FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

NI-NLM – Lecture 2

Tokenization & the Transformer architecture

Zdeněk Kasner

Recap & math refresher

Recap from last lecture

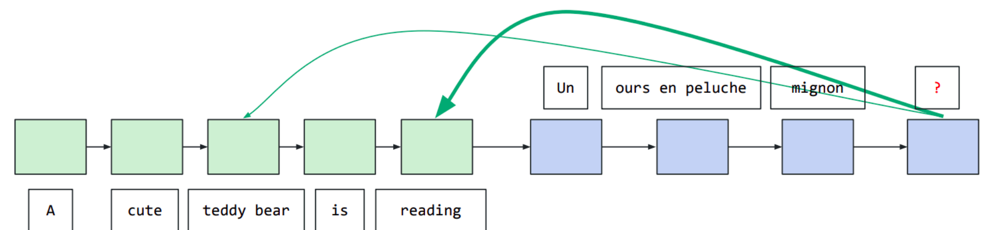
Last time, we covered:

- How to represent **language**?
- What is language **modeling**?
- How to process language with **neural networks**?
  - Also how to do language modeling with neural networks.

Today, we will cover:

How to process language in a more **efficient and scalable way**?

(→ tokenization, attention, Transformer)



source: Stanford CME295

Matrix multiplication

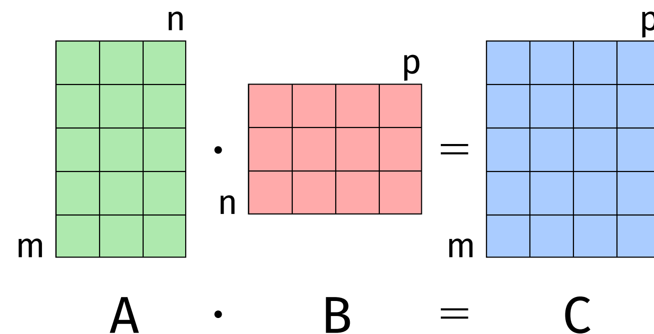
Given matrices $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{n \times p}$, their product $\boldsymbol{C} = \boldsymbol{AB}$ is:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

Example: matrix-vector multiplication:

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_{1,1}b_1 + a_{1,2}b_2 \\ a_{2,1}b_1 + a_{2,2}b_2 \end{pmatrix}$$

$$\underbrace{(2 \times 2)}_{\boldsymbol{A}} \cdot \underbrace{(2 \times 1)}_{\boldsymbol{b}} = \underbrace{(2 \times 1)}_{\boldsymbol{c}}$$



source: Wikipedia

Single-layer feed-forward neural network

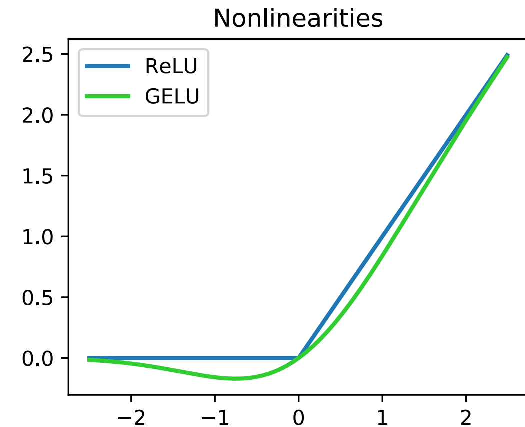
A single-layer **feed-forward neural network** (FFNN) applies a linear transformation (using the matrix W and bias b) followed by a non-linear function σ (ReLU, GELU, ...):

$$\boldsymbol{y}_1 = \sigma(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1)$$

Example: FFNN with ReLU applied on a vector x:

$$\underbrace{\begin{pmatrix} -0.38 & 1.35 \\ 0.70 & 0.30 \\ -1.03 & -1.03 \end{pmatrix}}_{\boldsymbol{W}_1 \in \mathbb{R}^{3\times 2}} \cdot \underbrace{\begin{pmatrix} -0.88 \\ 0.73 \end{pmatrix}}_{\boldsymbol{x} \in \mathbb{R}^2} + \underbrace{\begin{pmatrix} 0.10 \\ 0.21 \\ -0.48 \end{pmatrix}}_{\boldsymbol{b}_1 \in \mathbb{R}^3} = \underbrace{\begin{pmatrix} 1.42 \\ -0.19 \\ -0.33 \end{pmatrix}}_{\boldsymbol{z} \in \mathbb{R}^3}$$

$$\mathrm{ReLU}\left(\begin{pmatrix} 1.42 \\ -0.19 \\ -0.33 \end{pmatrix}\right) = \underbrace{\begin{pmatrix} 1.42 \\ 0.00 \\ 0.00 \end{pmatrix}}_{\boldsymbol{y}_1 \in \mathbb{R}^3}$$



source: Wikipedia

Softmax

Given a vector $\boldsymbol{x} \in \mathbb{R}^d$, **softmax** ensures that $\forall i \ \mathrm{softmax}(x_i) \geq 0$ and $\sum(\boldsymbol{x}) = 1$:

$$\mathrm{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{K} \exp(x_j)}$$

How to think about softmax

A convenient way how to transform any vector to a valid probability distribution.

Example:

$$\boldsymbol{z} = [2.0, 1.0, 0.1]$$

$$\mathrm{softmax}(\boldsymbol{z}) = [0.659, 0.243, 0.099]$$

Implementation in numpy:

```
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x))
```

Why softmax?

Given a vector $x \in \mathbb{R}^d$, **softmax** ensures that $\forall x_i \geq 0$ and $\sum(x) = 1$:

```
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x))
```

However, the same can be achieved with these two normalization functions:

```
def l1_norm(x):
    return np.abs(x) / np.sum(np.abs(x))
```

```
def min_sum_norm(x):
    x_norm = (x - np.min(x))
    return x_norm / np.sum(x_norm)
```

Question

What are the specifics and (dis)advantages of each of these three functions?

Tokenization

From words to tokens

Tokenization = splitting text into smaller units (**tokens**) that the model will process.

We assumed that our input is a **sequence of words**. That sounds good in theory, but does not work in practice:

Question

How would you split these strings into meaningful units?

- `Pneumonoultramicroscopicsilicovolcanoconiosis`
- 大型语言模型
- 🤗❤️🍾🥳

Why subword tokenization?

Idea

Keep the common words, split rare words into subwords.

Compare:

- Word-level: `["unhappiness"]` → out of vocabulary
- Character-level: `["u","n","h","a","p","p","i","n","e","s","s"]`
- Subword: `["un", "happiness"]` ✓

Question

How to decide where to split the subwords?

Byte pair encoding (BPE)

source: Sennrich et al. 2016

Originally a data compression algorithm, adapted for NLP tokenization.

BPE algorithm

1. Start with a vocabulary of all **individual characters** in the training corpus.
2. Count all **adjacent pairs** of tokens in the corpus.
3. **Merge** the most frequent pair into a new token.
4. **Repeat** steps 2–3 until the desired vocabulary size is reached.

The result is a **merge table** that defines how to tokenize any new text.

Exercise time!

BPE – step by step

source: BPE visualizer

Corpus: "aab baac aab baac caaba aac" → initial vocabulary: {a, b, c, _}

Step	Most frequent pair	New token
1	(a, a) → 6 times	aa
2	(aa, b) → 3 times	aab
3	(aa, c) → 3 times	aac
4	(aab, _) → 2 times	aab_

Final vocabulary: {_, a, b, c, aa, aab, aac, aab_}

Tokenization algorithms

Notes:

- In practice, the BPE algorithm **never crosses word boundaries**. → By convention, you will find the space character _ always *before* the other characters.
- Modern implementations use **Byte-Level BPE**, where the basic vocabulary has all the 256 possible byte values → no unknown characters.
- The **space "_" in BPE** is a special character with the byte value `0x20`.
  - This is often rendered as Ġ, which is artifact of GPT-2's byte-to-unicode mapping.
- There are other tokenization algorithms (SentencePiece, WordPiece, Unigram).
- Typical vocabulary size in practice: **50k–200k** tokens.

Tokenizer playground

How do real tokenizers split text?

👉 https://huggingface.co/spaces/Xenova/the-tokenizer-playground

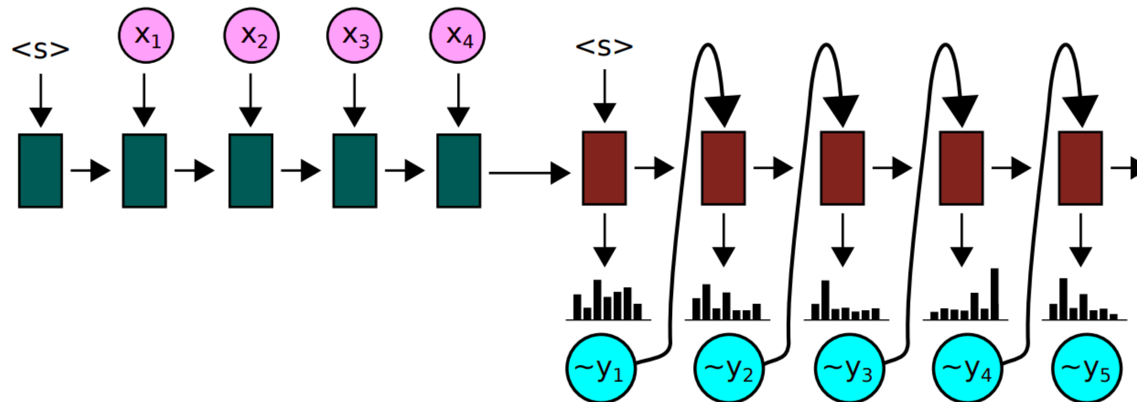
Notice the difference between languages.

Attention mechanism

The seq2seq bottleneck

Remember: the encoder compresses the **entire input** into a single vector h_N.

- Works OK for short sequences, but for longer inputs the **information gets lost**.
- The decoder has no way to "look back" at specific parts of the input.



source: NPFL116

Attention – the idea

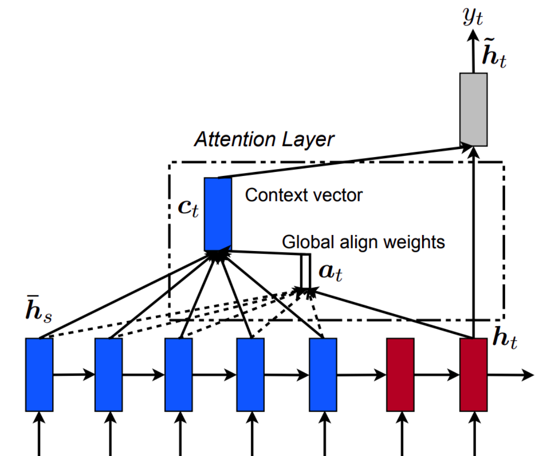
source: Luong et al. (2015)

Idea

Let the decoder **pick the information from the encoder hidden states**.

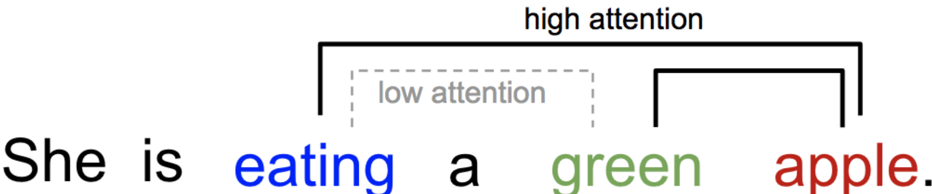
How exactly? At each decoding step t:

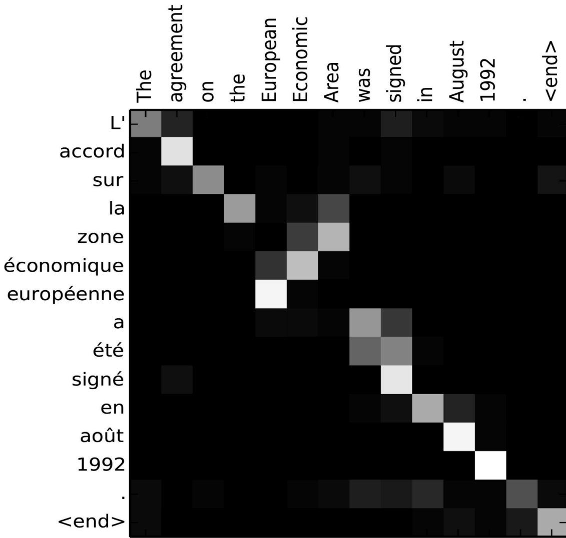
1. Compute a **score** for each encoder state $\boldsymbol{h}_i$.
2. Normalize scores with **softmax** → attention weights $\alpha_{t,i}$.
3. Compute a **weighted sum** of hidden states → context vector: $\boldsymbol{c}_t = \sum_i \alpha_{t,i} \boldsymbol{h}_i$.
4. Use $\boldsymbol{c}_t$ together with the decoder state to predict the next token.



Attention with RNNs

The attention weights correspond to **which input tokens the decoder focuses on** at each step.





source: Lil'Log

source: Bahdanau et al. (2014)

But it is slow...

RNN needs to process the input sequentially, token-by-token → there is no way to parallelize the process.

> **Idea**
>
> Can we get rid of the recurrence entirely?

The Transformer

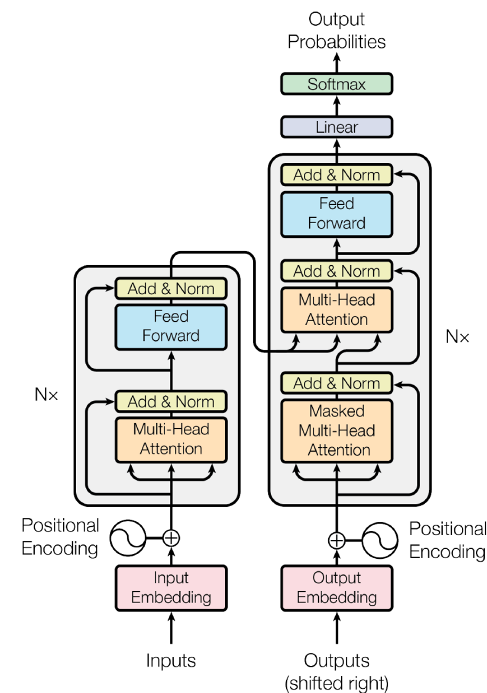
Attention is all you need

source: Vaswani et al. 2017

Paper by Google in 2017, originally proposed for improving MT systems.

How to get rid of recurrence:

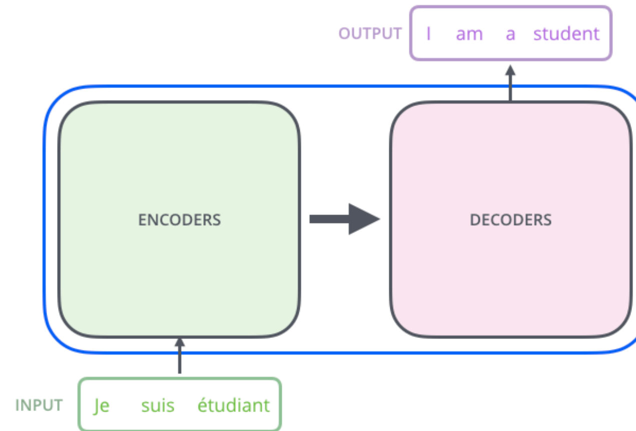
- Process each state **in parallel with FFNNs**.
- Use the **attention mechanism** to share information between the tokens.
- Apply it repeatedly in **layers** to make it more expressive.
- Use **positional encoding** to inject position information.



Transformer – high-level view

source: The Illustrated Transformer

The original Transformer follows the **encoder-decoder** pattern (→seq2seq generation).



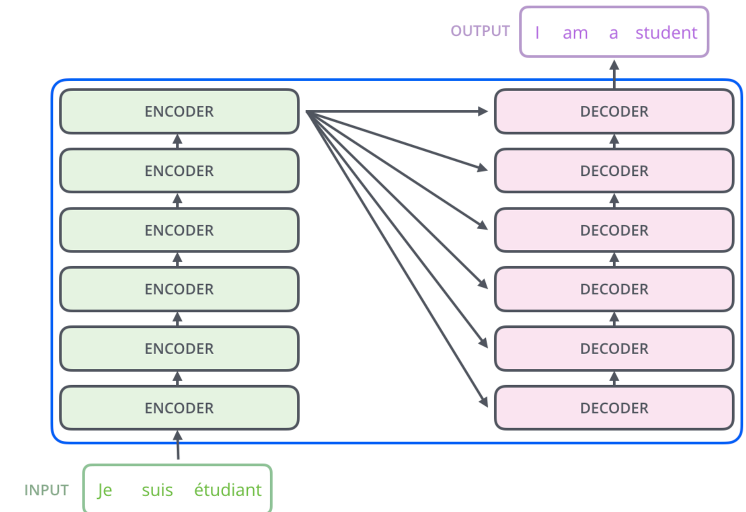
- **Encoder**: reads the input sequence, produces a rich representation.
- **Decoder**: generates the output sequence one token at a time.

The encoder-decoder stack

source: The Illustrated Transformer

Both encoder and decoder are **stacks of identical layers** (called “blocks”).

- The original Transformer uses 6 layers for both.
- Each layer refines the representation.
- The output of the last encoder layer is passed to **every decoder layer**.



Current scales

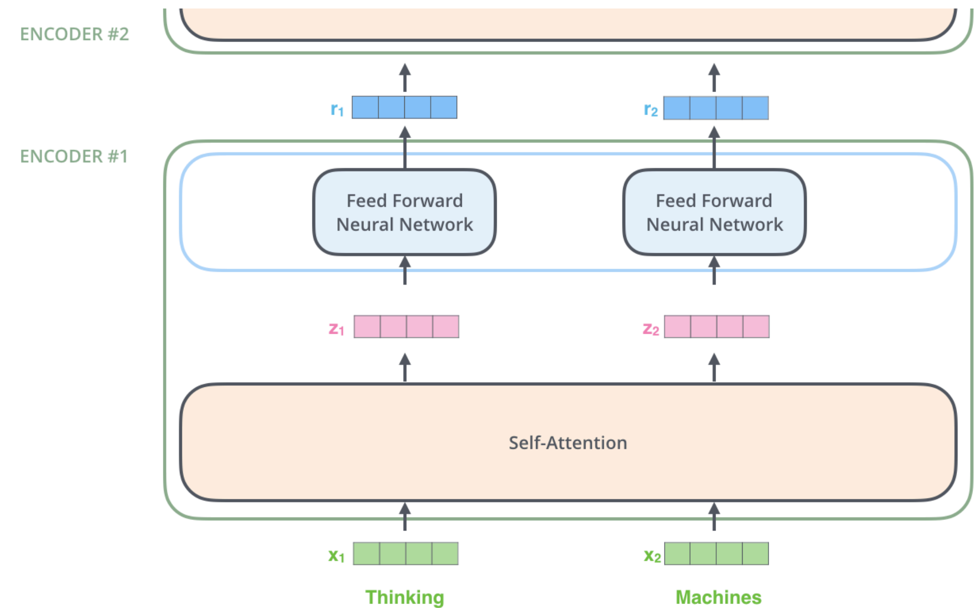
Modern LLMs use many more layers: e.g. Llama 3.3 70B has **80 (decoder) layers**.

Inside an encoder block

source: The Illustrated Transformer

Each encoder block has two sub-layers:

1. **Self-attention layer** → sharing information between the tokens.
2. **Feed-forward (FF) layer** → updating the token information.

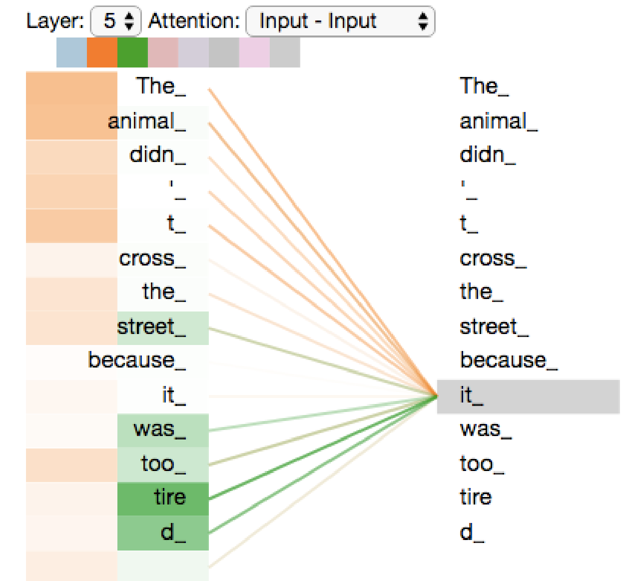


Self-attention

The original attention mechanism was useful for decoding a new sequence.

But what if we only want to **represent a sequence**?

→ We can turn it into **self-attention** (Cheng et al., 2016)



Example

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

When processing "*it*", the model should attend strongly to "*animal*" (not "*street*").

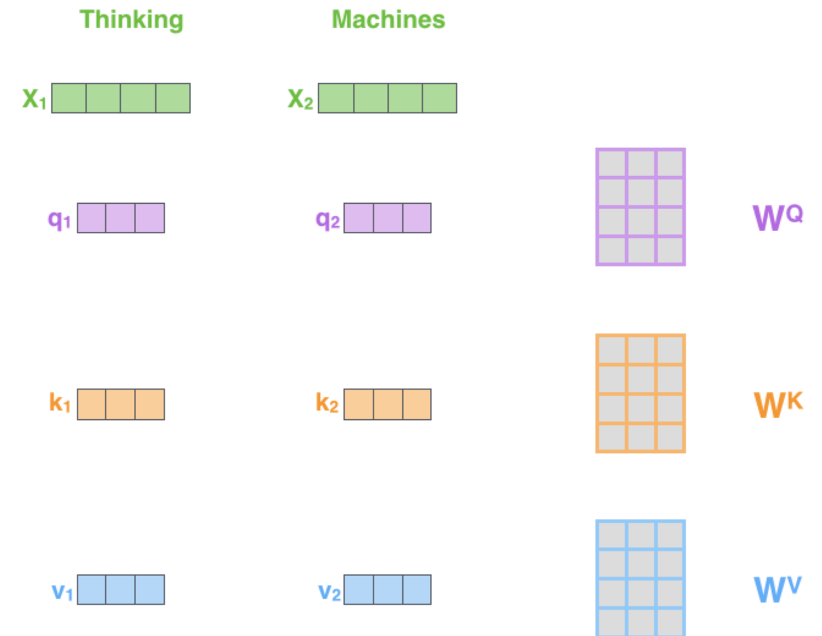
Self-attention – query, key, value

source: The Illustrated Transformer

For each token embedding x_i, we compute three vectors:

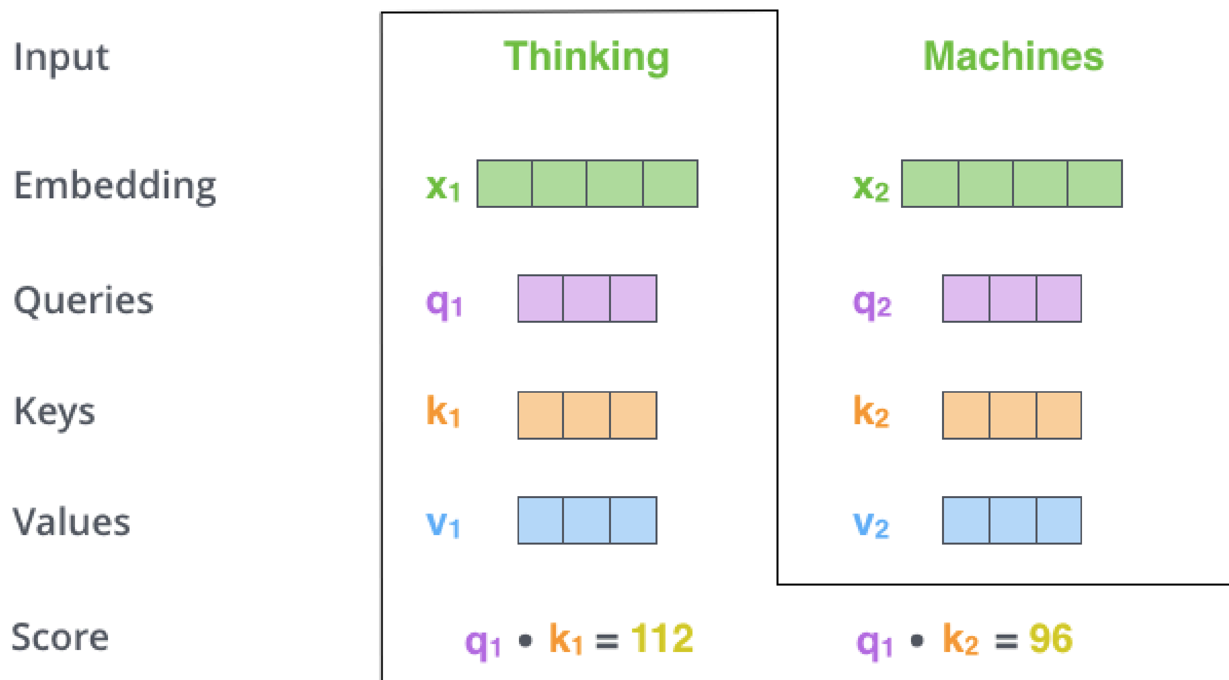
$$\boldsymbol{q}_i = \boldsymbol{W}^Q \boldsymbol{x}_i, \quad \boldsymbol{k}_i = \boldsymbol{W}^K \boldsymbol{x}_i, \quad \boldsymbol{v}_i = \boldsymbol{W}^V \boldsymbol{x}_i$$

- **Query** (q): “what am I looking for?”
- **Key** (k): “what do I contain?”
- **Value** (v): “what information do I provide?”



Self-attention – computation

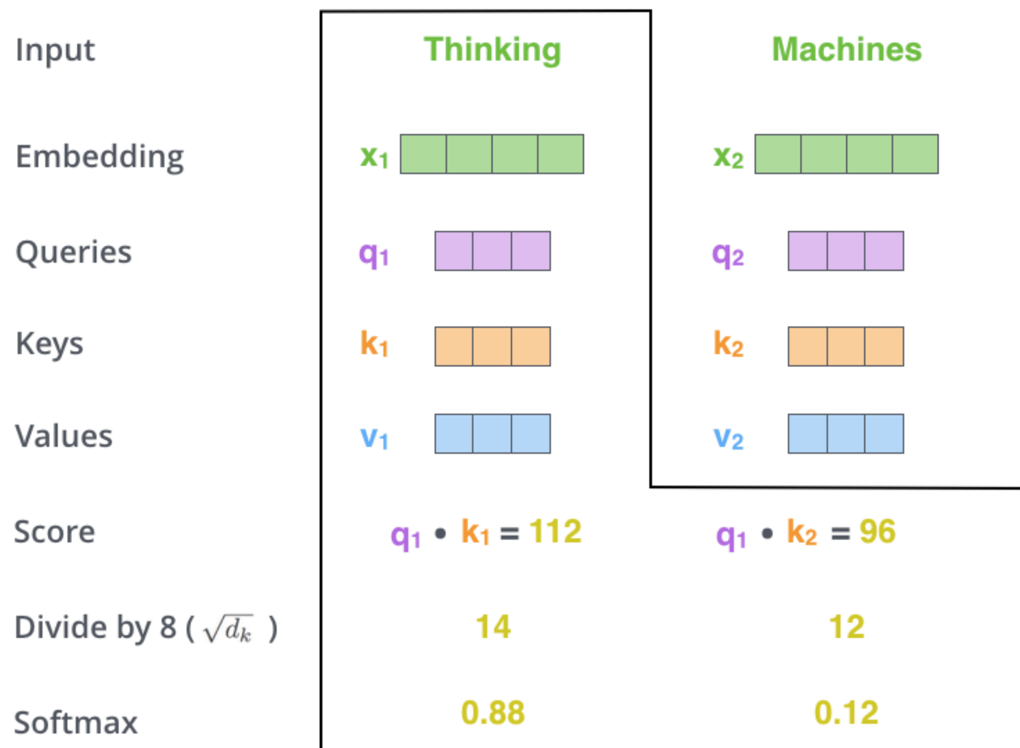
We multiply queries q_i by keys k_i → **raw (non-normalized) attention scores.**



Self-attention – computation

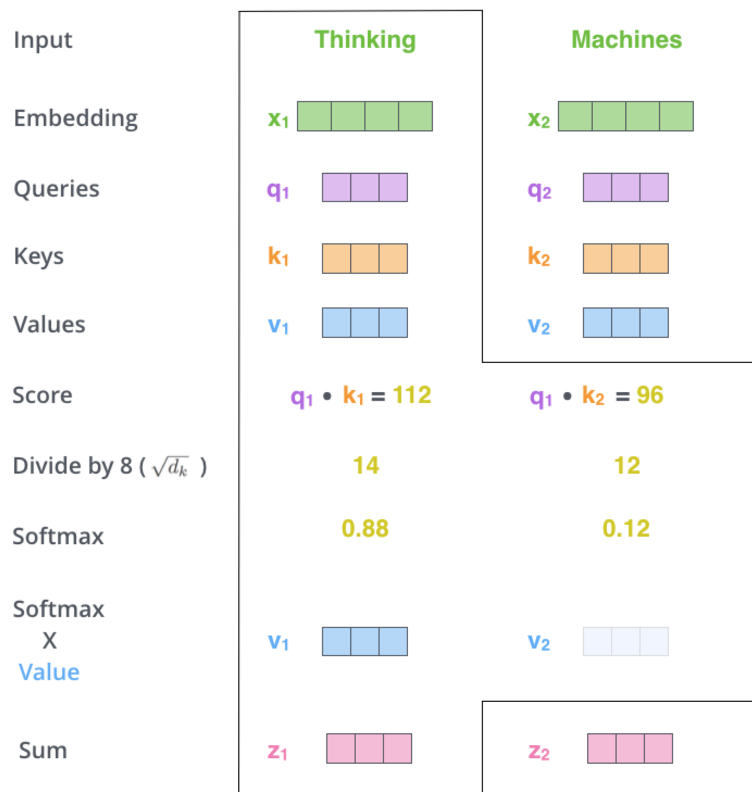
source: The Illustrated Transformer

We normalize the scores: divide by $\sqrt{d_k}$ and apply `softmax` → **attention weights**.



Self-attention – computation

The output for token i is a vector z_i: **weighted mixture** of all value vectors.



Self-attention: Pause and ponder

Question

Is it the only and the best way to compute attention? And do we need all three: queries, keys, and values?

Name	Alignment score function	See
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves et al. (2014)
Additive attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{v}_a^\top \tanh(\boldsymbol{W}_a[\boldsymbol{s}_{t-1}; \boldsymbol{h}_i])$	Bahdanau et al. (2015), Cheng et al. (2016)
Location-Base	$\alpha_{t,i} = \text{softmax}(\boldsymbol{W}_a \boldsymbol{s}_t)$	Luong et al. (2015)
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{W}_a \boldsymbol{h}_i$	Luong et al. (2015)
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong et al. (2015)
Scaled Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$	Vaswani et al. (2017)

source: Lil'Log

Self-attention – computing scores

Compactly, the operation in the attention sublayer can be written out as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V}$$

where $\boldsymbol{Q} \in \mathbb{R}^{n \times d_k}$, $\boldsymbol{K} \in \mathbb{R}^{n \times d_k}$, $\boldsymbol{V} \in \mathbb{R}^{n \times d_v}$.

source: Attention is All You Need Formula T-Shirt

Multi-head attention

source: The Illustrated Transformer

A single attention head captures **one type of relationship**. But words relate to each other in many ways: syntactic, semantic, positional, ...

Idea: multi-head attention

Run **multiple attentions ("heads") in parallel**, each with its own $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V$.

Then concatenate the outputs and project them to the original size.

Self-attention: putting it all together

source: The Illustrated Transformer

1) This is our input sentence*

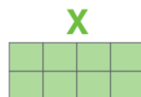
2) We embed each word*

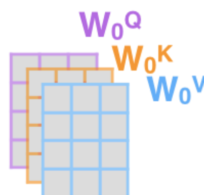
3) Split into 8 heads. We multiply X or R with weight matrices

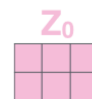
4) Calculate attention using the resulting Q/K/V matrices

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

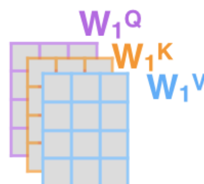
Thinking
Machines

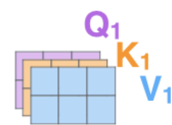


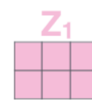




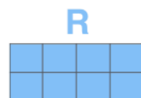
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

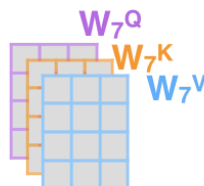


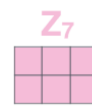


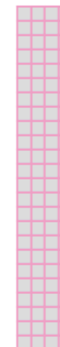


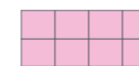
...











Residual connections & layer normalization

source: The Illustrated Transformer

Each sub-layer is wrapped in:

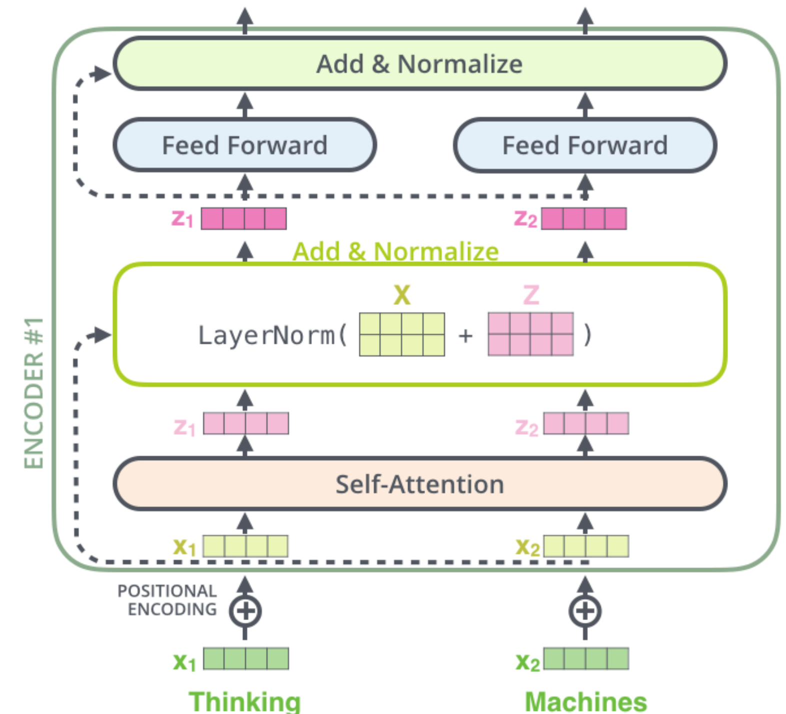
$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

Residual connection ($x + \ldots$):

The input to the sublayer is also passed as a identity (no modification) → the layer only needs to learn the **"delta"** (how to update the input).

Layer normalization:

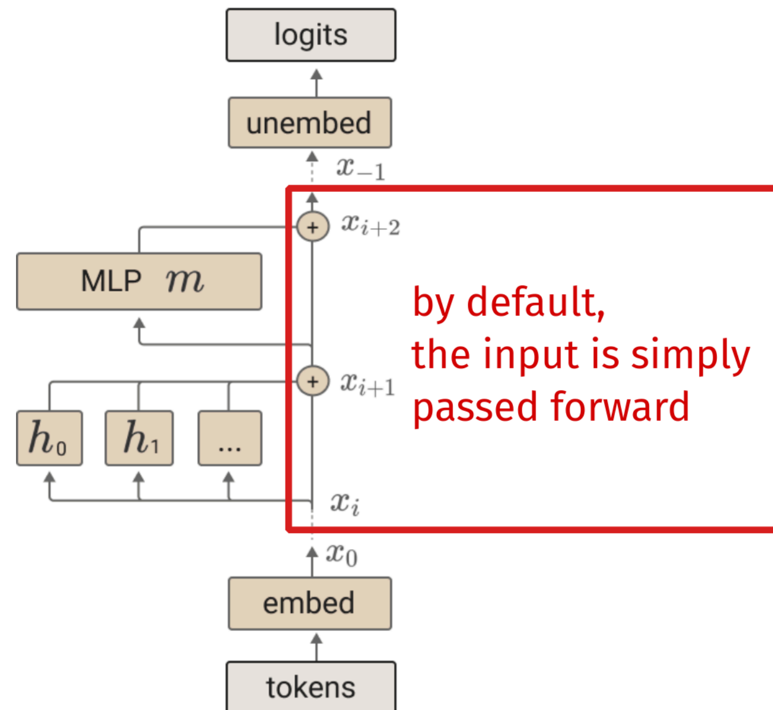
Normalizes values across features → more stable training.



Residual view of the Transformer

source: Transformer Circuits Thread

It may be even useful to think of attention and FF layers as mere **updates** to the “residual stream”:



Feed-forward layer

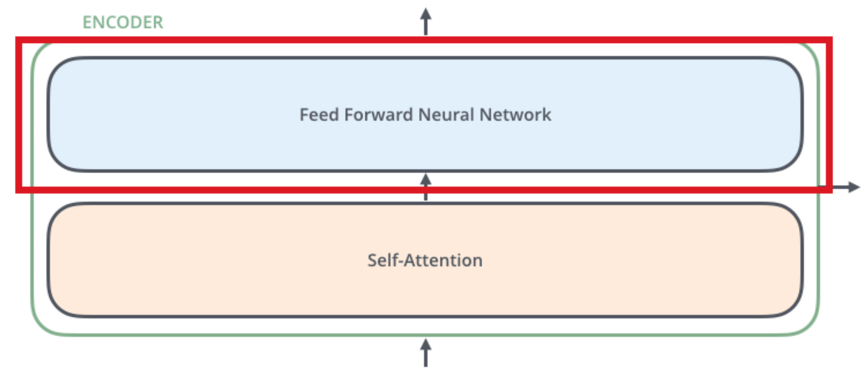
Applied to **each token independently** (same weights, different inputs):

$$\mathrm{FFN}(\boldsymbol{x}) = \boldsymbol{W}_2 \cdot \mathrm{ReLU}(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{d \times d_{\mathrm{ff}}}$, $\boldsymbol{W}_2 \in \mathbb{R}^{d_{\mathrm{ff}} \times d}$.

What happens in the feedforward layer?

- Model's knowledge injected into each token's representation.
- Up-projection → non-linearity → down-projection (given usual d and d_{ff}).



Positional encoding – why do we need it?

Self-attention is **permutation-invariant**: if we shuffle the input tokens, the attention scores would be the same (→ we did not have such an issue with RNNs!)

Why does it matter? The model now has no way to distinguish between sentences such as “dog bites man” and “man bites dog”.

Idea

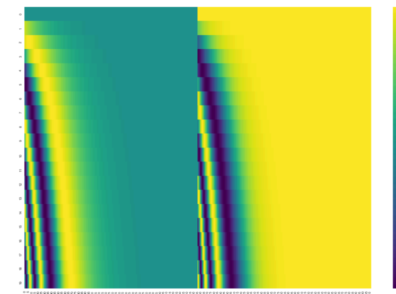
Let’s generate “**positional embeddings**” (PE): vectors that are specific for each position, but content-independent. We then add it with the input embeddings $\boldsymbol{x}_i = \mathrm{Emb}(w_i) + \mathrm{PE}(i)$, baking in the information about the position.

Sinusoidal positional embeddings

source: Vaswani et al. 2017

The original Transformer uses **fixed sinusoidal functions** for the positional embeddings (PE):

$$\mathrm{PE}(p, 2i) = \sin\left(\frac{p}{10000^{2i/d}}\right), \quad \mathrm{PE}(p, 2i+1) = \cos\left(\frac{p}{10000^{2i/d}}\right)$$



where p is the position and i is the dimension index.

PE's and tok. embeddings are added:

Tip

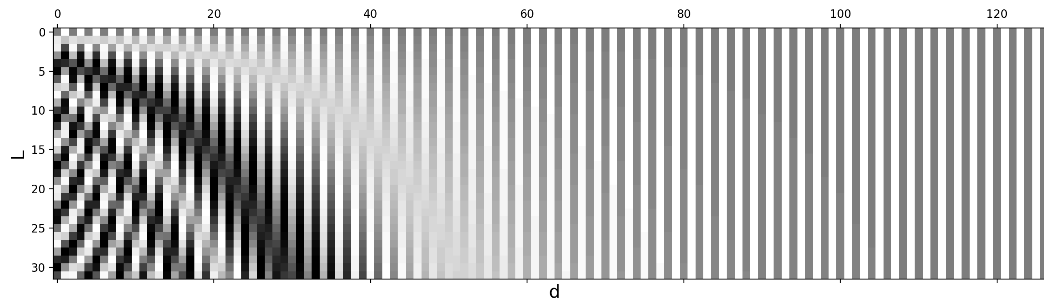
Do not memorize these formulas (they are not used in the modern models, anyway). Try to understand the principles that behind them (next slide)

Sinusoidal positional embeddings – principles

source: Lil'Log

- Each dimension of the PE vector is a **sinusoid**:
  - Different wavelengths in different dimensions: from 2π to $10,000 \cdot 2\pi$.

→ each position gets a **unique encoding**; nearby positions have **similar encodings**.



Question

Can you think of another way to get to the same principles?

The decoder

The decoder has **three sub-layers** per block:

1. **Masked self-attention**

Each token can only attend to previous tokens, future positions masked by $-\infty$. (**why?**)

2. **Encoder-decoder (cross) attention**

Queries come from the decoder, keys and values come from the encoder output.

3. **FF layer**

Same as in the encoder.

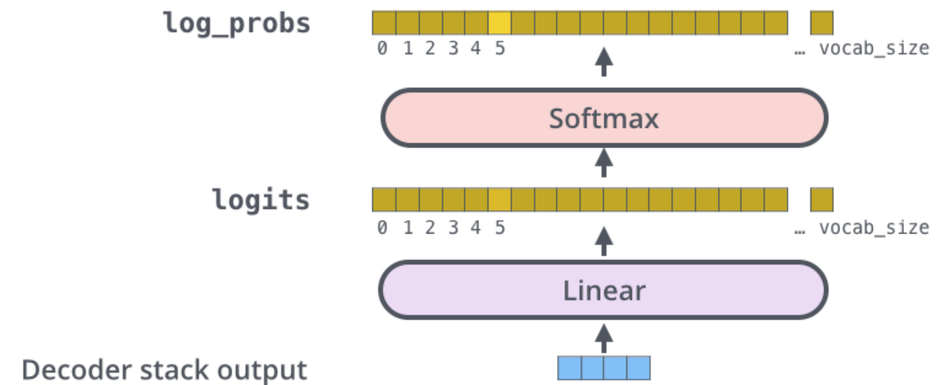
Output layer

source: The Illustrated Transformer

The final decoder output is projected to vocabulary size V and passed through **softmax**:

$$P(w_t \mid w_1, \ldots, w_{t-1}, \boldsymbol{x})$$
$$= \mathrm{softmax}(\boldsymbol{W}_{\text{out}} \boldsymbol{h}_t + \boldsymbol{b})$$

- How to train all these weights? → **Lecture 3**
- How to use the distribution for decoding? → **Lecture 4**



Stitching all pieces together

Slides 90-134: https://cme295.stanford.edu/slides/fall25-cme295-lecture1.pdf

Transformer vs. RNNs

Both architectures can be used for sequence-to-sequence processing.

RNNs

- Sequential processing
  - (-) slow, non-parallelizable
  - (+) implicit position information
- Hidden state is a bottleneck for sharing information.
- Gradient vanishing over long distances.

Transformer

- Parallel processing
  - (+) scalable on GPUs
  - (-) needs positional embeddings
- Token information can be easily cross-shared in the attention layer.
- Stable gradients with residual connections.

Summary

Summary

What we covered today

- **Math refresher**: matrix multiplication, FFNNs, softmax.
- **Tokenization**: subword tokenization with BPE.
- **Attention**: solving the hidden state bottleneck.
- **The Transformer**:
  - Self-attention with Q, K, V → scaled dot-product attention.
  - Multi-head attention → multiple parallel attention heads.
  - Positional encoding → injecting position information.
  - Residual connections + layer normalization.

Links

- The Illustrated Transformer – Jay Alammar
- Attention? Attention! – Lil'Log
- The Annotated Transformer – Harvard NLP
- Attention is all you need – Vaswani et al. (2017)
- Neural machine translation by jointly learning to align and translate – Bahdanau et al. (2015)
- Neural machine translation of rare words with subword units – Sennrich et al. (2016)
- Tokenizer playground – Hugging Face