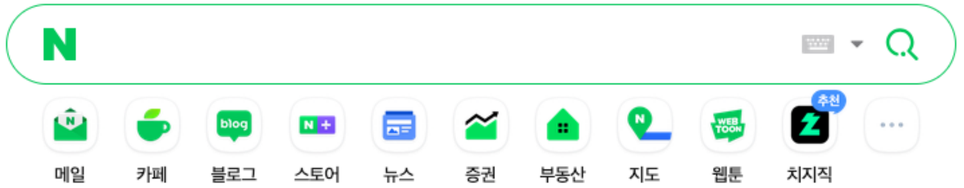
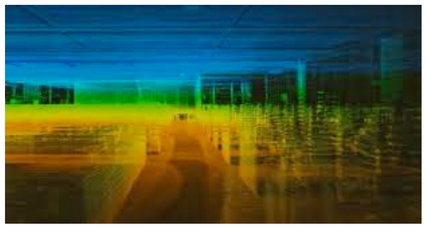
# Transformers architecture
# Large Language Models
# Intro to modern natural language processing
# Part I

**Maxime LOUIS**
Research Scientist
Naver LABS Europe
maxime.louis@naverlabs.com

# Naver Labs




- **Naver:**
  - Search engine (75% of market in Korea)
  - Line (70% of market in Japan/Indonesia/ Taiwan/Korea)
  - Webtoon (sharing manhwa)
  - Pay, cloud, computing …
- **Naver Labs Europe:** AI research lab in Meylan
  - LLM, NLP
  - Vision and multimodal models
  - Robotics application

# Naver Labs Europe

- I’m part of a team which works on :
  - Retrieval Augmented Generation: given a query, performing a search step before resorting to a Large Language Model
  - Agents
  - Compressed generation models

**There are internships available, feel free to reach us**

# Course outline

- Part I:
  - Quick reminders about machine/deep learning
  - Natural Language Processing: motivation for the transformers architecture
  - Tokenization
  - Text embeddings
  - Causal/Masked language Modeling: self supervised learning for NLP
  - The transformers architecture (part I): overview, attention layer
  - TP1: Byte-Pair encoding + implementing the attention layer
- PART II
  - The transformers architecture (part II): MLP, normalization, variations, limitations. Pretraining transformers: emergent behaviors. Fine-tuning transformers. TP2: Finishing implementing GPT-2/3

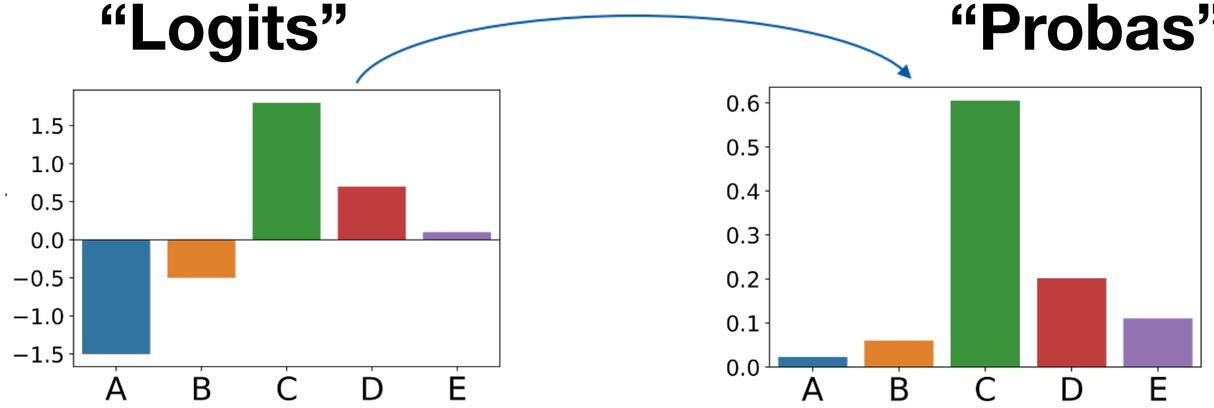
# Reminders about deep learning

# Useful functions: quick reminder

## Softmax:

$$z = [z_1, \ldots, z_K] \in \mathbb{R}^K \quad \rightarrow \quad s = [s_1, \ldots, s_K] : \; s_k \geqslant 0, \sum_{k=1}^{K} s_k = 1$$

**“Logits”** **“Probas”**




$$s_k = \frac{exp(z_k)}{\sum_{c=1}^{K} exp(z_c)}$$

Used e. g. in multiclass classfication

transforms a real-valued vector into a categorical distribution

```
import torch
import torch.nn.functional as F

# Example logits (pre-activation values)
logits = torch.tensor([2.0, 1.0, 0.1])

# Apply softmax
softmax_output = F.softmax(logits, dim=0)
# tensor([0.6590, 0.2424, 0.0986])
```

# Cross-entropy loss

$$L = -\sum_{i=1}^{N} \log(\hat{y}_{it_i})$$

- Where
  - N is the number of samples (e.g. in a batch)
  - $\hat{y}_{ij}$ : predicted probabilities of class j for sample i
  - $t_i$ is the ground truth class of sample I
- **Minimizing L maximizes** $\hat{y}_{i,t_i}$

**WARNING:** loss implementations may expect logits OR probas. They may also expect ints and not one-hot encoding for the labels

# Stochastic Gradient Descent

- Given a parametric function $f$ with parameters $\theta$ (e.g. a neural network), a batch of inputs $(x_1, \ldots, x_b)$, a batch of targets $(y_1, \ldots, y_b)$ and a loss function $L$. The stochastic gradient update on $\theta$ is:

$$\theta \leftarrow \theta - \eta \nabla_\theta \Big( \frac{1}{b} \sum L(f_\theta(x_i), y_i) \Big)$$

Where $\eta$ is the *learning rate*.

For neural network, the gradient is computed by back propagation.

# PyTorch basics

- VS CODE

# Natural language processing tasks

## Discriminative tasks

Text → ?

### Sentiment analysis

Food was awful, and the waiter spilled wine on our table → **negative**

### Intent classification

I would like to send an e-mail to the HR → **E-mail**

Natural language understanding (NLU)

## Generative tasks

? → Text

### Autocompletion

The truth is… →
**I don't have a lot of creative advice to give.**

### Generating captions for images

→ **a group of people jumping on the beach**

Natural language generation (NLG)

# Tasks solved in NLP

## Discriminative tasks

- Classification:
  - Sentiment analysis
  - Categorisation into topics
  - Spam detection
  - Intent recognition
  - Detecting automatically generated text
- Information extraction

## Tagging tasks

- Part-of-speech tagging
- Named entity recognition

## Generative tasks and seq2seq

- Machine translation
- Text summarisation
- Text simplification and style transfer
- Text generation, e. g. assistance in writing e-mails, reports, articles…
- Dialog
- Question answering
- Grammar correction

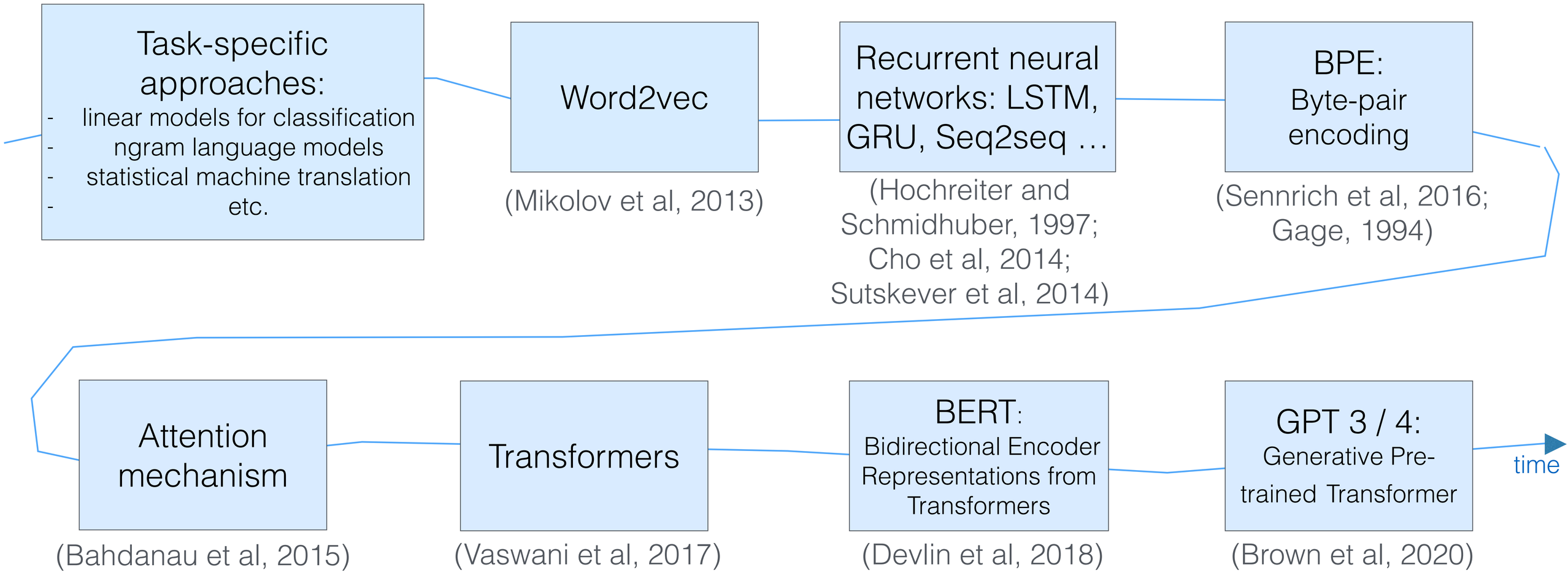
+speech

- Speech recognition
- Text-to-speech generation
- Speech translation

+code

- Automatic code commenting and documentation
- Code generation
- Automatic bug fixing

# (Very) brief history of widely-used approaches in NLP




* contains only a small part of influential works in NLP

**(In this course, we only look at transformers: a very short-sighted view of NLP)**

# Transformers

- Almost all state-of-the-art NLP algorithms now rely on a transformer model.
- There are base transformers models called foundation models or large language models (LLM), **pretrained** on large (huge) datasets.
- They are “easily” adaptable to new tasks by **fine-tuning**
- Since the introduction of the transformers (2017), progress in NLP is extremely fast, unpredictable, exciting, scary, dramatic, energy-consuming…
- The transformers architecture is now used in most domains of ML: vision, alphaFold, speech-to-text, video understanding/generation …

# Why the transformers ?

- It:
  - Is fast (because highly parallelizable)
  - Scales extremely well with data size and number of parameters
  - Is highly expressive
  - The attention mechanism allows for complex long-range learned interactions within sequences

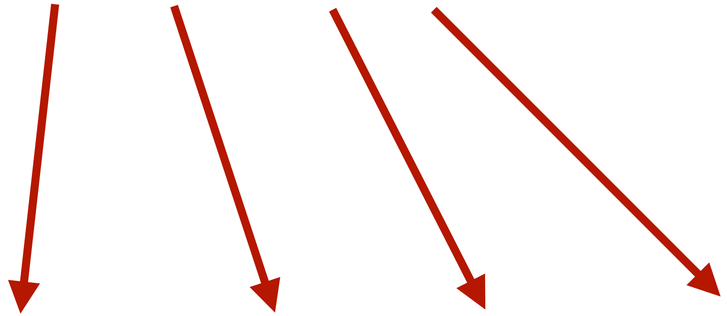
# Text preprocessing: tokenization

Which representation is best to feed text to a model ?

# Text representation: character-wise byte

- A text is a sequence of characters. Each character can be encoded into 8 bits

I started a band called 999 Megabytes. We haven’t gotten a gig yet.

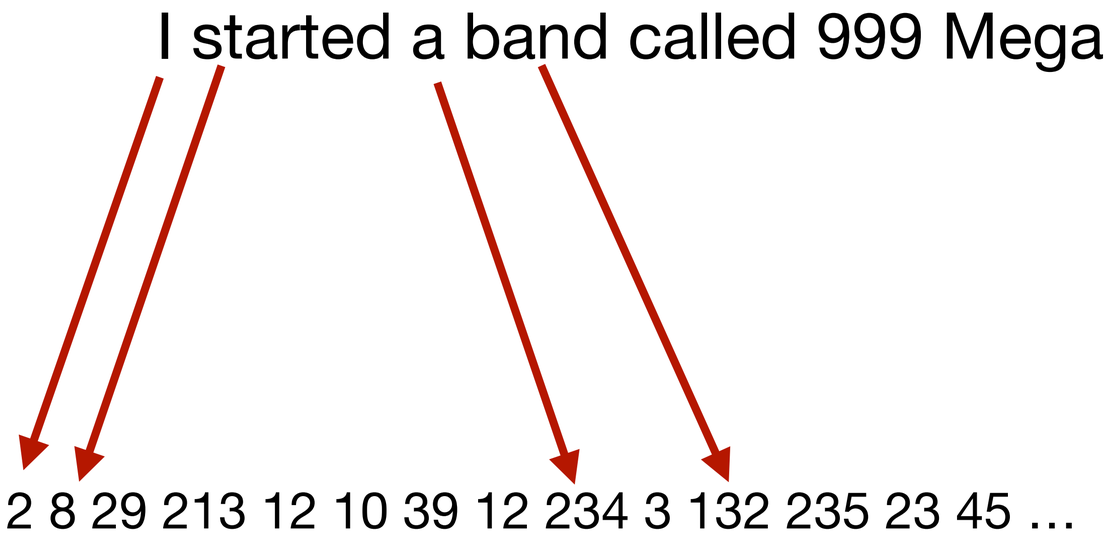


01001000 01101111 01110111 00100000 01100011 01100001 01101110 00100000
01001001 00100000 01110010 01100101 01110000 01110010 01100101 01110011
01100101 01101110 01110100 00100000 01100001 00100000 01110100 01100101
01111000 01110100 00100000 00111111

**Very bad representation! Too long, words are lost, sparsity.**

# Text representation: vocabulary=characters

- A text is a sequence of characters. Each character can be encoded into an integer in [0, 255]




**Bad representation! Too long, word structure is lost, sparsity (but less).**

# Text representation: vocabulary=words

- Parse your text corpus to compute the *vocabulary* V. Map each word to an integer in 1 … |V|.

```
'I'              → 9138
'ate'            → 2451
'a'              → 4943
'clock'          → 4606
'yesterday.'     → 8022
'It'             → 7625
'was'            → 8707
'very'           → 8307
'time-consuming' → 4866
```

- English wikipedia: **10 million words**

**=> poses learning problem**

# Text representation: vocabulary=words

- **Improvement:** lower-casing, punctuation removal, lemmatization (keep only the root of the word), rare words removal.

- Bring the count down to **100k words**. Still large but manageable in some cases

```
'Dr'            → 'dr' → 'dr' → 'dr' → 7197
'Watson's'      → 'watson's' → 'watson' → 'watson' → 1534
'Observations'  → 'observations' → 'observations' → 'observation' → 2845
'About'         → 'about' → 'about' → 'about' → 1922
'Sherlock's'    → 'sherlock's' → 'sherlock' → 'sherlock' → 3844
'Deductions'    → 'deductions' → 'deductions' → 'deduction' → 31
```

**Problems:**

- What about new words appearing at test time ?
- We cannot control the vocabulary size
- Rare words will rarely be seen (for training)

# Tokenization: Byte-pair encoding

Algorithm:

- 1) Start with V = {characters in corpus}
- 2) Count the most frequent adjacent character pairs in the corpus.
- 3) Merge the most frequent pair into a new *token* added to V
- 4) Repeat 2-3 until vocabulary reaches a predefined size

=> Used by GPT-3/4 models (with some small technical improvements)

DEMO

# Tokenization: wrap-up

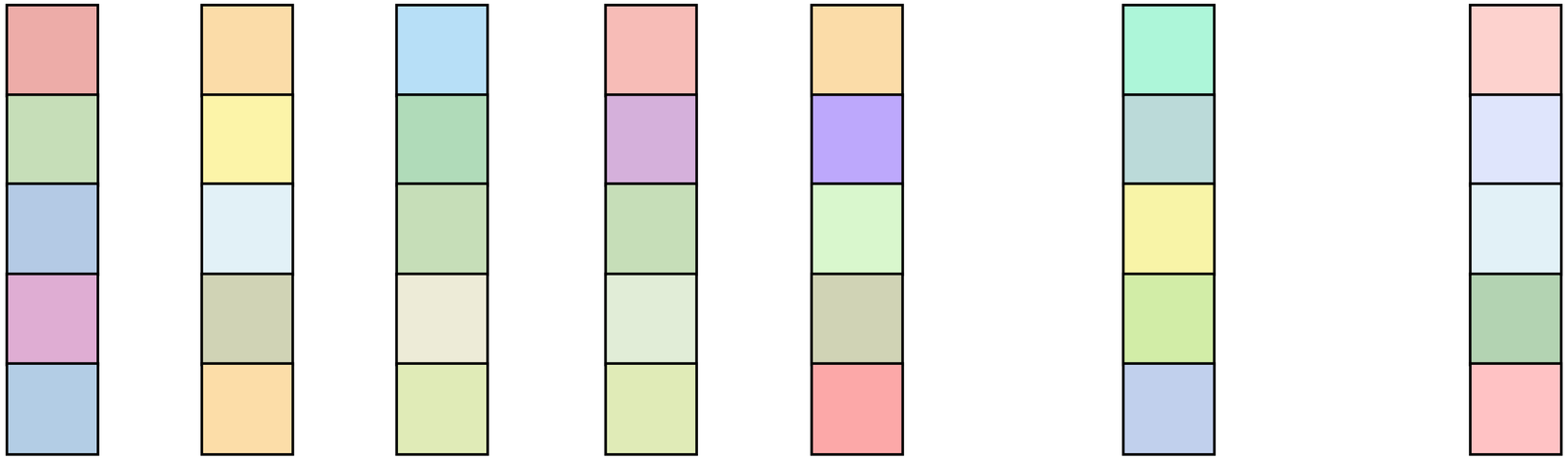
- Controllable vocabulary size
- Works well even without preprocessing of the text: lemmatization/lower-casing is not needed anymore
- Deals with unknown words (since V contains characters)
- Modern (multilingual) LLMs have about 100k tokens. 1 word ~ 4.2 tokens.

# Text embeddings

# Vector representations (embeddings)

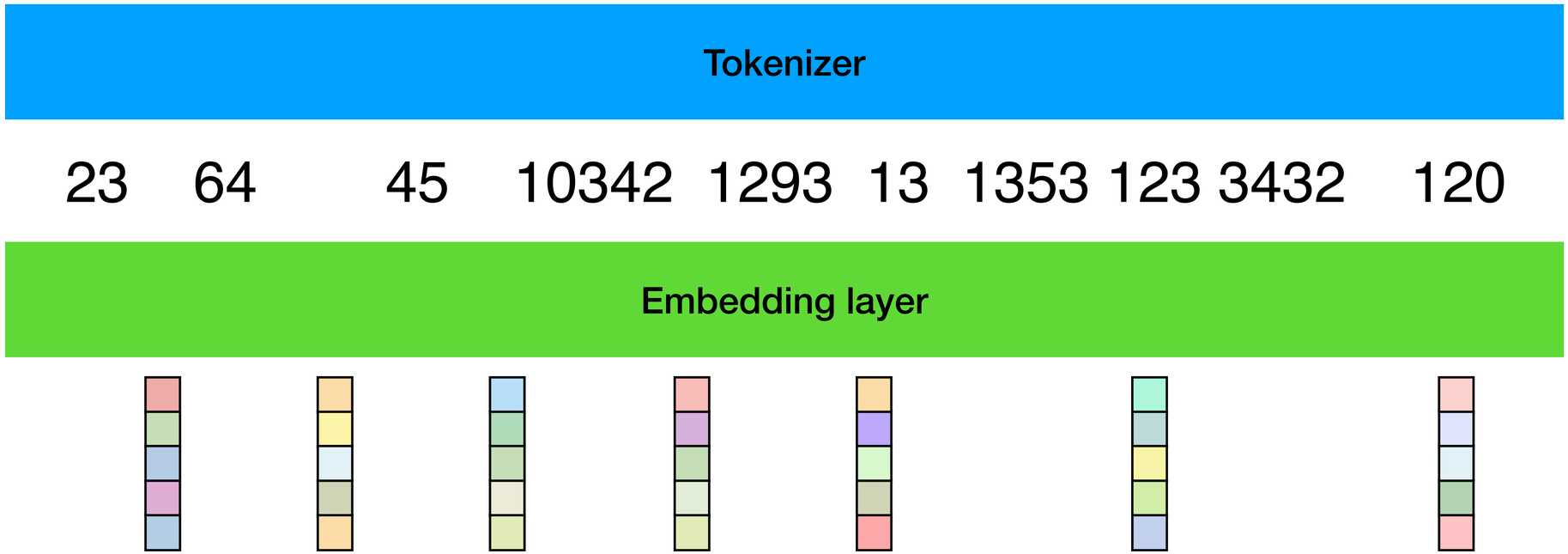
- Idea: embed each word into a real-valued vector of fixed dimensionality (e. g. 512)
- Vector “encodes” word’s meaning
- Vectors are learned automatically and may be hard to interpret

Python is a widely used programming language



# Embedding layer

The mapping token -> embedding is done by the **embedding layer. It's just a (Vocab_size, emb_dimension) matrix**



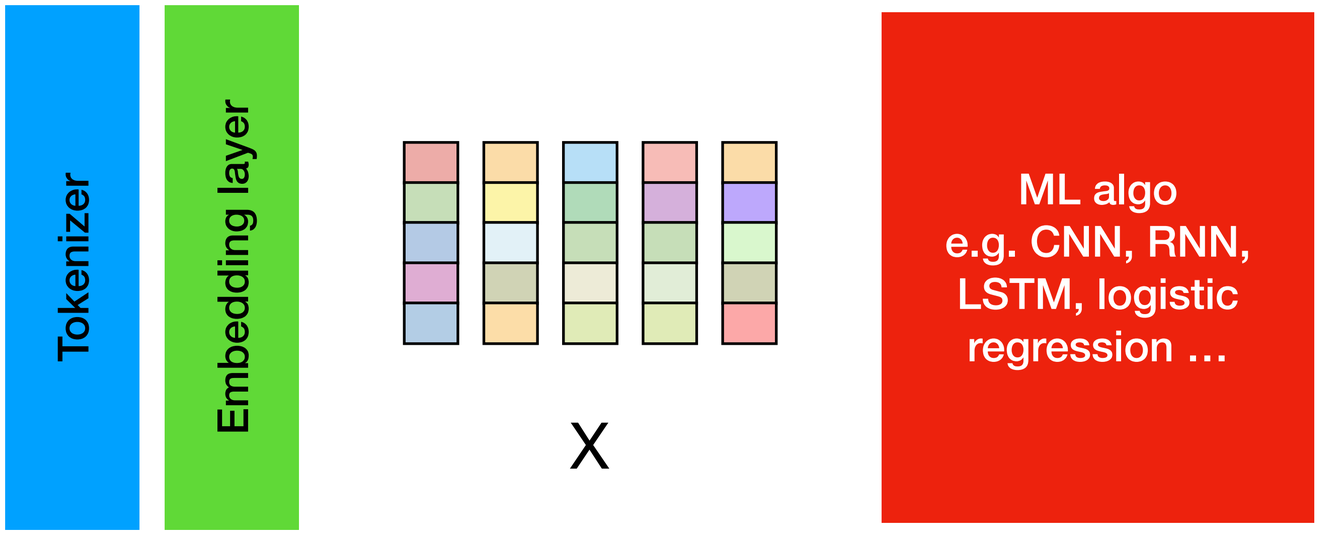
# Embedding layer: code

```python
import torch
import torch.nn as nn

# Define the vocabulary size and embedding dimension
vocab_size = 100  # Number of words in vocabulary
embedding_dim = 16  # Size of each word vector

# Create an embedding layer
embedding_layer = nn.Embedding(num_embeddings=vocab_size, embedding_dim=embedding_dim)

# Sample input: Batch of 3 sequences with word indices (each sequence has 4 tokens)
input_indices = torch.tensor([[1, 5, 8, 2], [3, 7, 9, 4], [6, 0, 2, 5]]) # shape is (3, 4)

# Get the embeddings for the input tokens
embedded_output = embedding_layer(input_indices) # shape is (3, 4, embedding_dim)
```

# Supervised learning

TEXT  Embeddings of its tokens

I hated this movie, it's the worst movie, it's really bad, really wrong actors …




Review rating:
1/5

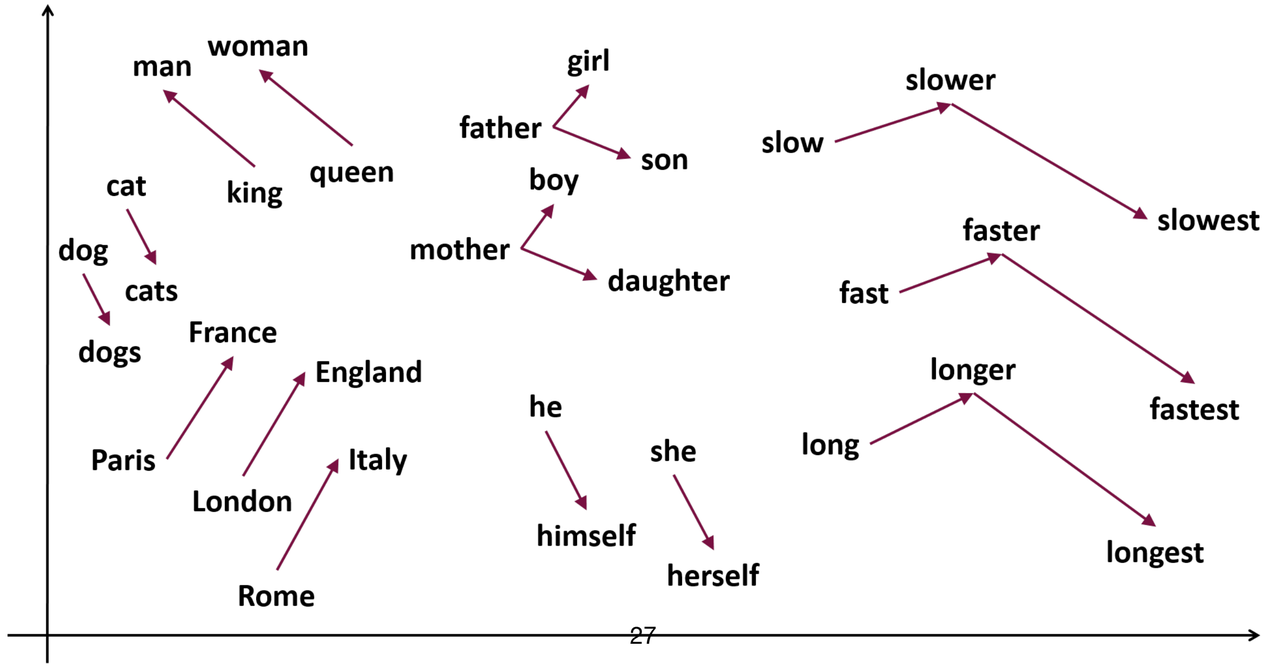
Y

Minimize loss ∑ L(Y, Y*)

We can already train classical ML algorithms *(provided we fix the length or handle the variable length somehow), given* **labels**

# Embedding

- Once an embedding layer is trained, its structure tells us a lot about the tokens !
- Using a pretrained embedding layer is a simple way to train NLP algorithms.



# Self-supervised learning: “free” labels for NLP

# Annotations for NLP ?

- Obtaining fine-grained annotations of text is in general expensive and time-consuming
- It’s easy to collect raw text from the web
  - https://huggingface.co/datasets/wikimedia/wikipedia
  - https://huggingface.co/datasets/HuggingFaceFW/fineweb

We can form supervised learning objectives from this raw text: this idea is called **self-supervised learning**

# Masked language Modeling

**Input:** I **<MASK>** my knee while ski-ing in **<MASK>** Laux.

Any Sequence to sequence ML algorithm (e.g. RNN)

**Target:** I **broke** my knee while ski-ing in **Sept** Laux.

Randomly hide input tokens and ask the model to reconstruct them

Shows emergent behaviors when scaled

Massive amount of data available.

# Causal language Modeling

**Input:** I broke my knee while ski-ing in Sept

Any Sequence-to-one ML algorithm (e.g. RNN, CNN with global pooling …)

**Target:** Laux

*An ML model trained to do this is called* ***a language model***

Here the target is the next token prediction.

Massive amount of data available.

This is at the basis of chatGPT/llama/mistral…

# Causal language Modeling

**Input:** I broke my knee while ski-ing in. Sept

Any Sequence-to-one ML algorithm (e.g. RNN, CNN with global pooling …)

**Target:** Broke my knee while ski-ing in Step Laux

In practice, **make predictions everywhere at once**:

**I -> broke**
**I broke my -> knee**
**I broke my knee -> while**
**…**

Provided the model

respects causality !

# Causal Language Modeling enables generation

- Given a language model F, and an input list of tokens (x_1, …, x_n), we can generate a continuation:
- $x_{n+1} \sim F(x_1, \ldots, x_n)$ # either select most likely, or sample along softmax distribution
- $x_{n+2} \sim F(x_1, \ldots, x_n, x_{n+1})$ # either select most likely, or sample along softmax distribution
- …

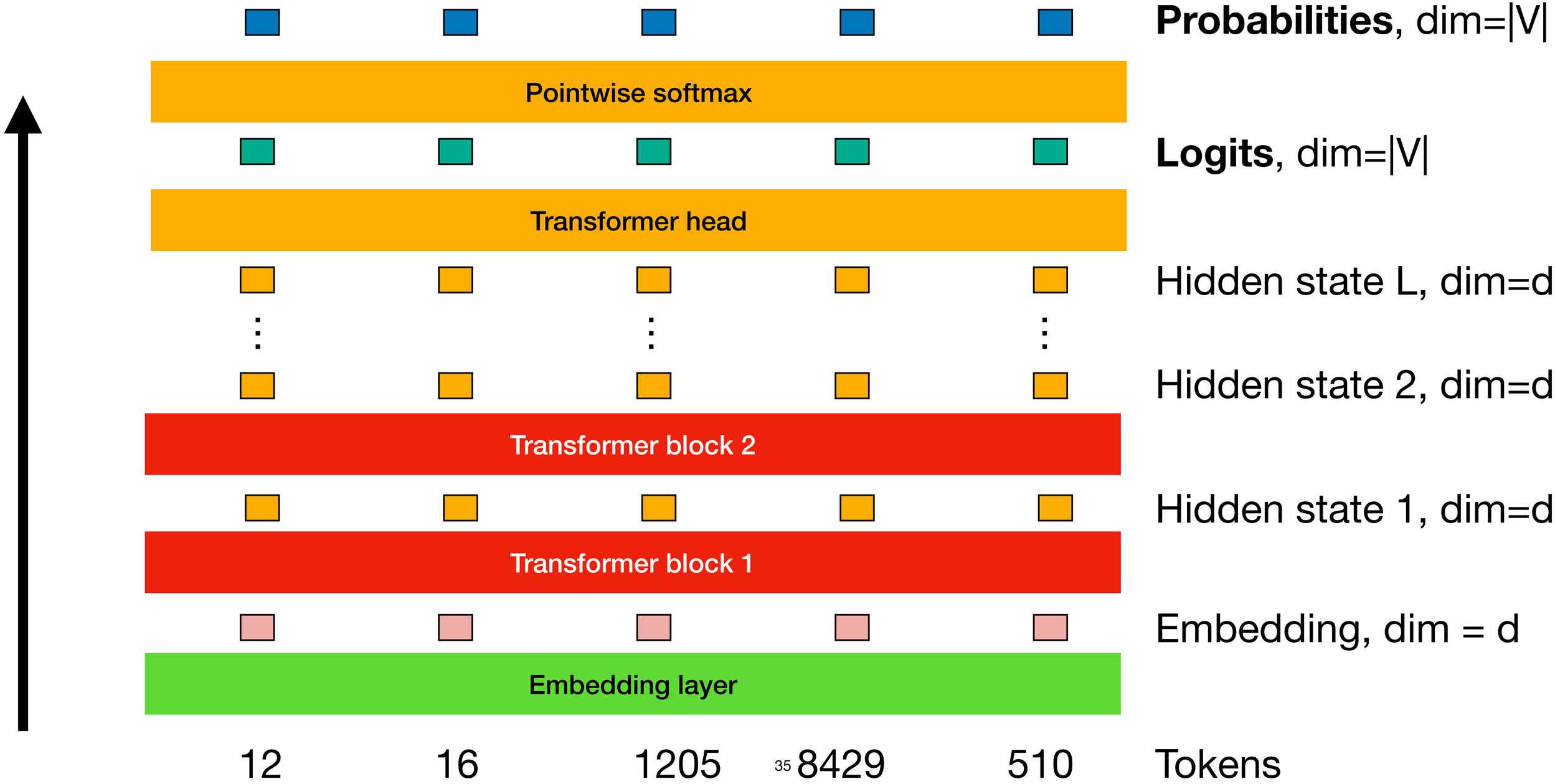
There are variations: greedy decoding, sampling, beam search …

Decoder-only

# Transformers architecture

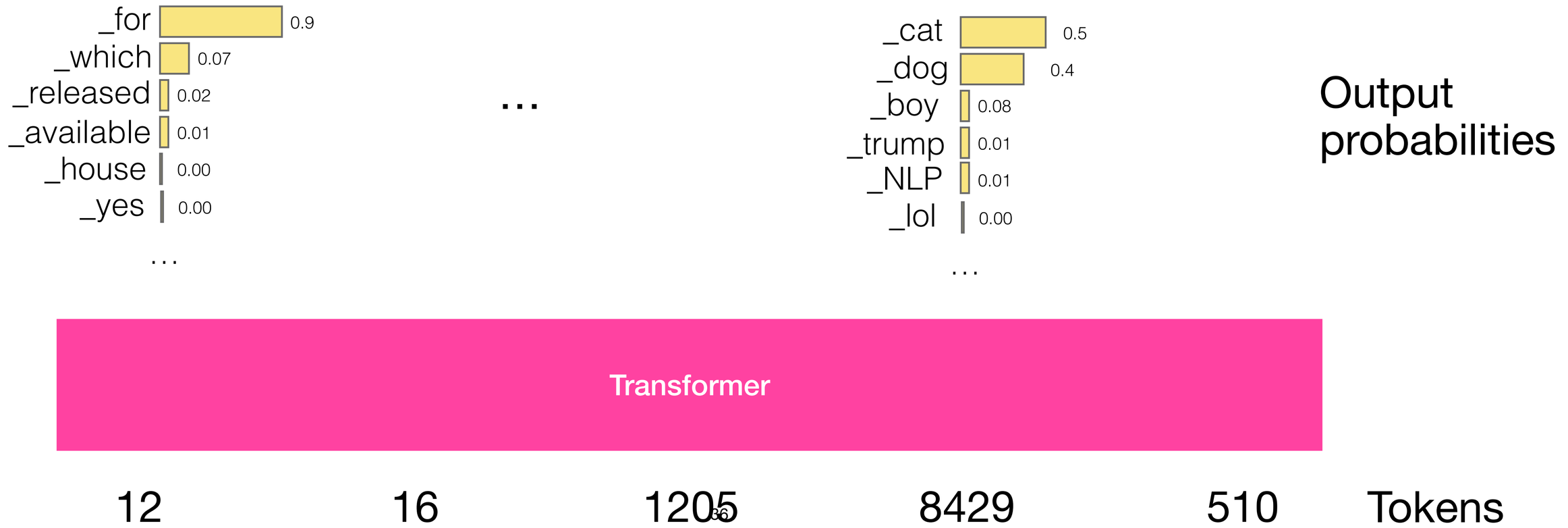
Step by step

# Transformer network from afar



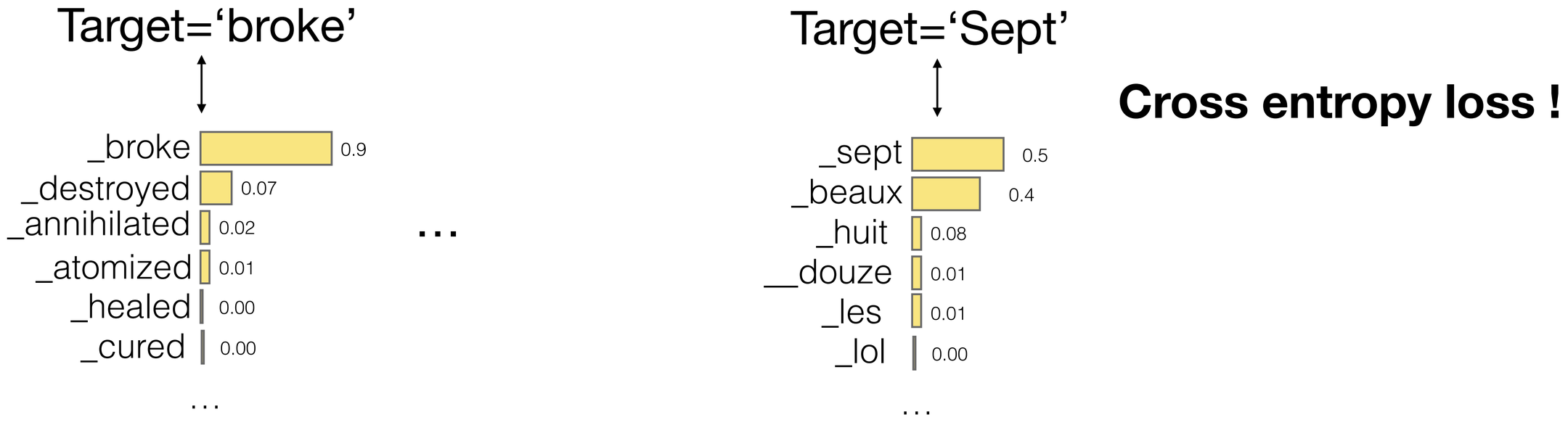
# Transformer network from afar

- **each input token is mapped to a categorical distribution over the vocabulary size**



# Training objective for transformer

## Masked Language Modeling

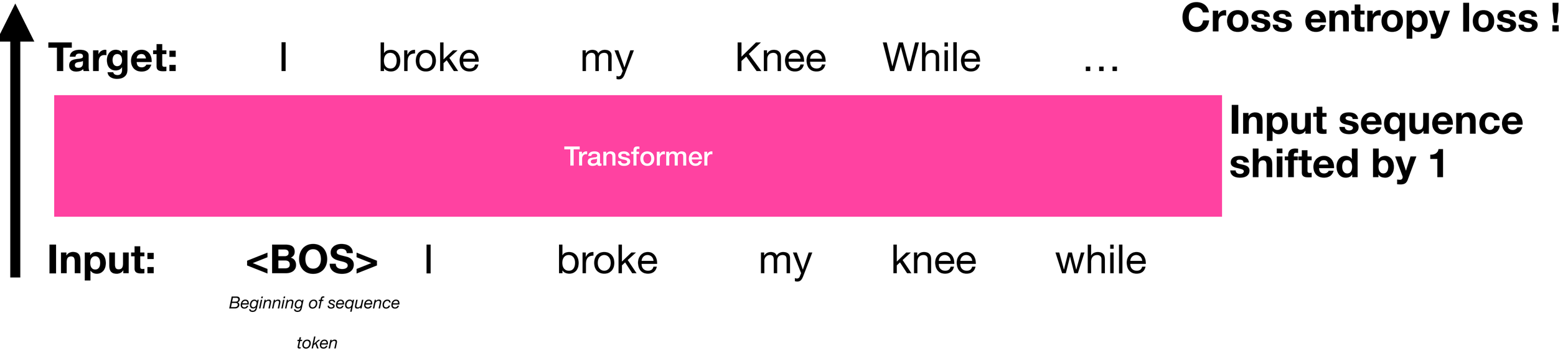



**Cross entropy loss !**

**Input:** I **<MASK>** my knee while ski-ing in **<MASK>** Laux.

# Training objective for transformer

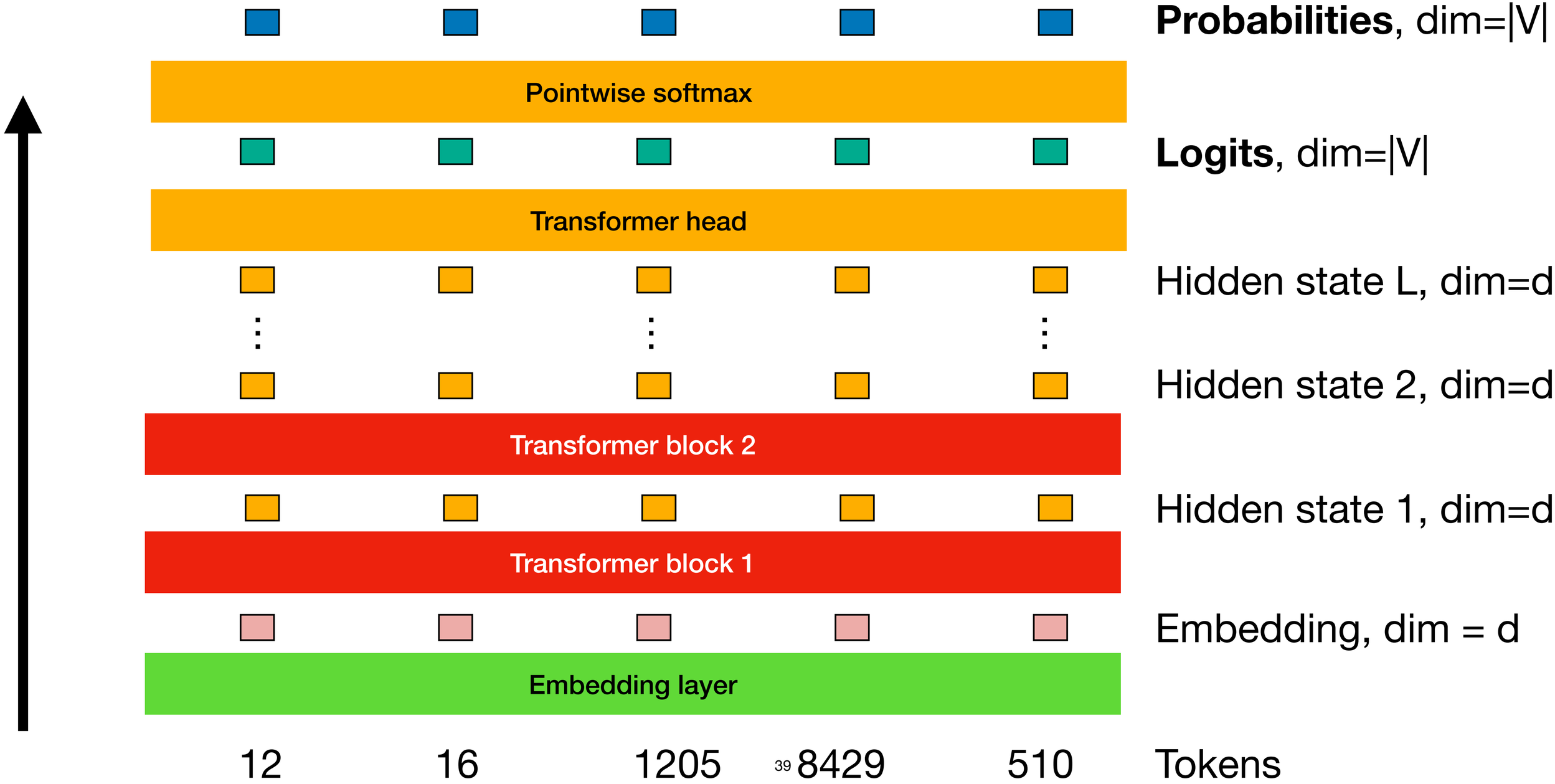
## Causal Language Modeling



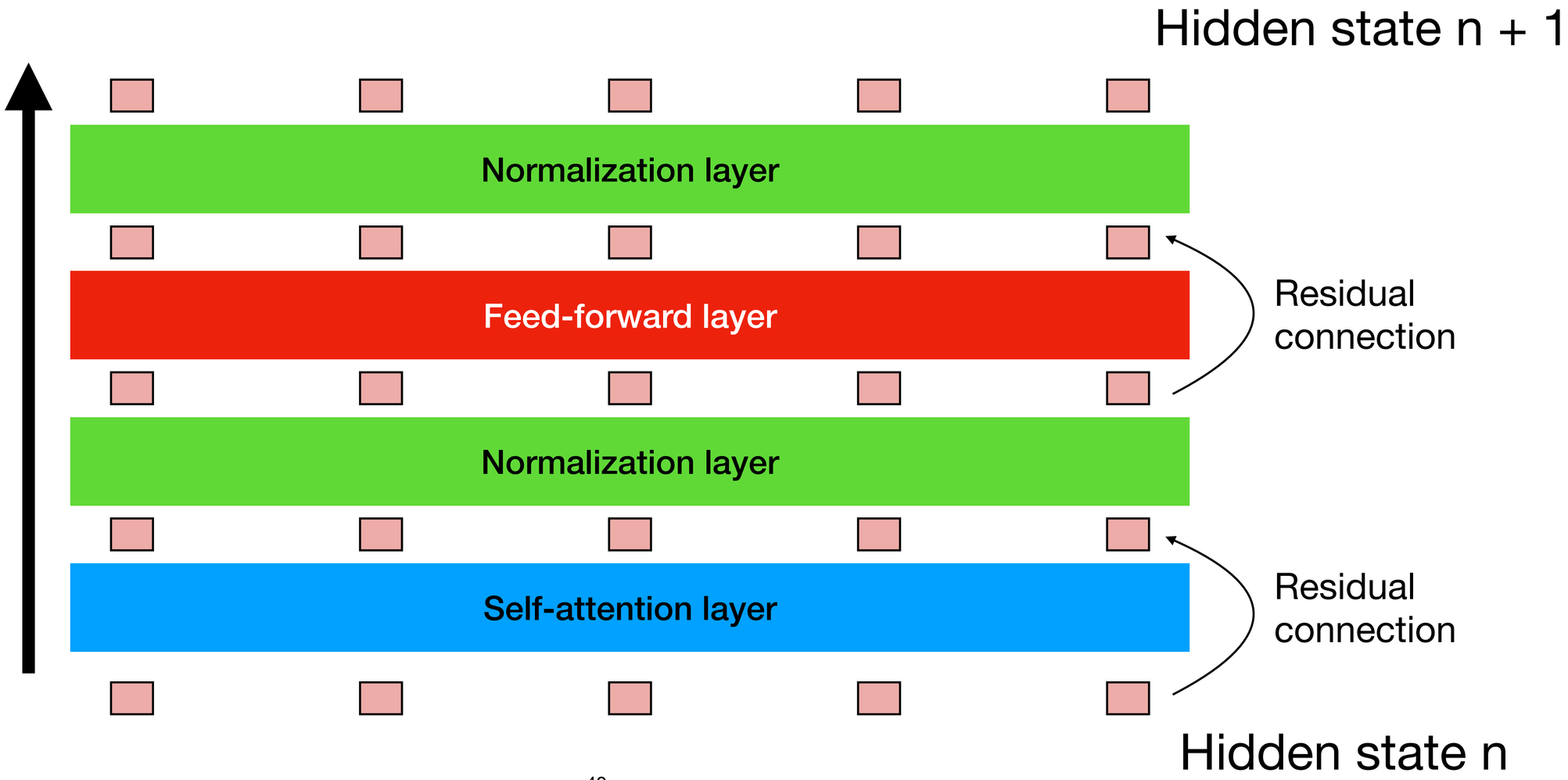

This is exactly the causal language modeling task presented before

Next token predictions are done for all tokens at once

# Transformer network from afar



# Inside a Transformer block



# Attention layer

- An attention layer takes as input a (batched) sequence of embeddings z, each of dimension $n_{\text{emb}}$
- **Step 1:** create **Keys, Queries and Values** for each entry embedding:
  - $K = W_K z$ where $W_K$ is a (d $\times$ $n_{\text{emb}}$) matrix
  - $Q = W_Q z$ where $W_Q$ is a (d $\times$ $n_{\text{emb}}$) matrix
  - $V = W_V z$ where $W_V$ is a (d $\times$ $n_{\text{emb}}$) matrix
- $W_K, W_Q, W_V$ are **learnable parameters of the layer**

# Attention layer

| | **Q_1** | **Q_2** | **Q_3** |
|---|---|---|---|
| **K_1** | K_1 • Q_1 | K_1 • Q_2 | K_1 • Q_3 |
| **K_2** | K_2 • Q_1 | K_2 • Q_2 | K_2 • Q_3 |
| **K_3** | K_3 • Q_1 | K_3 • Q_2 | K_3 • Q_3 |

- **Step 2:**
  - Compute all pairs of scalar products between keys and queries
  - **K_i • Q_i measure how much a token attends to another token**

# Attention layer

| | **Q_1** | **Q_2** | **Q_3** |
|---|---|---|---|
| **K_1** | K_1 • Q_1 | K_1 • Q_2 | K_1 • Q_3 |
| **K_2** | K_2 • Q_1 | K_2 • Q_2 | K_2 • Q_3 |
| **K_3** | K_3 • Q_1 | K_3 • Q_2 | K_3 • Q_3 |

- **Step 2:**
  - Compute all pairs of scalar products between keys and queries
  - **K_i • Q_i measure how much a token attends to another token**

# Attention layer

| | **Q_1** | **Q_2** | **Q_3** |
|---|---|---|---|
| **K_1** | 50 | -1 | 3 |
| **K_2** | 3.22 | 0.99 | 6 |
| **K_3** | -0.39 | 22 | 43 |

- **Step 2:**
  - Compute all pairs of scalar products between keys and queries
  - **K_i · Q_i measure how much a token attends to another token**

# Attention layer

| | **Q_1** | **Q_2** | **Q_3** |
|---|---|---|---|
| **K_1** | 50 | MASK | MASK |
| **K_2** | 3.22 | 0.99 | MASK |
| **K_3** | -0.39 | 22 | 43 |

- **Step 3:**
  - If we do CAUSAL language modeling: we need past tokens to NOT attend future tokens

# Attention layer

| | **Q_1** | **Q_2** | **Q_3** |
|---|---|---|---|
| **K_1** | 1 | MASK | MASK |
| **K_2** | 0.8 | 0.2 | MASK |
| **K_3** | 0.01 | 0.6 | 0.39 |

- Step 4:
  - Normalize these scores using softmax (row-wise)
  - **Technicality:** scores are divided by $\sqrt{d}$ before softmax to prevent too large logits.

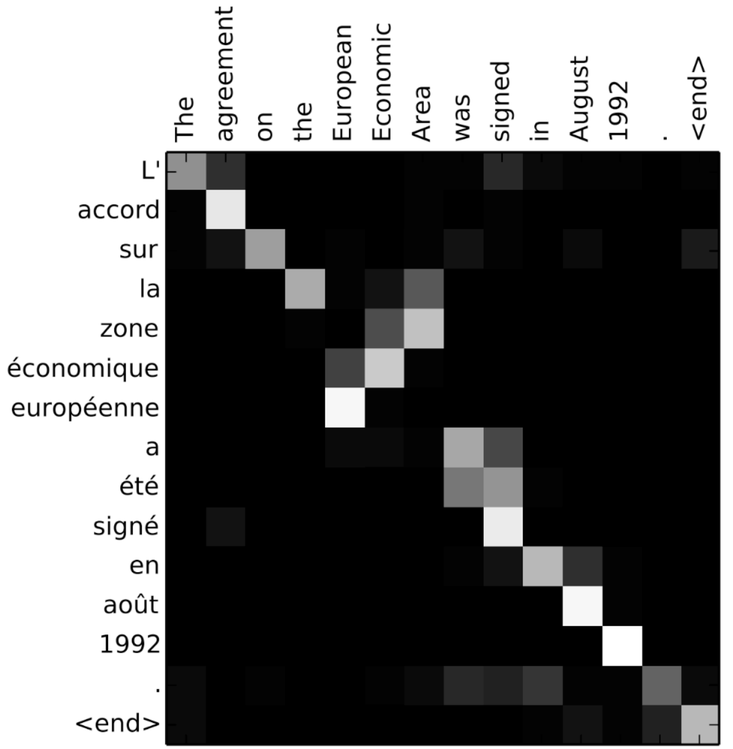
# Attention layer

- **Step 5:**
  - Form the output by averaging the values of each other token, weighted by the attention score:
  - e.g. for token 1:

$$\sum_{i=1}^{l} \text{softmax}\left(\frac{k_1^\top q_i}{\sqrt{d}}\right) v_i$$

# An example of attention map

- In practice, the attention layer makes connections between tokens.
- It **contextualizes** each embedding




Attention-matrix heatmap

*Bahdanau, et al. 2015. Neural machine translation by jointly learning to align and translate. In Proc. ICLR.*

# The attention layer: summary

**Layer output**

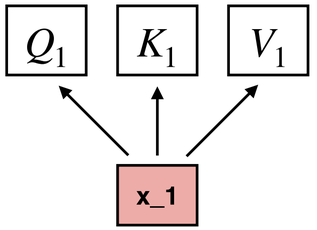
$$\sum_{i=1}^{l} \text{softmax}\left(\frac{k_1^\top q_i}{\sqrt{d}}\right) v_i \qquad \sum_{i=1}^{l} \text{softmax}\left(\frac{k_1^\top q_i}{\sqrt{d}}\right) v_i$$

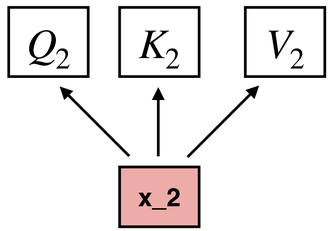
**Normalized Attention scores**

$$\text{softmax}\left(\frac{k_1^\top q_1}{\sqrt{d}}, \ldots, \frac{k_1^\top q_n}{\sqrt{d}}\right) \quad \text{softmax}\left(\frac{k_2^\top q_1}{\sqrt{d}}, \ldots, \frac{k_2^\top q_n}{\sqrt{d}}\right)$$

**Attention scores**

$$\left(\frac{k_1^\top q_1}{\sqrt{d}}, \ldots, \frac{k_1^\top q_n}{\sqrt{d}}\right) \qquad \left(\frac{k_2^\top q_1}{\sqrt{d}}, \ldots, \frac{k_2^\top q_n}{\sqrt{d}}\right)$$




$$Q_i = W_Q x_i$$
$$K_i = W_K x_i$$
$$V_i = W_V x_i$$

$W_Q, W_K, W_V$ **are the parameters of the layer**

**How does this scale with the number of input tokens ?**

# The attention layer

$$A = \textbf{softmax}\left(\frac{Q^\top K}{\sqrt{d}}\right)V$$

- Captures long-range dependencies
- Highly parallelizable
- Context-aware representation
- BUT quadratic in number of tokens
- Memory-intensive

...

# Multi-head attention

- In practice, multiple attention are computed in parallel and gathered
- For example, assuming a hidden size of 512 we can:
  - Have 4 attention heads, each producing new values in dimension 128
  - Concatenate each of these into a 512 vector
  - Apply a final linear layer (the ‘up’-projection in LLM lingo) to obtain the final output of the attention layer
- It’s an important part of transformer success
- https://bbycroft.net/llm

# TP1

- https://github.com/maxime-louis/transformers-intro/blob/main/TP1.ipynb

**Course adapted from:**

- Nadia Chirkova and Laurent Besacier NLP Course at Centrale Paris

(Huge thanks to Nadia)

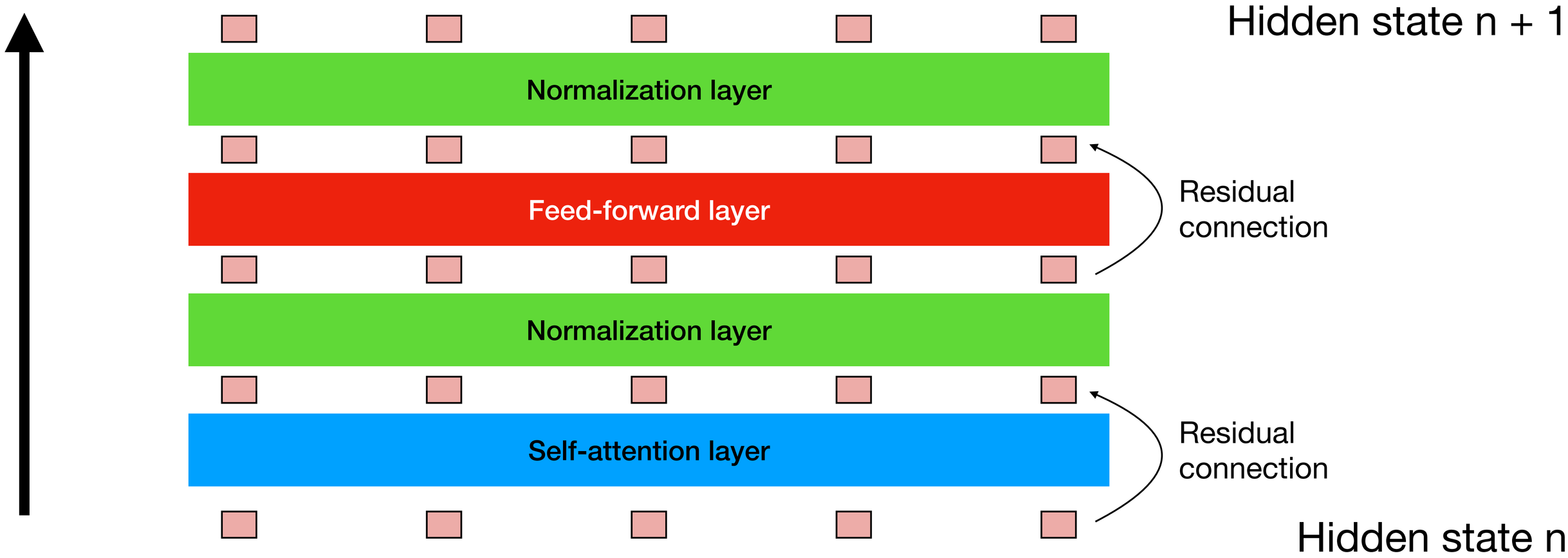
- Stanford transformers course

Other great resources:

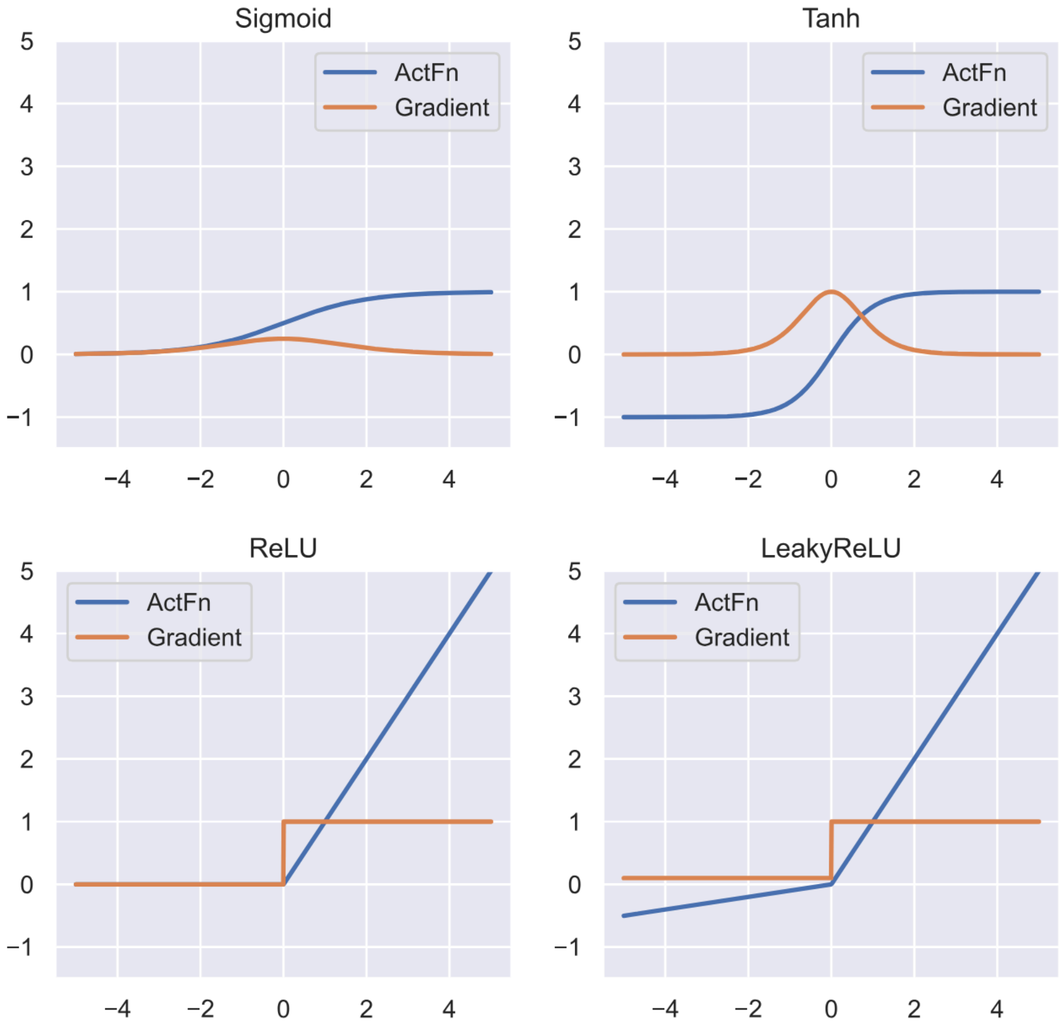
https://www.youtube.com/watch?v=wjZofJX0v4M&t (2Blue1Brown)

https://www.youtube.com/watch?v=kCc8FmEb1nY&t (openAI employee)

# Inside a Transformer block



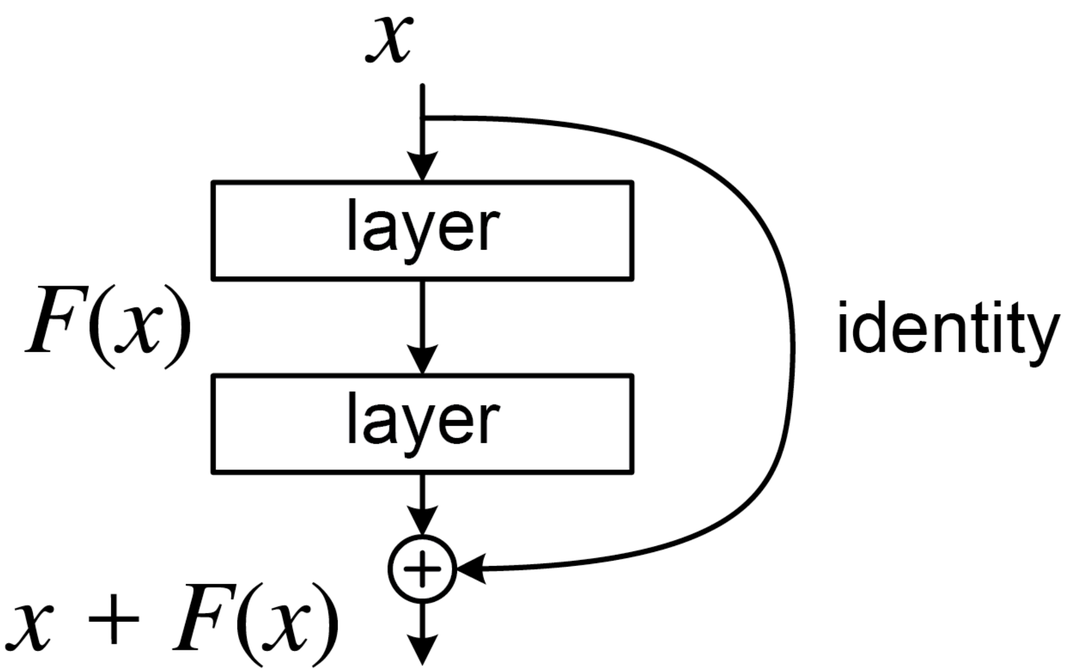
# The problem of vanishing gradients




- Iterative back propagation with DL activations leads to vanishing gradients
- Learning **very deep** neural networks with iterative layer is challenging
- One solution: residual connection

# Residual connection

## Avoiding vanishing gradients




- Add output of N-th layer to output of N+1-th layer
- Gradient computation involves the layer as before + the gradient through the identity: **no vanishing**
- BUT Different layers activations must match in dimension !
- **All large neural networks have residual connections.**

# Feed-forward layer

## Two fully connected layers

$$\text{MLP}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

- Acts token-wise

- Dropout is not always used (not in the latest LLMs)

```python
class TransformerMLP(nn.Module):
    def __init__(self, embed_dim, expansion_factor=4, dropout=0.1):
        super(TransformerMLP, self).__init__()

        hidden_dim = embed_dim * expansion_factor  # Expansion in FFN
        self.fc1 = nn.Linear(embed_dim, hidden_dim)  # First linear layer
        self.act = nn.ReLU()  # Activation function (GELU is also common)
        self.fc2 = nn.Linear(hidden_dim, embed_dim)  # Second linear layer
        self.dropout = nn.Dropout(dropout)  # Dropout for regularization

    def forward(self, x):
        x = self.fc1(x)  # First linear transformation
        x = self.act(x)  # Activation function
        x = self.dropout(x)  # Dropout
        x = self.fc2(x)  # Second linear transformation
        x = self.dropout(x)  # Dropout again
        return x
```

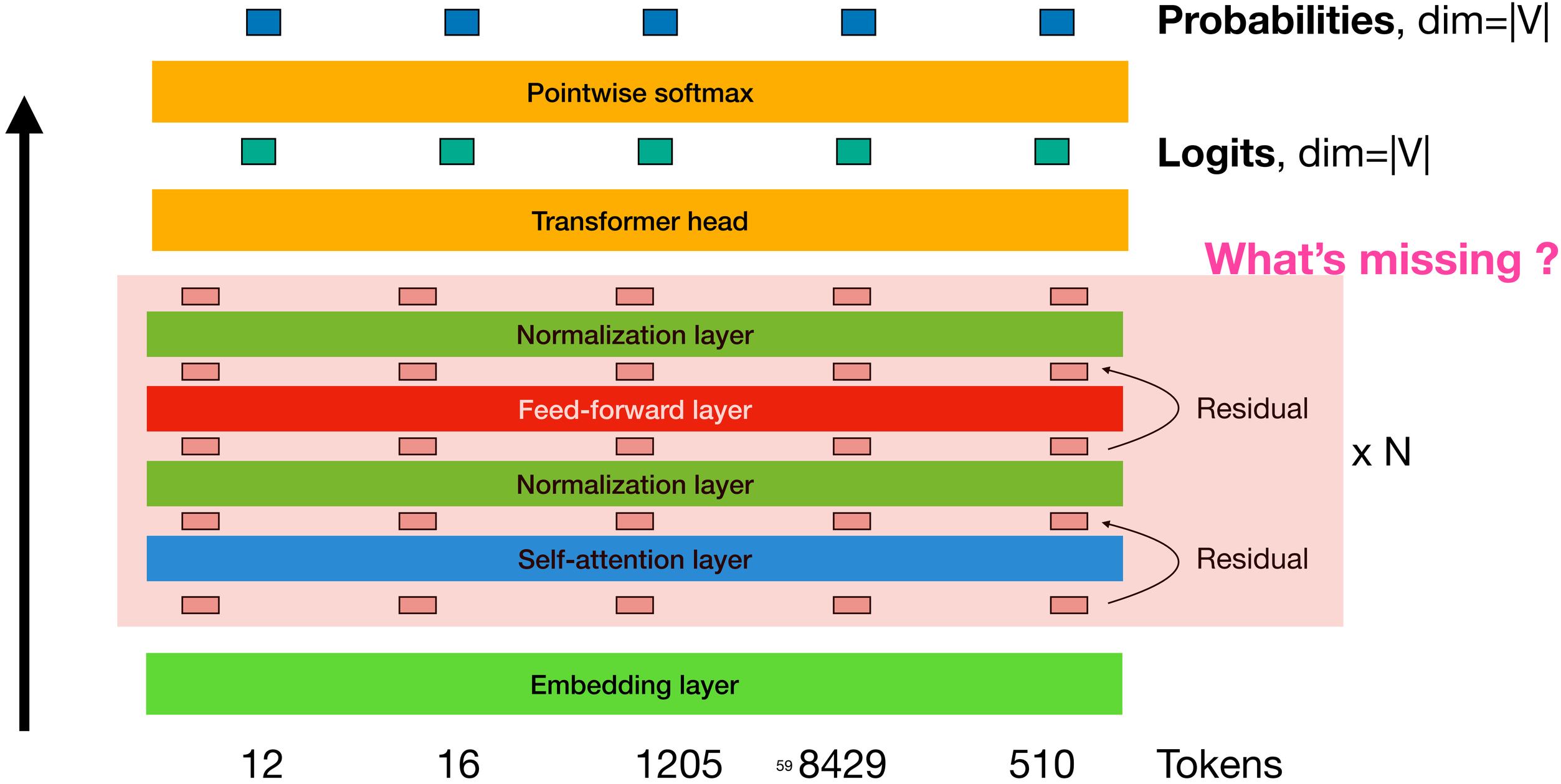
# Normalization layer

- **Internal covariate shift** causes optimization instabilities for large deep neural networks (variance of each layer input activation scales across batches through optimization)
- Therefore, a normalization of the activations (i.e. hidden states for a transformers) improves the learning procedure.
- Transformers typically use Layer Normalization:

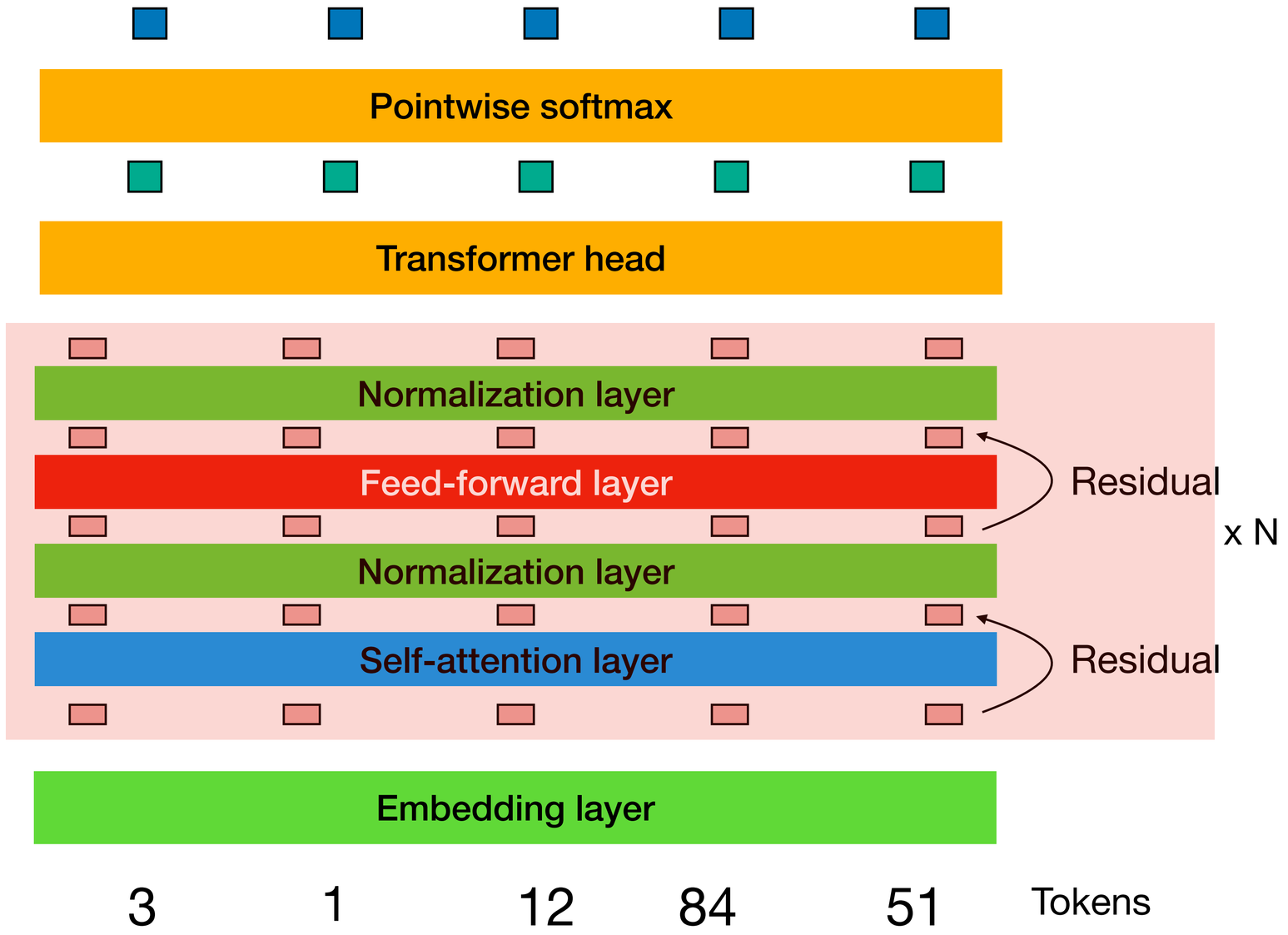
$$\hat{x}_i = \frac{x_i - \mu_L}{\sigma_L + \epsilon}$$

Where μ, σ are computed across the feature dimension

# Transformer network (current state)



# Transformer network (current state)

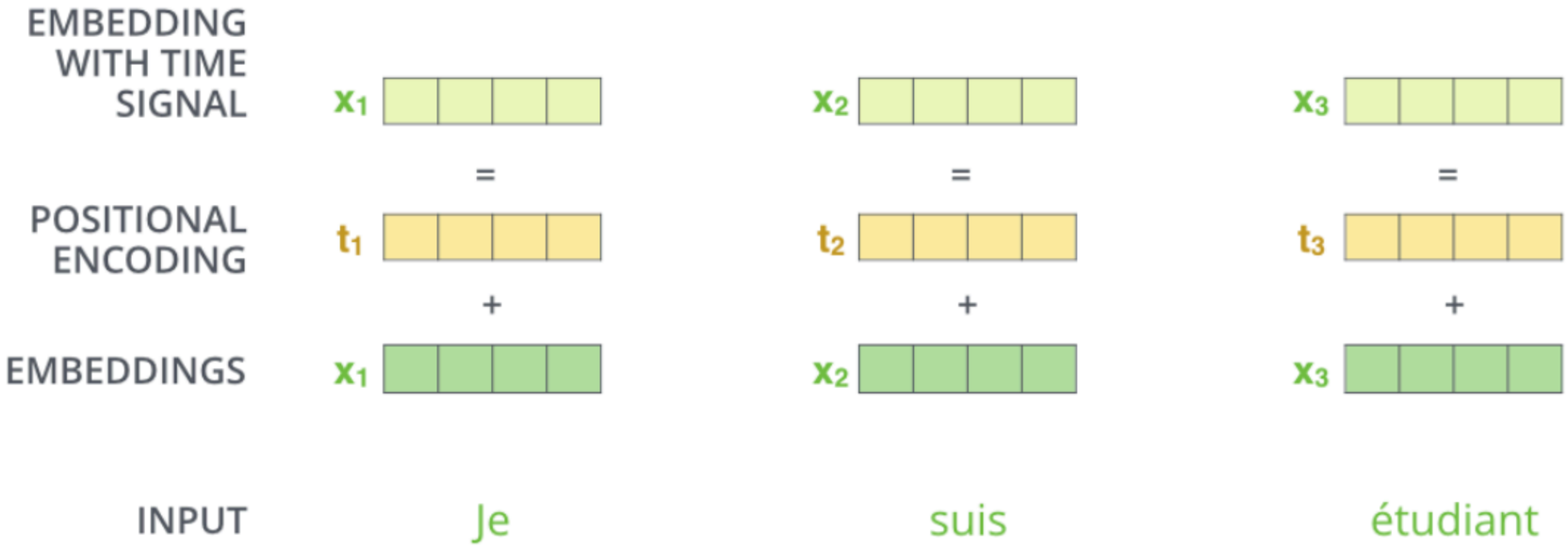


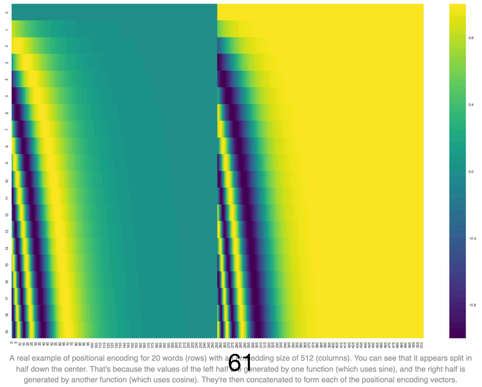

All operations are permutation-invariant: Information about the tokens position is lost !

There are multiple solutions. We look at:

- positional embeddings
- Rotary Positional embeddings

# Encoding position: positional embeddings



A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in half down the center. That's because the values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). They're then concatenated to form each of the positional encoding vectors.

Add a position-dependent Embedding
1 embedding=1 row
(Here in dim 512)

# Encoding position: RoPE

When computing the attention score, apply a rotation matrix:

$$Q_i^\top K_j \rightarrow Q^\top R_{\theta_i - \theta_j} K_j$$

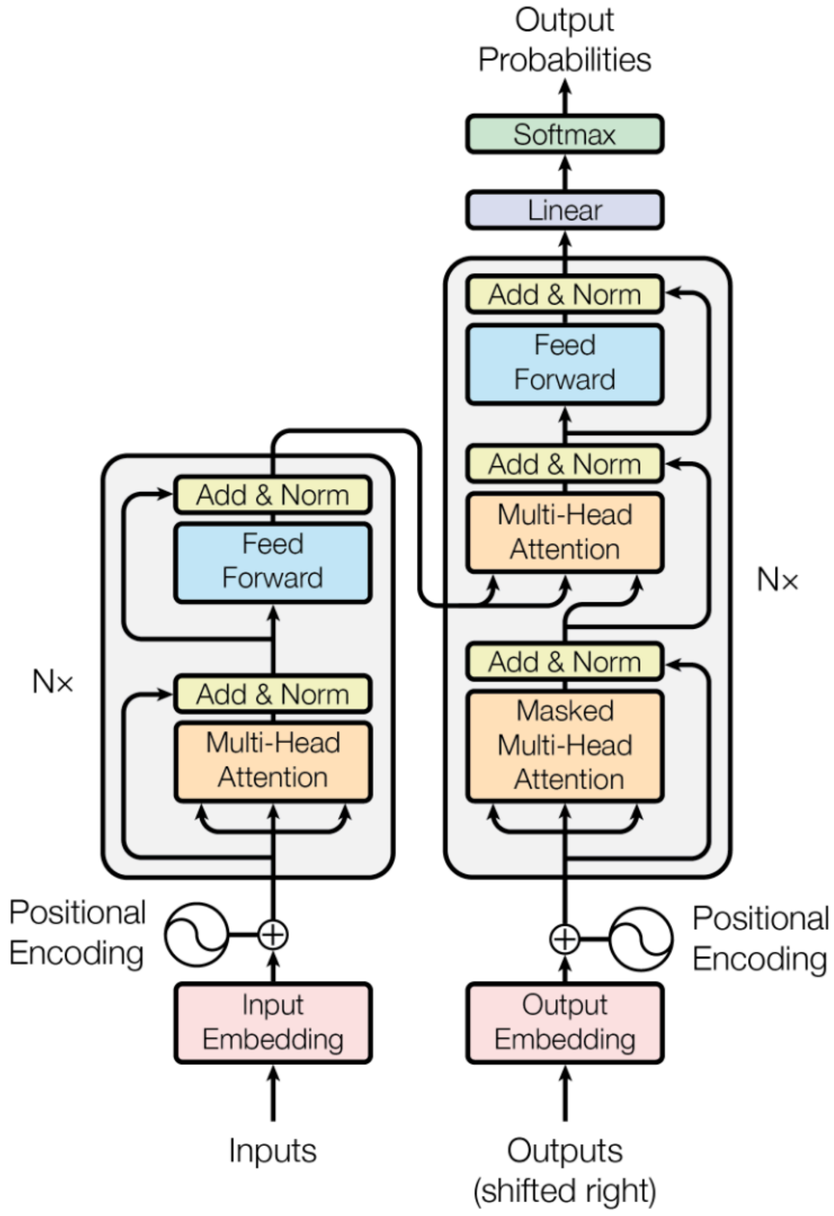
Whose angle depend on the distance between token i and j.

See https://arxiv.org/pdf/2104.09864 for details about theta and R.

- No additional parameter
- Shift invariant interaction between tokens
- Looser constraints on context size

# Encoder-Decoder transformer

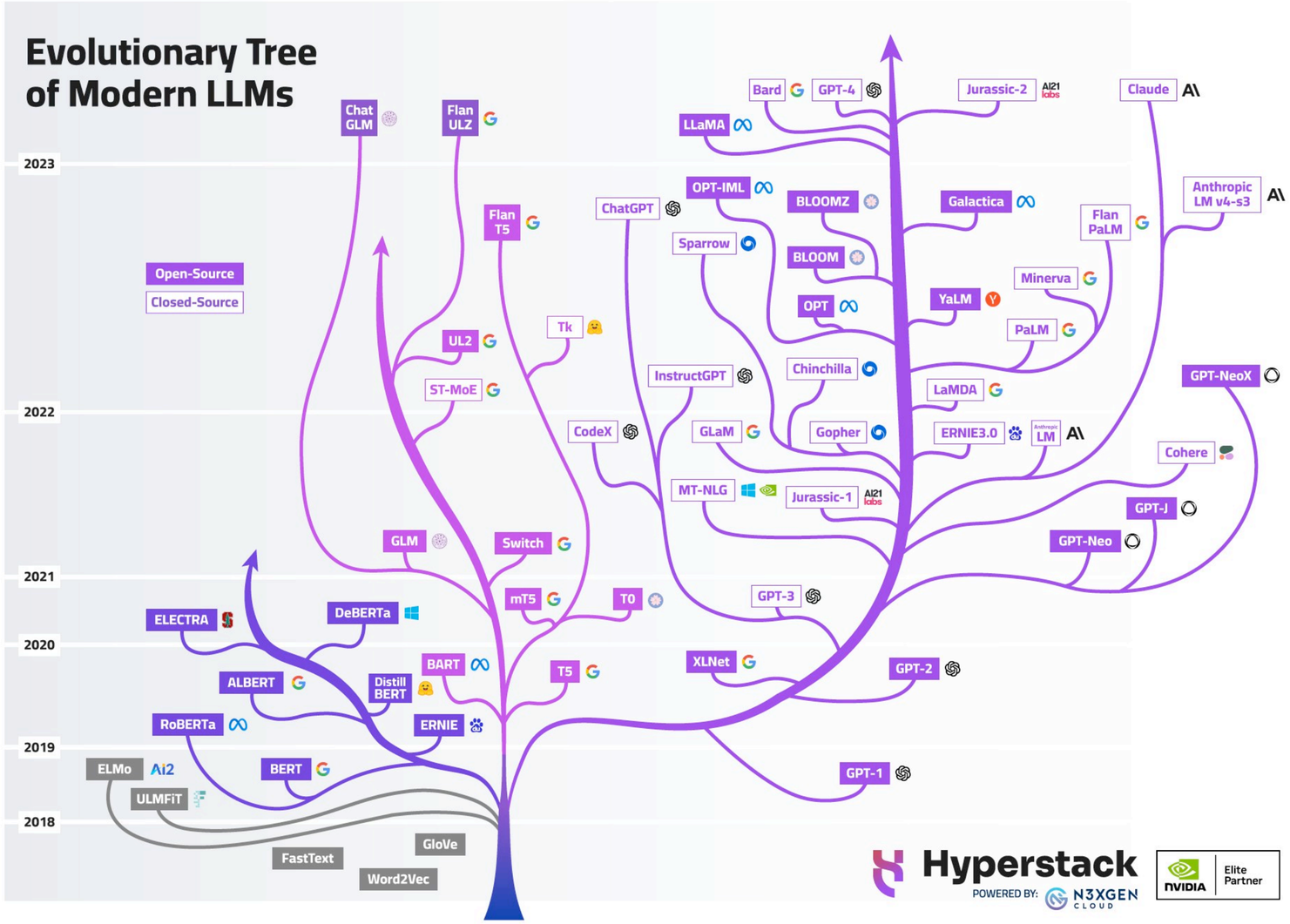
- Originally intended for sequence-to-sequence tasks
- Decoder only is easier to train in general



# Positional encoding

# Causal Masking

# Summary




All current LLMs are in between

-1B (for on-premise)

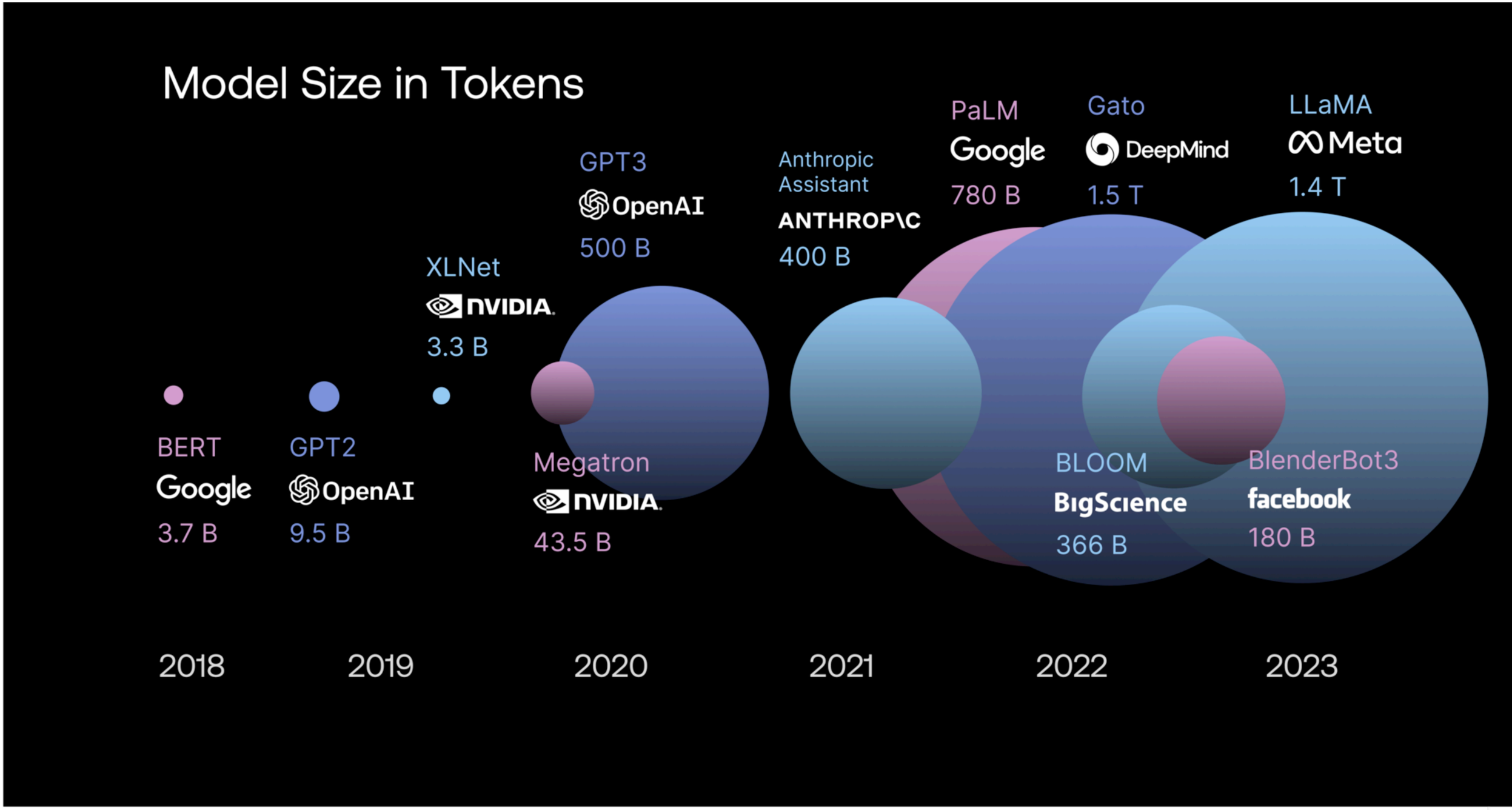
-1T (for optimal quality)

~175B parameters

~1B parameters

~110M parameters

~10M parameters


Model Size in Tokens
BERT
Google
3.7 B
GPT2
OpenAI
9.5 B
XLNet
NVIDIA.
3.3 B
GPT3
OpenAI
500 B
Megatron
NVIDIA.
43.5 B
Anthropic
Assistant
ANTHROP\C
400 B
PaLM
Google
780 B
Gato
DeepMind
1.5 T
LLaMA
Meta
1.4 T
BLOOM
BigScience
366 B
BlenderBot3
facebook
180 B
2018
2019
2020
2021
2022
2023

# LLM ecosystem

- LLMs are components of a large commercial market

Microsoft + OpenAI
Google + DeepMind
TESLA + xAI
Meta + Meta AI
amazon + ANTHROP\C

- There are excellent open-source LLMs and tools


Hugging Face