Transformers

Gustave Cortal

Transformers *vs* recurrent neural networks

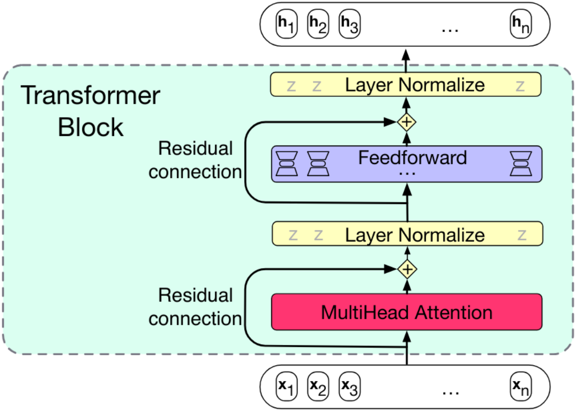
The transformer offers new mechanisms (**positional encodings** and **self-attention**) that help represent time and help focus on how words relate to each other over long distances

Transformers *vs* recurrent neural networks

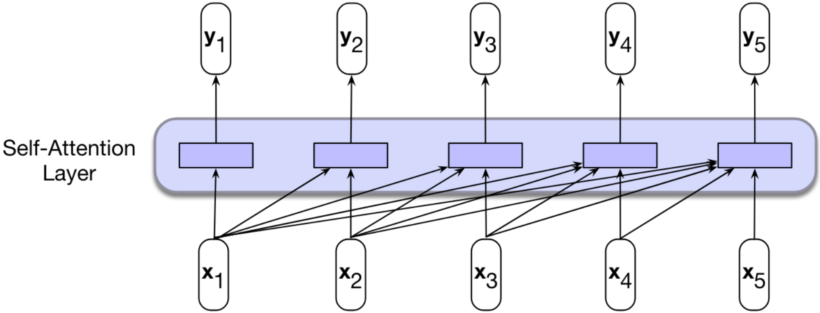
The transformer offers new mechanisms (**positional encodings** and **self-attention**) that help represent time and help focus on how words relate to each other over long distances

Unlike RNNs, the computations at each time step are **independent of all the other steps** and, therefore, can **be performed in parallel**

Transformer block

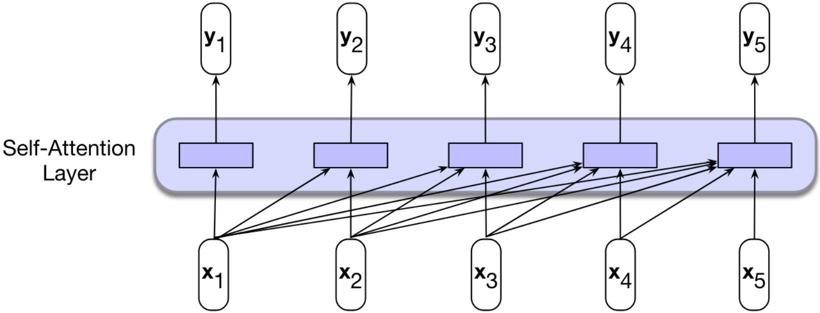


Self-attention layer



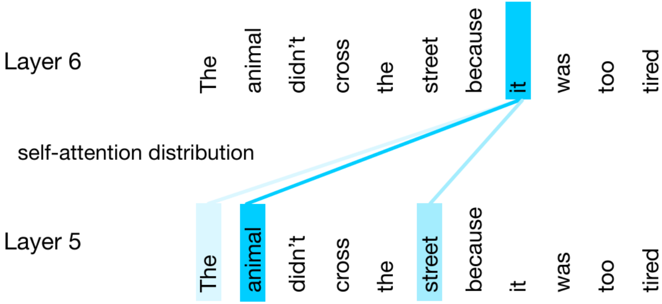
Self-attention directly extracts and uses information from arbitrarily large contexts without passing it through intermediate recurrent connections

Self-attention layer



Self-attention directly extracts and uses information from arbitrarily large contexts without passing it through intermediate recurrent connections

Attention visualization



Main idea of attention mechanisms

An attention-based approach is a set of **comparisons to relevant items** in some context, a **normalization** of those scores to provide a probability distribution, and a **weighted sum** using this distribution

Dot-product attention

A **dot product** is the simplest form of comparison between elements in a self-attention layer:

$$\text{score}(x_i, x_j) = x_i \cdot x_j$$

Dot-product attention

A **dot product** is the simplest form of comparison between elements in a self-attention layer:

$$\text{score}(x_i, x_j) = x_i \cdot x_j$$

Then, we **normalize** the scores with a softmax to create a vector of weights, α_{ij}, that indicates the proportional relevance of each input j to the input element i

Dot-product attention

A **dot product** is the simplest form of comparison between elements in a self-attention layer:

$$\text{score}(x_i, x_j) = x_i \cdot x_j$$

Then, we **normalize** the scores with a softmax to create a vector of weights, α_{ij}, that indicates the proportional relevance of each input j to the input element i

$$\begin{aligned}\alpha_{ij} &= \text{softmax}(\text{score}(x_i, x_j)) \quad \forall j \leq i \\ &= \frac{\exp(\text{score}(x_i, x_j))}{\sum_{k=1}^{i} \exp(\text{score}(x_i, x_k))} \quad \forall j \leq i\end{aligned}$$

Dot-product attention

A **dot product** is the simplest form of comparison between elements in a self-attention layer:

$$\text{score}(x_i, x_j) = x_i \cdot x_j$$

Then, we **normalize** the scores with a softmax to create a vector of weights, α_{ij}, that indicates the proportional relevance of each input j to the input element i

$$\begin{aligned}\alpha_{ij} &= \text{softmax}(\text{score}(x_i, x_j)) \quad \forall j \leq i \\ &= \frac{\exp(\text{score}(x_i, x_j))}{\sum_{k=1}^{i} \exp(\text{score}(x_i, x_k))} \quad \forall j \leq i\end{aligned}$$

Finally, we generate an output value y_i by taking the **sum** of the inputs seen so far, **weighted** by their respective α value.

$$y_i = \sum_{j \leq i} \alpha_{ij} x_j$$

Attention with queries, keys and values

But transformers create a **more sophisticated way** of representing how words can contribute to the representation of longer inputs. Consider the three roles each input embedding plays during the attention process:

- As the current focus of attention when being compared to all of the other preceding inputs $\rightarrow$ **query**
- In its role as a preceding input being compared to the current focus of attention $\rightarrow$ **key**
- And finally, as a **value** used to compute the output for the current focus of attention

Attention with queries, keys and values

But transformers create a **more sophisticated way** of representing how words can contribute to the representation of longer inputs. Consider the three roles each input embedding plays during the attention process:

- As the current focus of attention when being compared to all of the other preceding inputs $\rightarrow$ **query**
- In its role as a preceding input being compared to the current focus of attention $\rightarrow$ **key**
- And finally, as a **value** used to compute the output for the current focus of attention

To capture these three different roles, transformers introduce weight matrices W_Q, W_K, and W_V. These weights project each input vector x_i into a representation of its role as a key, query, or value:

$$\begin{aligned} q_i &= W_Q x_i, \\ k_i &= W_K x_i, \\ v_i &= W_V x_i \end{aligned}$$

$x_i \in \mathbb{R}^{d \times 1}$, $W_Q \in \mathbb{R}^{d \times d}$, $W_K \in \mathbb{R}^{d \times d}$, and $W_V \in \mathbb{R}^{d \times d}$.

Attention with queries, keys and values

Given these projections, the score between a current focus of attention, x_i, and an element in the preceding context, x_j, consists of a dot product between its query vector q_i and the preceding element's key vectors k_j:

$$\text{score}(x_i, x_j) = q_i \cdot k_j$$

Attention with queries, keys and values

Given these projections, the score between a current focus of attention, x_i, and an element in the preceding context, x_j, consists of a dot product between its query vector q_i and the preceding element's key vectors k_j:

$$\text{score}(x_i, x_j) = q_i \cdot k_j$$

The output calculation for y_i is now based on a weighted sum over the value vectors v:

$$y_i = \sum_{j \leq i} \alpha_{ij} v_j$$

Attention with queries, keys and values

Given these projections, the score between a current focus of attention, x_i, and an element in the preceding context, x_j, consists of a dot product between its query vector q_i and the preceding element's key vectors k_j:

$$\text{score}(x_i, x_j) = q_i \cdot k_j$$

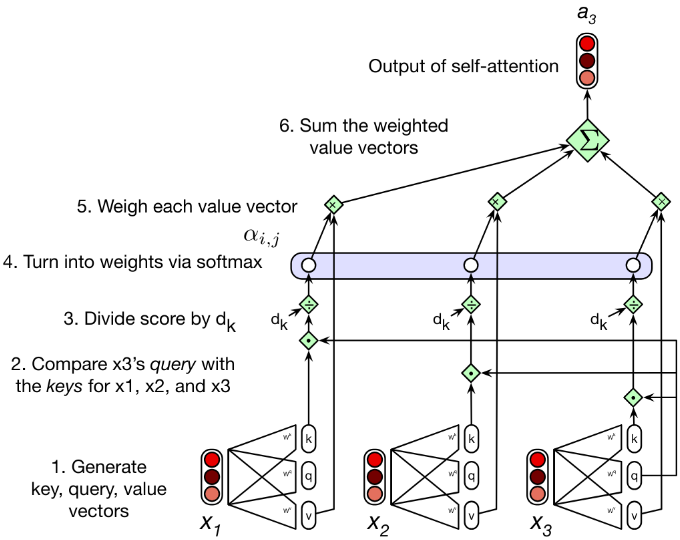
The output calculation for y_i is now based on a weighted sum over the value vectors v:

$$y_i = \sum_{j \leq i} \alpha_{ij} v_j$$

Exponentiating large values can lead to numerical issues. To avoid this, we **scale** the dot-product by a factor related to the size of the embeddings:

$$\text{score}(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d}}$$

Attention with queries, keys and values



Parallelization

Since each output y_i is computed independently, the entire process can be parallelized by taking advantage of matrix multiplication

Parallelization

Since each output y_i is computed independently, the entire process can be parallelized by taking advantage of matrix multiplication

Input tokens are packed into a single matrix $X \in \mathbb{R}^{N \times d}$. We multiply X by the key, query, and value matrices:

$$Q = XW_Q; \quad K = XW_K; \quad V = XW_V$$

$Q \in \mathbb{R}^{N \times d}$, $K \in \mathbb{R}^{N \times d}$, and $V \in \mathbb{R}^{N \times d}$

Parallelization

Since each output y_i is computed independently, the entire process can be parallelized by taking advantage of matrix multiplication

Input tokens are packed into a single matrix $X \in \mathbb{R}^{N \times d}$. We multiply X by the key, query, and value matrices:

$$Q = XW_Q; \quad K = XW_K; \quad V = XW_V$$

$Q \in \mathbb{R}^{N \times d}$, $K \in \mathbb{R}^{N \times d}$, and $V \in \mathbb{R}^{N \times d}$

We've reduced the self-attention step for a sequence of N tokens:

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

Masked attention matrix

N

q1•k1	−∞	−∞	−∞	−∞
q2•k1	q2•k2	−∞	−∞	−∞
q3•k1	q3•k2	q3•k3	−∞	−∞
q4•k1	q4•k2	q4•k3	q4•k4	−∞
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5

N

QK^T results in a score for each query value to every key value, including those that follow the query

Masked attention matrix

q1•k1	−∞	−∞	−∞	−∞
q2•k1	q2•k2	−∞	−∞	−∞
q3•k1	q3•k2	q3•k3	−∞	−∞
q4•k1	q4•k2	q4•k3	q4•k4	−∞
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5

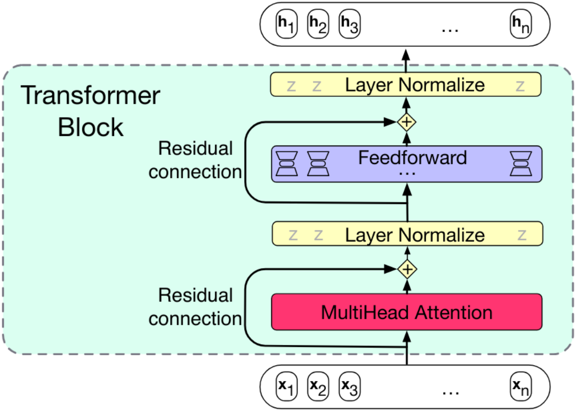
N

N

QK^T results in a score for each query value to every key value, including those that follow the query

This is inappropriate in language modeling since guessing the next word is pretty simple if you already know it. To fix this, the elements in the upper-triangular portion of the matrix are set to $-\infty$

Transformer block



Multihead attention

Different words in a sentence can relate to each other in many different ways simultaneously

Multihead attention

Different words in a sentence can relate to each other in many different ways simultaneously

It is difficult for a transformer block to capture all kinds of parallel relations among its inputs

Multihead attention

Different words in a sentence can relate to each other in many different ways simultaneously

It is difficult for a transformer block to capture all kinds of parallel relations among its inputs

Transformers address this issue with **multihead self-attention layers**, sets of self-attention layers, called heads, that reside in parallel layers at the same depth in a model, each with its own set of parameters

Multihead attention

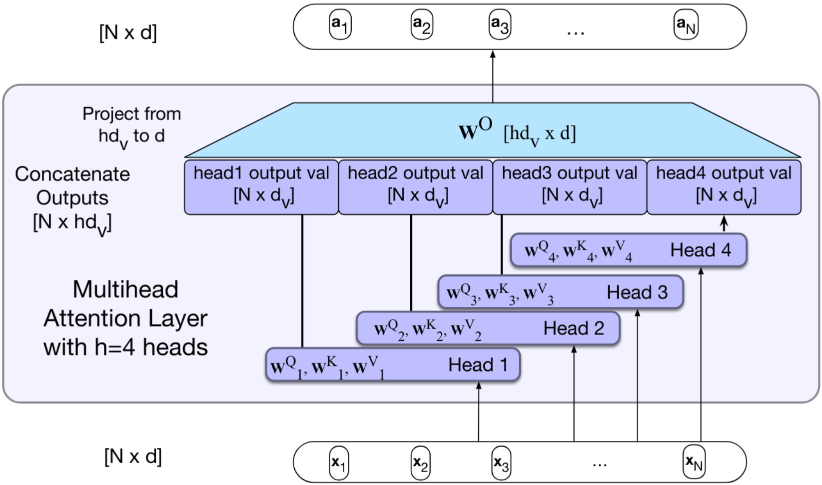
Different words in a sentence can relate to each other in many different ways simultaneously

It is difficult for a transformer block to capture all kinds of parallel relations among its inputs

Transformers address this issue with **multihead self-attention layers**, sets of self-attention layers, called heads, that reside in parallel layers at the same depth in a model, each with its own set of parameters

Given these distinct sets of parameters, each head can learn different aspects of the relationships among inputs at the same level of abstraction

Multihead attention



Multihead attention

To implement this notion, each head, i, in a self-attention layer is provided with its own set of key, query, and value matrices: W_i^K, W_i^Q, and W_i^V

Multihead attention

To implement this notion, each head, i, in a self-attention layer is provided with its own set of key, query, and value matrices: W_i^K, W_i^Q, and W_i^V

In multi-head attention, instead of using the model dimension d that's used for the input and output from the model, the key and query embeddings have dimensionality $d_k << d$

Multihead attention

To implement this notion, each head, i, in a self-attention layer is provided with its own set of key, query, and value matrices: W_i^K, W_i^Q, and W_i^V

In multi-head attention, instead of using the model dimension d that's used for the input and output from the model, the key and query embeddings have dimensionality $d_k << d$

$$\text{MultiHeadAttention}(X) = (\text{head}_1 \oplus \text{head}_2 \ldots \oplus \text{head}_h)W^O$$

$$Q_i = XW_i^Q; \quad K_i = XW_i^K; \quad V_i = XW_i^V$$

$$\text{head}_i = \text{SelfAttention}(Q_i, K_i, V_i)$$

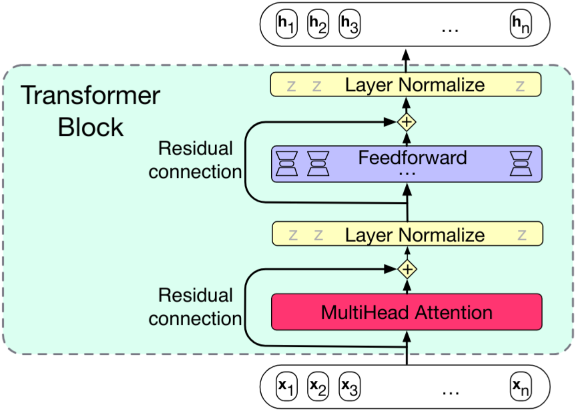
$X \in \mathbb{R}^{N \times d}$

$W_i^Q \in \mathbb{R}^{d \times d_k}$, $W_i^K \in \mathbb{R}^{d \times d_k}$, and $W_i^V \in \mathbb{R}^{d \times d_v}$

$Q \in \mathbb{R}^{N \times d_k}$, $K \in \mathbb{R}^{N \times d_k}$, and $V \in \mathbb{R}^{N \times d_v}$

$W^O \in \mathbb{R}^{hd_v \times d}$

Transformer block



Residual connections

Residual connections pass information from a lower layer to a higher layer without going through the intermediate layer

Residual connections

Residual connections pass information from a lower layer to a higher layer without going through the intermediate layer

Allowing information from the activation going forward and the gradient going backward to skip a layer improves learning and gives higher-level layers direct access to information from lower layers

Residual connections

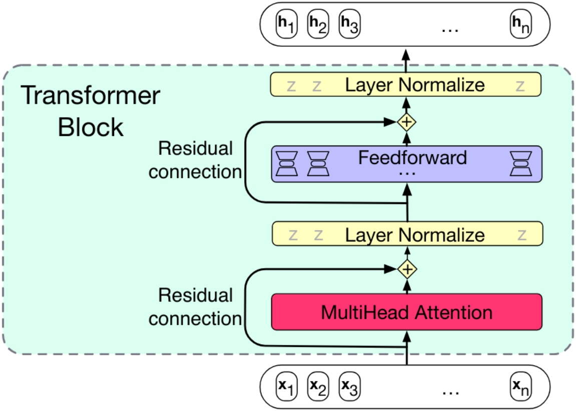
Residual connections pass information from a lower layer to a higher layer without going through the intermediate layer

Allowing information from the activation going forward and the gradient going backward to skip a layer improves learning and gives higher-level layers direct access to information from lower layers

If we think of a layer as one long vector of units, the resulting function computed in a transformer block can be expressed as:

$$O = \text{LayerNorm}(\mathbf{X} + \text{SelfAttention}(X))$$
$$H = \text{LayerNorm}(\mathbf{O} + \text{FFN}(O))$$

Transformer block



Layer normalization

$$O = \textbf{LayerNorm}(X + \text{SelfAttention}(X))$$
$$H = \textbf{LayerNorm}(O + \text{FFN}(O))$$

Layer normalization

$$O = \textbf{LayerNorm}(X + \text{SelfAttention}(X))$$
$$H = \textbf{LayerNorm}(O + \text{FFN}(O))$$

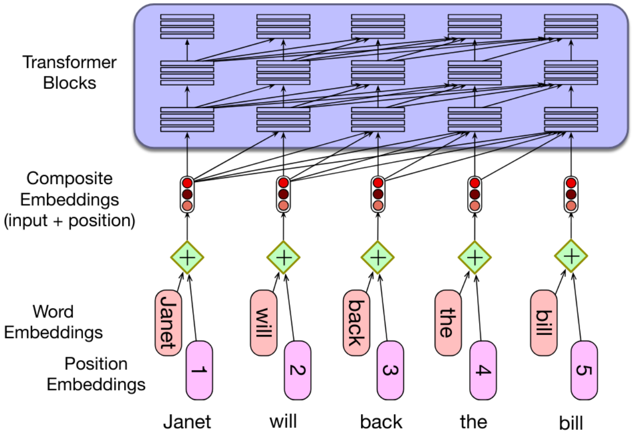
We calculate the mean, μ, and standard deviation, σ, over the elements of the vector to be normalized. Given a hidden layer with dimensionality d, these values are calculated as follows:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$$

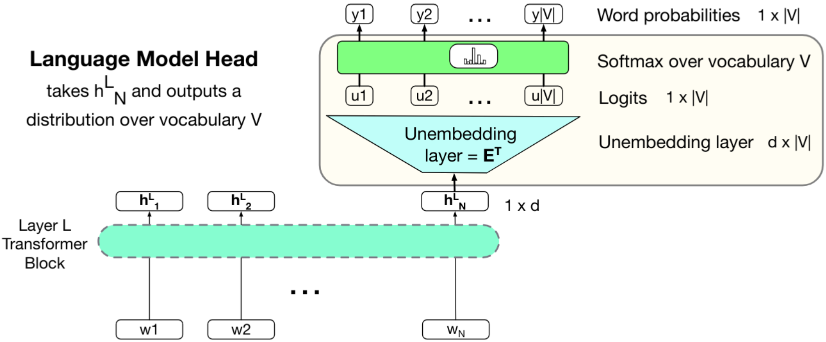
$$\hat{x} = \frac{(x - \mu)}{\sigma}$$

Positional encoding

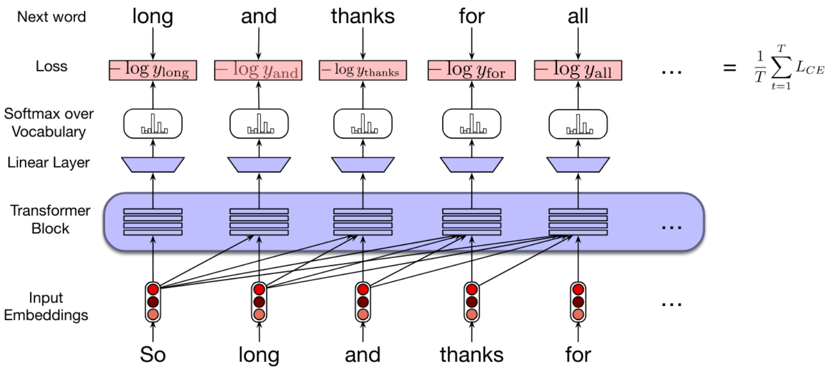


Train positional embeddings or use a static function that maps integer inputs to real-values vectors

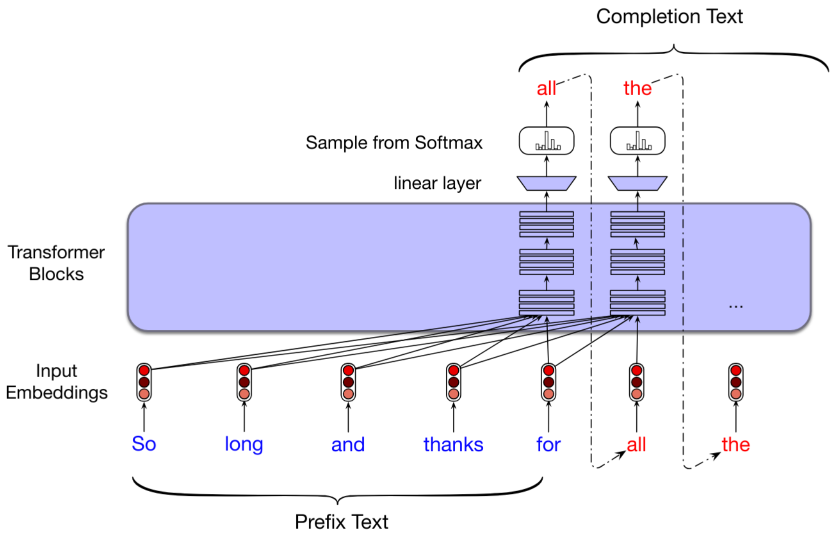
Language model head



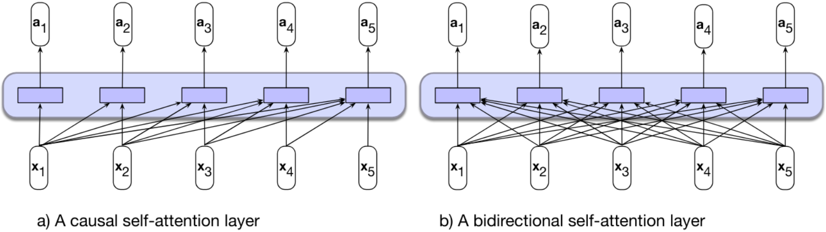
Language modeling using next word prediction



Conditional generation



Causal *vs* bidirectional language model



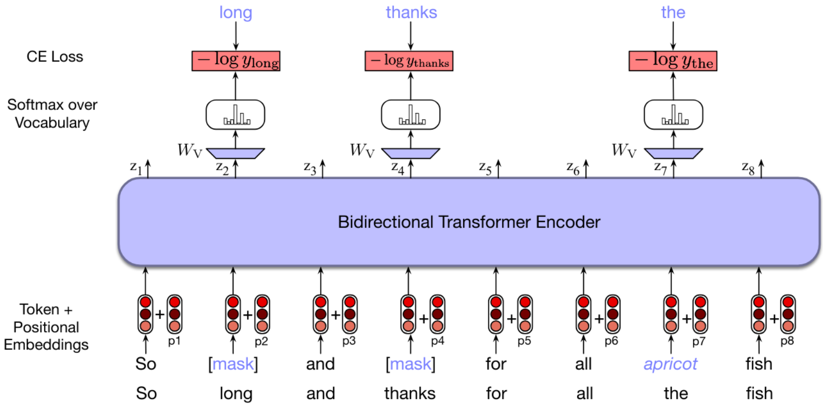
Attention matrix for bidirectional language model

N

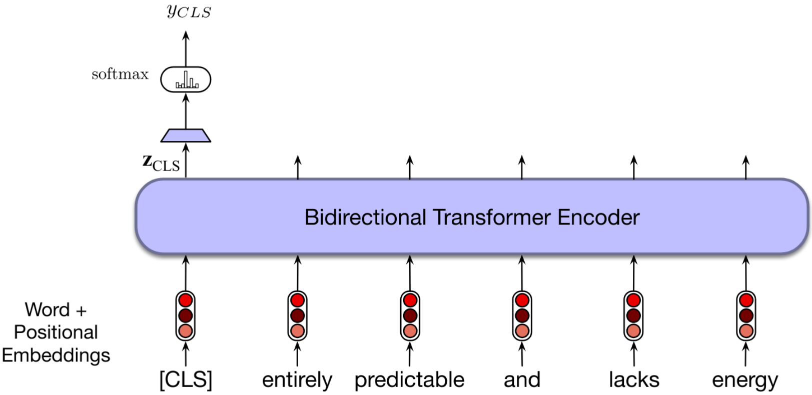
q1•k1	q1•k2	q1•k3	q1•k4	q1•k5
q2•k1	q2•k2	q2•k3	q2•k4	q2•k5
q3•k1	q3•k2	q3•k3	q3•k4	q3•k5
q4•k1	q4•k2	q4•k3	q4•k4	q4•k5
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5

N

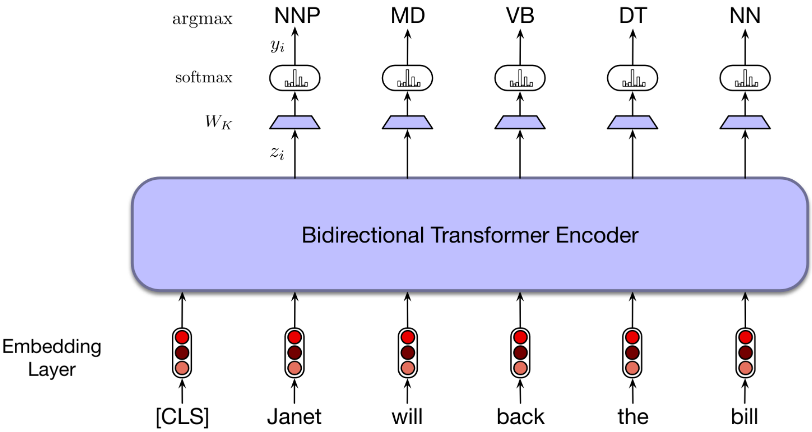
Masked language modeling



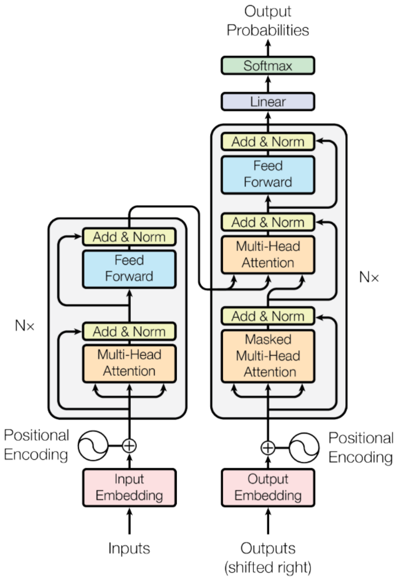
Sequence classification



Token classification



Transformer architecture from *Attention is All you Need*



Architecture, size, and hyperparameters of GPT-3 from *Language Models are Few-Shot Learners*

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Conclusion

Tokenization is splitting text into individual tokens

Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

N-grams are simple probabilistic language models based on Markov assumption

Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

N-grams are simple probabilistic language models based on Markov assumption

Naive bayes classifiers are generative models based on class-specific unigram

Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

N-grams are simple probabilistic language models based on Markov assumption

Naive bayes classifiers are generative models based on class-specific unigram

Embedding represents word meaning as a vector

Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

N-grams are simple probabilistic language models based on Markov assumption

Naive bayes classifiers are generative models based on class-specific unigram

Embedding represents word meaning as a vector

Logistic regressions are discriminative models based on the sigmoid function

Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

N-grams are simple probabilistic language models based on Markov assumption

Naive bayes classifiers are generative models based on class-specific unigram

Embedding represents word meaning as a vector

Logistic regressions are discriminative models based on the sigmoid function

Feedforward neural networks handle longer inputs and generalize better compared to N-grams thanks to embeddings, have fixed context windows

Conclusion

Recurrent neural networks handle temporal data inherently in the architecture, have infinite context windows, hidden states have local information

Conclusion

Recurrent neural networks handle temporal data inherently in the architecture, have infinite context windows, hidden states have local information

Information flow is better in gated recurrent networks due to better context management

Conclusion

Recurrent neural networks handle temporal data inherently in the architecture, have infinite context windows, hidden states have local information

Information flow is better in gated recurrent networks due to better context management

Attention mechanisms solve the bottleneck problem to produce dynamically derived context vectors

Conclusion

Recurrent neural networks handle temporal data inherently in the architecture, have infinite context windows, hidden states have local information

Information flow is better in gated recurrent networks due to better context management

Attention mechanisms solve the bottleneck problem to produce dynamically derived context vectors

Transformers use self-attention layers combined with feedforward layers to handle more complex distant relationships between tokens, enable parallelization due to independent computation between tokens, have fixed context windows

Ressources

Alammar, J. *The Illustrated Transformer*. https://jalammar.github.io/illustrated-transformer/

Alammar, J. *The Illustrated GPT-2*. https://jalammar.github.io/illustrated-gpt2/

3Blue1Brown's videos on *neural networks*. https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

Ressources

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. *Attention Is All You Need.* arXiv. https://doi.org/10.48550/arXiv.1706.03762

Phuong, M., & Hutter, M. *Formal Algorithms for Transformers.* arXiv. https://doi.org/10.48550/arXiv.2207.09238

Amirhossein Kazemnejad's blog. *Transformer Architecture: The Positional Encoding.* https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

Weng, L. *Attention? Attention!* https://lilianweng.github.io/posts/2018-06-24-attention/

Ressources

Harvard NLP. *The Annotated Transformer.* https://nlp.seas.harvard.edu/annotated-transformer/

Peter Bloem. *Transformers from scratch.* https://peterbloem.nl/blog/transformers

Andrej Karpathy. *Let's build GPT: From scratch, in code, spelled out.* https://www.youtube.com/watch?v=kCc8FmEb1nY

Warner, B. Creating a Transformer From Scratch - Part One: The Attention Mechanism. https://benjaminwarner.dev/2023/07/01/attention-mechanism.html

Warner, B. Creating a Transformer From Scratch - Part Two: The Rest of the Transformer. https://benjaminwarner.dev/2023/07/28/rest-of-the-transformer.html

Raschka, S. *Understanding and coding the self-attention mechanism from scratch*. https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html

Ressources

Collège de France, « Apprendre les langues aux machines »:
https://www.college-de-france.fr/fr/agenda/cours/apprendre-les-langues-aux-machines

Dan Jurafsky and James H. Martin, *Speech and Language Processing*:
https://web.stanford.edu/~jurafsky/slp3/ed3bookfeb3_2024.pdf

3Blue1Brown, *Essence of linear algebra* and *Neural Networks* playlists :
https://www.youtube.com/@3blue1brown/playlists

AI News: we summarize top AI discords + AI reddits + AI X/Twitters, and send you a roundup each day!
https://buttondown.email/ainews