

# Attention and Transformer



**Dr. Xiao Wang**  
**J.O. Berger and M.E. Bock Professor of Statistics**  
**Purdue University**  
**URL: <https://www.stat.purdue.edu/~wangxiao/>**

# Content

**1 Attention Mechanisms**

**2 Positional Encoding**

**3 Transformer Architecture**

**4 Transformer Applications**

# Content

**1 Attention Mechanisms**

2 Positional Encoding

3 Transformer Architecture

4 Transformer Applications

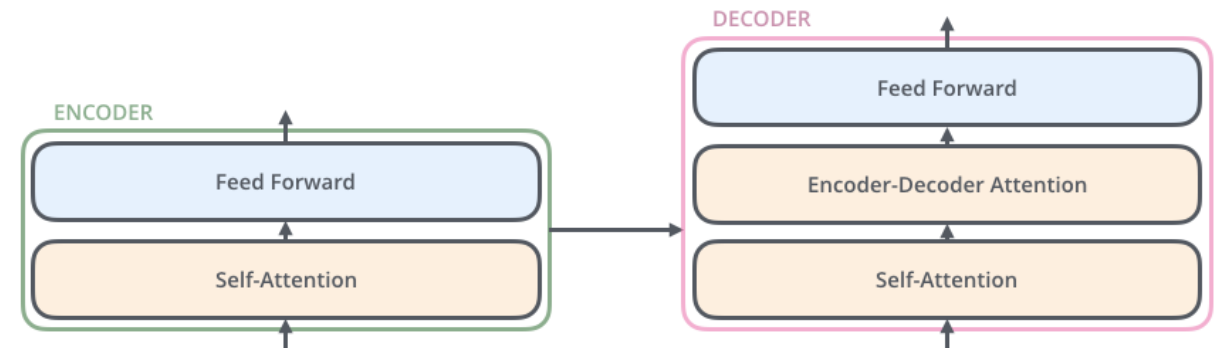
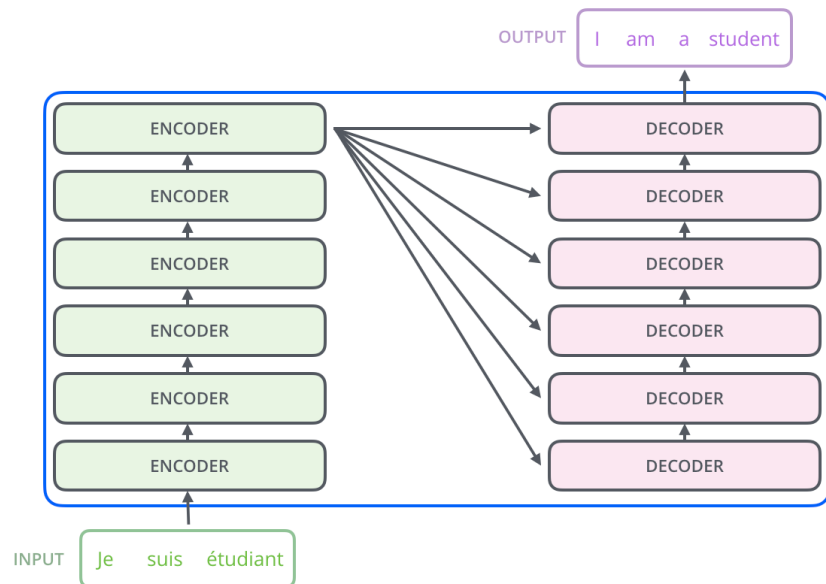
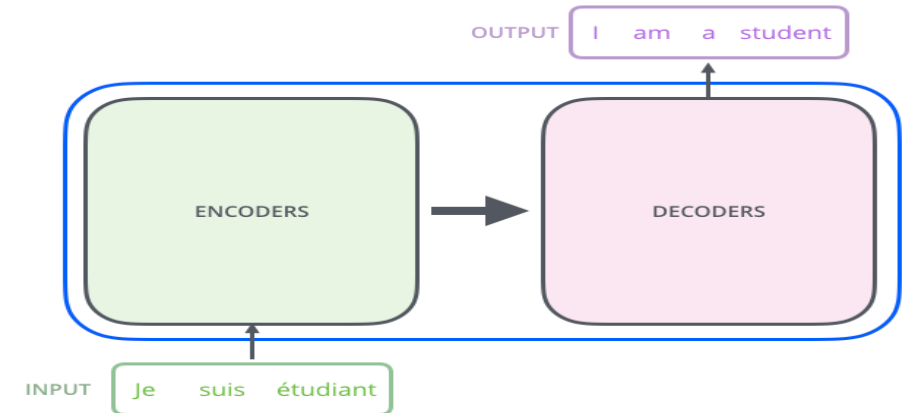
# Motivation

The dog chased another dog

The animal didn't cross the street because it was too tired

- How to encode the text?
- “dog” refers to two different entities in the first sentence
- What does “it” refer to in the second sentence?
- Is there any other way to model dependence?

# A High-Level Look



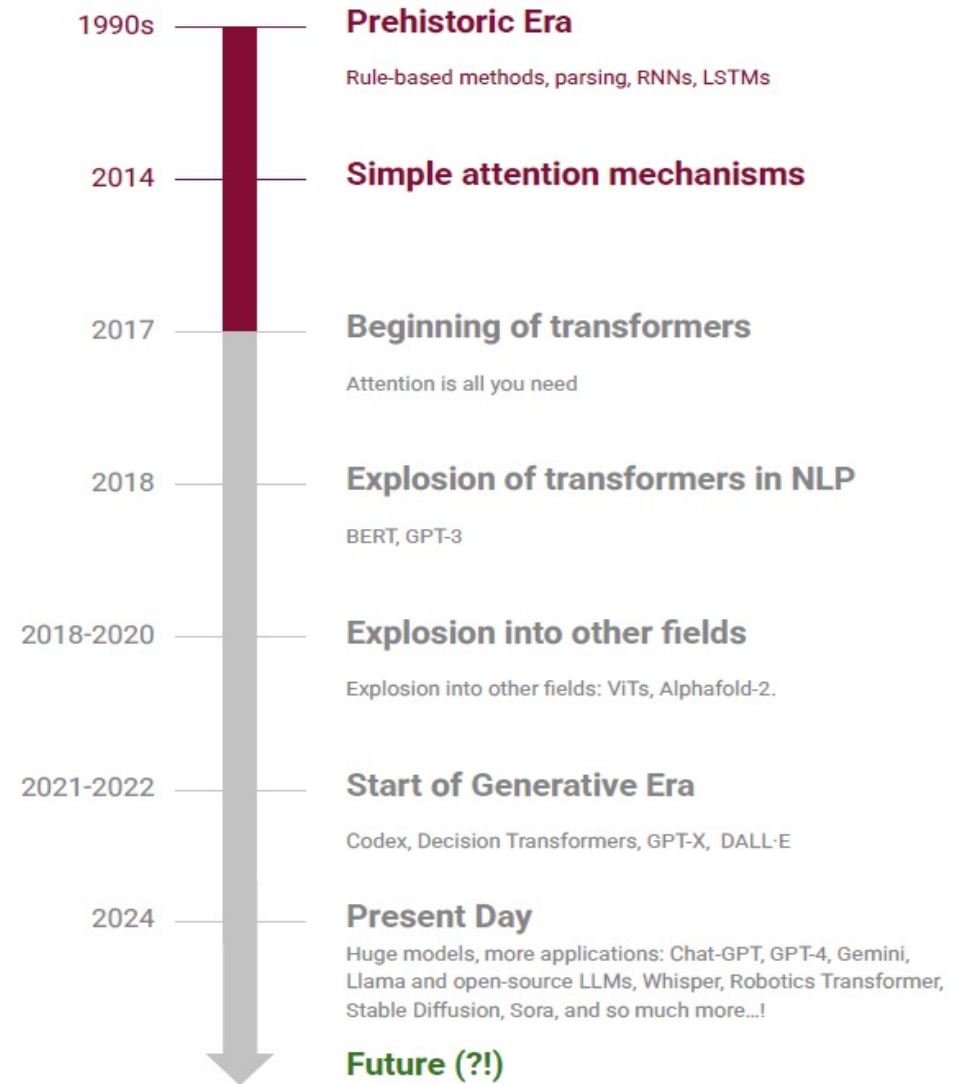
[The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. \(jalammar.github.io\)](https://jalammar.github.io)

# Attention Timeline

This timeline highlights the evolution of deep learning and natural language processing. Starting in the 1990s with rule-based methods, RNNs, and LSTMs, it progressed to the introduction of simple attention mechanisms in 2014.

The Transformer era began in 2017 with the groundbreaking paper "Attention is All You Need," leading to a rapid adoption in NLP with models like BERT and GPT by 2018. From 2018 to 2020, Transformers expanded into fields like vision (ViTs) and protein folding (AlphaFold-2). The generative era began in 2021-2022 with models like Codex, GPT-X, and DALL-E.

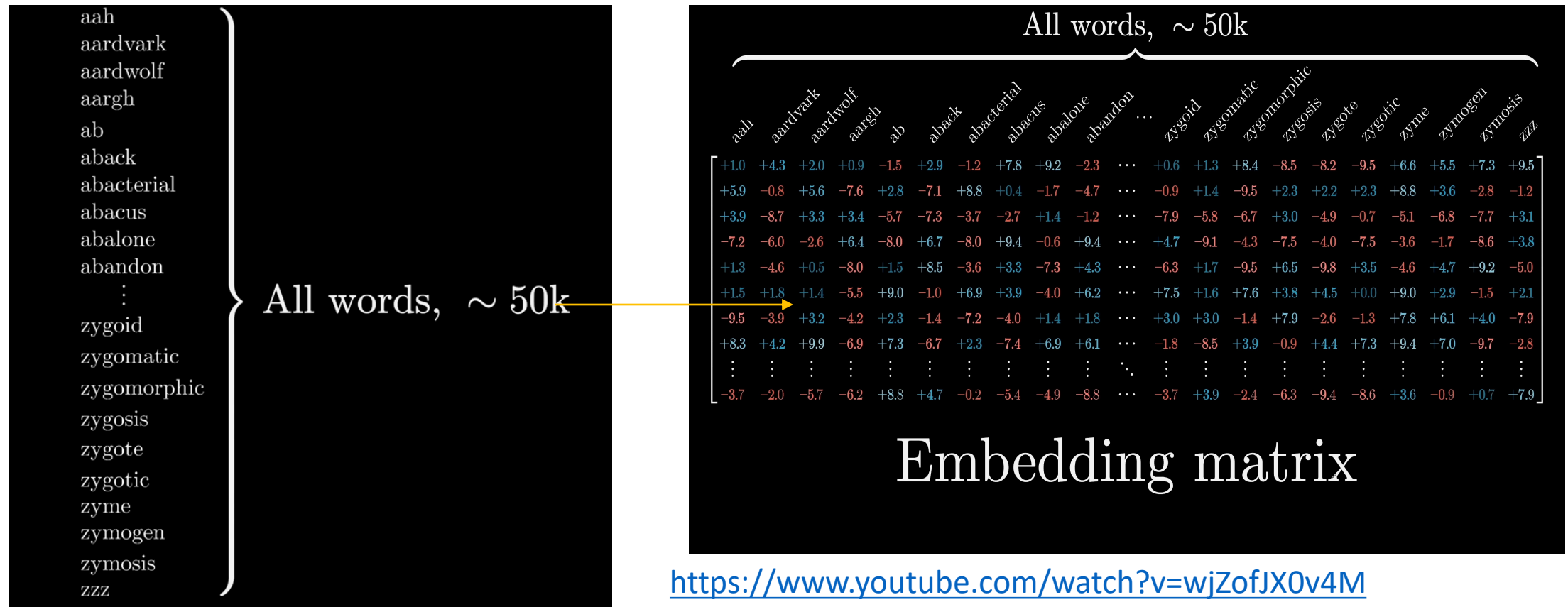
By spring 2025, Transformers power massive models like ChatGPT, DeepSeek and open new applications in diverse areas, with exciting prospects for the future.



(Yang & Hashimoto, 2025)

# Word Embeddings in NLP

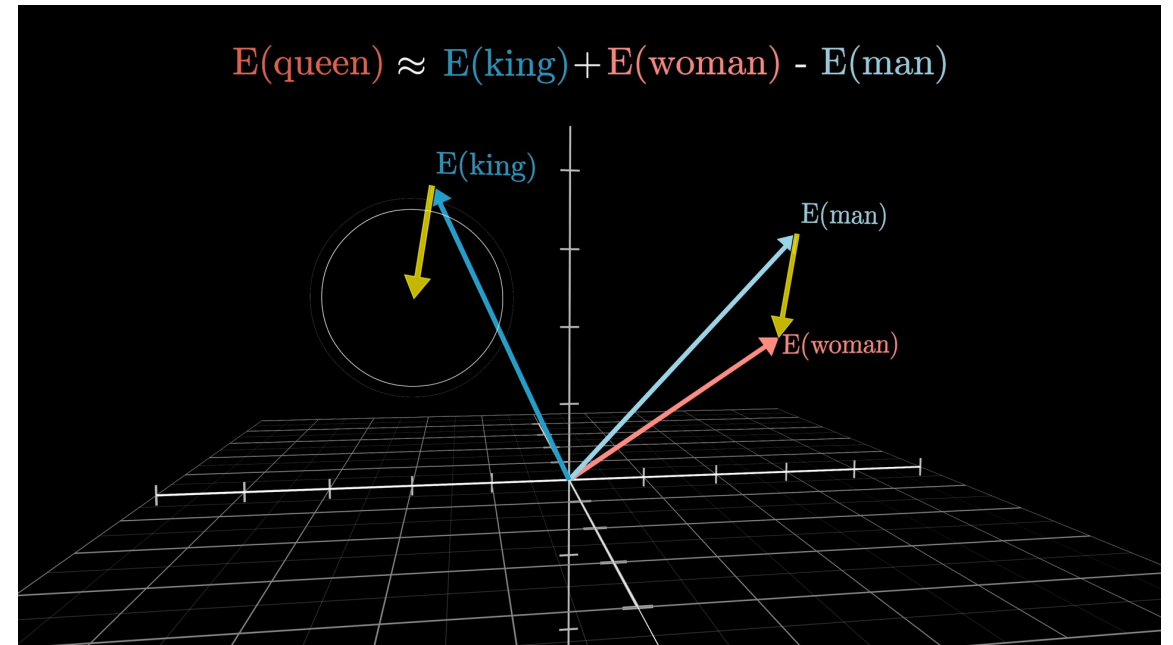
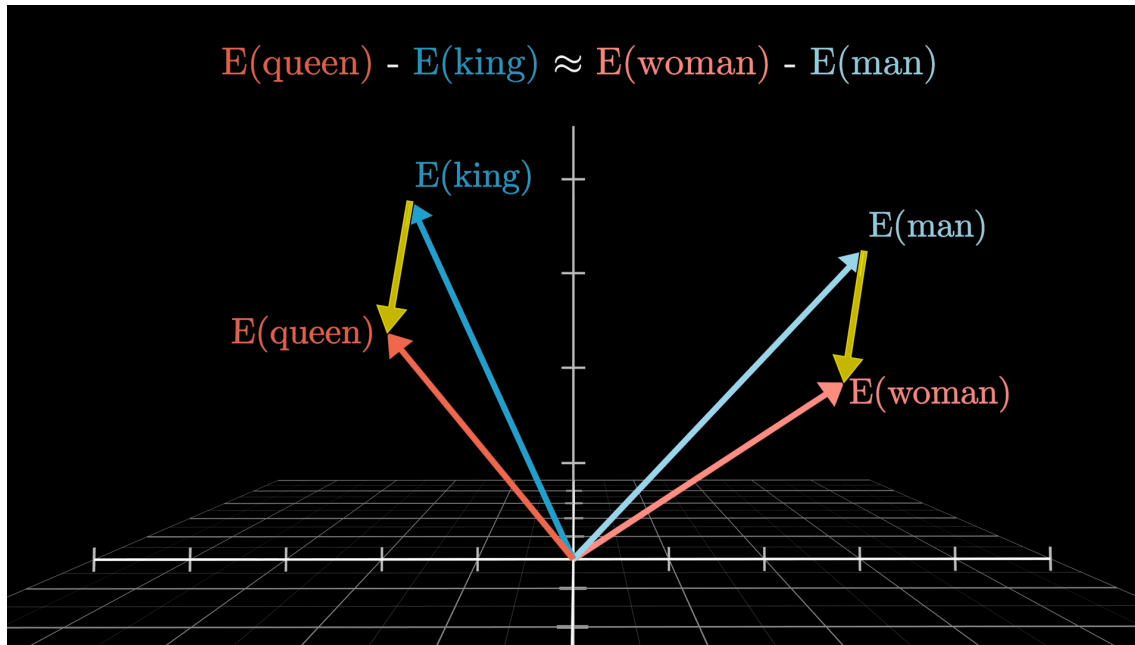
Before diving into the attention mechanism, recall a common preprocessing step in NLP. Words are typically represented as dense vectors called word embeddings. These embeddings are stored in an embedding matrix  $W_E$ , where  $d$  is the embedding dimension and  $n$  is the vocabulary size. Embeddings provide the foundation for queries, keys, and values in attention.



<https://www.youtube.com/watch?v=wjZofJX0v4M>

# Word Vector Embedding

- The model tends to settle on a set of embeddings where **directions** in the space have a kind of **semantic meaning**.
- One classical example is that, the difference between vectors of “**man**” and “**woman**” is very similar to that between “**king**” and “**queen**”. Consequently, we can simply find the embedding of a female monarch by taking “**king**”, adding the difference “**woman**” – “**man**” and search such an embedding.

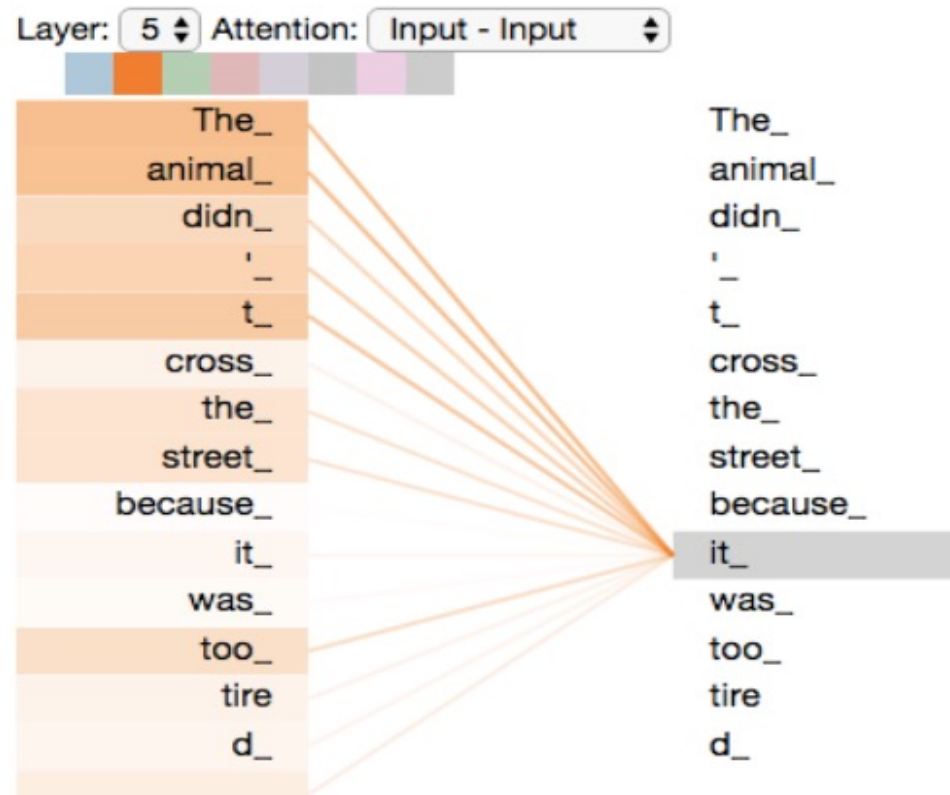


<https://www.youtube.com/watch?v=wjZofJX0v4M>



# Self-Attention: Motivation

- Suppose the following sentence is an input sentence we want to translate: *The animal didn't cross the street because it was too tired.* What does “it” in the sentence refer to? Is it referring to the street or the animal? It's a simple question to a human, but not as simple to an algorithm.
- The self-attention mechanism enables the model to associate “it” with “animal”.

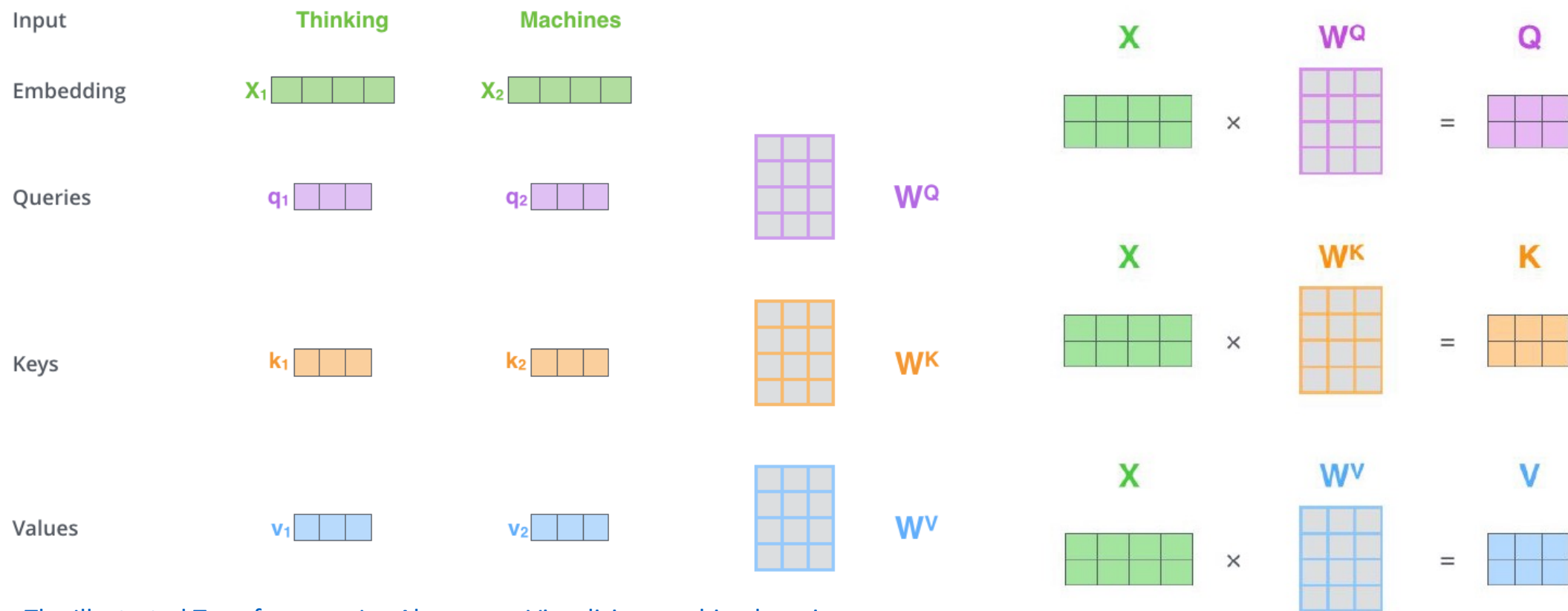


As we are encoding the word “it” in encoder #5, which is the top encoder in the stack, part of the attention mechanism was focusing on “The Animal”, and baked a part of its representation into the encoding of “it”.

[The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. \(jalammar.github.io\)](https://jalammar.github.io/)

# Self-Attention: Details

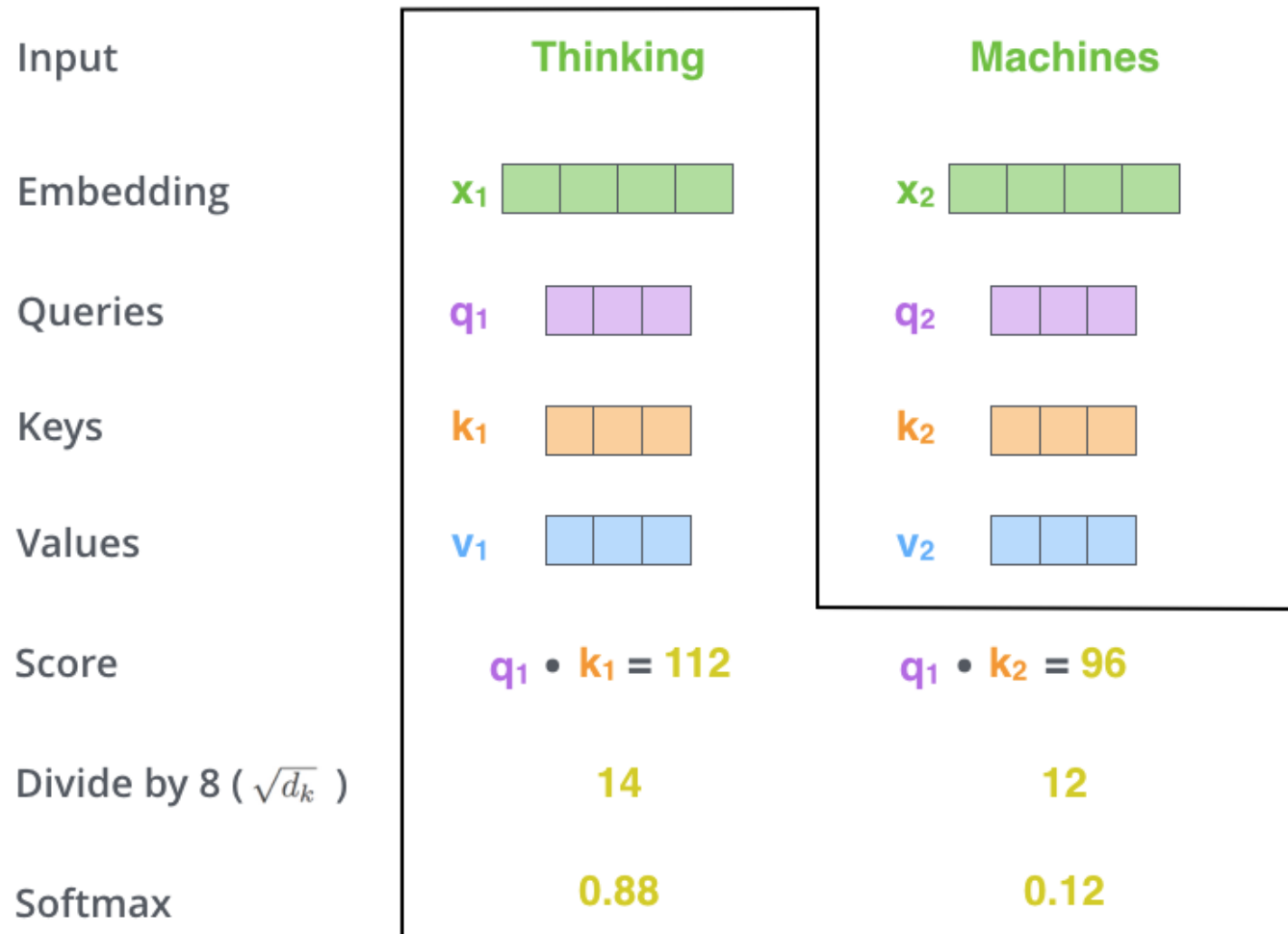
- **The first step is to create three vectors from each of encoder's input vectors.** These vectors are created by multiplying the embedding by three matrices that we trained during the training process (usually smaller in dimension than the embedding vector).



[The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. \(jalammar.github.io\)](https://jalammar.github.io/)

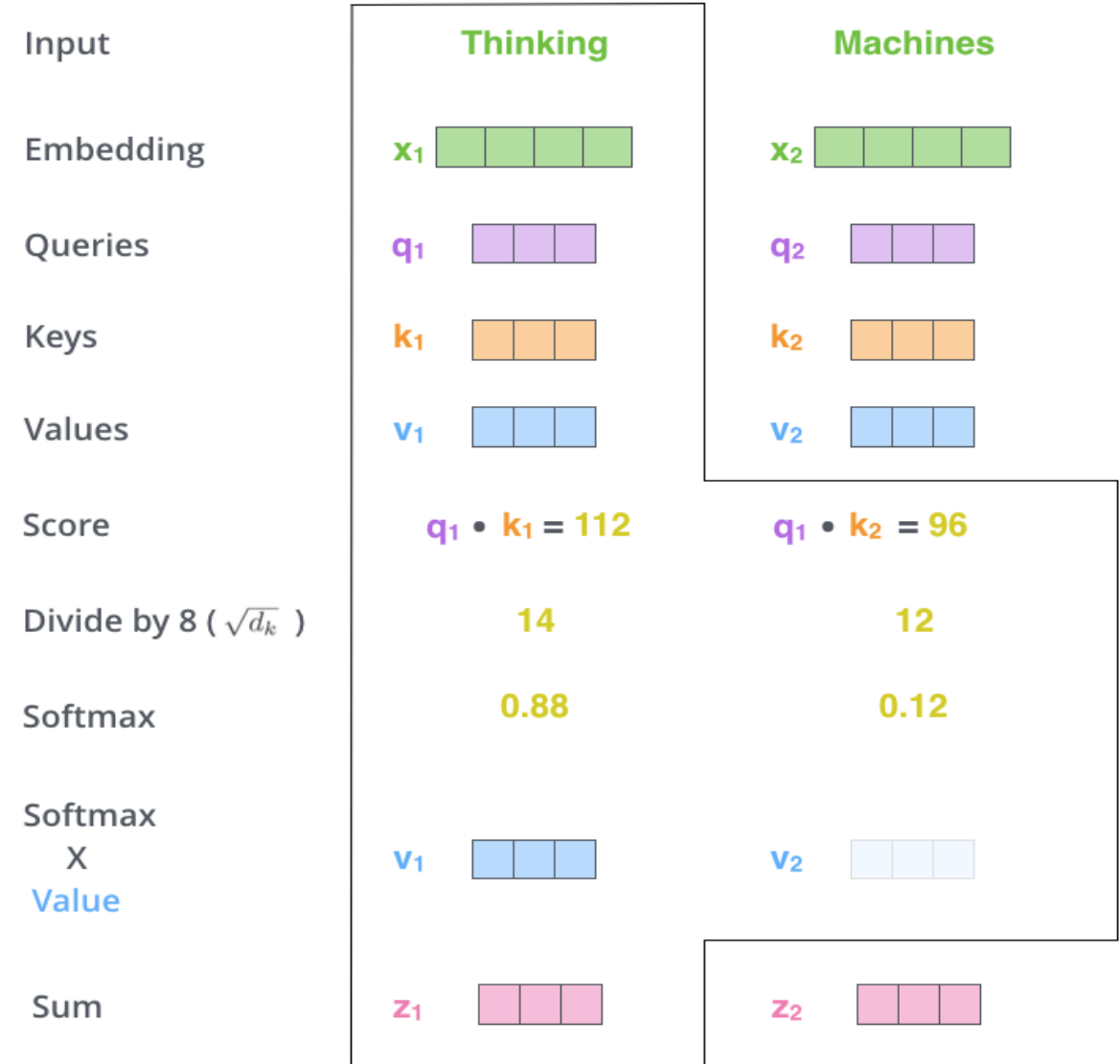
# Self-Attention: Details

- The second step is to calculate the scoring function and then divide it by the square root of the dimension of the key vectors.



# Self-Attention: Details

- The third step is to multiply each value vector by the softmax score and sum up the weighted value vectors.
- The resulting vector is the one we can send along to the feed-forward neural network.
- In the actual implementation, such calculation is done in matrix form for faster processing.



# Self-Attention: Matrix Calculation

- Query, key and value matrices are calculated through matrix multiplication.

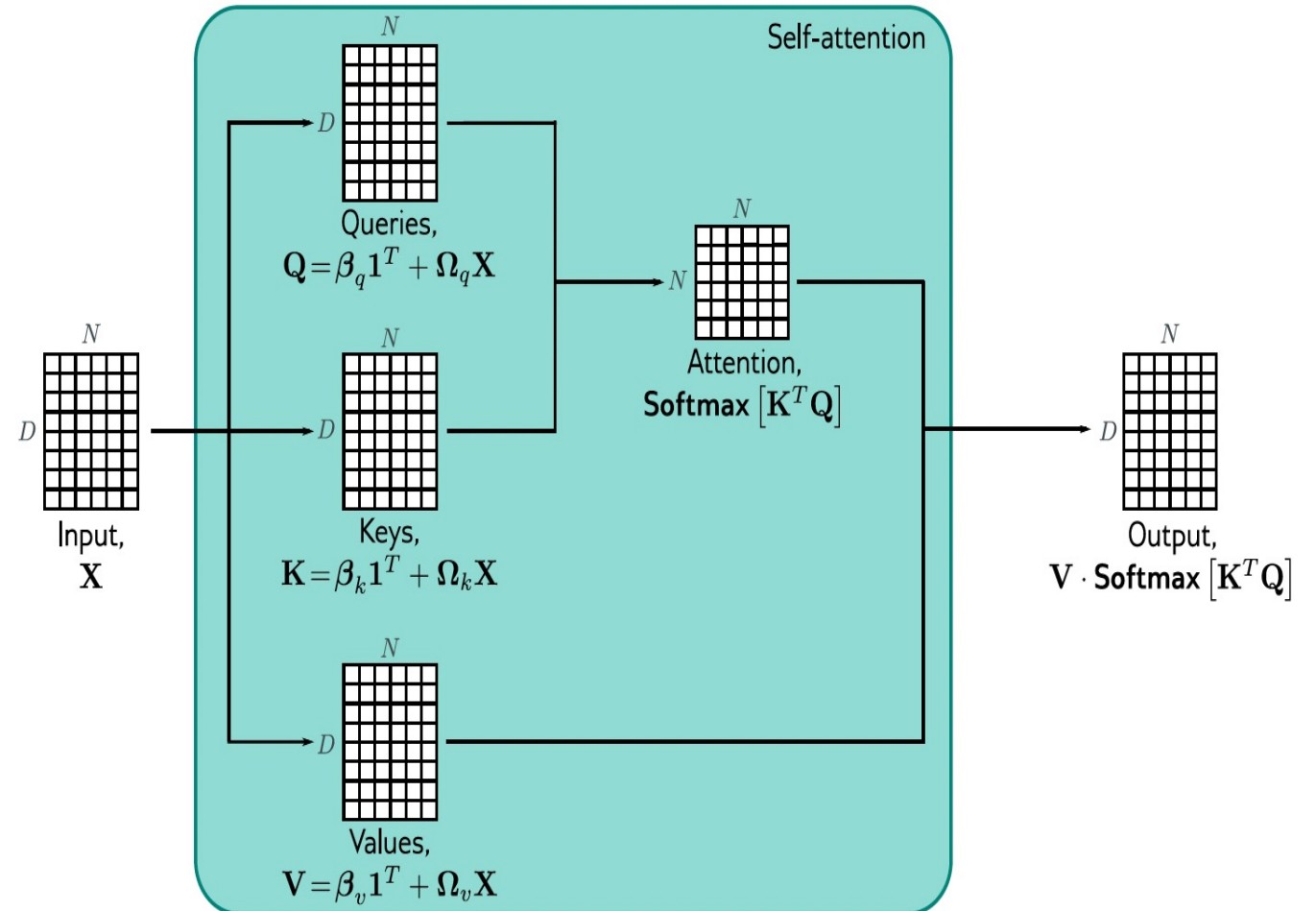
$$Q = W_Q X = \beta_q 1^T + \Omega_q X$$

$$K = W_K X = \beta_k 1^T + \Omega_k X$$

$$V = W_V X = \beta_v 1^T + \Omega_v X$$

$$Y = \text{Softmax}(QK^T)V$$

$$\text{softmax}\left(\frac{\begin{matrix} \text{Q} \\ \text{6x3 grid} \end{matrix} \times \begin{matrix} \text{K}^T \\ \text{3x6 grid} \end{matrix}}{\sqrt{d_k}}\right) \begin{matrix} \text{V} \\ \text{6x3 grid} \end{matrix} = \begin{matrix} \text{Z} \\ \text{6x3 grid} \end{matrix}$$



# Self-Attention: Equivariance and Invariance

## Definition

- ▶  $T_\pi(X) = XP_\pi$  is a spatial permutation.
- ▶ Equivariant:  $A(T_\pi(X)) = T_\pi(A(X))$
- ▶ Invariant:  $A(T_\pi(X)) = A(X)$

## Sketch

- ▶ Let  $X \in \mathbb{R}^{d \times n}$ ,  $P_\pi \in \mathbb{R}^{n \times n}$  a permutation matrix.
- ▶  $T_\pi(X) = XP_\pi$  applies the permutation to columns.
- ▶ Compute attention on  $XP_\pi$ :

$$O' = W_v X P_\pi \cdot \text{softmax}((W_k X P_\pi)^\top (W_q X P_\pi))$$

- ▶ Since softmax is equivariant and dot products are permutation-consistent:

$$O' = (W_v X \cdot \text{softmax}((W_k X)^\top (W_q X))) P_\pi = O P_\pi$$

## Theorem

- ▶ Self-attention is equivariant:

$$As(T_\pi(X)) = T_\pi(As(X))$$

- ▶ Attention with learned query is invariant:

$$AQ(T_\pi(X)) = AQ(X)$$

# RNN, CNN, and Self-Attention

**RNNs** are best for capturing sequential dependencies but struggle with long-range patterns.

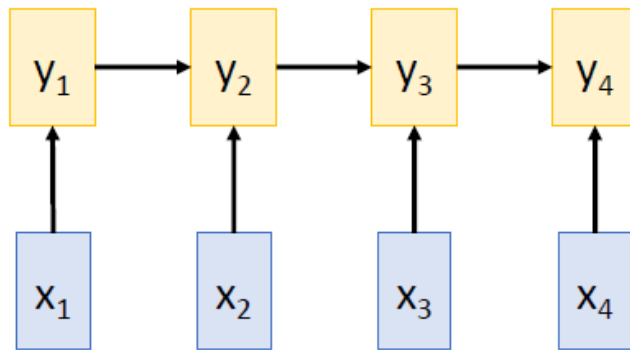
**CNNs** excel at local spatial features and parallel processing but are not well-suited for sequence data.

**Self-Attention** provides flexibility in capturing long-range dependencies and global contexts but is computationally expensive for very long sequences.

Aspect	RNNs	CNNs	Self-Attention
Primary Purpose	Sequential data processing	Spatial data processing	Capturing global relationships and dependencies
Data Handling	Temporal and ordered sequences	Local and spatial features	Long-range and global dependencies
Model Type	Recurrent neural networks	Convolutional neural networks	Attention-based, often in transformers
Core Mechanism	Recurrence and hidden states	Convolutions with filters/kernels	Query-Key-Value attention mechanism

# Comparing RNNs, CNNs and Self-Attention

## Recurrent Neural Network

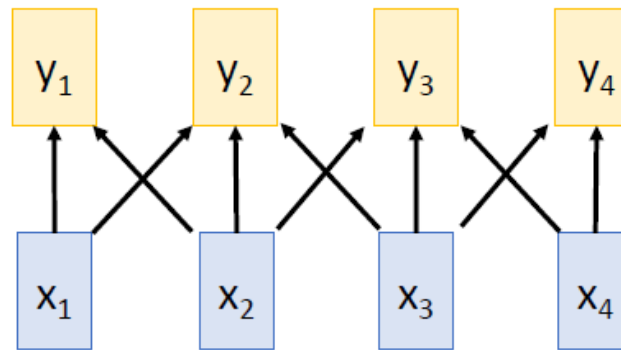


Works on **Ordered Sequences**

(+) **Good at long sequences:** After one RNN layer,  $h_T$  "sees" the whole sequence

(-) **Not parallelizable:** need to compute hidden states sequentially

## 1D Convolution



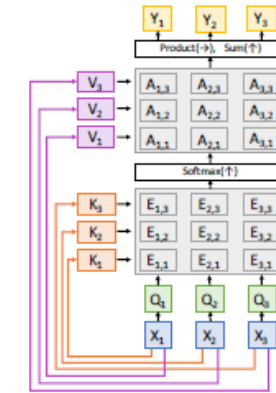
Works on **Multidimensional Grids**

(-) **Bad at long sequences:** Need to stack many conv layers for outputs to "see" the whole sequence

(+) **Highly parallel:** Each output can be computed in parallel

(Johnson, 2022)

## Self-Attention



Works on **Sets of Vectors**

(-) **Good at long sequences:** after one self-attention layer, each output "sees" all inputs!

(+) **Highly parallel:** Each output can be computed in parallel

(-) **Very memory intensive**



# Multi-Head Attention

- In practice, given the same set of queries, keys and values we may want our model to combine knowledge from different behaviors of the same attention mechanism, such as capturing dependencies of various ranges (e.g., shorter-range vs. longer-range) within a sequence. Thus, it may be beneficial to allow our attention mechanism to jointly use different representation subspaces of queries, keys and values.
- Instead of performing a single attention pooling, queries, keys, and values can instead be transformed with  $h$  independently learned linear projections. These  $h$  projected queries, keys and values are fed into attention pooling in parallel.
- This design is called **multi-head attention**, where each of the  $h$  attention pooling outputs is **a head**.

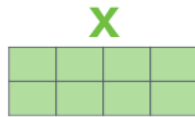
# Multi-Head Attention

Multi-head attention expands the model's ability to focus on different positions, as well as gives the attention layer multiple “representation subspaces”, thus improving the expressivity of the model.

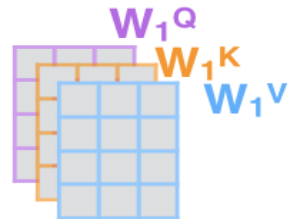
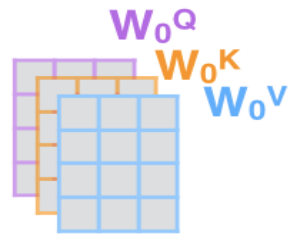
1) This is our input sentence\*

Thinking  
Machines

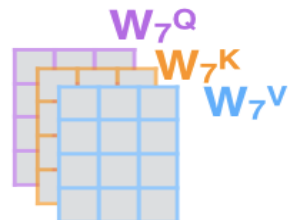
2) We embed each word\*



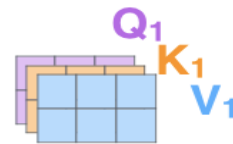
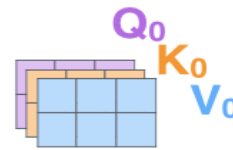
3) Split into 8 heads. We multiply  $X$  or  $R$  with weight matrices



...



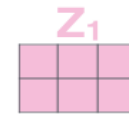
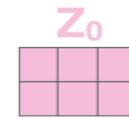
4) Calculate attention using the resulting  $Q/K/V$  matrices



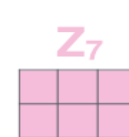
...



5) Concatenate the resulting  $Z$  matrices, then multiply with weight matrix  $W^O$  to produce the output of the layer



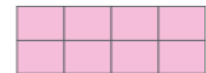
...



$W^O$



$Z$



\* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



# Content

1 Attention Mechanisms

**2 Positional Encoding**

3 Transformer Architecture

4 Transformer Applications

# Positional Embedding

- Unlike RNNs, which recurrently process tokens of a sequence one-by-one, self-attention ditches sequential operations in favor of parallel computation.
- However, self-attention by itself does not preserve the order of sequence, that is, it is equivariant to permutations of the inputs. What should we do to account for the order of words in the input sequence when it really matters?

**what we see:**

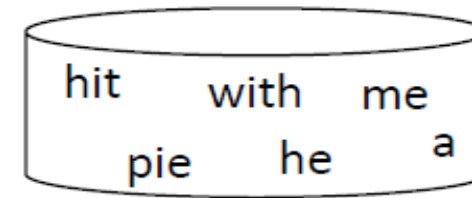
he hit me with a pie

**what naïve self-attention sees:**

a pie hit me with he

a hit with me he pie

he pie me with a hit



(Levine, 2021)

- The dominant approach for preserving information about the order of tokens is to represent this to the model as an additional input associated with each token. These inputs are called *positional embedding*, and can either be learned or a fixed *a priori*.

# Positional Embedding

- The naïve positional encoding would just append  $t$  to the input:  $\overline{x}_t = \begin{pmatrix} x_t \\ t \end{pmatrix}$ . However, it would not be a great idea, because the **absolute** position is less important than the **relative** position.

I walk my dog every day



every single day I walk my dog



The fact that “my dog” is right after “I walk” is the important part, not its absolute position

- ❖ Therefore, we want to represent position in a way that tokens with similar **relative position** will have similar position encoding.
- ❖ What about using frequency-based representations?

Sequence	Index of token	Positional Encoding Matrix			
I	0	$P_{00}$	$P_{01}$	...	$P_{0d}$
am	1	$P_{10}$	$P_{11}$	...	$P_{1d}$
a	2	$P_{20}$	$P_{21}$	...	$P_{2d}$
Robot	3	$P_{30}$	$P_{31}$	...	$P_{3d}$

Positional Encoding Matrix for the sequence 'I am a robot'

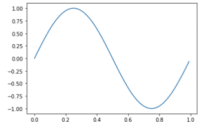
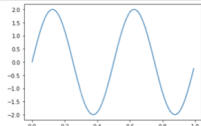
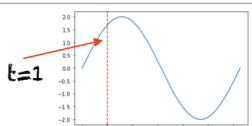
# Positional Embedding: Example

- One typical scheme for fixed positional encodings are based on sine and cosine functions.
- Suppose that  $X \in \mathbb{R}^{n \times d}$  contains the  $d$ -dimensional embeddings for  $n$  tokens of a sequence. The positional encoding outputs  $X + P$  using a positional embedding matrix  $P \in \mathbb{R}^{n \times d}$  of the same shape, whose element on the  $i$ th row and the  $(2j)$ th or  $(2j + 1)$ th column is given by

$$p_{i,2j} = \sin\left(\frac{i}{1000 \frac{2j}{d}}\right), p_{i,2j+1} = \cos\left(\frac{i}{1000 \frac{2j}{d}}\right)$$

## ► Angular Frequency ( $\omega_j$ ):

- $\omega_j = \frac{1}{1000 \frac{2j}{d}}$
- Each increment  $i \rightarrow i + 1$  increases the argument by  $\omega_j$ .

Equation	Graph	Frequency	Wavelength
$\sin(2\pi t)$		1	1
$\sin(2 * 2\pi t)$		2	1/2
$\sin(t)$		$1/2\pi$	$2\pi$
$\sin(ct)$	Depends on $c$	$c/2\pi$	$2\pi/c$

## ► Wavelength ( $\lambda_j$ ):

- $\lambda_j = \frac{2\pi}{\omega_j} = 2\pi \times 1000 \frac{2j}{d}$
- A larger exponent  $\frac{2j}{d}$  yields a larger wavelength.

# Positional Embedding: Example

- Equivalent to

$$p_{i,2j} = \sin\left(2\pi \frac{pos}{\lambda_j}\right), p_{i,2j+1} = \cos\left(2\pi \frac{pos}{\lambda_j}\right)$$

🎵 **Think of each dimension as a musical string:**

- Low-frequency dimensions (large  $\lambda$ ) → **slow, smooth waves**
- High-frequency dimensions (small  $\lambda$ ) → **fast, jittery waves**

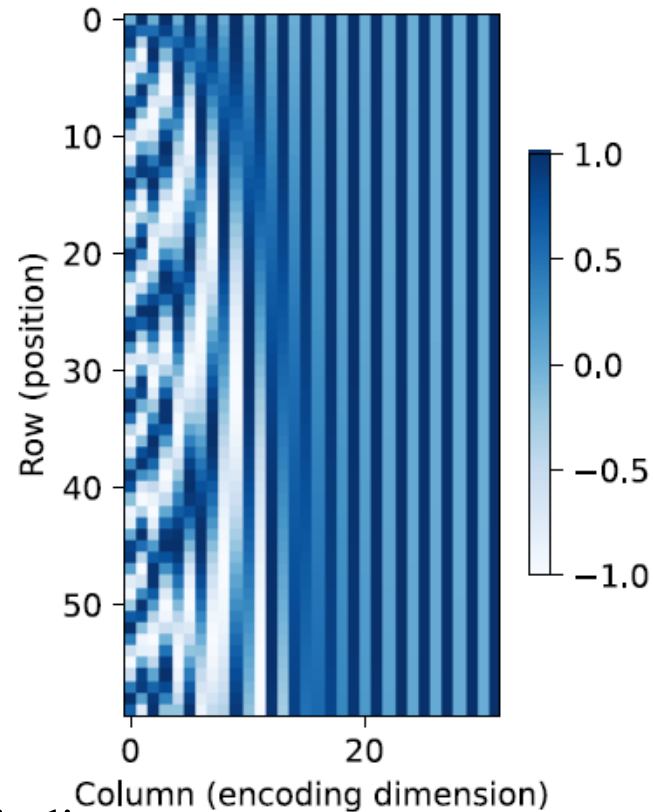
By combining them, the model gets a **richer encoding** of position, similar to combining bass and treble to form a musical tone.

- **How the Model Uses This**

- The **self-attention mechanism** doesn't know order inherently.
- Positional encodings **shift the vector space** so tokens at different positions are **separable**, even if their word embeddings are similar.
- This enables the model to **attend to relative and absolute positions** of tokens.

# Positional Embedding: Example

0 in binary is 000  
1 in binary is 001  
2 in binary is 010  
3 in binary is 011  
4 in binary is 100  
5 in binary is 101  
6 in binary is 110  
7 in binary is 111



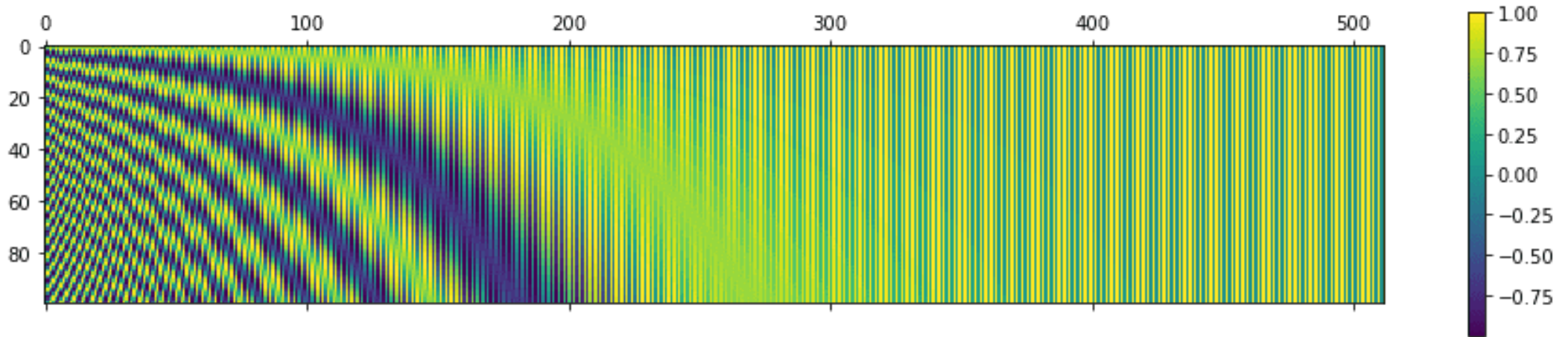
“even-odd” indicator

“first half-second half indicator” indicator

- ▶ **Monotonically decreasing frequency:** In sinusoidal positional encoding, each higher dimension has a lower frequency.
  - ▶  $\omega_j = \frac{1}{10000 \frac{2^j}{d}}$  decreases with  $j$ .
  - ▶ Encodes absolute position using increasingly coarse granularity.
- ▶ **Binary representation analogy:**
  - ▶ High-order bits change less frequently than low-order bits.
  - ▶ Similarly, high- $j$  dimensions oscillate more slowly in positional encoding.
- ▶ **Continuous vs. Binary:** Float-based sinusoidal encodings are *more space-efficient* and allow fractional offsets.



# Positional Embedding



## Relative Intuition

Because sinusoids are **shifted versions of one another**, subtracting positional encodings of positions  $a$  and  $b$  gives a pattern that the model can use to infer **relative positions**.

That's why:

- Positional encoding enables **absolute and relative** positioning,
- Without needing recurrence or convolution.

## Summary of Intuition

- **Wavelength** controls how fast the position-based values change.
- Using a range of wavelengths gives the model **fine to coarse positional resolution**.
- The **model learns patterns of where things are** via combinations of sinusoids, like Fourier encoding.

# Alternative Positional Embeddings

## Learnable Positional Embeddings:

- ▶ Treat position embeddings as trainable parameters.
- ▶ No mathematical constraint, but less interpretable.

## Relative Positional Encoding:

- ▶ Encodes the difference between positions.
- ▶ Helps with tasks where relative order matters (e.g., text generation).

## Rotary Positional Encoding:

- ▶ Efficiently integrates positional information into attention.
- ▶ Particularly effective for long-sequence tasks.

## Vision Transformer

### Patch-Based Tokens:

- ▶ Image is split into patches and embedded as tokens.
- ▶ A 2D coordinate (x, y ) is mapped to position embeddings.

$$PE_{(x,y)} = \text{concat}(PE(x), PE(y))$$

### 2D Positional Embeddings:

- ▶ Often a learnable embedding for each patch index.
- ▶ Alternatively, sinusoidal in each spatial dimension, then combined.

### Why It Works:

- ▶ Preserves spatial relationships for tasks like recognition, detection.
- ▶ Vision Transformers handle global and local contexts effectively.

# Content

1 Attention Mechanisms

2 Self-Attention and Positional Encoding

**3 Transformer Architecture**

4 Transformer Applications

# Why Transformers?

## Parallel Computation + Short Path Length:

- ▶ Self-attention can handle all tokens in parallel.
- ▶ Minimal path length for global dependencies, vital for deep architectures.

## Transformer Dominance in NLP:

- ▶ Nearly all state-of-the-art language tasks use Transformer based models.
- ▶ Default approach: "Grab a large pretrained Transformer" (BERT, GPT, T5, etc.).

## Vision Transformer (ViT):

- ▶ Patch-based input turned into token embeddings.
- ▶ Now a go-to model for image recognition, detection, and segmentation.

Islam, S., Elmekki, H., Elsebai, A., Bentahar, J., Drawel, N., Rjoub, G., & Pedrycz, W. (2024). A comprehensive survey on applications of transformers for deep learning tasks. *Expert Systems with Applications*, 241, 122666.

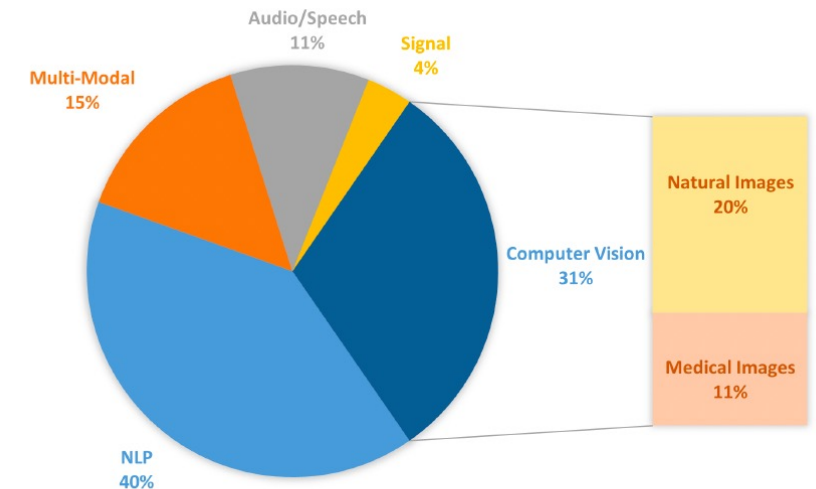
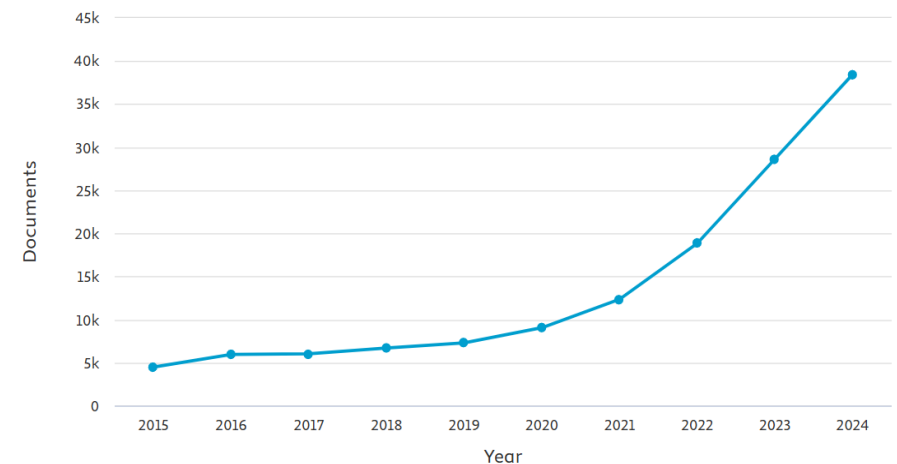


Figure 4: Proportion of transformer application in Top-5 fields

Documents by year

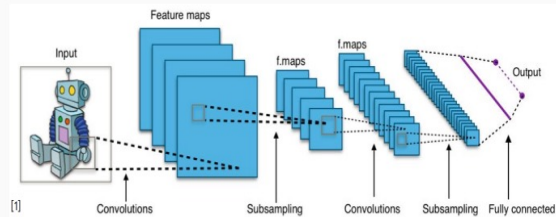


# Why Transformers?

## Before Transformers

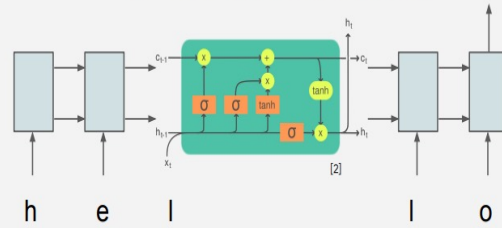
### Computer Vision

Convolutional NNs (+ResNets)



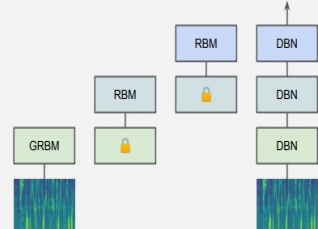
### Natural Lang. Proc.

Recurrent NNs (+LSTMs)



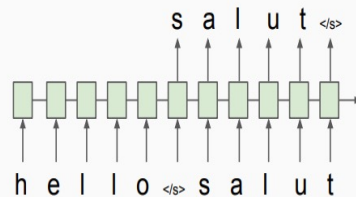
### Speech

Deep Belief Nets (+non-DL)



### Translation

Seq2Seq



### RL

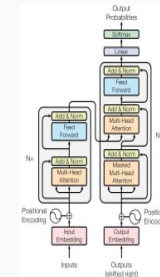
BC/GAIL

**Algorithm 1** Generative adversarial imitation learning

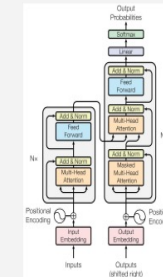
- 1: **Input:** Expert trajectories  $\tau_E \sim \pi_E$ , initial policy and discriminator parameters  $\theta_0, w_0$
- 2: **for**  $t = 0, 1, 2, \dots$  **do**
- 3: Sample trajectories  $\tau_t \sim \pi_{\theta_t}$
- 4: Update the discriminator parameters from  $w_t$  to  $w_{t+1}$  with the gradient
 
$$\hat{E}_{\tau_t} [\nabla_w \log(D_w(s, a))] + \hat{E}_{\tau_E} [\nabla_w \log(1 - D_w(s, a))] \quad (17)$$
- 5: Take a policy step from  $\theta_t$  to  $\theta_{t+1}$ , using the TRPO rule with cost function  $\log(D_{w_{t+1}}(s, a))$ . Specifically, take a KL-constrained natural gradient step with
 
$$\hat{E}_{\tau_t} [\nabla_{\theta} \log \pi_{\theta}(a|s) Q(s, a)] - \lambda \nabla_{\theta} H(\pi_{\theta}), \quad (18)$$
 where  $Q(s, a) = \hat{E}_{\tau_E} [\log(D_{w_{t+1}}(s, a))] | a_0 = s, a_0 = a |$
- 6: **end for**

## After Transformers

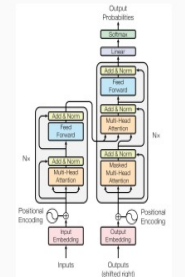
### Computer Vision



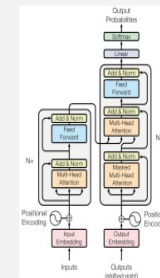
### Natural Lang. Proc.



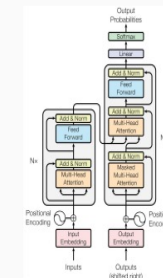
### Reinf. Learning



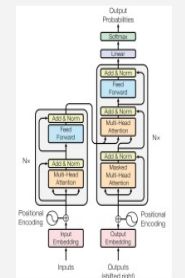
### Speech



### Translation



### Graphs/Science



(Bertasius, 2024)

# From Attention Mechanism to Transformer

- If we have attention, do we even need recurrent connections?
- Can we transform our RNN into a purely attention-based model?

---

## Attention Is All You Need

---

Neurips 2017

**Ashish Vaswani\***  
Google Brain  
avaswani@google.com

**Noam Shazeer\***  
Google Brain  
noam@google.com

**Niki Parmar\***  
Google Research  
nikip@google.com

**Jakob Uszkoreit\***  
Google Research  
usz@google.com

**Llion Jones\***  
Google Research  
llion@google.com

**Aidan N. Gomez\* †**  
University of Toronto  
aidan@cs.toronto.edu

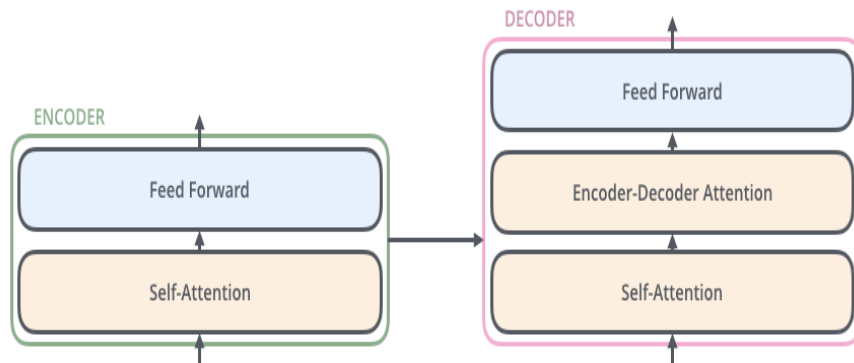
**Łukasz Kaiser\***  
Google Brain  
lukaszkaizer@google.com

**Illia Polosukhin\* ‡**  
illia.polosukhin@gmail.com

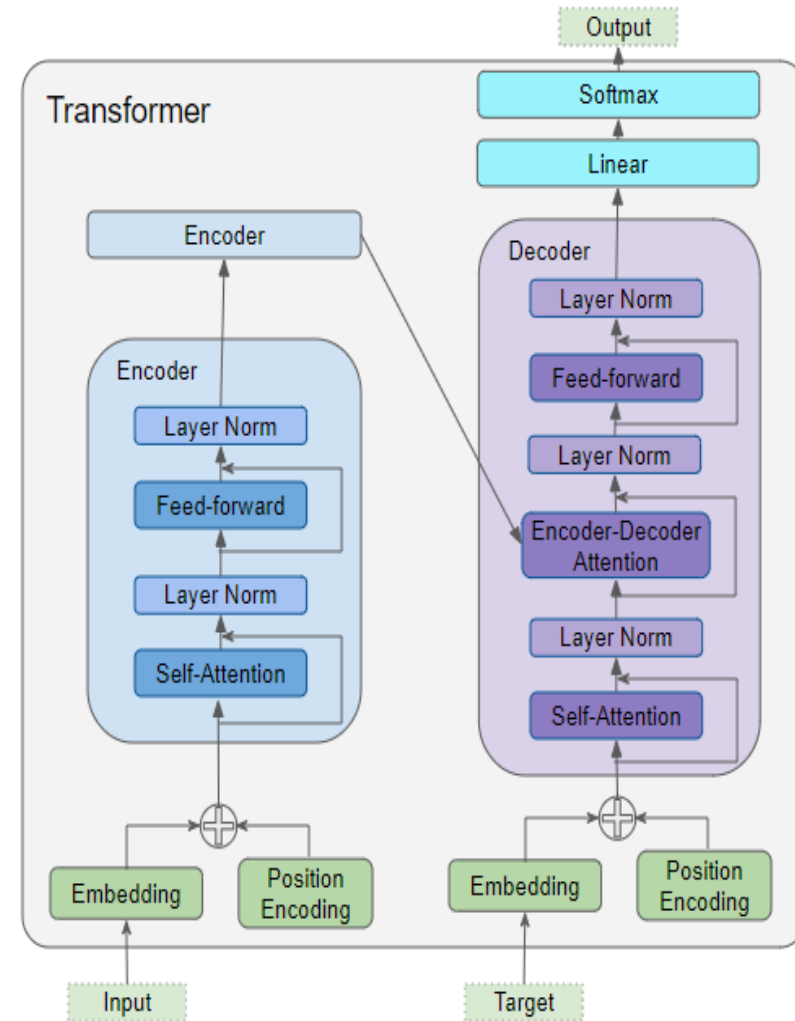
**Citation: 173883+**

# Transformer

- **Transformer** is composed of an **encoder** and a **decoder**.
- The **input and output sequence embeddings** are added with **positional encoding** before being fed into the encoder and the decoder.
- The **encoder** is a stack of multiple Transformer layers, used to **transform the text embeddings into a representation** that can support a variety of tasks.
- The **decoder** is also a stack of multiple Transformer layers, used to **predict the next token to continue the input text**. It also inserts a sub-layer, known as the **encoder-decoder attention**.



<https://jalammar.github.io/illustrated-transformer/>



<https://towardsdatascience.com/transformers-explained-visually-part-2-how-it-works-step-by-step-b49fa4a64f34/>



# Transformer Layer (Block)

► **Input:**  $D \times N$  matrix of word embeddings, where  $D$  is the embedding dimension and  $N$  the sequence length.

► **Multi-Head Attention:**

- ❖ Each token can attend to every other token.
- ❖ Output dimension is  $D \times N$ .
- ❖ Residual connection: add the original inputs back.

► **LayerNorm:**

- Applied to each column (token) independently.
- Normalizes across the embedding dimension.

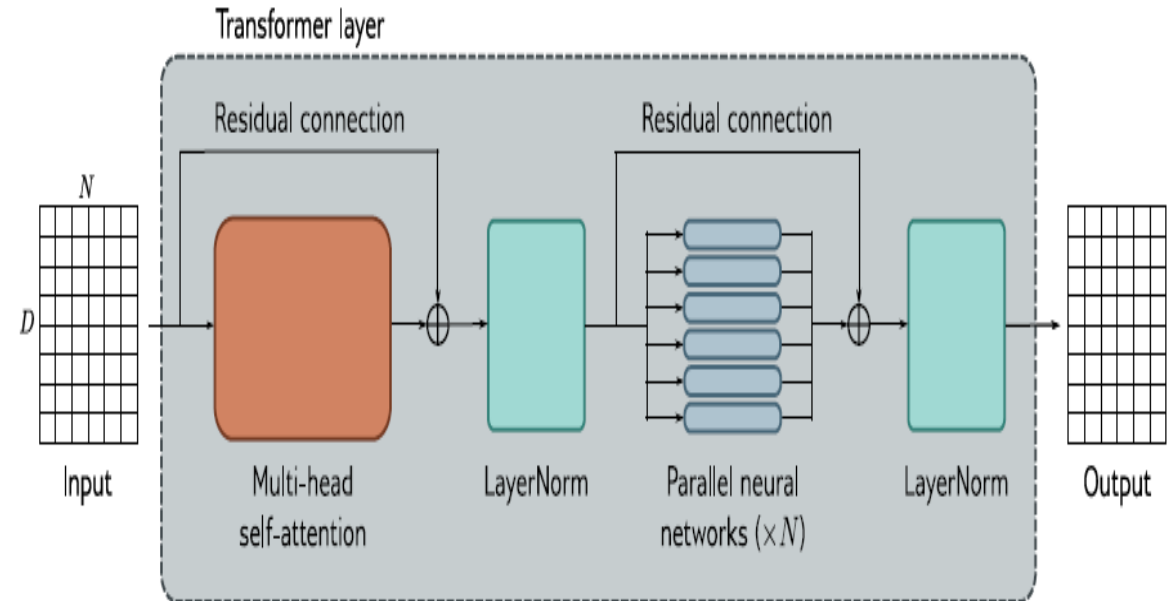
► **Fully Connected Feed-Forward:**

- ✓ Same MLP applied to each column.
- ✓ Residual connection again.

► **Final LayerNorm:**

- ❑ Normalizes outputs across  $D$  for each token.

► **Result:** Output is a  $D \times N$  matrix with updated token representations.



$$\mathbf{X} \leftarrow \mathbf{X} + \text{MhSa}[\mathbf{X}]$$

$$\mathbf{X} \leftarrow \text{LayerNorm}[\mathbf{X}]$$

$$\mathbf{x}_n \leftarrow \mathbf{x}_n + \text{mlp}[\mathbf{x}_n] \quad \forall n \in \{1, \dots, N\}$$

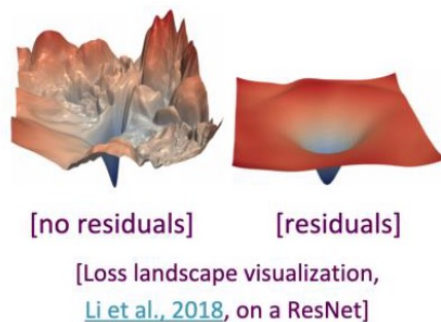
$$\mathbf{X} \leftarrow \text{LayerNorm}[\mathbf{X}],$$



# Transformer Layer: Residual Connection

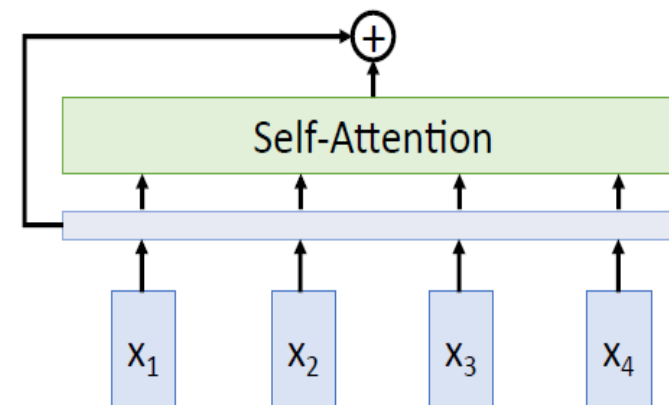
- Residual connection is a simple but powerful technique from computer vision.
- Observation: Deep neural networks are surprisingly bad at learning the identity function.
- Therefore, directly passing “raw” embeddings to the next layer would be very helpful!
$$x_l = f(x_{l-1}) + x_{l-1}$$
- This prevents the network from “forgetting” or distorting important information as it is processed by many layers.

Residual connections are also thought to smooth the loss landscape and make training easier!



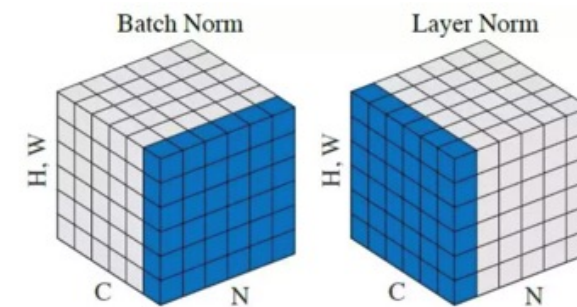
```
class AddNorm(nn.Module):  
    """The residual connection followed by layer normalization."""  
    def __init__(self, norm_shape, dropout):  
        super().__init__()  
        self.dropout = nn.Dropout(dropout)  
        self.ln = nn.LayerNorm(norm_shape)  
  
    def forward(self, X, Y):  
        return self.ln(self.dropout(Y) + X)
```

Residual connection  
All vectors interact



# Transformer Layer: Layer Normalization

- **Problem:** Deep neural networks often suffer from internal covariate shift, where the distribution of inputs to each layer changes during training, making optimization difficult.
- **Solution:** Reduce variation by normalizing to zero mean and standard deviation of one within each layer.



**Batch norm**  $d$ -dimensional vectors  $a_1, a_2, \dots, a_B$  for each sample in batch

$$\mu = \frac{1}{B} \sum_{i=1}^B a_i \quad \sigma = \sqrt{\frac{1}{B} \sum_{i=1}^B (a_i - \mu)^2}$$

$d$ -dim  $\mu$   $\sigma$

$$\bar{a}_i = \frac{a_i - \mu}{\sigma} \gamma + \beta$$

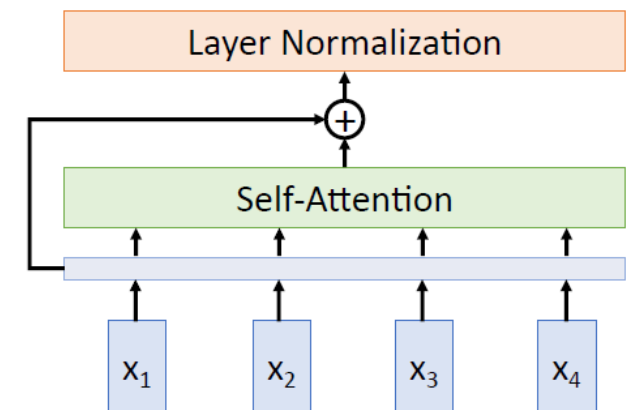
**Layer norm**  $a$  different dimensions of  $a$

$$\mu = \frac{1}{d} \sum_{j=1}^d a_j \quad \sigma = \sqrt{\frac{1}{d} \sum_{j=1}^d (a_j - \mu)^2}$$

$1$ -dim  $\mu$   $\sigma$

$$\bar{a} = \frac{a - \mu}{\sigma} \gamma + \beta$$

Residual connection  
All vectors interact  
with each other

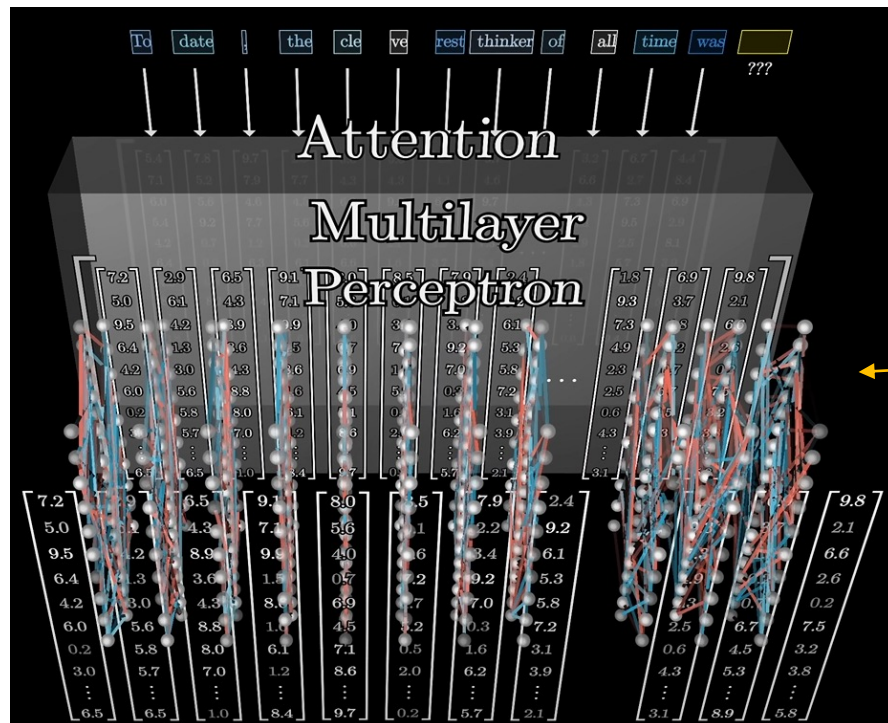


Layer norm is not applied to an entire transformer layer, but just to the embedding vector of a single token.

# Position-wise Feed-Forward Networks

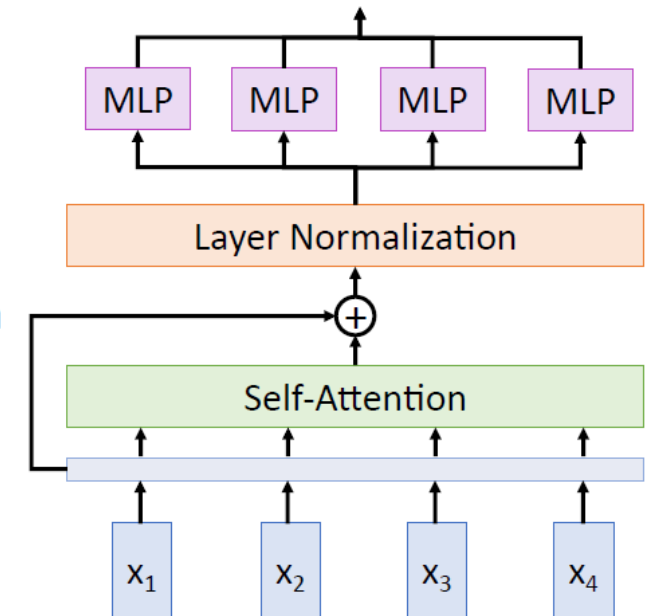
- The position-wise feed-forward network transforms the representation at all the sequence positions using the same MLP. This is why we call it *position-wise*.

```
class PositionWiseFFN(nn.Module):  
    """The positionwise feed-forward network."""  
    def __init__(self, ffn_num_hiddens, ffn_num_outputs):  
        super().__init__()  
        self.dense1 = nn.Linear(ffn_num_hiddens, ffn_num_hiddens)  
        self.relu = nn.ReLU()  
        self.dense2 = nn.Linear(ffn_num_hiddens, ffn_num_outputs)  
  
    def forward(self, X):  
        return self.dense2(self.relu(self.dense1(X)))
```



MLP independently  
on each vector

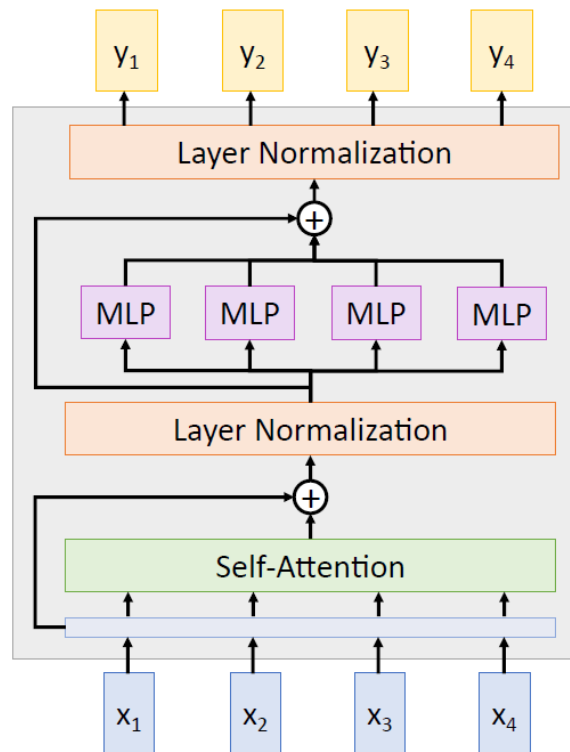
Residual connection  
All vectors interact  
with each other



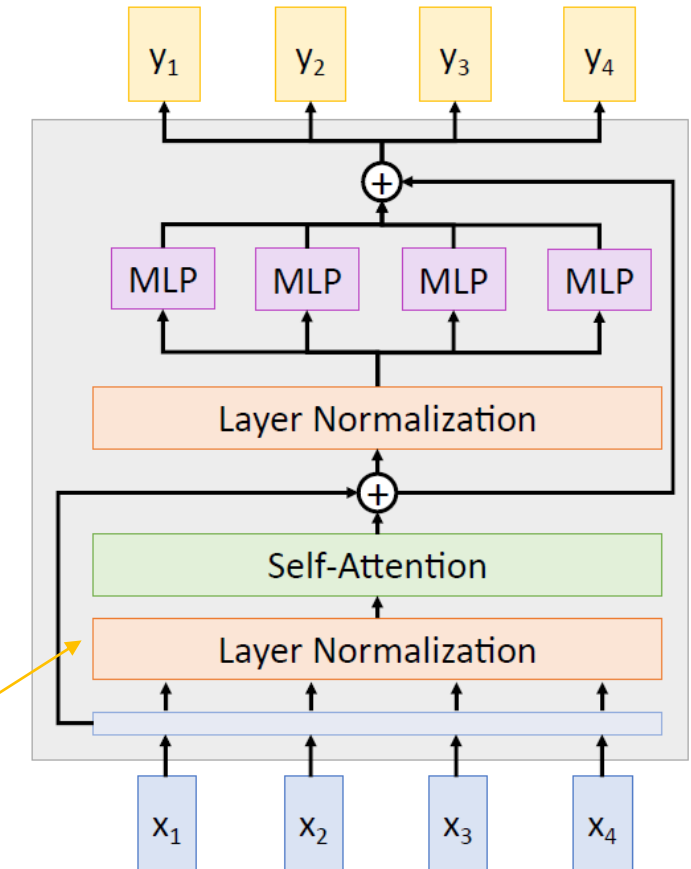
<https://www.youtube.com/watch?v=wjZofJX0v4M>

# Putting it All Together

- Self-attention is the only interaction between vectors.
- Layer normalization and MLP work independently per vector.
- The structure is highly scalable and highly parallelizable.



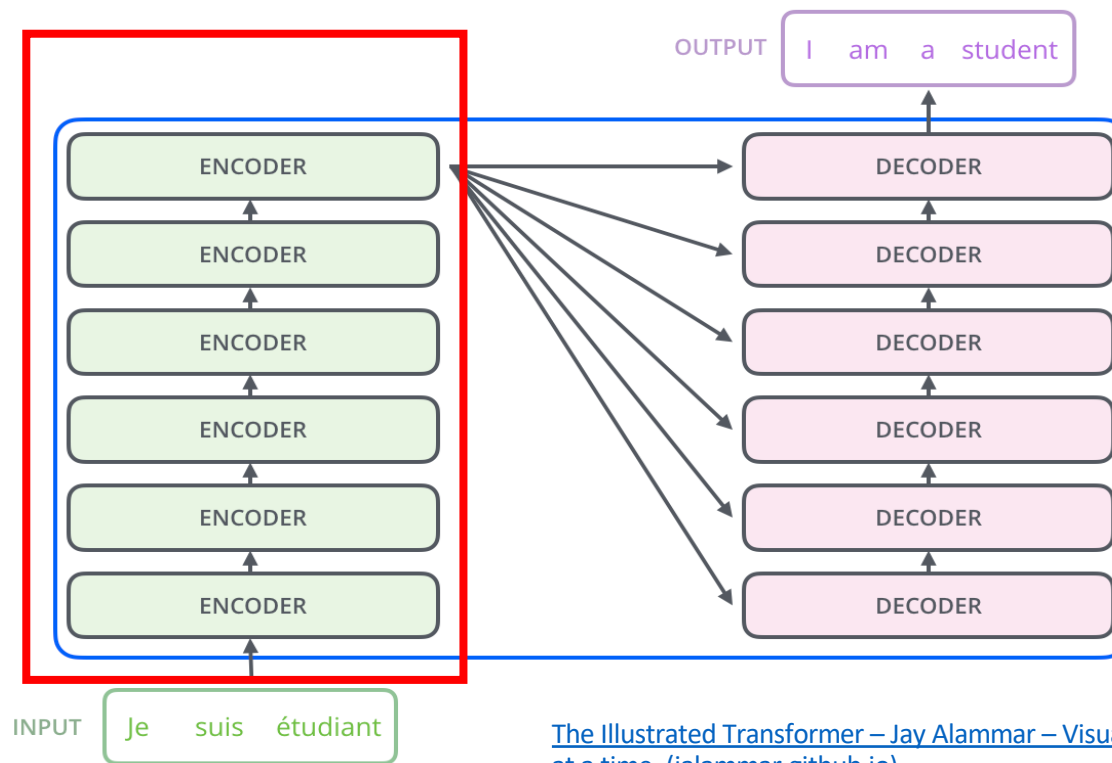
(Johnson, 2022)



In practice, we often put the layer normalization inside the residual attention, which tend to give more stable training and is commonly used in practice.

# Encoder

- The Transformer encoder consists of multiple identical Transformer layers that process the input sequence in parallel. Each layer refines the input representation by capturing dependencies across all positions. ( $N = 6$  in the paper *Attention is all you need*).
- The encoder outputs a contextualized representation for each token, which serves as input to the decoder.



- ▶ **Input:**  $D \times N_{\text{enc}}$  matrix of embeddings (or projected tokens).
- ▶ **Self-Attention Layer:**
  - ▶ Each token attends to every other token in the source.
  - ▶ Multi-head mechanism for capturing diverse relationships.
  - ▶ Residual connection + LayerNorm keep gradients stable.
- ▶ **Feed-Forward Layer:**
  - ▶ Position-wise MLP applied to each token's embedding.
  - ▶ Another residual connection + LayerNorm.
- ▶ **Stacking Layers:**
  - ▶ Typically  $L$  identical encoder layers.
  - ▶ Output is  $D \times N_{\text{enc}}$ , providing contextualized embeddings for each source token.

[The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. \(jalammar.github.io\)](https://github.com/jalammar/the-illustrated-transformer)

# Transformer Decoder

## ► Decoder Sublayers:

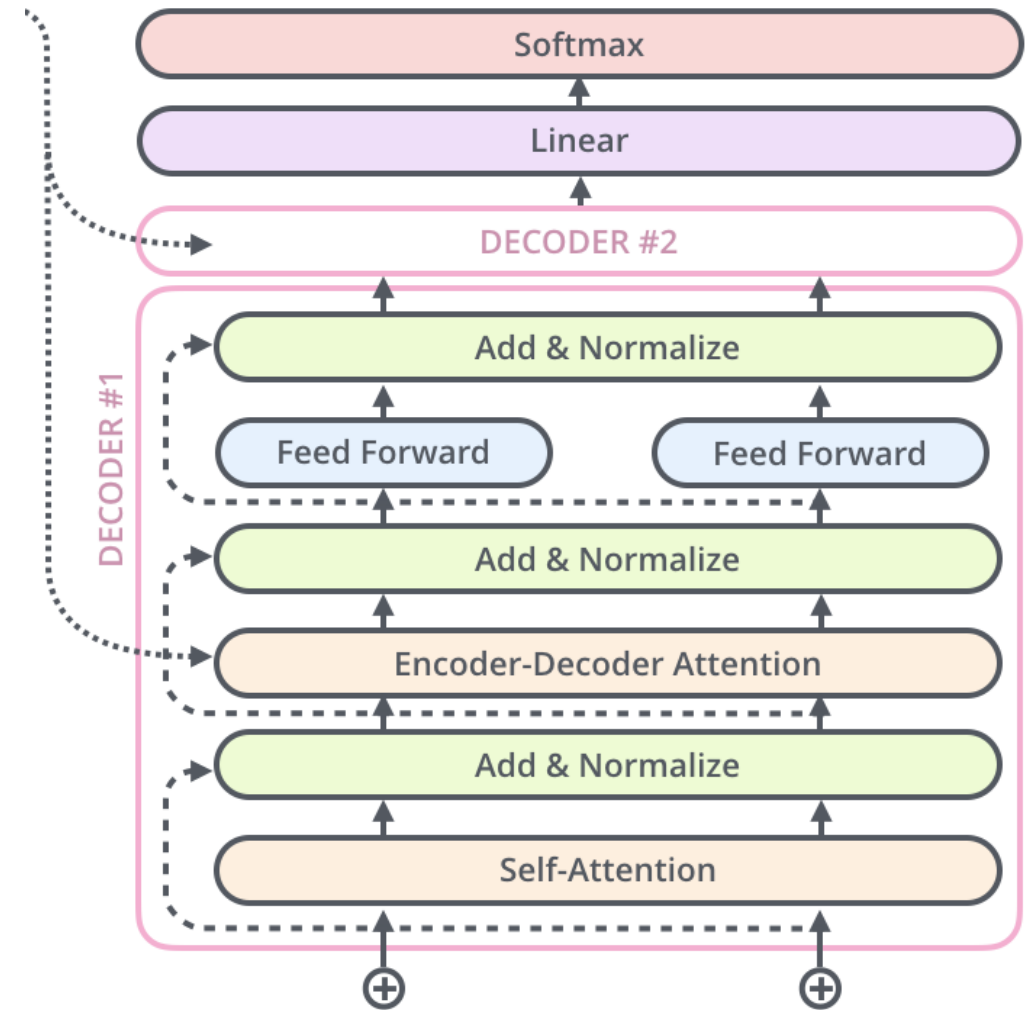
1. **Masked Self-Attention:** Targets attending to themselves (past tokens).
2. **Cross-Attention:** Queries from the decoder, Keys and Values from encoder.
3. **Feed-Forward Network:** Applies position-wise transformations to each token.

## ► Residuals + LayerNorm:

- Each sublayer uses skip connections and normalization.
- Ensures stable training and consistent dimensionality.

## ► Outcome:

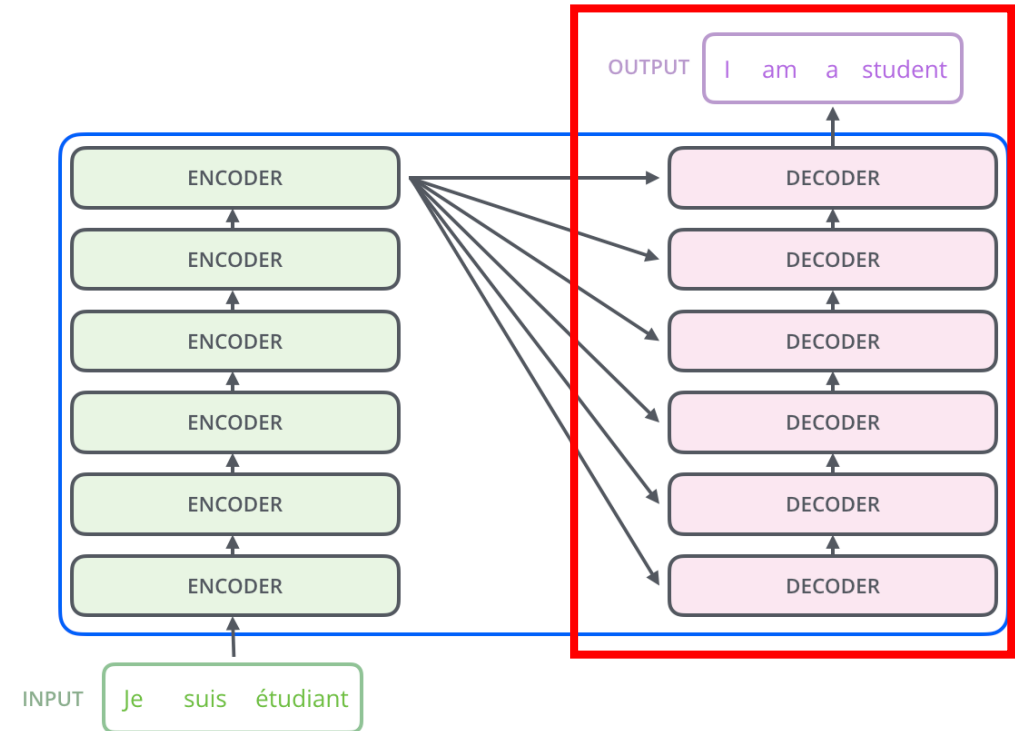
- Decoder hidden states are enriched with relevant source info.
- Final step: linear layer + softmax for next-token prediction.



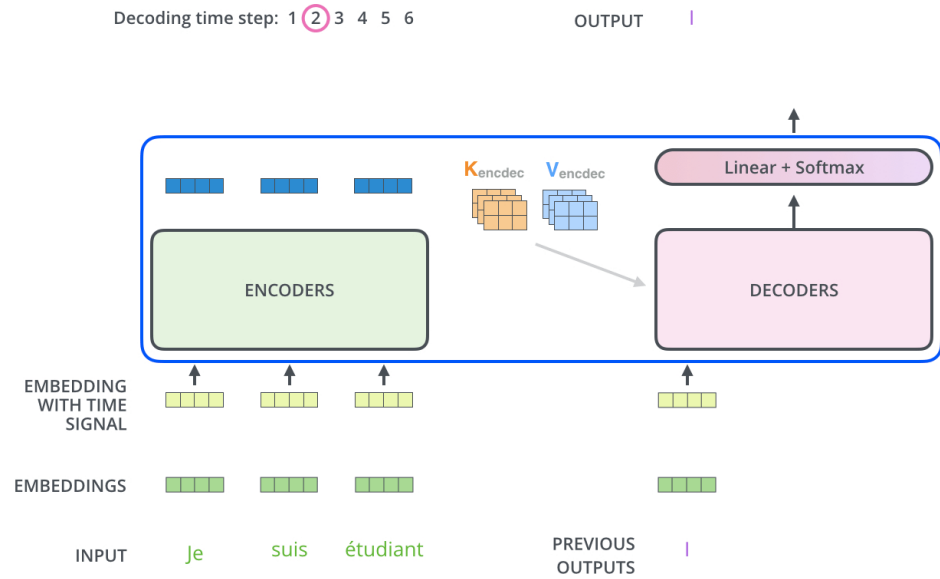
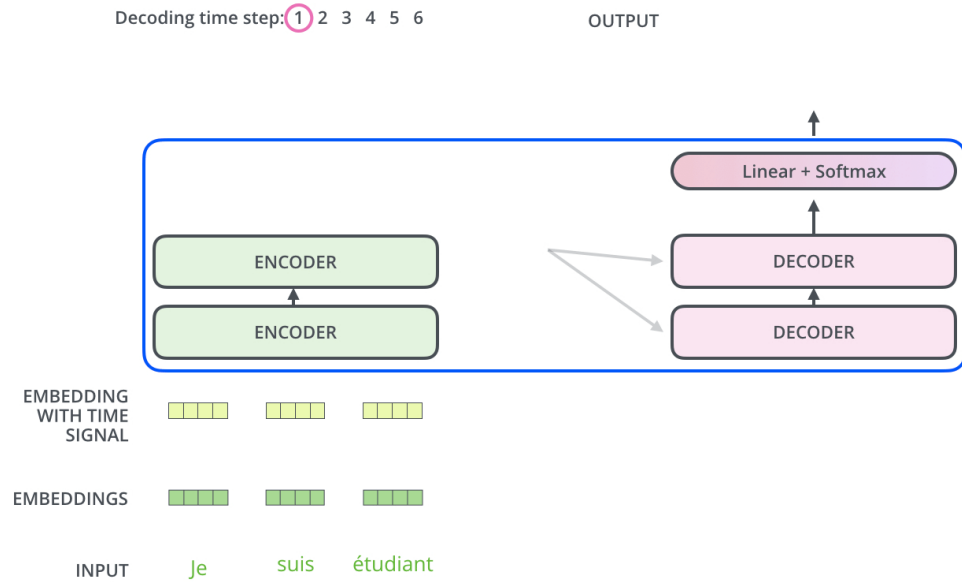


# Decoder: Masked Self-Attention

- The Transformer decoder is also composed of multiple layers and generates the output sequence step by step. In the **decoder self-attention**, queries, keys and values are all from the outputs of the previous decoder layer.
- However, each position in the decoder is allowed only to attend to all positions in the decoder up to that position. This **masked attention** preserves the autoregressive property, ensuring that the prediction only depends on those output tokens that have been generated.



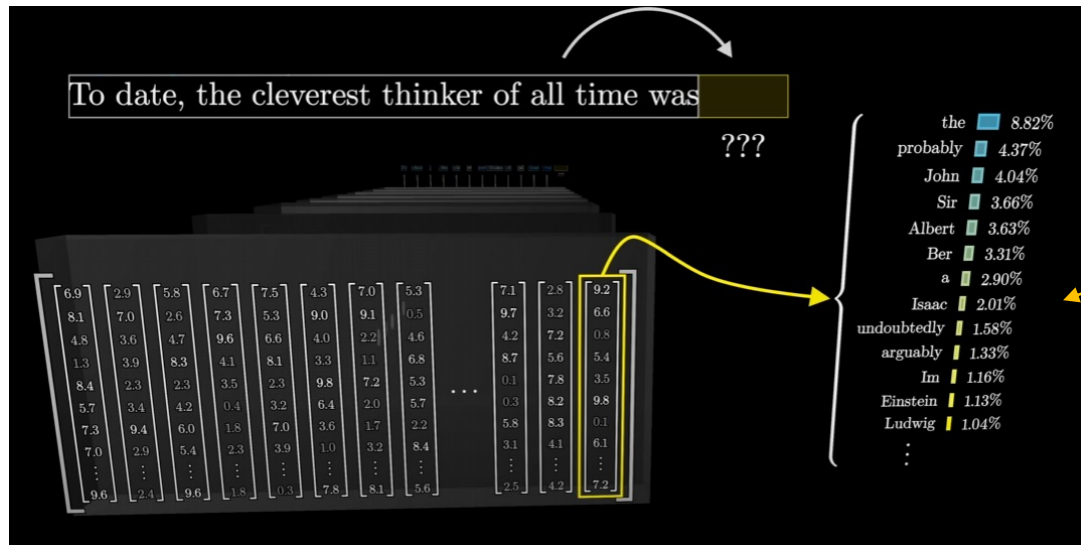
# Decoder





# Decoder: Final Layer

- The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.
- The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0).
- The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.



Which word in our vocabulary  
is associated with this index?

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

log\_probs



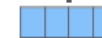
Softmax

logits



Linear

Decoder stack output



am

5

# Transformer: Putting it All Together

A Transformer is a sequence of Transformer layers.

multi-head attention keys and values  
 $k_{t,1}^\ell, \dots, k_{t,m}^\ell$  and  $v_{t,1}^\ell, \dots, v_{t,m}^\ell$

6 layers, each with  $d = 512$

$\bar{h}_t^\ell = \text{LayerNorm}(\bar{a}_t^\ell + h_t^\ell)$   
passed to next layer  $\ell + 1$

$$h_t^\ell = W_2^\ell \text{ReLU}(W_1^\ell \bar{a}_t^\ell + b_1^\ell) + b_2^\ell$$

2-layer neural net at each position

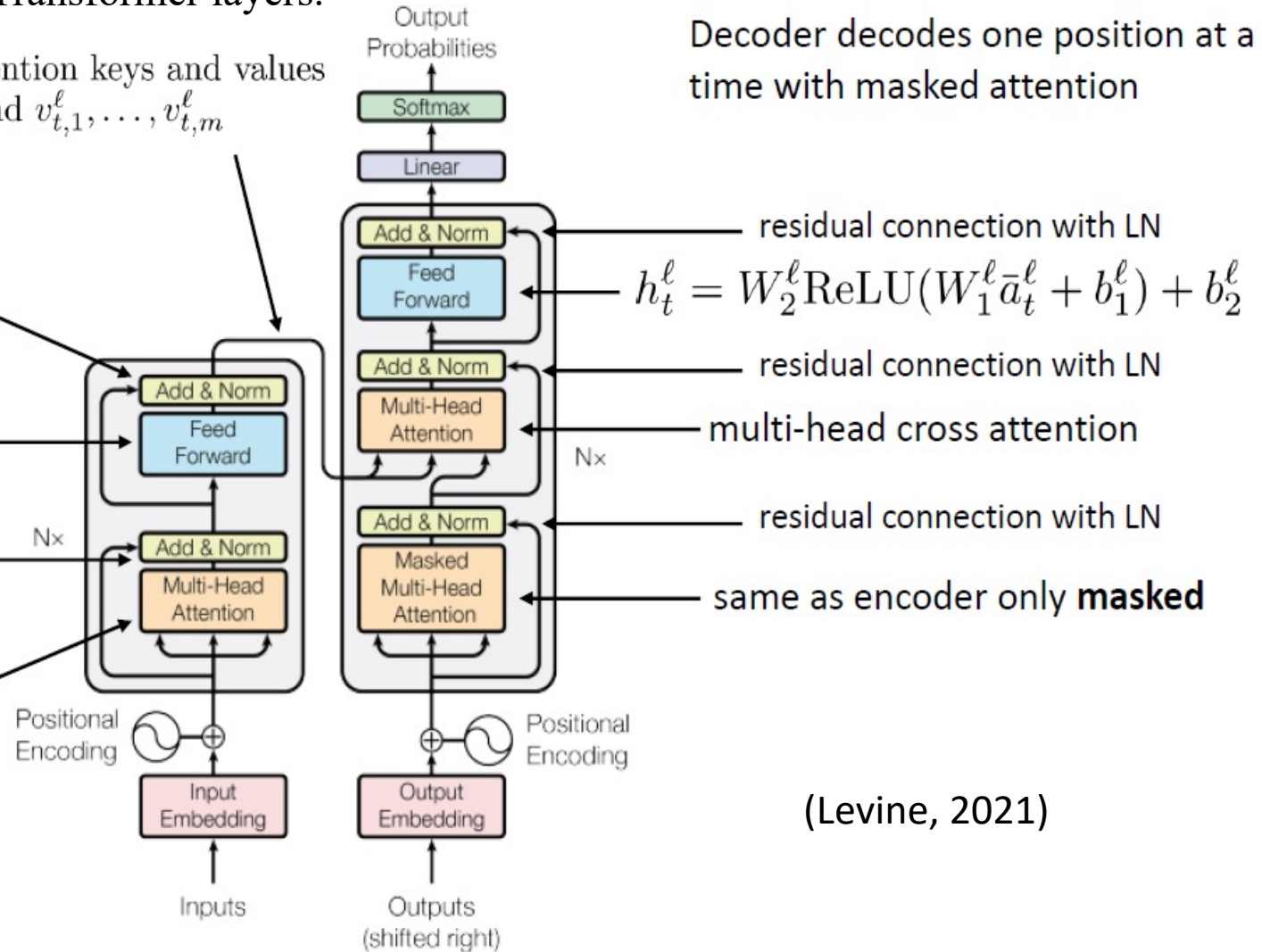
$$\bar{a}_t^\ell = \text{LayerNorm}(\bar{h}_t^{\ell-1} + a_t^\ell) -$$

essentially a residual connection with LN

input:  $\bar{h}_t^{\ell-1}$   
output:  $a_t^\ell$

concatenates attention from all heads

Decoder decodes one position at a time with masked attention



(Levine, 2021)

# Content

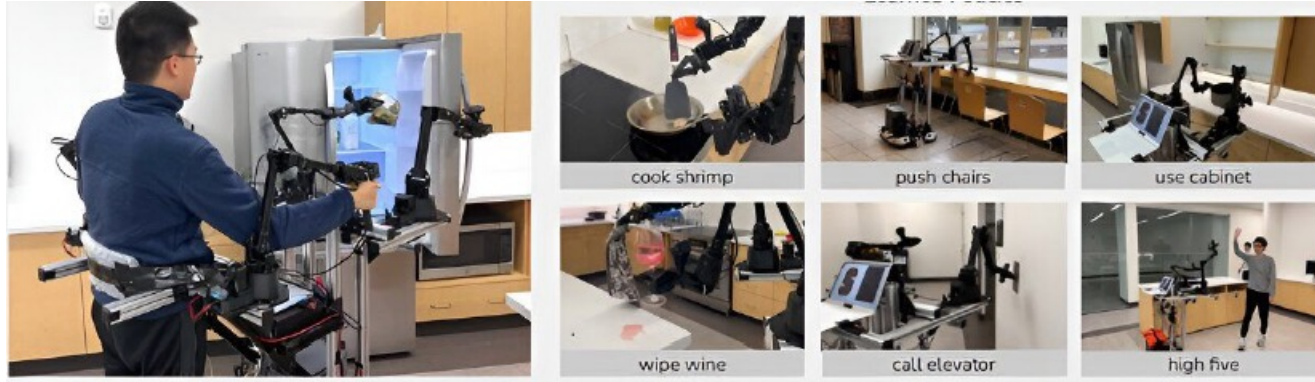
1 Attention Mechanisms

2 Self-Attention and Positional Encoding

3 Transformer Architecture

**4 Transformer Applications**

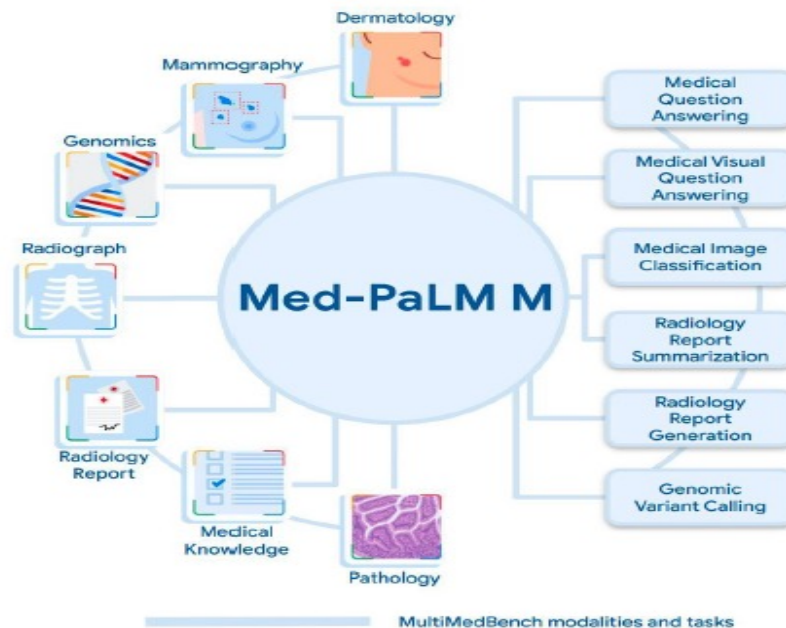
# Transformers are Everywhere Now!



Robotics, Simulations, Physical Tasks



Playing Games

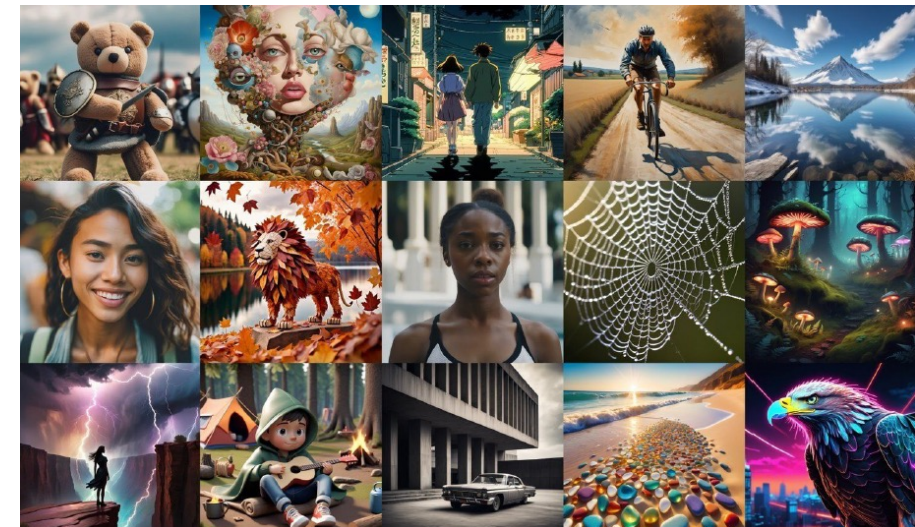
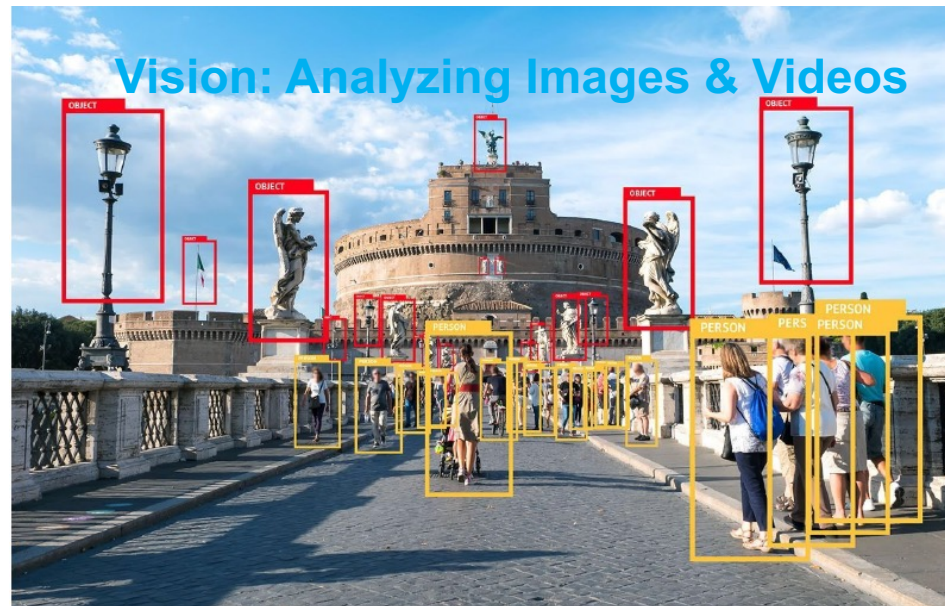
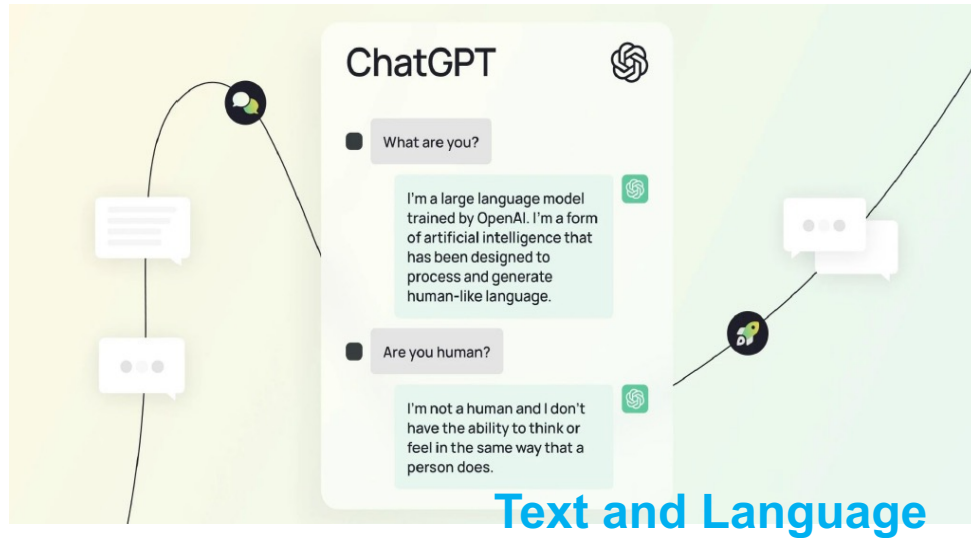


Biology + Healthcare





# Transformers are Everywhere Now!



# Why Transformers?

- **Downsides:**

- Attention computations are technically  $O(N^2)$
- Somewhat more complex to implement (positional encodings, etc.)

- **Benefits:**

- Much better long-range connections
  - Much easier to parallelize
  - In practice, can make it much deeper than RNN.
- The benefits seem to **vastly** outweigh the downsides, and Transformers work much better than RNNs and LSTMs in many cases. Arguably, Transformer is one of the most important sequence modeling improvements of the past decade.

# Pretraining and Fine-Tuning

**Definition:** Train a model on a large, general-purpose dataset.

## Objective:

- ▶ Capture grammar, semantics, and world knowledge.
- ▶ Develop universal language representations.

## Benefits:

- ▶ Model gains broad patterns (e.g., BERT, GPT, etc.).
- ▶ Reduces the amount of data needed for future tasks.
- ▶ Often uses large corpora (Wikipedia, BookCorpus, etc.).

## Examples:

- ▶ Masked language modeling (BERT).
- ▶ Next token prediction (GPT).

**Definition:** Further training a pretrained model on a smaller, task-specific dataset.

## Goal:

- ▶ Leverage general knowledge from pretraining.
- ▶ Specialize for a target task (classification, QA, NER, etc.).

## Advantages:

- ▶ Requires far less data than training from scratch.
- ▶ Faster convergence, lower computational cost.
- ▶ Often leads to state-of-the-art performance on downstream tasks.

## Process:

- ▶ Load pretrained weights, replace final layer with task-specific output.
- ▶ Train on the smaller labeled dataset for a few epochs.

# The Rise of Large Language Models (LLM)

- Scaled up versions of Transformer architecture, e.g. billions/trillions of parameters
- Typically trained on massive amounts of “general” textual data (e.g. web corpus)
- Training objective is typically “next token prediction”:  $P(W_{t+1}|W_t, W_{t-1}, \dots, W_1)$
- Emergent abilities as they scale up (e.g. chain-of-thought reasoning)
- Heavy computational cost (time, money, GPUs)
- Larger general ones: “plug-and-play” with few or zero-shot learning
  - Train once, then adapt to other tasks without needing to retrain
  - E.g. in-context learning and prompting
- Why do LLMs work so well? What happens as you scale up?
- Potential explanation: emergent abilities!
- An ability is emergent if it is present in larger but not smaller models
- Not have been directly predicted by extrapolating from smaller models
- Performance is near-random until a certain critical threshold, then improves heavily



Gemini / Bard  
(Google)



ChatGPT / GPT-4  
(OpenAI)



Claude 3  
(Anthropic)



Llama 3  
(Meta)



# Scaling up Transformers

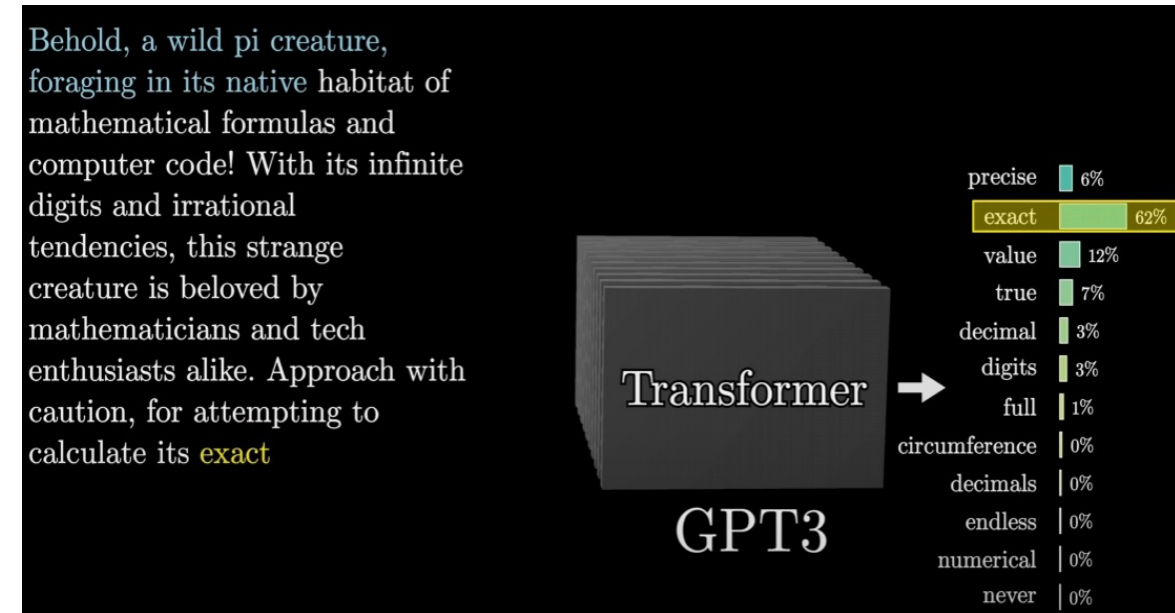
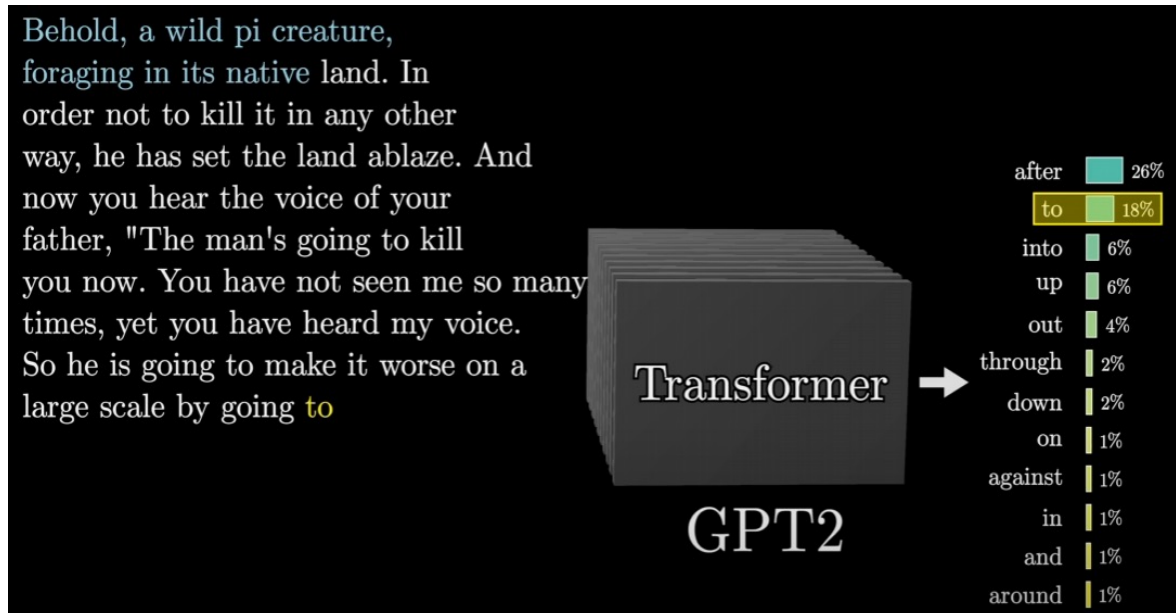
Scaling up Transformers **\$3,768,320 on Google Cloud (eval price)**

Model	Layers	Width	Heads	Params	Data	Training
Transformer-Base	12	512	8	65M		8x P100 (12 hours)
Transformer-Large	12	1024	16	213M		8x P100 (3.5 days)
BERT-Base	12	768	12	110M	13 GB	
BERT-Large	24	1024	16	340M	13 GB	
XLNet-Large	24	1024	16	~340M	126 GB	512x TPU-v3 (2.5 days)
RoBERTa	24	1024	16	355M	160 GB	1024x V100 GPU (1 day)
GPT-2	48	1600	?	1.5B	40 GB	
Megatron-LM	72	3072	32	8.3B	174 GB	512x V100 GPU (9 days)
Turing-NLG	78	4256	28	17B	?	256x V100 GPU
GPT-3	96	12,288	96	175B	694GB	?
Gopher	80	16,384	128	280B	10.55 TB	4096x TPUv3 (38 days)

(Johnson, 2022)

# Scaling Laws & Beyond Scaling

- With Transformers, language modeling performance improves smoothly as we increase model size, training data, and compute resources in tandem.
- This power-law relationship has been observed over multiple orders of magnitude with no sign of slowing!
- While scaling is a factor in emergent abilities, it is not the only factor! E.g. new architectures (DeepSeek, as discussed later), higher-quality data, and improved training procedures, could enable emergent abilities on smaller models

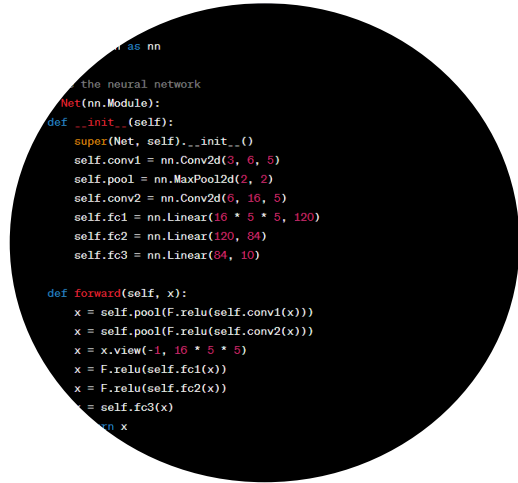


<https://www.youtube.com/watch?v=eMlx5fFNoYc>

# References

- Ansar, W., Goswami, S., & Chakrabarti, A. (2024). A Survey on Transformers in NLP with Focus on Efficiency. *arXiv preprint arXiv:2406.16893*.
- Alammar, J. (2018). *The Illustrated Transformer*. Retrieved from <https://jalammar.github.io/illustrated-transformer/>
- Bertasius, G. (2024). *Visual Recognition with Transformers* [Lecture slides]. COMP 590/790: Visual Recognition with Transformers, Spring 2024. University of North Carolina at Chapel Hill. Retrieved from <https://uncch.instructure.com/courses/49024>
- Brock, A., De, S., Smith, S. L., & Simonyan, K. (2021). High-performance large-scale image recognition without normalization. *International Conference on Machine Learning* (pp. 1–10). PMLR.
- Feng, S., Garg, D., Bunnapradist, E., & Lee, S. (2024). *Overview of Transformers* [Lecture slides]. CS25: Transformers United V4, Spring 2024. Stanford University. Retrieved from <https://web.stanford.edu/class/cs25/>
- Gao, C., Cao, Y., Li, Z., He, Y., Wang, M., Liu, H., ... & Fan, J. (2024). Global convergence in training large-scale transformers. *Advances in Neural Information Processing Systems*, 37, 29213-29284.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* (Vol. 196). MIT Press.
- Johnson, J. (2022). *Attention* [Lecture slides]. EECS 498.008 / 598.008: Deep Learning for Computer Vision, Winter 2022. University of Michigan. Retrieved from <https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/schedule.html>
- Jurafsky, D., & Martin, J. H. (2025). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition with language models* (3rd ed.). Online manuscript released January 12, 2025.
- Levine, S. (2021). *Transformers* [Lecture slides]. CS W182 / 282A: Designing, Visualizing and Understanding Deep Neural Networks, Spring 2021. University of California, Berkeley. Retrieved from <https://cs182sp21.github.io/>
- Lin, T., Wang, Y., Liu, X., & Qiu, X. (2022). A survey of transformers. *AI open*, 3, 111-132.
- Prince, S. J. D. (2023). *Understanding deep learning*. MIT Press.
- Szalata, A., Hrovatin, K., Becker, S., Tejada-Lapuerta, A., Cui, H., Wang, B., & Theis, F. J. (2024). Transformers in single-cell omics: a review and new perspectives. *Nature methods*, 21(8), 1430-1443.
- Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems*, 27.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30.
- Yang, D., & Hashimoto, T. (2025). *Transformers* [Lecture slides]. CS224N: Natural Language Processing with Deep Learning, Winter 2025. Stanford University. Retrieved from <https://web.stanford.edu/class/cs224n/>
- Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2021). Dive into deep learning. *arXiv preprint arXiv:2106.11342*.

# How to succeed in this course?



**Practice**



**Explore**



**Visualize**

**Discuss**



**Ask**

