LLM LE2 VT2026

Basics

Fredrik Heintz

Dept. of Computer Science
Linköping University

fredrik.heintz@liu.se
@FredrikHeintz

Outline:

- Language Models
- Neural Language Models
- Word embeddings
- Transformers

LINKÖPING UNIVERSITY

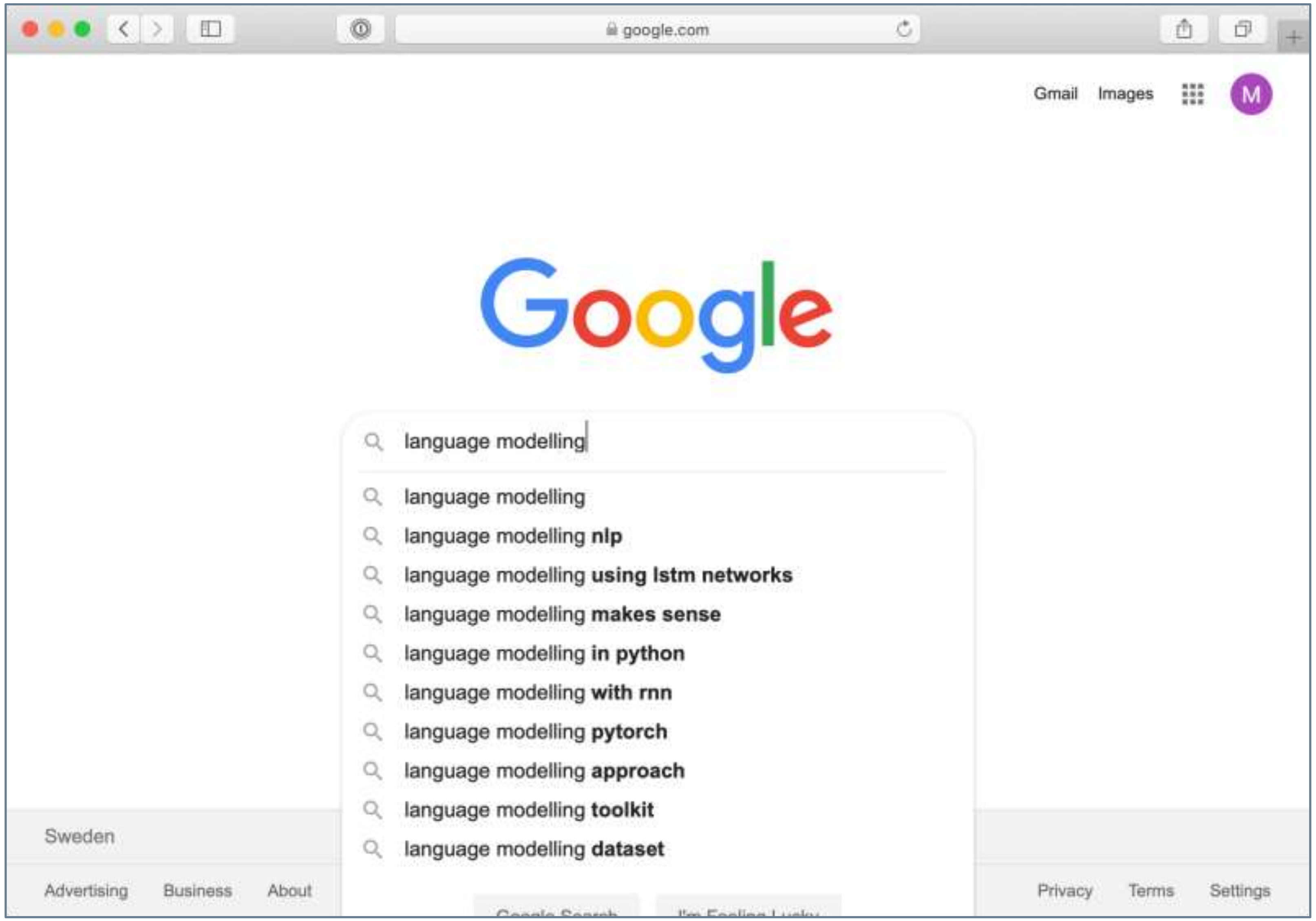
What is Language Modeling?

The task of computing $P(w \mid h)$, the probability distribution of possible words w from a vocabulary given some history (sequence of words) h.

… and thanks for all the _____

fish	0.70
memories	0.10
support	0.05
help	0.05
love	0.04
time	0.02
work	0.02
fun	0.01
…	…

Funded by the European Union



language modelling

language modelling
language modelling **nlp**
language modelling **using lstm networks**
language modelling **makes sense**
language modelling **in python**
language modelling **with rnn**
language modelling **pytorch**
language modelling **approach**
language modelling **toolkit**
language modelling **dataset**

Language modelling

- **Language modelling** is the task of predicting which word comes next in a sequence of words.
- More formally, given a sequence of words $w_1, \dots, w_t$ we want to know the probability of the next word, w_{t+1}:

$$P(w_{t+1} \mid w_1, \dots, w_t)$$

- We are assuming that w_{t+1} comes from a finite vocabulary *V*.

  language models = classifiers

An alternative view on language models

- Rather than as predictive models, language models can also be viewed as models that assign a probability to a piece of text.

  How likely is it that this piece of text is written in Swedish? French?

- These two views are equivalent, as the probability of a sequence can be expressed as a product of conditional probabilities: *

$$P(w_1 \cdots w_N) = \prod_{t=1}^{N} P(w_t \mid w_1, \ldots, w_{t-1})$$

Motivation for Language Modeling

- Generating more plausible sentences.
  - P(“I saw a van”) > P(“eyes awe of an”)
  - Many NLP applications: correcting grammar or spelling errors, machine translation, speech recognition, content summarization, conversational agents, etc.

- More importantly, Large Language Models are built by training them on this task!

Funded by the European Union

An Approximation

- A problem: computing *P(w | h)* exactly is infeasible for arbitrary history *h* since language is **creative** and *h* might have never occurred before!

- Idea: the Markov assumption: approximate the history by just the last few words.

$$P(w_k|w_{1:k-1}) \approx P(w_k|w_{k-n+1:k-1})$$

- Example: n-gram models: look at n-1 words in the history. n=2 is bigrams, n=3 is trigrams, etc.

  LLMs use **much** larger n, in the thousands or even millions!

Funded by the European Union

N-gram language models

- An **n-gram** is a contiguous sequence of *n* words (or characters).

Sherlock **Holmes** had **sprung out** and seized the intruder **by the collar**.

unigram **bigram** **trigram**

- An **n-gram model** specifies conditional probabilities for the last word in an *n*-gram, given the previous words:

$$P(w_n \mid w_1 \cdots w_{n-1})$$

Intuition behind n-gram models

- By the chain rule, the probability of a sequence of N words can be computed using conditional probabilities as

$$P(w_1 \cdots w_N) = \prod_{k=1}^{N} P(w_k \mid w_1 \cdots w_{k-1})$$

- To make probability estimates more robust, we approximate the full history $w_1, \dots, w_N$ by overlapping n-gram windows:

$$P(w_1 \cdots w_N) = \prod_{k=1}^{N} P(w_k \mid w_{k-n+1} \cdots w_{k-1})$$

Funded by the European Union

Formal definition of an n-gram model

n	the model's order (1 = unigram, 2 = bigram, …)
V	a finite set of possible words; the vocabulary
$P(w\|u)$	a probability that specifies how likely it is to observe the word *w* after the context (*n*-1)-gram *u* one value for each combination of a word *w* and a context *u*

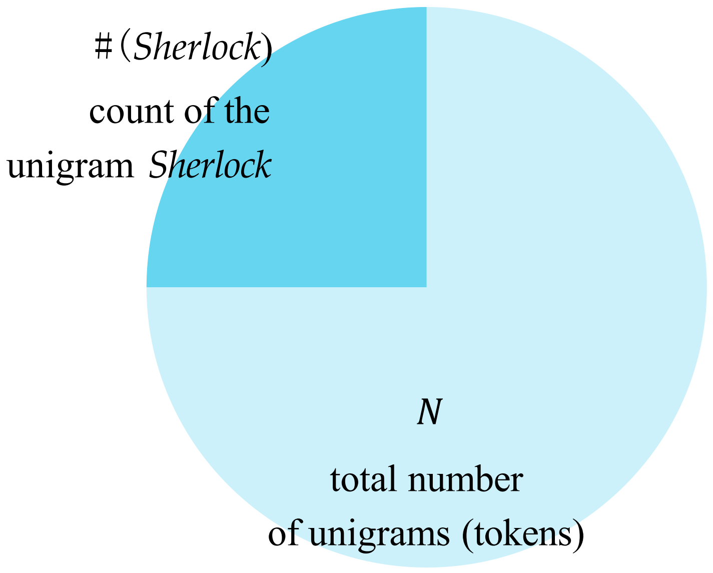
Estimation of n-gram models

- The simplest method for estimating n-gram models is **maximum likelihood estimation (MLE)**.
  - maximise the likelihood of the observations given the parameters
- We want to find model parameters (here, probabilities) that maximise the likelihood of some text data.
- It turns out that we can solve this problem by simply counting occurrences of n-grams and normalising.
  - formal derivation uses Lagrange multipliers

Funded by the European Union

MLE of unigram probabilities

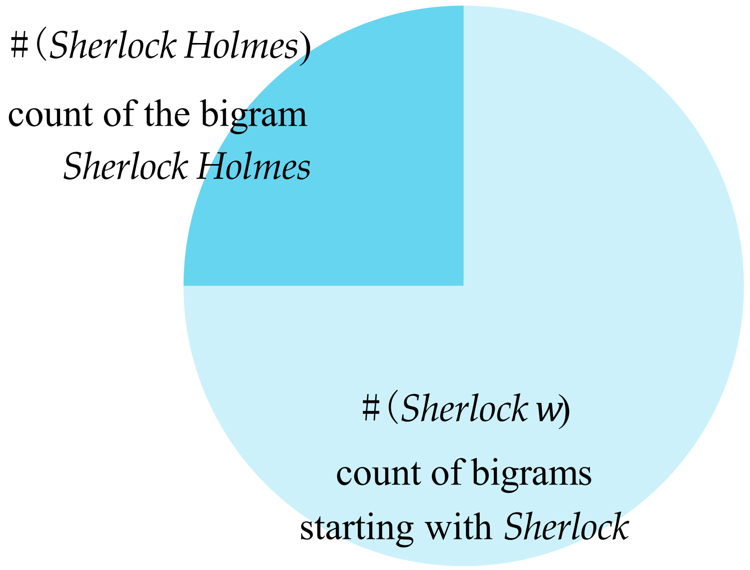
$P(\textit{Sherlock})$



$$P(w) = \frac{\#(w)}{N}$$

MLE of bigram probabilities

$P(\mathit{Holmes} \mid \mathit{Sherlock})$


#(*Sherlock Holmes*)
count of the bigram *Sherlock Holmes*
#(*Sherlock w*)
count of bigrams starting with *Sherlock*

$$P(w \mid u) = \frac{\#(uw)}{\#(u\bullet)}$$

$$P(w \mid u) = \frac{\#(uw)}{\#(u)}$$

Sparsity problems

$$P(w \mid \text{students opened their}) = \frac{\#(\text{students opened their } w)}{\#(\text{students opened their})}$$

Funded by the European Union

Smoothing

- In **smoothing**, we "spread out the probability mass" over the possible outcomes more evenly than MLE would do.
- A substantial amount of research in language modelling has been devoted to the development of advanced smoothing techniques.
  - additive smoothing, absolute discounting, Kneser–Ney smoothing, …

An Approximation

- A problem: computing *P(w | h)* exactly is infeasible for arbitrary history *h* since language is **creative** and *h* might have never occurred before!

- Idea: the Markov assumption: approximate the history by just the last few words.

$$P(w_k|w_{1:k-1}) \approx P(w_k|w_{k-n+1:k-1})$$

- Example: n-gram models: look at n-1 words in the history. n=2 is bigrams, n=3 is trigrams, etc.

  LLMs use **much** larger n, in the thousands or even millions!

Funded by the European Union

Estimating Probabilities

Question: What is the objective of this model?

Goal: Train a language model P_θ with parameters (weights) θ such that $P_\theta(h)$ computes a probability distribution over the vocabulary of all possible words.

Start with a dataset $D_{train} = \{ (h_1, w_1), \ldots, (h_n, w_n) \}$.

Assumption: i.i.d. (independent and identically distributed)

A good model is one that makes the data look probable. Therefore, choose θ such that

$$P(D_{train}) = \prod_{i=1}^{n} P(h_i).P_\theta(w_i \mid h_i) \text{ is maximized.}$$

Funded by the European Union

Multiplying Probabilities

$$P(D_{train}) = \prod_{i=1}^{n} P(h_i).P_\theta(w_i \mid h_i)$$

Multiplying together many numbers <= 1

$$\log P(D_{train}) = \sum_{i=1}^{n} \log P(h_i) + \log P_\theta(w_i \mid h_i) = \sum_{i=1}^{n} \log P_\theta(w_i \mid h_i) + \cancel{\text{const}}$$

$$\theta^* \leftarrow \underset{\theta}{\arg\max} \sum_{i=1}^{n} \log P_\theta(w_i \mid h_i)$$

maximum likelihood estimation (MLE)

This is our **loss function**

$$\theta^* \leftarrow \underset{\theta}{\arg\min} -\frac{1}{n} \sum_{i=1}^{n} \log P_\theta(w_i \mid h_i)$$

negative log likelihood (NLL)

Also called *cross-entropy*

Neural Language Models

TrustLLM

Funded by
the European Union

Neural Language Models: Unconditioned vs Conditioned

Neural language models can either be designed to just predict the next word given the previous ones, or they can be designed to predict the next word given the previous ones and some additional conditioning sequence.

Unconditioned: $P(Y)$

- At each step the LM predicts: $P(y_t \mid y_{1:t-1})$
- Examples:
  - GPT
  - LLaMA

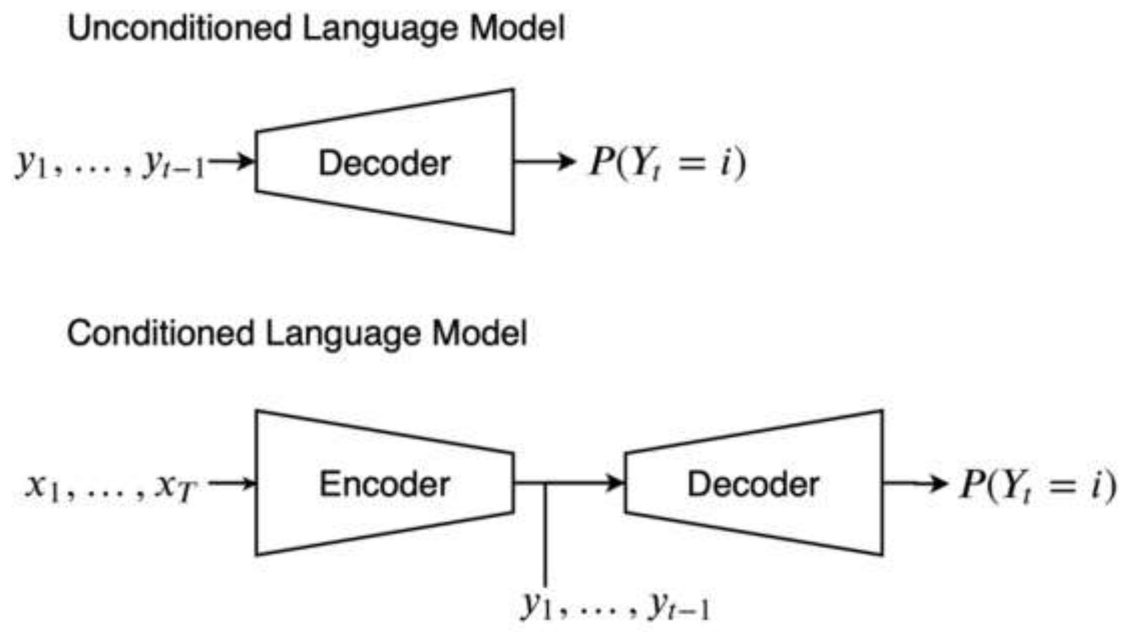
Conditioned: $P(Y|X)$

- At each step the LM predicts: $P(y_t \mid y_{1:t-1}, x_{1:T})$
- Examples
  - T5
  - Most machine translation models
  - Sometimes called sequence-to-sequence or seq2seq models.

Funded by the European Union

Neural Language Models: Unconditioned vs Conditioned

- Unconditioned neural language models only have a decoder.
- Conditioned ones have an encoder and a decoder.



Neural Language Models: Unconditioned vs Conditioned

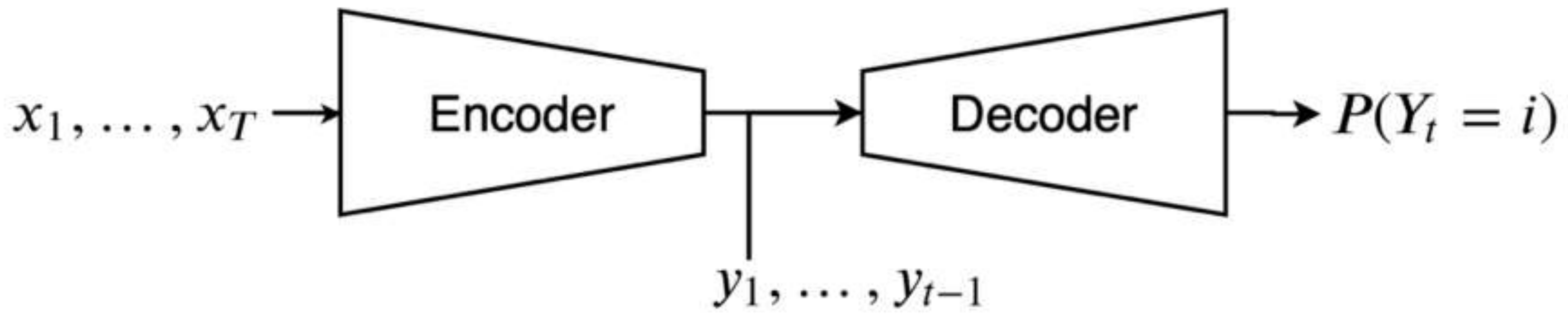
- Theoretically, any task designed for a decoder-only architecture can be turned into one for an encoder-decoder architecture, and vice-versa.
- TASK: Continue the sequence.
- Decoder-only version:
  - P(Y="Once upon a time there lived a dreadful ogre.")
- Encoder-decoder version:
  - P(Y="lived a dreadful ogre." | X="Once upon a time there")

Neural Language Models: Unconditioned vs Conditioned

- Theoretically, any task designed for a decoder-only architecture can be turned into one for an encoder-decoder architecture, and vice-versa.
- TASK: Translate from English to French.
- Decoder-only version:
  - P(Y="English: The hippo ate my homework. French: L'hippopotame a mangé mes devoirs.")
- Encoder-decoder version:
  - P(Y="L'hippopotame a mangé mes devoirs." | X="The hippo ate my homework.")

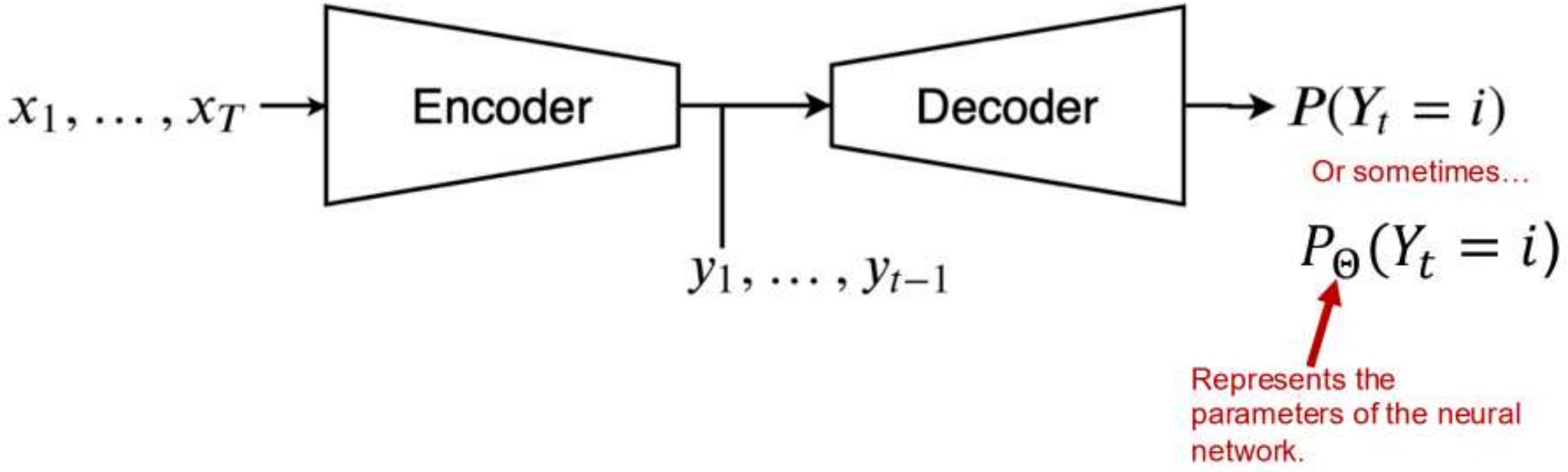
Language Models: Important Terms

Input sequence: $x_1, \ldots, x_T$
Target sequence: $y_1, \ldots, y_T$



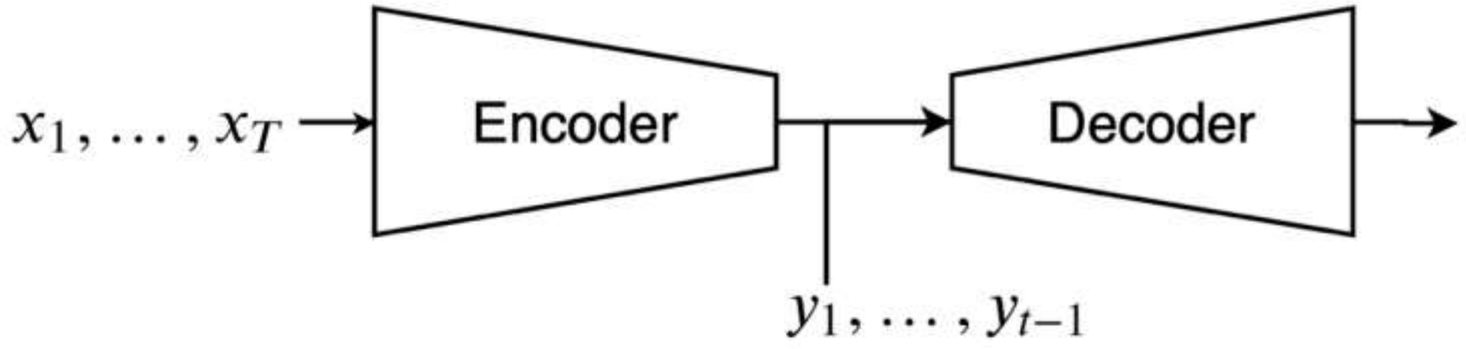
Language Models: Important Terms

Input sequence: $x_1, \ldots, x_T$
Target sequence: $y_1, \ldots, y_T$



Language Models: Important Terms

Input sequence: $x_1, \dots, x_T$
Target sequence: $y_1, \dots, y_T$



Or sometimes…

$$P_\Theta(Y_t = i)$$

Or sometimes…

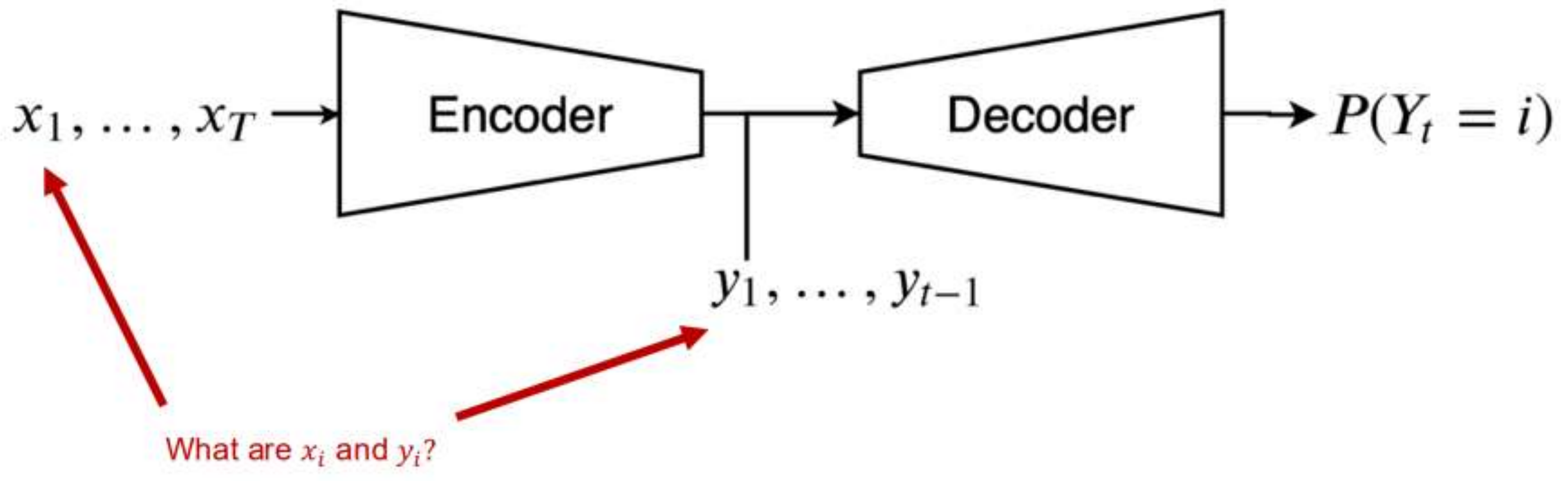
$$P(Y_t = i | y_1, \dots, y_{t-1}; x_1, \dots, x_T; \Theta)$$

Funded by the European Union

Language Models: Important Terms

Input sequence: $x_1, \dots, x_T$

Target sequence: $y_1, \dots, y_T$

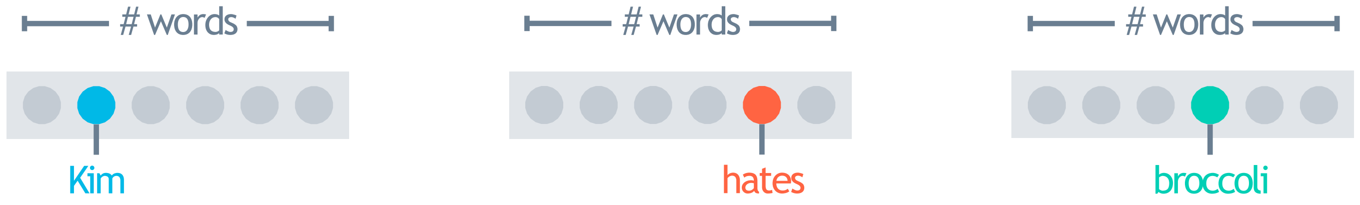


Word Embeddings

Funded by the European Union

One-hot vectors

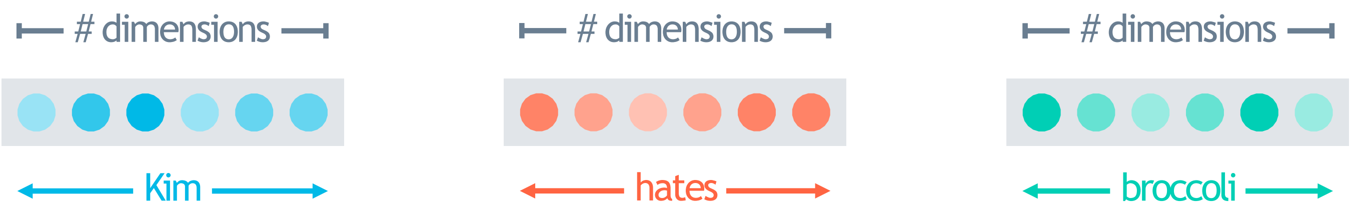
- To process words using neural networks, we need to represent them as vectors of numerical values.
- The classical way to do this is to use **one-hot vectors** – vectors in which all components but one are zero.



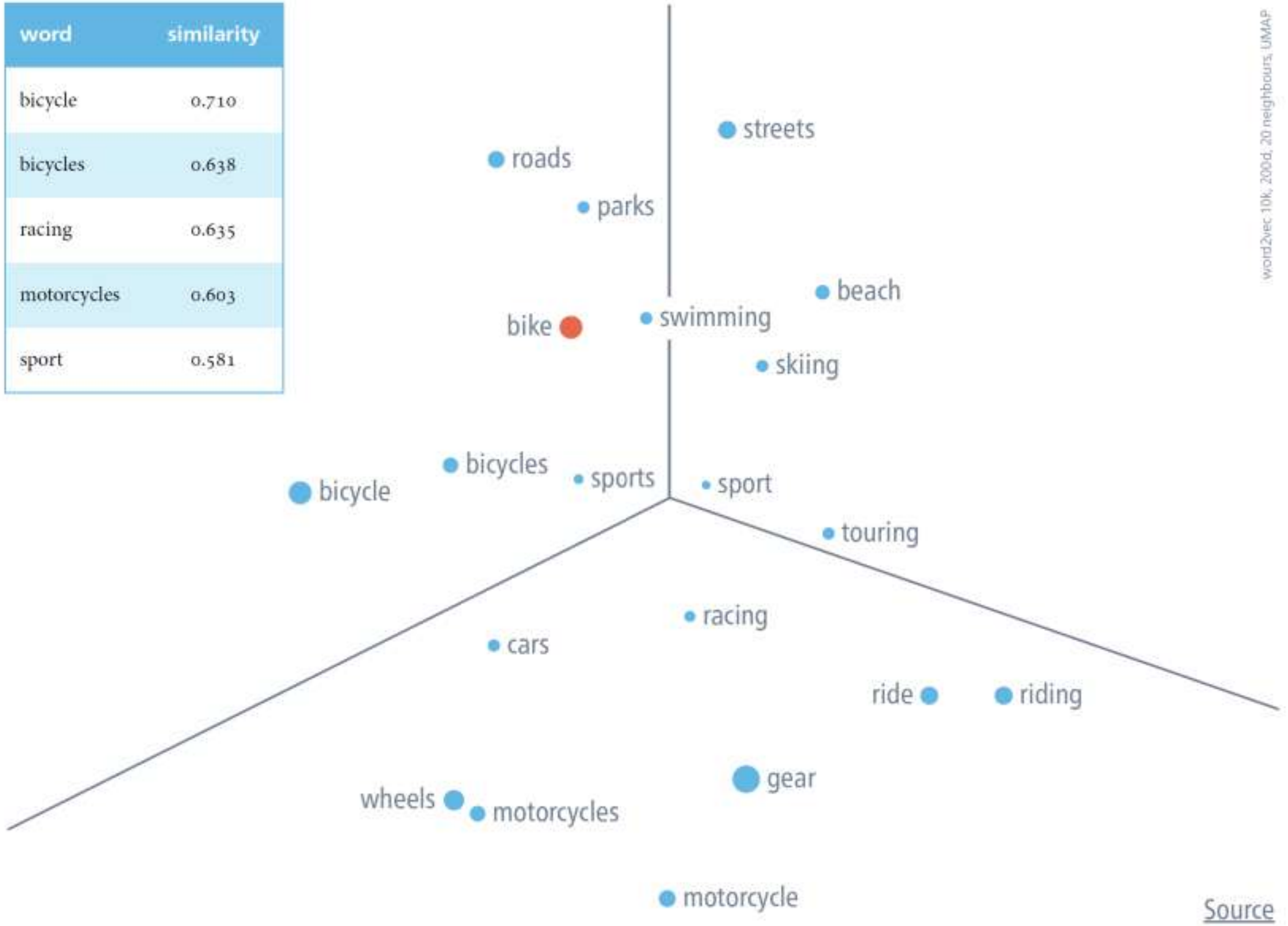
Word embeddings

Compared to one-hot vectors, **word embeddings**

- are shorter but dense
- support a useful notion of similarity
- can be learned from data



word	similarity
bicycle	0.710
bicycles	0.638
racing	0.635
motorcycles	0.603
sport	0.581



Source

TrustLLM

Funded by the European Union

You shall know a word by the company it keeps

What do the following sentences tell us about *Garrotxa*?

- *Garrotxa* is made from milk.
- *Garrotxa* pairs well with crusty country bread.
- *Garrotxa* is aged in caves to enhance mould development.

Sentences taken from the English Wikipedia

Funded by the European Union

The distributional hypothesis

- The **distributional hypothesis** states that words with similar distributions have similar meanings.

  with similar distributions = are used and occur in the same contexts

- This suggests that we can learn word representations from co-occurrence statistics.

  similar co-occurrence distributions = similar meanings

Funded by the European Union

Co-occurrence matrix

	cheese	bread	goat	sheep
cheese				
bread				
goat				
sheep				

as olives cheese or bread

Funded by the European Union

Co-occurrence matrix

	cheese	bread	goat	sheep
cheese		1		
bread				
goat				
sheep				

as olives cheese or bread

of sheep cheese and milk

TrustLLM

Funded by the European Union

Co-occurrence matrix

	cheese	bread	goat	sheep
cheese		1		1
bread				
goat				
sheep				

as olives cheese or bread

of sheep cheese and milk

goat milk cheese can be

Funded by the European Union

Co-occurrence matrix

	cheese	bread	goat	sheep
cheese		1	1	1
bread				
goat				
sheep				

as olives cheese or bread

of sheep cheese and milk

goat milk cheese can be

bread and cheese for breakfast

Funded by the European Union

Co-occurrence matrix

	cheese	bread	goat	sheep
cheese		2	1	1
bread				
goat				
sheep				

as olives cheese or bread

of sheep cheese and milk

goat milk cheese can be

bread and cheese for breakfast

macaroniand cheese with bread

Funded by the European Union

Co-occurrence matrix

	cheese	bread	goat	sheep
cheese		3	1	1
bread				
goat				
sheep				

as olives cheese or bread

of sheep cheese and milk

goat milk cheese can be

bread and cheese for breakfast

macaroniand cheese with bread

Funded by the European Union

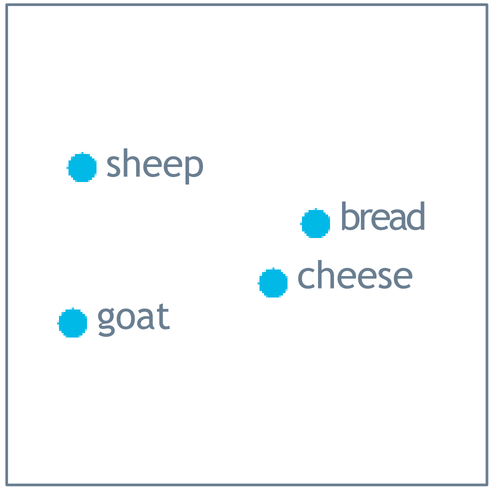
Co-occurrence matrix

	cheese	bread	goat	sheep
chees e	14	7	5	1
bread	7	12	0	0
goat	5	0	8	12
sheep	1	0	12	2

word vector for *cheese*

Funded by the European Union

Vector similarity = meaning similarity



vector space (PCA)

	cheese	bread	goat	sheep
cheese	1.00	0.80	0.49	0.38
bread	0.80	1.00	0.17	0.04
goat	0.49	0.17	1.00	0.67
sheep	0.38	0.04	0.67	1.00

cosine similarities $\cos(x, y) = \frac{x^{\top} y}{\|x\| \|y\|}$

Funded by the European Union

Learning word embeddings

- **Count-based methods: Matrix factorisation**

  Minimise the difference between the co-occurrence matrix and an approximate reconstruction of it from word embeddings.

- **Prediction-based methods: Neural networks**

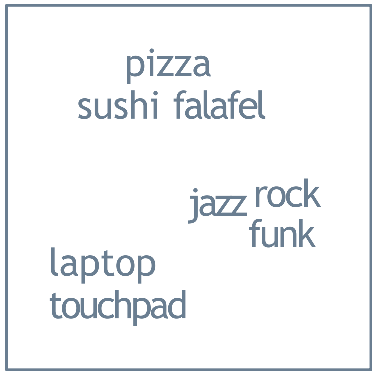
  Maximise the likelihood of a corpus under a probability model that is conditioned on the word embeddings.

Funded by the European Union

Two different perspectives on word embeddings

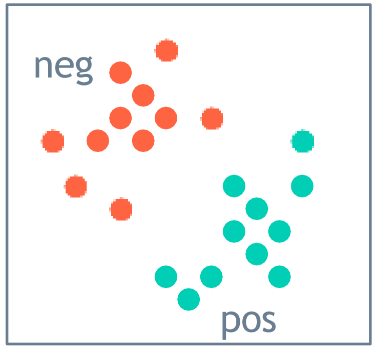
- **Count-based approach**

  similar embeddings ⇒ the corresponding words have similar distributions



- **Prediction-based approach**

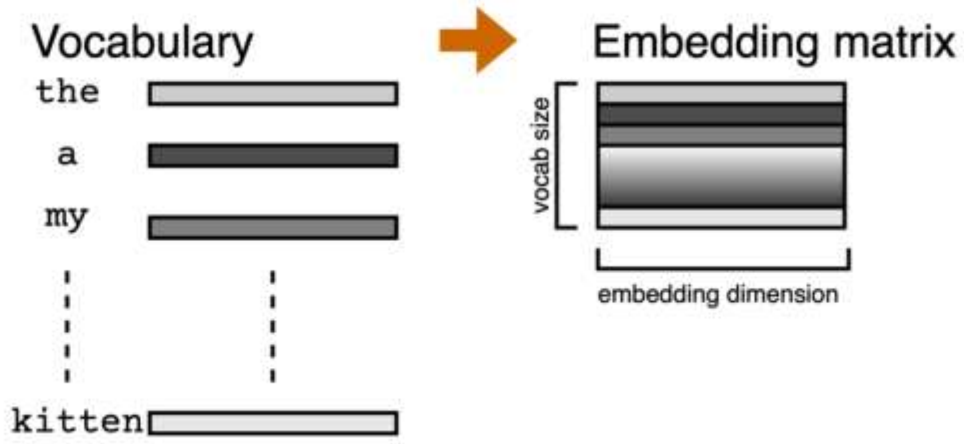
  similar embeddings ⇒ the corresponding words behave similarly in learning tasks



Turning Discrete Tokens into Continuous Vectors

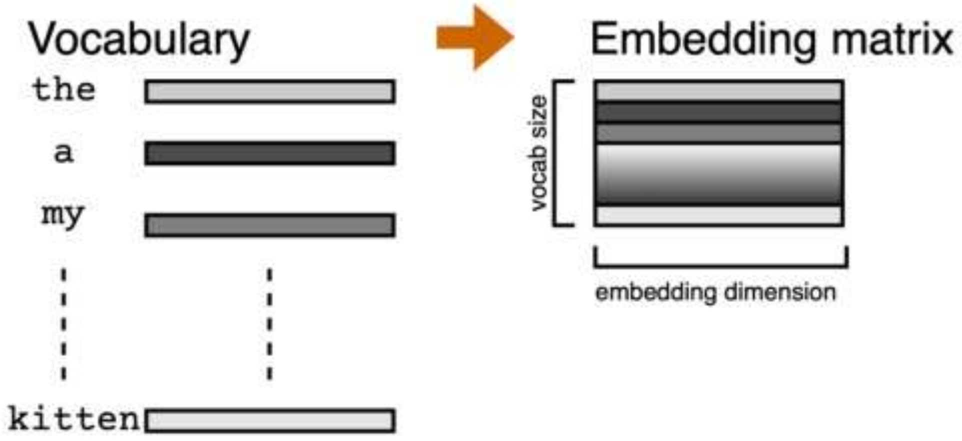
Neural networks cannot operate on discrete tokens.

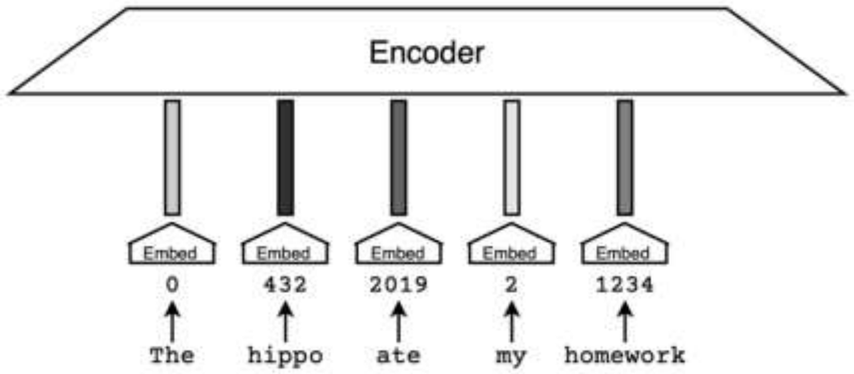
Instead, we build an **embedding matrix** which associates each token in the vocabulary with a vector embedding.



Encoder Inputs and Outputs

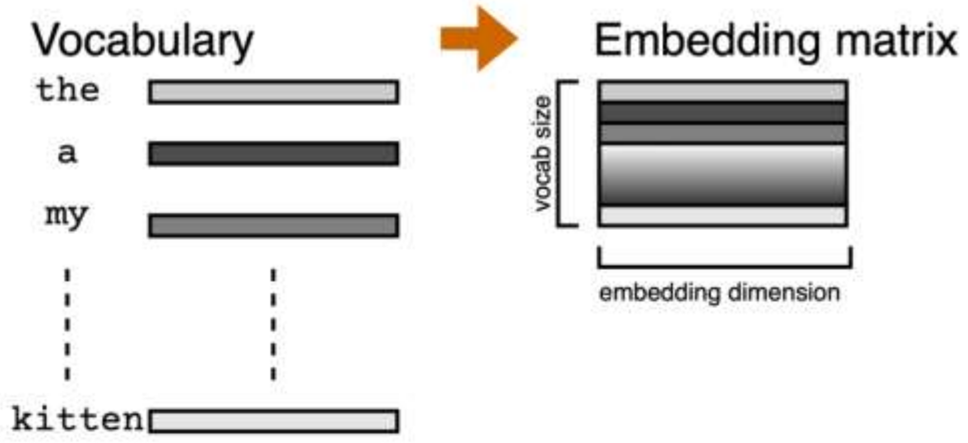
The encoder takes as input the vector representations of each token in the input sequence.

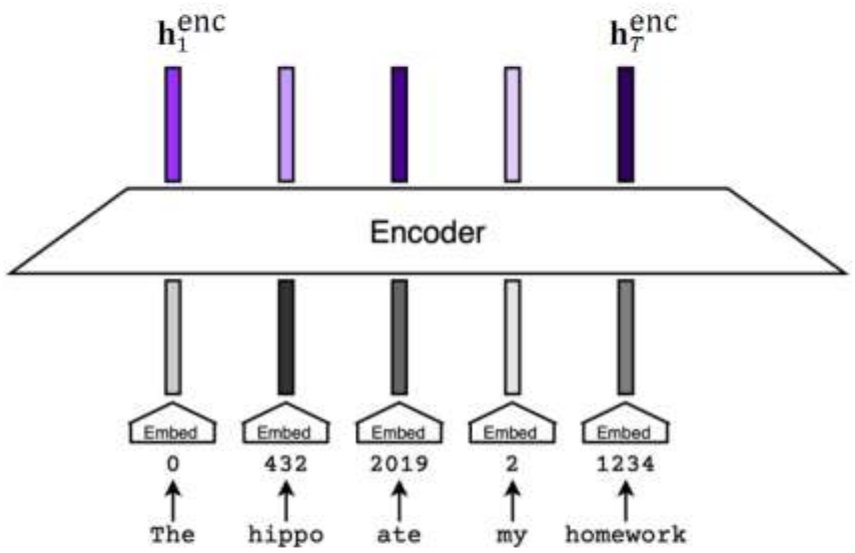




Encoder Inputs and Outputs

The encoder outputs a sequence of embeddings called **hidden states**.

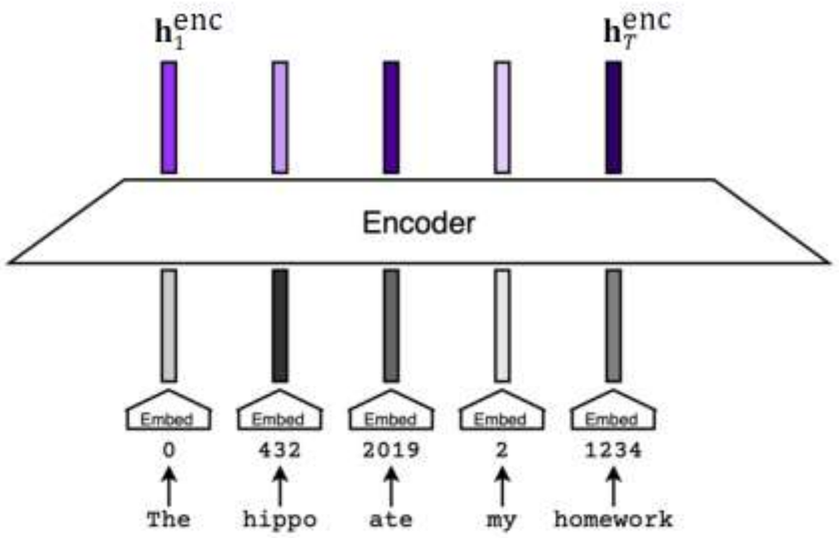


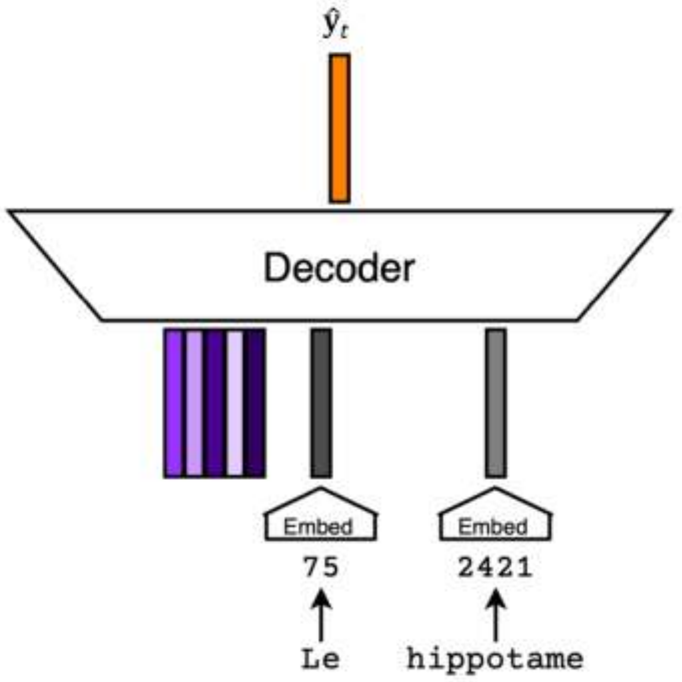


Decoder Inputs and Outputs

The decoder takes as input the hidden states from the encoder as well as the embeddings for the tokens seen so far in the target sequence.

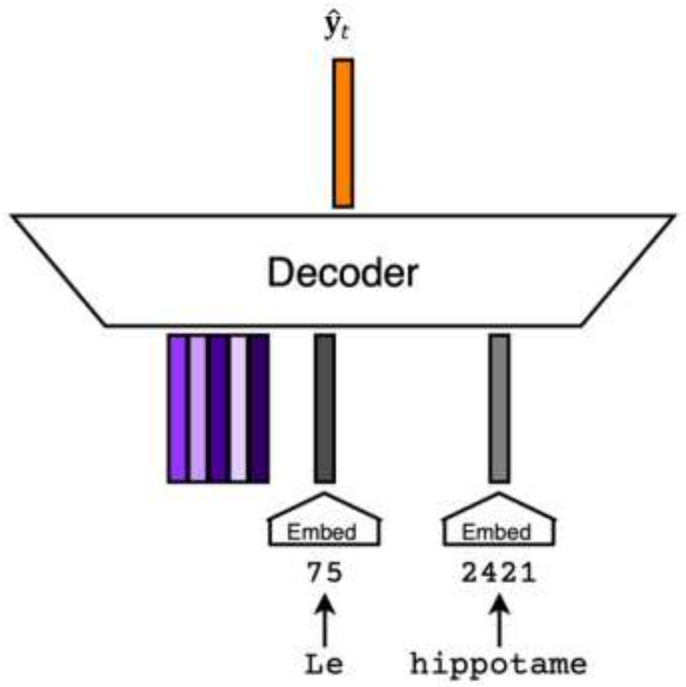
It outputs an embedding $\hat{\mathbf{y}}_t$.





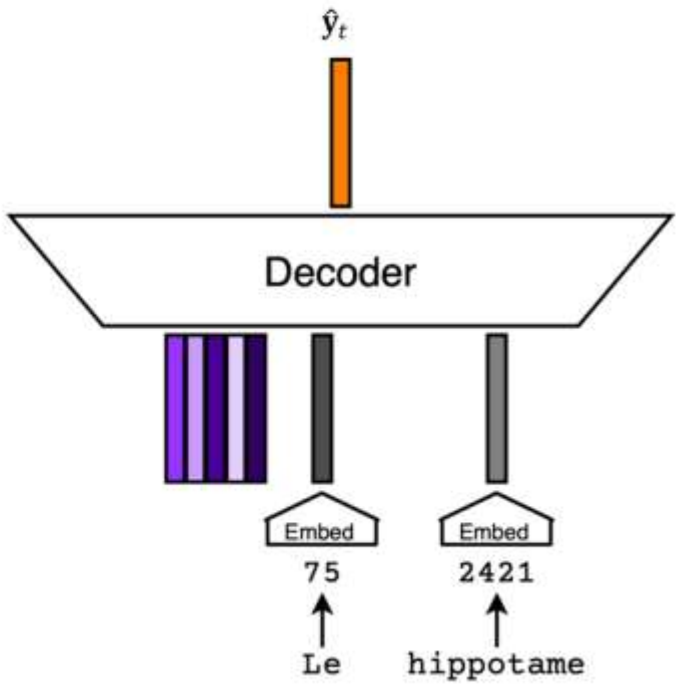
Decoder Inputs and Outputs

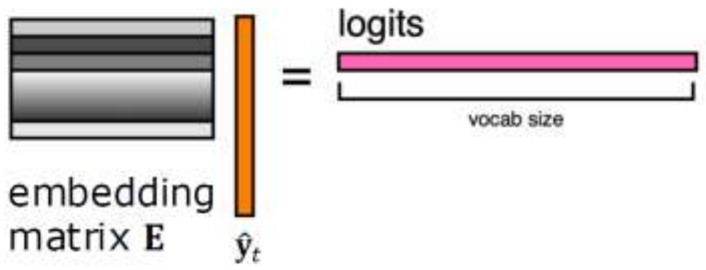
Ideally, $\hat{\mathbf{y}}_t$ would be as close as possible to the embedding of the true next token.



Decoder Inputs and Outputs

We multiply the predicted embedding $\hat{\mathbf{y}}_t$ by our vocabulary embedding matrix to get a score for each vocabulary word. These scores are referred to as **logits**.

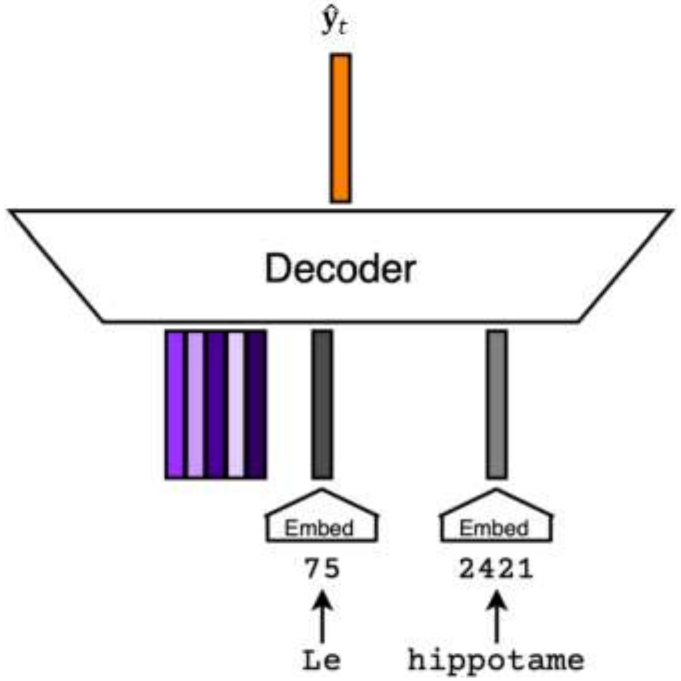


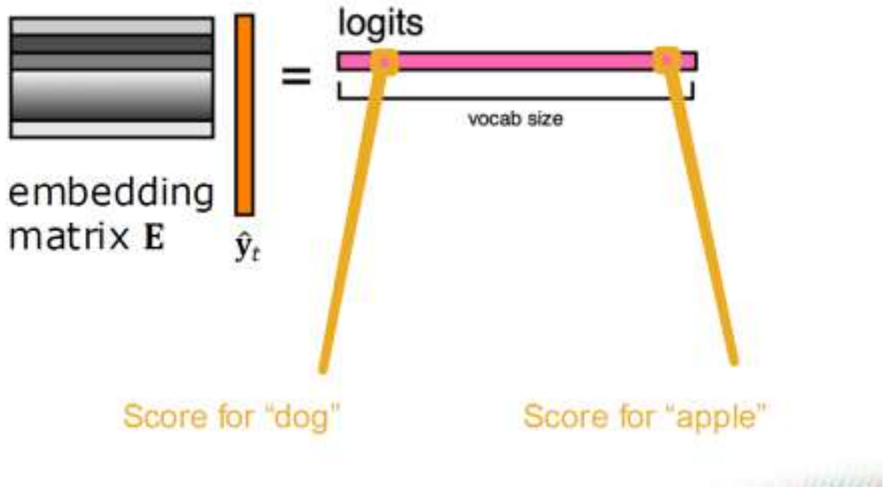


Funded by the European Union

Decoder Inputs and Outputs

We multiply the predicted embedding $\hat{\mathbf{y}}_t$ by our vocabulary embedding matrix to get a score for each vocabulary word. These scores are referred to as **logits**.





Decoder Inputs and Outputs

We multiply the predicted embedding $\hat{\mathbf{y}}_t$ by our vocabulary embedding matrix to get a score for each vocabulary word. These scores are referred to as **logits**.

The **softmax function** is used to turn the logits into probabilities.

$$P(Y_t = i | \mathbf{x}_{1:T}, \mathbf{y}_{1:t-1}) = \frac{\exp(\mathbf{E}\hat{\mathbf{y}}_t[i])}{\sum_j \exp(\mathbf{E}\hat{\mathbf{y}}_t[j])}$$

Example: Suppose we are trying to predict the 5th word in the sequence "the dog chased the". We want to know the probability the next word is "cat".

$$P(Y_5 = \text{“}cat\text{”} | \text{“}the\ dog\ chase\ the\text{”}) = \frac{\exp(\text{score in logits for “}cat\text{”})}{\text{normalization term}} = 0.321$$

Funded by the European Union

Evaluation of word embeddings

Eisenstein § 14.6

- visualisation of the embedding space

  Requires dimensionality reduction (PCA, t-SNE, UMAP)

- computing relative similarities

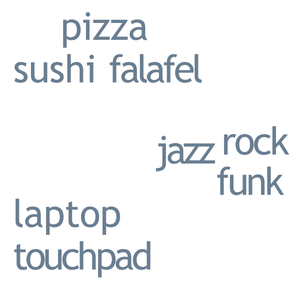
  cosine similarity, Euclidean distance

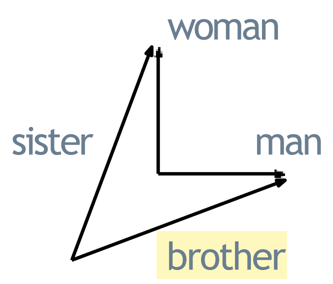
- similarity benchmarks

  Example: odd one out – *breakfast lunch dinner* *surgery*

- analogy benchmarks

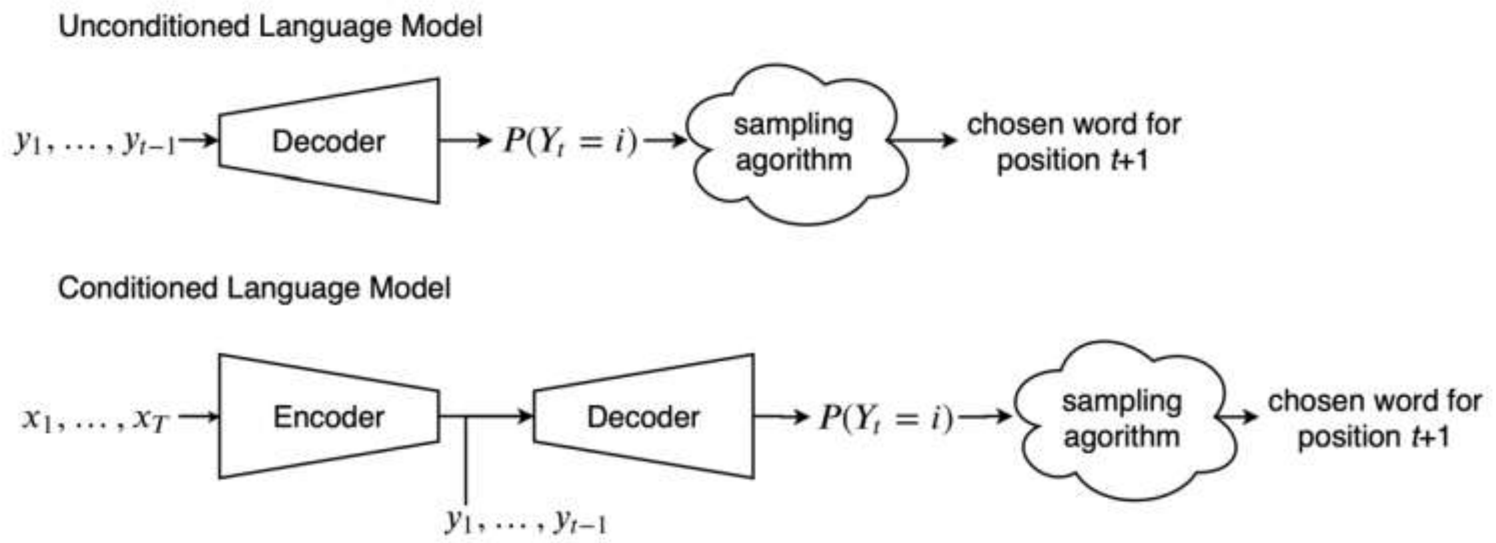
  Example: *woman* is to *man* as *sister* is to *?*





How Do we Do the Generation?

To do generation, we need a **sampling algorithm** that selects a word given the predicted probability distribution $P(Y_t = i|y_{1:t-1})$.

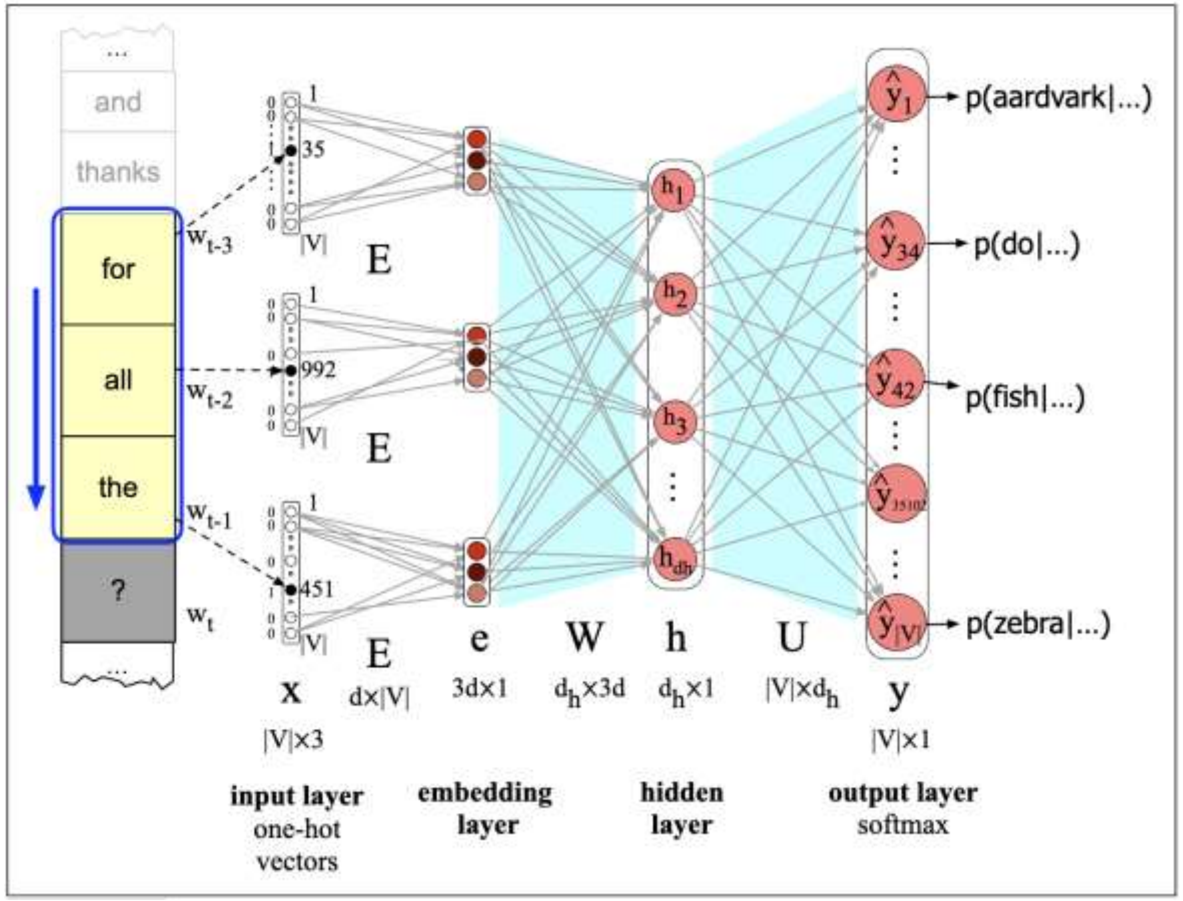


Feedforward Neural Network for Language Modeling

TrustLLM

Funded by the European Union

Architecture and Illustration of Forward Pass

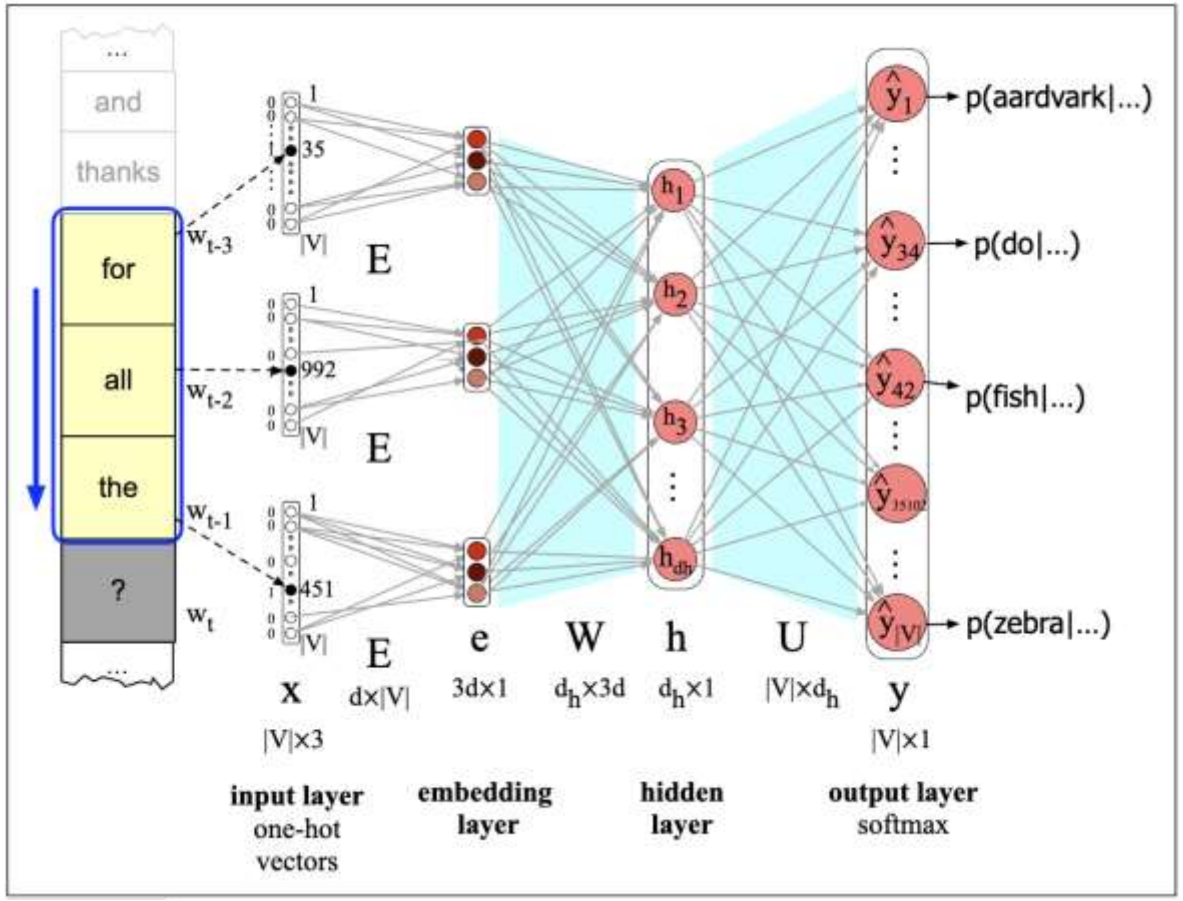


Sketch of feedforward neural language model with 3-word input context.

Picture Credit: Chapter 7 of Jurafsky/Martin's book "Speech and Language Processing" (3rd ed).

Y. Benjio et al. [A Neural Probabilistic Language Model](). JMLR 2003.

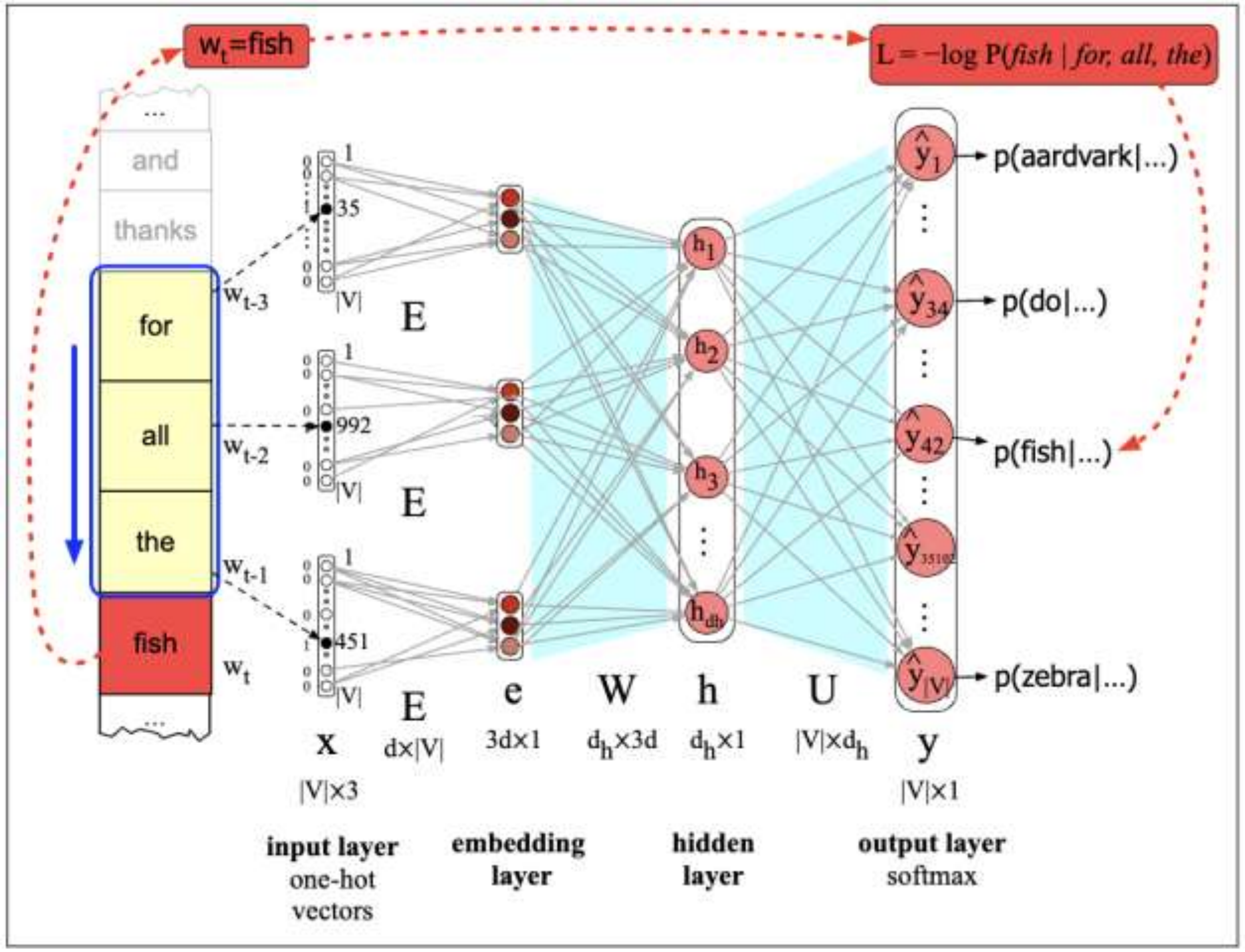
Architecture and Illustration of Forward Pass



At each timestep **t**, the network:

1. computes a **d**-dimensional embedding for each context word (by multiplying a one-hot vector by embedding matrix **E**)
2. concatenates the 3 resulting embeddings to produce embedding layer **e**
3. **e** is multiplied by a weight matrix **W** and then an activation function is applied element-wise to produce hidden layer **h**
4. **h** is then multiplied by another weight matrix **U**
5. finally, a softmax output layer predicts at each node **i** the probability that the next word $\mathbf{w_t}$ will be vocabulary word $\mathbf{V_i}$

Loss Function: Negative Log Likelihood (NLL)

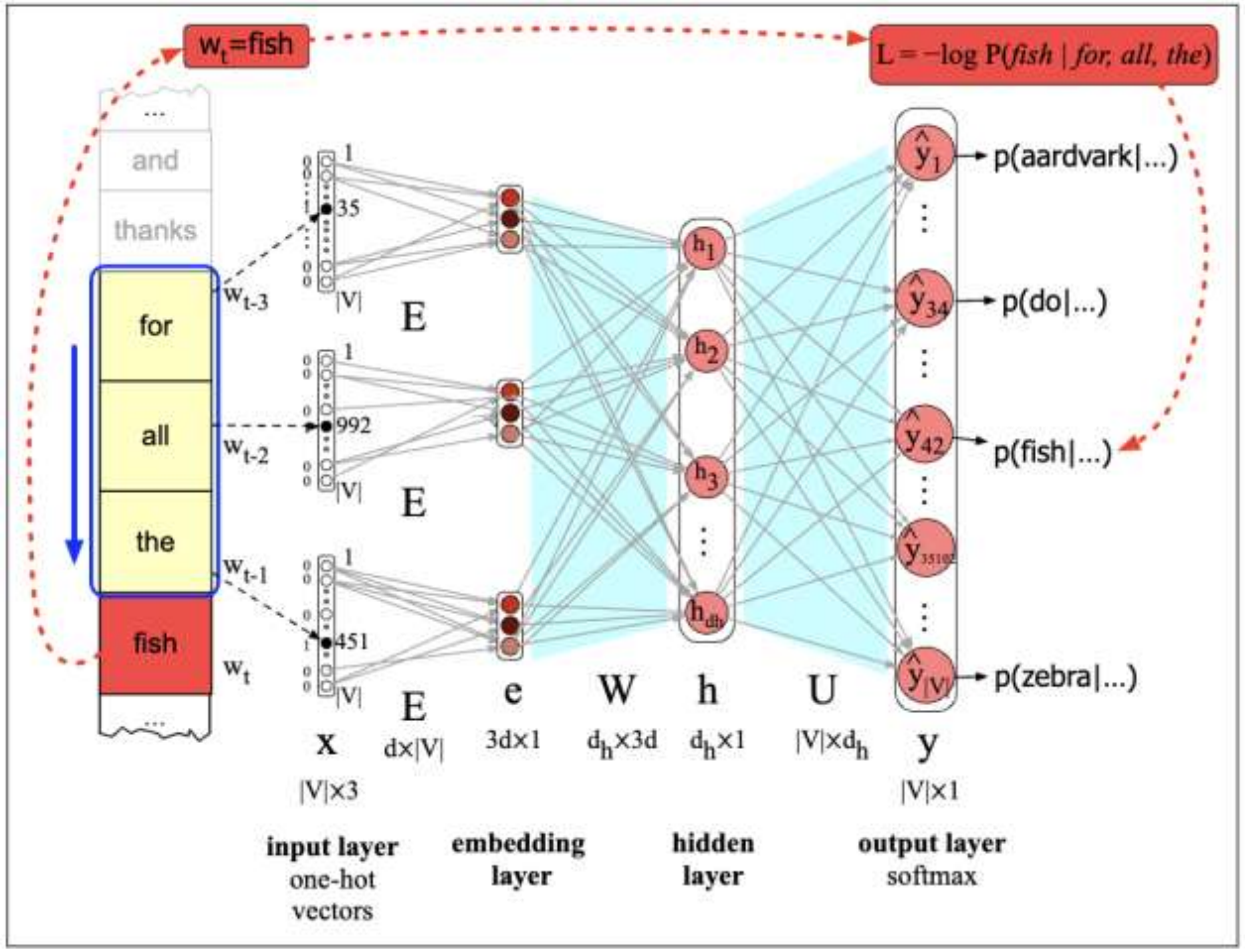


The parameter update for stochastic gradient descent for cross-entropy loss L from step s to s+1 is:

$$\theta^{s+1} = \theta^{s} - \eta \frac{\partial\left[-\log p(w_t|w_{t-1},\ldots,w_{t-n+1})\right]}{\partial \theta}$$

This gradient can be computed in any standard neural network framework (e.g. Pytorch) which will then backpropagate through θ = **E**, **W**, **U**.

Forward Pass and Loss Function in Equations



$$\mathbf{e} = [\mathbf{E}\,\mathbf{x}_{t-3}; \mathbf{E}\,\mathbf{x}_{t-2}; \mathbf{E}\,\mathbf{x}_{t-1}]$$

$$\mathbf{h} = \sigma\,(\mathbf{W}\,\mathbf{e})$$

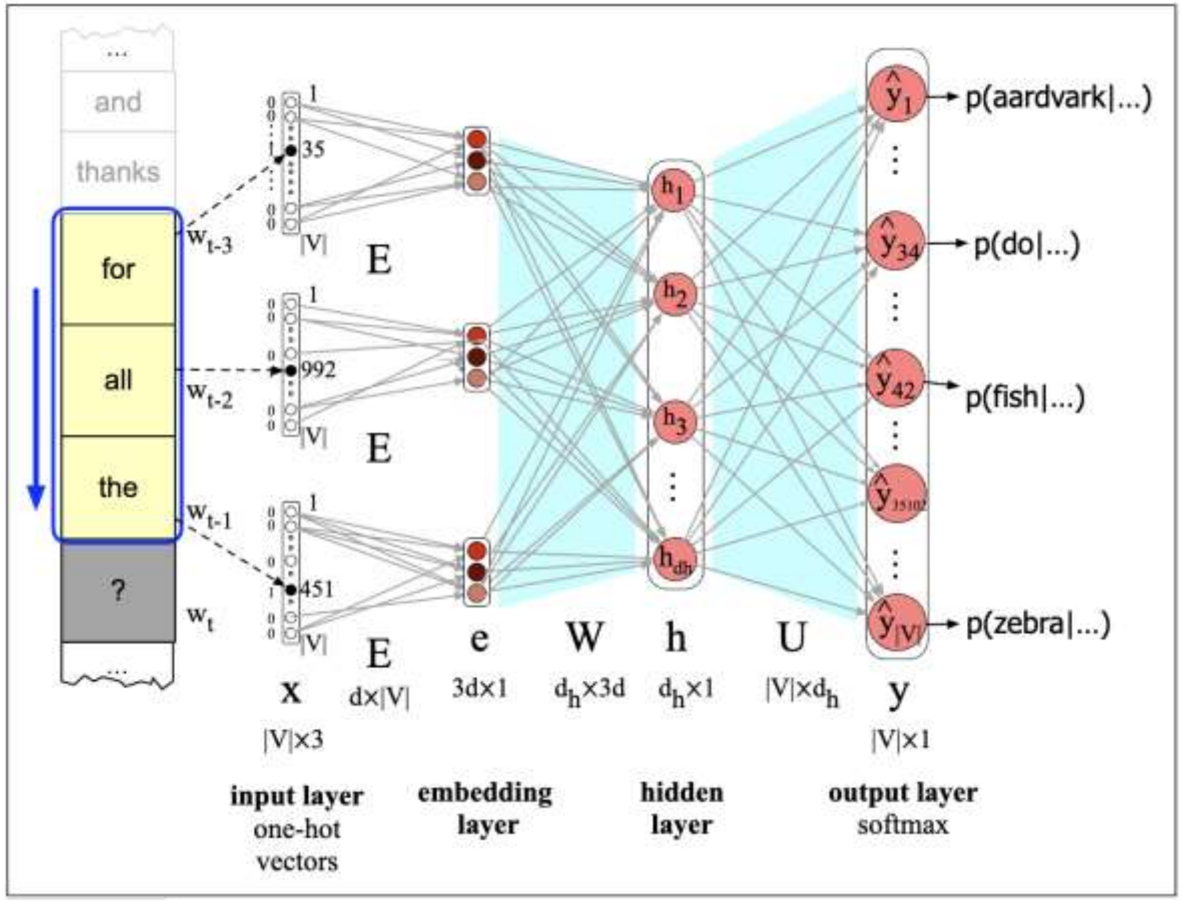
$$\mathbf{z} = \mathbf{U}\,\mathbf{h}$$

$$\mathbf{y} = \text{softmax}(\mathbf{z})$$

$$\mathbf{L} = -\,\mathbf{y}\,(\log \mathbf{y})$$

where **y** is one-hot vector representing ground truth.

Pros/Cons of Feedforward Neural Network as Language Model



Improvements over n-gram LM:

- No sparsity problem.
- No need to store all observed n-grams.

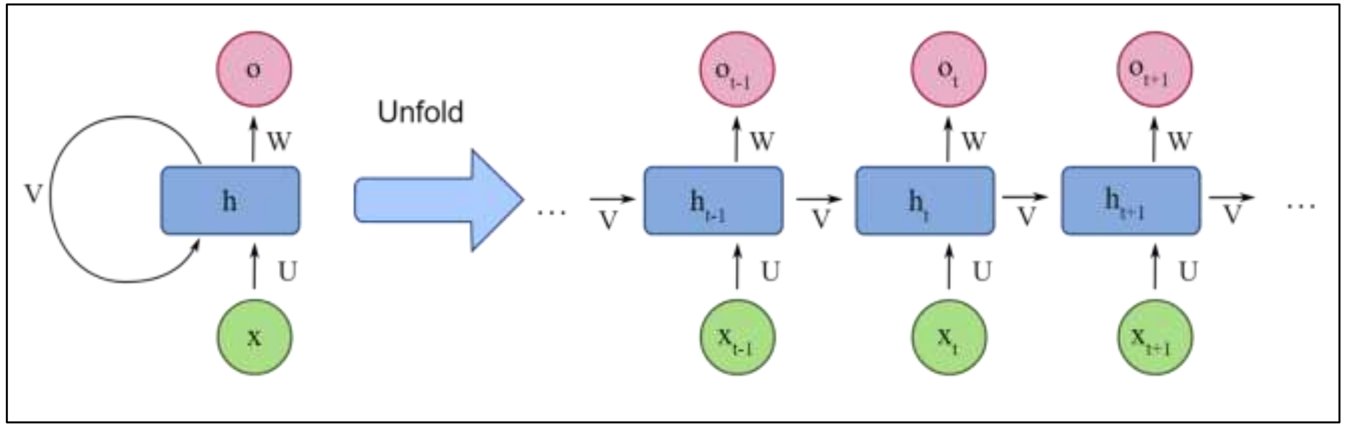
Remaining problems:

- Fixed window is too small.
- Model size (**W**) increases for longer input context.
- Window can never be large enough!
- Each $\mathbf{x_i}$ is multiplied by completely different weights in **W**, so no symmetry in how the inputs are processed.

Funded by the European Union

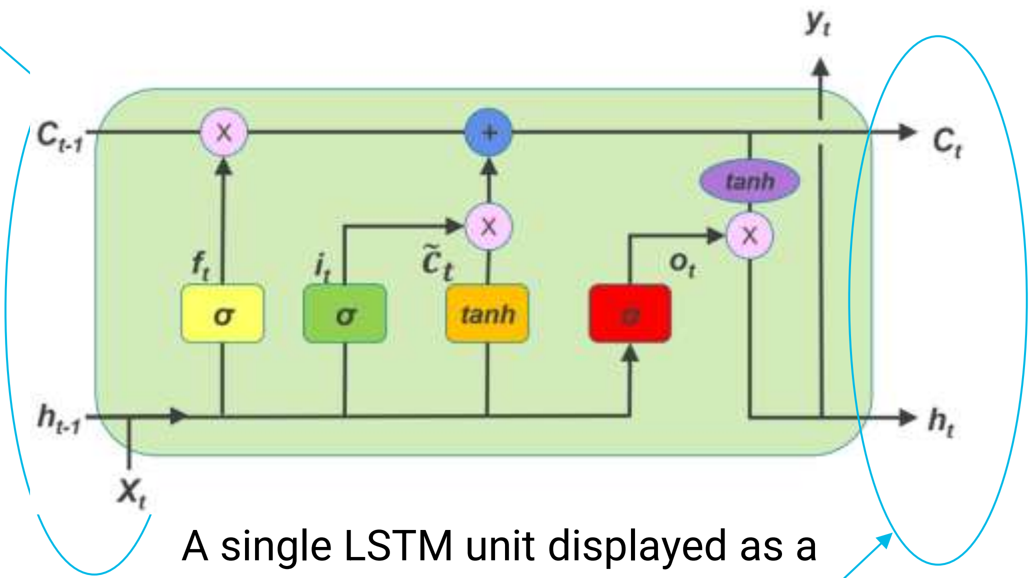
Seq2Seq Models

- Recurrent Neural Networks (RNNs)
- Long Short-Term Memory Networks (LSTMs)
- Capture dependencies between input tokens
- Gates control the flow of information



A simple RNN shown unrolled in time. Network layers are recalculated for each time step, while weights U, V and W are shared across all time steps.

The inputs to each unit consists of the current input x_t, previous hidden state h_{t-1}, and previous context c_{t-1}



A single LSTM unit displayed as a computation graph.

The outputs are a new hidden state h_t and an updated context c_t.

TrustLLM
Funded by the European Union

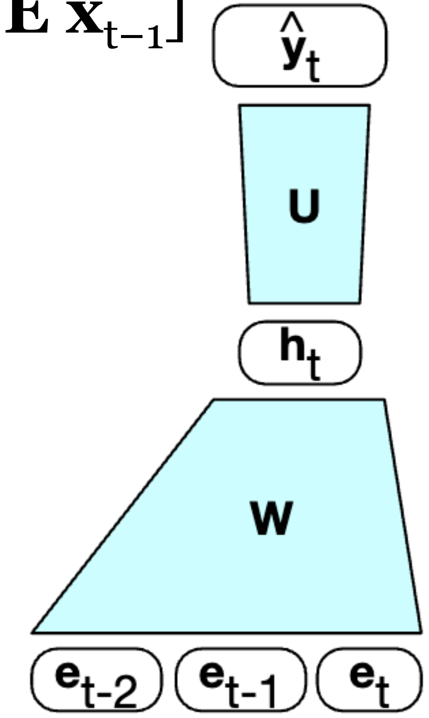
Feedforward vs. Recurrent Neural Network as Language Model

$\mathbf{e} = [\mathbf{E}\,\mathbf{x}_{t-3}; \mathbf{E}\,\mathbf{x}_{t-2}; \mathbf{E}\,\mathbf{x}_{t-1}]$

$\mathbf{h} = \sigma\,(\mathbf{W}\,\mathbf{e})$

$\mathbf{z} = \mathbf{U}\,\mathbf{h}$

$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$

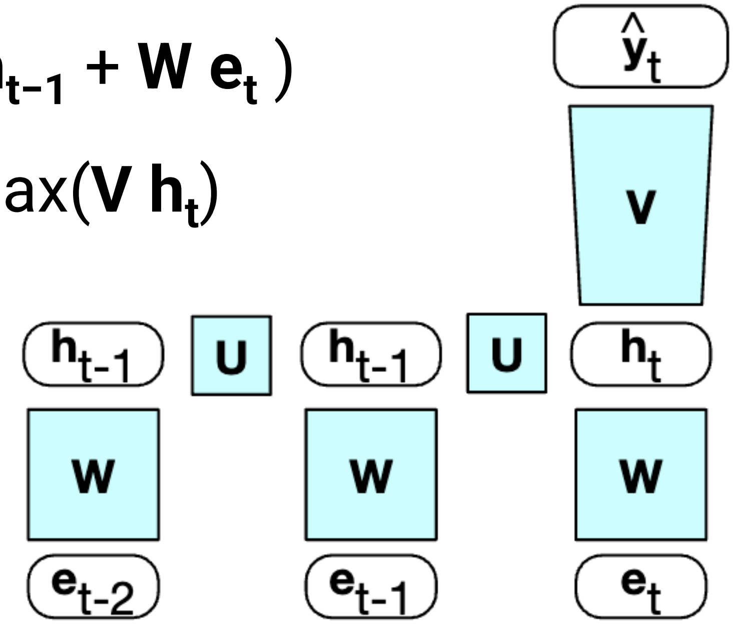


Feedforward Neural Network

$\mathbf{e}_t = \mathbf{E}\,\mathbf{x}_t$

$\mathbf{h}_t = \sigma\,(\mathbf{U}\,\mathbf{h}_{t-1} + \mathbf{W}\,\mathbf{e}_t)$

$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{V}\,\mathbf{h}_t)$



Recurrent Neural Network

TrustLLM
Funded by the European Union

Pros/Cons of Recurrent Neural Network as Language Model

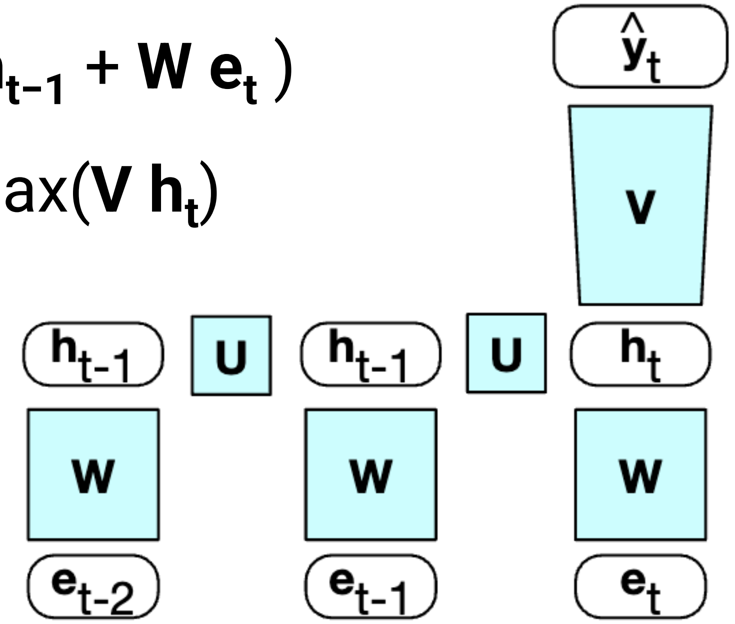
Pros:

- Can process any length input!
- Computation for step **t** can (in theory) use information from many steps back.
- Model size (**W**) doesn't increase for longer input context.
- Each $\mathbf{x}_i$ is multiplied by same weights in **W**, so there is symmetry in how inputs are processed.

$$\mathbf{e}_t = \mathbf{E}\,\mathbf{x}_t$$

$$\mathbf{h}_t = \sigma\,(\mathbf{U}\,\mathbf{h}_{t-1} + \mathbf{W}\,\mathbf{e}_t)$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{V}\,\mathbf{h}_t)$$



Cons:

- Recurrent computation is slow (cannot be parallelized).
- In practice, it is difficult to access information from many steps back. ←

Vanishing Gradient Problem!
RNN extensions such as LSTMs (or other varieties like GRUs) are commonly used.

TrustLLM

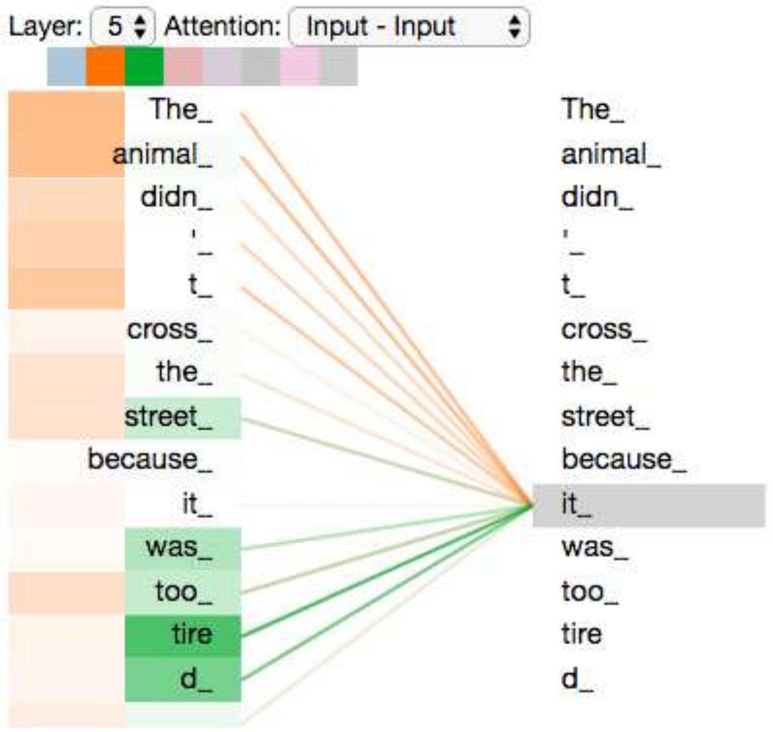
Funded by the European Union

Self-Attention and Transformers

- Allows to "focus attention" on particular aspects of the input while generating the output.
- Done by using a set of parameters, called "weights," that determine how much attention should be paid to each input token at each time step.
- These weights are computed using a combination of the input and the current hidden state of the model.

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

A. Vaswani et al. Attention Is All You Need. NeurIPS 2017.



In encoding the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired". The model's representation of the word "it" thus bakes in some of the representation of both "animal" and "tired". https://jalammar.github.io/illustrated-transformer/

Funded by the European Union

Transformers

TrustLLM

Funded by the European Union

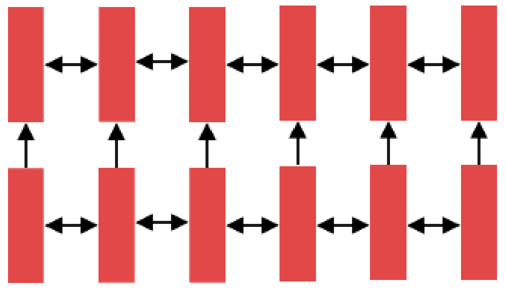
From Recurrence to Attention

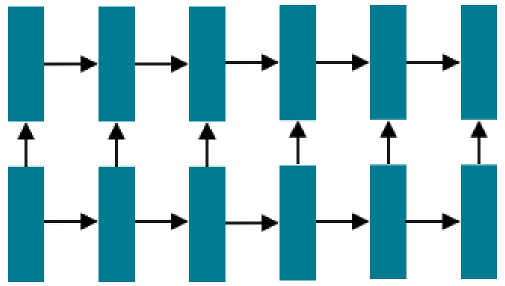
TrustLLM

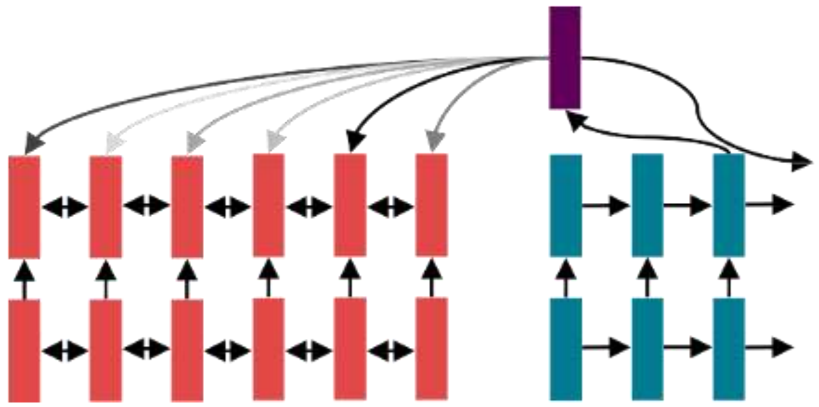
Funded by the European Union

The Story So Far: RNNs for (Most) NLP

- Circa 2016, the de facto strategy in NLP is to encode sentences with a bidirectional LSTM (e.g., the source sentence in a translation).
- Define your output (parse, sentence, summary) as a sequence, and use an LSTM to generate it.
- Use attention to allow flexible access to memory.



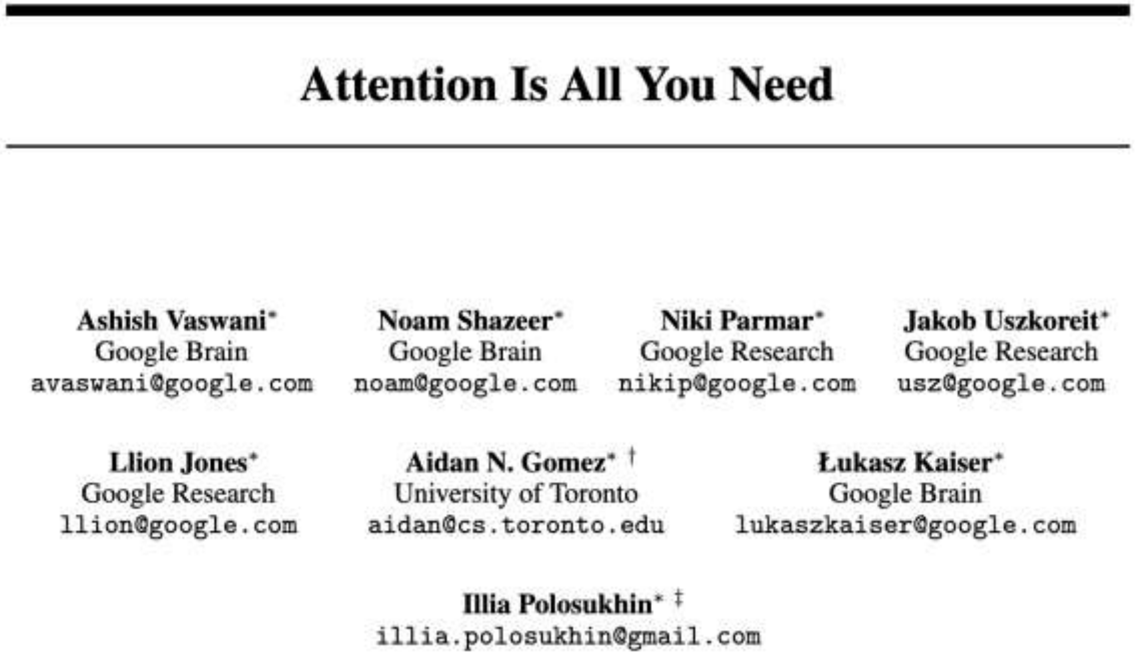




Attention Is All You Need

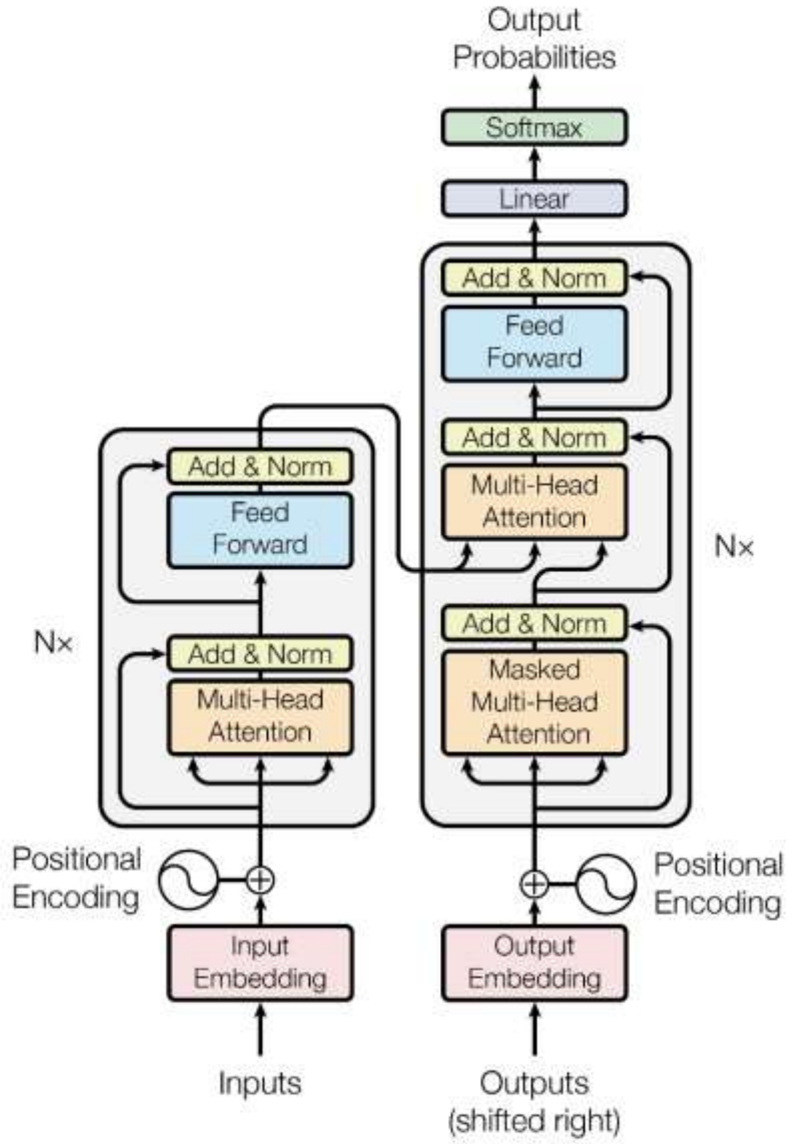
We saw that attention dramatically improves the performance of RNNs.

Transformers take this idea one step further!



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



Motivation for Transformer

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

Funded by the European Union

1. Transformer Motivation: Computational Complexity Per Layer

When sequence length (n) << representation dimension (d), the complexity per layer is lower for a Transformer model compared to RNN models.

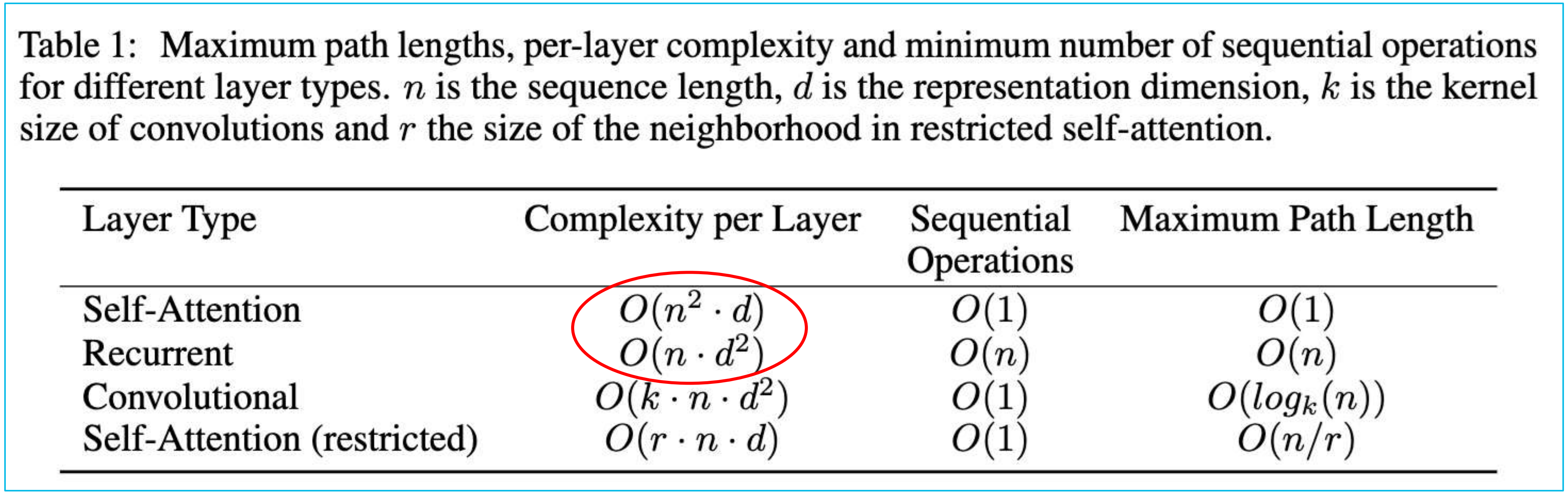


Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

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)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

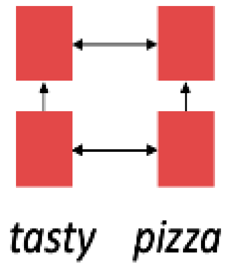
Table 1 of paper by A. Vaswani et al. Attention Is All You Need. NeurIPS 2017.

Funded by the European Union

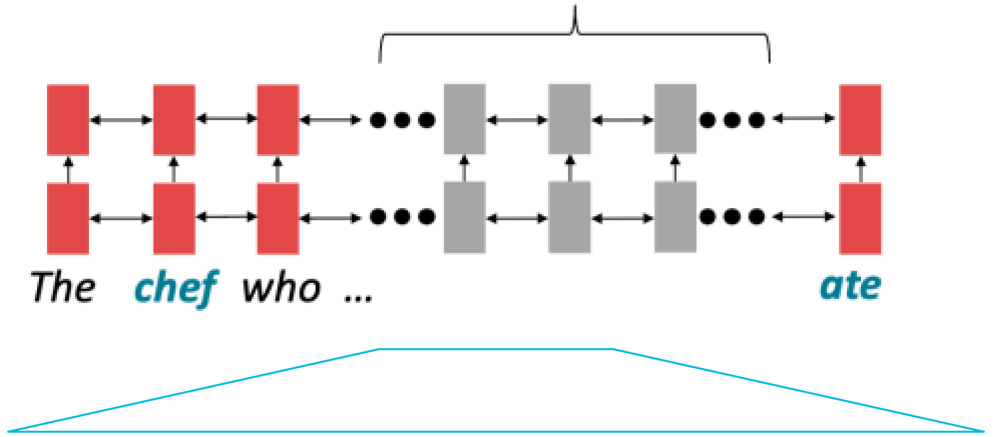
2. Transformer Motivation: Minimize Linear Interaction Distance

- RNN is unrolled "left-to-right".
- It encodes linear locality: a useful heuristic!
- **Problem:** RNN takes **O(sequence length)** steps for distant word pairs to interact.
  - Hard to learn long-distance dependencies due to gradient problems.
  - Linear order of words is "baked in"; we already know sequential structure doesn't tell the whole story...

Nearby words often affect each other's meanings



Info about ***chef*** has gone through O(sequence length) many layers!

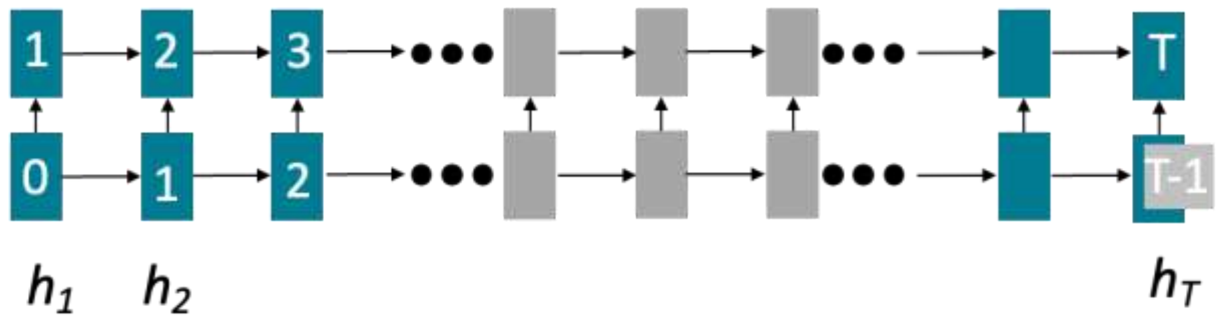


meticulously prepared the finest gourmet dishes, using only the freshest, most exquisite ingredients sourced from around the world, finally sat down at the end of a long day in the kitchen. Exhausted but satisfied, he carefully plated a small meal for himself, savoring each bite as he

3. Transformer Motivation: Maximize Parallelizability

Forward and backward passes have **O(sequence 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.



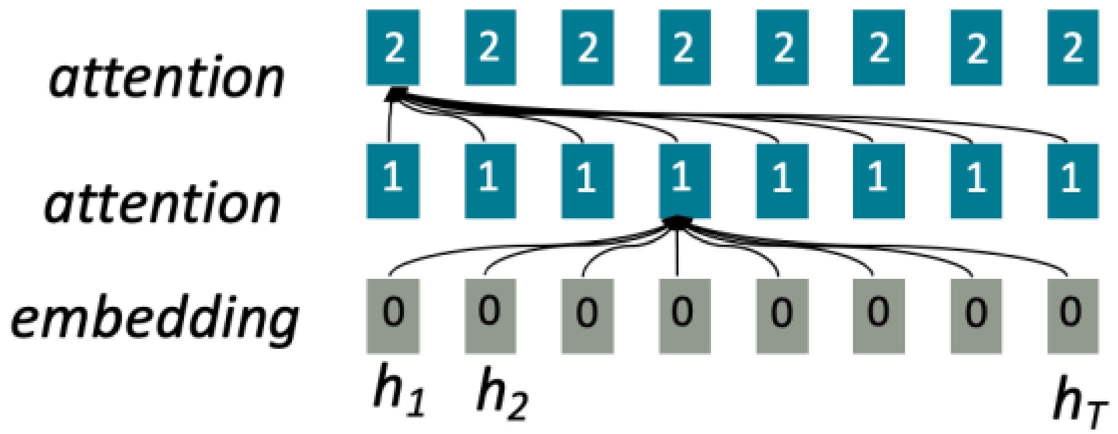
Numbers indicate min # of steps before a state can be computed

Funded by the European Union

High-Level Architecture: Transformer is all about (Self) Attention

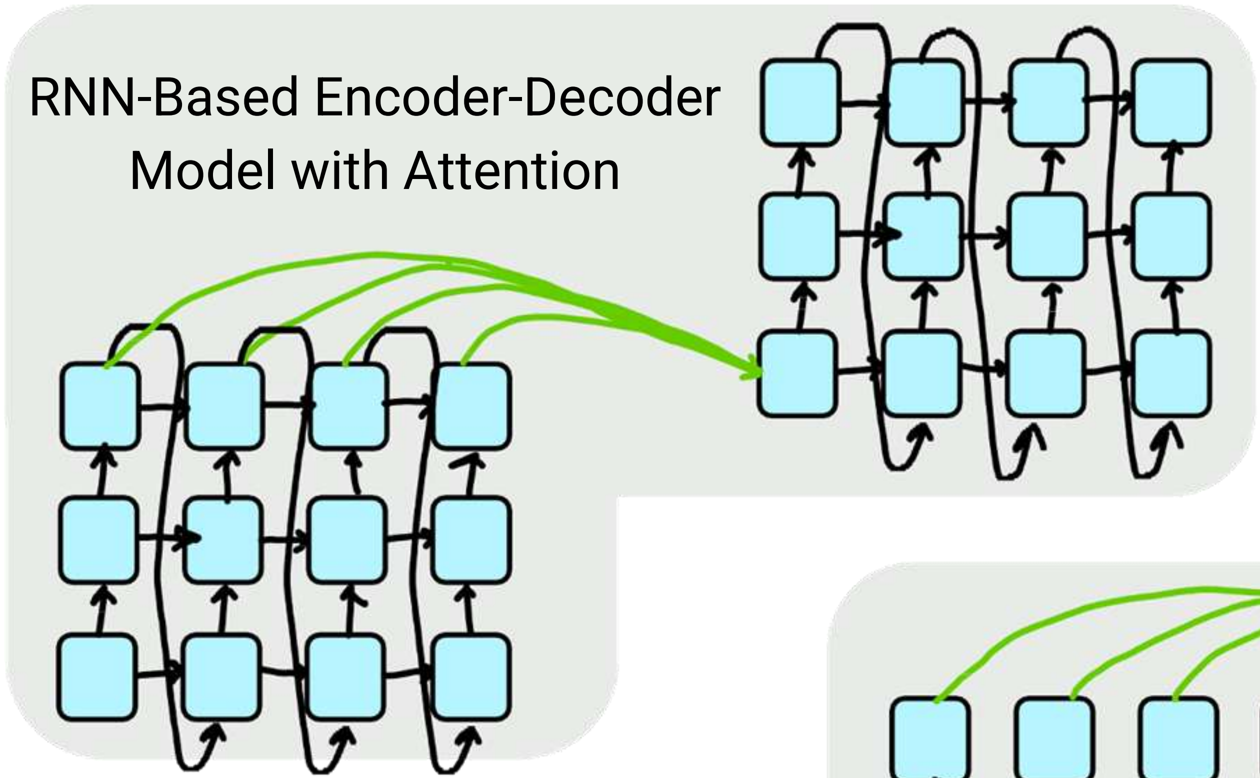
Earlier, we saw attention from the decoder to the encoder in a recurrent sequence-to-sequence model.

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



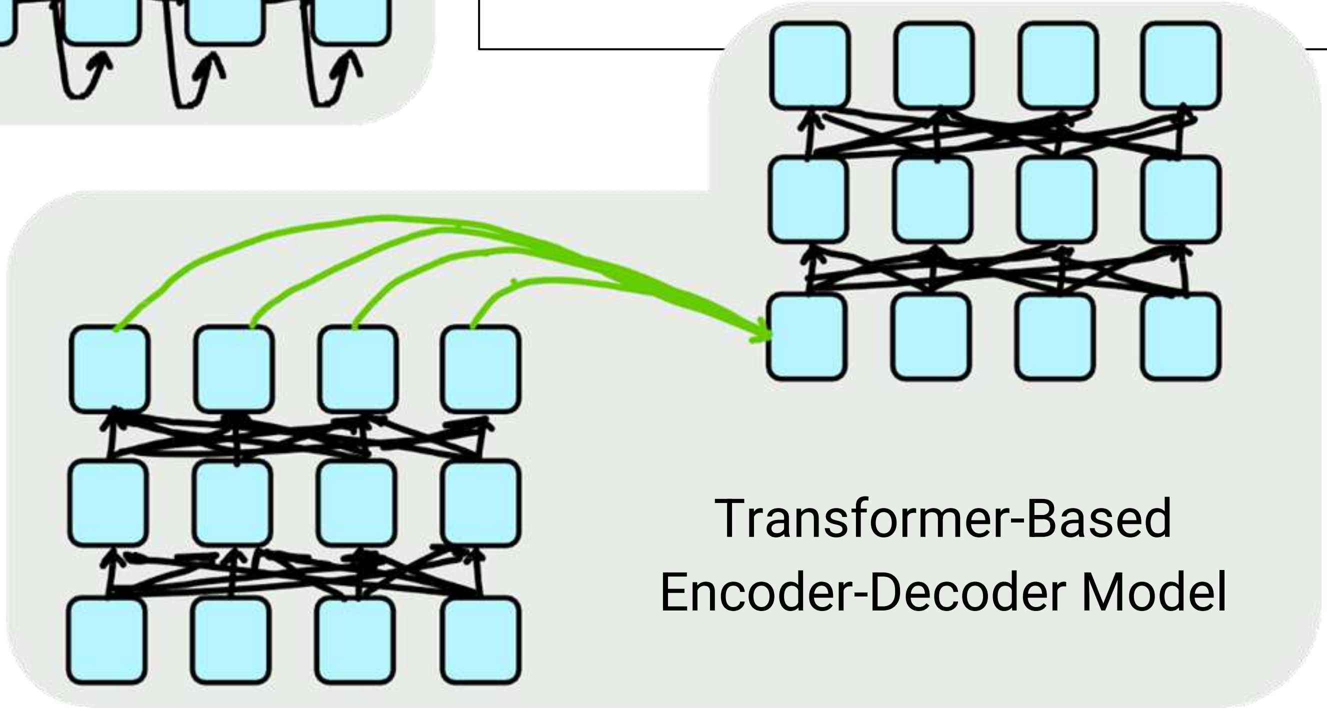
All words attend to all words in previous layer; most arrows here are omitted.

Computational Dependencies for Recurrence vs. Attention



Transformer Advantages:

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



The Transformer Block

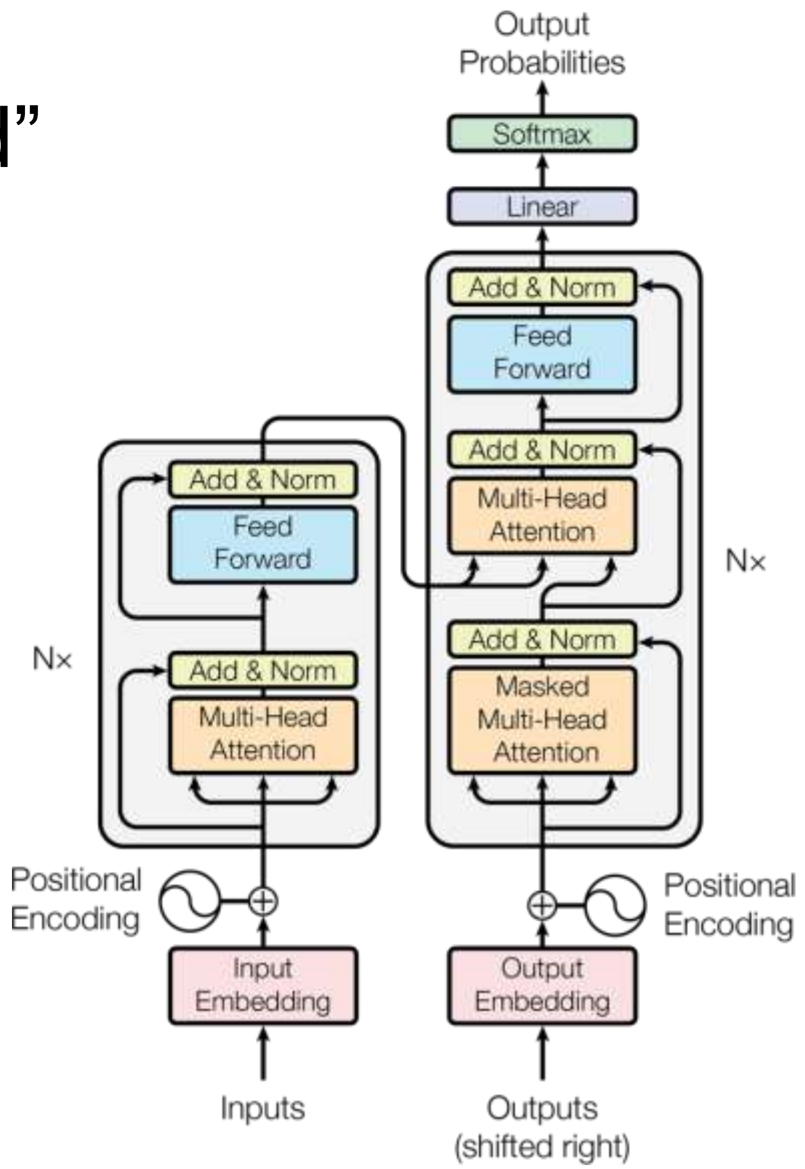
TrustLLM

Funded by the European Union

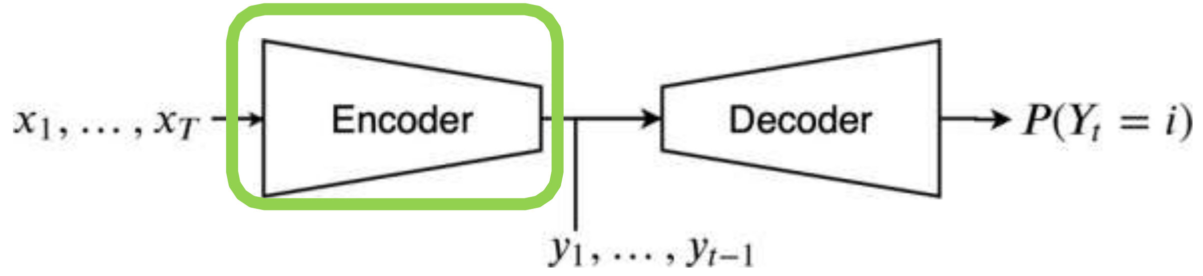
Why drop the recurrence and only use attention?

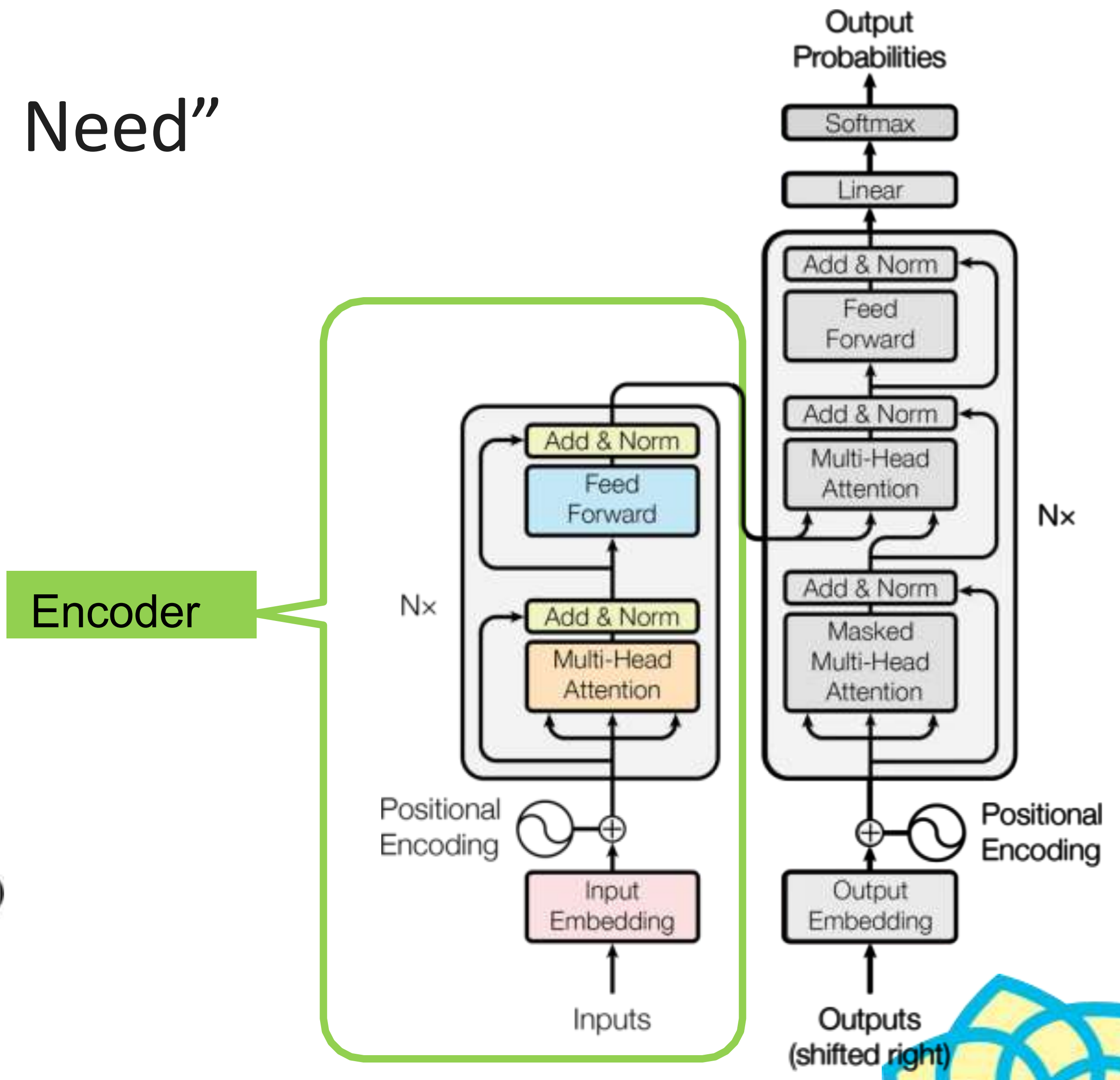
- Recurrent neural networks are slow to train. Computation cannot be parallelized.
    - The computation at position *t* is dependent on first doing the computation at position *t*-1.
- Recurrent neural networks do poorly with long contexts.
    - If two tokens are *K* positions apart, there are *K* opportunities for knowledge of the first token to be erased from the hidden state before a prediction is made at the position of the second token.
- Transformers solve both these problems.

Transformers: “Attention is All You Need”

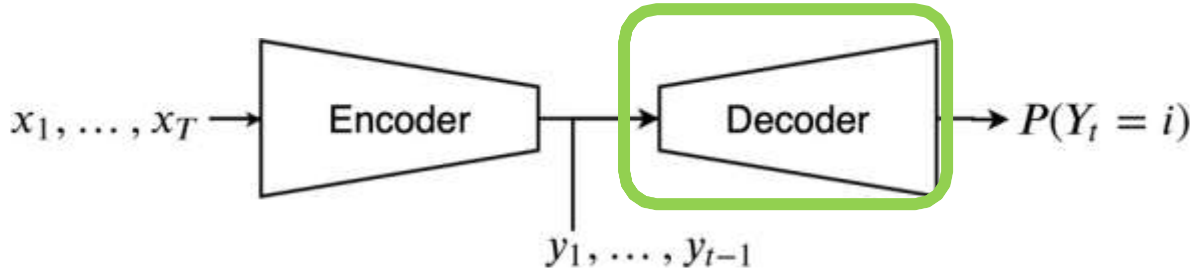


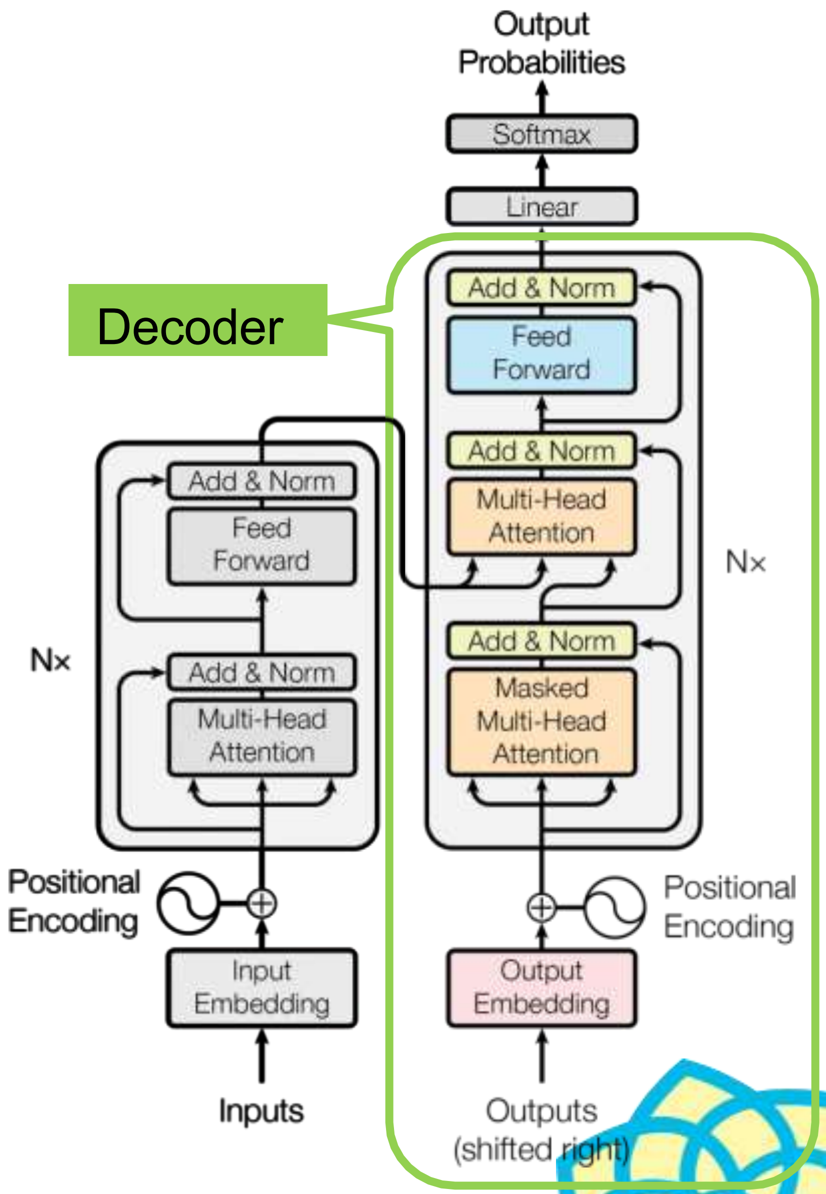
Transformers: “Attention is All You Need”



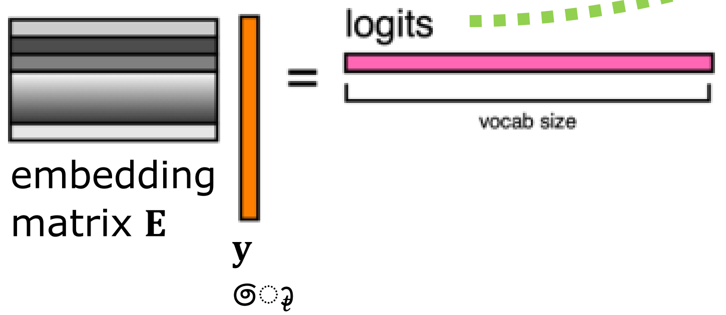


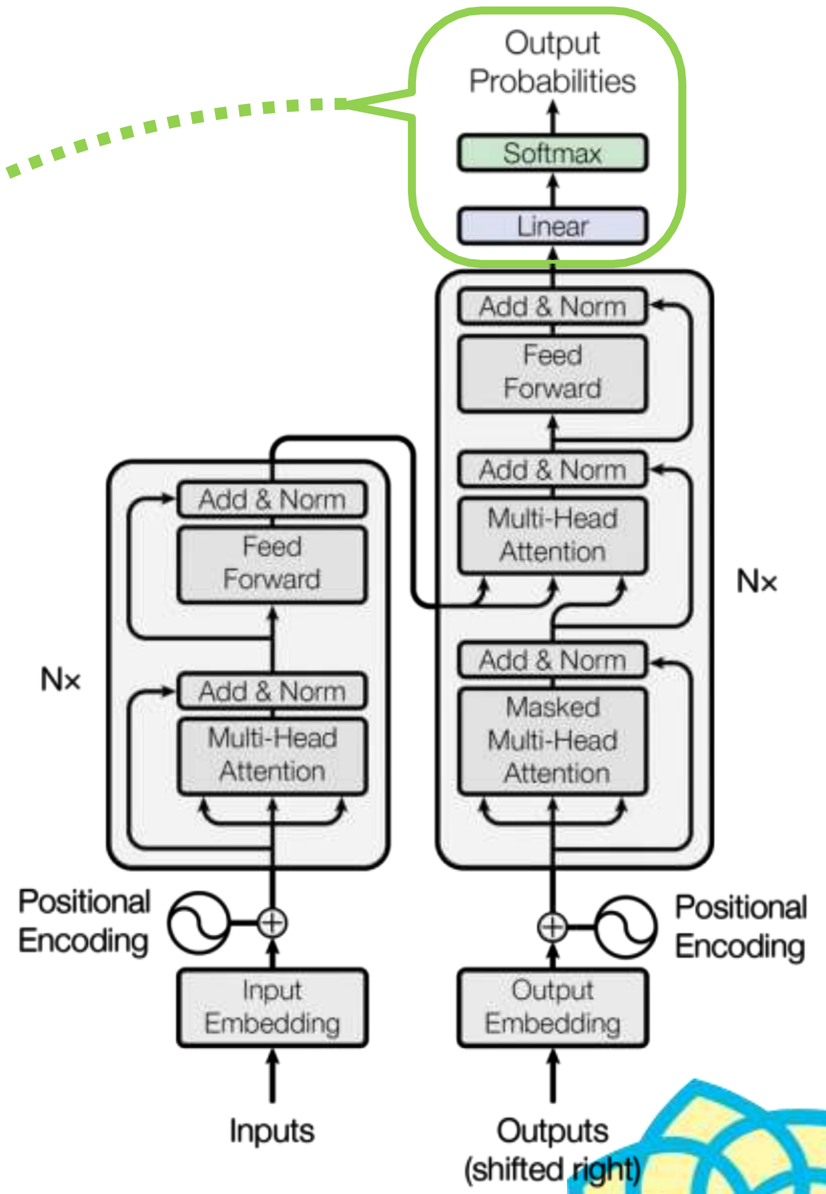
Transformers: “Attention is All You Need”





Transformers: “Attention is All You Need”





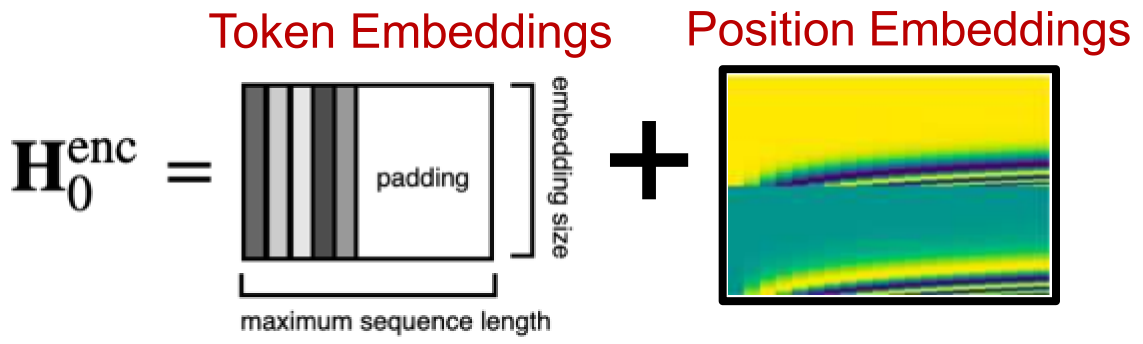
$$P(Y_t = i | \mathbf{x}_{1:T}, \mathbf{y}_{1:t-1}) = \frac{\exp(\mathbf{E}\mathbf{y}_t[i])}{\sum_j \exp(\mathbf{E}\mathbf{y}_t[j])}$$

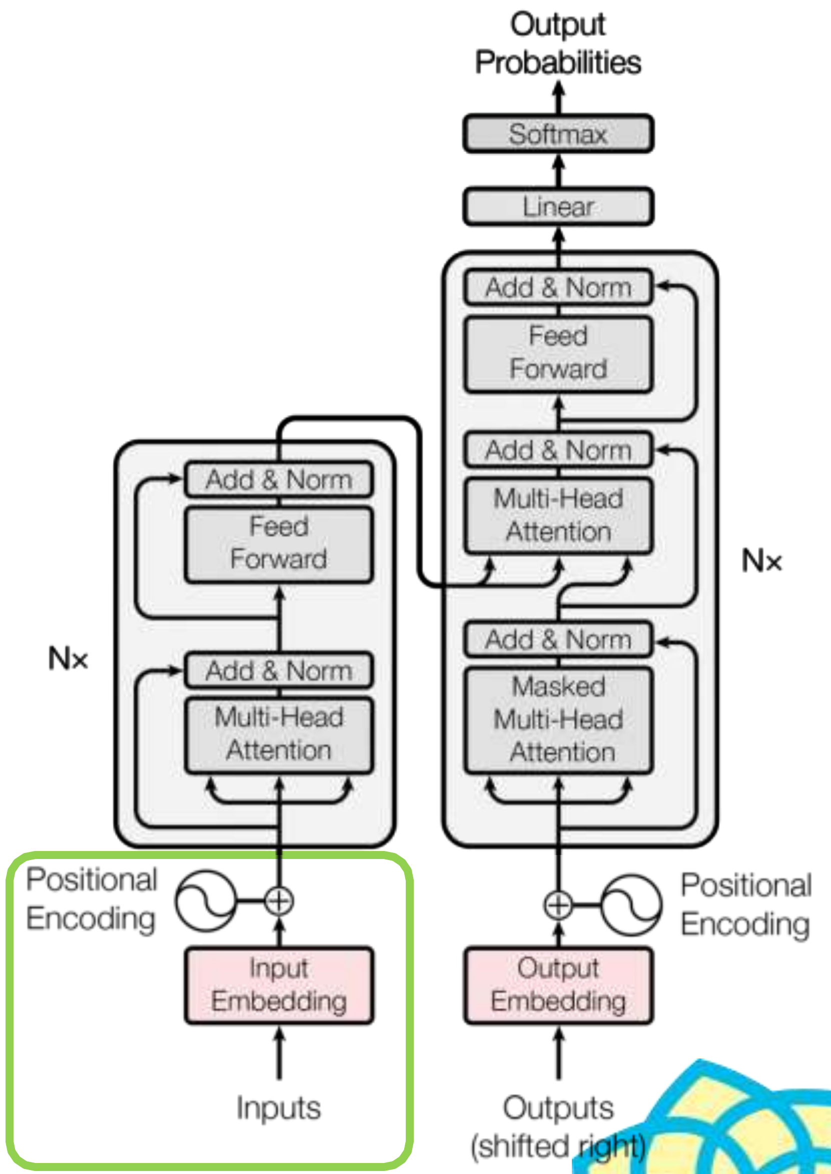
TrustLLM

Funded by the European Union

Transformers: “Attention is All You Need”

The input into the encoder looks like:





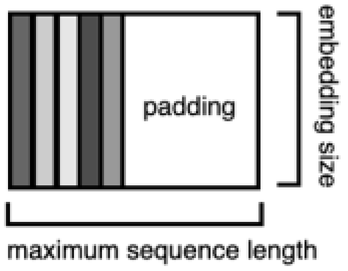
Funded by the European Union

Transformers: “Attention is All You Need”

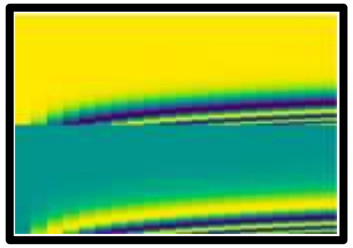
The input into the encoder looks like:

Token Embeddings Position Embeddings

$\mathbf{H}_0^{\text{enc}} =$



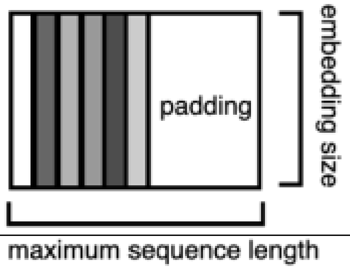
$+$



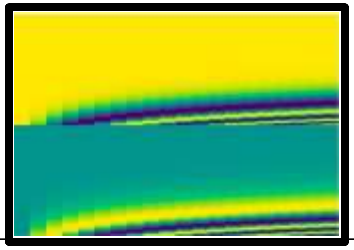
The input to the decoder looks like:

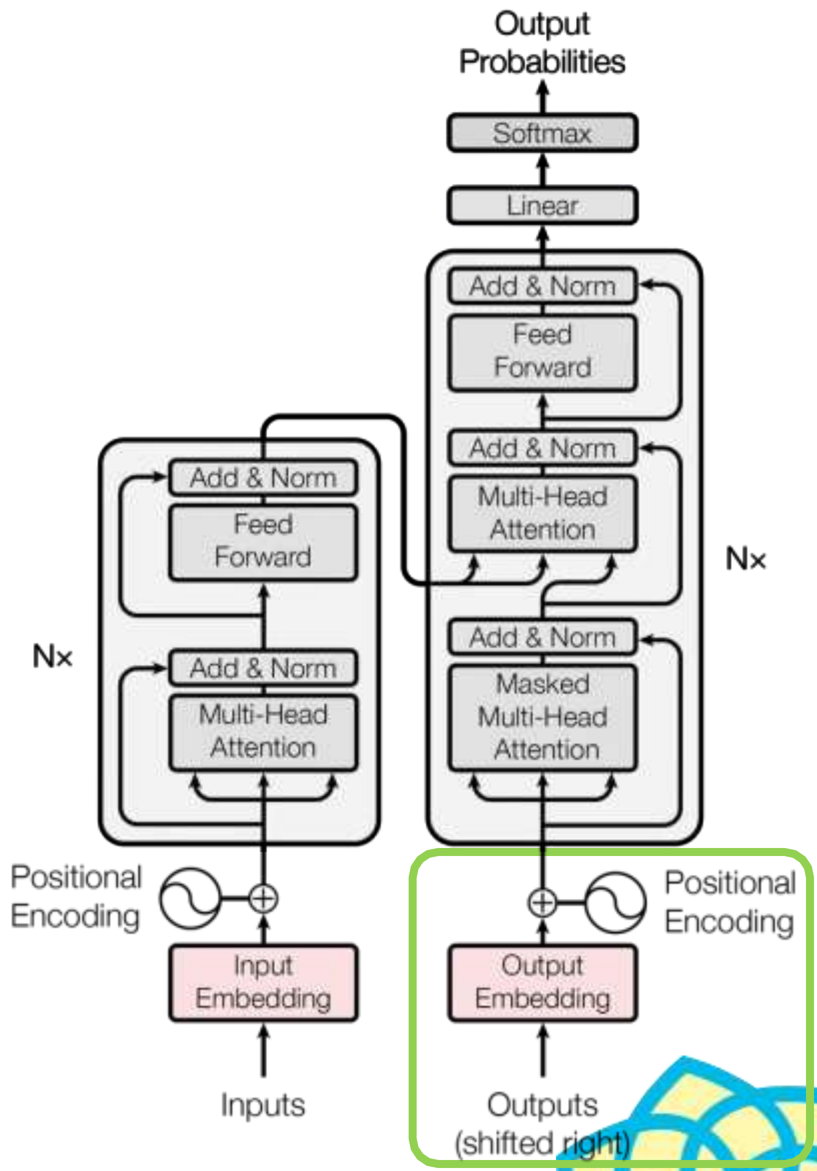
Shifted Token Embeddings Position Embeddings

$\mathbf{H}_0^{\text{dec}} =$



$+$



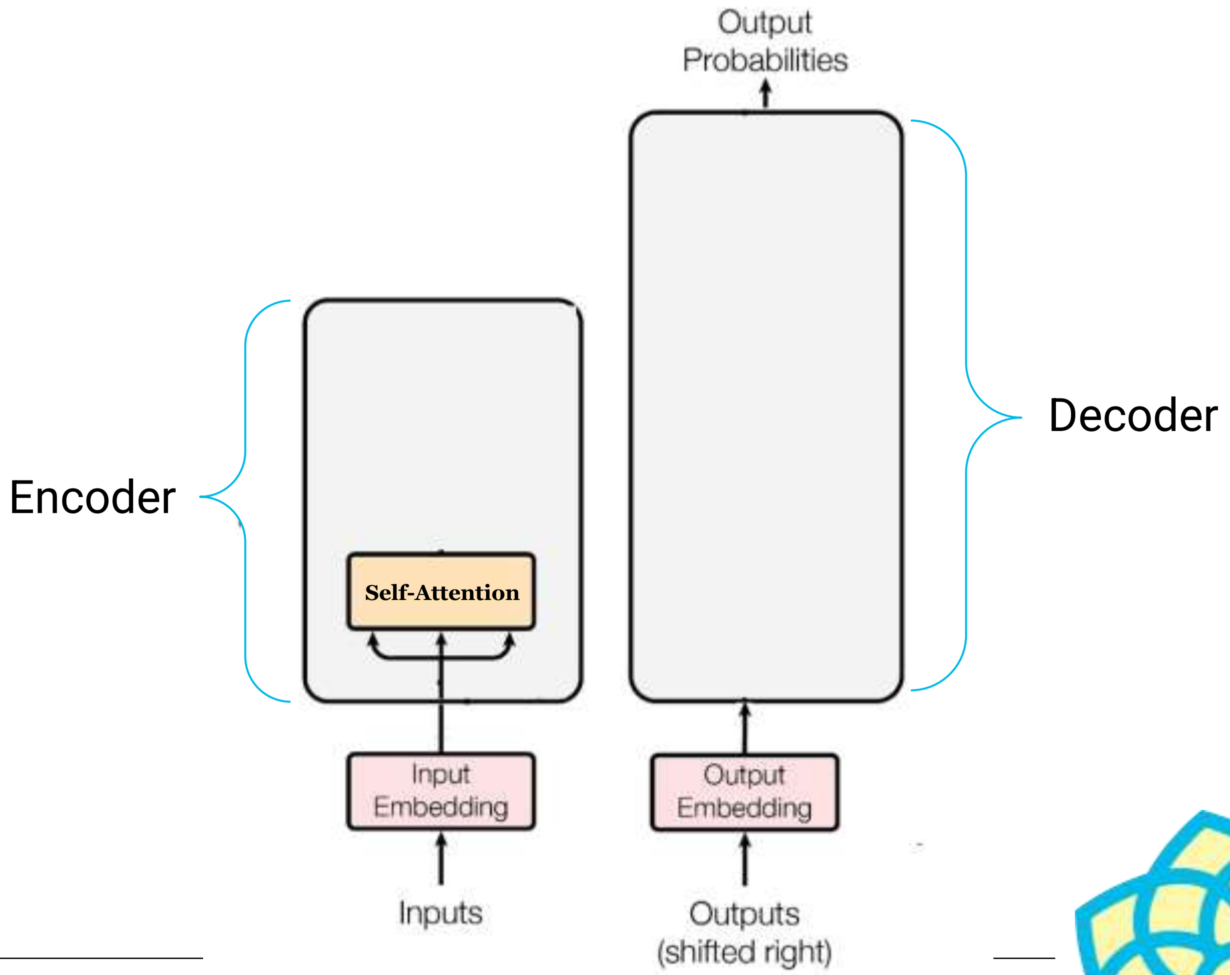


TrustLLM

Funded by the European Union

Encoder: Self-Attention

Self-Attention is the core building block of Transformer, so let's first focus on that!

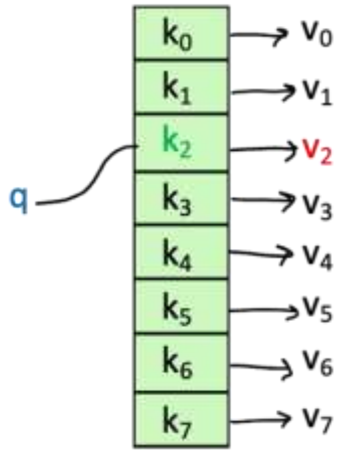


Funded by the European Union

Intuition for Attention Mechanism

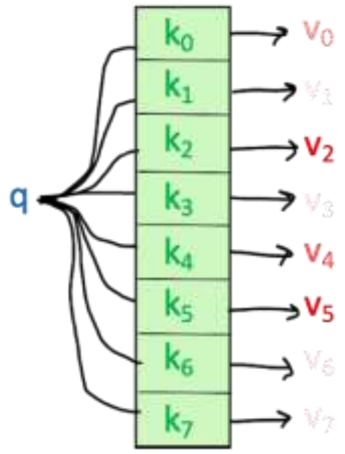
Let’s think of attention as a "fuzzy" or approximate hashtable. To look up a value, we compare a query against keys in a table.

In a hashtable:



Each query (hash) maps to exactly one key-value pair.

In (self-)attention:



Each query matches each key to varying degrees. We return a sum of values weighted by the query-key match.

TrustLLM Funded by the European Union

Recipe for 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.

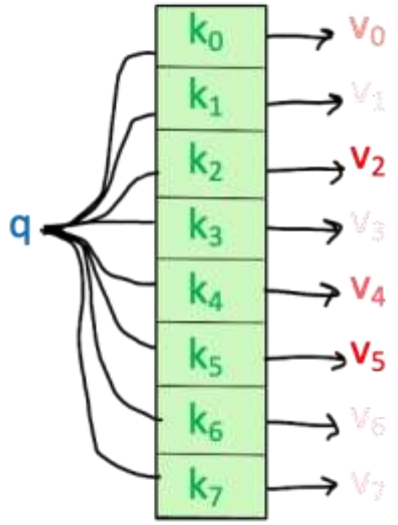
$$e_{ij} = q_i \cdot k_j$$

- 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.

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



Recipe for (Vectorized) Self-Attention in the Transformer Encoder

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

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

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

$$E = QK^T$$

- Step 3: Take the softmax to normalize attention scores.

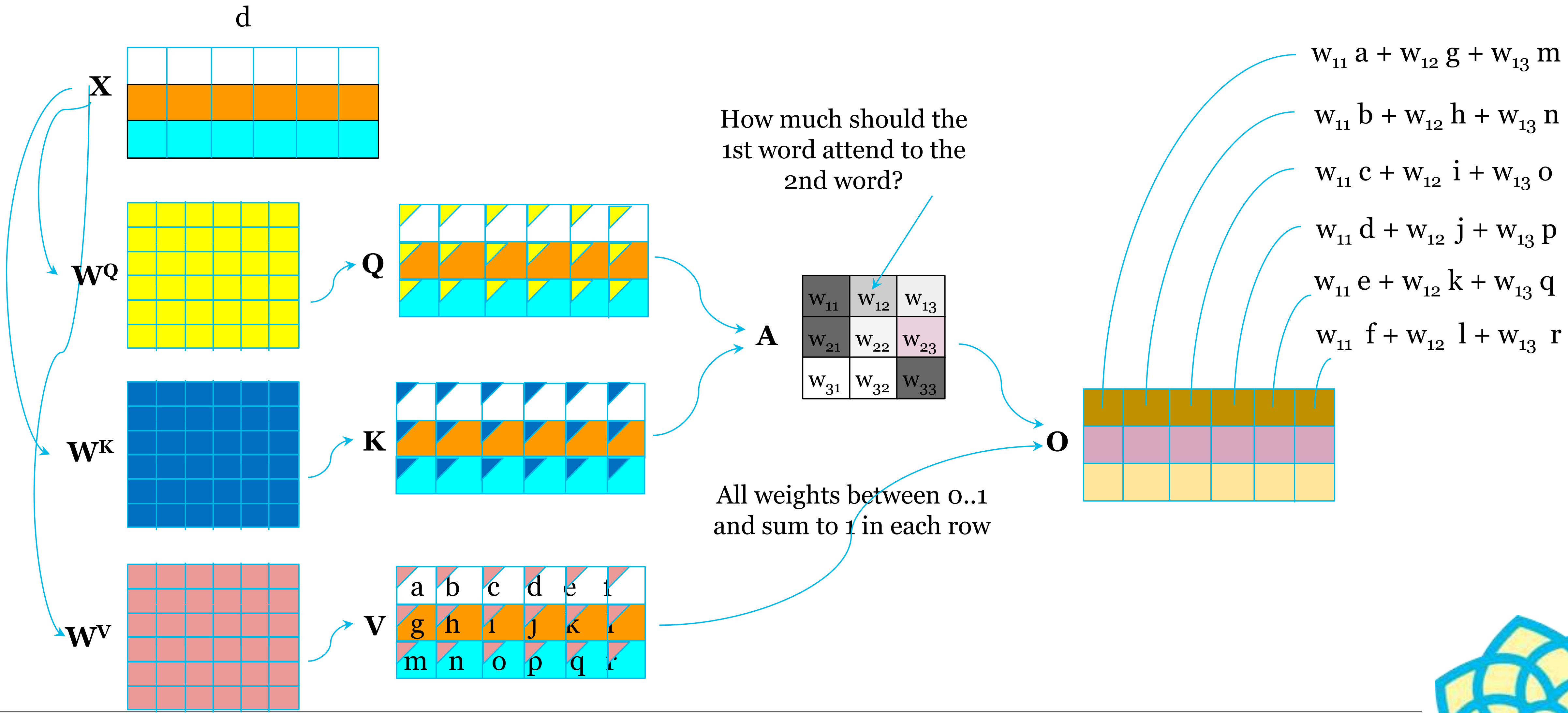
$$A = softmax(E)$$

- Step 4: Take a weighted sum of values.

$$Output = AV$$

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

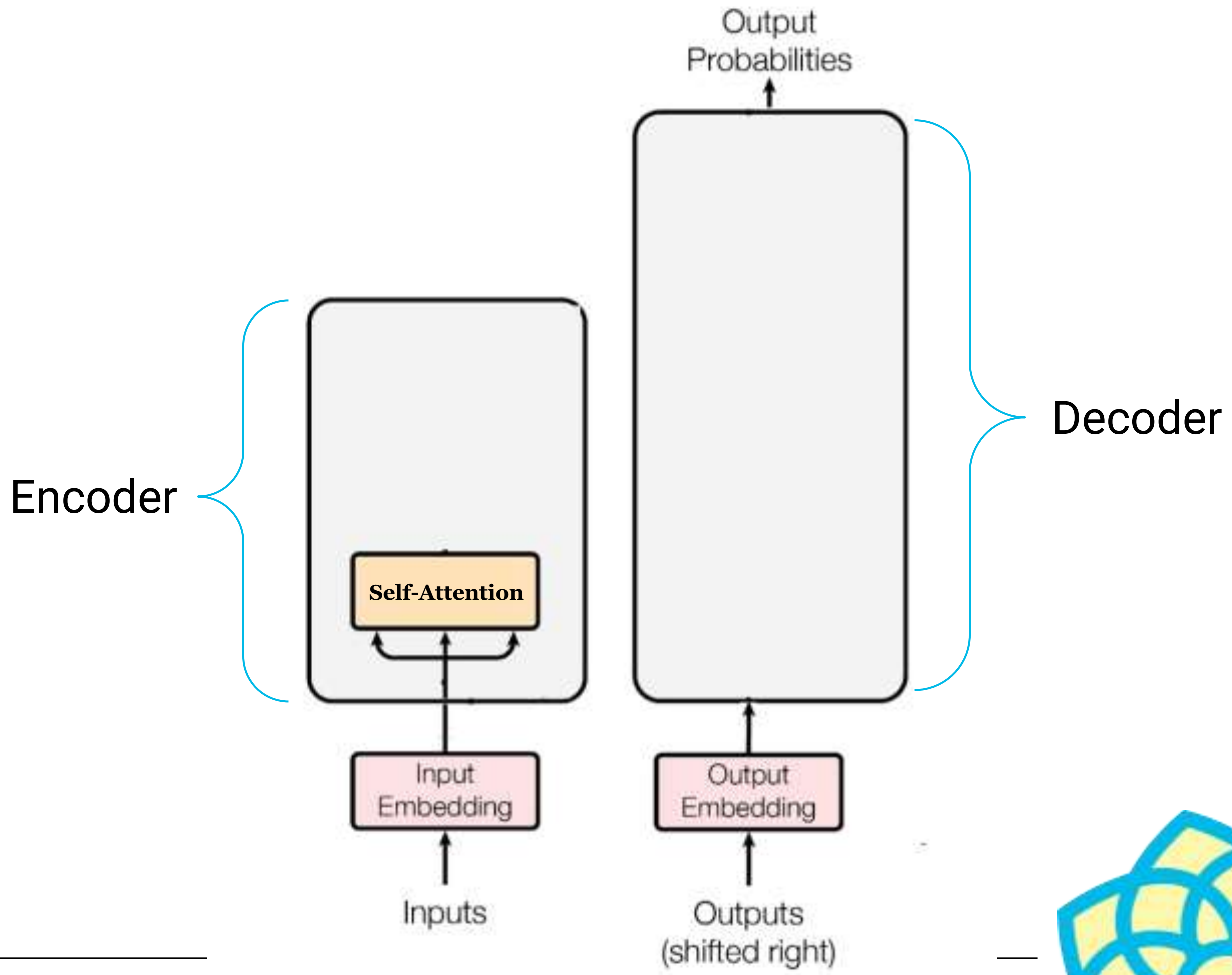
In Pictures (N = 3, d = 6, h = 1)



How much should the 1st word attend to the 2nd word?

All weights between 0..1 and sum to 1 in each row

What We Have So Far: (Encoder) Self-Attention!

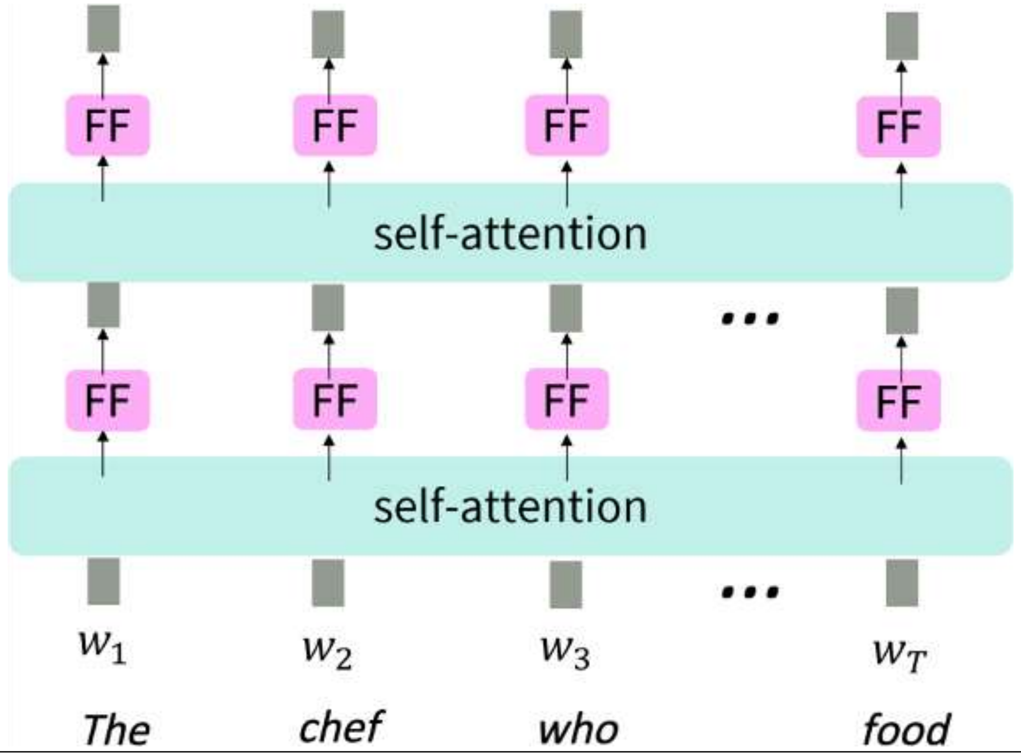


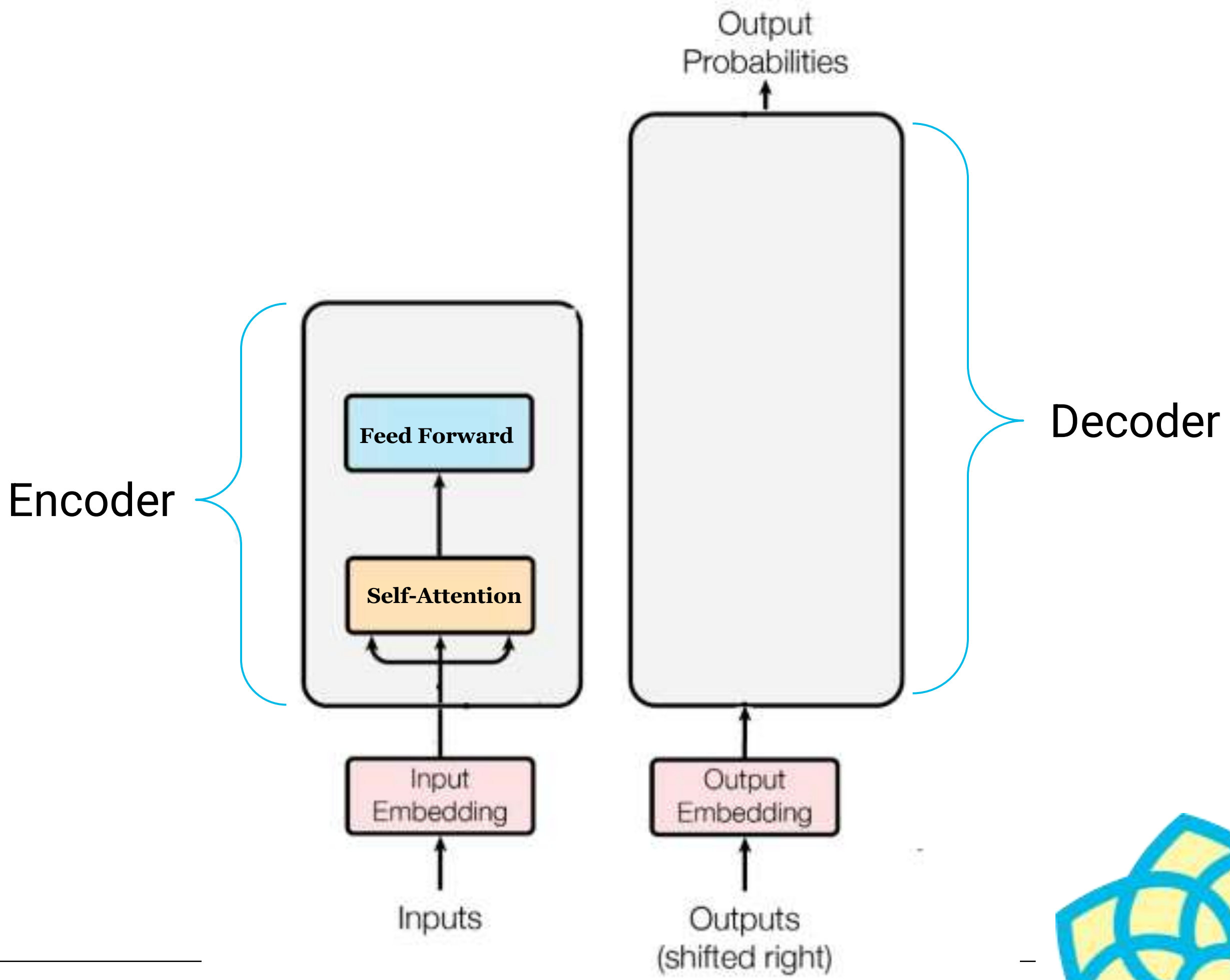
Funded by the European Union

Attention Isn't Quite All You Need!

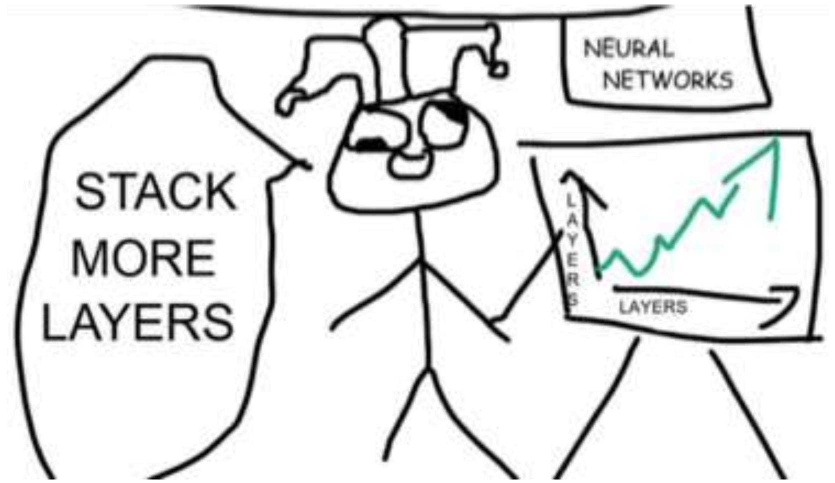
Equation for Feed Forward Layer

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





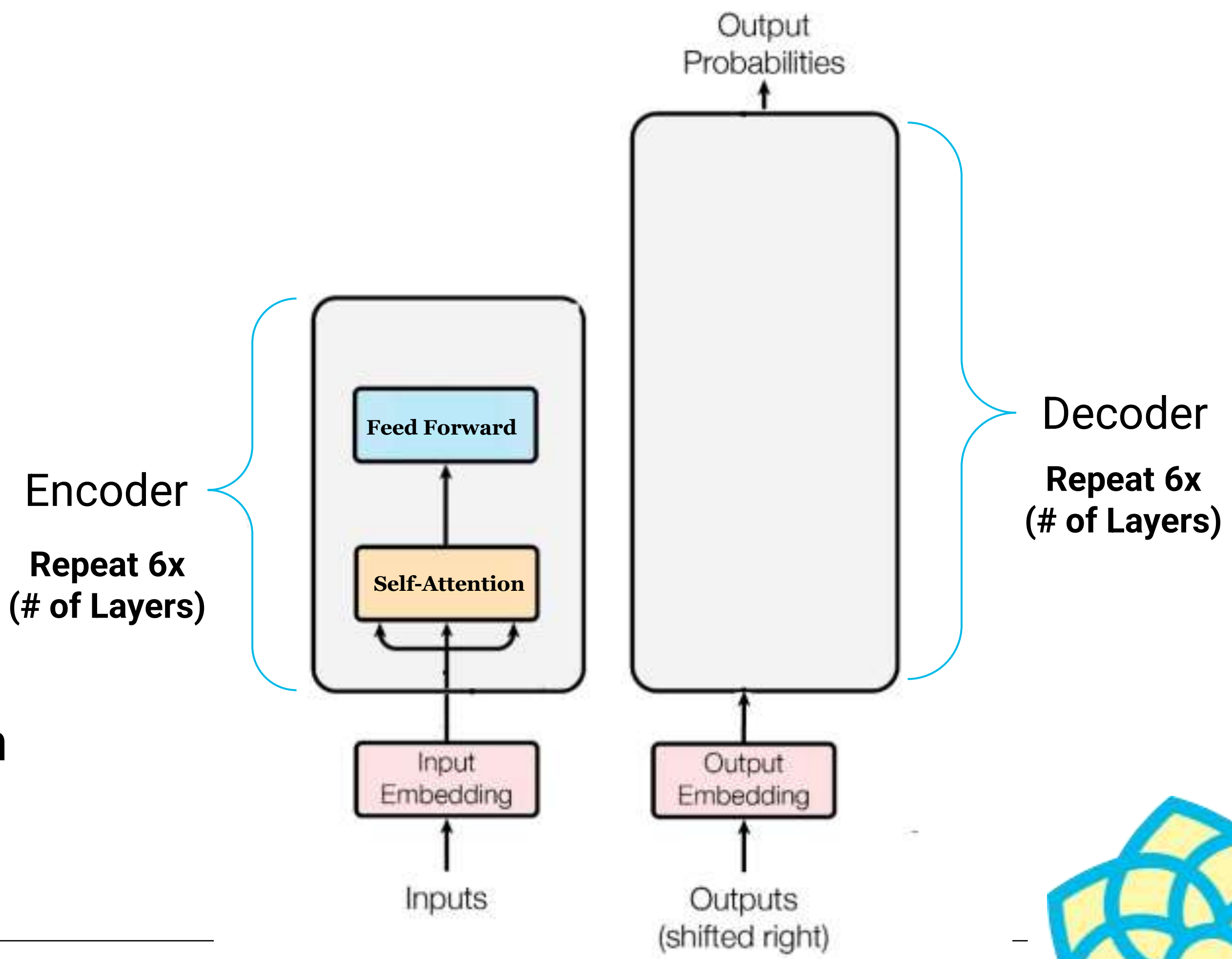
Making This Work For Deep Networks



Training Trick #1: Residual Connections

Training Trick #2: LayerNorm

Training Trick #3: Scaled Dot Product Attention



Training Trick #1: Residual Connections [He et al., 2016]

- 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!

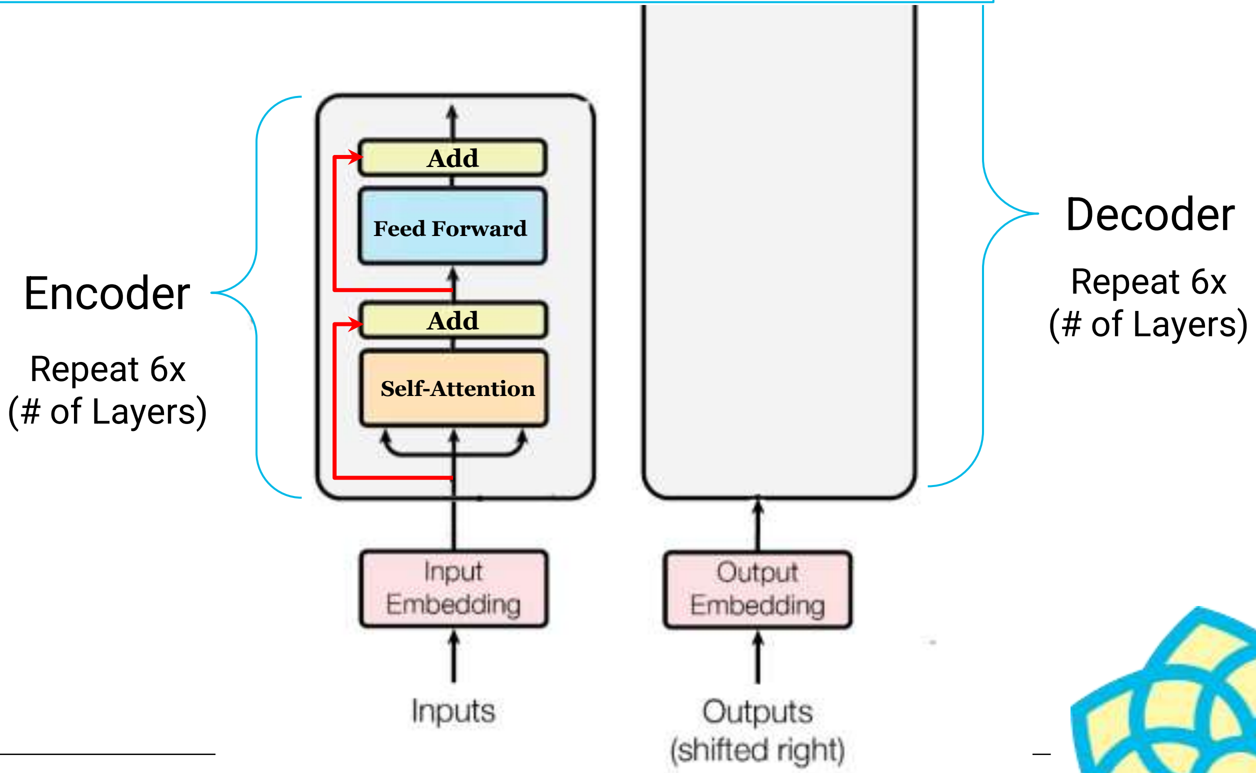
  This prevents the network from "forgetting" or distorting important information as it is processed by many layers.

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

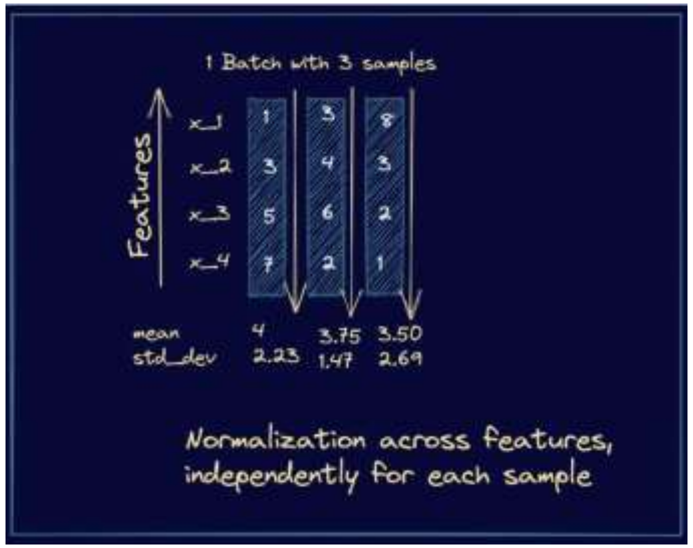
Residual connections are also thought to smooth the loss landscape and make training easier!

[no residuals]
[residuals]

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



Training Trick #2: Layer Normalization [Ba et al., 2016]


1 Batch with 3 samples
Features
x_1
x_2
x_3
x_4
mean
std_dev
Normalization across features, independently for each sample

	Sample 1	Sample 2	Sample 3
x_1	1	3	8
x_2	3	4	3
x_3	5	6	2
x_4	7	2	1
mean	4	3.75	3.50
std_dev	2.23	1.47	2.69

An Example of How LayerNorm Works
(Image by Bala Priya C, Pinecone)

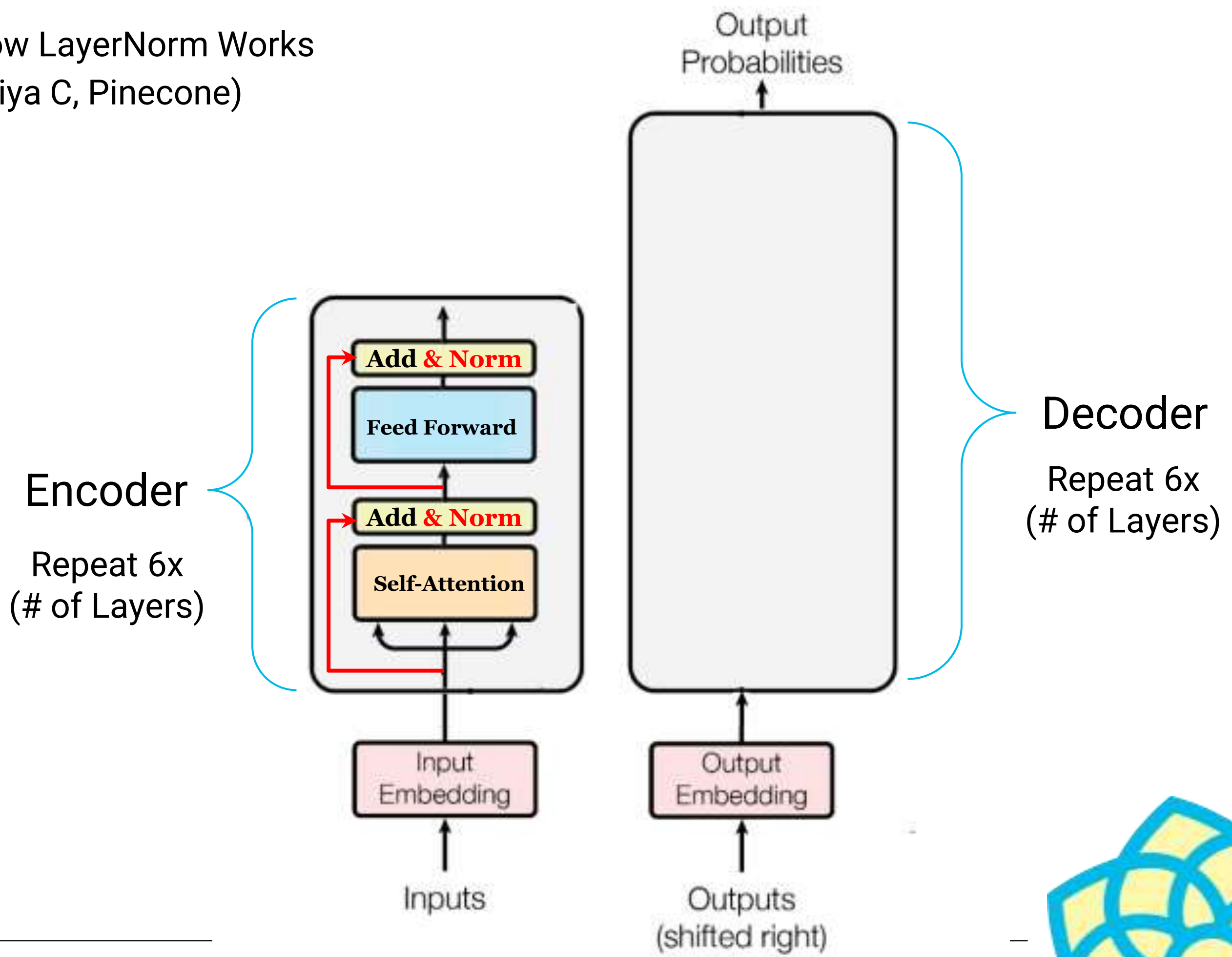
Mean:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$$

Standard Deviation:

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

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



TrustLLM

Training Trick #3: Scaled Dot Product Attention

After LayerNorm, the mean and variance of vector elements is 0 and 1, respectively.

However, the dot product still tends to take on extreme values, as its variance scales with dimensionality d_k.

Quick Statistics Review:

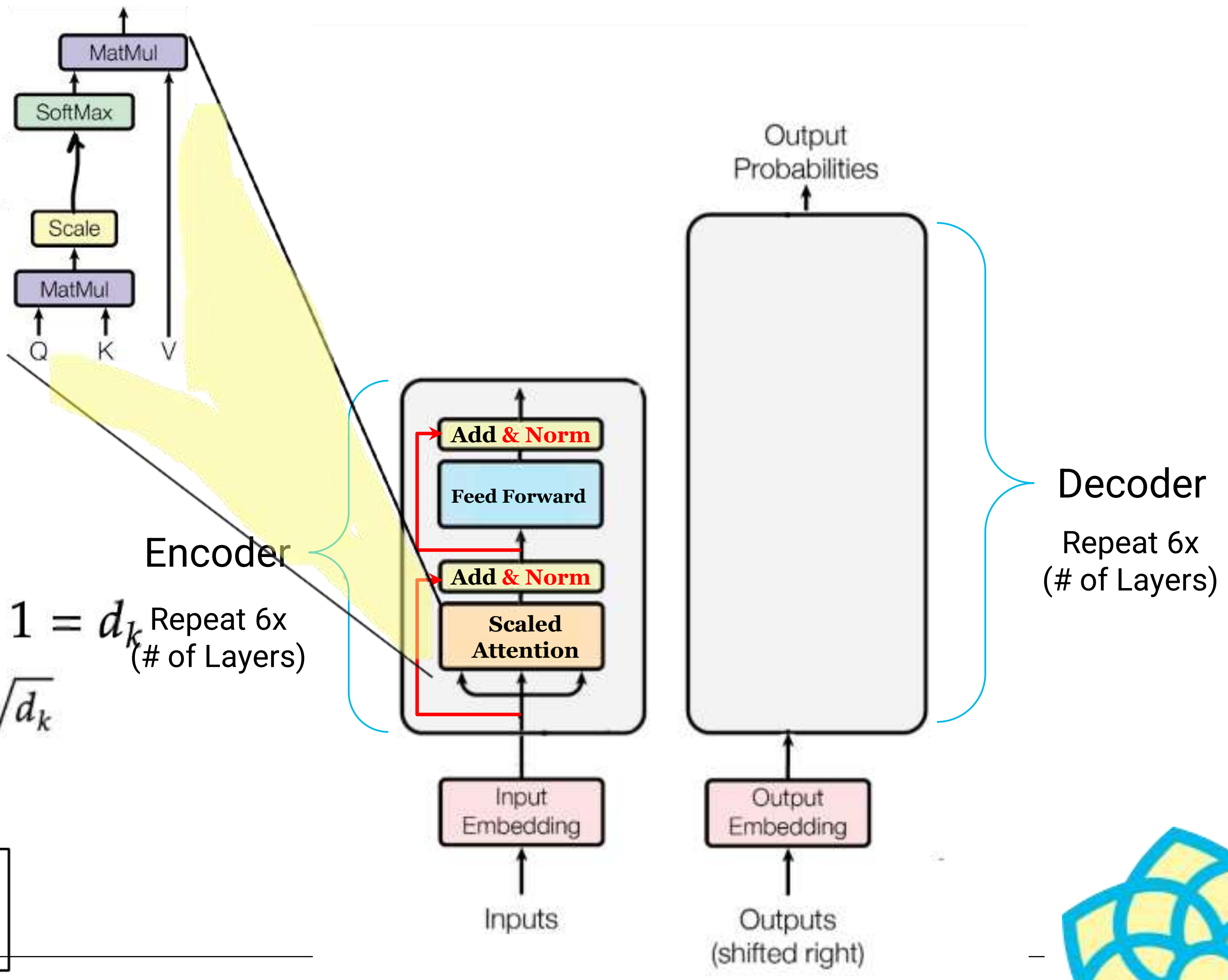
Mean of sum = sum of means = $d_k * 0 = 0$

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

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



Positional Encodings

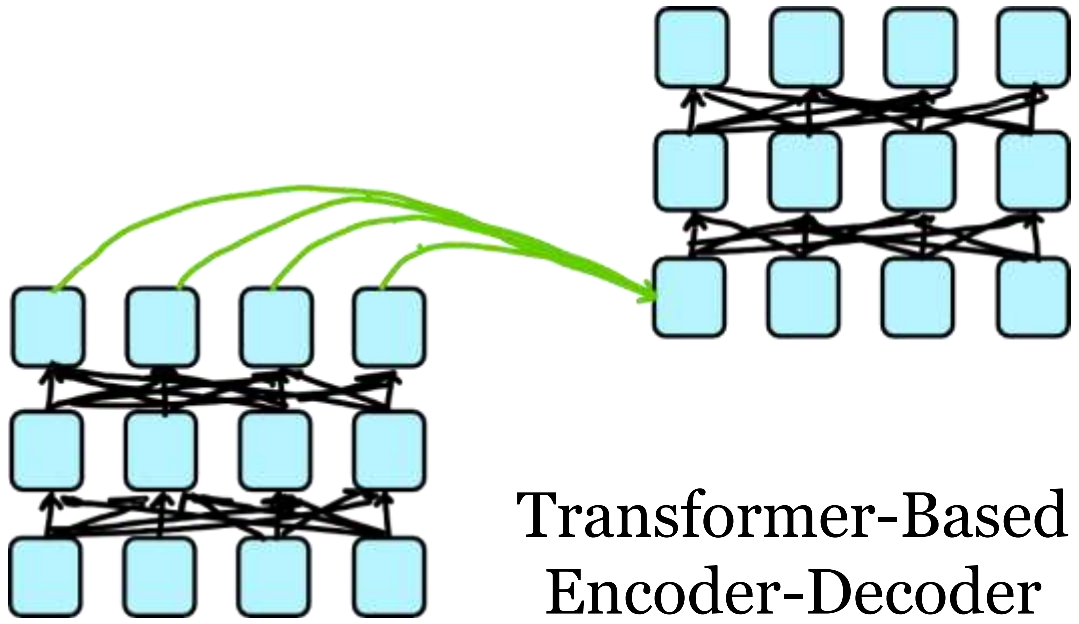
We're almost done with the Encoder, but we have a problem!

Consider this sentence: "Man eats small dinosaur."

Order doesn't impact the network at all!

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

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



Transformer-Based Encoder-Decoder Model

Funded by the European Union

Positional Encodings

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

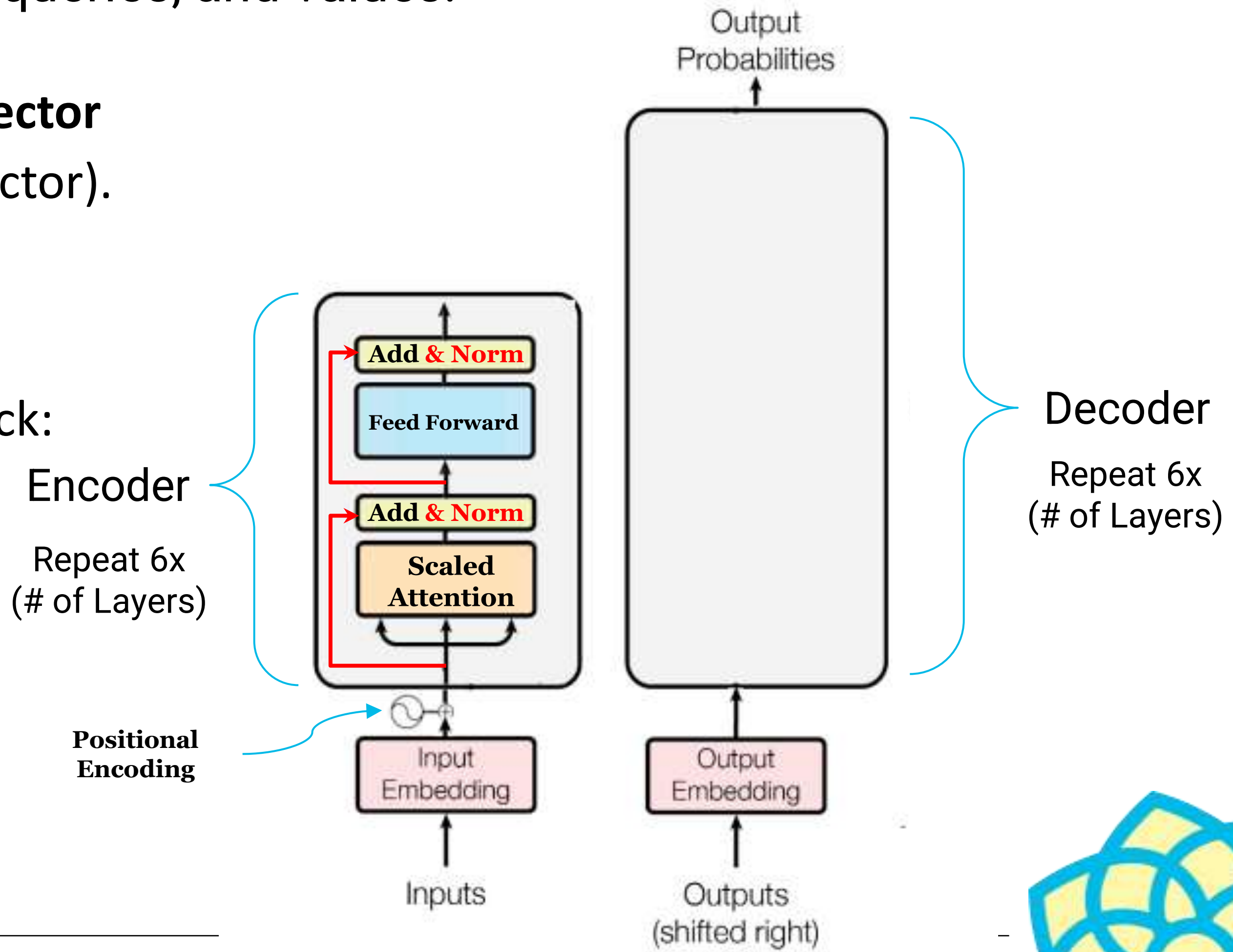
Consider representing each **sequence index** as a **vector** $p_i \in \mathbb{R}^d$, for $i \in \{1,2, \dots, T\}$ (called position vector).

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

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

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

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



Position Representation Vectors Using Sinusoids (Original)

Sinusoidal position representations: concatenate sinusoidal functions of varying periods

We will study pros and cons when we look at two alternatives of such **absolute** position encodings: **relative** position encodings and **rotary** position encodings.

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

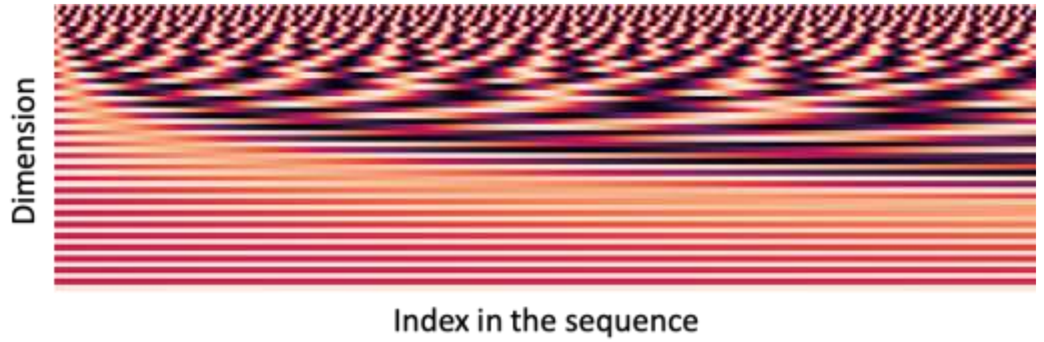
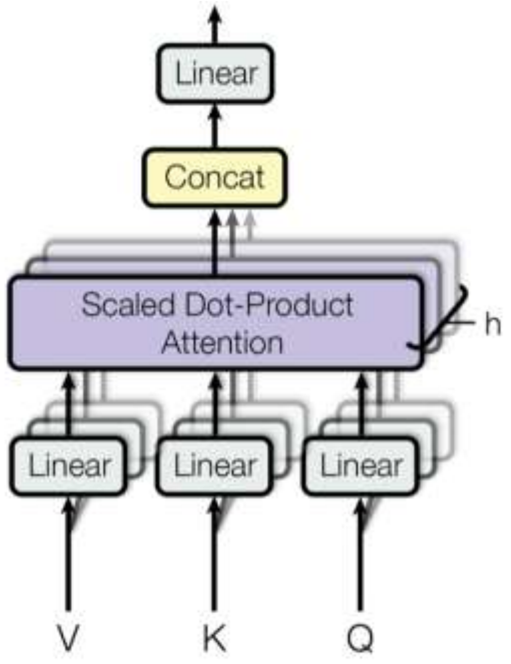


Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

Multi-Headed Self-Attention: k heads are better than 1!

High-Level Idea: Perform self-attention multiple times in parallel and combine the results.



[Vaswani et al. 2017]

Wizards of the Coast, Artist: Todd Lockwood

The Transformer Encoder: 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 $x_i^\top Q^\top K x_j$ s high, but maybe we want to focus on different j for different reasons?

Define **multiple attention "heads"** through multiple Q, K, V matrices!

Let $Q_P, K_P, V_P \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and P ranges from 1 to h.

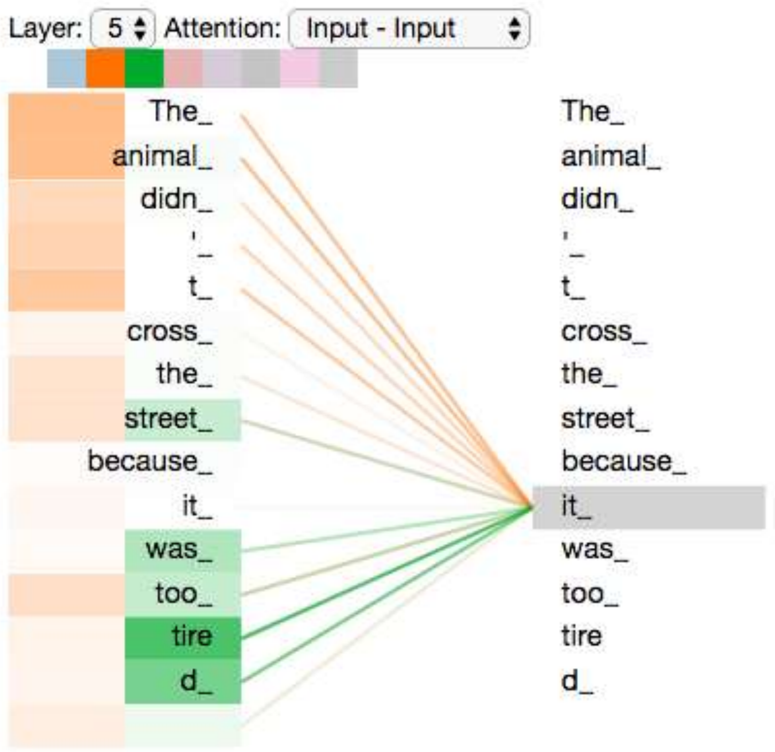
Each attention head performs attention independently:

$$\text{output}_P = \text{softmax}(XQ_PK_P^\top X^\top) * XV_P, \text{ where output}_P \in \mathbb{R}^{d/h}$$

Then the outputs of all the heads are combined!

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

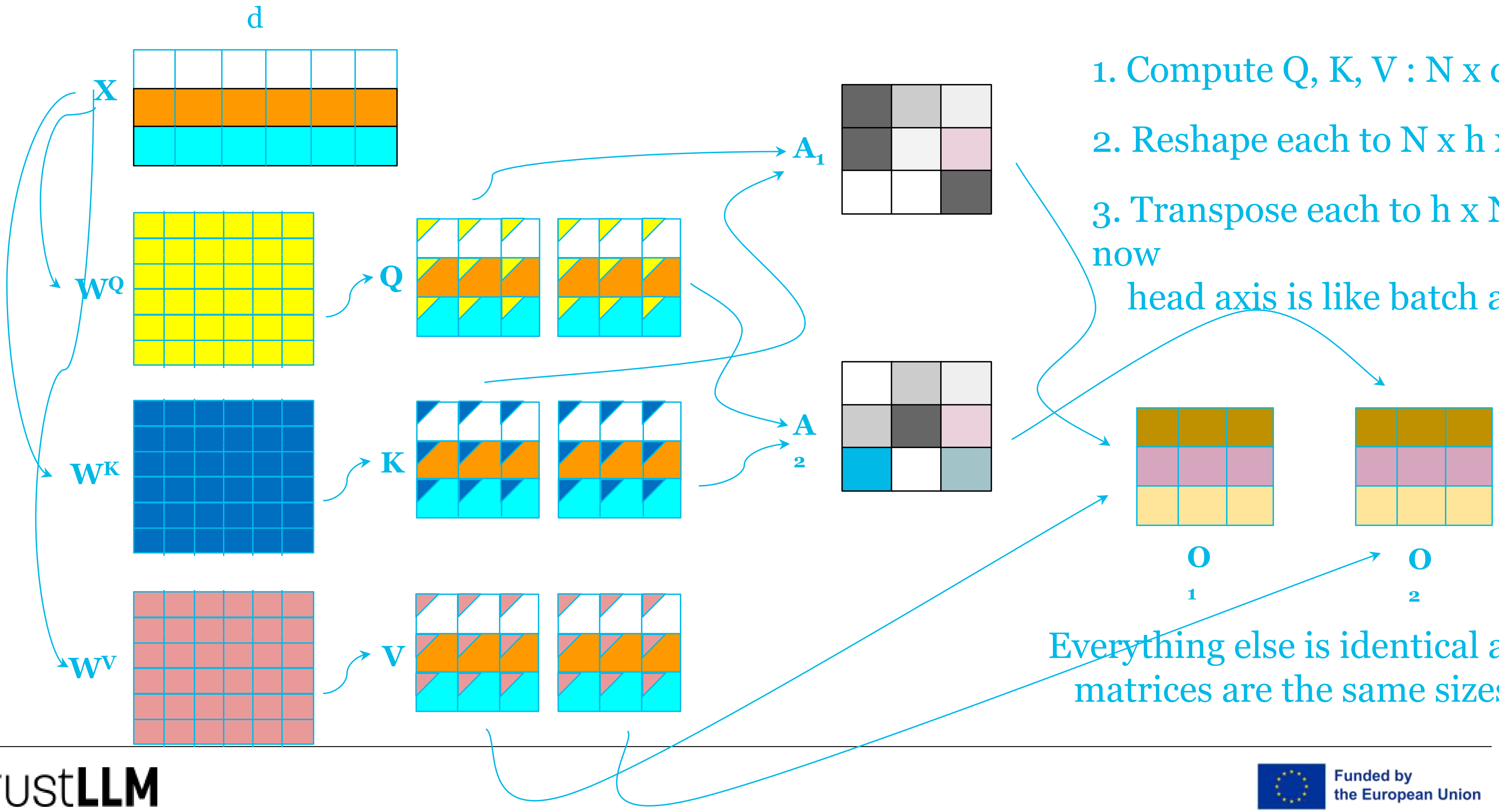
Each head gets to "look" at different things, and construct value vectors differently.



In encoding the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired". The model's representation of the word "it" thus bakes in some of the representation of both "animal" and "tired".
https://jalammar.github.io/illustrated-transformer/

TrustLLM

In Pictures (N = 3, d = 6, h = 2)



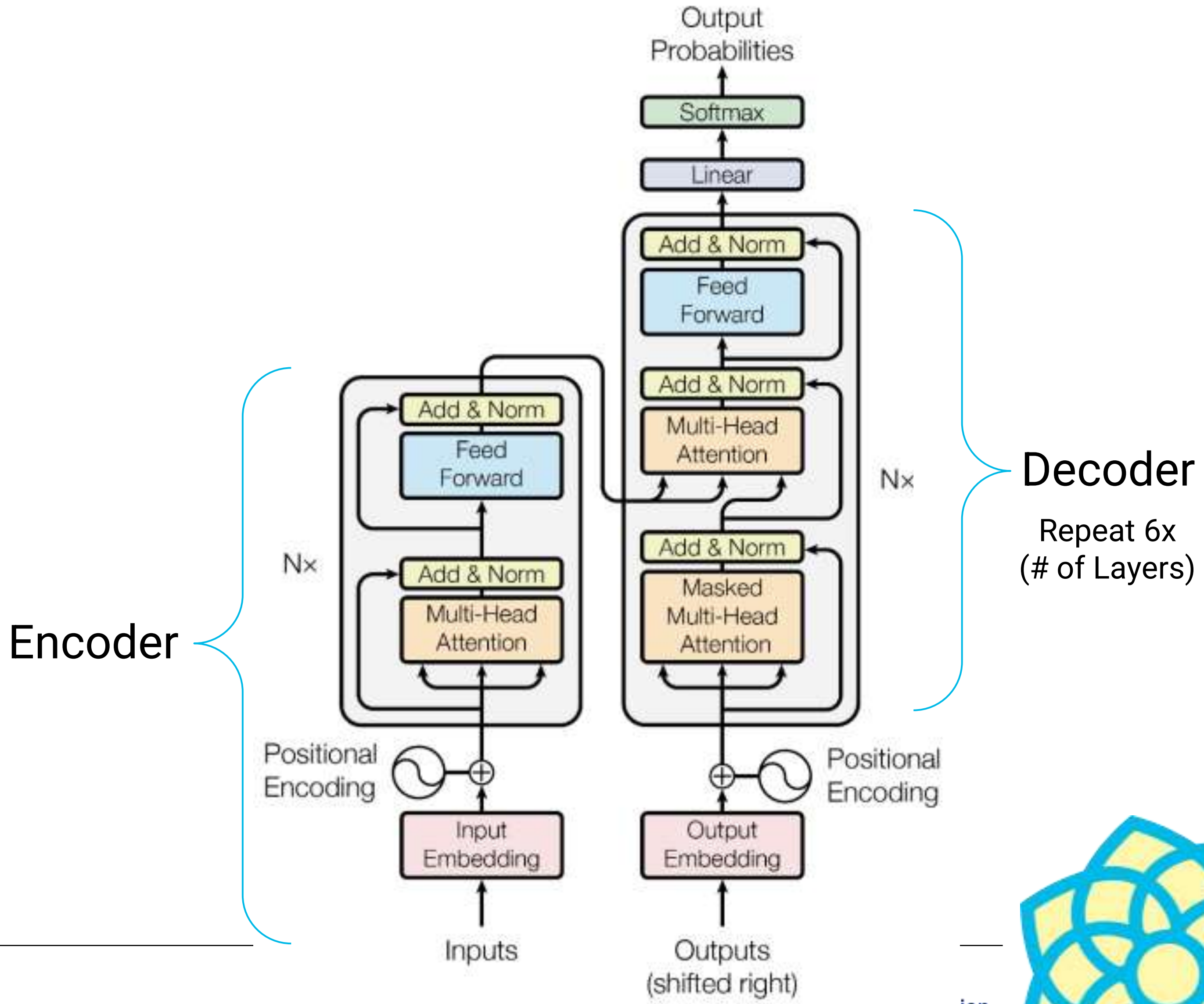
1. Compute Q, K, V : N x d as before
2. Reshape each to N x h x d/h
3. Transpose each to h x N x d/h; now
   head axis is like batch axis

Everything else is identical and matrices are the same sizes!

Funded by the European Union

The Story So Far ...

We've completed the Encoder!

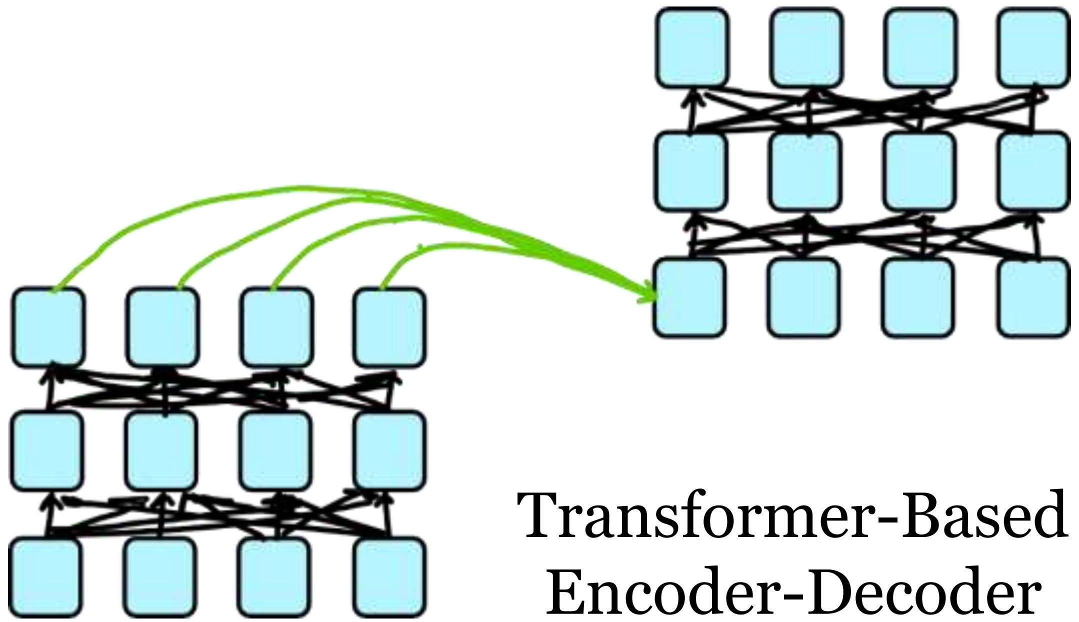


Decoder: Masked Self-Attention

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.



Transformer-Based Encoder-Decoder 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 –∞.

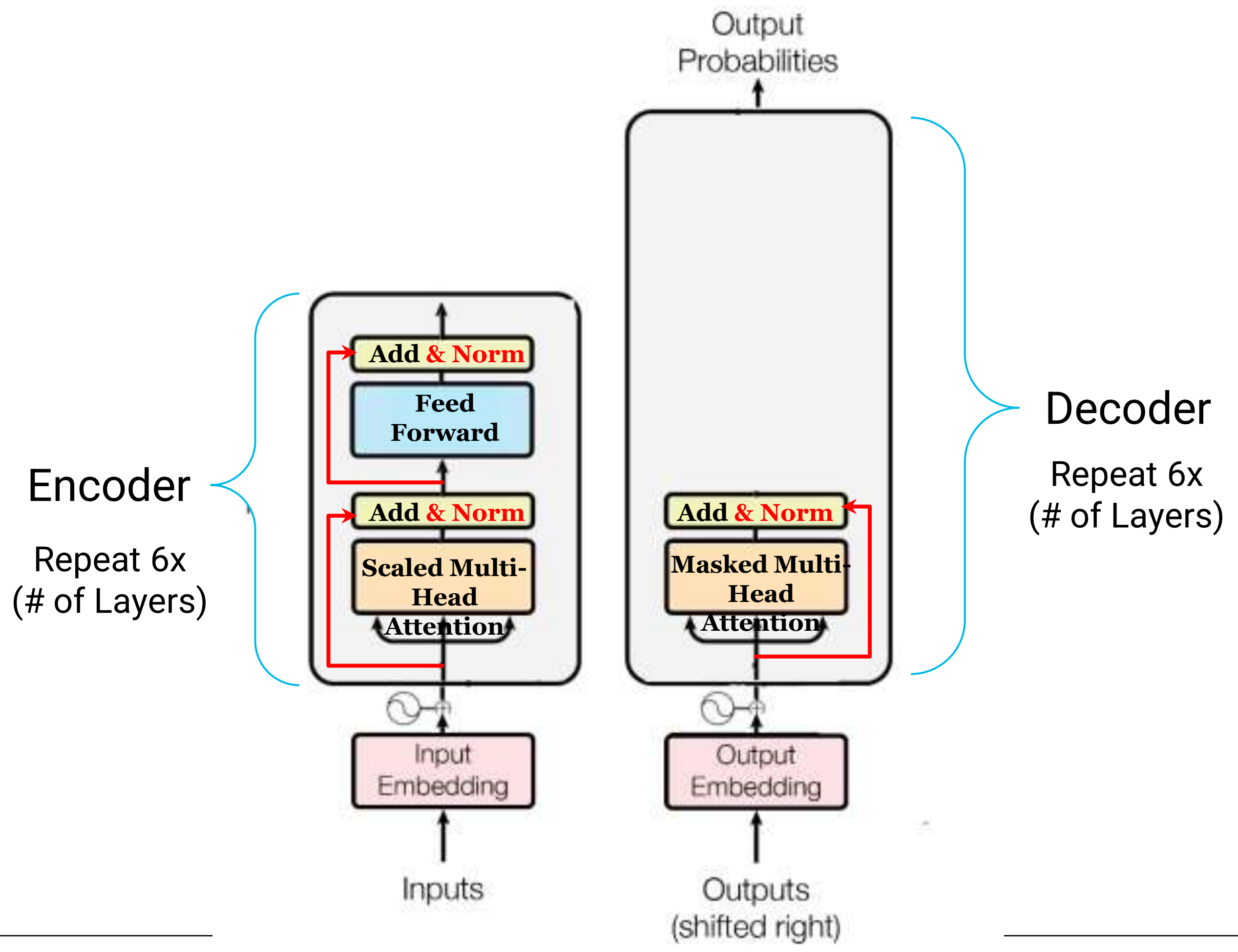
$$e_{ij} = \begin{cases} q_i^\top k_j, j < i \\ -\infty, j \geq i \end{cases}$$

We can look at these (not greyed out) words

For encoding these words

$$\begin{array}{c|cccc} & \text{[START]} & \text{The} & \text{chef} & \text{who} \\ \hline \text{[START]} & -\infty & -\infty & -\infty & -\infty \\ \text{The} & & -\infty & -\infty & -\infty \\ \text{chef} & & & -\infty & -\infty \\ \text{who} & & & & -\infty \end{array}$$

Decoder: Masked Multi-Head Self-Attention



Funded by the European Union

Encoder-Decoder Attention

How does the decoder focus on appropriate places in the input sequence? Using encoder-decoder / cross attention!

Let $\mathbf{h}_1, \ldots, \mathbf{h}_N$ be output vectors from the encode

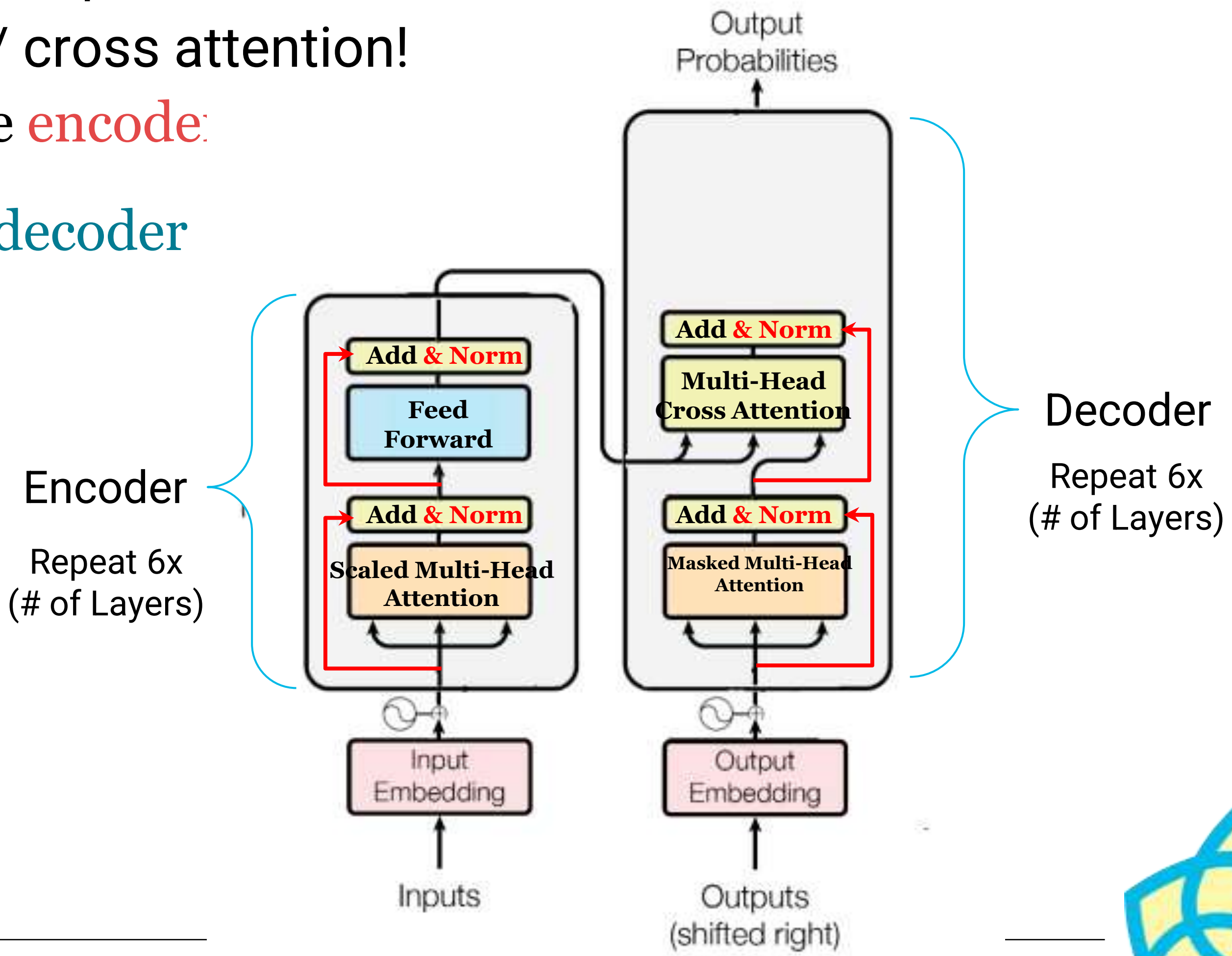
Let $\mathbf{z}_1, \ldots, \mathbf{z}_N$ be input vectors from the decoder

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

$$\mathbf{k}_\mathbf{i} = \mathbf{W}^\mathbf{K}\,\mathbf{h}_\mathbf{i} \text{ and } \mathbf{v}_\mathbf{i} = \mathbf{W}^\mathbf{V}\,\mathbf{h}_\mathbf{i}$$

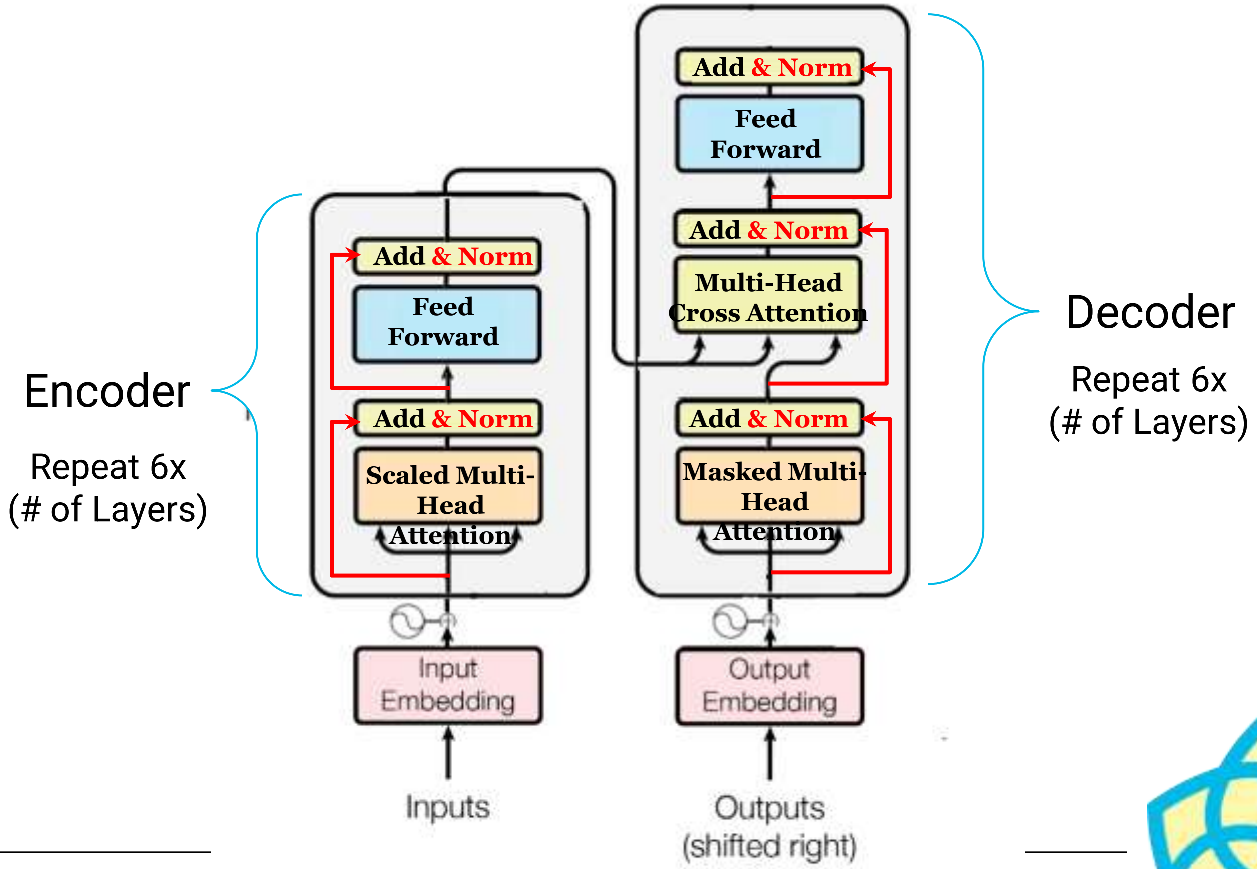
whereas the **queries** are drawn from the decoder:

$$\mathbf{q}_\mathbf{i} = \mathbf{W}^\mathbf{Q}\,\mathbf{z}_\mathbf{i}$$



Decoder: Finishing Touches!

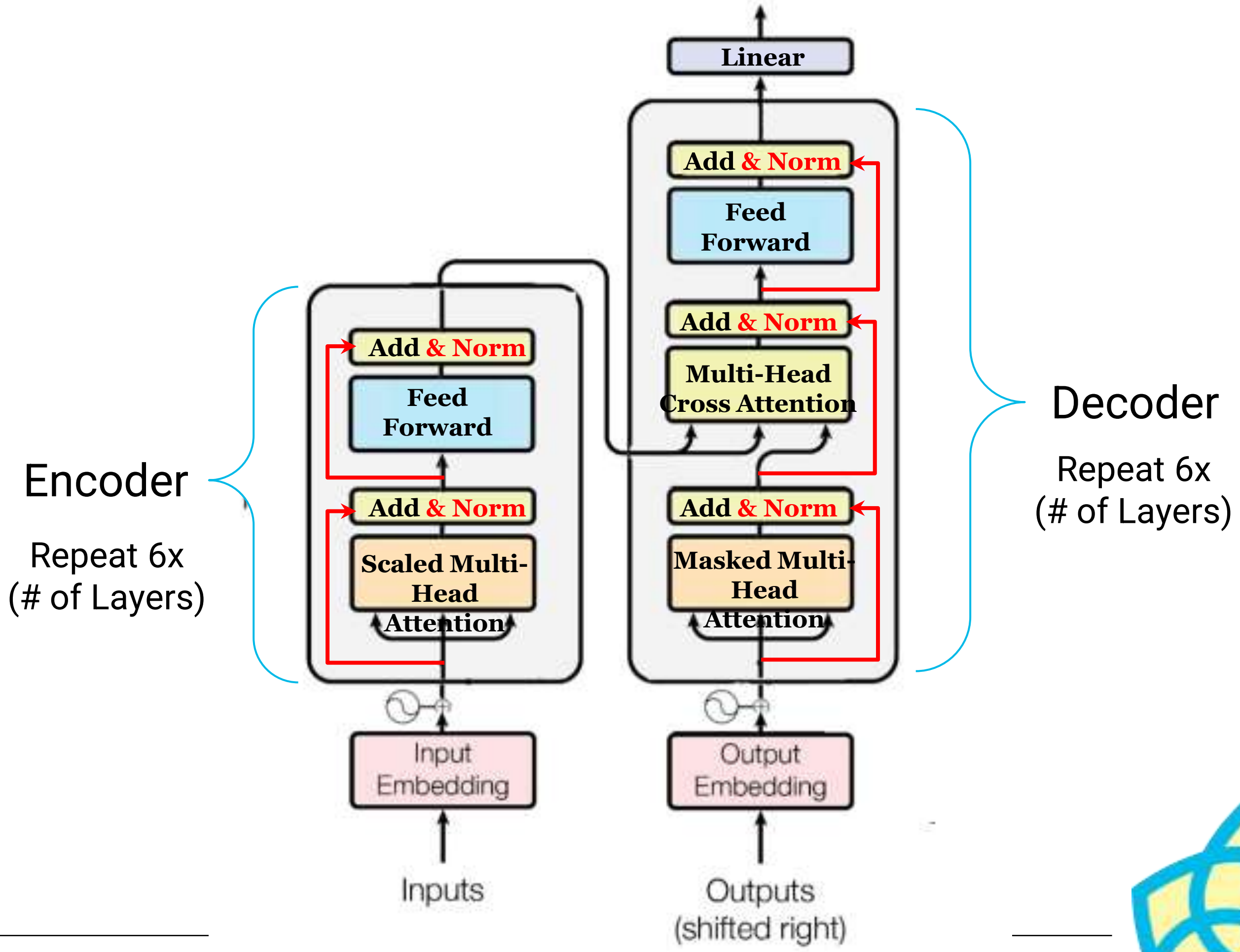
Add a feed forward layer (with residual connections and layer norm).



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).

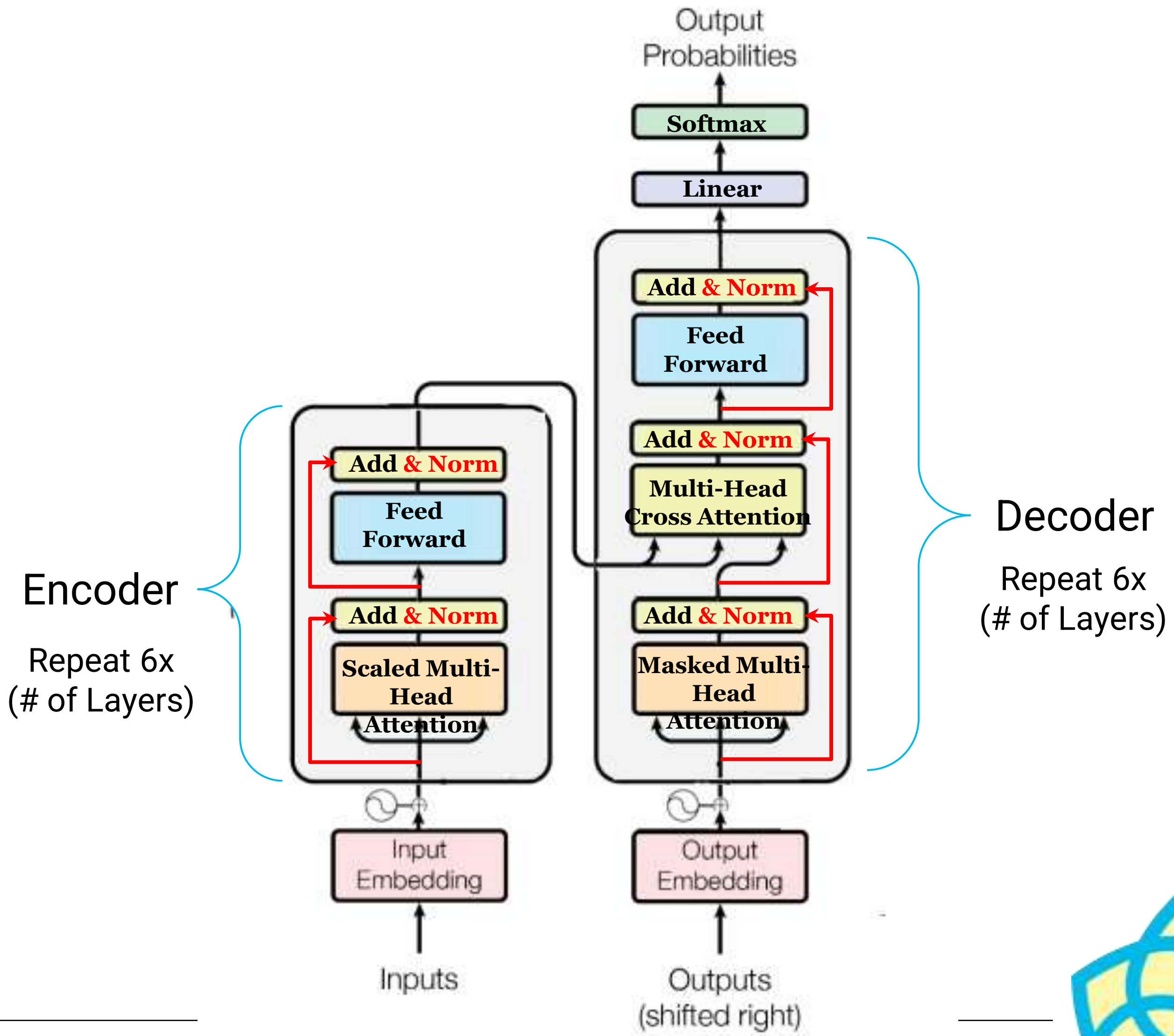


Decoder: Finishing Touches!

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

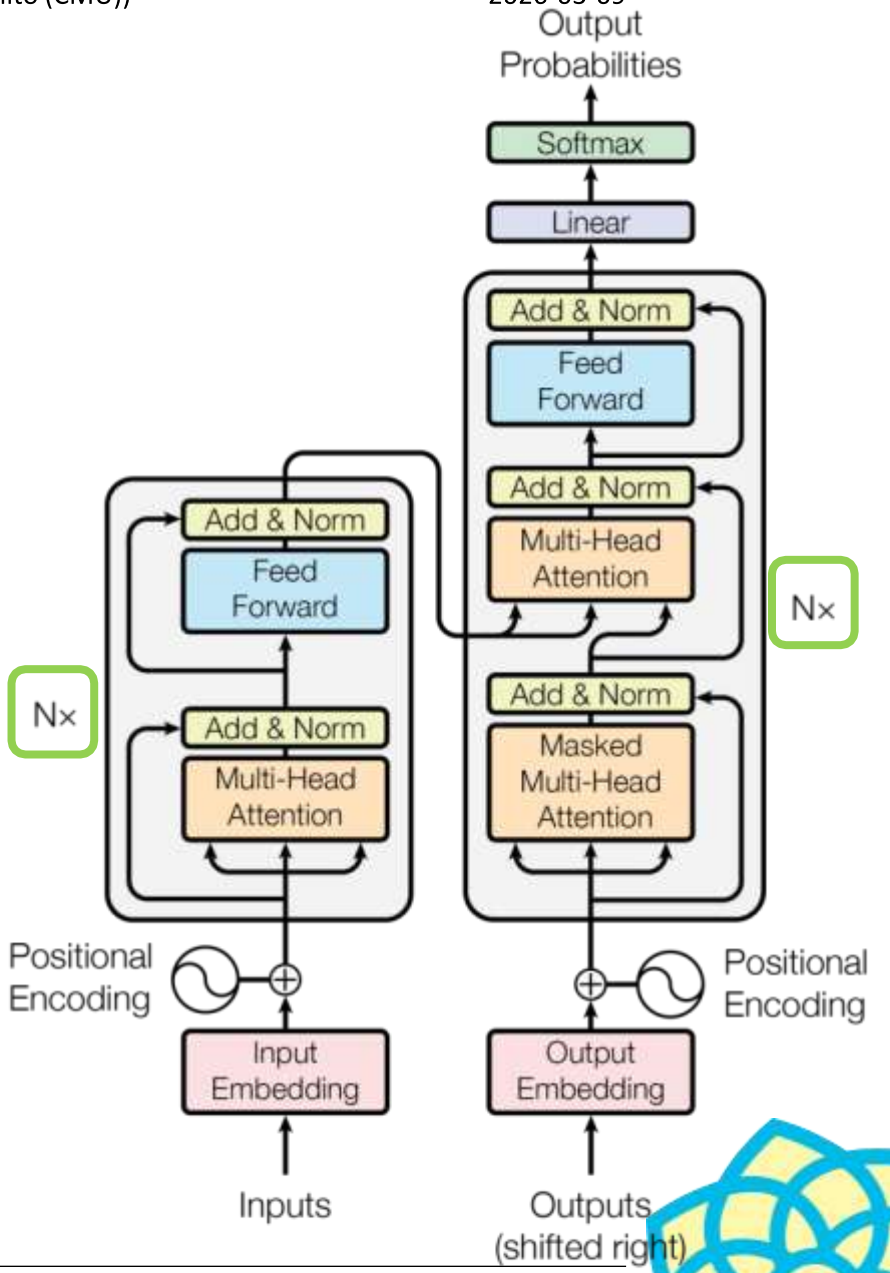
Add a final linear layer to project the embeddings into a much longer vector of length vocab size (logits).

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



Multiple Layers

In practice, there are many attention layers.



Funded by the European Union

Differences in Attention Mechanism of RNN vs. Transformer

Feature	**RNN with Attention (Bahdanau et al. 2015)**	**Transformer**
Attention Type	Additive (Bahdanau) Attention	Scaled Dot Product Attention
Alignment	Based on decoder hidden state and encoder hidden states	Based on dot-product of query and keys (global attention)
Efficiency	Processes sequences step-by-step	Parallel processing of all positions
Context	Weighted sum of encoder hidden states at each step	Attends to all encoder positions for every output
Self-Attention	Not used	Self-attention in both encoder and decoder

Funded by the European Union

- LAIM LE2 VT2026:

  Language Models
  Neural Language Models
  Word embeddings
  Transformers

www.ida.liu.se/~frehe08/llm

LINKÖPING UNIVERSITY