RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Inria

AILYS

ENS DE LYON

Large Language Models and Transformers

Mathurin Massias

https://mathurinm.github.io

Inria, OCKHAM team

16/06/2026

Outline and disclaimers

- I am not an expert in LLMs
- audience: computer scientists and mathematicians, a priori not doing ML
- goal: get a broad understanding of how a LLM works
- ⚠ some content is blatantly wrong but simplified to grasp the big picture

Outline and disclaimers

- I am not an expert in LLMs
- audience: computer scientists and mathematicians, a priori not doing ML
- goal: get a broad understanding of how a LLM works
- ⚠ some content is blatantly wrong but simplified to grasp the big picture
- I am not an expert in LLMs

Outline

Deep learning in a nutshell

Large language models

Transformers and Attention

More

Machine learning: learn to map inputs to outputs

In Machine Learning we have:

- *inputs* $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ e.g. flattened images of dogs and cats
- *labels* $y_1, \ldots, y_n \in \{0, 1\}$ e.g. $y = 0$ if image of cat, 1 if image of dog

We want:

- to learn to predict 0 or 1 for a new image $\mathbf{x}$ (= if it's a cat or a dog)

learning = finding a good function $f : \mathbb{R}^d \to \{0, 1\}$

this is only "binary classification", there is much more to ML

What should f output?

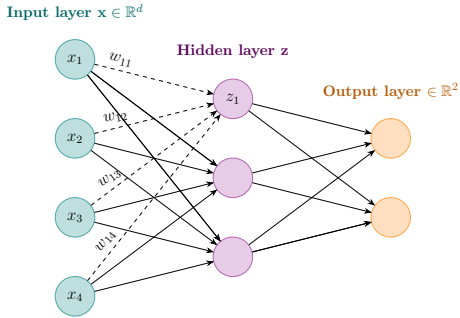
- predicting discrete values $f(\mathbf{x}) \in \{0, 1\}$ is a pain
- remedy: predict score vector $\in \mathbb{R}^2$:

$$f(\mathbf{x}) \approx \begin{pmatrix} \mathbb{P}(y = \text{cat}|\mathbf{x}) \\ \mathbb{P}(y = \text{dog}|\mathbf{x}) \end{pmatrix}$$

- this idea adapts to more labels e.g. cat/dog/horse/cow, $f(\mathbf{x}) \in \mathbb{R}^4$

Let's do this with Deep learning

- deep neural networks are parametric functions (here $f : \mathbb{R}^d \to \mathbb{R}^2$)

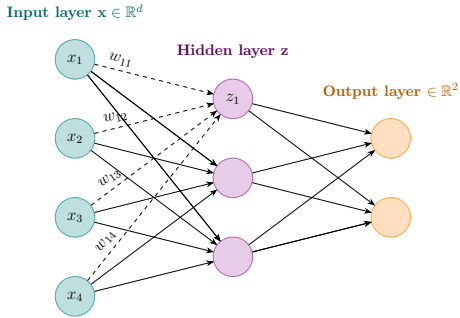


- connections between neurons have **weights** $w_{ij} \in \mathbb{R}$ (parameters: can be tuned):

$$z_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4 \in \mathbb{R}$$

Let's do this with Deep learning

- deep neural networks are parametric functions (here $f : \mathbb{R}^d \to \mathbb{R}^2$)

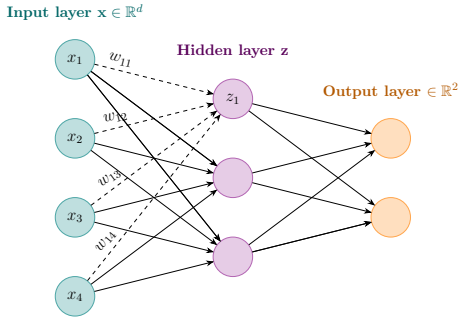


- connections between neurons have **weights** $w_{ij} \in \mathbb{R}$ (parameters: can be tuned):

$$z_1 = \phi(w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4) \in \mathbb{R} \qquad \phi : \mathbb{R} \to \mathbb{R} : \textit{ non linearity}$$

Let's do this with Deep learning

- deep neural networks are parametric functions (here $f : \mathbb{R}^d \to \mathbb{R}^2$)

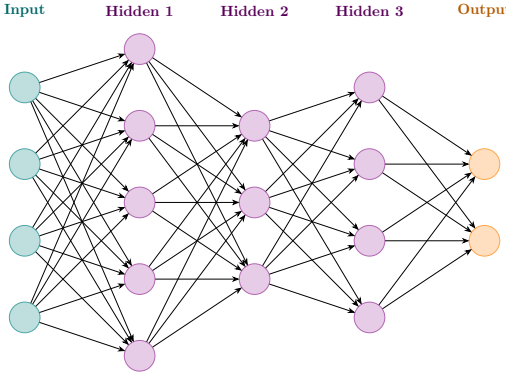


- connections between neurons have **weights** $w_{ij} \in \mathbb{R}$ (parameters: can be tuned):

$$z_1 = \phi(w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4) \in \mathbb{R} \qquad \phi : \mathbb{R} \to \mathbb{R} : \textit{ non linearity}$$

- compact notation: $\mathbf{z} = \phi(\mathbf{W}\mathbf{x}) \in \mathbb{R}^3, \quad \mathbf{W} \in \mathbb{R}^{3\times 4}$ ϕ applied pointwise

Moving on to the next layer



each layer has its weights $\mathbf{W}^\ell$

- $\mathbf{z}^1 = \phi(\mathbf{W}^1\mathbf{x})$
- $\mathbf{z}^2 = \phi(\mathbf{W}^2\mathbf{z}^1)$
- $\mathbf{z}^3 = \phi(\mathbf{W}^3\mathbf{z}^2)$
- $f(\mathbf{x}) = \phi(\mathbf{W}^4\mathbf{z}^3)$

$$f(\mathbf{x}) = \phi(\mathbf{W}^4\phi(\mathbf{W}^3\phi(\mathbf{W}^2\phi(\mathbf{W}^1\mathbf{x})))) \in \mathbb{R}^2$$

How to get probability vectors as output?

- remember we want $f(\mathbf{x}) \approx \begin{pmatrix} \mathbb{P}(y = \text{cat}|\mathbf{x}) \\ \mathbb{P}(y = \text{dog}|\mathbf{x}) \end{pmatrix}$: must be ≥ 0 and sum to 1
- the last layer nonlinearity is different: we use this nonlinearity is not pointwise

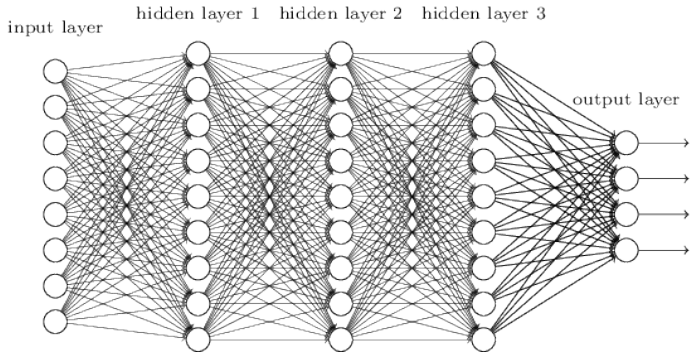
$$\text{softmax}(\mathbf{v}) : \mathbb{R}^p \to \Delta_p$$

$$\mathbf{v} \mapsto \left(\frac{\exp(v_i)}{\sum_{j=1}^{p} \exp(v_j)} \right)_i$$

- gives a positive vector & entries sum to 1 by design
- ex: $\text{softmax}(\begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}) \approx \begin{pmatrix} 0.67 \\ 0.09 \\ 0.24 \end{pmatrix}$

DEEP neural networks

- simply means many hidden layers:



- large models: easily millions of parameters (quadratic in number of neurons in layer)
- ChatGPT: $\approx$ 175 billion parameters

Learning the weights (optimization)

- fix *architecture* (number of layers, size of layers)

Learning the weights (optimization)

- fix *architecture* (number of layers, size of layers)
- pick random initial values for weights $\mathbf{W}^{(\ell)}$ in each layer ℓ

Learning the weights (optimization)

- fix *architecture* (number of layers, size of layers)
- pick random initial values for weights $\mathbf{W}^{(\ell)}$ in each layer ℓ
- for each image $\mathbf{x}_i$ in training set, compare true label y_i with network prediction $f(\mathbf{x}_i)$

Learning the weights (optimization)

- fix *architecture* (number of layers, size of layers)
- pick random initial values for weights $\mathbf{W}^{(\ell)}$ in each layer ℓ
- for each image $\mathbf{x}_i$ in training set, compare true label y_i with network prediction $f(\mathbf{x}_i)$
- modify weights a bit to make prediction closer to true label y_i, and iterate
  keywords: stochastic gradient descent, ADAM, Muon, cross-entropy loss

Outline

Deep learning in a nutshell

Large language models

Transformers and Attention

More

The big picture

Large Language Models simply do **next word prediction**:

- input $\mathbf{x}$ = last $T \in \mathbb{N}$ words of discussion
- output $f(\mathbf{x})$ = next word
- model is called *autoregressive*: it adds the prediction to the discussion, then predicts again the next word, and again and again

$$f\Big(\text{"can"}, \text{"you"}, \text{"help"}, \text{"me"}, \text{"?"}\Big) = \text{"Of"}$$

$$f\Big(\text{"you"}, \text{"help"}, \text{"me"}, \text{"?"}, \text{"Of"}\Big) = \text{"course"}$$

$$f\Big(\text{"help"}, \text{"me"}, \text{"?"}, \text{"Of"}, \text{"course"}\Big) = \ldots$$

The big picture

Large Language Models simply do **next word prediction**:

- input $\mathbf{x}$ = last $T \in \mathbb{N}$ words of discussion
- output $f(\mathbf{x})$ = next word
- model is called *autoregressive*: it adds the prediction to the discussion, then predicts again the next word, and again and again
- in fact output $f(\mathbf{x})$ = vector of probability over all possible words

$$f\Big(\text{"can", "you", "help", "me", "?"}\Big) = \begin{pmatrix} \mathbb{P}(\text{"a"}|\mathbf{x}) \\ \mathbb{P}(\text{"the"}|\mathbf{x}) \\ \mathbb{P}(\text{"Of"}|\mathbf{x}) \\ \vdots \end{pmatrix}$$

- this is exactly the same setting as with cats and dogs! but much bigger output space

The big picture

Large Language Models simply do **next word prediction**:

- input $\mathbf{x}$ = last $T \in \mathbb{N}$ words of discussion
- output $f(\mathbf{x})$ = next word
- model is called *autoregressive*: it adds the prediction to the discussion, then predicts again the next word, and again and again
- in fact output $f(\mathbf{x})$ = vector of probability over all possible words

$$f\Big(\text{"can", "you", "help", "me", "?"}\Big) = \begin{pmatrix} \mathbb{P}(\text{"a"}|\mathbf{x}) \\ \mathbb{P}(\text{"the"}|\mathbf{x}) \\ \mathbb{P}(\text{"Of"}|\mathbf{x}) \\ \vdots \end{pmatrix}$$

- this is exactly the same setting as with cats and dogs! but much bigger output space
- prediction can be random or not (sample from estimated probability distribution, or use most probable word)

The big picture

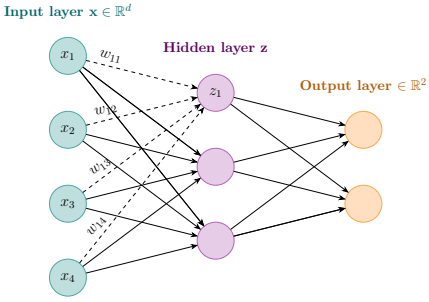
Large Language Models simply do **next word prediction**:

- input $\mathbf{x}$ = last $T \in \mathbb{N}$ words of discussion
- output $f(\mathbf{x})$ = next word
- model is called *autoregressive*: it adds the prediction to the discussion, then predicts again the next word, and again and again
- in fact output $f(\mathbf{x})$ = vector of probability over all possible words

$$f\Big(\text{"can", "you", "help", "me", "?"}\Big) = \begin{pmatrix} \mathbb{P}(\text{"a"}|\mathbf{x}) \\ \mathbb{P}(\text{"the"}|\mathbf{x}) \\ \mathbb{P}(\text{"Of"}|\mathbf{x}) \\ \vdots \end{pmatrix}$$

- this is exactly the same setting as with cats and dogs! but much bigger output space
- prediction can be random or not (sample from estimated probability distribution, or use most probable word)
- super easy to find training data with true answers

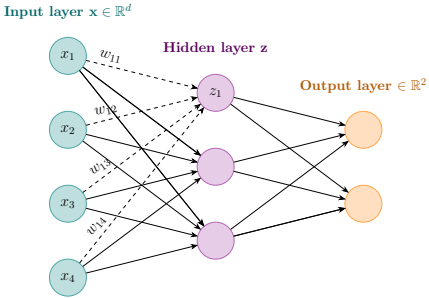
About the input: embedding



$$\mathbf{x} = (\text{"can"}, \text{"you"}, \text{"help"}, \text{"me"})$$

$$z_1 = \phi(w_{11} \times \text{"can"} + w_{12} \times \text{"you"} + ...) \text{ ???????}$$

About the input: embedding



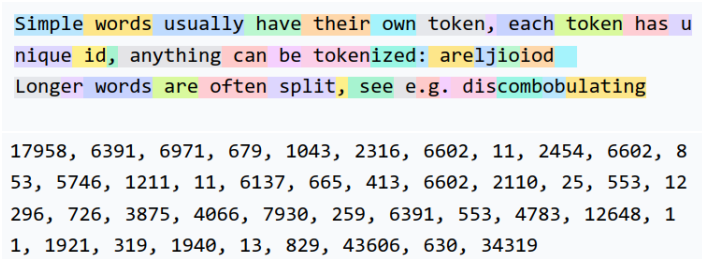
$$\mathbf{x} = (\text{"can"}, \text{"you"}, \text{"help"}, \text{"me"})$$

$$z_1 = \phi(w_{11} \times \text{"can"} + w_{12} \times \text{"you"} + ...) \text{ ???????}$$

- in practice each word in the dictionary is mapped to its own *embedding* $\in \mathbb{R}^e$
  typically $e \approx 10\,000$
- we hence have a numerical input $\mathbf{x} \in \mathbb{R}^{e \times T}$ (concatenation of embeddings)
- the embedding vectors are learned, along with the network's weights

About the input: tokenization

- working with words is inefficient: instead, use *tokens* (≈ pieces of words)
- set of tokens is learned independently with the Byte Pair Encoding (BPE) algorithm
  `https://www.youtube.com/watch?v=HEikzVL-lZU`
- play with various tokenizers: `https://tiktokenizer.vercel.app/`



- typically $n_{\text{tokens}} \approx$ 150 000 distinct tokens

Tokenization and embedding: recap

- the input prompt is tokenized into T tokens (T fixed)
  typically context length $T \approx 10\,000$
- each token is mapped to its *embedding* $\in \mathbb{R}^e$
  typically embedding size $e \approx 10\,000$
- the input of model f consists of T embeddings, and it outputs a vector of probabilities over all possible tokens; in the end a LLM is:

$$f : \mathbb{R}^{e \times T} \to \mathbb{R}^{n_{\text{tokens}}}$$

Tokenization and embedding: recap

- the input prompt is tokenized into T tokens (T fixed)
  - typically context length $T \approx 10\,000$
- each token is mapped to its *embedding* $\in \mathbb{R}^e$
  - typically embedding size $e \approx 10\,000$
- the input of model f consists of T embeddings, and it outputs a vector of probabilities over all possible tokens; in the end a LLM is:

$$f : \mathbb{R}^{e \times T} \to \mathbb{R}^{n_{\text{tokens}}}$$

- in practice your GPT prompt is prefixed with a fixed set of instructions (see answer to "what's your name?")
- how do we do if input prompt is shorter than T?

Tokenization and embedding: recap

- the input prompt is tokenized into T tokens (T fixed)
  typically context length $T \approx 10\,000$
- each token is mapped to its *embedding* $\in \mathbb{R}^e$
  typically embedding size $e \approx 10\,000$
- the input of model f consists of T embeddings, and it outputs a vector of probabilities over all possible tokens; in the end a LLM is:

$$f : \mathbb{R}^{e \times T} \to \mathbb{R}^{n_{\text{tokens}}}$$

- in practice your GPT prompt is prefixed with a fixed set of instructions (see answer to "what's your name?")
- how do we do if input prompt is shorter than T? special `PAD` token
- when does the model decide to stop its answer?

Tokenization and embedding: recap

- the input prompt is tokenized into T tokens (T fixed)
  typically context length $T \approx 10\,000$
- each token is mapped to its *embedding* $\in \mathbb{R}^e$
  typically embedding size $e \approx 10\,000$
- the input of model f consists of T embeddings, and it outputs a vector of probabilities over all possible tokens; in the end a LLM is:

$$f : \mathbb{R}^{e \times T} \to \mathbb{R}^{n_{\text{tokens}}}$$

- in practice your GPT prompt is prefixed with a fixed set of instructions (see answer to "what's your name?")
- how do we do if input prompt is shorter than T? special `PAD` token
- when does the model decide to stop its answer? special `END` token

Outline

Deep learning in a nutshell

Large language models

Transformers and Attention

More

Transformers

- the T in GPT
- 2017 paper, 253 775 citations as of yesterday

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

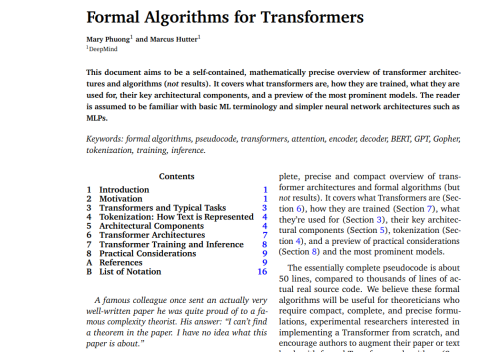
Łukasz Kaiser[*]
Google Brain
lukaszkaiser@google.com

Illia Polosukhin[*] [‡]
illia.polosukhin@gmail.com

- a special neural network architecture, designed for sequence modelling

Two great ressources

- Phuong & Hutter, *Formal Algorithms for Transformers*
  `https://arxiv.org/abs/2207.09238`
- Karpathy, *Let's build GPT: from scratch, in code, spelled out.*
  `https://www.youtube.com/watch?v=kCc8FmEb1nY`



Formal Algorithms for Transformers

Mary Phuong[1] **and Marcus Hutter**[1]
[1]DeepMind

This document aims to be a self-contained, mathematically precise overview of transformer architectures and algorithms (*not* results). It covers what transformers are, how they are trained, what they are used for, their key architectural components, and a preview of the most prominent models. The reader is assumed to be familiar with basic ML terminology and simpler neural network architectures such as MLPs.

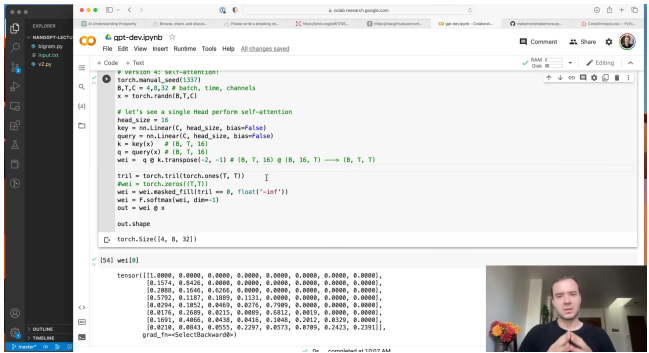
Keywords: formal algorithms, pseudocode, transformers, attention, encoder, decoder, BERT, GPT, Gopher, tokenization, training, inference.

Contents

A famous colleague once sent an actually very well-written paper he was quite proud of to a famous complexity theorist. His answer: "I can't find a theorem in the paper. I have no idea what this paper is about."

plete, precise and compact overview of transformer architectures and formal algorithms (but *not* results). It covers what Transformers are (Section 6), how they are trained (Section 7), what they're used for (Section 3), their key architectural components (Section 5), tokenization (Section 4), and a preview of practical considerations (Section 8) and the most prominent models.

The essentially complete pseudocode is about 50 lines, compared to thousands of lines of actual real source code. We believe these formal algorithms will be useful for theoreticians who require compact, complete, and precise formulations, experimental researchers interested in implementing a Transformer from scratch, and encourage authors to augment their paper or text



The Attention mechanism

- remember our inputs consist of embedded tokens:

$$\mathbf{x} = (\mathbf{t}_1, \ldots, \mathbf{t}_T) \in (\mathbb{R}^e)^T$$

  with e embedding size, T context length

- the Attention layer is a layer that maps a sequence of T token embeddings to a sequence of T transformed token embedding:

$$(\mathbf{t}_1, \ldots, \mathbf{t}_T) \mapsto (\mathbf{t}'_1, \ldots, \mathbf{t}'_T) \in (\mathbb{R}^e)^T$$

  I will say “token” when I should say “token embedding”

- Transformers are basically stacked Attention layers (intertwined with fully connected layers)

$$(\mathbf{t}_1, \ldots, \mathbf{t}_T) \mapsto (\mathbf{t}'_1, \ldots, \mathbf{t}'_T) \mapsto (\mathbf{t}''_1, \ldots, \mathbf{t}''_T) \mapsto \ldots$$

The idea of Attention

- value of transformed token number i, $\mathbf{t}'_i$, is chosen to be a combination of all input token values $\mathbf{t}_j$:

$$\mathbf{t}'_i = \sum_{j=1}^{T} \lambda_{ij} \mathbf{t}_j$$

- weight λ_{ij} should be large if $\mathbf{t}_i$ similar to $\mathbf{t}_j$, small otherwise
- weights (λ_{ij}) should be positive and sum to 1 over j

The idea of Attention

- value of transformed token number i, $\mathbf{t}'_i$, is chosen to be a combination of all input token values $\mathbf{t}_j$:

$$\mathbf{t}'_i = \sum_{j=1}^{T} \lambda_{ij} \mathbf{t}_j$$

- weight λ_{ij} should be large if $\mathbf{t}_i$ similar to $\mathbf{t}_j$, small otherwise
- weights (λ_{ij}) should be positive and sum to 1 over j
- $\rightsquigarrow$ use $\lambda_{i\cdot} = \operatorname{softmax}\left(\left(\langle \mathbf{t}_i, \mathbf{t}_j \rangle\right)_{j=1,\ldots,T}\right) \in \Delta_T$ Δ is the simplex

The idea of Attention

- value of transformed token number i, $\mathbf{t}'_i$, is chosen to be a combination of all input token values $\mathbf{t}_j$:

$$\mathbf{t}'_i = \sum_{j=1}^{T} \lambda_{ij} \mathbf{t}_j$$

- weight λ_{ij} should be large if $\mathbf{t}_i$ similar to $\mathbf{t}_j$, small otherwise
- weights (λ_{ij}) should be positive and sum to 1 over j
- $\rightsquigarrow$ use $\lambda_{i\cdot} = \mathrm{softmax}\left((\langle \mathbf{t}_i, \mathbf{t}_j \rangle)_{j=1,\dots,T}\right) \in \Delta_T$ Δ is the simplex
- gain flexibility by using three linear transformations of tokens:

$$\mathbf{t}'_i = \sum_{j=1}^{T} \lambda_{ij} \mathbf{V} \mathbf{t}_j$$

with $\lambda_{i\cdot} = \mathrm{softmax}\left((\langle \mathbf{Q}\mathbf{t}_i, \mathbf{K}\mathbf{t}_j \rangle)_{j=1,\dots,T}\right) \in \Delta_T$

key, query, value matrices of adequate size
(different matrices for each attention layer, learnable)

Intuition behind Q, K, V

$$\mathbf{t}'_i = \sum_{j=1}^{T} \lambda_{ij} \mathbf{V}\mathbf{t}_j$$

with $\lambda_{i\cdot} = \mathrm{softmax}\left(\left(\langle \mathbf{Q}\mathbf{t}_i, \mathbf{K}\mathbf{t}_j \rangle\right)_{j=1,\dots,T}\right) \in \Delta_T$

- Attention is a “content-based retrieval mechanism”
- Query $\mathbf{Q}\mathbf{t}_i$ asks: “what is token i looking for?”
- Keys $\mathbf{K}\mathbf{t}_j$ describe: “what does token j contain?”
- Values $\mathbf{V}\mathbf{t}_j$ are: “what information does token j pass on?”

Intuition behind Q, K, V

$$\mathbf{t}'_i = \sum_{j=1}^{T} \lambda_{ij} \mathbf{V}\mathbf{t}_j$$

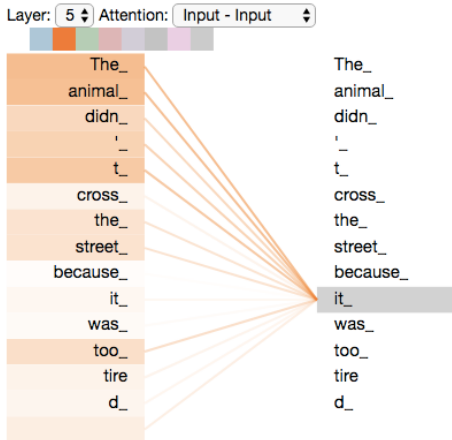
with $\lambda_{i\cdot} = \mathrm{softmax}\left(\left(\langle \mathbf{Q}\mathbf{t}_i, \mathbf{K}\mathbf{t}_j \rangle\right)_{j=1,\dots,T}\right) \in \Delta_T$

- Attention is a “content-based retrieval mechanism”
- Query $\mathbf{Q}\mathbf{t}_i$ asks: “what is token i looking for?”
- Keys $\mathbf{K}\mathbf{t}_j$ describe: “what does token j contain?”
- Values $\mathbf{V}\mathbf{t}_j$ are: “what information does token j pass on?”
- Matching score:

  $$\langle \mathbf{Q}\mathbf{t}_i, \mathbf{K}\mathbf{t}_j \rangle$$

  measures compatibility between request and memory

Visualizing attention



source: `https://jalammar.github.io/illustrated-transformer/`

The whole Transformer architecture

- chain several Attention layers (≈ 50)
- end up with sequence of T embeddings
- at last layer, use fully connected layer to map to $\mathbb{R}^{n_{\text{tokens}}}$ + softmax to get probability vector of next token

The whole Transformer architecture

- chain several Attention layers (≈ 50)
- end up with sequence of T embeddings
- at last layer, use fully connected layer to map to $\mathbb{R}^{n_{\text{tokens}}}$ + softmax to get probability vector of next token
- not covered: multi-head attention, normalization layers, residual connections, causal masking, positional embedding

What make models different?

GPT, DeepSeek, Gemini, Claude, LeChat, Emmy...

- all based on Transformer architecture
- performance differences arise from:
  - model size (parameters, context length)
  - training data and filtering (quality data matter)
  - training and finetuning
  - broader choices (e.g. mixture of experts)
  - inference-time techniques (chain of thought)

Outline

Deep learning in a nutshell

Large language models

Transformers and Attention

More

Pretraining vs finetuning and alignment

- LLMs are first pretrained on massive corpora ($\approx$ the whole web); pretraining learns general linguistic and world knowledge
- finetuning = small retraining of pretrained model, on curated input/output pairs, aimed at adapting the model to a specific task (coding, maths)
  keyword: LoRA (Low Rank Adaptation)
- alignment = making sure the model performs as desired in some cases
  keyword: RLHF (Reinforcement Learning with Human Feedback)

Mixture of Experts (MoE)

- replace one large network by many experts network
- a router selects which experts process each token
- only a few experts are active at a time
- model size grows while computation remains nearly constant

Discrete diffusion: The next generation of text generation?

- Continuous diffusion: take clean data, progressively transform it into pure Gaussian noise (left to right); learn to perform the *reverse* task (right to left) to generate images

- Discrete diffusion: take clean documents, progressively mask each token, learn to invert the process (right to left)

- difference with autoregressive models: non causal, much faster ($4\times$?)
- GemmaDiffusion by Google, June 10th

`https://blog.google/innovation-and-ai/technology/developers-tools/diffusion-gemma-faster-text-generation/`

Illustrations courtesy of Georges Le Bellier and Orel Mazor