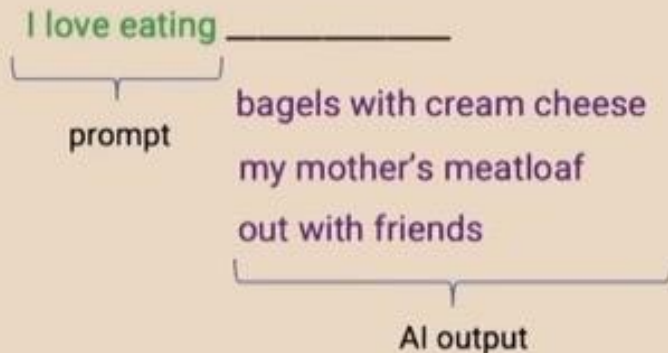


Transformer

Language Models

This decade: Generative AI

Text generation process



How it works

Generative AI is built by using supervised learning ($A \rightarrow B$) to repeatedly predict the next word.

My favorite food is a bagel with cream cheese and lox.

Input (A)	Output (B)
My favorite food is a	bagel
My favorite food is a bagel	with
My favorite food is a bagel with	cream

When we train a very large AI system on a lot of data (hundreds of billions of words) we get a Large Language Model like ChatGPT.

Stanford

Andrew Ng



Unsupervised Learning

learning by observation

no target information → kind of “vague” pattern recognition (but plenty of data)

can be cast as **self-supervised learning**:

- input-output mapping like supervised learning
- but generating labels itself from input information

generative AI as unsupervised learning: generate variations of training data

How Much Information is the Machine Given during Learning?

- ▶ **“Pure” Reinforcement Learning (cherry)**
 - ▶ The machine predicts a scalar reward given once in a while.
 - ▶ **A few bits for some samples**
- ▶ **Supervised Learning (icing)**
 - ▶ The machine predicts a category or a few numbers for each input
 - ▶ Predicting human-supplied data
 - ▶ **10→10,000 bits per sample**
- ▶ **Self-Supervised Learning (cake génoise)**
 - ▶ The machine predicts any part of its input for any observed part.
 - ▶ Predicts future frames in videos
 - ▶ **Millions of bits per sample**

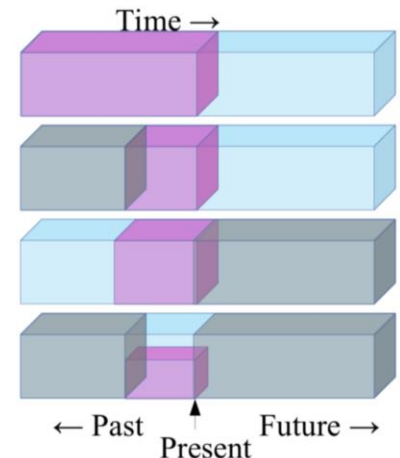


© 2019 IEEE International Solid-State Circuits Conference 1.1: Deep Learning Hardware: Past, Present, & Future

59

Self-Supervised Learning

- ▶ Predict any part of the input from any other part.
- ▶ Predict the **future** from the **past**.
- ▶ Predict the **future** from the **recent past**.
- ▶ Predict the **past** from the **present**.
- ▶ Predict the **top** from the **bottom**.
- ▶ Predict the **occluded** from the **visible**
- ▶ **Pretend there is a part of the input you don't know and predict that.**



© 2019 IEEE International Solid-State Circuits Conference 1.1: Deep Learning Hardware: Past, Present, & Future

58

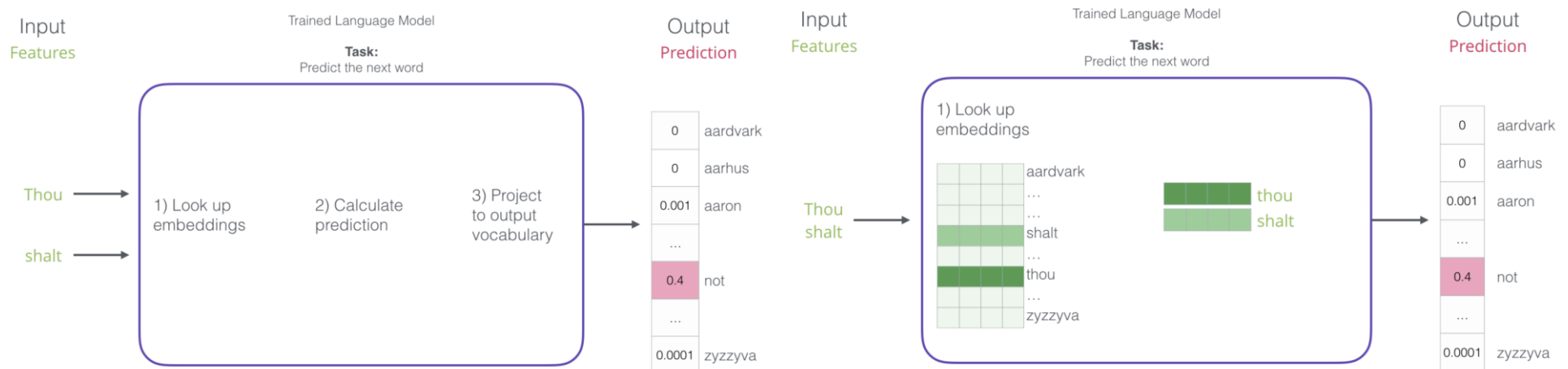
Word Embeddings as Part of Language Model

language models contain embedding matrix as part of learned parameters

- can be extracted and subsequently used as pre-trained embeddings for other task
- typically several hundred dimensions for word vectors
- trained on huge data sets (millions in vocabulary)



next-word
prediction:



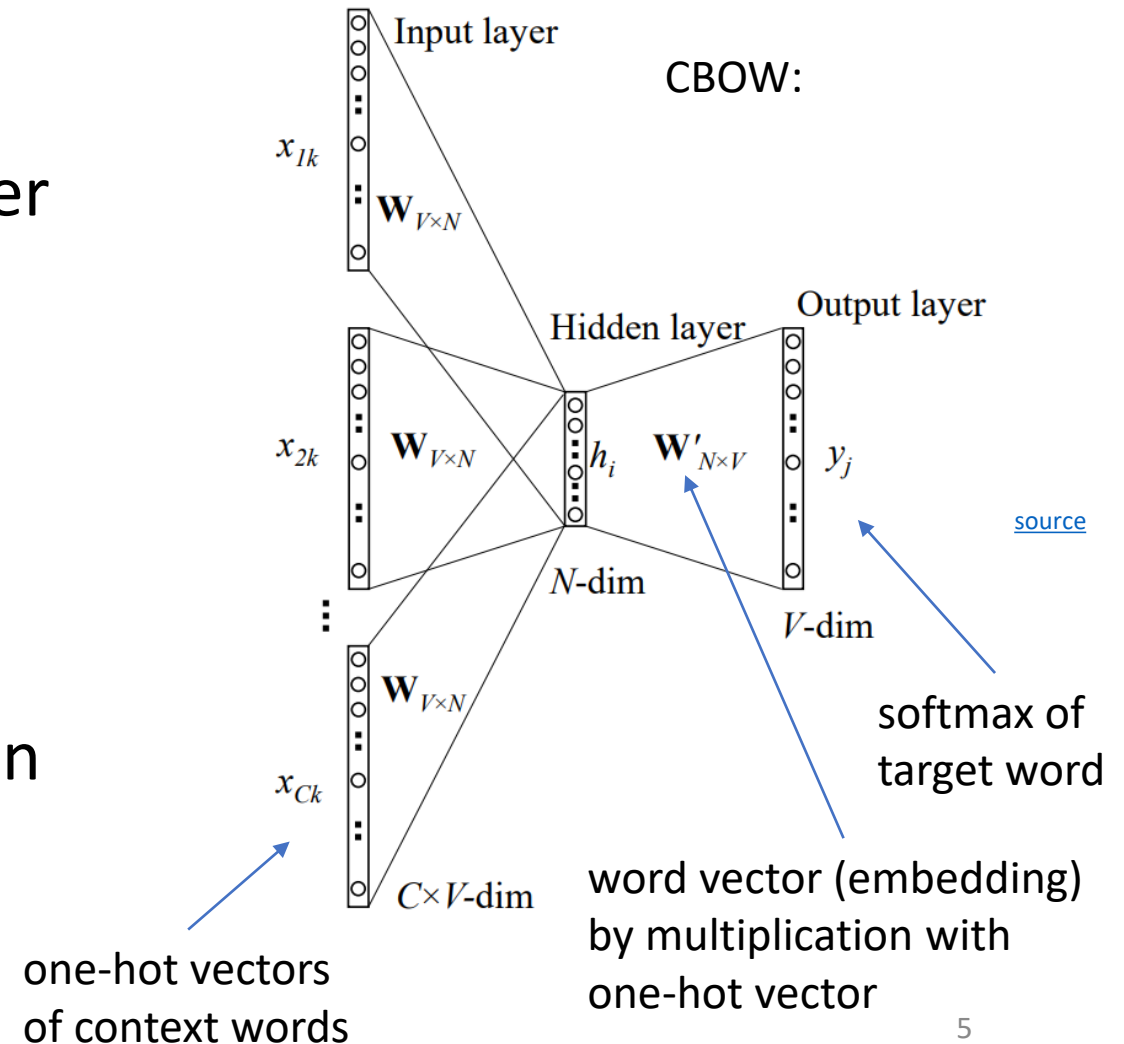
[source](#)

Some Thoughts on Word Embeddings

can be implemented as

- neural network with single hidden layer (linear activation)
- using, e.g., bag-of-words approach (predict masked word from its surroundings)

→ not context-aware (need for attention or RNN)



Context Awareness: Transformer

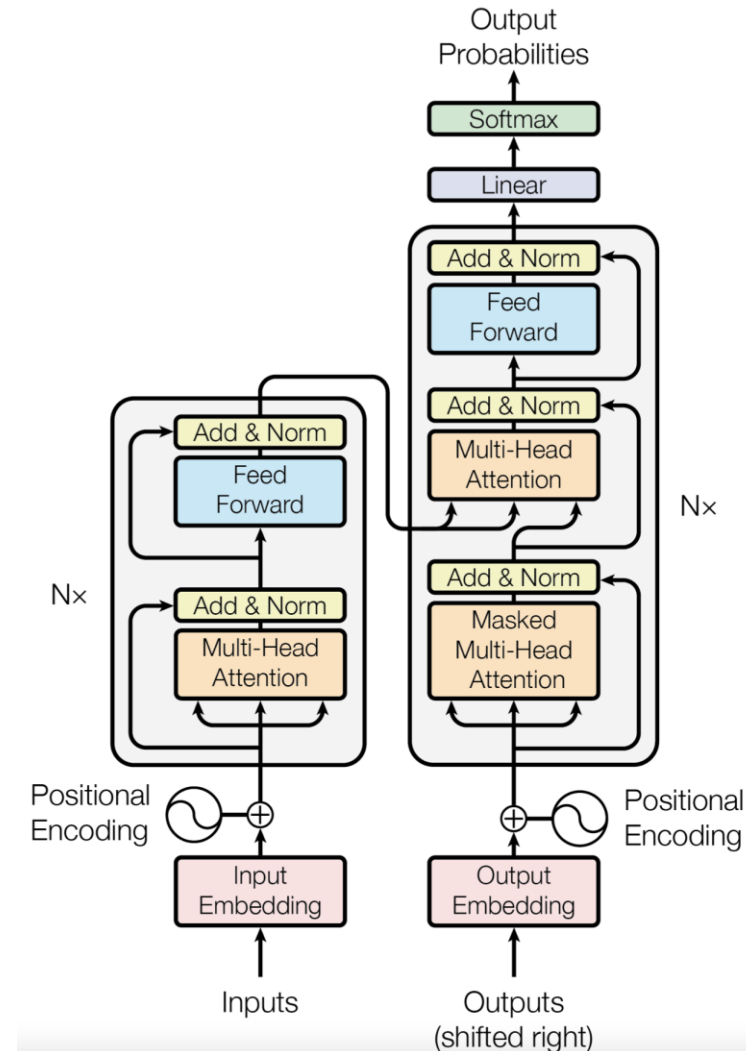
attention is all you need: RNNs replaced by multi-headed self-attention (implemented with matrix multiplications and feed-forward neural networks)

- allowing for much more parallelization
- allowing for bigger models (more parameters)

better long-range dependencies thanks to shorter path lengths in network (less sequential operations)

Let's go through it step by step ...

original transformer: sequence-to-sequence model (e.g., for machine translation)



[source](#)

Tokenization and Embeddings

tokenization: *breaking text in chunks*

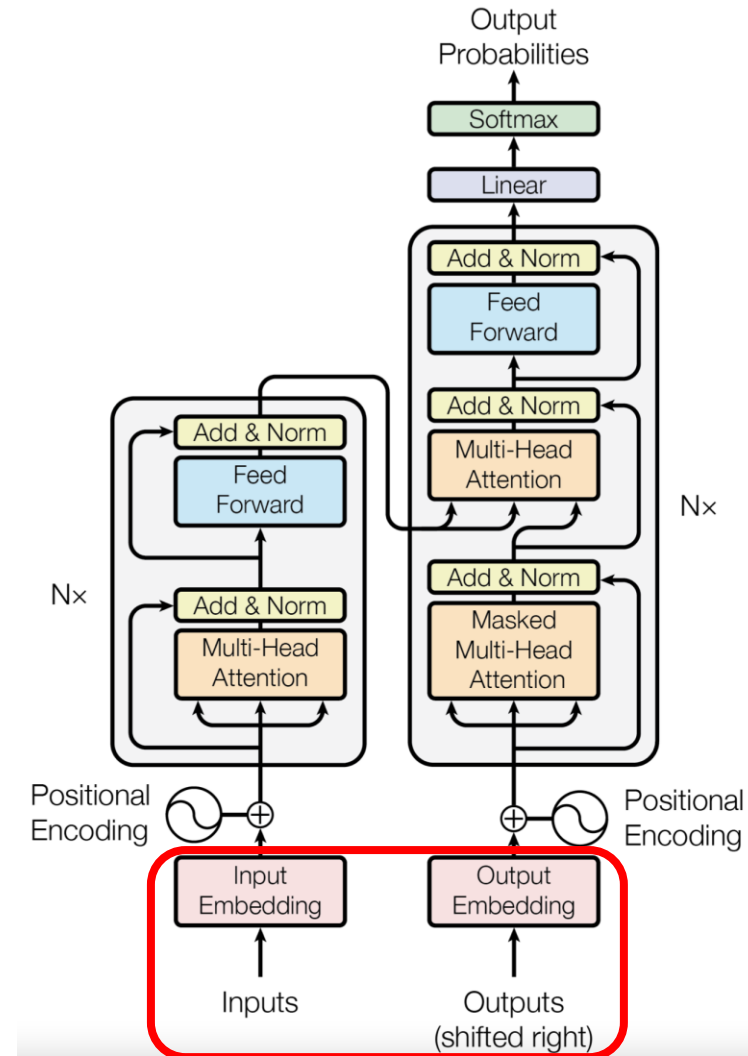
- word tokens: different forms, spellings, etc → undefined and vast vocabulary (need for stemming, lemmatization)
- character tokens: not enough semantic content (longer sequences)

→ byte-pair encoding as compromise for tokenization

one-hot encoding on tokens → token (word) embeddings:
only before bottom-most encoder/decoder



[source](#)



[source](#)

Byte-Pair Encoding

data compression method used for encoding text as sequence of tokens

- merging token pairs (starting with characters) with maximum frequency
 - continue merging until defined fixed vocabulary size (hyperparameter) is reached
- common words encoded as single token
- rare words encoded as sequence of tokens (representing word parts)

alternative: direct operation on bytes (e.g., [ByT5](#))

aaabdaaabc

ZabdZabc
Z=aa

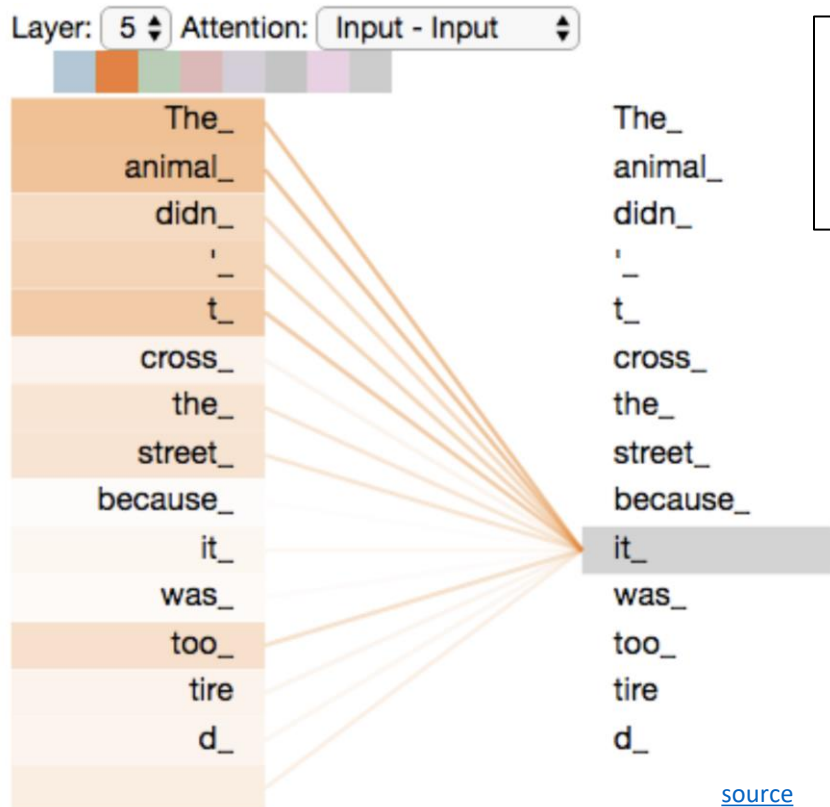
ZYdZYac
Y=ab
Z=aa

XdXac
X=ZY
Y=ab
Z=aa

example from wikipedia

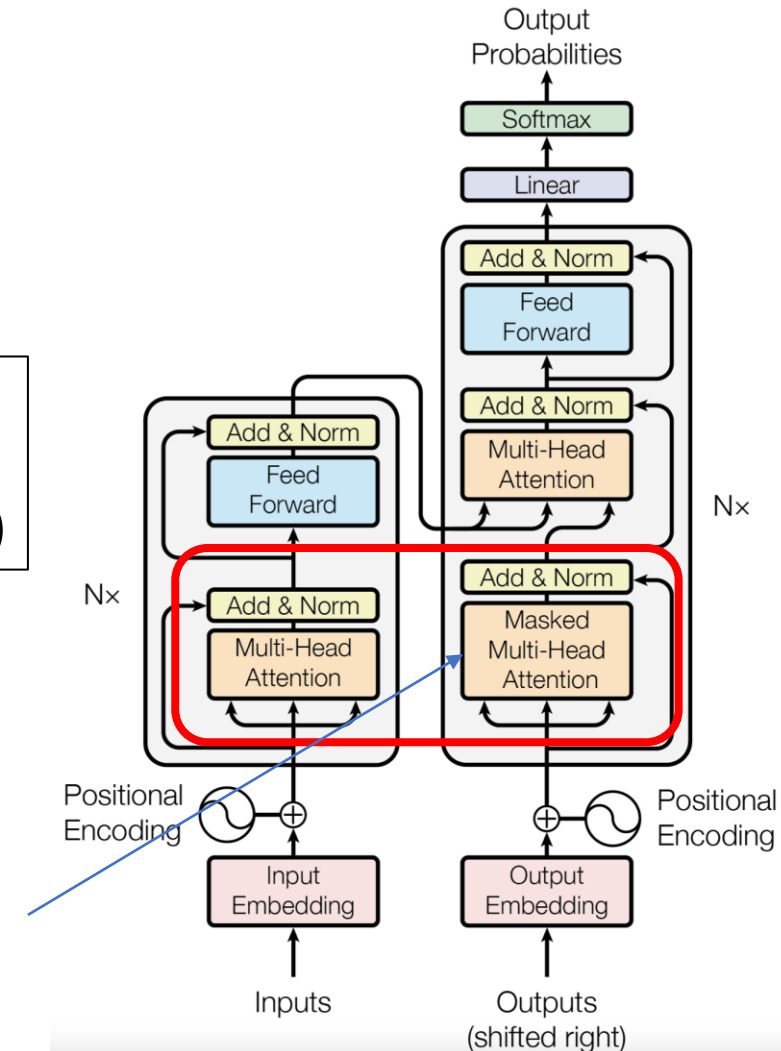
Self-Attention

evaluating other input words in terms of relevance for encoding of given word



computational complexity
quadratic in length of input (each
token attends to each other token)

masked self-attention in
decoder: only allowed to
attend to earlier positions in
output sequence (masking
future positions by setting
them to $-\infty$)

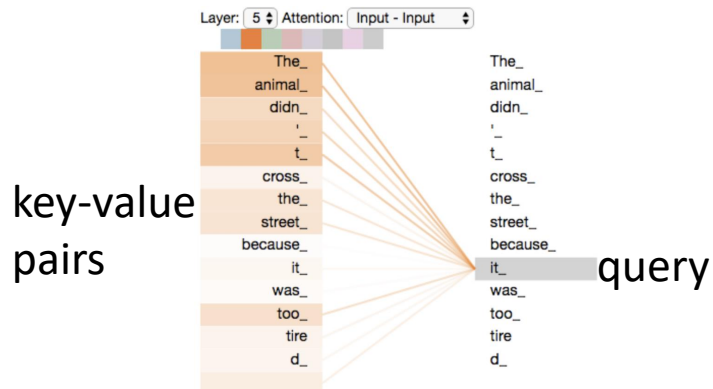


[source](#)

Scaled Dot-Product Attention

3 abstract matrices created from inputs (e.g., word embeddings) by multiplying inputs with 3 different weight matrices

- query Q
- key K
- value V



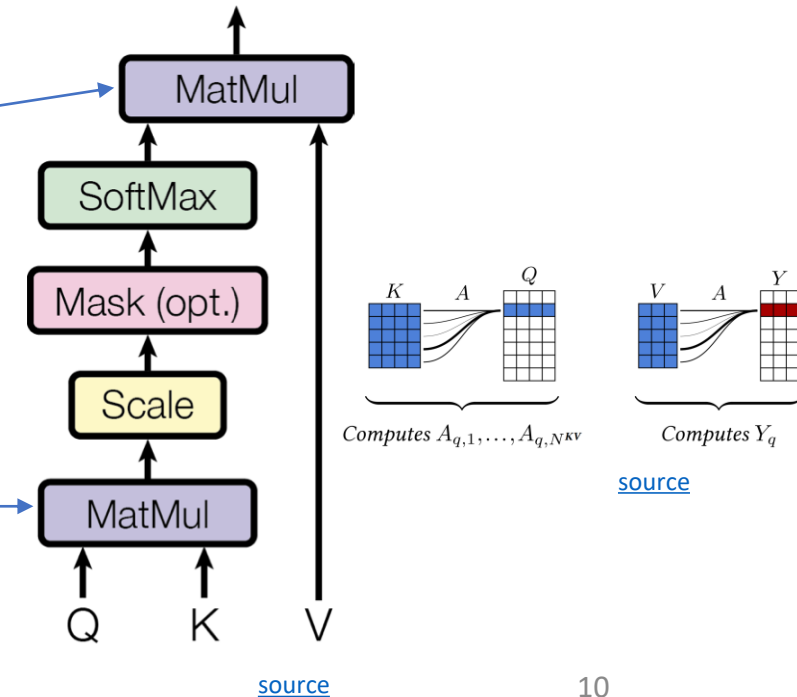
filtering: multiplication of attention probabilities with corresponding key word values

scoring each of the key words (context) with respect to current query word: multiplication of inputs (in contrast to inputs times weights in neural networks)

softmax not scale invariant: largest inputs dominate output for large inputs (more embedding dimensions d_k)

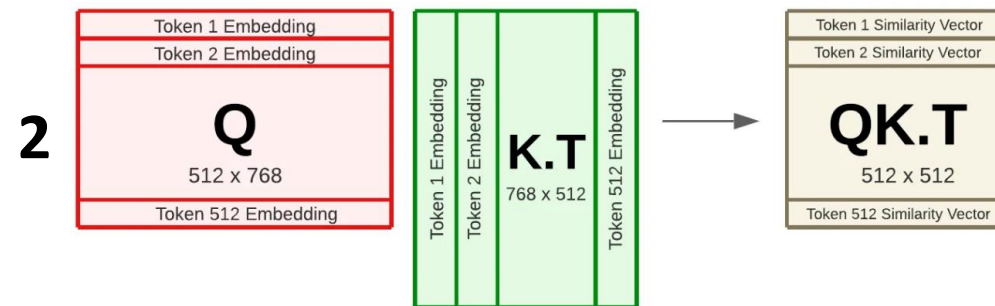
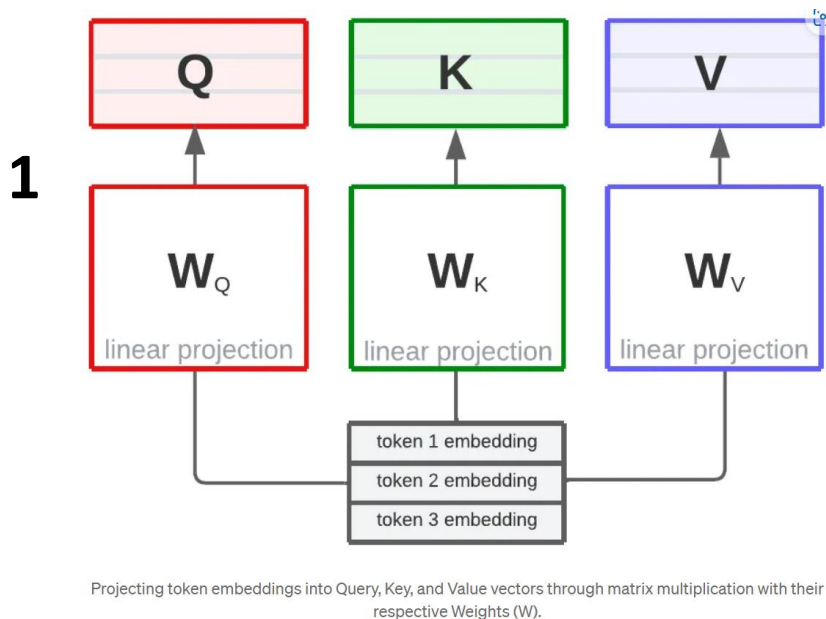
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Scaled Dot-Product Attention

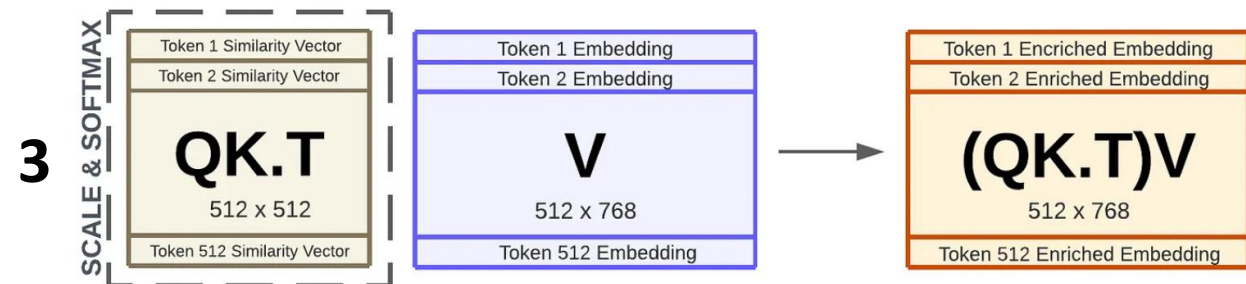


Self-Attention as Weighted Average

weighted average: reflecting to what degree a token is paying attention to the other tokens in the sequence



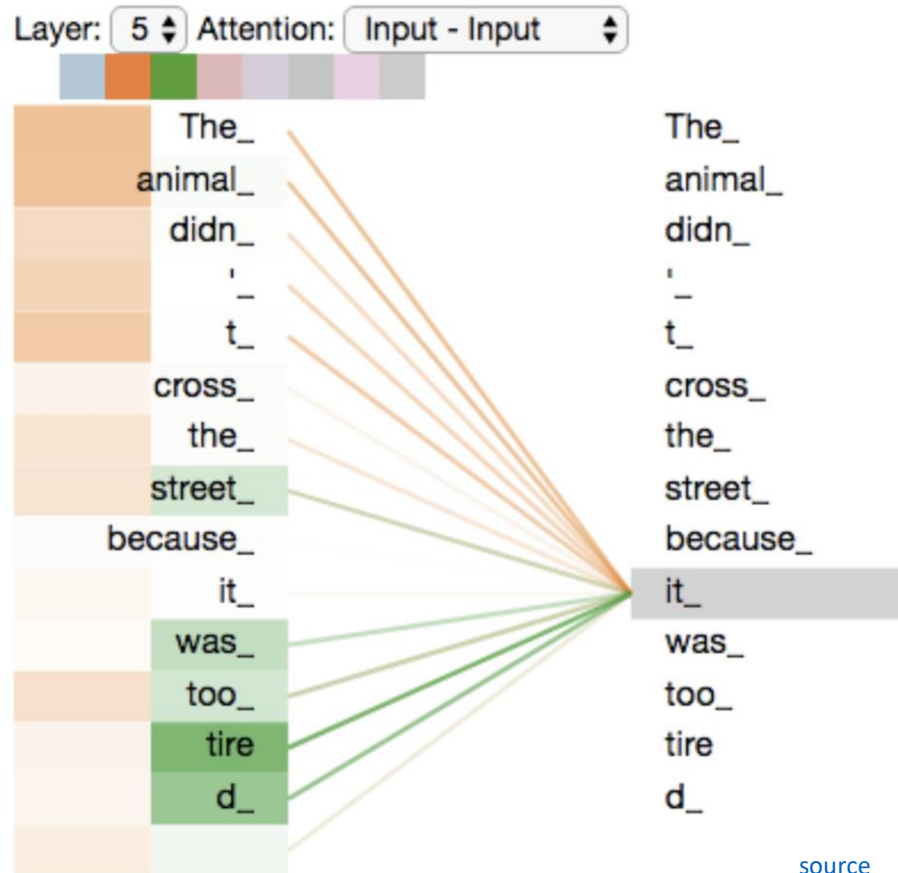
The Query (Q) matrix multiplied with the Key (K.T) resulting in the matrix of similarity of scores (QK.T).



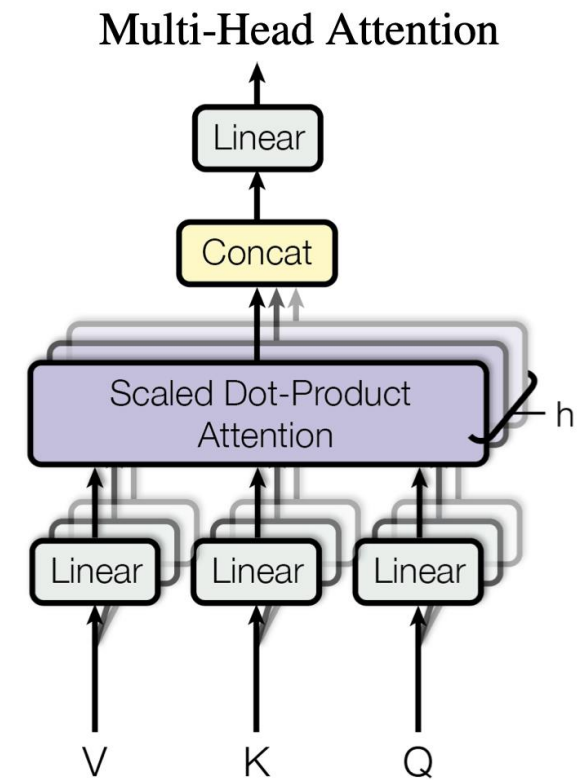
The scaled and soft similarity scores matrix multiplied with the values (V) resulting in enriched embeddings.

Multi-Head Attention

multiple heads: several attention layers running in parallel



different heads can pay attention to different aspects of input (multiple representation sub-spaces)



[source](#)

Involved Matrix Calculations

parameters
to be learned

1) This is our
input sentence*

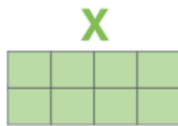
2) We embed
each word*

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

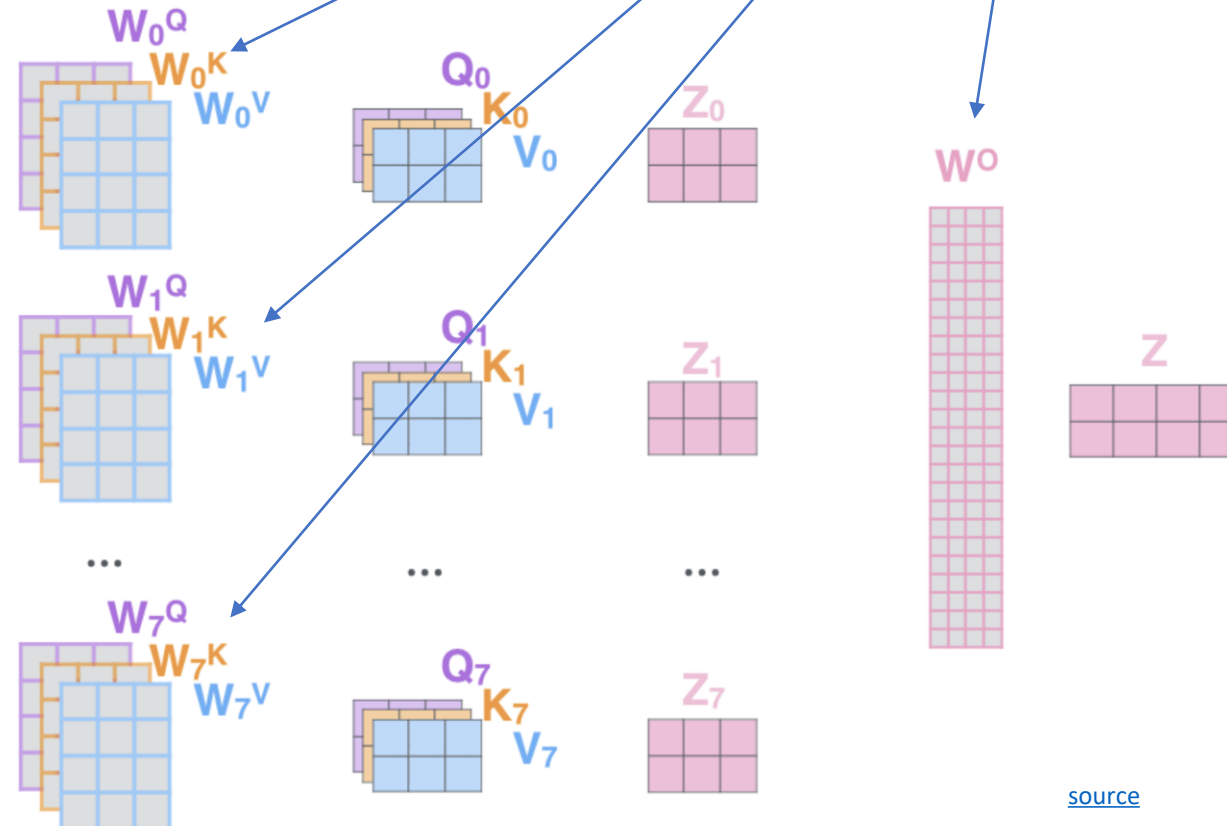
4) Calculate attention
using the resulting
 $Q/K/V$ matrices

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

Thinking
Machines



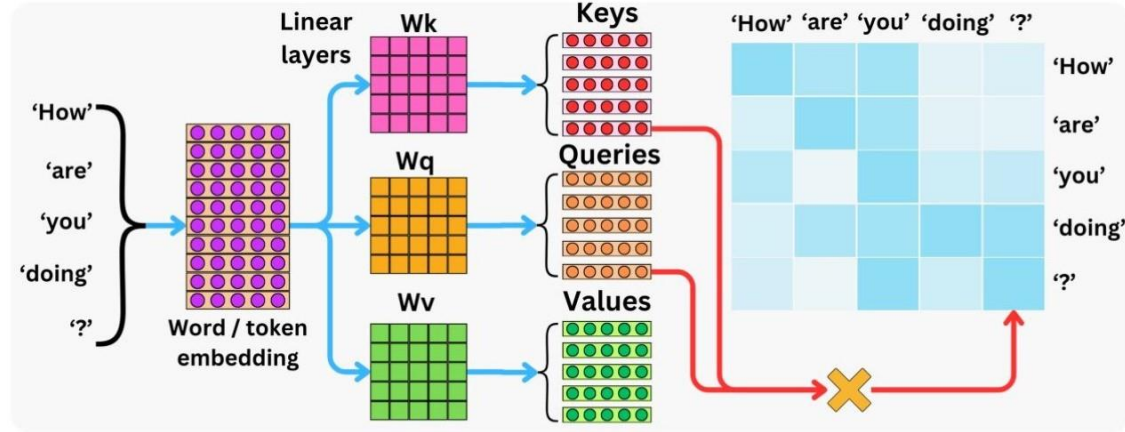
* In all encoders other than #0,
we don't need embedding.
We start directly with the output
of the encoder right below this one



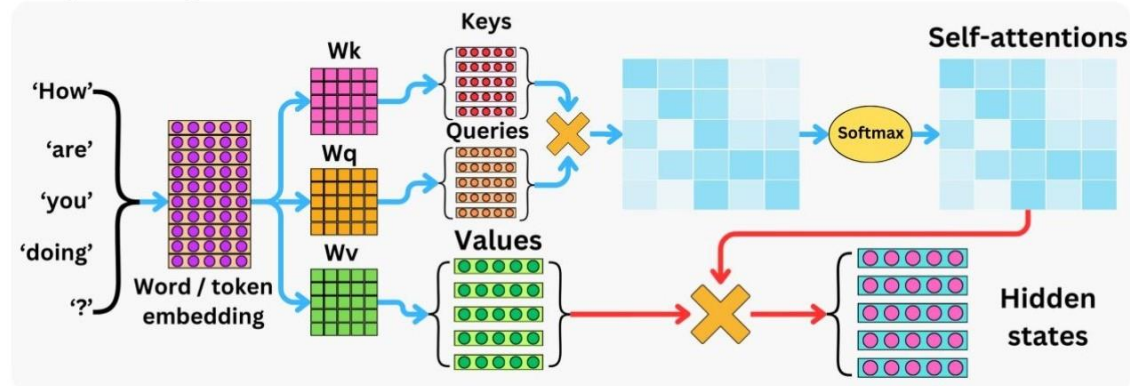
[source](#)

Attention is all you need!

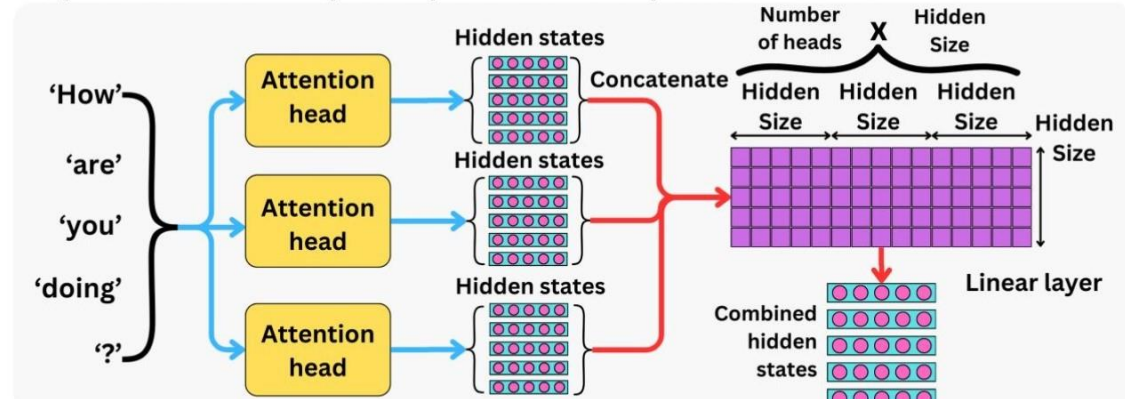
Step 1: Compute the scores that captures the token-token interaction



Step 2: Compute the hidden states



Step 3: Combine multiple outputs from multiple Attention heads



Encoder-Decoder Attention

aka cross-attention

connection between encoders and decoders

attention layer helping decoder to focus on relevant parts of input sentence (similar to attention in seq2seq models)

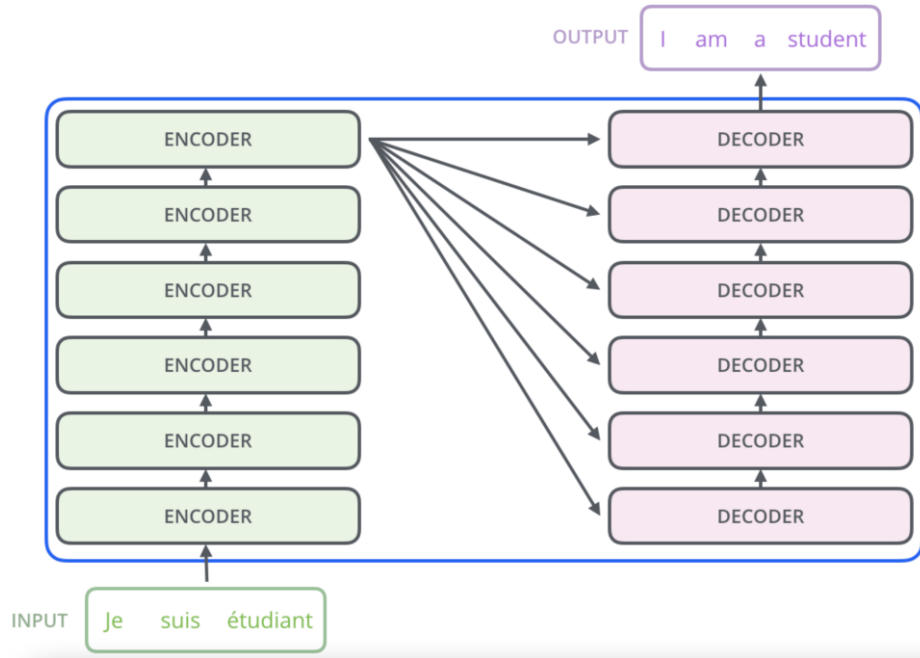
output of last encoder transformed into set of attention matrices K and $V \rightarrow$ fed to each decoder's cross-attention layer (redundancy)

multiheaded self-attention with Q from decoder layer below and K, V from output of encoder stack

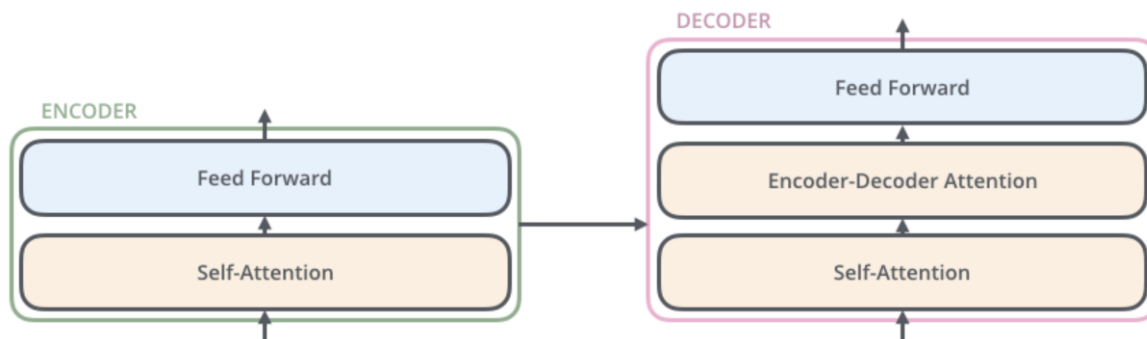


[source](#)

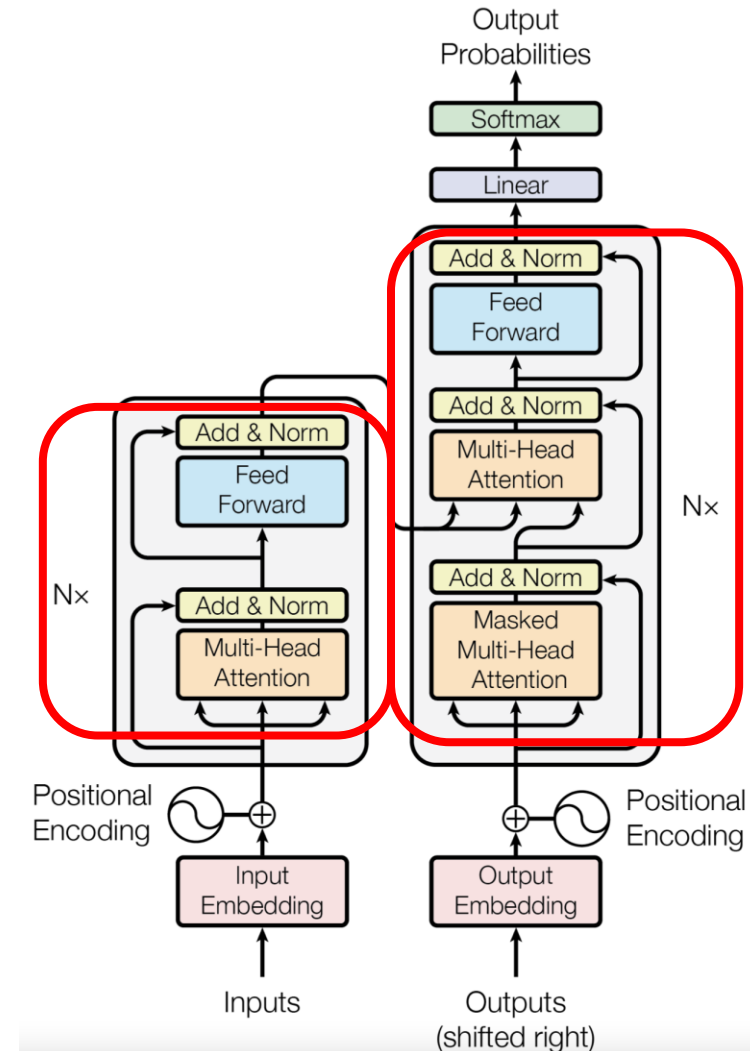
Encoder and Decoder Stacks



output of encoders/decoders
fed as input to next ones



[source](#)



[source](#)

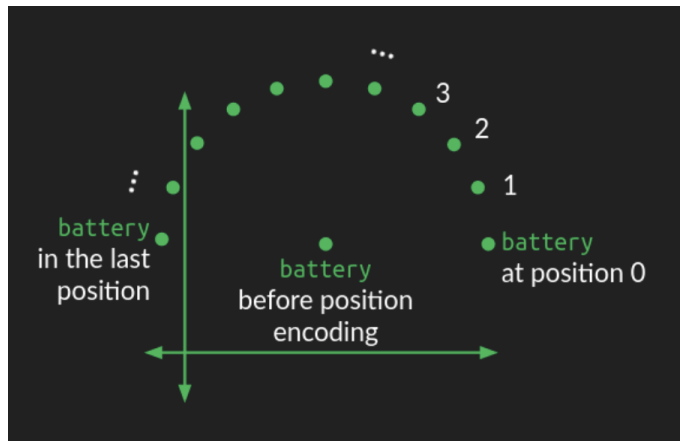
Positional Encoding

attention permutation invariant → need for positional encoding to learn from order of sequence

added to input embeddings (same dimension d_{model})

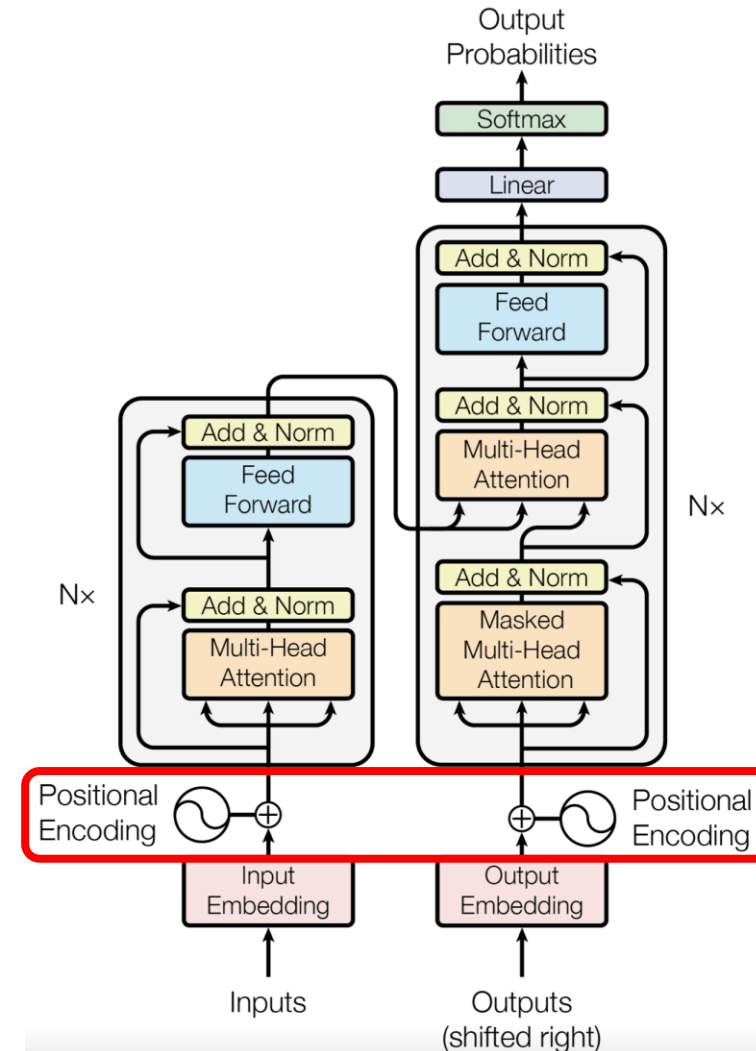
different choices for positional encoding:

- learned (by including absolute position in embedding)
- fixed, e.g., sine/cosine functions for each dimension i



$$PE_{pos,2i} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$
$$PE_{pos,2i+1} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

[source](#)

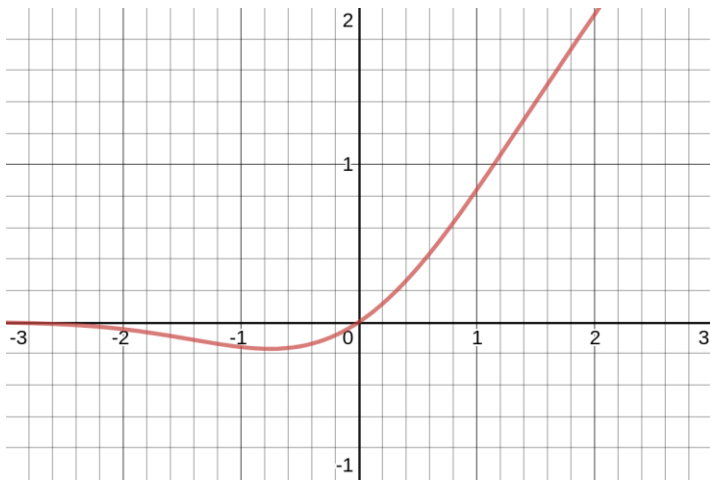


[source](#)

Position-Wise Feed-Forward Networks

for each encoder or decoder layer: identical feed-forward network independently applied to each position

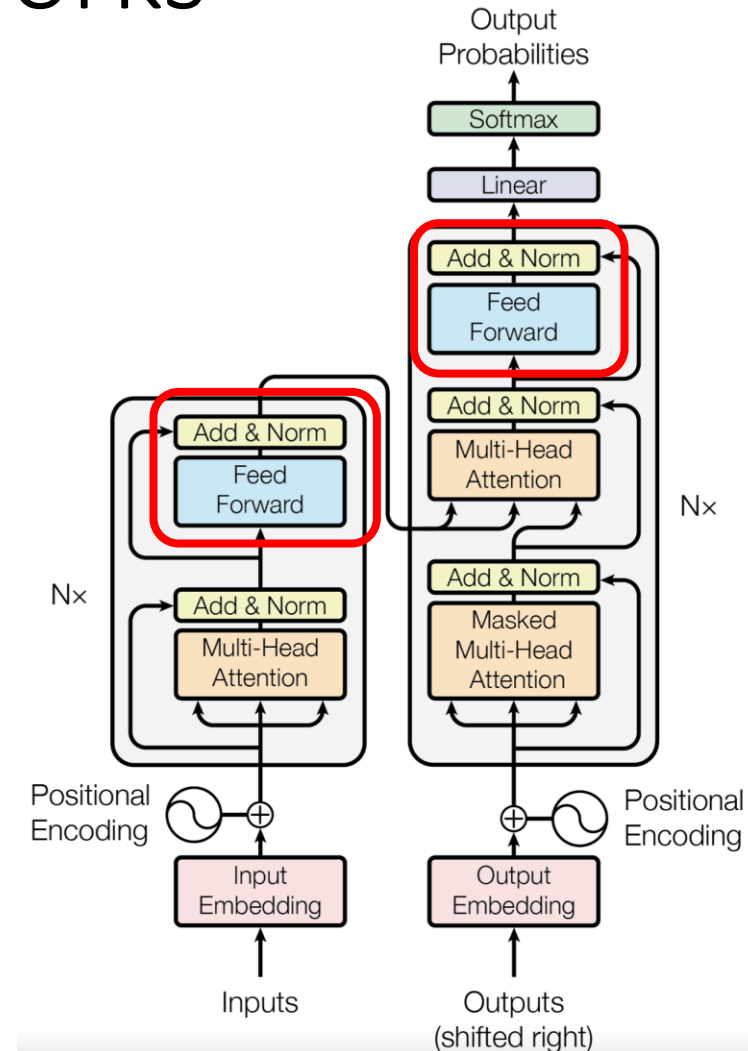
GeLU



from wikipedia

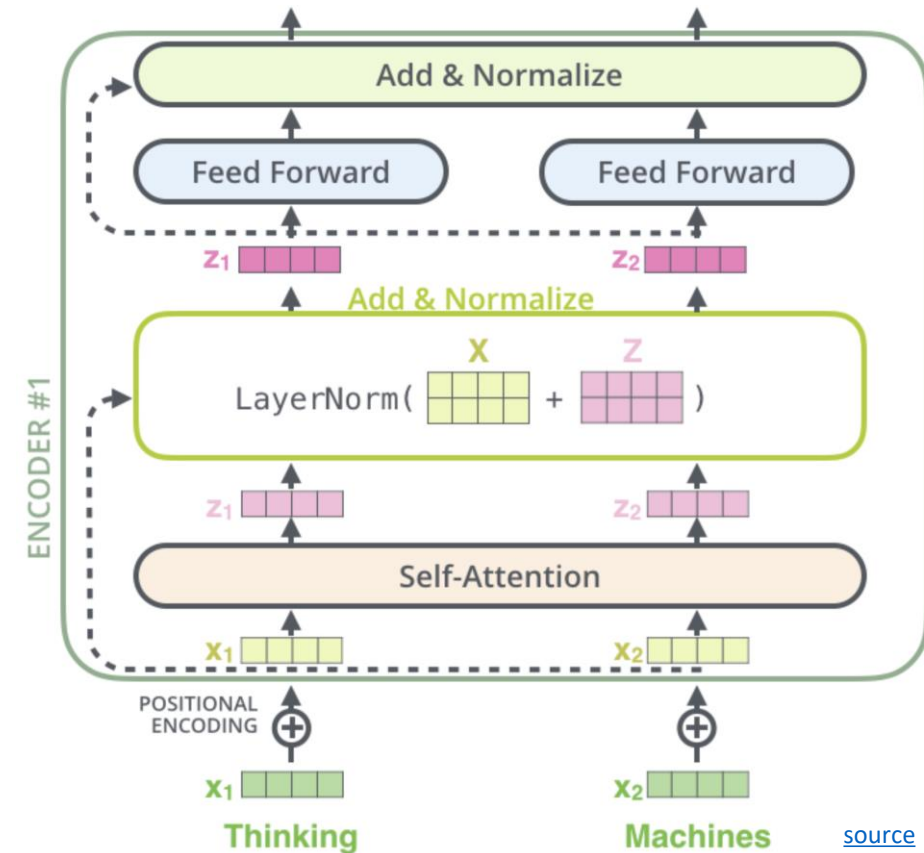
attention is just weighted averaging
→ need for non-linearities (often GeLU activations) to capture more complex patterns

typically expand-and-contract (two layers) network



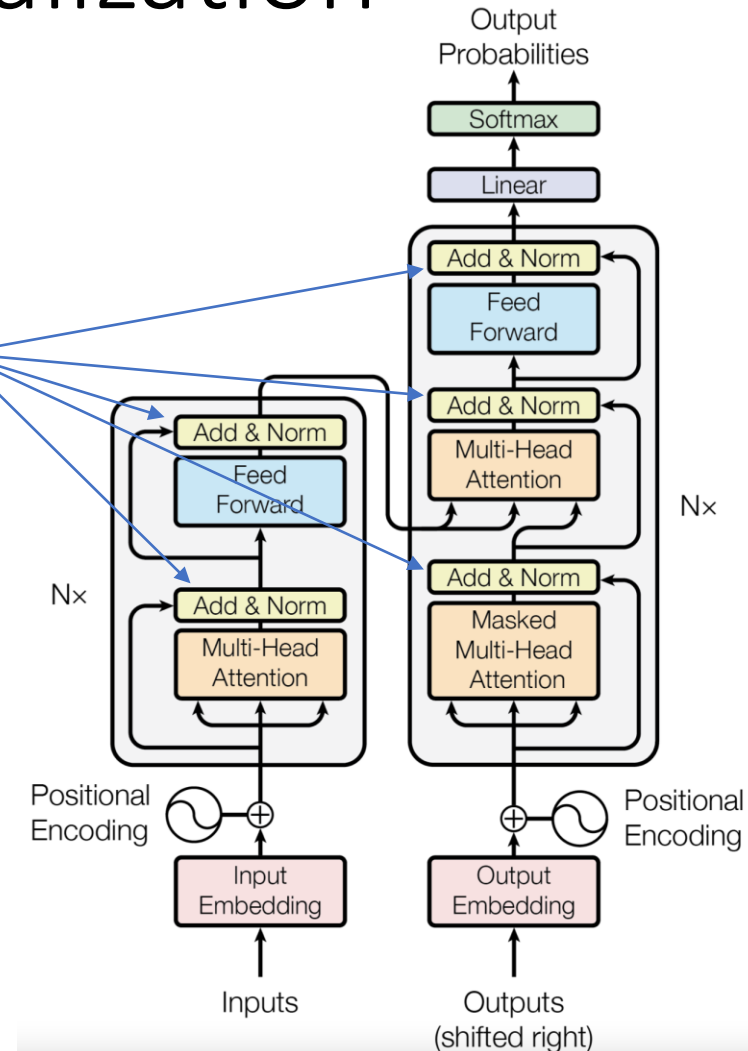
[source](#)

Skip Connections and Layer Normalization



skip connections and layer normalization for each sub-layer

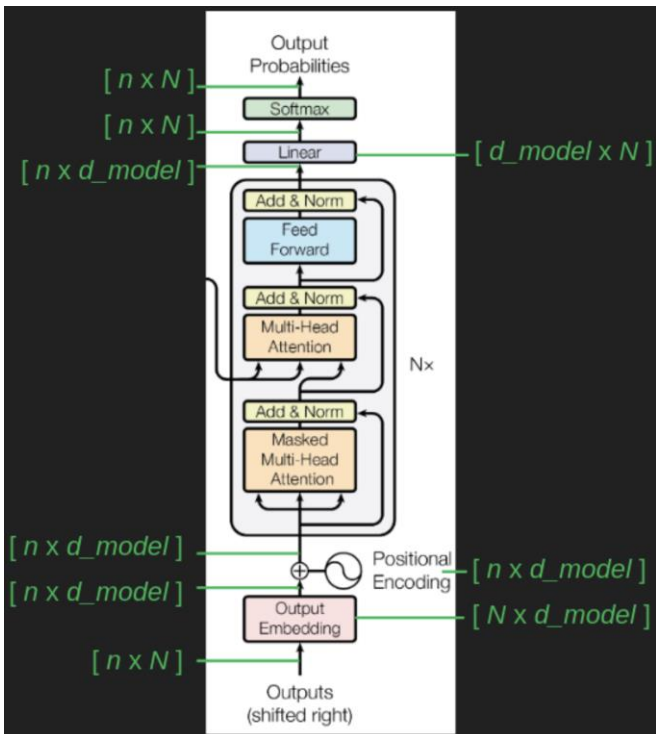
- skip connections improve robustness by preserving original input (attention layers as filters) as well as gradients (mitigate vanishing-gradient problem)
- easier learning of identity functions (useful for disregarding modules that do not improve model performance)



[source](#)

De-Embedding and Softmax

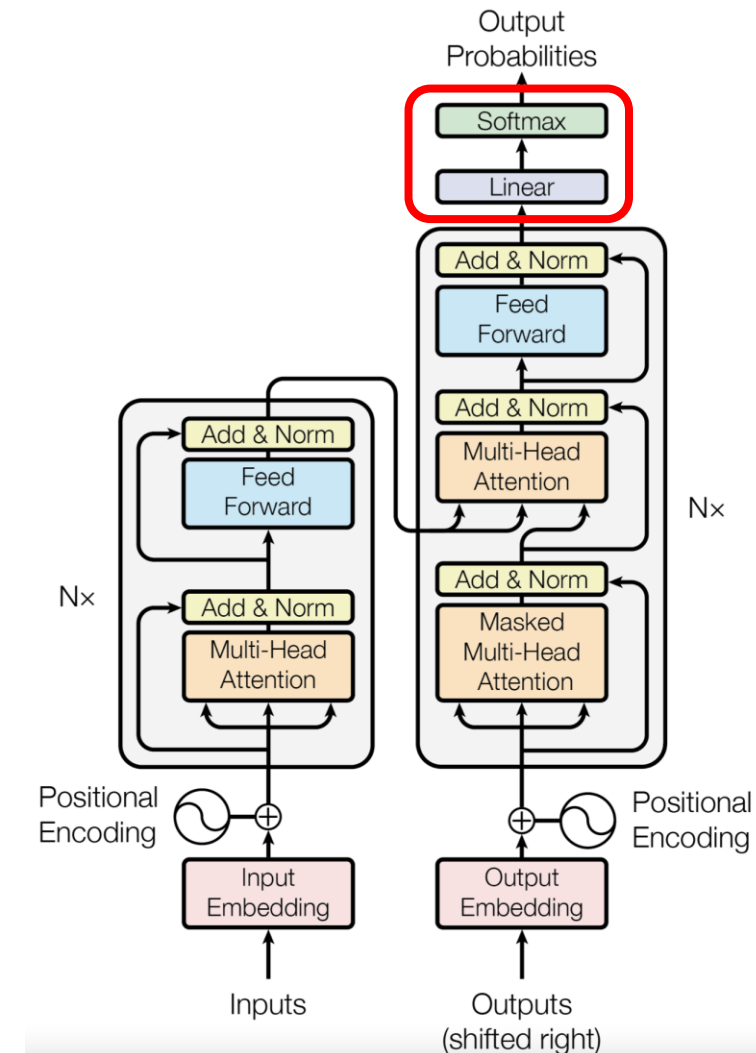
n : maximum sequence length
 N : vocabulary size
 d_{model} : embedding dimensions



conversion of final decoder output to predicted next-token probabilities for output vocabulary

de-embedding: linear transformation (matrix multiplication / fully connected neural network layer)

softmax: transformation to probabilities ("softness" can be controlled by hyperparameter temperature)

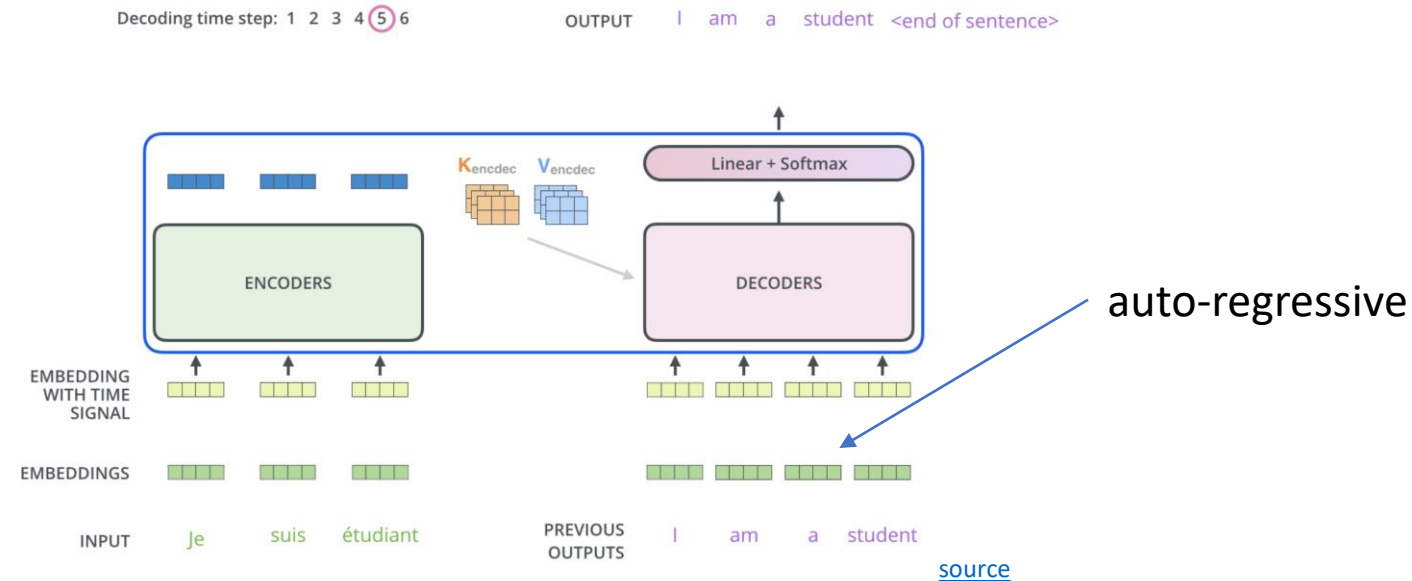


[source](#)

Sequence Completion

- greedily picking the one with highest probability
- pick according to probabilities (degree of randomness controlled by softmax temperature)
- beam search

for each step/token (iteratively), choose one output token to add to decoder input sequence → increasing uncertainty



prompt: externally given initial sequence for running start and context on which to build rest of sequence ([prompt engineering](#))

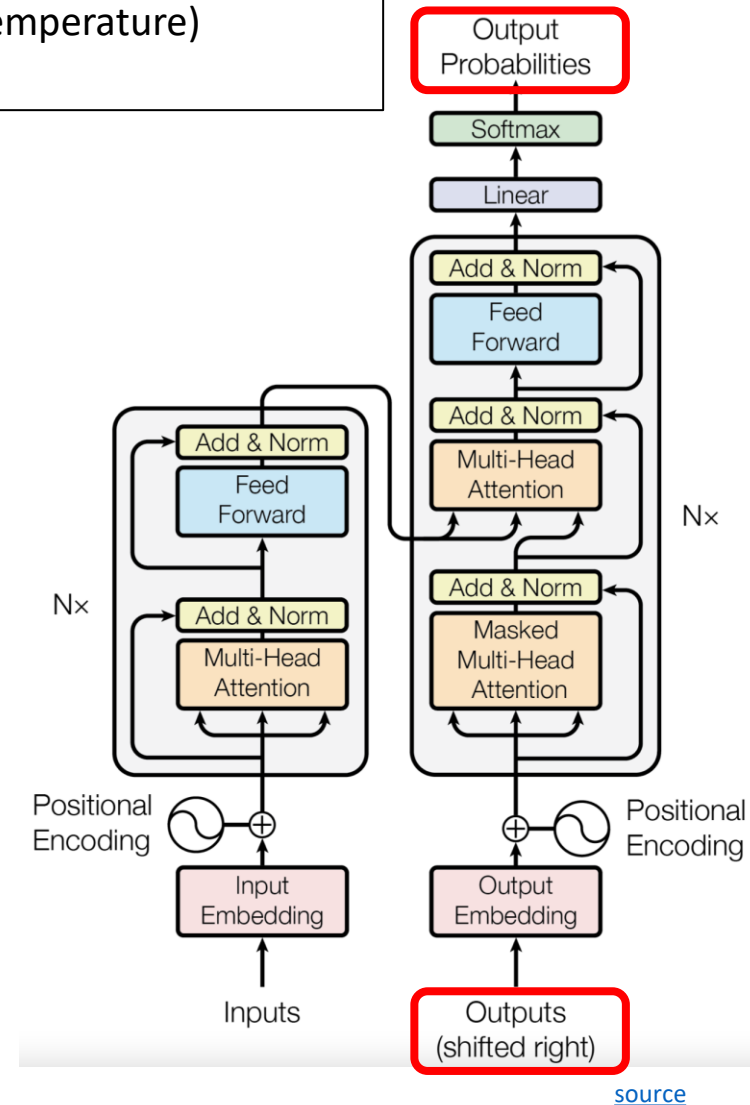


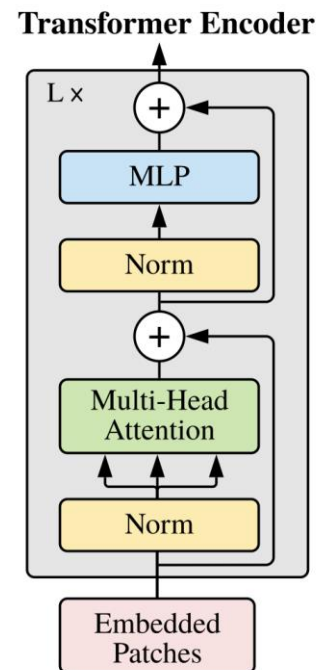
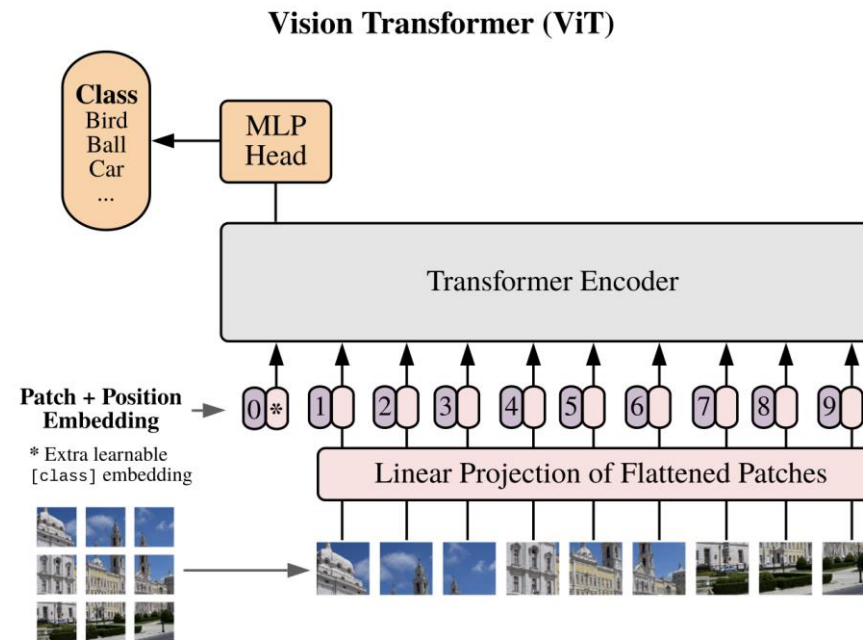
Image Classification with Vision Transformer (ViT)

formulation as sequential problem:

- split image into patches and flatten → use as tokens
- produce linear embeddings and add positional embeddings

processing by transformer encoder:

- pre-train with image labels
- fine-tune on specific data set



[source](#)

Open-Source Implementations

lightweight PyTorch re-implementation of GPT (decoder-only transformer):

[minGPT](#)

more powerful:

[nanoGPT](#), [LitGPT](#)