Introduction to NLP

CSE5321/CSEG321

Lecture 11. Transformers (2)

Hwaran Lee (hwaranlee@sogang.ac.kr)

Materials are referenced from Stanford CS224n

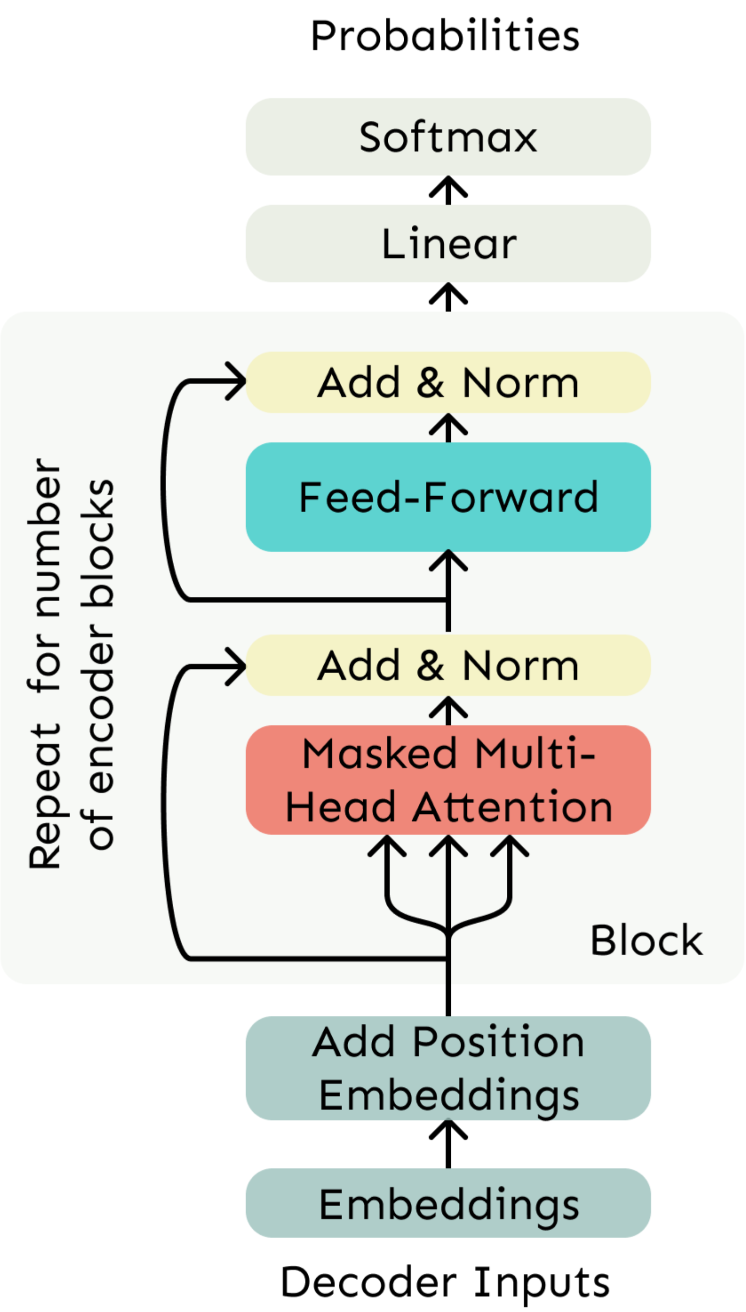
Lecture Plan

Lecture 11: Transformers

1. Notification on "Discussion in Class"
2. Recap Transformer Architectures
3. Positional Embeddings
4. Great results with Transformers
5. Drawbacks and variants of Transformers

The Transformer Decoder

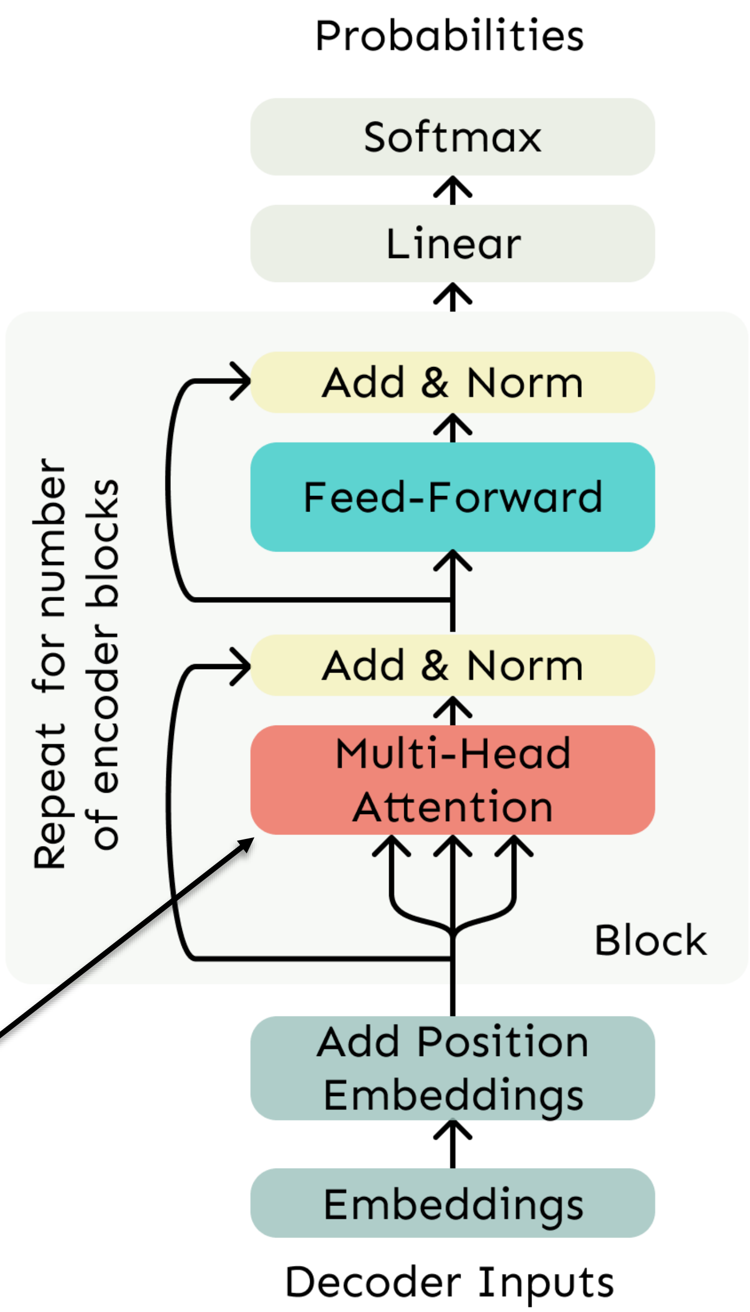
- The Transformer Decoder is a stack of Transformer Decoder **Blocks**.
- Each Block consists of:
  - Self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm
- That's it! We've gone through the Transformer Decoder.



The Transformer Encoder

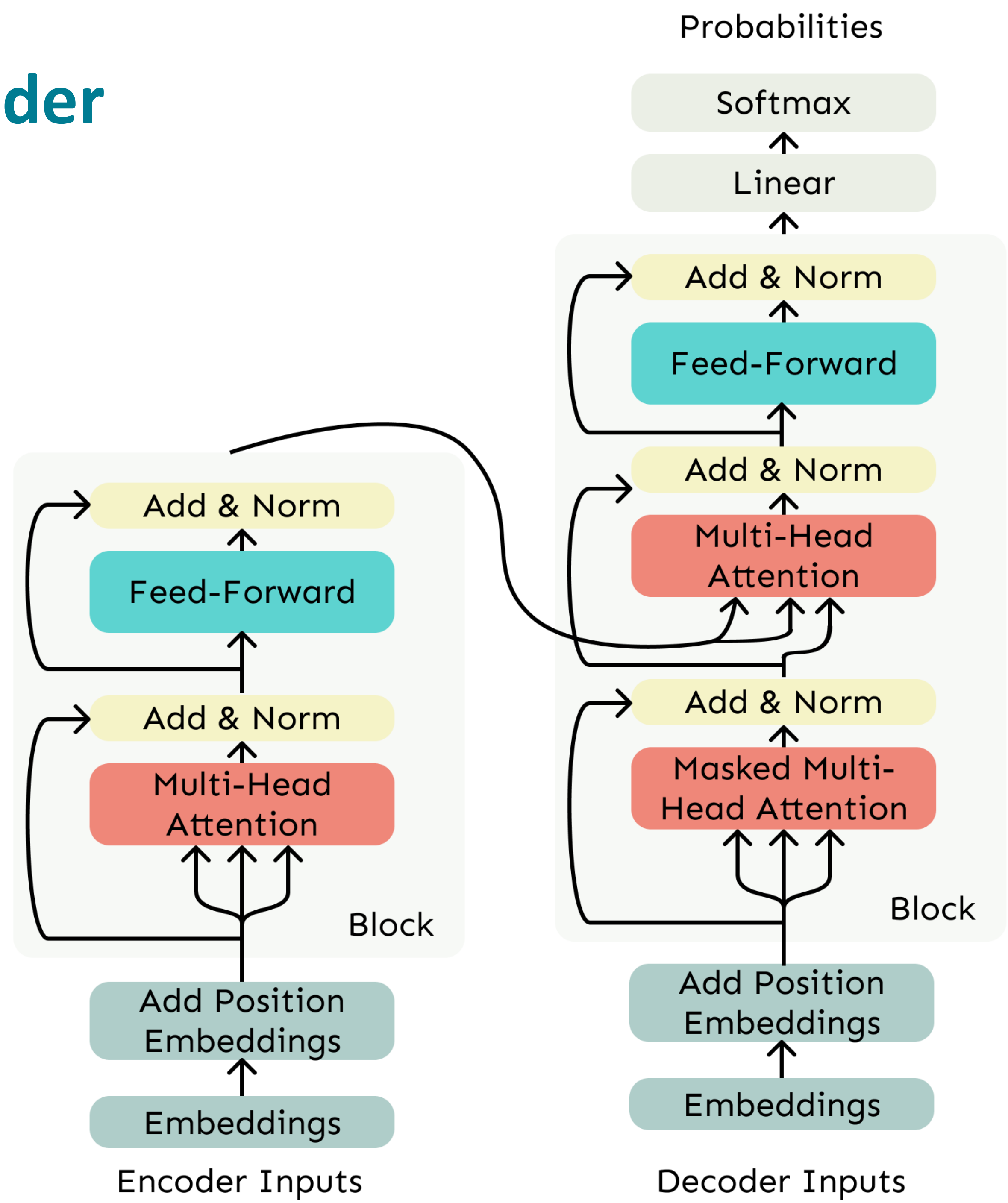
- The Transformer Decoder constrains to **unidirectional context**, as for **language models.**
- What if we want **bidirectional context**, like in a bidirectional RNN?
- This is the Transformer Encoder. The only difference is that we **remove the masking** in the self-attention.

No Masking!



The Transformer Encoder-Decoder

- Recall that in machine translation, we processed the source sentence with a **bidirectional** model and generated the target with a **unidirectional model**.
- For this kind of seq2seq format, we often use a Transformer Encoder-Decoder.
- We use a normal Transformer Encoder.
- Our Transformer Decoder is modified to perform **cross-attention** to the output of the Encoder.



Fixing the first self-attention problem: **sequence order**

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$$\boldsymbol{p}_i \in \mathbb{R}^d, \text{ for } i \in \{1,2,\dots,n\} \text{ are position vectors}$$

- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the $\boldsymbol{p}_i$ to our inputs!
- Recall that $\boldsymbol{x}_i$ is the embedding of the word at index i. The positioned embedding is:

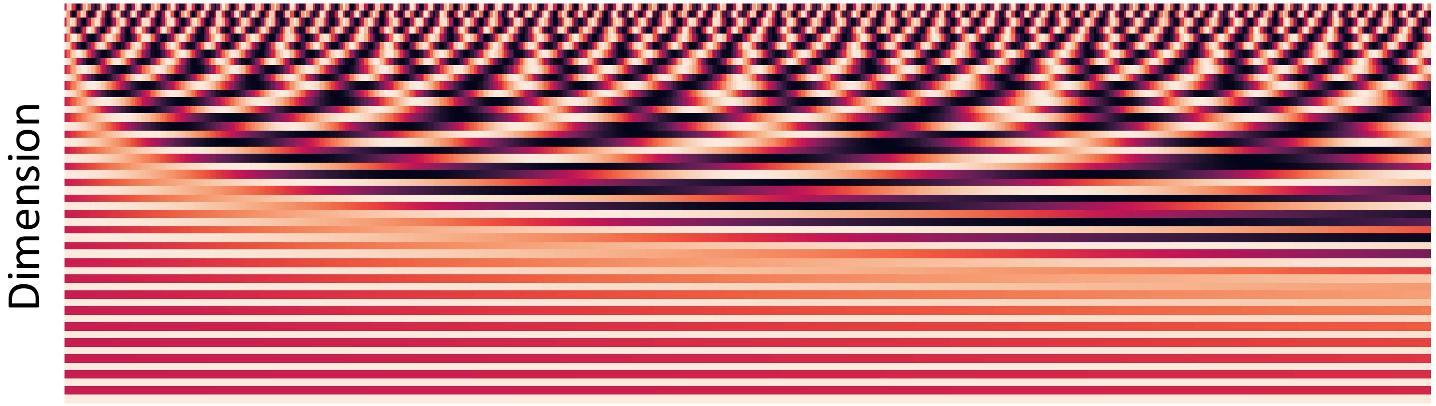
$$\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add…

Position representation vectors through sinusoids

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

$$
\boldsymbol{p}_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}
$$



- Pros:
  - Periodicity indicates that maybe “absolute position” isn’t as important
  - Maybe can extrapolate to longer sequences as periods restart!
- Cons:
  - Not learnable; also the extrapolation doesn’t really work!

Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

Position representation vectors learned from scratch

- **Learned absolute position representations:** Let all p_i be learnable parameters!
  Learn a matrix $\boldsymbol{p} \in \mathbb{R}^{d \times n}$, *and let each* $\boldsymbol{p}_i$ *be a column of that matrix!*

- Pros:
  - Flexibility: each position gets to be learned to fit the data
- Cons:
  - Definitely can't extrapolate to indices outside $1, \dots, n$.
- Most systems use this!

- Sometimes people try more flexible representations of position:
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]

Positional Embeddings

- Motivating Example:

  - The **dog** chased another **dog**

  - -> Without any positional information, the output is *identical* for the same token in different positions.

https://huggingface.co/blog/designing-positional-encoding

Positional Embeddings

- Motivating Example:

  - The **dog** chased another **dog**

  - -> Without any positional information, the output is *identical* for the same token in different positions.

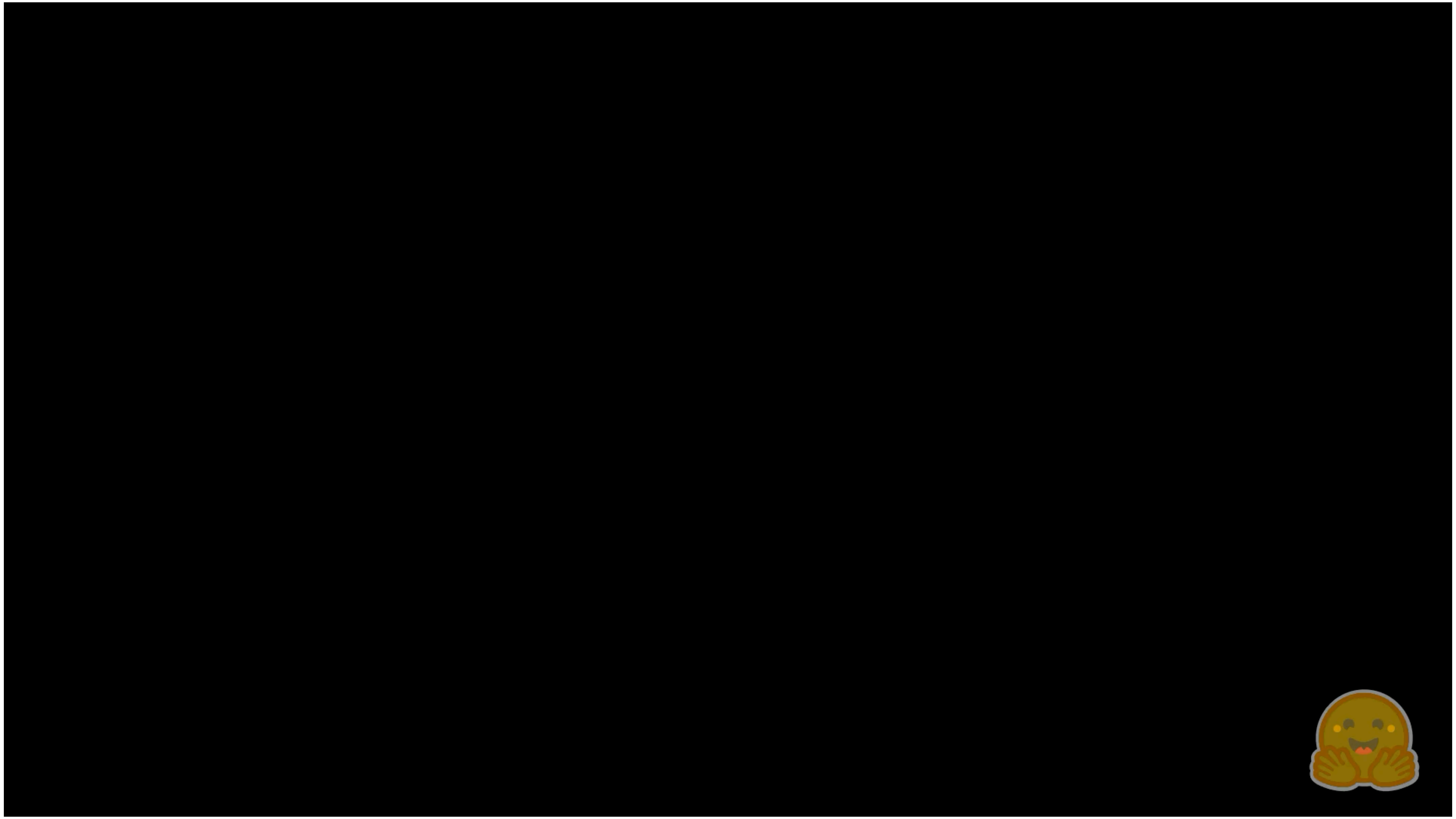
```python
import torch
import torch.nn as nn
from transformers import AutoTokenizer, AutoModel

model_id = "meta-llama/Llama-3.2-1B"
tok = AutoTokenizer.from_pretrained(model_id)
model = AutoModel.from_pretrained(model_id)

text = "The dog chased another dog"
tokens = tok(text, return_tensors="pt")["input_ids"]
embeddings = model.embed_tokens(tokens)
hdim = embeddings.shape[-1]

W_q = nn.Linear(hdim, hdim, bias=False)
W_k = nn.Linear(hdim, hdim, bias=False)
W_v = nn.Linear(hdim, hdim, bias=False)
mha = nn.MultiheadAttention(embed_dim=hdim, num_heads=4, batch_first=True)

with torch.no_grad():
    for param in mha.parameters():
        nn.init.normal_(param, std=0.1) # Initialize weights to be non-negligible

output, _ = mha(W_q(embeddings), W_k(embeddings), W_v(embeddings))

dog1_out = output[0, 2]
dog2_out = output[0, 5]
print(f"Dog output identical?: {torch.allclose(dog1_out, dog2_out, atol=1e-6)}")
```

https://huggingface.co/blog/designing-positional-encoding

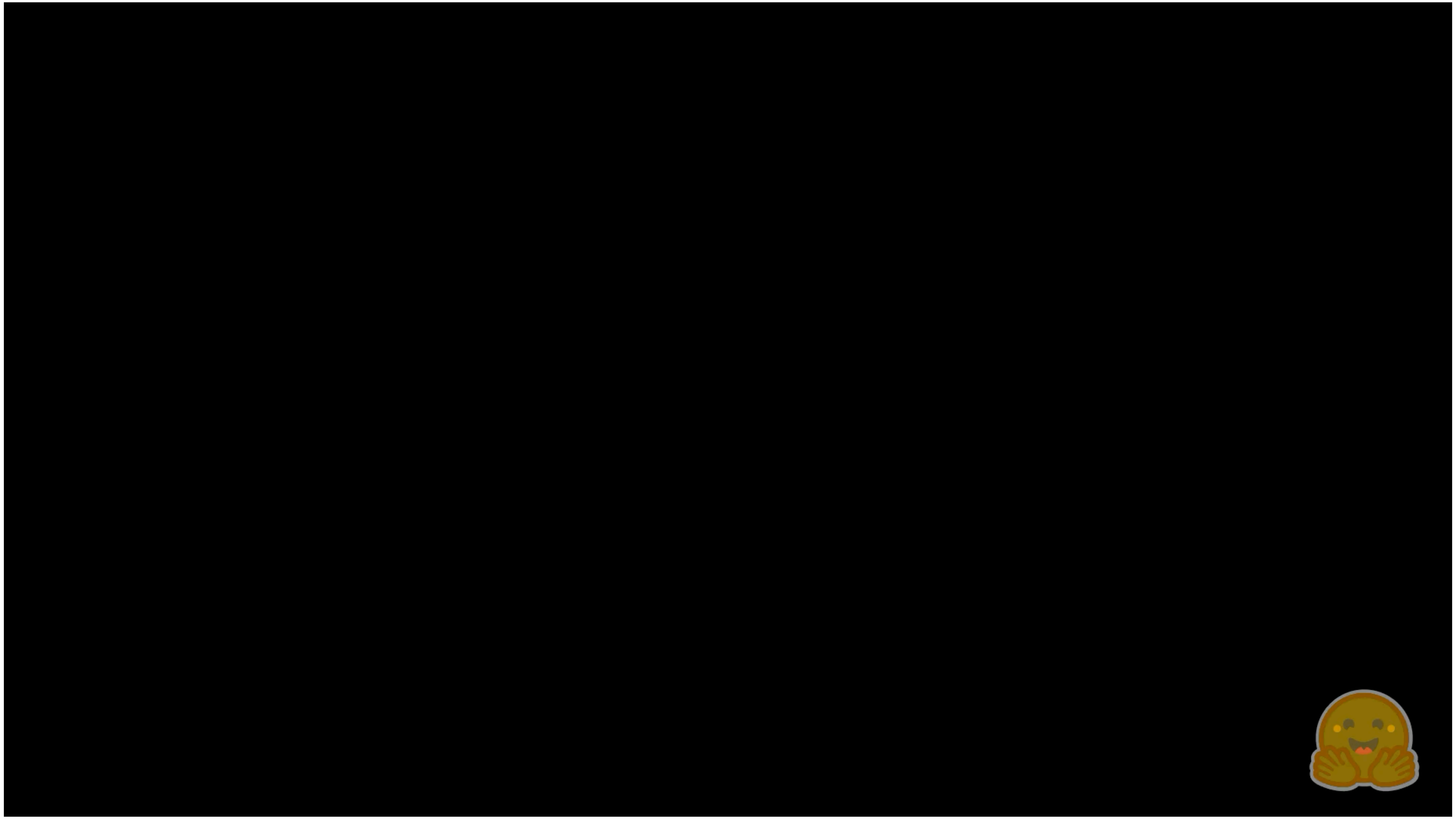
Positional Embeddings

- Desirable Properties
  - Property 1 - Unique encoding for each position (across sequences)
  - Property 2 - Linear relation between two encoded positions
  - Property 3 - Generalizes to longer sequences than those encountered in training
  - Property 4 - Generated by a deterministic process the model can learn
  - Property 5 - Extensible to multiple dimensions

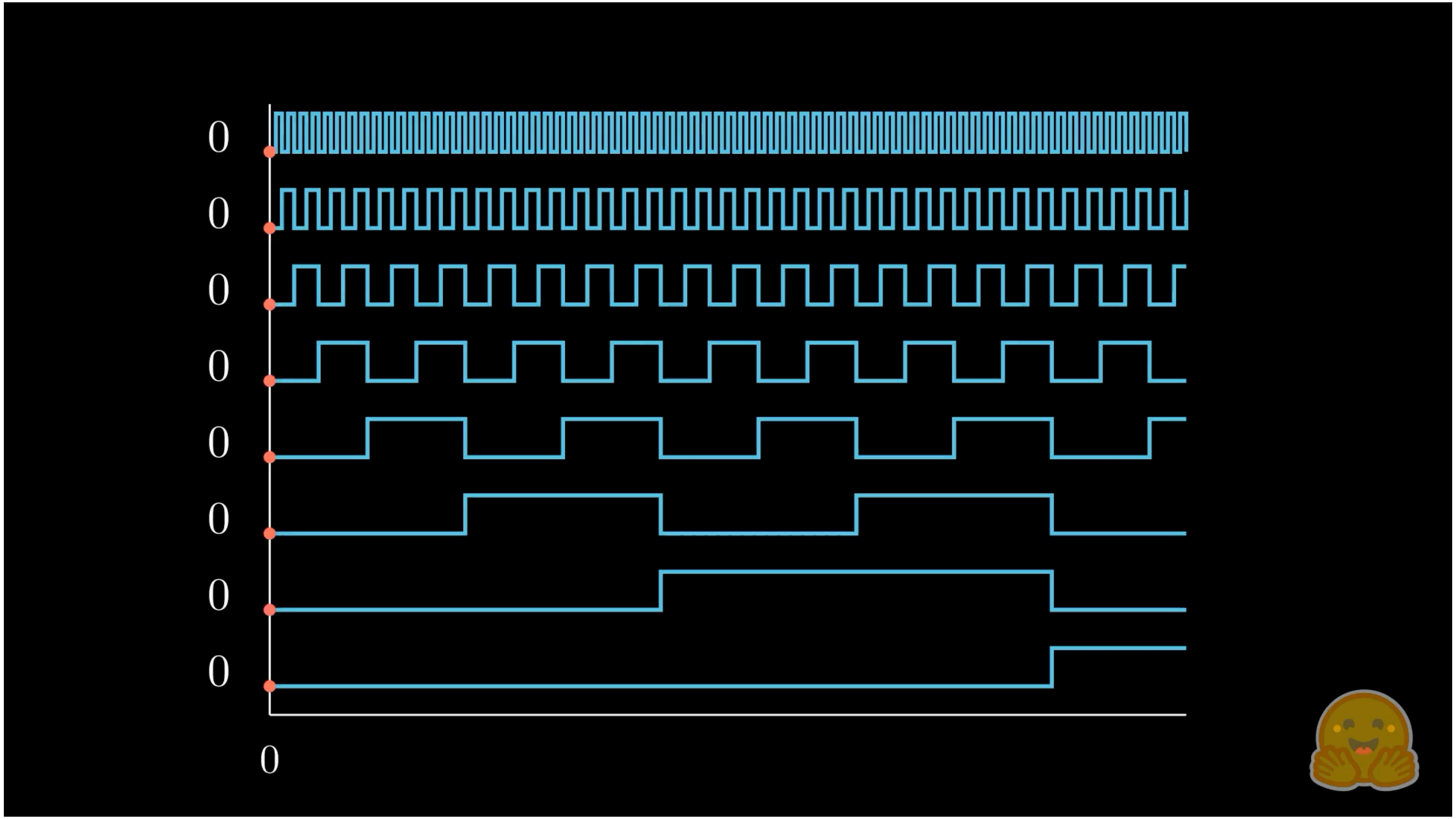
Positional Embeddings: Integer Position Encoding



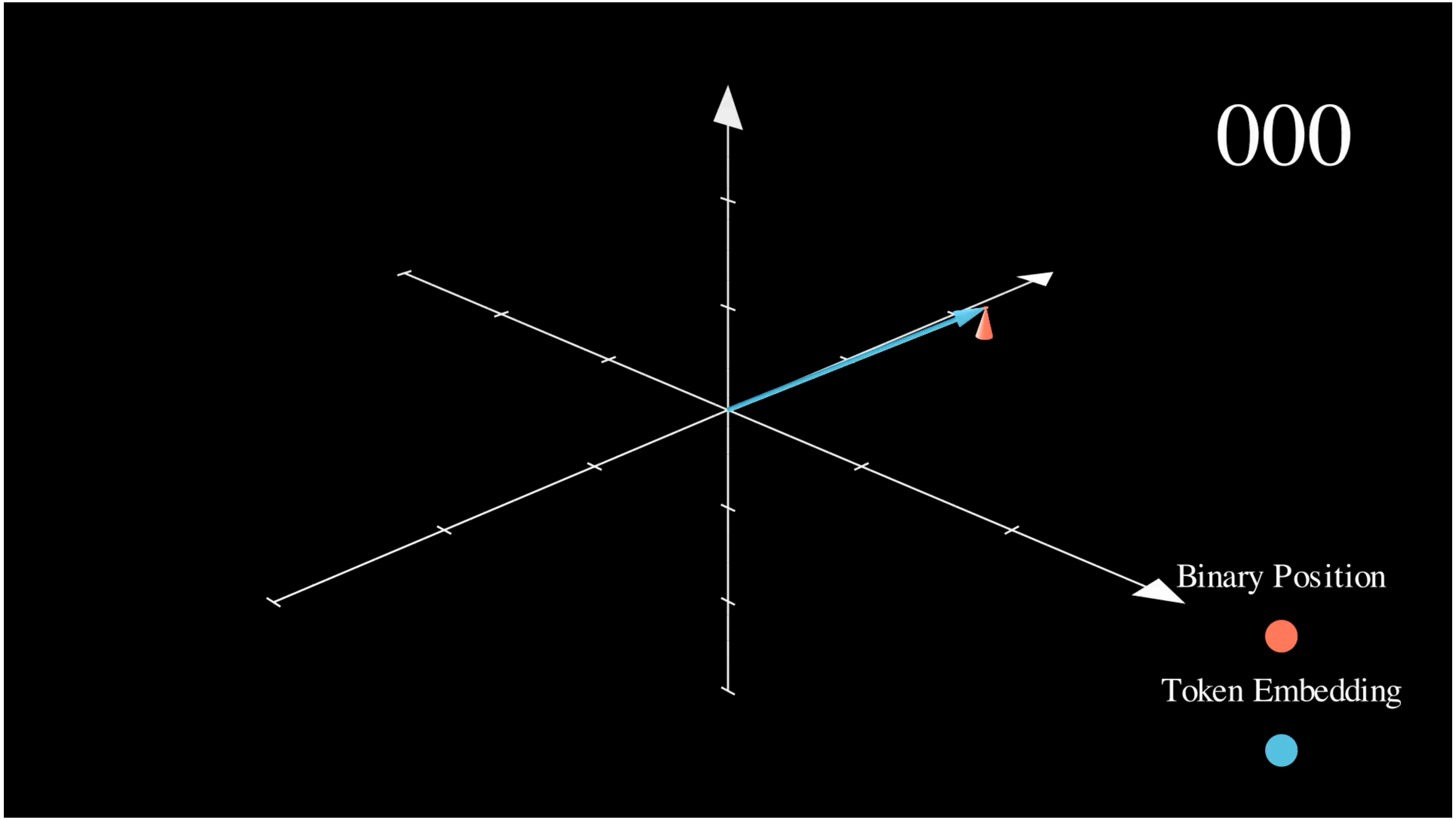
Positional Embeddings: Binary Position Encoding



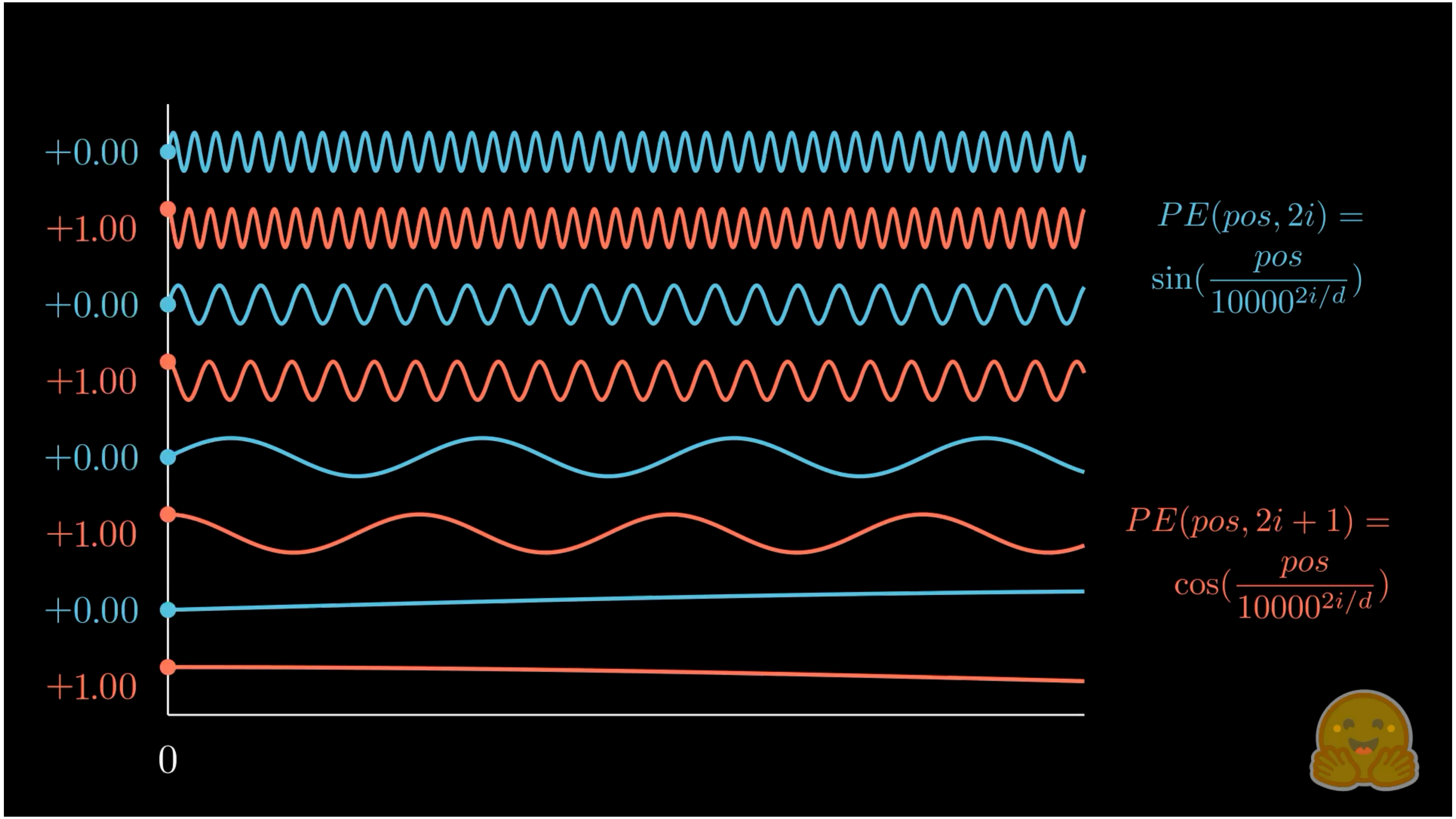
Positional Embeddings: Binary Position Encoding



Positional Embeddings: Binary Position Encoding



Positional Embeddings: Sinusoidal Positional Encoding



Common, modern position embeddings - RoPE

High level thought process: a *relative* position embedding should be some $f(x, i)$ s.t.

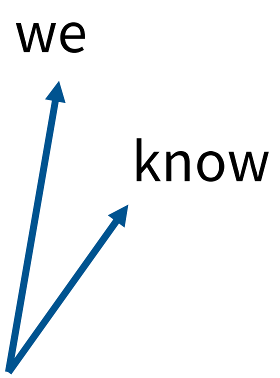
$$\langle f(x, i), f(y, j) \rangle = g(x, y, i - j)$$

That is, the attention function *only* gets to depend on the relative position (i-j). How do existing embeddings not fulfill this goal?

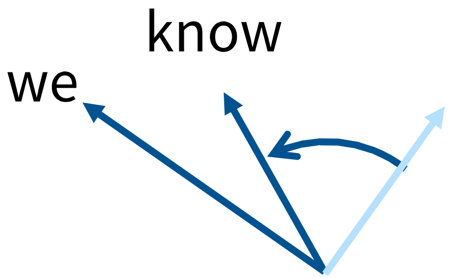
RoPE – Embedding via rotation

How can we solve this problem?

- We want our embeddings to be invariant to absolute position
- We know that inner products are invariant to arbitrary rotation.

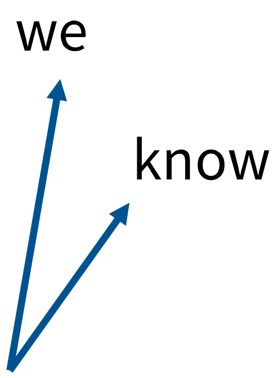


Position independent embedding



Embedding
“of course we know”

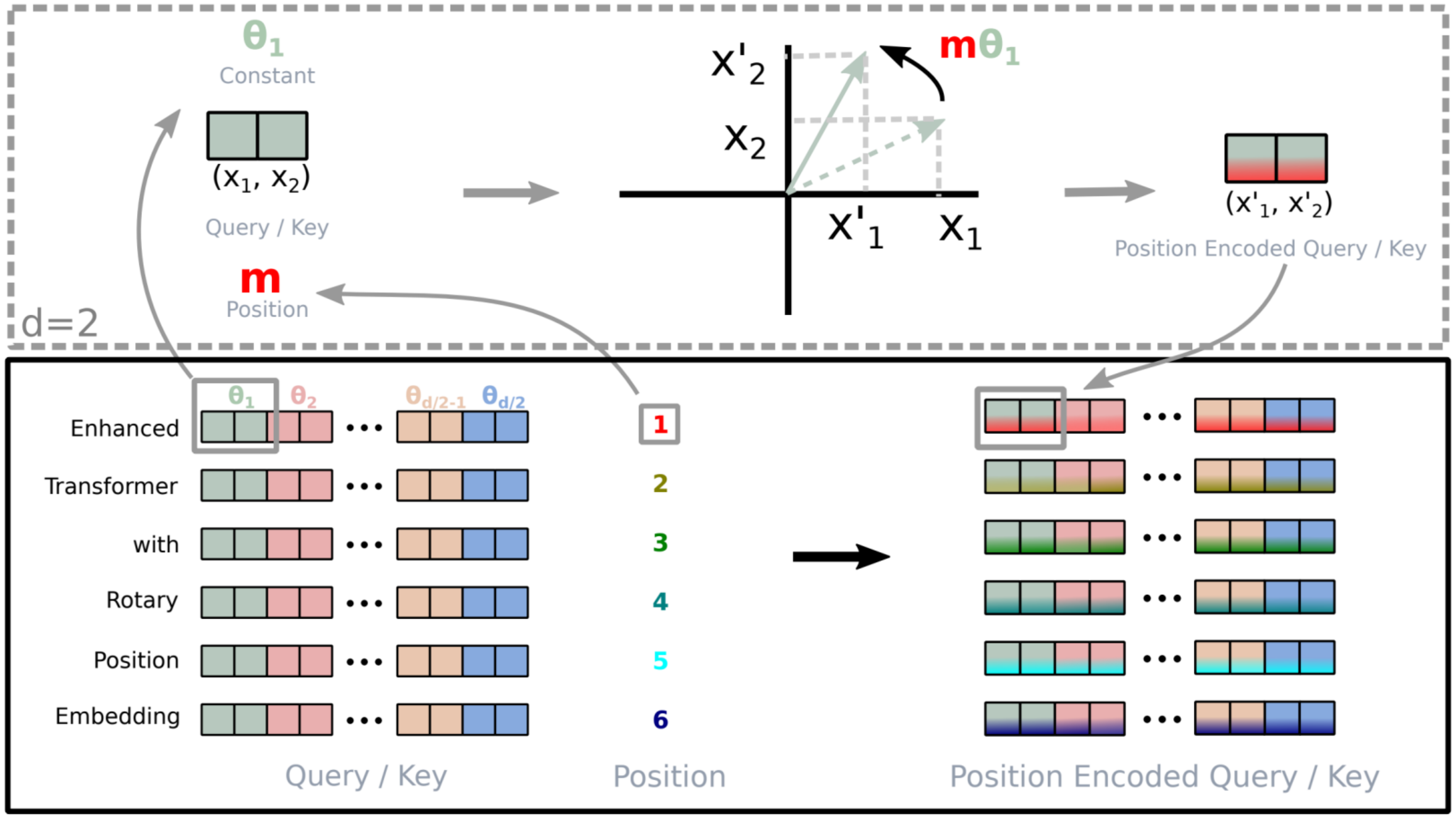
Rotate by ‘2 positions’



Embedding
“we know that”

Rotate by ‘0 positions’

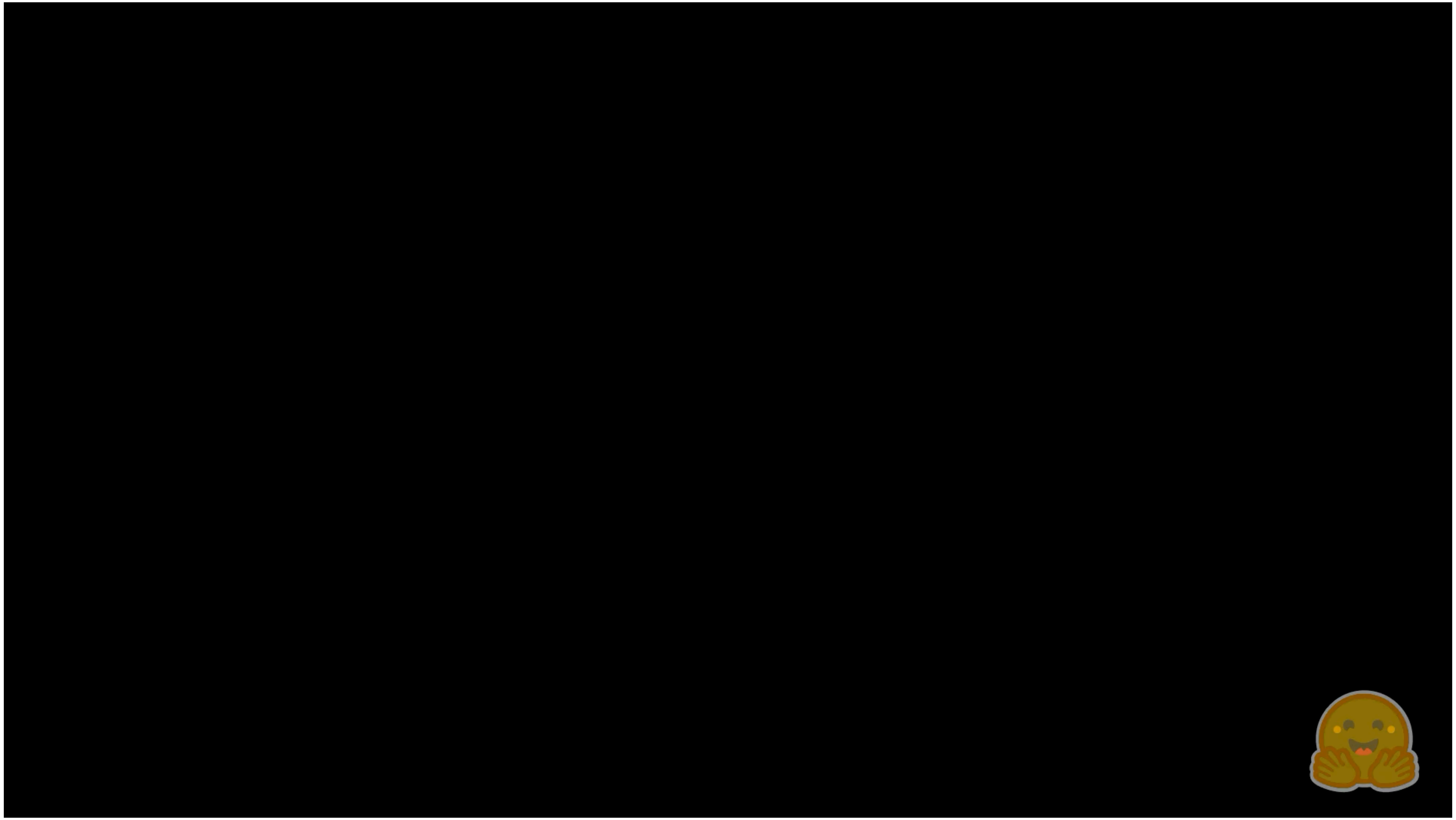
RoPE – From 2 to many dimensions



[Su et al 2021]

Just pair up the coordinates and rotate them in 2d (motivation: complex numbers)

Positional Embeddings: RoPE



Great Results with Transformers

First, Machine Translation from the original Transformers paper!

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

 [Test sets: WMT 2014 English-German and English-French]

[Vaswani et al., 2017]

Great Results with Transformers

Next, document generation!

Model	Test perplexity	ROUGE-L
seq2seq-attention, $L = 500$	5.04952	12.7
Transformer-ED, $L = 500$	2.46645	34.2
Transformer-D, $L = 4000$	2.22216	33.6
Transformer-DMCA, no MoE-layer, $L = 11000$	2.05159	36.2
Transformer-DMCA, MoE-128, $L = 11000$	1.92871	37.9
Transformer-DMCA, MoE-256, $L = 7500$	1.90325	38.8

The old standard

Transformers all the way down.

[Liu et al., 2018]; WikiSum dataset

Great Results with Transformers

Before too long, most Transformers results also included **pretraining**, a method we'll go over next.

Transformers' parallelizability allows for efficient pretraining, and have made them the de-facto standard.

On this popular aggregate benchmark, for example:

GLUE

All top models are Transformer (and pretraining)-based.

	Rank	Name	Model	URL	Score
	1	DeBERTa Team - Microsoft	DeBERTa / TuringNLRv4	↗	90.8
	2	HFL iFLYTEK	MacALBERT + DKM		90.7
+	3	Alibaba DAMO NLP	StructBERT + TAPT	↗	90.6
+	4	PING-AN Omni-Sinitic	ALBERT + DAAF + NAS		90.6
	5	ERNIE Team - Baidu	ERNIE	↗	90.4
	6	T5 Team - Google	T5	↗	90.3

[Liu et al., 2018]

Outline

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers

What would we like to fix about the Transformer?

- **Training instabilities (Pre vs Post norm)**

- **Quadratic compute in self-attention** :
  - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
  - For recurrent models, it only grew linearly!

Pre vs Post norm

The one thing *everyone* agrees on (in 2024)

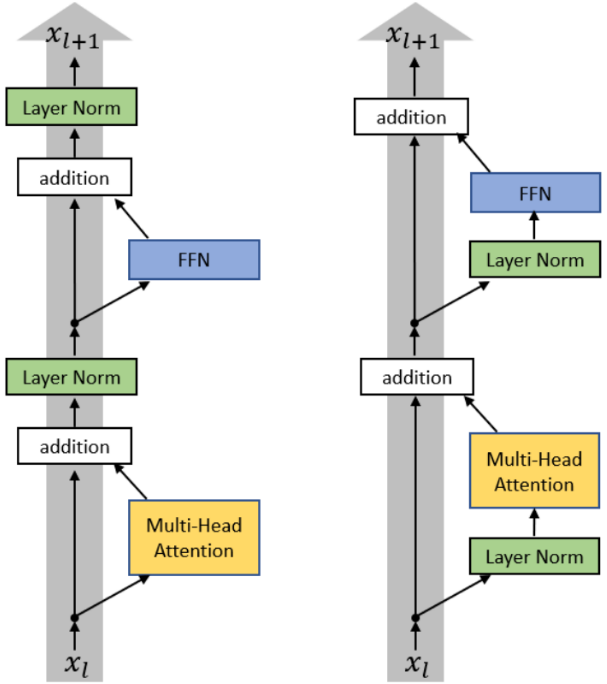


Figure from Xiong 2020

Post-LN Transformer	Pre-LN Transformer
$x^{post,1}_{l,i} = \text{MultiHeadAtt}(x^{post}_{l,i}, [x^{post}_{l,1}, \cdots, x^{post}_{l,n}])$	$x^{pre,1}_{l,i} = \text{LayerNorm}(x^{pre}_{l,i})$
$x^{post,2}_{l,i} = x^{post}_{l,i} + x^{post,1}_{l,i}$	$x^{pre,2}_{l,i} = \text{MultiHeadAtt}(x^{pre,1}_{l,i}, [x^{pre,1}_{l,1}, \cdots, x^{pre,1}_{l,n}])$
$x^{post,3}_{l,i} = \text{LayerNorm}(x^{post,2}_{l,i})$	$x^{pre,3}_{l,i} = x^{pre}_{l,i} + x^{pre,2}_{l,i}$
$x^{post,4}_{l,i} = \text{ReLU}(x^{post,3}_{l,i} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$	$x^{pre,4}_{l,i} = \text{LayerNorm}(x^{pre,3}_{l,i})$
$x^{post,5}_{l,i} = x^{post,3}_{l,i} + x^{post,4}_{l,i}$	$x^{pre,5}_{l,i} = \text{ReLU}(x^{pre,4}_{l,i} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$
$x^{post}_{l+1,i} = \text{LayerNorm}(x^{post,5}_{l,i})$	$x^{pre}_{l+1,i} = x^{pre,5}_{l,i} + x^{pre,3}_{l,i}$
	Final LayerNorm: $x^{pre}_{Final,i} \leftarrow \text{LayerNorm}(x^{pre}_{L+1,i})$

Set up LayerNorm so that it doesn't affect the main residual signal path (on the left)

Almost all modern LMs use pre-norm (but BERT was post-norm)

(One somewhat funny exception – OPT350M. I don't know why this is post-norm)

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(n^2 d)$, where n is the sequence length, and d is the dimensionality.

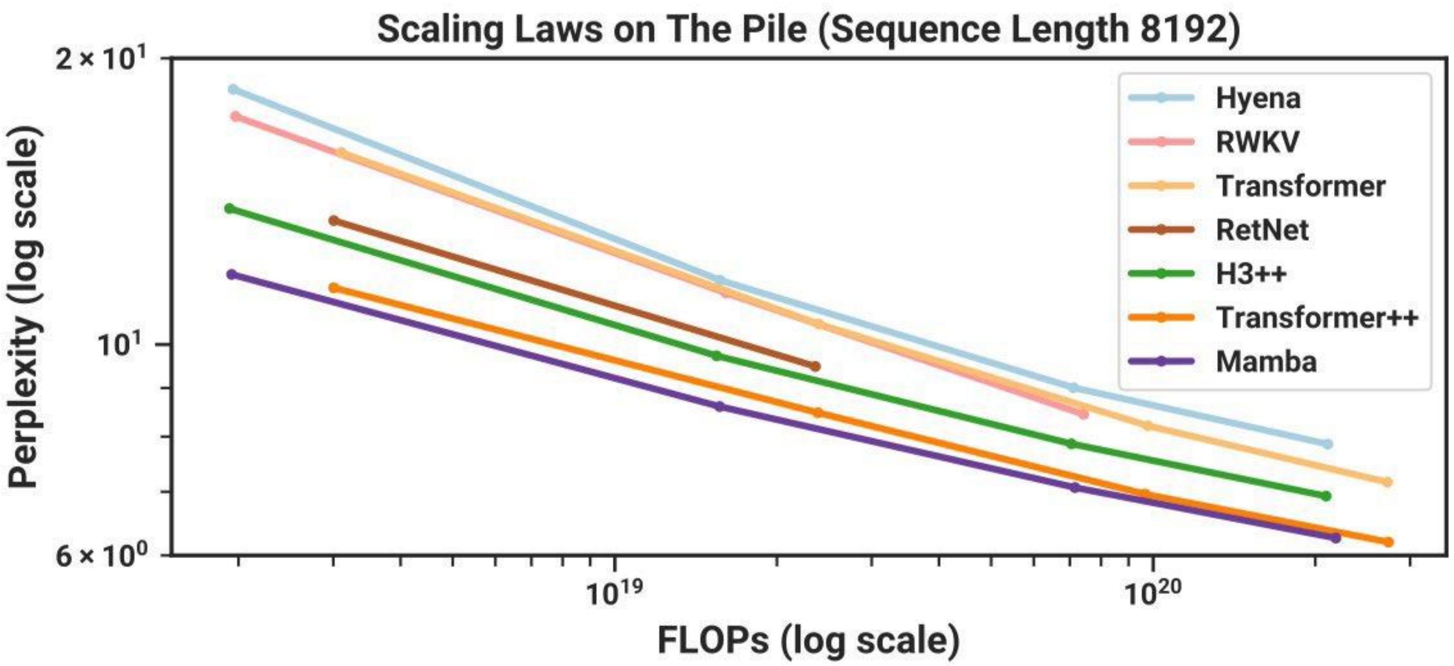
$$XQ \quad K^\top X^\top = XQK^\top X^\top \in \mathbb{R}^{n\times n}$$

Need to compute all pairs of interactions! $O(n^2 d)$

- Think of d as around $\mathbf{1,000}$ (though for large language models it's much larger!).
  - So, for a single (shortish) sentence, $n \le 30$; $n^2 \le \mathbf{900}$.
  - In practice, we set a bound like $n = 512$.
  - **But what if we'd like $\boldsymbol{n} \ge \mathbf{50,000}$?** For example, to work on long documents?

Back to the future – RNNs are back!

RNNs only require $O(nd^2)$ computations!



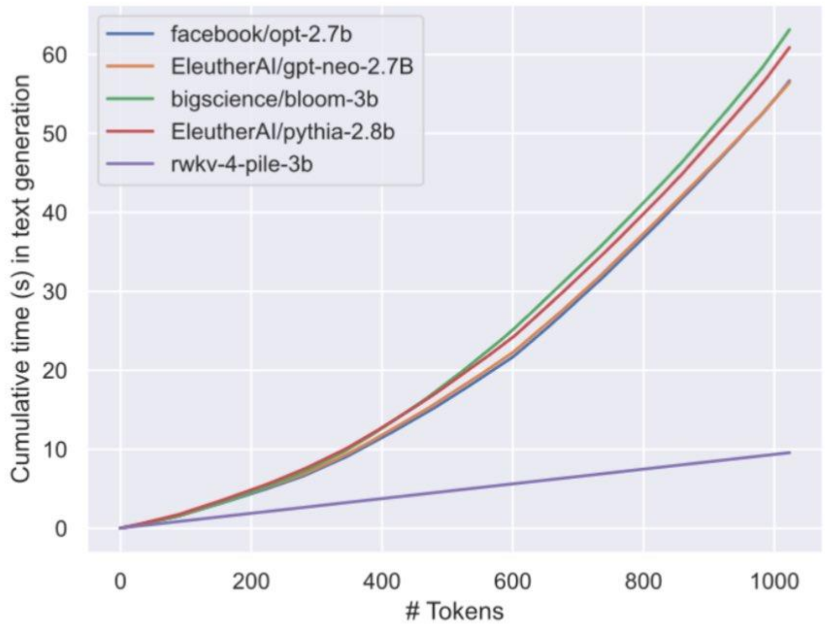
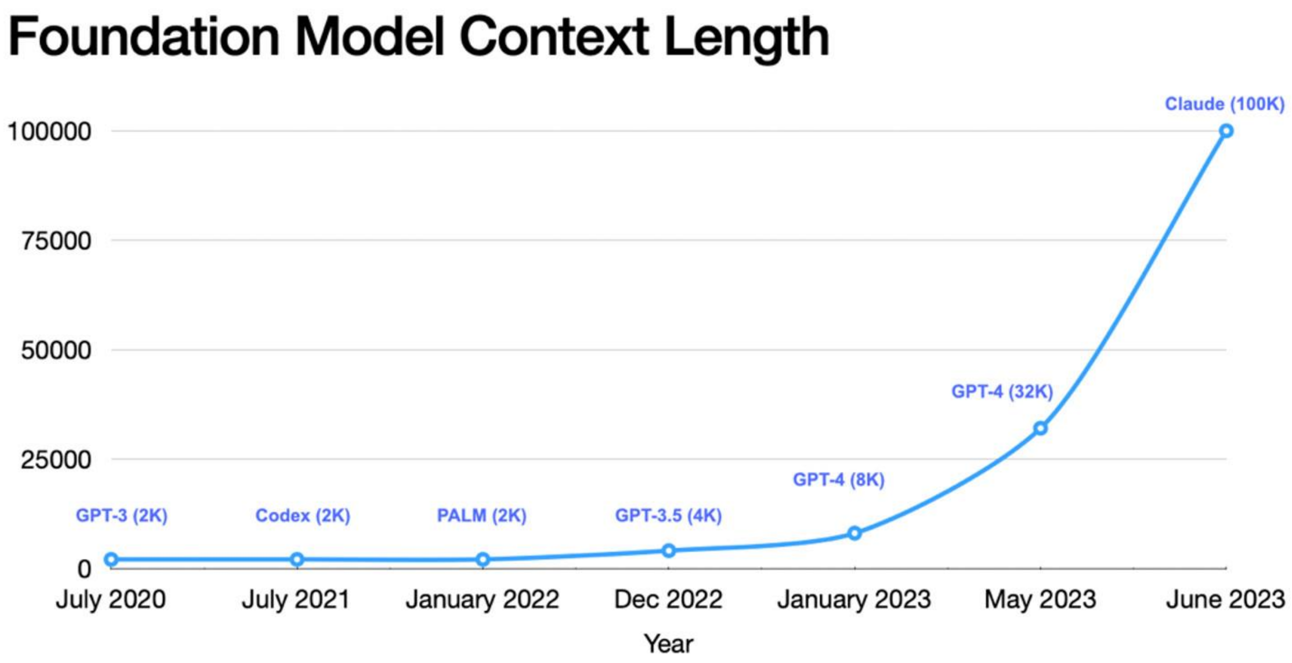


Figure 7: Cumulative time on text generation for LLM
Unlike transformers, RWKV exhibits linear scaling.

If you want *really* long context, RNNs provide this (linear complexity).
Modern RNNs (RWKV, Mamba, etc) are getting better!

Do we even need to remove the quadratic cost of attention?

- As Transformers grow larger, a larger and larger percent of compute is **outside** the self-attention portion, despite the quadratic cost.
- In practice, **production Transformer language models use quadratic cost attention**
  - The cheaper methods tend not to work as well at scale.
  - Systems optimizations work well (Flash attention – Jun 2022)



Do Transformer Modifications Transfer?

- "Surprisingly, we find that most modifications do not meaningfully improve performance."

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02	26.62
GeLU	223M	11.1T	3.58	2.179 ± 0.003	1.838	**75.79**	**17.86**	**25.13**	26.47
Swish	223M	11.1T	3.62	2.186 ± 0.003	1.847	**73.77**	17.74	**24.34**	**26.75**
ELU	223M	11.1T	3.56	2.270 ± 0.007	1.932	67.83	16.73	23.02	26.08
GLU	223M	11.1T	3.59	2.174 ± 0.003	**1.814**	**74.20**	**17.42**	24.34	**27.12**
GeGLU	223M	11.1T	3.55	2.130 ± 0.006	**1.792**	**75.96**	**18.27**	**24.87**	**26.87**
ReGLU	223M	11.1T	3.57	2.145 ± 0.004	**1.803**	**76.17**	**18.36**	**24.87**	**27.02**
SeLU	223M	11.1T	3.55	2.315 ± 0.004	1.948	68.76	16.76	22.75	25.99
SwiGLU	223M	11.1T	3.53	2.127 ± 0.003	**1.789**	**76.00**	**18.20**	**24.34**	**27.02**
LiGLU	223M	11.1T	3.59	2.149 ± 0.005	**1.798**	**75.34**	**17.97**	**24.34**	26.53
Sigmoid	223M	11.1T	3.63	2.291 ± 0.019	1.867	**74.31**	17.51	23.02	26.30
Softplus	223M	11.1T	3.47	2.207 ± 0.011	1.850	**72.45**	17.65	**24.34**	**26.89**
RMS Norm	223M	11.1T	3.68	2.167 ± 0.008	**1.821**	**75.45**	**17.94**	**24.07**	**27.14**
Rezero	223M	11.1T	3.51	2.262 ± 0.003	1.939	61.69	15.64	20.90	26.37
Rezero + LayerNorm	223M	11.1T	3.26	2.223 ± 0.006	1.858	70.42	17.58	23.02	26.29
Rezero + RMS Norm	223M	11.1T	3.34	2.221 ± 0.009	1.875	70.33	17.32	23.02	26.19
Fixup	223M	11.1T	2.95	2.382 ± 0.012	2.067	58.56	14.42	23.02	26.31
24 layers, $d_{ff} = 1536$, $H = 6$	224M	11.1T	3.33	2.200 ± 0.007	1.843	**74.89**	17.75	**25.13**	**26.89**
18 layers, $d_{ff} = 2048$, $H = 8$	223M	11.1T	3.38	2.185 ± 0.005	**1.831**	**76.45**	16.83	**24.34**	**27.10**
8 layers, $d_{ff} = 4608$, $H = 18$	223M	11.1T	3.69	2.190 ± 0.005	1.847	**74.58**	17.69	**23.28**	**26.85**
6 layers, $d_{ff} = 6144$, $H = 24$	223M	11.1T	3.70	2.201 ± 0.010	1.857	**73.55**	17.59	**24.60**	**26.66**
Block sharing	65M	11.1T	3.91	2.497 ± 0.037	2.164	64.50	14.53	21.96	25.48
+ Factorized embeddings	45M	9.4T	4.21	2.631 ± 0.305	2.183	60.84	14.00	19.84	25.27
+ Factorized & shared embeddings	20M	9.1T	4.37	2.907 ± 0.313	2.385	53.95	11.37	19.84	25.19
Encoder only block sharing	170M	11.1T	3.68	2.298 ± 0.023	1.929	69.60	16.23	23.02	26.23
Decoder only block sharing	144M	11.1T	3.70	2.352 ± 0.029	2.082	67.93	16.13	**23.81**	26.08
Factorized Embedding	227M	9.4T	3.80	2.208 ± 0.006	1.855	70.41	15.92	22.75	26.50
Factorized & shared embeddings	202M	9.1T	3.92	2.320 ± 0.010	1.952	68.69	16.33	22.22	26.44
Tied encoder/decoder input embeddings	248M	11.1T	3.55	2.192 ± 0.002	1.840	**71.70**	17.72	**24.34**	26.49
Tied decoder input and output embeddings	248M	11.1T	3.57	2.187 ± 0.007	**1.827**	**74.86**	17.74	**24.87**	**26.67**
Untied embeddings	273M	11.1T	3.53	2.195 ± 0.005	**1.834**	**72.99**	17.58	**23.28**	26.48
Adaptive input embeddings	204M	9.2T	3.55	2.250 ± 0.002	1.899	66.57	16.21	**24.07**	**26.66**
Adaptive softmax	204M	9.2T	3.60	2.364 ± 0.005	1.982	**72.91**	16.67	21.16	25.56
Adaptive softmax without projection	223M	10.8T	3.43	2.229 ± 0.009	1.914	**71.82**	17.10	23.02	25.72
Mixture of softmaxes	232M	16.3T	2.24	2.227 ± 0.017	**1.821**	**76.77**	17.62	22.75	**26.82**
Transparent attention	223M	11.1T	3.33	2.181 ± 0.014	1.874	54.31	10.40	21.16	**26.80**
Dynamic convolution	257M	11.8T	2.65	2.403 ± 0.009	2.047	58.30	12.67	21.16	17.03
Lightweight convolution	224M	10.4T	4.07	2.370 ± 0.010	1.989	63.07	14.86	23.02	24.73
Evolved Transformer	217M	9.9T	3.09	2.220 ± 0.003	1.863	**73.67**	10.76	**24.07**	26.58
Synthesizer (dense)	224M	11.4T	3.47	2.334 ± 0.021	1.962	61.03	14.27	16.14	**26.63**
Synthesizer (dense plus)	243M	12.6T	3.22	2.191 ± 0.010	1.840	**73.98**	16.96	**23.81**	**26.71**
Synthesizer (dense plus alpha)	243M	12.6T	3.01	2.180 ± 0.007	**1.828**	**74.25**	17.02	**23.28**	26.61
Synthesizer (factorized)	207M	10.1T	3.94	2.341 ± 0.017	1.968	62.78	15.39	**23.55**	26.42
Synthesizer (random)	254M	10.1T	4.08	2.326 ± 0.012	2.009	54.27	10.35	19.56	26.44
Synthesizer (random plus)	292M	12.0T	3.63	2.189 ± 0.004	1.842	**73.32**	17.04	**24.87**	26.43
Synthesizer (random plus alpha)	292M	12.0T	3.42	2.186 ± 0.007	**1.828**	**75.24**	17.08	**24.08**	26.39
Universal Transformer	84M	40.0T	0.88	2.406 ± 0.036	2.053	70.13	14.09	19.05	23.91
Mixture of experts	648M	11.7T	3.20	2.148 ± 0.006	**1.785**	**74.55**	**18.13**	**24.08**	**26.94**
Switch Transformer	1100M	11.7T	3.18	2.135 ± 0.007	**1.758**	**75.38**	**18.02**	**26.19**	**26.81**
Funnel Transformer	223M	1.9T	4.30	2.288 ± 0.008	1.918	67.34	16.26	22.75	23.20
Weighted Transformer	280M	71.0T	0.59	2.378 ± 0.021	1.989	69.04	16.98	23.02	26.30
Product key memory	421M	386.6T	0.25	2.155 ± 0.003	**1.798**	**75.16**	17.04	**23.55**	**26.73**

Do Transformer Modifications Transfer Across Implementations and Applications?

Sharan Narang* Hyung Won Chung Yi Tay William Fedus
Thibault Fevry† Michael Matena† Karishma Malkan† Noah Fiedel
Noam Shazeer Zhenzhong Lan† Yanqi Zhou Wei Li
Nan Ding Jake Marcus Adam Roberts Colin Raffel†

Transformer

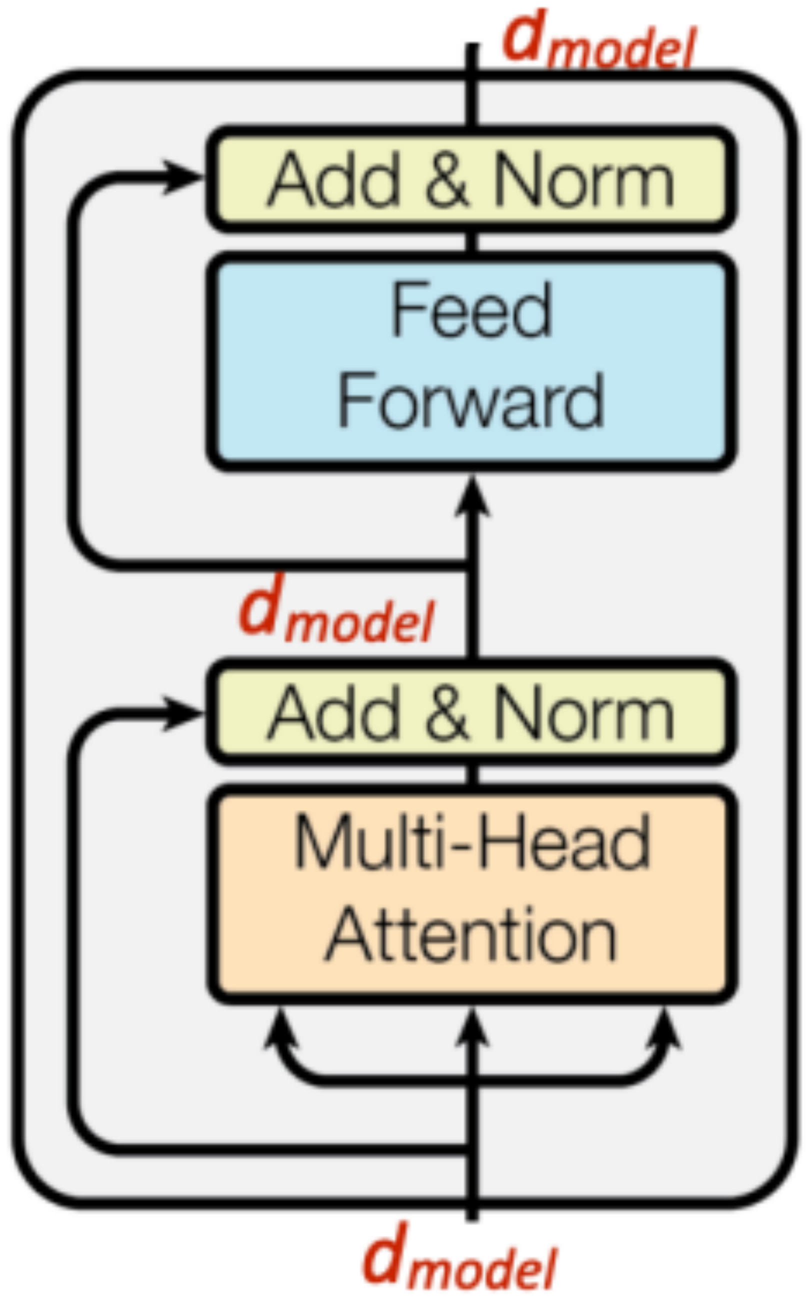
- Transformer architecture specifications

	N	d_{model}	d_{ff}	h	d_k	d_v
base	6	512	2048	8	64	64

  - From Vaswani et al.

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}
GPT-3 Small	125M	12	768	12	64
GPT-3 Medium	350M	24	1024	16	64
GPT-3 Large	760M	24	1536	16	96
GPT-3 XL	1.3B	24	2048	24	128
GPT-3 2.7B	2.7B	32	2560	32	80
GPT-3 6.7B	6.7B	32	4096	32	128
GPT-3 13B	13.0B	40	5140	40	128
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128

  - From GPT-3; d_{head} is our d_k

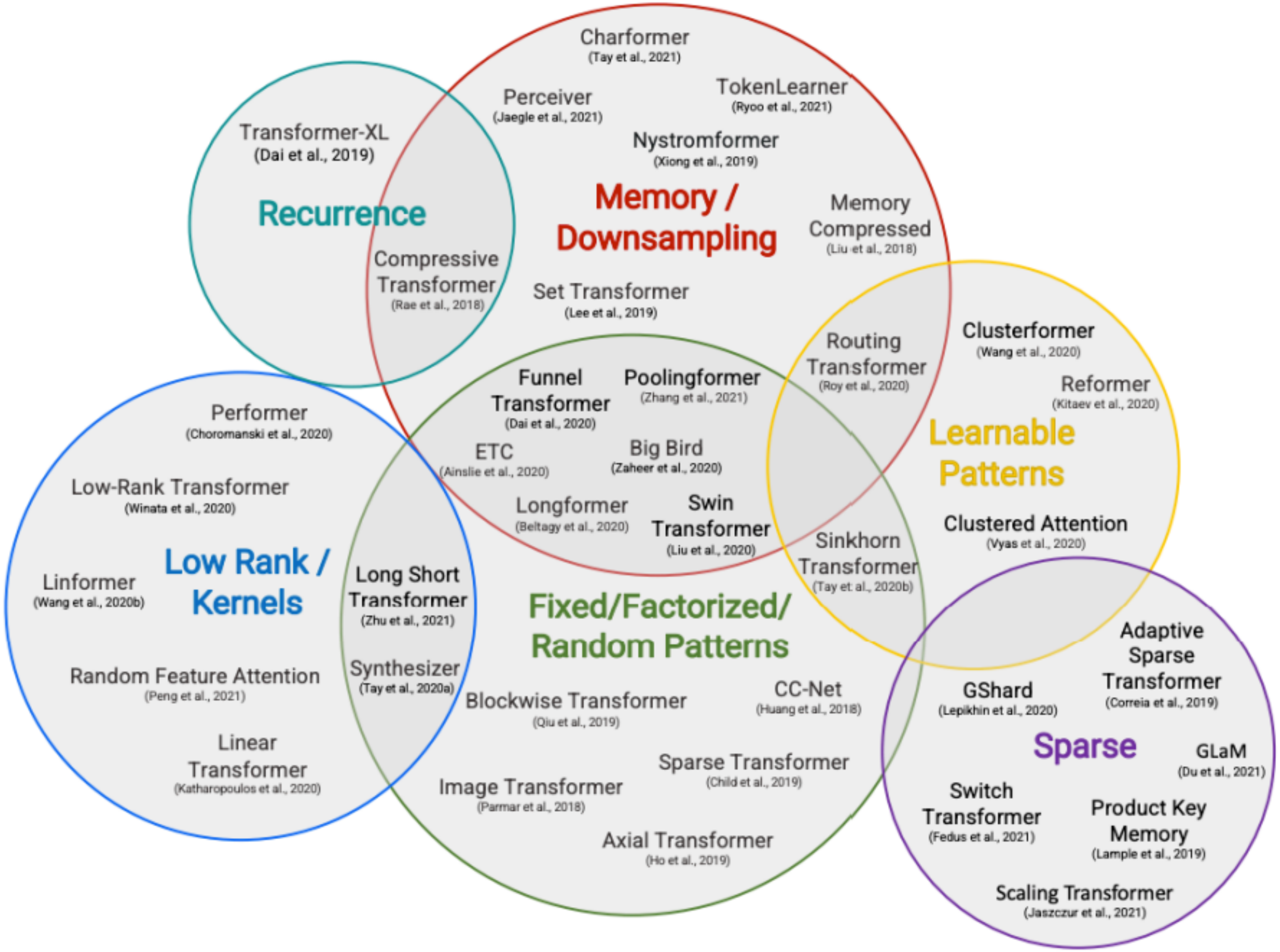


Transformer

- Advantages
  - Easier to capture long-range dependencies: we draw attention between every pair of word
  - Easier to parallelize
- Drawbacks
  - Are positional encodings enough to capture positional information?
    - Otherwise self-attention is an un unordered function of its input
  - Quadratic computation in self-attention
    - Can become very slow when the sequence length is large

Transformer

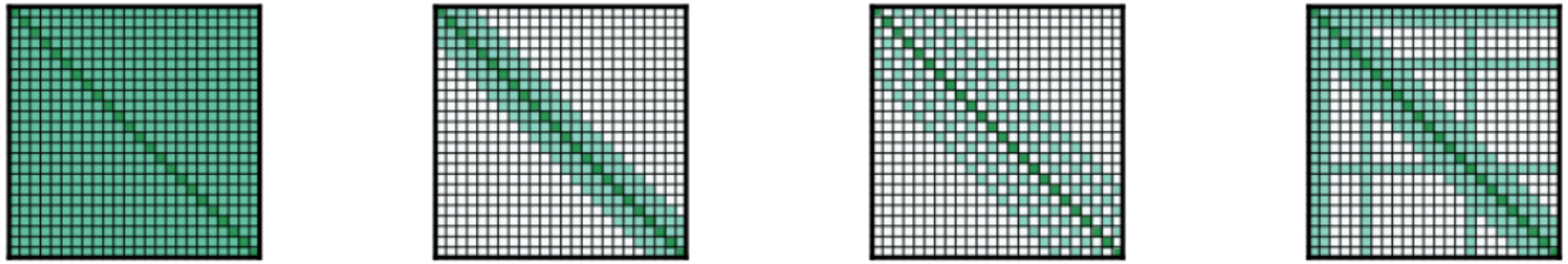
- Efficient Transformers



Transformer

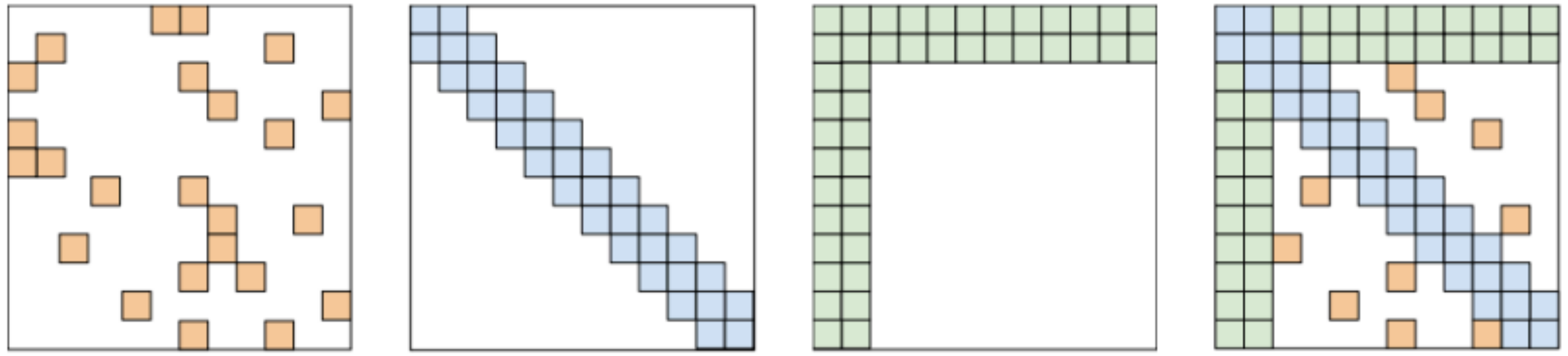
- Longformer / Big Bird

Key idea: use sparse attention patterns!



(a) Full n^2 attention (b) Sliding window attention (c) Dilated sliding window (d) Global+sliding window

(Beltagy et al., 2020): Longformer: The Long-Document Transformer



(a) Random attention (b) Window attention (c) Global Attention (d) BigBird

(Zaheer et al., 2021): Big Bird: Transformers for Longer Sequences

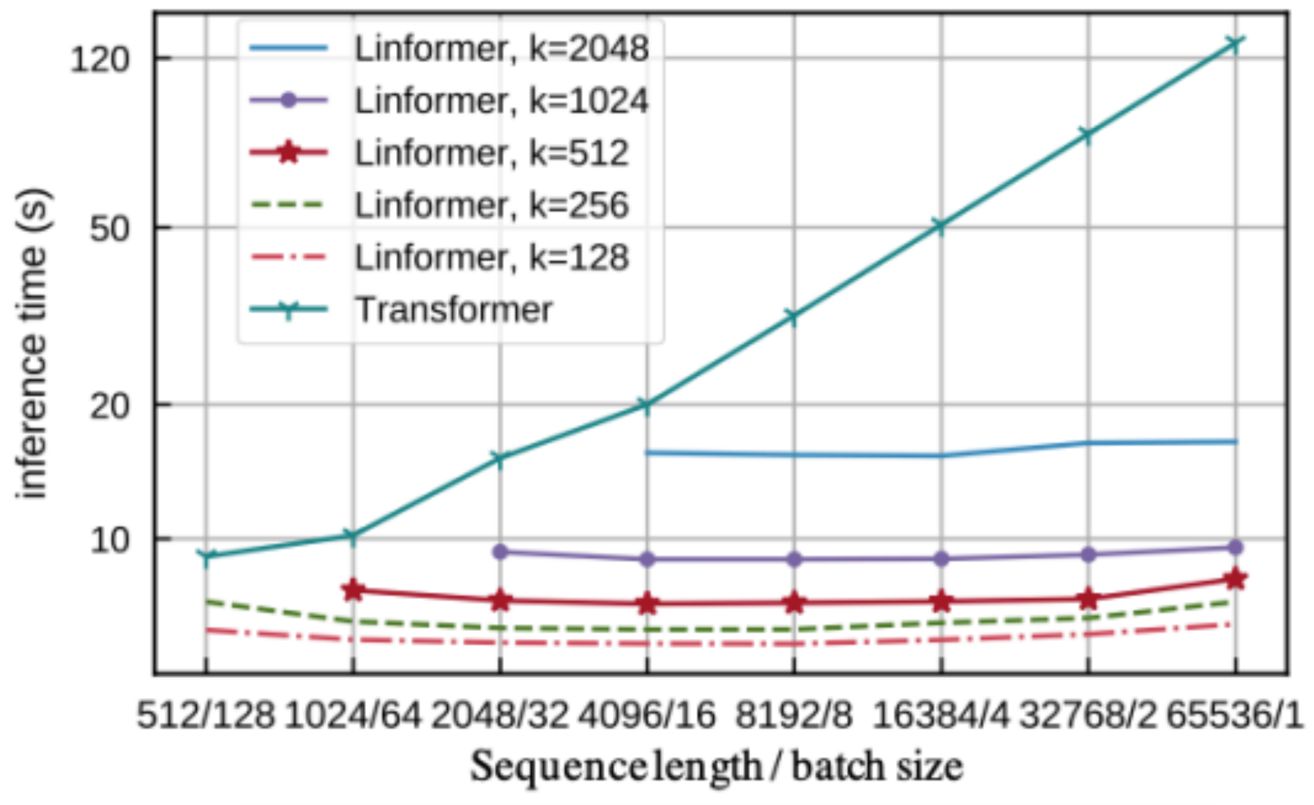
Transformer

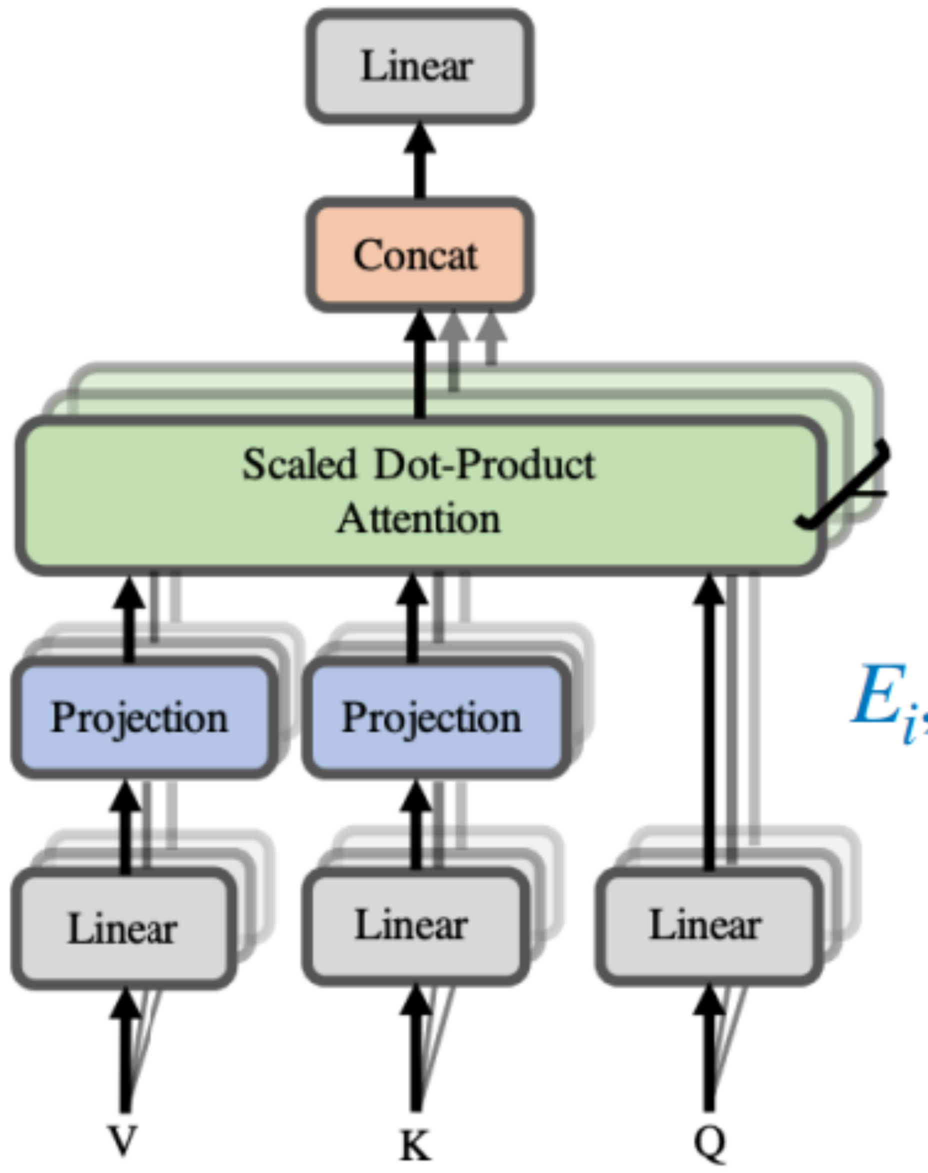
- Linformer

Key idea: The attention matrix $e_{i,j}$ can be approximated by a low-rank matrix

Map the sequence length dimension to a lower-dimensional space for values, keys

$$\overline{\mathrm{head}_i} = \mathrm{Attention}(QW_i^Q, E_iKW_i^K, F_iVW_i^V)$$
$$= \underbrace{\mathrm{softmax}\left(\frac{QW_i^Q(E_iKW_i^K)^T}{\sqrt{d_k}}\right)}_{\bar{P}:n\times k} \cdot \underbrace{F_iVW_i^V}_{k\times d},$$



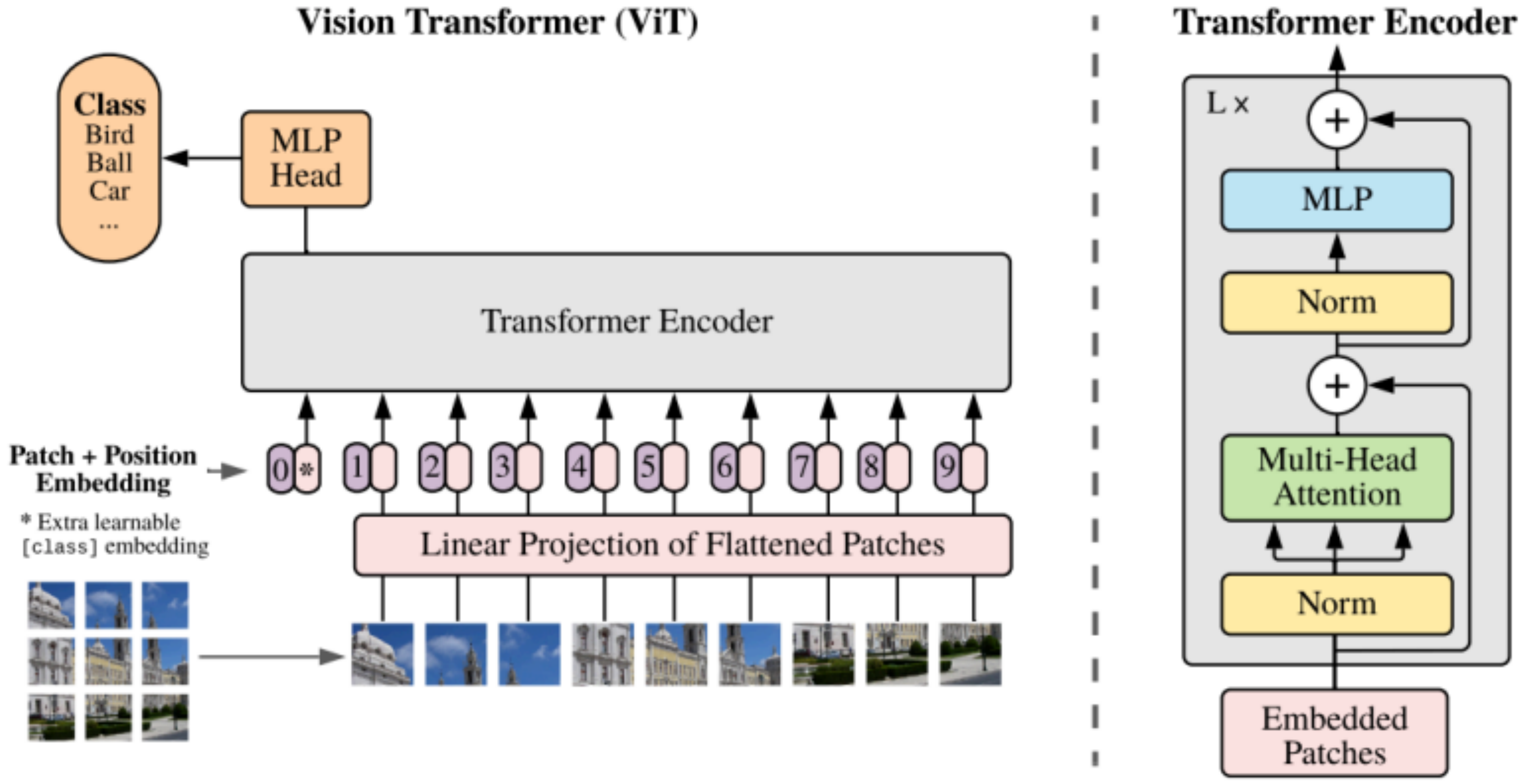


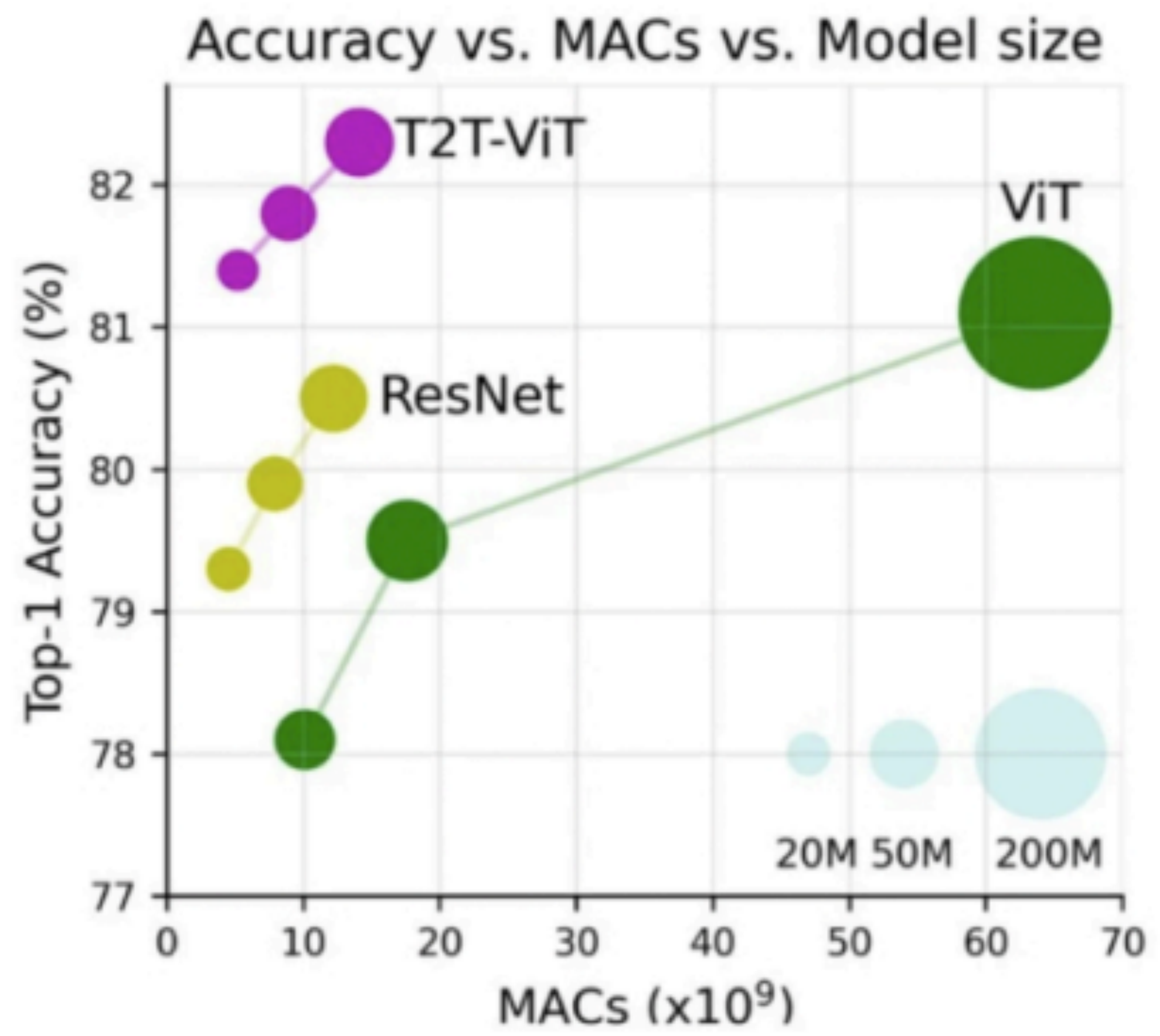
$E_i, F_i \in \mathbb{R}^{k\times n}$

(Wang et al., 2020): Linformer: Self-Attention with Linear Complexity

Transformer

- Vision Transformer (ViT)

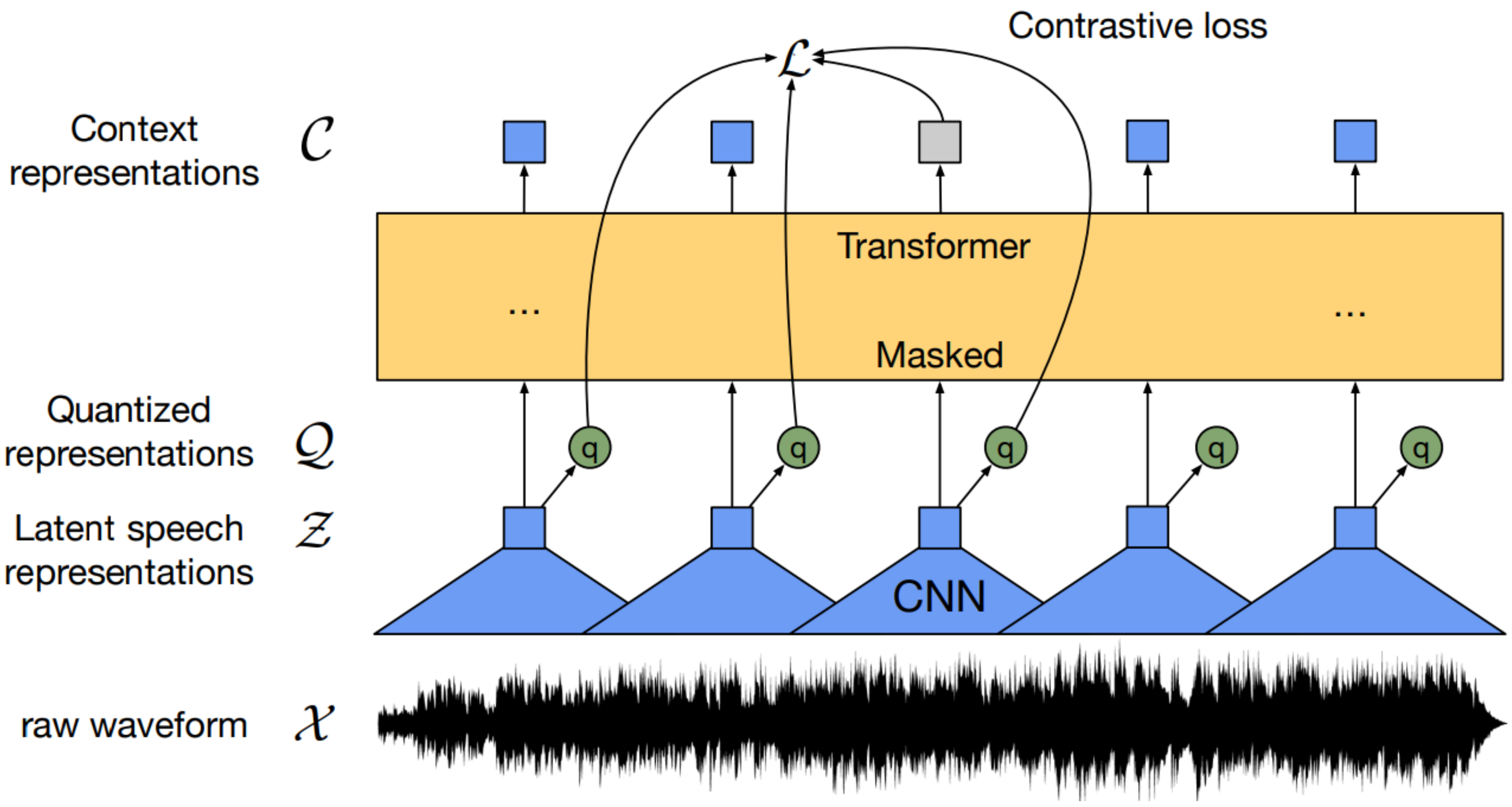




(Dosovitskiy et al., 2021): An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

Transformer

- Wav2vec 2.0



(Baevski et al., 2020): wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations