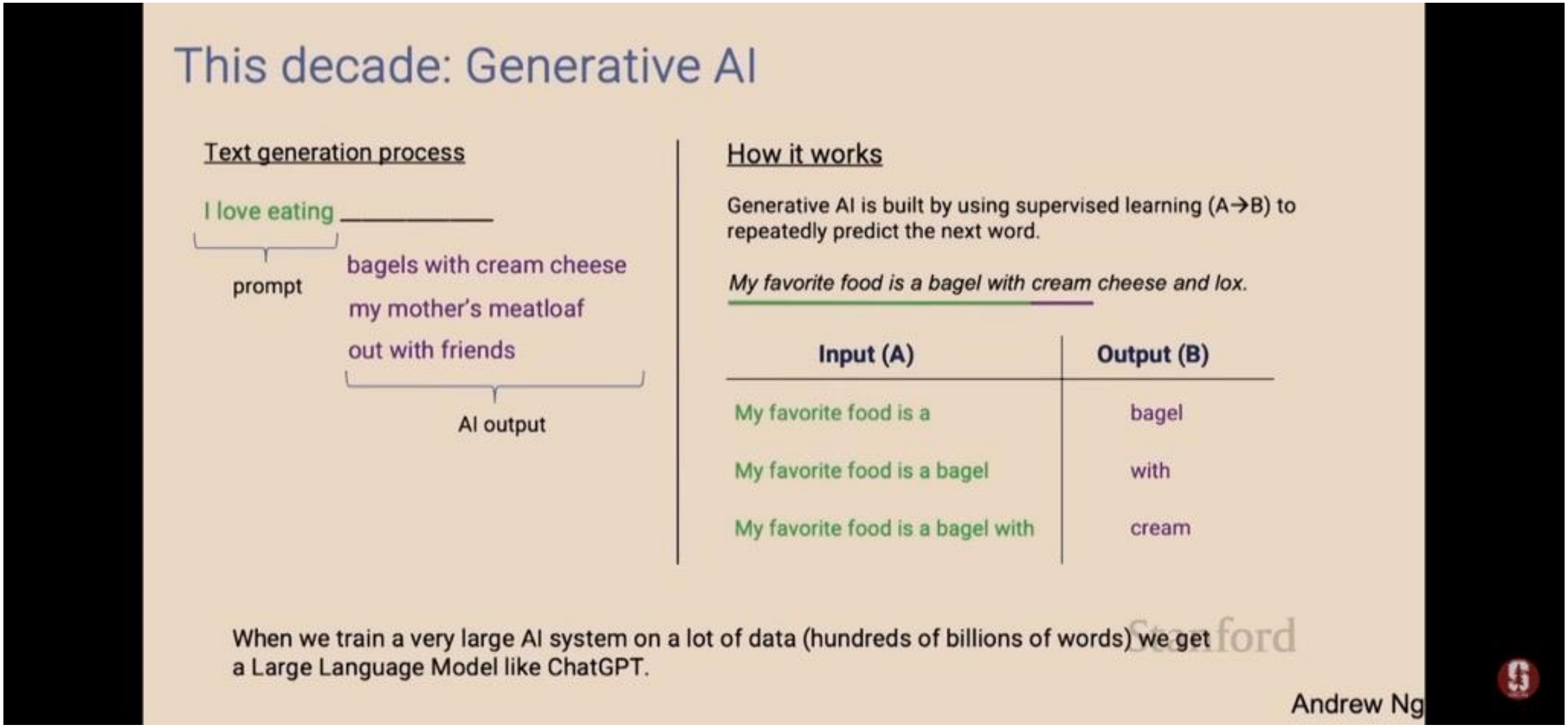
# Transformer

# Language Models

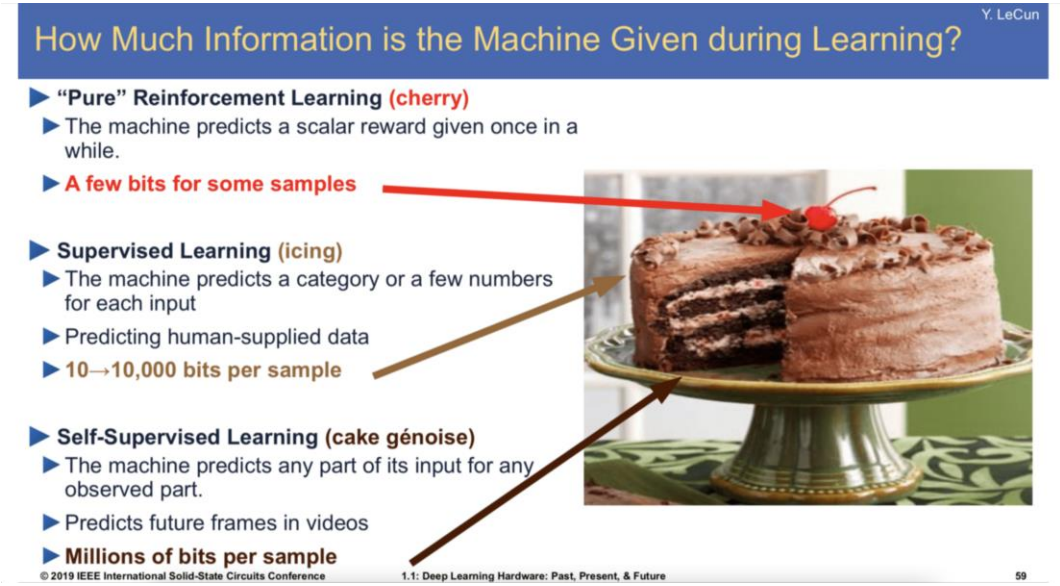


ML needs lots of training data

# Unsupervised Learning

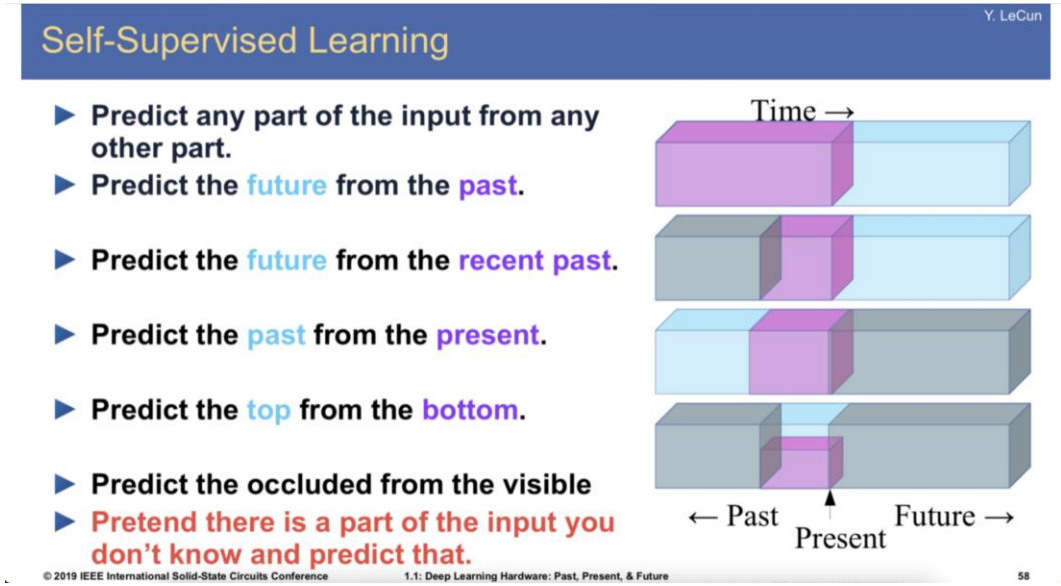
**learning by observation**

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



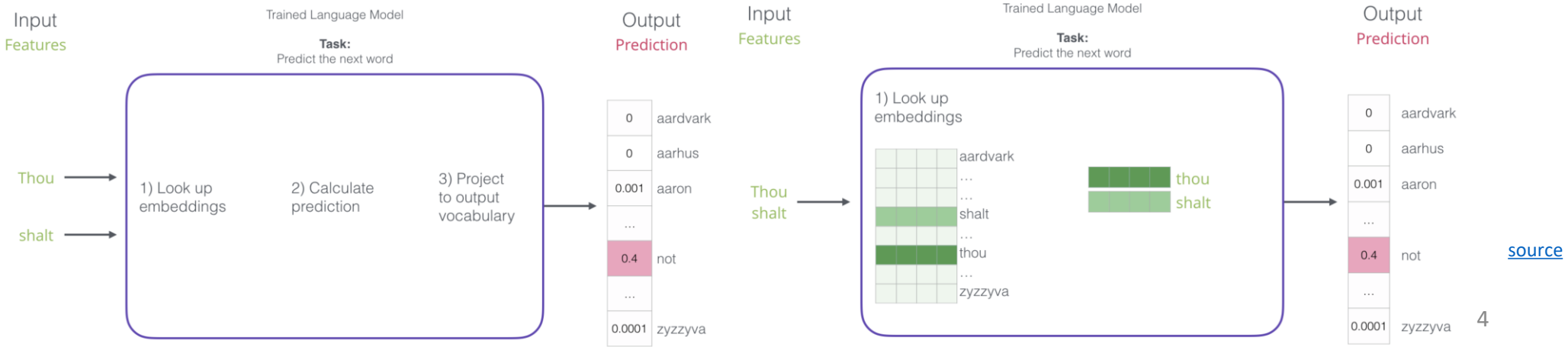

generative AI as unsupervised learning: generate variations of training data

# 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

# Context Awareness: Transformer

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

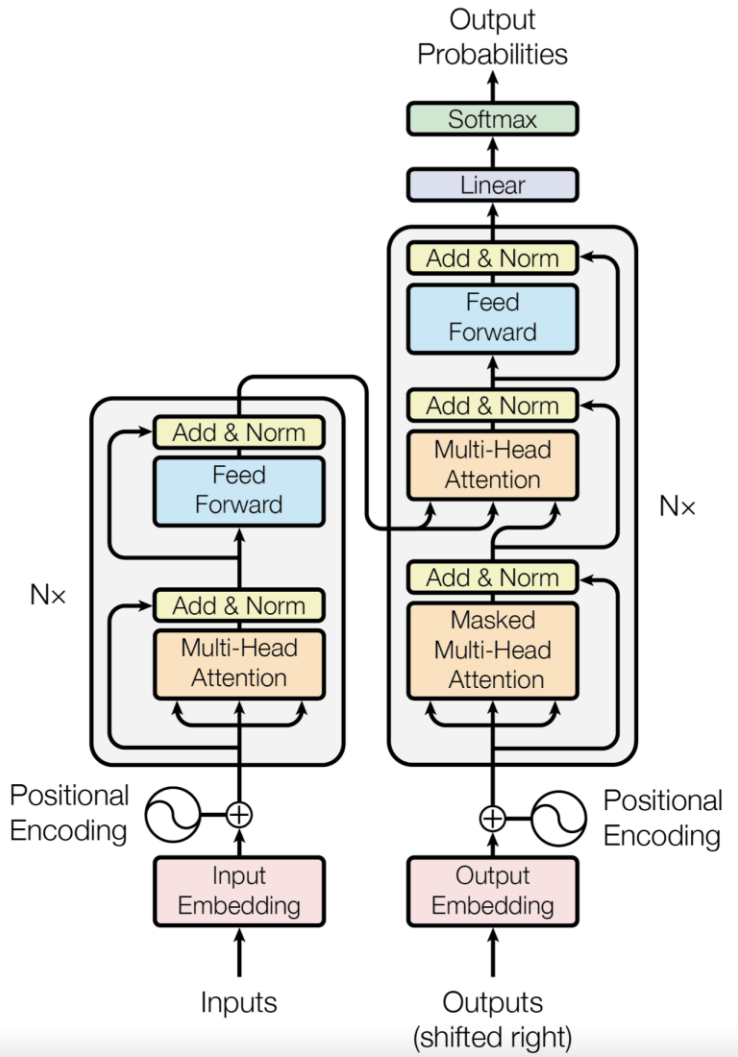
→ allowing for much more parallelization

→ 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

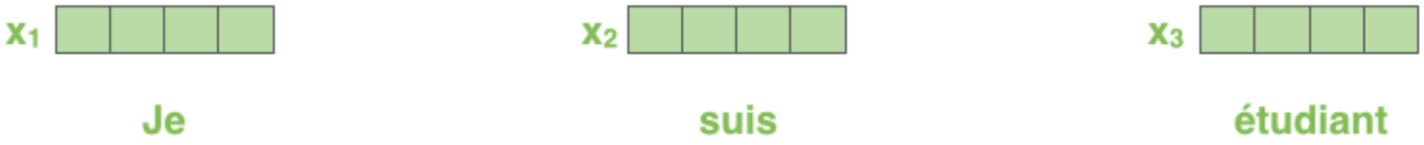
tokenization: *breaking text in chunks*

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

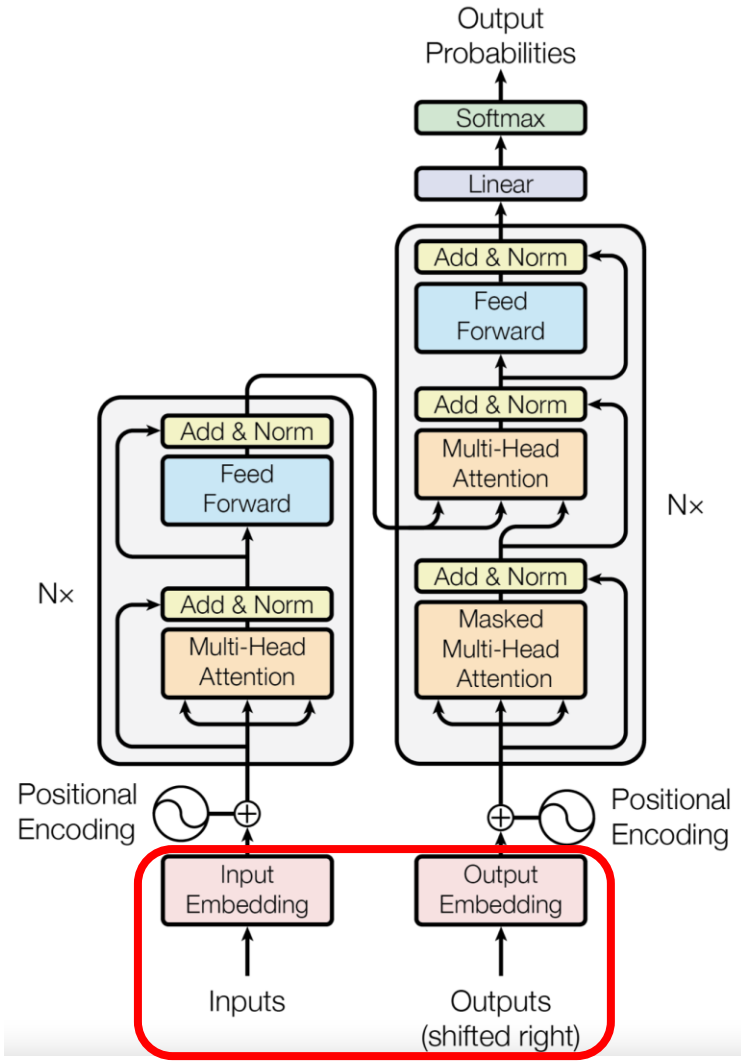
→ byte-pair encoding as compromise for tokenization

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)

aaabdaaabac

ZabdZabac
Z=aa

ZYdZYac
Y=ab
Z=aa

XdXac
X=ZY
Y=ab
Z=aa

example from wikipedia

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

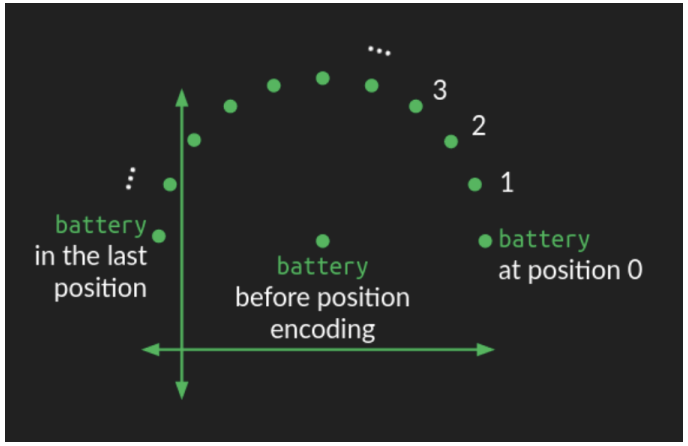
# Positional Encoding

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

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

different choices for positional encoding:

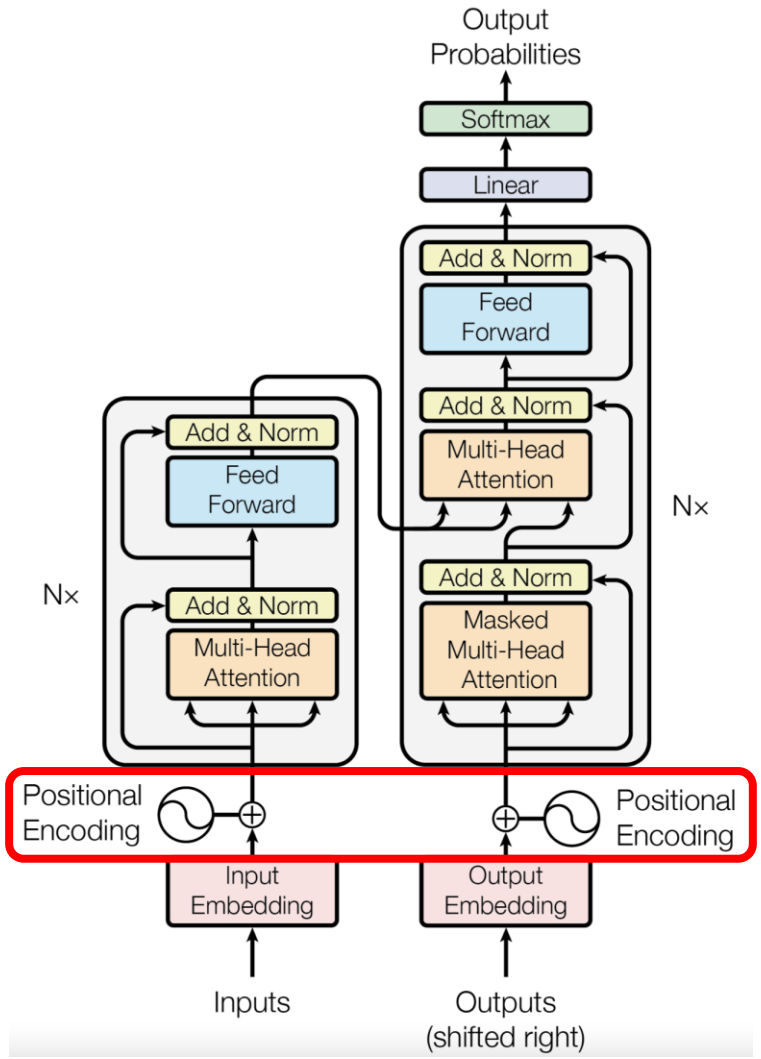
- learned (by including absolute position in embedding)
- fixed, e.g., sine/cosine functions for each dimension $i$




source

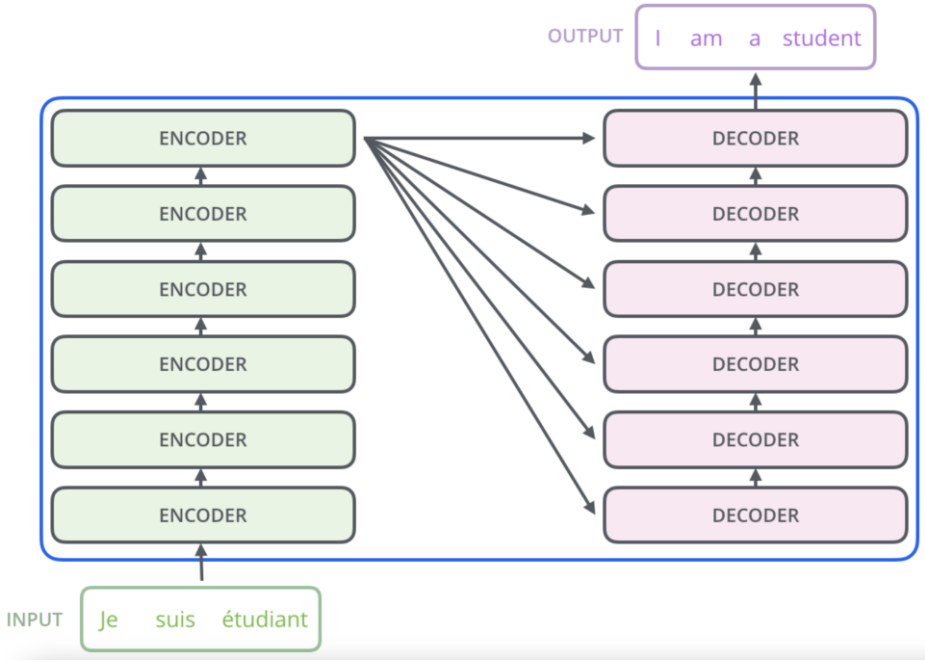
$$PE_{pos,2i} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

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



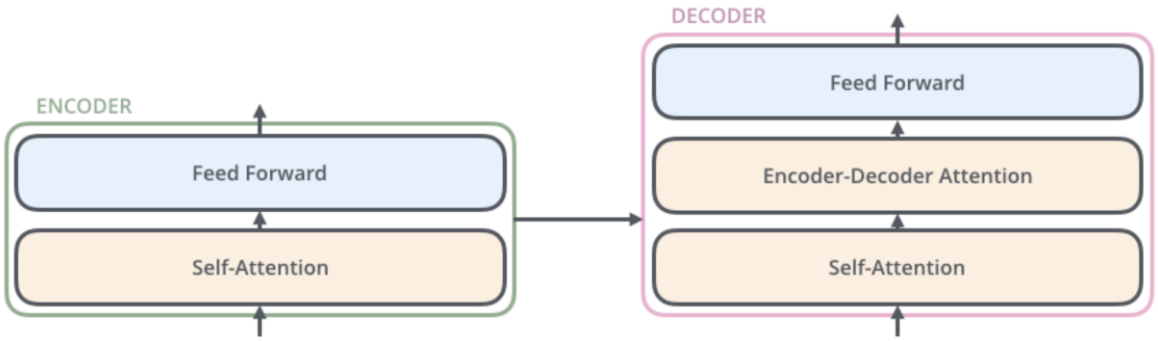

source

# Encoder and Decoder Stacks

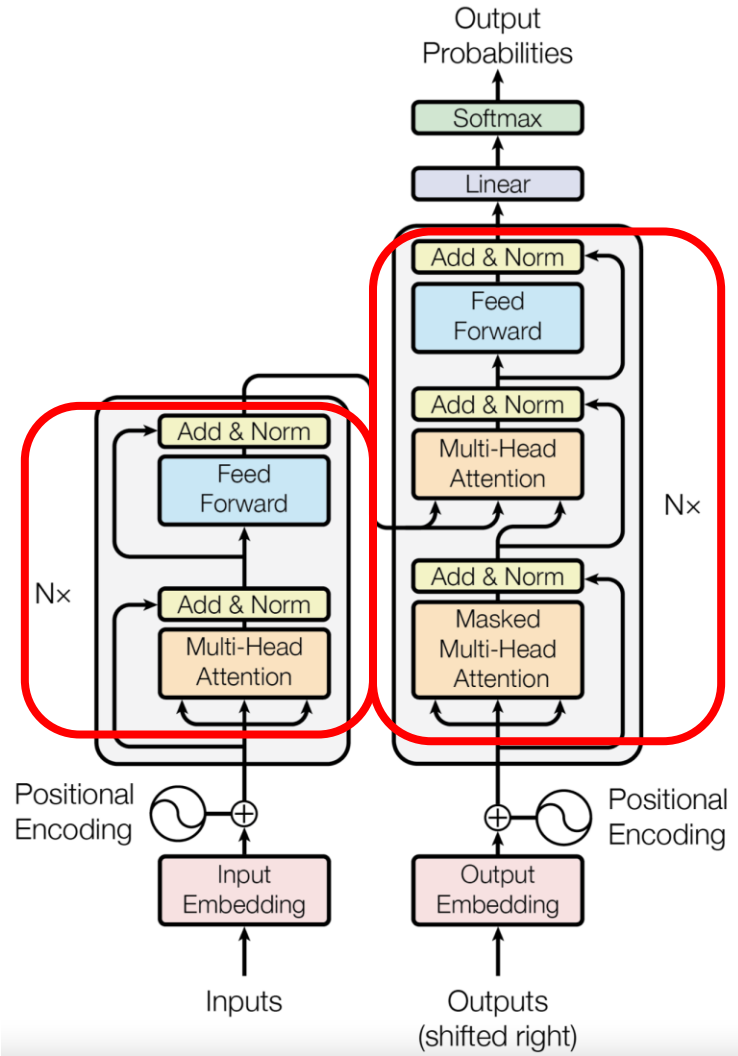



output of encoders/decoders fed as input to next ones

idea of depth:
providing redundancy
(even more important here than increasingly sophisticated abstraction, like, e.g., in CNN)

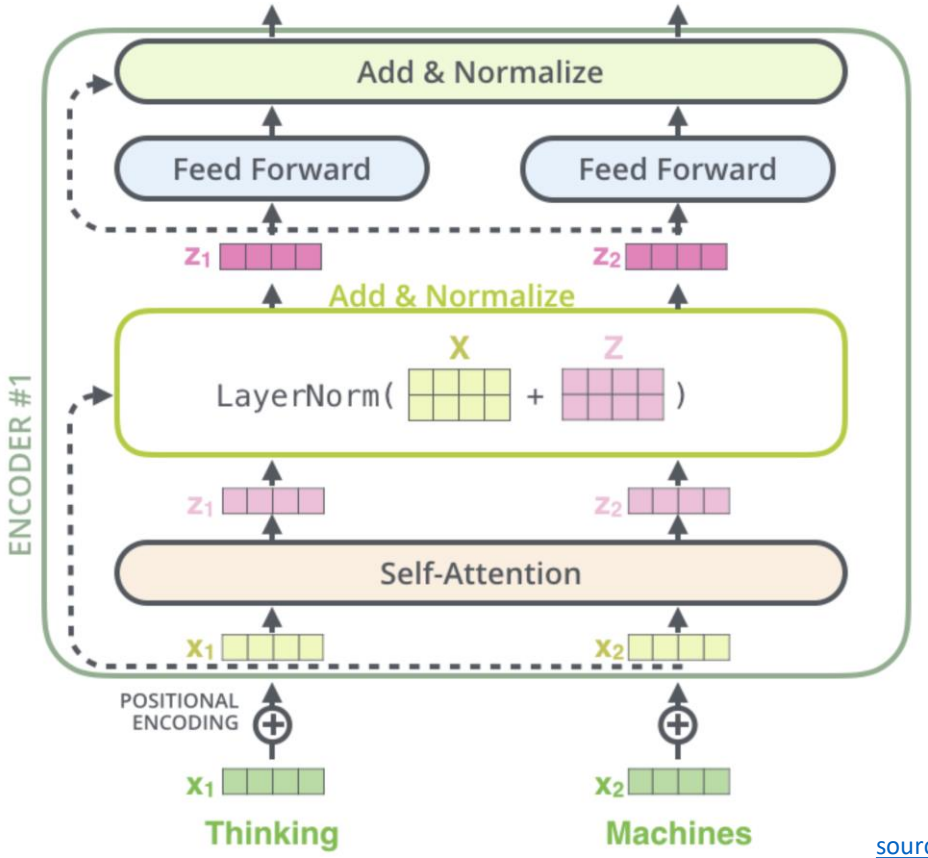



source




source

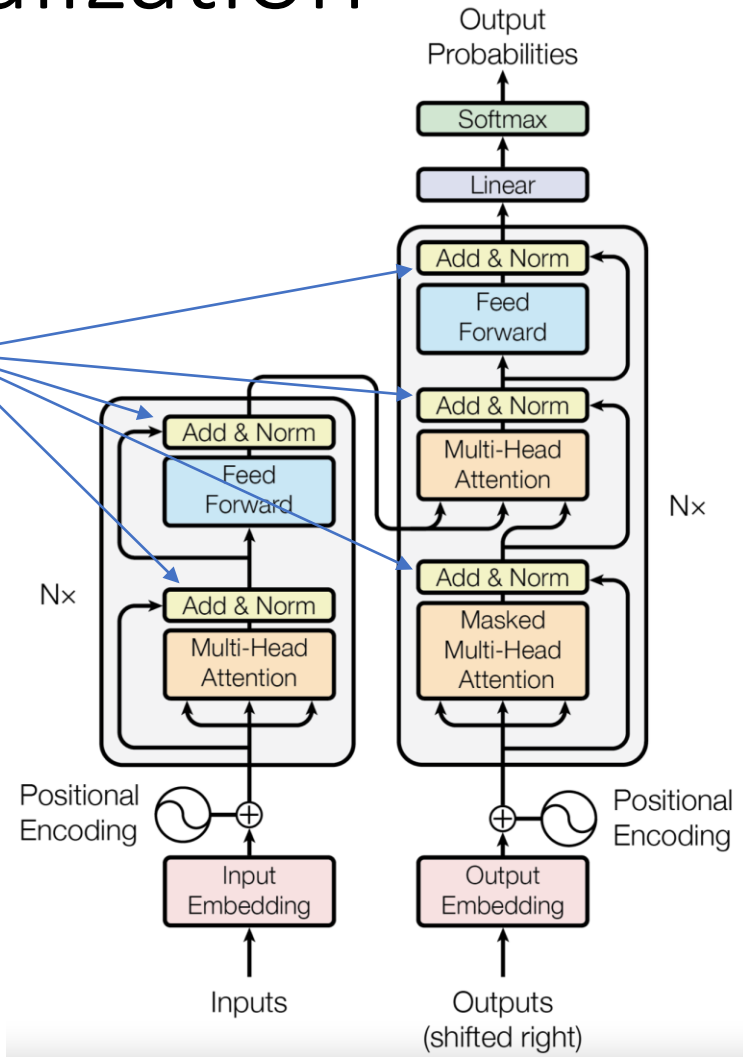
# Skip Connections and Layer Normalization




source

skip connections and layer normalization for each sub-layer

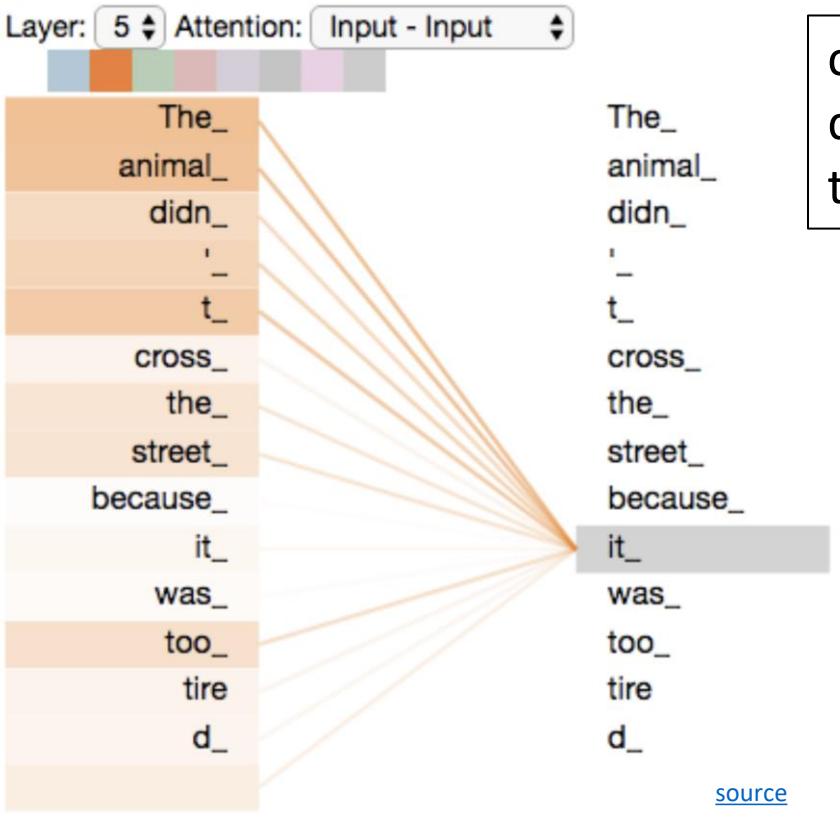
skip connections improve robustness by preserving original input (attention layers as filters)




source

# Self-Attention

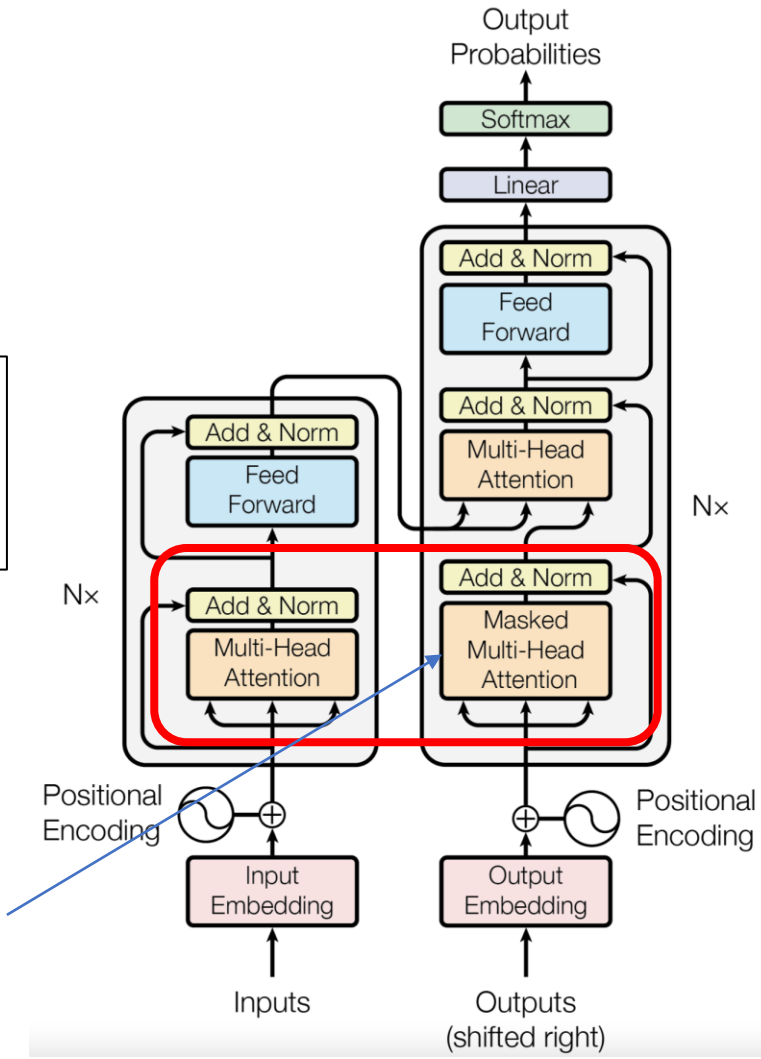
evaluating other input words in terms of relevance for encoding of given word




source

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

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

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

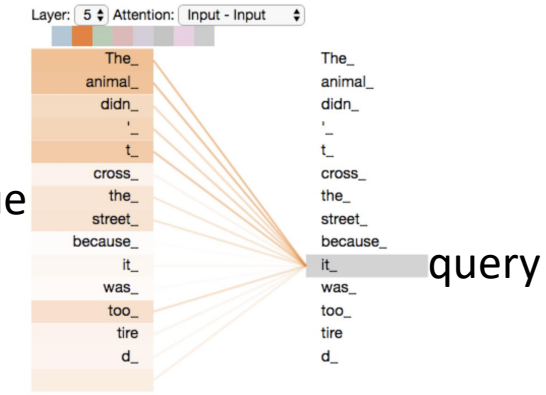
Scaled Dot-Product Attention

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

- query Q
- key K
- value V

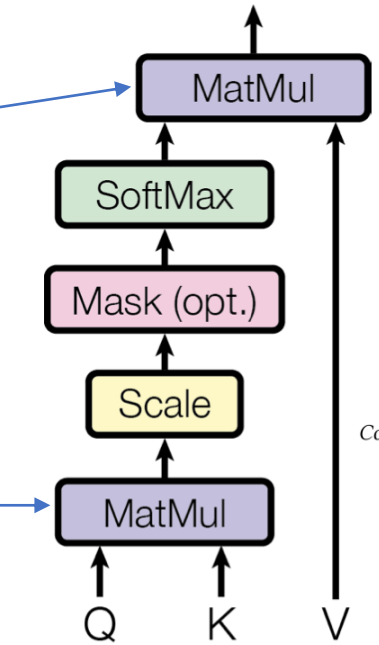
filtering: multiplication of attention probabilities with corresponding key word values

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)

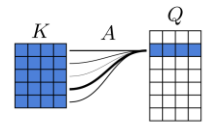


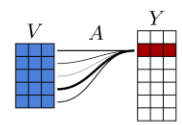

key-value pairs

query




source

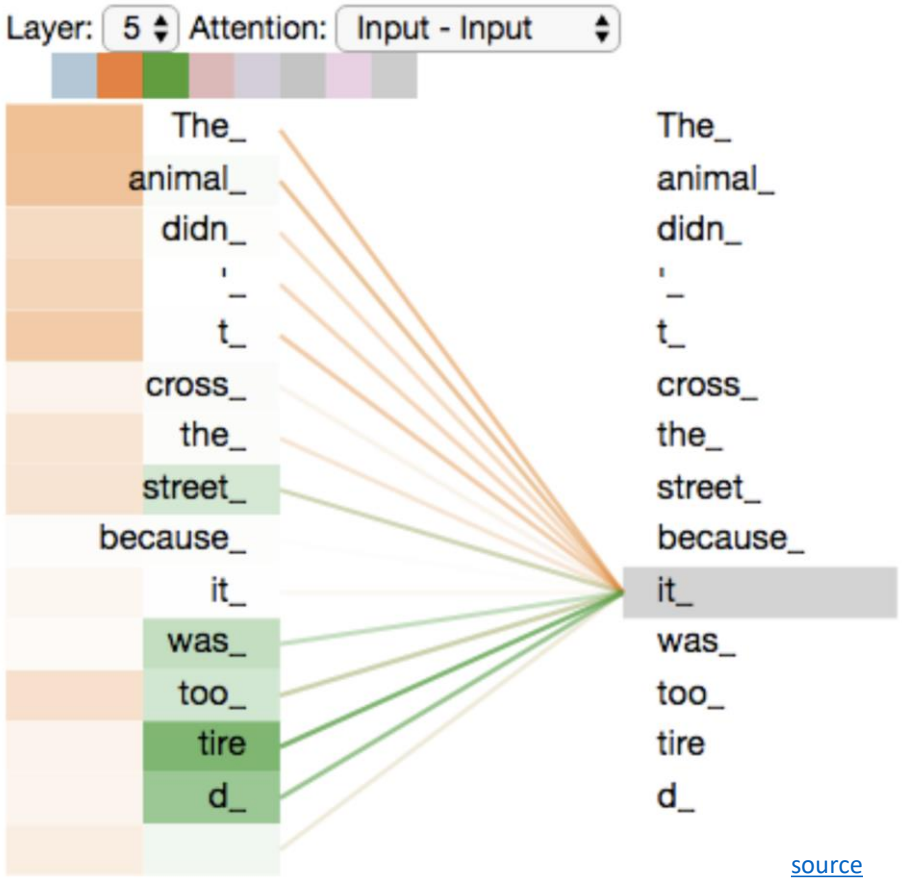



source

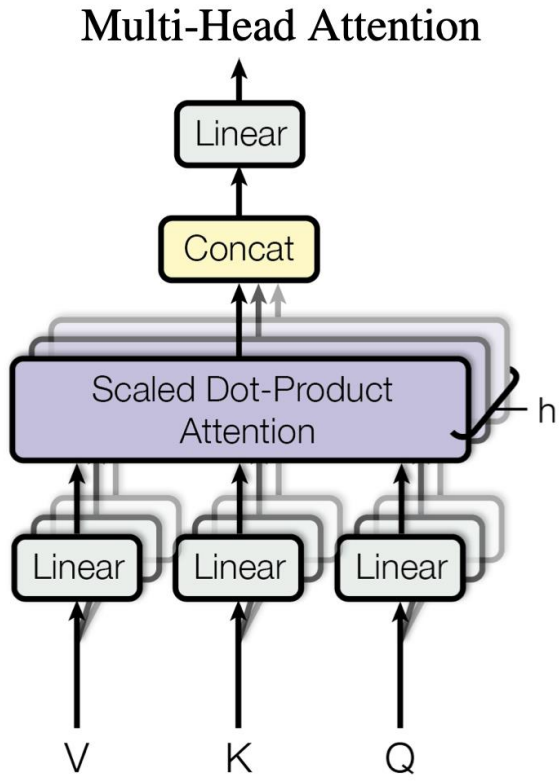
# Multi-Head Attention

multiple heads: several attention layers running in parallel




source

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*

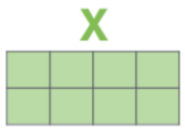
2) We embed each word*

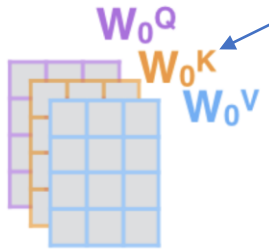
3) Split into 8 heads. We multiply X or R with weight matrices

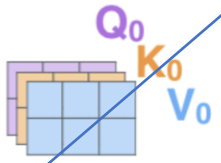
4) Calculate attention using the resulting Q/K/V matrices

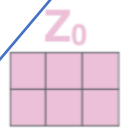
5) Concatenate the resulting Z matrices, then multiply with weight matrix $W^O$ to produce the output of the layer

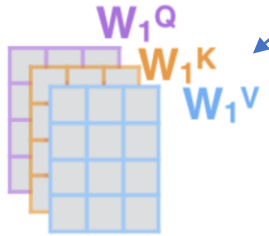
Thinking Machines

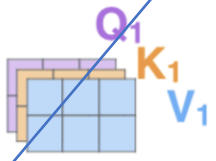


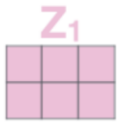



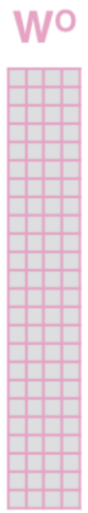



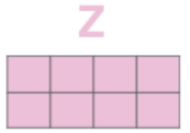



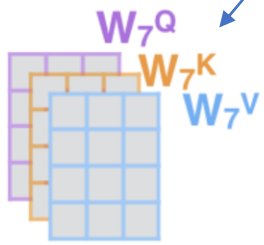



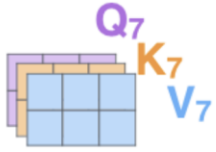



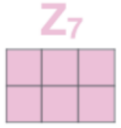




* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



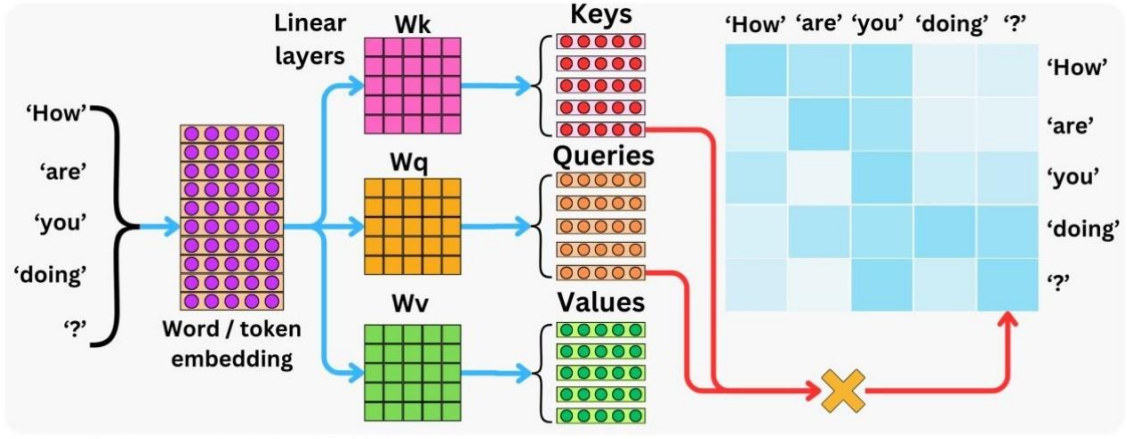




source

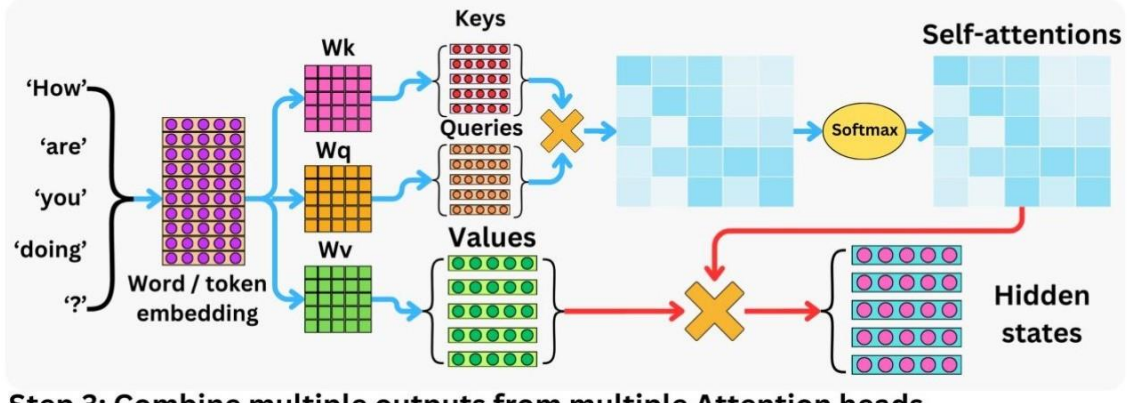
# Attention is all you need!

TheAiEdge.io

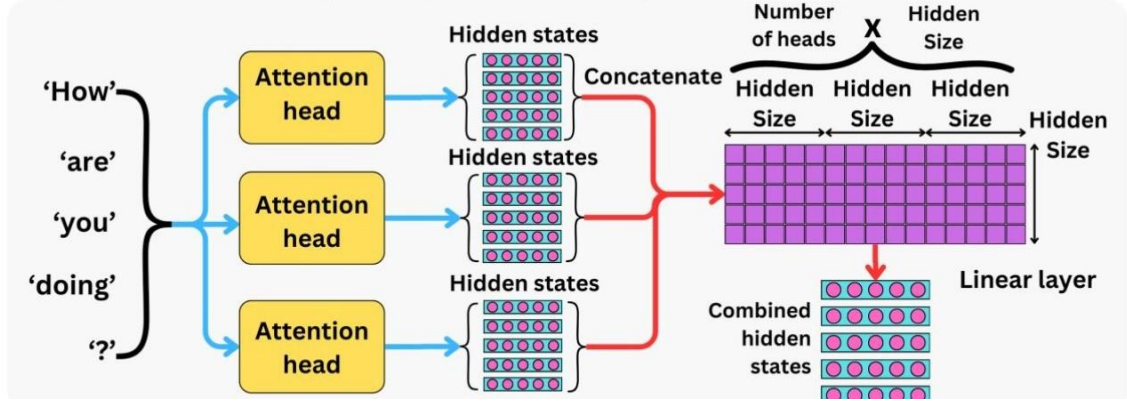
**Step 1: Compute the scores that captures the token-token interaction**




**Step 2: Compute the hidden states**

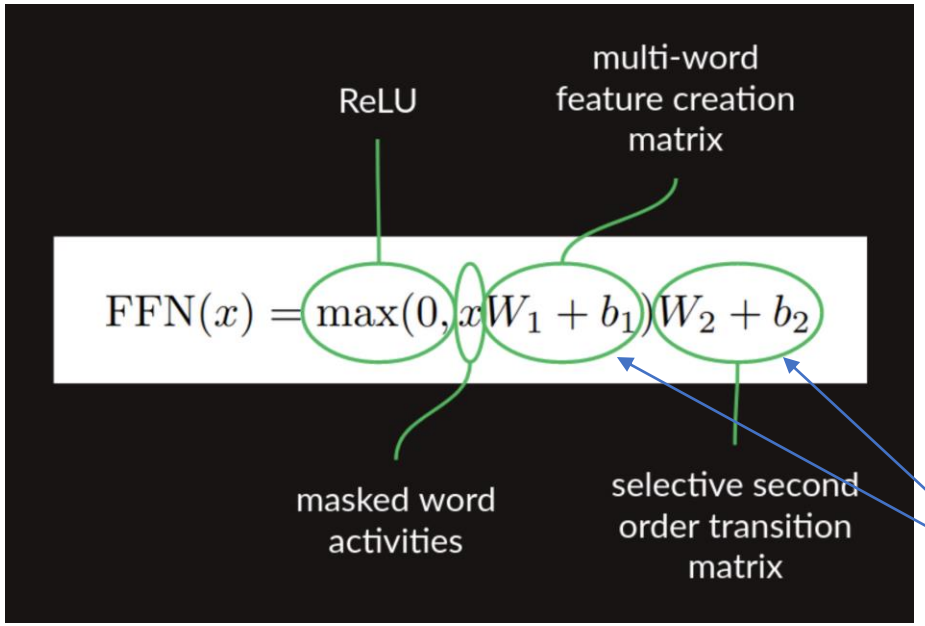



**Step 3: Combine multiple outputs from multiple Attention heads**



# Position-Wise Feed-Forward Networks

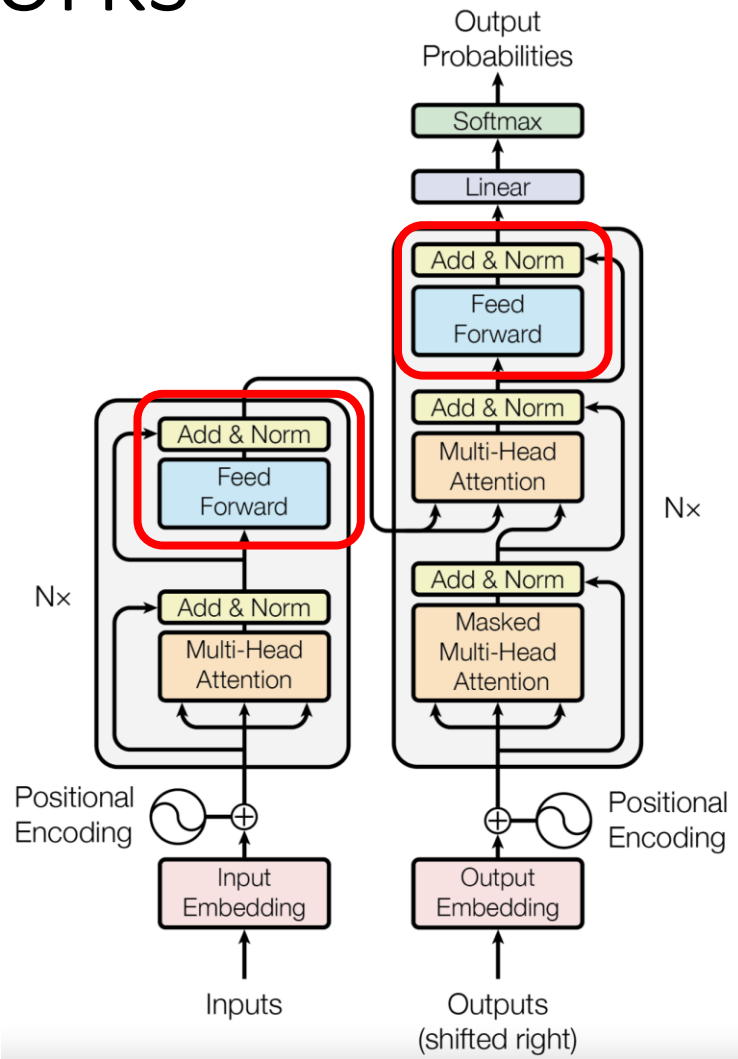
for each encoder or decoder layer: identical feed-forward network independently applied to each position




source

attention is just weighted averaging → need for non-linearities (neural networks): creating multi-word features from (self-)attention outputs (selectively masked words)

two network layers




source

# Encoder-Decoder Attention

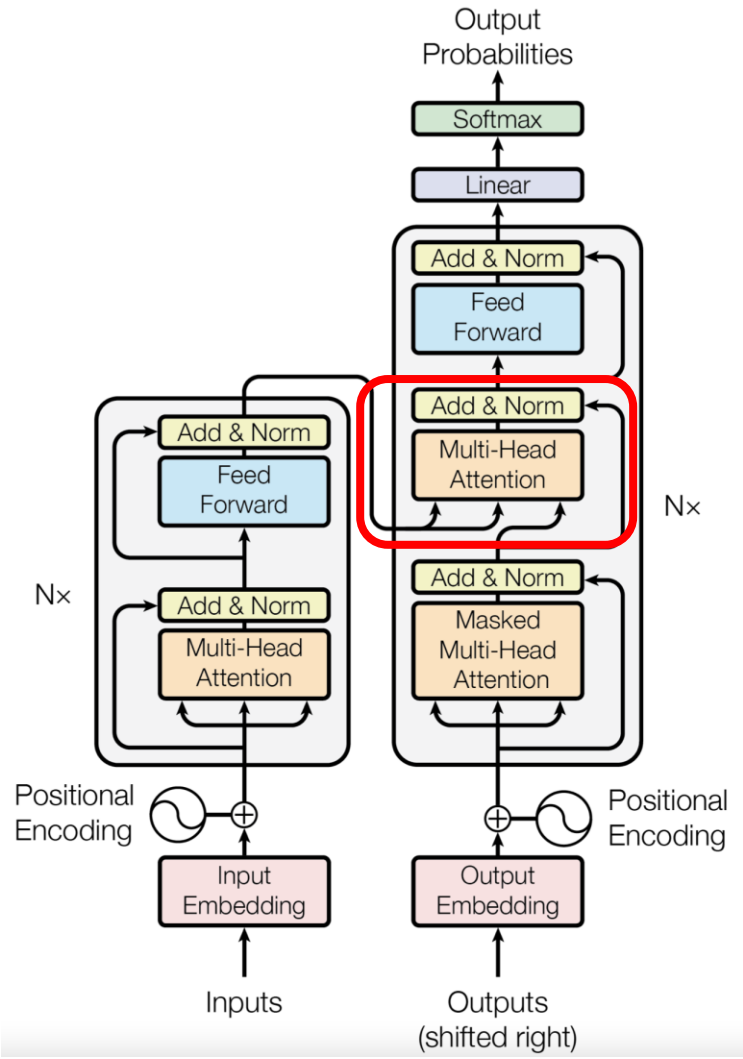
aka cross-attention

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 $\mathrm{K}$ and $\mathrm{V}$ → fed to each decoder's cross-attention layer (redundancy)

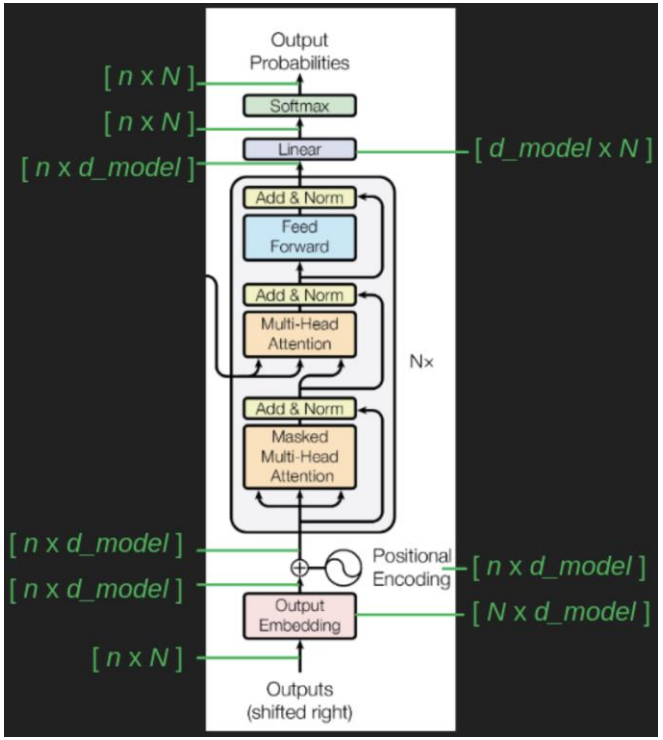
multiheaded self-attention with $\mathrm{Q}$ from decoder layer below and $\mathrm{K}$, $\mathrm{V}$ from output of encoder stack




source

# De-Embedding and Softmax

*n*: maximum sequence length
*N*: vocabulary size
*d_model*: embedding dimensions

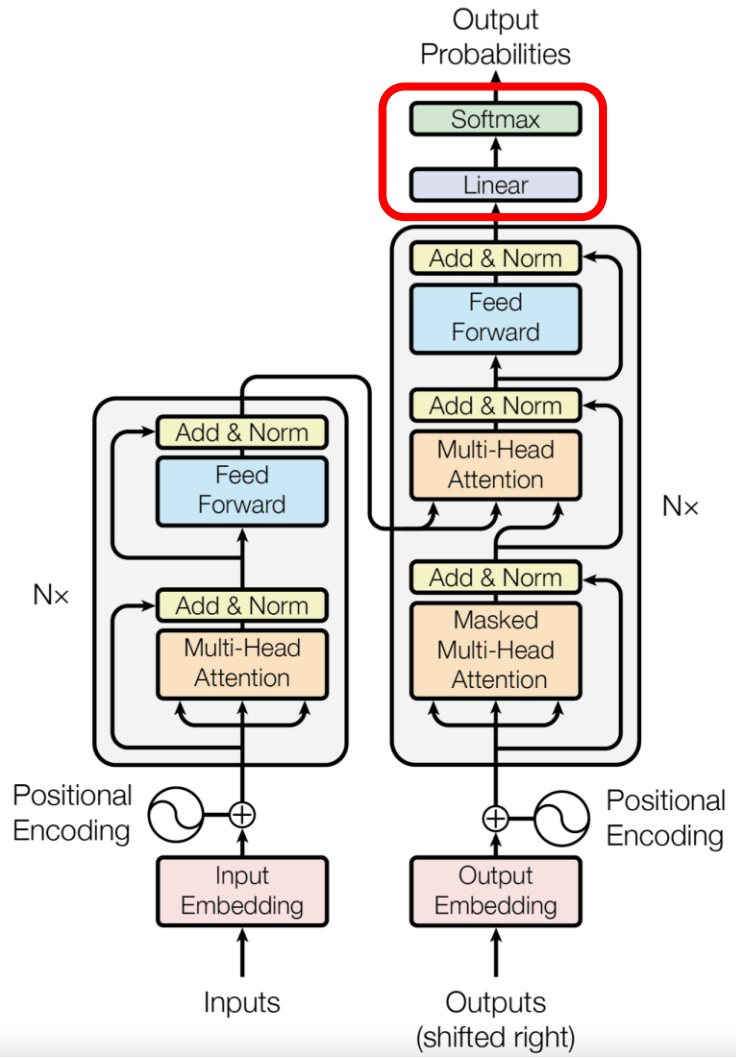



source

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)

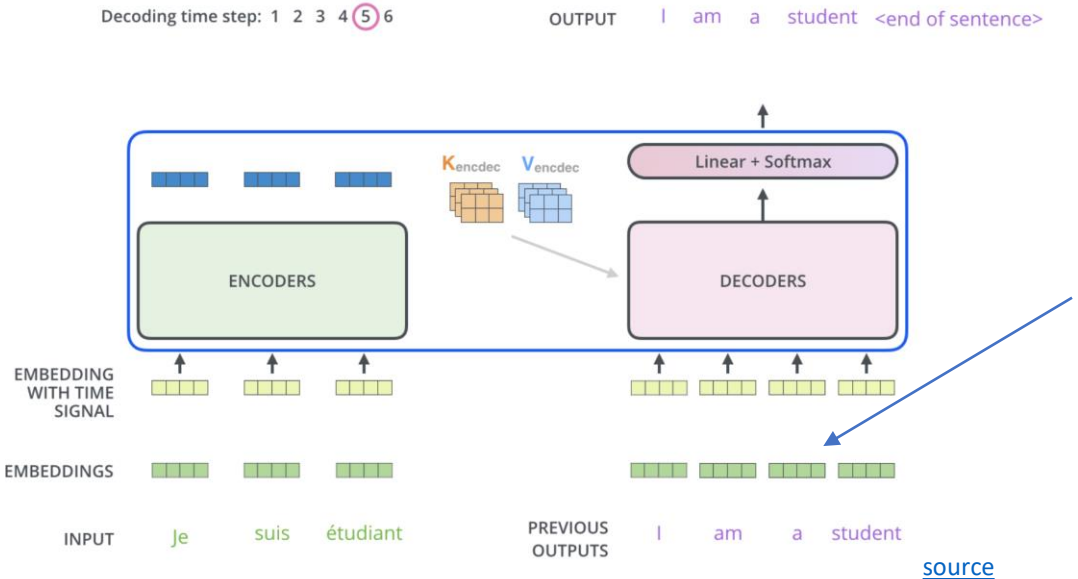



source

# Sequence Completion

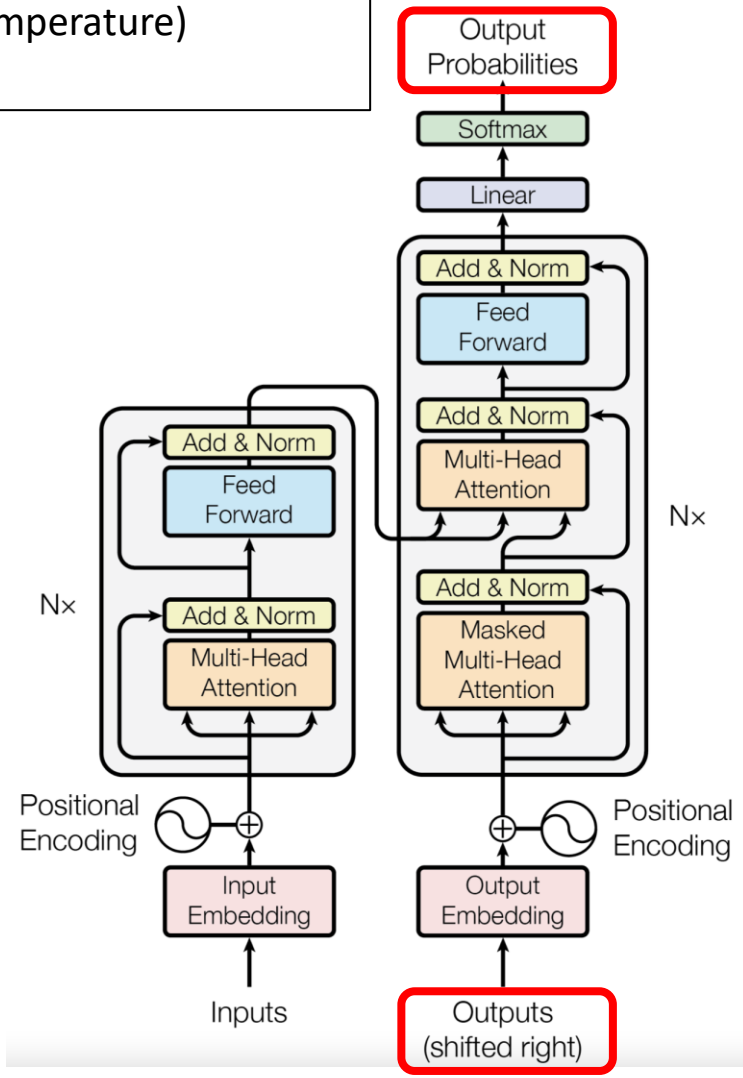
- greedily picking the one with highest probability
- pick according to probabilities (degree of randomness controlled by softmax temperature)
- beam search

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




source

auto-regressive




source

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

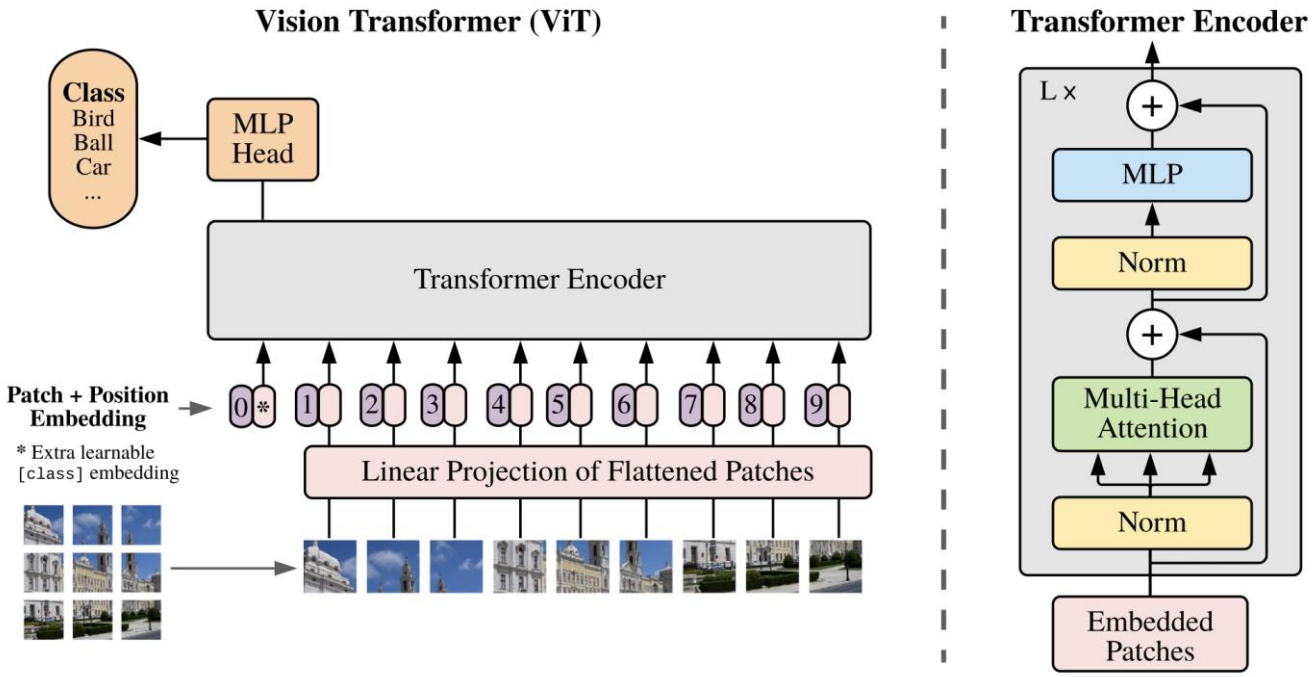
# Image Classification with Vision Transformer (ViT)

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, Lit-GPT