



National Engineering School of Tunis
University of Tunis El Manar

Chapter Two

Large Language Models : NLP Tokens, Word Embeddings, and Transformers

Mariem ZAOUALI
mariem.zaouali@enit.utm.tn

Powered by:



Plan



Séance 1 Chapter 1 : Introduction to Generative AI

Séance 2 Chapter 2 : Large Language Models – Hands-On

Séance 3-4 Chapter 3 : Building Applications + Certif

Séance 3-4 Hands-On : Prompt Engineering -First Assessment – Challenge revealed

Séance 5 Hands-On : LLM Orchestration – Agentic AI + Certif

Séance 6 Chapter 4 : Advanced Topics and Closing thoughts + Certif

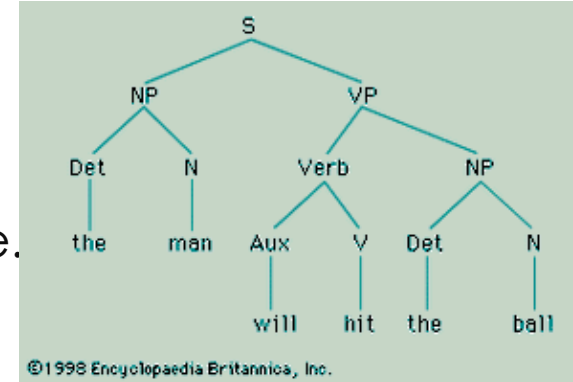
Séance 7 Challenge validation and Final Assessment

Introduction to NLP: History and Applications

1950s – 1960s: Rules-based

Challenges:

- Scalability issues due to manual rule creation.
- Limited ability to handle ambiguity in natural language.



1960s – 1980s: Statistical Methods

Challenges:

- Dependence on large labeled datasets.
- Difficulty handling long-range dependencies in language.

N-Gram

N=1: This is a sentence

Uni-grams

this,
is,
a,
sentence

N=2: This is a sentence

Bi-grams

this is,
is a,
a sentence

N=3: This is a sentence

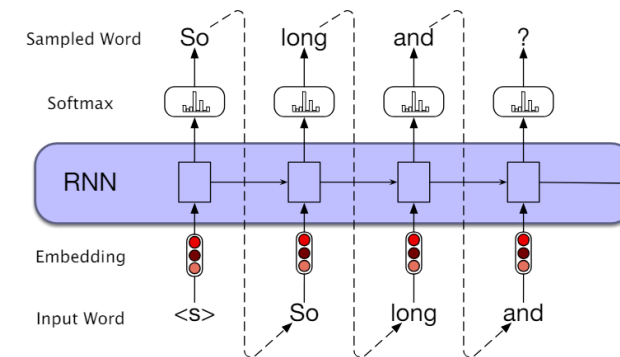
Tri-grams

this is a,
is a sentence

1990s – 2010s: Machine Learning Methods

Challenges:

- Difficulty scaling RNNs to handle very long sequences.
- Training inefficiencies for deep networks.



Classical NLP Methods - Limitations

1. Lexical Ambiguity

Words with multiple meanings (e.g., “bank” as a financial institution vs. river bank).

Syntactic Ambiguity:

Different ways to parse a sentence (e.g., “I saw the man with a telescope”).

Contextual Limitations:

Classical methods often struggle with determining meaning based on context.

2. Fixed-Size Context Windows:

N-gram models and early statistical methods consider only short-term dependencies.



Classical NLP Methods - Limitations

3. Handling Out-of-Vocabulary (OOV) Words

Vocabulary Limitations:

Classical methods struggle with unseen words or variations (e.g., “tweeted” vs. “tweet”).

Morphological Variations:

Inflexible handling of word forms (e.g., “run,” “running,” “ran”).

4. Difficulty in Capturing Semantics

Surface-Level Understanding:

Classical methods focus on syntax and word frequency rather than meaning.

No Contextual Word Representation:

Words are treated independently, missing nuances and context (e.g., “bat” as an animal vs. sports equipment).



Classical NLP Methods - Limitations

5. Poor Scalability with Large Data

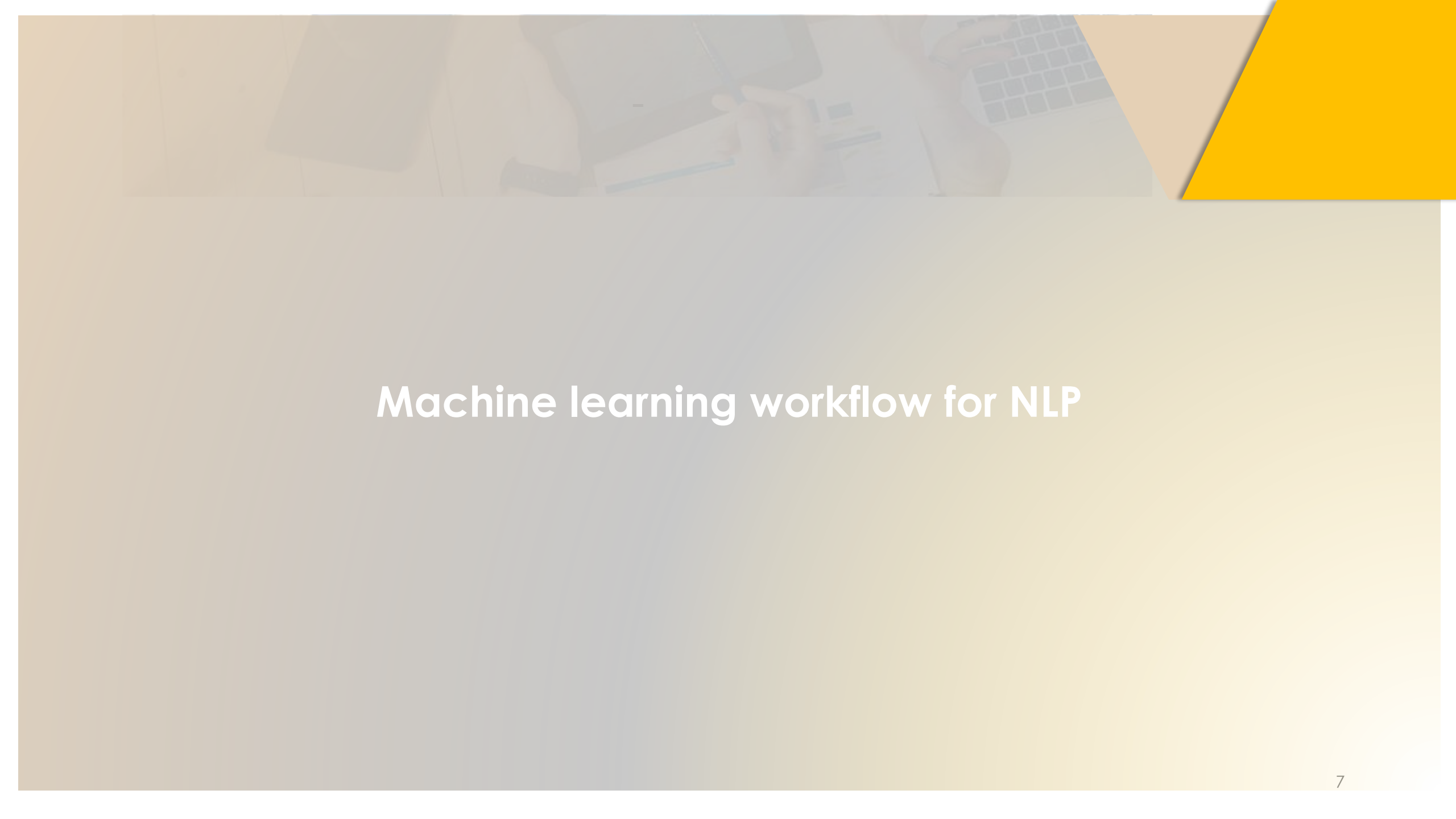
Rule Explosion:

As data size grows, maintaining and expanding rule sets becomes impractical.

Statistical Models:

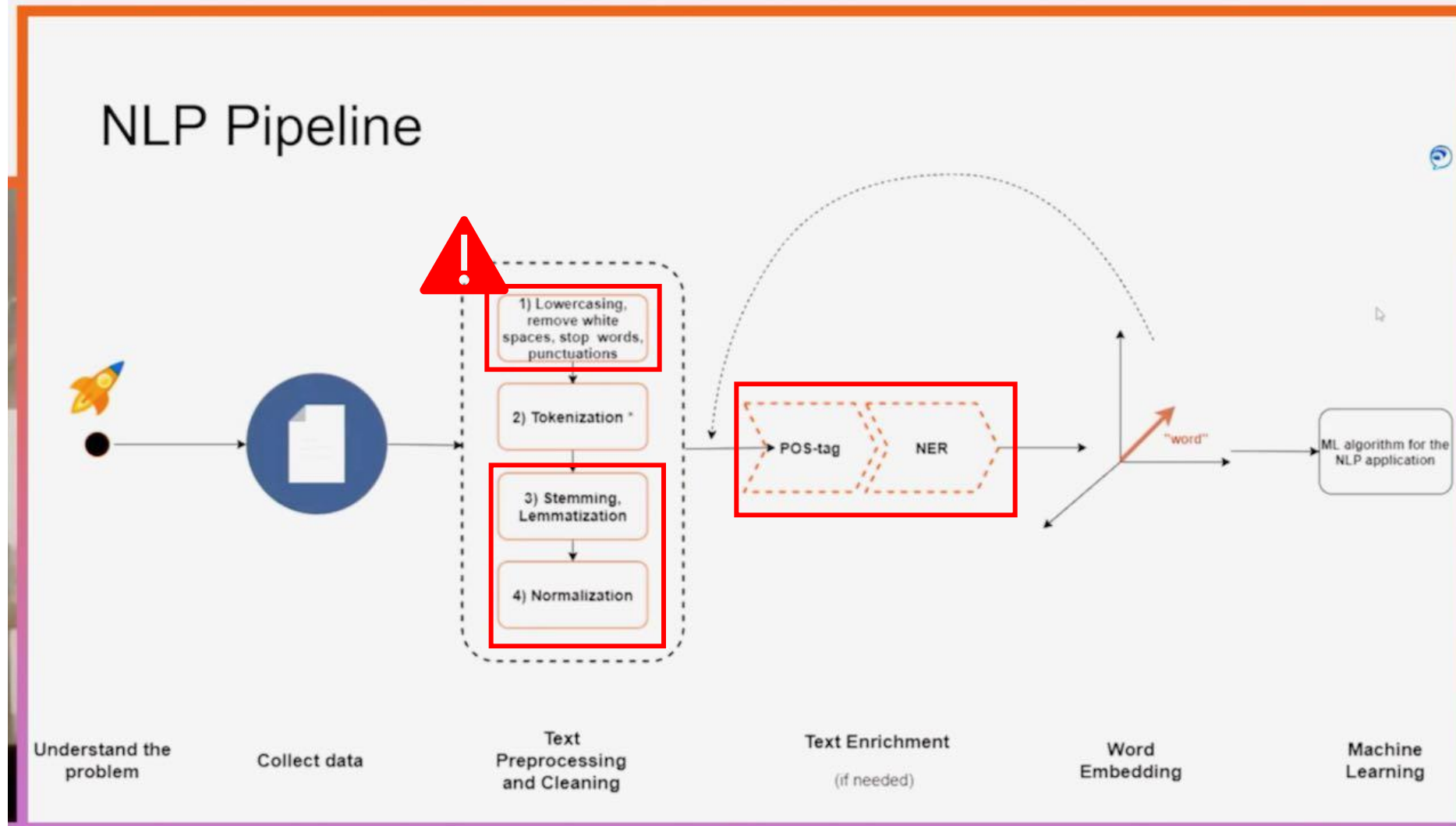
Often require extensive feature engineering and large labeled datasets.

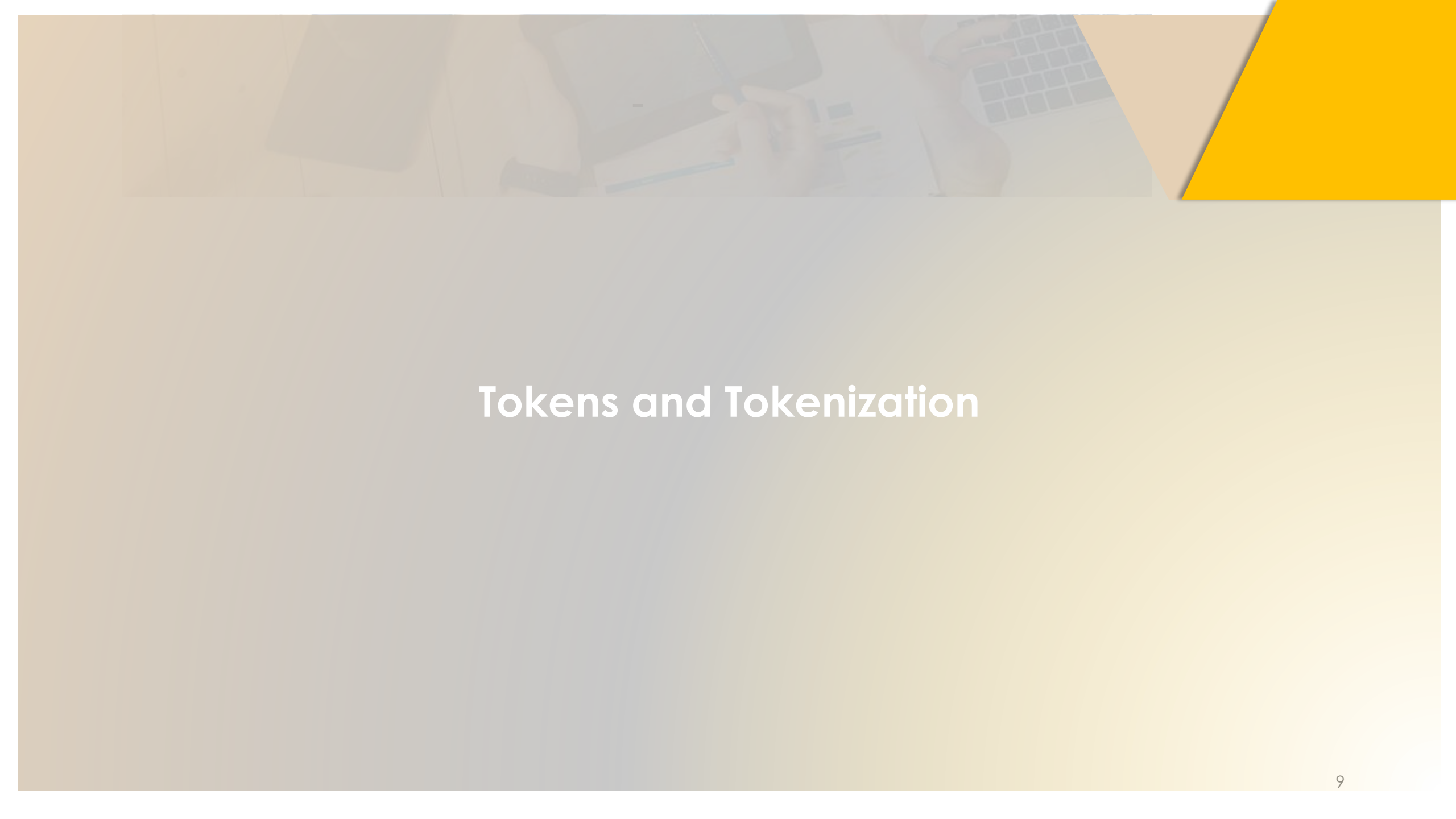




Machine learning workflow for NLP

Machine learning workflow for NLP





Tokens and Tokenization

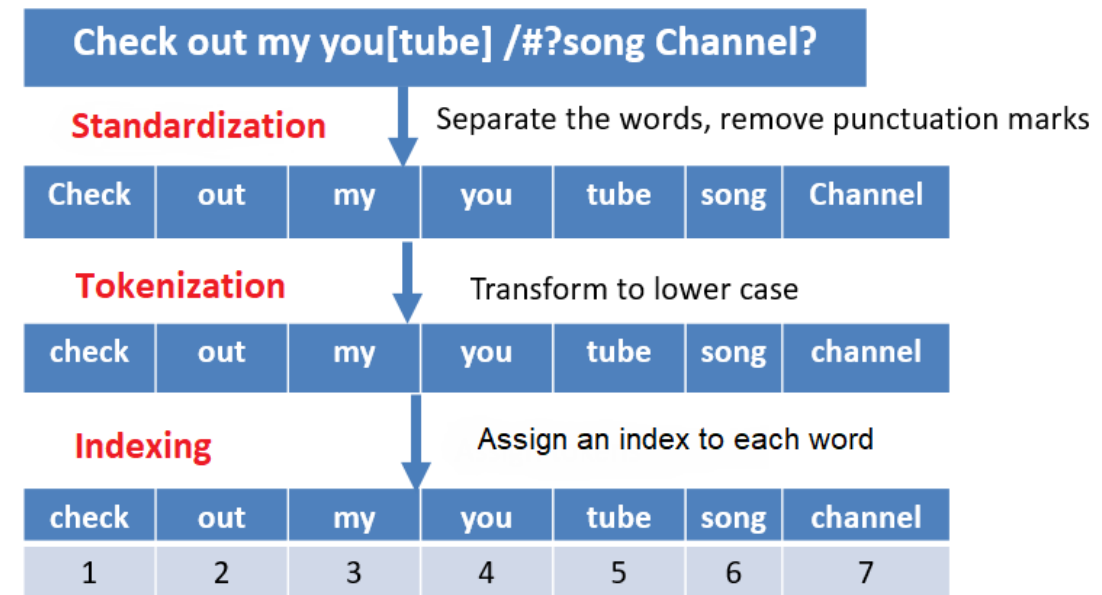
What are tokens?

Tokens **are the building blocks** of all natural language tasks and language models.

They can be stored as a key-value pairing, to enable the expression of text as a sequence of numbers.

Language can be processed or **encoded as a series of tokens IDs**.

How these tokens are built for a given language, or dataset varies and can have a **drastic impact on the performance** and utility of the language models



Important terms

When discussing tokens and tokenization, it is important to consider some terms that describe how a natural language phrase is tokenized.

If we take the following phrase, or **sequence**, we need to use our **vocabulary**, to split it into separate **tokens**.

The moon, Earth's only natural satellite, has been a subject of fascination and wonder for thousands of years.

Token: Basic building block

- The
- Moon
- ,
- Earth's
- Only
-
- years

Sequence: Sequential list of tokens

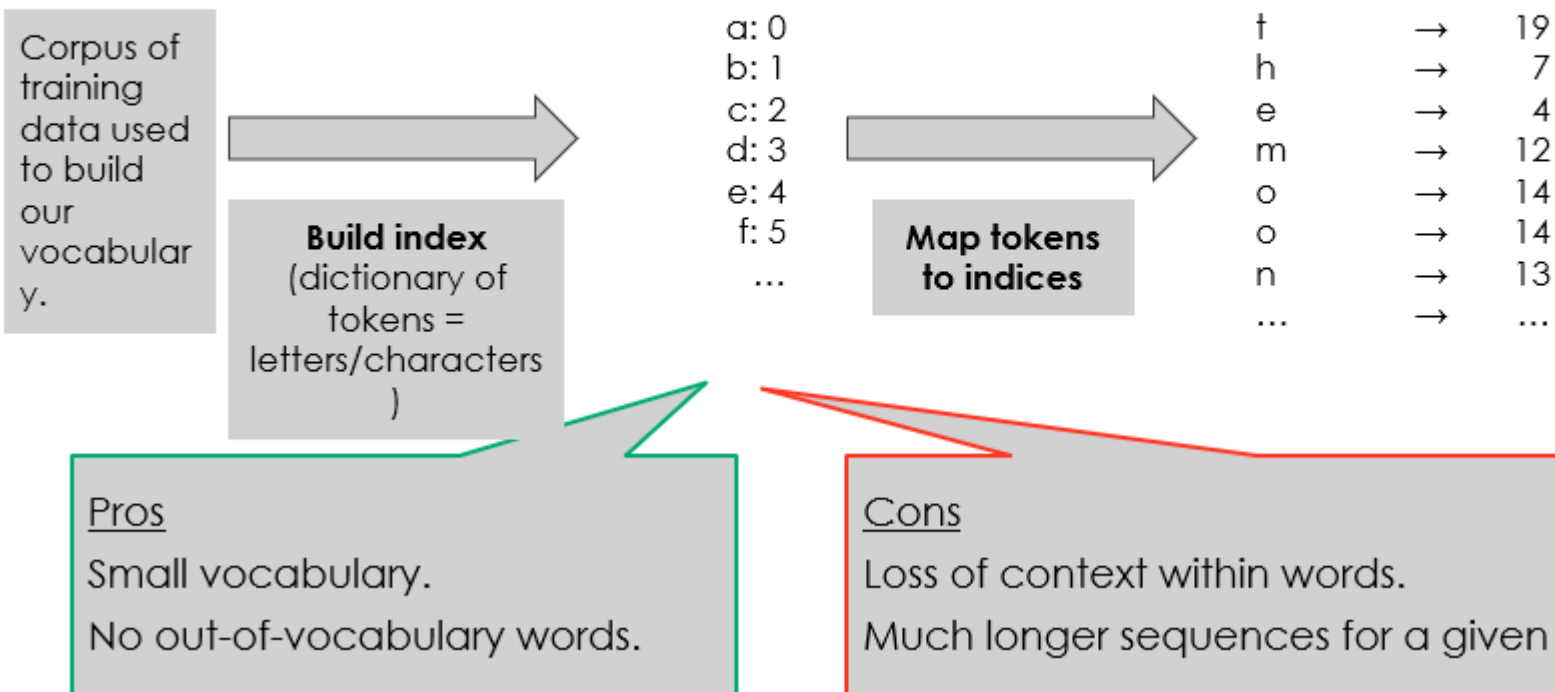
- The moon,
- Earth's only natural satellite
- Has been a subject of
-
- Thousands of years

Vocabulary: Complete list of known tokens

```
{  
  1:"The",  
  569:"moon",  
  122: ",",  
  430:"Earth",  
  50:"**'s",  
  ...}
```

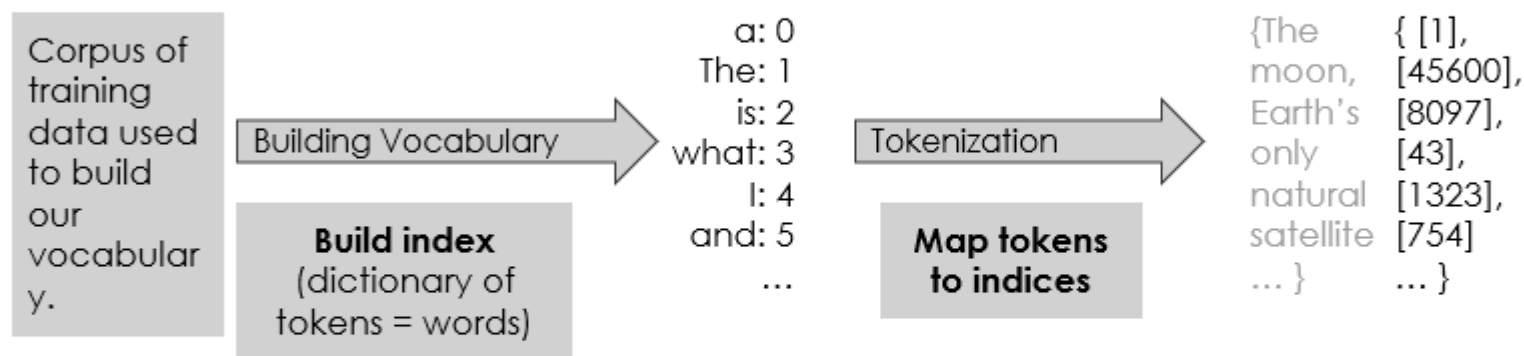
Tokenization : Character-level

The moon, Earth's only natural satellite, has been a subject of fascination and wonder for thousands of years.



Tokenization word-level

The moon, Earth's only natural satellite, has been a subject of fascination and wonder for thousands of years.



Pros

Intuitive.

Quick and simple to implement

Cons

Big vocabularies.

Complications such as handling misspellings and other out-of-vocabulary words.

Effect of Tokenization – Spelling and Char-level issues

1. LLMs Struggle with Spelling and Rare Words:

Tokenization splits words into sub-words (e.g., “unpredictable” → [“un”, “predict”, “able”]).

Misspelled words (e.g., “teh”) fragment into low-probability sub-words, causing errors.

2. String Manipulation Is Difficult:

LLMs operate at the token level, not character level.

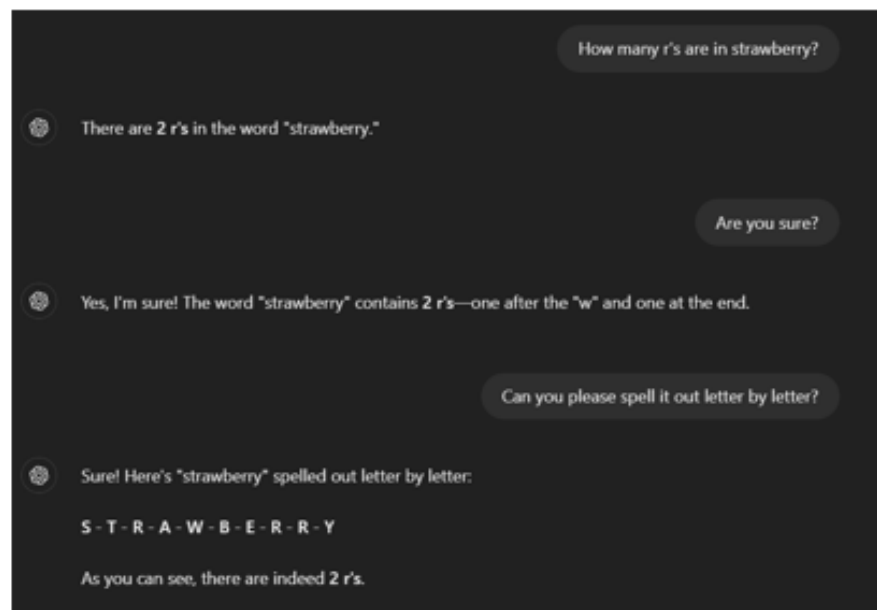
Tasks like reversing strings (“hello” → “olleh”) require access to individual characters, which tokenization obscures.

3. Granularity Mismatch:

Sub-word tokenization prioritizes meaning over form, leading to poor performance on character-sensitive tasks.

Why It Happens:

LLMs are trained to understand semantics, not fine-grained character-level details.



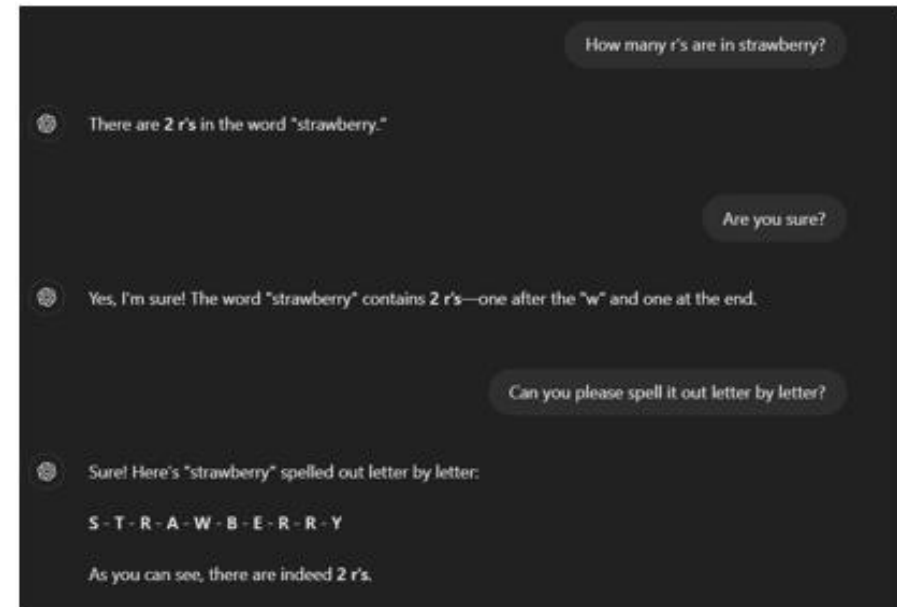
The “strawberry problem”

Effect of Tokenization – Spelling and Char-level issues

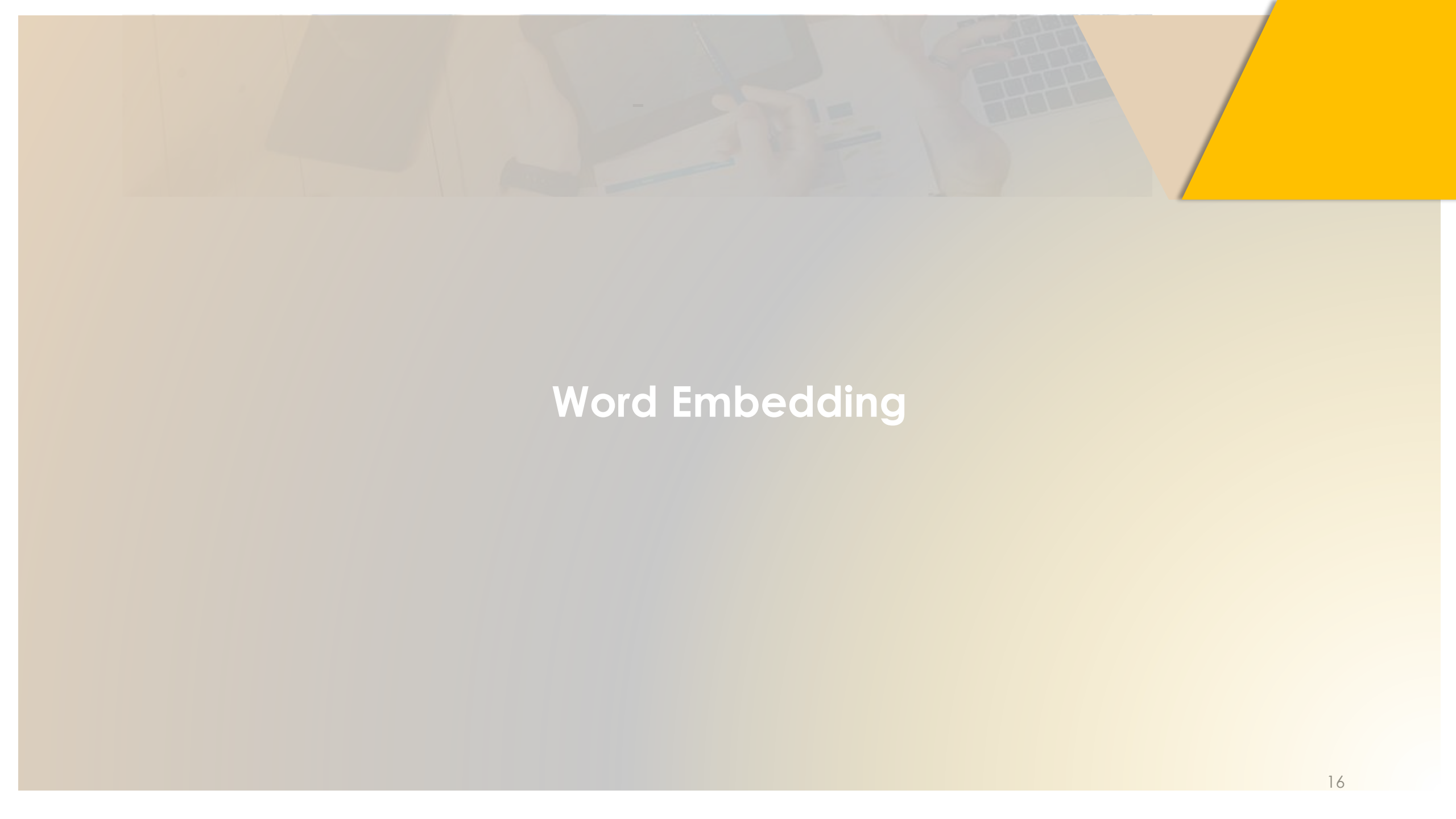
Tokenization with Byte Pair Encoding or subword methods like WordPiece is sometimes not efficient because, in certain cases, they cannot handle individual words properly.

To address this, OpenAI and others have added to their datasets:

- spelling tests
- letter counting tasks
- word-level reasoning
- synthetic datasets where words are split **character by character**



The "strawberry problem"



Word Embedding

Translating text into numbers

For any language model, we need a way to convert and represent the human-native languages, written in text and spoken verbally, into a format that can be utilized by our computational language models.

Encoding this information will lead us to the concept of word embedding vectors. But it will help to first see how we get there.

Let's answer these questions:

- Why is this difficult?
- How is it helpful?
- Is there a perfect way to do it?

Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols:

hotel, *conference*, *motel* — a localist representation

one 1, the rest 0's

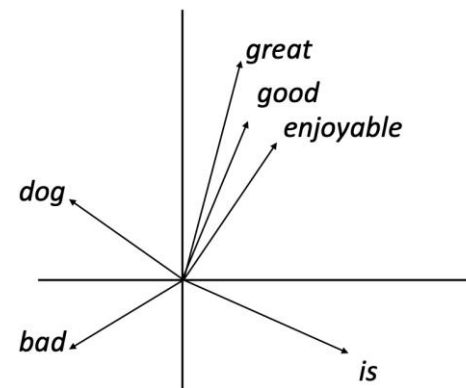
Words can be represented by **one-hot** vectors:

hotel = [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

motel = [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]

Vector dimension = number of words in vocabulary (e.g., 500,000)

There is no way to encode similarity of words in these vectors!



Translating text into numbers

Originally, NLP models were used to perform tasks such as document similarity and matching.

To do this, a common approach was to convert a document (or sentence) into a single vector where each dimension corresponded to a word in the vocabulary

	MARY	IS	HUNGRY	HAPPY	FOR	APPLES	NOT	JOHN	HE	
"Mary is hungry for apples."	1	1	1	0	1	1	0	0	0	→ [1, 1, 1, 0, 1, 1, 0, 0, 0]
"John is happy he is not hungry for apples."	0	2	1	1	1	1	1	1	1	→ [0, 2, 1, 1, 1, 1, 1, 1, 1]

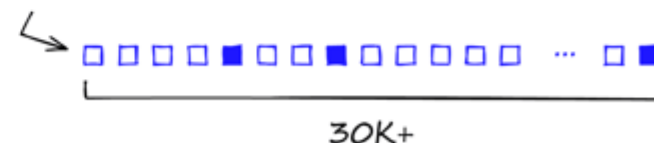
What this results in are known as "sparse" vectors.

For smaller examples these approaches might seem reasonable but imagine a vocabulary of all English words ~500k...

Any given document would have a tiny set of unique words, meaning that the vector of that document would be almost entirely made up of 0s.

sparse

$[0, 0, 0, 1, 0, \dots 0]$



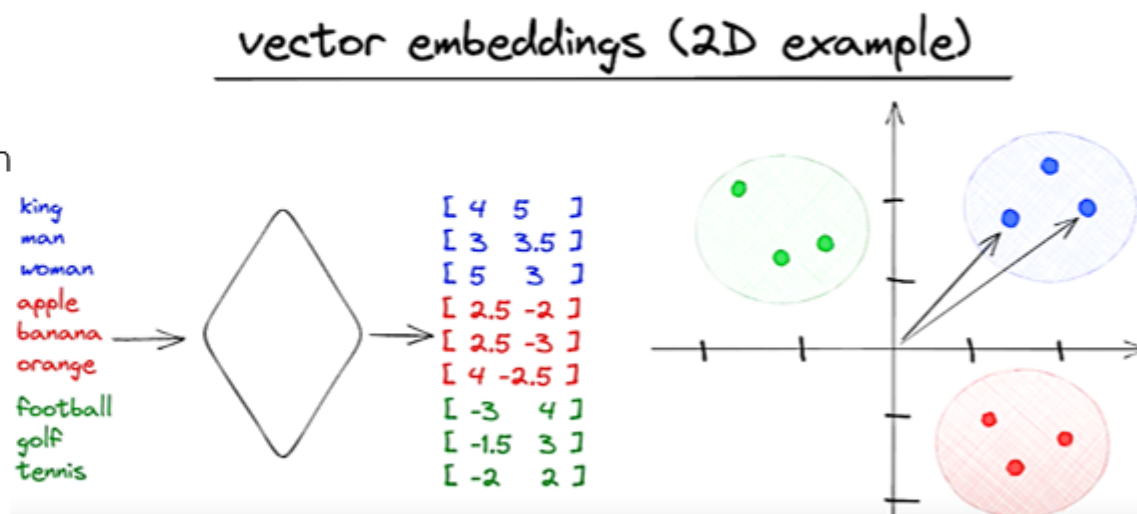
Translating text into numbers

Word2Vec is a family of algorithms proposed in 2013.

Unlike previous approaches to create document vectors with each word as a dimension, Word2Vec starts with each word being represented as a multidimensional dense vector of 100s of dimensions in what is known as an **embedding space**. Each unique word corresponds to a unique location in this space.

Two approaches are used to generate these embedding vectors:

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

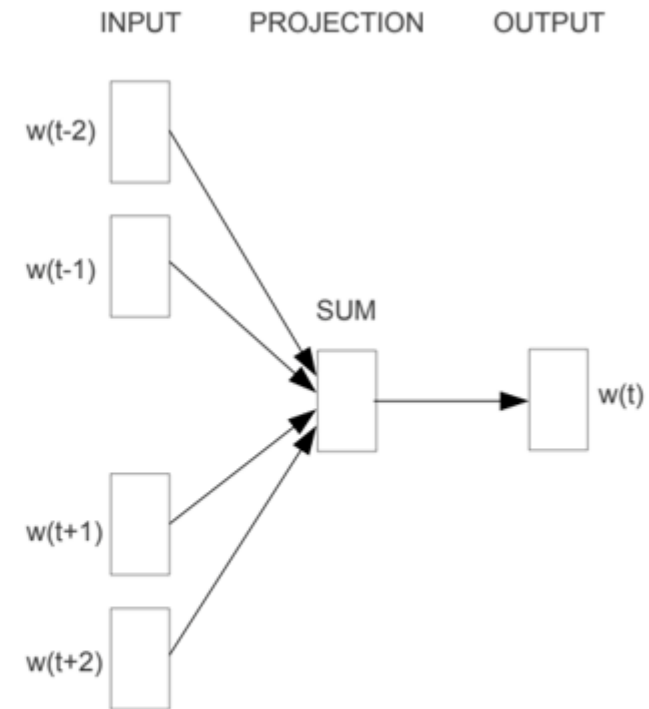


Translating text into numbers

CBOW can be viewed as a 'fill in the blank' task, where the word embedding represents the way the word influences the relative probabilities of other words in the context window.

Words which are semantically similar should influence these probabilities in similar ways, because semantically similar words should be used in similar contexts.

The order of context words does not influence prediction (bag of words assumption).



CBOW

Translating text into numbers

In the continuous skip-gram architecture, the word2vec model uses the target word to predict the surrounding window of context words.

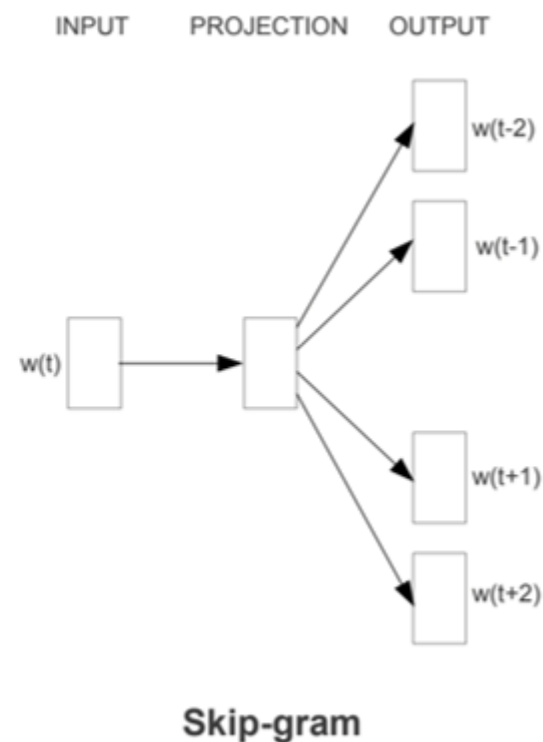
The skip-gram model weighs those context words nearby more heavily.

Pros:

- **Better for Rare Words:** It handles infrequent words well, as it generates multiple context-word pairs, improving their embeddings.
- **Captures Detailed Semantic Relationships:** It excels in tasks requiring nuanced word representations, like analogies (e.g., “king - man + woman = queen”).

Cons:

- **Slower Training:** Skip-Gram generates one prediction per word-context pair, making it computationally more expensive.
- **Requires Larger Data:** Performs better with larger datasets, as it can struggle with sparse data due to its granular focus on word-context pairs.



Translating text into numbers

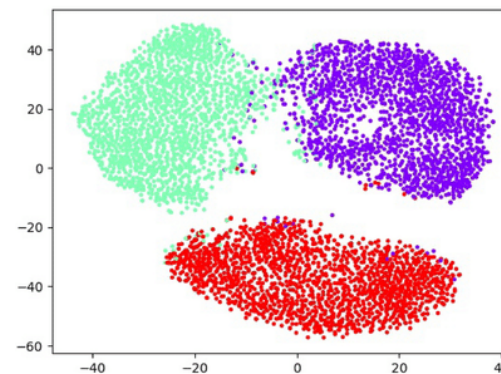
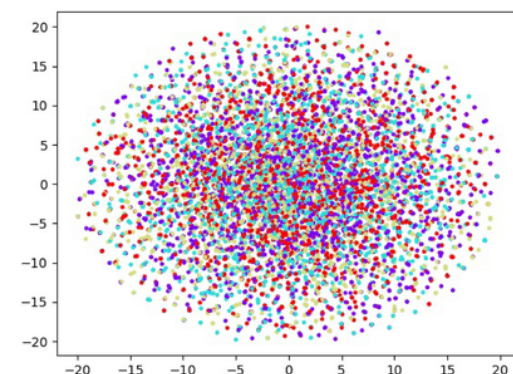
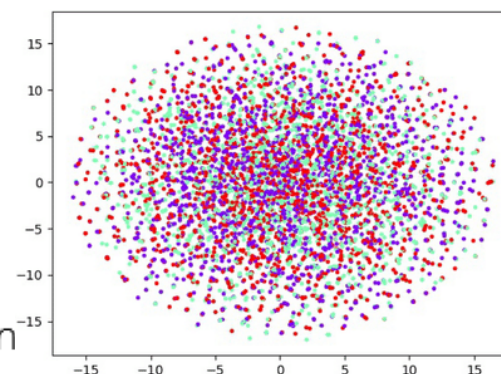
Once trained, these word2vec models represent a latent space for the word embeddings. With each word represented as a dense vector, they possess several useful features:

- Words with similar meanings (e.g., “king” and “queen”) are closer together.
- Linear relationships emerge, allowing analogies like “king - man + woman = queen.”
- Embedding spaces typically have 100–300 dimensions, capturing much of the linguistic structure efficiently.

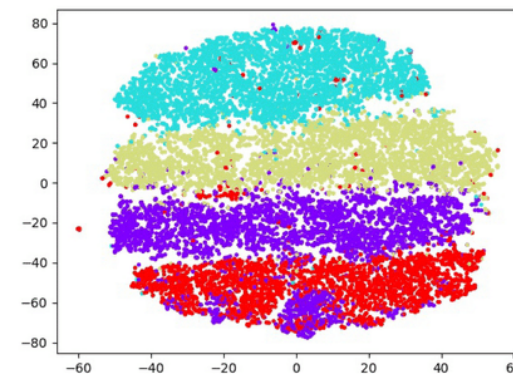
Useful tasks with word embeddings:

Semantic Similarity: Finding related words (e.g., for recommendations or clustering).

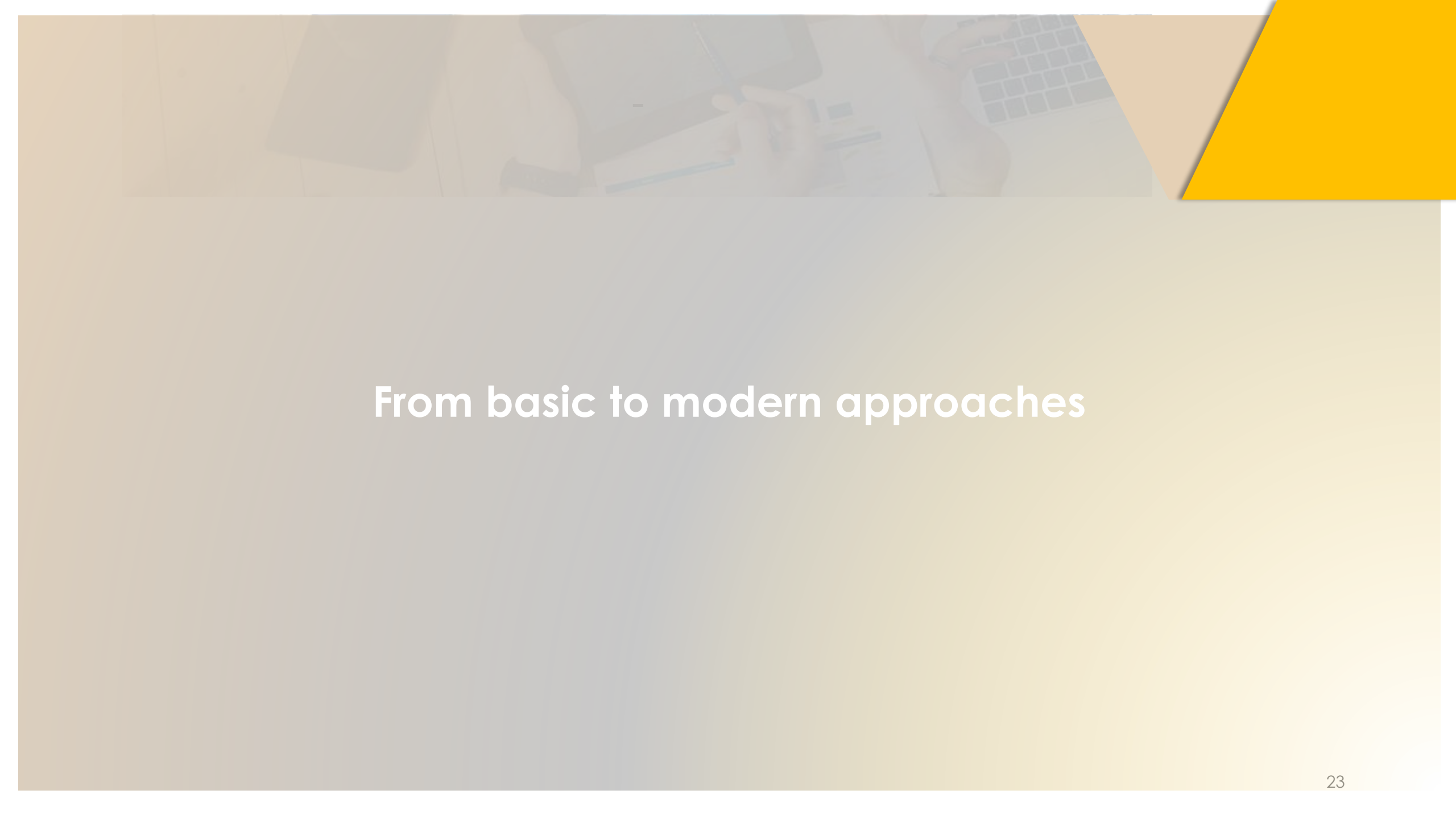
Downstream Tasks: Serving as input to models for tasks like sentiment analysis, translation, or question answering.



(a) Latent space distribution of emotion



(b) Latent space distribution of action



From basic to modern approaches

Classical NLP Methods

Rules-based

Rule-based models use a set of **predefined rules to interpret** and understand natural language. These models **work by breaking down text into smaller parts**, such as words or phrases, and then applying a set of rules to those parts to extract meaning.

Rule-Based Data Preprocessing

Tokens	the	expressions	are	formed	using	functions
Possible POS tags	{DT}	{NNS}	{VBP}	{VBD, VBN}	{VBG, NN}	{VBZ, NNS}
Candidate POS tag sets	{DT}	{NNS}	{VBP}	{VBN}	{VBG, NN}	{VBZ, NNS}

Model Inference

Tokens	the	expressions	are	formed	using	functions
Part-of-Speech	{DT}	{NNS}	{VBP}	{VBN}	{MASK}	{MASK}
	{DT}	{NNS}	{VBP}	{VBN}	{VBG}	{NNS}

Classical NLP Methods - Limitations

Bag of Words (BoW)

Description:

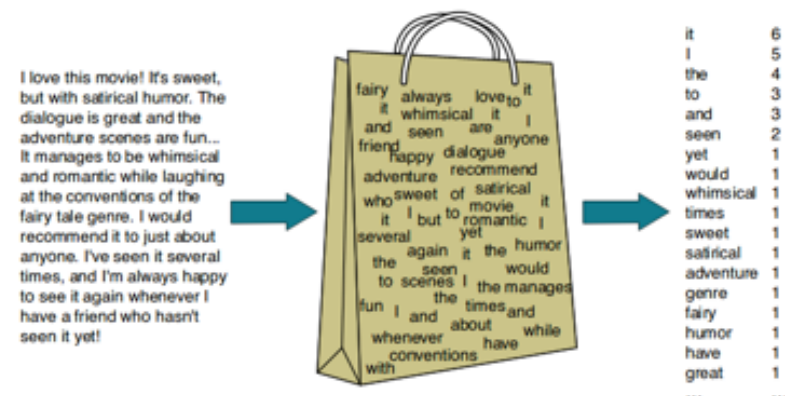
Represents text as an unordered collection of words, ignoring grammar and word order.

Pros:

- Simple and Efficient: Easy to implement and computationally inexpensive.
- Good for Short Texts: Works well when context and order aren't crucial (e.g., spam detection).

Cons:

- Ignores Word Order: Loses context and meaning (e.g., "not good" vs. "good").
- Sparse Representation: Large vocabulary leads to high-dimensional, sparse vectors.
- Contextual Limitations: Struggles with understanding relationships between words.



N-Gram Models

Description:

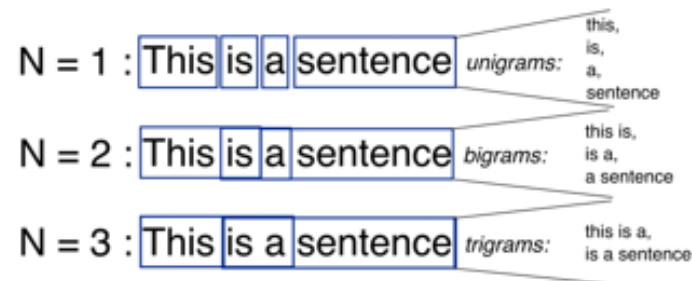
Represents text as sequences of n -length contiguous words (e.g., bigrams = 2-word sequences).

Pros:

- Captures Word Order: Preserves local context and basic syntax.
- Better Context Representation: Improves performance over BoW for tasks like language modeling.

Cons:

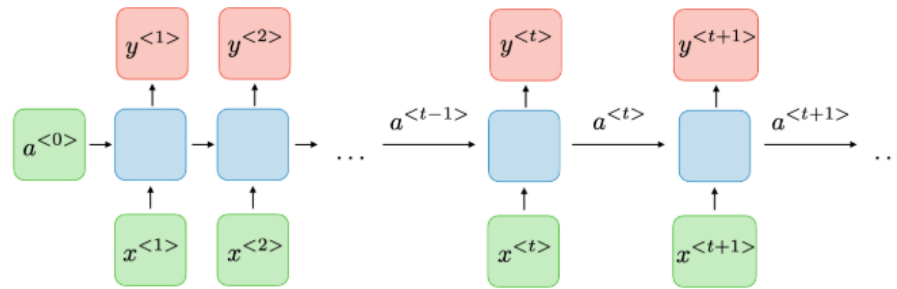
- Limited Context Window: Can't capture long-range dependencies (e.g., beyond 3-4 words).
- Data Sparsity: Rare n -grams can lead to poor generalization.
- Scalability Issues: Large n -grams increase computational and memory demands.



$P(\text{"There was heavy rain"}) \sim P(\text{"There"}) P(\text{"was"} | \text{"There"}) P(\text{"heavy"} | \text{"was"}) P(\text{"rain"} | \text{"heavy"})$

Classical NLP Methods - Limitations

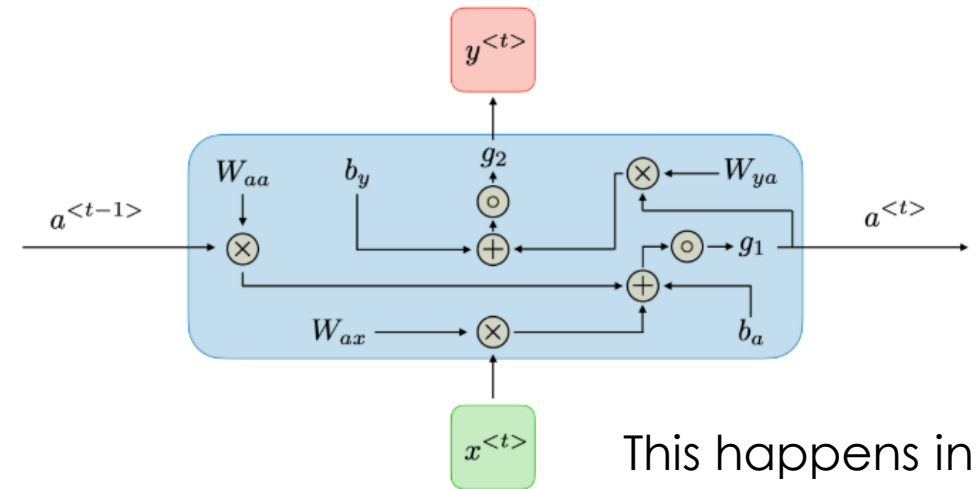
Recurrent neural networks, also known as RNNs, are a class of neural networks that **allow previous outputs to be used as inputs (across time and across layers)** while having hidden states. They are typically as follows:



For each timestep t , the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \quad \text{and} \quad y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

where W_{ax} , W_{aa} , W_{ya} , b_a , b_y are coefficients that are shared temporally and g_1 , g_2 activation functions.



This happens in on HL!

Classical NLP Methods - Limitations

Key Characteristics of RNNs:

- **Sequential Processing:**

RNNs process input one step at a time, making them well-suited for tasks where data has a temporal or sequential structure.

- **Recurrent Connections:**

At each time step, the hidden state of the network is updated based on the current input and the hidden state from the previous time step. This allows the network to "remember" information from earlier in the sequence.

- **Shared Weights:**

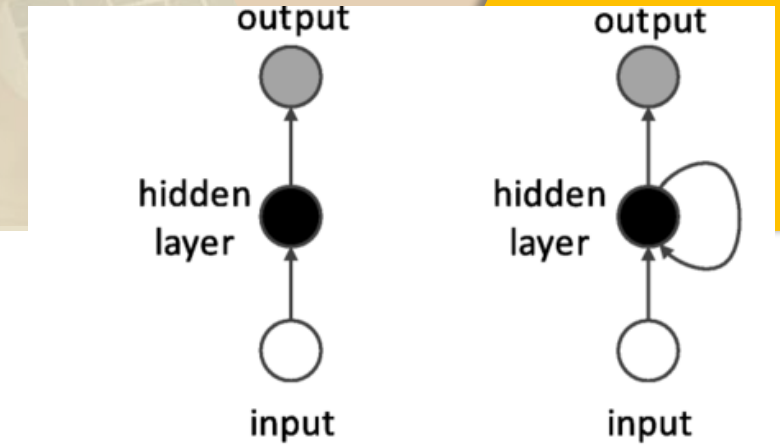
The weights used for processing are shared across time steps, which reduces the number of parameters and helps capture temporal dependencies.

- **Memory:**

The hidden state serves as a form of memory, enabling the network to use information from earlier inputs to influence later outputs.

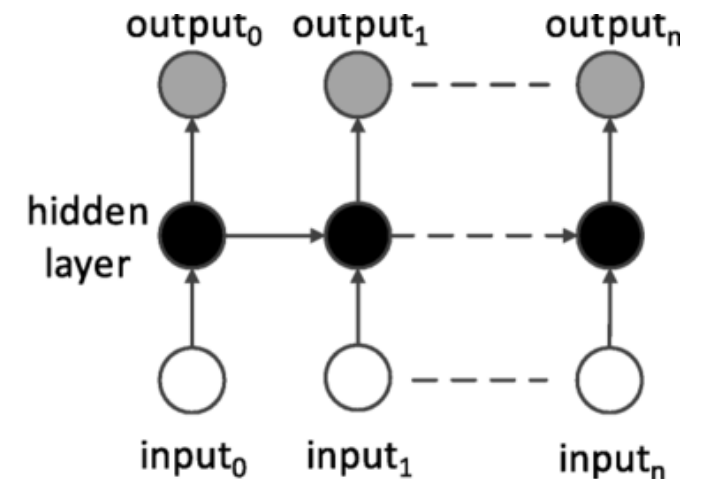
- **Training via Backpropagation Through Time (BPTT):**

The gradient is computed for all time steps of the sequence to train the network. However, this can lead to issues like vanishing or exploding gradients.



(a) A traditional neural network

(b) RNN



(c) RNN in unfolded form

Classical NLP Methods - Limitations

Challenges with RNNs

Struggle **with long input sequences**: Information from earlier words gets diluted as it propagates through the sequence.

- **Vanishing Gradient Problem**: Gradients shrink over time, reducing the ability to learn long-term dependencies.

Solution: LSTMs

Submit a description of LSTMs and a Figure in TikZ format to explain the improvement of LSTMs **over** RNNs

□ **Types of gates** — In order to remedy the vanishing gradient problem, specific gates are used in some types of RNNs and usually have a well-defined purpose. They are usually noted Γ and are equal to:

$$\Gamma = \sigma(Wx^{<t>} + Ua^{<t-1>} + b)$$

where W , U , b are coefficients specific to the gate and σ is the sigmoid function. The main ones are summed up in the table below:

Type of gate	Role	Used in
Update gate Γ_u	How much past should matter now?	GRU, LSTM
Relevance gate Γ_r	Drop previous information?	GRU, LSTM
Forget gate Γ_f	Erase a cell or not?	LSTM
Output gate Γ_o	How much to reveal of a cell?	LSTM

Classical NLP Methods - Limitations

Sequential Processing Bottleneck

RNNs and LSTMs process inputs step-by-step, making training and inference slow for long sequences.

- **Vanishing Gradients**

Gradients diminish as they backpropagate through many time steps, limiting the network's ability to learn relationships across long sequences.

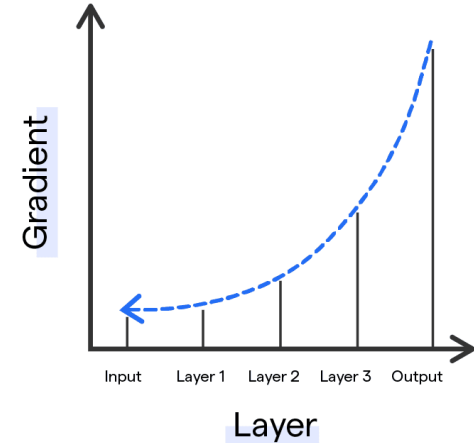
- **Challenges with Long-Range Dependencies**

Even with LSTMs, retaining information from distant parts of a sequence is difficult, leading to a loss of context over time.

- **Focus on Local Context**

RNNs and LSTMs prioritize immediate neighboring words but struggle to model relationships across the entire input effectively.

Vanishing Gradient Problem

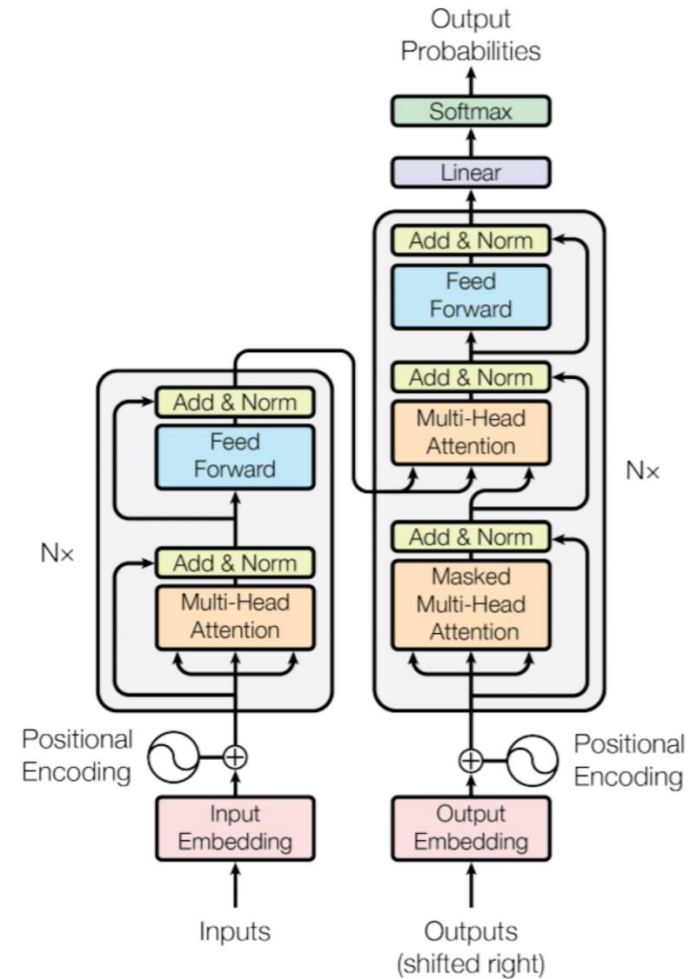


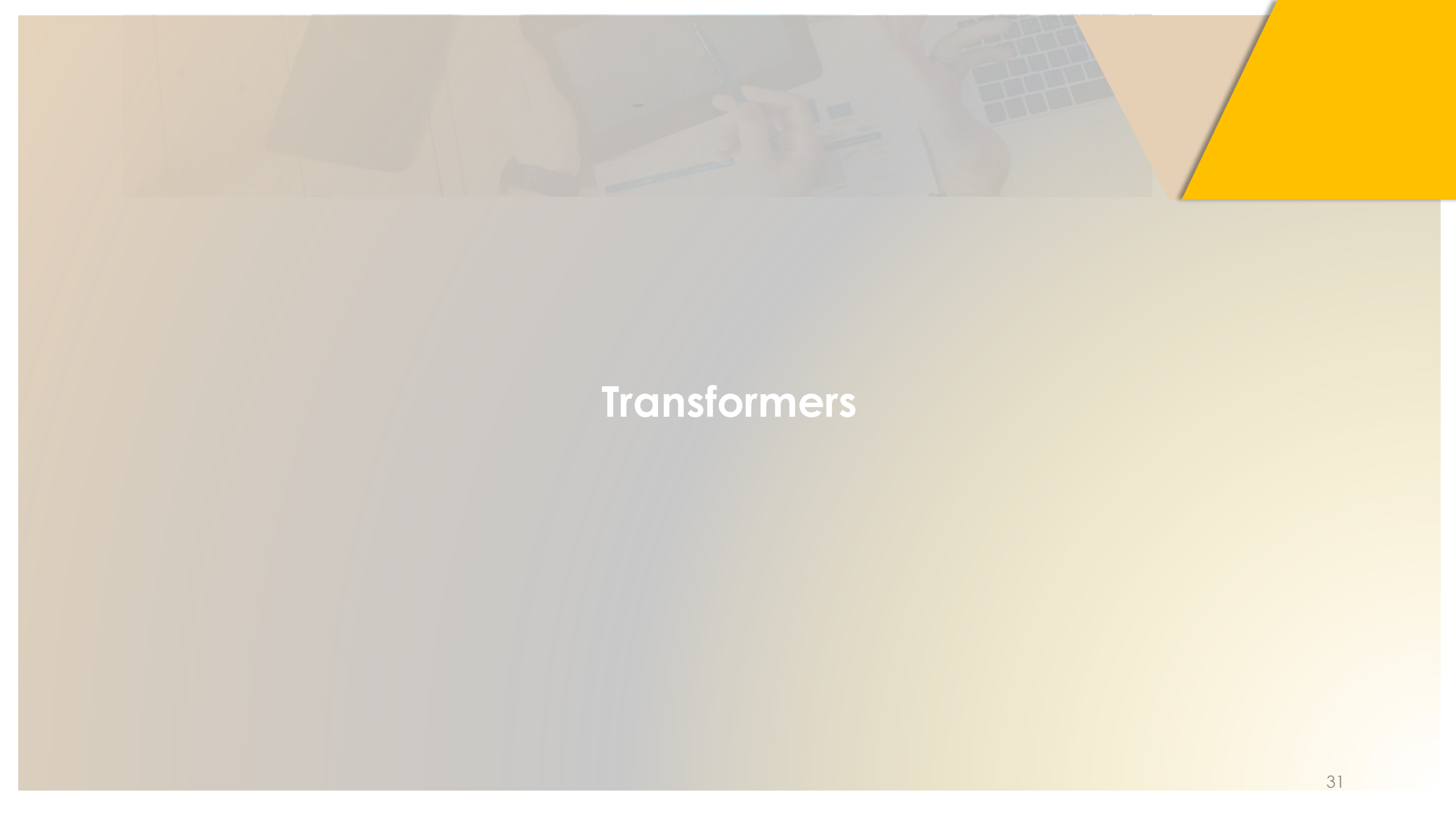
The solution?

A new mechanism that would enable parallel process, dynamically focusing on relevant sequence parts, and capturing long-range dependencies without the limitations of step-by-step computing or vanishing gradients...

Modern approaches

A neural network architecture designed for handling sequential data using a **self-attention mechanism**. Introduced in the paper “*Attention Is All You Need*” (2017), Transformers are the foundation for modern models like BERT, GPT, and T5.





Transformers

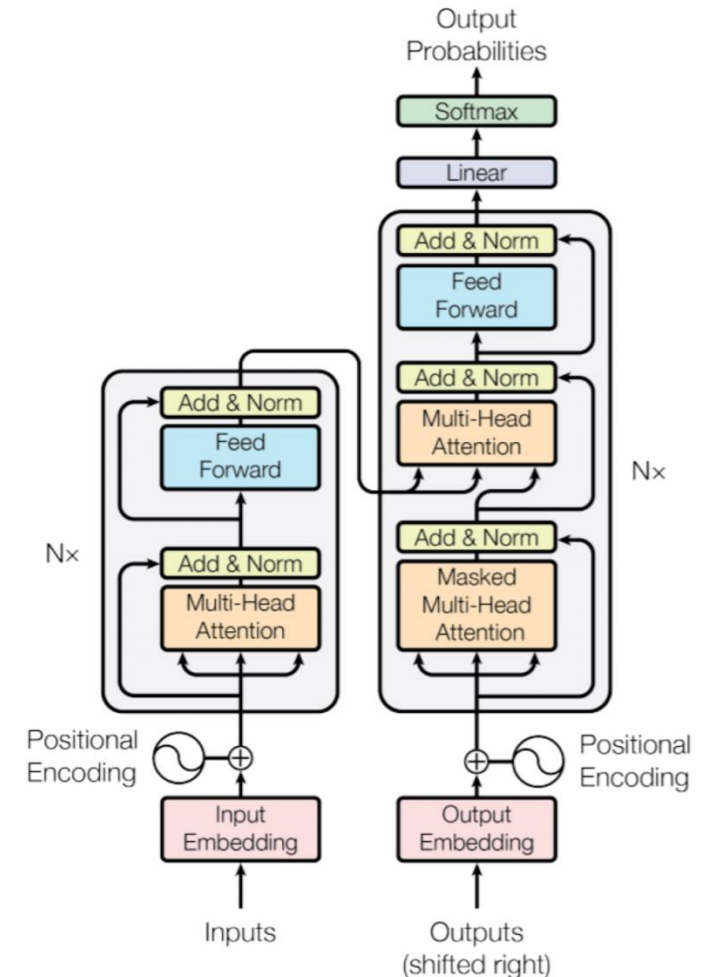
Attention in all you need - Vaswani et.al 2017

This paper 'The paper 'Attention Is All You Need' introduced the **Transformer architecture**, a new approach based exclusively on attention mechanisms.

Transformer architecture completely removed the reliance on recurrence to process sequences.

Self-Attention: What Does It Actually Mean?

Self-attention is a mechanism that allows a model **to dynamically focus on the most relevant parts** of a sequence when computing representations for each token.



Attention in all you need - Vaswani et.al 2017

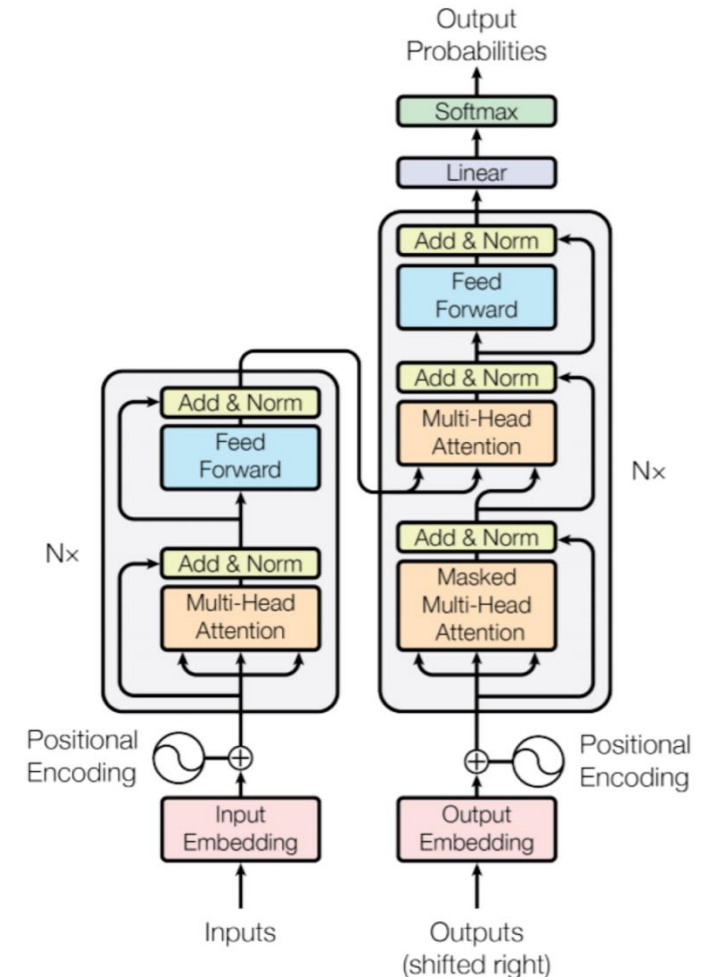
Key Concept:

Every token in the sequence can *attend* to every other token, including **itself**, to understand its relationship and importance in the context of the entire sequence.

E.g.

In the sentence: "**The cat chased the mouse**,"

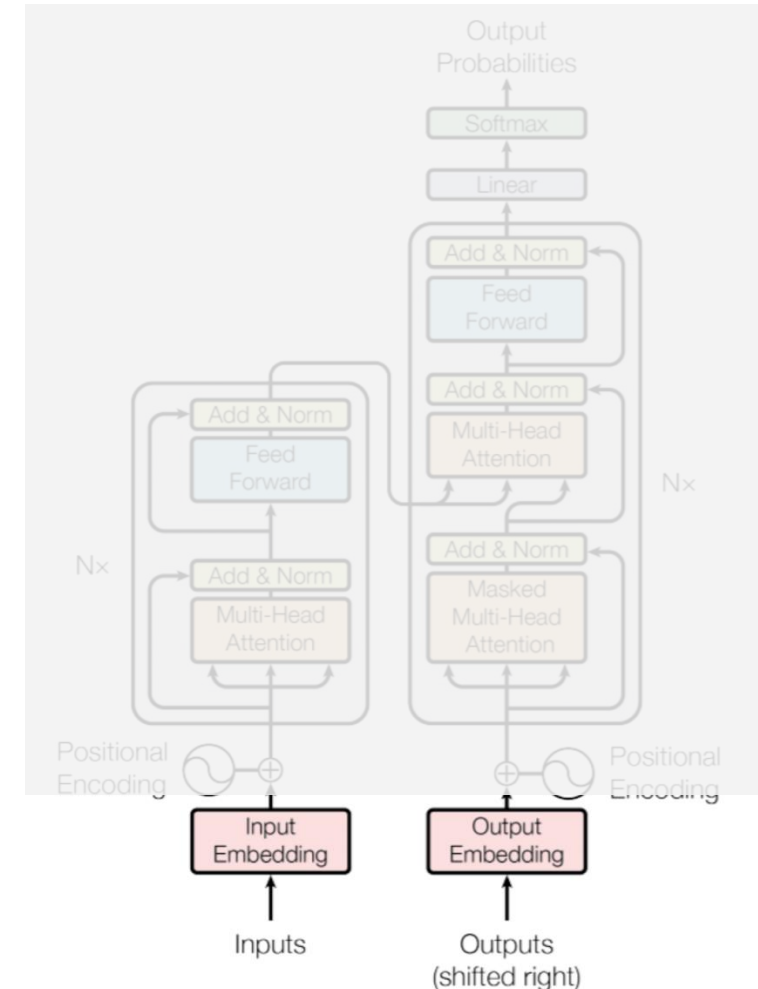
Self-attention helps the model understand that "*the mouse*" is what "*the cat*" **chased**, by focusing on the semantic relationship between "chased" and "the mouse."



How does self-attention mechanism work?

Every text-generative Transformer consists of these **three key components**:

1.Embedding: Text input is divided into smaller units called tokens, which can be words or subwords. These tokens are converted into numerical vectors called embeddings, which capture the semantic meaning of words.



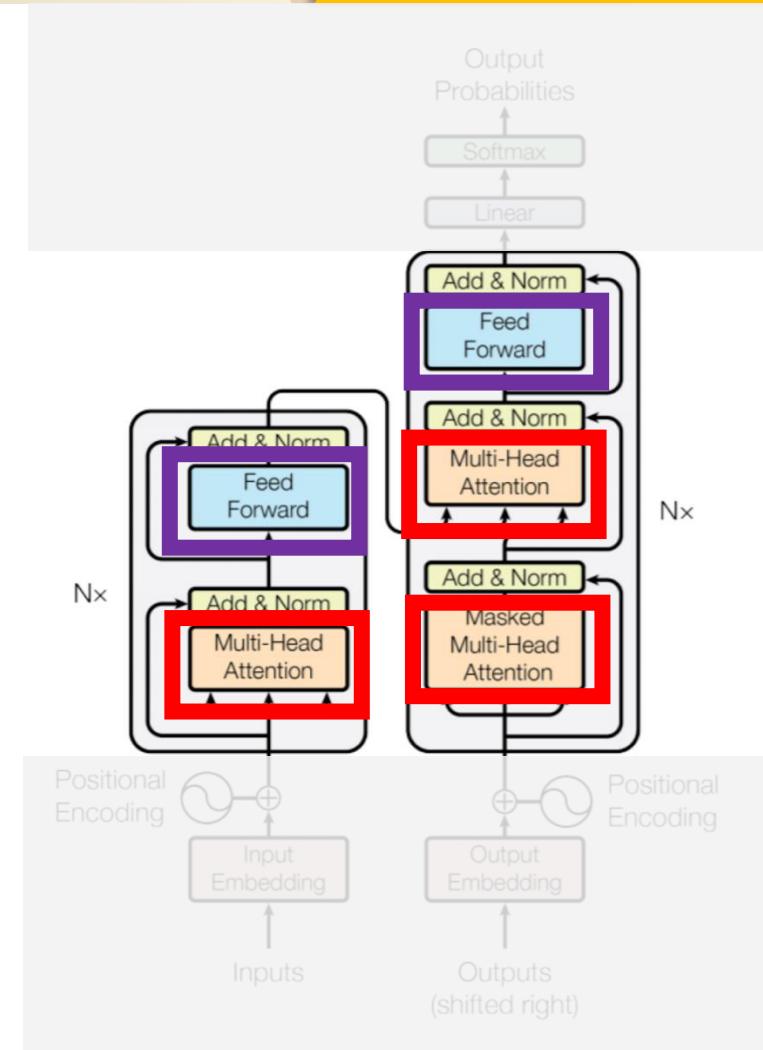
How does self-attention mechanism work?

Every text-generative Transformer consists of these **three key components**:

2,Transformer Block is the fundamental building block of the model that processes and transforms the input data. Each block includes:

1. Attention Mechanism, the core component of the Transformer block. It allows tokens to communicate with other tokens, capturing contextual information and relationships between words.

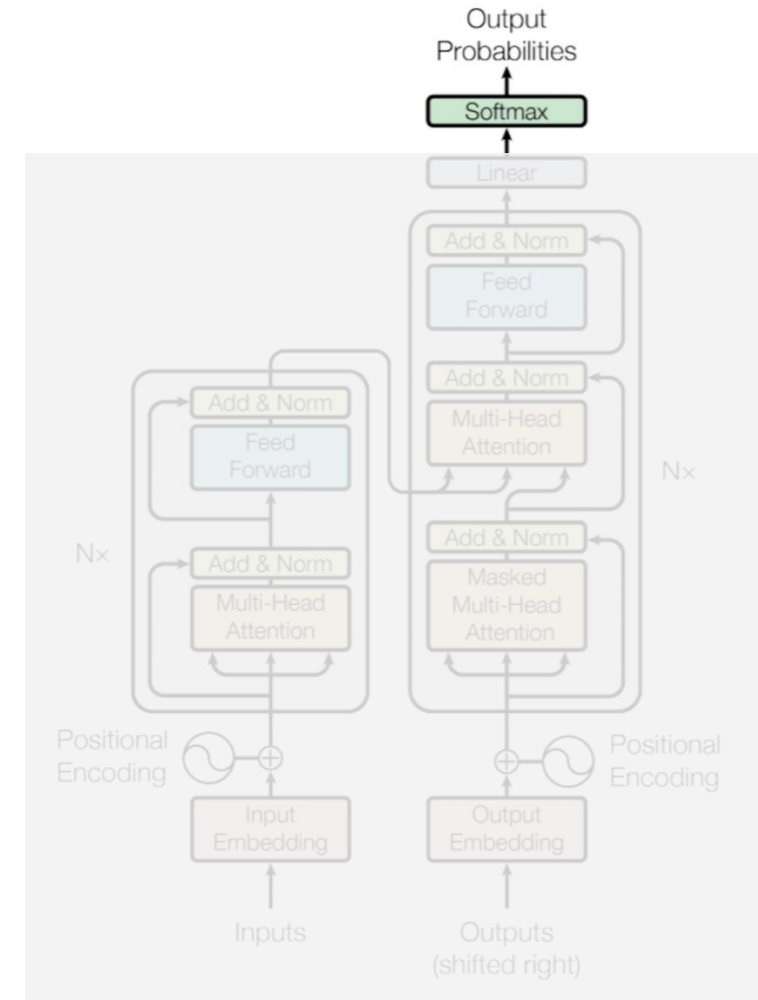
2. MLP (Multilayer Perceptron) Layer, a feed-forward network that operates on each token independently. While the goal of the attention layer is to route information between tokens, the goal of the MLP is to refine each token's representation.



How does self-attention mechanism work?

Every text-generative Transformer consists of these **three key components**:

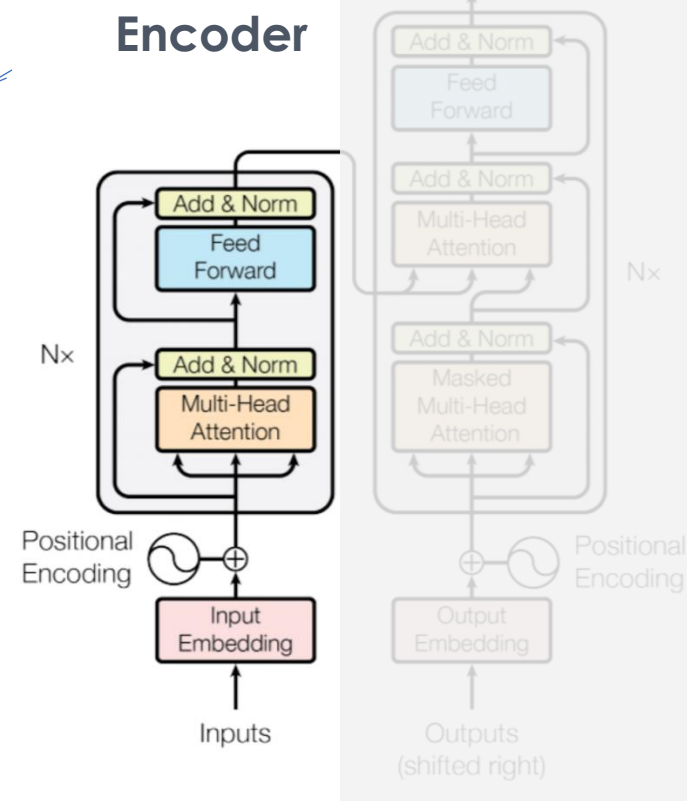
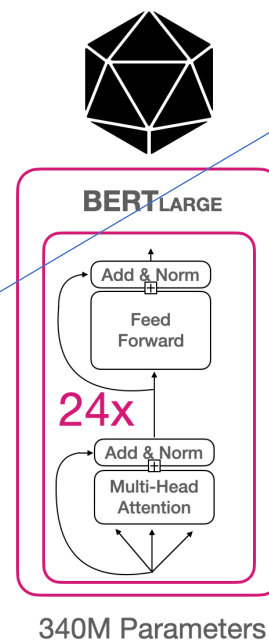
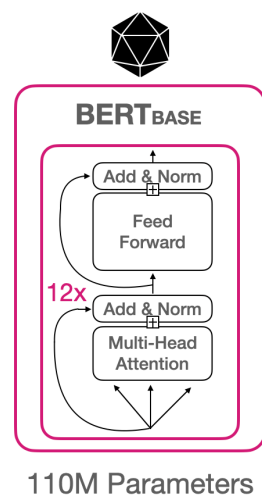
3. Output Probabilities: The final linear and softmax layers transform the processed embeddings into probabilities, enabling the model to make predictions about the next token in a sequence.



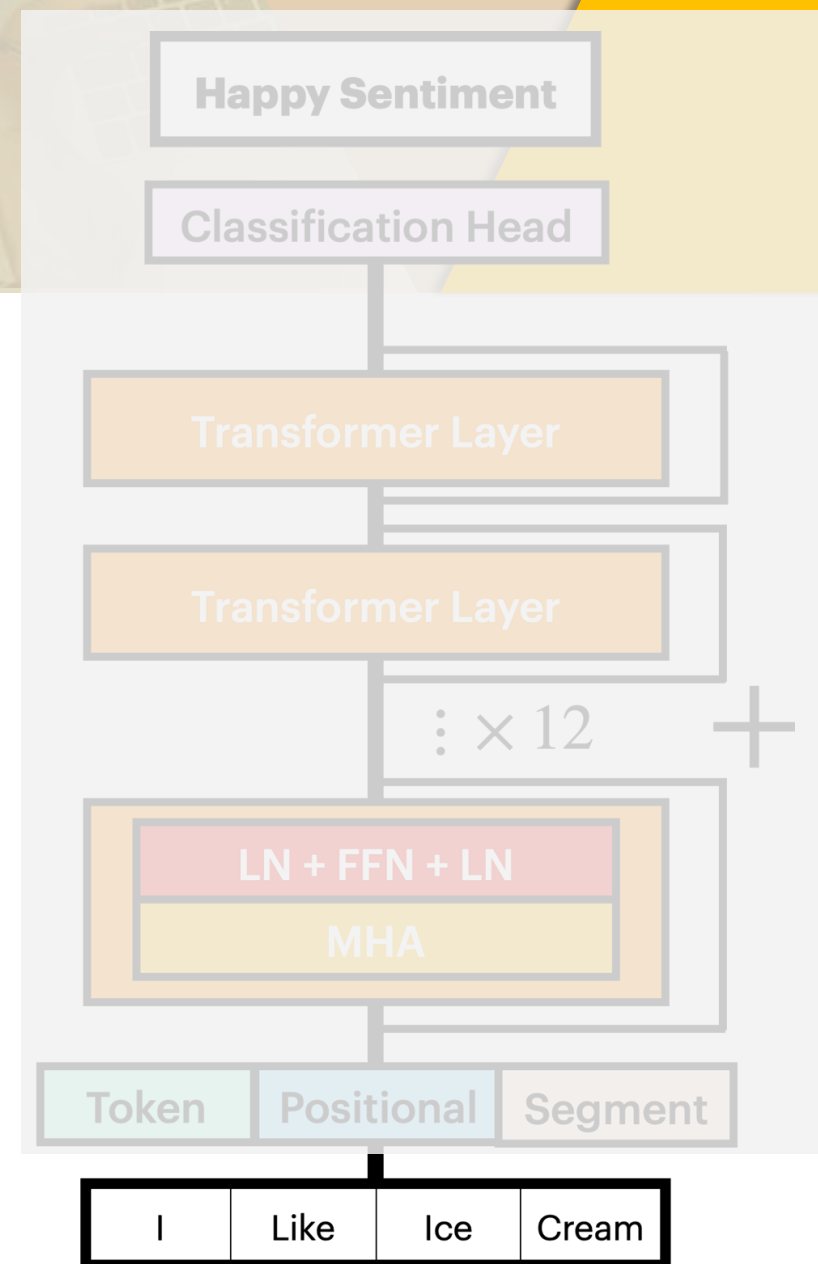
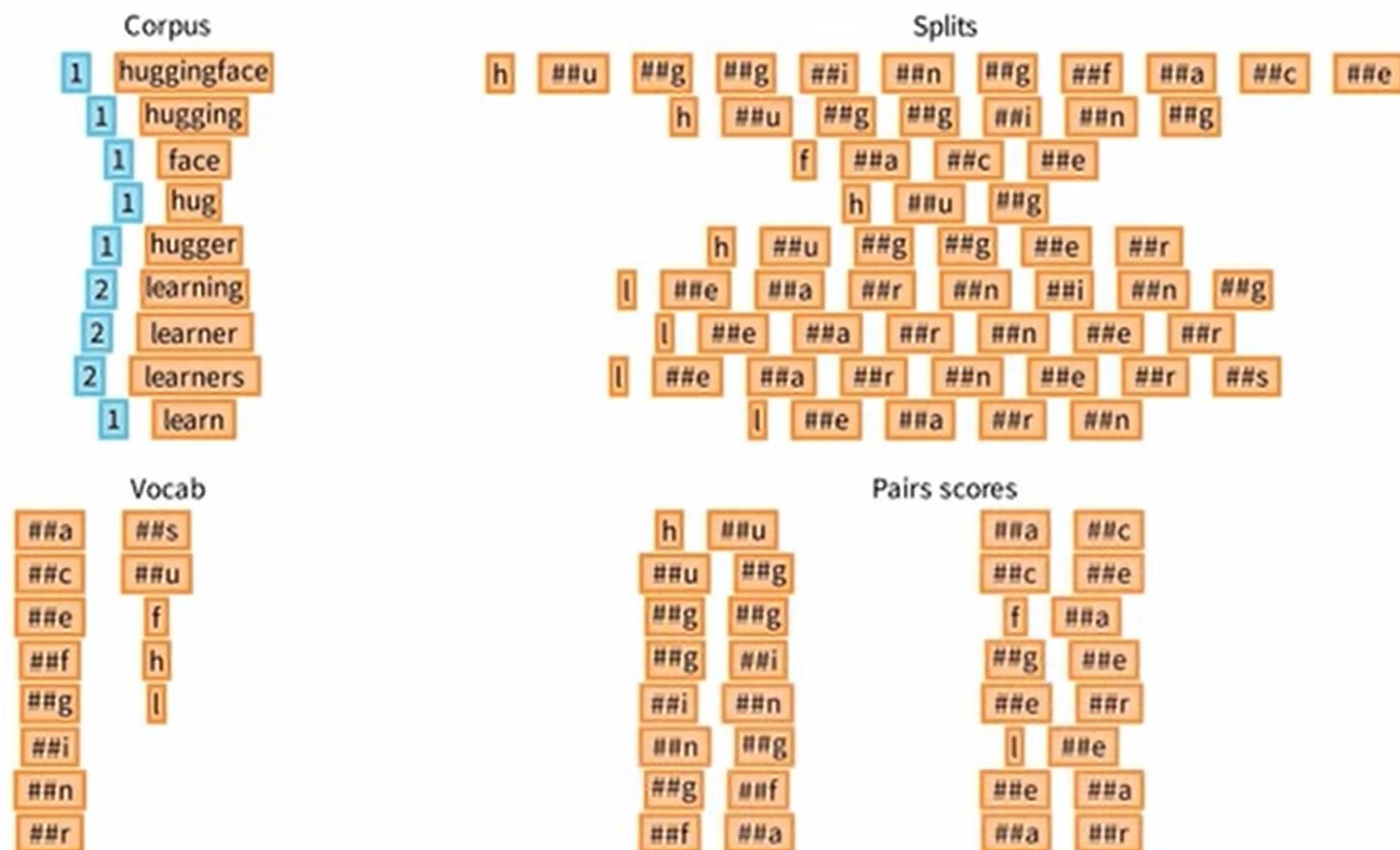
Let's focus on the encoder part

Transformers visually explained : <https://poloclub.github.io/transformer-explainer/>

Bidirectional Encoder Representations from Transformers : BERT



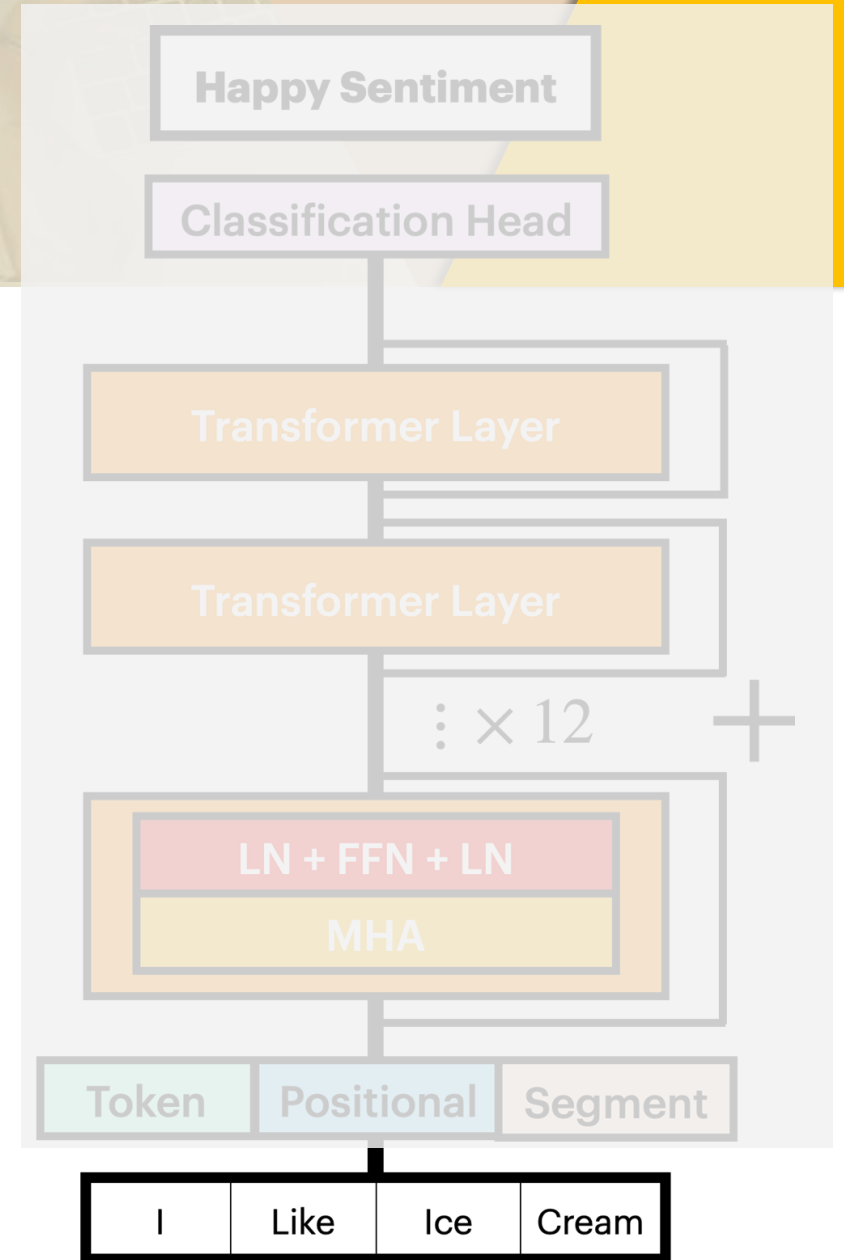
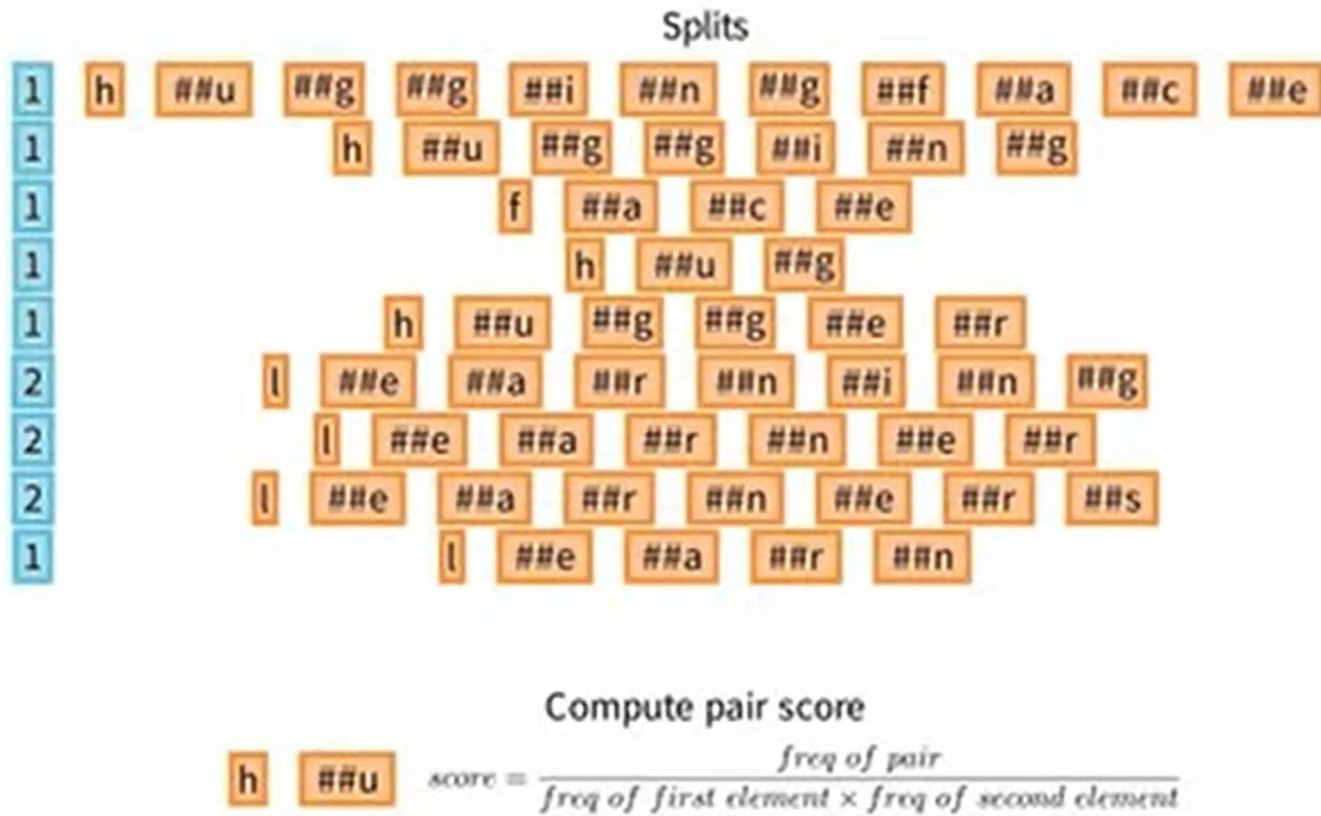
[Bert] Tokenization level : WordPiece



Animated explanation

https://www.youtube.com/watch?v=qpv6ms_t_1A&embeds_referring_euri=https%3A%2F%2Fhuggingface.co%2F

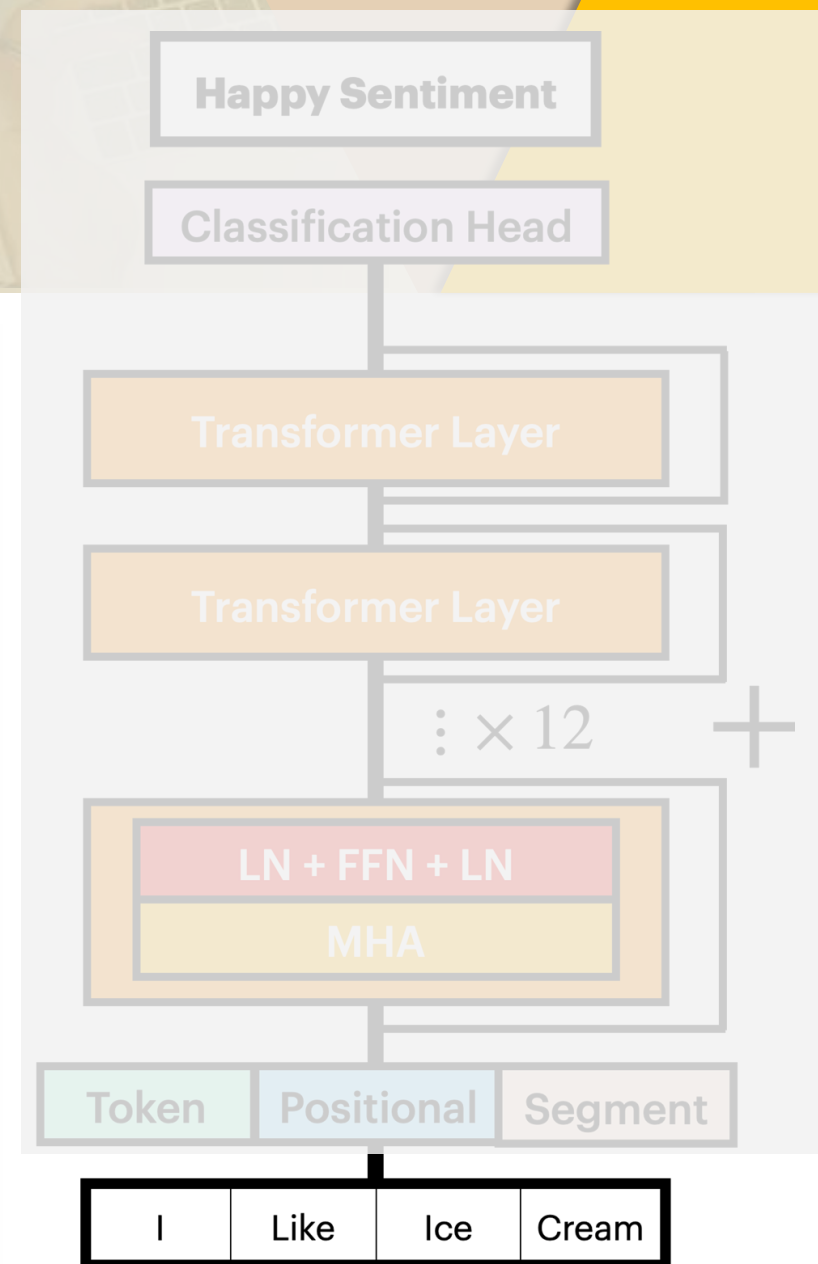
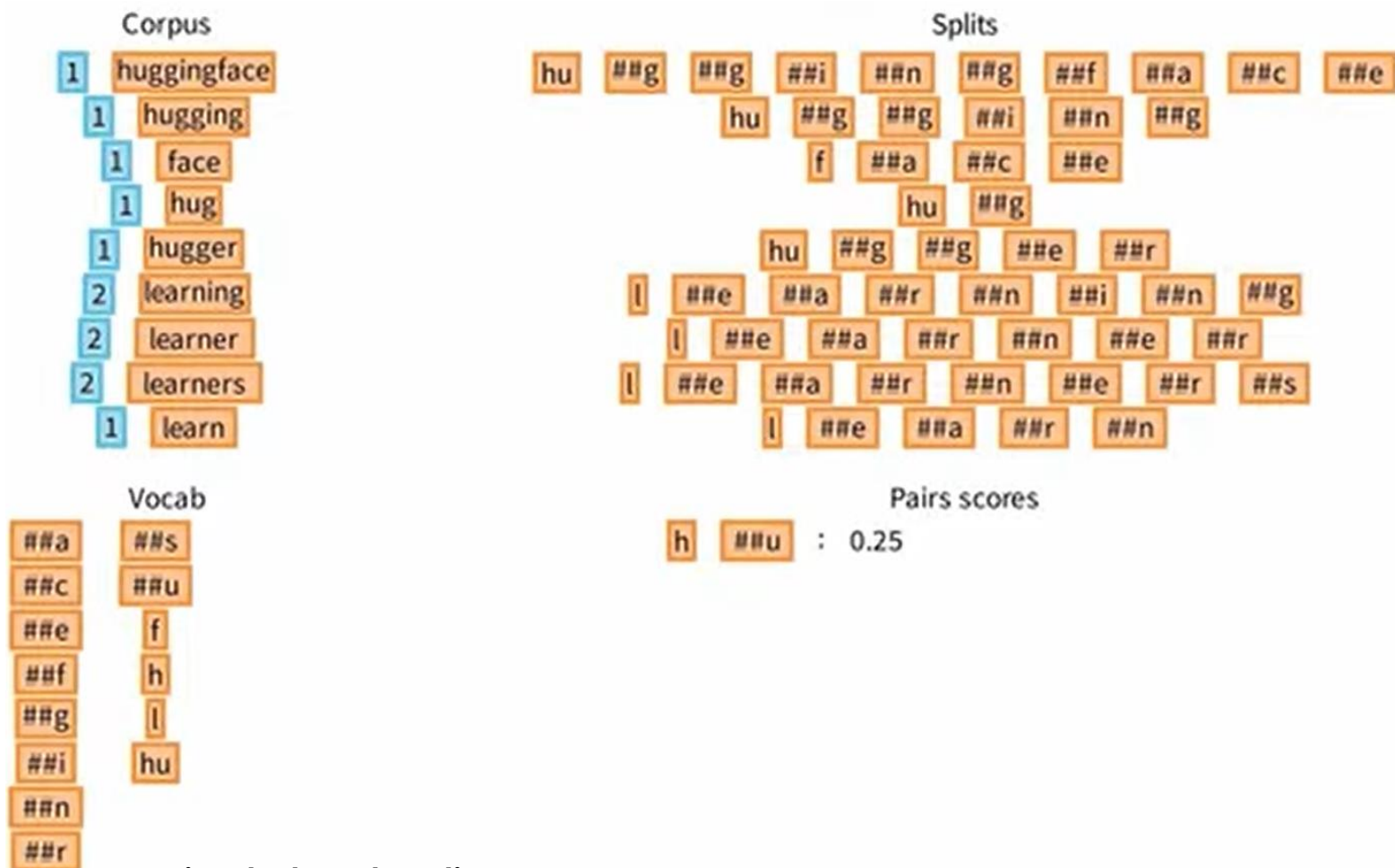
[Bert] Tokenization level : WordPiece



Animated explanation

https://www.youtube.com/watch?v=qpv6ms_t_1A&embeds_referring_euri=https%3A%2F%2Fhuggingface.co%2F

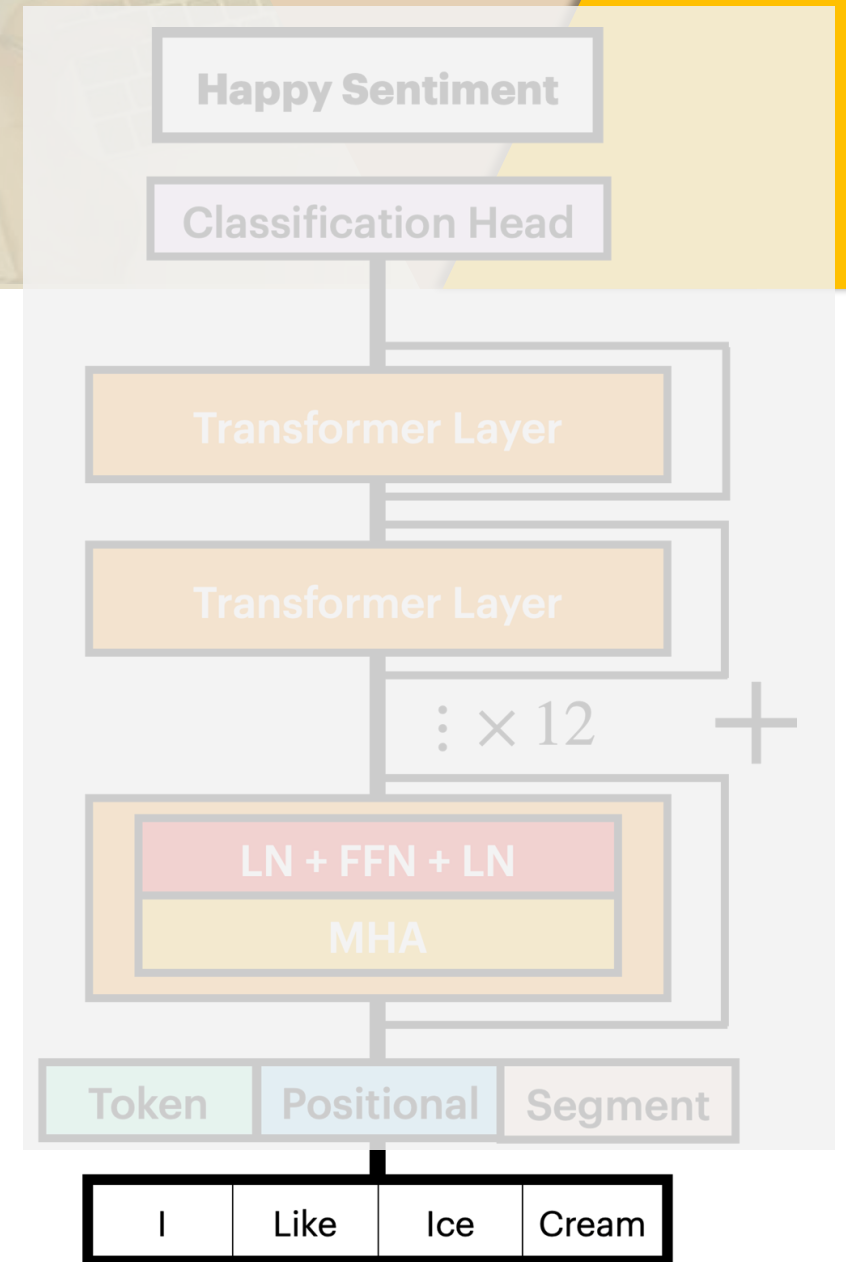
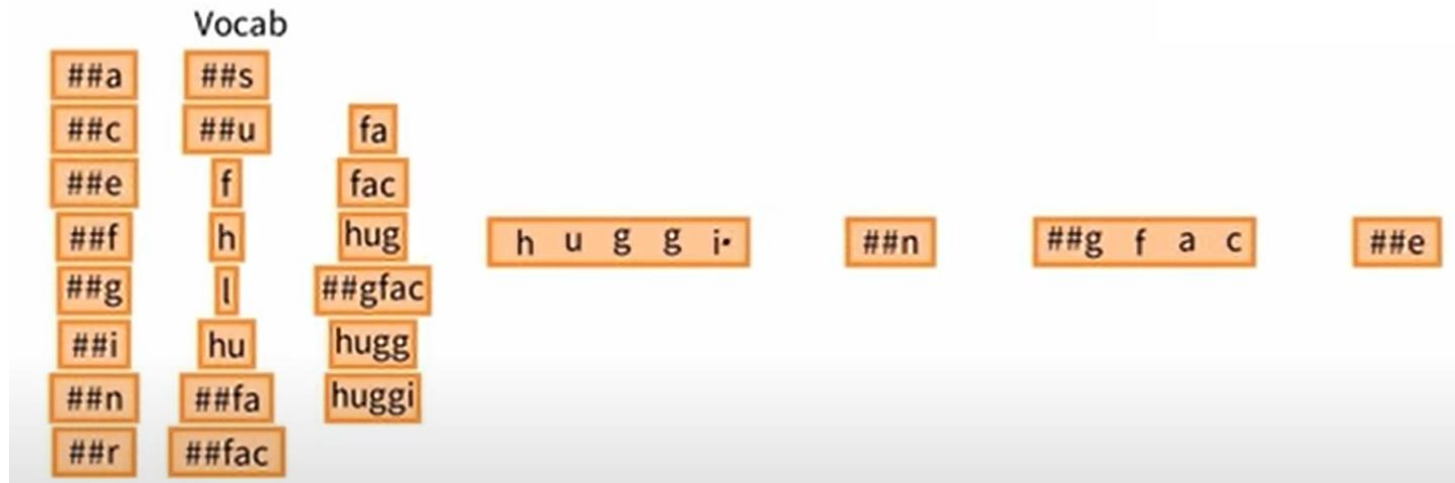
[Bert] Tokenization level : WordPiece



Animated explanation

https://www.youtube.com/watch?v=qpv6ms_t_1A&embeds_referring_euri=https%3A%2F%2Fhuggingface.co%2F

[Bert] Tokenization level : WordPiece



Animated explanation

https://www.youtube.com/watch?v=qpv6ms_t_1A&embeds_referring_euri=https%3A%2F%2Fhuggingface.co%2F

[Bert] Embedding level

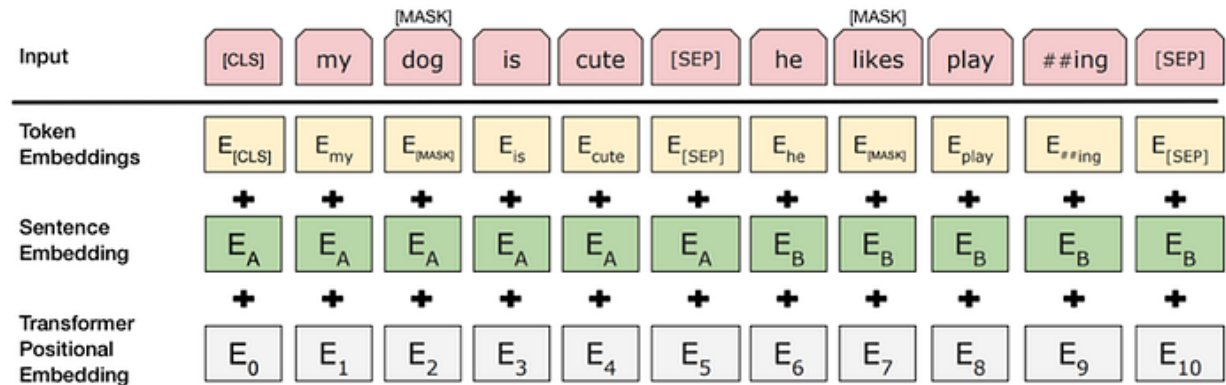
- **BERT's sequence length is 512 tokens**

This is the **maximum input length per sequence**, including all tokens ([CLS], [SEP]) and all tokens of the text. Any input longer than this will be *truncated*; shorter inputs are *padded*.

- **Embedding matrix**

BERT uses a **word embedding matrix** of size $V \times H$, where V is the vocabulary size and H is the hidden size (e.g., 768 for BERT-base).

- Each token is first converted to its embedding using this matrix.
- BERT then **adds segment embeddings** (to distinguish sentence A from sentence B) and **positional embeddings** (to encode token positions in the sequence) to produce the final input embeddings.



[Bert] Embedding level

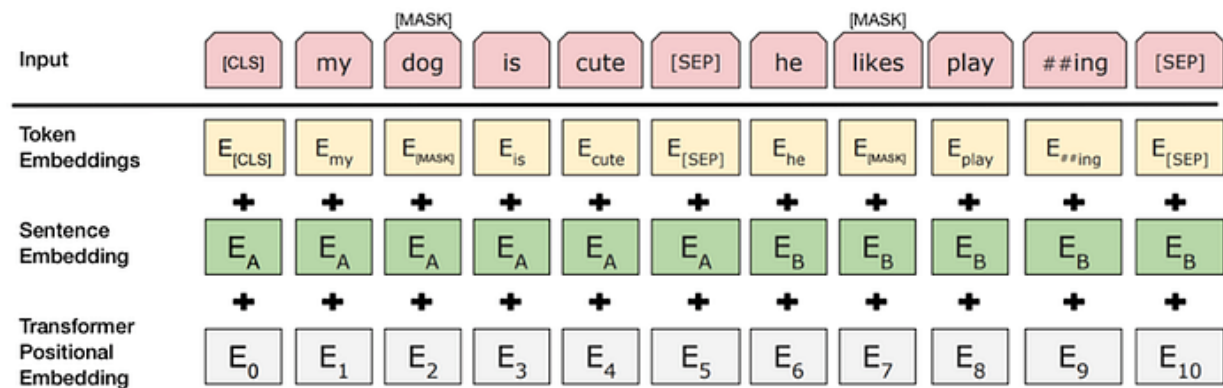
- **[SEP] token**

The [SEP] token is **not based on meaning or punctuation**. It is used to:

1. **Separate two segments** (e.g., sentence A and sentence B)
2. **Mark the end of a single sequence**

- **[CLS] token**

- **[CLS]** is the **classification token** placed at the beginning of the sequence.
 - Its **final hidden state** (output of the last Transformer layer) is used for **classification tasks**, such as sentence classification or next-sentence prediction.



[Bert] Encoder

Before entering the encoder, **multi-head attention** is the first component that analyzes your token embeddings.

- From your token embeddings, which have a dimension of **number_of_tokens × 768**,
- BERT generates three matrices: **K (Key)**, **Q (Query)**, and **V (Value)**:

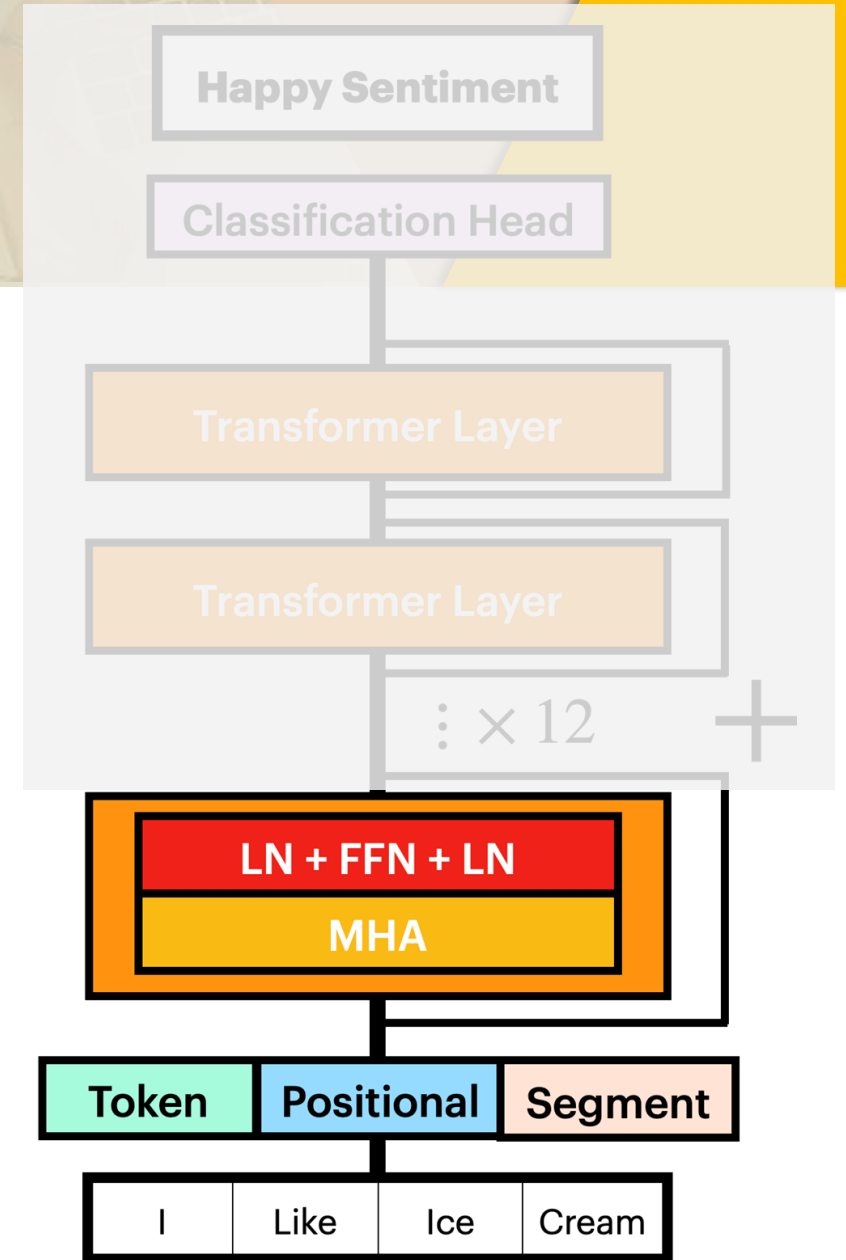
$$K = W_K \cdot \text{Embedding_matrix},$$

$$Q = W_Q \cdot \text{Embedding_matrix},$$

$$V = W_V \cdot \text{Embedding_matrix}$$

Each of **K**, **Q**, and **V** has the shape **number_of_tokens × 768**.

BERT uses **12 attention heads** in each layer.
Each head only processes **64 dimensions** of each token in parallel (since $768 \div 12 = 64$).



[Bert] Encoder (inside one head attention)

In **multi-head attention(MHA)**, each head learns different projections of **Q, K, and V**, allowing each head to capture token relationships from a different “subspace” or perspective.

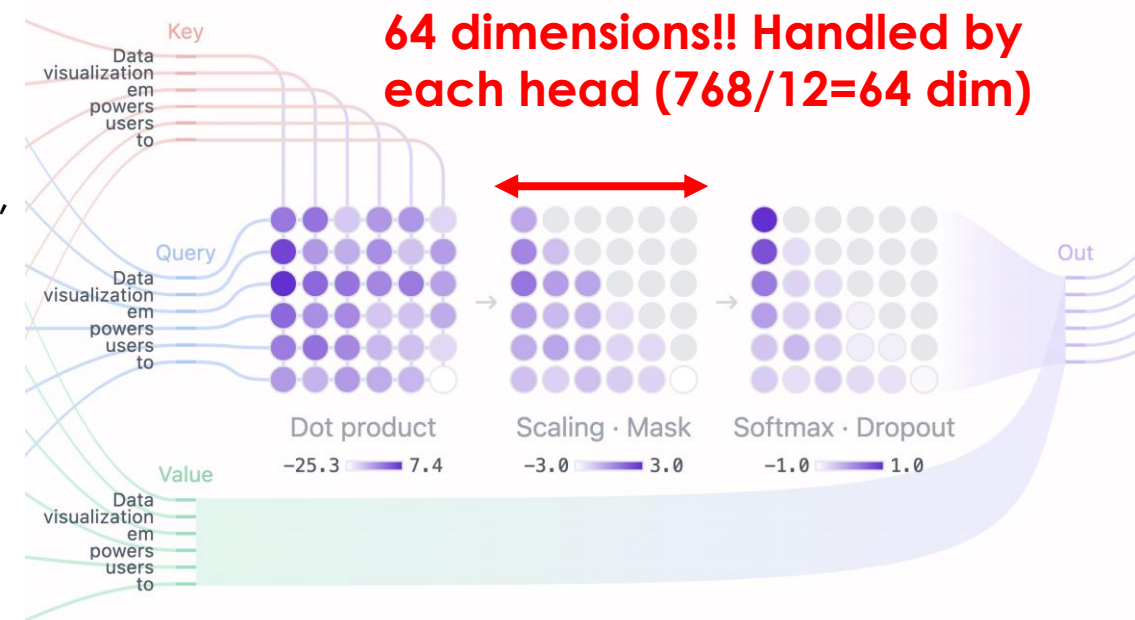
This is why we say each head “sees token relationships from another side”:

- One head might focus on **local syntactic relations**,
- Another head might capture **long-range dependencies**,
- Yet another might pick up **semantic patterns**.

Essentially, we compute the **co-occurrence scores** between **Q and K**, and **V** contains the **weighted token representations** according to these attention scores.

After all heads compute attention in parallel, their outputs are **concatenated** and passed through a **linear layer** to form the final representation for the layer.

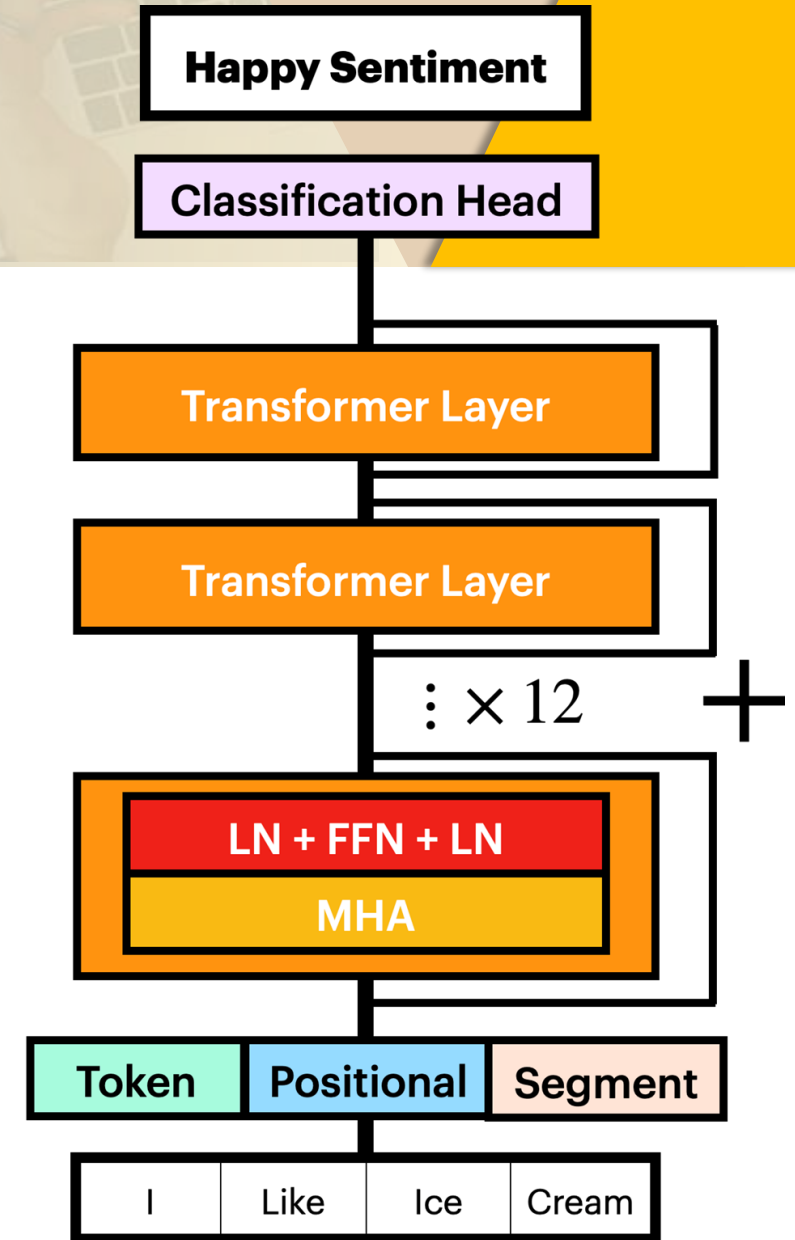
$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V$$



[Bert] Encoder .. 12 times!

Only MHA are parallel
You have 12 MHA in each encoder

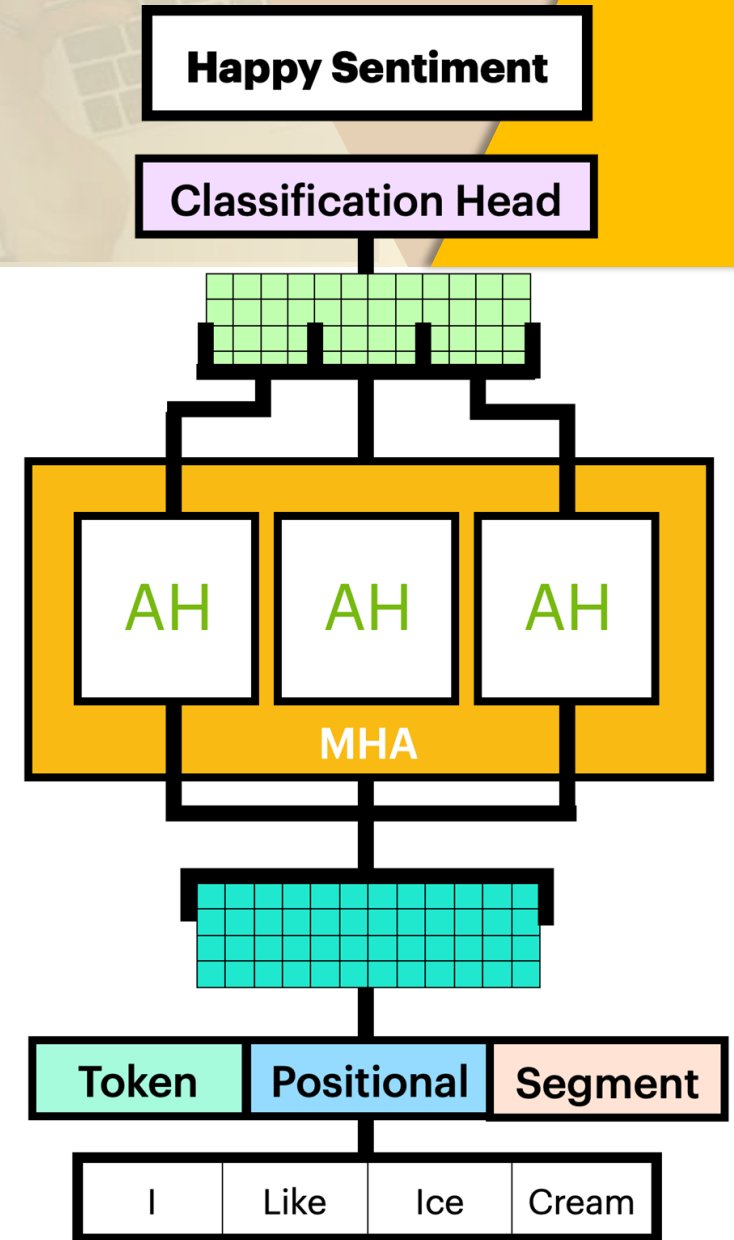
And you have 12 Encoders to be executed sequentially



Role of each component in BERT

Problem: Need Deeper Reasoning

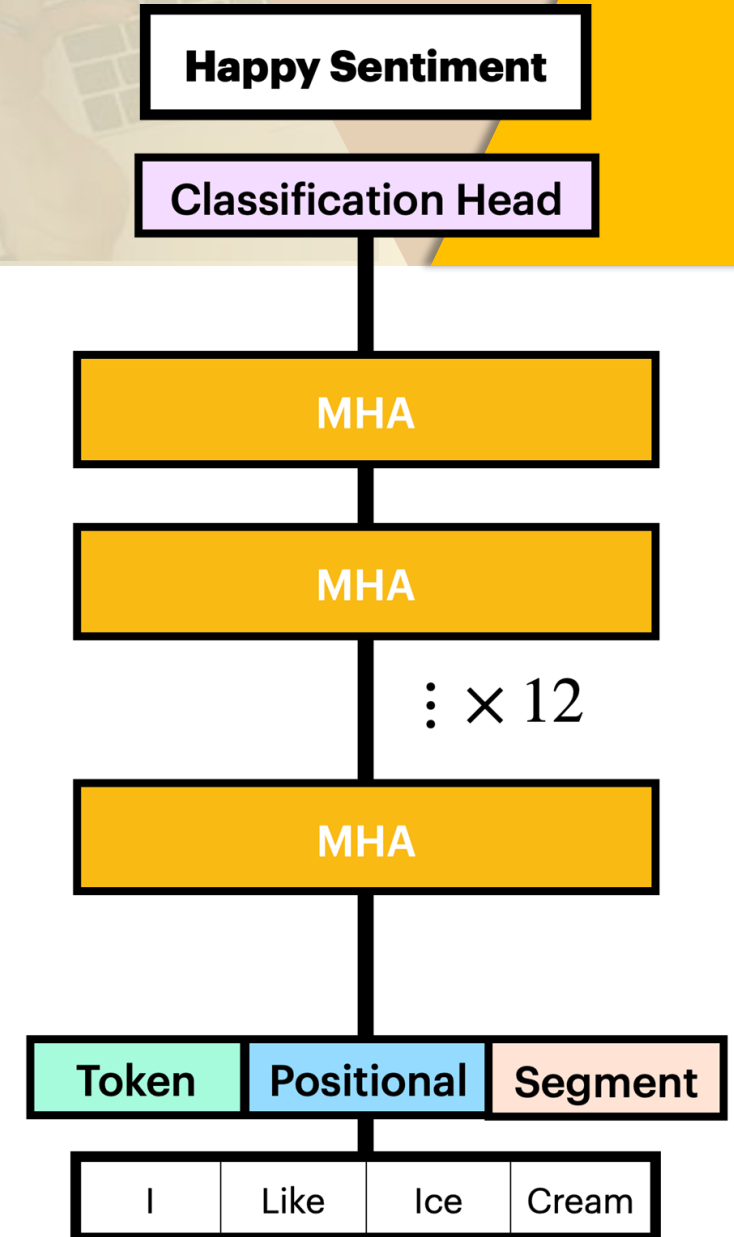
— On same data representation? **Multi-Headed Attention**



Role of each component in BERT

Problem: Need Deeper Reasoning

- On same data representation? **Multi-Headed Attention**
- Need more representations? **Multiple Transformer Layers**



Role of each component in BERT

Problem: Need Deeper Reasoning

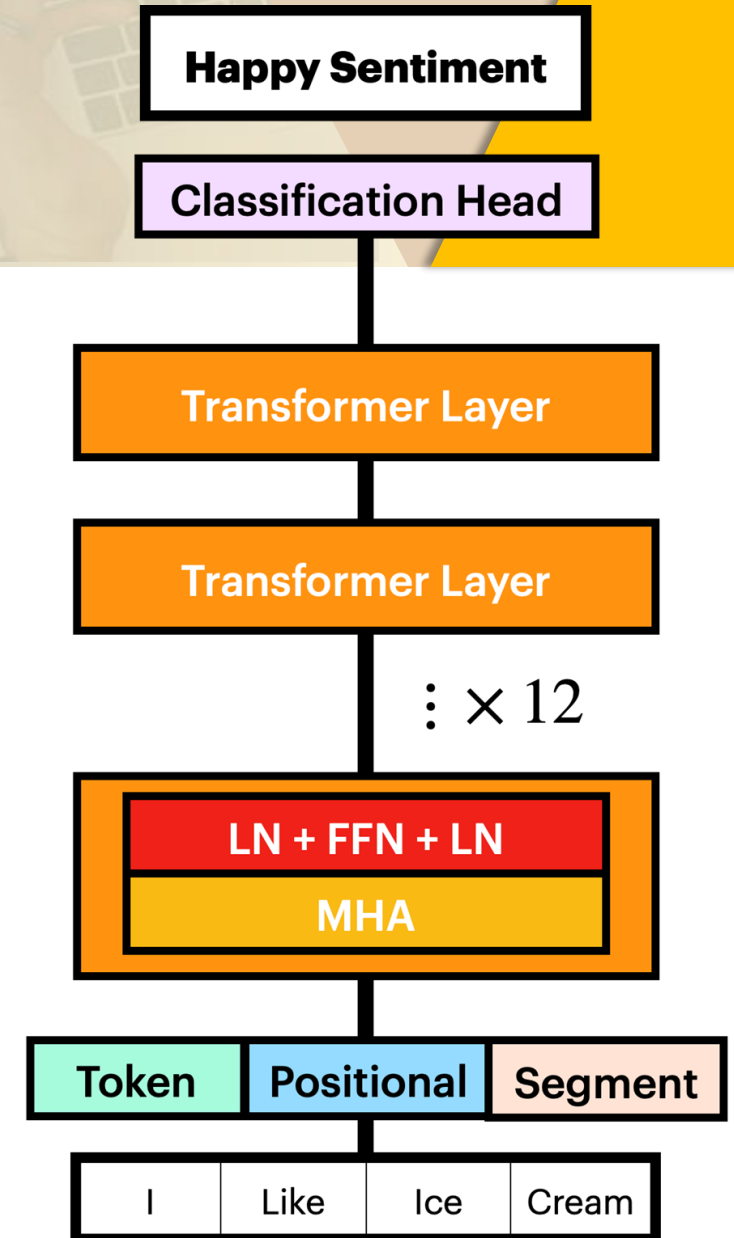
- On same data representation? **Multi-Headed Attention**
- Need more representations? **Multiple Transformer Layers**

Problem: Volatile gradients behavior, distribution shifts.

- Layer Normalization**
- Gaussian Error Linear Unit (GELU)**
- Dropout During Training**



FFN weights and biases do not change during inference or feature extraction, but they are updated through backpropagation when BERT is fine-tuned.



From transformers to BERT

Problem: Need Deeper Reasoning

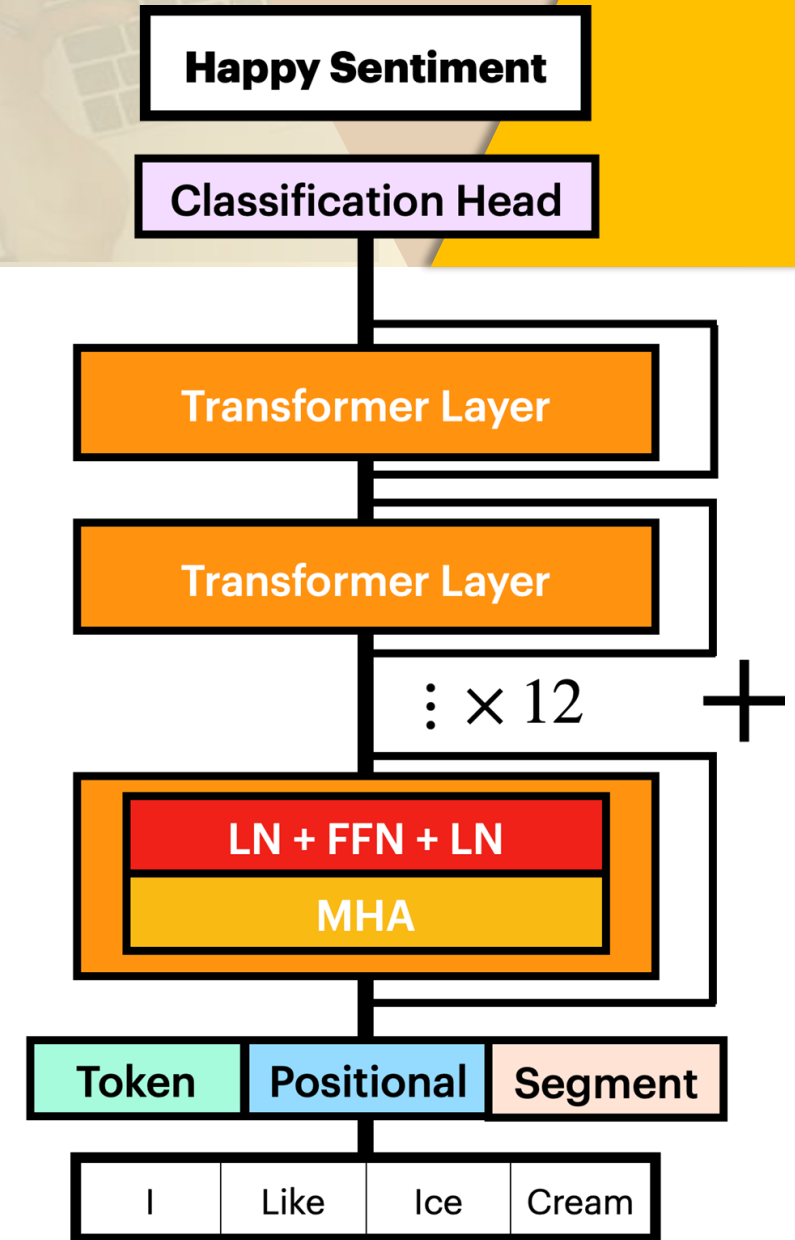
- On same data representation? **Multi-Headed Attention**
- Need more representations? **Multiple Transformer Layers**

Problem: Volatile gradients behavior, distribution shifts.

- Layer Normalization**
- Gaussian Error Linear Unit (GELU)**
- Dropout During Training**

Problem: Network too deep; Gradients are vanishing

- Residual Connections**



Pretraining BERT – Self Supervision Learning

BERT is pretrained using **two self-supervised tasks**:

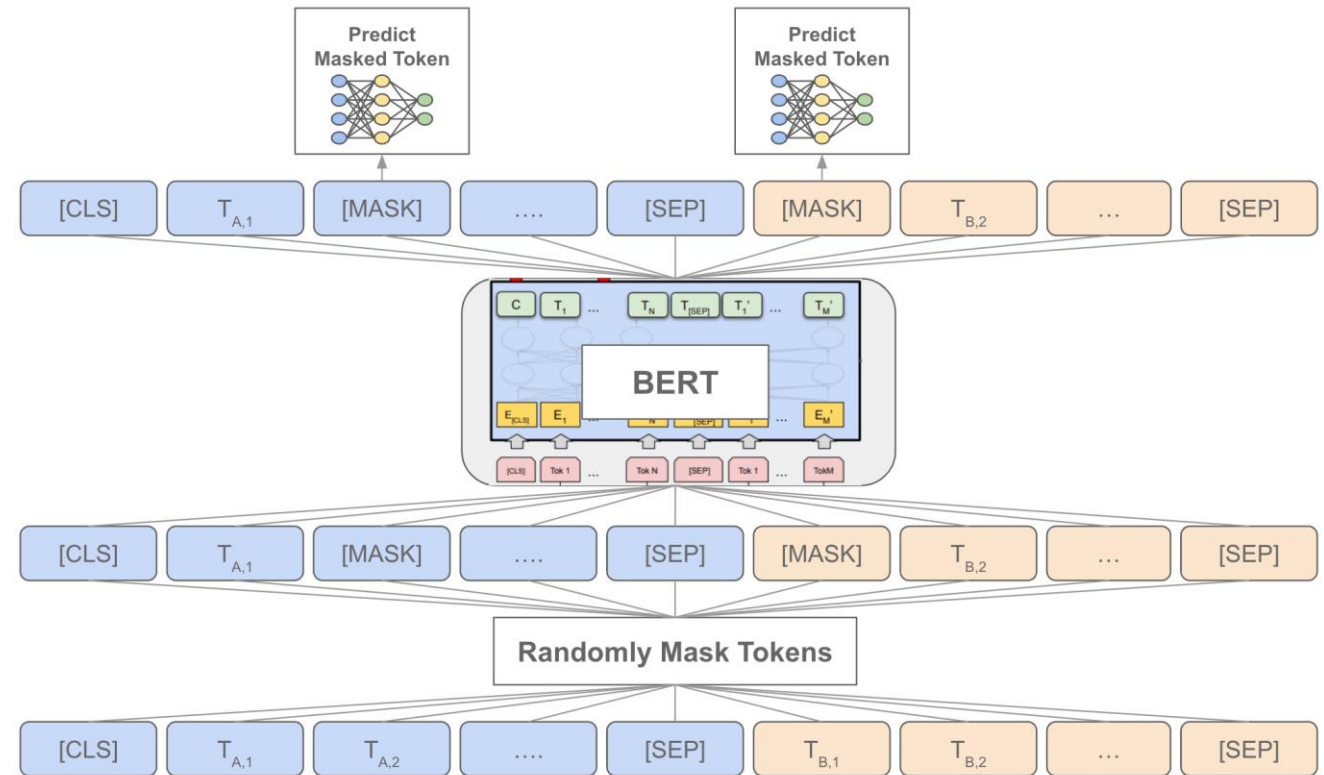
1. Masked Language Modeling (MLM)

1. Randomly masks some tokens in the input sentence (e.g., [MASK]) and asks the model to **predict the original token**.
2. Helps BERT **learn deep contextual word representations**.

2. Next Sentence Prediction (NSP)

- Given a pair of sentences, the model predicts whether **the second sentence actually follows the first one** in the original text.

RoBERTa(Robustly Optimized BERT Approach) is proposed essentially to enhance BERT pretraining phase



BERT for sentiment analysis

```
python

from transformers import BertTokenizer, BertForSequenceClassification

import torch

# Load pretrained BERT tokenizer and model
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)

# Example text
texts = ["I love this product!", "This is terrible."]
labels = torch.tensor([1, 0]) # 1=positive, 0=negative

# Tokenize
inputs = tokenizer(texts, padding=True, truncation=True, return_tensors="pt")

# Forward pass
outputs = model(**inputs, labels=labels)
loss = outputs.loss
logits = outputs.logits

# Convert logits to predicted classes
preds = torch.argmax(logits, dim=1)
print(preds)
```

 Copy code

- **BertTokenizer**: converts text into token IDs, attention masks, and special tokens using WordPiece.
- **BertForSequenceClassification**: BERT with a classification head on top of the [CLS] token, suitable for tasks like sentiment analysis, spam detection, or intent classification. (“Build a classification head on top of [CLS]” = add a linear layer that takes [CLS] as input and outputs class scores.)

← Not crucial for production

BERT for sentiment analysis

Step 2: Tokenize the dataset

- Use BERT's tokenizer to convert text → token IDs, attention masks.
- Decide **max sequence length** (e.g., 128 or 256 tokens).
- Optionally, save tokenized dataset to speed up training.

python

 Copy code

```
inputs = tokenizer(texts, padding=True, truncation=True, return_tensors="pt", max_length=128)
```

Step 3: Create DataLoader for batching

- Convert your tokenized inputs and labels into a **PyTorch Dataset**.
- Use **DataLoader** to handle batching, shuffling, and GPU loading.

python

 Copy code

```
from torch.utils.data import DataLoader, TensorDataset

dataset = TensorDataset(inputs['input_ids'], inputs['attention_mask'], labels)
dataloader = DataLoader(dataset, batch_size=16, shuffle=True)
```

BERT for sentiment analysis

Step 4: Define optimizer and learning rate

- Usually `AdamW` is used for fine-tuning BERT.

```
python

from torch.optim import AdamW

optimizer = AdamW(model.parameters(), lr=2e-5)
```

 Copy code

Step 5: Fine-tune BERT

- Loop over **epochs** and **batches**:

```
python

model.train()
for epoch in range(num_epochs):
    for batch in dataloader:
        input_ids, attention_mask, batch_labels = batch
        outputs = model(input_ids=input_ids,
                        attention_mask=attention_mask,
                        labels=batch_labels)

        loss = outputs.loss
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

 Copy code

Step 6: Evaluation / Prediction

- After training, switch to evaluation mode:

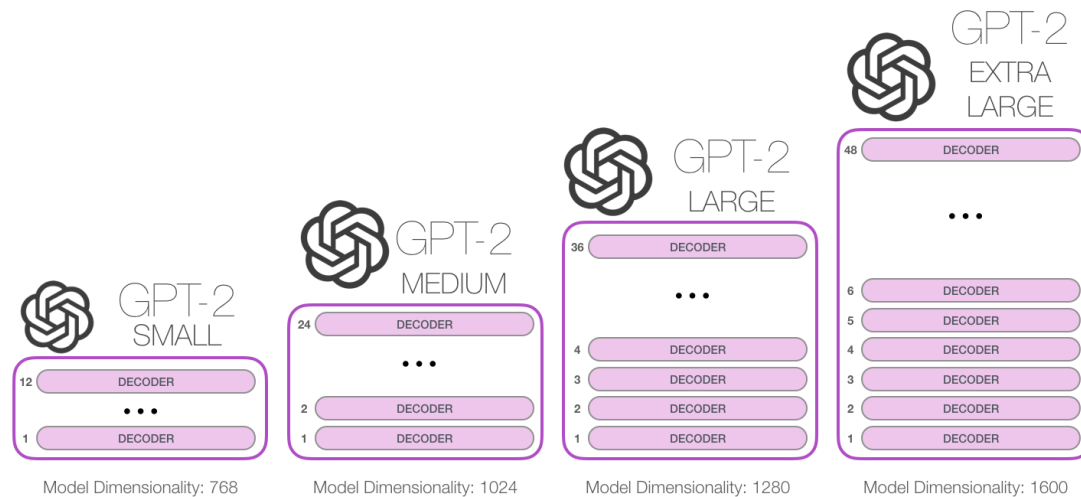
```
python

model.eval()
with torch.no_grad():
    outputs = model(input_ids=input_ids, attention_mask=attention_mask)
    predictions = torch.argmax(outputs.logits, dim=1)
```

 Copy code

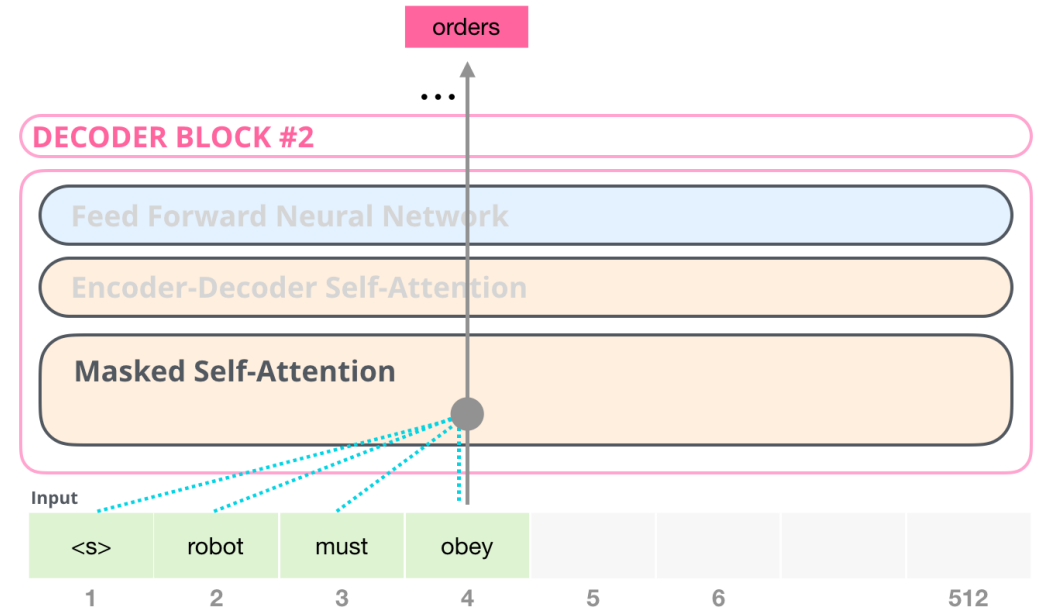
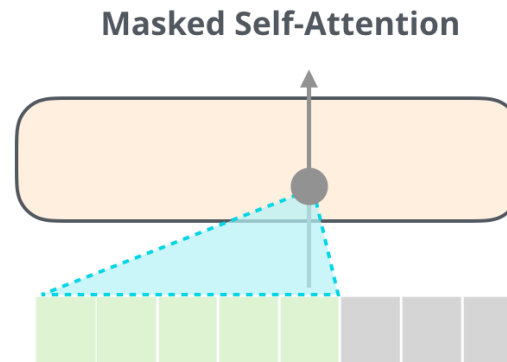
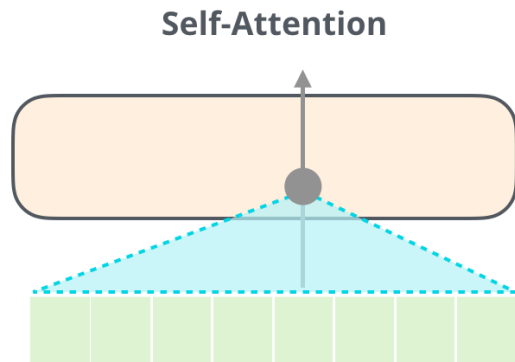
Decoder Part –GPT2

The GPT-2 is built using transformer decoder blocks. GPT2, like traditional language models, outputs one token at a time.



Decoder Part –GPT2

Autoregressive Modeling: A training approach where the model predicts the next token in a sequence based only on the tokens that come before it. This is implemented using causal language modeling, which applies a mask in self-attention to prevent the model from “seeing” future tokens.



Decoder Part –GPT2

in the beginning, we look up the embedding of the start token **<s>** in the embedding matrix.

Before handing that to the first block in the model, we need to incorporate positional encoding – a signal that indicates the order of the words in the sequence to the transformer blocks.

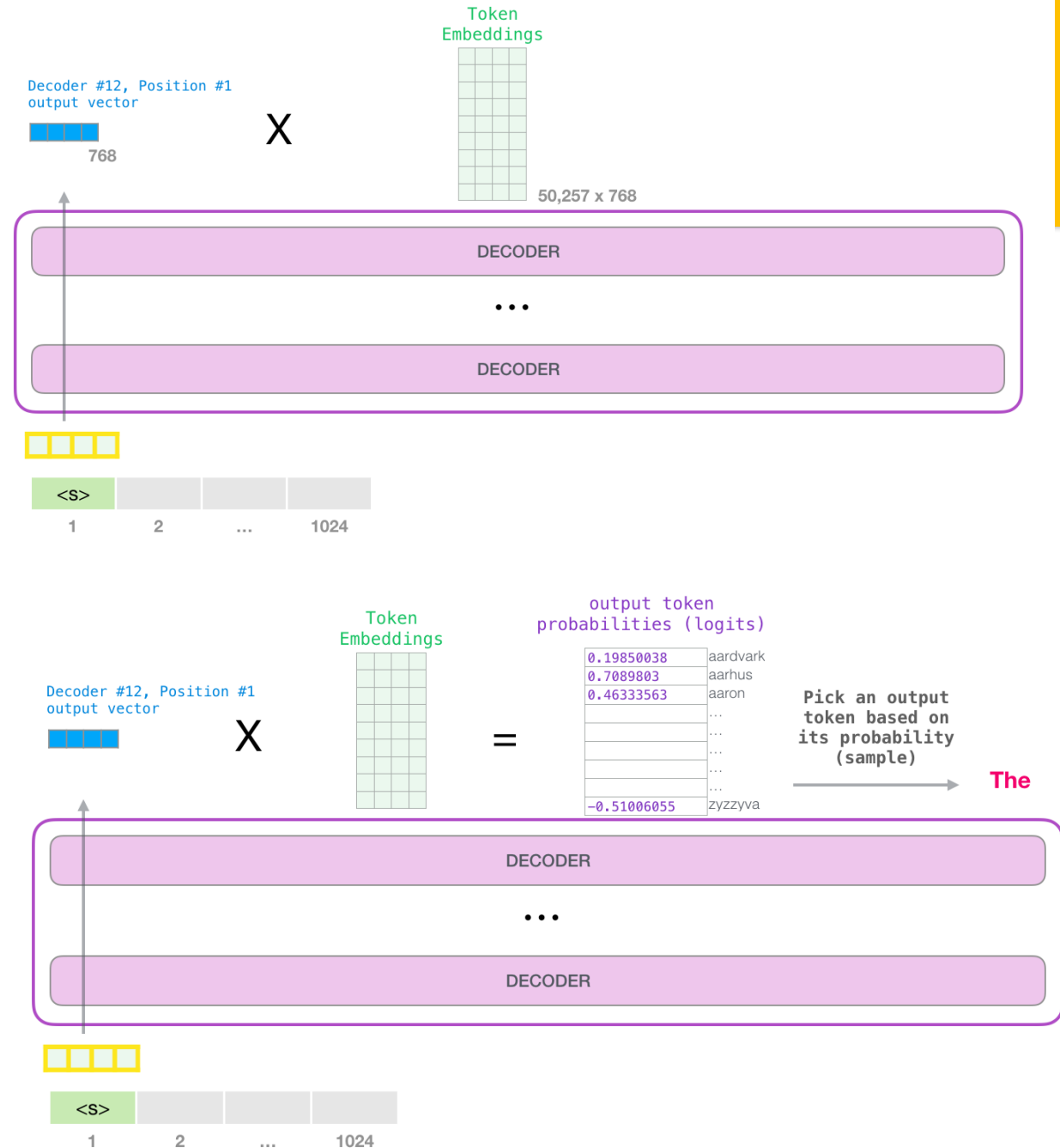
Part of the trained model is a matrix that contains a positional encoding vector for each of the 1024 positions in the input.



Decoder Part –GPT2

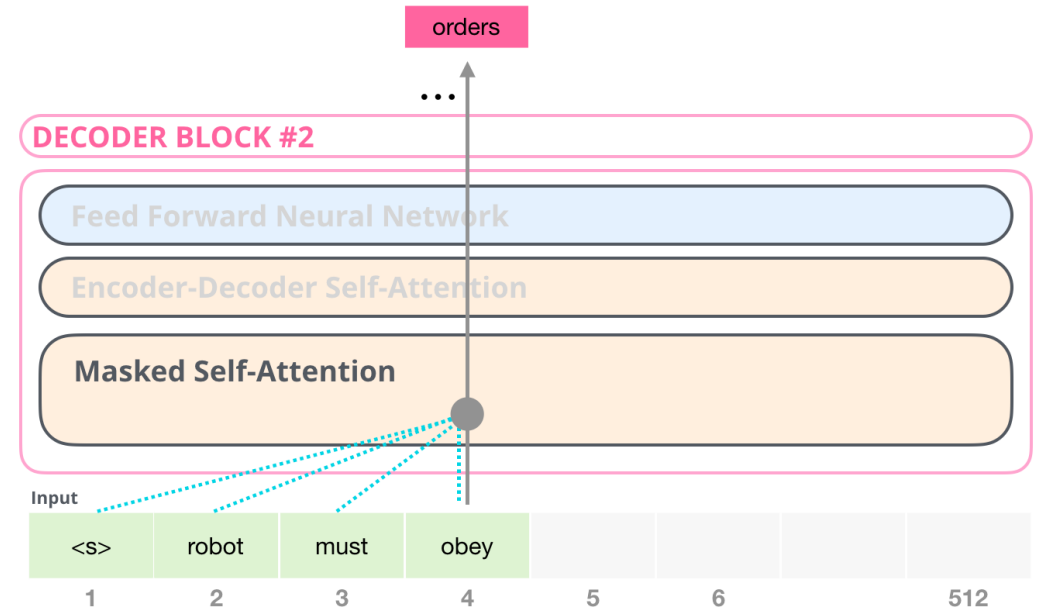
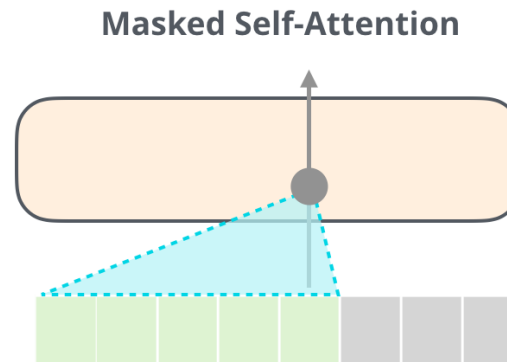
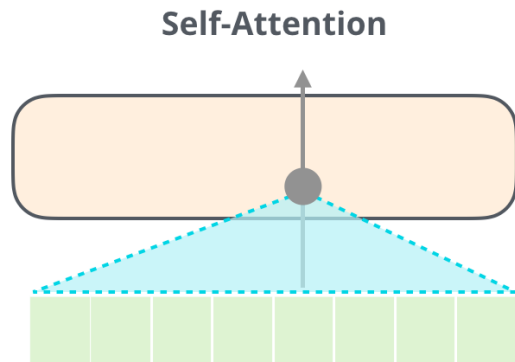
When the top block in the model produces its output vector (the result of its own self-attention followed by its own neural network), the model multiplies that vector by the embedding matrix.

Recall that each row in the embedding matrix corresponds to the embedding of a word in the model's vocabulary. The result of this multiplication is interpreted as a score for each word in the model's vocabulary.



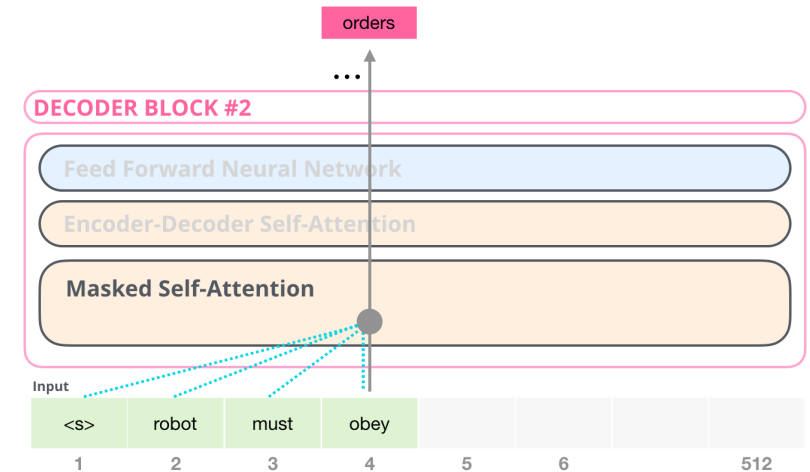
Decoder Part –GPT2

- Once we have a trained autoregressive model, we need to generate text. Common strategies:
- Greedy Decoding:** pick the highest-probability next token each time.
- Beam Search:** keep multiple hypotheses alive to find a more likely overall sequence.
- Top-k / Top-p Sampling:** sample from the top k tokens or the top tokens whose cumulative probability is p.



Decoder Part –GPT2

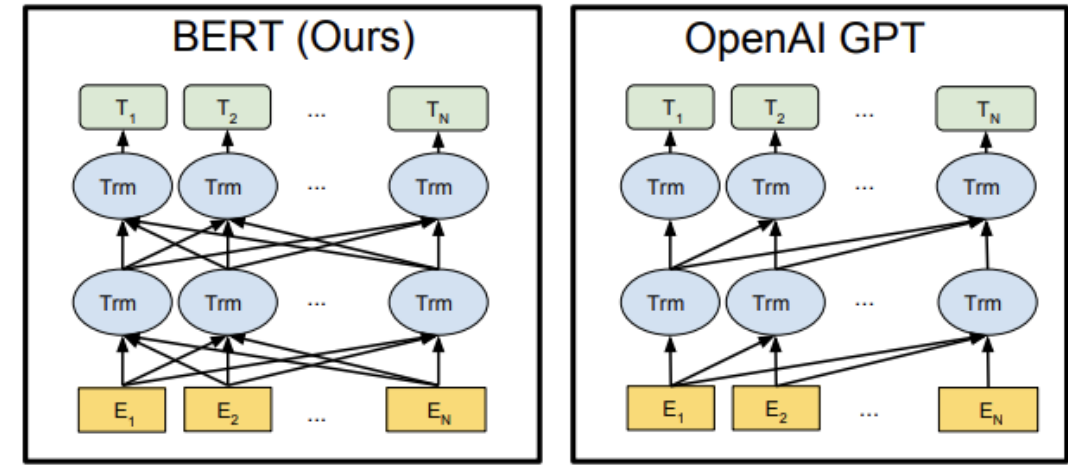
- **Pretraining Causal Language Modeling (CLM)**
- GPT learns to **predict the next token** given all previous tokens
- To enforce causality, GPT uses **masked (causal) self-attention** where each token can only attend to earlier tokens.
- GPT is pretrained on **very large unlabeled corpora**, such as:
- Books, websites (Common Crawl), articles code (GitHub),...
- **During Pretraining**
- It learns *general world knowledge and language structure*:
- grammar, syntax, semantic relations
- how concepts relate reasoning patterns, how instructions typically look, how code is structured common patterns in human text
- It does **not** learn specific tasks (classification, summarization, etc.) yet.



Decoder Part –GPT2

Training a transformer recap

- Transformers like BERT/GPT use the unlabeled text data as the sources of both input and output.
- In Autoregressive training, and in Masked Language Modeling the model aims to fill in the missing (or next) token.
- It uses the attention mechanism to add extra context to determine what the right token should be.
- This means no labels are needed and all kinds of data can be used from text, to images, to speech, to video!



Transformer – The big picture

Seq2Seq how does he work ?

Task Example

We want to translate:

Input (English): "I love apples"

Output (French): "J'aime les pommes"

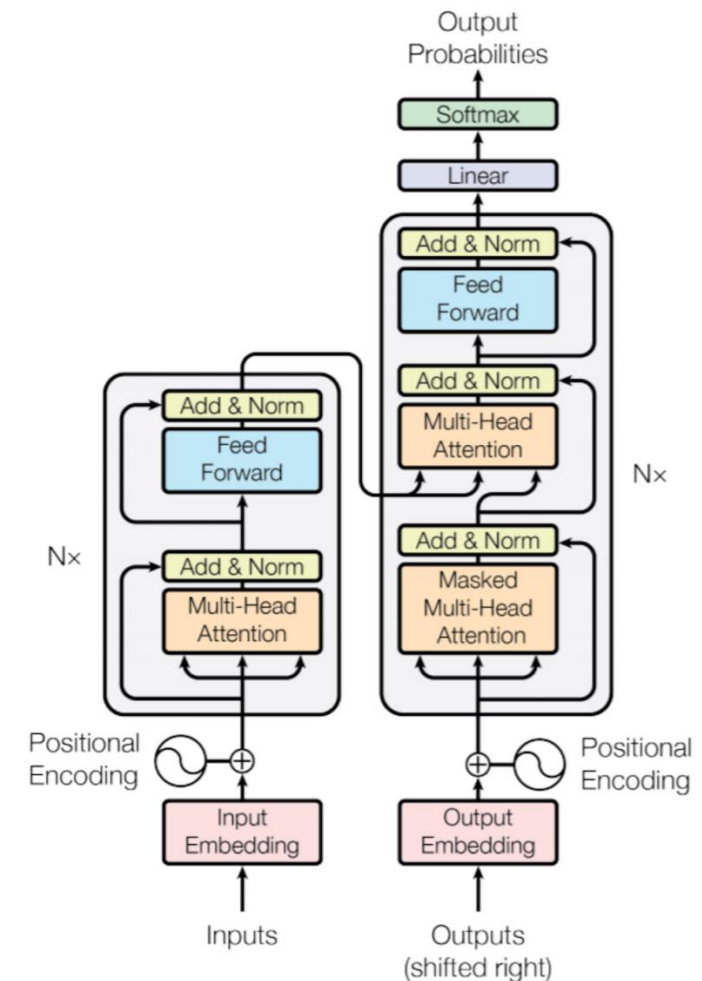
Step 1: Input Tokenization

- Convert words into **tokens** the model understands (subwords via WordPiece/BPE):

Input tokens (English): [I, love, apples] → [101, 7592, 6207, 102] # 101=CLS, 102=SEP

Output tokens (French): [J', aime, les, pommes] → [101, 1557, 5663, 2075, 102]

- These are **numerical IDs** fed into the model.



Transformer – The big picture

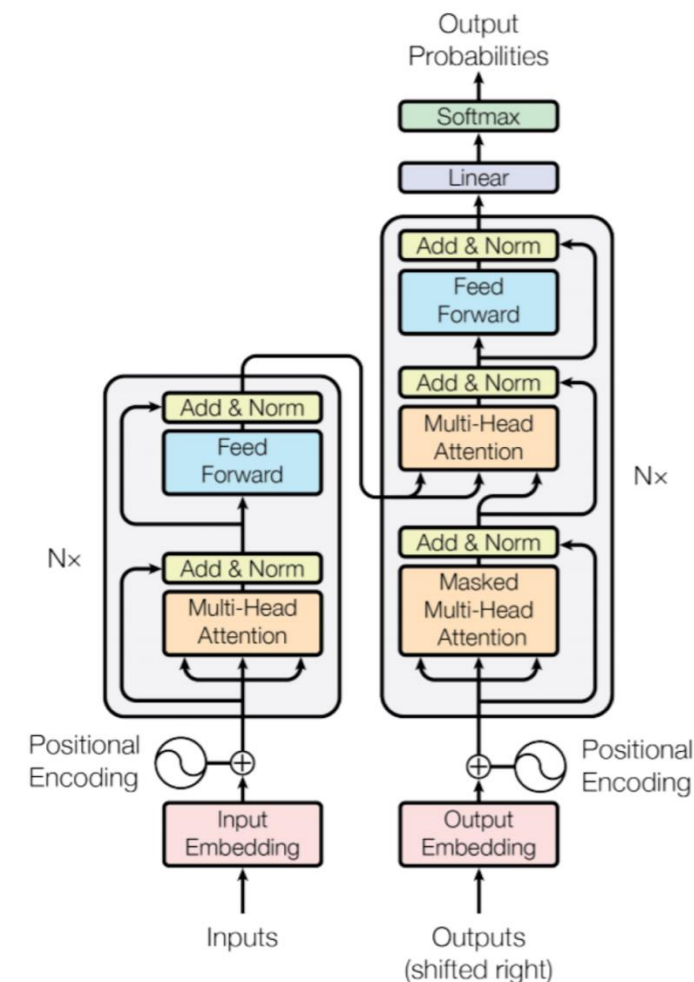
Step 2: Encoder

Purpose:

- Transform the **input sequence** into **contextual embeddings**.
- Understand **relationships between tokens**.

Steps inside the encoder:

1. **Input Embeddings:** convert token IDs $\rightarrow$ vectors
Example: [I, love, apples] $\rightarrow$ 3 vectors of size 512 or 768.
2. **Positional Encoding:** add position info (so the model knows the order of words)
3. **Self-Attention Layers** (multi-head):
 - Each token “looks at” every other token to understand context.
 - Example: “love” attends to “I” to understand subject.
4. **Feedforward Network**
5. **Stack multiple encoder layers** (e.g., 6 layers)
6. **Output:** Contextual embeddings for each token
7. Encoder output = [E_I, E_love, E_apples]



Transformer – The big picture

Step 3: Decoder

Purpose:

Generate **output sequence token by token** using **encoder outputs**.

Steps inside the decoder for autoregressive generation:

Input tokens so far: e.g., start with <BOS> or <S> or <CLS> token

Decoder input step 1: <BOS> or <S>

Masked Self-Attention

Each token can only see **tokens generated before it**

Prevents “cheating” (looking at future tokens)

Encoder-Decoder Attention

Each token in the decoder attends to **all encoder outputs**

Allows decoder to incorporate **context from the input sequence**

Example: decoder at step 1 looks at [E_I, E_love, E_apples]

Feedforward layer

Linear + Softmax

Produces **probability distribution over the vocabulary**

Example: step 1 probabilities: J' 0.7, Je 0.2, Nous 0.1

Decoding strategy

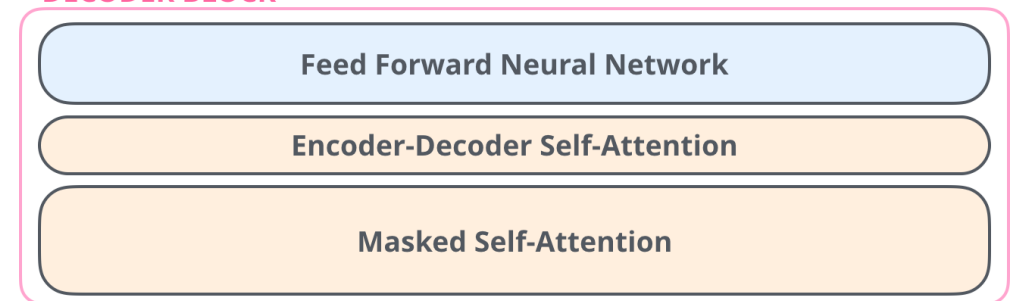
Greedy / Beam / Top-k / Nucleus → selects next token

Append selected token to decoder input, repeat until <EOS>



THE TRANSFORMER

DECODER BLOCK



Input



T5 model : Encoder - Decoder

T5 (Text-to-Text (or Seq2Seq) Transfer Transformer) is a **transformer encoder-decoder model** from Google.

Text-to-Text paradigm:

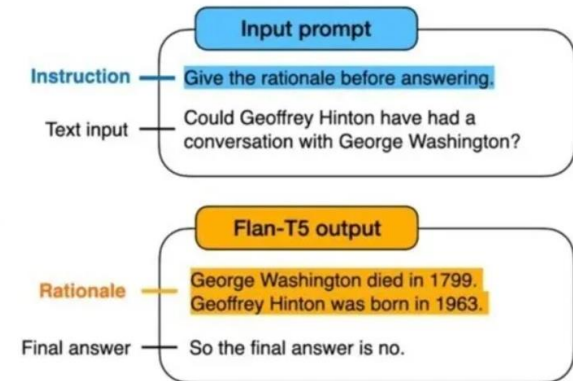
Every task is converted into a **text input** → **text output** problem.

Examples:

- Translation: "translate English to French: I am happy" → "Je suis heureux"
- Summarization: "summarize: <long text>" → <summary>

Encoder-decoder structure:

- **Encoder:** processes the input text and produces hidden representations
- **Decoder:** autoregressively generates output tokens based on encoder outputs



T5 model : Encoder – Decoder: prompting / reasoning technique

The **purpose of reasoning techniques in language models** is to guide the model to reason better **without changing its weights**. (to provide during inference in the prompt, not possible with BERT)

Chain-of-Thought means the model **generates intermediate reasoning steps before the final answer**.

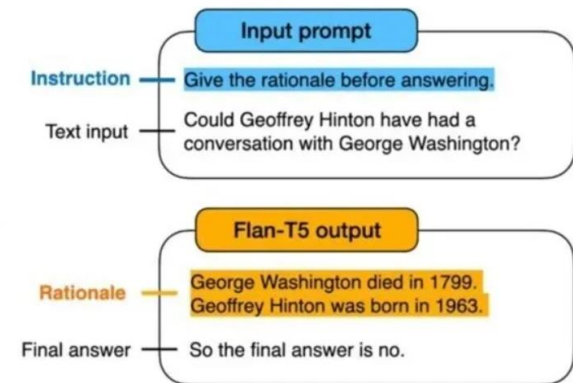
Self-Consistency (SC) : Instead of generating **one chain of thought**, generate **multiple reasoning paths** and take the **most frequent final answer**.

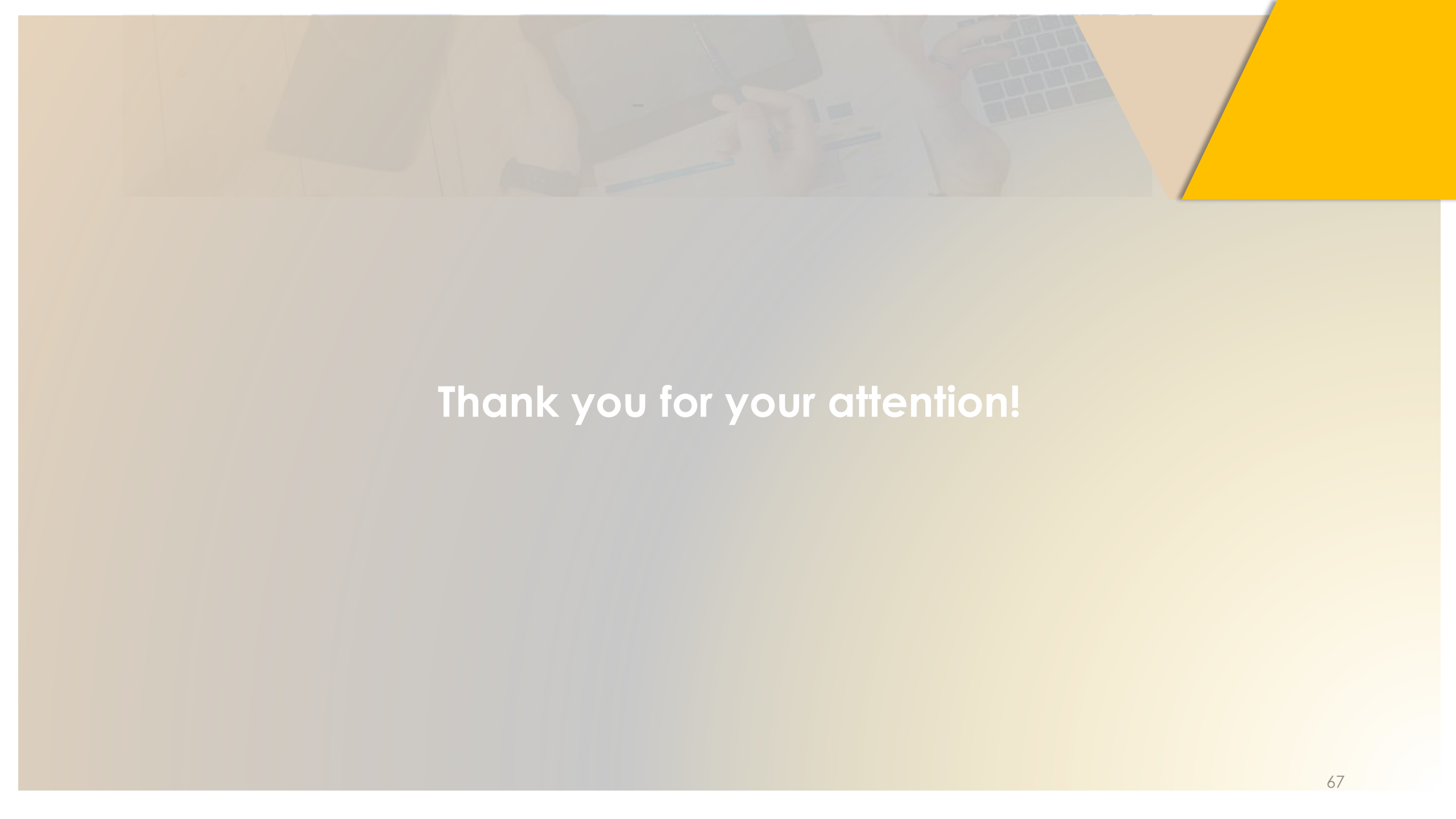
In-Context Learning = the model **learns from examples provided in the prompt**, without updating its weights.

Prompt Engineering / Zero/Few-Shot Reasoning

Techniques include:

- Adding “**step by step**” instructions
- Providing **few-shot examples** with reasoning steps
- Asking for **structured output** like numbered steps





Thank you for your attention!