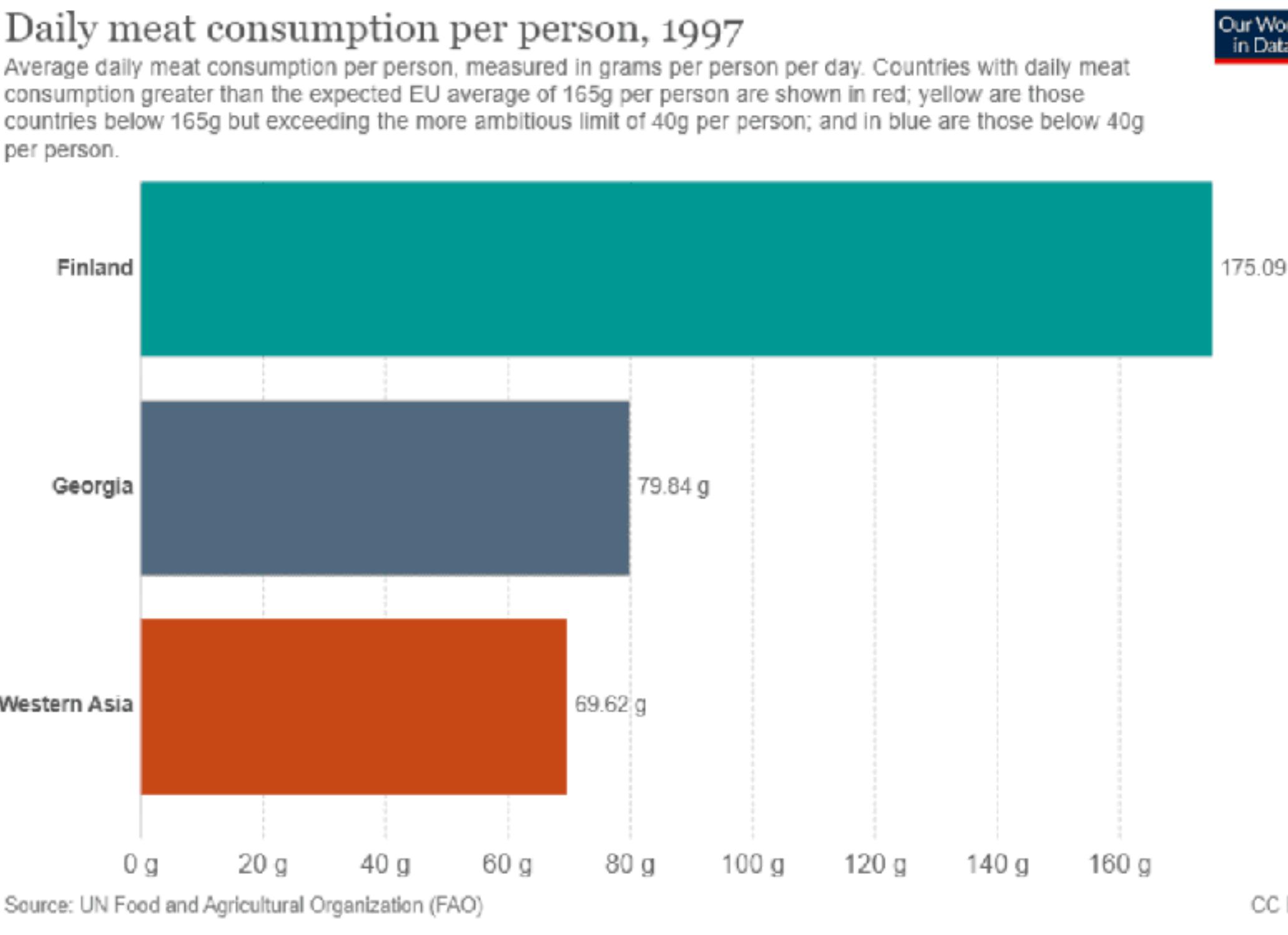
Panel 3: A close-up of the VGA connector with a small Lightning connector (used for charging iPhones and other Apple devices) at the end.

The humor in this image comes from the absurdity of plugging a large, outdated VGA connector into a small, modern smartphone charging port.

Visual inputs: chart reasoning

Sample 2 of 7

User What is the sum of average daily meat consumption for Georgia and Western Asia? Provide a step-by-step reasoning before providing your answer.



GPT-4 To find the sum of average daily meat consumption for Georgia and Western Asia, we will follow these steps:

1. Identify the average daily meat consumption for Georgia.
2. Identify the average daily meat consumption for Western Asia.
3. Add the values from steps 1 and 2.

Step 1: From the provided chart, we see that the average daily meat consumption for Georgia is 79.84 grams per person per day.

Step 2: From the chart, we see that the average daily meat consumption for Western Asia is 69.62 grams per person per day.

Step 3: Add the values from steps 1 and 2.

$$79.84 \text{ g (Georgia)} + 69.62 \text{ g (Western Asia)} = 149.46 \text{ g}$$

GPT-4 Performance Benchmarks

- OpenAI evaluated GPT-4 by simulating exams designed for humans, such as **the Uniform Bar Examination and LSAT for lawyers, and the SAT for university admission.**
- The results showed that GPT-4 achieved human-level **performance on various professional and academic benchmarks.**

Simulated exams	GPT-4 estimated percentile	GPT-4 (no vision) estimated percentile	GPT-3.5 estimated percentile
Uniform Bar Exam (MBE+MEE+MPT) ¹	298 / 400 ~90th	298 / 400 ~90th	213 / 400 ~10th
LSAT	163 ~88th	161 ~89rd	149 ~40th
SAT Evidence-Based Reading & Writing	710 / 800 ~93rd	710 / 800 ~93rd	670 / 800 ~87th
SAT Math	700 / 800 ~89th	690 / 800 ~89th	590 / 800 ~70th
Graduate Record Examination (GRE) Quantitative	163 / 170 ~80th	157 / 170 ~62nd	147 / 170 ~25th
Graduate Record Examination (GRE) Verbal	169 / 170 ~99th	165 / 170 ~96th	154 / 170 ~63rd
Graduate Record Examination (GRE) Writing	4 / 6 ~54th	4 / 6 ~54th	4 / 6 ~54th
USABO Semifinal Exam 2020	87 / 150 99th - 100th	87 / 150 99th - 100th	43 / 150 31st - 33rd
USNCO Local Section Exam 2022	36 / 60	38 / 60	24 / 60
Medical Knowledge Self-Assessment Program	75 %	75 %	53 %
Codeforces Rating	392 below 5th	392 below 5th	260 below 5th
AP Art History	5 86th - 100th	5 86th - 100th	5 86th - 100th
AP Biology	5 85th - 100th	5 85th - 100th	4 62nd - 85th
AP Calculus BC	4 43rd - 59th	4 43rd - 59th	1 0th - 7th

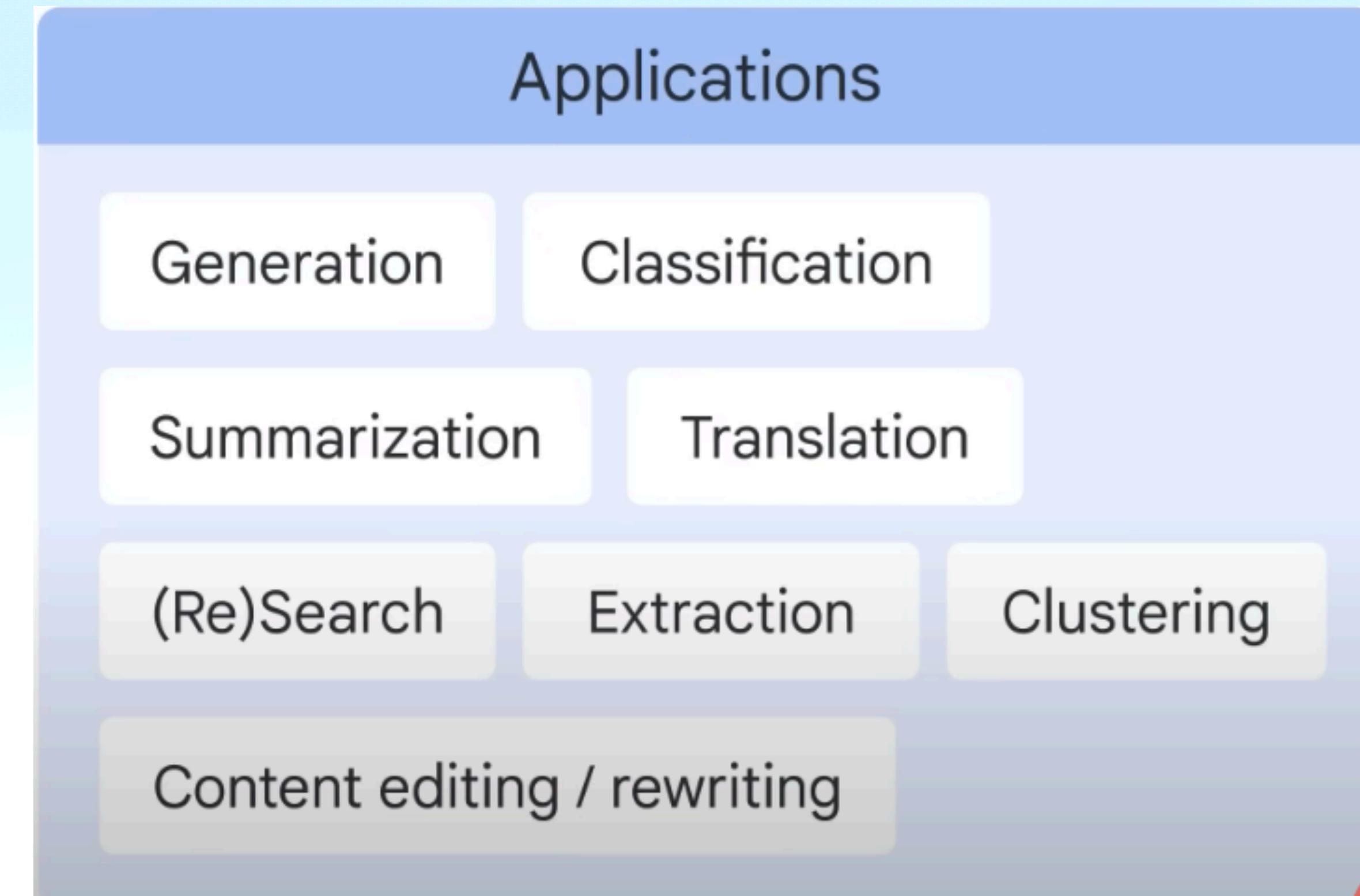
How LLM works



Model Types

text-to-text

Text-to-text models take a natural language input and produce text output. These models are trained to learn the mapping between a pair of texts (e.g. translation from one language to another).



Model Types

Text-to-image

Text-to-image models are relatively new and are trained on a large set of images, each captioned with a short text description. Diffusion is one method used to achieve this.

Applications

Image generation

Image editing

Model Types

Text-to-video

Text-to-3D

Text-to-video models aim to generate a video representation from text input. The input text can be anything from a single sentence to a full script, and the output is a video that corresponds to the input text.

Similarly Text-to-3D models generate three-dimensional objects that correspond to a user's text description (for use in game or other 3D worlds)

Applications

Video generation

Video editing

Game assets

Model Types

Text-to-task

Text-to-task models are trained to perform a specific task or action based on text input. This task can be a wide range of actions such as answering a question, performing a search, making a prediction, or taking some sort of action. For example, a text-to-task model could be trained to navigate web UI or make changes to a doc through the GUI.

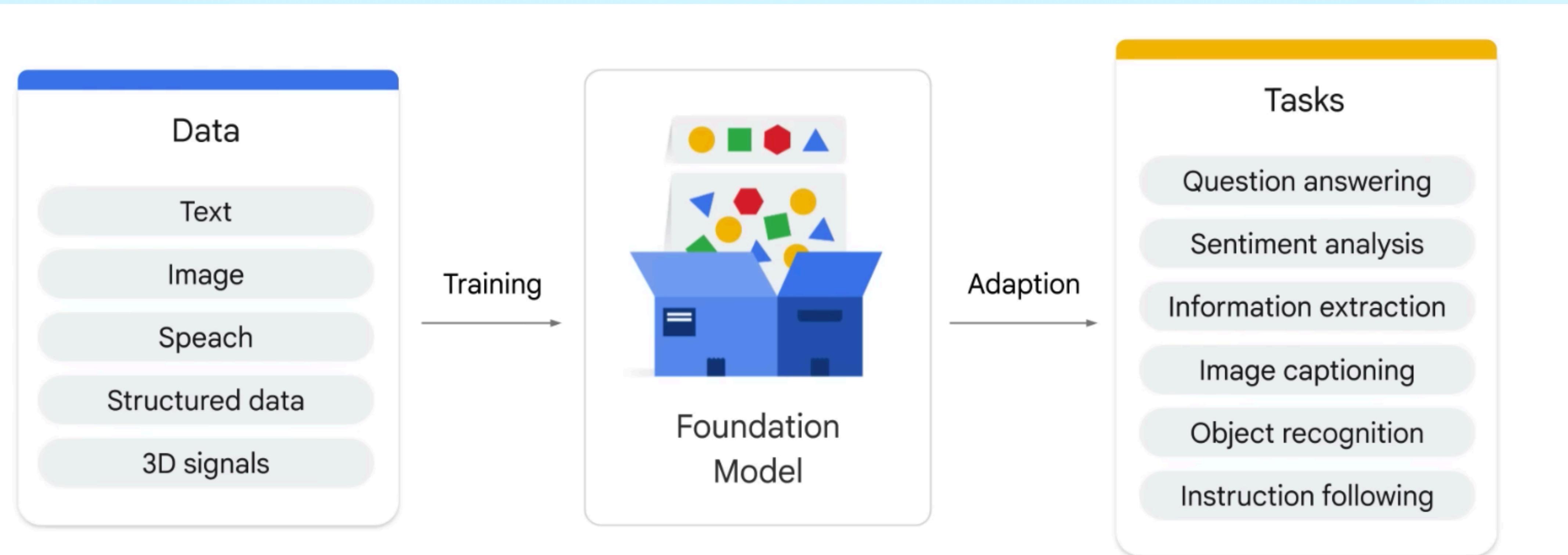
Applications

Software agents

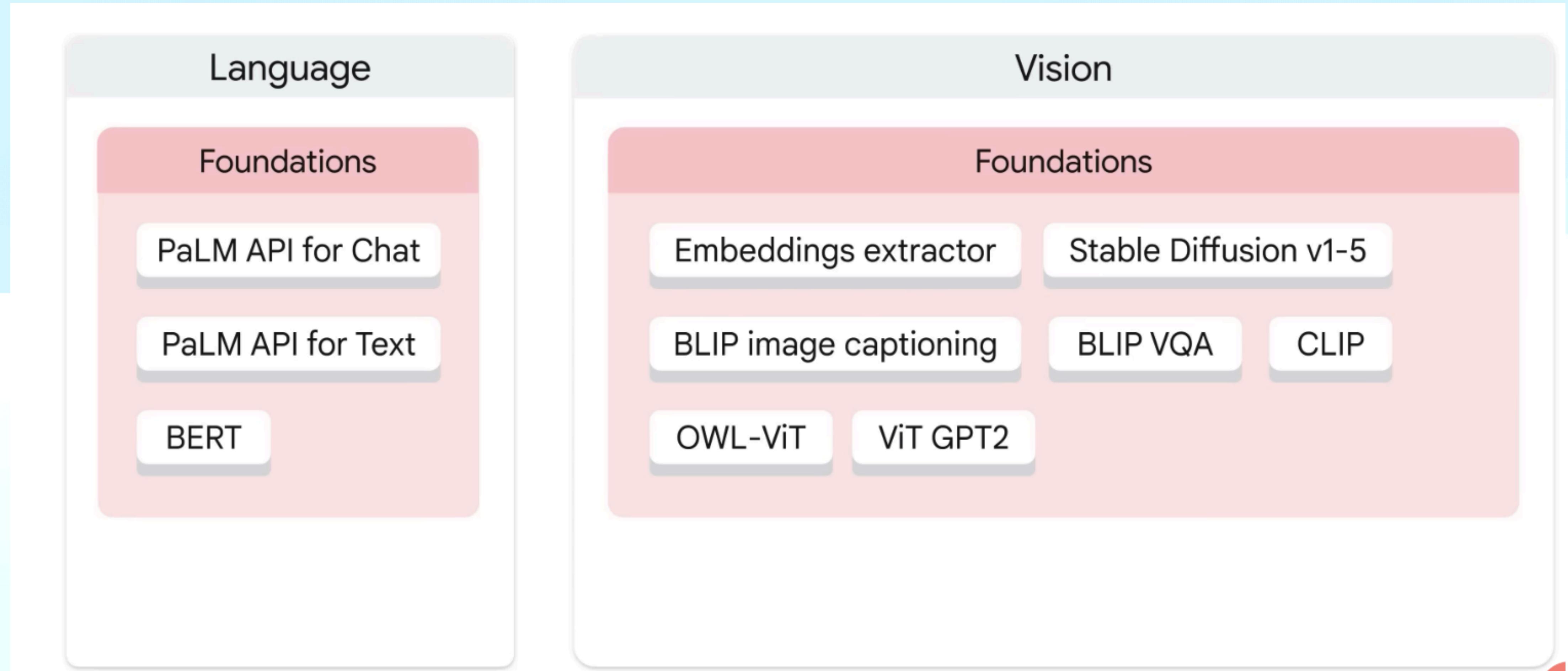
Virtual assistants

Automation

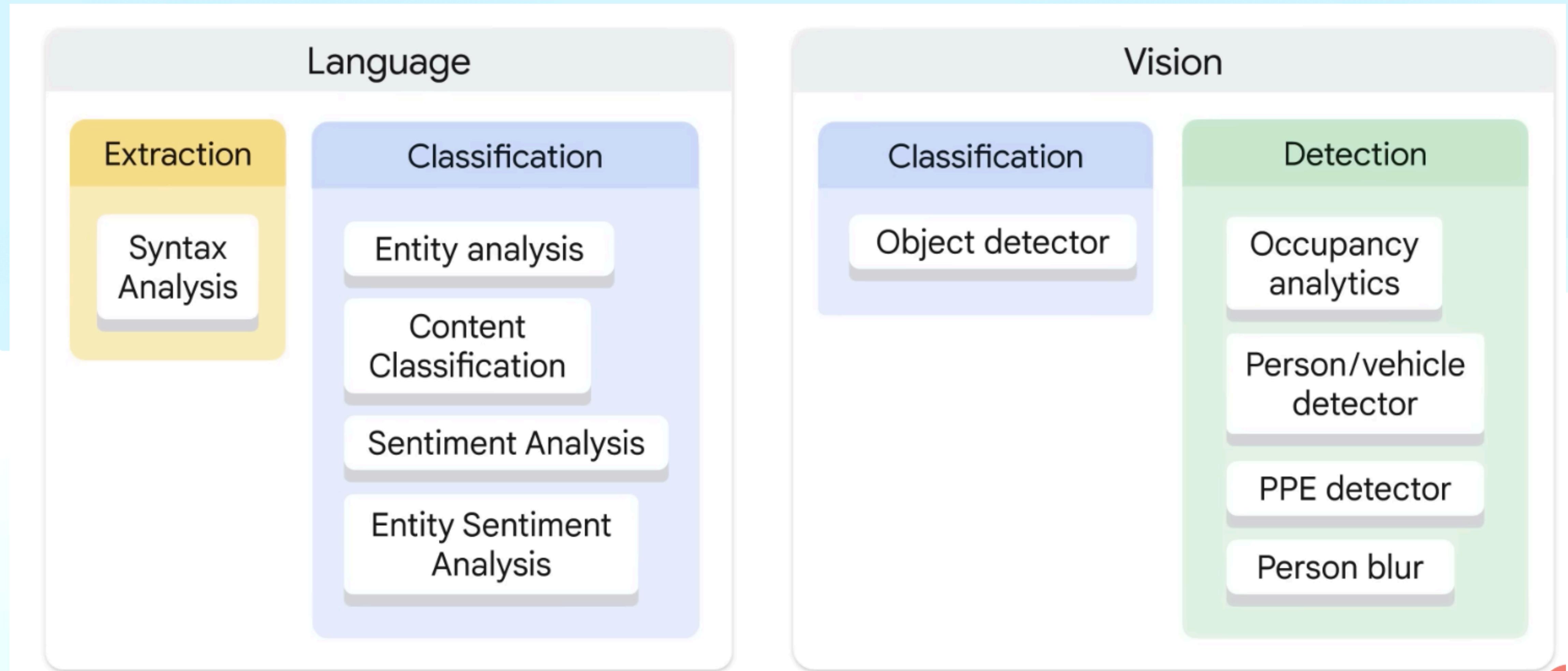
Foundation Model



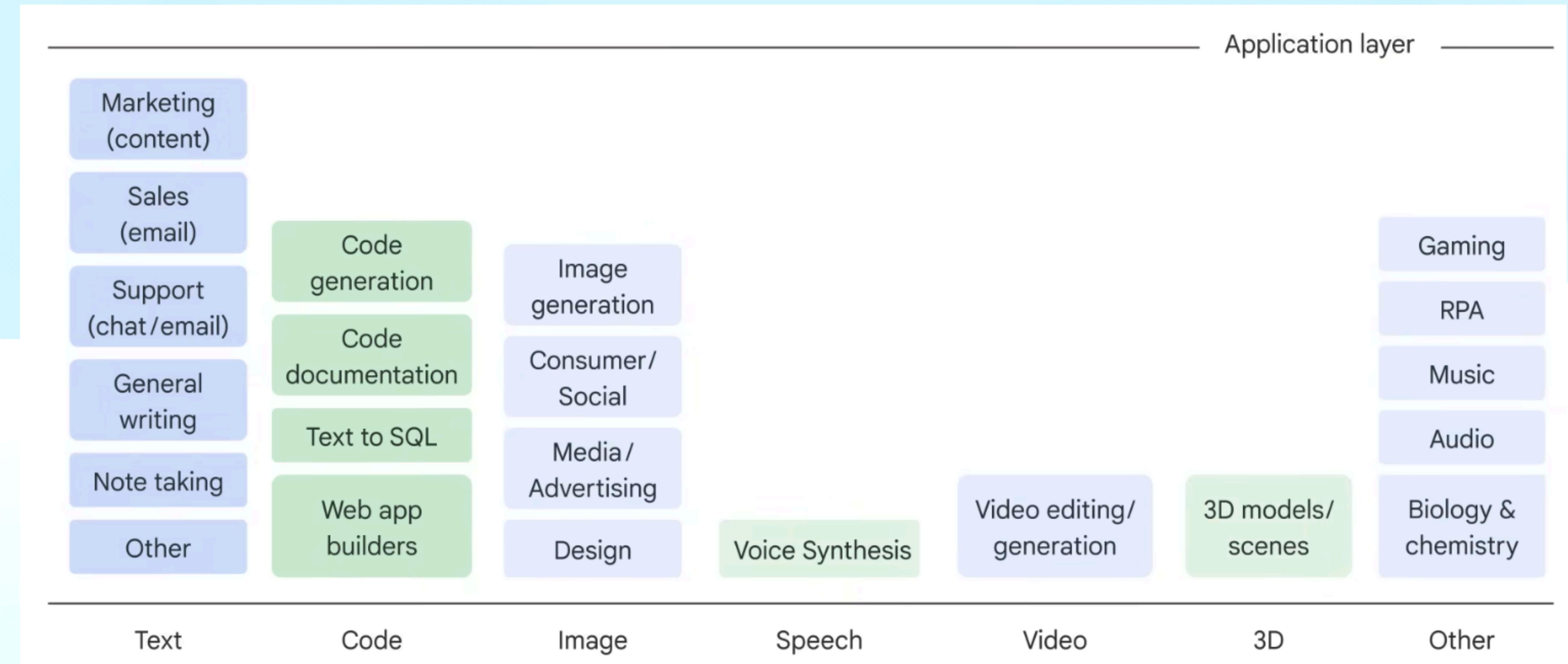
Foundation Model Example : Google Cloud



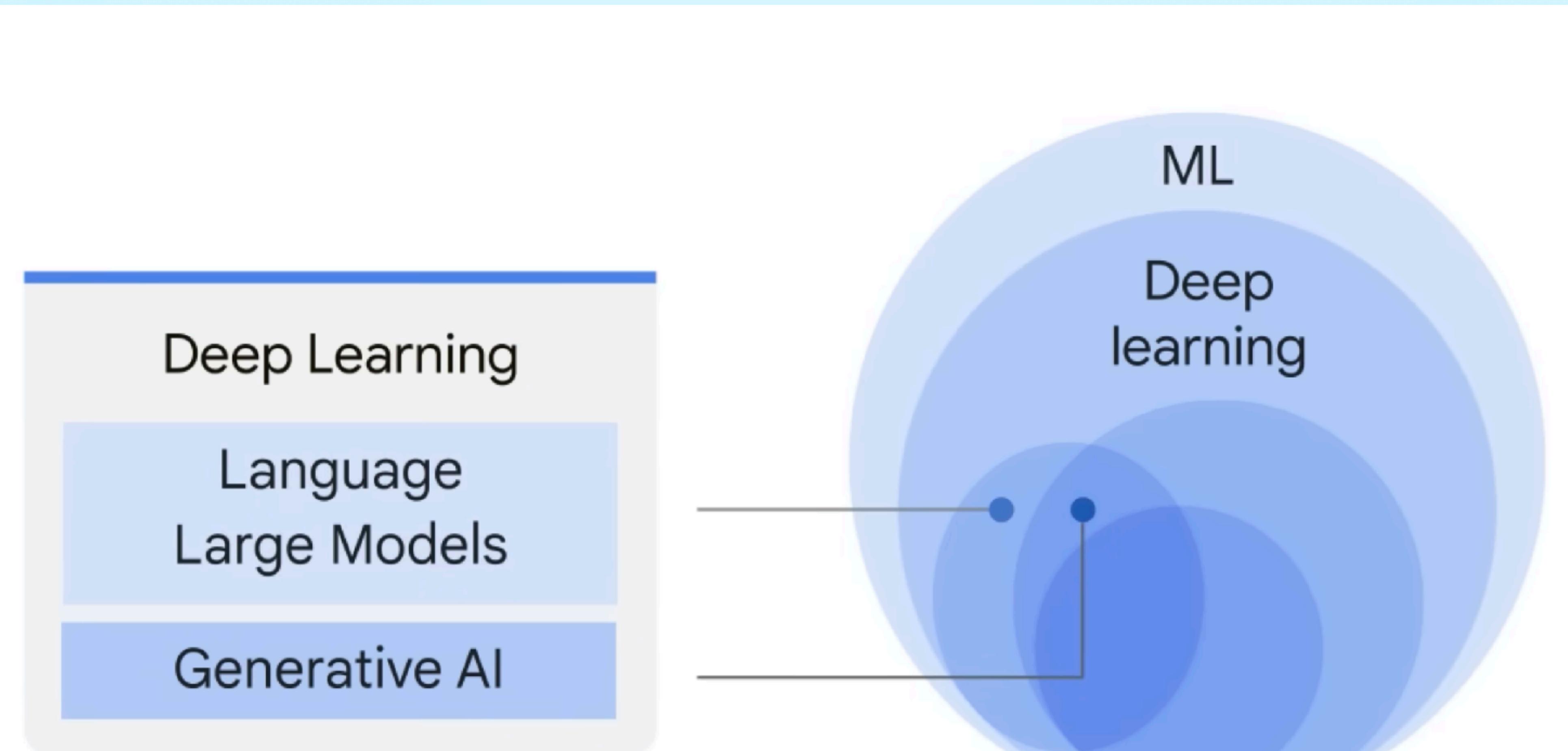
Foundation Model Examples : Google



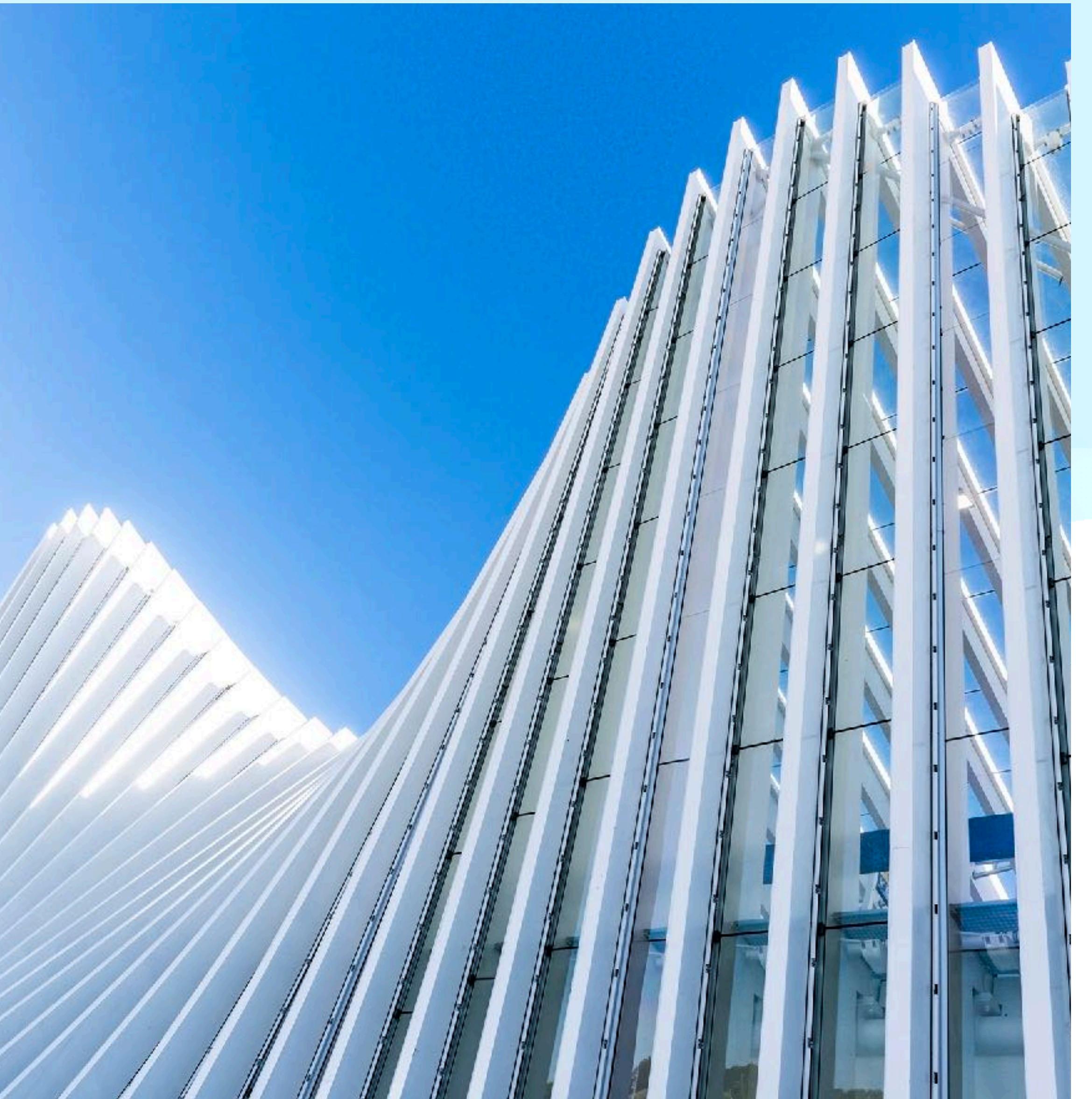
AI Application Landscape



Generative AI vs. Large Language Models



Getting on to GPT-3 and GPT-4



Here is an overview of the different API components and their functionalities:

Execution Engine: It determines the language model used for execution. Choosing the right engine is the key to determining your model's capabilities and in turn getting the right output.

Response Length: The response length sets a limit on how much text the API includes in its completion. Because OpenAI charges by the length of text generated per API call, response length is a crucial parameter for anyone on a budget. A higher response length will cost more.

Temperature: The temperature controls the randomness of the response, represented as a range from 0 to 1. A lower value of temperature means the API will respond with the first thing that the model sees; a higher value means the model evaluates possible responses that could fit into the context before spitting out the result.

Top P: Top P controls how many random results the model should consider for completion, as suggested by the temperature dial, thus determining the scope of randomness. Top P's range is from 0 to 1. A lower value limits creativity, while a higher value expands its horizons.

Here is an overview of the different API components and their functionalities:

Frequency and Presence Penalty: The frequency penalty decreases the likelihood that the model will repeat the same line verbatim by “punishing” it. The presence penalty increases the likelihood that it will talk about new topics.

Best of: This parameter lets you specify the number of completions (n) to generate on the server-side and returns the best of “n” completions.

Stop Sequence: A stop sequence is a set of characters that signals the API to stop generating completions.

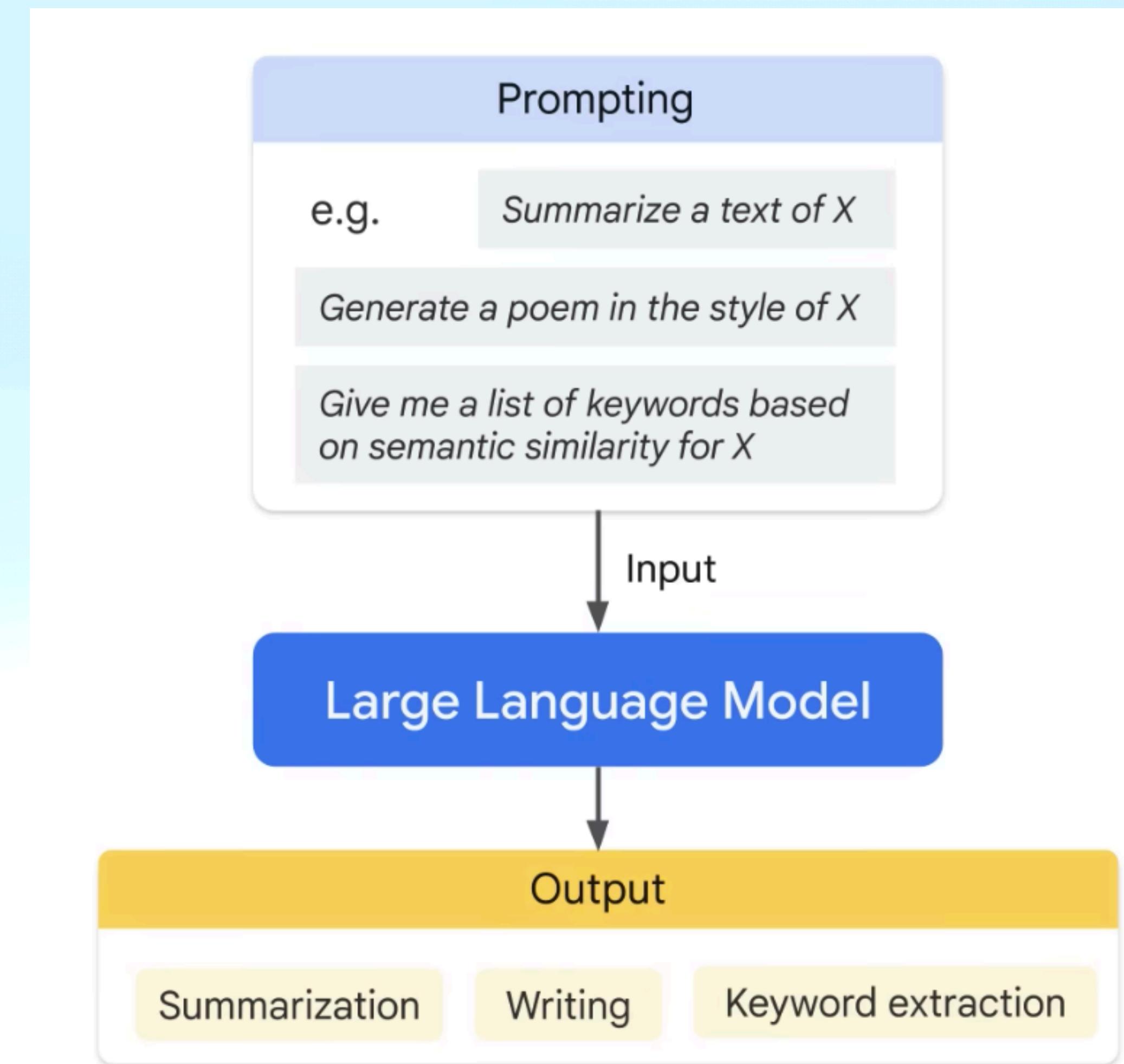
Inject Start & Restart Text: The inject start text and inject restart text parameters allow you to insert text at the beginning or end of the completion, respectively.

Show Probabilities: This option lets you debug the text prompt by showing the probability of tokens that the model can generate for a given input.

Execution Engines

The OpenAI API offers four different execution engines that differ in the number of parameters used, performance capabilities, and price. The primary engines in increasing order of their capabilities and size are **Ada** (named after Ada Lovelace), **Babbage** (named after Charles Babbage), **Curie** (named after Madame Marie Curie) and **Davinci** (named after Leonardo da Vinci).

Workflow for LLM



Model Tuning vs Prompt Tuning vs Prompt Engineering

Model Tuning (Fine-Tuning):

- This involves continuing the training of a pre-trained model on a task-specific dataset.
- This updates the model's parameters (or weights) to optimize the model's performance on the specific task.
- The advantage of fine-tuning is that it can significantly improve the model's performance on task-specific or domain-specific applications.
- The downside is that it can be computationally expensive and may require a substantial amount of labeled data.

Model Tuning vs Prompt Tuning vs Prompt Engineering

Prompt Tuning:

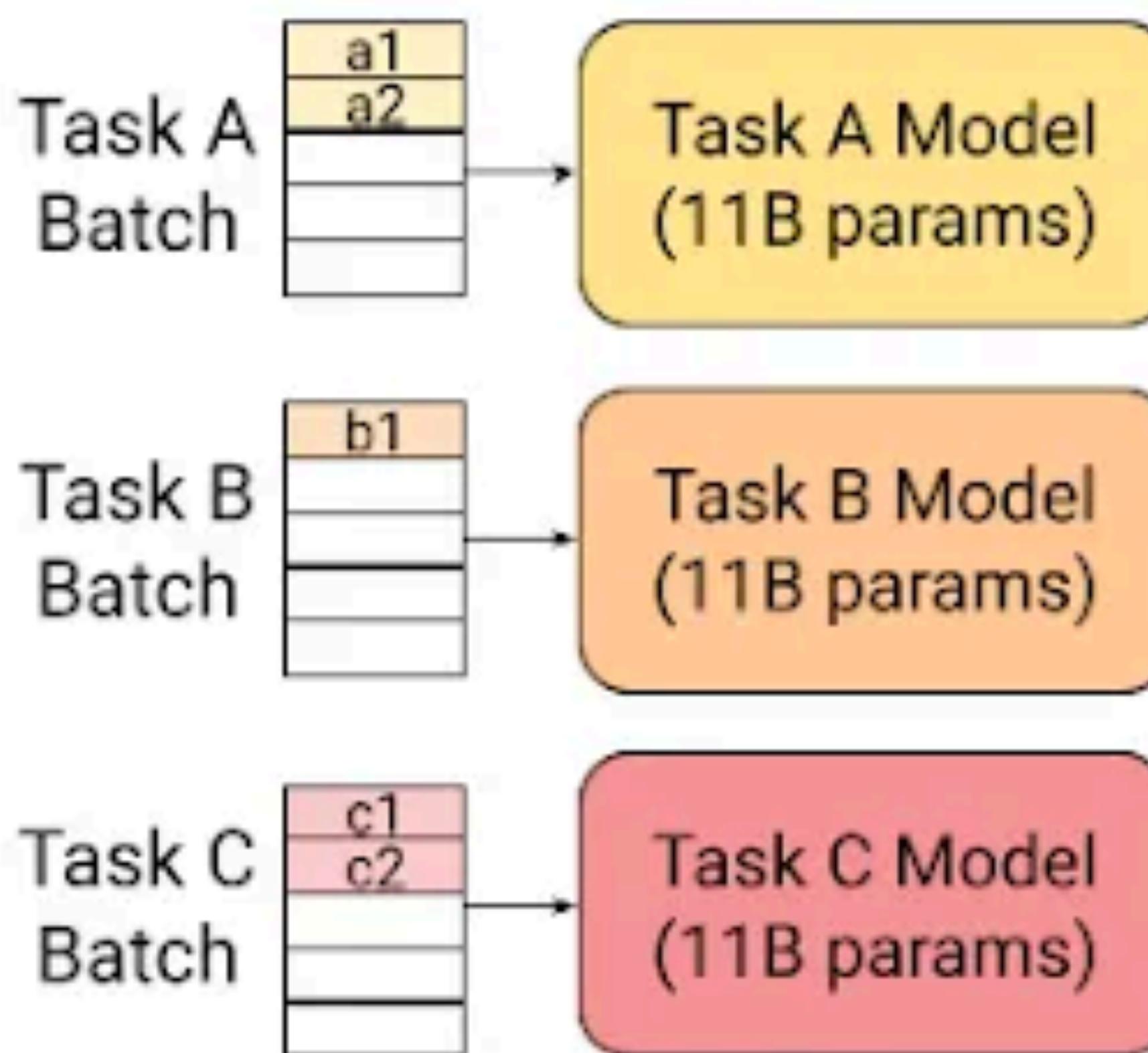
- Prompt tuning is a method where you don't modify the model itself, but instead, optimize the input prompt given to the model.
- In other words, it's about finding an optimal way to phrase or structure your prompt to get the desired output from the model.
- This is less computationally expensive than fine-tuning and can be effective in many cases.
- However, it may not lead to as large improvements as fine-tuning, especially for complex or highly domain-specific tasks.

Model Tuning vs Prompt Tuning vs Prompt Engineering

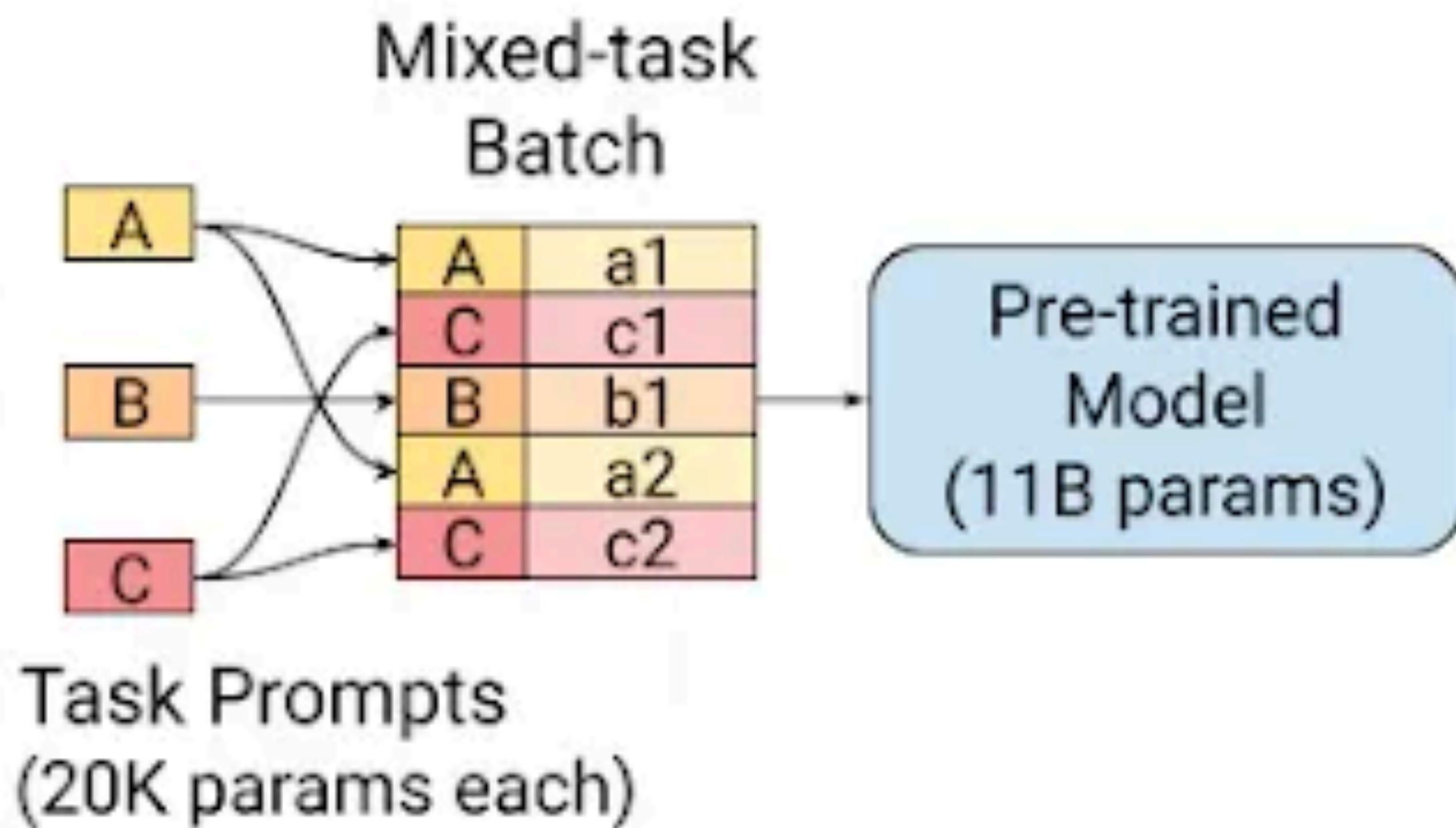
Prompt Engineering:

- Prompt engineering is a manual, heuristic approach to crafting effective prompts.
- It involves creatively designing prompts based on your understanding of how the model works and what kind of prompts it responds to well.
- This is a somewhat experimental approach and requires a lot of trial and error.
- However, it can be effective when done well, and it requires no additional training or computational resources.

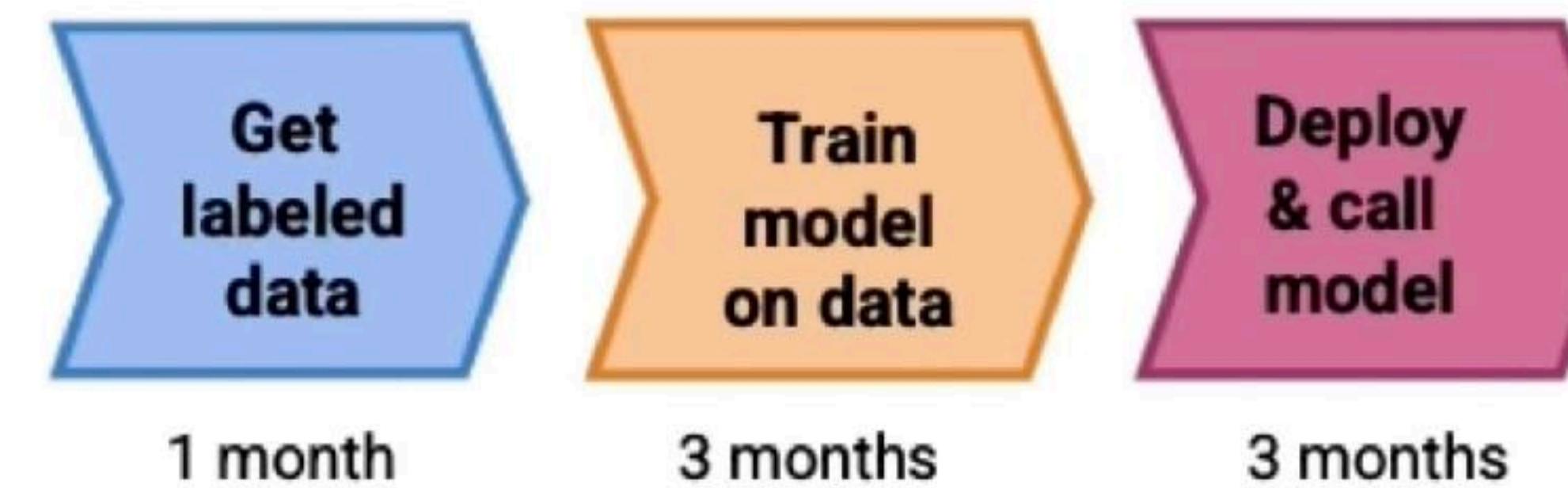
Model Tuning



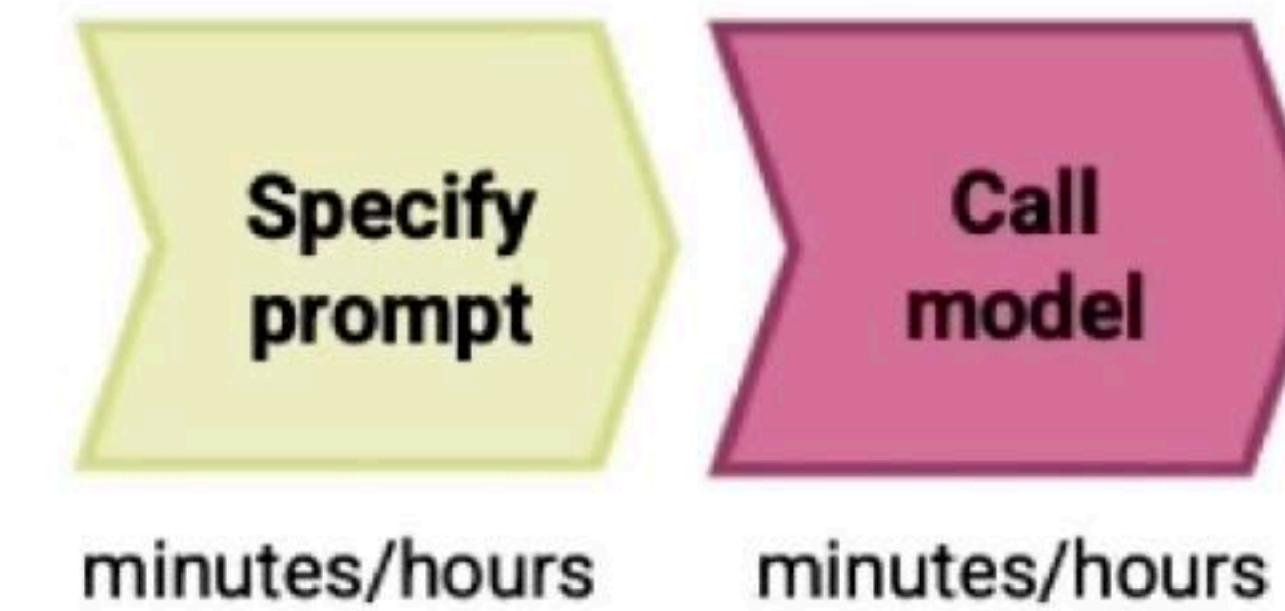
Prompt Tuning



Supervised learning

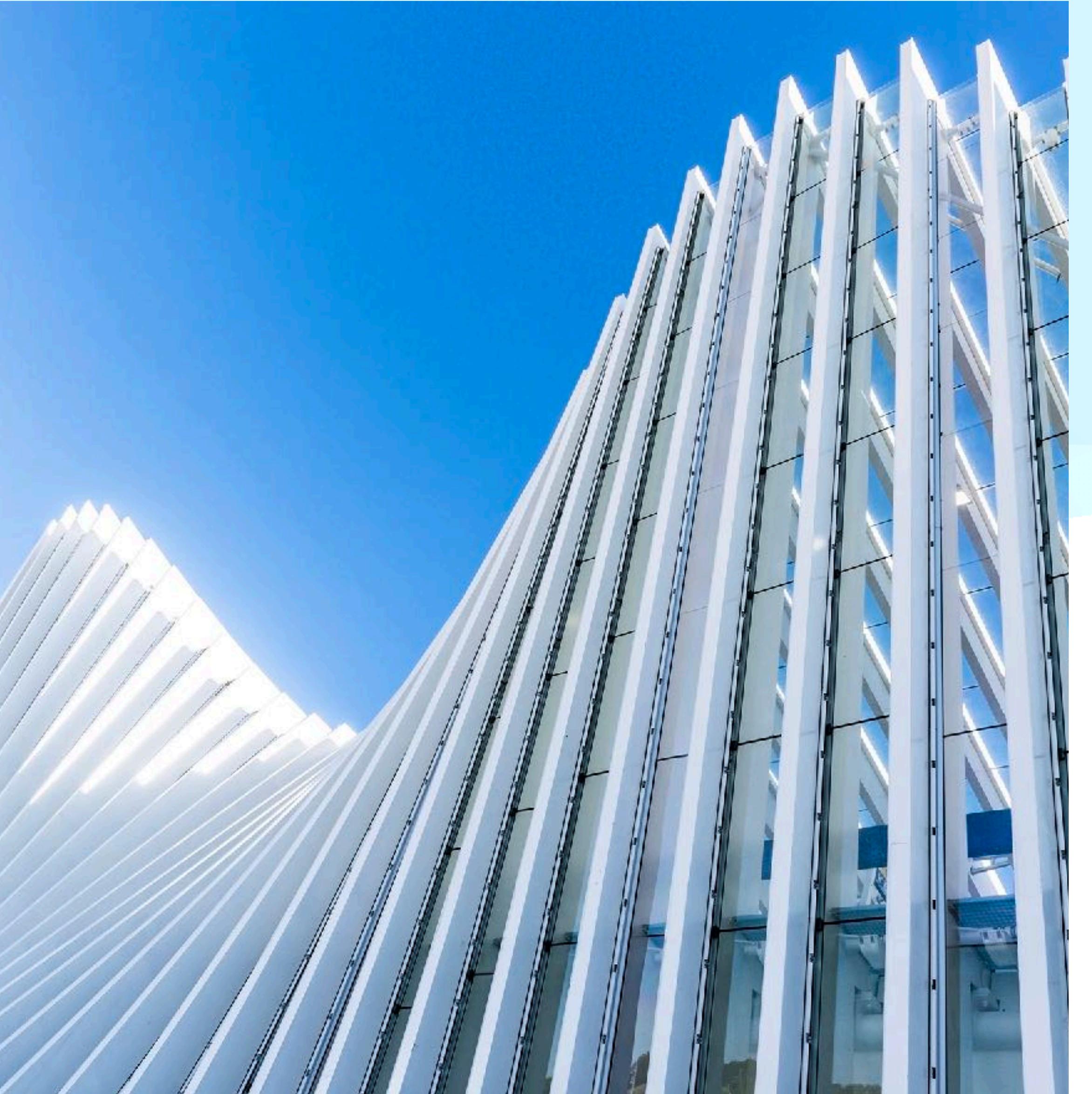


Prompt-based AI

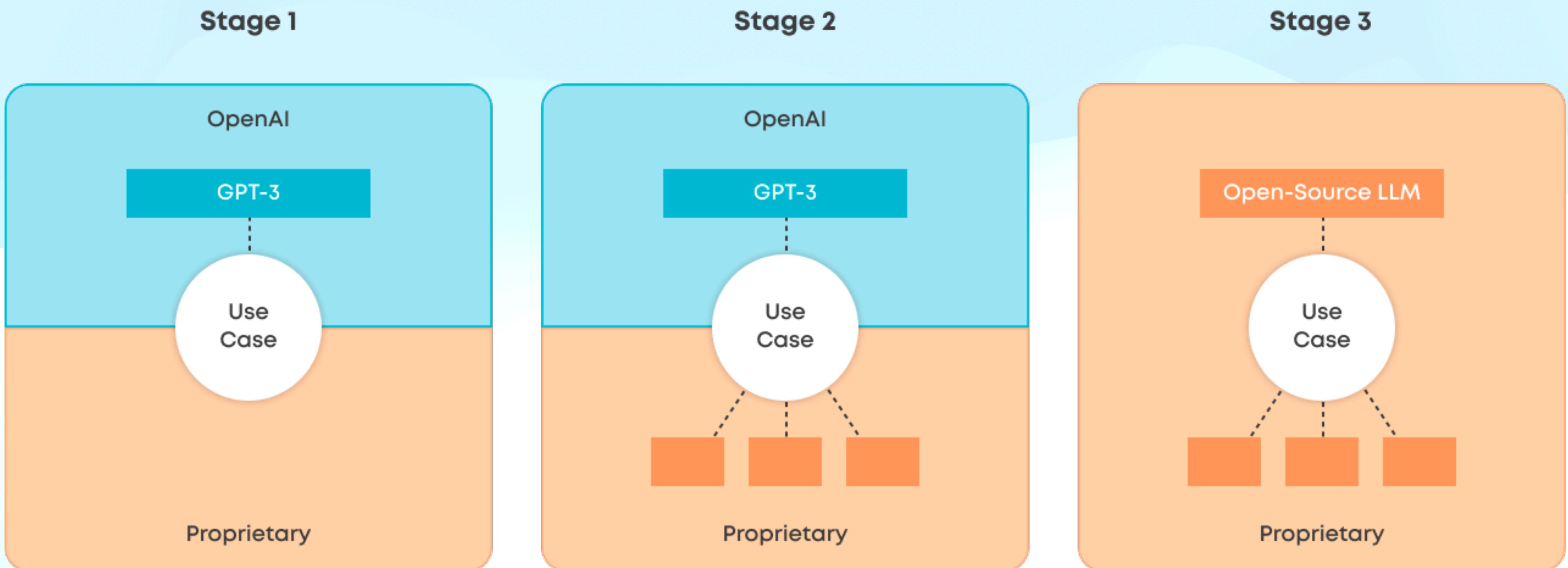


Workshop # 6 : Build LLM application using OpenAI API

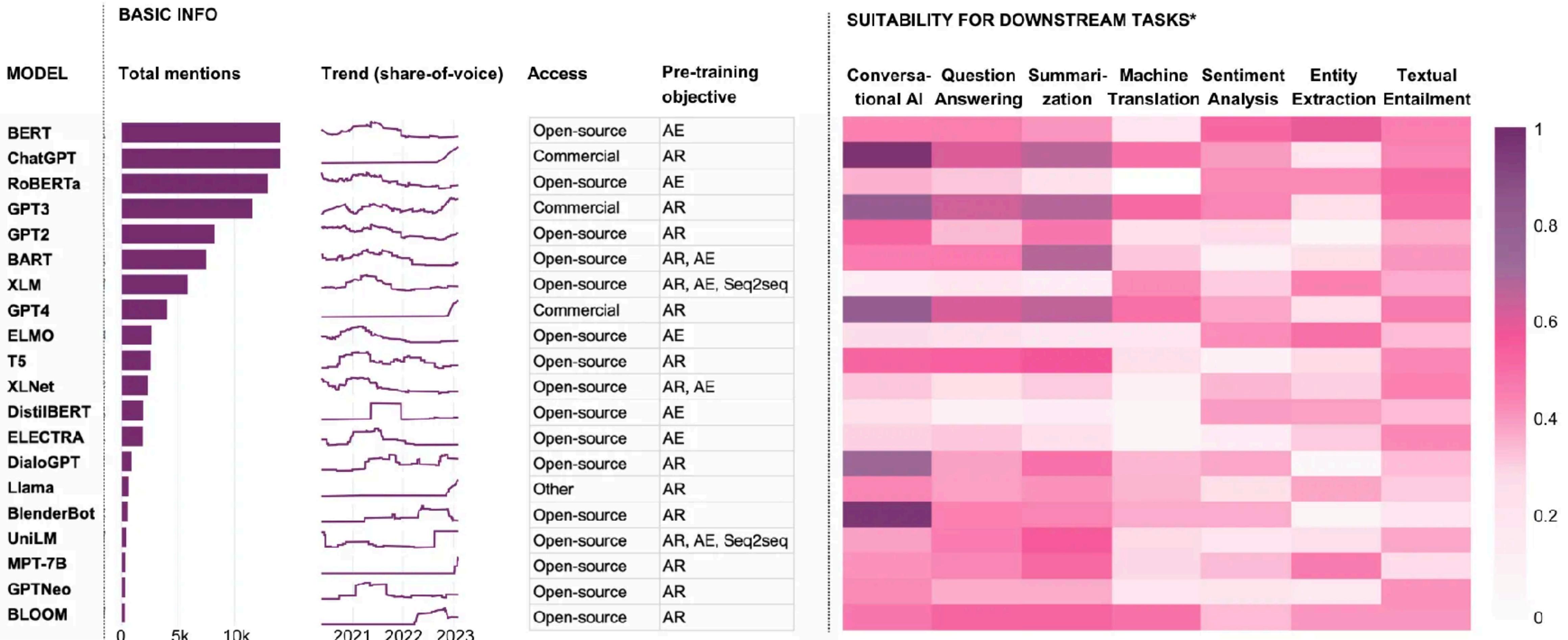
Opensource Alternative to GPT



Recipe for Building Application using LLM



OVERVIEW OVER THE MOST POPULAR LLMS (May 2023)



Opensource LLM : Falcon LLM

- **Falcon LLM is a 40 billion parameter model** that has been trained on a massive dataset of text and code. It is the newest and the greatest addition to the growing list of open-source LLMs. To promote collaboration and drive innovation the Falcon LLM, its training data, and weights are open-sourced. It comes as yet another threat to the dominance of ChatGPT and Bard which are restrictive AI models and are very expensive to inference from external sources.
- Having Falcon open-sourced means that it gives the developers the freedom to develop more innovative solutions with greater freedom, less cost and avoid making api calls to external sources for inferencing.

Opensource LLM : Cerebras GPT

- **Cerebras** has recently announced their latest innovation: Cerebras GPT, the highest performing language processing tool available today. Building upon the foundations established by leading academic and industrial efforts such as BERT, GPT-2/GPT-J, and Gopher, Cerebras GPT breaks ground in accelerated natural language understanding and generation. Drawing upon the company's expertise in ASIC design and assembly of proprietary chipsets optimized for matrix operations, Cerebras achieves 7x inference speedup over publicly reported PyTorch implementations.
- Furthermore, the Cerebras Research Group has developed custom models and libraries tailored towards efficient deployment and scaling of neural network models on the platform. As a result, developers and research teams alike can harness state-of-the-art language processing capabilities, empowering a new wave of AI systems capable of tackling complex human-centric interactions.
- From conversational AIs and virtual assistants to advanced content creation tools and recommendation engines, the possibilities enabled by Cerebras GPT are endless, laying the foundation for a brighter, smarter, and more collaborative digital future.
- All Cerebras-GPT models are available on Hugging Face.
- The family includes 111M, 256M, 590M, 1.3B, 2.7B, 6.7B, and 13B models. License: Apache 2.0. Architecture: GPT-3 style architecture

Opensource LLM : Google Flan T5

- **Google Flan T5** is another impressive top Opensource LLM models and is a creation from the search giant that has garnered significant attention in recent months. It stands apart from traditional language models because of its novel architecture that relies on tensor factorization instead of autoregressive masking mechanisms.
- This innovative design makes it significantly faster, cheaper, and simpler to train, allowing organizations to generate high-quality responses under resource constraints that were previously thought impossible. As a result, T5 offers improved speed, higher quality, and better control over various generation tasks, such as translation, question answering, text classification, summarization, code completion, and others.
- In essence, it delivers exceptional results at scale while retaining competitive accuracy compared to industry benchmarks, solidifying Flan and T5's place among top-tier solutions for AI and NLP applications.
- T5 comes in different sizes:
 - t5-small
 - t5-base
 - t5-large
 - t5-3b
 - t5-11b.

Opensource LLM : GPT NeoX 20B

- **GPT Neo XL 20B** builds upon the already impressive capabilities of the original GPT Neo architecture. With four times the number of parameters and almost twice the number of tokens, the newest iteration sets a new bar in the field of language generation and comprehension.
- Boasting improved precision and recall, enhanced context handling abilities, and the ability to generate novel answers in response to open-ended questions, GPT Neo XL is becoming the go-to choice for organizations looking to integrate cutting-edge NLP into their workflows.
- GPT-NeoX-20B can be loaded using the ***AutoModelForCausalLM*** functionality:

Opensource LLM : FastChat-T5 Model Card

- **FastChat T5** is a compact and commercial friendly chatbot trained by finetuning Flan-t5-xl (3B parameters) on user-shared 70K conversations collected from ShareGPT.
- It is a language model trained by researchers from Large Model Systems Organization (LMSYS). Its purpose is to assist with various natural language processing tasks such as answering questions, providing information, and generating text.
- It is based on an encoder-decoder transformer architecture and can autoregressively generate responses to users' inputs. It was released in April 2023. by The FastChat developers.



Introduction to LLaMA 2

- LLaMA 2 is an **auto-regressive large language model** developed by Meta AI. It's based on an **optimized transformer architecture** and is open-source for both commercial and non-commercial use.
- The model **comes in sizes with 7, 13, 34, or 70 billion parameters**, and it was trained on **2 trillion tokens** with over **a million human annotations**.
- LLaMA 2 can handle a **context length of about 4000 tokens**, doubling the capability of its predecessor.
- A fine-tuned variant, LLaMA 2-Chat, has been developed specifically for chat applications.

Performance and Training of LLaMA 2

- LLaMA 2 performs excellently compared to other open-source models such as Falcon-40B or MosaicML's MPT, leading the Hugging Face Open LLM Leaderboard.

Model	Size	Code	Commonsense Reasoning	World Knowledge	Reading Comprehension	Math	MMLU	BBH	AGI Eval
MPT	7B	20.5	57.4	41.0	57.5	4.9	26.8	31.0	23.5
	30B	28.9	64.9	50.0	64.7	9.1	46.9	38.0	33.8
Falcon	7B	5.6	56.1	42.8	36.0	4.6	26.2	28.0	21.2
	40B	15.2	69.2	56.7	65.7	12.6	55.4	37.1	37.0
LLAMA 1	7B	14.1	60.8	46.2	58.5	6.95	35.1	30.3	23.9
	13B	18.9	66.1	52.6	62.3	10.9	46.9	37.0	33.9
	33B	26.0	70.0	58.4	67.6	21.4	57.8	39.8	41.7
	65B	30.7	70.7	60.5	68.6	30.8	63.4	43.5	47.6
LLAMA 2	7B	16.8	63.9	48.9	61.3	14.6	45.3	32.6	29.3
	13B	24.5	66.9	55.4	65.8	28.7	54.8	39.4	39.1
	34B	27.8	69.9	58.7	68.0	24.2	62.6	44.1	43.4
	70B	37.5	71.9	63.6	69.4	35.2	68.9	51.2	54.2

Table 3: Overall performance on grouped academic benchmarks compared to open-source base models.

Performance and Training of LLaMA 2

- When compared to proprietary models like OpenAI's GPT models, LLaMA 2 shows room for improvement, but it outperforms GPT-3.5 in some benchmarks, like the HellaSwag benchmark.

Benchmark (shots)	GPT-3.5	GPT-4	PaLM	PaLM-2-L	LLAMA 2
MMLU (5-shot)	70.0	86.4	69.3	78.3	68.9
TriviaQA (1-shot)	–	–	81.4	86.1	85.0
Natural Questions (1-shot)	–	–	29.3	37.5	33.0
GSM8K (8-shot)	57.1	92.0	56.5	80.7	56.8
HumanEval (0-shot)	48.1	67.0	26.2	–	29.9
BIG-Bench Hard (3-shot)	–	–	52.3	65.7	51.2

Table 4: Comparison to closed-source models on academic benchmarks. Results for GPT-3.5 and GPT-4 are from OpenAI (2023). Results for the PaLM model are from Chowdhery et al. (2022). Results for the PaLM-2-L are from Anil et al. (2023).

Performance and Training of LLaMA 2

- The training process of LLaMA 2 involved **pre-training on publicly available online sources, supervised fine-tuning with the help of human experts, and reinforcement learning from human feedback (RLHF) for iterative refinement.**
- Alongside the LLaMA 2 release, Microsoft announced its expanded partnership with Meta to support the LLaMA 2 family of large language models on Azure and Windows.

Implications and Potential of LLaMA 2

- Llama 2 is free for **research and commercial use**.
- The release of LLaMA 2 opens up a multitude of potential applications that can leverage its advanced features and capabilities.
- Meta's move to provide almost free availability and openness with LLaMA 2 is highly commendable and is expected to lead to the development of new, more powerful models or applications.

Demo using open source LLM

See Jupyter Notebook

Building a private LLM

Building a large language model is a complex task requiring significant computational resources and expertise. There is no single “correct” way to build an LLM, as the specific architecture, training data and training process can vary depending on the task and goals of the model.

How Do You Train LLMs from Scratch?

Step

1. Continuing the Text

The training process of the LLMs that continue the text is known as pretraining LLMs. These LLMs are trained in self-supervised learning to predict the next word in the text. We will exactly see the different steps involved in training LLMs from scratch.

- a. Dataset Collection
- b. Dataset Preprocessing
- c. Dataset Preparation

Input	Output
DHS	stands
DHS stands	for
DHS stands for	DataHack
DHS stands for DataHack	Summit

How Do You Train LLMs from Scratch?

Step

1. Continuing the Text

The training process of the LLMs that continue the text is known as pretraining LLMs. These LLMs are trained in self-supervised learning to predict the next word in the text. We will exactly see the different steps involved in training LLMs from scratch.

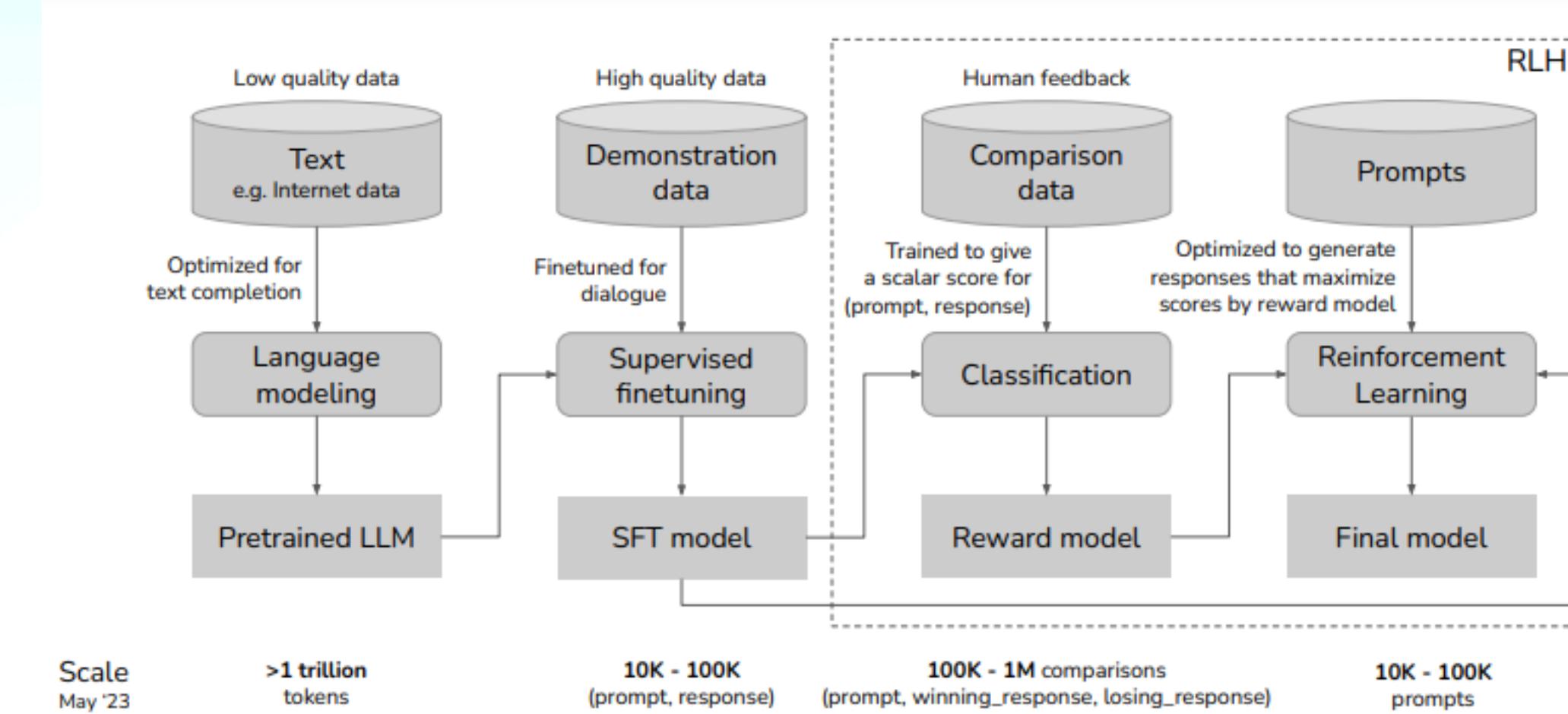
- a. Dataset Collection
- b. Dataset Preprocessing
- c. Dataset Preparation
- d. Model Architecture
- e. Hyperparameter Search

Input	Output
DHS	stands
DHS stands	for
DHS stands for	DataHack
DHS stands for DataHack	Summit

How Do You Train LLMs from Scratch?

2. Dialogue-optimized LLMs

In the dialogue-optimized LLMs, the first step is the same as the pretraining LLMs discussed above. After pretraining, these LLMs are now capable of completing the text. Now, to generate an answer for a specific question, the LLM is finetuned on a supervised dataset containing questions and answers. By the end of this step, your model is now capable of generating an answer to a question.



Introduction to LangChain



Large Language Models (LLMs) like OpenAI's GPT, Google's BERT, and Meta's LLaMA are transforming industries with their ability to generate varied text outputs.

LangChain is a framework that enhances software developers' and data engineers' productivity when integrating LLM-based AI into applications and data pipelines.

It leverages APIs of several LLMs, creating an accessible and efficient way to integrate AI functionality into software.

Problems Addressed by LangChain

Problem 1: Boilerplate Text in Prompts - Redundancy in creating prompts for LLMs.

Solution: LangChain provides **prompt templates**, simplifying the task of writing prompts.

Problem 2: Unstructured Responses - Responses from LLMs are unstructured, making them harder to process.

Solution: LangChain uses output **parser tools** to convert the text response into structured data.

Problem 3: Difficulty in Switching Between LLMs - Without an abstraction layer, it's difficult to switch between different LLMs.

Solution: LangChain introduces an **LLM class abstraction**, enabling easy swapping or integration of multiple models.

Problems Addressed by LangChain

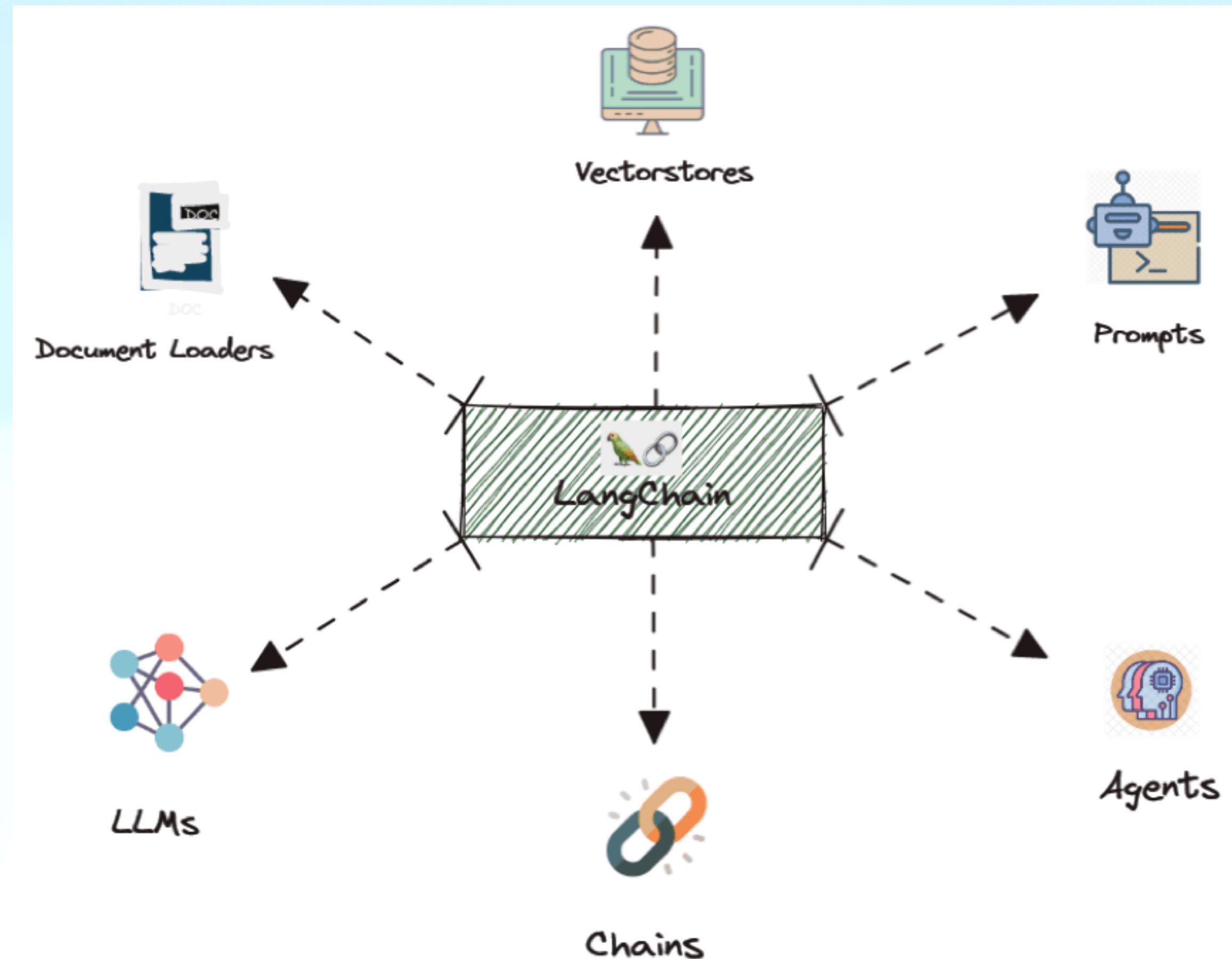
Problem 4: Short Memory of LLMs - LLMs have limited memory, affecting their ability to maintain consistency in long conversations.

Solution: LangChain's chat message history tools extend the memory of LLMs, leading to improved consistency.

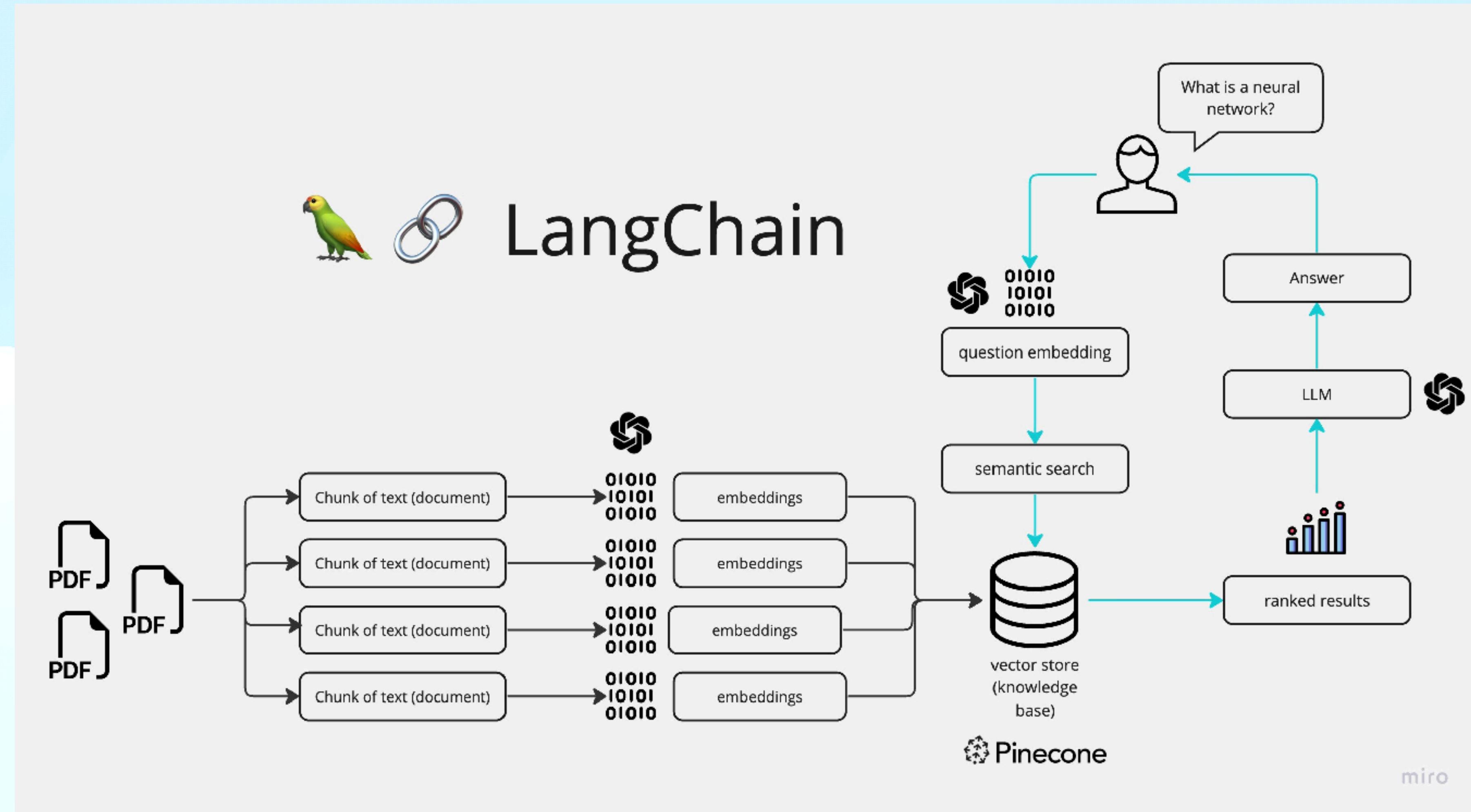
Problem 5: Difficulties in LLM Integration - It's challenging to integrate LLMs into data pipelines and software applications.

Solution: LangChain provides tools for pipeline-type workflows, streamlining the integration process.

Problems Addressed by LangChain



LangChain Application Architecture



Demo and workshop on LangChain



