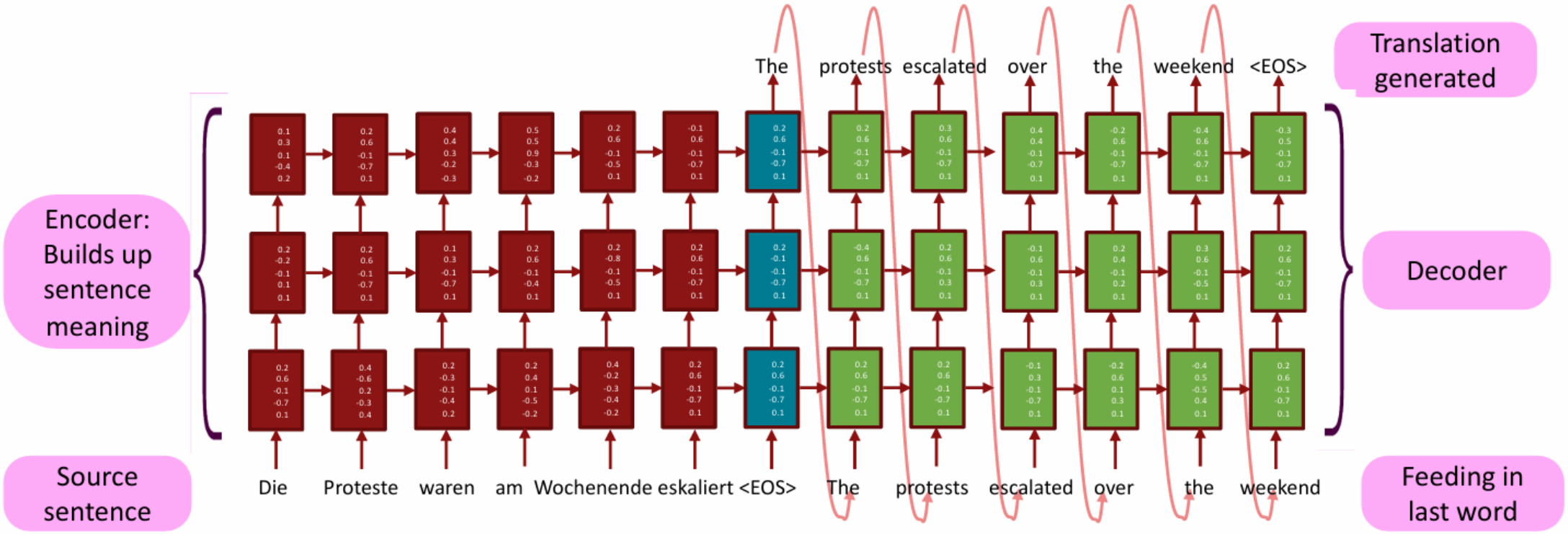
Attention, Self Attention, Transformers

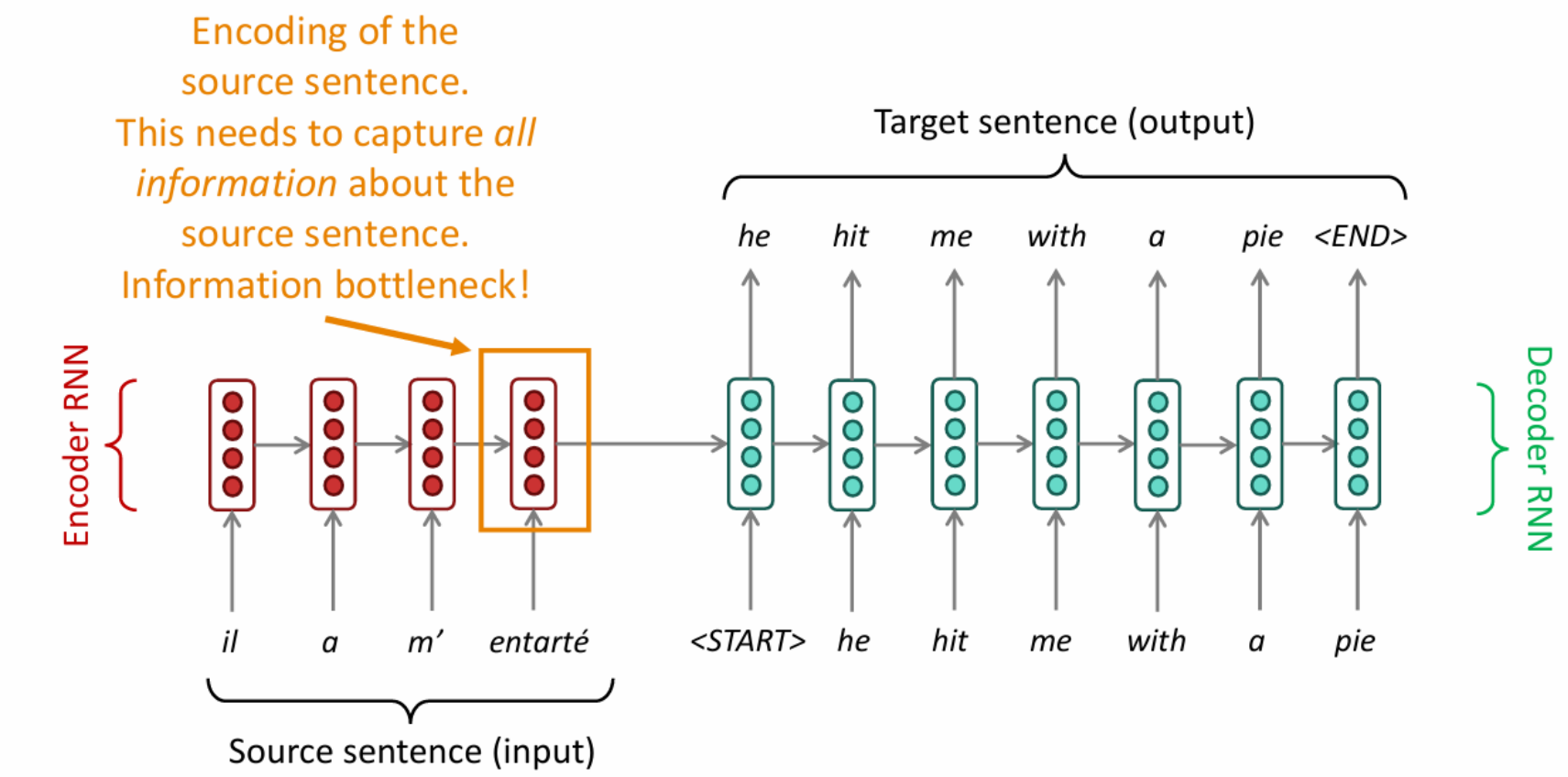
HESAM HOSSEINI

SUMMER 2024

Review: encoder-decoder machine translation net



Why attention? Sequence-to-sequence: the bottleneck problem

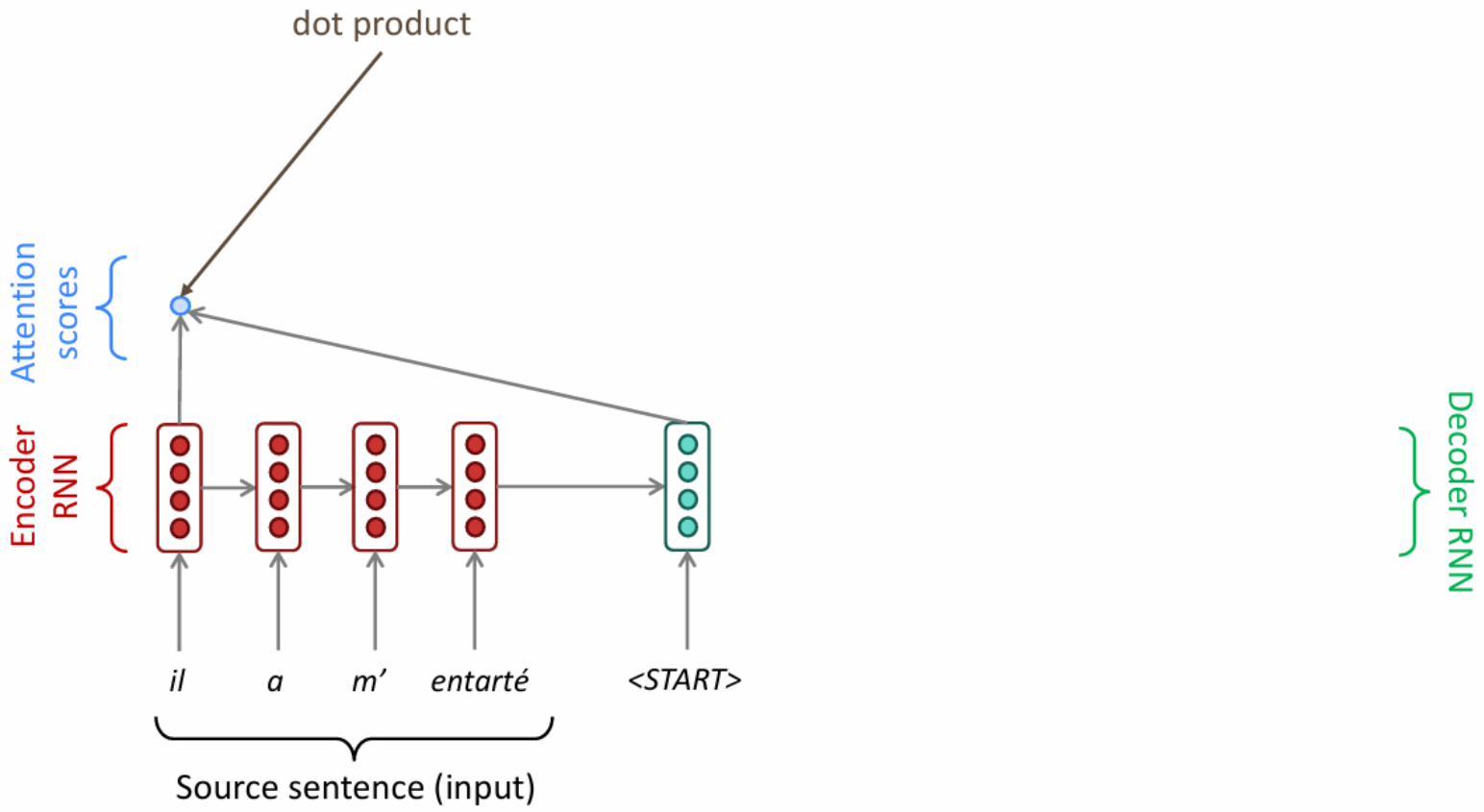


Attention

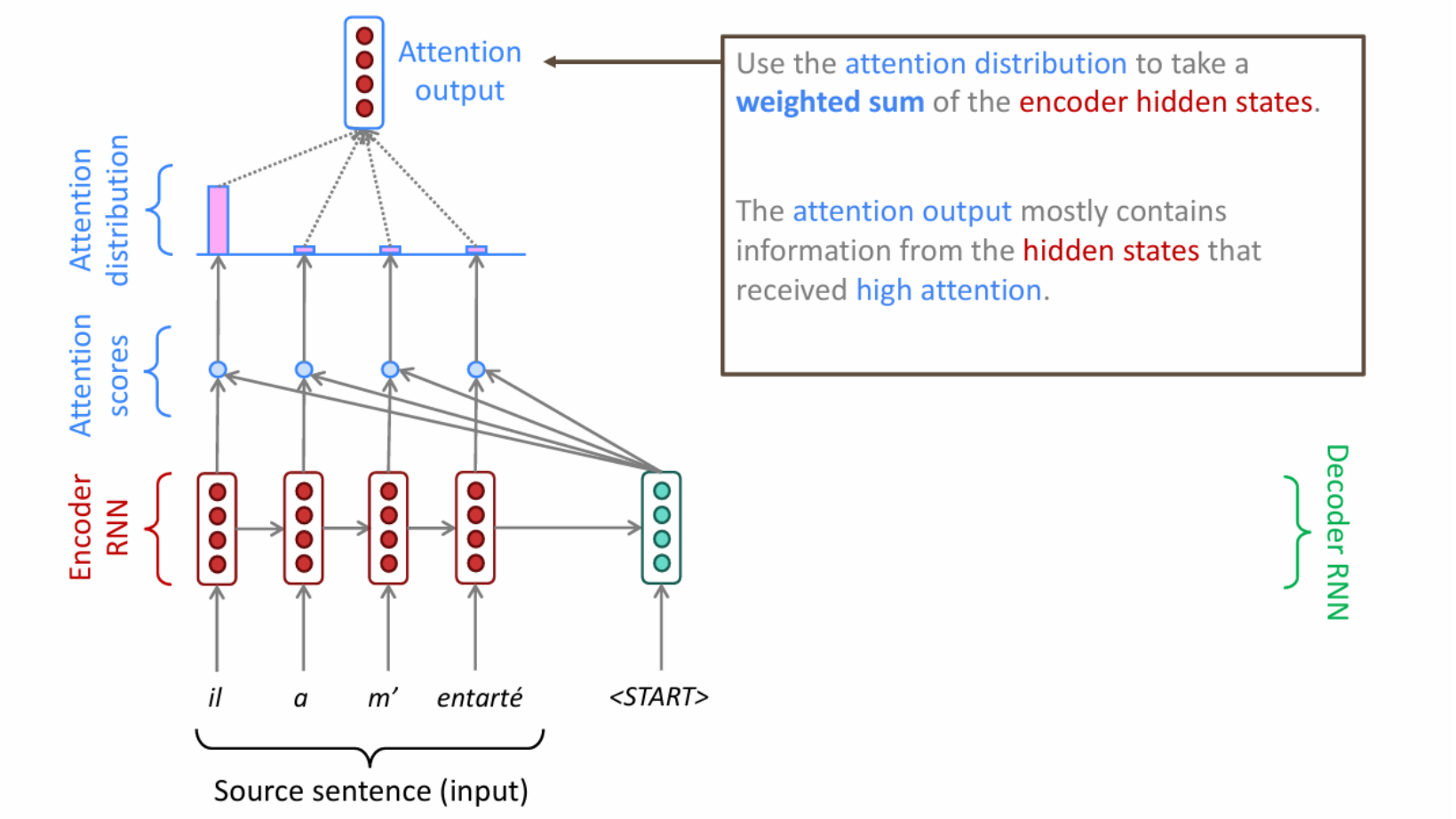
Attention provides a solution to the bottleneck problem.

Core idea: on each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence

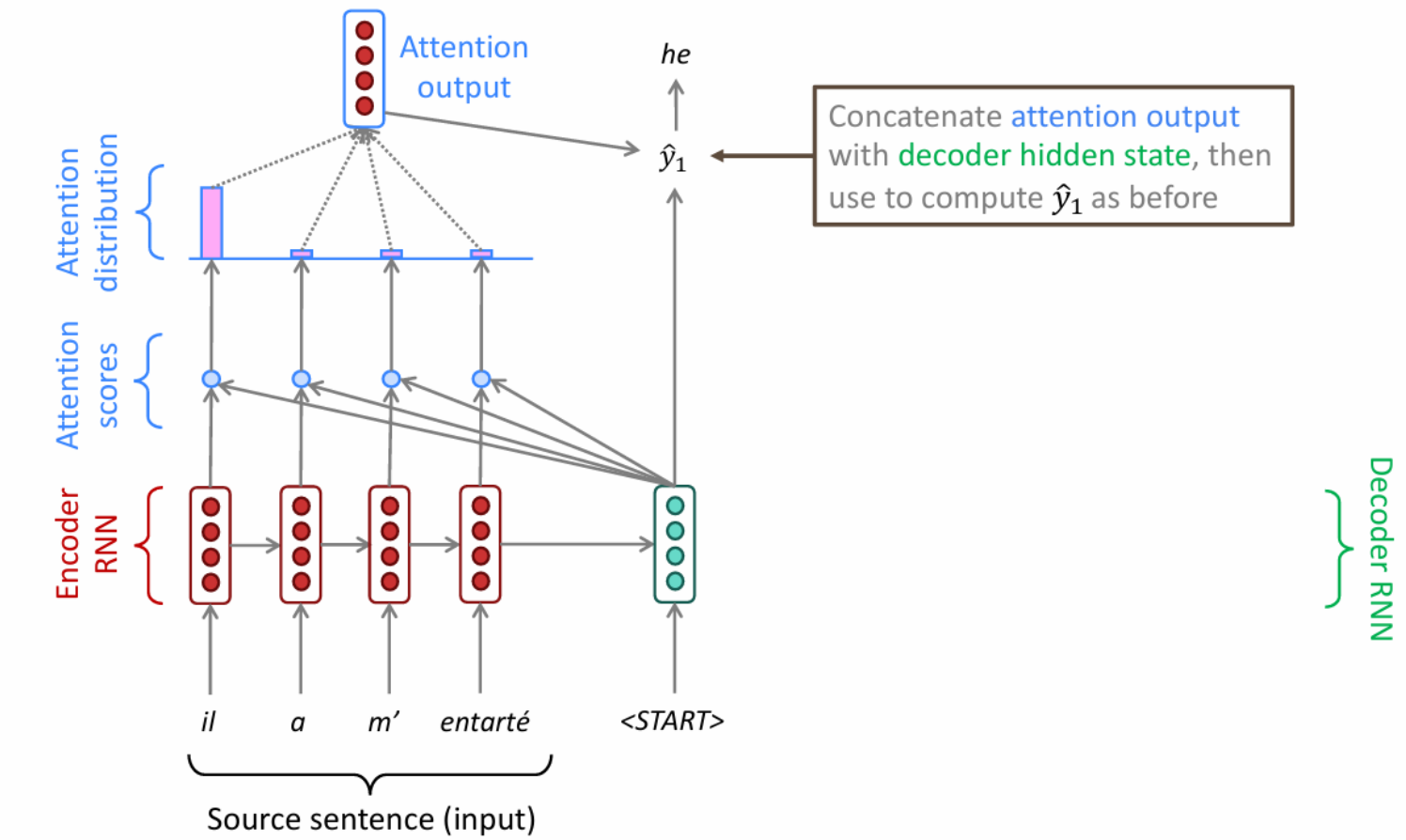
Sequence-to-sequence with attention



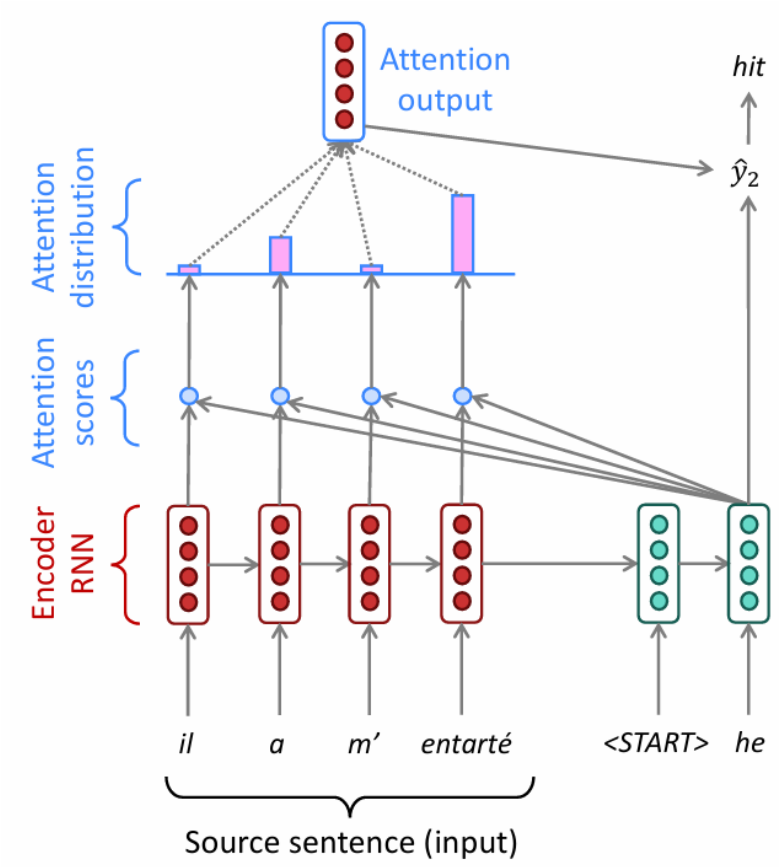
Sequence-to-sequence with attention

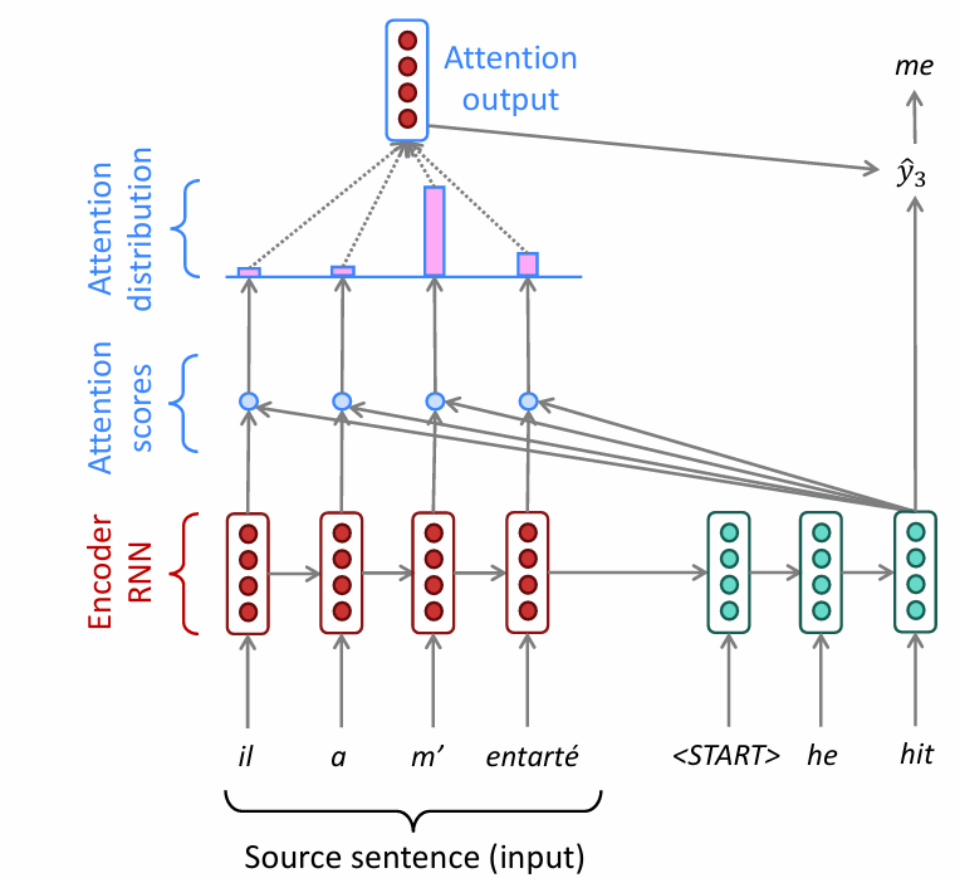


Sequence-to-sequence with attention

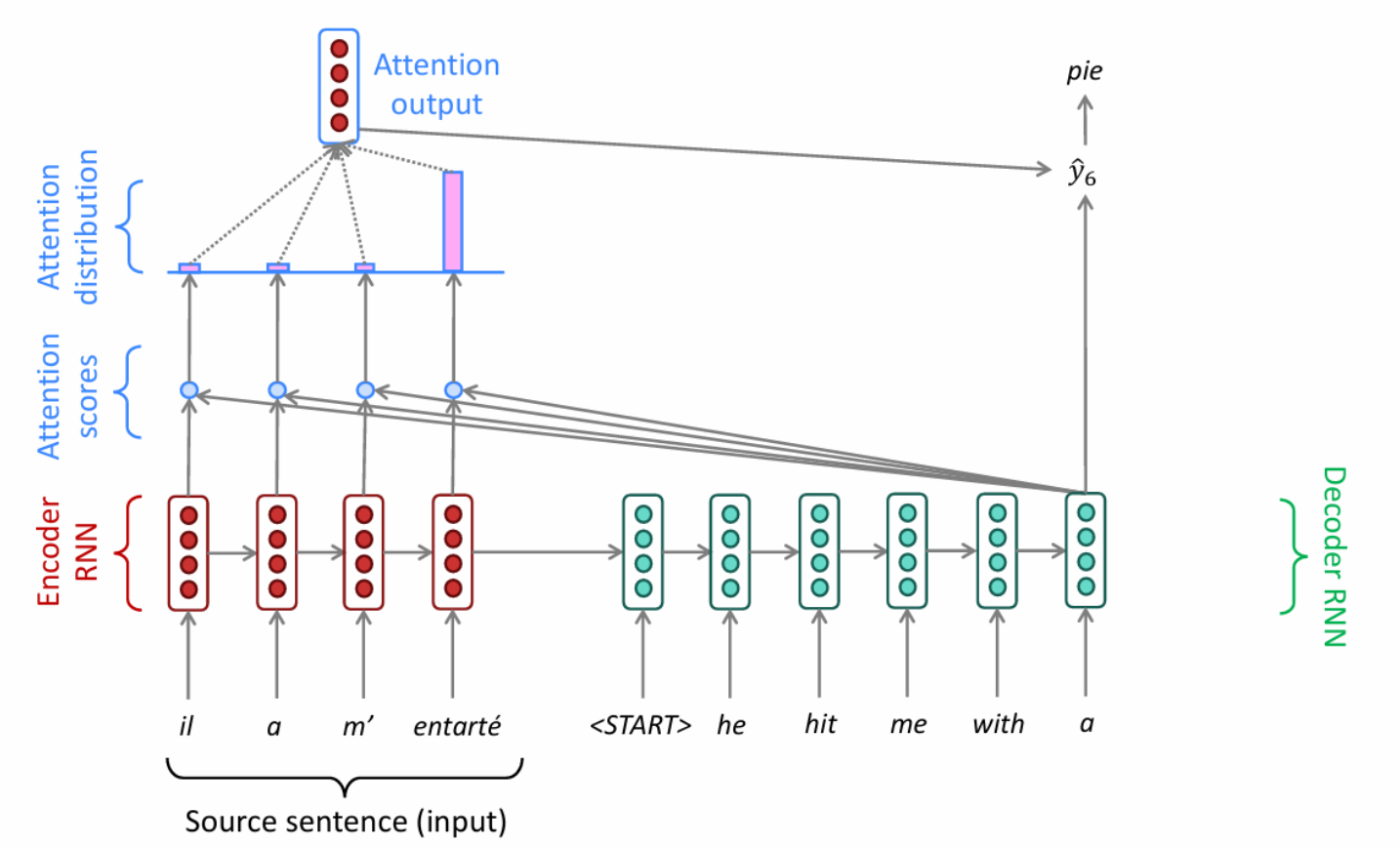


Sequence-to-sequence with attention





Sequence-to-sequence with attention



Attention: in equations

We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$

On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$

We get the attention scores $\boldsymbol{e}^t$ for this step:

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \text{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

Finally we concatenate the attention output $\boldsymbol{a}_t$ with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

Attention is great!

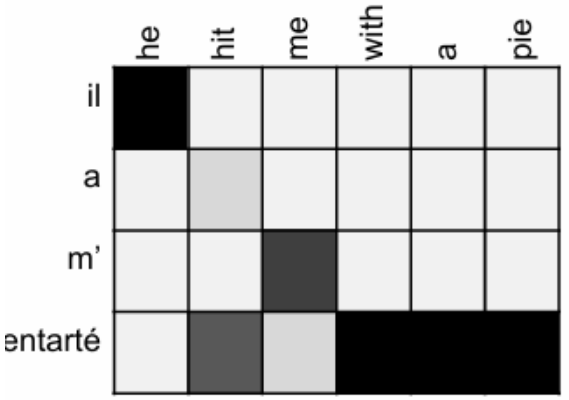
Attention significantly improves NMT performance
- It's very useful to allow decoder to focus on certain parts of the source

Attention provides a more "human-like" model of the MT process
- You can look back at the source sentence while translating, rather than needing to remember it all

Attention solves the bottleneck problem
- Attention allows decoder to look directly at source; bypass bottleneck

Attention helps with the vanishing gradient problem
- Provides shortcut to faraway states

Attention provides some interpretability
- By inspecting attention distribution, we see what the decoder was focusing on
- We get (soft) alignment for free!
- This is cool because we never explicitly trained an alignment system
- The network just learned alignment by itself



Attention is a general Deep Learning technique

We've seen that attention is a great way to improve the sequence-to-sequence model for Machine Translation.

However : You can use attention in many architectures (not just seq2seq) and many tasks (not just MT)

More general definition of attention :

- Given a set of vector values, and a vector query, attention a weighted sum of the values, dependent on the query.

We sometimes say that the query attends to the values.

For example, in the seq2seq + attention model, each decoder hidden state (query) attends to all the encoder hidden states (values).

Transformers: Is Attention All We Need?

Attention Is All You Need

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

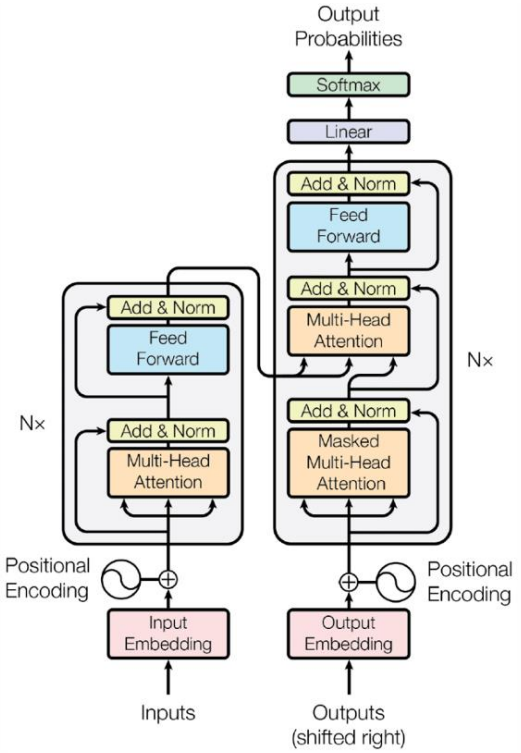
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Transformers Have Revolutionized the Field of NLP

By the end of this lecture, you will deeply understand the neural architecture that underpins virtually every state-of-the-art NLP model today!



Why Move Beyond Recurrence?

The Transformers authors had 3 desirata when designing this architecture:

1. Minimize (or at least not increase) computational complexity per layer.
2. Minimize path length between any pair of words to facilitate learning of long-range dependencies.
3. Maximize the amount of computation that can be parallelized

1. Computational Complexity Per Layer

When sequence length (n) << representation dimension (d), complexity per layer is lower for a Transformer compared to the recurrent models we've learned about so far.

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

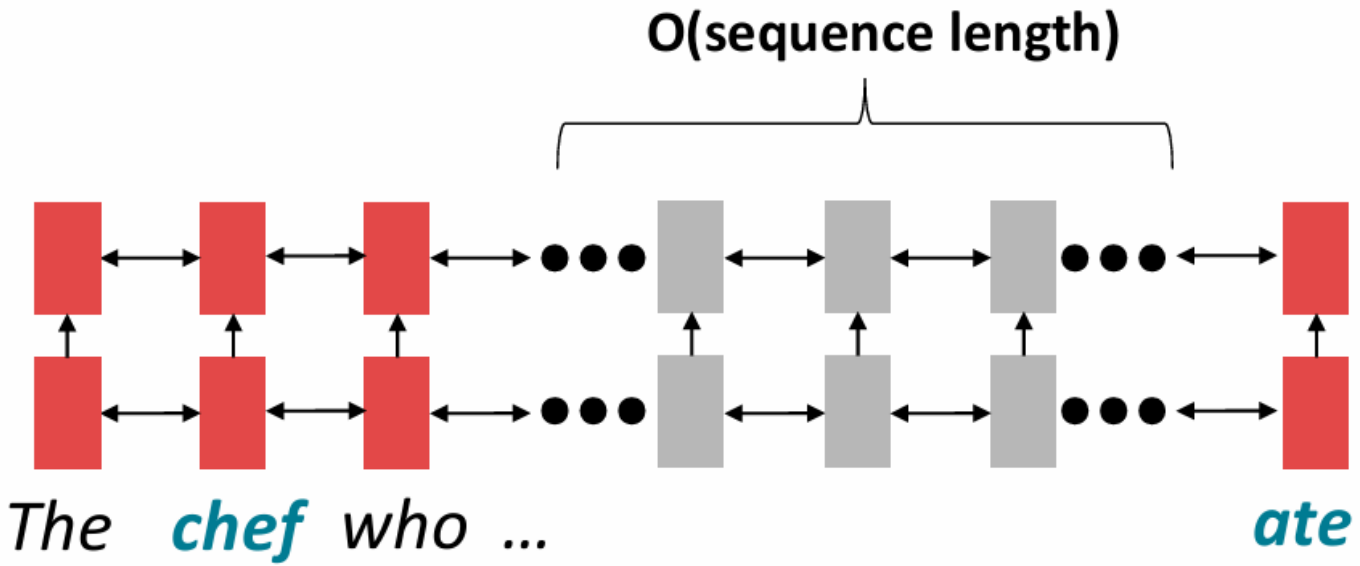
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

2. Minimize Linear Interaction Distance

RNNs are unrolled “left-to-right”.

It encodes linear locality: a useful heuristic! Nearby words often affect each other’s meanings

Problem: RNNs take O(sequence length) steps for distant word pairs to interact.

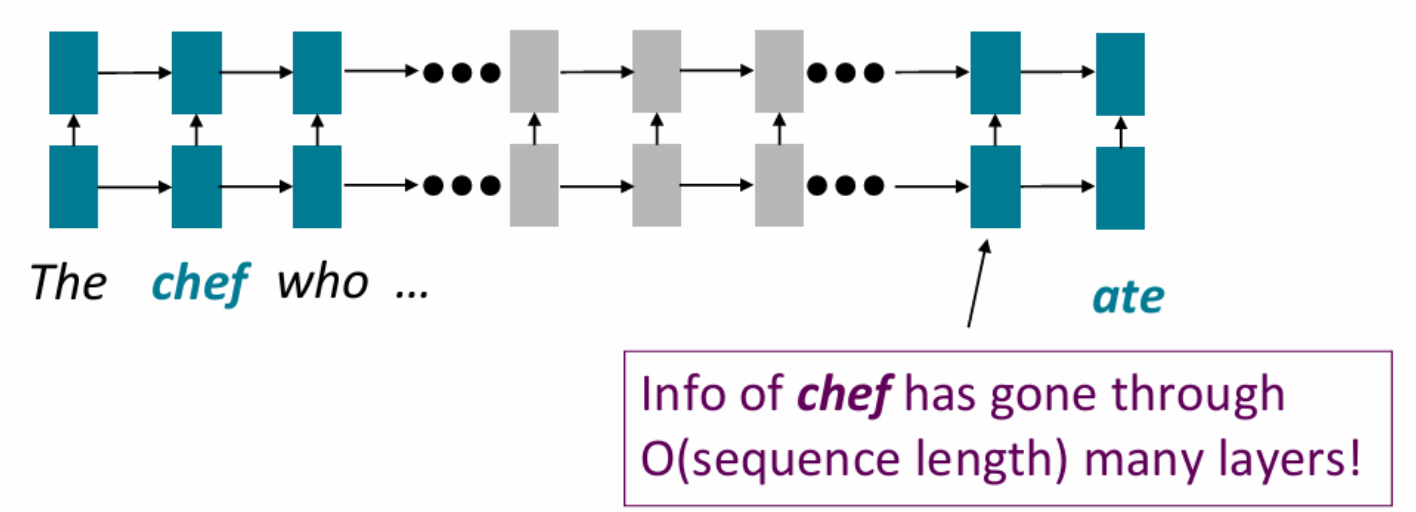


2. Minimize Linear Interaction Distance

O(sequence length) steps for distant word pairs to interact means:

Hard to learn long-distance dependencies (because gradient problems!)

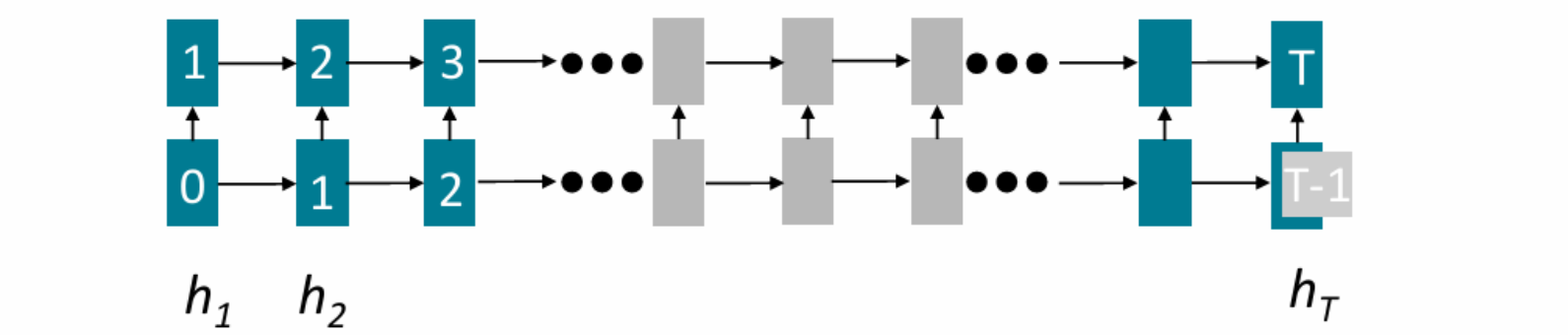
Linear order of words is “baked in”; we already know sequential structure doesn't tell the whole story...



3. Maximize Parallelizability

Forward and backward passes have O(seq length) unparallelizable operations

- GPUs (and TPUs) can perform many independent computations at once!
- But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
- Inhibits training on very large datasets!
- Particularly problematic as sequence length increases, as we can no longer batch many examples together due to memory limitations



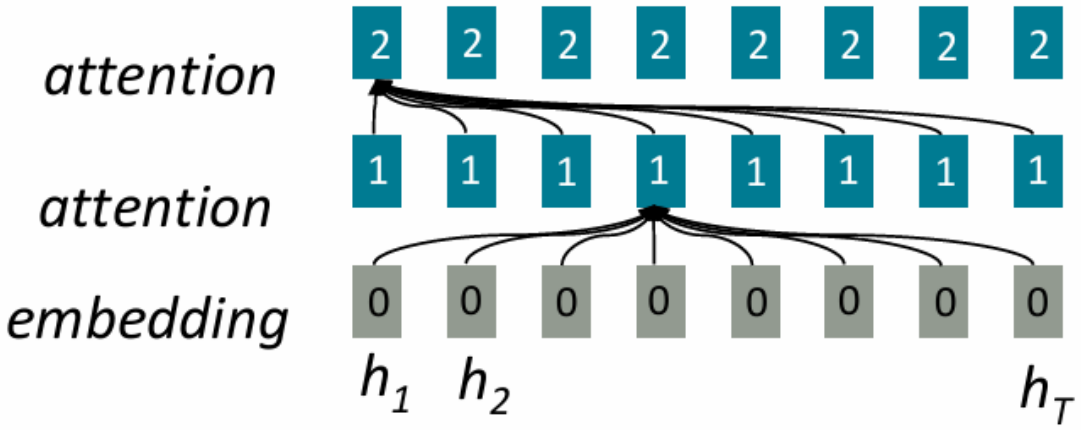
Numbers indicate min # of steps before a state can be computed

Transformer High-Level Architecture

To recap, attention treats each word's representation as a query to access and incorporate information from a set of values.

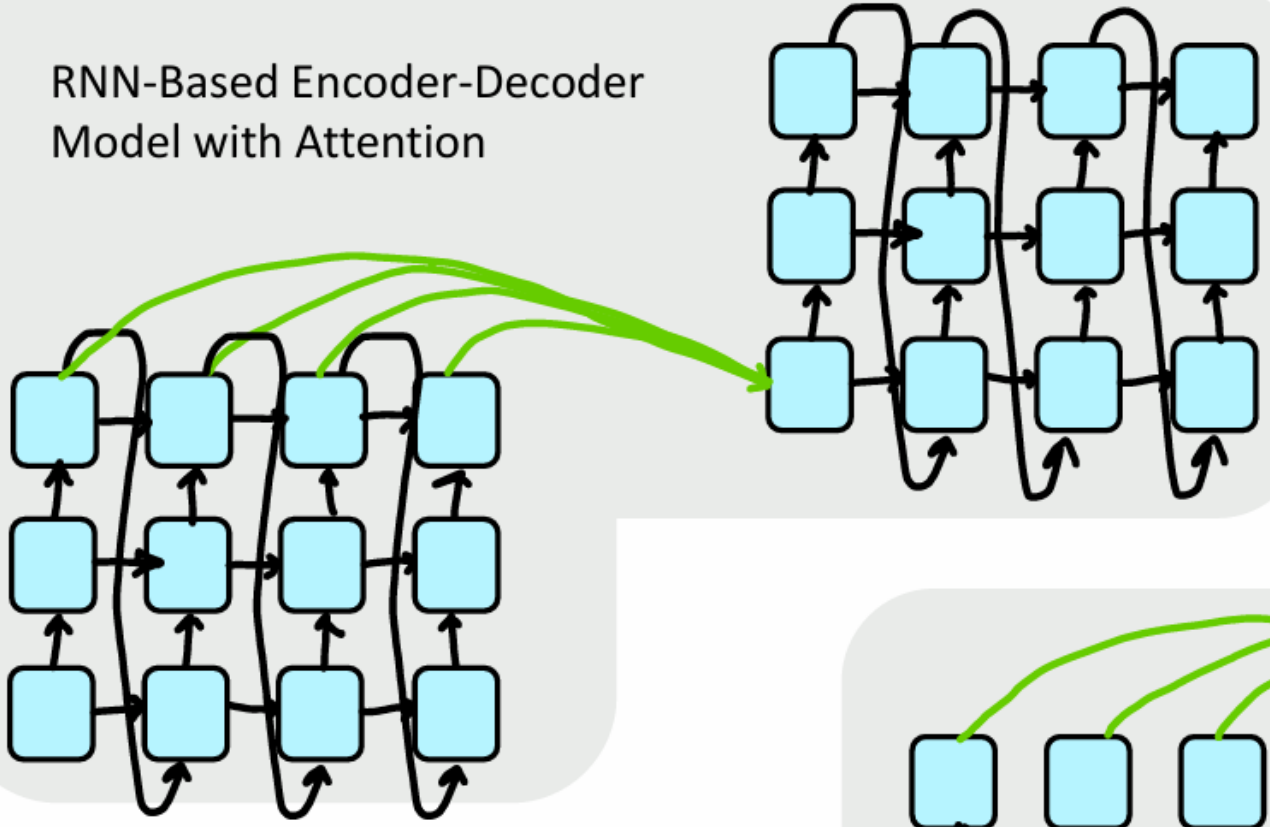
we saw attention from the decoder to the encoder in a recurrent sequence-to-sequence model

Self-attention is encoder-encoder (or decoder-decoder) attention where each word attends to each other word within the input (or output).



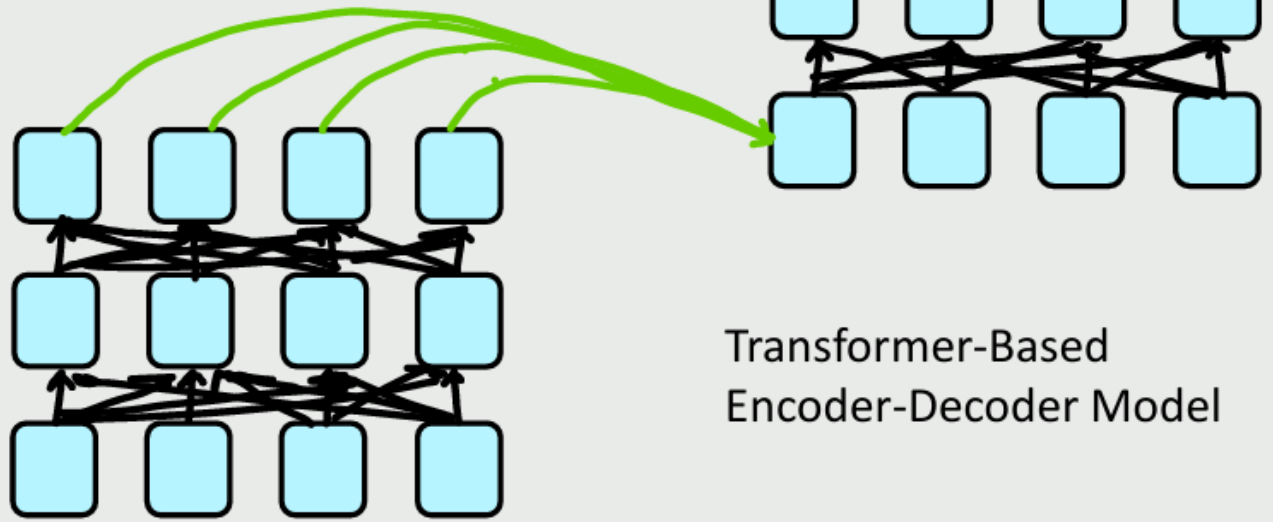
All words attend to all words in previous layer; most arrows here are omitted

RNN-Based Encoder-Decoder Model with Attention



Transformer Advantages:

- Number of unparallelizable operations does not increase with sequence length.
- Each "word" interacts with each other, so maximum interaction distance is O(1).

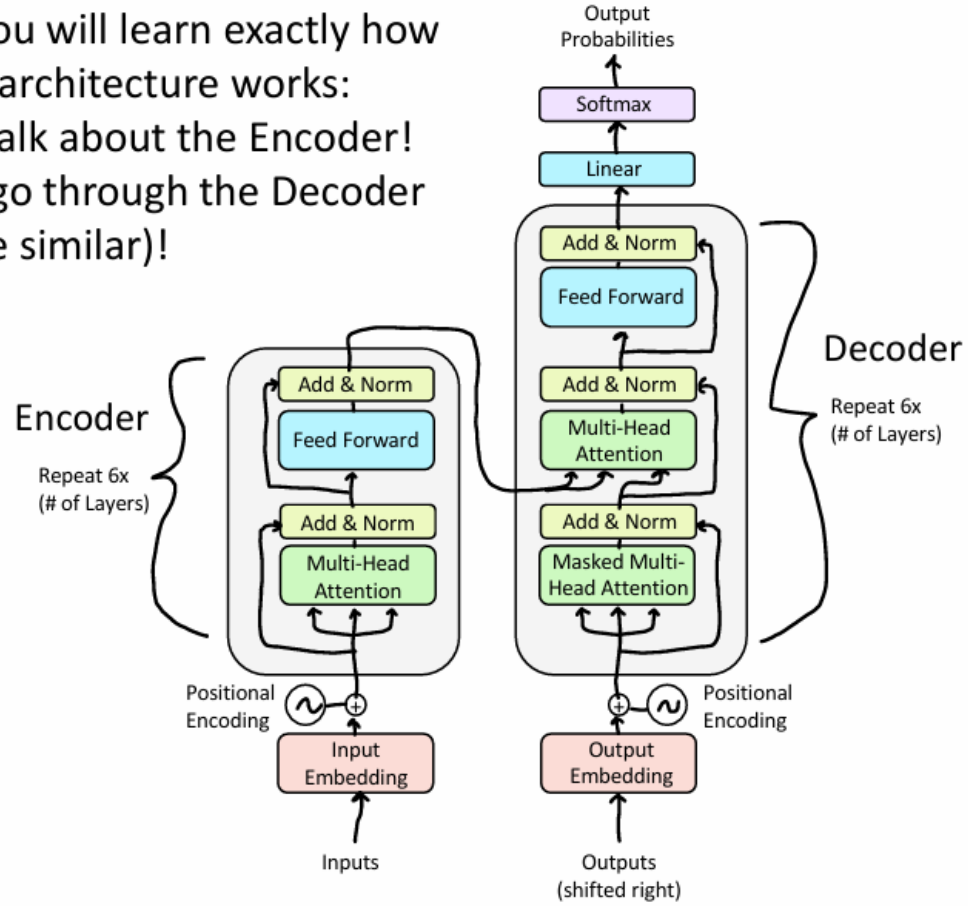


Transformer-Based Encoder-Decoder Model

The Transformer Encoder-Decoder

In this section, you will learn exactly how the Transformer architecture works:

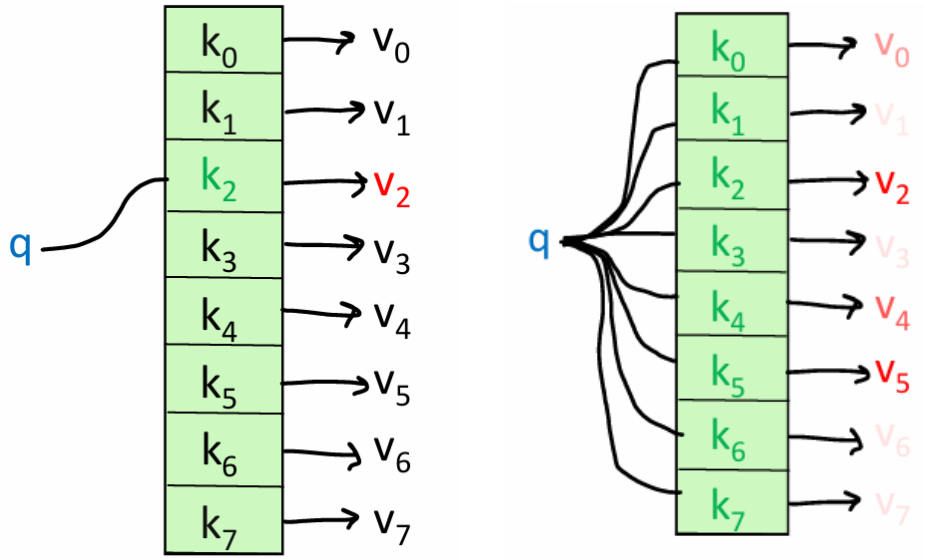
- First, we will talk about the Encoder!
- Next, we will go through the Decoder (which is quite similar)!



Intuition for Attention Mechanism

Let's think of attention as a "fuzzy" or approximate hashtable:

- To look up a value, we compare a query against keys in a table.
- In a hashtable (shown on the bottom left):
  - Each query (hash) maps to exactly one key-value pair.
- In (self-)attention (shown on the bottom right):
  - Each query matches each key to varying degrees.
  - We return a sum of values weighted by the query-key match.



Self-Attention in the Transformer Encoder

Step 1: For each word x_i , calculate its **query**, **key**, and **value**.

$$q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i$$

Step 2: Calculate attention score between **query** **and** **keys.**

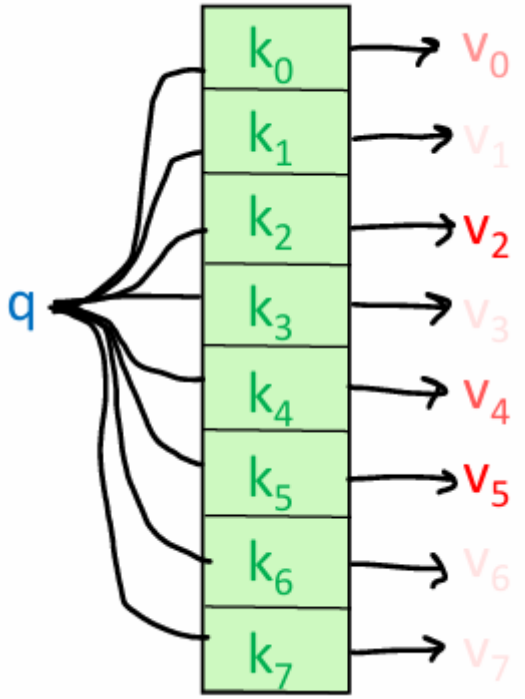
$$e_{ij} = q_i \cdot k_j$$

Step 3: Take the softmax to normalize attention scores.

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(e_{ij})}{\sum_k exp(e_{ik})}$$

Step 4: Take a weighted sum of **values**.

$$Output_i = \sum_j \alpha_{ij} v_j$$



(Vectorized) Self-Attention in the Transformer Encoder

Step 1: With embeddings stacked in X, calculate **queries**, **keys**, and **values**.

$$Q = XW^Q \quad K = XW^K \quad V = XW^V$$

Step 2: Calculate attention scores between **query** **and** **keys.**

$$E = QK^T$$

Step 3: Take the softmax to normalize attention scores.

$$A = softmax(E)$$

Step 4: Take a weighted sum of **values**.

$$Output = AV$$

$$Output = softmax\left(QK^T\right)V$$

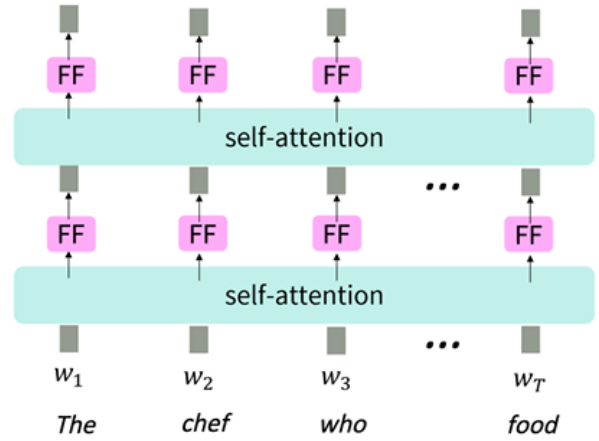
But attention isn't quite all you need!

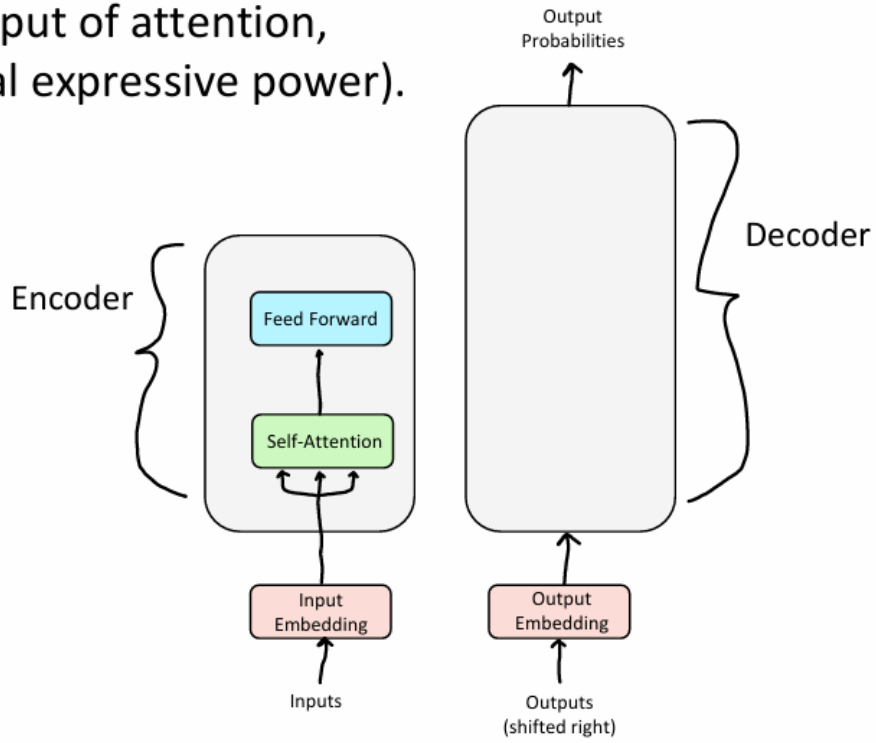
Problem: Since there are no element-wise non-linearities, self-attention is simply performing a re-averaging of the value vectors.

Easy fix: Apply a feedforward layer to the output of attention, providing non-linear activation (and additional expressive power).

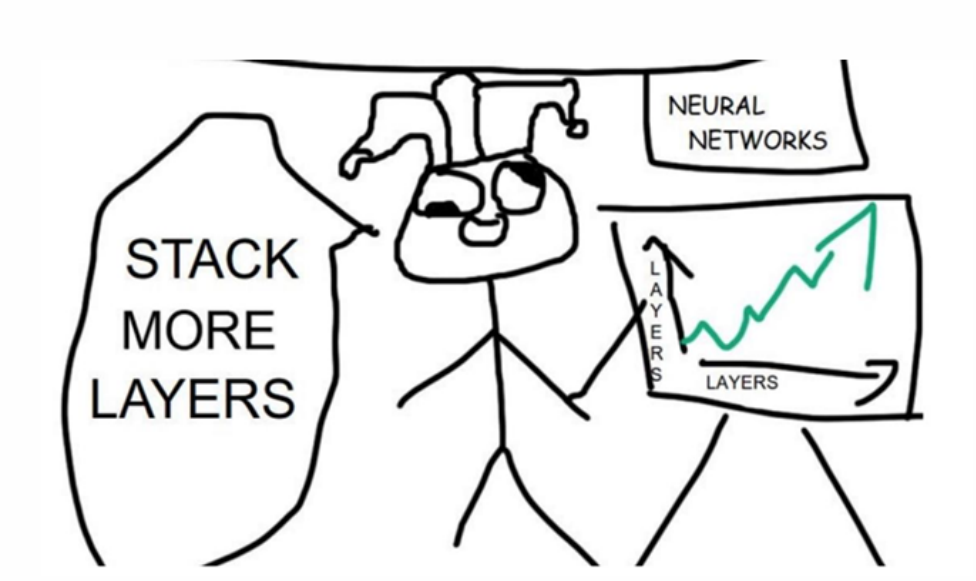
Equation for Feed Forward Layer

$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2$$

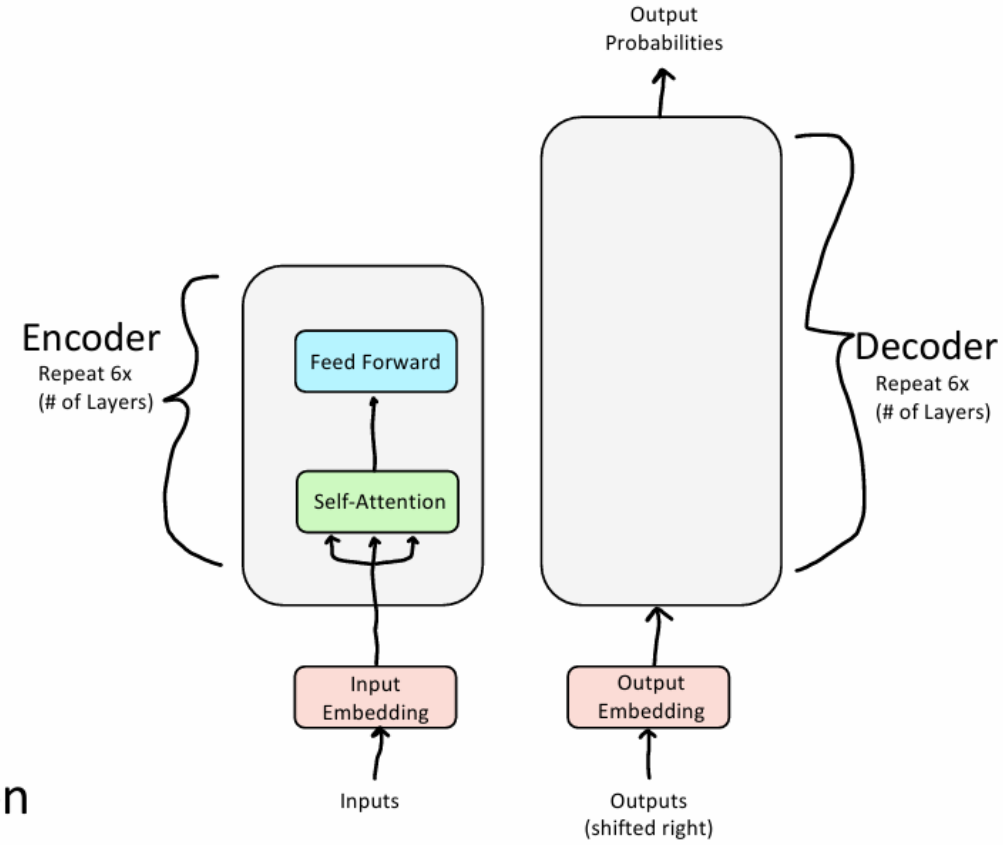




how do we make this work for deep networks



Training Trick #1: Residual Connections
Training Trick #2: LayerNorm
Training Trick #3: Scaled Dot Product Attention



Residual Connections

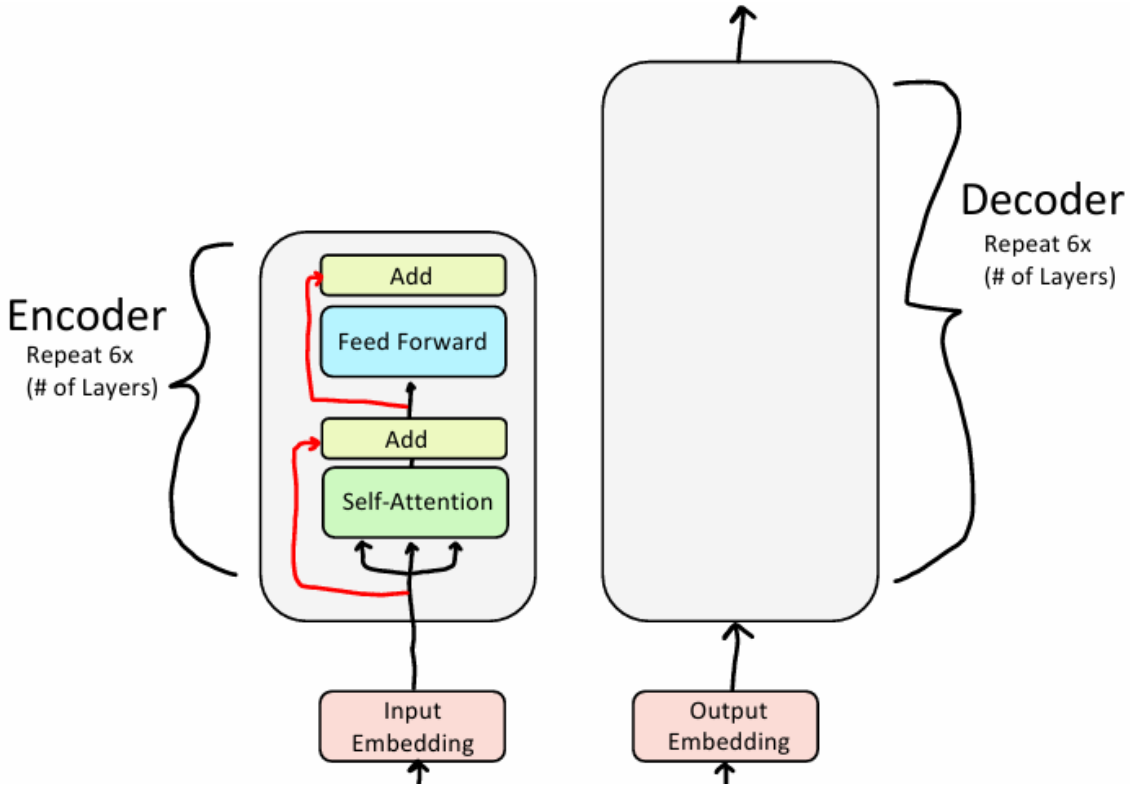
Residual connections are a simple but powerful technique from computer vision.

Deep networks are surprisingly bad at learning the identity function!

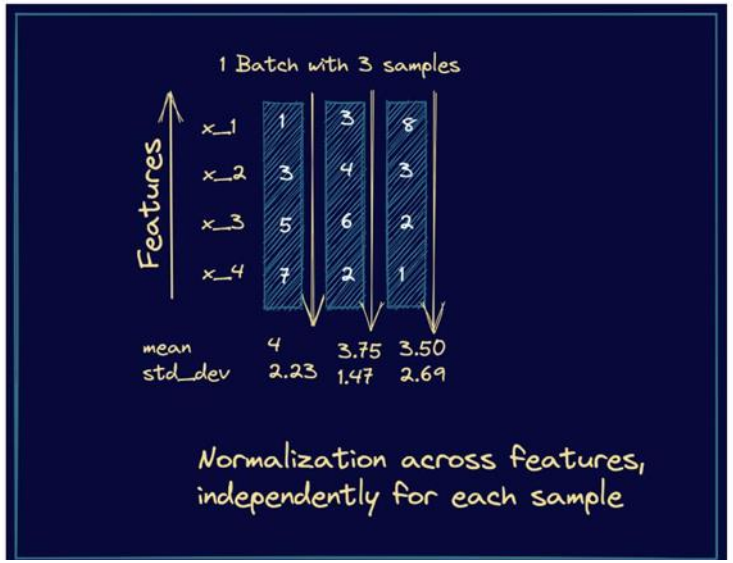
Therefore, directly passing "raw" embeddings to the next layer can actually be very helpful!

$$x_\ell = F(x_{\ell-1}) + x_{\ell-1}$$

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

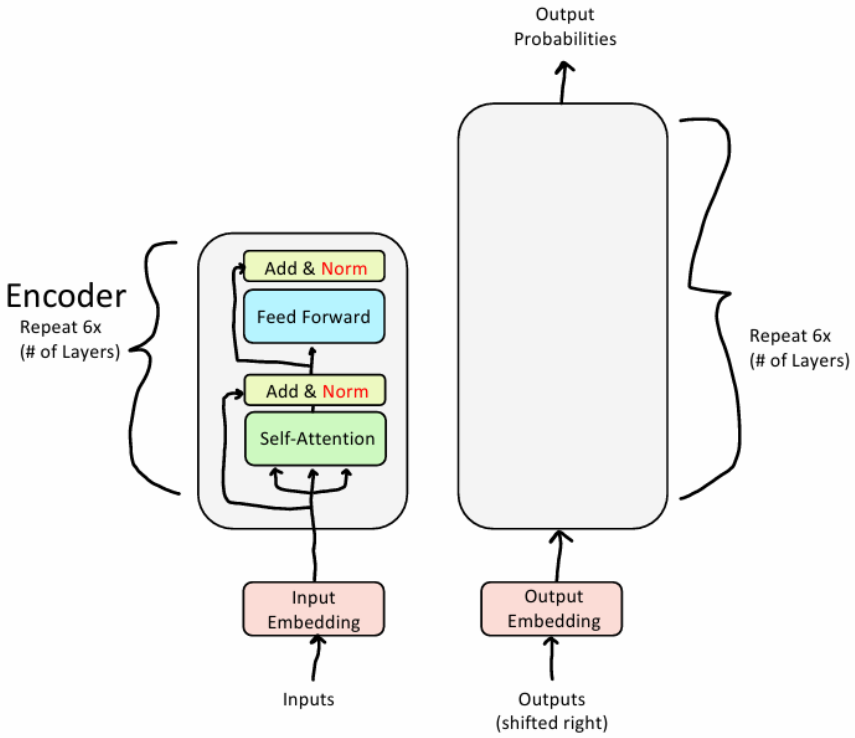


Layer Normalization



$$x^{\ell\prime} = \frac{x^{\ell} - \mu^{\ell}}{\sigma^{\ell} + \epsilon}$$

Mean: $\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$ Standard Deviation: $\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$

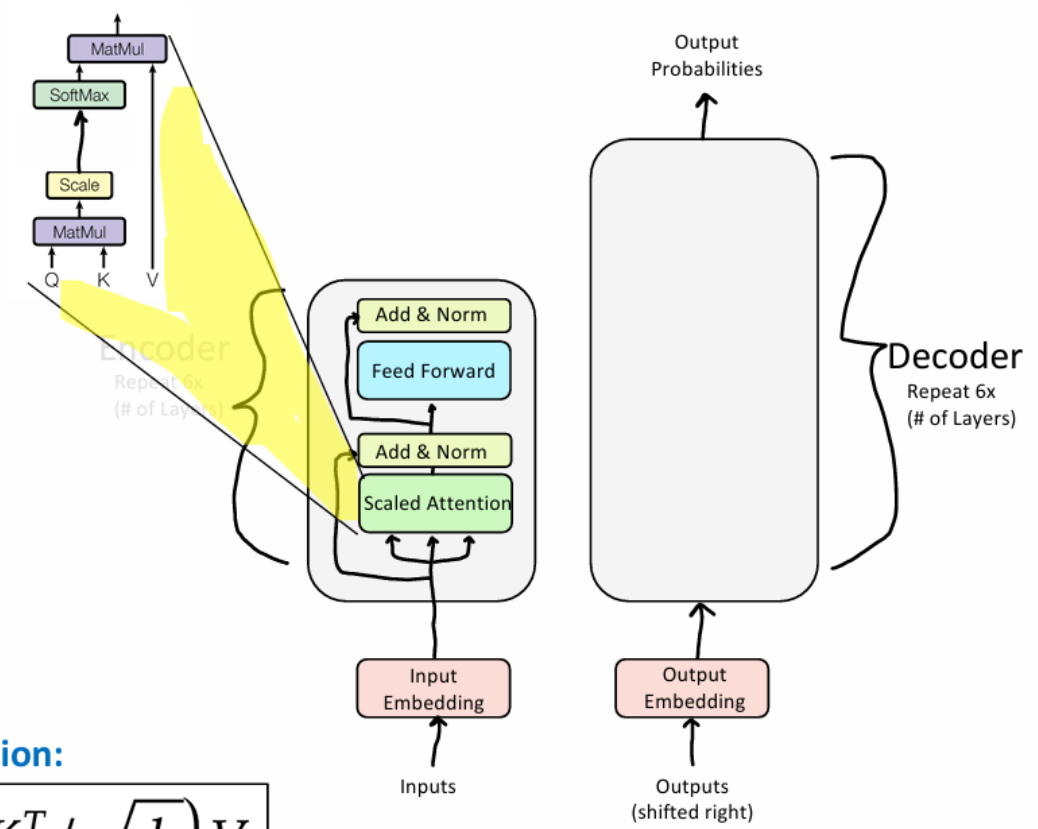


Scaled Dot Product Attention

- After LayerNorm, the mean and variance of vector elements is 0 and 1, respectively. (Yay!)
- However, the dot product still tends to take on extreme values, as its variance scales with dimensionality d_k

Quick Statistics Review:

- Mean of sum = sum of means = $d_k * 0 = 0$
- Variance of sum = sum of variances = $d_k * 1 = d_k$
- To set the variance to 1, simply divide by $\sqrt{d_k}$!



Updated Self-Attention Equation:

$$Output = softmax\left(QK^T / \sqrt{d_k}\right)V$$

Major issue!

We're almost done with the Encoder, but we have a major problem! Has anyone spotted it?

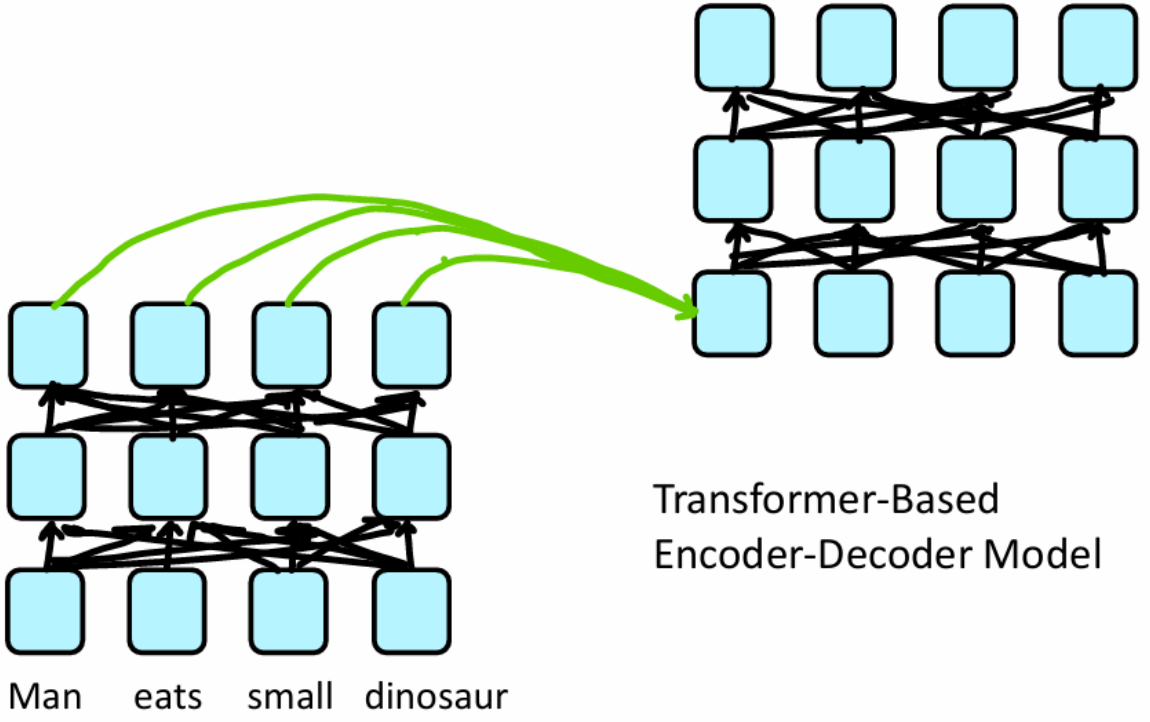
Consider this sentence:

- "Man eats small dinosaur."

Wait a minute, order doesn't impact the network at all!

This seems wrong given that word order does have meaning in many languages, including English!

$$Output = softmax\left(QK^T / \sqrt{d_k}\right)V$$



sequence order

Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.

Consider representing each sequence index as a vector

$p_i \in \mathbb{R}^d, for\ i \in \{1,2,\ldots,T\}$ are position vectors

Don't worry about what the pi are made of yet!

Easy to incorporate this info into our self-attention block: just add the pi to our inputs!

Let $\tilde{v}_i\ \tilde{k}_i, \tilde{q}_i$ be our old values, keys, and queries.

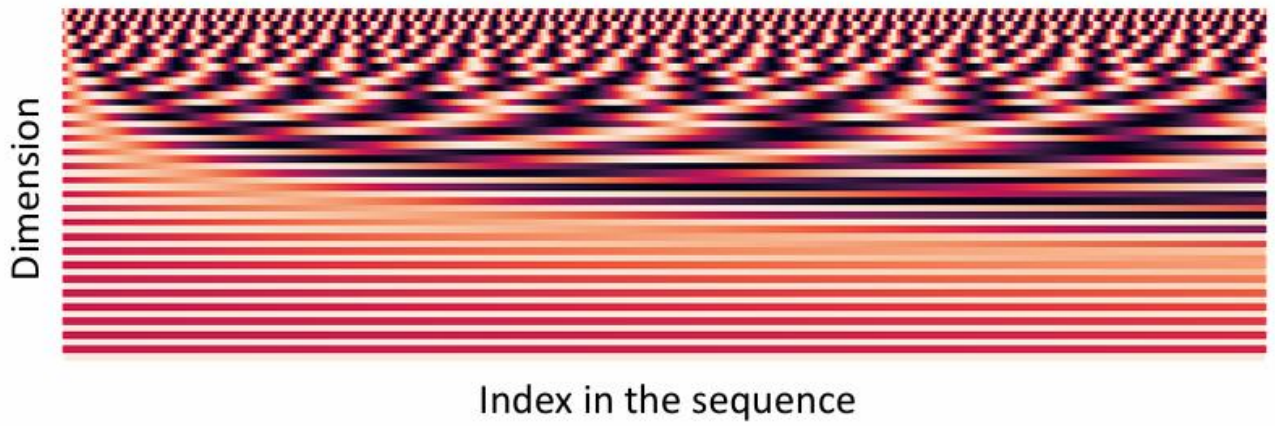
$$v_i = \tilde{v}_i + p_i$$
$$q_i = \tilde{q}_i + p_i$$
$$k_i = \tilde{k}_i + p_i$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add…

Position representation vectors

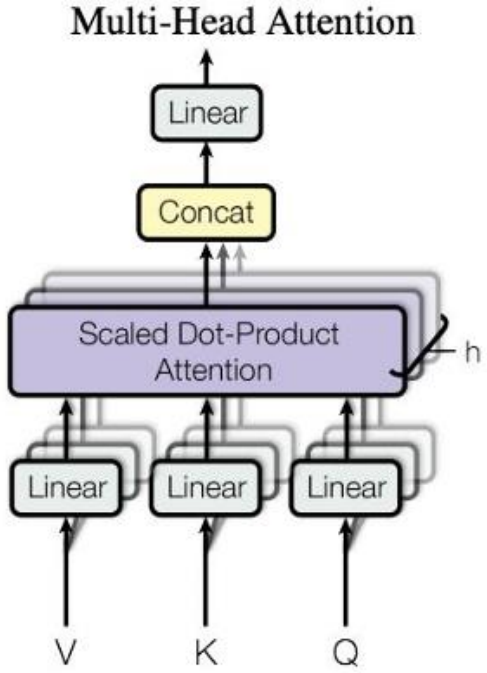
Sinusoidal position representations: concatenate sinusoidal functions of varying periods

$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



Multi-Headed Self-Attention

High-Level Idea: Let's perform self-attention multiple times in parallel and combine the results



Multi-headed Self-Attention

What if we want to look in multiple places in the sentence at once?

- For word i, self-attention "looks" where $x_i^\top Q^\top K x_j$ is high, but maybe we want to focus on different j for different reasons?

We'll define **multiple attention "heads"** through multiple Q,K,V matrices

Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and ℓ ranges from 1 to h.

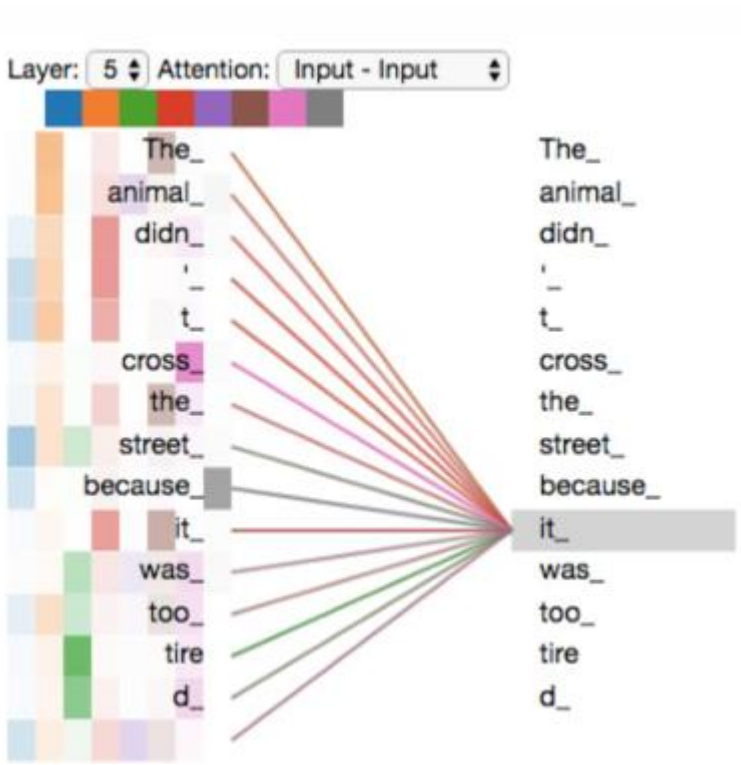
Each attention head performs attention independently:

- $\text{output}_\ell = \text{softmax}\left(XQ_\ell K_\ell^\top X^\top\right) * XV_\ell$, where $\text{output}_\ell \in \mathbb{R}^{d/h}$

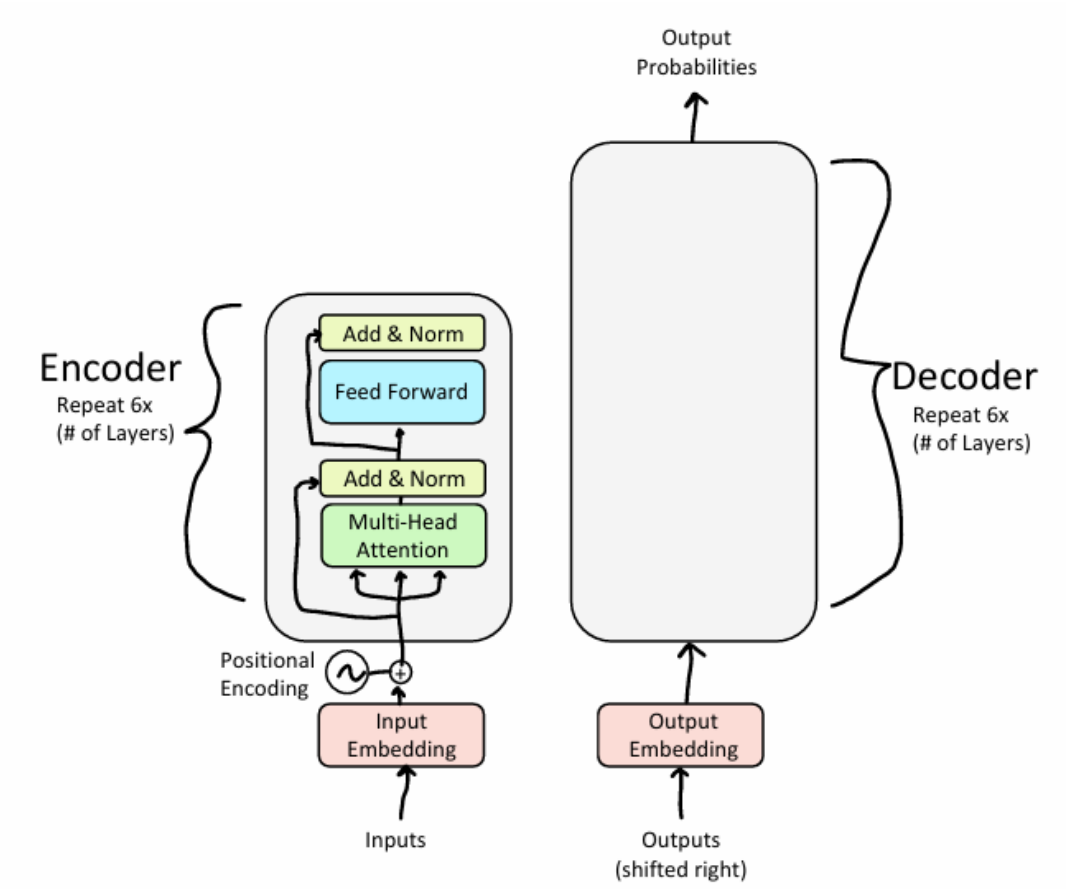
Then the outputs of all the heads are combined!

- $\text{output} = Y[\text{output}_1; \ldots; \text{output}_h]$, where $Y \in \mathbb{R}^{d \times d}$

Each head gets to "look" at different things, and construct value vectors differently.



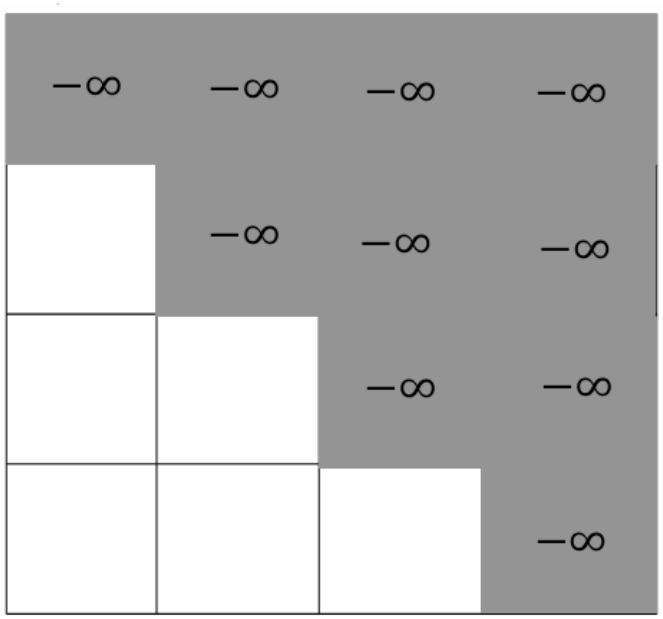
we've completed the Encoder! Time for the Decoder.



Decoder: Masked Multi-Head Self-Attention

Problem: How do we keep the decoder from “cheating”? If we have a language modeling objective, can't the network just look ahead and "see" the answer?

Solution: Masked Multi-Head Attention. At a high-level, we hide (mask) information about future tokens from the model.



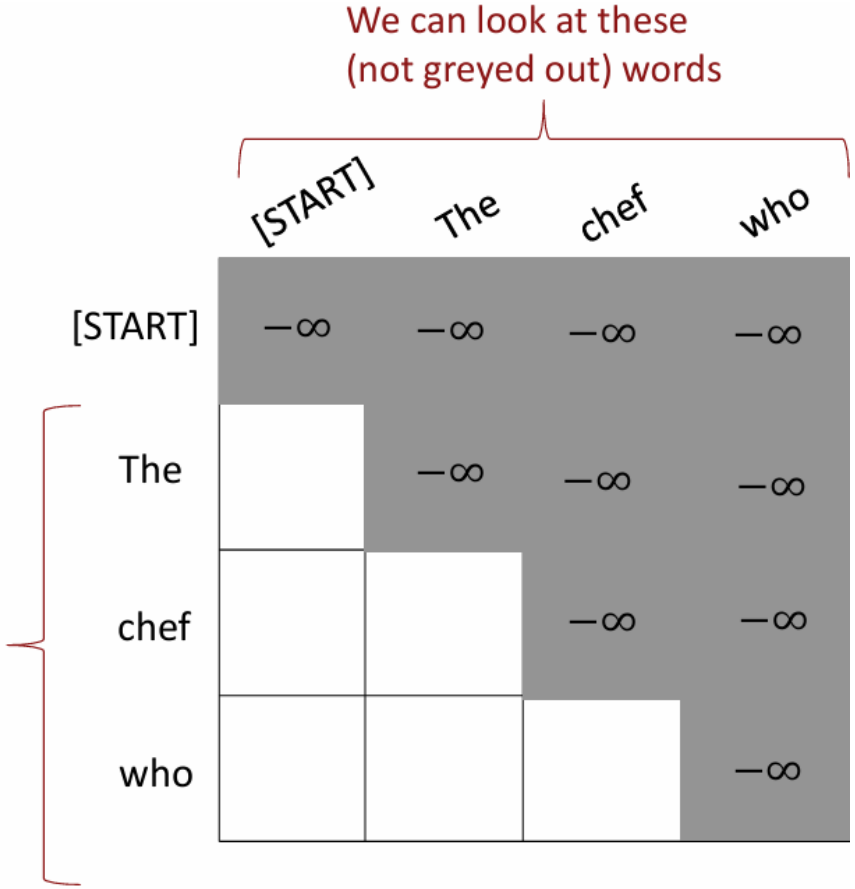
Masking the future in self-attention

To use self-attention in **decoders**, we need to ensure we can't peek at the future.

At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)

To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^\top k_j, j < i \\ -\infty, j \geq i \end{cases}$$



Encoder-Decoder Attention

We saw that self-attention is when keys, queries, and values come from the same source.

In the decoder, we have attention that looks more like what we saw last week.

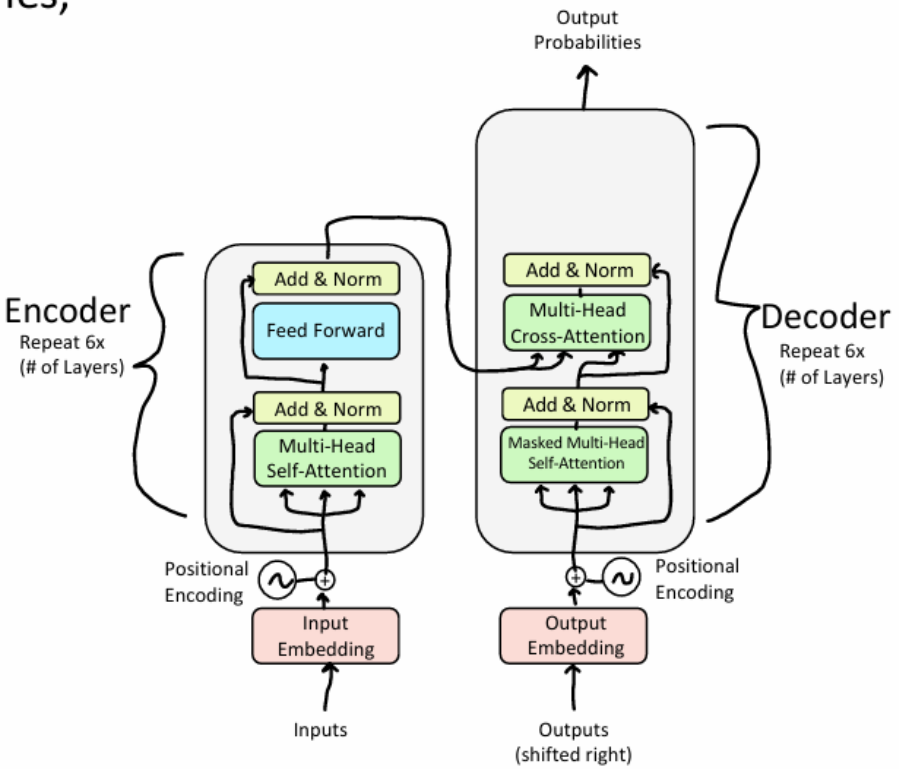
Let $h_1, \ldots, h_T$ be **output** vectors **from** the Transformer **encoder**; $x_i \in \mathbb{R}^d$

Let $z_1, \ldots, z_T$ be input vectors from the Transformer **decoder**, $z_i \in \mathbb{R}^d$

Then keys and values are drawn from the **encoder** (like a memory):

- $k_i = Kh_i, v_i = Vh_i$.

And the queries are drawn from the **decoder**, $q_i = Qz_i$.

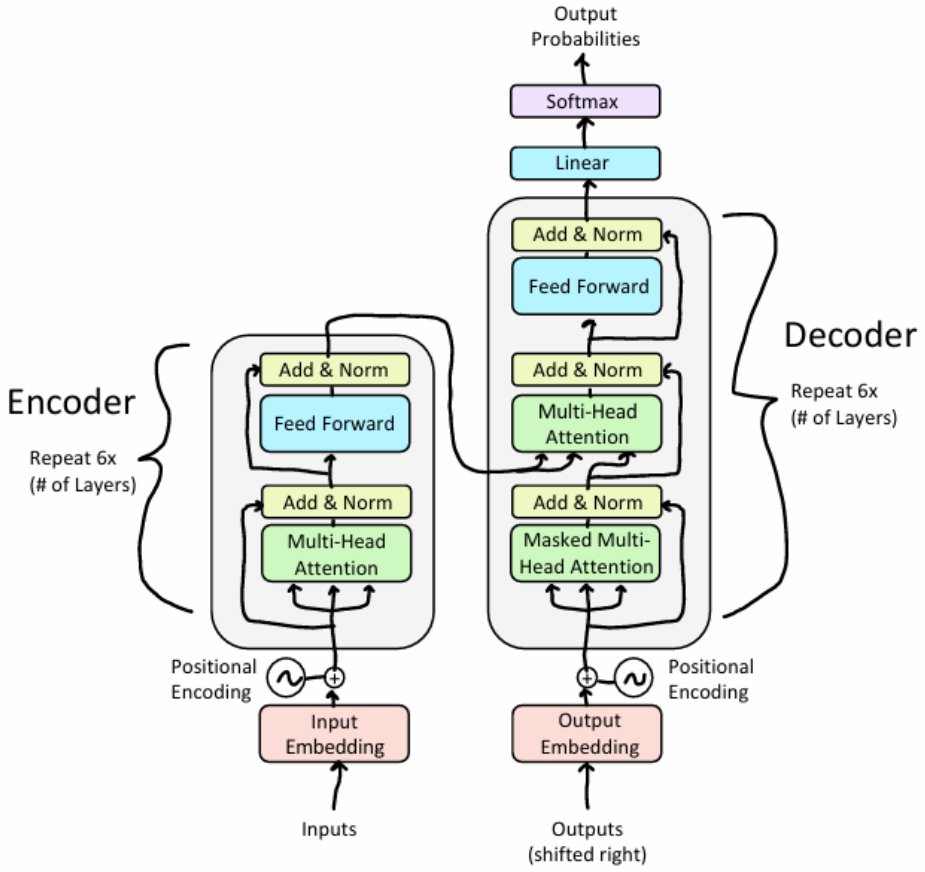


Decoder: Finishing touches!

Add a feed forward layer (with residual connections and layer norm)

Add a final linear layer to project the embeddings into a much longer vector of length vocab size (logits)

Add a final softmax to generate a probability distribution of possible next words



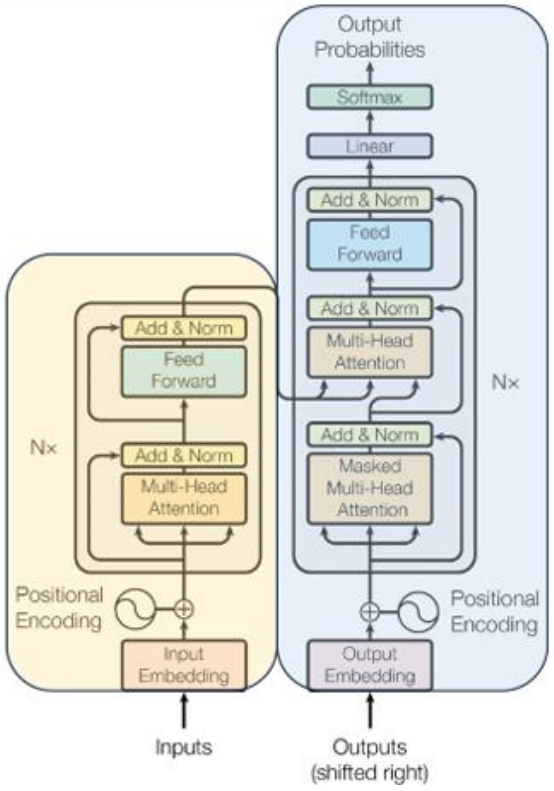
Transformer and LLM

Input – input tokens
Output – hidden states

Model can see all timesteps

Does not usually output tokens, so no inherent auto-regressivity

Representation



Input – output tokens and hidden states*
Output – output tokens

Model can only see previous timesteps

Model is auto-regressive with previous timesteps' outputs

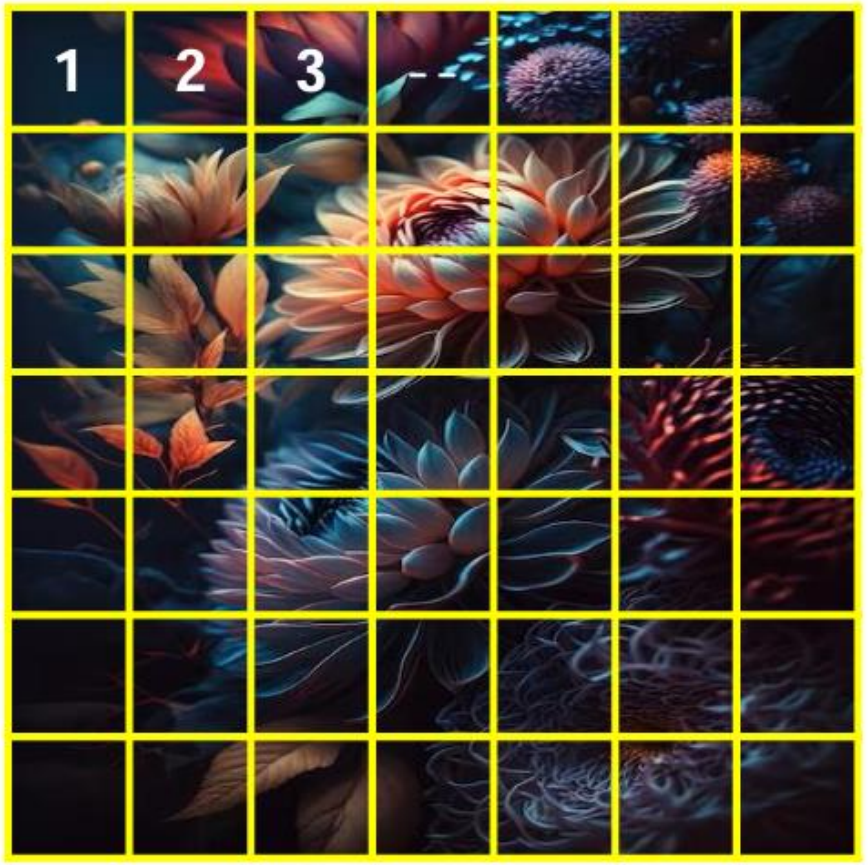
Generation

Vision Transform (ViT)

Image and Sequence?

Token~ Image Patches

Scan the patch and build sequence

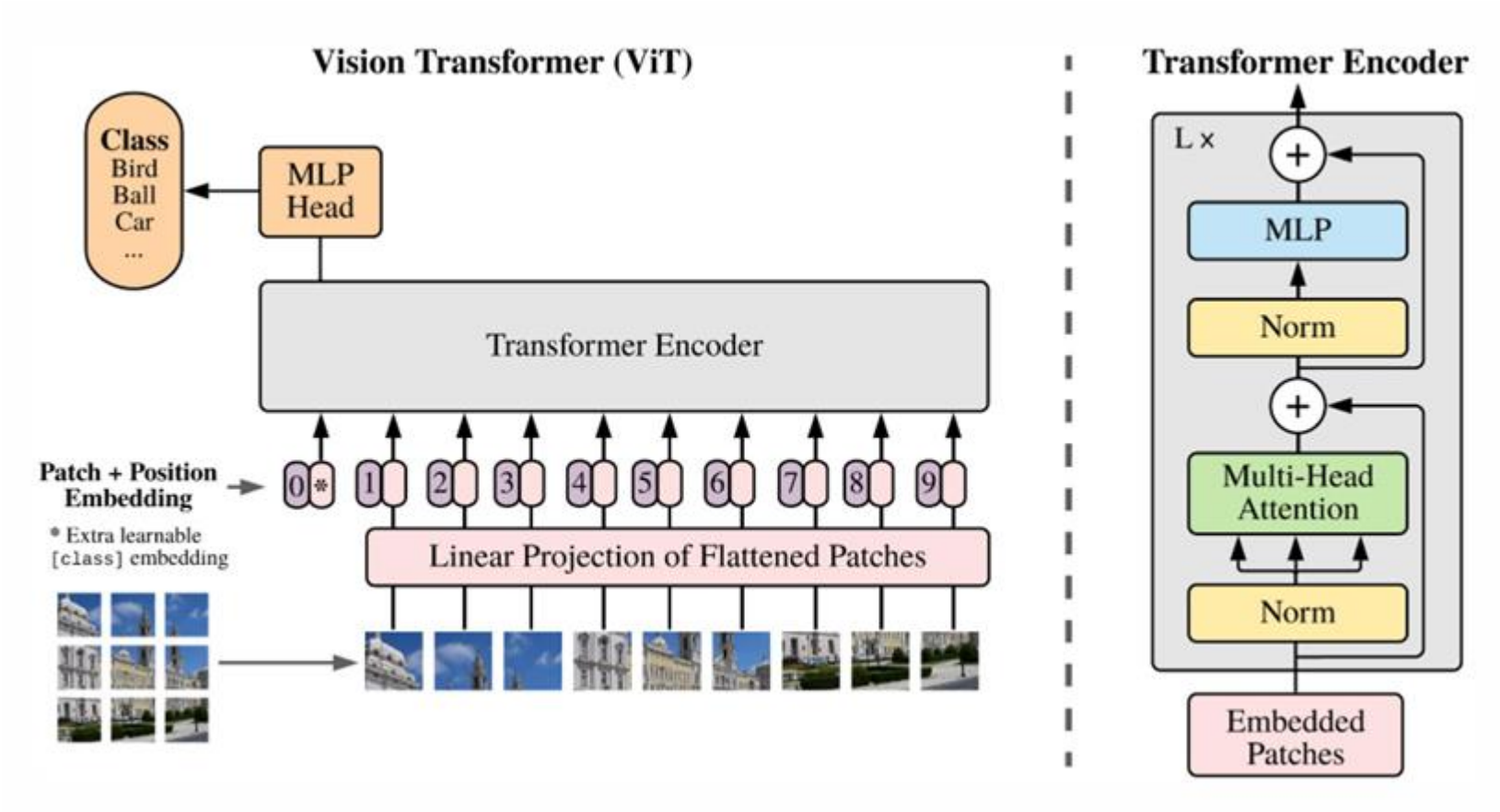


VIT

ViT steps:

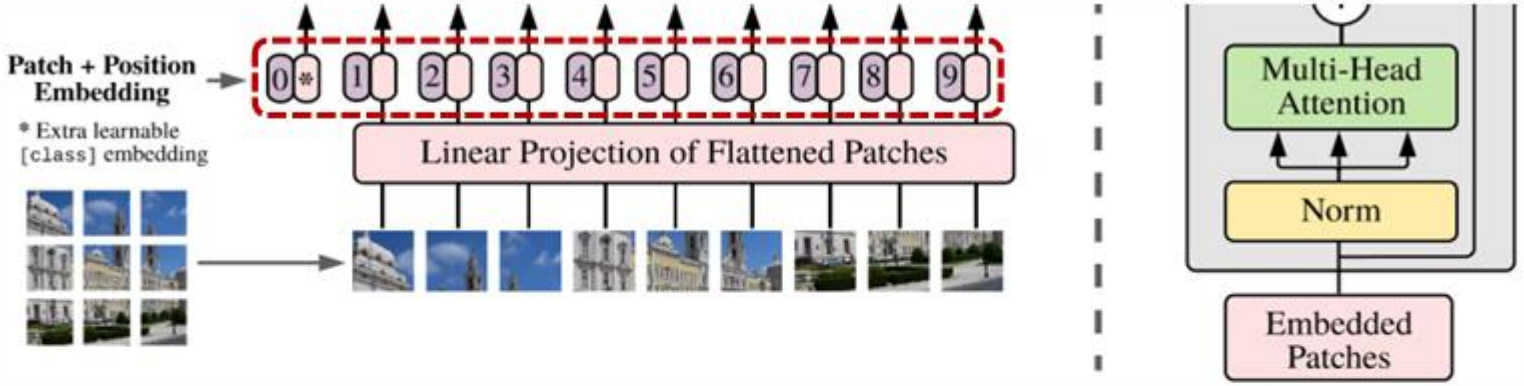
- Split an image into patches ($\mathbb{R}D\times D$)
- Flatten the patches ($\mathbb{R}D2$)
- Produce a linear embeddings from the flattened patches (learnable), known as patch embedding.
- Add positional embeddings
- Feed the sequence as an input to a standard transformer encoder
- Pretrain the model with image labels (fully supervised on a huge dataset)
- Finetune on the downstream dataset for image classification

ViT Configuration



ViT Embedding

(Patch+Class)/Position Embedding:



$0 - 1 - 2 - \cdots - N$: $(N + 1)$ Position Encoding (0 : class token position embedding)

$* - E_1 - E_2 - \cdots - E_N$: $(N + 1)$ Patch Embedding ($*$: learnable class token)

The output of [class] token is transformed into a class prediction via a small MLP with tanh as non-linearity in the single hidden layer.

Application

- Image Classification
- Image Captioning
- Image Segmentation
- Anomaly detection
- Action Recognition
- Autonomous Driving