Intro to deep learning

Dr. Janoś Gabler, University of Bonn

Lecture 11: Transformers

Motivation

Today you finally learn how transformers work and how they revolutionized NLP

Topics

- Problems before transformers
- The essence of transformers
- Smart engineering tricks
- Model architectures
- How should you use transformers

Before transfomers

Problems of RNNs: Short memory

- Hidden state h_t is the only thing that has memory
- $h_t = tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$
- Intuition: Each time step:
  - Adds new information
  - Destroys some old information
- Final state has only vague memory of words that were incorporated a long time ago

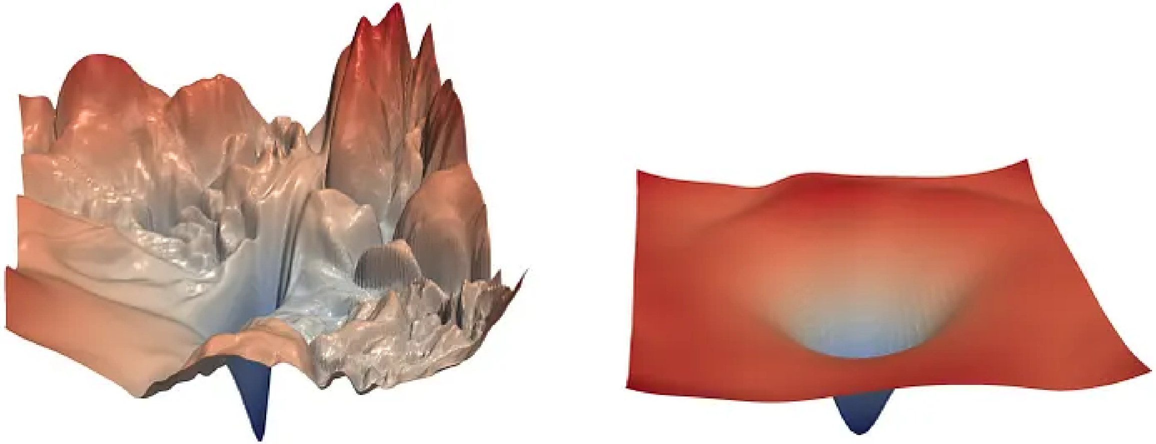
Why do we need long memory?

- Example: "The animal did not cross the street because it was too tired"
- "it" refers to "The animal" but there are 6 words in between
- Similar connections can span multiple sentences!
- Important for machine translation or language modelling!

Problems of RNNs: Vanishing or exploding gradients

- Gradients of RNNs can easily
  - Explode, i.e. become infinity/undefined
  - Vanish, i.e. become zero
- Boths cases are problematic during training
- LSTM formulation improves this but does not completely avoid it

How bad is it?



(a) without skip connections

(b) with skip connections

Problems of RNNs: No parallelism

- Fundamental RNN equations:
  - $h_t = tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$
  - $y_t = W_{hy} \cdot h_t$
- By construction, RNNs are hard to parallelize
  - No parallelism between time-steps
  - Product with W_{hh} and W_{xh} could be done in parallel

How bad is it?

- Training at the scale of GPT would have been completely impossible
- Even GPUS back then could not be used efficiently
- Latest GPUs only can do their magic on parallelizable workloads!

Language Modelling before transformers

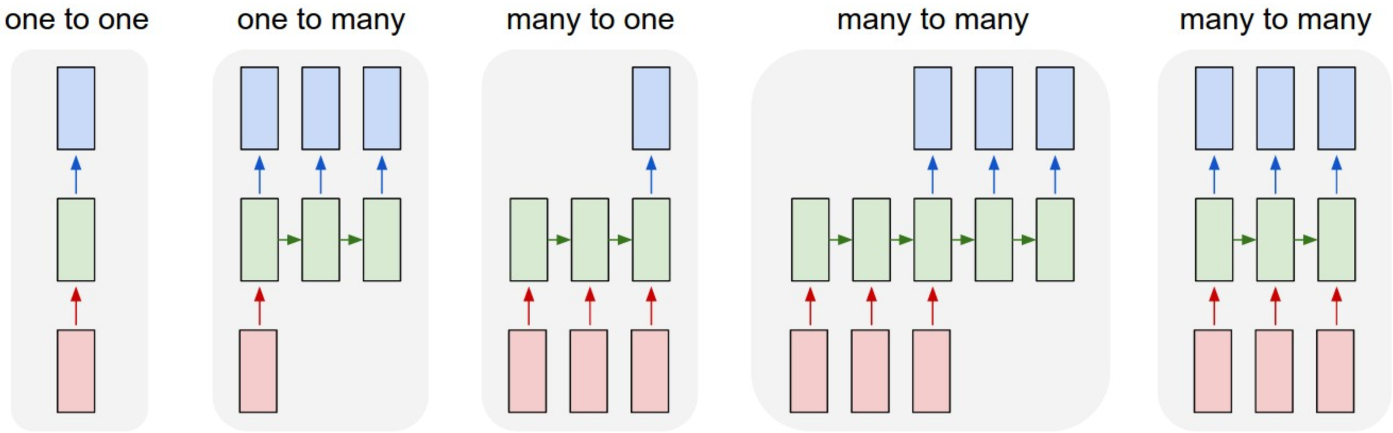
- Only used for text generation
- Trained from scratch for specific topics
- Use LSTM-RNNs
- Most people did not see the enormous potential!
- RNNs were good enough to be state of the art
- Transformers completely replaced them after 2017

Machine translation before transformers

- Hybrid of rule-based and statistical approaches
- Hand-crafted components
  - Alignment models
  - Translation models
  - …
- RNNs were used somewhat successfully since 2015
- Google translate uses them since 2016
- Attention is all you need introduces transformers in 2017

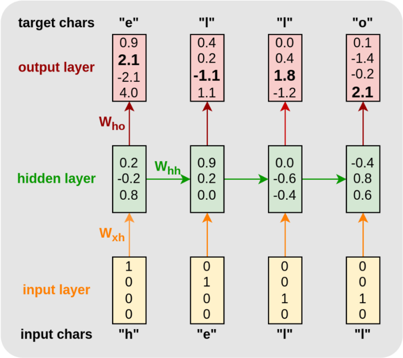
From RNNs to transformers

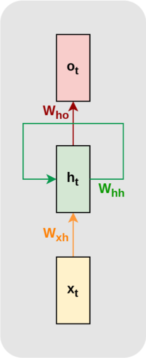
Model interfaces



1. Image classification
2. Image captioning
3. Text classification
4. Machine translation
5. Language modelling

What do we need for language models

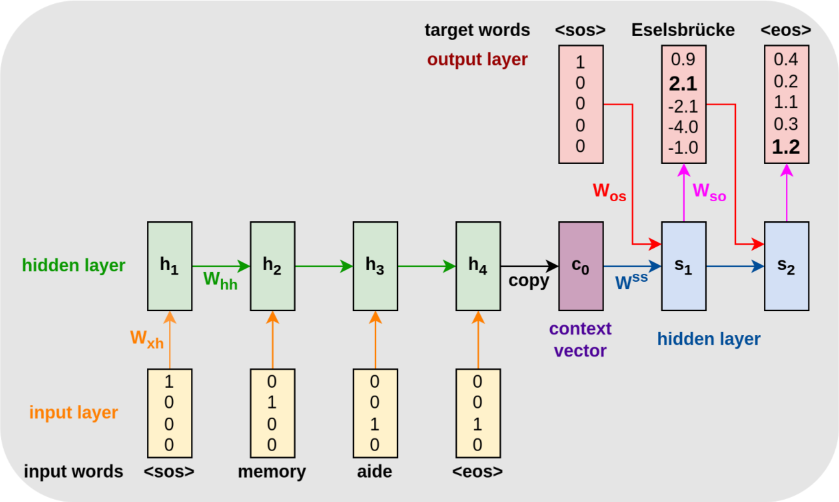




Source: Tobias Stenzel's Blog

- Ultimately need a model that produces a list of hidden states
- One hidden state per output word
- Cannot peak into the future!
- Let's keep embedding and linear output layer + softmax unchanged

What do we need for translation



Source: Tobias Stenzel's Blog

- Need an encoder hidden state, that encodes the entire sentence
- Decoder needs are the same as language modeling needs

The famous transformer graph

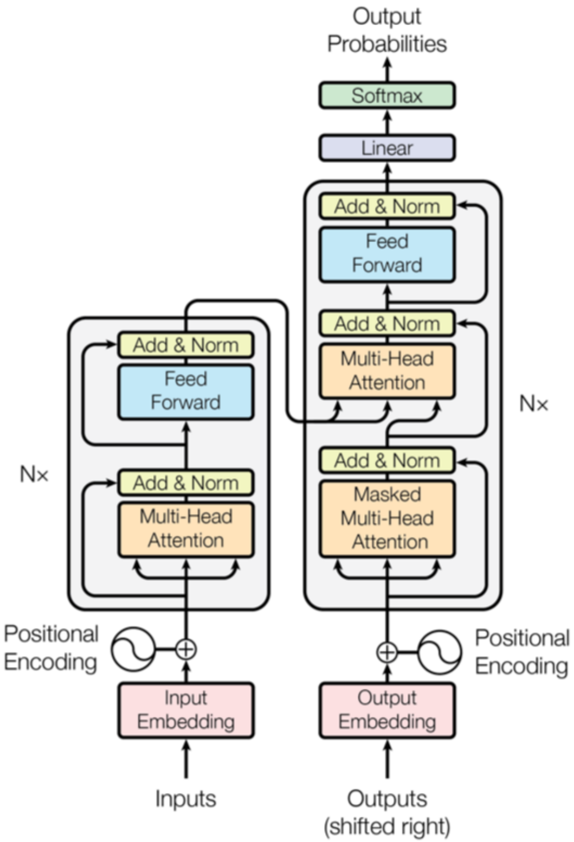


Figure 1: The Transformer - model architecture.

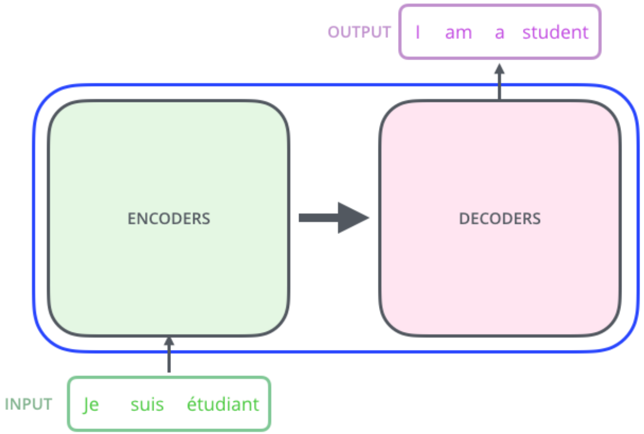
- Left: Encoder
- Right: Decoder
- Main questions:
  - What is (Multi-Head) Attention?
  - What is masked Attention?
  - Wat is the role of the Feed-Forward network?
  - How does this relate to RNNs?

The essence of transformers

The illustrated transformer

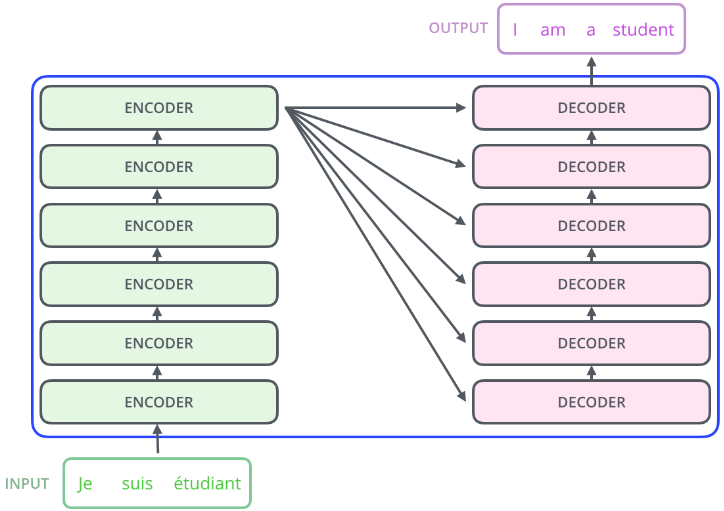
- We do not have the basics to understand the chart from the transformer paper
- Most of the following is from the amazing illustrated transformer blogpost!
- For now, we stick to the absolute essence of transformers
- For even more details, look at the blogpost

High level interface



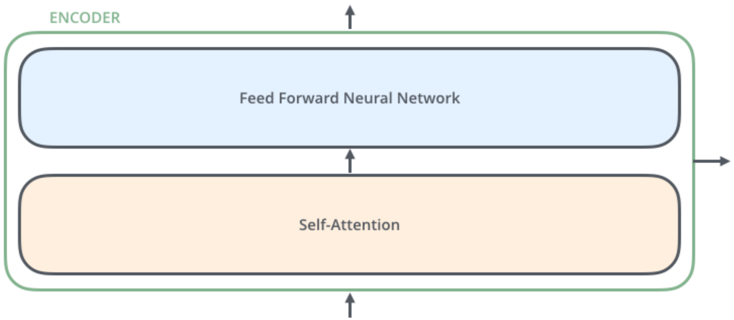
- Transformers can be
  - Encoder-only (BERT-style)
  - Encoder-decoder (Original, Machine translation)
  - Decoder only (GPT-style)
- We will first focus on the encoder

A deep encoder-decoder stack



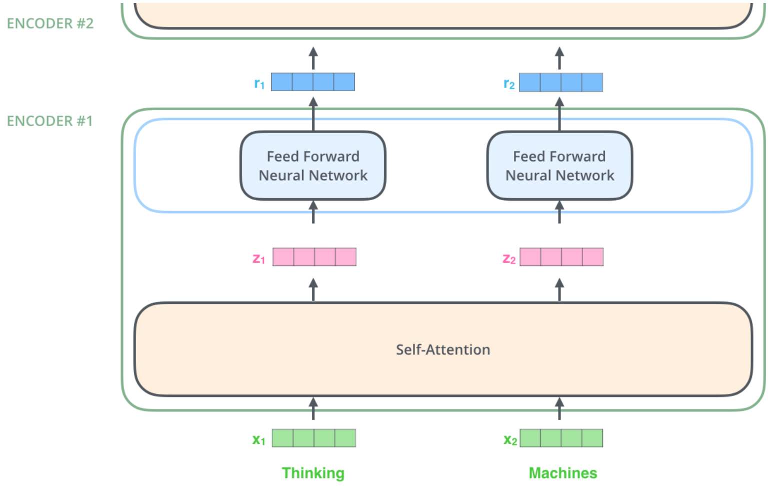
- So far, this could be a deep RNN
- There are multiple stacked encoder cells and decoder cells
- All decoder cells have access to the last encoder hidden state

The steps of a transformer encoder cell



- Remember: Encoders look at the entire sentence!
- **Attention** averages vectors corresponding to different words
- **Feed-Forward Network** transforms vectors and plays the role of a nonlinearity

The interface of an RNN cell

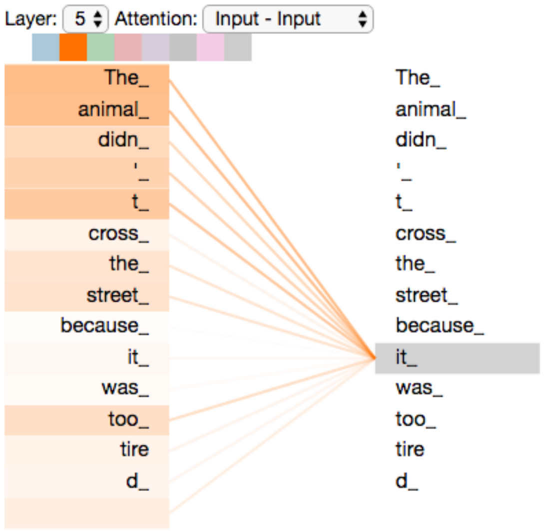


- Takes: One vector per input word
- Returns: One vector per input word (the list of hidden states)
- Typically: All vectors have same length (`n_hidden`)

Summary

- The encoder cell does two things:
  - Calculate different weighted averages of input vectors via attention
  - Transform those averages via a neural network
- All the magic will come from trainable parameters!

How can Attention be so powerful?

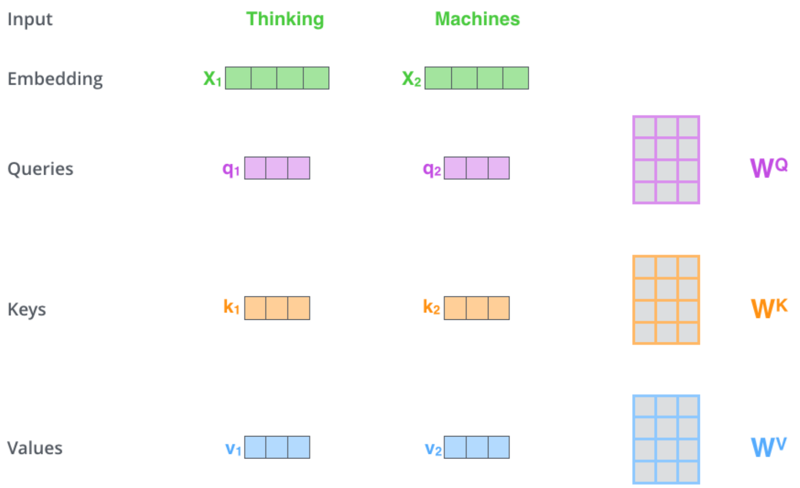


- The hidden representation of "it" will encode that "it" refers to "The animal"
- No problem that there were several words in between
- In fact, word order is irrelevant for attention and we will have to use a trick to encode position information!

Roadmap

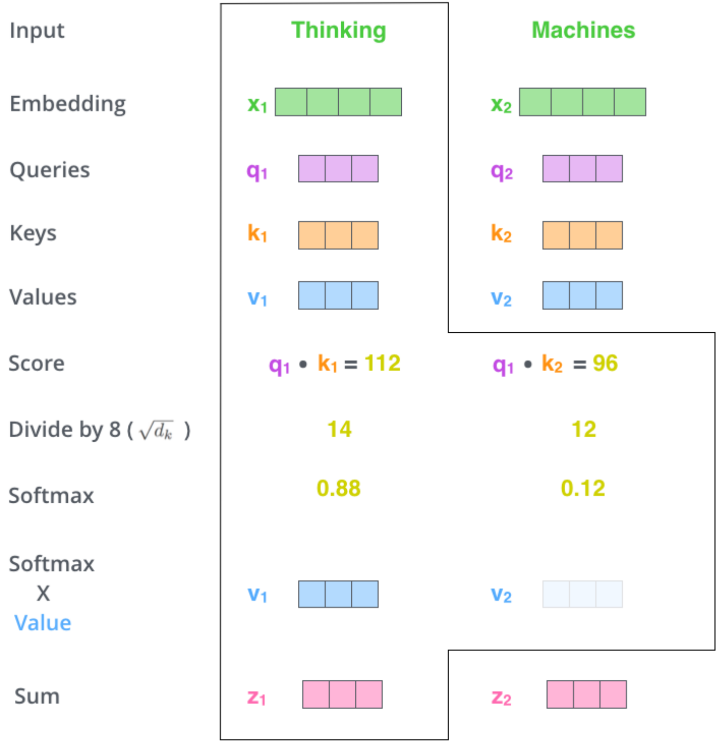
- Goal: Take optimally weighted averages of vectors
- Steps:
  - Calculate query, key and value vectors
  - Use query and key vectors + softmax to calculate weights
  - Take weighted sums of value vectors
- Optimality comes from training the weight matrices

Query, key and value vectors



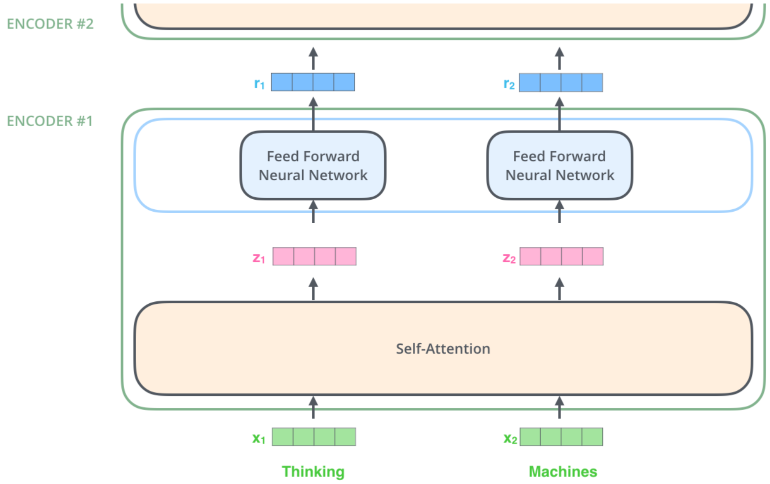
- The matrices W^Q, W_K and W^V are trainable parameters
- the vectors q, k and v are calculated by multiplying the input vectors x by weight matrices
- Example: $q_1 = W^Q \times x_1$
- W^V is a bit similar to W_{xh} in our RNN

Calculating weights



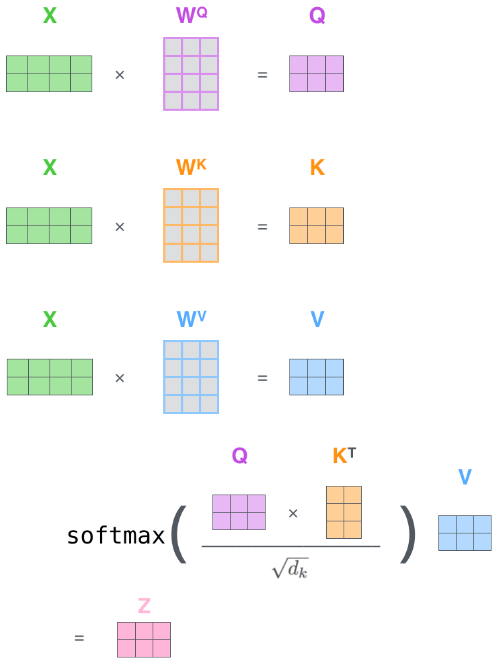
- Calculate attention scorces for output vector 1
  - Use query vector 1
  - Multiply with all key vectors
- Division is a training trick!
- Softmax produces weights that sum to 1
- z_1 will be mostly x_1 but have some x_2 mixed in

Summary



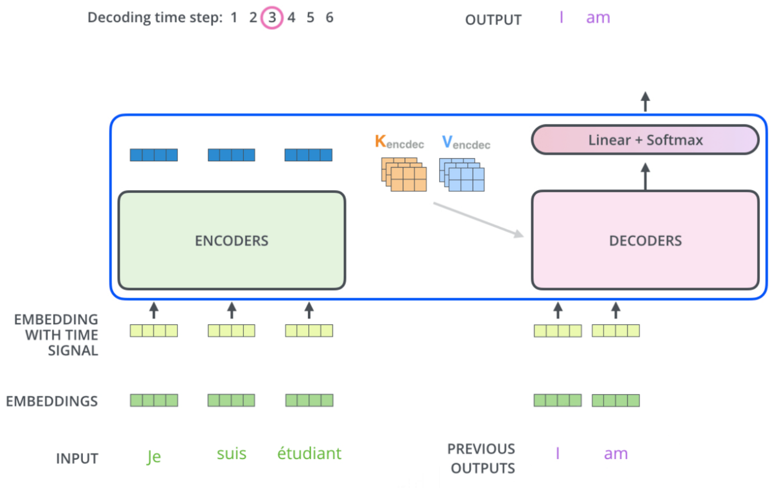
- We now understand how one encoder cell works
  - W^V is used to transform the inputs
  - Attention weights (from queries and keys) are used to average the the transformed inputs
- Feed Forward network plays the role of nonlinearities

Efficient implementation



- Concise way of expressing same calculations as before
- Matrix multiplications are implemented very efficiently on GPUs

What about the decoder



- The output vectors of the last encoder cell are passed to all decoders
- Decoder uses two types of attention
  - Cross attention to encoder states
  - Masked self-attention to already decoded outputs
- Without masking, decoding would be trivial

Transformer vs. RNN

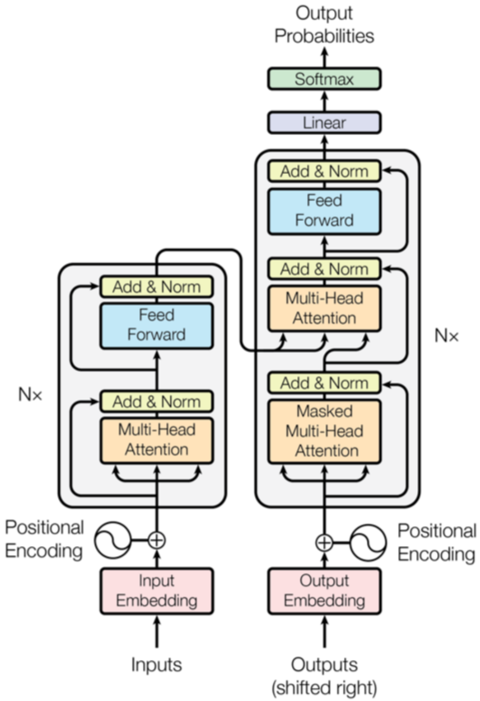
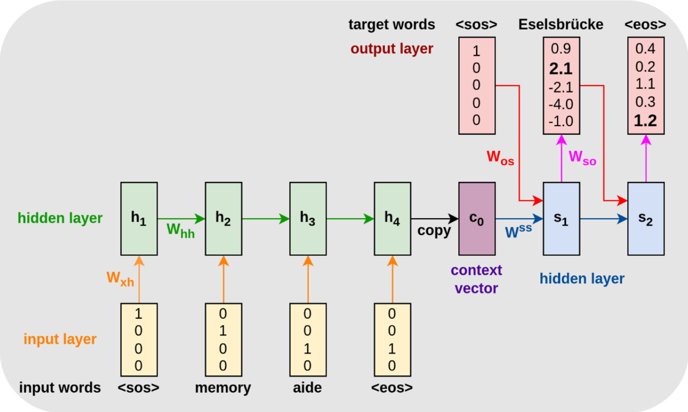


Figure 1: The Transformer - model architecture.



Computational advantages

- There are not time-steps anymore
- All words can be processed independently (in parallel)
- Even decoding can be done in parallel during training!
- No repeated multiplication with $W_h h$ -> better gradients

What we skipped

Multi-Headed attention

- In practice, all transformer models use multi-headed attention
- Repeat attention calculation k times with different sets of weight matrices
- Produces "too many output vectors"
- They are concatenated and projected down to desired shape

Multi-Headed attention

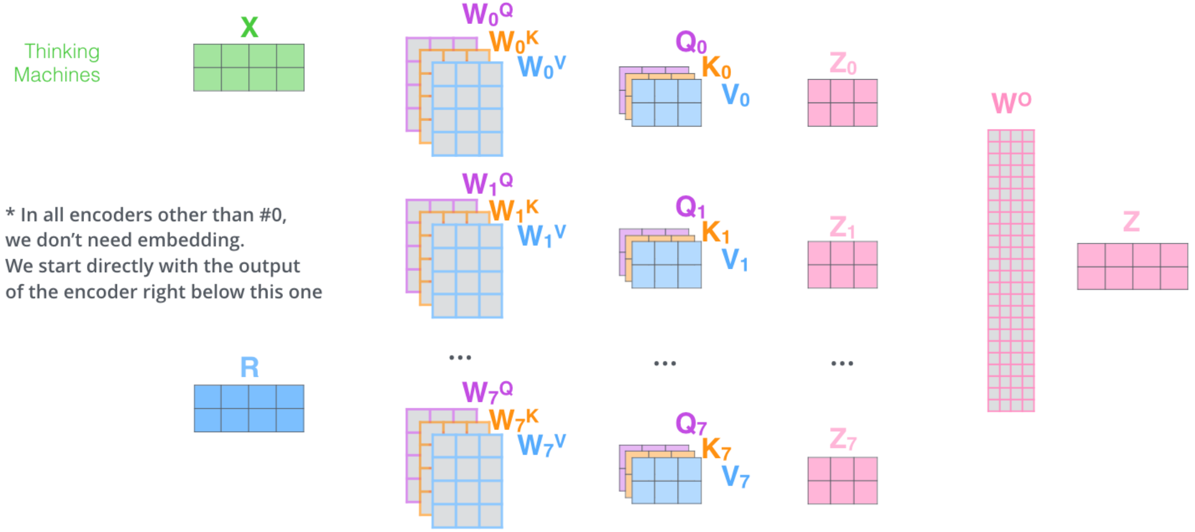
1) This is our input sentence*

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

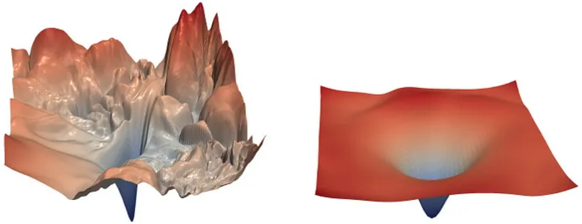


Positional encoding

- Transfomers ignore order of inputs
- However, order is relevant for meaning of language
- Need separate positional encoding to make transfomers aware of order
- Solution: Add position embedding to word embedding!
- Different methods to produce position embeddings, see blogpost

Gradient problems

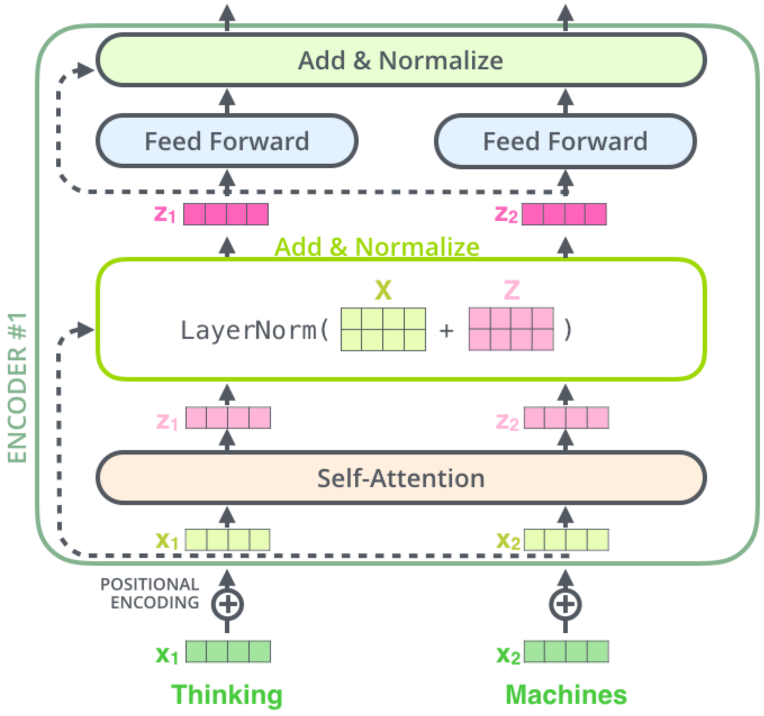
- Transformers can still have vanishing or exploding gradients
- To avoid this, they use residual connections and layer norm



(a) without skip connections

(b) with skip connections

Residual connections and layer norm



- Remember, a transformer cell takes x_i and produces z_i
- Transformer cells with residual connections instead return $\tilde{z}_i = x_i + z_i$

Recap: The full transformer

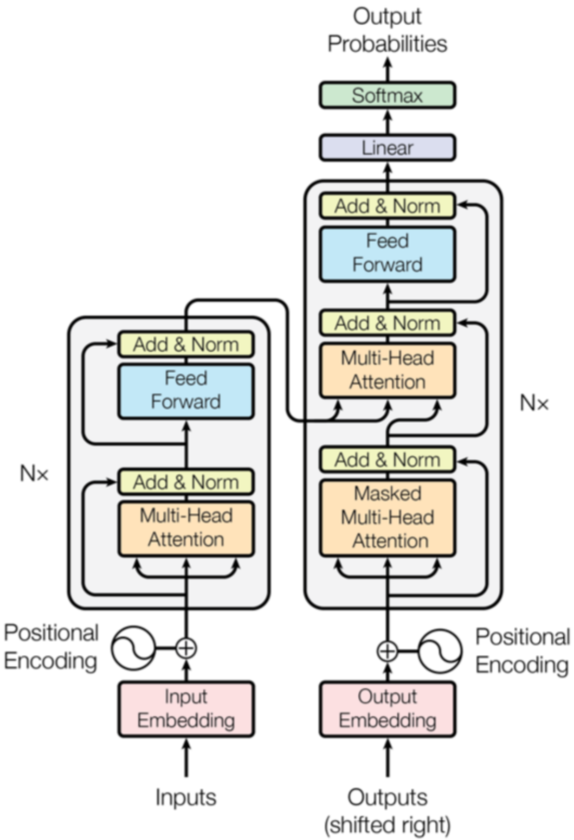
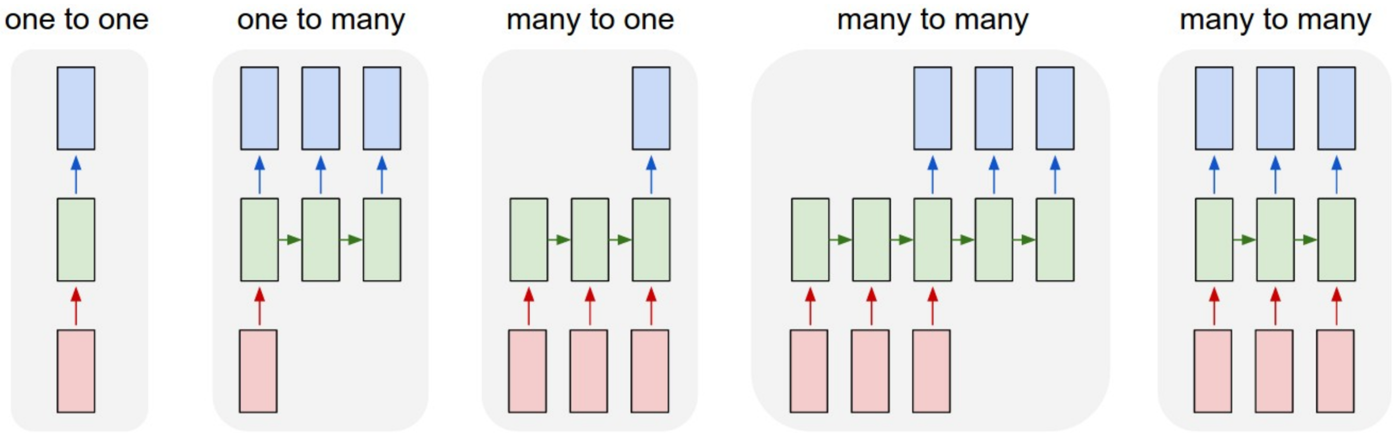


Figure 1: The Transformer - model architecture.

- We now know all components of the transformer
- You have intuition how and why attention works
- You also know many of the engineering tricks
- Ultimately, the transformer is used because it is successful in practice and the training tricks are as important as the basic architecture

Model architectures

Tasks and architectures



3. Text classification: Encoder only
4. Machine translation: Encoder-decoder
5. Language modelling: Decoder only

Encoder: BERT

Encoder-Decoder: T5

Decoder: GPT2

Wrapping up

How to use transformers

- Giant Models (would not run on your computer):
    - APIs like OpenAi
- Smaller models
    - Find pre-trained model on huggingface
    - Fine-tune it on a GPU if necessary
    - Run the fine-tuned model on your computer
- Do not attempt to pre-train a large language model from scratch!

Outlook

- Guest lecture on webscraping/crawling
- Deadline for final project topics