CSCI 5541: Natural Language Processing

Lecture 10: Deep Dive on Transformers

Dongyeop Kang (DK), University of Minnesota

dongyeop@umn.edu | twitter.com/dongyeopkang | dykang.github.io

Courtesy of Paramount Pictures

Using some slides borrowed from Anna Goldie (Google Brain) and John Hweitt (Stanford)

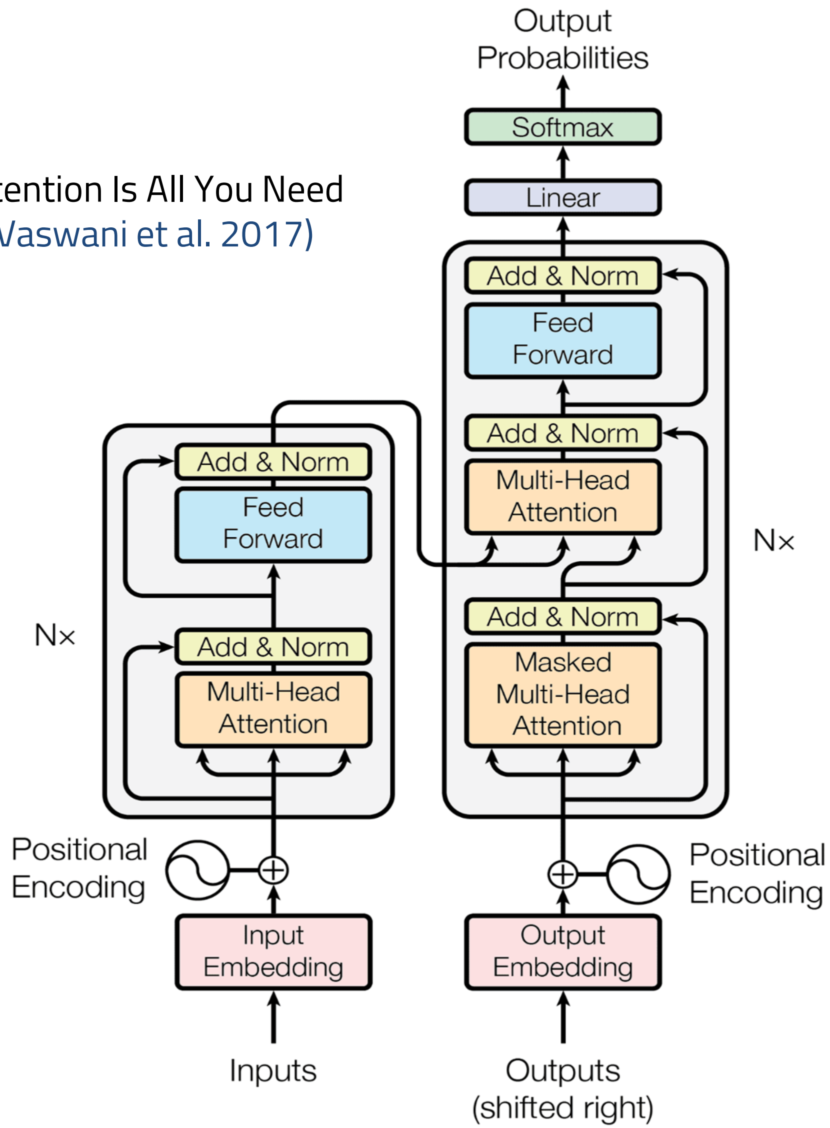
University of Minnesota

Driven to Discover®

Summary of Transformers

- A sequence-to-sequence model based entirely on attention
- Strong results on translation and a wide variety of other tasks
- Fast: only matrix multiplications

Attention Is All You Need
(Vaswani et al. 2017)



Strong results/findings and applications of Transformers

Strong results with Transformers on machine translation

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)

Strong results with Transformers on document summarization

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

WikiSum dataset (Liu et al., 2018)

Strong results with (pre-trained) Transformers on classification tasks

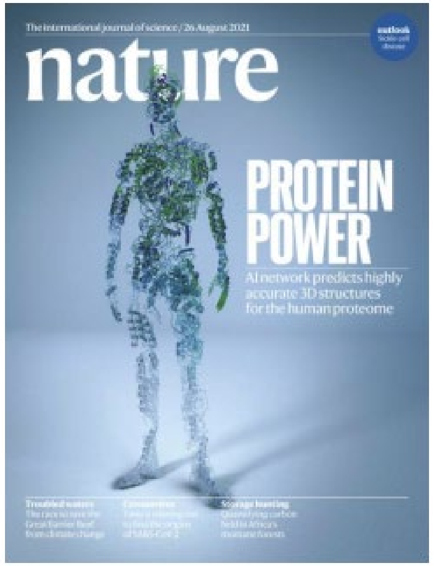
Sentiment classification on SST-2 dataset

Rank	Model	Accuracy	Paper	Code	Result	Year	Tags
1	**SMART-RoBERTa Large**	97.5	SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models through Principled Regularized Optimization			2019	Transformer
2	**T5-3B**	97.4	Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer			2019	Transformer
3	**MUPPET Roberta Large**	97.4	Muppet: Massive Multi-task Representations with Pre-Finetuning			2021	
4	**ALBERT**	97.1	ALBERT: A Lite BERT for Self-supervised Learning of Language Representations			2019	Transformer
5	**T5-11B**	97.1	Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer			2019	Transformer
6	**StructBERTRoBERTa ensemble**	97.1	StructBERT: Incorporating Language Structures into Pre-training for Deep Language Understanding			2019	Transformer
7	**XLNet** (single model)	97	XLNet: Generalized Autoregressive Pretraining for Language Understanding			2019	Transformer
8	**ELECTRA**	96.9	ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators			2020	
9	**EFL**	96.9	Entailment as Few-Shot Learner			2021	Transformer
10	**XLNet-Large** (ensemble)	96.8	XLNet: Generalized Autoregressive Pretraining for Language Understanding			2019	Transformer
11	**RoBERTa**	96.7	RoBERTa: A Robustly Optimized BERT Pretraining Approach			2019	Transformer

https://paperswithcode.com/

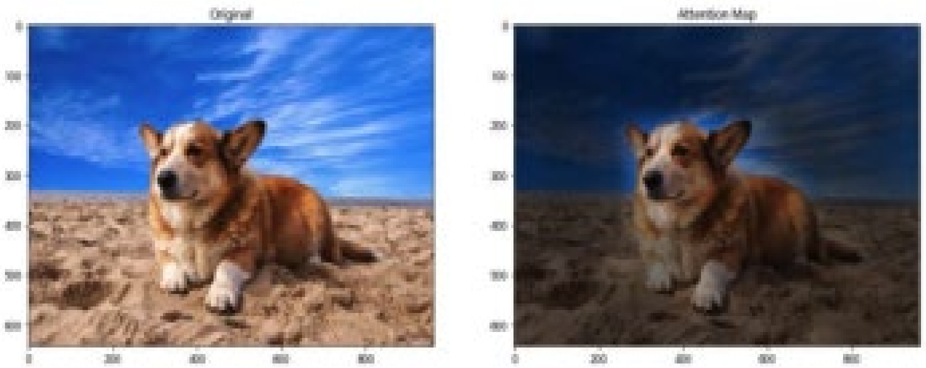
Transformers used outside of NLP

Protein folding



Alpha?Fold2 (Jumper et al., 2021)

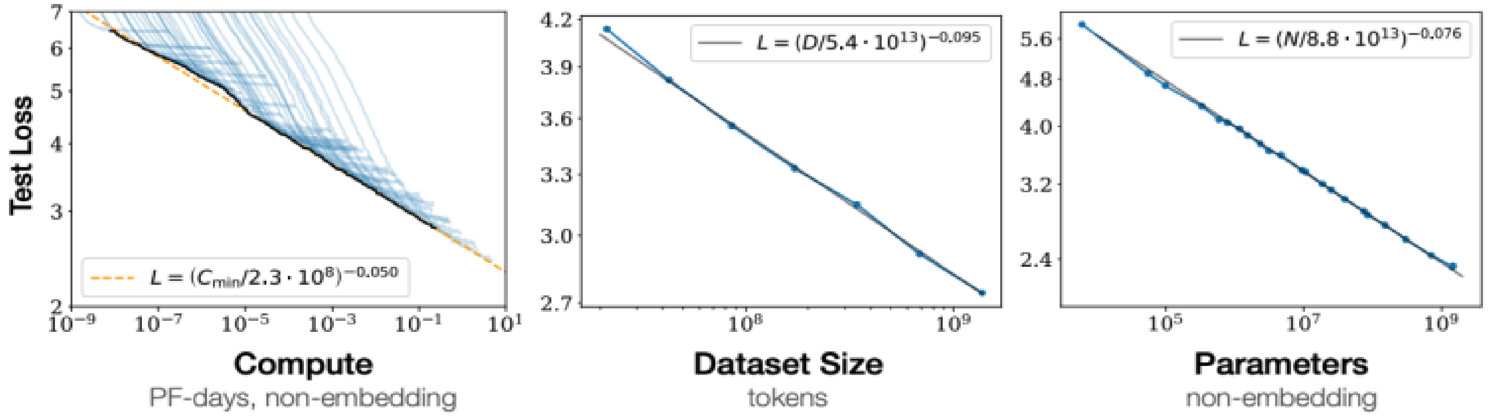
Image Classification



Vision Transformer (ViT) outperforms ResNet-based baselines with substantially less compute (Dosovitskiy et al. 2020)

Scaling laws

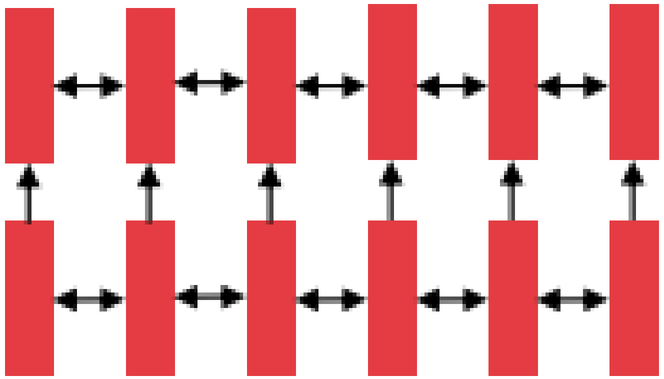
- With Transformers, language modeling performance improves smoothly as we increase model size, training data, and computing resources.
- This power-law relationship has been observed over multiple orders of magnitude with no sign of slowing down!
- If we keep scaling up these models (with no change to the architecture), could they eventually match or exceed human-level performance?



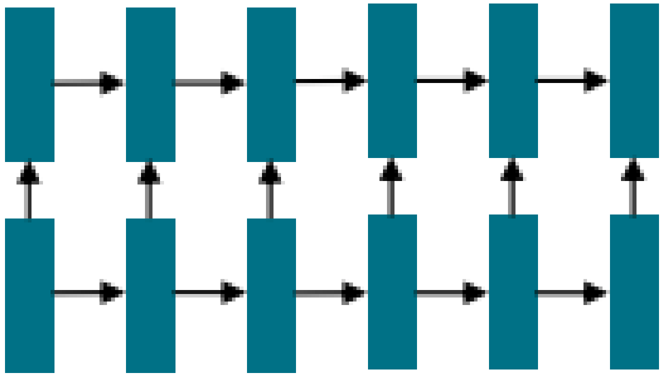
Kaplan et al., 2020, *Scaling Laws for Neural Language Models*

Why self-attention?

Recurrence in RNNs

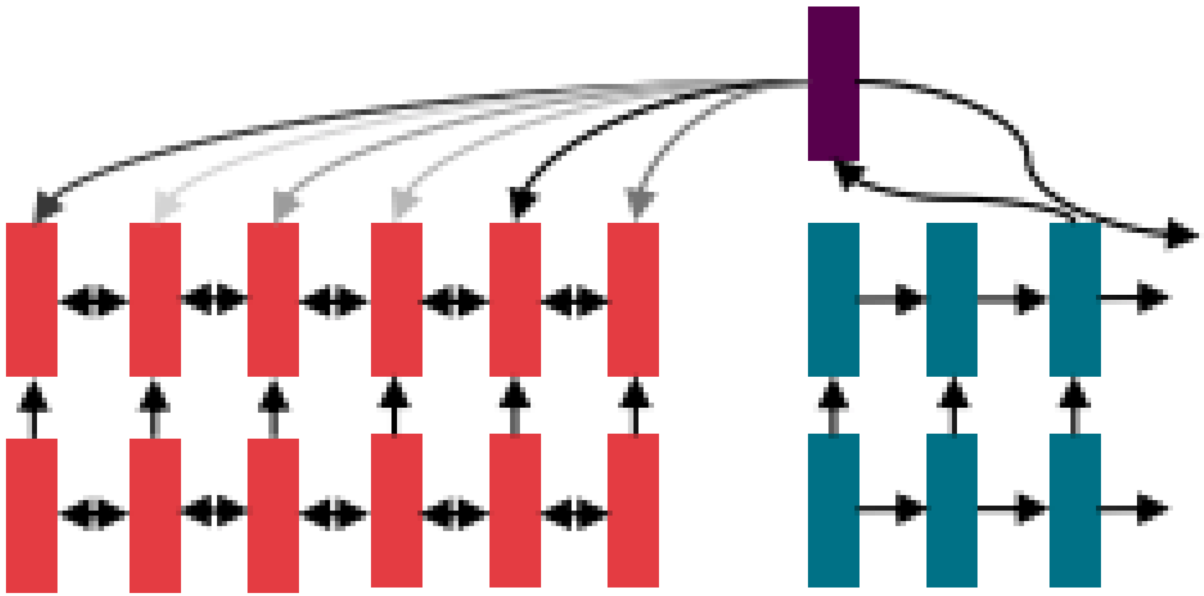


Encoding: Encode input sentences with bi-directional LSTM



Decoding: Define your outputs (parse, sentence, summary) as a sequence/label, and use LSTM to decode it.

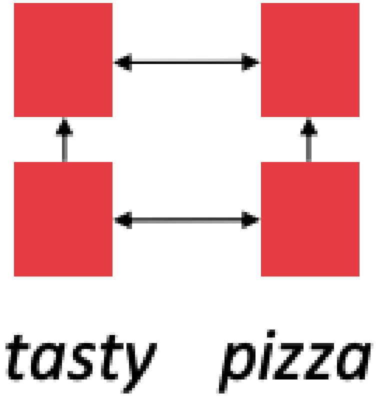
Sequence-to-sequence with attention



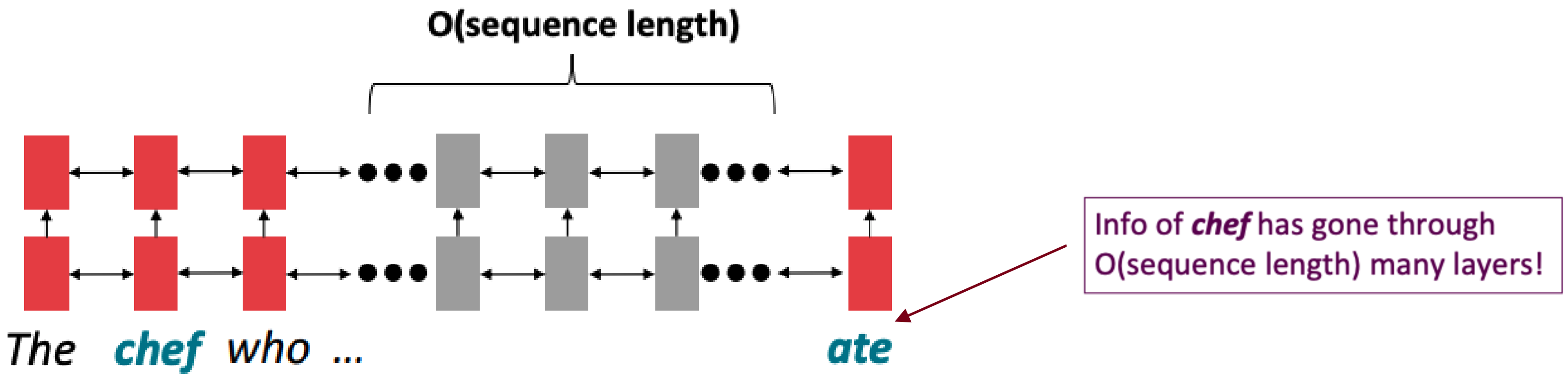
Use **attention** to allow
flexible access to memory

Issues with recurrent models: **Linear interaction distance**

- Forward RNNs are unrolled “left-to-right”.
- It encodes linear locality:
  - Nearby words often affect each other’s meanings

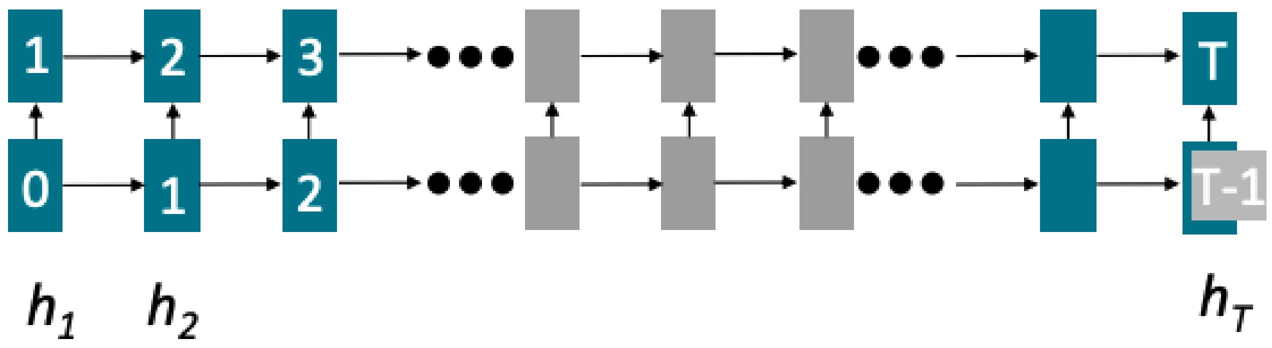


- **Problem**: RNNs take **O(sequence length) steps** for distant word pairs to interact



Issues with recurrent models: **Lack of parallelizability**

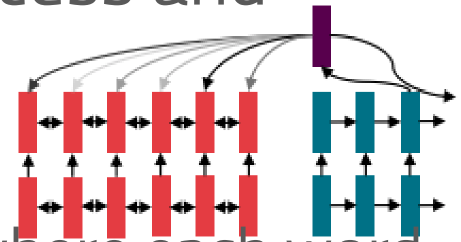
- Forward and backward passes have **O(seq length) un-parallelizable** operations
  - GPUs (and TPUs) can perform many independent computations at once!
  - But future RNN hidden states can't be computed fully before past RNN hidden states have been computed
  - Particularly problematic as sequence length increases, as we can no longer batch many examples together due to memory limitations

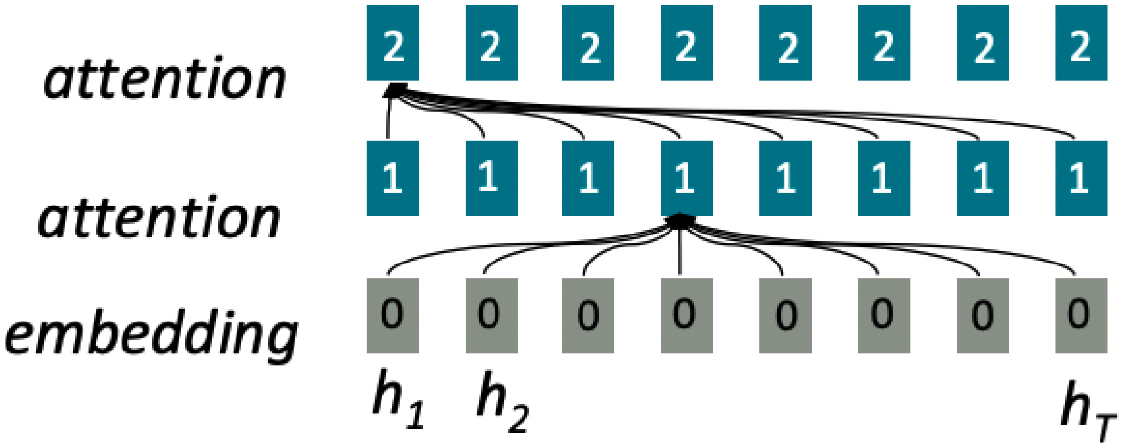


Numbers indicate min # of steps before a state can be computed

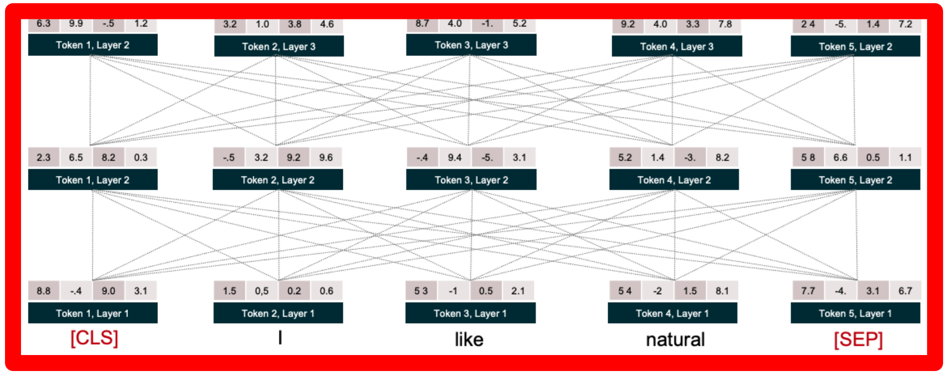
If not recurrence, then what? How about (self) attention?

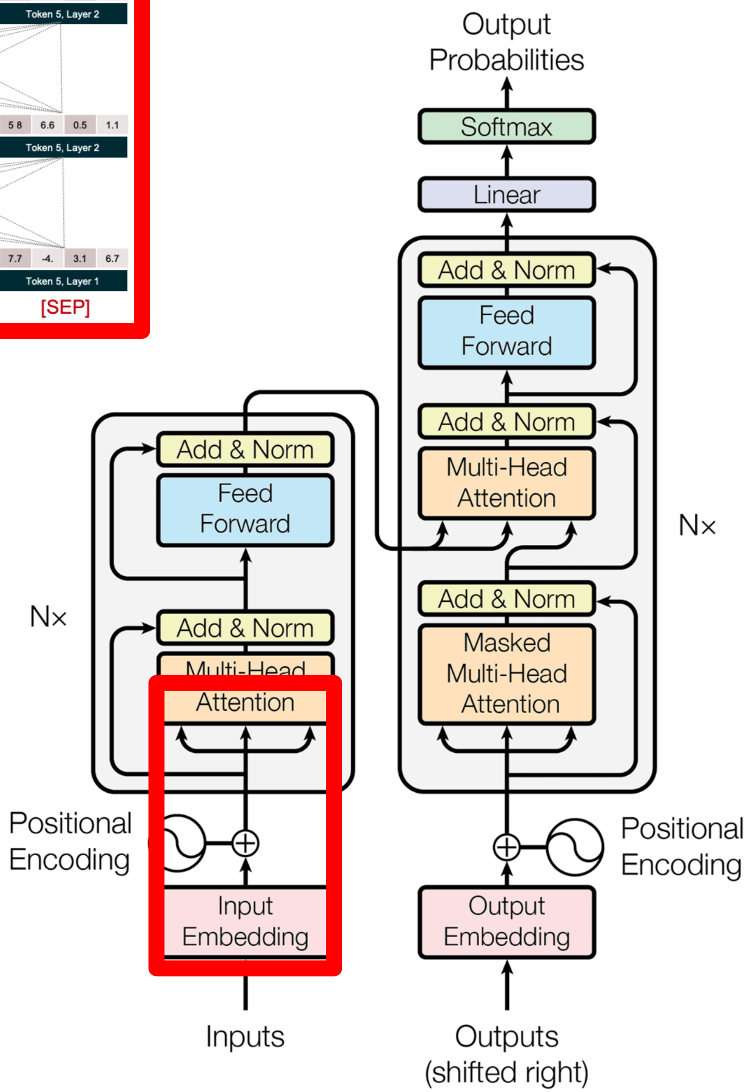
- **Attention** treats each word's representation as a **query** to access and incorporate information from a **set of values**.
  - We saw attention from the decoder to the encoder;
  - Self-attention is encoder-encoder (or decoder-decoder) attention where each word attends to each other word within the input (or output).





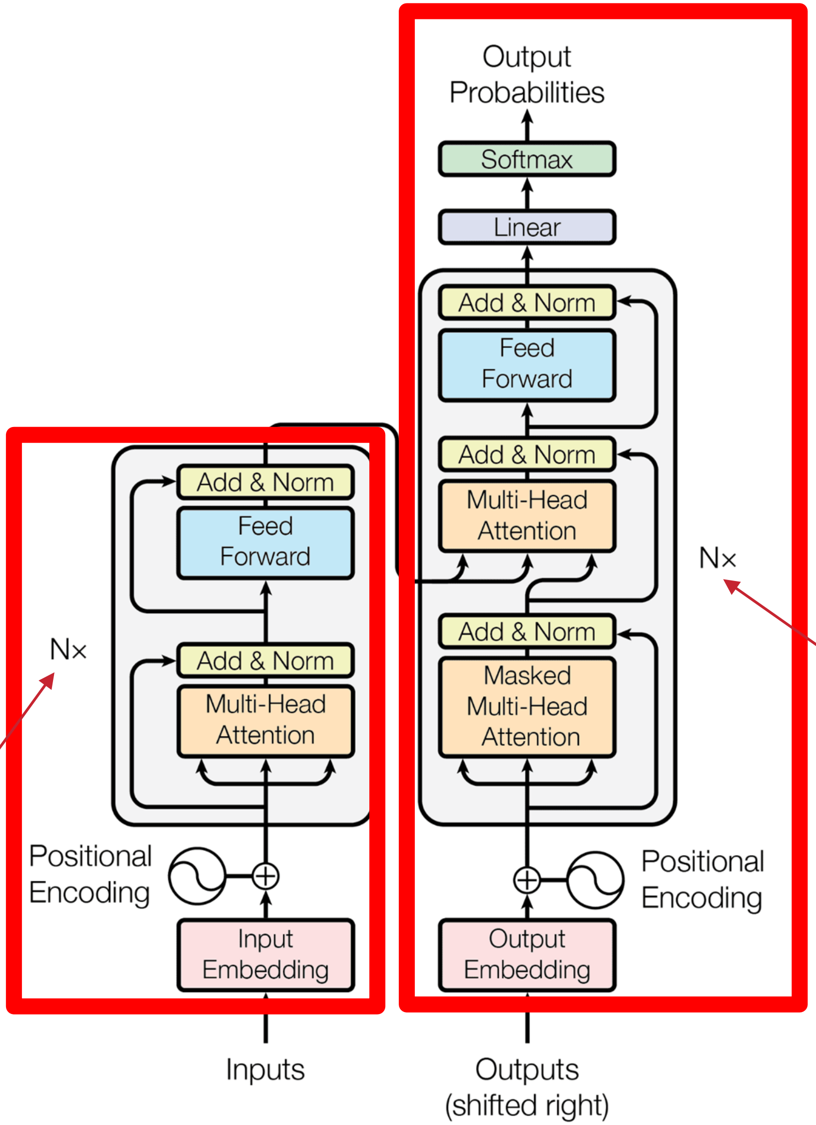
All words attend to all words in previous layer; most arrows are omitted


Token 1, Layer 2
Token 2, Layer 3
Token 3, Layer 3
Token 4, Layer 3
Token 5, Layer 2
Token 1, Layer 2
Token 2, Layer 2
Token 3, Layer 2
Token 4, Layer 2
Token 5, Layer 2
Token 1, Layer 1
Token 2, Layer 1
Token 3, Layer 1
Token 4, Layer 1
Token 5, Layer 1
[CLS]
I
like
natural
[SEP]


Output
Probabilities
Softmax
Linear
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Add & Norm
Masked
Multi-Head
Attention
Nx
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Nx
Positional
Encoding
Input
Embedding
Positional
Encoding
Output
Embedding
Inputs
Outputs
(shifted right)

Encoder

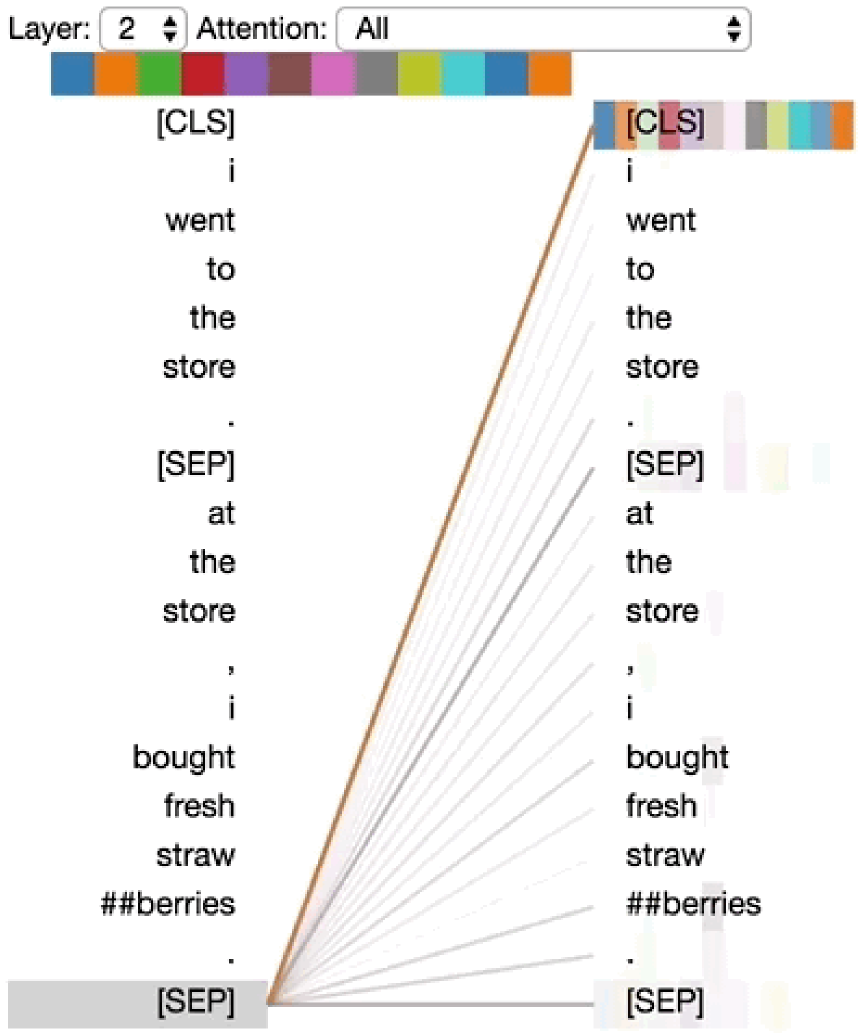
Repeat N times (number of layers)

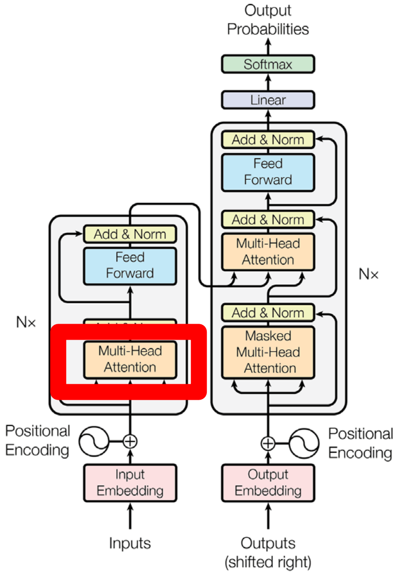


Decoder

Repeat N times (number of layers)

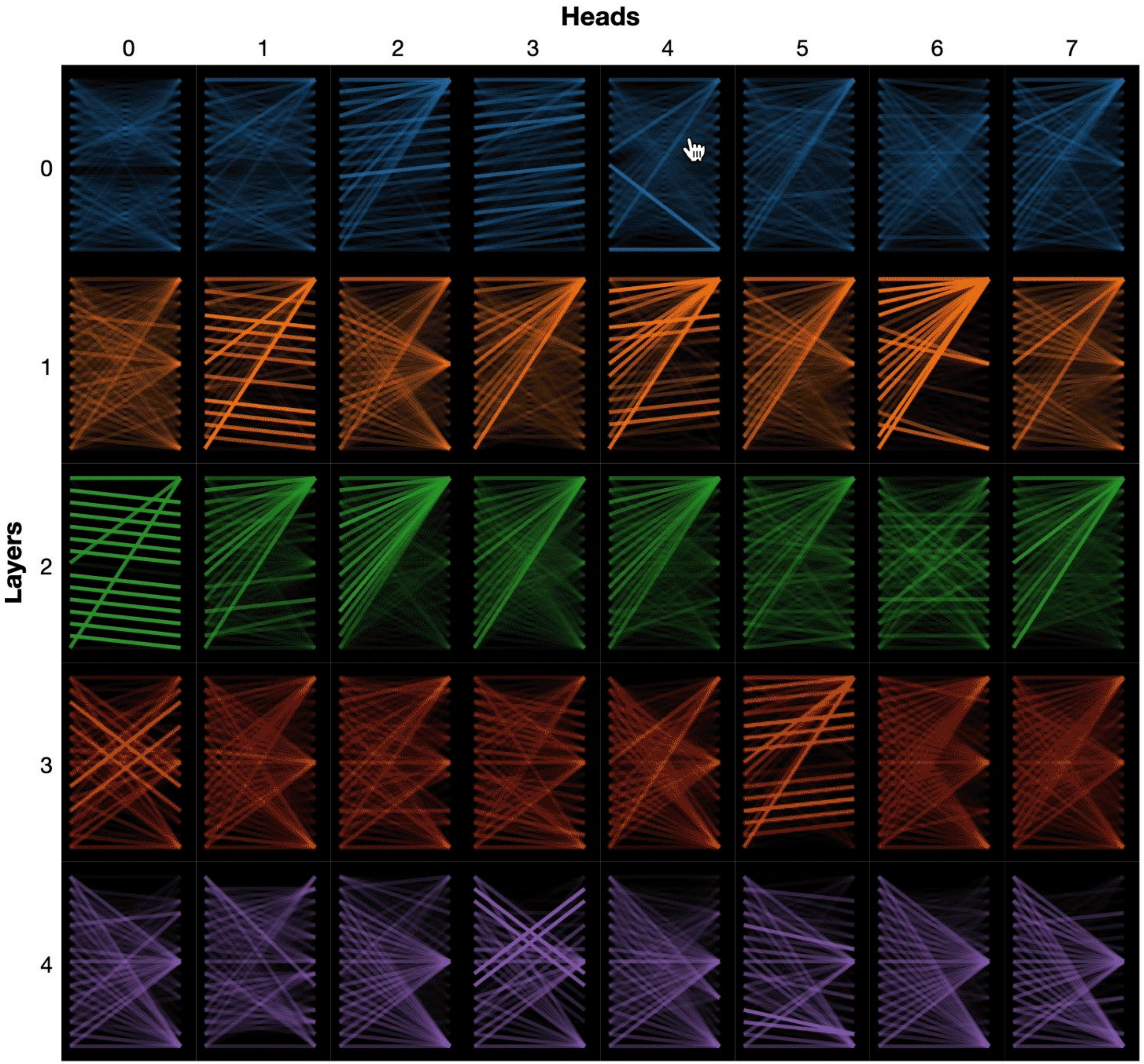
"I went to the store. At the store, I bought fresh strawberries."

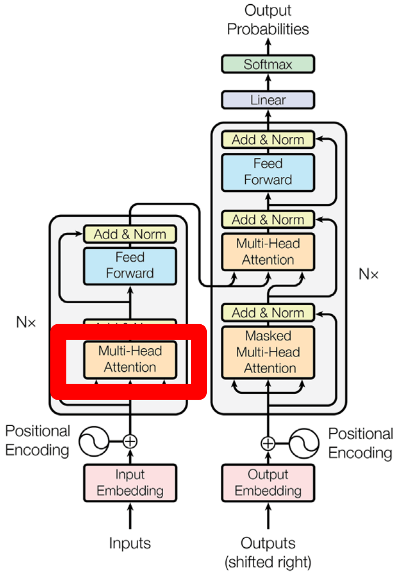




https://github.com/jessevig/bertviz

https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb

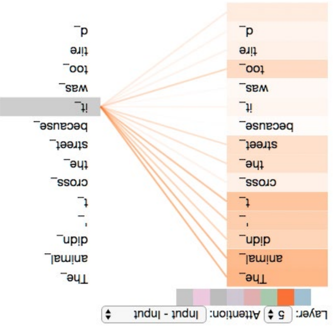


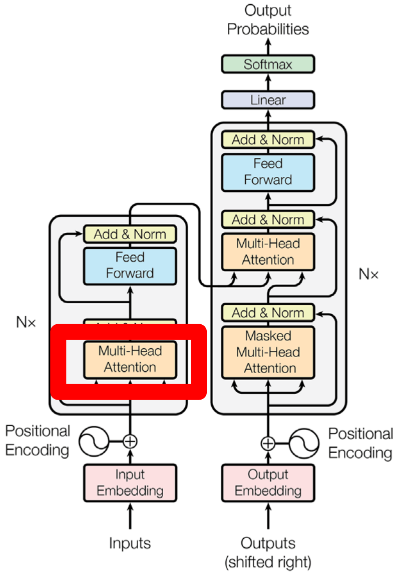


https://github.com/jessevig/bertviz

https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb

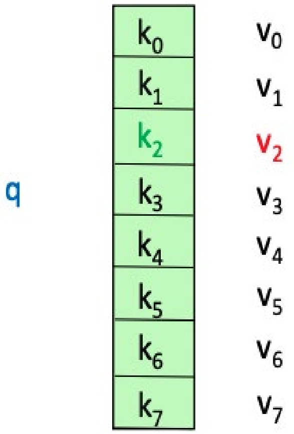
Encoder: Self-Attention

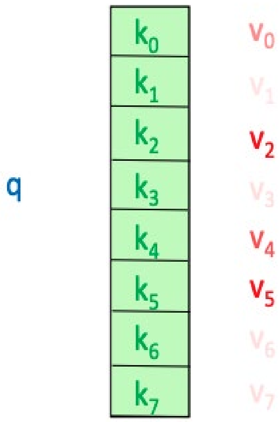




Let's think of attention as a "fuzzy" or approximate ha

- To look up a value, we compare a query against keys in a ta
- In a hashtable
  - Each query (hash) maps to exactly one key-value pair.
- In (self-)attention:
  - Each query matches each key to varying degrees.
  - We return a sum of values weighted by the query-key match.





Recipe for Self-Attention in the Transformer Encoder

- Step 1: For each word , calculate its query, key, and value.

$$q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i$$

- Step 2: Calculate attention score between query and keys.

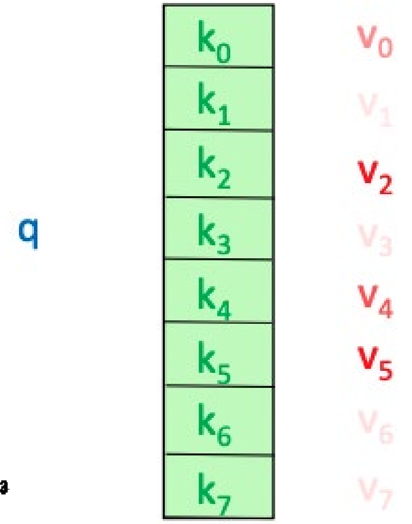
$$e_{ij} = q_i \cdot k_j$$

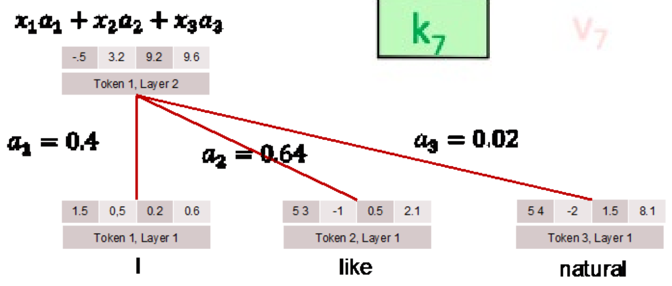
- Step 3: Take the softmax to normalize attention scores.

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(e_{ij})}{\sum_k exp(e_{ik})}$$

- Step 4: Take a weighted sum of values.

$$Output_i = \sum_j \alpha_{ij} v_j$$





Recipe for (Vectorized) Self-Attention in the Transformer Encoder

- Step 1: For each word , calculate its query, key, and value.

$$Q = XW^Q \quad K = XW^K \quad V = XW^V$$

- Step 2: Calculate attention score between query and keys.

$$E = QK^T$$

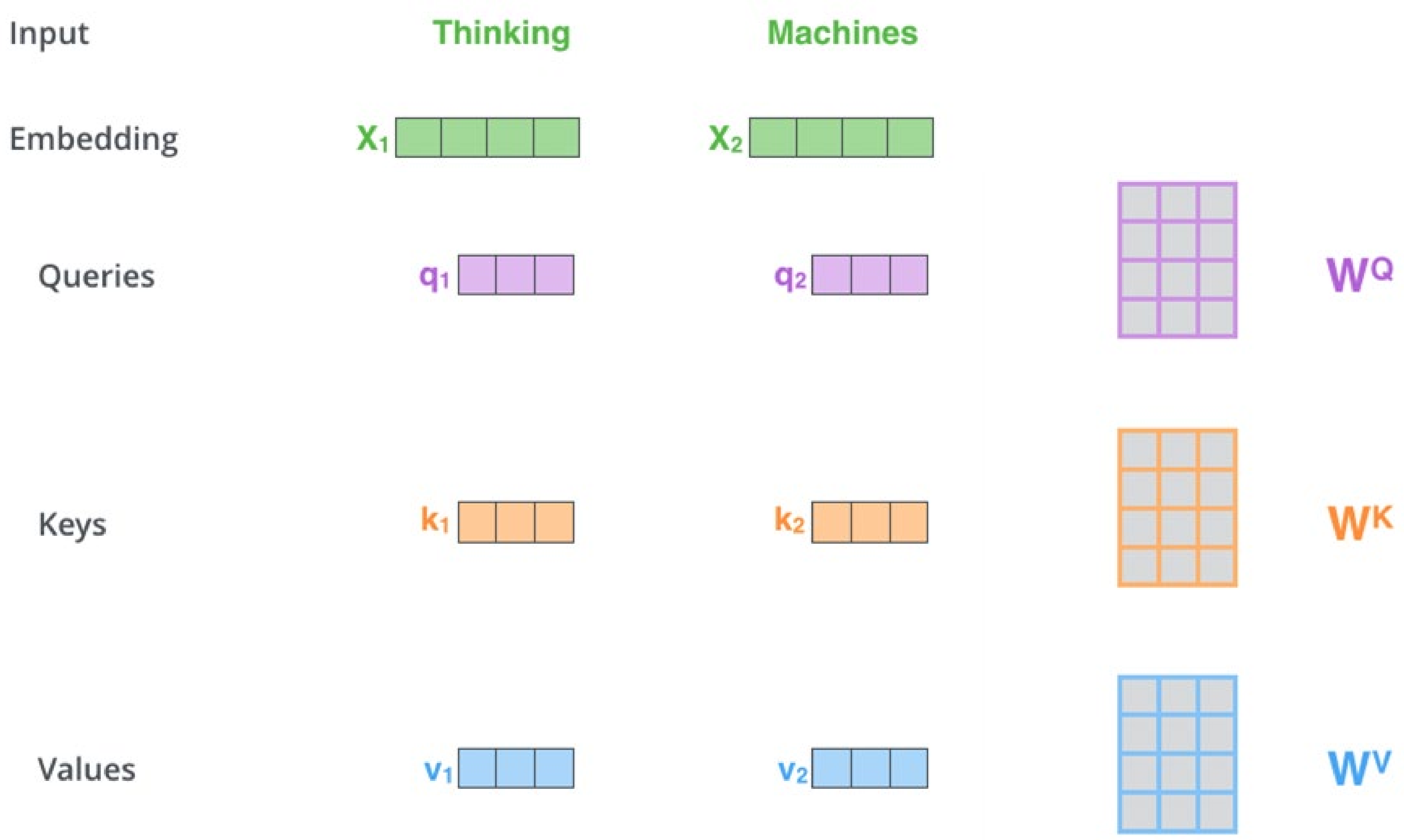
- Step 3: Take the softmax to normalize attention scores.

$$A = softmax(E)$$

- Step 4: Take a weighted sum of values.

$$Output = AV$$

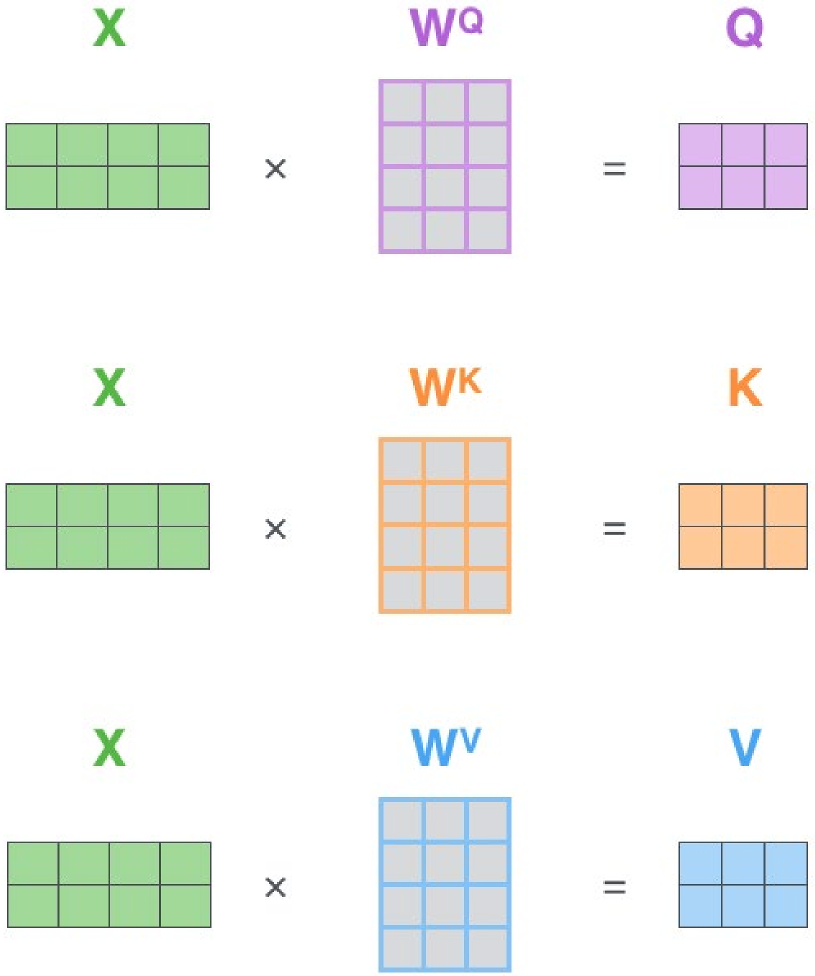
$$\boxed{Output = softmax\left(QK^T\right)V}$$



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

- Step 1: For each word , calculate its query, key, and value.

$$Q = XW^Q \quad K = XW^K \quad V = XW^V$$



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

- Step 2: Calculate attention score between query and keys.

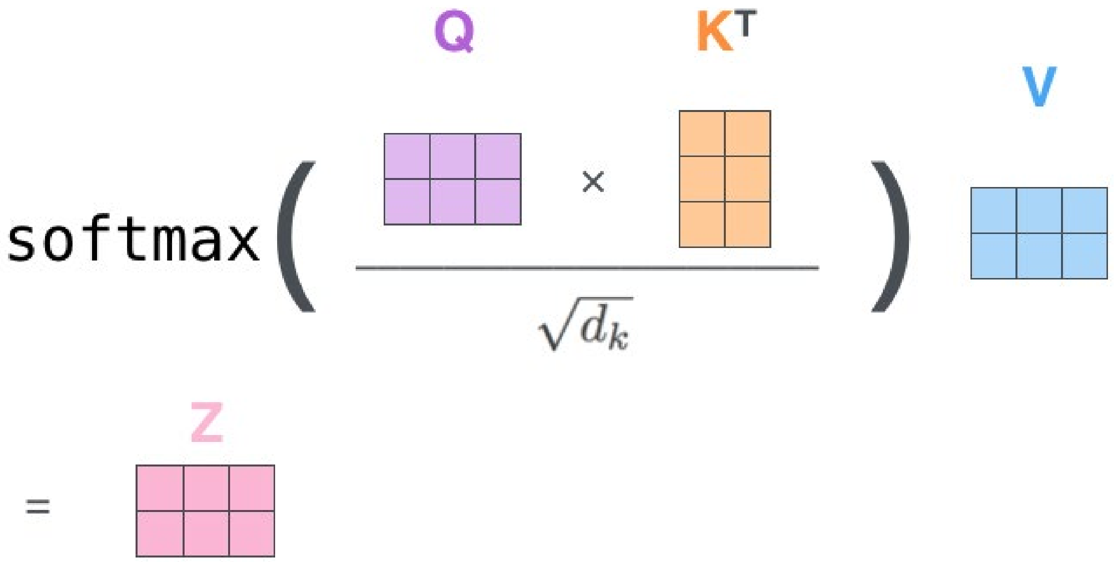
$$E = QK^T$$

- Step 3: Take the softmax to normalize attention scores.

$$A = softmax(E)$$

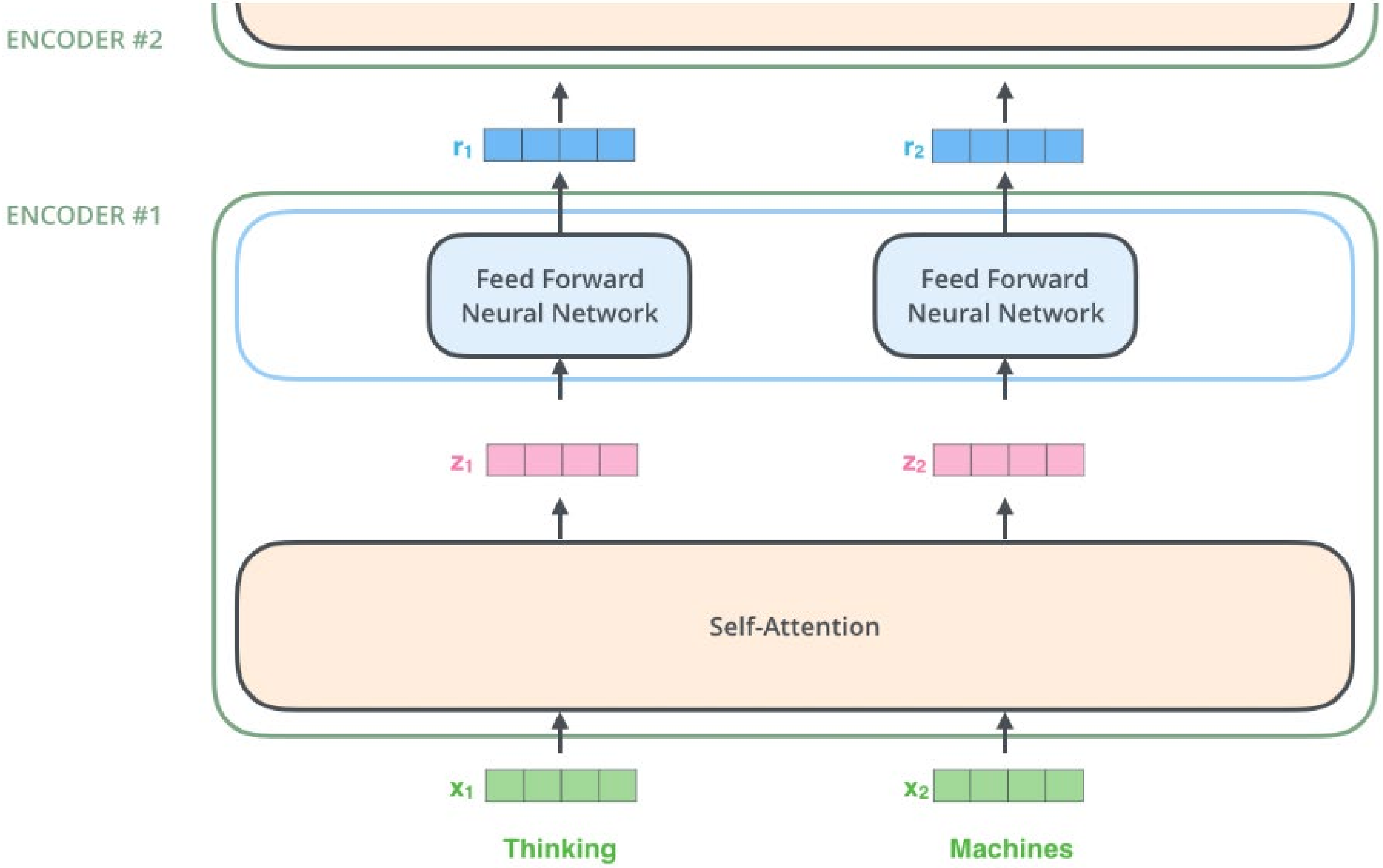
- Step 4: Take a weighted sum of values.

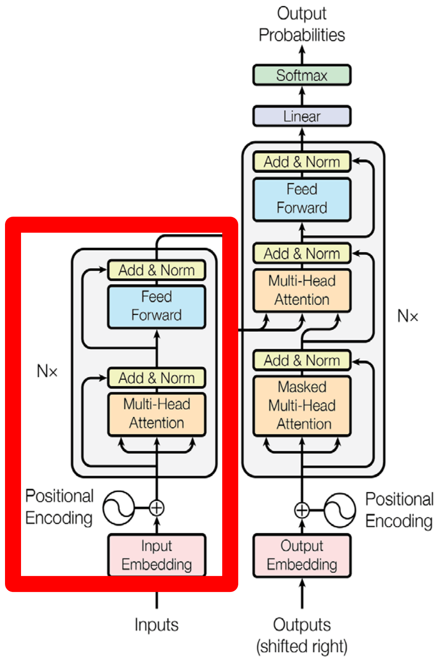
$$Output = AV$$



$$\boxed{Output = softmax\left(QK^T\right)V}$$

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

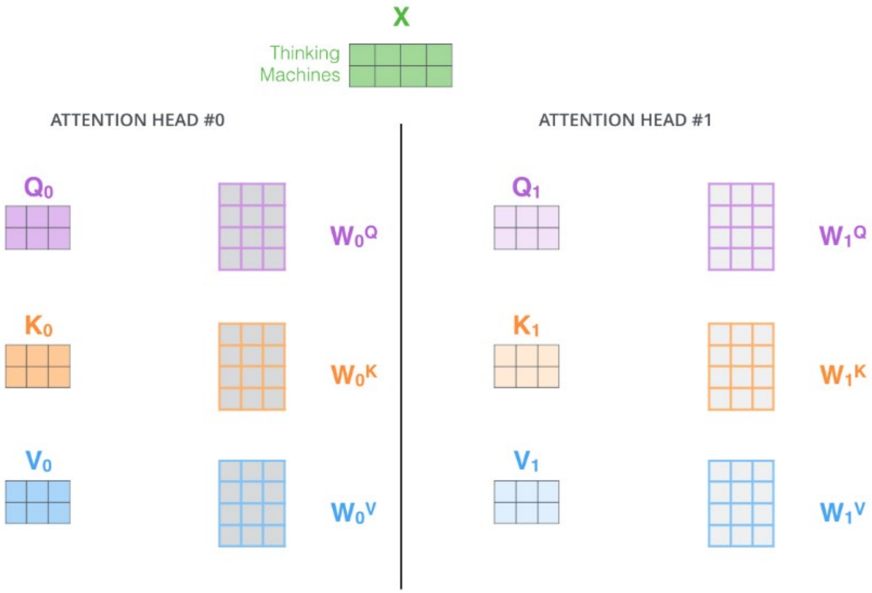


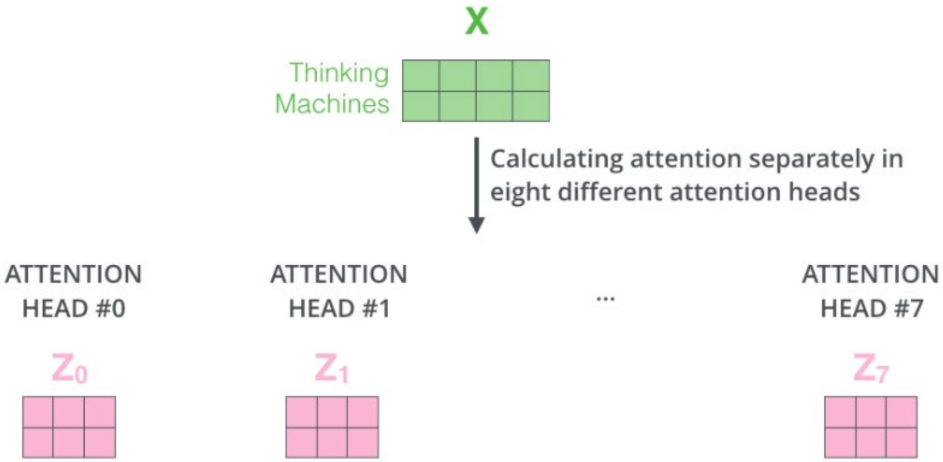


https://jalammar.github.io/illustrated-transformer/

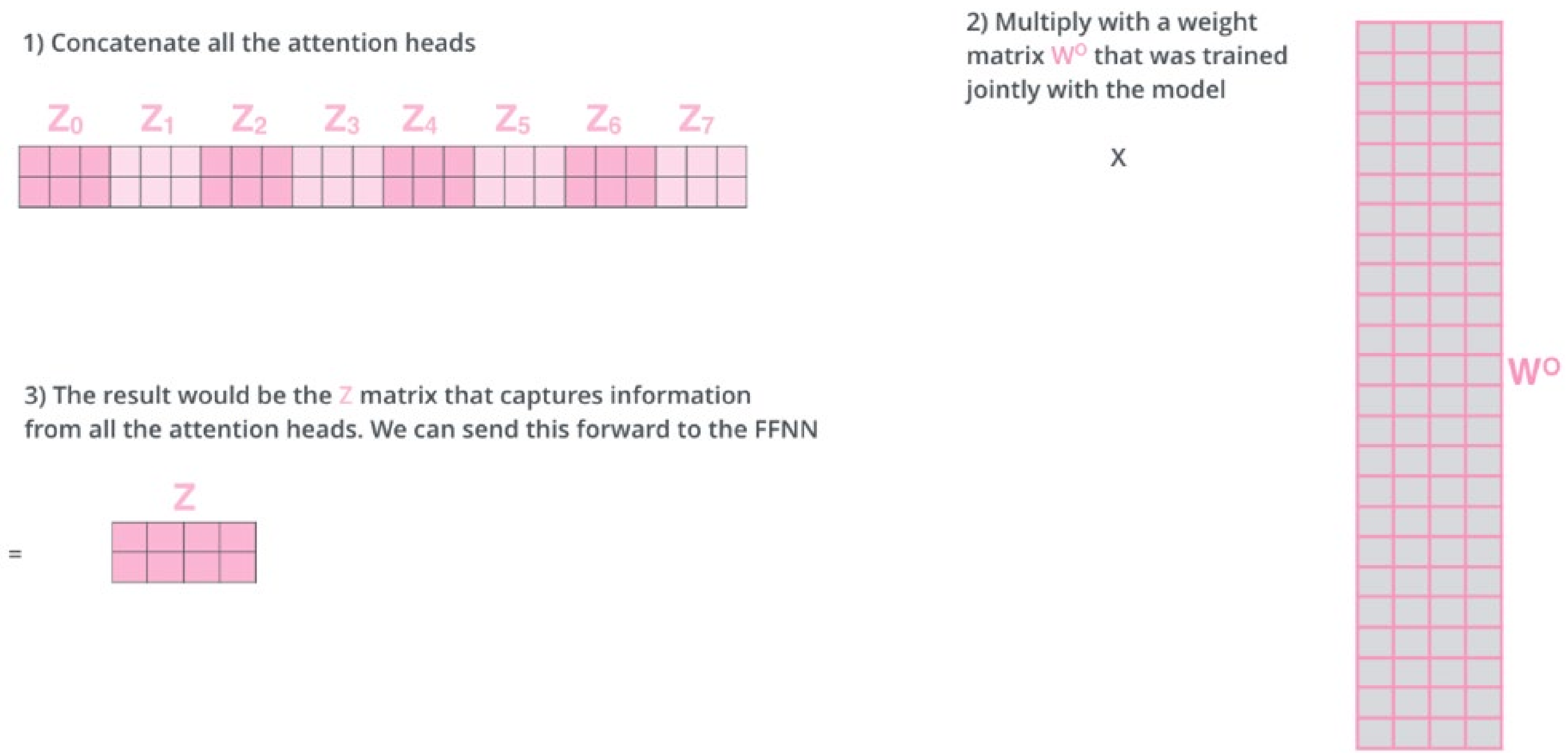
Multi-headed self-attention

- It gives the attention layer multiple "representation subspaces"
- Multiple sets of Query/Key/Value weight matrices (Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized.

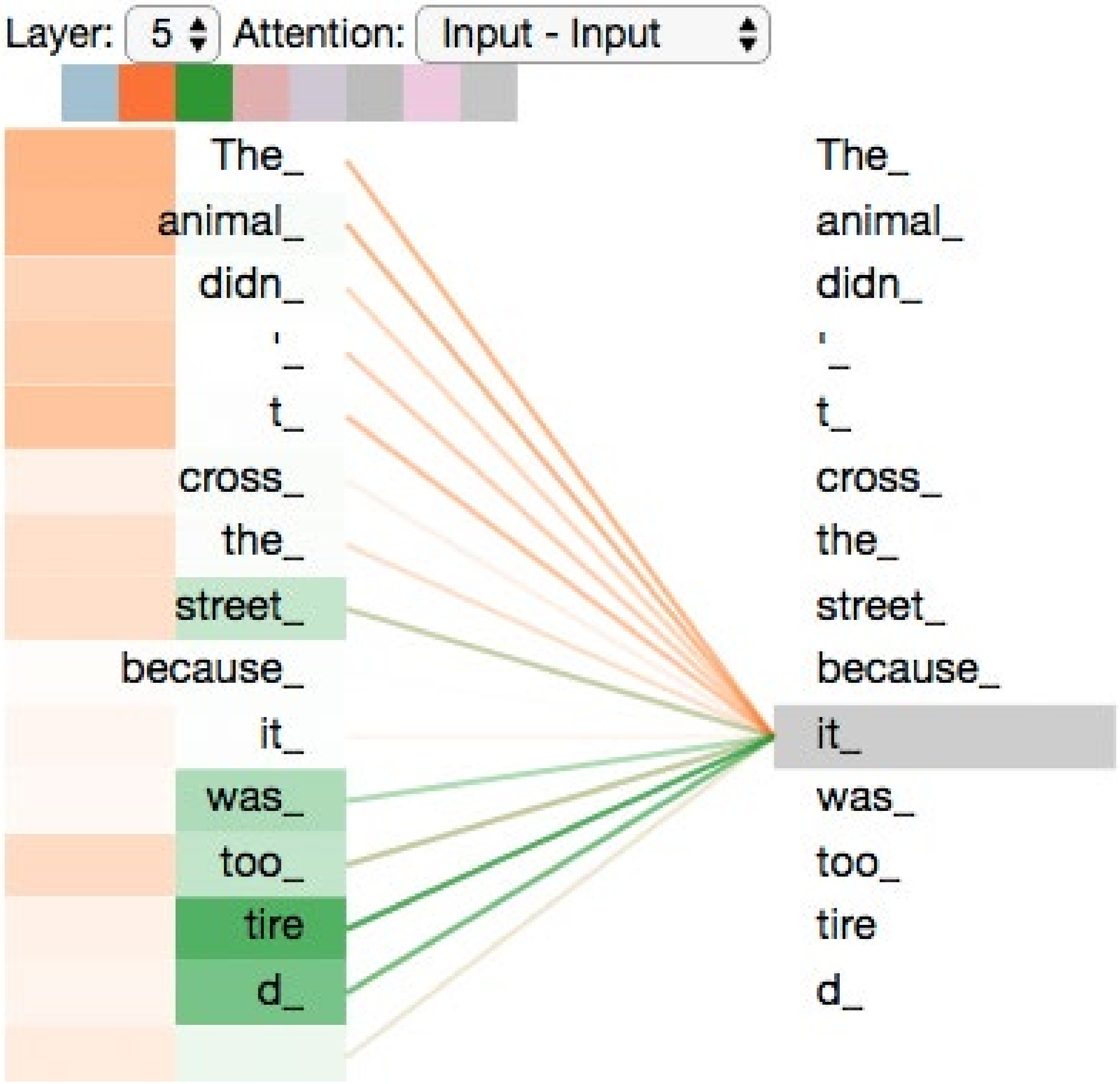




Condensing multi-head attentions into a single matrix

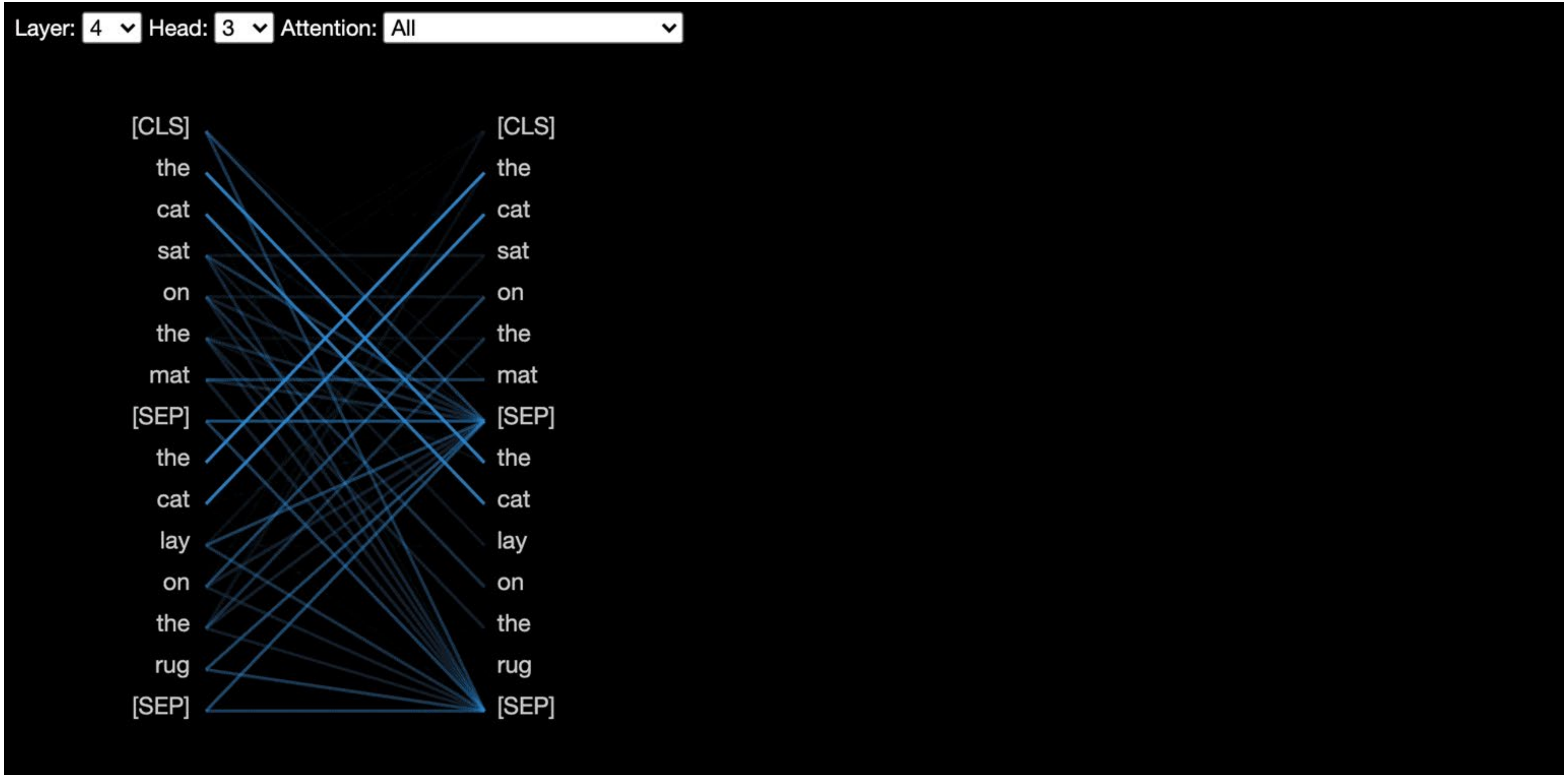


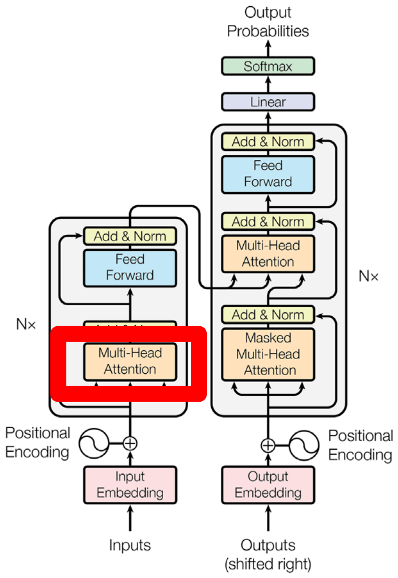
https://jalammar.github.io/illustrated-transformer/



https://github.com/jessevig/bertviz

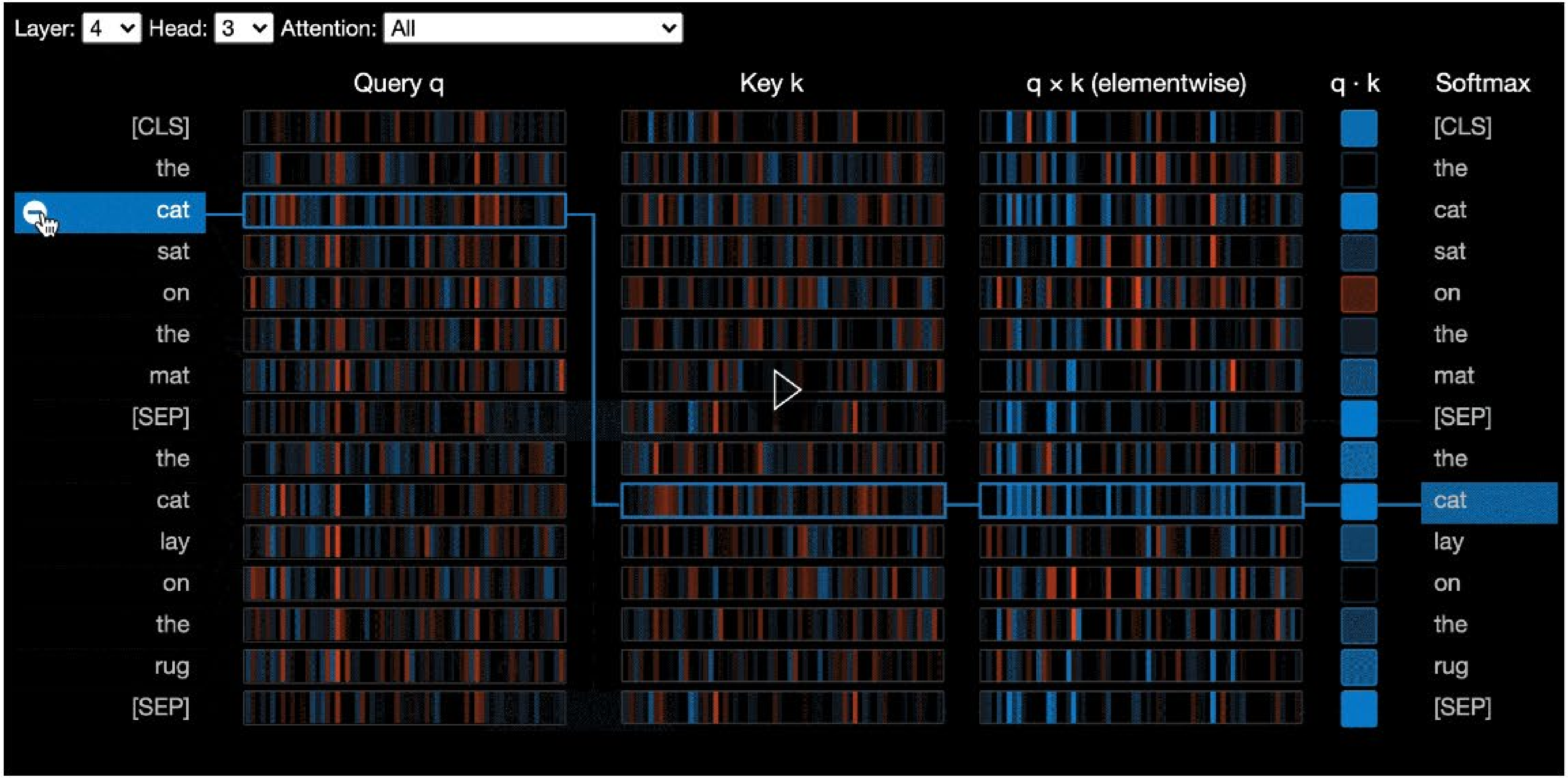
https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb

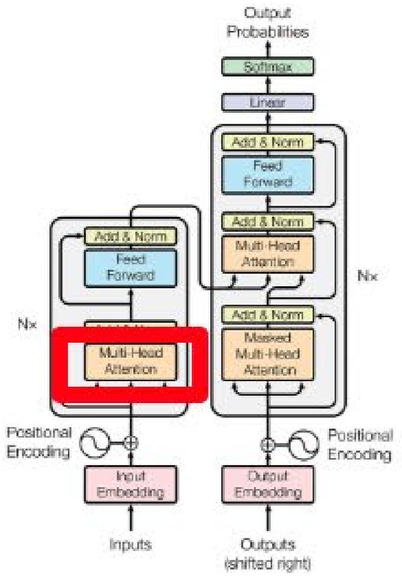




https://github.com/jessevig/bertviz

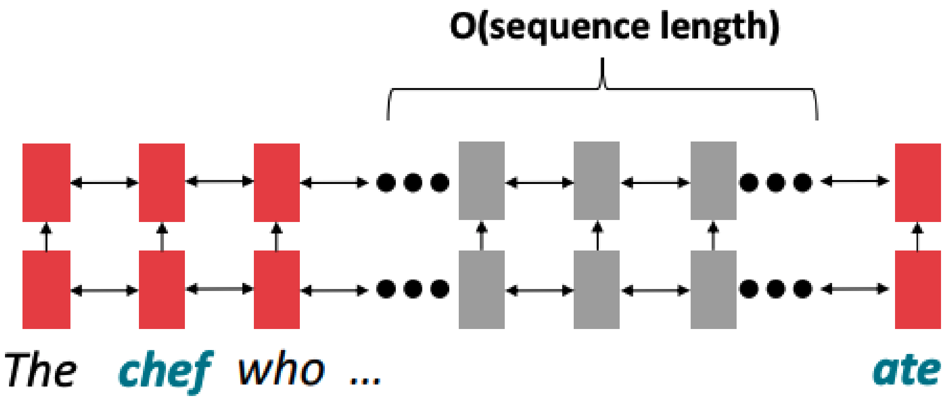
https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb



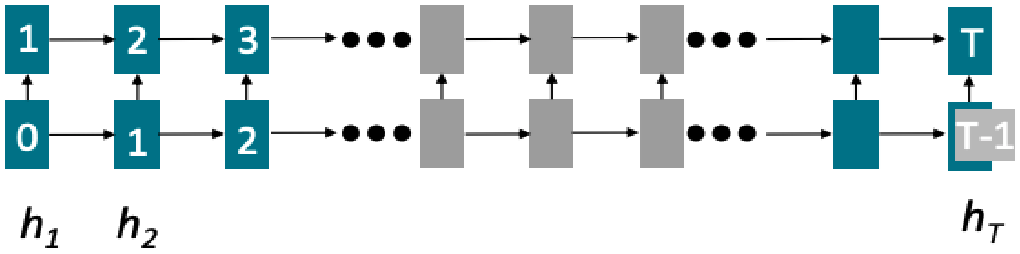


https://github.com/jessevig/bertviz

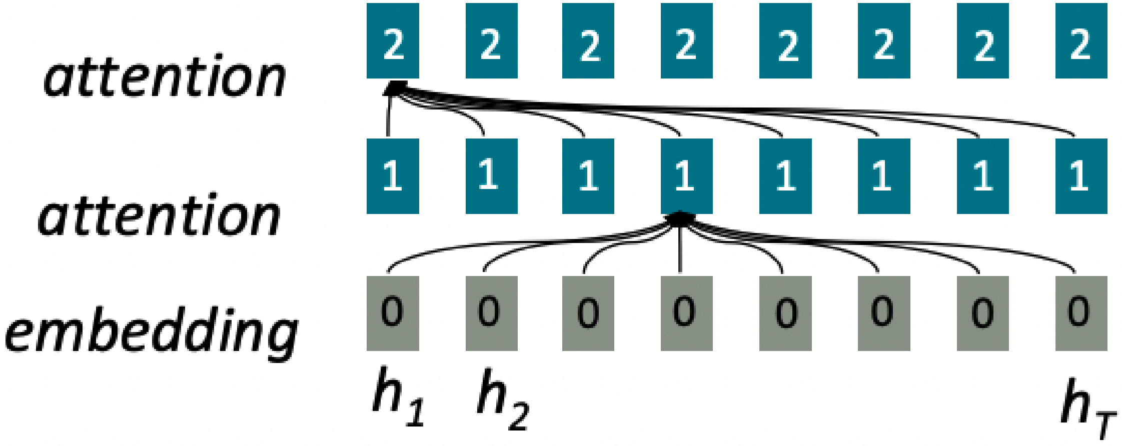
https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb



Linear interaction distance: O(T)

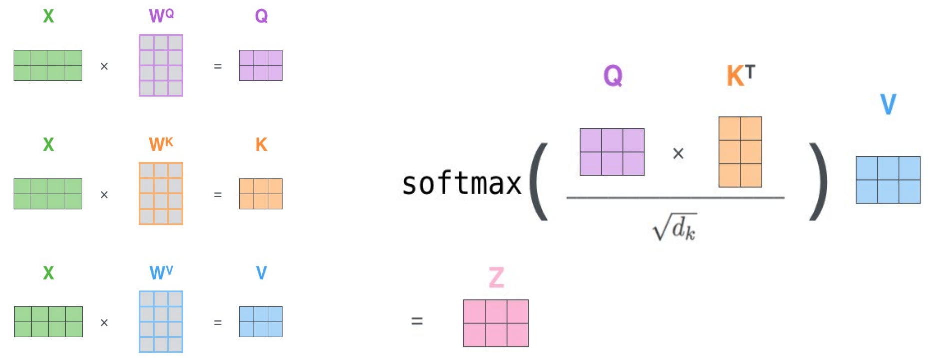


Lack of parallelization: O(T)

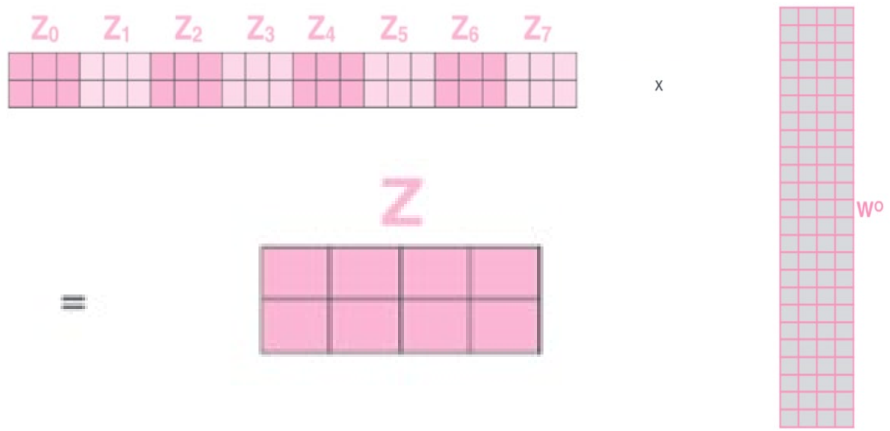


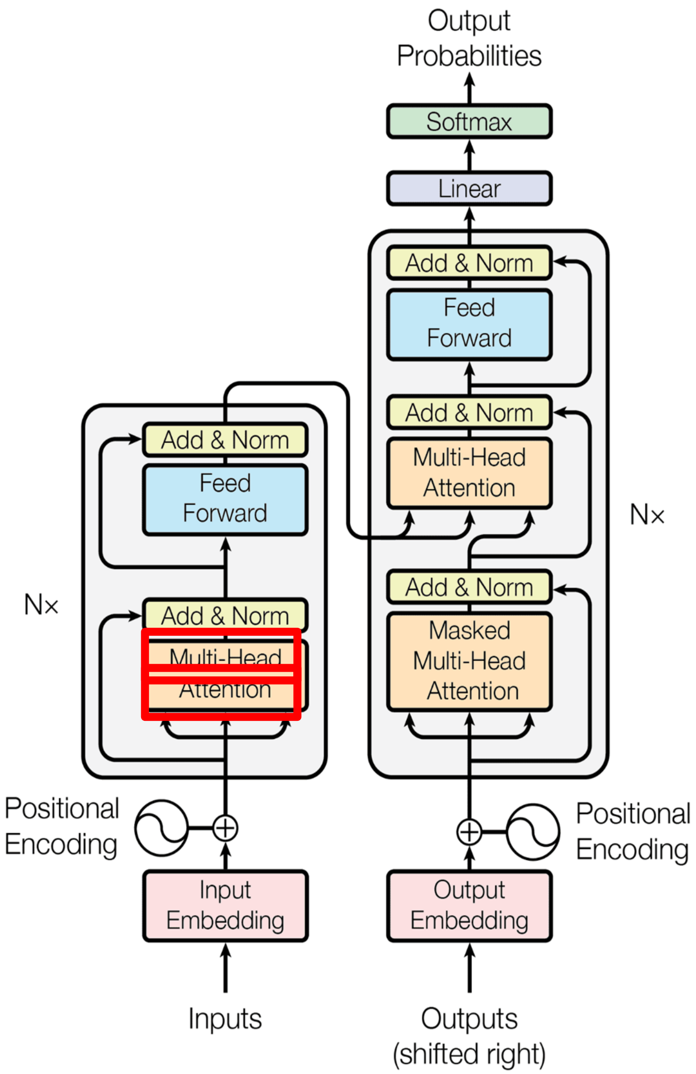
Self-attention: O(L)

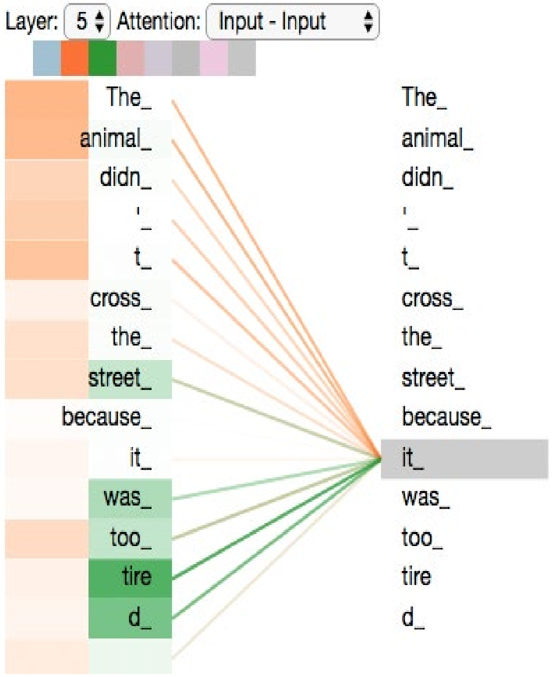
$$Q = XW^Q \quad K = XW^K \quad V = XW^V$$



$$Output = softmax(QK^T)V$$

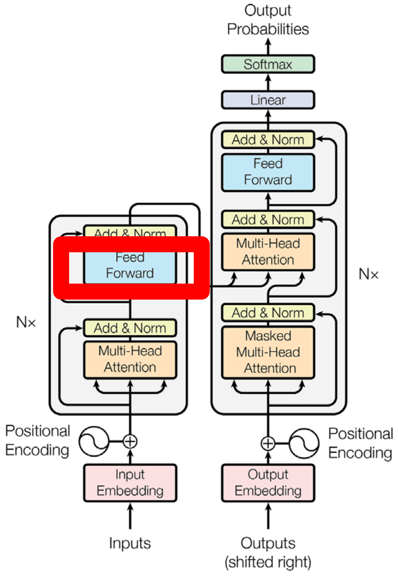




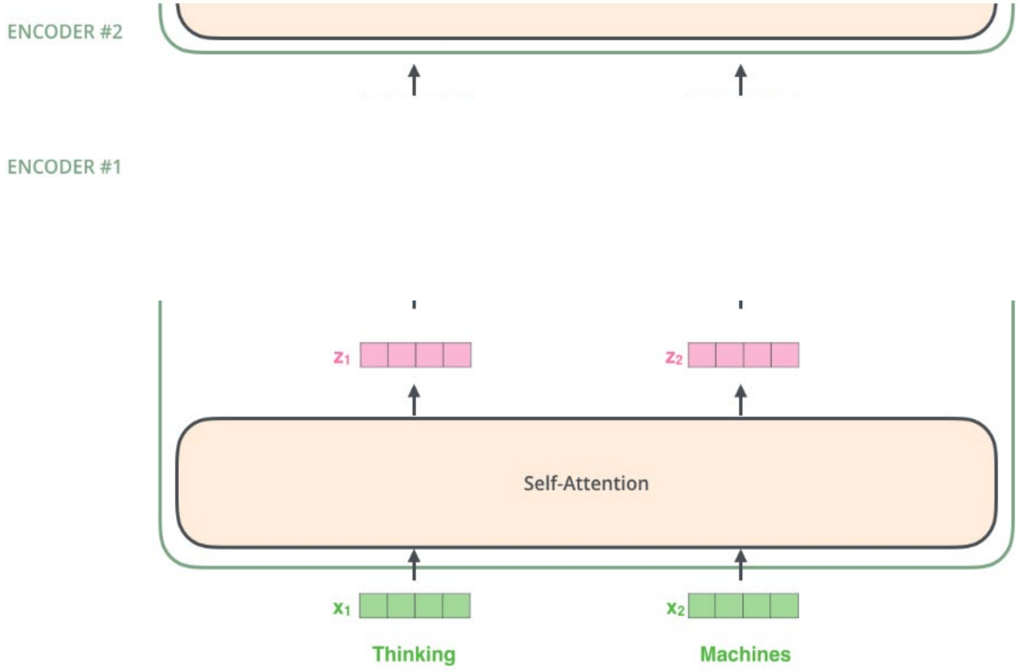


Other tricks than attention?

But attention isn't quite all you need!



- **Problem**: Since there are no element-wise non-linearities, self-attention is simply performing a re-averaging of the value vectors.
- **Easy fix**: Apply a feedforward layer to the output of attention, providing non-linear activation (and additional expressive power).

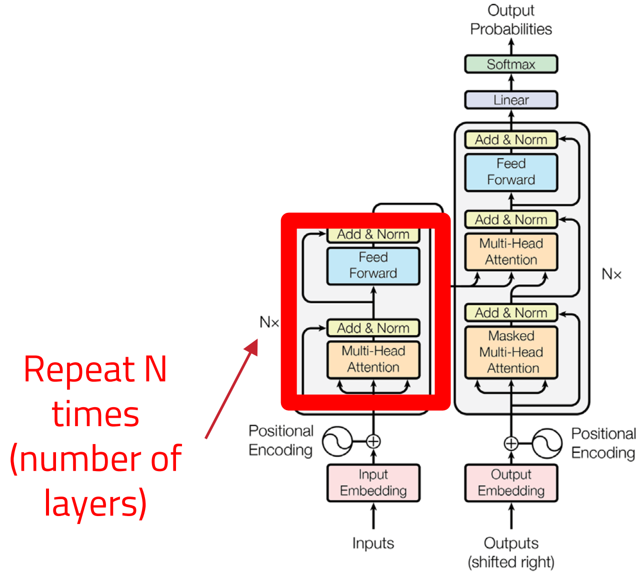


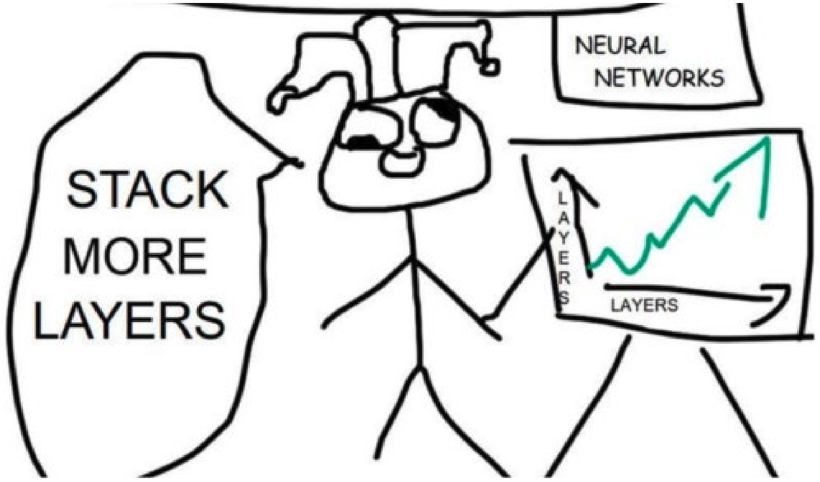
Equation for Feed-Forward layer

$$
\begin{aligned}
m_i &= MLP(\text{output}_i) \\
&= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2
\end{aligned}
$$

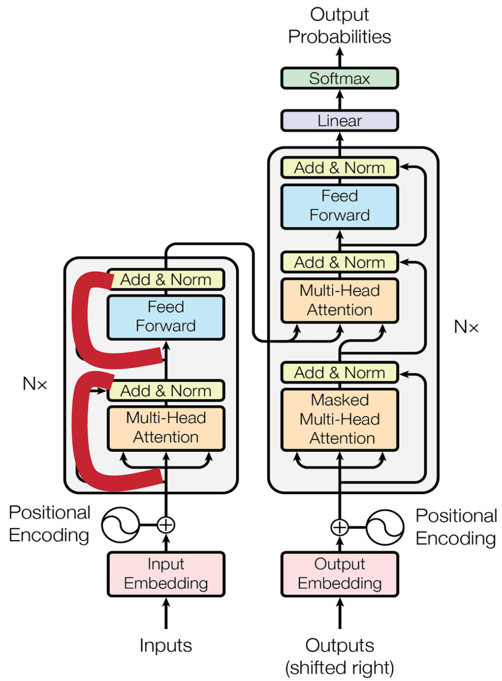
Stacking deep neural nets

- ❑ Training trick #1: Residual Connections
- ❑ Training trick #2: LayerNorm
- ❑ Training trick #3: Scaled Dot Product Attention





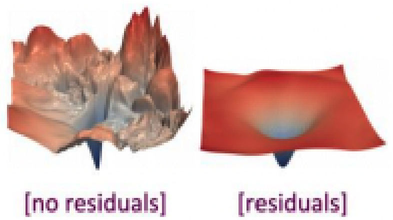
Trick #1: Residual Connections [He et al., 2016]



- ❑ Residual connections are a simple but powerful technique from computer vision.
- ❑ Directly passing "raw" embeddings to the next layer prevents the network from "forgetting" or distorting important information as it is processed by many layers.

$$x_\ell = F(x_{\ell-1}) + x_{\ell-1}$$

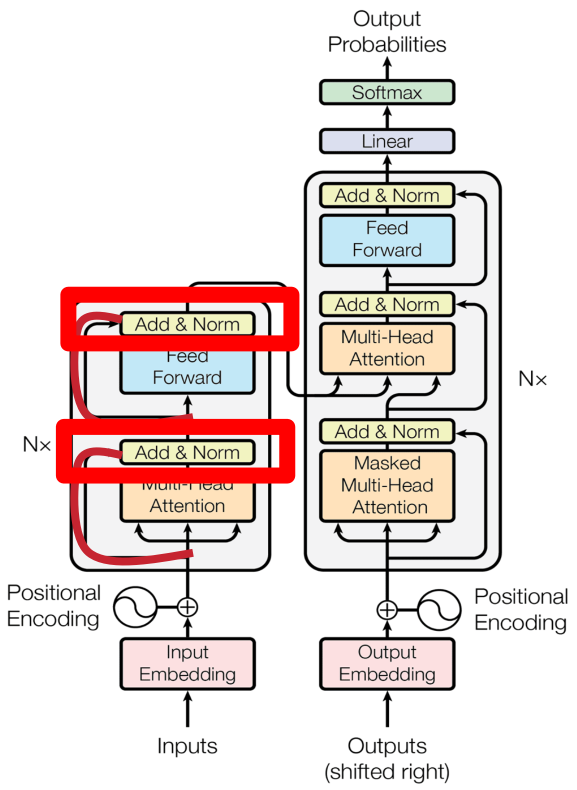
Residual connections are also thought to smooth the loss landscape and make training easier!



[Loss landscape visualization, Li et al., 2018, on a ResNet]

Trick #2: Layer Normalization [Ba et al., 2016]

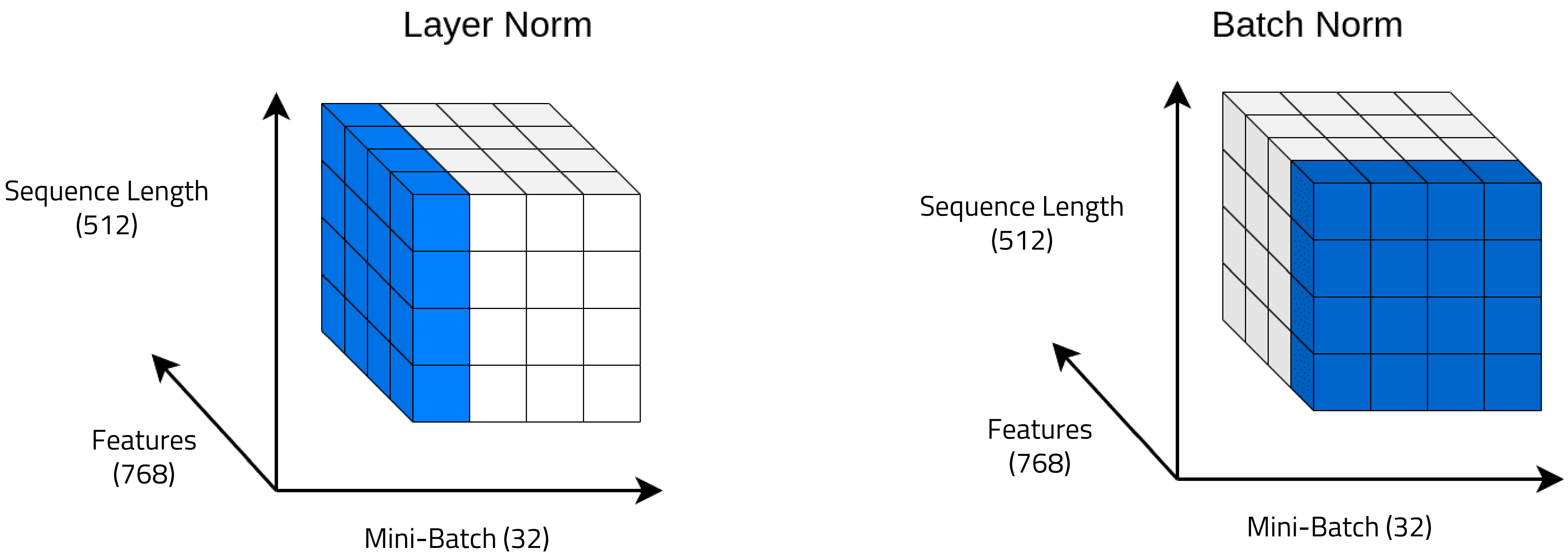
- **Problem**: Difficult to train the parameters of a given layer because its input from the layer beneath keeps shifting.
- **Solution**: Reduce uninformative variation by normalizing to zero mean and standard deviation of one within each layer.

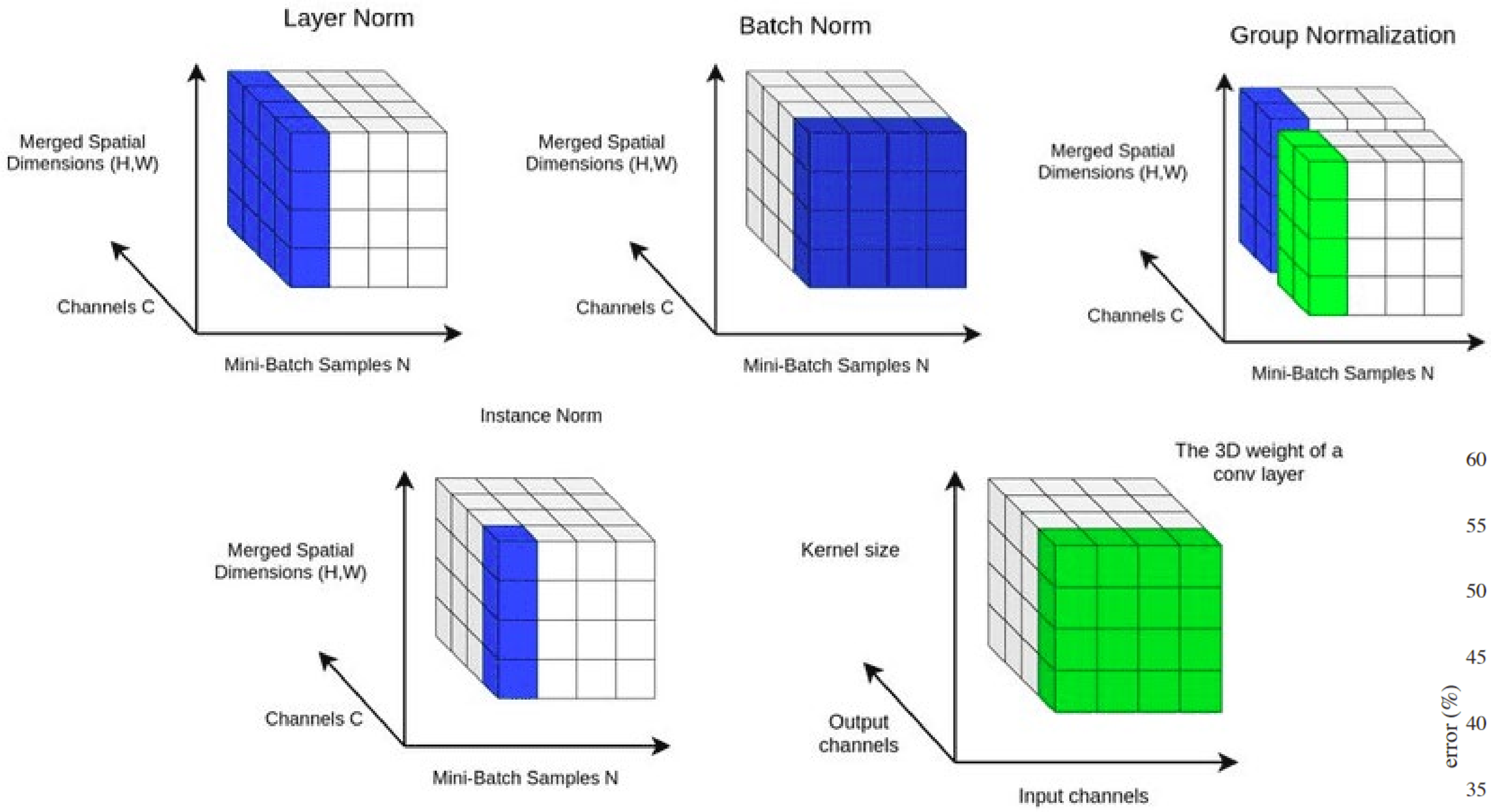


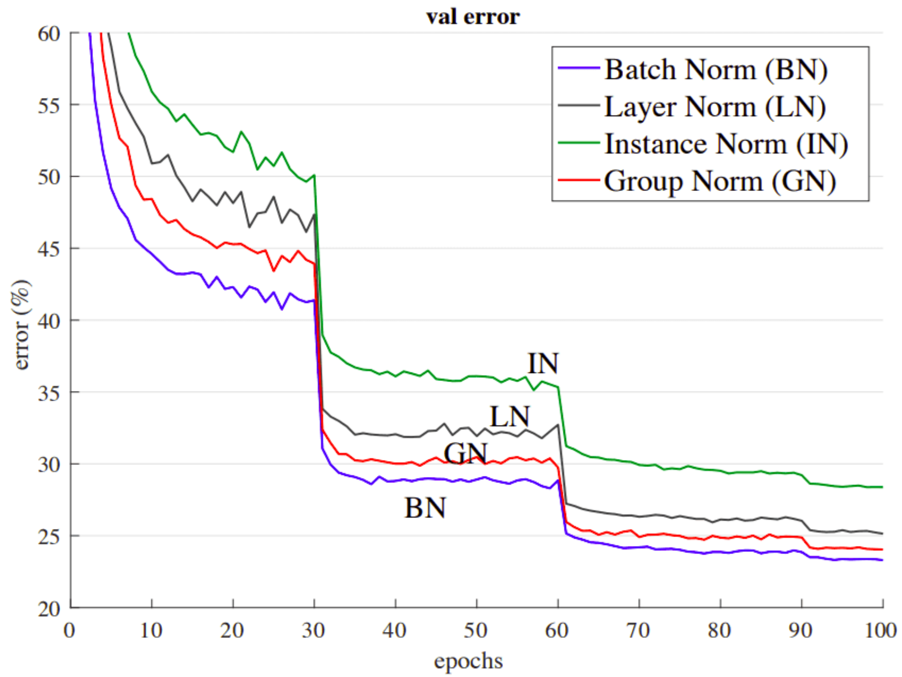
Mean: $\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$ **Standard Deviation:** $\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$

$$x^{\ell\prime} = \frac{x^\ell - \mu^\ell}{\sigma^\ell + \epsilon}$$

Layer norm vs Batch norm



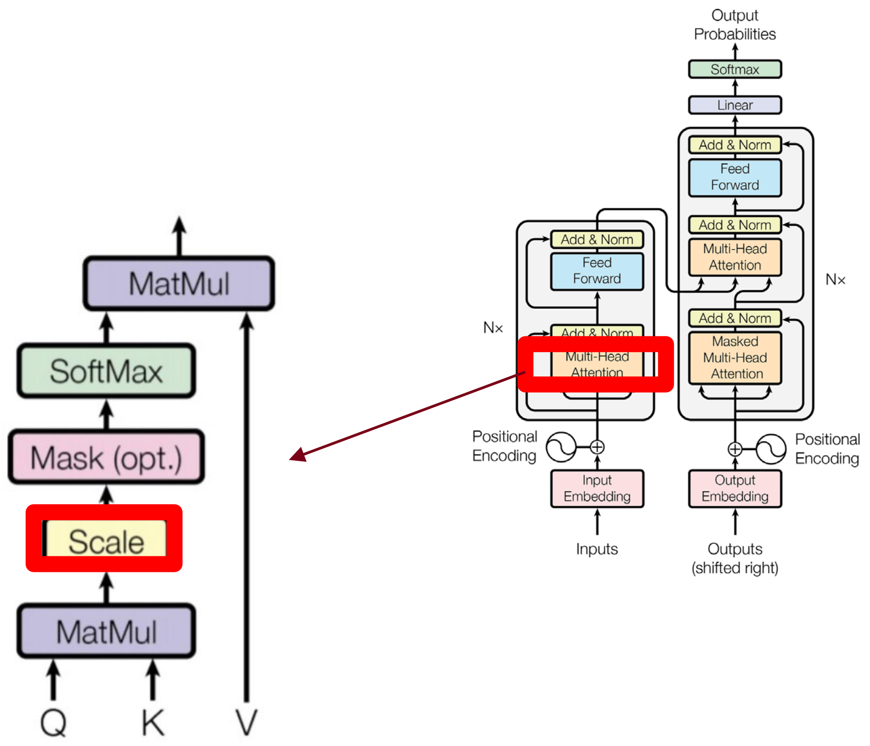




https://theaisummer.com/normalization

Trick #3: Scaled Dot Product Attention

- ❑ After LayerNorm, the mean and var of vecto elements is 0 and 1, respectively.
- ❑ But, the dot product still tends to take on extreme values, as its variance scales with dimensionality d_k



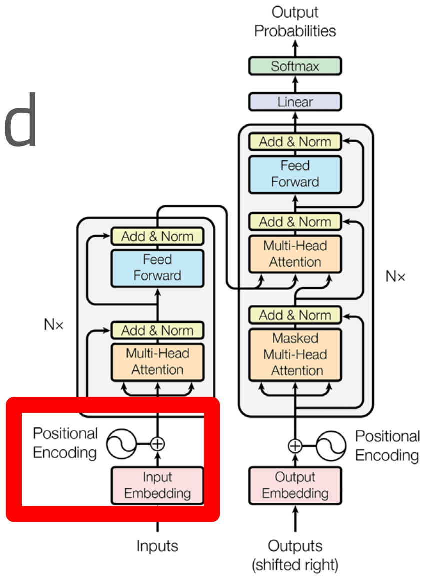
$$Output = softmax\left(QK^T\right)V$$

➡

Updated Self-Attention Equation

$$Output = softmax\left(QK^T / \boxed{\sqrt{d_k}}\right)V$$

Representing The Order of The Sequence Using Positional Encoding



- 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**

$p_i \in \mathbb{R}^d$, for $i \in \{1,2,\ldots,T\}$ are position vectors

- Easy to incorporate this info into our self-attention block: just add the pi to our inputs!

$$v_i = \tilde{v}_i + p_i$$
$$q_i = \tilde{q}_i + p_i$$
$$k_i = \tilde{k}_i + p_i$$

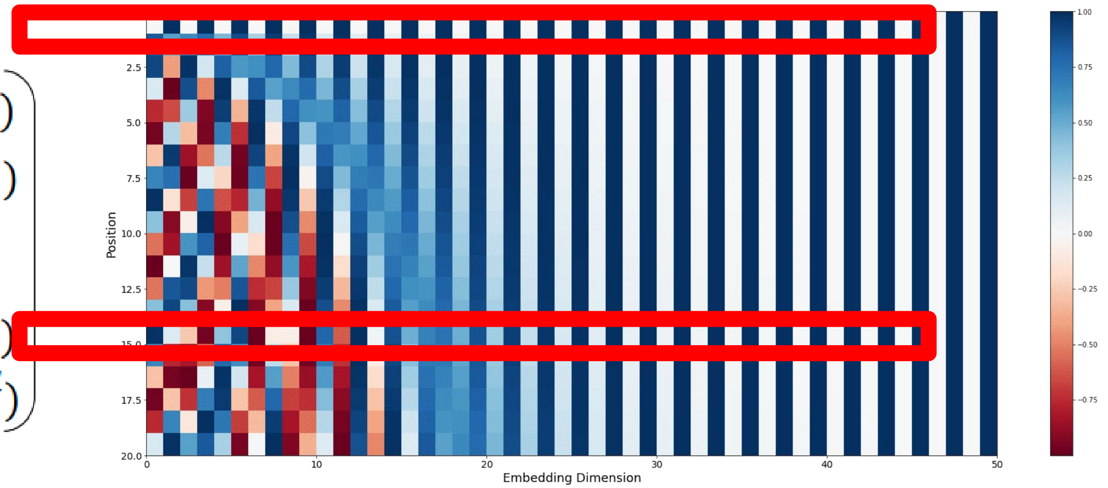
In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add...

Position representation vectors through sinusoids

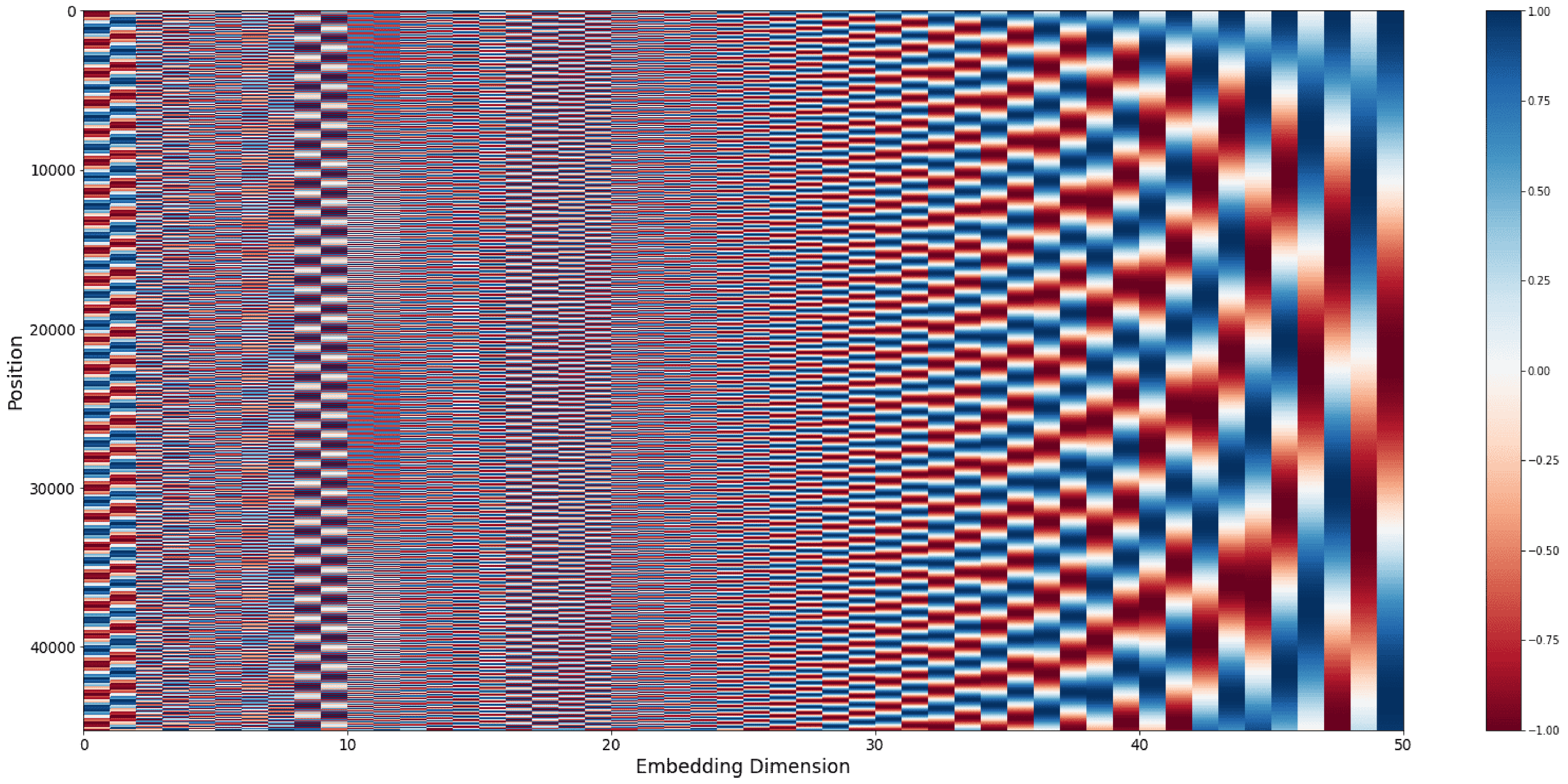
- Sinusoidal position representations: concatenate sinusoidal functions of varying periods:

$$\vec{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k.t), & \text{if } i = 2k \\ \cos(\omega_k.t), & \text{if } i = 2k+1 \end{cases}$$

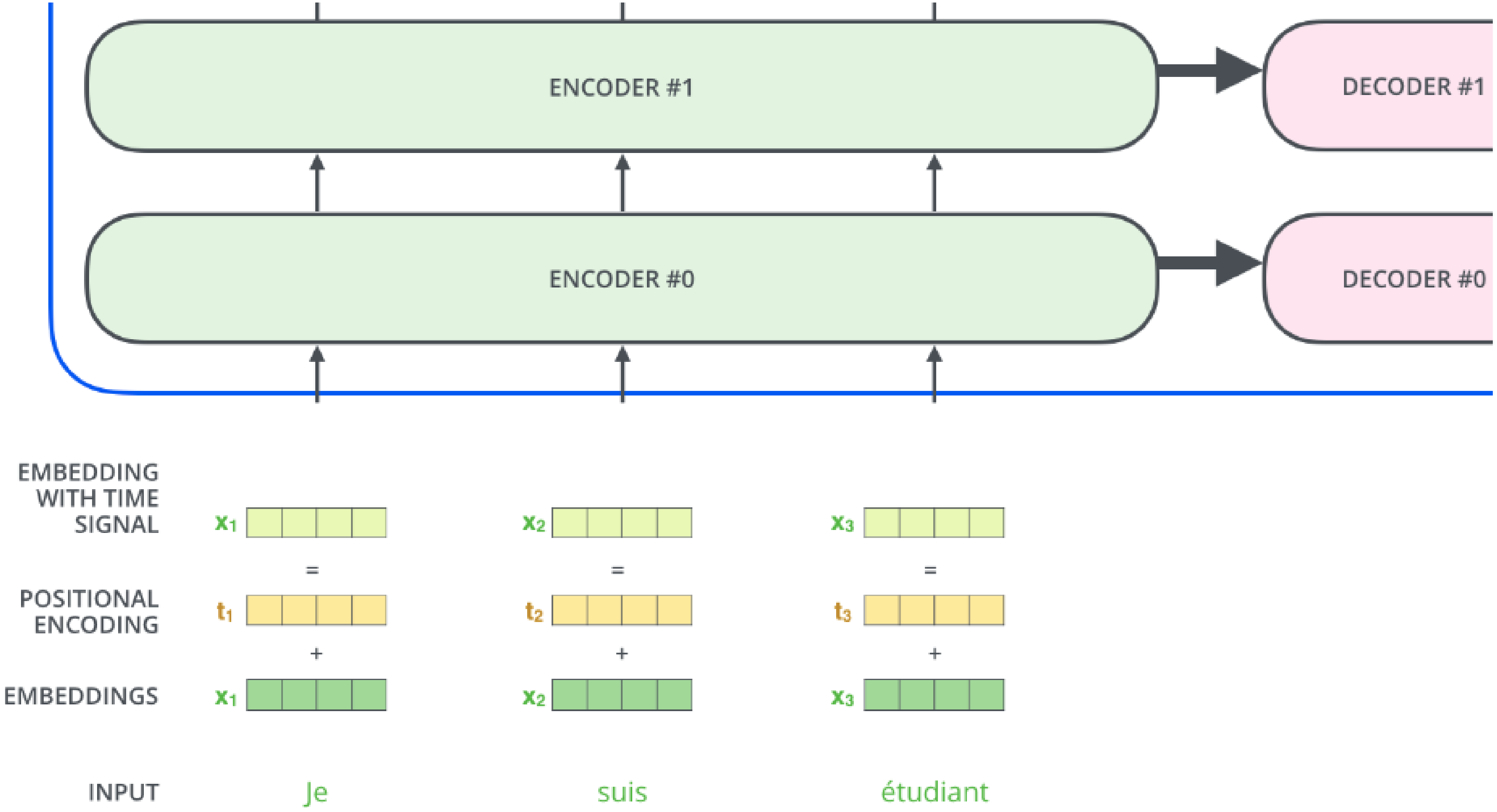
$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$

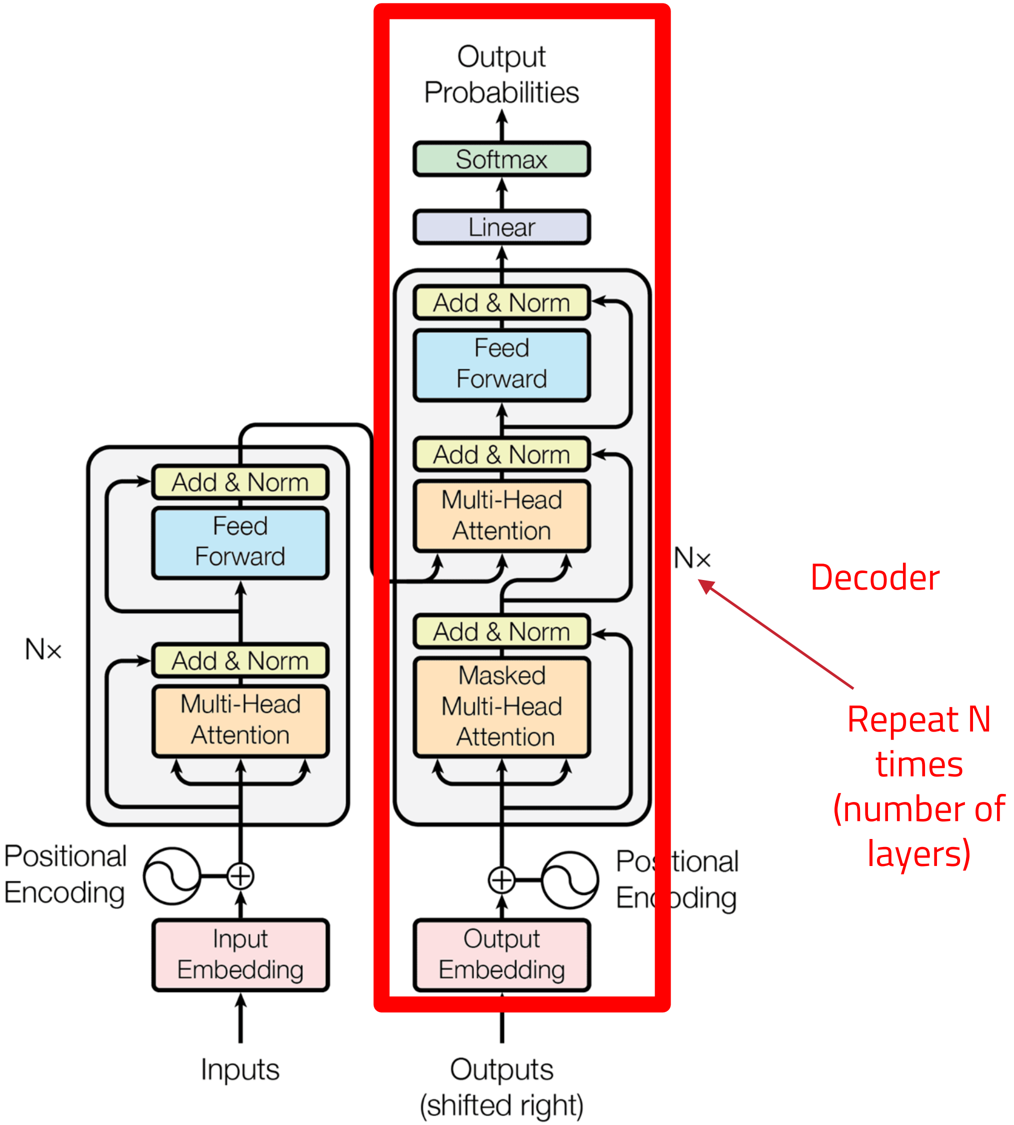


- Pros: Periodicity indicates that maybe “absolute position” isn’t as important
- Cons: Not learnable; also the extrapolation doesn’t really work

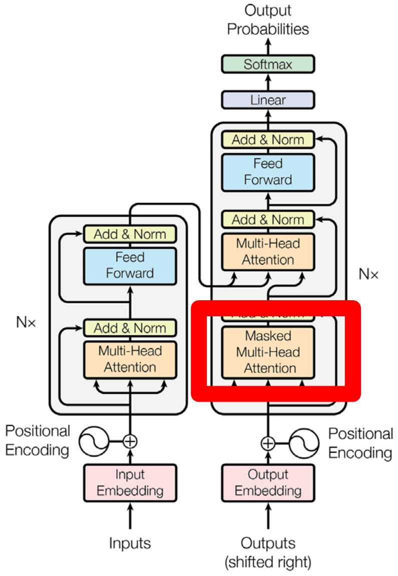


Position
0
10000
20000
30000
40000
Embedding Dimension
0
10
20
30
40
50
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


ENCODER #1
DECODER #1
ENCODER #0
DECODER #0
EMBEDDING WITH TIME SIGNAL
x1
x2
x3
=
POSITIONAL ENCODING
t1
t2
t3
+
EMBEDDINGS
x1
x2
x3
INPUT
Je
suis
étudiant


Output
Probabilities
Softmax
Linear
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Add & Norm
Masked
Multi-Head
Attention
Nx
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Nx
Positional
Encoding
Positional
Encoding
Input
Embedding
Output
Embedding
Inputs
Outputs
(shifted right)
Decoder
Repeat N
times
(number of
layers)

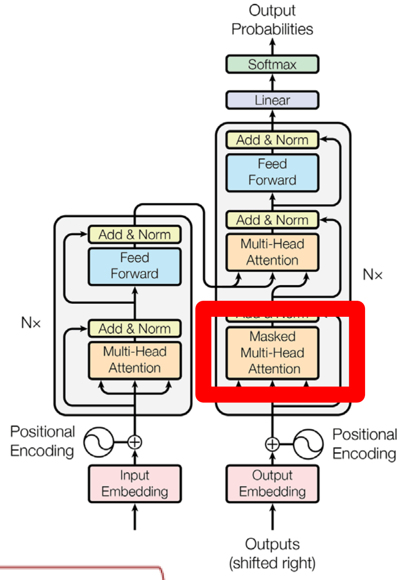
Decoder: Masked Multi-Head Self-Attention



- **Problem**: How do we prevent the decoder from "cheating"? If we have a language modeling objective, can't the network just look ahead and "see" the answer?

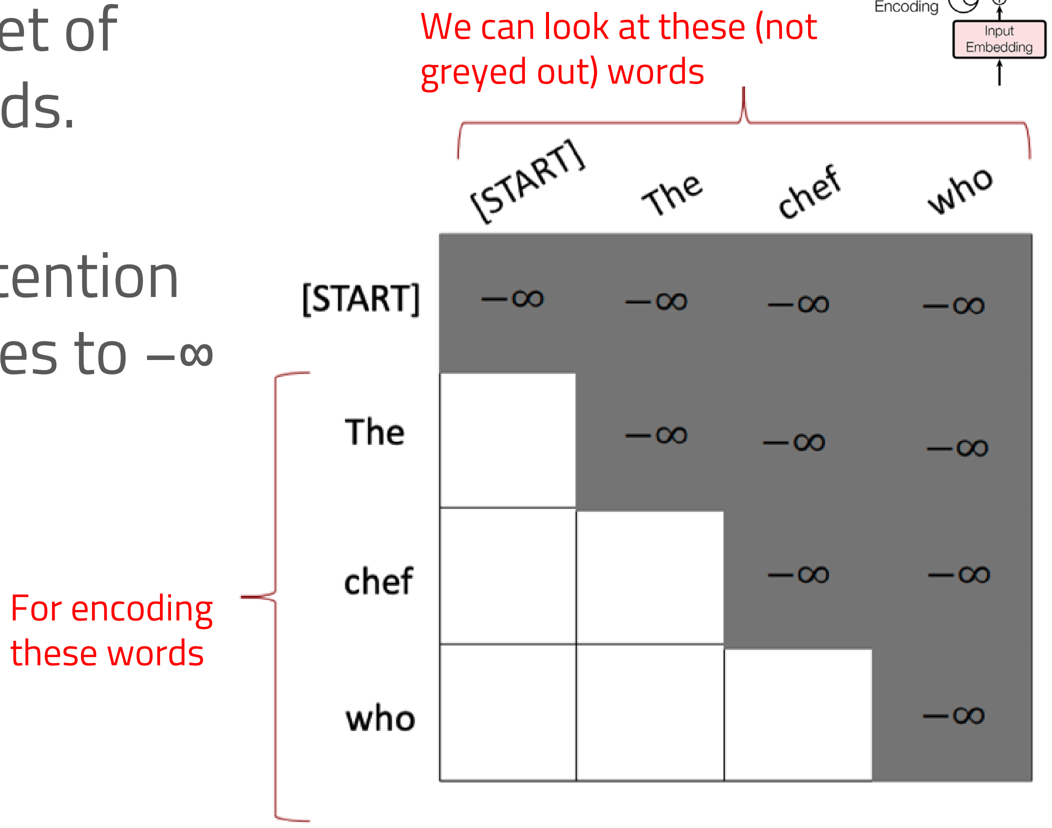
- **Solution**: Masked Multi-Head Attention. At a high-level, we hide (mask) information about future tokens from the model.

Masking the future in self-attention

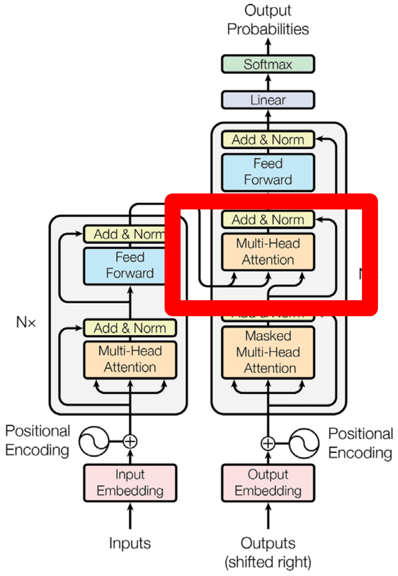


- To use self-attention in decoders, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of keys and queries to include only past words. (Inefficient!)
- To enable parallelization, we mask out attention to future words by setting attention scores to $-\infty$

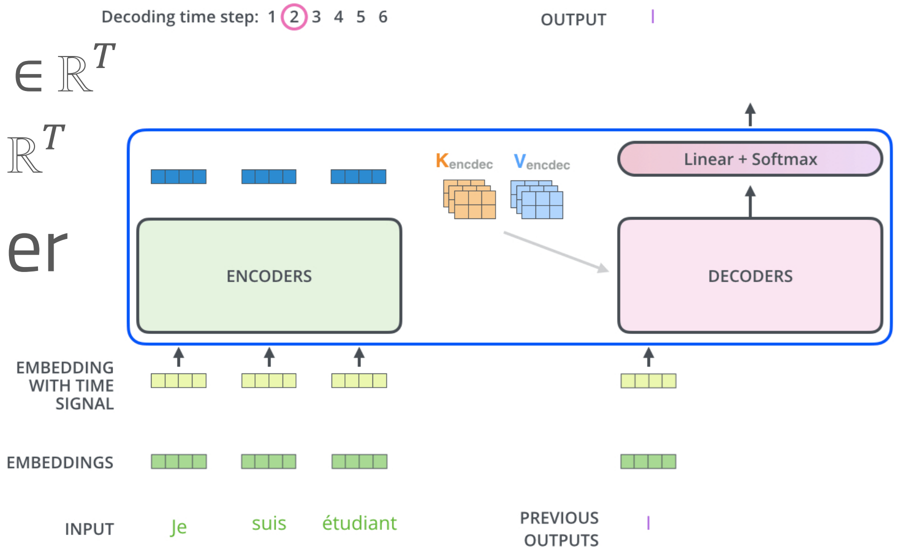
$$e_{ij} = \begin{cases} q_i^\top k_j, j < i \\ -\infty, j \geq i \end{cases}$$

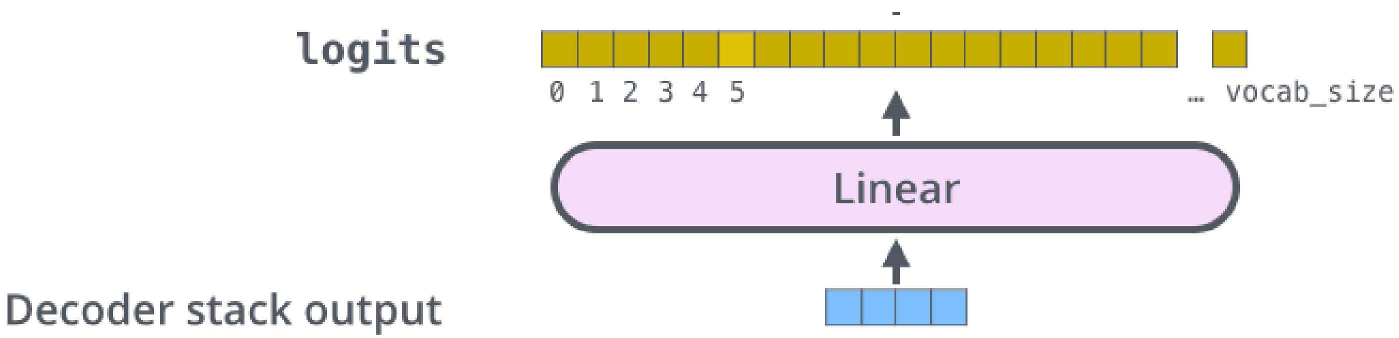


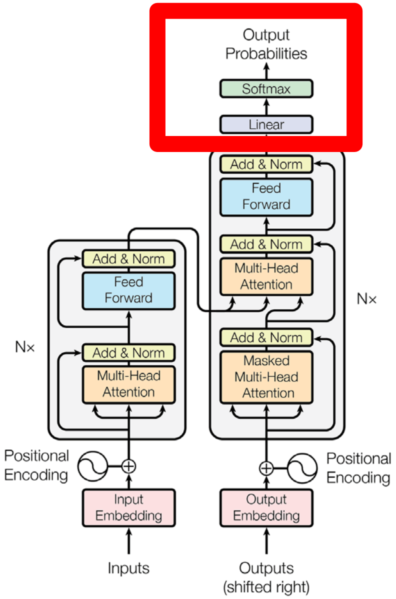
Encoder-Decoder Attention



- We saw that self-attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like seq2seq with attention.
  - Let $h_1 .. h_T$ be output vectors from the Transformer encoder; $x_i \in \mathbb{R}^T$
  - Let $z_1 .. z_T$ be input vectors from the Transformer decoder, $z_i \in \mathbb{R}^T$
- Then keys and values are drawn from the encoder (like a memory):
  - $k_i = K h_i$, $v_i = V h_i$.
- And the queries are drawn from the decoder,
  - $q_i = Q z_i$




logits
0 1 2 3 4 5
… vocab_size
Linear
Decoder stack output


Output
Probabilities
Softmax
Linear
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Add & Norm
Masked
Multi-Head
Attention
N×
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
N×
Positional
Encoding
Positional
Encoding
Input
Embedding
Output
Embedding
Inputs
Outputs
(shifted right)

Which word in our vocabulary
is associated with this index?

am

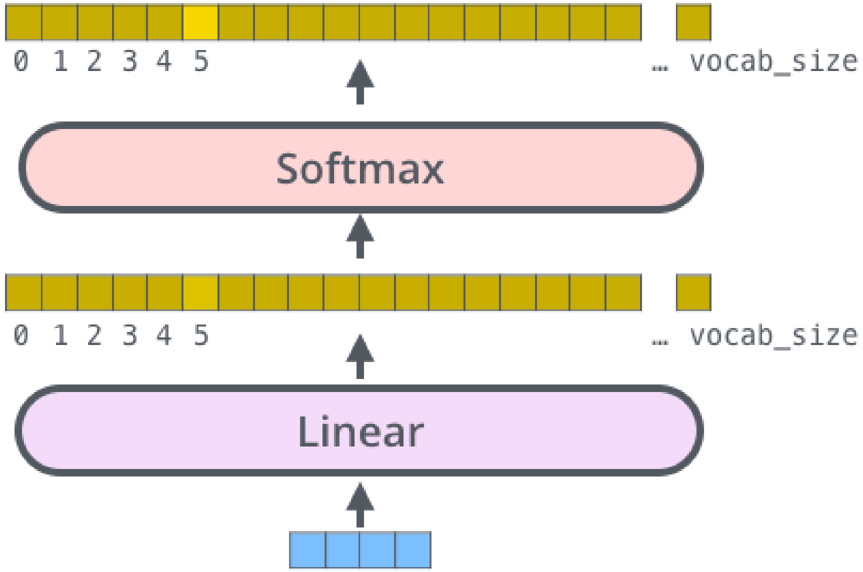
Get the index of the cell
with the highest value
`(argmax)`

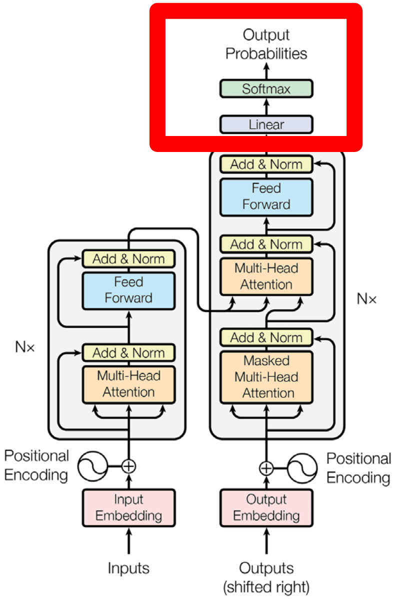
5

`log_probs`

`logits`

Decoder stack output





Drawback of Transformer

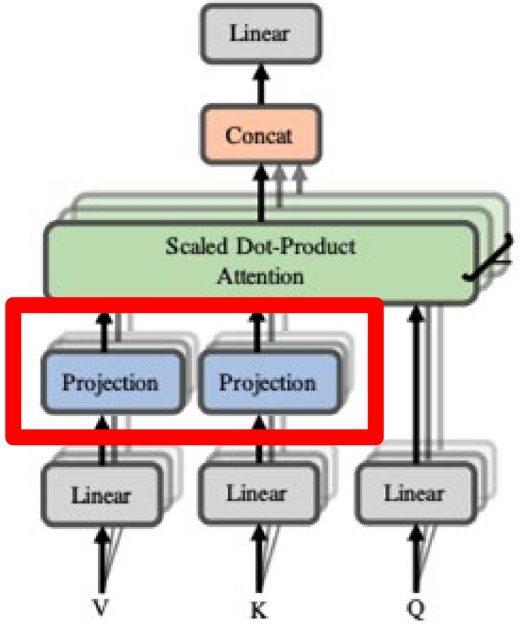
Drawback of Transformer

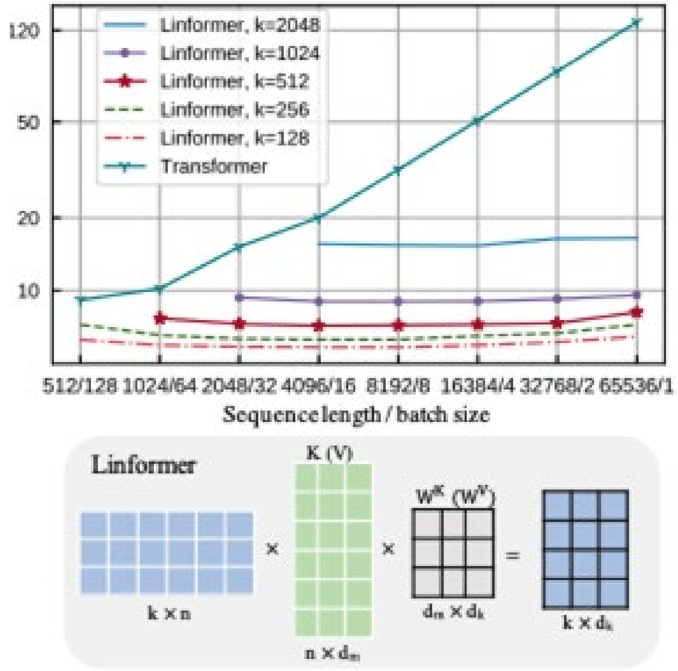
- **Quadratic compute in self-attention:**
  - Computing all pairs of interactions means our computation grows quadratically with the sequence length! For recurrent models, it only grew linearly!
  - Reduce $O(T^2)$ all-pairs self-attention cost?

- **Position representations:**
  - Are simple absolute indices the best we can do to represent position?
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]

Reduce $O(T^2)$ all-pairs self-attention cost?

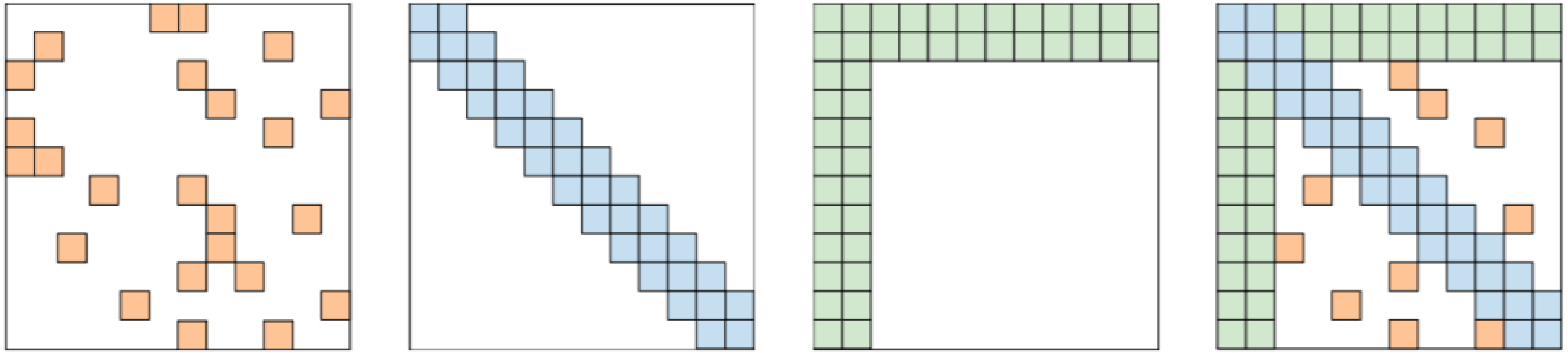
- LinFormer (Wang et al., 2020); O(T^2) -> O(T)
  - Map the sequence length dimension to a lower-dimensional space for values, keys





Reduce $O(T^2)$ all-pairs self-attention cost?

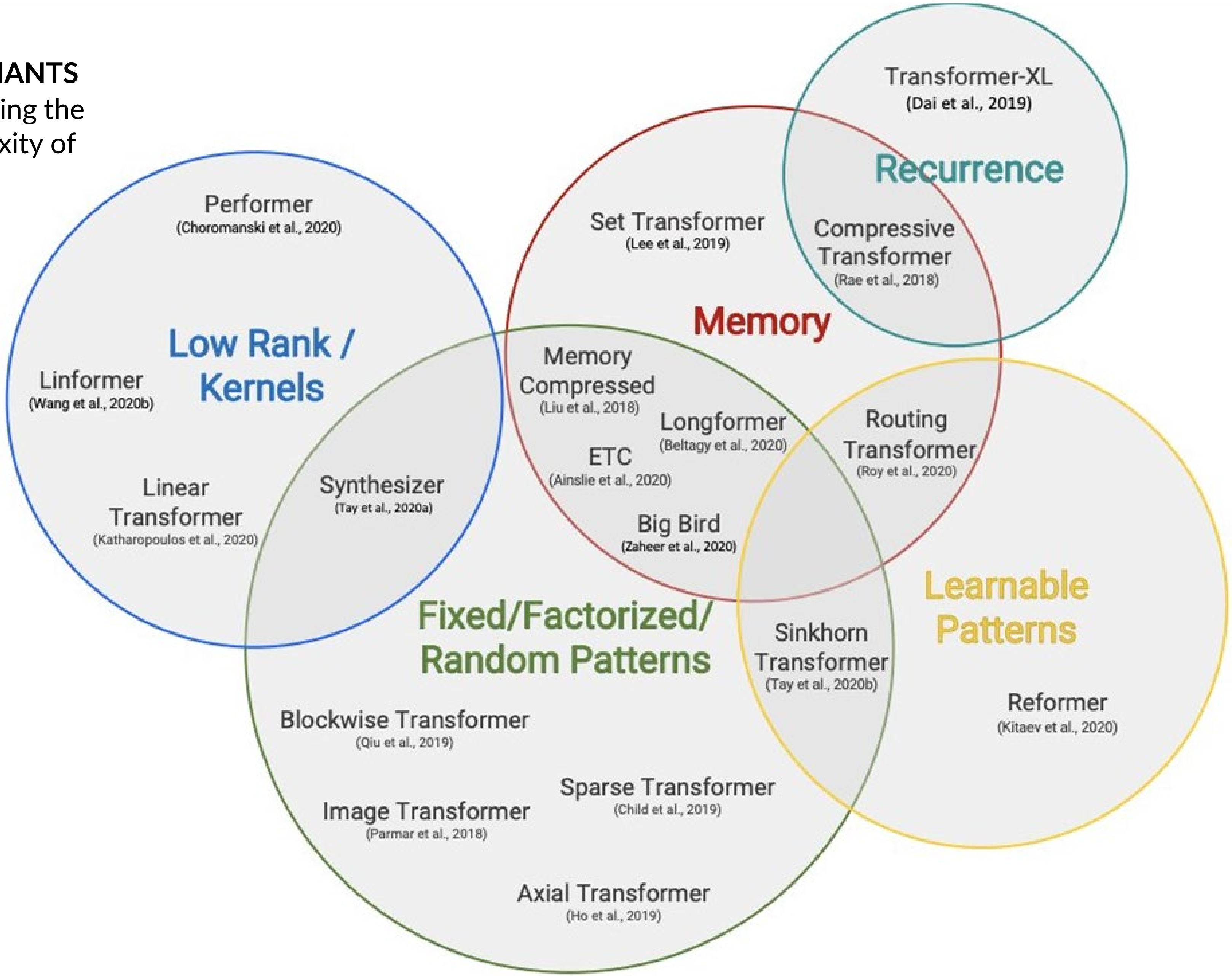
- BigBird (Zaheer et al., 2021)
  - Replace all-pairs interactions with a family of other interactions, like local windows, looking at everything, and random interactions.



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

TRANSFORMER VARIANTS

Lots of focus on reducing the computational complexity of transformer models.



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†

Scaling up Transformer

Model	Layers	Width	Heads	Params	Data	Training
Transformer-Base	12	512	8	65M		8x P100 (12 hrs)
Transformer-Large	12	1024	16	213M		8x P100 (3.5 days)

http://hal.cse.msu.edu/teaching/2020-fall-deep-learning/14-nlp-and-transformers/#/22/0/9

Scaling up Transformer

Model	Layers	Width	Heads	Params	Data	Training
Transformer-Base	12	512	8	65M		8x P100 (12 hrs)
Transformer-Large	12	1024	16	213M		8x P100 (3.5 days)
BERT-Base	12	768	12	110M	13GB	
BERT-Large	24	1024	16	340M	13GB	

http://hal.cse.msu.edu/teaching/2020-fall-deep-learning/14-nlp-and-transformers/#/22/0/9

Scaling up Transformer

Model	Layers	Width	Heads	Params	Data	Training
Transformer-Base	12	512	8	65M		8x P100 (12 hrs)
Transformer-Large	12	1024	16	213M		8x P100 (3.5 days)
BERT-Base	12	768	12	110M	13GB	
BERT-Large	24	1024	16	340M	13GB	
XLNet-Large	24	1024	16	340M	126GB	512x TPU-v3 (2.5 days)
RoBERTa	24	1024	16	355M	160GB	1024x V100 (1 day)
GPT-2	48	1600	?	1.5B	40GB	
Megatron-LM	72	3072	32	8.3B	174GB	512x V100 (9 days)
Turing-NLG	78	4256	28	17B	?	256x V100
GPT-3	96	12288	96	175B	694GB	?

Brown et al, "Language Models are Few-Shot Learners", arXiv 2020

http://hal.cse.msu.edu/teaching/2020-fall-deep-learning/14-nlp-and-transformers/#/22/0/9

Summary

- Transformers are a new neural network model that only uses attention (and many other training tricks!!)
- However, the models are extremely expensive
- Improvements (unfortunately) seem to mostly come from even more expensive models and more data
- If you can afford large data and large compute, transformers are the go to architecture, instead of CNNs, RNNs, etc.
    - Why? On our way back to fully-connected models, throwing out the inductive bias of CNNs and RNNs.