Bios 740- Chapter 7. Transformer and Attention Mechanism

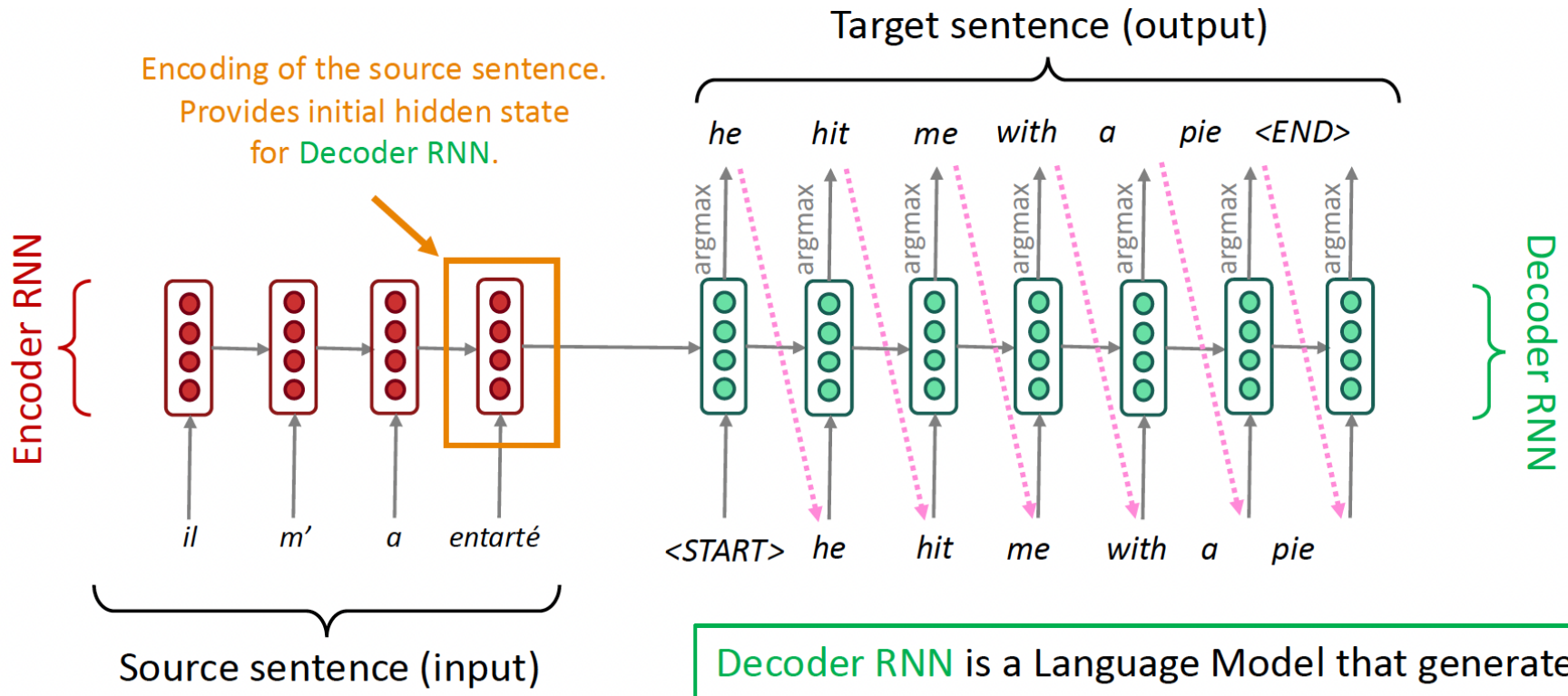
Acknowledgement: Many thanks to Mr. Mingchen Hu for preparing some of these slides and to Dr. Xiao Wang for sharing his slides. I also drew on material from the lecture presentations of Stanford CS224n, UNC COMP 590/790, and content generated by ChatGPT.

Content

Content

The Sequence-to-sequence Model

Neural Machine Translation



Encoder RNN produces an encoding of the source sentence.

Decoder RNN is a Language Model that generates target sentence, *conditioned on encoding*.

Note: This diagram shows **test time** behavior: decoder output is fed in ⋯→ as next step's input

Stanford CS224n

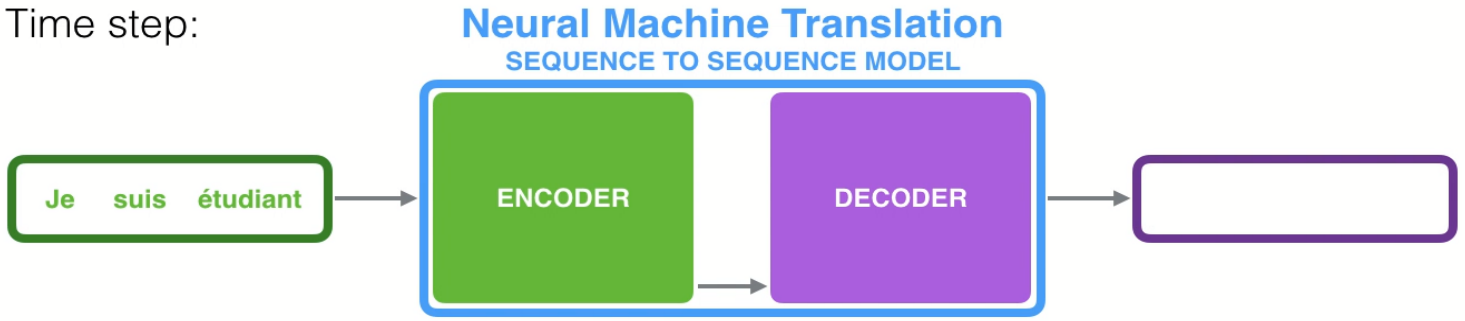
Sequence-to-sequence Model

Sequence-to-Sequence Overview

- A general encoder-decoder architecture.
- **Encoder:** maps input to a neural representation.
- **Decoder:** generates output from the neural representation.
- When both input and output are sequences: seq2seq model.

Seq2Seq Applications: Sequence-to-sequence is versatile!

- ❖ **Machine Translation:** French --->English
- ❖ **Summarization:** long text ----> concise summary
- ❖ **Dialogue:** previous utterances ---> next response
- ❖ **Parsing:** input text---> parse tree (as a sequence)
- ❖ **Code Generation:** natural language --> Python code



In the visualization, each pulse from the encoder or decoder represents an RNN processing its input and updating its hidden state based on current and past inputs.

https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Neural Machine Translation

Sequence-to-Sequence as a Conditional Language Model

Like any language model, the decoder predicts the next word in the target sequence Y.

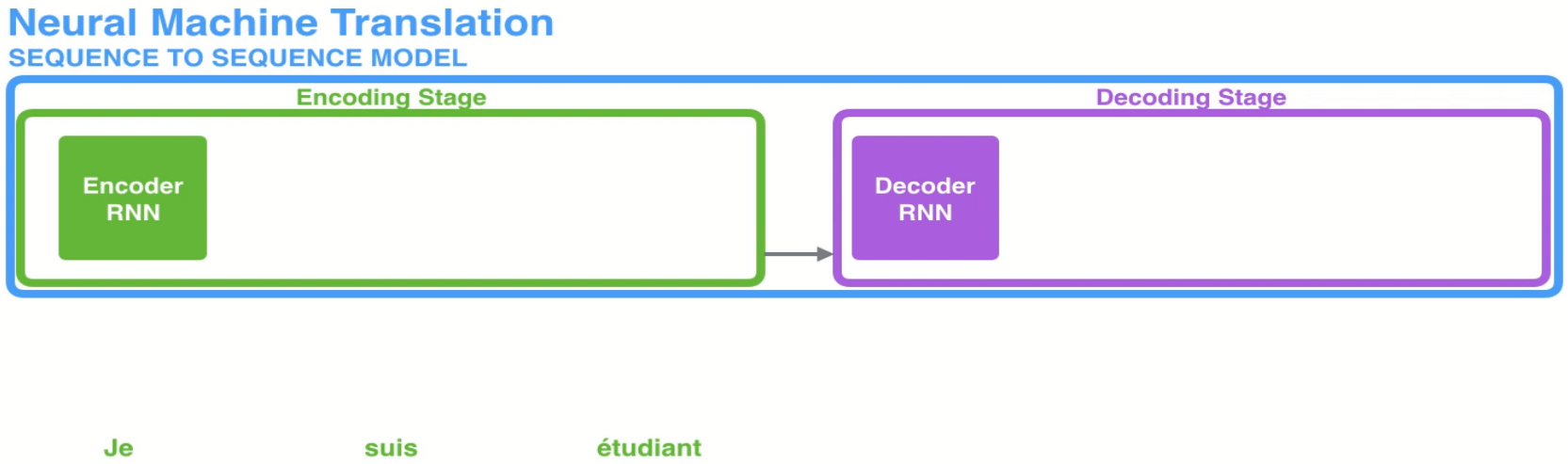
It is conditional because predictions are made based on both:

- Previously generated target tokens.
- The full source sequence X (via the encoder).

This allows the model to generate context-aware translations or responses.

$$P(y|x) = P(y_1|x)\,P(y_2|y_1,x)\,P(y_3|y_1,y_2,x)\ldots\underbrace{P(y_T|y_1,\ldots,y_{T-1},x)}$$

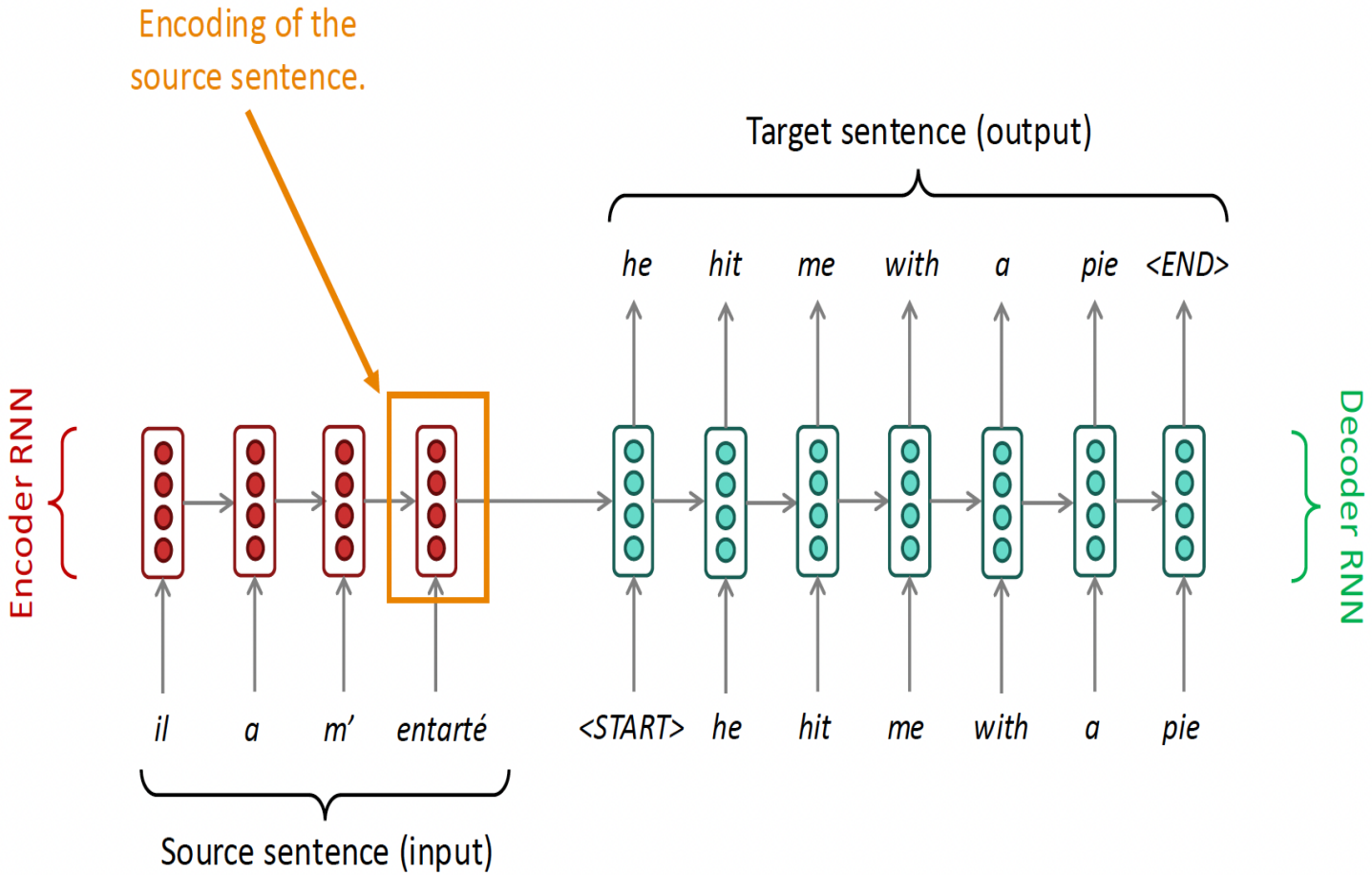
Probability of next target word, given
rget words so far and source sentence *x*



The context vector turned out to be a bottleneck for these types of models. It made it challenging for the models to deal with long sentences.

https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Limitations of RNNs for Long Sequences



Stanford CS224n

Problems with this architecture?

Long-distance dependencies are hard to learn:

- ❖ Distant word pairs require O sequence length steps to interact.
- ❖ Leads to vanishing or exploding gradients.

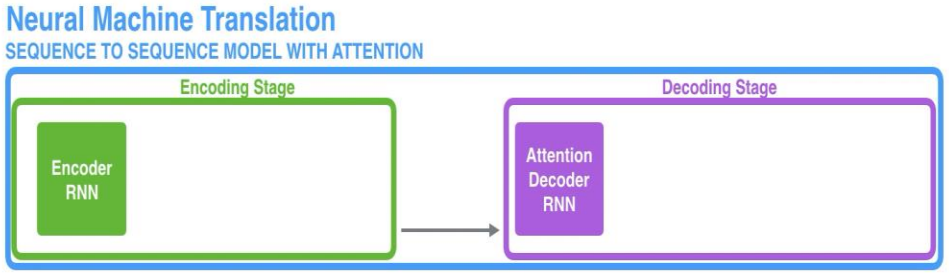
RNNs are inherently sequential:

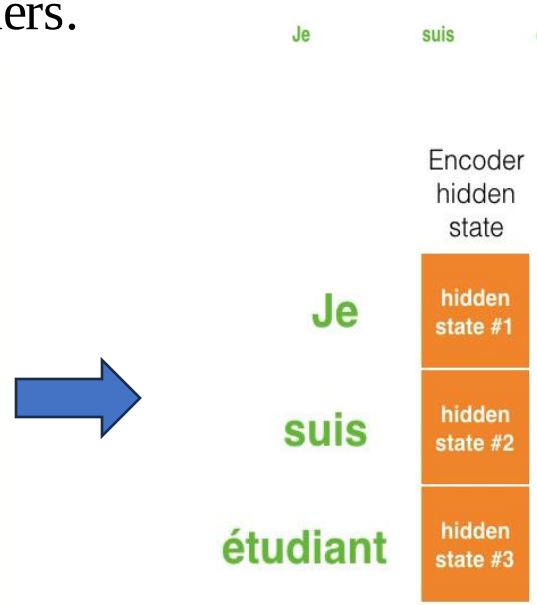
- ➢ Forward/backward passes require O(sequence length) dependent operations.
- ➢ Each hidden state of RNN must be computed in order.

These challenges motivated the development of **attention-based and transformer** models.

Attention: A Key Breakthrough

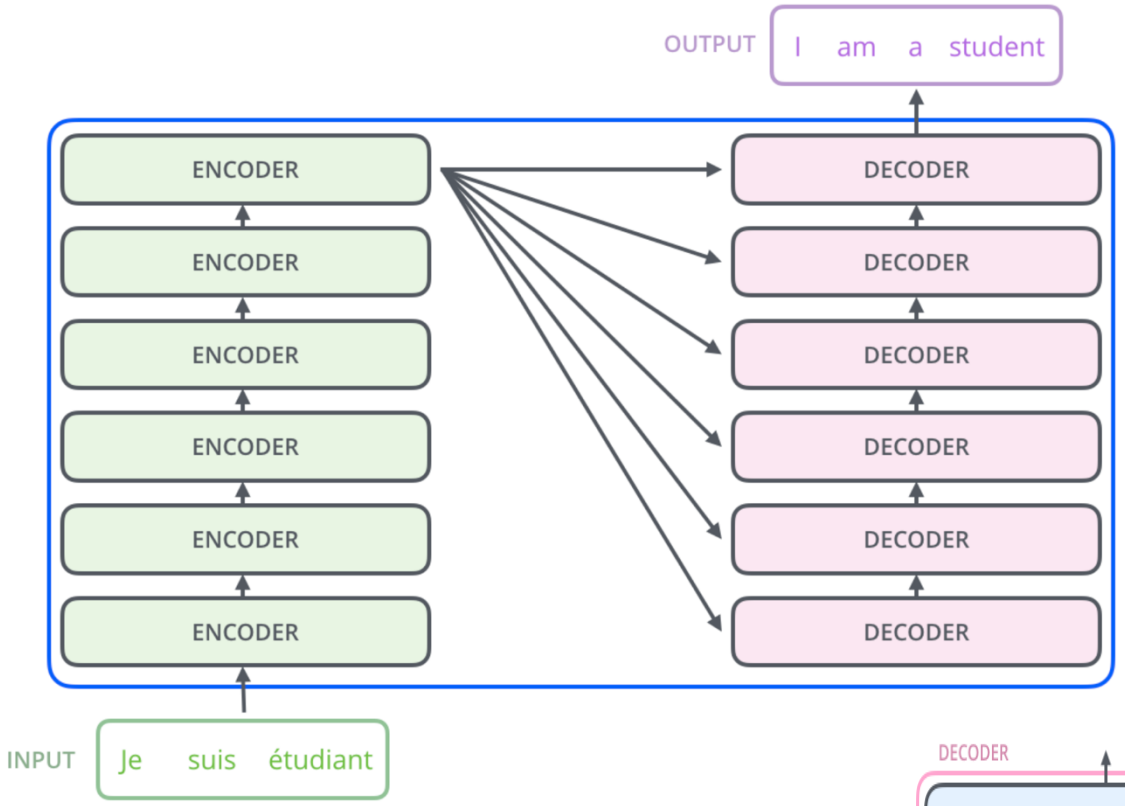
A major solution was proposed in Bahdanau et al. (2014) and refined by Luong et al. (2015). They introduced a mechanism called Attention. Attention allows the model to focus on the most relevant parts of the input sequence when generating each output token. This led to substantial improvements in machine translation systems. It laid the groundwork for modern architectures like Transformers.

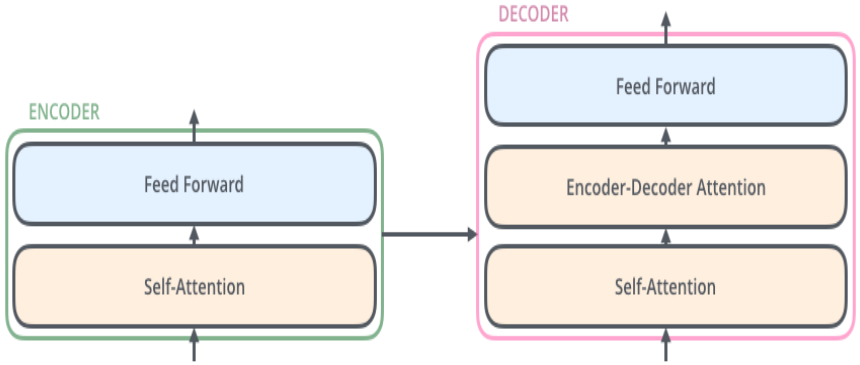




https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Transformer: Encoder and Decoder Layers





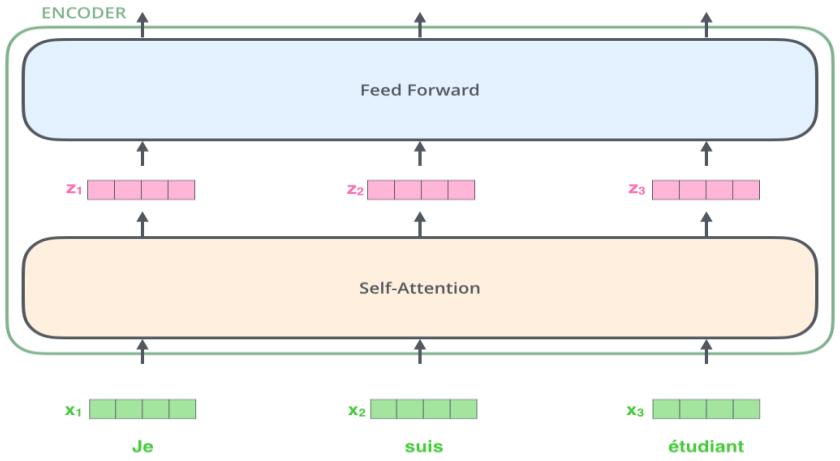
Encoder:

- Inputs pass through a self-attention layer, allowing each token to attend to others in the sequence.
- Followed by a **feed-forward neural network**, applied identically to each position.

Decoder:

- Contains a self-attention layer and a feed-forward layer.
- Additionally includes **an encoder-decoder attention layer** to focus on relevant encoder outputs.

This design enables both dynamic focus and parallelization.



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

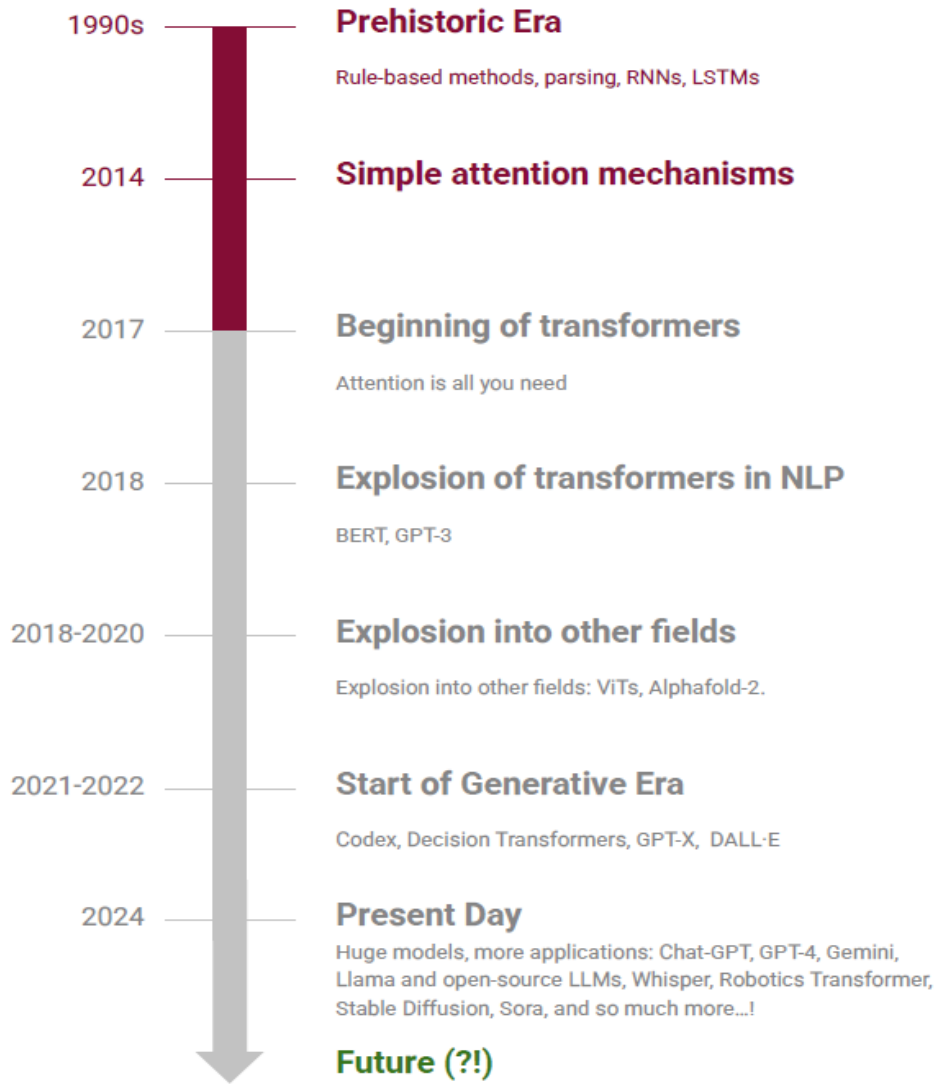
Content

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.

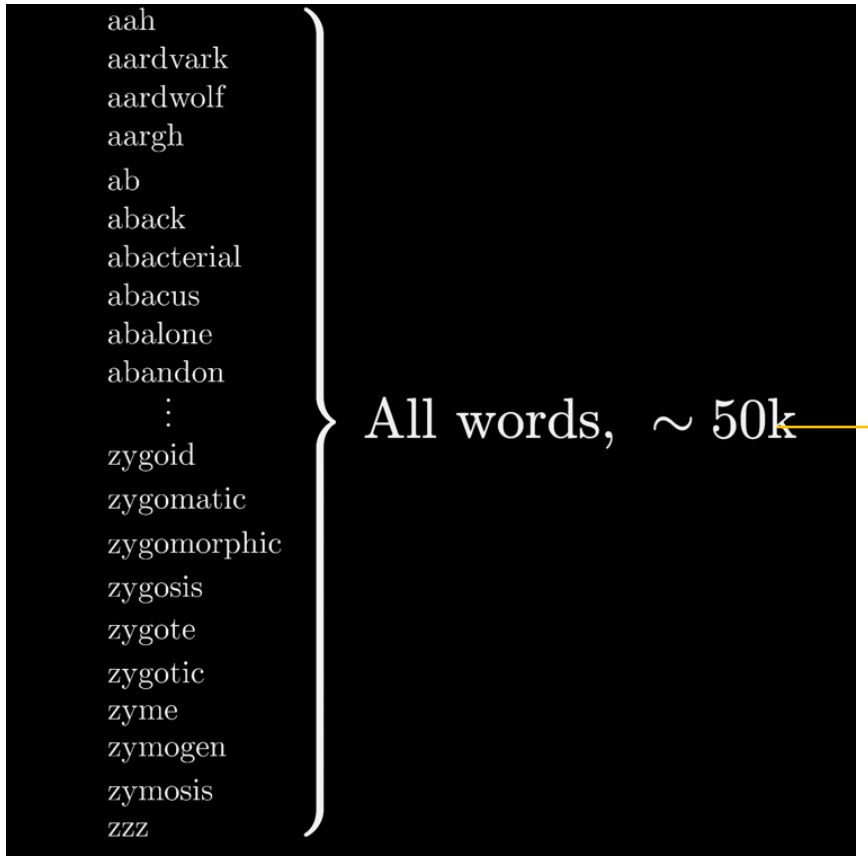
By spring 2025, Transformers power massive models like ChatGPT, DeepSeek and open new applications in diverse areas, with exciting prospects for the future.

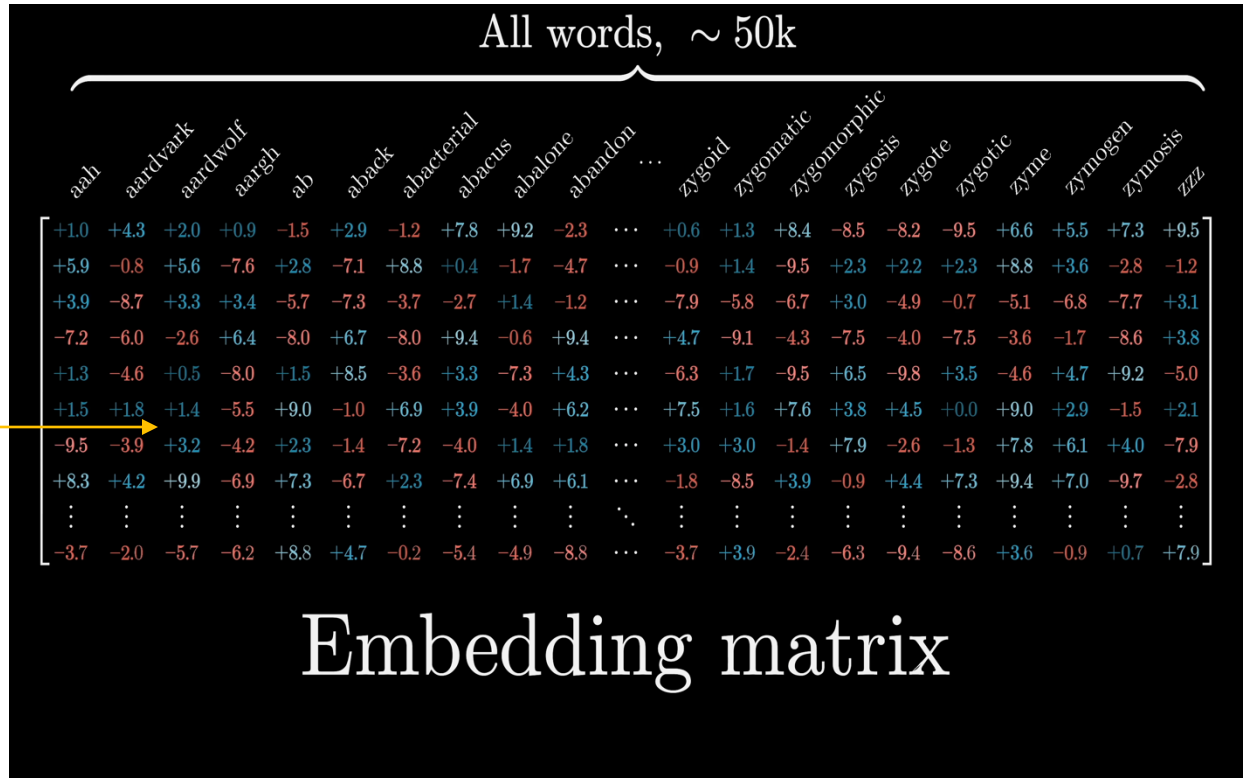


(Yang & Hashimoto, 2025)

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.

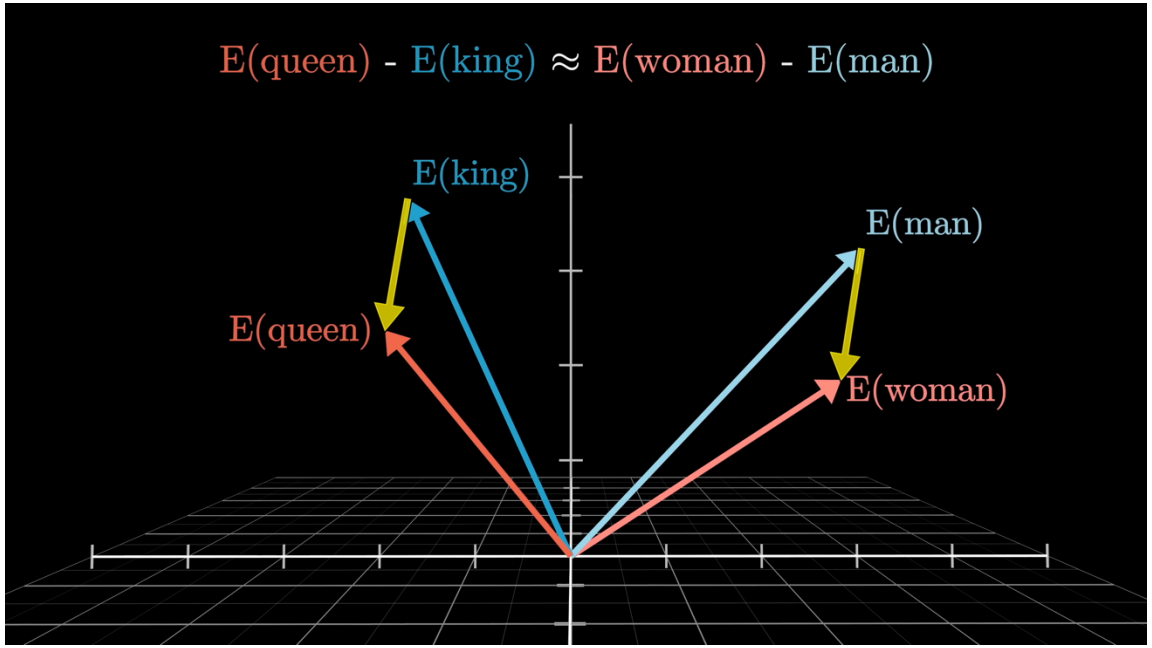


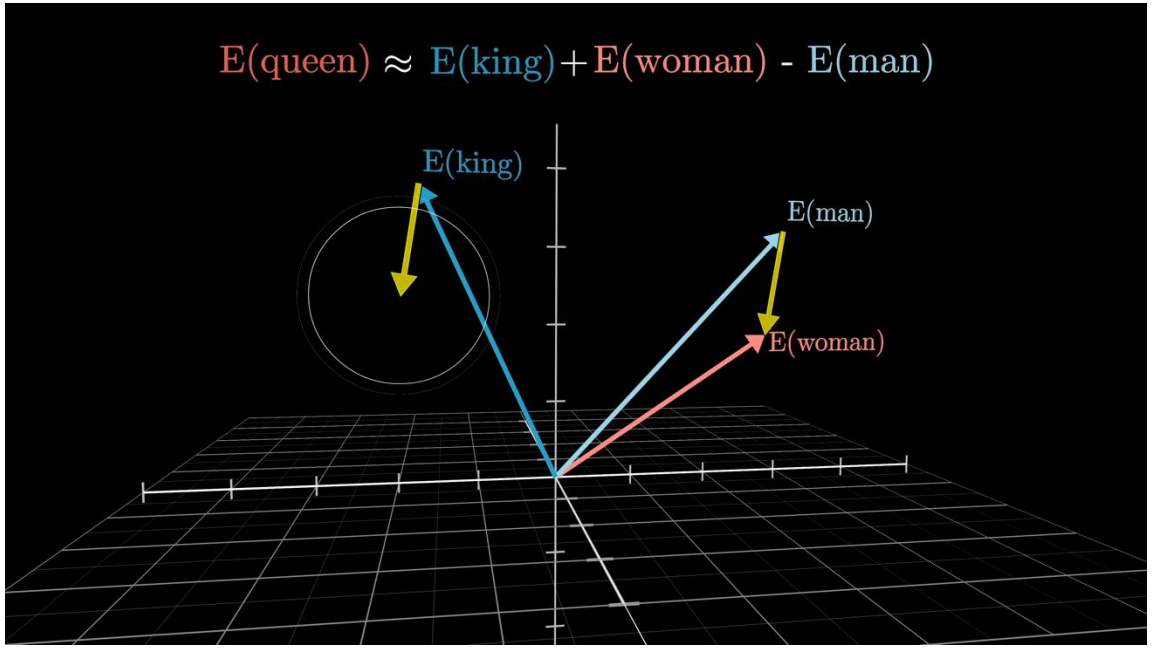


https://www.youtube.com/watch?v=wjZofJX0v4M

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

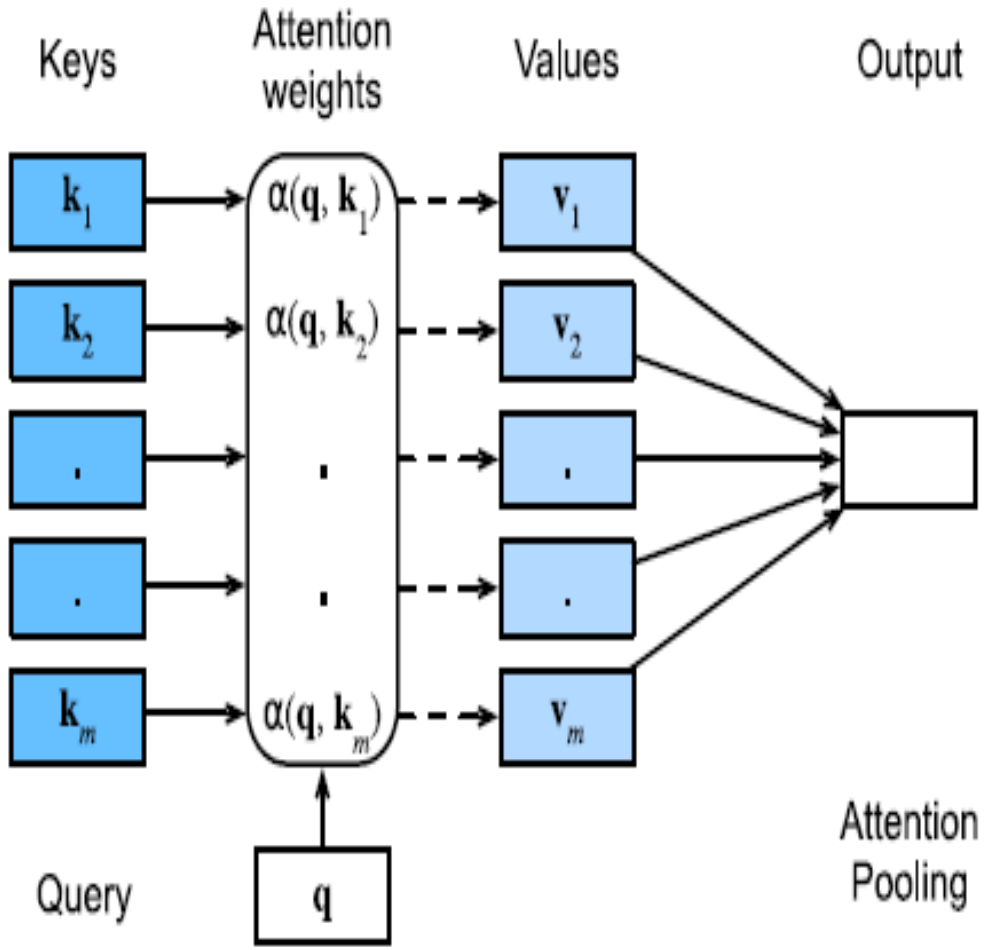
Queries, Keys and Values

- Define by $D = \{(k_1, v_1), \ldots, (k_m, v_m)\}$ a dataset of m tuples of ***keys*** and ***values*** and denote q a ***query***. Then we can define the ***attention*** over D as $\boldsymbol{Attention}(q, D) = \sum_{i=1}^{m} a(q, k_i)v_i$ where $a(q, k_i)v_i \in \mathbb{R}$ are scalar nonnegative attention weights.
- The name **"attention"** derives from the fact that operation pays particular attention to the terms for which the weight α is significant. As such, the attention over D generates a linear combination of values contained in the database.
- To ensure that weights $\alpha(q, k_i)$ sum up to 1, one can normalize them via $\alpha(q, k_i) = \frac{\alpha(q,k_i)}{\sum_j \alpha(q,k_j)}$.

  To further ensure that weights are also nonnegative, one can resort to the exponentiation via **softmax operation** $\alpha(q, k_i) = \frac{\exp(a(q,k_i))}{\sum_j \exp(a(q,k_j))}$.

Queries, Keys and Values: Analogy

- Imagine you're looking for information on a specific topic (query) in a library system.
- Each book in the library has a summary (key) that helps identify if it contains the information you're looking for.
- Once you find a match between your query and a summary, you access the book to get the detailed information (value) you need.
- Here in attention, we do a "soft-match" across multiple values. That is, we get into from multiple books ("Book1 is most relevant, then book2, the book3…")

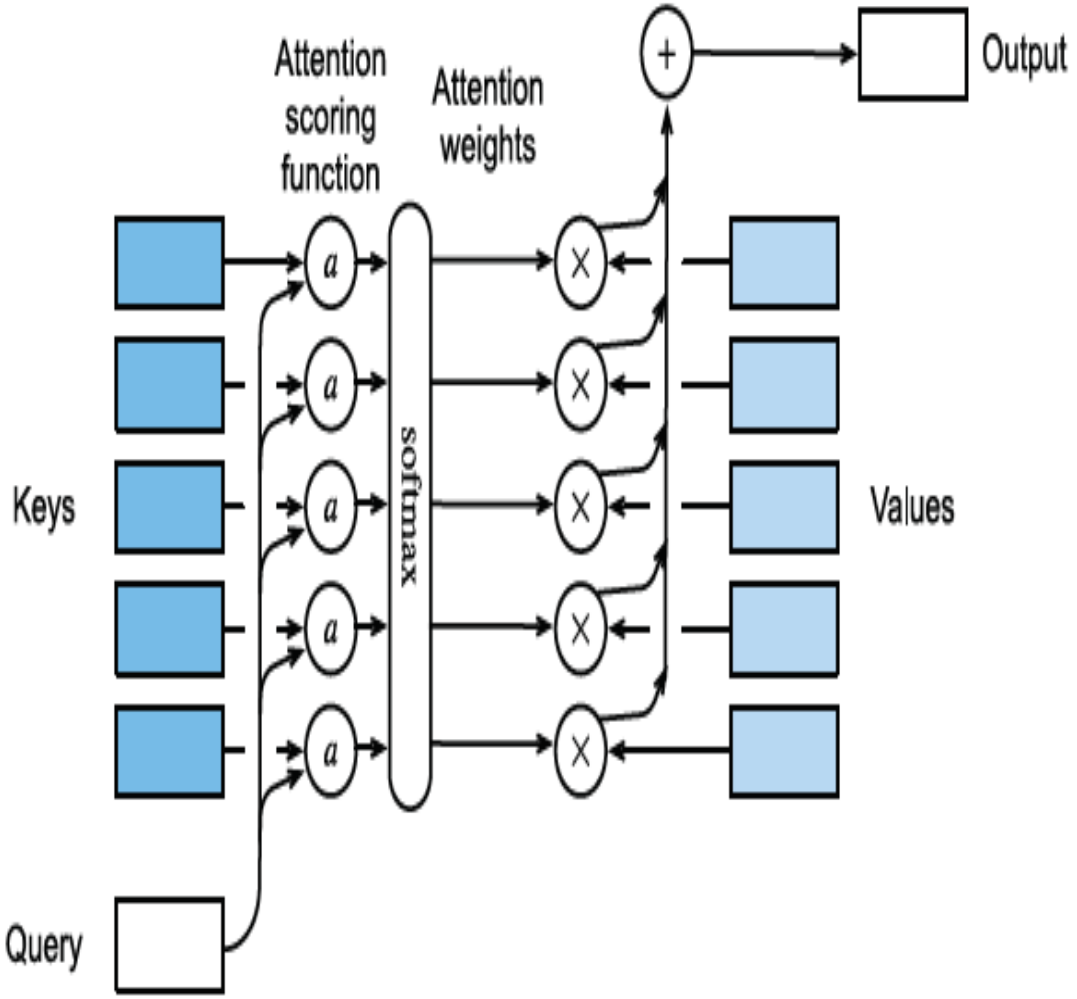


Attention Scoring Functions

- We refer to $a(q, k_i)$ before the softmax operation as the *attention scoring functions*. The mechanism Computing the output of attention pooling as a weighted average of values, where weights are computed with the attention scoring function a and the softmax operation.

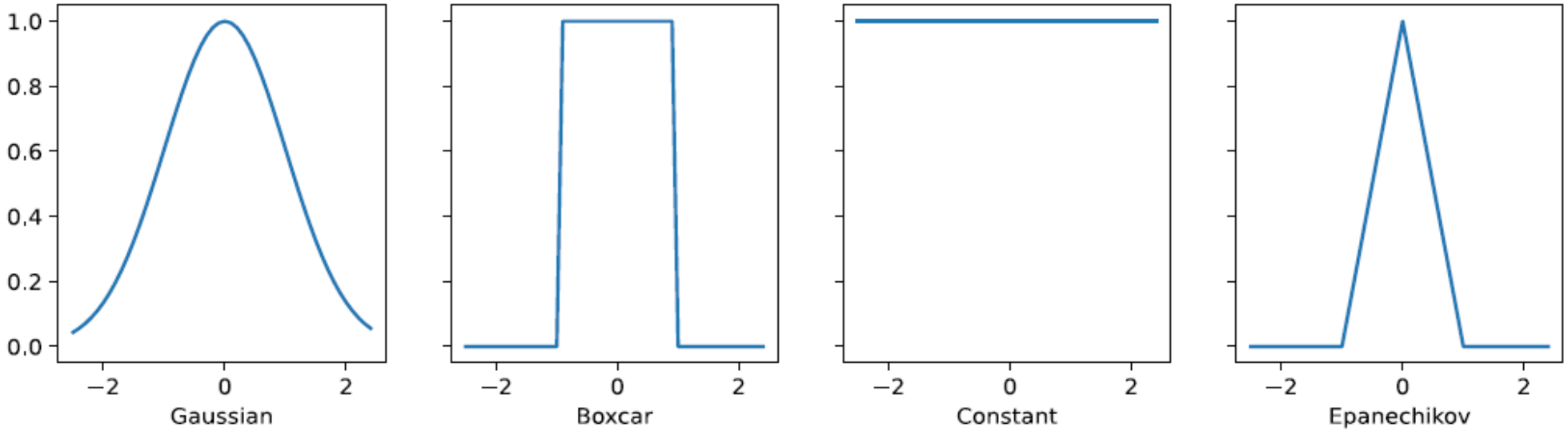
Examples include:

- Dot product: $a(q, k_i) = q^\top k_i$
- Scaled dot product: $a(q, k_i) = \frac{q^\top k_i}{\sqrt{d_k}}$
- Additive attention: $a(q, k_i) = v^\top \tanh(W_1 q + W_2 k_i)$



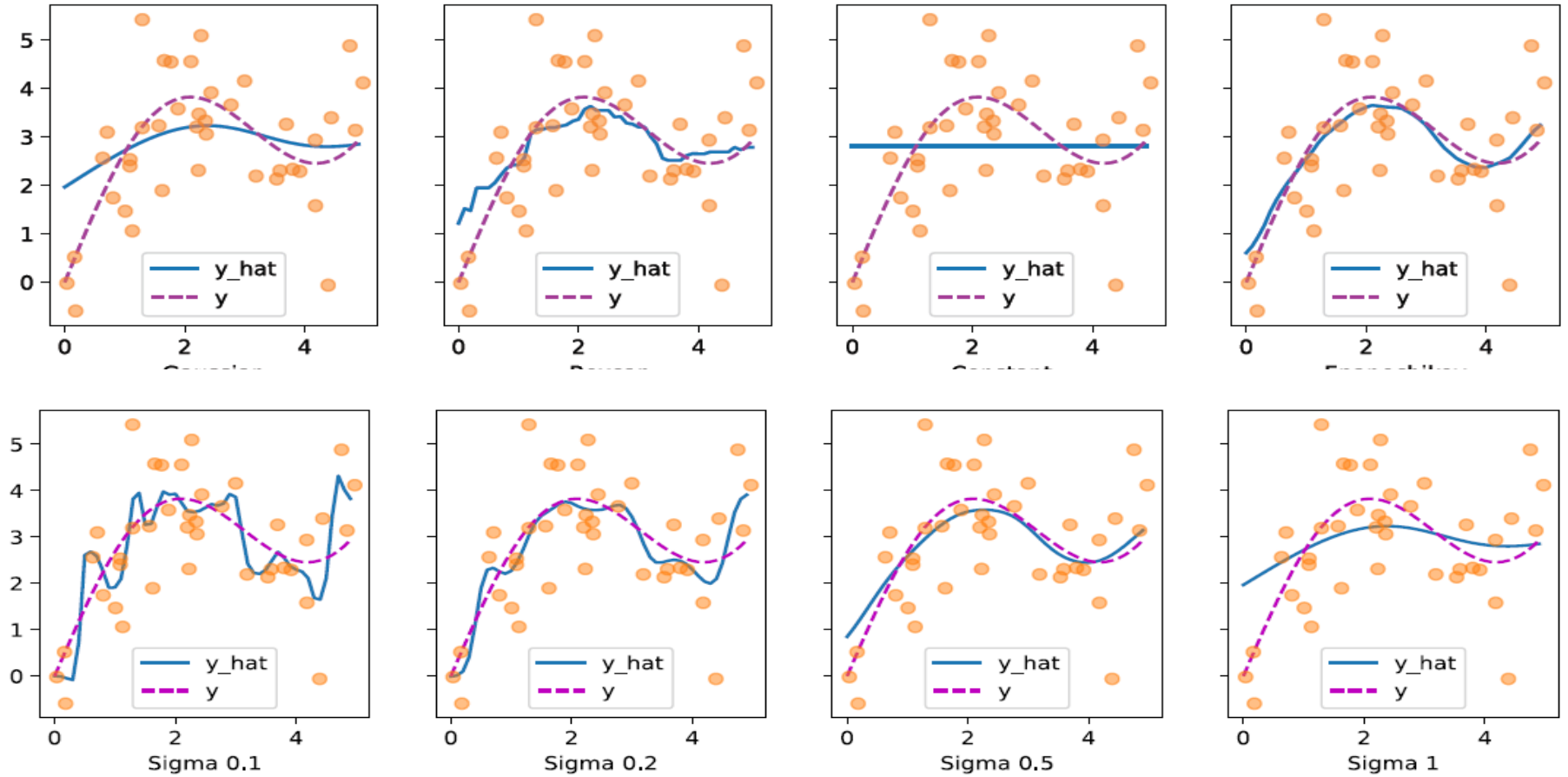
Nadaraya–Watson Kernel Estimator

- The similarity kernels have already been used in regression and classification via kernel density estimation.
- At their core, Nadaraya-Watson estimators rely on some similarity kernel $\alpha(q, k)$ relating queries q to keys k. Some common kernels include $a(q, k) = \exp(-\frac{1}{2}\left|\left|q - k\right|\right|^2)$ (Gaussian), $a(q, k) = 1\ if\ \left|\left|q - k\right|\right| \leq 1$ (Boxcar) and $a(q, k) = \max(0, 1 - ||q - k||)$ (Epanechikov).
- For example, we can pick key $k = 0$ and will yield following graphs:



Nadaraya–Watson Kernel Estimator

- All of these kernels lead to the equation for regression and classification alike: $f(q) = \sum_i \frac{a(q,k_i)}{\sum_j a(q,\mathrm{k_j})} \mathrm{v}_i$. Such Nadaraya-Watson kernel regression is an early precursor of the current attention mechanisms.



Attention Mechanisms: Overview

Given:

- n query vectors $q_1, \ldots, q_n \in \mathbb{R}^d$
- m key vectors $k_1, \ldots, k_m \in \mathbb{R}^d$
- m value vectors $v_1, \ldots, v_m \in \mathbb{R}^p$

The attention mechanism outputs n vectors $o_1, \ldots, o_n \in \mathbb{R}^q$ b

$$o_j = \frac{1}{C} \sum_{i=1}^{m} f(q_j, k_i) g(v_i)$$

where $f(q_j, k_i)$ measures similarity and $g(v_i)$ is a linear transformation.

- $f(q_j, k_i)$ is often the **embedded Gaussian similarity**:

$$f(q_j, k_i) = \exp\left(\theta(q_j)^\top \phi(k_i)\right)$$

- $\theta(q_j) = W_q q_j$, $\phi(k_i) = W_k k_i$
- $g(v_i) = W_v v_i$
- $C = \sum_{i=1}^{m} f(q_j, k_i)$ normalizes the scores

This setup lets each output vector depend on all input vectors.

Attention Mechanisms: Overview

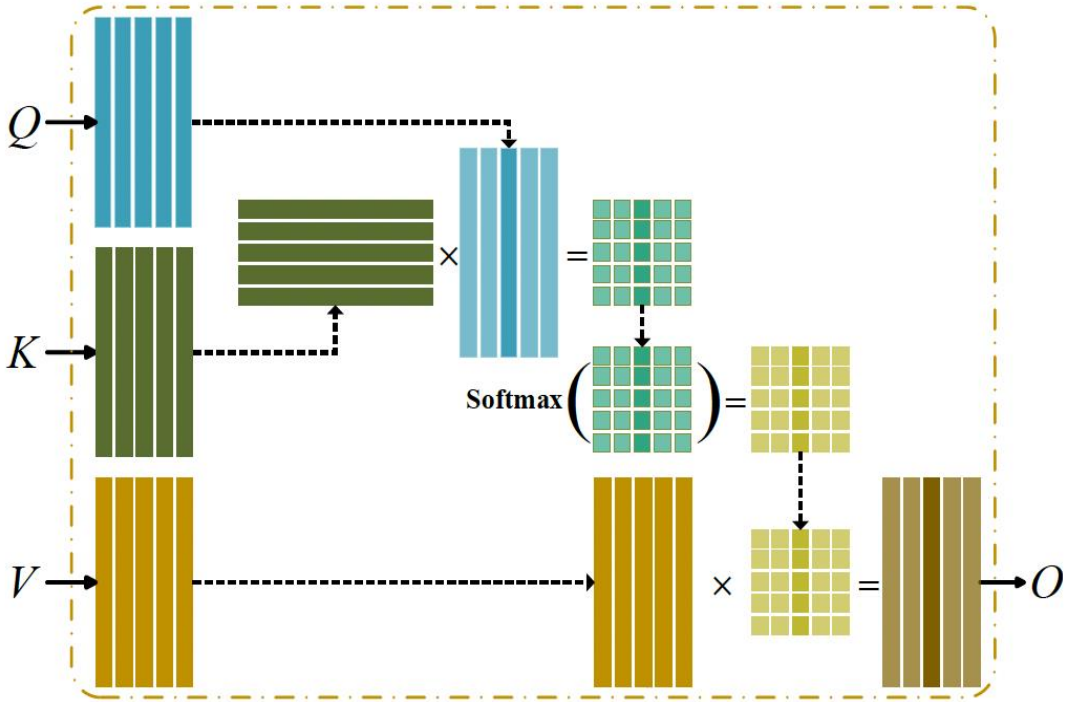
$$Q = [q_1, q_2, \ldots, q_n] \in \mathbb{R}^{d \times n}$$
$$K = [k_1, k_2, \ldots, k_m] \in \mathbb{R}^{d \times m}$$
$$V = [v_1, v_2, \ldots, v_m] \in \mathbb{R}^{p \times m}$$

$$O = W_v V \cdot \text{softmax}\left((W_k K)^\top (W_q Q)\right)$$

- softmax$(\cdot)$ normalizes similarity scores along each column.
- Ensures attention weights sum to 1 for each query.
- Output matrix $O = [o_1, \ldots, o_n] \in \mathbb{R}^{q \times n}$.
- Each o_j is a weighted combination of transformed value vectors.
- Output count matches number of queries.

Dimensions:

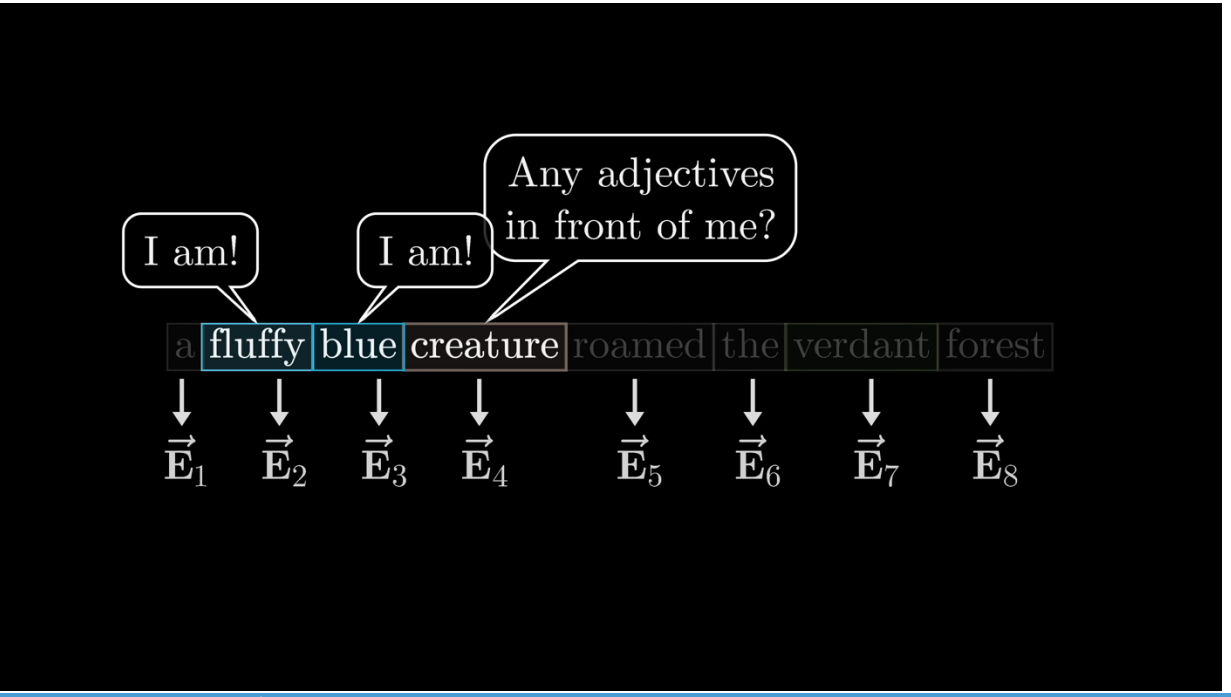
- $W_q \in \mathbb{R}^{r \times d}$ (queries mapped to $\mathbb{R}^r$)
- $W_k \in \mathbb{R}^{r \times d}$ (keys mapped to $\mathbb{R}^r$)
- $W_v \in \mathbb{R}^{q \times p}$ (values mapped to $\mathbb{R}^q$)

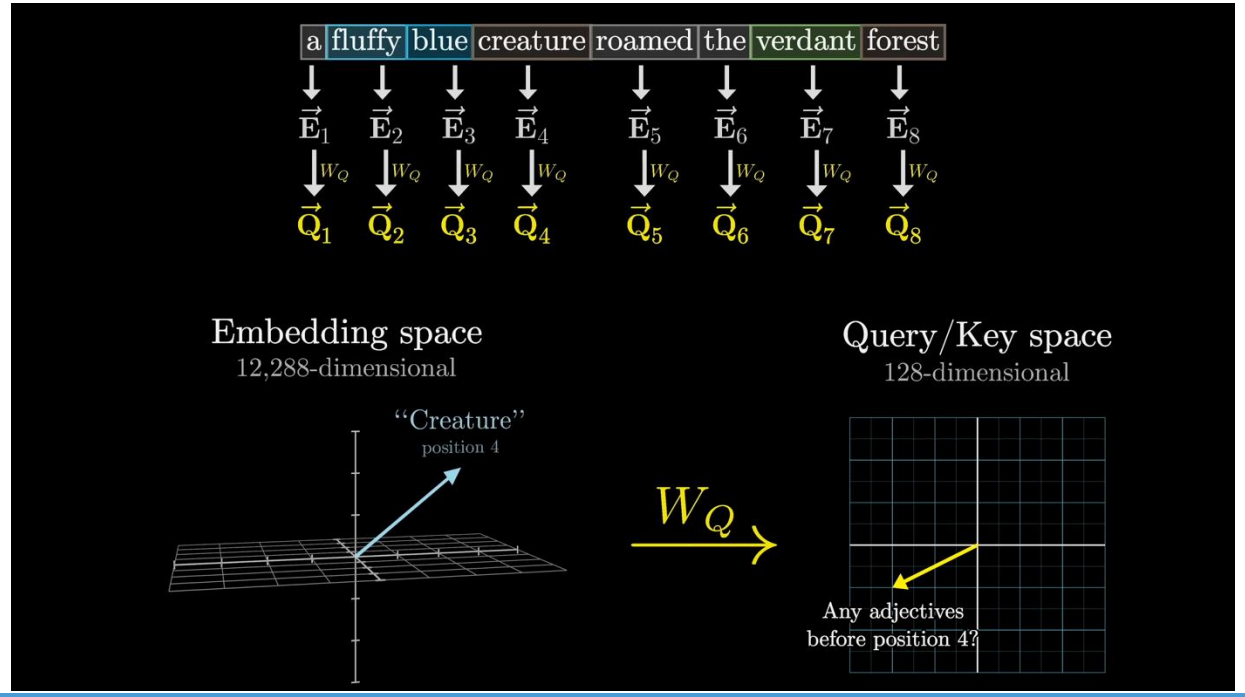


Query: Illustration

- Consider the sentence: *A fluffy blue creature roamed the verdant forest*.
- Imagine the situation, where each **noun** like "creature" is asking the question: "Hey, are there any **adjectives** sitting in front of me?
- Then the word "blue" would answer: "Yes, I am an adjective and I am in that position!"
- This question can be encoded in the *query vector*, which has a much smaller dimension than the embedding space.

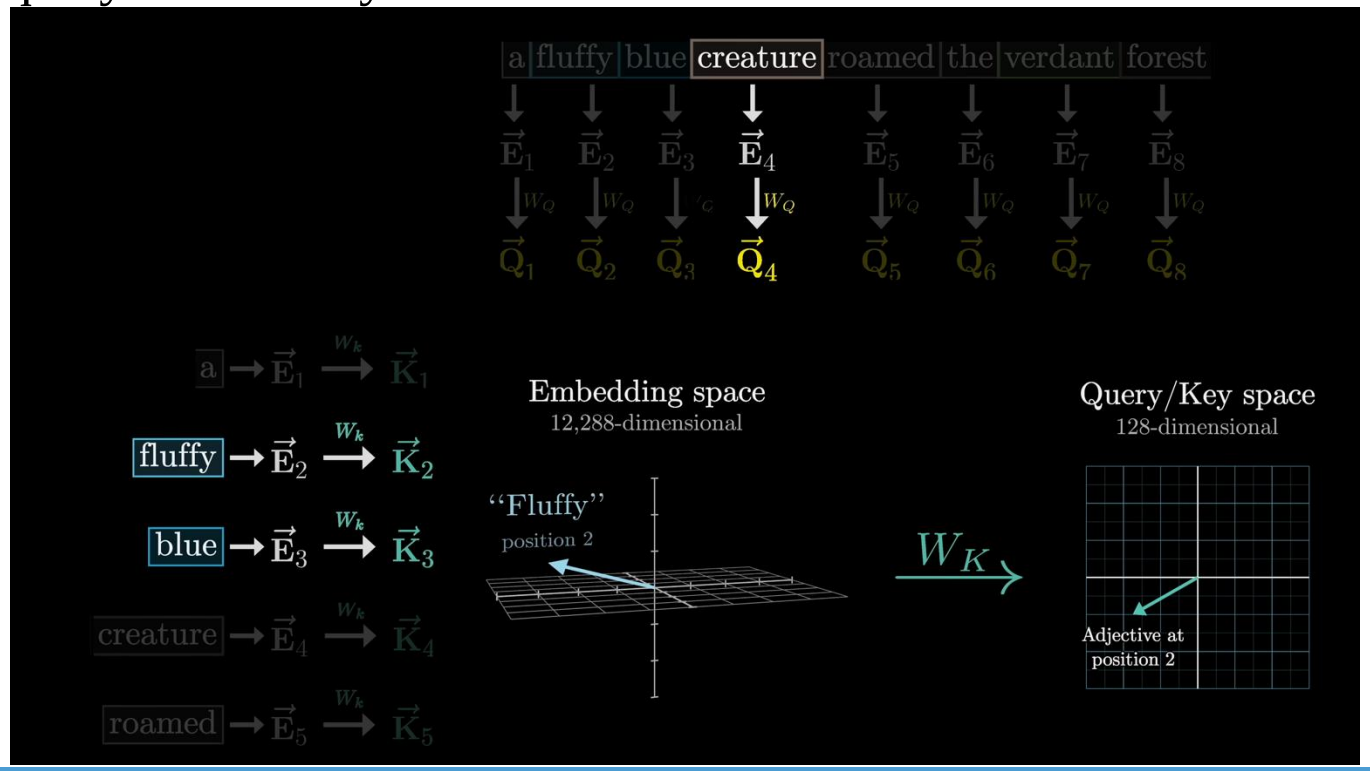
https://www.youtube.com/watch?v=eMlx5fFNoYc





Key: Illustration

- *Keys* can be interpreted as potentially answering the queries, which also has a much smaller dimension than the whole embedding space.
- As an example, the key matrix would map the adjectives like “fluffy” to vectors that are closely aligned with the query produced by the word “creature”, leading to some large positive dot products between the query and the key.



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

Value: Illustration

- Unlike query and key, *value vector* lives in the same, very high dimensional space as the embeddings.
- For example, here we add large proportions of the re-scaled value vectors for “fluffy” and “blue” to the original context-free embedding of “creature”, we get a refined vector that has contextually-rich meaning, like a “fluffy blue creature”.

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

Convenience Functions

- To deals with string of variable lengths more easily, one often pad shorter sequences with dummy tokens "$< blank >$".
- *Masked softmax operation*: Since we do not want the attention model to attend to these blanks, this operation limit $\sum_{i=1}^{n} \alpha(q, k_i) v_i$ to $\sum_{i=1}^{l} \alpha(q, k_i) v_i$ where $l \leq n$ is the length of actual sentence.

```python
def masked_softmax(X, valid_lens):
    """Perform softmax operation by masking elements on the last axis."""
    # X: 3D tensor, valid_lens: 1D or 2D tensor
    def _sequence_mask(X, valid_len, value=0):
        maxlen = X.size(1)
        mask = torch.arange((maxlen), dtype=torch.float32,
                            device=X.device)[None, :] < valid_len[:, None]
        X[~mask] = value
        return X

    if valid_lens is None:
        return nn.functional.softmax(X, dim=-1)
    else:
        shape = X.shape
        if valid_lens.dim() == 1:
            valid_lens = torch.repeat_interleave(valid_lens, shape[1])
        else:
            valid_lens = valid_lens.reshape(-1)
        # On the last axis, replace masked elements with a very large negative
        # value, whose exponentiation outputs 0
        X = _sequence_mask(X.reshape(-1, shape[-1]), valid_lens, value=-1e6)
        return nn.functional.softmax(X.reshape(shape), dim=-1)
```

Illustration: Consider a minibatch of two examples of size 2×4, where their valid lengths are 2 and 3 respectively:

```python
masked_softmax(torch.rand(2, 2, 4), torch.tensor([2, 3]))
```

```
tensor([[[0.4448, 0.5552, 0.0000, 0.0000],
         [0.4032, 0.5968, 0.0000, 0.0000]],

        [[0.2795, 0.2805, 0.4400, 0.0000],
         [0.2798, 0.3092, 0.4110, 0.0000]]])
```

Scaled Dot-Product Attention

- Scaled Dot-Product Attention is a key component of the Transformer architecture, widely used in modern deep learning models such as BERT and GPT. The formula is as follows:

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

- Scaled Dot-Product Attention possess the following key advantages:
  - **Efficiency:** It relies on simple matrix multiplications and scaling, making it highly efficient and suitable for parallel computation on GPUs/TPUs.
  - **Scalability:** It performs well on large-scale datasets and is the foundation of many state-of-the-art models.
  - **Flexibility:** It can handle sequences of different lengths, as the attention mechanism dynamically computes the relationships between queries and keys.

Why scale by $\sqrt{d_k}$?

- Elements of $q_i^\top k_j$ have variance d_k if q_i, k_j are independent with unit variance.
- Without scaling, large dot products push softmax into regions with small gradients.
- Scaling by $\sqrt{d_k}$ normalizes the variance:

$$\text{Var}(q_i^\top k_j) = d_k \Rightarrow \frac{q_i^\top k_j}{\sqrt{d_k}} \text{ has unit variance}$$

This stabilizes gradients during training and improves convergence.

Additive Attention (Bahdanau Attention)

- Additive Attention, also known as Bahdanau Attention, was introduced in the context of RNN-based sequence-to-sequence models. It computes the attention weights by passing the concatenation of the query (Q) and key (K) through a feedforward neural network with a tanh activation function, followed by a learned weight vector w_v. The attention distribution is obtained by applying a softmax function to the output of this network. The formula is as follows:

$$Attention(Q, K, V) = softmax\left({w_v}^T \cdot \tanh\left(W_q Q + W_k K\right)\right)V$$

- Additive Attention has certain advantages:
  - **Flexibility:** It can model complex relationships between queries and keys due to the use of a learnable feedforward network.
  - **Interpretability:** The attention weights are often more interpretable in certain tasks.
- However, the additional parameters and computations compared to Scaled Dot-Product Attention limits creates computational cost. It is also less suitable for large-scale parallel computation.

Attention: Implementation

$$Attention(Q, K, V) = softmax\left(w_v^T \cdot \tanh\left(W_q Q + W_k K\right)\right)V$$

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

```python
class AdditiveAttention(nn.Module):
    """Additive attention."""
    def __init__(self, num_hiddens, dropout, **kwargs):
        super(AdditiveAttention, self).__init__(**kwargs)
        self.W_k = nn.LazyLinear(num_hiddens, bias=False)
        self.W_q = nn.LazyLinear(num_hiddens, bias=False)
        self.w_v = nn.LazyLinear(1, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, queries, keys, values, valid_lens):
        queries, keys = self.W_q(queries), self.W_k(keys)
        # After dimension expansion, shape of queries: (batch_size, no. of
        # queries, 1, num_hiddens) and shape of keys: (batch_size, 1, no. of
        # key-value pairs, num_hiddens). Sum them up with broadcasting
        features = queries.unsqueeze(2) + keys.unsqueeze(1)
        features = torch.tanh(features)
        # There is only one output of self.w_v, so we remove the last
        # one-dimensional entry from the shape. Shape of scores: (batch_size,
        # no. of queries, no. of key-value pairs)
        scores = self.w_v(features).squeeze(-1)
        self.attention_weights = masked_softmax(scores, valid_lens)
        # Shape of values: (batch_size, no. of key-value pairs, value
        # dimension)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

```python
class DotProductAttention(nn.Module):
    """Scaled dot product attention."""
    def __init__(self, dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout)

    # Shape of queries: (batch_size, no. of queries, d)
    # Shape of keys: (batch_size, no. of key-value pairs, d)
    # Shape of values: (batch_size, no. of key-value pairs, value dimension)
    # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
    def forward(self, queries, keys, values, valid_lens=None):
        d = queries.shape[-1]
        # Swap the last two dimensions of keys with keys.transpose(1, 2)
        scores = torch.bmm(queries, keys.transpose(1, 2)) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, valid_lens)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

Content

Self-Attention and Learnable Query

Self-Attention:

- $Q = K = V = X \in \mathbb{R}^{d \times n}$
- Computes internal correlations:

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

Learnable Query:

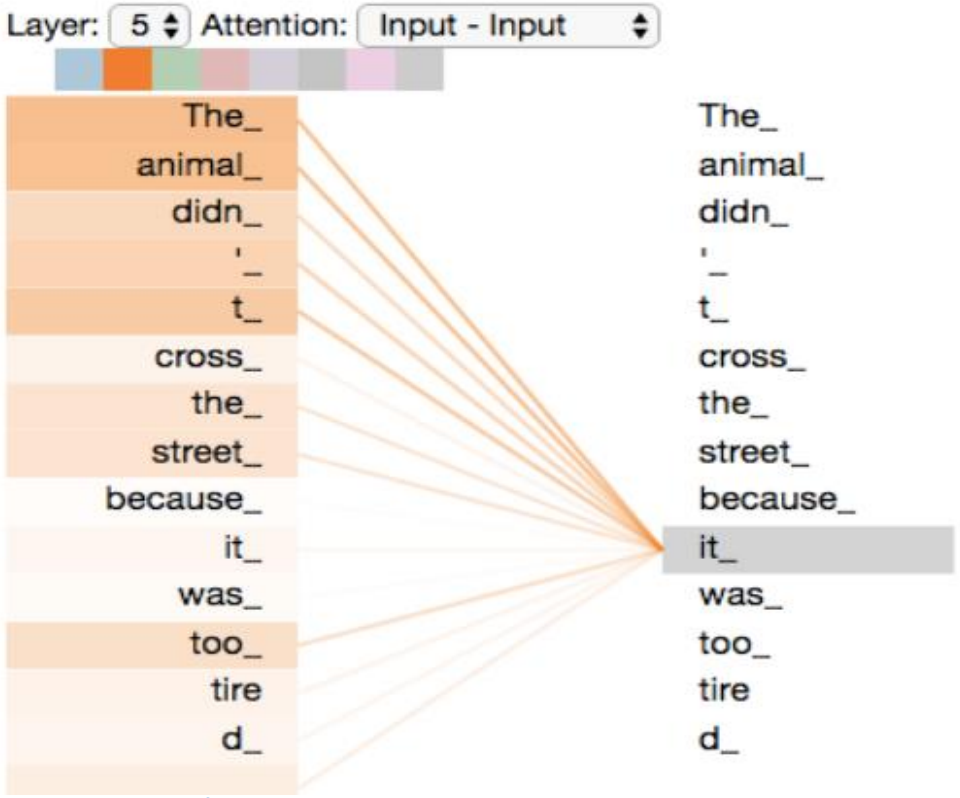
- $K = V = X$, Q is trainable and independent of X
- Output size is fixed, independent of input size:

$$O = W_v X \cdot \text{softmax}((W_k X)^\top Q)$$

Aspect	Self-Attention	Learnable Query
Query Definition	Q=K=V=X (input data itself)	Q is independent and learnable
Input Dependency	Input size determines query size	Query size is fixed and independent
Flexibility	Suitable for variable-length inputs	Suitable when fixed-length output is needed
Use Case	Typically used for internal relationships in input data (like word dependencies)	Often used when fixed-size representation is needed, regardless of input size
Example	Transformer encoder, where all tokens attend to each other	Object detection, where fixed number of queries are learned to detect objects

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)

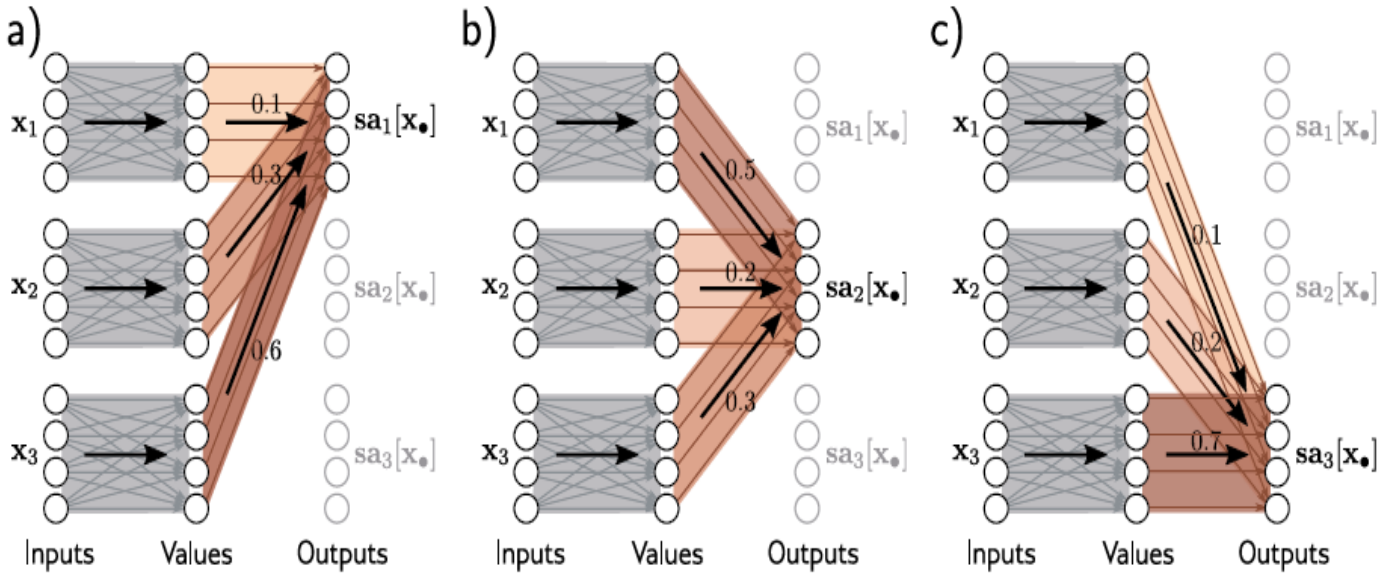
Self-Attention

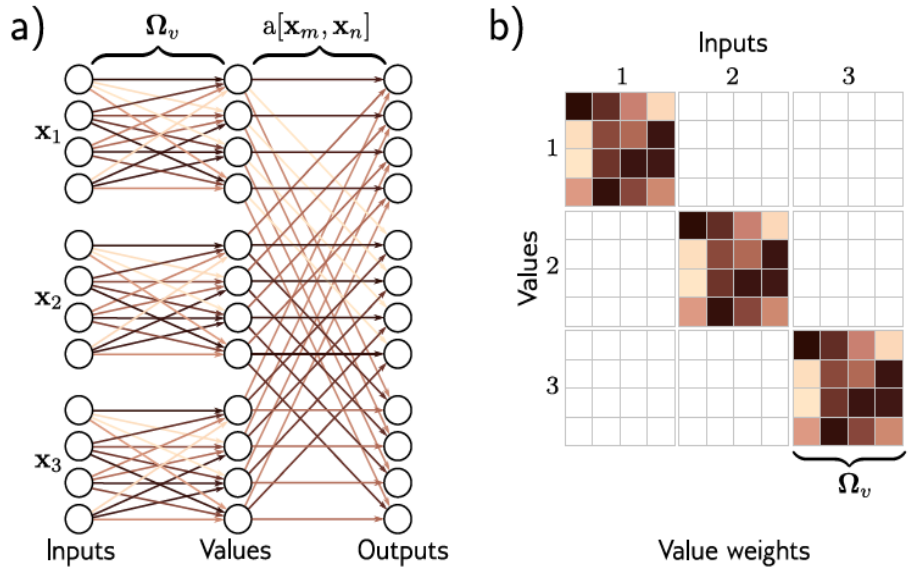
- **Self-attention:** Every token is attending to each other token, that is routing the values in different proportions to create each output.

- A self-attention block takes n inputs, each of dimension $d \times 1$, and returns n output vectors of the same size. Given a sequence of input tokens $x_1, \ldots, x_n$ where any $x_i \in \mathbb{R}^d$, **its self attention** outputs a sequence of same length $y_1, .., y_n \in \mathbb{R}^d$.

$$v_i = \beta_v + \Omega_v x_i$$

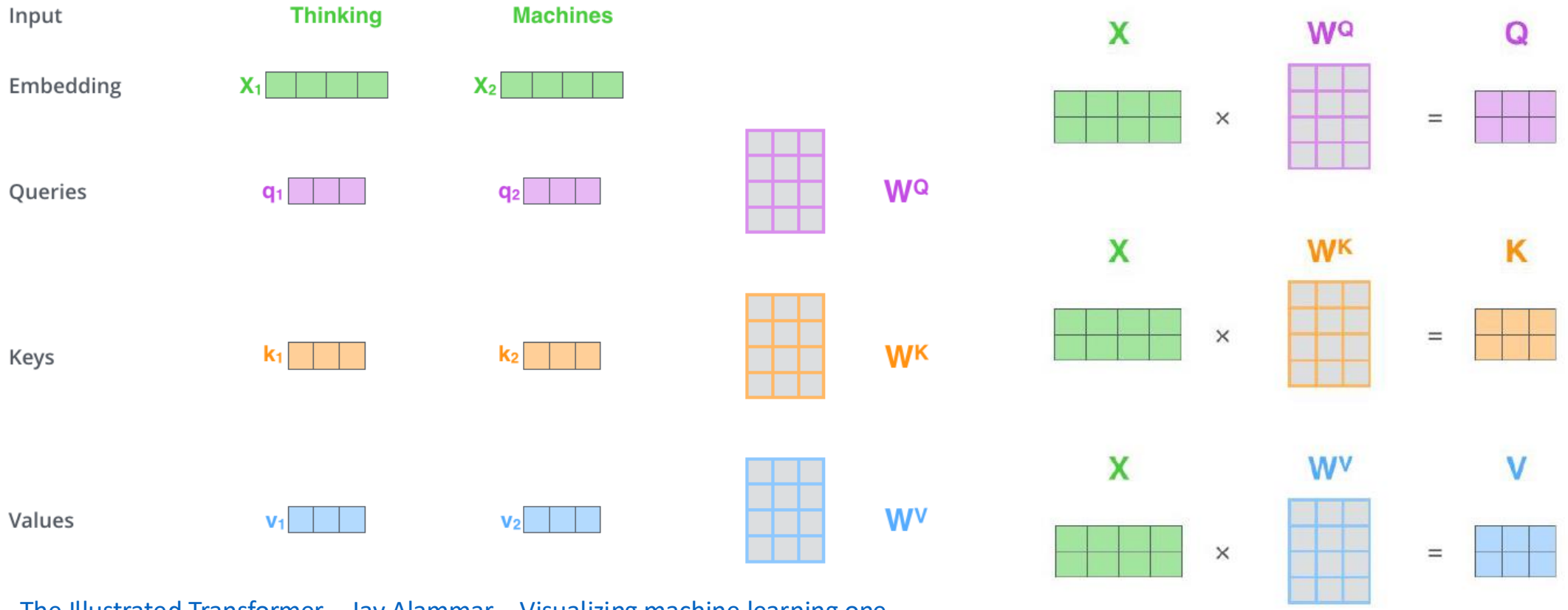
$$y_i = \sum_j a(x_j, x_i)\, v_i$$





Self-Attention: Details

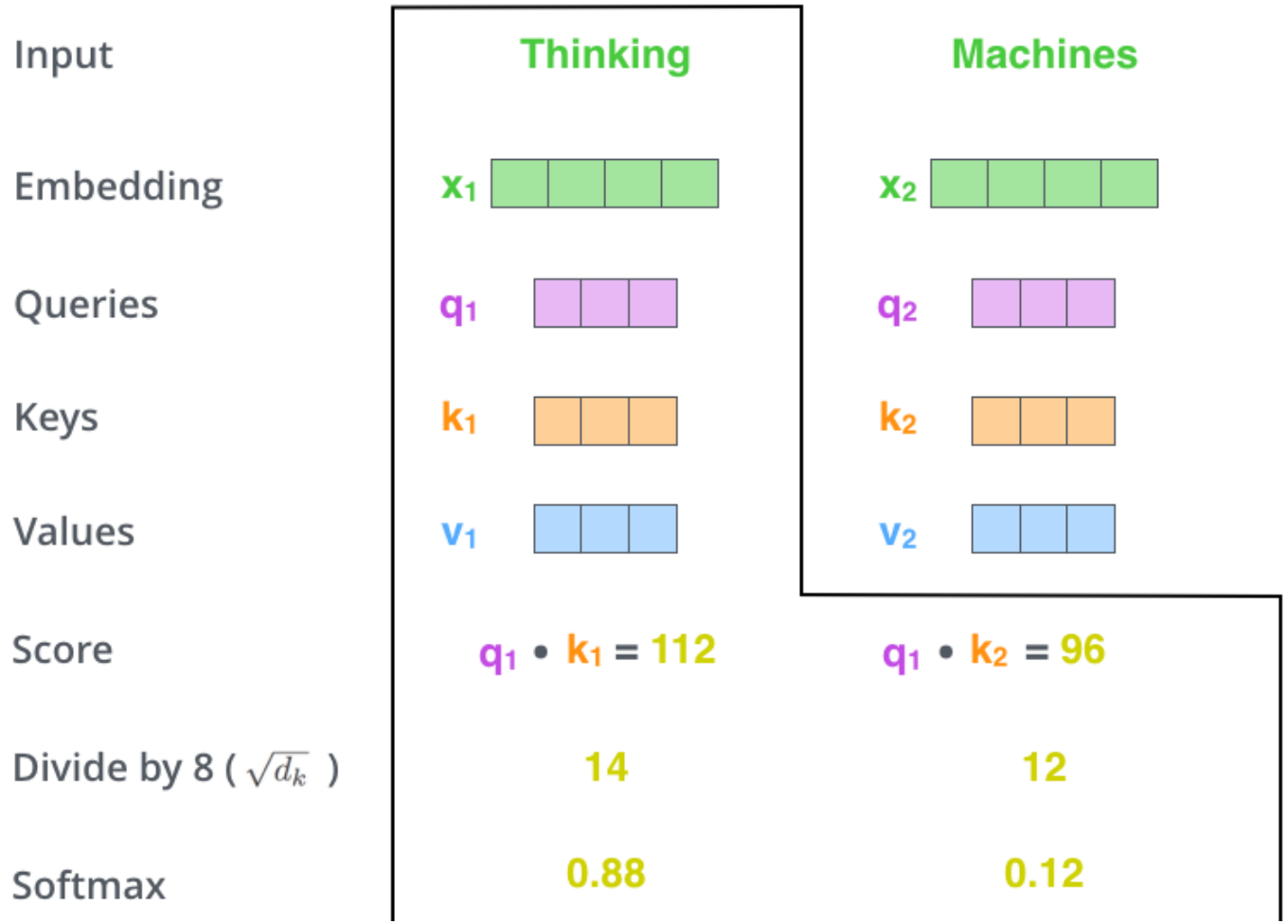
- **The first step is to create three vectors from each of encoder's input vectors.** These vectors are created by multiplying the embedding by three matrices that we trained during the training process (usually smaller in dimension than the embedding vector).



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

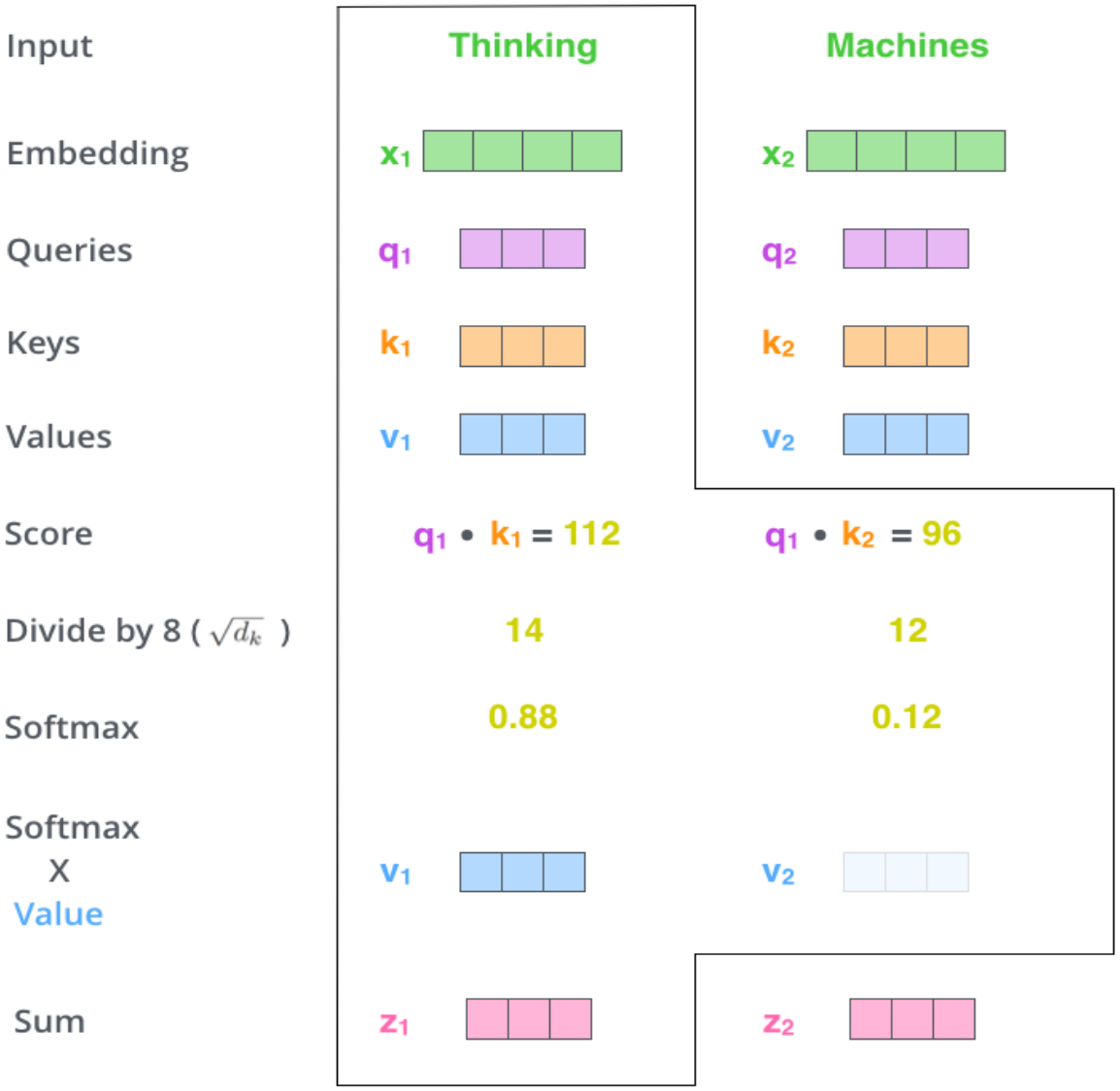
Self-Attention: Details

- **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

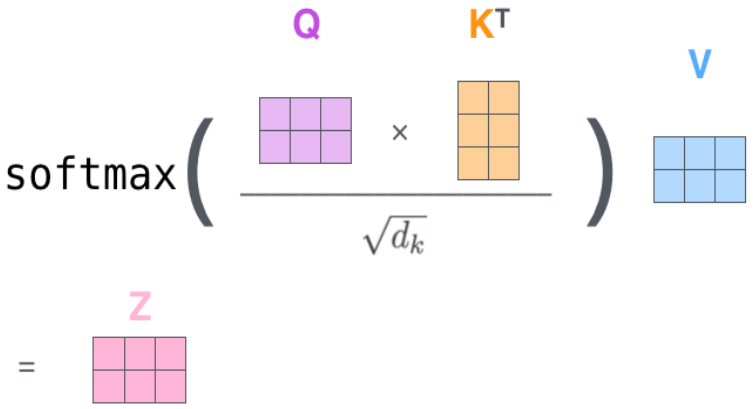
- **The third step is to multiply each value vector by the softmax score and sum up the weighted value vectors.**
- The resulting vector is the one we can send along to the feed-forward neural network.
- In the actual implementation, such calculation is done in matrix form for faster processing.

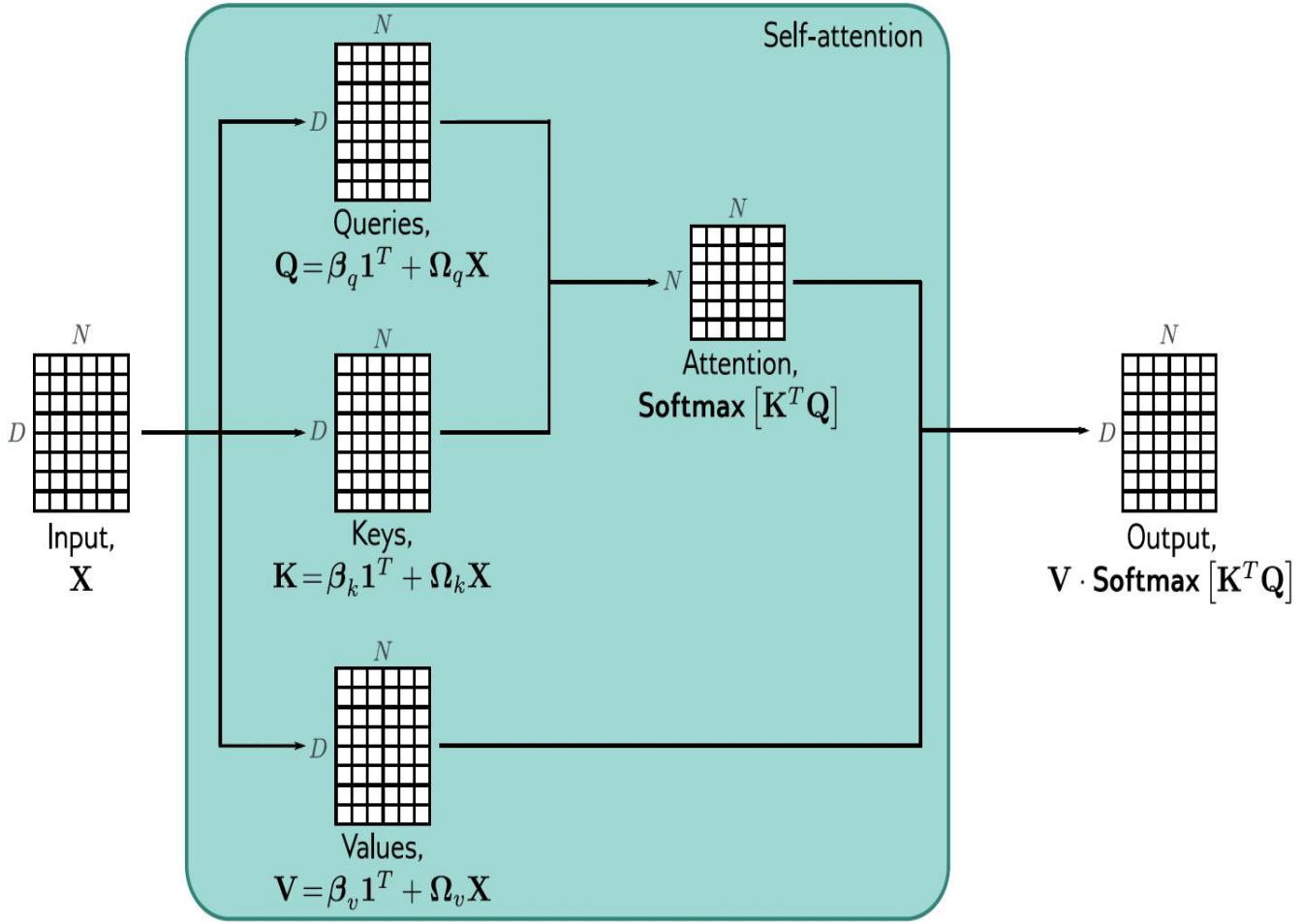


Self-Attention: Matrix Calculation

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

$$Q = W_Q X = \beta_q 1^T + \Omega_q X$$
$$K = W_K X = \beta_k 1^T + \Omega_k X$$
$$V = W_V X = \beta_v 1^T + \Omega_v X$$
$$Y = Softmax(QK^T)V$$





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: $A(T_\pi(X)) = A(X)$

Theorem

- Self-attention is equivariant:

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

- Attention with learned query is invariant:

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

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_π:

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

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

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

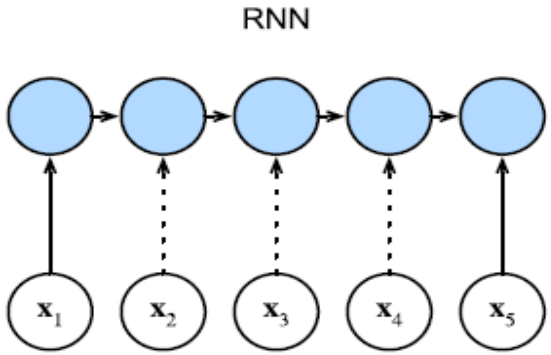
RNN, CNN, and Self-Attention

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.

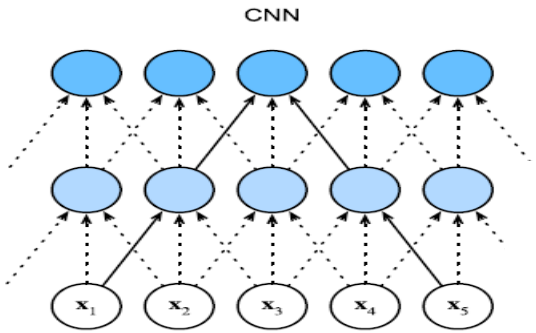
Aspect	RNNs	CNNs	Self-Attention
Primary Purpose	Sequential data processing	Spatial data processing	Capturing global relationships and dependencies
Data Handling	Temporal and ordered sequences	Local and spatial features	Long-range and global dependencies
Model Type	Recurrent neural networks	Convolutional 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

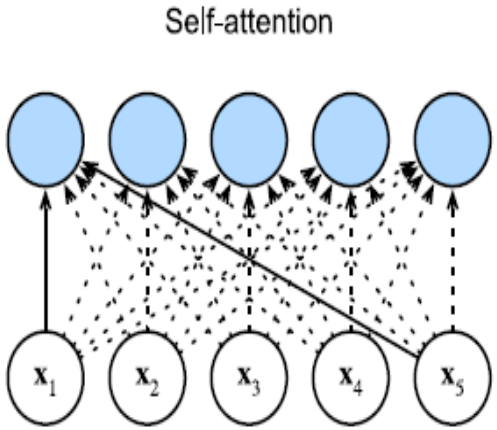
- RNN: d dimensional hidden state
  - $O(nd^2)$ computational complexity
  - $O(n)$ sequential operations
  - $O(n)$ maximal path length



- CNN: Given a sequence of length n, kernel size k and numbers of input and output channels d.
  - $O(knd^2)$ computational complexity
  - $O(1)$ sequential operations
  - $O\left(\frac{n}{k}\right)$ maximal path length

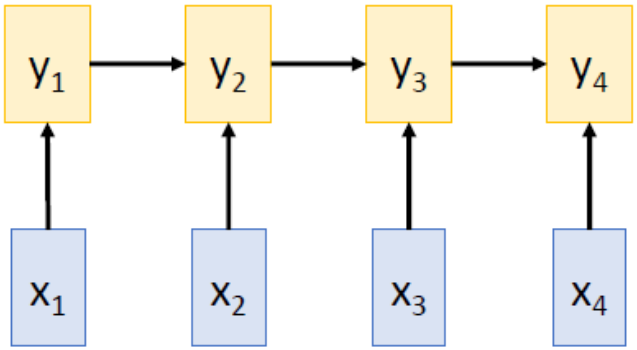


- Self-Attention: queries, keys and values are all $n \times d$
  - $O(n^2 d)$ computational complexity $\rightarrow$ **prohibitively slow and improved by later works**
  - $O(1)$ sequential operations
  - $O(1)$ maximal path length $\rightarrow$ **easier to capture long range dependencies**



Comparing RNNs, CNNs and Self-Attention

Recurrent Neural Network

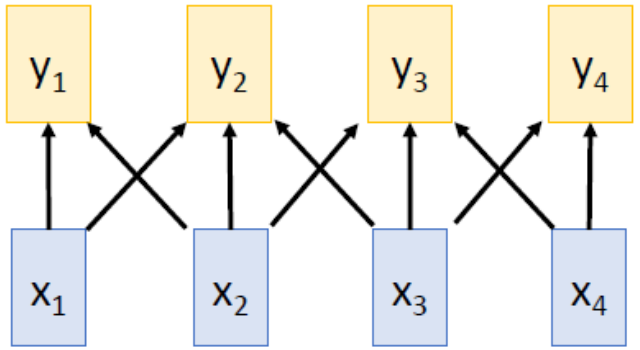


Works on **Ordered Sequences**

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

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

1D Convolution



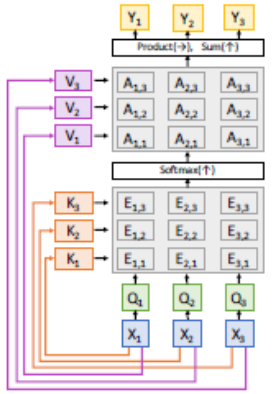
Works on **Multidimensional Grids**

(-) 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

From Self-Attention to Transformers

- The basic concept of self-attention can be used to develop a very powerful type of sequence model, called **a Transformer.**
- But to make this actually work, we need to develop a few additional components to address some functional limitations.
  - **Masked encoding:** How to prevent attention lookups into the future? Ensures causality in autoregressive models.
  - **Multi-head Attention**: Allows querying multiple positions at each layer. Increases model capacity to capture complex patterns.
  - **Positional encoding**: Address lack of sequence information, especially for images and videos. Provides sequence order awareness, crucial for text, images, and videos.

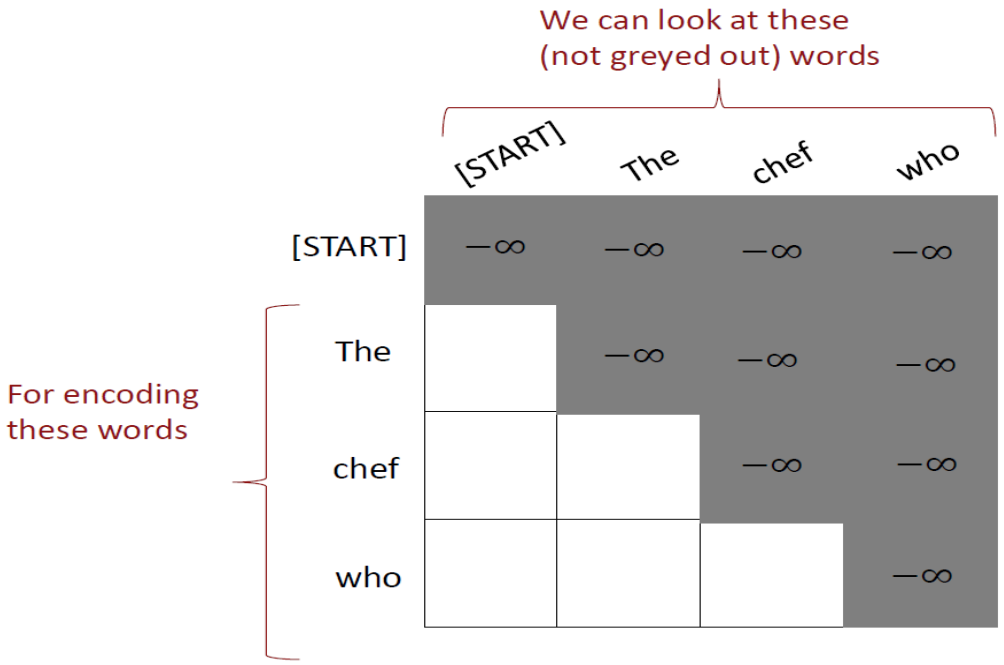
The combination of these techniques makes Transformers robust and efficient for sequence modeling.

Masked Attention

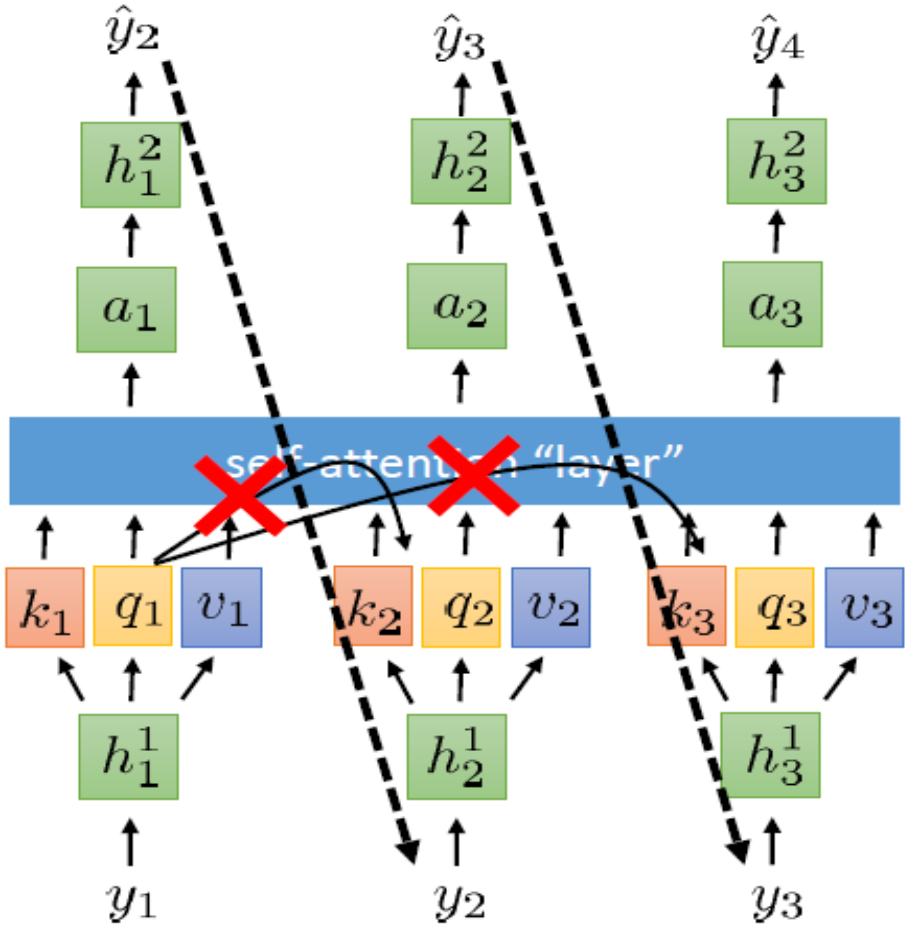
- **Problem:** During training, self-attention might access future tokens, leading to data leakage.
- **Solution:** Use a mask to block attention from future positions.
- **Implementation:** Add a mask matrix M:

$$Scores = \frac{QK^T}{\sqrt{d_k}}, \qquad Mask = \begin{cases} 0, past \\ -\infty, future \end{cases}$$

$$Masked\ Scores = Scores + Mask$$



A **crude** self-attention "language model":



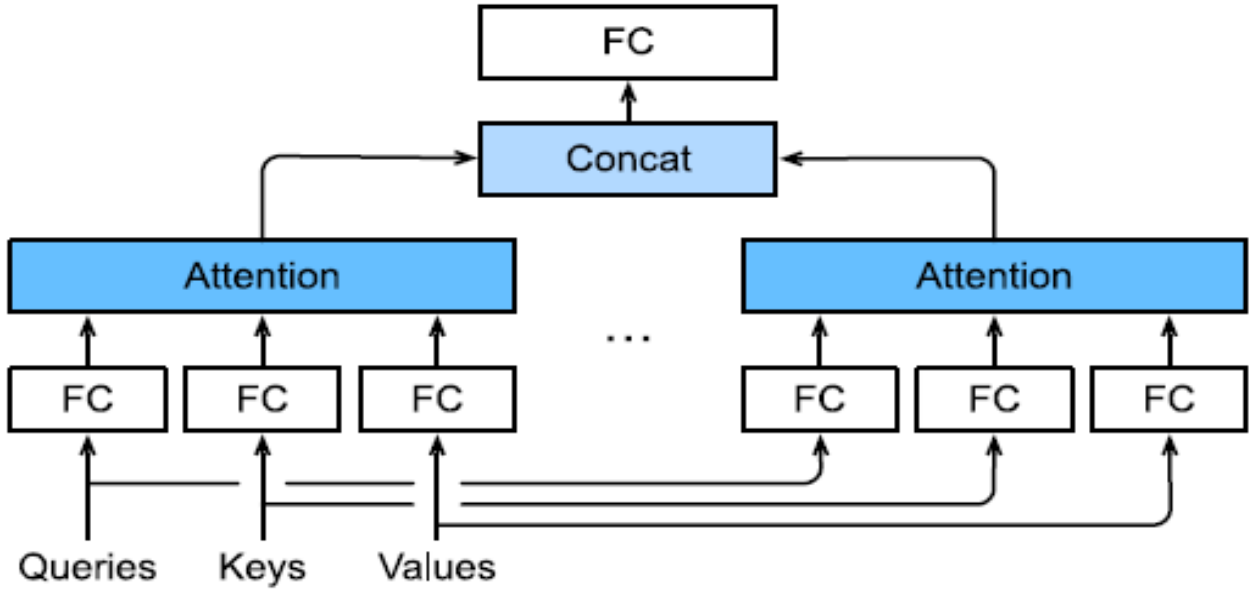
(Levine, 2021)

Multi-Head Attention

- In practice, given the same set of queries, keys and values we may want out 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**.

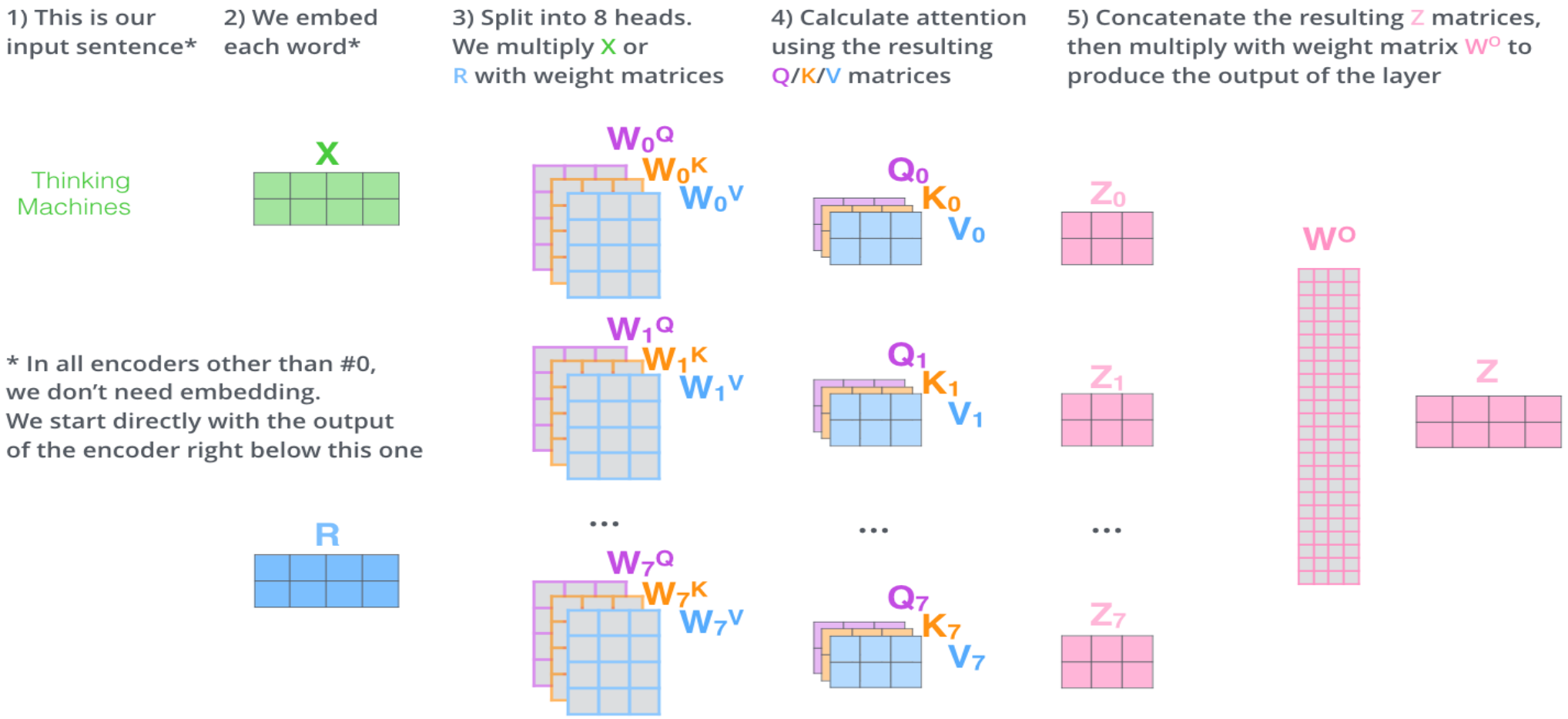
Multi-Head Attention

- Give a query $q \in \mathbb{R}^{d_q}$, a key k $\in \mathbb{R}^{d_k}$ and a value $v \in \mathbb{R}^{d_v}$, each attention head h_i is computed as $h_i = f\left(W_i^{(q)}q, W_i^{(k)}k, W_i^{(v)}v\right) \in \mathbb{R}^{p_v}$ where $W_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$, $W_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$ and $W_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$ are learnable parameters and f is attention pooling such as additive attention or scaled dot product attention.
- The feed-forward layer is not expecting multiple matrices. Therefore, the output is another linear transformation via learnable parameters $W_o \in \mathbb{R}^{p_o \times Mp_v}$ of concatenation of M heads: $W_o \begin{pmatrix} h_1 \\ \dots \\ h_M \end{pmatrix} \in \mathbb{R}^{p_o}$.



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.



Multi-Head Attention

```python
class MultiHeadAttention(Module):
    """Multi-head attention."""
    def __init__(self, num_hiddens, num_heads, dropout, bias=False, **kwargs):
        super().__init__()
        self.num_heads = num_heads
        self.attention = DotProductAttention(dropout)
        self.W_q = nn.LazyLinear(num_hiddens, bias=bias)
        self.W_k = nn.LazyLinear(num_hiddens, bias=bias)
        self.W_v = nn.LazyLinear(num_hiddens, bias=bias)
        self.W_o = nn.LazyLinear(num_hiddens, bias=bias)

    def forward(self, queries, keys, values, valid_lens):
        # Shape of queries, keys, or values:
        # (batch_size, no. of queries or key-value pairs, num_hiddens)
        # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
        # After transposing, shape of output queries, keys, or values:
        # (batch_size * num_heads, no. of queries or key-value pairs,
        # num_hiddens / num_heads)
        queries = self.transpose_qkv(self.W_q(queries))
        keys = self.transpose_qkv(self.W_k(keys))
        values = self.transpose_qkv(self.W_v(values))

        if valid_lens is not None:
            # On axis 0, copy the first item (scalar or vector) for num_heads
            # times, then copy the next item, and so on
            valid_lens = torch.repeat_interleave(
                valid_lens, repeats=self.num_heads, dim=0)

        # Shape of output: (batch_size * num_heads, no. of queries,
        # num_hiddens / num_heads)
        output = self.attention(queries, keys, values, valid_lens)
        # Shape of output_concat: (batch_size, no. of queries, num_hiddens)
        output_concat = self.transpose_output(output)
        return self.W_o(output_concat)
```

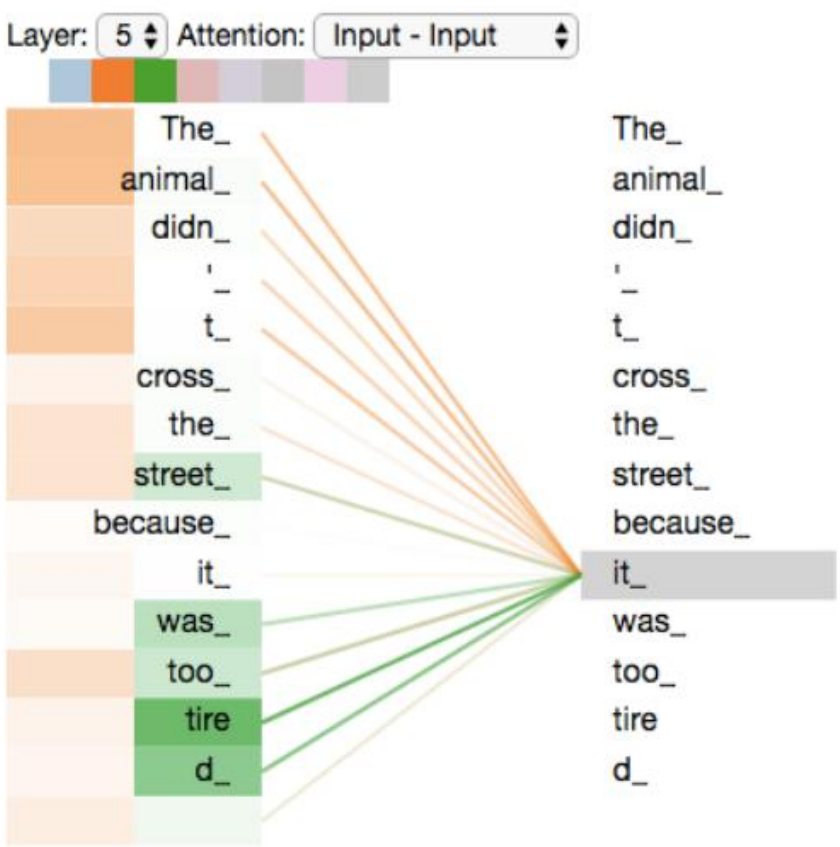
```python
def transpose_qkv(self, X):
    """Transposition for parallel computation of multiple attention heads."""
    # Shape of input X: (batch_size, no. of queries or key-value pairs,
    # num_hiddens). Shape of output X: (batch_size, no. of queries or
    # key-value pairs, num_heads, num_hiddens / num_heads)
    X = X.reshape(X.shape[0], X.shape[1], self.num_heads, -1)
    # Shape of output X: (batch_size, num_heads, no. of queries or key-value
    # pairs, num_hiddens / num_heads)
    X = X.permute(0, 2, 1, 3)
    # Shape of output: (batch_size * num_heads, no. of queries or key-value
    # pairs, num_hiddens / num_heads)
    return X.reshape(-1, X.shape[2], X.shape[3])
```

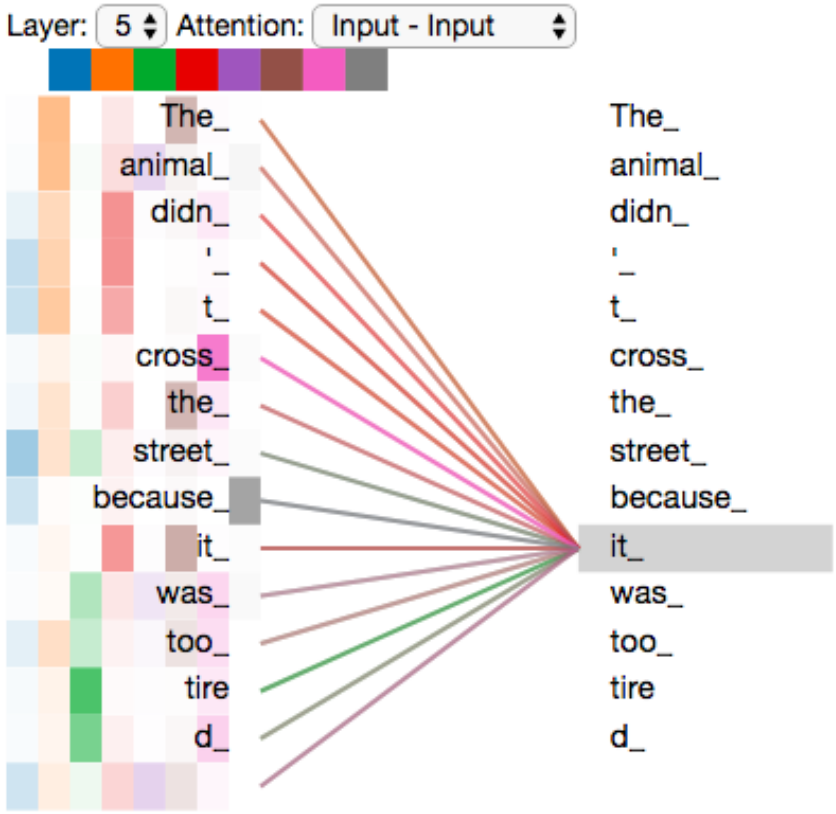
```python
def transpose_output(self, X):
    """Reverse the operation of transpose_qkv."""
    X = X.reshape(-1, self.num_heads, X.shape[1], X.shape[2])
    X = X.permute(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

Multi-Head Attention: Example

As we encode the word **“it”,** one attention head is focusing most on **“the animal”,** while another is focusing on **“tired”** -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

However, once all attention heads are to be considered, things can get started to be harder for interpretation.





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:

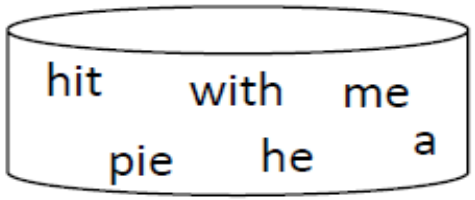
he hit me with a pie

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

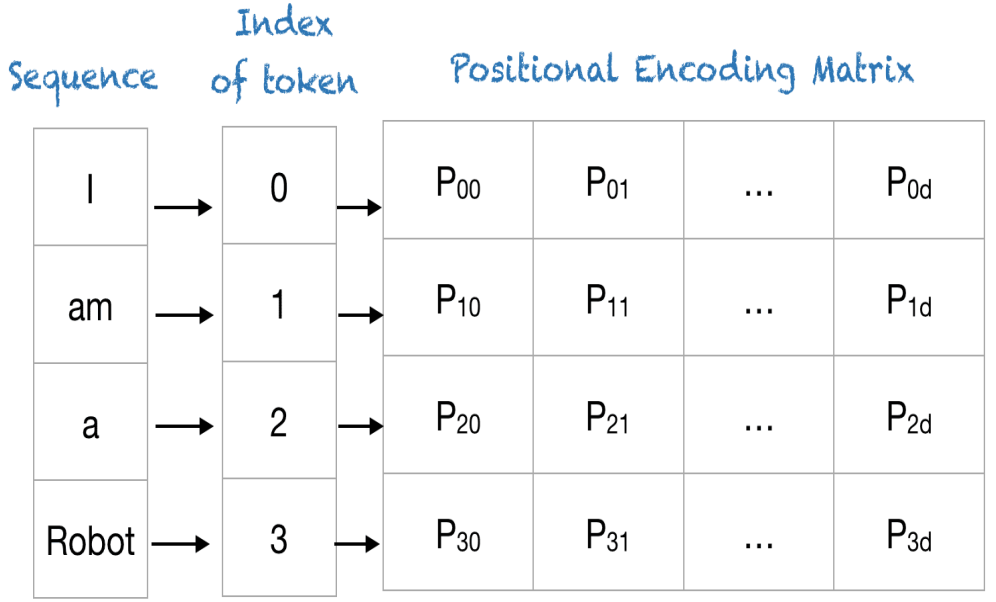
- The naïve positional encoding would just append t to the input: $\overline{x_t} = \binom{x_t}{t}$. However, it would not be a great idea, because the **absolute** position is less important than the **relative** position.

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?

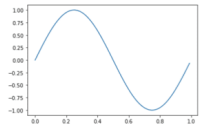
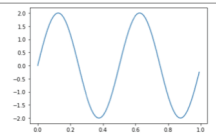
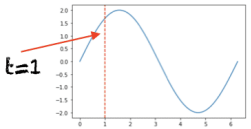


Positional Encoding Matrix for the sequence 'I am a robot'

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 ith row and the $(2j)$th or $(2j+1)$th column is given by

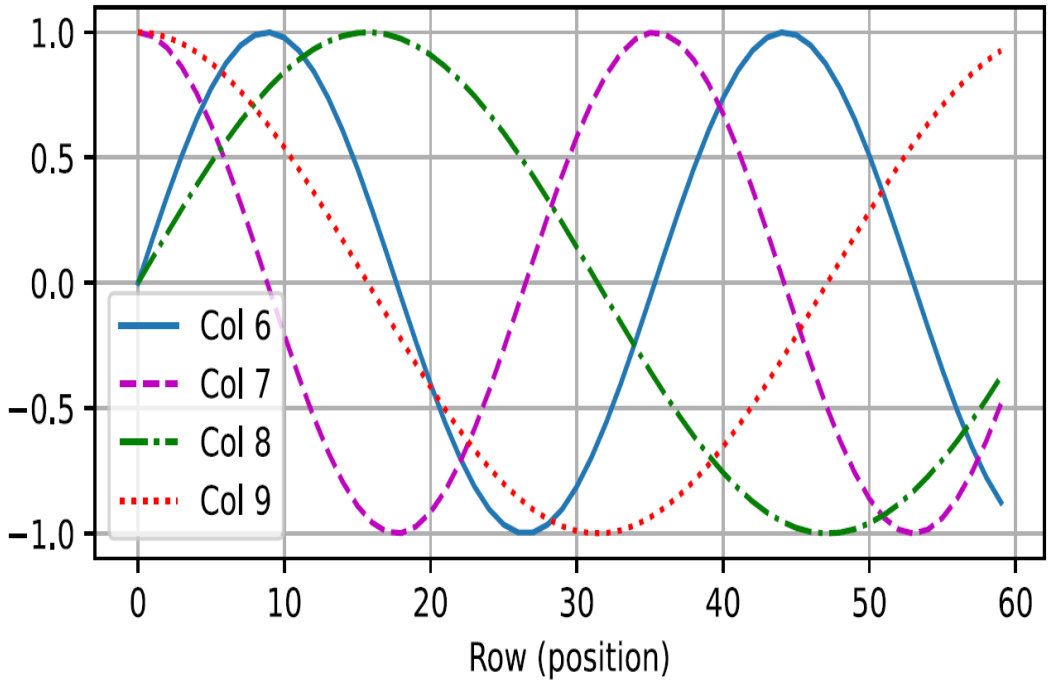
$$p_{i,2j} = \sin\left(\frac{i}{1000^{\frac{2j}{d}}}\right), p_{i,2j+1} = cos\left(\frac{i}{1000^{\frac{2j}{d}}}\right)$$

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

- **Angular Frequency (ω_j):**
  - $\omega_j = \frac{1}{1000^{\frac{2j}{d}}}$
  - Each increment $i \to i+1$ increases the argument by ω_j.
- **Wavelength (λ_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

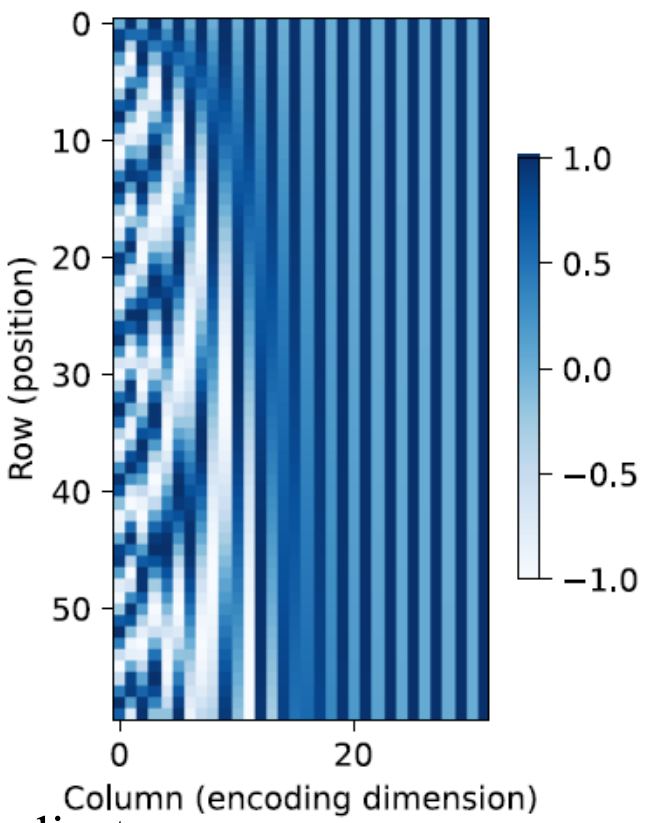
- In the positional embedding matrix P, rows correspond to positions within a s sequence and columns represent different positional encoding dimensions.
- In the example below, we can see that the 6^{th} and the 7^{th} columns of positional embedding matrix have a higher frequency than the 8^{th} and the 9^{th} columns.



- **Lower j:**
  - $\omega_j \approx 1$ for small $\frac{2j}{d}$, $\lambda_j \approx 6.28$ steps in discrete math.
  - Rapid oscillations over fewer positions.
- **Higher j:**
  - ω_j becomes very small, yielding large λ_j.
  - The sinusoid changes more slowly, capturing coarse-grained position.
- Suppose $d = 4$, base $= 1000$, and $j = 0, 1$.
  - For $j = 0$: $\omega_0 = 1/1000^0 = 1$, so $\lambda_0 = 2\pi$.
  - For $j = 1$: $\omega_1 = 1/1000^{\frac{2}{4}} = 1/1000^{0.5} = 1/\sqrt{1000}$, $\lambda_1 = 2\pi\sqrt{1000}$.
- Position i increments the phase by ω_j.
- This ensures some encodings oscillate quickly, others slowly.

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
```

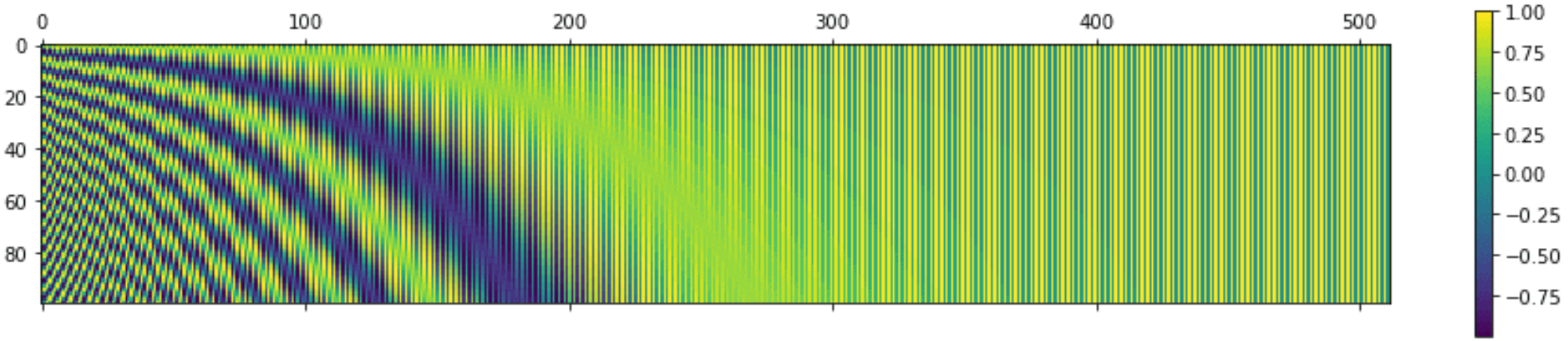


"even-odd" indicator

"first half-second half indicator" indicator

- **Monotonically decreasing frequency:** In sinusoidal positional encoding, each higher dimension has a lower frequency.
  - $\omega_j = \frac{1}{10000^{\frac{2j}{d}}}$ 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



- **Absolute Position:**
  - Different frequencies let the model pinpoint an absolute location in the sequence.
  - Example: $i = 10$ vs. $i = 100$ produce noticeably different multi-sinusoidal phases.
- **Relative Position:**
  - Key property: For an offset δ, $PE(i + \delta)$ can be derived from $PE(i)$ via a linear transformation (thanks to trig identities).
  - Allows model to easily learn relative shifts (e.g., paying attention to positions $i + 1, i + 2$) without separate parameters.
- **Result:** The encoding provides both absolute and relative cues in a single representation.

- **Positional Encoding Analogy:**
  - In dimension 0, sine wave might oscillate within a handful of steps.
  - In dimension $d - 1$, the wave might only complete a fraction of a single period over the entire sequence.
  - This layered range of frequencies parallels the layered range of bit flips in binary.
- **Space Efficiency:**
  - Instead of storing a discrete bit for each position, the continuous sinusoidal approach uses fewer dimensions to encode both small and large positional changes.
  - Facilitates interpolation and generalization across different sequence lengths.

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

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.

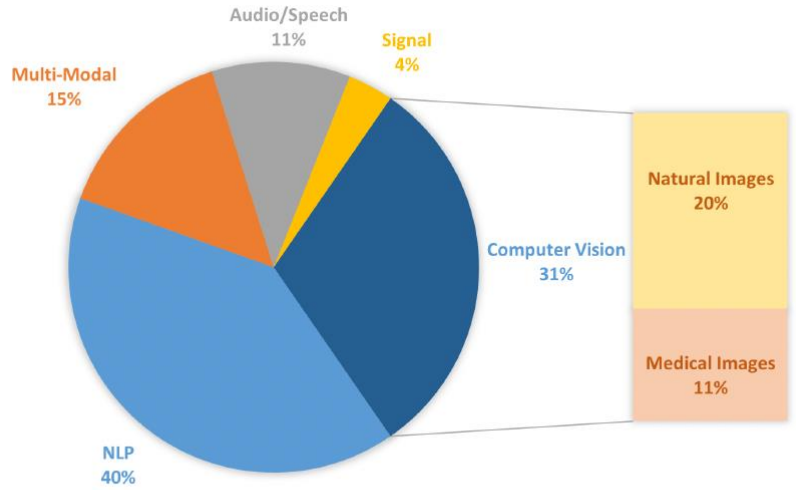
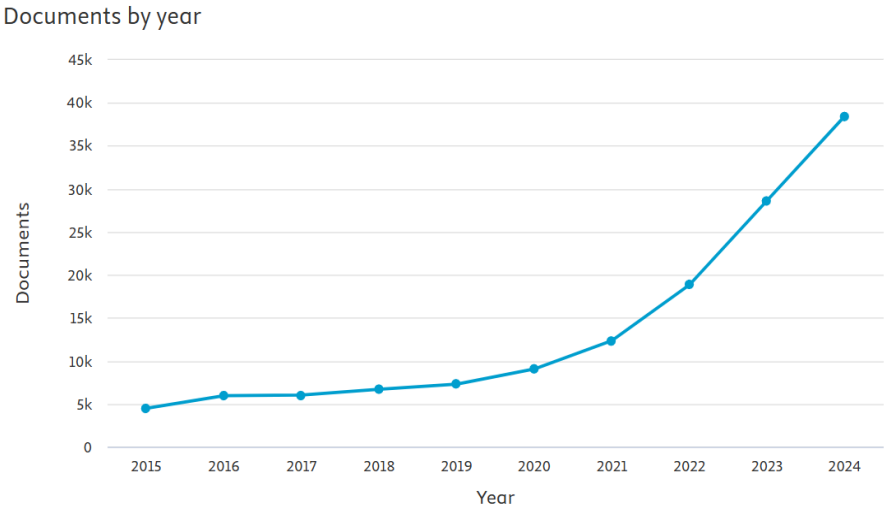


Figure 4: Proportion of transformer application in Top-5 fields



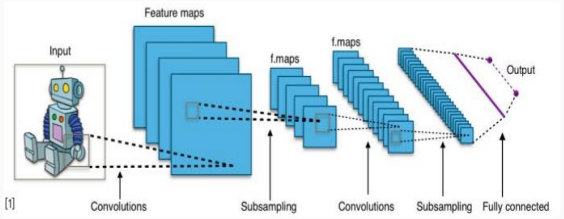
Islam, S., Elmekki, H., Elsebai, A., Bentahar, J., Drawel, N., Rjoub, G., & Pedrycz, W. (2024). A comprehensive survey on applications of transformers for deep learning tasks. *Expert Systems with Applications*, *241*, 122666.

Why Transformers?

Before Transformers

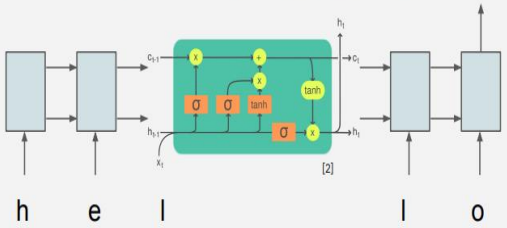
Computer Vision

Convolutional NNs (+ResNets)



Natural Lang. Proc.

Recurrent NNs (+LSTMs)

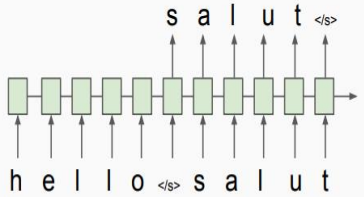


Speech

Deep Belief Nets (+non-DL)

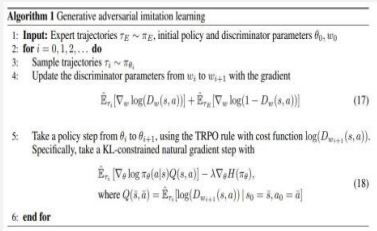
Translation

Seq2Seq



RL

BC/GAIL



Algorithm 1 Generative adversarial imitation learning

1: **Input:** Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters θ_0, w_0
2: **for** $i = 0, 1, 2, \ldots$ **do**
3: Sample trajectories $\tau_i \sim \pi_{\theta_i}$
4: Update the discriminator parameters from w_i to w_{i+1} with the gradient

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_w \log(D_w(s,a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_w \log(1 - D_w(s,a))] \quad (17)$$

5: Take a policy step from θ_i to θ_{i+1}, using the TRPO rule with cost function $\log(D_{w_{i+1}}(s,a))$. Specifically, take a KL-constrained natural gradient step with

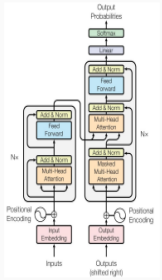
$$\hat{\mathbb{E}}_{\tau_i}[\nabla_\theta \log \pi_\theta(a|s) Q(s,a)] - \lambda \nabla_\theta H(\pi_\theta), \quad \text{where } Q(\bar{s},\bar{a}) = \hat{\mathbb{E}}_{\tau_i}[\log(D_{w_{i+1}}(s,a))|s_0 = \bar{s}, a_0 = \bar{a}] \quad (18)$$

6: **end for**

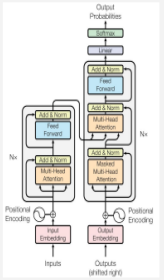
[1] CNN image CC-BY-SA by Aphex34 for Wikipedia https://commons.wikimedia.org/wiki/File:Typical_cnn.png
[2] RNN image CC-BY-SA by GChe for Wikipedia https://commons.wikimedia.org/wiki/File:The_LSTM_Cell.svg

After Transformers

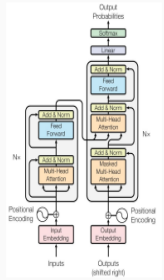
Computer Vision



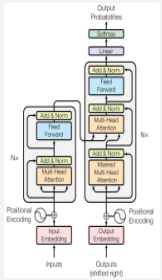
Natural Lang. Proc.



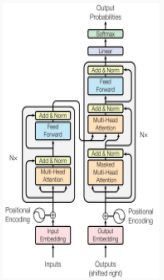
Reinf. Learning



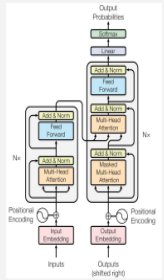
Speech



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

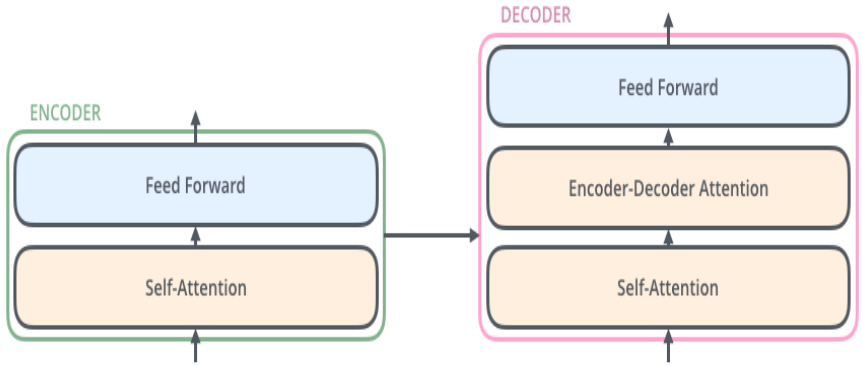
Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

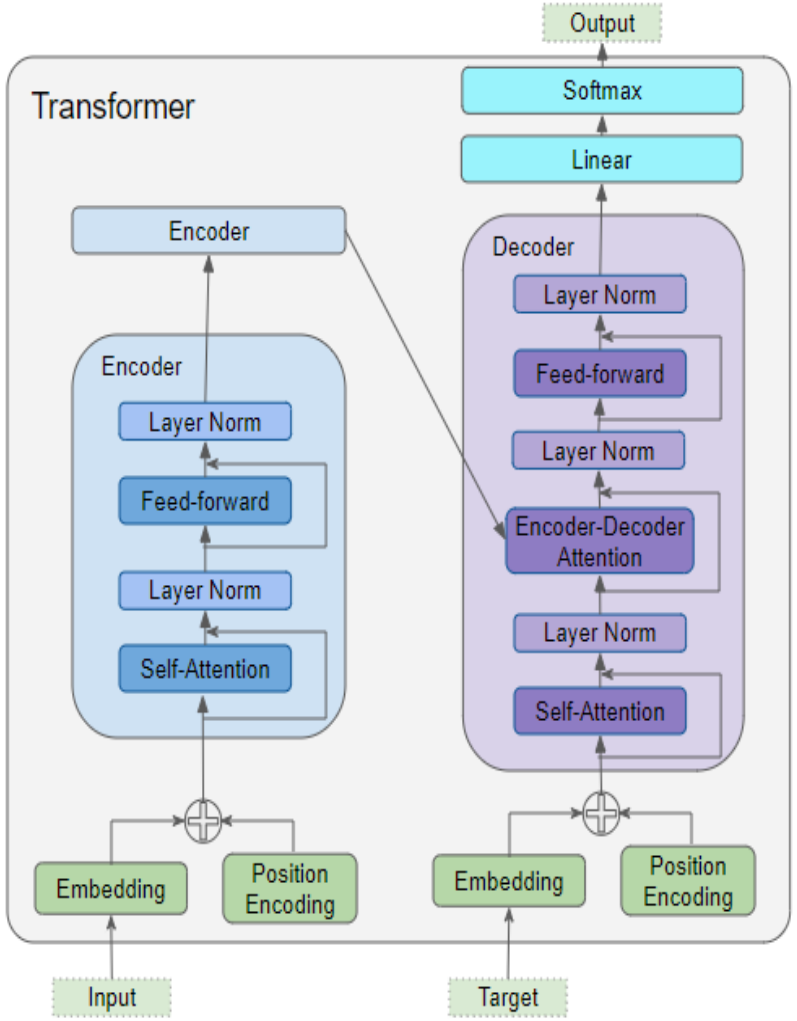
Citation: 173883+

Transformer: A High-Level Look

- **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/

Tokenization and Vocabulary Construction

- **Goal:**

Convert raw text into model-readable units (tokens).

Common Methods:

- Whitespace splitting (simple but limited).
- Word-level tokenization

(vulnerable to out-of-vocabulary words).

- Subword tokenization (BPE)

(compromise between letters and full words).

Importance: Effective tokenization impacts model performance.

- **Definition:** The vocabulary is the set of all tokens recognized by the tokenizer.

Size:

- Typically a user-defined hyperparameter (e.g., 30k, 50k tokens).
- Larger vocab covers more words but uses more memory.

Handling Rare Words:

- Full-word approach leads to frequent OOV (out-of-vocabulary) issues.
- Subword approaches store partial words, enabling composition of unknown words.
- Example: BERT uses a WordPiece vocabulary of 30k tokens.

Whitespace Splitting Example:

- Input: "Hello world!"
- Tokens: ["Hello", "world!"]
- Issue: Punctuation remains attached to "world!"

Word-Level Example:

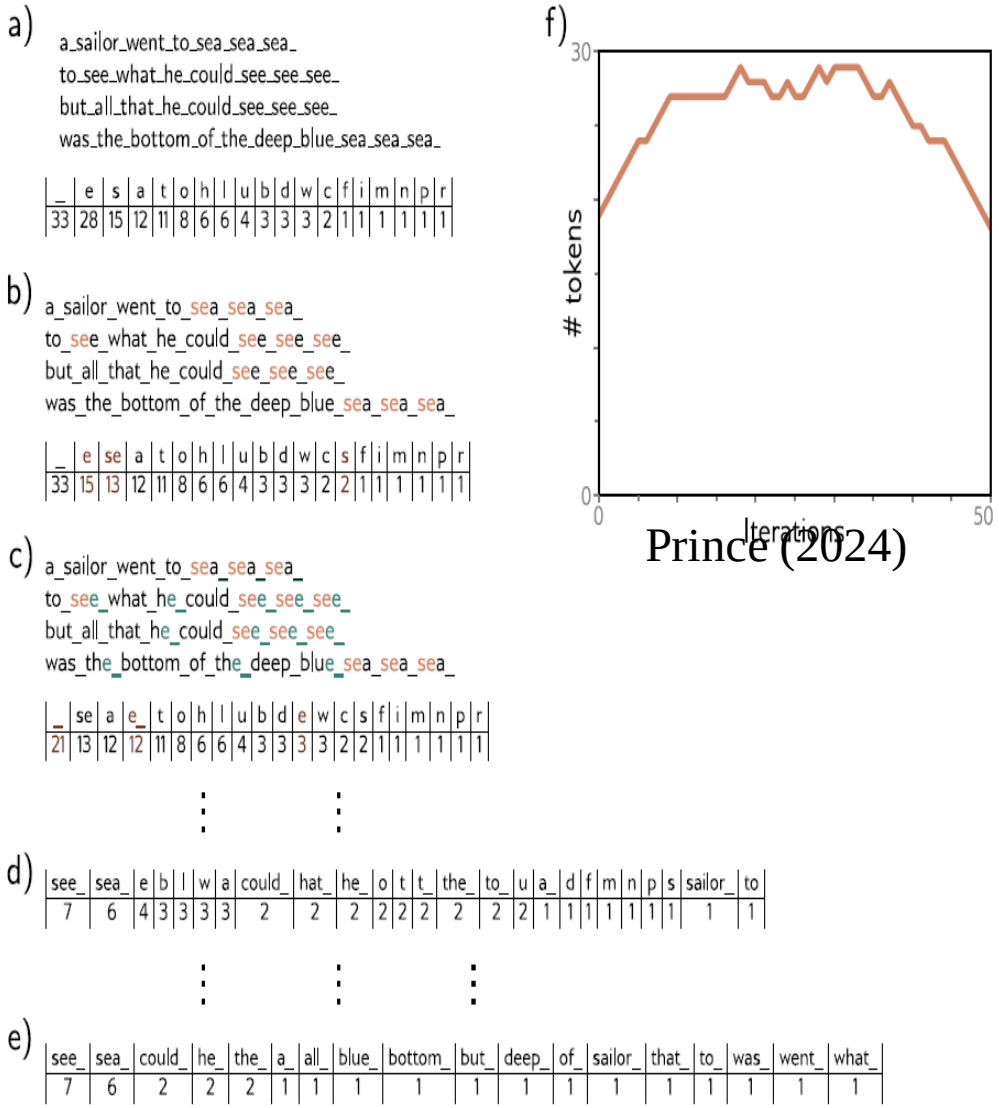
- Often attempts to separate punctuation from words.
- Tokens: ["Hello", "world", "!"]
- OOV Problem: If "Hello" isn't in the vocab, it becomes. "<UNK>".

BPE:

- Example 1: "unbelievable"
- If "unbelievable" is rare, split into subwords: Tokens: ["un", "believ", "able"]
- Example 2: "low-frequent-words"
- If "-" is a token, might see: Tokens: ["low", "-", "frequent", "-", "words"]

Illustration of BPE

- **Initial Characters:** The passage is split into single letters + whitespace (e.g., for space). Frequencies are tallied.
- **Iteration 1: Merge Most Common Pair**
  - ❖ E.g., merging s and e into the new token se.
  - ❖ Token counts for s and e decrease accordingly.
  - ❖ Constraint: Cannot merge across words (no merging if last char is whitespace).
- **Subsequent Iterations:**
  - ➢ Continue merging the most frequently adjacent pair, e.g. e and _, forming e_.
  - ➢ Over many iterations, tokens become a mix of letters, fragments, and common words.
- **Vocabulary Growth and Shrink:**
- As merges occur, the vocabulary expands with new fragments, then contracts as merges become less frequent.
- In practice, we stop once we reach a predefined vocabulary size.
- **Real-World Use:** Large corpora, punctuation, uppercase letters all handled as separate input characters, with a final vocab determined by the token count limit



Prince (2024)

Transformer Layer (Block)

▶ **Input:** D × 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 × N.

❖ Residual connection: add the original inputs back.

▶ **LayerNorm:**

➢ Applied to each column (token) independently.

➢ Normalizes across the embedding dimension.

▶ **Fully Connected Feed-Forward:**

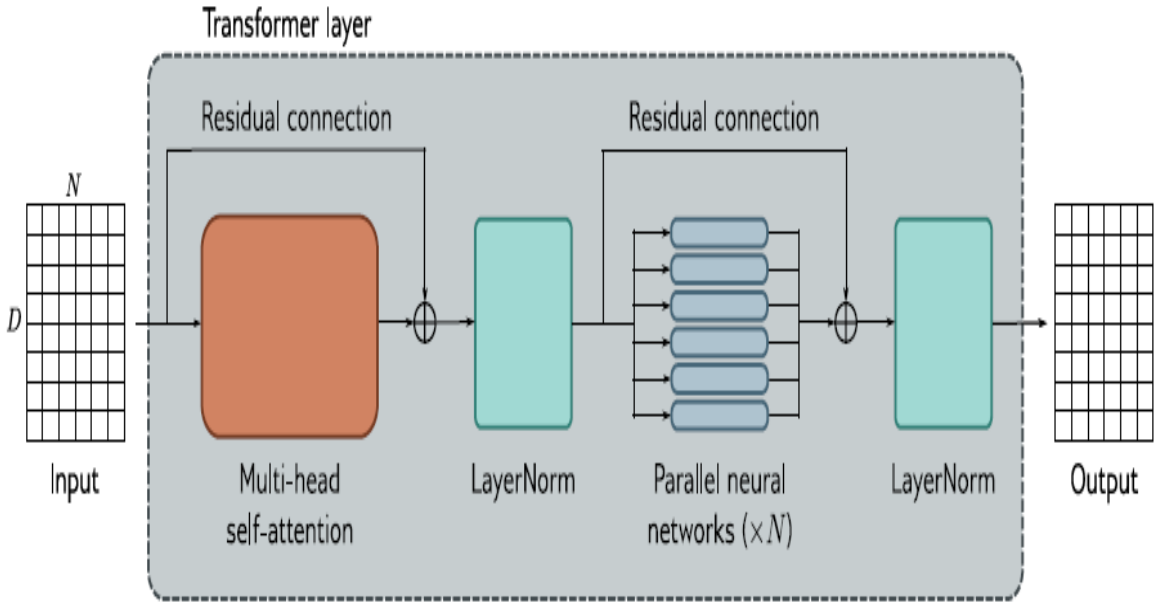
✓ Same MLP applied to each column.

✓ Residual connection again.

▶ **Final LayerNorm:**

❑ Normalizes outputs across D for each token.

▶ **Result:** Output is a D × N matrix with updated token representations.



$$
\begin{aligned}
\mathbf{X} &\leftarrow \mathbf{X} + \mathbf{MhSa}[\mathbf{X}] \\
\mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}] \\
\mathbf{x}_n &\leftarrow \mathbf{x}_n + \mathbf{mlp}[\mathbf{x}_n] \quad \forall\, n \in \{1, \ldots, N\} \\
\mathbf{X} &\leftarrow \mathbf{LayerNorm}[\mathbf{X}],
\end{aligned}
$$

Transformer Layer: Residual Connection

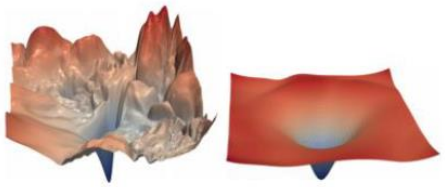
- Residual connection is a simple but powerful technique from computer vision.
- Observation: Deep neural networks are surprisingly bad at learning the identity function.
- Therefore, directly passing "raw" embeddings to the next layer would be very helpful!

$$x_l = f(x_{l-1}) + x_{l-1}$$

- This prevents the network from "forgetting" or distorting important information as it is processed by many layers.

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

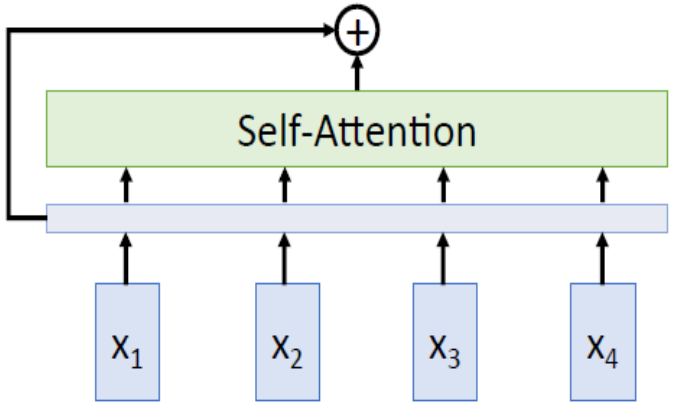


[no residuals] [residuals]

[Loss landscape visualization, Li et al., 2018, on a ResNet]

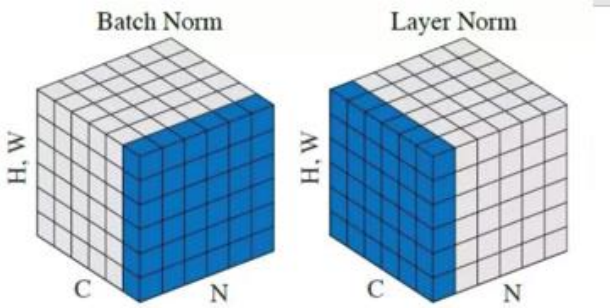
Residual connection

All vectors interact with each other



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 for each sample in batch

$a_1, a_2, \ldots, a_B$

d-dim

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

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

Layer norm — different *dimensions* of a

a

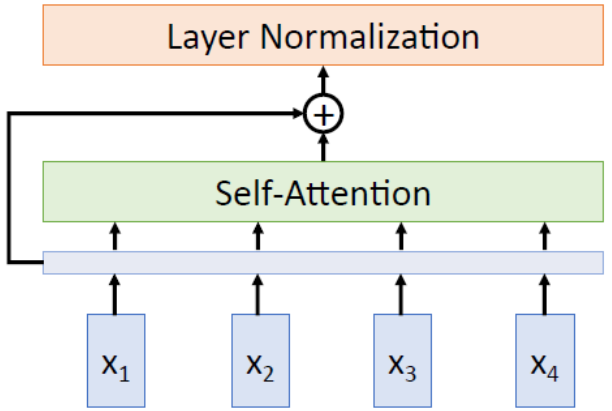
1-dim

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

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

Residual connection

All vectors interact with each other

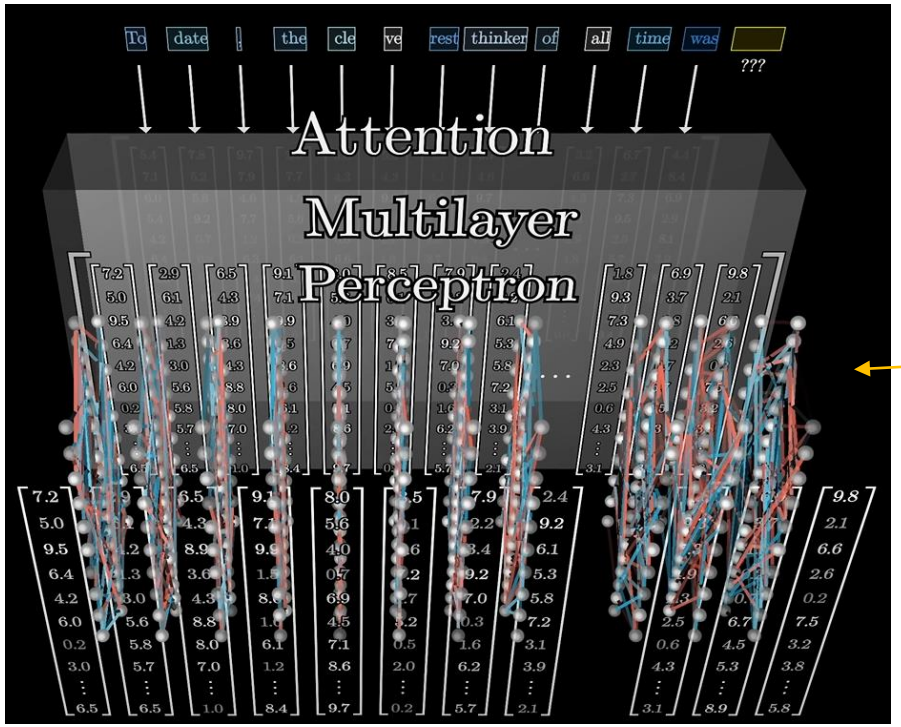


Layer norm is not applied to an entire transformer layer, but just to the embedding vector of a single token.

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*.

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

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

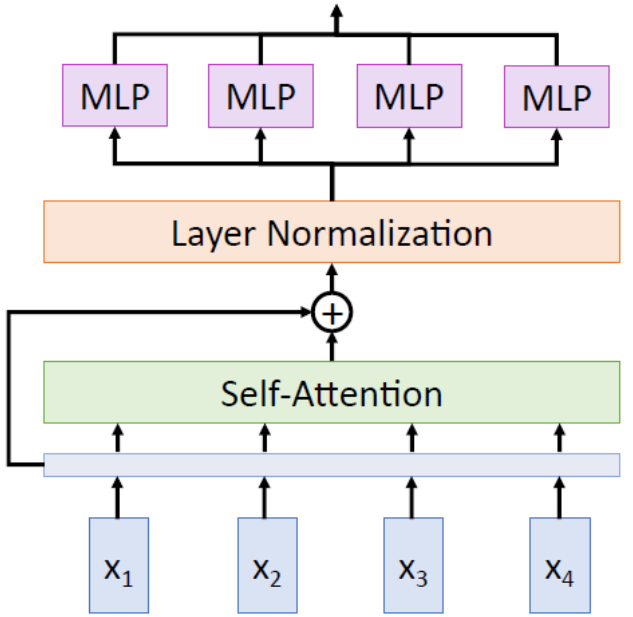


https://www.youtube.com/watch?v=wjZofJX0v4M

MLP independently on each vector

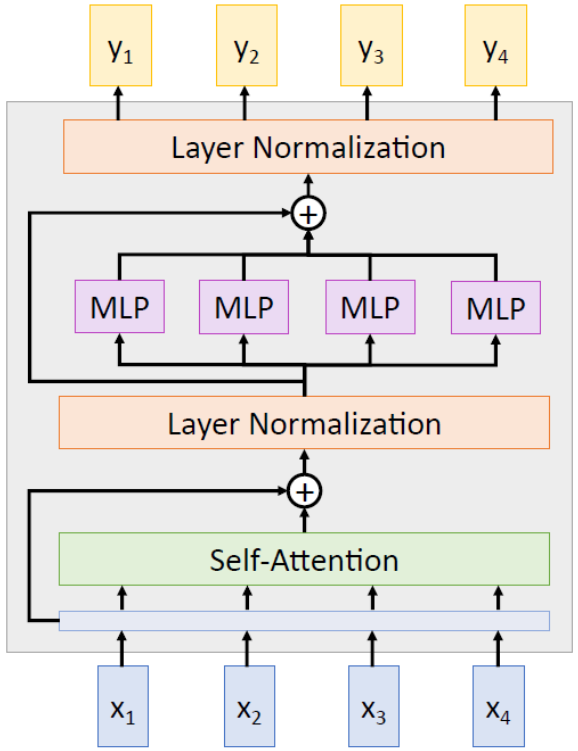
Residual connection

All vectors interact with each other

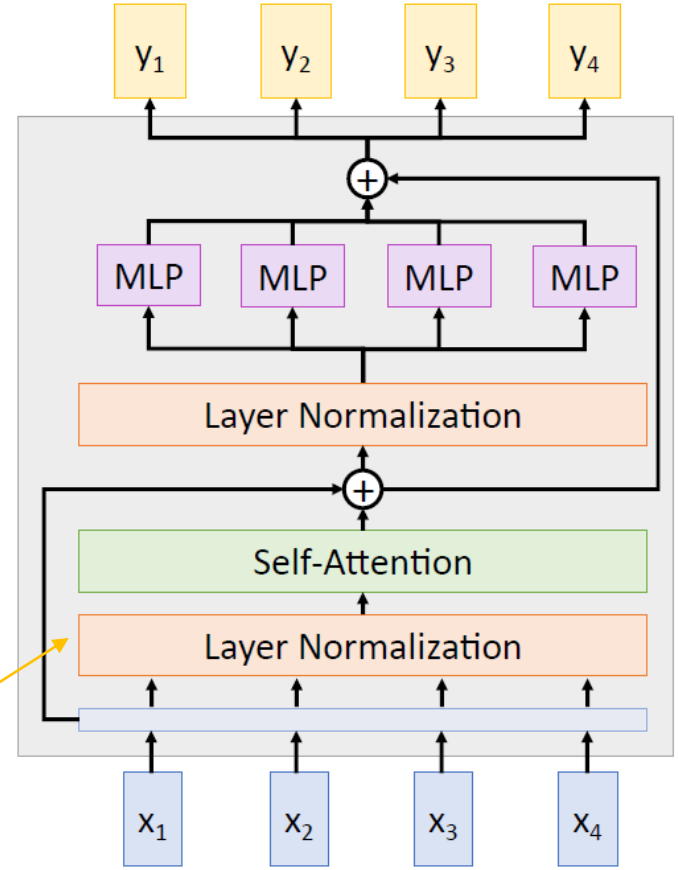


Putting it All Together

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



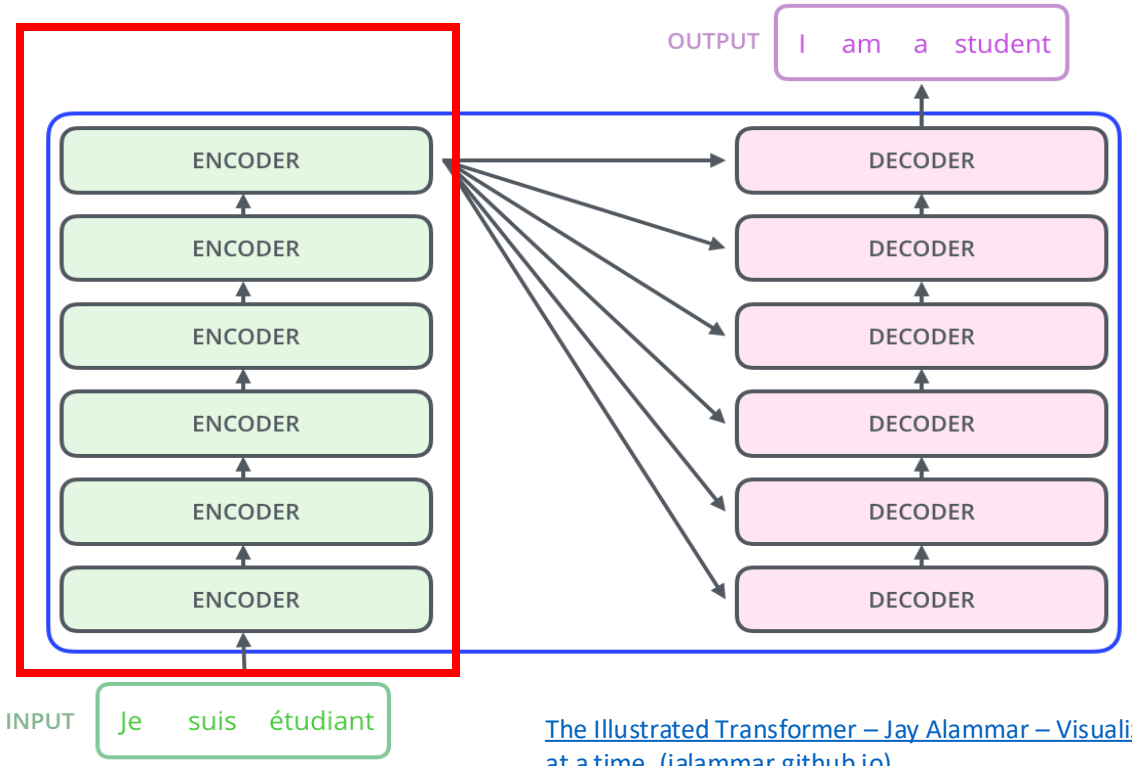
(Johnson, 2022)



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)

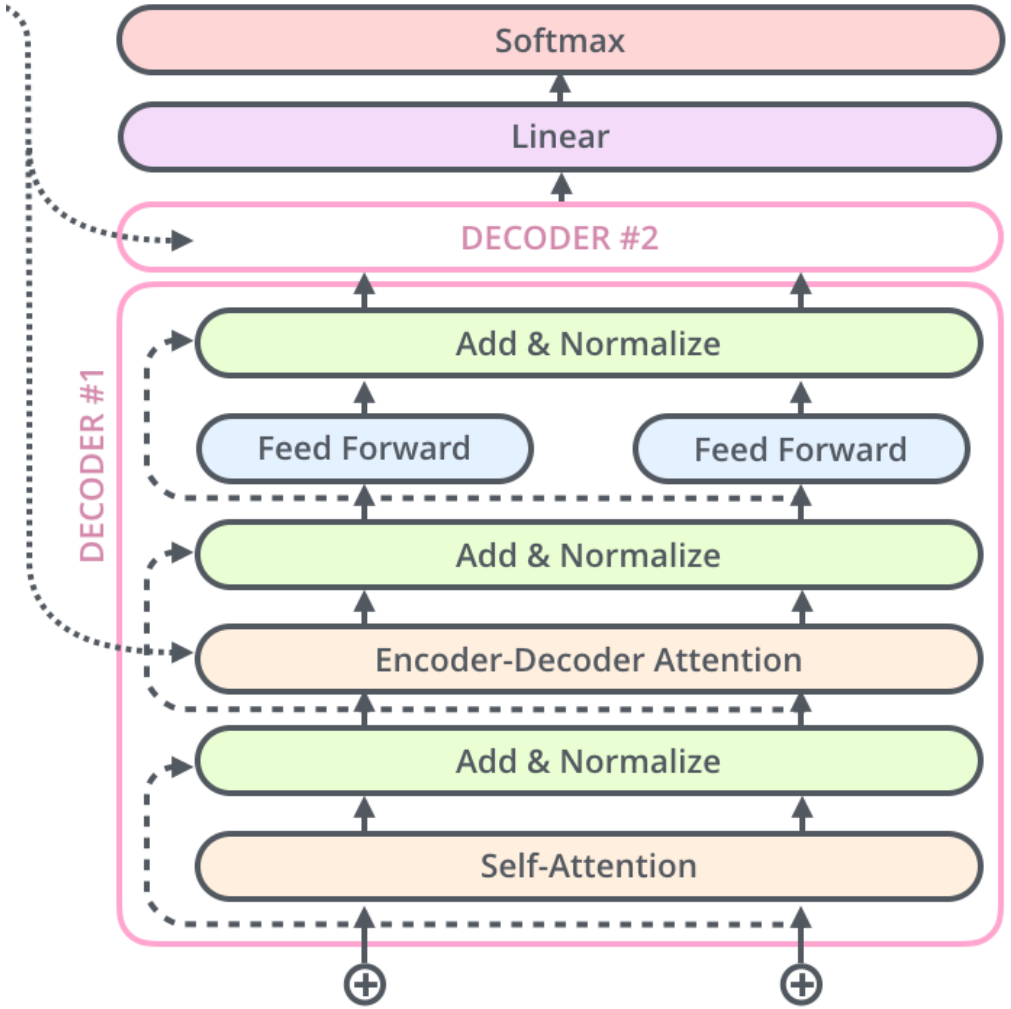
Encoder: Implementation

```python
class TransformerEncoderBlock(nn.Module):
    """The Transformer encoder block."""
    def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout,
                 use_bias=False):
        super().__init__()
        self.attention = MultiHeadAttention(num_hiddens, num_heads,
                                            dropout, use_bias)
        self.addnorm1 = AddNorm(num_hiddens, dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(num_hiddens, dropout)

    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        return self.addnorm2(Y, self.ffn(Y))
```

```python
class TransformerEncoder(Encoder):
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens,
                 num_heads, num_blks, dropout, use_bias=False):
        super().__init__()
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_blks):
            self.blks.add_module("block"+str(i), TransformerEncoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, use_bias))

    def forward(self, X, valid_lens):
        # Since positional encoding values are between -1 and 1, the embedding
        # values are multiplied by the square root of the embedding dimension
        # to rescale before they are summed up
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[
                i] = blk.attention.attention.attention_weights
        return X
```

Transformer Decoder

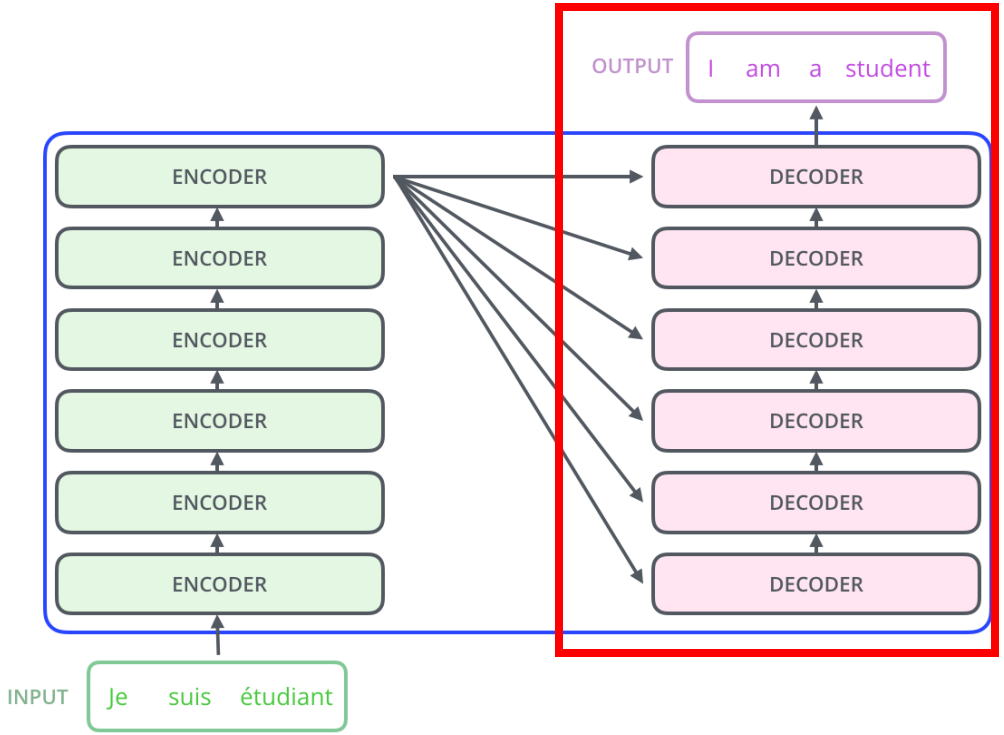
- **Decoder Sublayers:**
  1. Masked Self-Attention: Targets attending to themselves (past tokens).
  2. **Cross-Attention:** Queries from the decoder, Keys and Values from encoder.
  3. Feed-Forward Network: Applies position-wise transformations to each token.
- **Residuals + LayerNorm:**
  - Each sublayer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



Thinking Machines

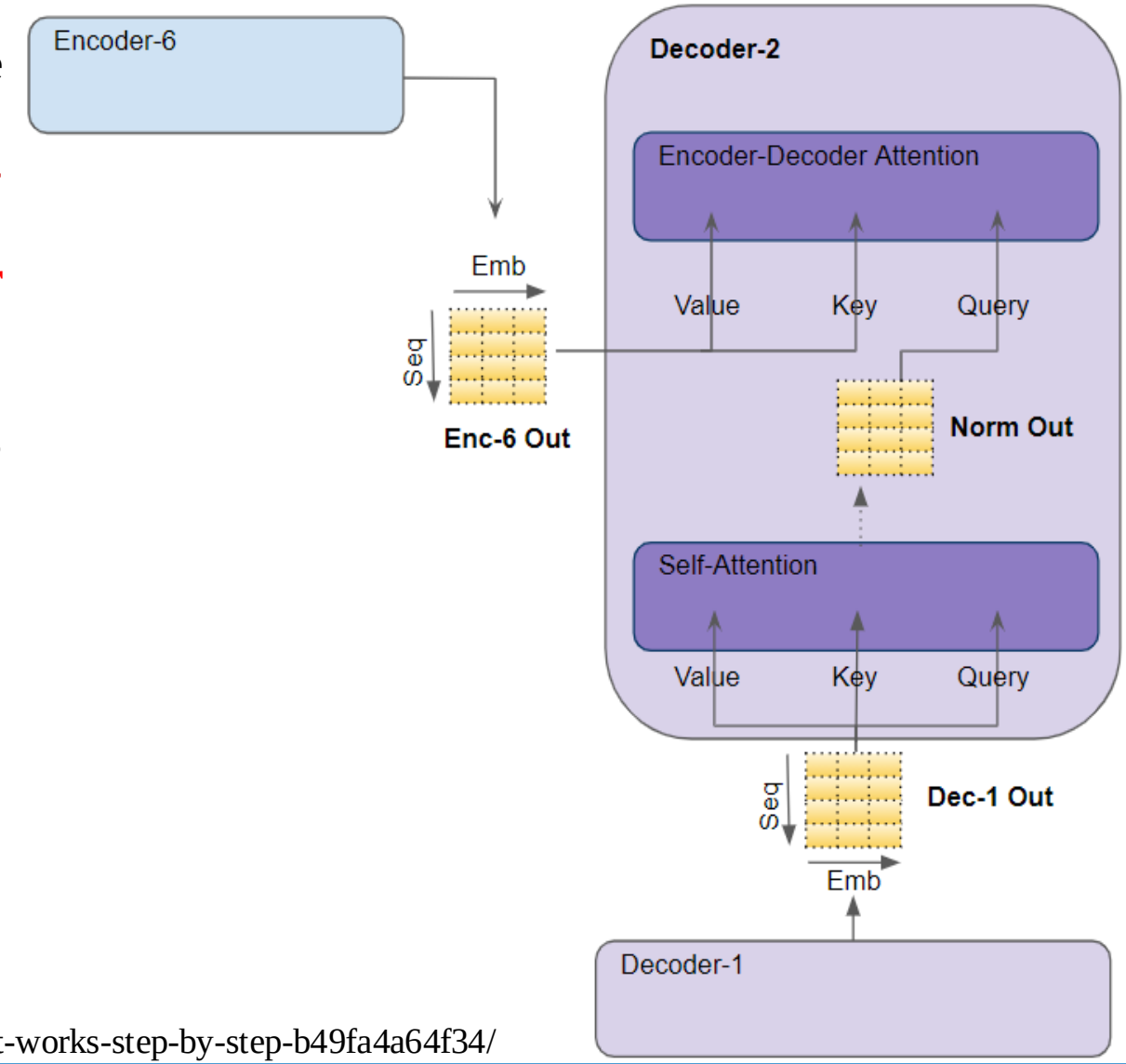
Decoder: Masked Self-Attention

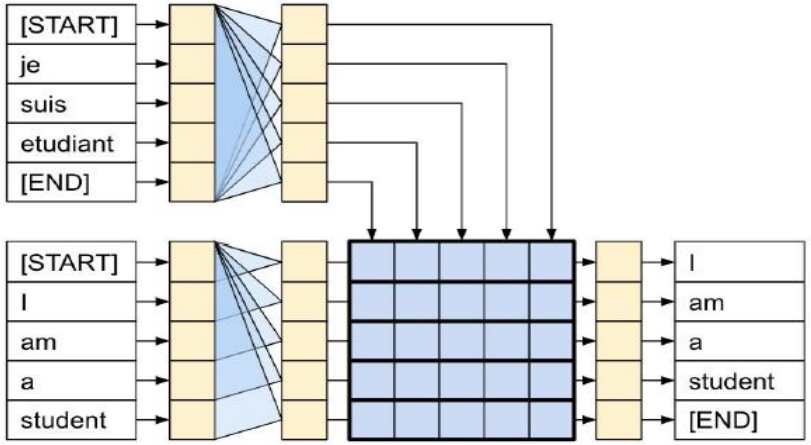
- The Transformer decoder is also composed of multiple layers and generates the output sequence step by step. In the **decoder self-attention**, queries, keys and values are all from the outputs of the previous decoder layer.
- However, each position in the decoder is allowed only to attend to all positions 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: Cross-Attention

- Cross-attention in the decoder allows it to incorporate information from the encoder's output.
- **Queries are from the outputs of the decoder's self-attention sublayer (decoder's hidden states).**
- The **keys and values are from the Transformer encoder outputs**.
- In reality, cross-attention is also multi-headed.
- Such design enables the model to align generated tokens with relevant input features dynamically.

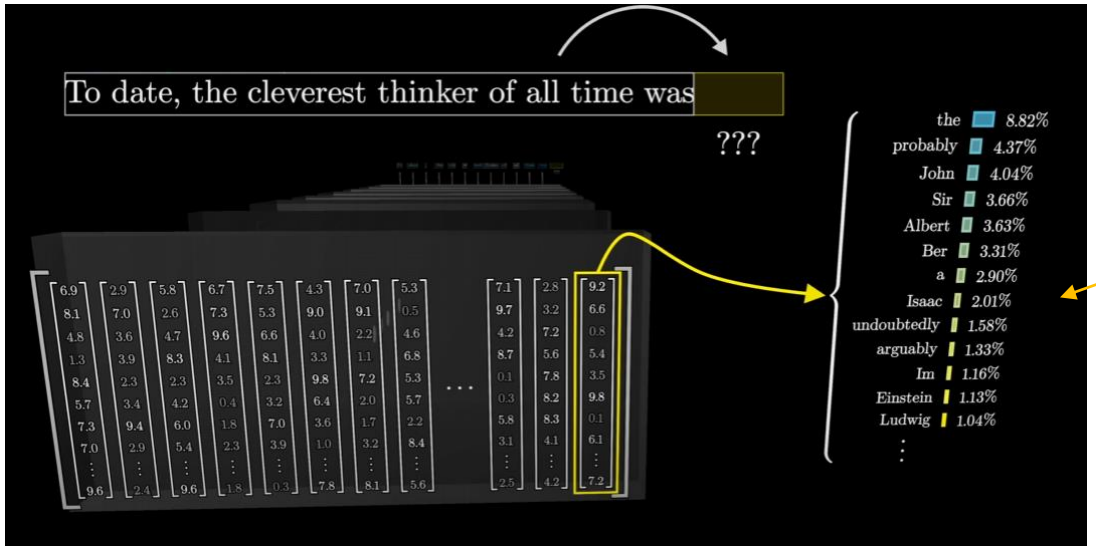


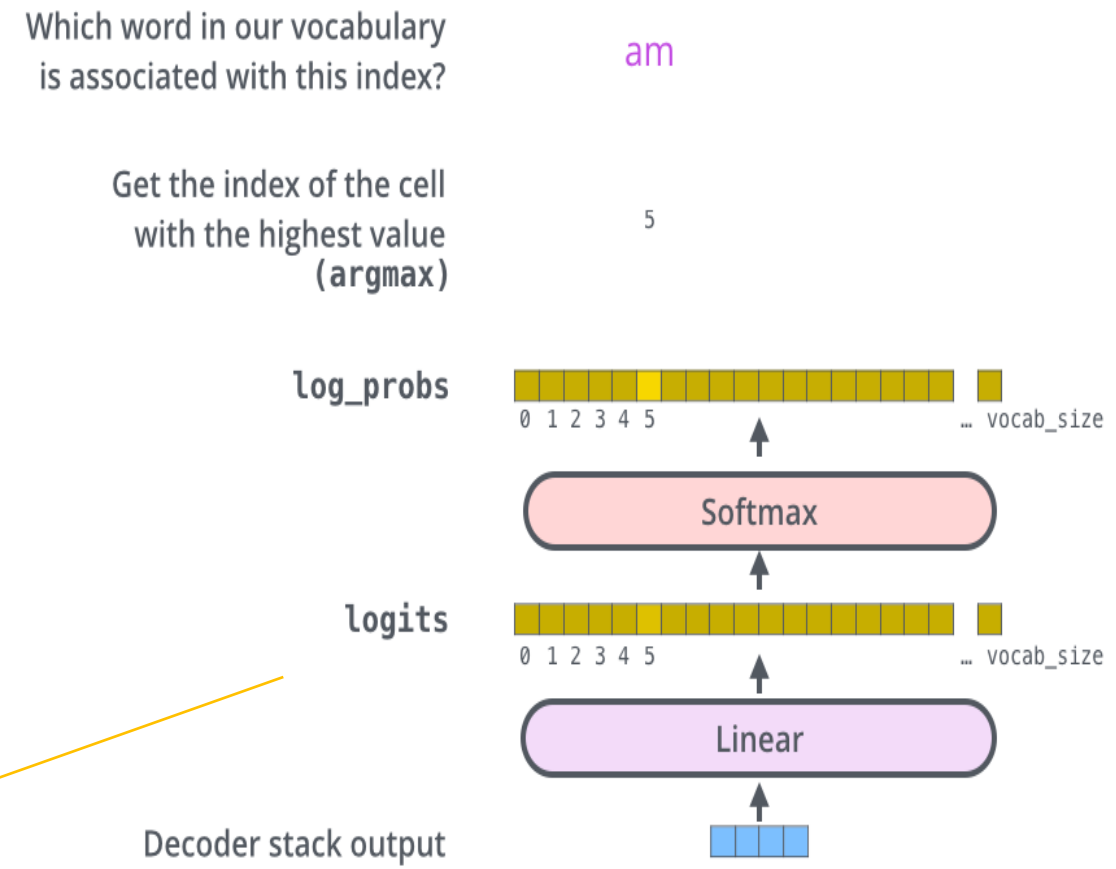


https://towardsdatascience.com/transformers-explained-visually-part-2-how-it-works-step-by-step-b49fa4a64f34/

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.





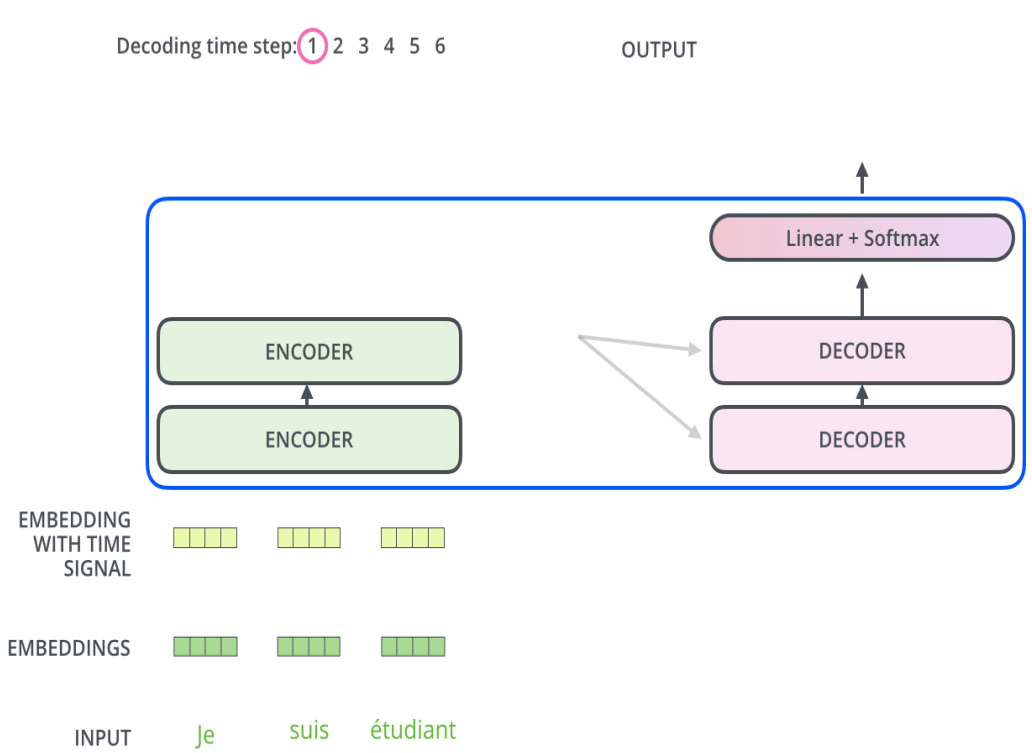
Decoder: Implementation

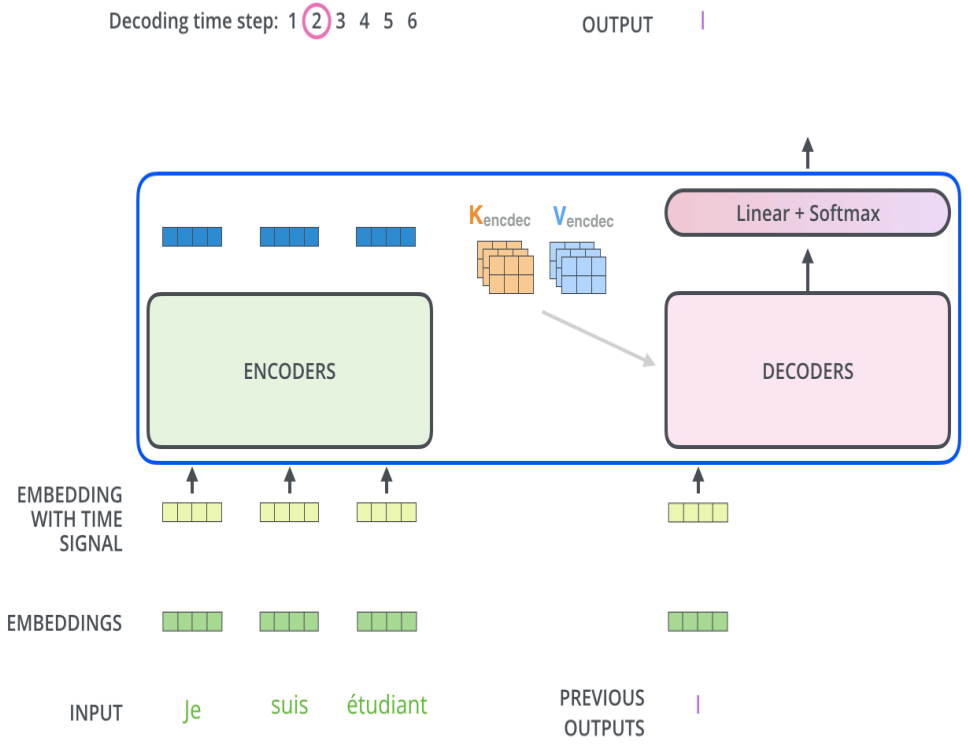
To preserve autoregression in the decoder, the masked self-attention specifies a valid length so that any query only attends to all positions in the decoder up to the query position.

```python
class TransformerDecoderBlock(nn.Module):
    # The i-th block in the Transformer decoder
    def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout, i):
        super().__init__()
        self.i = i
        self.attention1 = MultiHeadAttention(num_hiddens, num_heads,
                                             dropout)
        self.addnorm1 = AddNorm(num_hiddens, dropout)
        self.attention2 = MultiHeadAttention(num_hiddens, num_heads,
                                             dropout)
        self.addnorm2 = AddNorm(num_hiddens, dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm3 = AddNorm(num_hiddens, dropout)

    def forward(self, X, state):
            enc_outputs, enc_valid_lens = state[0], state[1]
            if state[2][self.i] is None:
                key_values = X
            else:
                key_values = torch.cat((state[2][self.i], X), dim=1)
            state[2][self.i] = key_values
            if self.training:
                batch_size, num_steps, _ = X.shape
                dec_valid_lens = torch.arange(
                    1, num_steps + 1, device=X.device).repeat(batch_size, 1)
            else:
                dec_valid_lens = None
            # Self-attention
            X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
            Y = self.addnorm1(X, X2)
            # Encoder-decoder attention. Shape of enc_outputs:
            # (batch_size, num_steps, num_hiddens)
            Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
            Z = self.addnorm2(Y, Y2)
            return self.addnorm3(Z, self.ffn(Z)), state
```

```python
class TransformerDecoder(Decoder):
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                 num_blks, dropout):
        super().__init__()
        self.num_hiddens = num_hiddens
        self.num_blks = num_blks
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_blks):
            self.blks.add_module("block"+str(i), TransformerDecoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, i))
        self.dense = nn.LazyLinear(vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens):
        return [enc_outputs, enc_valid_lens, [None] * self.num_blks]

    def forward(self, X, state):
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self._attention_weights = [[None] * len(self.blks) for _ in range (2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            # Decoder self-attention weights
            self._attention_weights[0][
                i] = blk.attention1.attention.attention_weights
            # Encoder-decoder attention weights
            self._attention_weights[1][
                i] = blk.attention2.attention.attention_weights
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

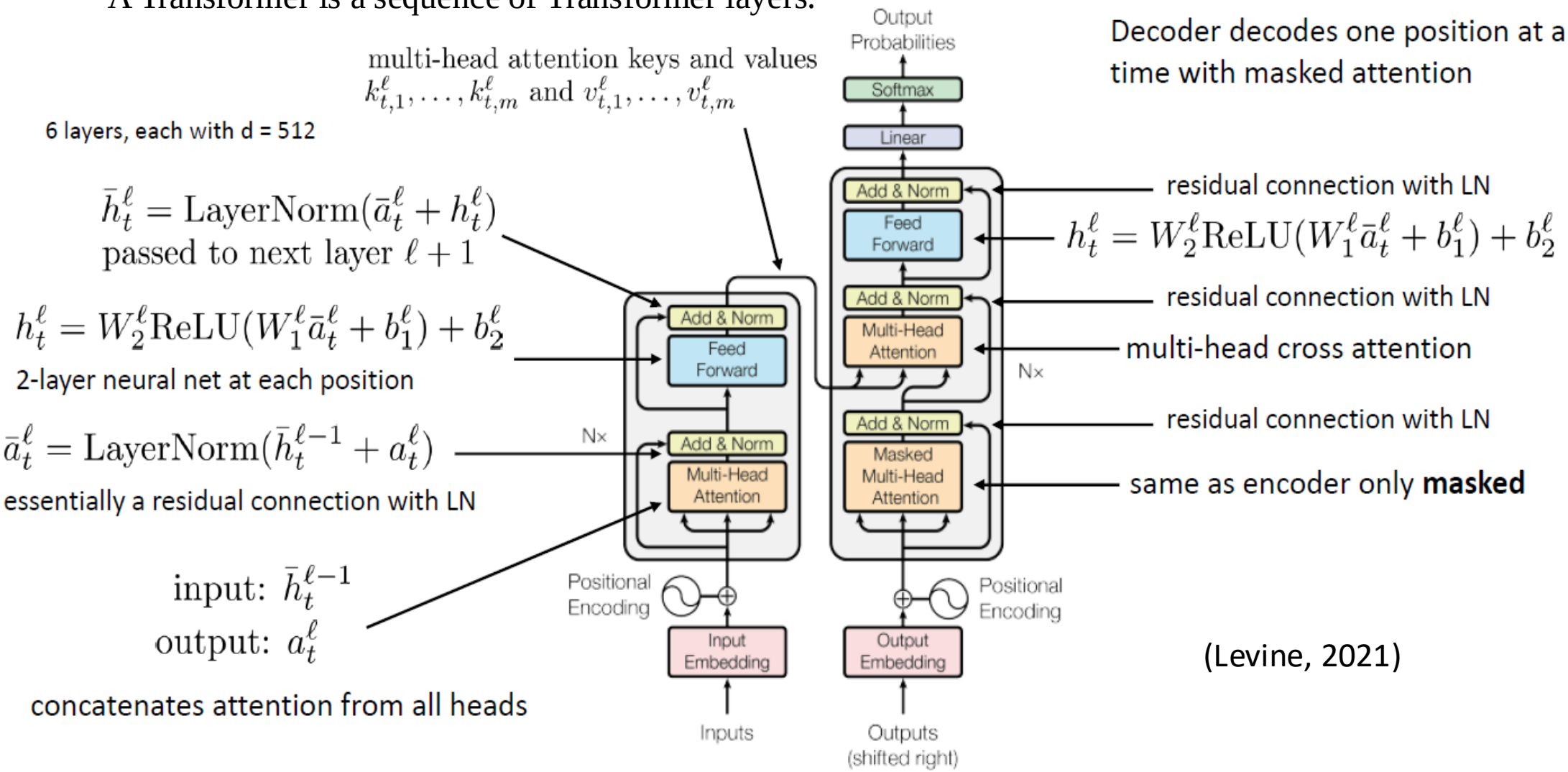
Combine Encoder and Decoder Values: Animation





Transformer: Putting it All Together

A Transformer is a sequence of Transformer layers.



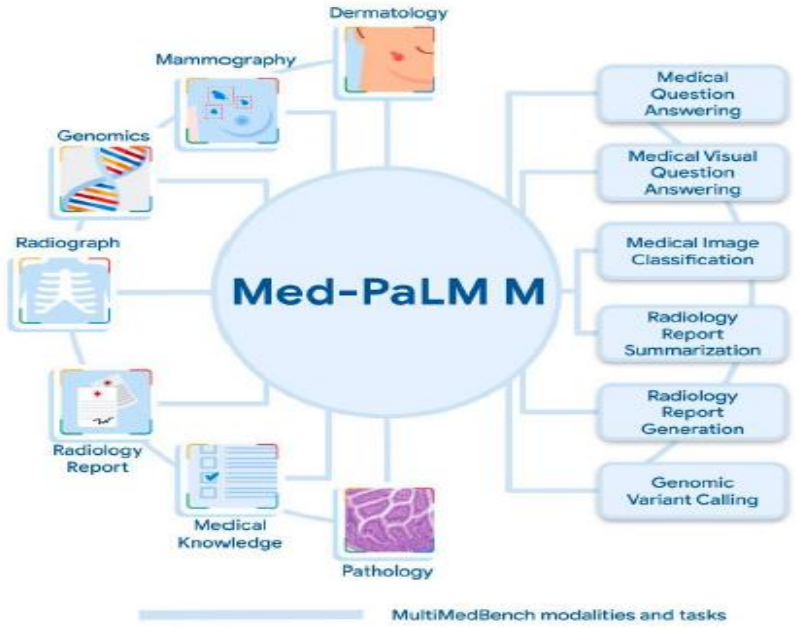
(Levine, 2021)

Content

Transformers are Everywhere Now!

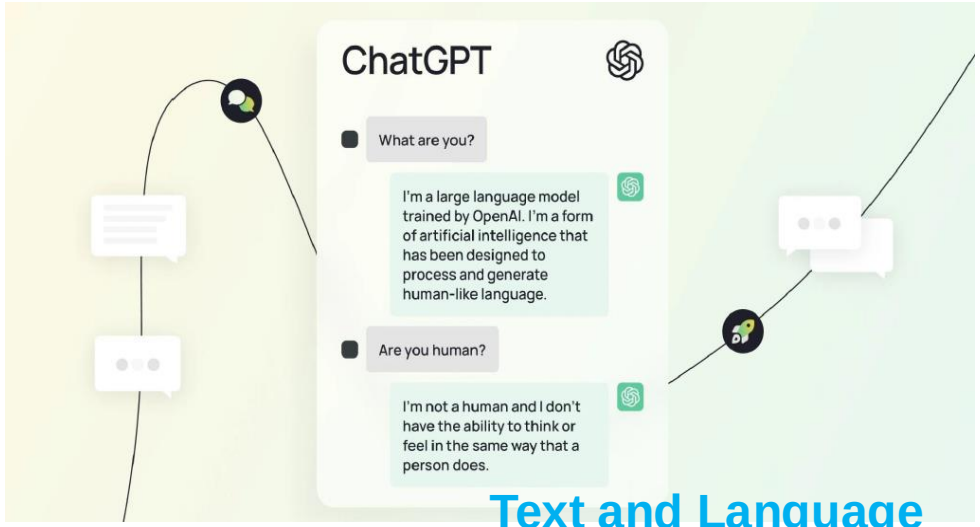
Robotics, Simulations, Physical Tasks

Playing Games

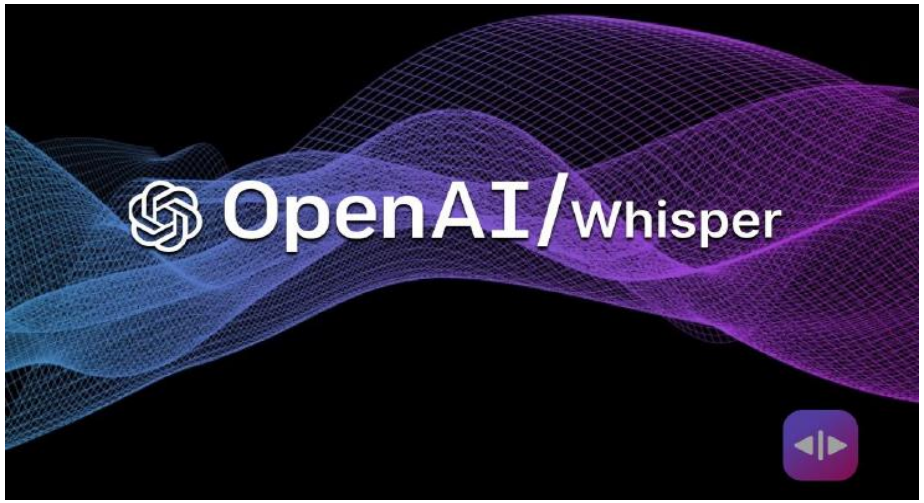


Biology + Healthcare

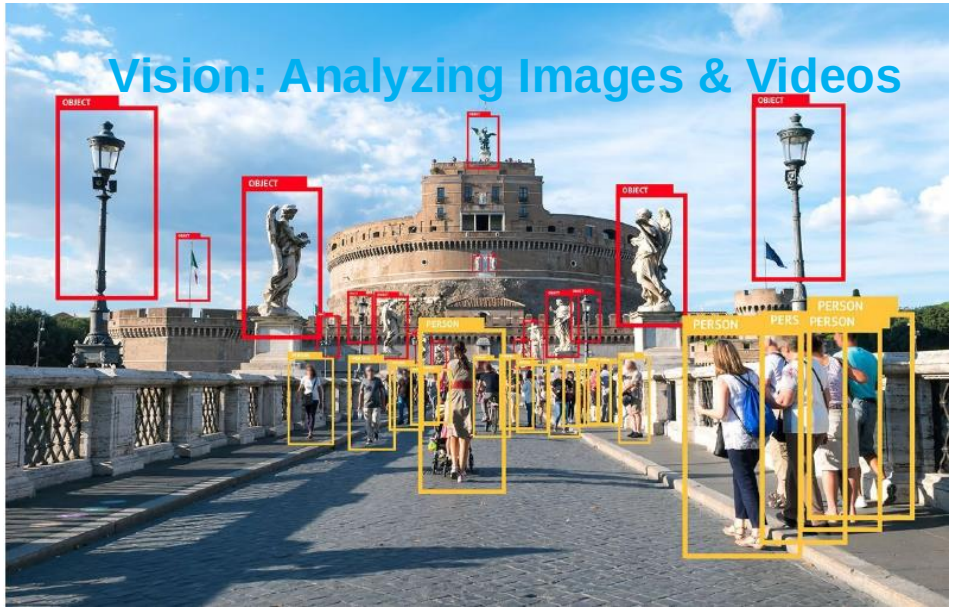
Transformers are Everywhere Now!



Text and Language



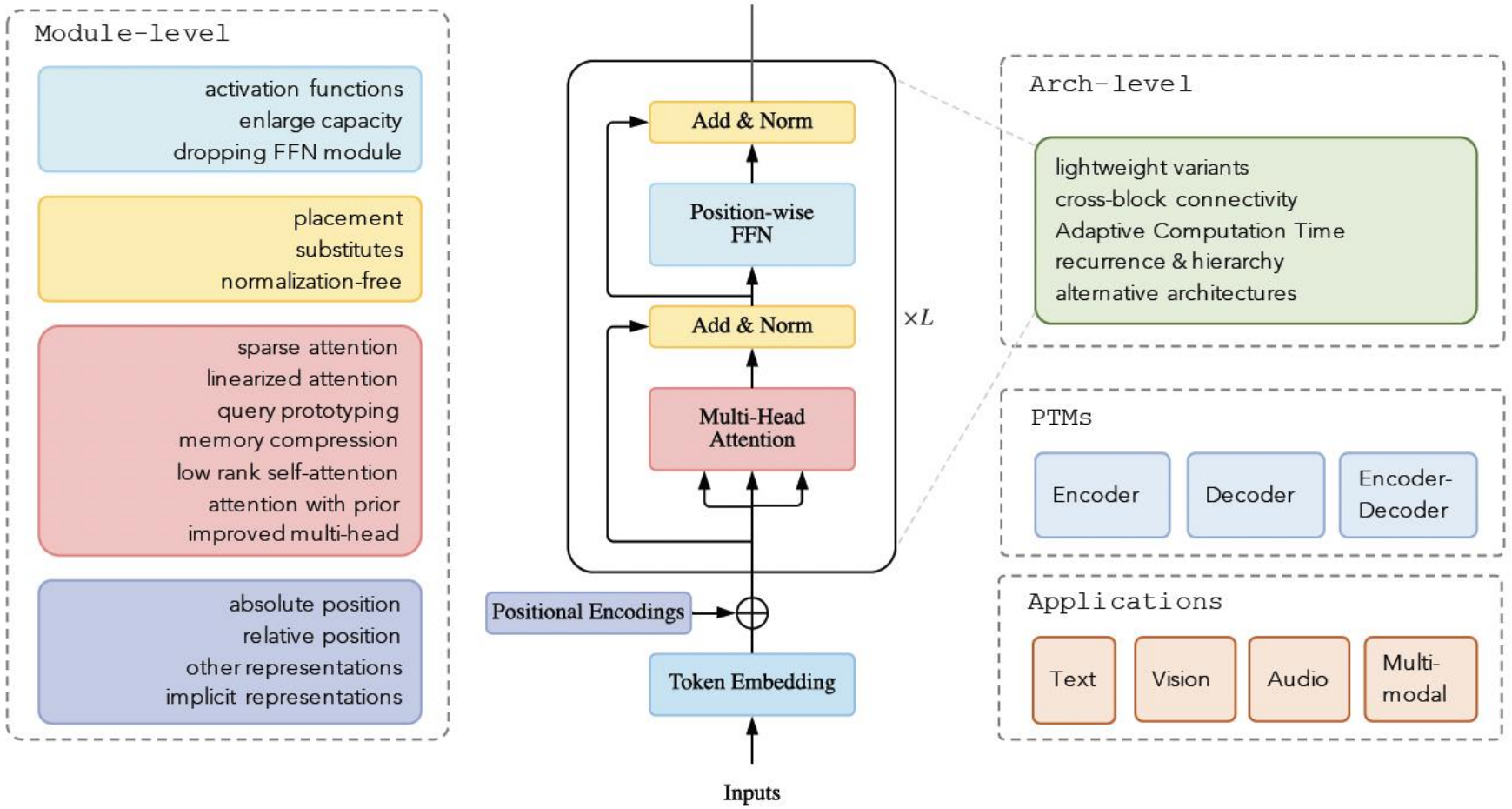
Audio: Speech + Music



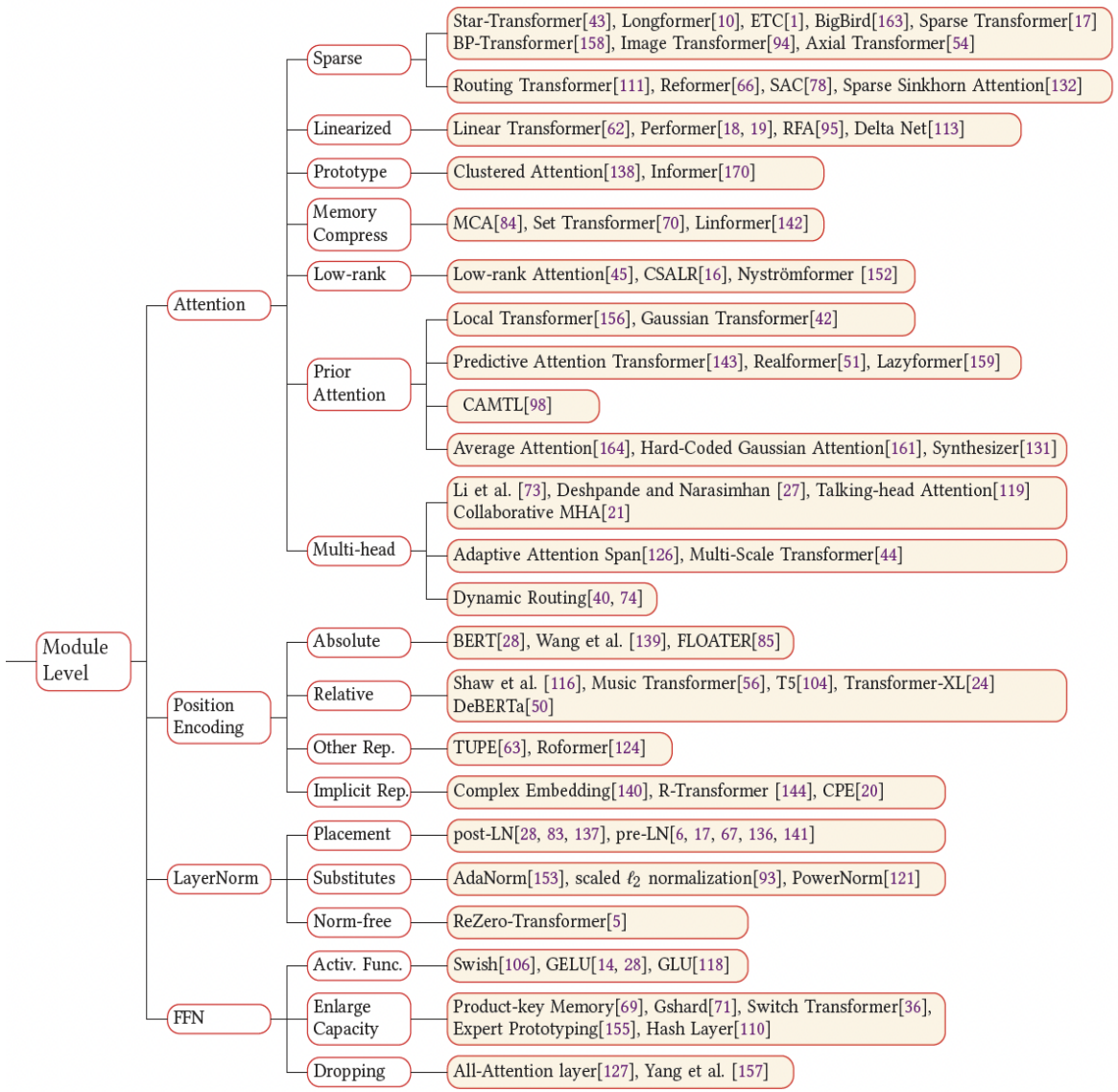
Vision: Analyzing Images & Videos

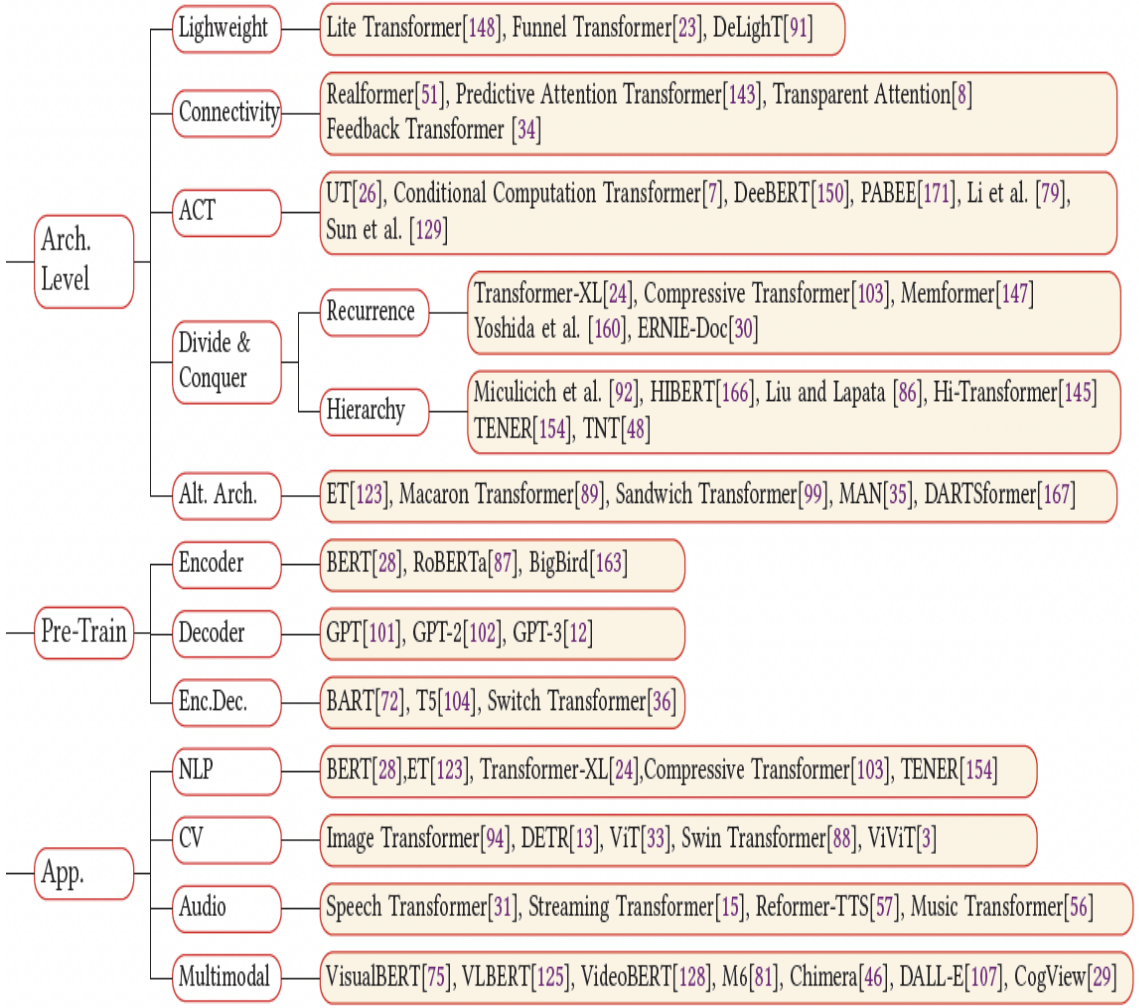
Vision: Generating Images & Video

Categorization of Transformer variants



X-Transformers





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.

Why Transformers?

In practice, this means we can use larger models for the same cost

larger model = better performance

much faster training

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

great translation results

Vaswani et al. **Attention Is All You Need.** 2017.

Text summarization

previous state of the art seq2seq model

Model	Test perplexity	ROUGE-L
seq2seq-attention, L = 500	5.04952	12.7
Transformer-ED, L = 500	2.46645	34.2
Transformer-D, L = 4000	2.22216	33.6
Transformer-DMCA, no MoE-layer, L = 11000	2.05159	36.2
Transformer-DMCA, MoE-128, L = 11000	1.92871	37.9
Transformer-DMCA, MoE-256, L = 7500	1.90325	38.8

lower is better (this metric is similar to 1/likelihood)

We'll learn more about the power of transformers as **language models** next time!

Liu et al. **Generating Wikipedia by summarizing long sequences.** 2018.

(Levine, 2021)

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.

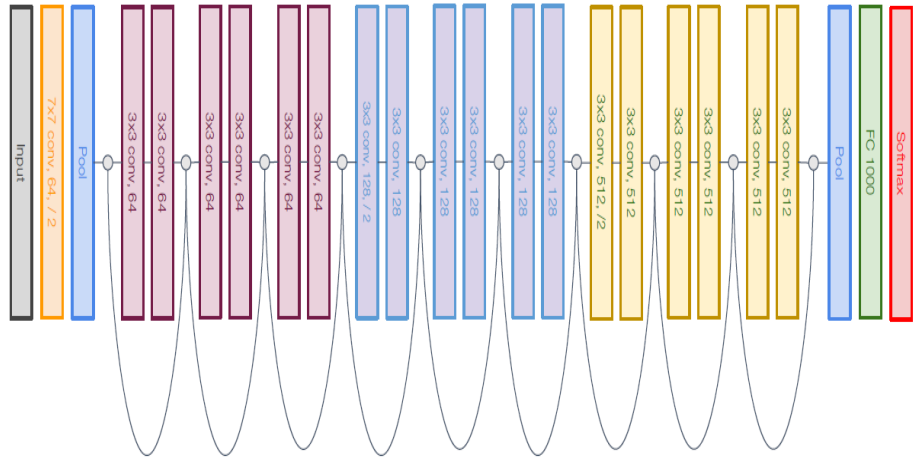
Process:

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

Attention/Transformers for Vision

Idea #1: Add attention to existing CNNs

- Start from standard CNN architecture (e.g. ResNet)
- Add Self-Attention blocks between existing ResNet blocks



Key Idea:

▶ Introduce long-range self-attention into GANs for image generation.

▶ Traditional conv-based GANs rely on spatially local features.

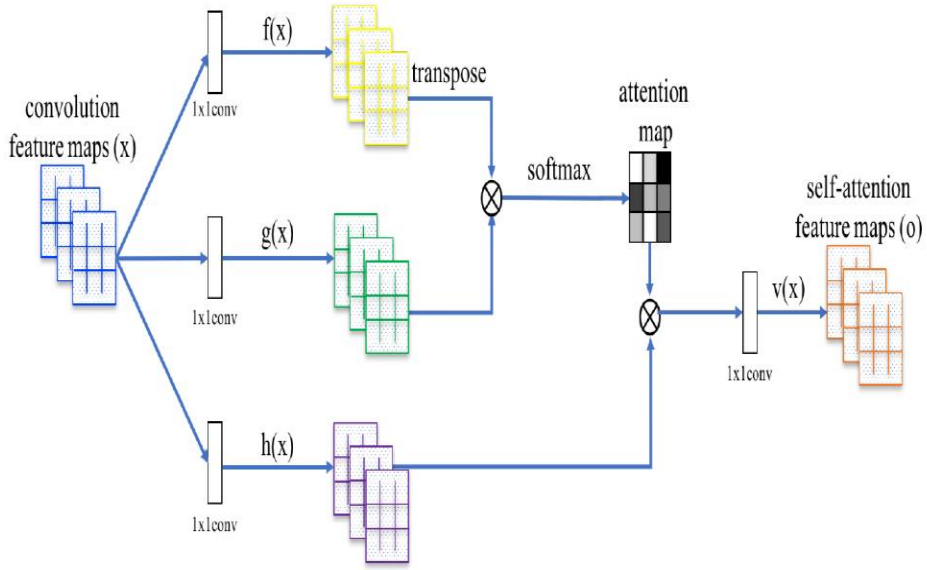
▶ SAGAN allows any feature location to influence the generation of high-resolution details.

Results on ImageNet:

▶ Inception score boosted from **36.8 to 52.52.**

▶ Frechet Inception Distance (FID) reduced from **27.62 to 18.65.**

▶ Visualization shows generator attends to object-like shapes, not just local patches.



Zhang et al, "Self-Attention Generative Adversarial Networks", ICML 2018
Wang et al, "Non-local Neural Networks", CVPR 2018

Attention/Transformers for Vision

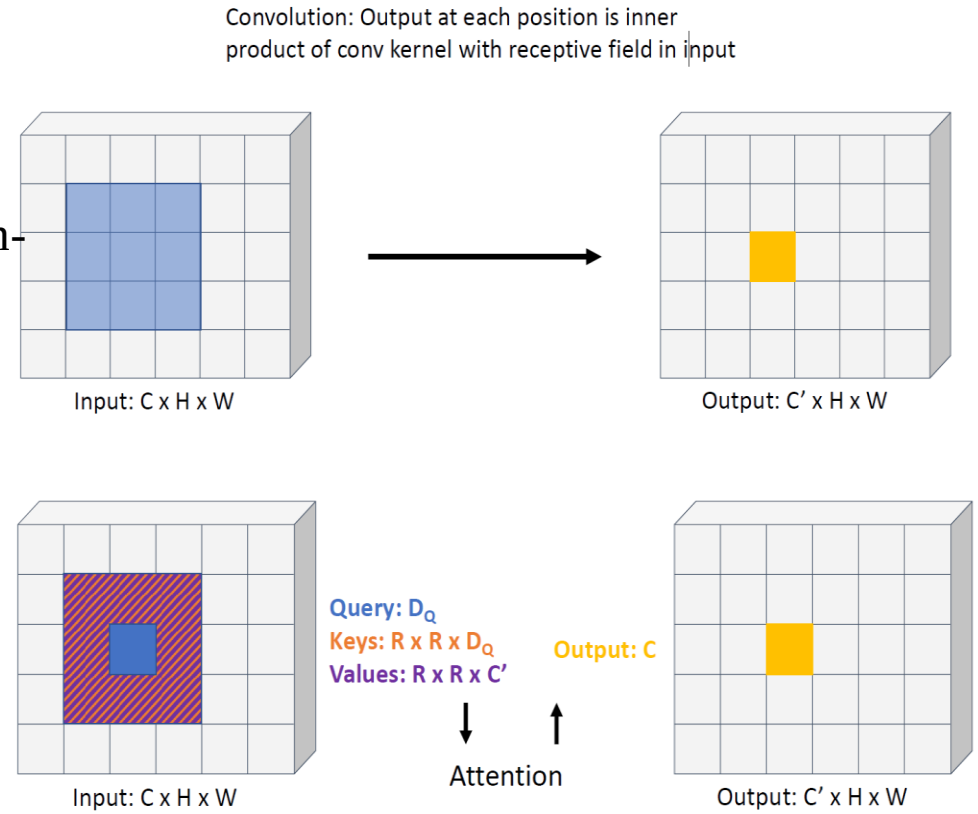
Idea #2: Replace convolution entirely

Beyond Convolutions:

- Convolutions excel at local feature extraction.
- Self-attention is especially beneficial in later layers and long-range dependencies require content-based interactions (e.g., self-attention, non-local blocks).

Key question: Can self-attention fully replace spatial convolutions?

- Replacing Convs with Self-Attention:
- Modify ResNet by swapping each spatial convolution for a self-attention module.
- Gains on ImageNet: outperforms baseline with 12% fewer FLOPs and 29% fewer parameters.
- On COCO detection, pure self-attention matches baseline mAP with 39% fewer FLOPs and 34% fewer parameters.



Hu et al, "Local Relation Networks for Image Recognition", ICCV 2019;
Ramachandran et al, "Stand-Alone Self-Attention in Vision Models", NeurIPS 2019

Attention/Transformers for Vision

Unfortunately, the performance is not satisfactory:

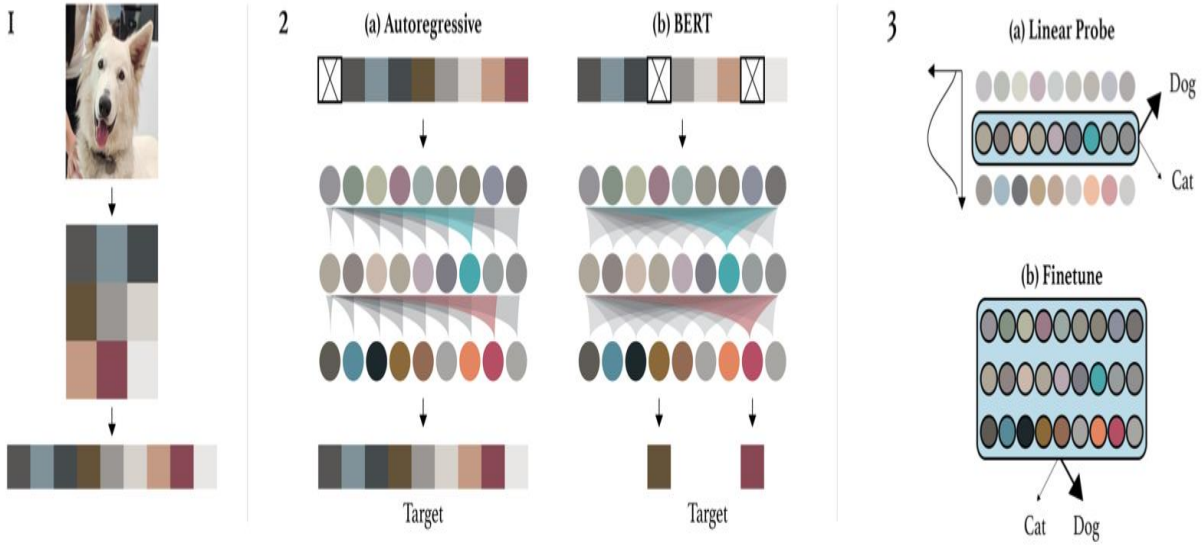
- Lots of tricky details
- Hard to implement
- Only marginally better than ResNets

stage	output	ResNet-50	**LR-Net-50 (7×7, m=8)**
res1	112×112	7×7 conv, 64, stride 2	**1×1, 64** **7×7 LR, 64, stride 2**
res2	56×56	3×3 max pool, stride 2	3×3 max pool, stride 2
		[1×1, 64 3×3 conv, 64 1×1, 256] ×3	[1×1, 100 **7×7 LR, 100** 1×1, 256] ×3
res3	28×28	[1×1, 128 3×3 conv, 128 1×1, 512] ×4	[1×1, 200 **7×7 LR, 200** 1×1, 512] ×4
res4	14×14	[1×1, 256 3×3 conv, 256 1×1, 1024] ×6	[1×1, 400 **7×7 LR, 400** 1×1, 1024] ×6
res5	7×7	[1×1, 512 3×3 conv, 512 1×1, 2048] ×3	[1×1, 800 **7×7 LR, 800** 1×1, 2048] ×3
	1×1	global average pool 1000-d fc, softmax	global average pool 1000-d fc, softmax
# params		**25.5**×10^6	**23.3**×10^6
FLOPs		**4.3**×10^9	**4.3**×10^9

Attention/Transformers for Vision

Idea #3: Standard Transformer on Pixels

- Treat an image as a set of pixel values, and then feed it as input to standard Transformer
- Problem: Too much memory usage! An $R \times R$ image requires R^4 elements per attention matrix. Then for a 128×128 image with 48 layers, 16 heads per layer, it would take 768GB for attention matrices!



▶ **Idea:** Apply a similar auto-regressive Transformer to pixels without explicit 2D priors.

▶ **Setup:**

▶ Trained on low-resolution ImageNet in a purely auto-regressive manner. No 2D convolutions, treats images as 1D sequences.

▶ **Results on CIFAR-10:**

▶ 96.3% accuracy with a linear probe, beating a supervised Wide ResNet.

▶ 99.0% accuracy when fully fine-tuned, matching top supervised pretrained models.

Self-Supervised Benchmarks on ImageNet:

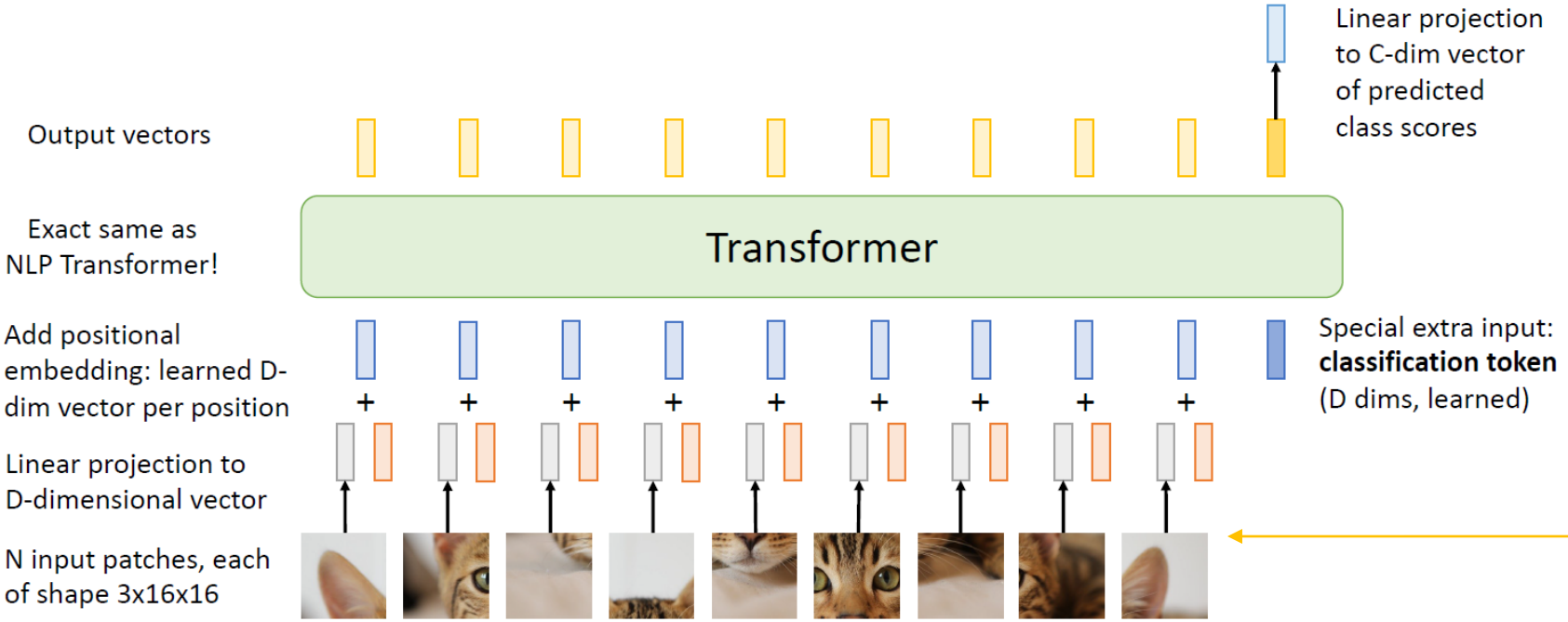
▶ Demonstrates strong learned representations even without explicit image patches.

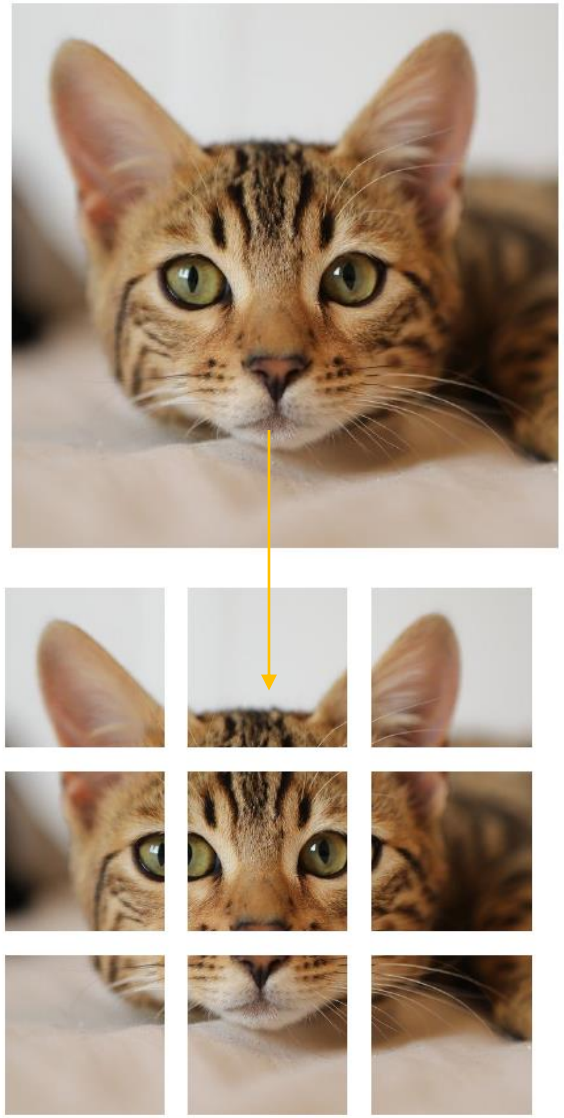
Chen et al, "Generative Pretraining from Pixels", ICML 2020

Vision Transformers (ViT)

Idea #4: Standard Transformer on **Patches**

In practice, we take 224x224 input image, divide into 14x14 grid of 16x16 pixel patches . Each attention matrix then takes around 150 KB.



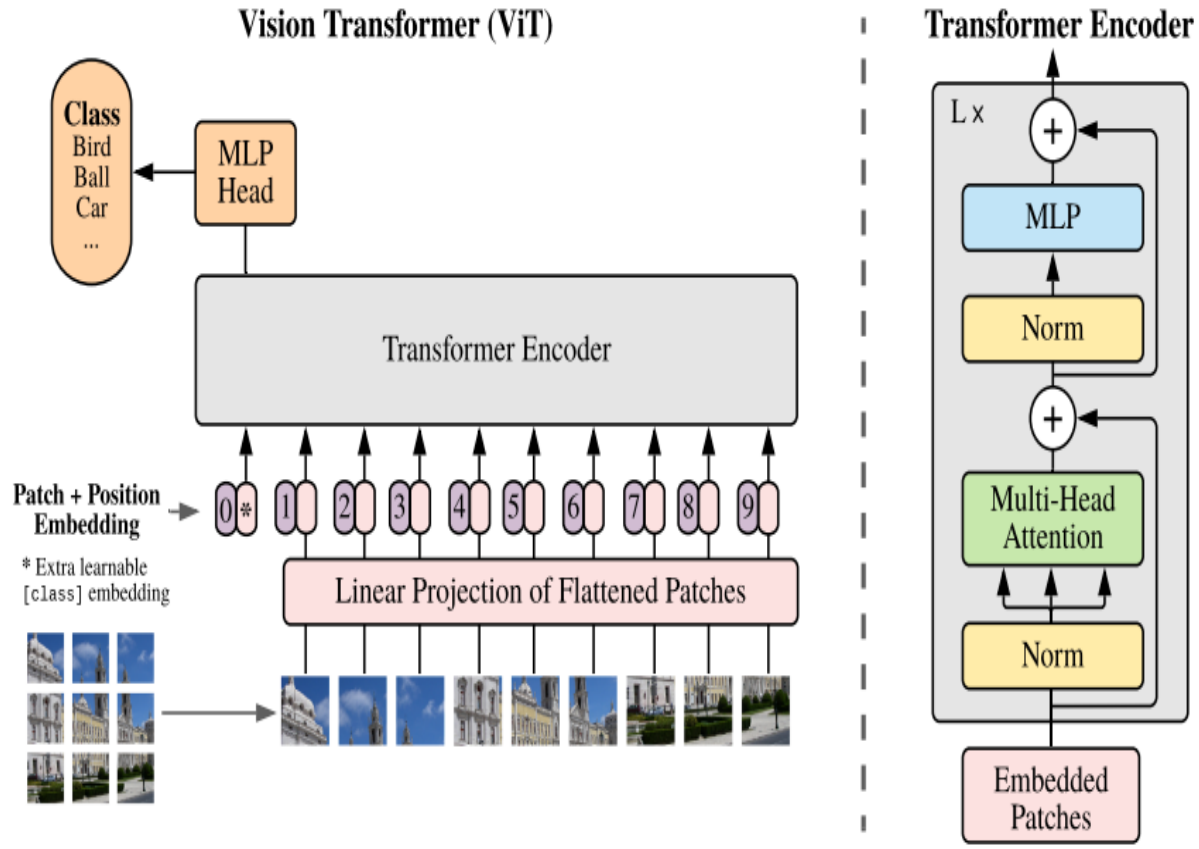


(Johnson, 2022)

Dosovitskiy et al, “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”, ICLR 2021

Vision Transformers (ViT)

- In the field of vision, the reliance on CNN is not necessary and a pure transformer applied directly can perform very well on image classification tasks.
- Vision Transformer(ViT) try to do fewest possible modification by splitting image into patches and provide the sequence of linear embeddings of these patches as input.
- Unlike prior works, ViT doesn't introduce specific inductive biases into architecture except the initial patch extraction step. Instead, image is treated as a sequence of patches and is processed by a standard Transformer encoder
- Convolutional inductive bias is useful for smaller datasets; while for larger ones, learning the relevant patterns is even beneficial.

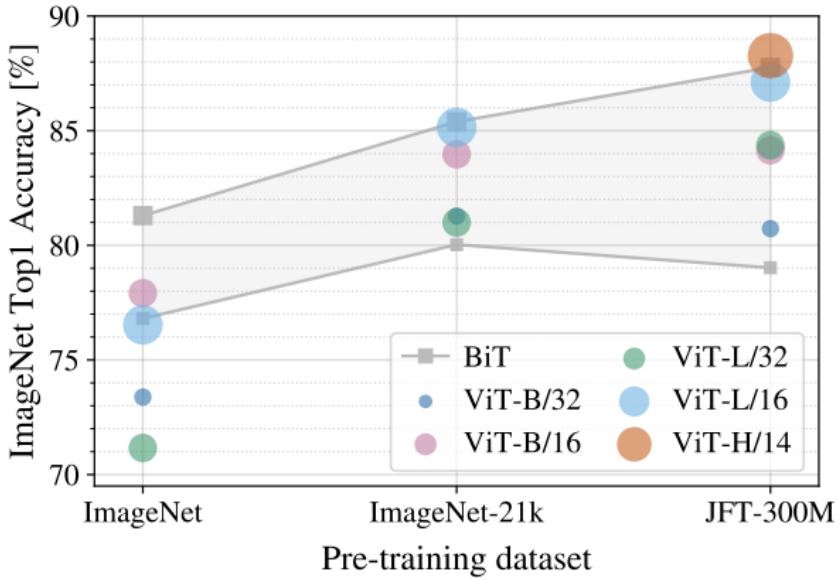


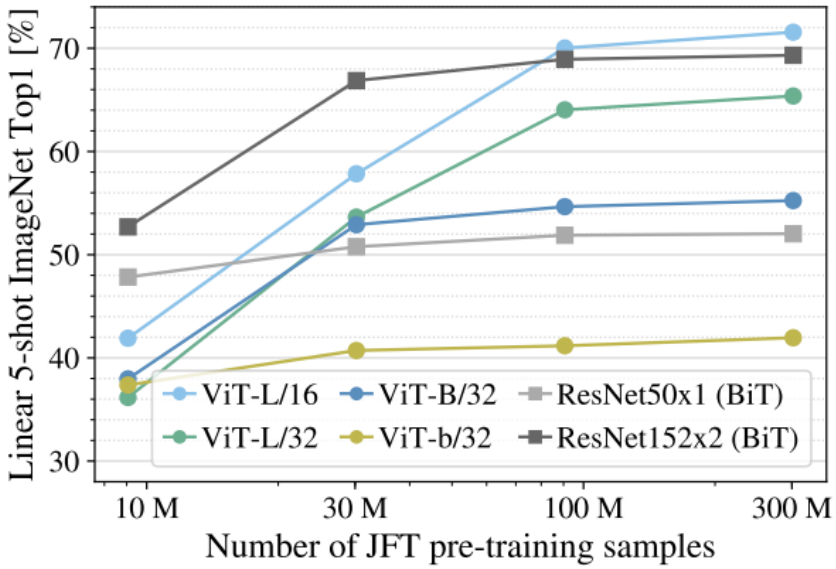
Vision Transformers (ViT)

- When trained on ImageNet, ViT models perform worse than ResNets.
- If you pretrain on JFT and finetune on ImageNet, large ViTs outperform large ResNets.
- ViT can make more efficient use of GPU/TPU (tensor processing unit) hardware, as matrix multiplication is more hardware-friendly than convolution.

B = Base
L = Large
H = Huge

/32, /16, /14 is patch size;
smaller patch size is a bigger model (more patches)



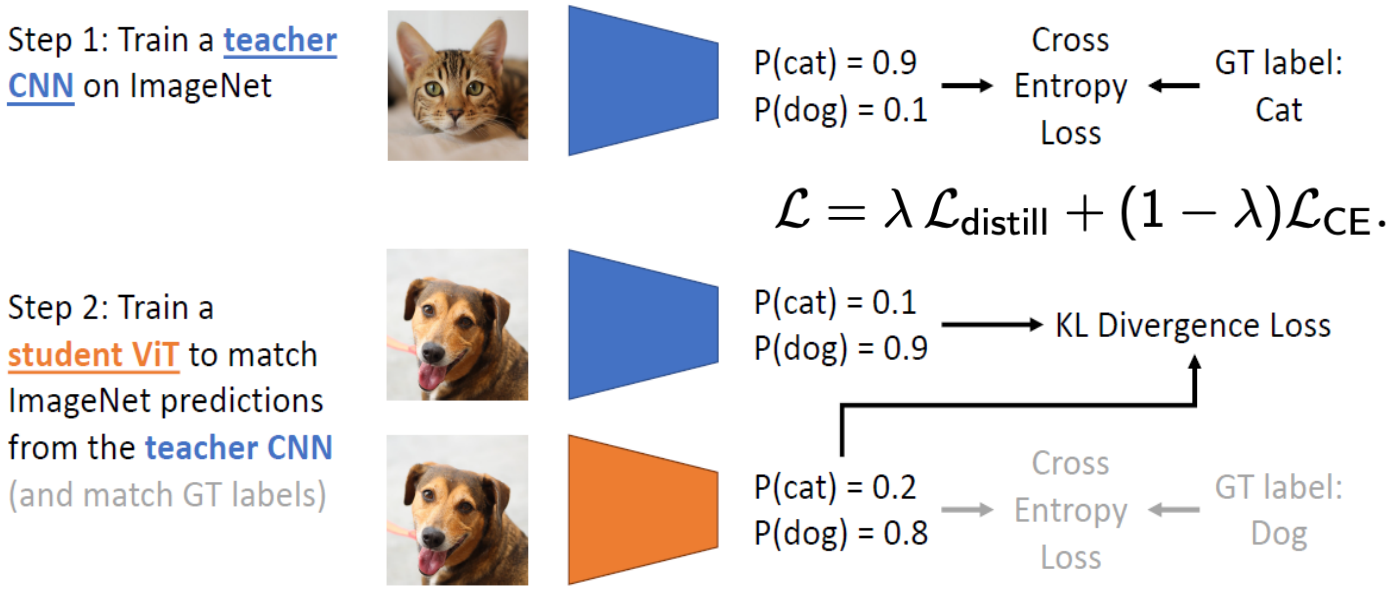


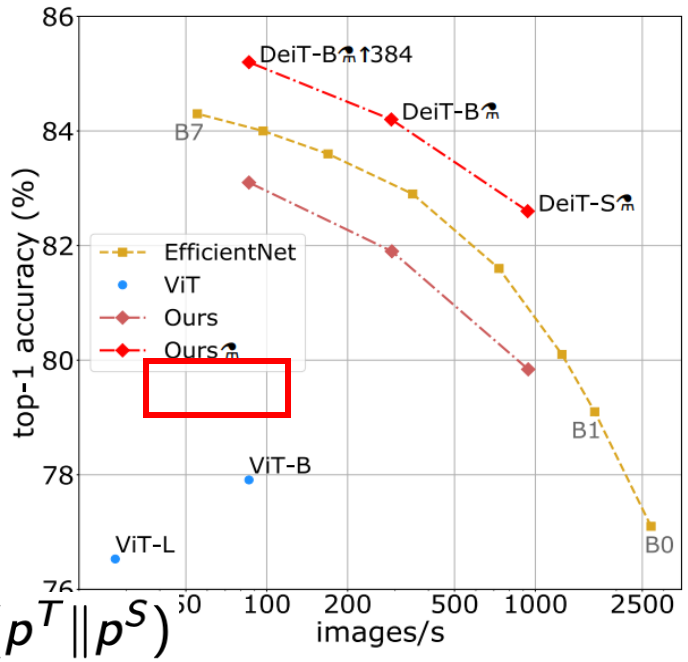
Improving ViT: Distillation (DeiT)

Data-efficient Image Transformer (DeiT) is an improved version of the ViT, designed to achieve high performance with smaller datasets and less computational cost.

DeiT is trained entirely on ImageNet-1k by leveraging knowledge distillation. It introduces a distillation token that learns from a pre-trained teacher model (e.g., a CNN), allowing the Transformer to benefit from the teacher's soft labels and achieve better generalization.

Additionally, DeiT also employs advanced data augmentation and efficient training strategies, enabling faster convergence and competitive performance without the need for external data.





Touvron et al, "Training data-efficient image transformers & distilla $\mathcal{L}_{\text{distill}} = \tau^2 \cdot \text{KL}(p^T \| p^S)$ attention", ICML 2021

Scaling (ViT-22B)

Three main modifications to the original ViT:

- Instead of sequentially applying self-attention and MLP blocks, put them in parallel for additional parallelization.
- Query/Key normalization to ensure stable gradient.
- Omit bias term for QKV projections to accelerate utilization

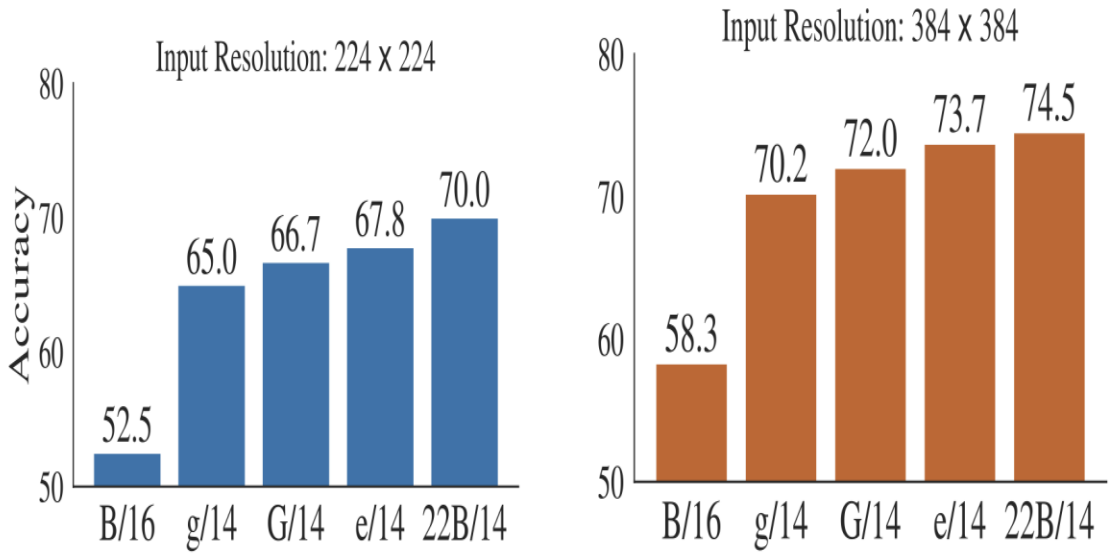


Table 1: ViT-22B model architecture details.

Name	Width	Depth	MLP	Heads	Params [M]
ViT-G	1664	48	8192	16	1843
ViT-e	1792	56	15360	16	3926
ViT-22B	6144	48	24576	48	21743

Table 3: Zero-shot transfer results on ImageNet (variants).

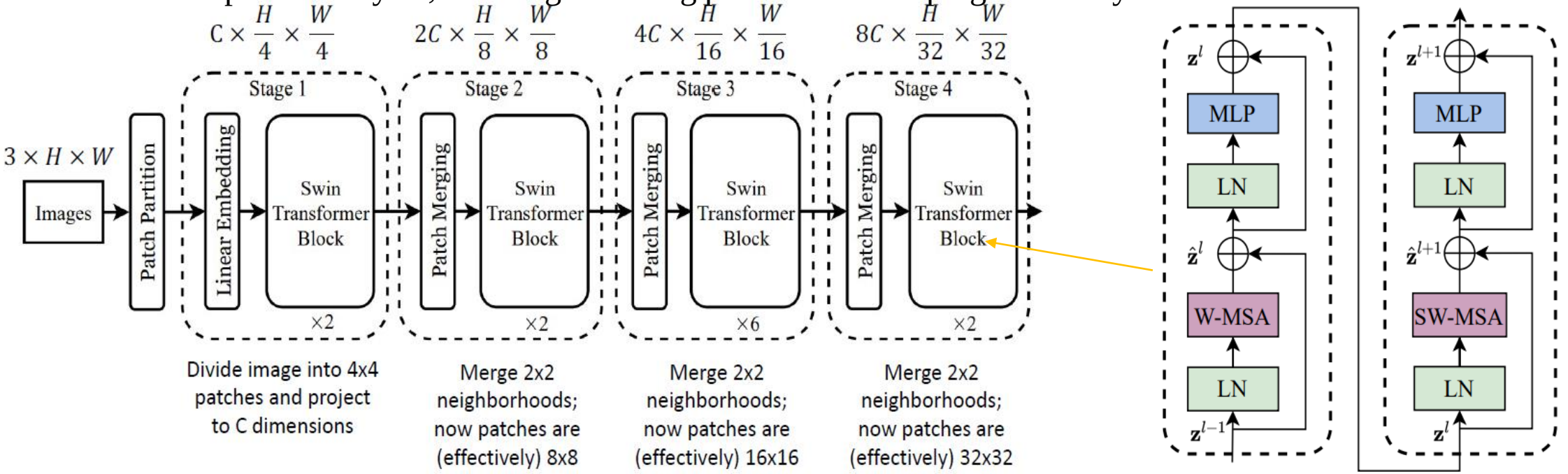
Model	IN	IN-v2	IN-R	IN-A	ObjNet	ReaL
CLIP	76.2	70.1	88.9	77.2	72.3	-
ALIGN	76.4	70.1	92.2	75.8	72.2	-
BASIC	85.7	80.6	95.7	85.6	78.9	-
CoCa	86.3	80.7	96.5	90.2	82.7	-
LiT-g/14	85.2	79.8	94.9	81.8	82.5	88.6
LiT-e/14	85.4	80.6	96.1	88.0	84.9	88.4
LiT-22B	85.9	80.9	96.0	90.1	87.6	88.6

Dehghani, Mostafa, et al. "Scaling vision transformers to 22 billion parameters." *International Conference on Machine Learning*. PMLR, 2023.

Swin Transformer

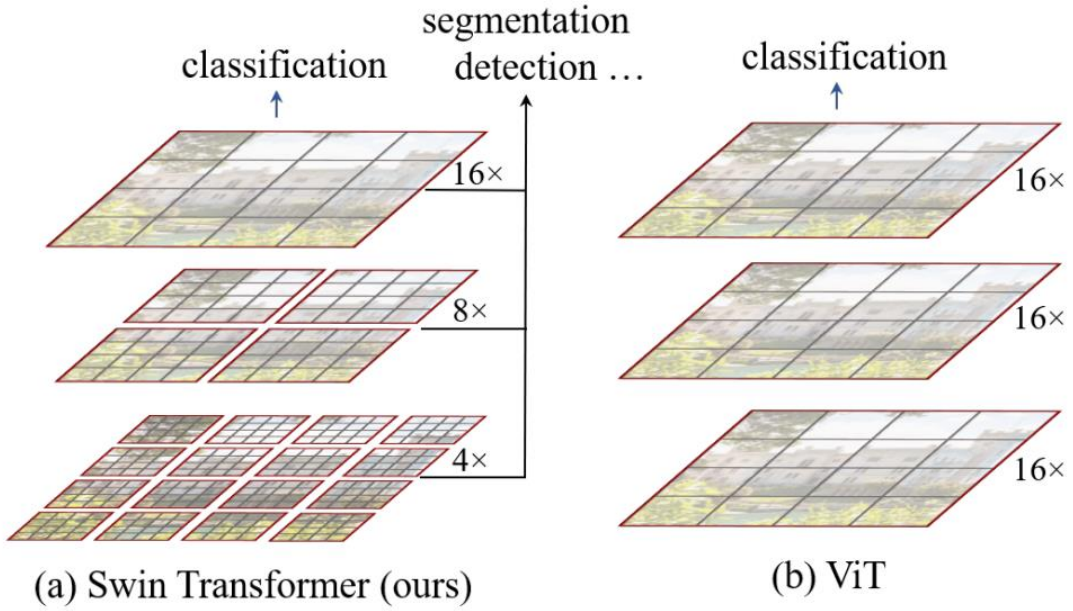
Swin Transformer highlights how a carefully adapted Transformer architecture—with local windows, shifting, and hierarchical stages—can match or exceed CNN performance in classification, detection, and segmentation, paving the way for broader adoption of Transformers as a universal backbone in computer vision.

- **Hierarchical Feature Representation:** Splits the input image into patches and constructs feature maps of progressively lower resolution. This design enables the model to handle **multi-scale visual entities** effectively and to integrate seamlessly with common dense prediction frameworks (e.g., FPN, U-Net).
- **Shifted Window Partitioning** between consecutive layers lets each window see content from neighboring windows in previous layers, boosting modeling power while keeping efficiency.

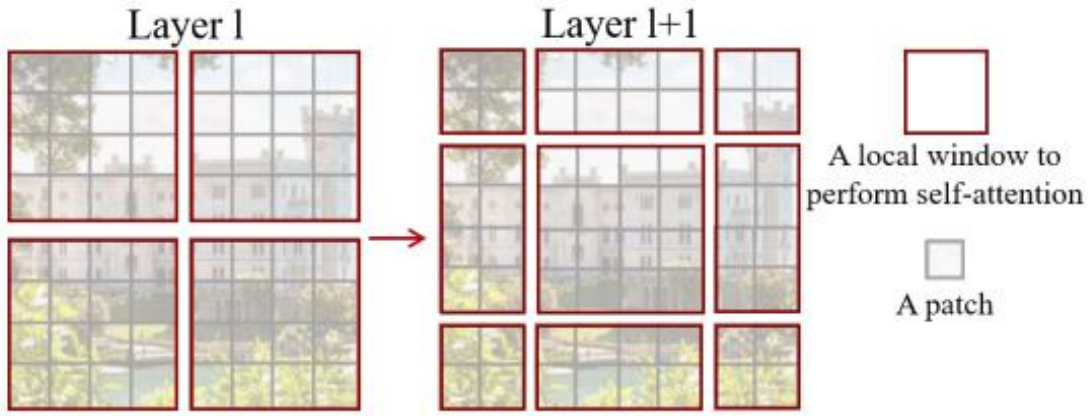


Liu, Ze, et al. "Swin transformer: Hierarchical vision transformer using shifted windows." *Proceedings of the IEEE/CVF international conference on computer vision*. 2021.

Swin Transformer

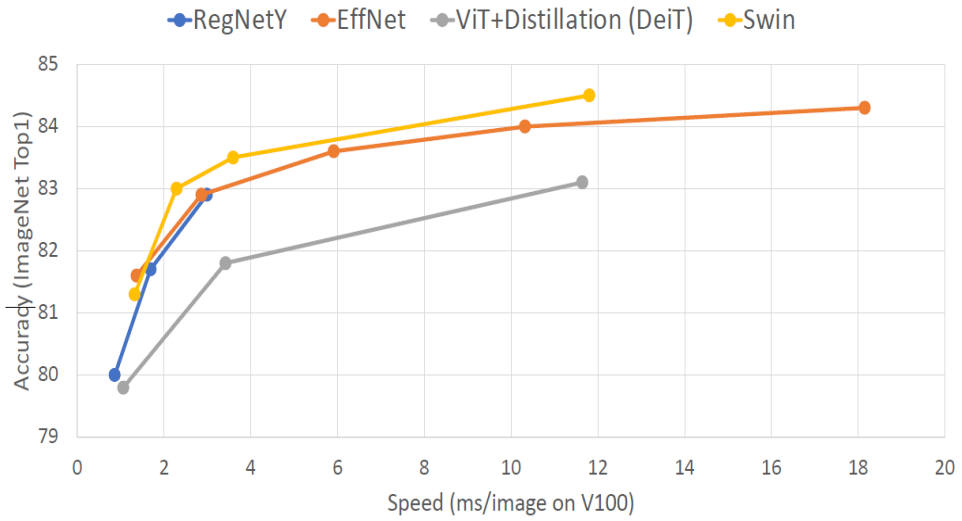


The shifted windows bridge the windows of the preceding layer, providing connections among them that significantly enhance modeling power.



Strong Performance Across Vision Tasks

- **Image Classification**: Achieves top-tier accuracy on ImageNet (e.g., 87.3% top-1 with Swin-L).
- **Object Detection**: Improves box AP and mask AP on COCO, surpassing prior SOTA by a significant margin (e.g., +2.7 AP).
- **Semantic Segmentation**: Attains new best mIoU on ADE20K, showing broad applicability beyond classification.

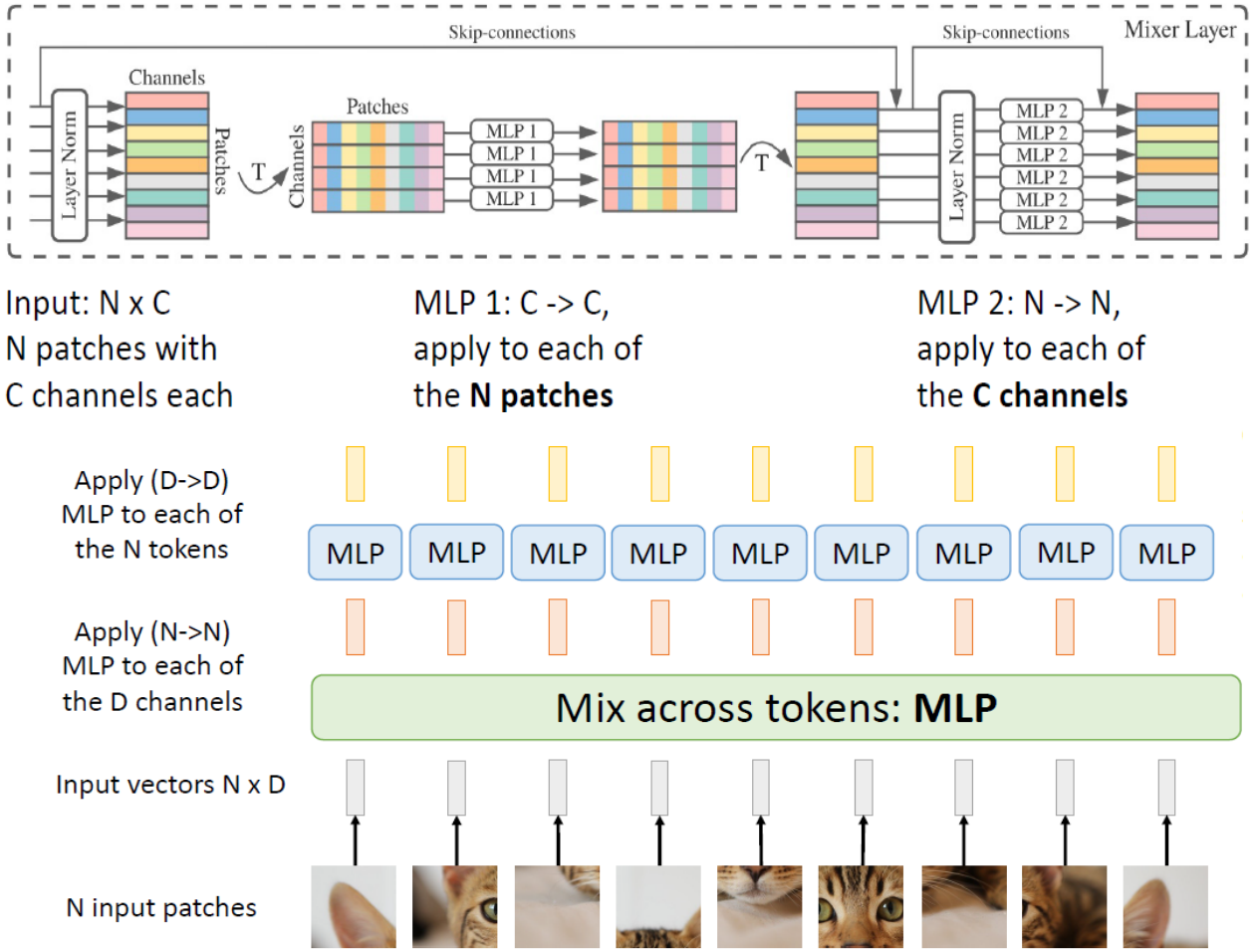


MLP-Mixer: An All-MLP Architecture

Input image is divided into non-overlapping patches – similar to ViT.

- Each patch is linearly projected, forming a patches(x)channels table, X.
- MLP-Mixer separates mixing of features into two kinds of MLP layers:
- Token-Mixing MLP: Operates across patches (rows of X^T).
- Channel-Mixing MLP: Operates across feature channels (rows of X).
- Each layer is a simple MLP with skip connections and LayerNorm – repeated L times.

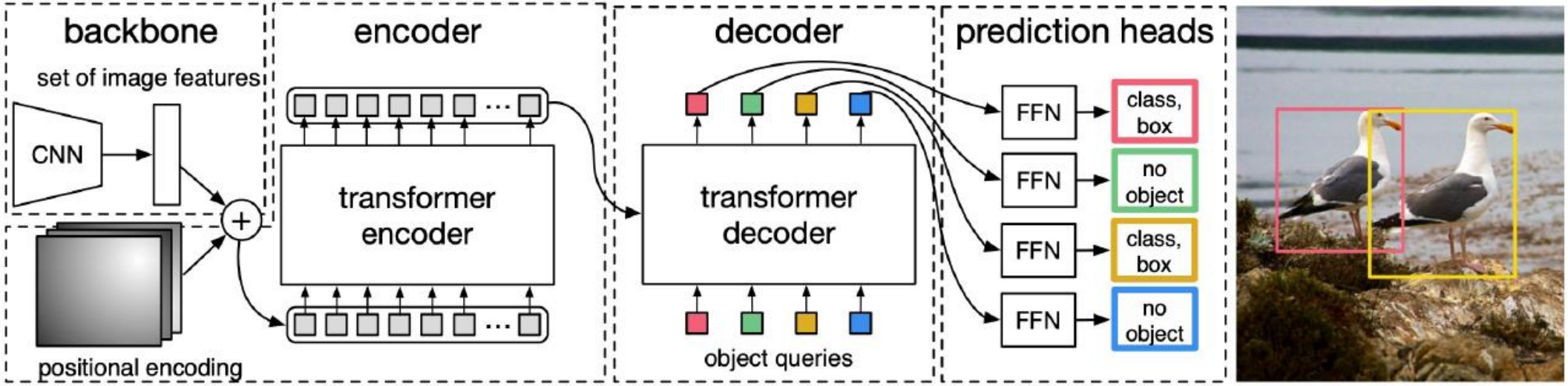
However, its initial result on ImageNet is not very compelling, which gets better when using JFT for pretraining. There are many follow-ups to this work.



Tolstikhin et al, "MLP-Mixer: An all-MLP architecture for vision", NeurIPS 2021

DETR: Object Detection with Transformers

- DEtection TRansformer (DETR) is a novel object detection framework that simplifies the traditional pipeline by directly predicting a set of bounding boxes using a Transformer model.
- DETR uses a bipartite matching approach to match predicted boxes to ground-truth boxes. Specifically, it employs the Hungarian algorithm to find the optimal one-to-one matching between predictions and ground truths, ensuring each prediction is assigned to a unique ground-truth box.
- The model is then trained to regress the box coordinates and classify objects based on this matching. This end-to-end approach eliminates the need for hand-designed components like anchors and non-maximum suppression (NMS), making DETR both simpler and more efficient while achieving competitive performance on object detection tasks.

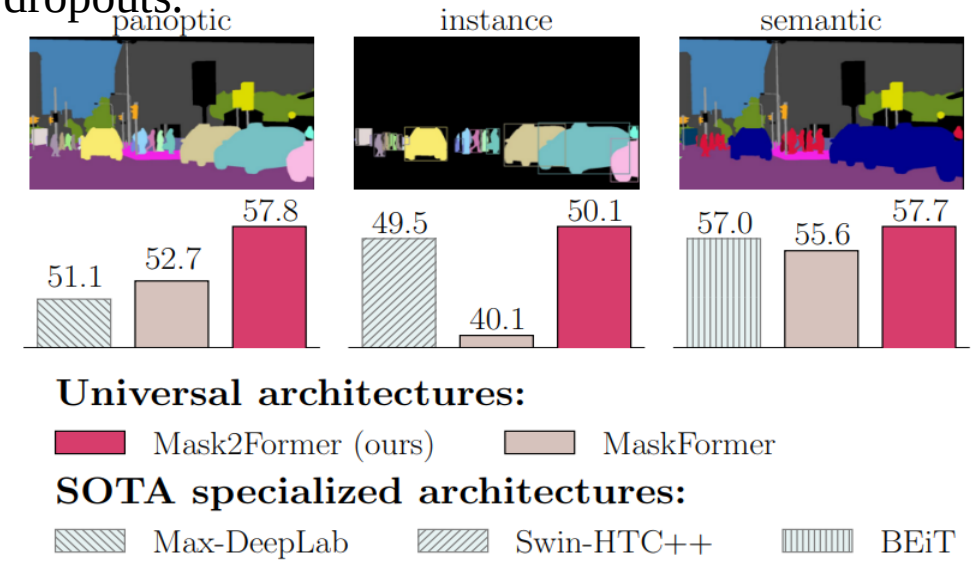


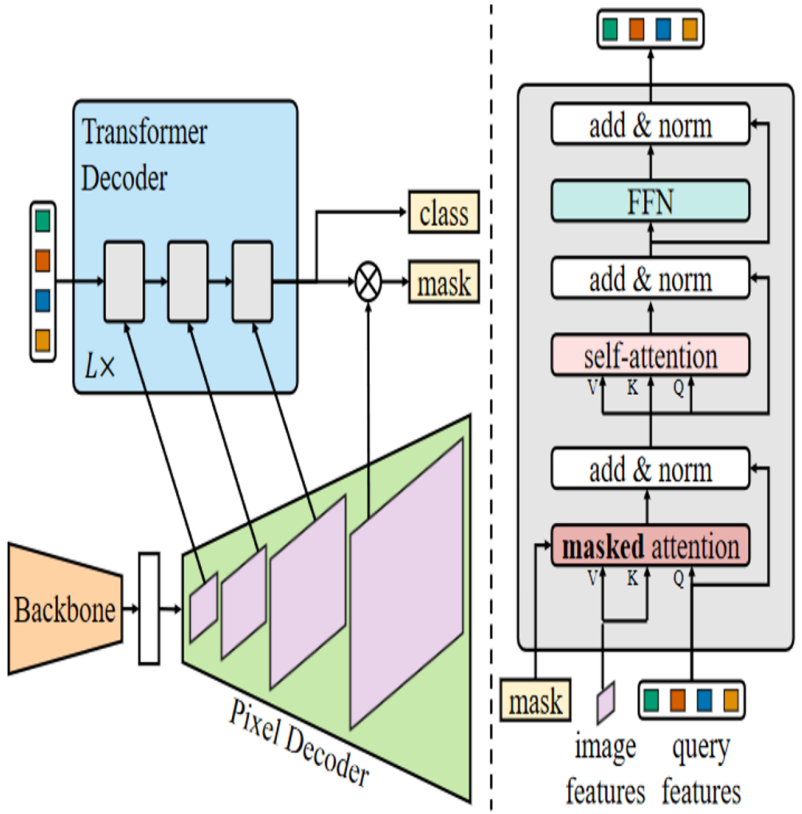
Carion et al, "End-to-End Object Detection with Transformers", ECCV 2020

Masked-attention Mask Transformer for Universal Image Segmentation

A universal image segmentation that outperforms specialized architectures, while still easy to train on every task.

- Masked attention in decoder to restrict attention to localized features centered around predicted segments, which can be either objects or regions depending on specific semantic.
- Use multi-scale high-resolution features to help model to segment small objects/regions.
- Optimization improvements such as switching the order of self and cross attention, making query features learnable and removing dropouts.





Cheng, Bowen, et al. "Masked-attention mask transformer for universal image segmentation." *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022.

Key Takeaways of ViTs

- **ViTs are an evolution, not a revolution. We can still fundamentally solve the same problems as with CNNs.**
- Inductive biases of CNNs might not be as harmful as previously claimed (even in big-data regimes), and they might even benefit Transformers (e.g., Swin Transformer).
- The flexibility of Transformers is helpful when considering multimodal data.
- Don't give in to the hype but instead critically evaluate each paper based on the empirical evidence.
- Pay attention to hidden implementation details (e.g., optimization, training schedule, data augmentation, etc.).
- Learn to appreciate simple yet effective ideas.
- Consider the big picture of each paper (e.g., potential future impact of the paper).
- **Attention is NOT all you need (but it can still be useful).**
- Having said this, currently, the model choice still largely depends on the task that we want to solve.

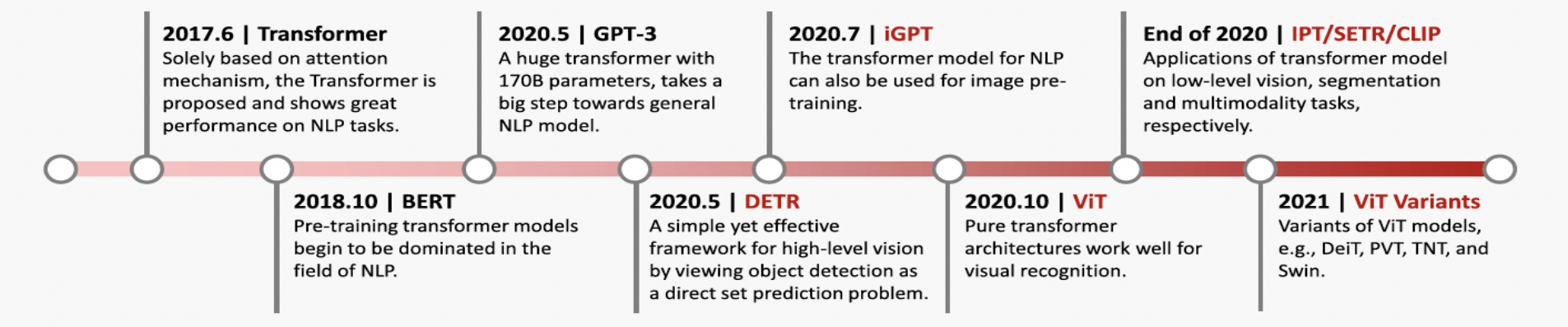
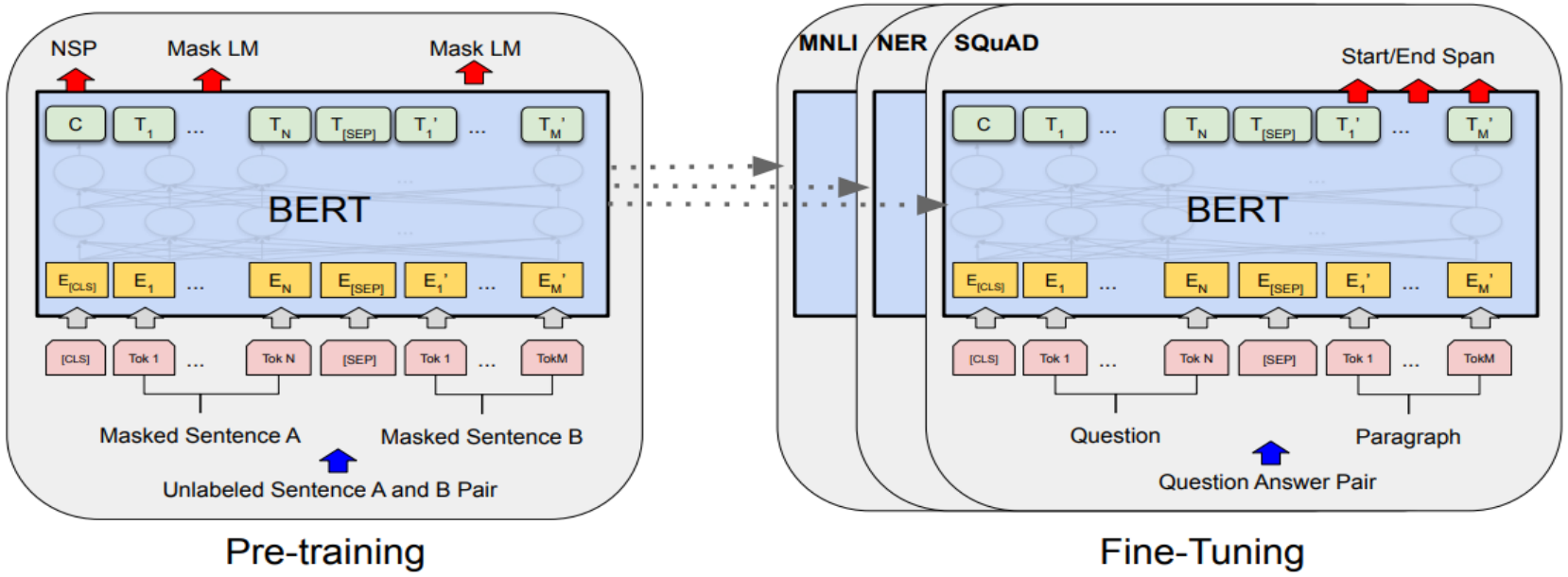


Fig. 1. Key milestones in the development of transformer. The vision transformer models are marked in red.

Han, Kai, et al. "A survey on vision transformer." *IEEE transactions on pattern analysis and machine intelligence* 45.1 (2022): 87-110.

BERT: Encoder-Only Model

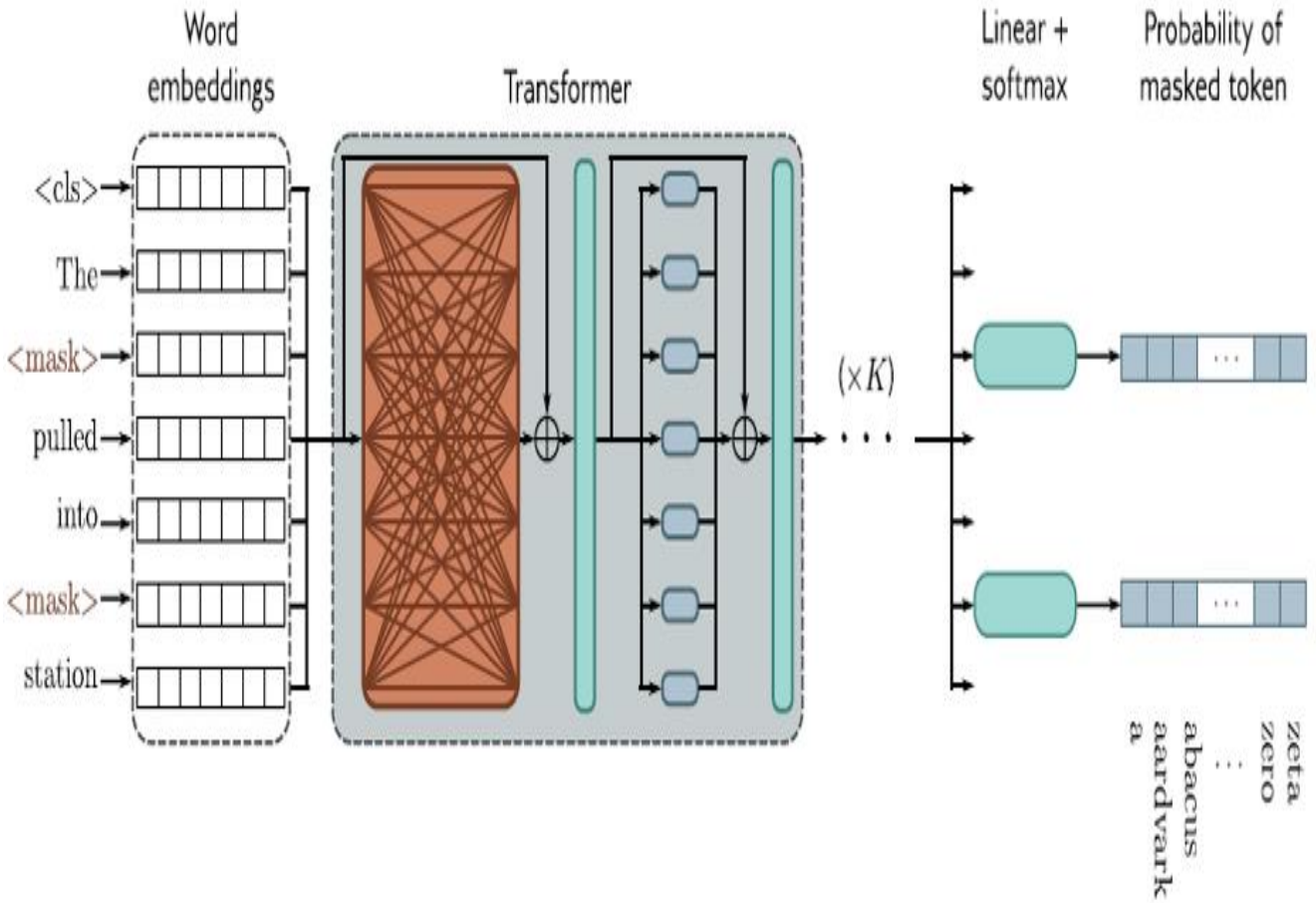
- Bidirectional Encoder Representations from Transformers (BERT) is an encoder-only Transformer design that uses a vocabulary of 30,000 tokens. Input tokens are converted to 1024-dimensional word embeddings and passed through 24 transformer layers, each containing a self-attention mechanism with 16 heads. The queries, keys, and values for each head are of dimension.
- BERT exploits transfer learning. During pretraining, parameters are learned using self-supervision from a large corpus of text. The goal here is for the model to learn general information about the statistics of language. In the fine-tuning stage, the resulting network is adapted to solve a particular task using a smaller body of supervised training data.



Devlin, Jacob. "Bert: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805* (2018).

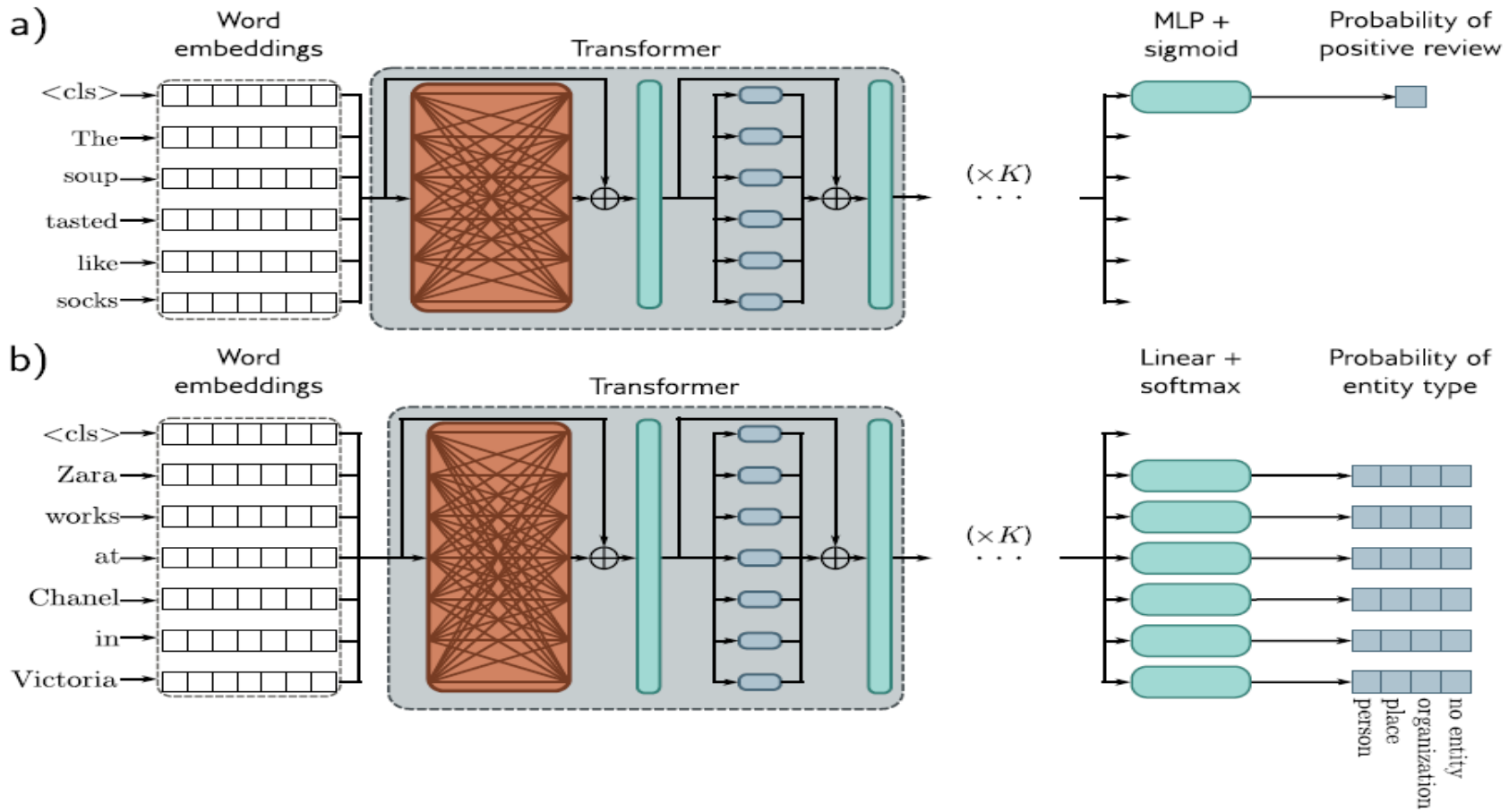
BERT: Pretraining

- In the pre-training stage, the network is trained using self-supervision. The self-supervision task consists of predicting missing words from sentences from a large internet corpus.
- Such prediction forces the Transformer network to understand some syntax. For example, it may learn that the adjective “red” is often found before nouns like “house” or “car” but never before a verb like “scout”.
- It can also learn superficial common sense about the world. For example, after training, the model will assign a higher probability to the missing word “train” before “station” than it would to the world “peanut”.
- However, the degree of “understanding” this type of model can ever have is limited.



BERT: Finetuning

- In the fine-tuning stage, the model parameters are adjusted to specialize the network to a particular task. An extra layer is appended onto the transformer network to convert the output vectors to the desired output format.
- Examples include text classification, word classification and text span prediction.



GPT3: Decoder-Only Model

GPT-3 (Generative Pre-trained Transformer 3):

- Released by OpenAI in 2020, with up to 175B parameters.
- Primarily a decoder-only Transformer for language modeling.

Context-Only Generation:

- Takes in prompt text and generates the next tokens auto-regressively.
- Admits no separate encoder & no bidirectional attention over input tokens.

Key Strengths:

- Zero-shot and few-shot learning abilities.
- Extremely large scale leads to surprising emergent capabilities.

Training Data:

- 499B tokens from diverse sources (Common Crawl, WebText2, Books).

Scaling Laws:

- Larger model + more data + more compute = improved results (Kaplan et al. 2020).
- GPT-3 extends to near trillion parameter regime feasibility.

GPT3: Decoder-Only Model

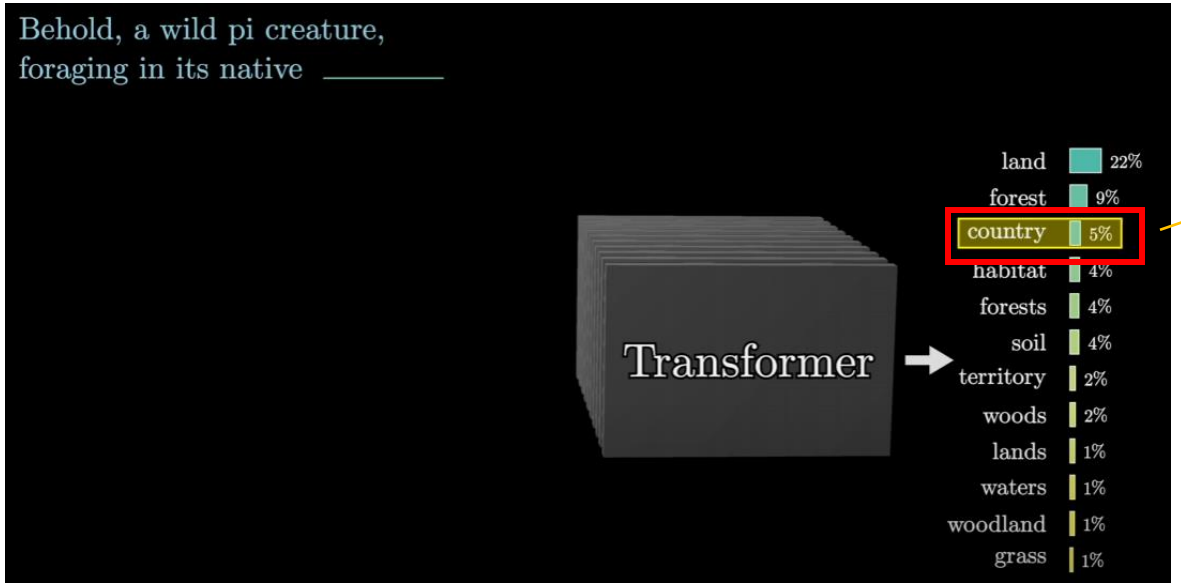
Standard Transformer Blocks:

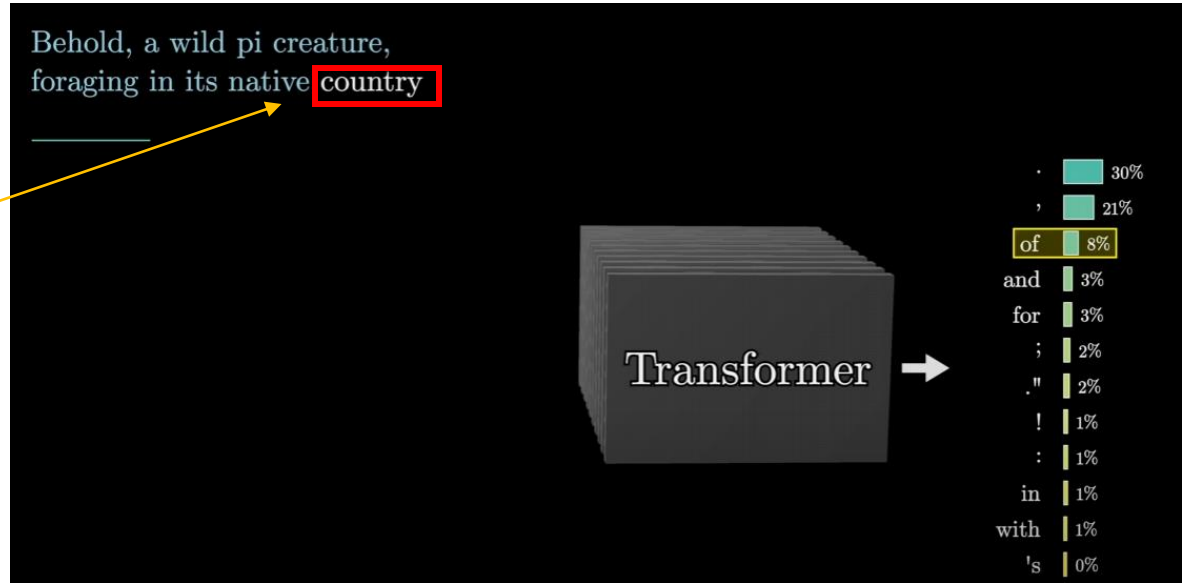
- Self-attention layers (causal/masked).
- Feed-forward MLP sub-layer, layer normalization, residual connections.

No Encoder Module:

- Input text is fed directly to the decoder stack (positional embeddings + token embeddings).
- Output hidden states map to next-token probabilities.

Specifically, GPT3 models the language by constructing an **autoregressive** language model.





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

Flamingo: a Visual Language Model

- Developed by DeepMind to enable language models to interpret and generate text grounded in images.
- Extends the idea of large language models (LLMs) into the visual domain.

Core Architecture:

❖ Visual Encoder + Language Model:

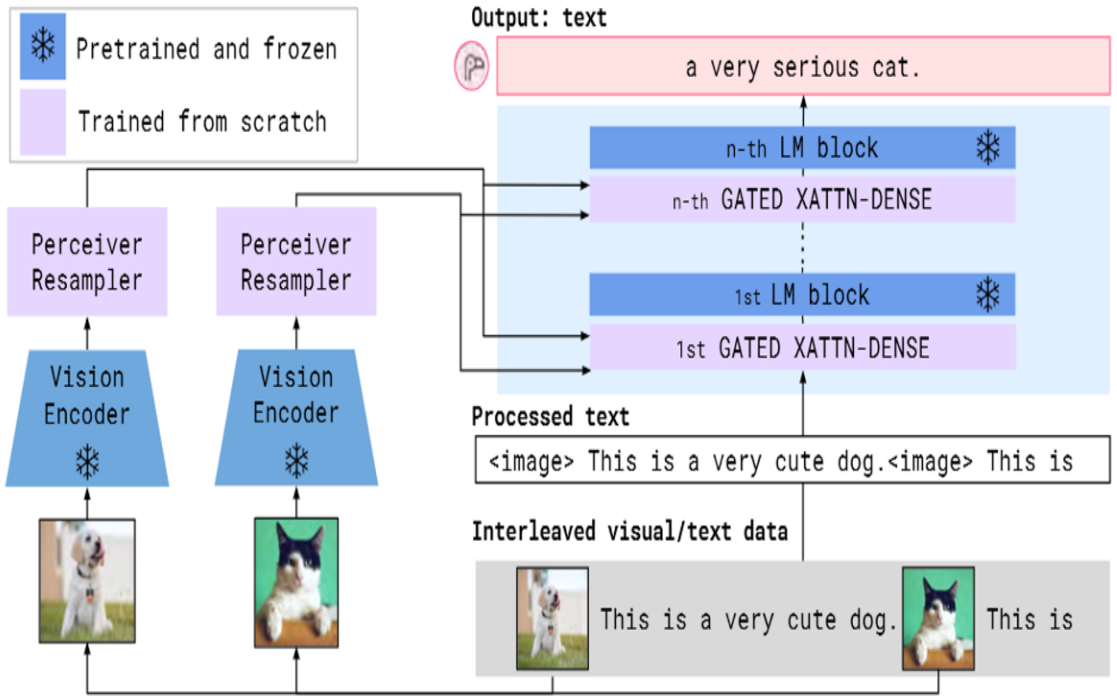
- Flamingo uses a CNN or Vision Transformer (ViT) to embed images.
- Connects to a LLM backbone (e.g., GPT-style) via cross-attention layers.

❖ Perceiver Resampler (DeepMind approach):

- Adapts the visual features into a compact set of tokens fed into the language model.
- Minimizes overhead when dealing with high-res images.

❖ Decoder-Only LM:

- Flamingo extends the LM with cross-attention blocks to handle image-conditioned text generation.



Alayrac, Jean-Baptiste, et al. "Flamingo: a visual language model for few-shot learning." *Advances in neural information processing systems* 35 (2022): 23716-23736.

ViperGPT

ViperGPT: A novel system for performing reasoning on visual data.

LLM-Generated Python Code:

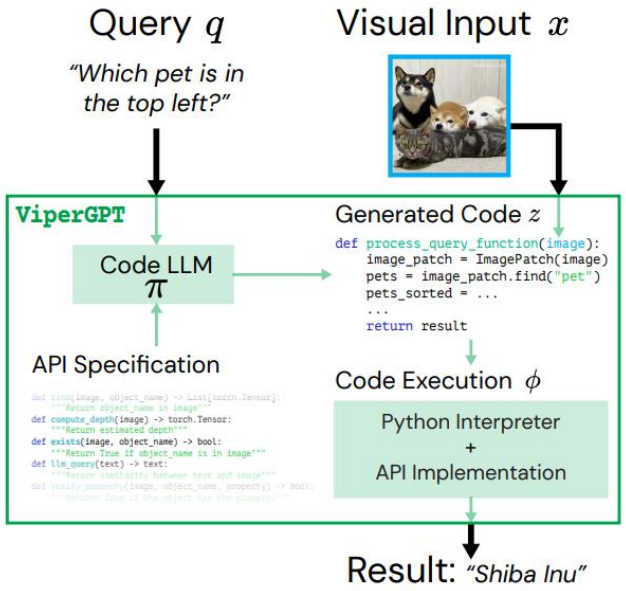
- ViperGPT uses a LLM (like GPT) to generate small Python snippets.
- These snippets call specialized vision functions (e.g., detection, segmentation, classification) to gather info.

Execution + Re-evaluation:

- The code runs in a sandbox, returning results to the LLM.
- The model integrates these results to refine or correct its approach, forming a loop of reasoning.

Architecture Overview:

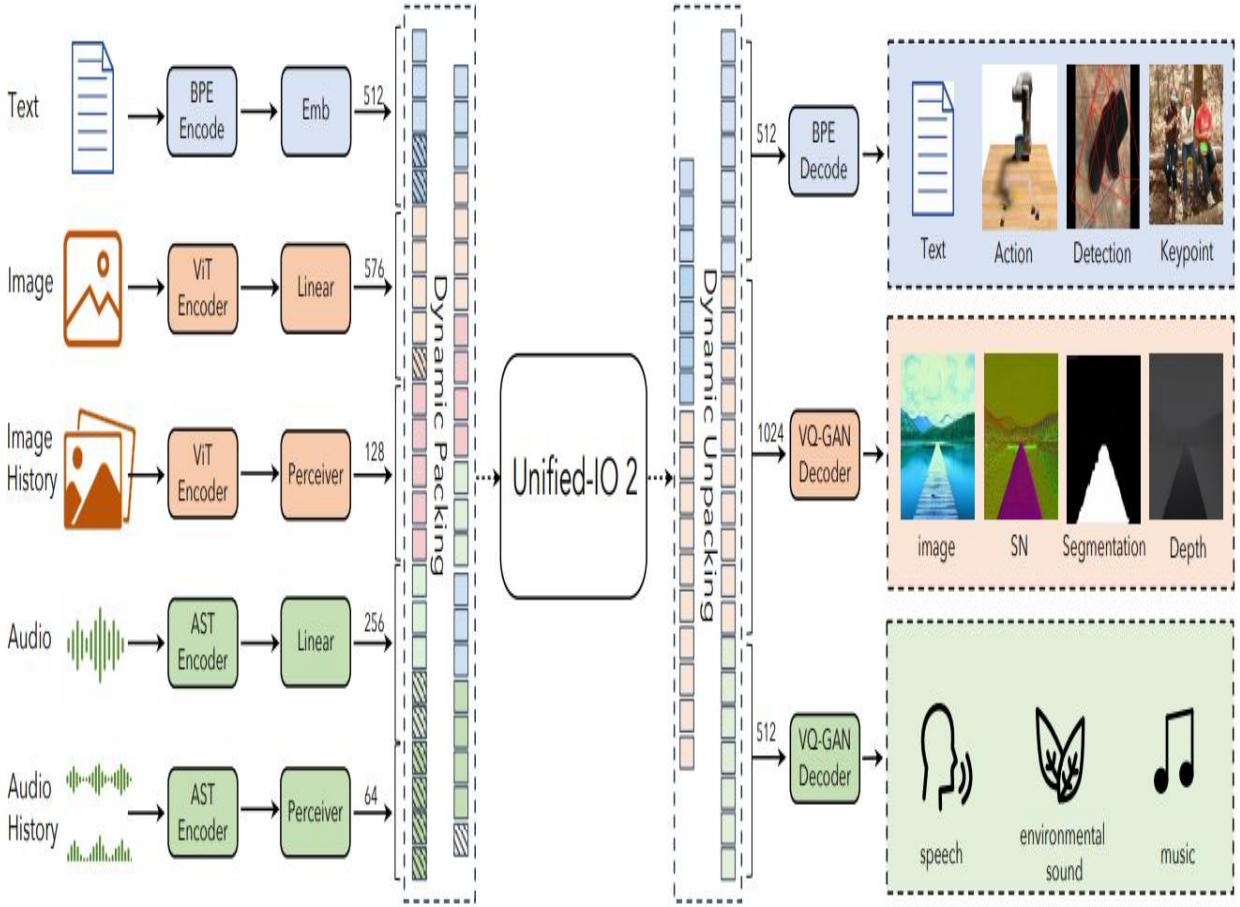
- **Language Model Brain:** Proposes code to interpret or transform image data.
- **Vision Backend:** Toolset of image-processing APIs for detection, OCR, bounding boxes, etc.
- **Execution Environment:** Python interpreter running the LLM-generated code.
- **Feedback Loop:** Model reads code outputs, decides next step (rewrite code, answer, etc.).



Surís, Dídac, Sachit Menon, and Carl Vondrick. "Vipergpt: Visual inference via python execution for reasoning." *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023.

Unified-IO 2

- The first autoregressive multi-modal model that is capable of understanding and generating image, text, audio and action.
- Processes all modalities with a single unified encoder-decoder Transformer, which is made possible after encoding various inputs and outputs into sequences of tokens.
  - Texts and actions are tokenized using byte-pair encoding from LLaMA by Meta.
  - Images are encoded using pre-trained ViT.
  - Audios are encoded up to 4.08 seconds of audio into a spectrogram, which is then encoded with a pre-trained Audio Spectrogram Transformer (AST).



Lu, Jiasen, et al. "Unified-IO 2: Scaling Autoregressive Multimodal Models with Vision Language Audio and Action." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024.

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

Scaling up Transformers

Scaling up Transformers

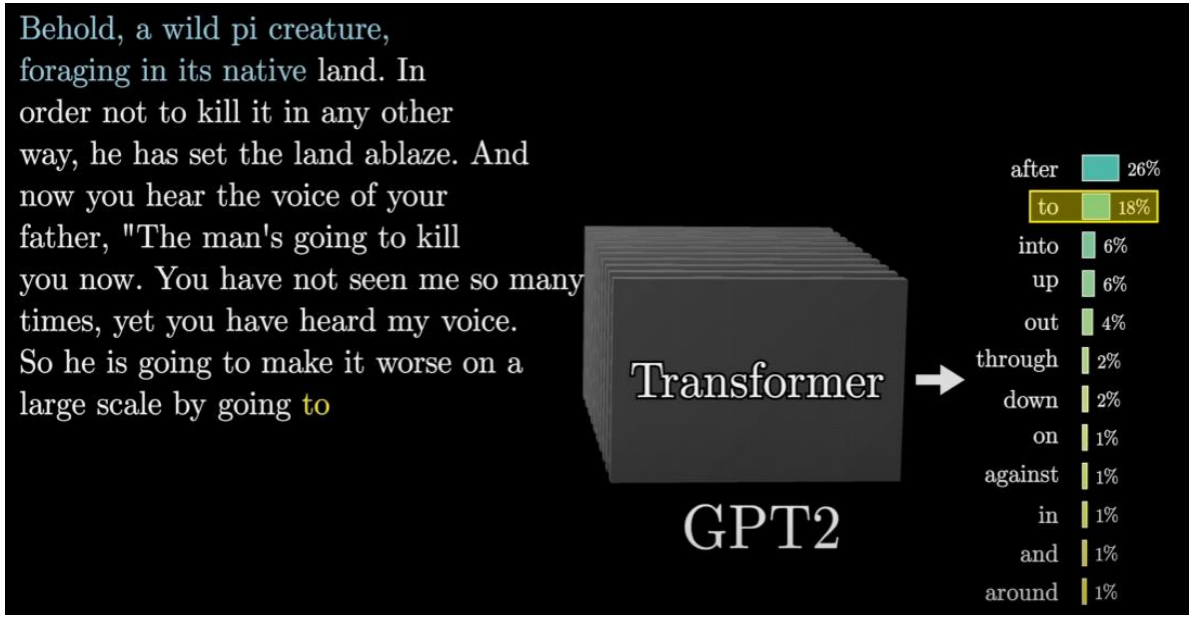
$3,768,320 on Google Cloud (eval price)

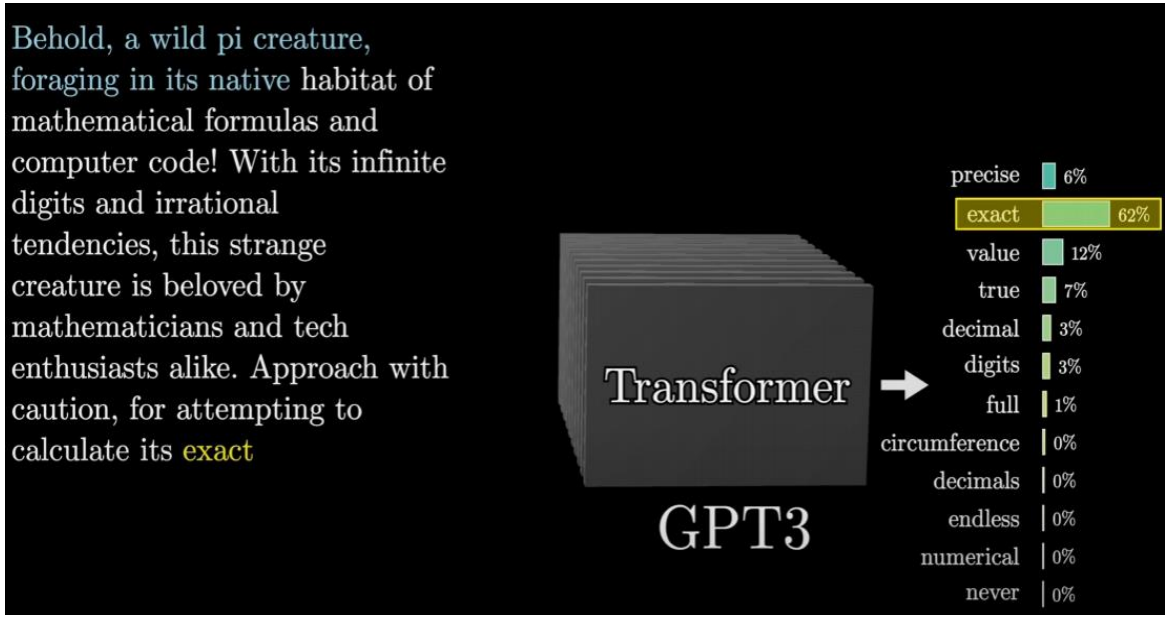
Model	Layers	Width	Heads	Params	Data	Training
Transformer-Base	12	512	8	65M		8x P100 (12 hours)
Transformer-Large	12	1024	16	213M		8x P100 (3.5 days)
BERT-Base	12	768	12	110M	13 GB	
BERT-Large	24	1024	16	340M	13 GB	
XLNet-Large	24	1024	16	~340M	126 GB	512x TPU-v3 (2.5 days)
RoBERTa	24	1024	16	355M	160 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

GPT4 vs. Gemini

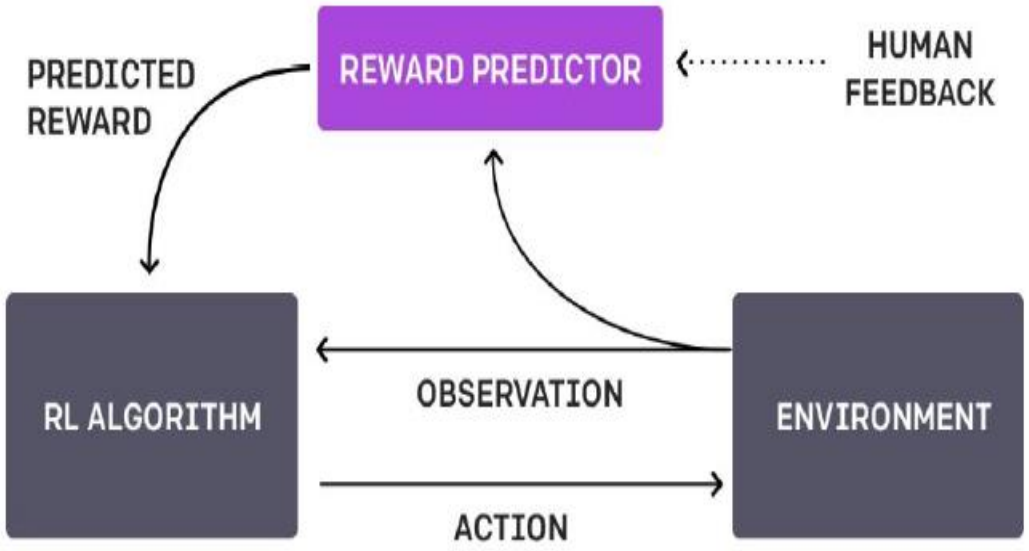
- Supervised learning on large dataset, then RLHF.
- GPT-4 trained on both **images** and text
- Discuss humor in images, summarize screenshot text, etc.
- GPT-4 is "more reliable, creative, and able to handle much more nuanced instructions than GPT-3.5”
- Much longer context windows of 8,192 and 32,768 tokens
- Does exceptionally well on standardized tests
- No technical details of GPT-4 released

- Based on a Mixture-of-Experts (MoE) model
- Goal: have several models/“experts” work together to
- solve a problem, each expert may be specialized for a task/purpose
- Combination of multiple small Neural networks known as “Experts” which are trained and capable of handling particular data and performing specialized tasks.
- “Gating network” which predicts which response is best suited to address the request.

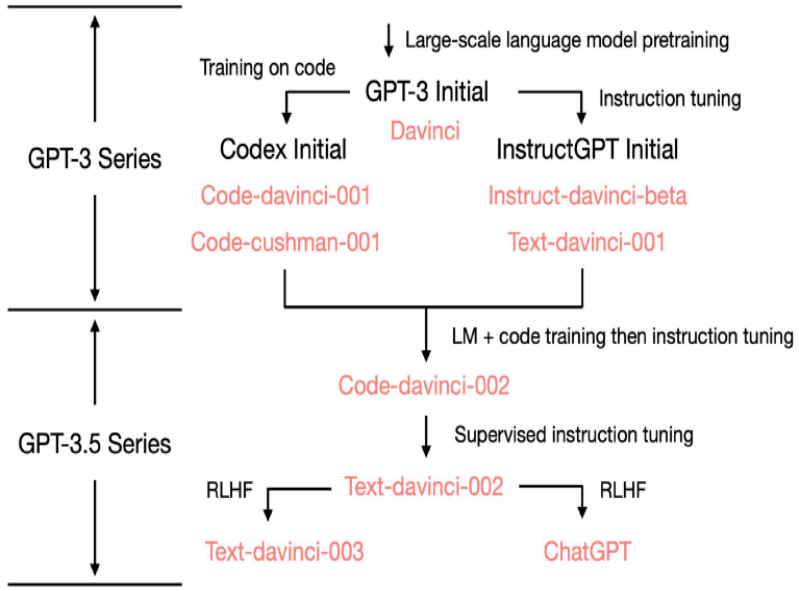
Gemini 1.5

Reinforcement Learning with Human Feedback (RLHF) and ChatGPT

- RLHF: Technique that trains a "reward model" directly from human feedback.
- Uses the model as a reward function to optimize an agent's policy using reinforcement learning (RL) through an optimization algorithm.
- Ask humans to rank instances of the agent's behavior, e.g. which produced response is better.



- ChatGPT is finetuned on GPT-3.5, which is a series of models trained on a mix of text and code using instruction tuning and RLHF
- Taken the world by storm!



(Feng, Garg, Bunnapradist, & Lee, 2024)

Chain of Thought (CoT) Reasoning

- Chain-of-thought (CoT) - series of intermediate reasoning steps
- Shown to improve LLM performance on complex reasoning tasks
- Inspired by human thought process: decompose multi-step problems
- Also provides an interpretable window into behavior of the model (how it arrived at an answer, where it goes wrong in its reasoning path)
- CoT exploits the fact that deep down in the model's weights, it knows more about the problem than just prompting it to get a response

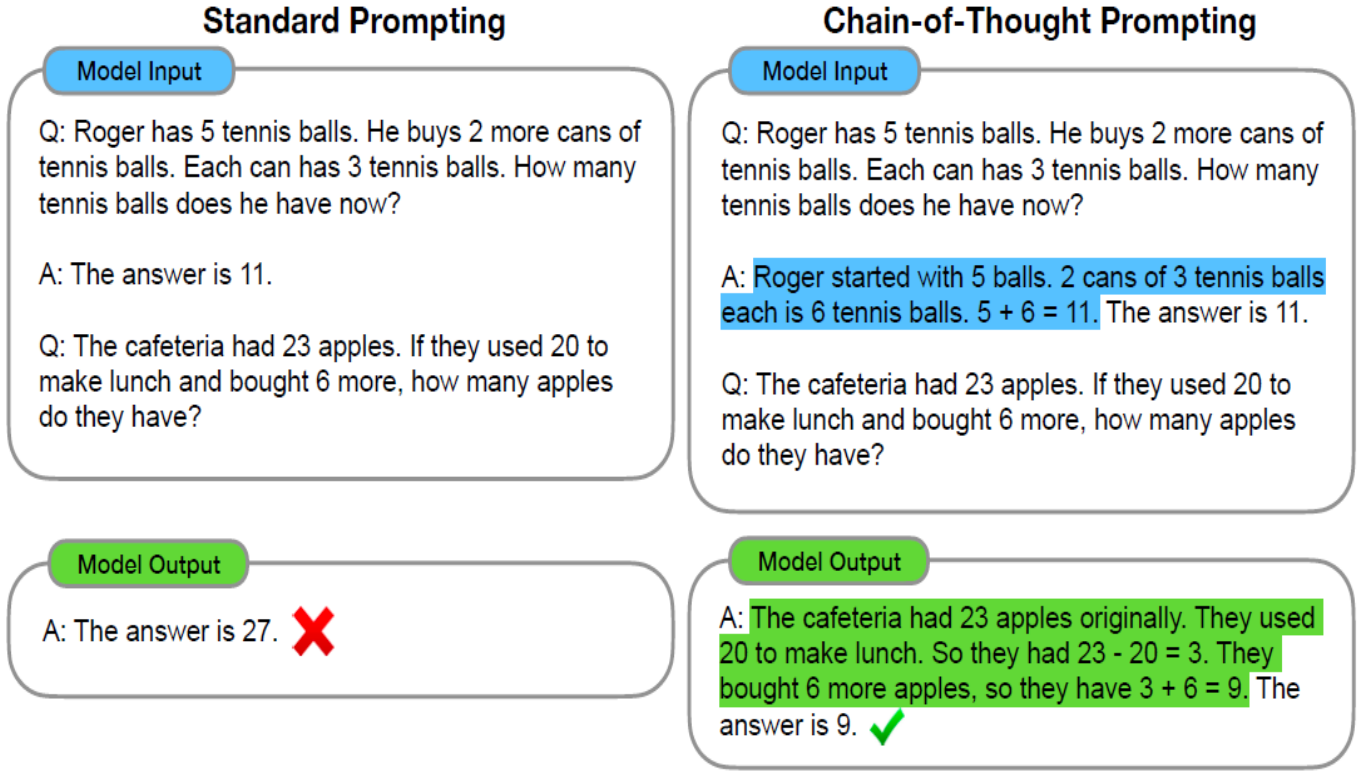


Figure 1: Chain-of-thought prompting enables large language models to tackle complex arithmetic, commonsense, and symbolic reasoning tasks. Chain-of-thought reasoning processes are highlighted.

Wei, Jason, et al. "Chain-of-thought prompting elicits reasoning in large language models." *Advances in neural information processing systems* 35 (2022): 24824-24837.

Content

Theoretical Properties

Transformers have demonstrated remarkable empirical success but face unresolved **theoretical challenges** relating to:

- ❖ Their **exact expressive power** and limitations.
- ❖ **Computational complexity** and how to handle extremely long sequences.
- ❖ **Optimization** behaviors (loss landscapes, generalization, implicit bias).
- ❖ **Interpretability** and **attention** mechanics not always equaling causal explanation.
- ❖ Integrating multiple **modalities** robustly, with quantifiable performance bounds.

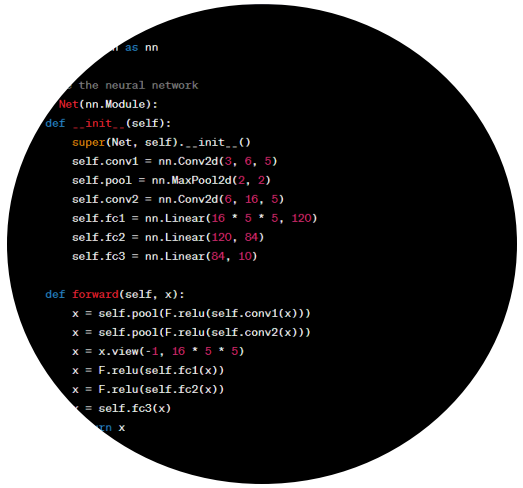
Building a rigorous mathematical foundation for these areas is an active field of research, aimed at closing the gap between the extraordinary practical performance of Transformers and our incomplete theoretical understanding of why and when they excel.

- ▶ Representation Limits: How do Transformers compare to universal approximators (e.g., RNNs, CNNs)?
- ▶ Positional Encoding: Is sinusoidal or learned encoding fully capturing sequence structure?
- ▶ Loss Landscape: Highly high-dimensional, and stability for deep Transformer stacks is not fully understood.
- ▶ Generalization Bounds: Empirical scaling laws show bigger data + bigger models = better results, but no rigorous proofs.
- ▶ Implicit Biases: Transformers, like other networks, exhibit hidden inductive biases from gradient descent – yet these remain partially unexplained.
- ▶ Masked LM / Next-Token Prediction: Why do these tasks alone suffice to learn so many language/vision capabilities?
- ▶ Empirical success vs. theoretical backing – little clarity on*why* it transfers so well to so many downstream tasks.
- ▶ Scaling Laws: Observed empirically, but not proven in general.

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.
- Szałata, 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