Transformers

Arto Klami

April 4, 2025

Where are we?

- CNNs and RNNs as architectures that can solve more interesting problems than fully connected networks
- Parameter sharing and tying as the practical means of reducing the number of parameters and encoding desired properties
- Alternatively: Ways of encoding inductive biases (equivariance, memory for sequential processing, ...) into the architecture itself
- Already hinted that transformer models have largely overtaken both CNNs and RNNs as the architecture of choice for very large problems
- Today: The basic transformer architecture, focusing especially on the self-attention mechanism

Transformers

Similar to how CNNs are networks built from blocks of convolutions replacing fully connected layers, we have:

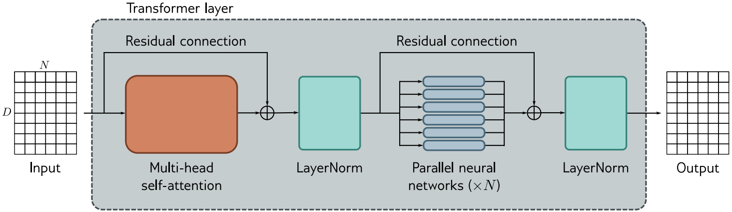
- *Transformers* are a family of neural network architectures
- The core component is the *self-attention operation* that replaces a fully connected layer
- The full architecture uses the self-attention in a block structure, where the block includes also other standard components (fully connected layers, residual connections)

Transformers

- Transformers are often used for sequential data, but actually are models for sets of inputs; we need additional tricks to account for sequences
- The name *attention* captures the intuition well: The layer learns how much we should pay attention to each of the inputs
- ...but *self-attention* has a bit of a legacy, referring to the concept of attention as it was used in some specific RNN models first
- Many modern and highly impressive models are surprisingly simple transformers: BERT, GPT3, AlphaFold2, ...
- Consequently: After this lecture, you could implement any of them!

Setup

- The self-attention operation and the basic transformer block take as input a set of N vectors $\mathbf{x}_i$ of D dimensions
- The output has the exact same shape and interpretation: We have N output vectors $\mathbf{y}_i$ of D dimensions, each corresponding to one input (a bit like in convolution)
- In typical applications both N and D are large; e.g. $D = 1024$ and N is hundreds or thousands (and much larger in recent language models)



See: Prince, Chapter 12

Setup: Alternatives

Fully connected layer for mapping from $N \times D$ input to output of the same size

- Can easily consider inputs from arbitrarily far away
- Permuting the inputs changes everything, so definitely not modelling the inputs as a set
- Needs $D^2 N^2$ parameters, in the order of 10^{12} for typical choices $\Rightarrow$ completely unreasonable

Setup: Alternatives

Convolutional layer for mapping from $N \times D$ input to output of the same size

- Can only consider local neighborhoods; the output for $\mathbf{y}_i$ only depends on inputs within the kernel width as $\mathbf{y}_i = \sum_{j=-S}^{S} \mathbf{w}(i+j)\mathbf{x}_{i+j}$
- To connect inputs further away we need to stack multiple layers, possibly very many
- Only makes sense for ordered inputs
- We could do 1×1 convolution, but it processes each input completely independently

Setup: Alternatives

Recurrent layer mapping from $N \times D$ input to output of the same size

- Again only makes sense for ordered inputs
- Efficient parameterization and already a single layer can in principle remember inputs infinitely far away
- LSTM/GRU help in remembering the history, but need to spend 'more effort' for keeping information far away in the sequence relevant: The further they are, the more computational operations we have between them
- In practice struggle with long sequences
- Many practical sequence-to-sequence RNNs (e.g. in machine translation) started including an *attention* mechanism where the hidden states of the RNN are kept in memory and the output part of the network can select which ones it uses

Goal

- We want a layer that can efficiently process reasonably large sets of inputs (N is hundreds or thousands)
- It should be able to make long-term connections between different inputs, ideally as easily as short-term connections
- The key conceptual idea is that of *self-attention*: We want to explicitly model which inputs are relevant for determining a particular output
- This is done internally in the model, rather than building a dedicated attention mechanism on top of a RNN
- The attention is in practice implemented using concepts from *information retrieval*: We *search* for inputs that are relevant for this output

Background: Information retrieval

How old-fashioned search engines work:

- The search phrase is converted to a *query* vector $\mathbf{q}$, (e.g.) a vector that counts how often each word appears in the search phrase
- Each document in described by a *key* vector $\mathbf{k}_i$, with the same feature space as the query (so counting the words in the document)
- The relevance of each document is computed as suitably normalized inner product of the two: $r_i = \mathbf{q}^T \mathbf{k}_i$
- For the best ranked documents we return the *value* $\mathbf{v}_i$, for instance the url or the full content

There is no notion of order in the collection of documents, but rather it is a set

Self-attention as information retrieval

- For computing the output $\mathbf{y}_i$ we use the corresponding input $\mathbf{x}_i$ as a *query* vector to find relevant information for that index
- As the *keys* we again use the inputs, computing $r_{ij} = \mathbf{x}_i^T \mathbf{x}_j$ as the relevance of the jth input for the ith output
- The output is formed by summing over the *values* for the different inputs, weighting the inputs based on the relevances

$$\mathbf{y}_i = \sum_{j=1}^{N} \rho_{ij} \mathbf{v}_j$$

  where the weights are simply the relevances normalized to sum to one with softmax

$$a_{ij} = \mathrm{Softmax}(r_{ij}) = \frac{e^{r_{ij}}}{\sum_k e^{r_{ik}}}$$

- There are no parameters at this point; a_{ij} is a deterministic function of the inputs

Self-attention as information retrieval

- The output is formed by summing over the *values* for the different inputs, weighting the inputs based on the relevances

$$\mathbf{y}_i = \sum_{j=1}^{N} a_{ij}\mathbf{v}_j$$

- But: What are the values $\mathbf{v}_j$?
- They need to relate to the specific input, but simply passing the input itself as the value does not sound very interesting
- Let's model the values as a linear transformation of the input instead

$$\mathbf{v}_j = \mathbf{W}_\mathbf{v}\mathbf{x}_j$$

- Now $\mathbf{W}_\mathbf{v} \in \mathbb{R}^{D\times D}$ is a parameter of the layer
- It is a dense matrix but shared for all inputs and hence small (a fully connected layer would have N^2D^2 parameters)

Self-attention as information retrieval

- This would (kind of) already work as a self-attention layer, but we can easily go further
- We used the inputs $\mathbf{x}_i$ directly as both queries and inputs, and hence we e.g. always have high self-similarity because $\mathbf{x}_i^T \mathbf{x}_i$ is the norm of the input
- For the values we used a linear transformation of the inputs instead
- Nothing stops us doing the same for the queries and keys:

$$
\begin{aligned}
\mathbf{q}_i &= \mathbf{W}_q \mathbf{x}_i \\
\mathbf{k}_i &= \mathbf{W}_k \mathbf{x}_i \\
r_{ij} &= \mathbf{q}_i^T \mathbf{k}_j = (\mathbf{W}_q \mathbf{x}_i)^T (\mathbf{W}_k \mathbf{x}_j) = \mathbf{x}_i^T (\mathbf{W}_q^T \mathbf{W}_k) \mathbf{x}_j
\end{aligned}
$$

where $\mathbf{W}_q, \mathbf{W}_k \in \mathbb{R}^{D \times D}$ are again parameters of the layer

Self-attention layer

- In matrix form we have (omitting biases for clarity)

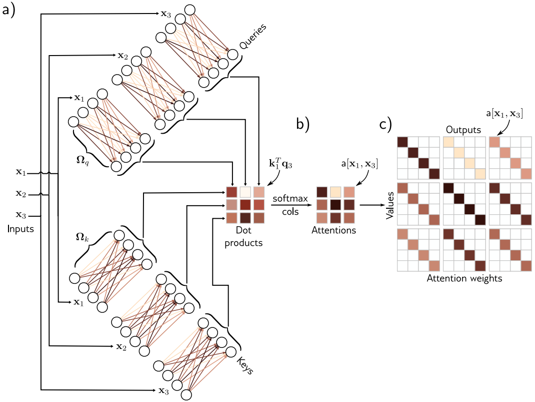
$$\mathbf{Q} = \mathbf{W}_q\mathbf{X}$$
$$\mathbf{K} = \mathbf{W}_k\mathbf{X}$$
$$\mathbf{V} = \mathbf{W}_v\mathbf{X}$$

- The whole computation is given by

$$\mathbf{Y} = \mathbf{V} \cdot \text{Softmax}\left(\frac{\mathbf{Q}^T\mathbf{K}}{\sqrt{D}}\right)$$

  where the division by $\sqrt{D}$ scales the dot-product (this is known to help in practice) and some broadcasting is needed to make the product correct
- All terms are functions of the inputs $\mathbf{X}$ but we usually do not write it explicitly

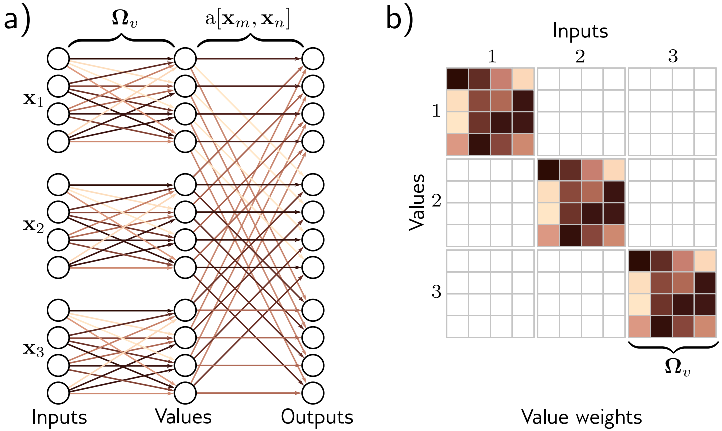
Computing the attention weights



Source: Prince (2023) CC-BY-NC-ND

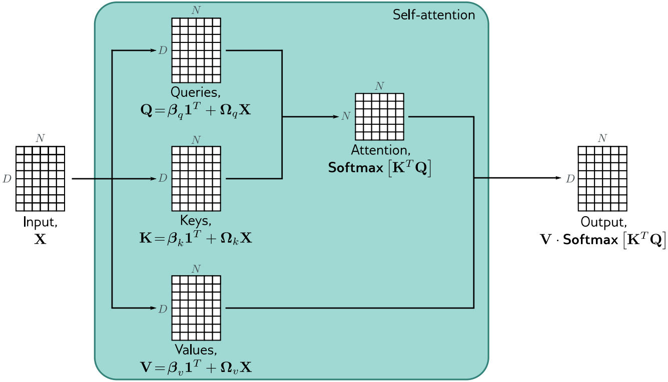
Self-attention layer

Computing the values (Notation $\mathbf{\Omega} = \mathbf{W}$)



Source: Prince (2023) CC-BY-NC-ND

Full computation in matrix form



Multi-head attention

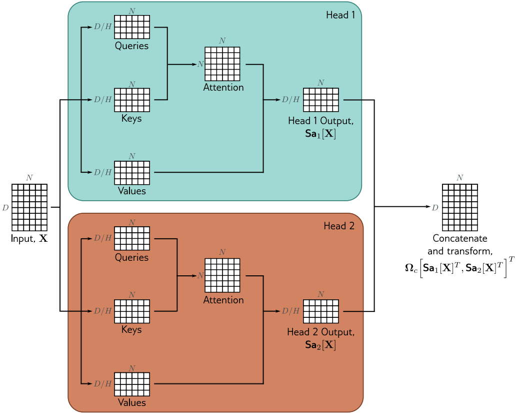
- Similar to using multiple filters in a convolutional layer, we can (and should) use multiple self-attention operations in one layer
- Called *multi-head attention*, and helps simultaneously learning alternative attention mechanisms for complementary needs
- Simply introduce M different parameter matrices $\mathbf{W}_v^m$, $\mathbf{W}_q^m$ and $\mathbf{W}_k^m$
- For each, compute separately the attention weights a_{ij}^m and then the output $\mathbf{Y}^m$
- Concatenate the outputs (increases dimensionality by a factor of M) $\mathbf{Y} = [\mathbf{Y}^1; \ldots; \mathbf{Y}^M]$
- Pull back to original dimensionality with $\mathbf{BY}$, where $\mathbf{B} \in \mathbb{R}^{D \times MD}$

Multi-head attention

- Computation and number of parameters scales with M
- In practice often implemented so that the total cost remains constant
- Instead of $\mathbf{W}_q^m \in \mathbb{R}^{D \times D}$ we use $\mathbf{W}_q^m \in \mathbb{R}^{D/M \times D}$ (and likewise for keys and values), so that the outputs for individual heads are smaller by a factor of M
- Now their concatenation has the same dimensionality as a single head would have
- We still include the final transformation $\mathbf{BY}$ to combine them more flexibly, but it no longer changes the dimensionality
- If $D = 1024$ we can use e.g. $M = 4$ so that the individual heads still operate in 256-dimensional space

Multi-head attention

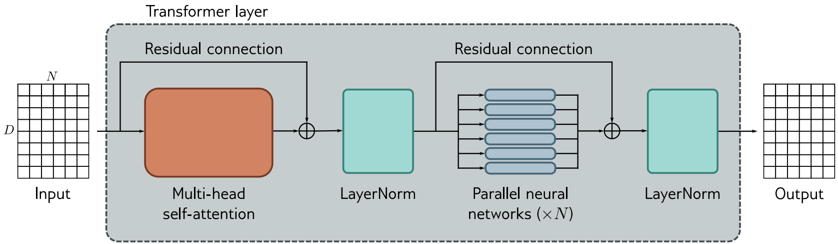
Full computation in matrix form



Source: Prince (2023) CC-BY-NC-ND

Transformer block

- A typical block combines multi-head attention with a residual connection, layer normalization, and fully connected network processing each input independently
- Why exactly this? Quite significant engineering effort in finding something that works well, and several alternatives have been proposed



Source: Prince (2023) CC-BY-NC-ND

Beyond sets: Encoding position

- The transformer block takes as input a *set*, which means it pays no attention to the order of the inputs. Permuting the inputs permutes the outputs in identical way
- However: Transformers were largely motivated by limitations of RNNs and the most famous examples are for processing text (BERT, GPT, ...)
- We can process sequences by *explicitly* encoding the position of the inputs
- Both absolute position ('this input is 5th') and relative position ('this input is 3 indices before that') can be encoded

Positional encoding

- For each input $\mathbf{x}_i$ we introduce another vector $\mathbf{p}_i \in \mathbb{R}^D$ that describes its position
- Instead of $\mathbf{x}_i$ we use $\mathbf{x}_i + \mathbf{p}_i$ as input for keys and queries (and perhaps values as well)
- The unnormalized attention is then

$$
\begin{aligned}
r_{ij} &= \mathbf{q}_i^T \mathbf{k}_j = (\mathbf{W}_q(\mathbf{x}_i + \mathbf{p}_i))^T (\mathbf{W}_k(\mathbf{x}_j + \mathbf{p}_j)) \\
&= \mathbf{x}_i^T (\mathbf{W}_q^T \mathbf{W}_k)\mathbf{x}_j + \mathbf{x}_i^T (\mathbf{W}_q^T \mathbf{W}_k)\mathbf{p}_j + \mathbf{p}_i^T (\mathbf{W}_q^T \mathbf{W}_k)\mathbf{x}_j + \mathbf{p}_i^T (\mathbf{W}_q^T \mathbf{W}_k)\mathbf{p}_j
\end{aligned}
$$

- Encodes both general properties about the positions (the last term) and some sort of interactions between the inputs and the positional embeddings (middle terms)

Positional encoding

- We need to have $\mathbf{p}_i \neq \mathbf{p}_j$ for $i \neq j$ to determine the positions uniquely
- It is nice if we somehow retain similarity: $\mathbf{p}_i$ should be closer to $\mathbf{p}_j$ than $\mathbf{p}_k$ if $|i - j| < |i - k|$
- The embeddings can be pre-determined or learnt
- Example: The original transformer paper used

$$\mathbf{P}_{i,2j} = \sin\left(\frac{i}{c^{2j/D}}\right)$$
$$\mathbf{P}_{i,2j+1} = \cos\left(\frac{i}{c^{2j/D}}\right)$$

where $c = 10,000$

Transformer models

Transformers are frequently used for

- Language processing
- Image processing
- Time series
- ...and quite a bit other tasks as well

Training:

- Typically trained on large data, either in supervised or self-supervised manner (more about that in next lecture)
- Supervised fine-tuning to work better on specific tasks, as form of transfer learning

See: Exercise problem

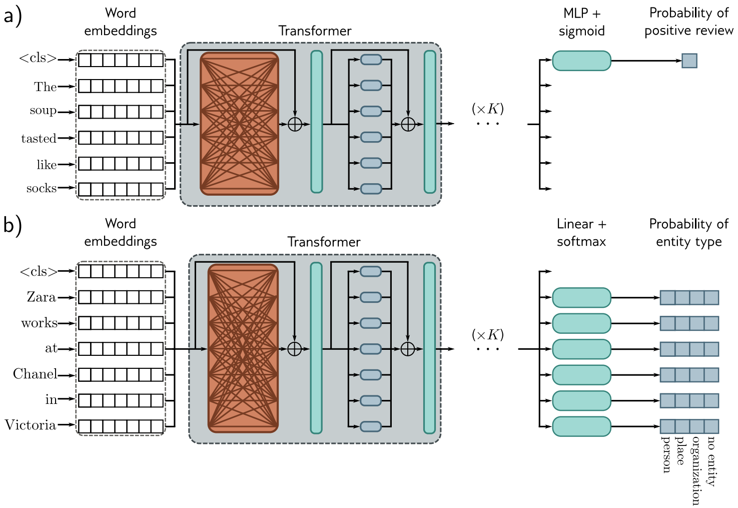
Processing language

- Transformers (and RNNs) for language start by *tokenizing* the text, splitting it into finite number of *tokens*
- The tokens could be characters, words, or (typically) fragments of words that offer a nice balance between vocabulary size and meaning
- Each input $\mathbf{x}_i$ is a D-dimensional *embedding* of the corresponding token
- Could be precomputed or learned together with the model
- We feed fixed-length sequence of the word embeddings as input for the transformer block
    - Too short passage? Fill with zeroes to just ignore the missing inputs
    - Too long passage? Truncate

Examples: BERT

- BERT is so-called *encoder* model: It takes as input a sequence and finds a representation for it, to be used for solving some supervised learning task
- Vocabulary of $30,000$ tokens, $D = 1024$, input length $N = 512$
- 24 transformer blocks with 16 heads (of size 64) each, with 4096-dimensional latent representation in the end
- Total of 340 million parameters, which was a lot in 2018 when it was released
- One extra input token `<cls>` for storing information for the whole sequence, with a learnable input. The output selects which inputs it needs to attend to.

Examples: BERT

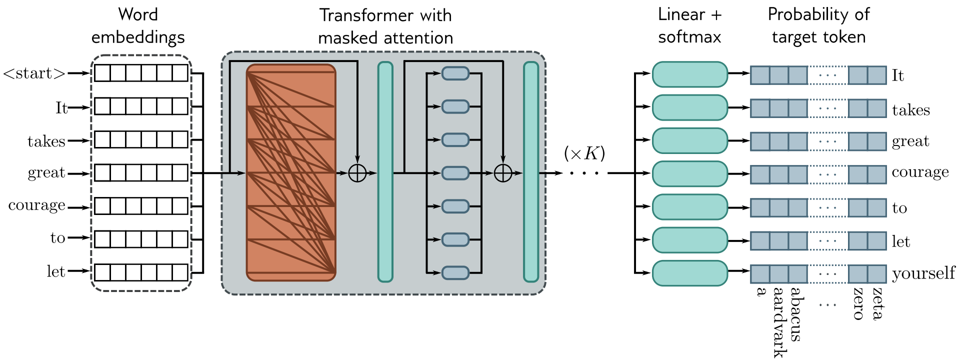


Source: Prince (2023) CC-BY-NC-ND

Examples: GPT3

- GPT is so-called *decoder* model: It predicts continuation of the sequence
- $p(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_1, \mathbf{x}_2) \dots p(\mathbf{x}_N|\mathbf{x}_1, \dots, \mathbf{x}_{N-1})$
- Input as in BERT: Embedding for each token
- We need to change how the self-attention layer works, by killing connections to 'future' tokens: When computing $\mathbf{y}_i$ we can only sum over the previous indices: $r_{ij} = -\infty$ for $j \geq 1$ so that $a_{ij} = 0$ for them
- $N = 2048$ tokens, 96 transformer layers etc, for a total of 175 billion (!) parameters trained on 300 billion (!) tokens

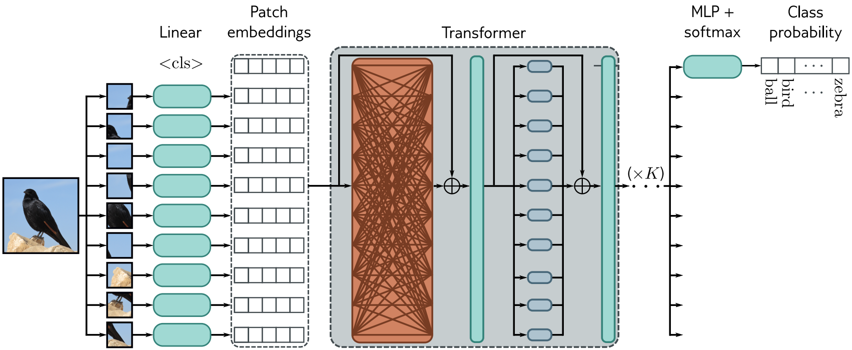
Examples: GPT3



Examples: Vision Transformer (ViT)

- For images we do not have clear tokens
- Instead, we use local image patches
- Each patch mapped into an embedding by a learnable transformation, and the embeddings are processed by standard self-attention
- Positional encoding in two dimensions, to tell about relationships between patches
- The first ViT paper from 2021 introduced three models of different sizes:
  - 16×16 grid of patches
  - $D = 768$ to $D = 1280$
  - 12-32 standard transformer layers of 12-16 heads, for 86-632M parameters
  - Supervised training on 303 million labeled examples, with 18,000 classes

Examples: Vision Transformer



Summary

- Transformers are models that use the self-attention operation as a processing layer
- Usually combined with residual connection and normalization + fully-connected layer
- Self-attention processes a set of inputs and gives output for each, so that the output can depend on arbitrary subset of the inputs
- For sequences we need to encode the position explicitly
- A deep stack of transformer blocks is in itself extremely capable, effectively state-of-the-art in both language and image tasks once trained on massive data collections
- BERT, GPT3, ViT and many other modern models are surprisingly simple, on the level of an exercise problem: you should all be able to implement any of these from scratch after checking how layer normalization works and even that is a single-line expression