Transformers, I

LING 574 Deep Learning for NLP
Shane Steinert-Threlkeld

Announcements

- HW3 ref available

- Cross entropy loss:

- BCE in HW3 similar

$$\frac{1}{\text{batch-size}} \sum_{\text{row}_i} \text{c-ent}(\text{labels}[\text{row}_i], \text{probabilities}[\text{row}_i])$$

$$= \frac{1}{\text{batch-size}} \sum_{\text{row}_i} -\sum_{j} \text{labels}[\text{row}_i]_j \times \log \text{probabilities}[\text{row}_i]_j$$

$$= -\frac{1}{\text{batch-size}} \sum_{\text{row}_i} \sum_{j} \text{labels}[\text{row}_i]_j \times \log \text{probabilities}[\text{row}_i]_j$$

Today's Plan

- Attention

- Limitations of Recurrent Models

- Transformers: building blocks

  - Self-attention

  - Encoder architecture

Limitations of Recurrent Models

RNNs Unrolling

- Recall: RNNs are "unrolled" across time, same operation at each step

- This has at least two issues:

  - Creates "long path lengths" between sequence positions
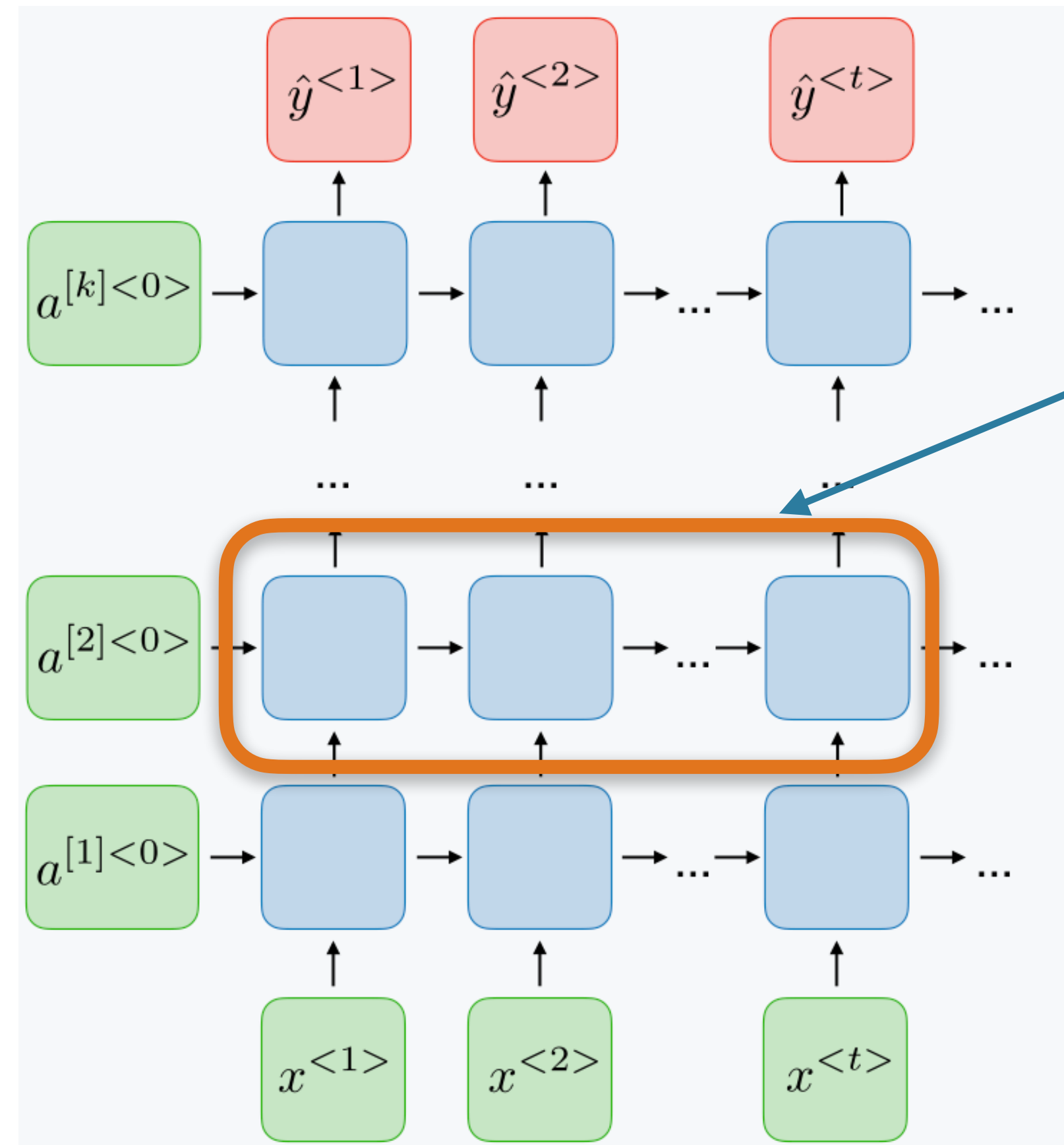
  - Not parallelizable

Long Path Lengths

- Gating mechanisms help RNNs learn long distance dependencies, by alleviating the vanishing gradient problem

- But: still takes a linear number of computations for one token to influence another

  - Long-distance dependencies are still hard!
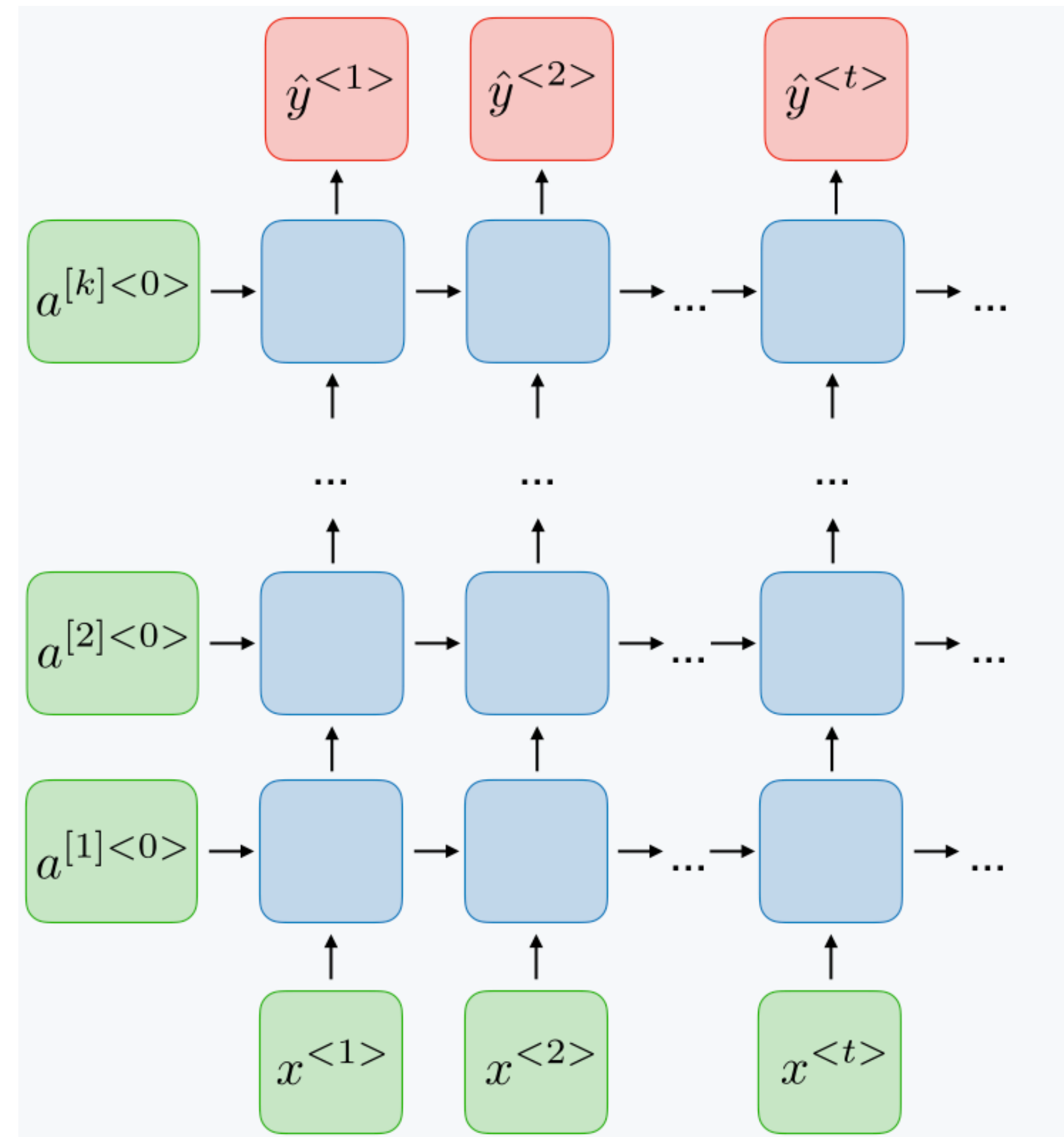
Students who … enjoy

Long Path Lengths

- Gating mechanisms help RNNs learn long distance dependencies, by alleviating the vanishing gradient problem

- But: still takes a linear number of computations for one token to influence another

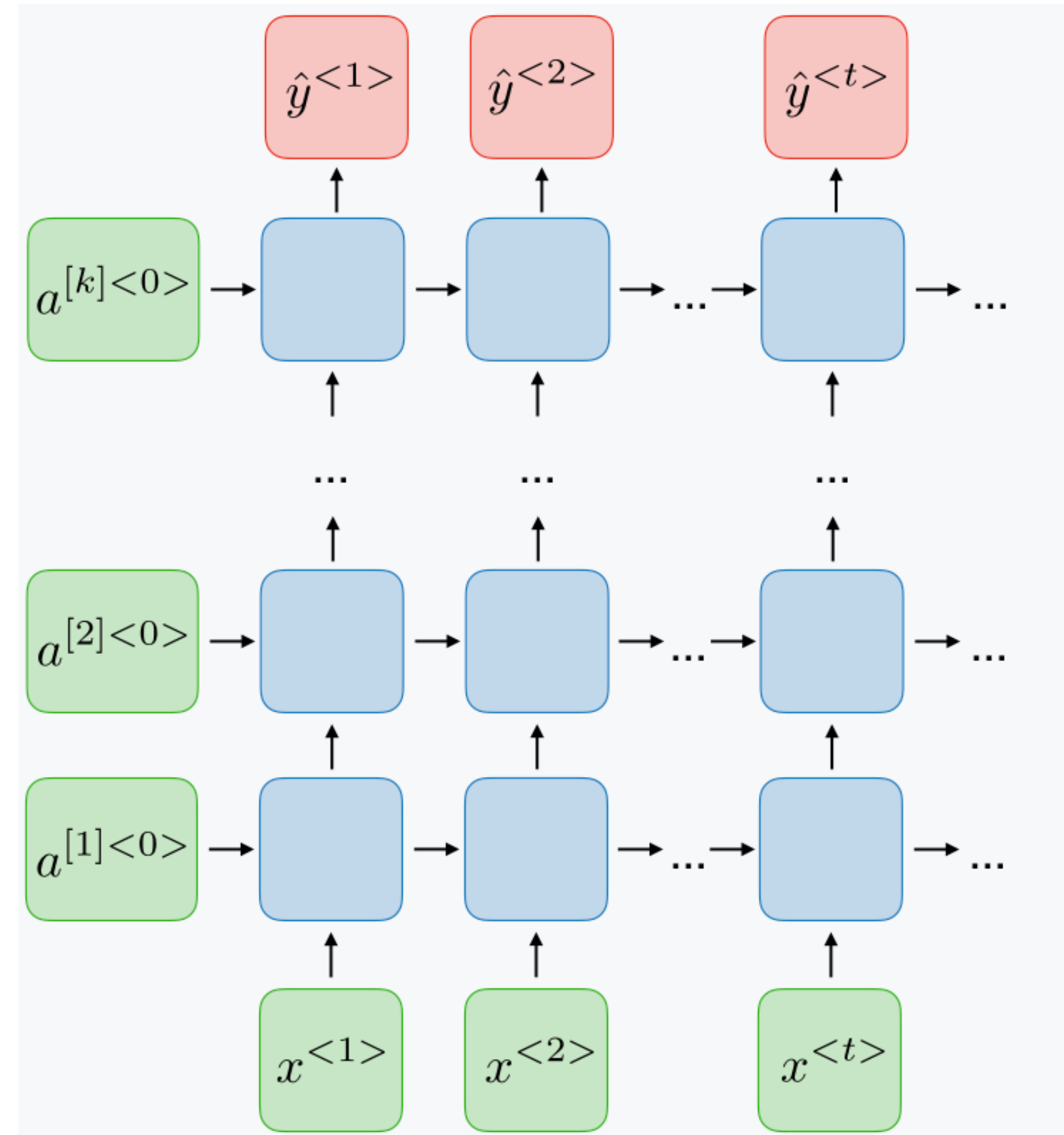 - Long-distance dependencies are still hard!

Linear "path length" for interaction between tokens

Students who … enjoy
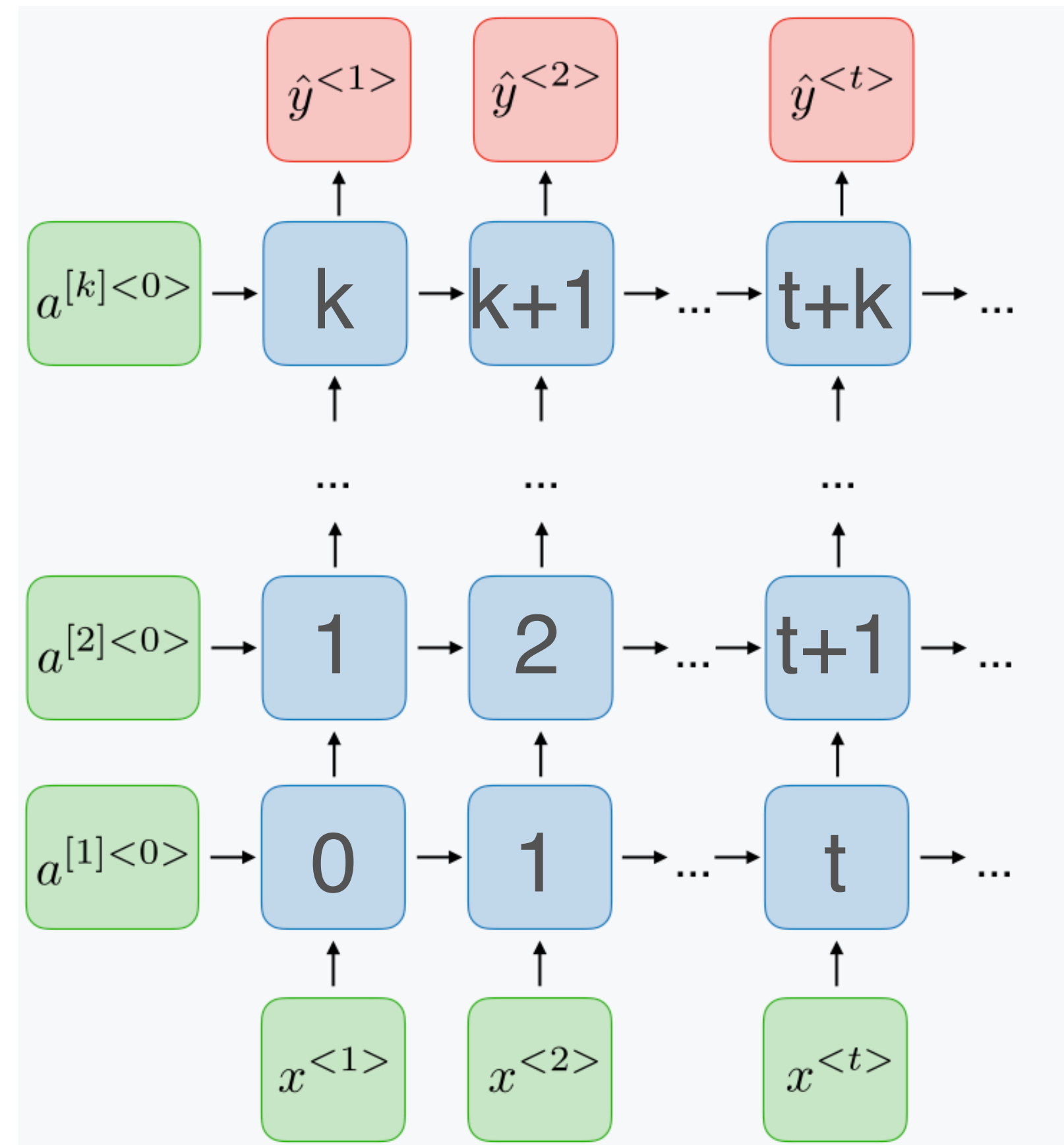
Lack of Parallelizability

- Modern hardware (e.g. GPUs) are very good at doing *independent* computations in parallel

- RNNs are inherently serial:

  - Cannot compute future time steps without the past

- Bottleneck that makes scaling up difficult

Students who … enjoy

Lack of Parallelizability

- Modern hardware (e.g. GPUs) are very good at doing *independent* computations in parallel

- RNNs are inherently serial:

  - Cannot compute future time steps without the past

- Bottleneck that makes scaling up difficult

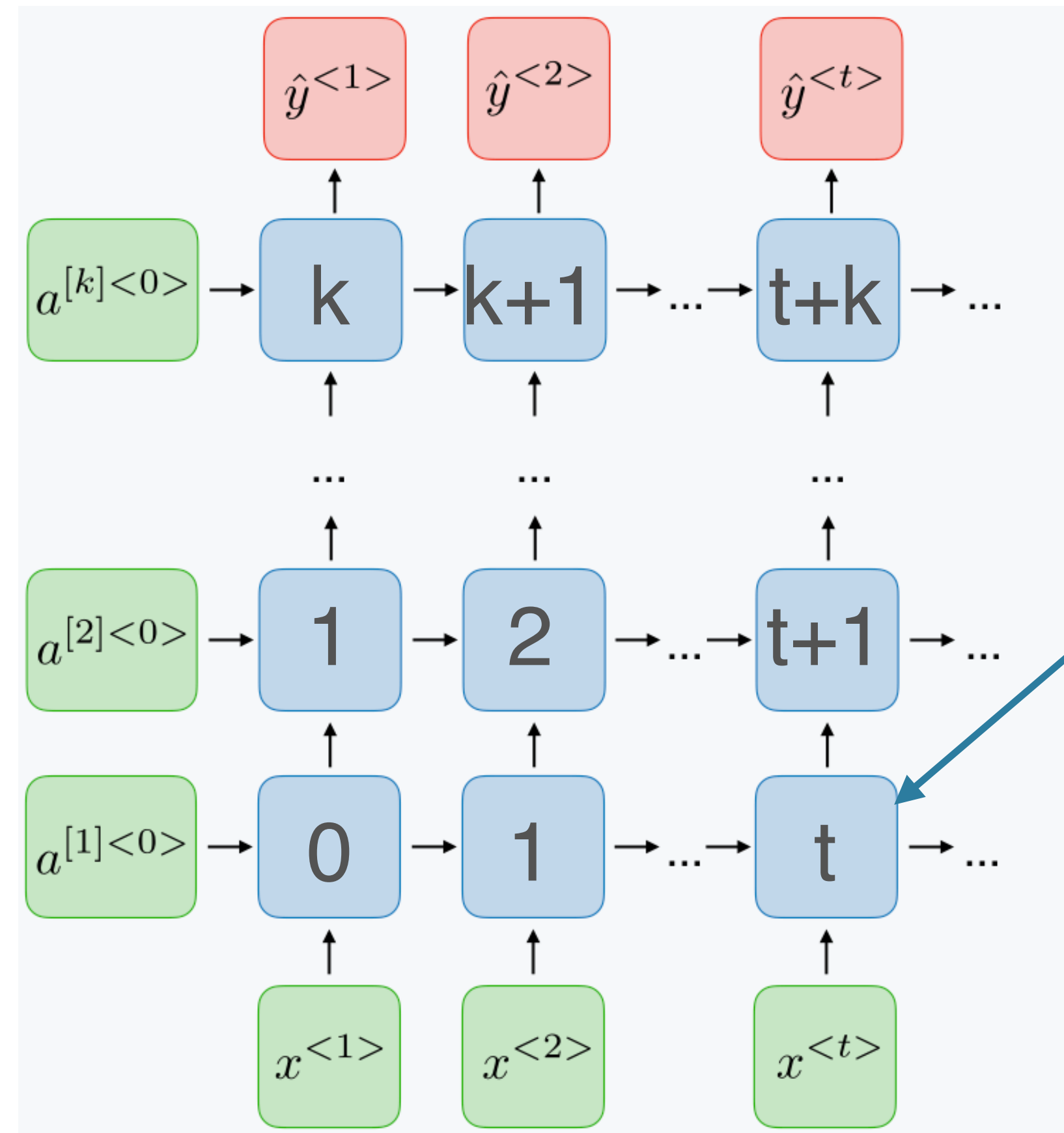Students who … enjoy

Lack of Parallelizability

- Modern hardware (e.g. GPUs) are very good at doing *independent* computations in parallel

- RNNs are inherently serial:
  - Cannot compute future time steps without the past

- Bottleneck that makes scaling up difficult

Number of computation steps required: linear in sequence length

Students who … enjoy

Transformer Architecture

Attention Is All You Need

Ashish Vaswani[*]
Google Brain
avaswani@google.com

Noam Shazeer[*]
Google Brain
noam@google.com

Niki Parmar[*]
Google Research
nikip@google.com

Jakob Uszkoreit[*]
Google Research
usz@google.com

Llion Jones[*]
Google Research
llion@google.com

Aidan N. Gomez[* †]
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser[*]
Google Brain
lukaszkaiser@google.com

Illia Polosukhin[* ‡]
illia.polosukhin@gmail.com

Paper link

(but see Annotated and Illustrated Transformer)

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.
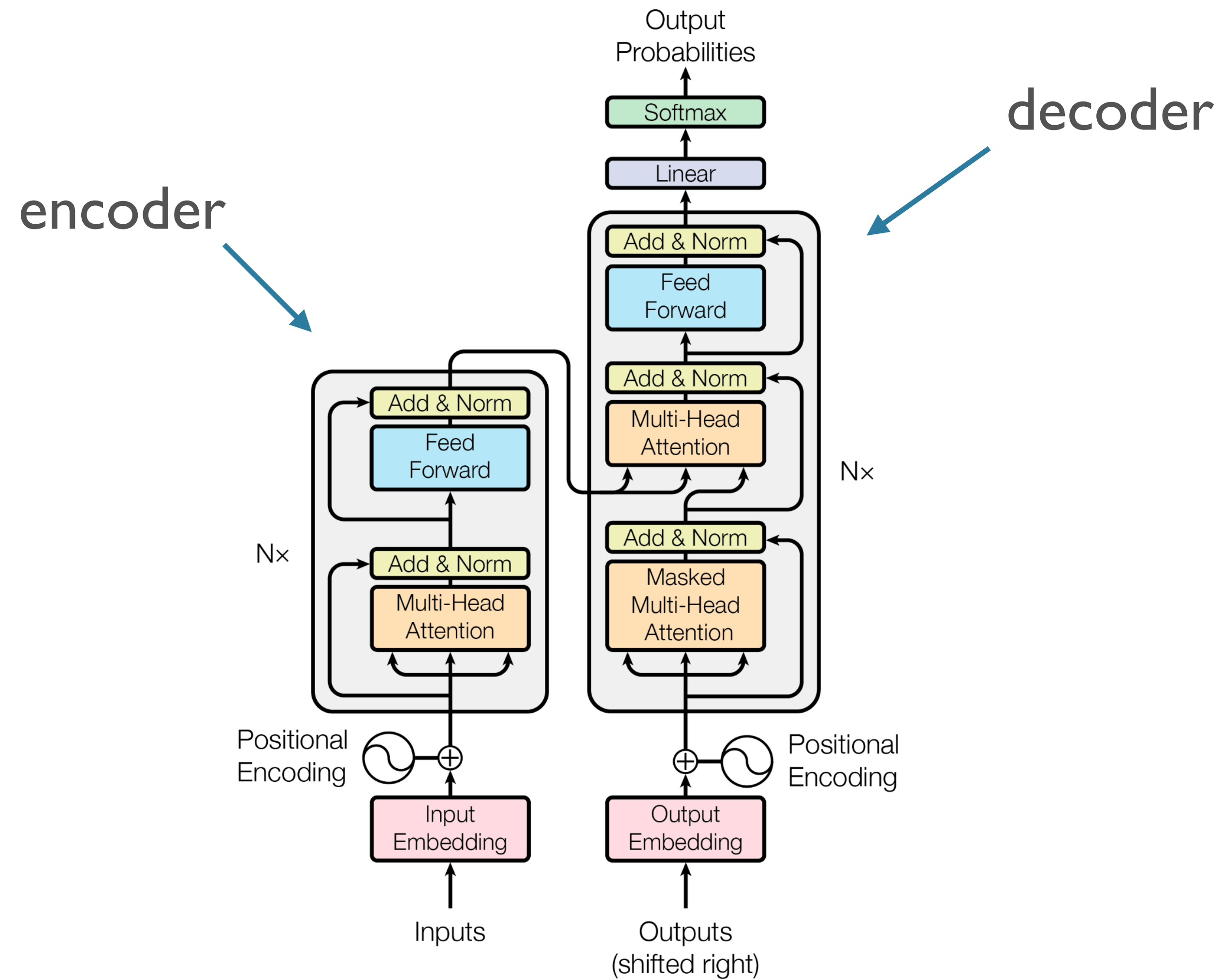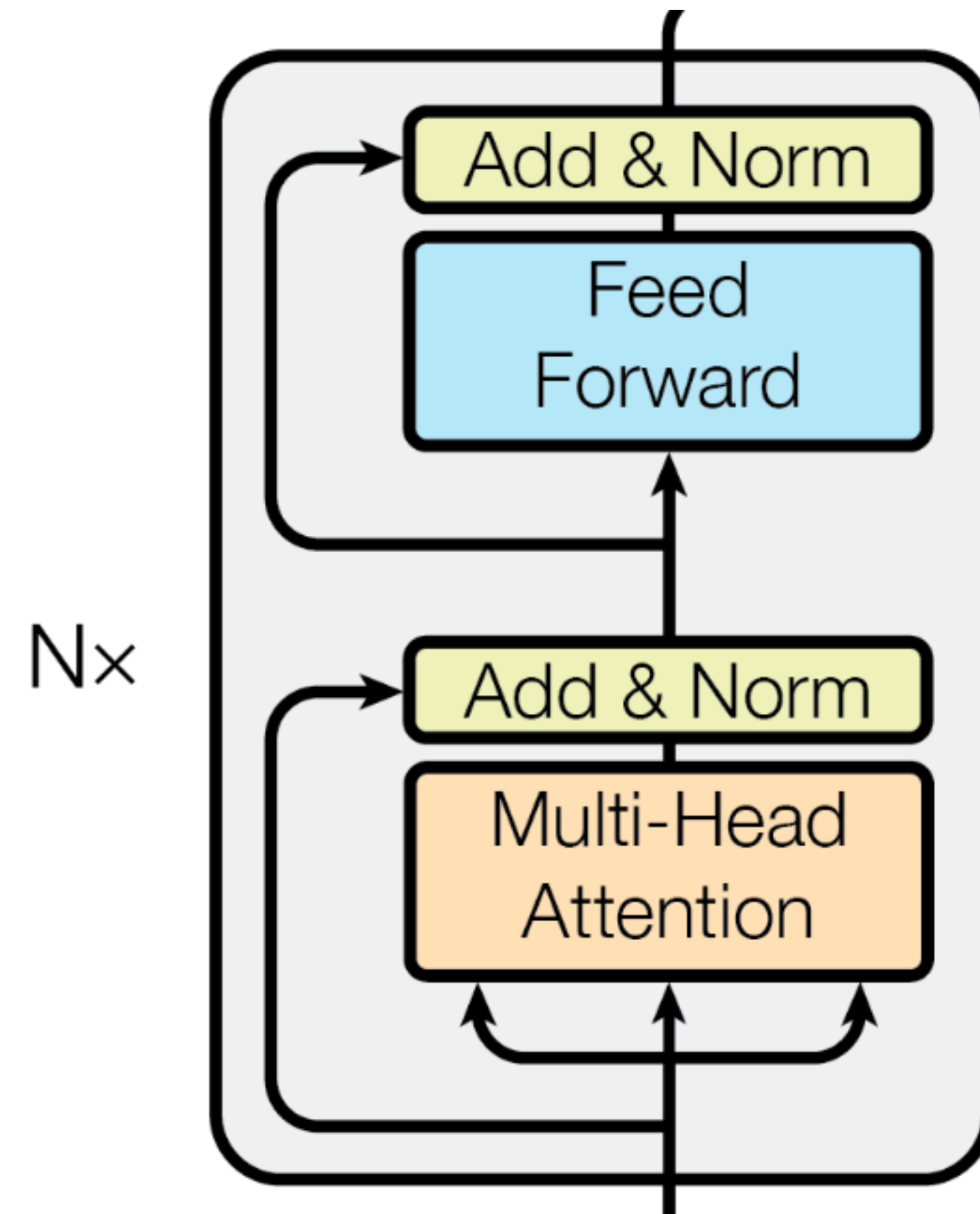
Key Idea

- Recurrence: not parallelizable, long "path lengths"

- *Attention*:

  - Parallelizable, short path lengths

- Transformer: "replace" recurrence with attention mechanism

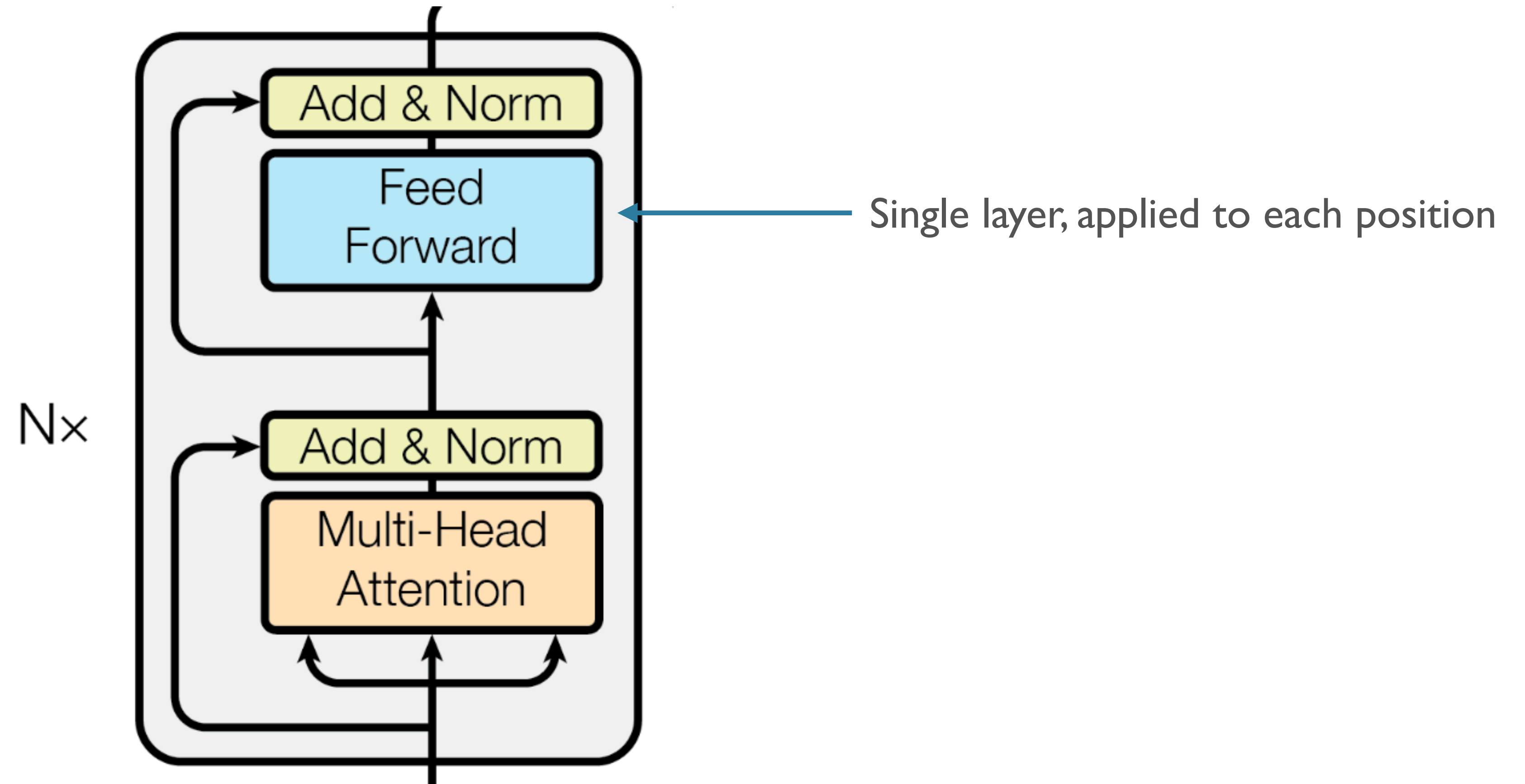 - Subtle issues in making this work, which we we will see
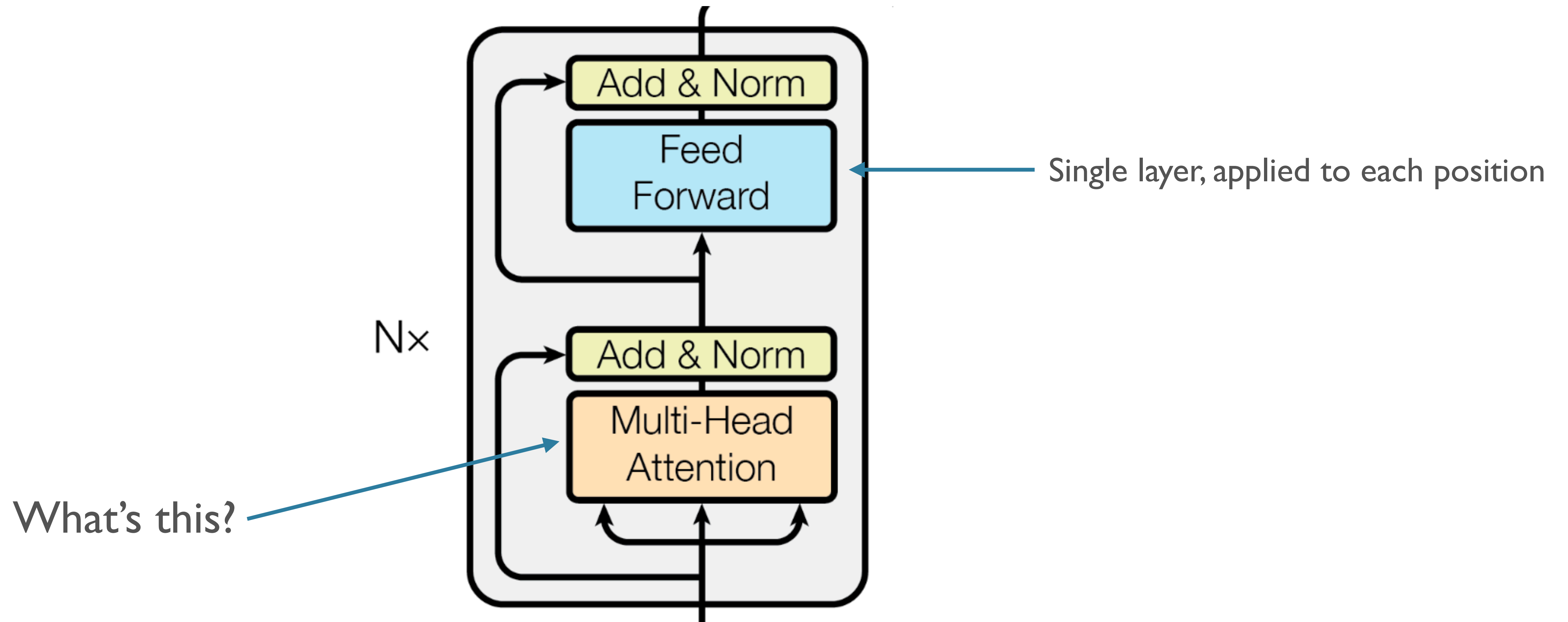
Full Model

Transformer Block

Transformer Block

Single layer, applied to each position

Transformer Block

Single layer, applied to each position

Nx

What's this?

Scaled Dot-Product Attention

- Recall:

- Putting it together: (keys/values in matrices)

$$\text{Attention}(q, K, V) = \sum_j \frac{e^{q \cdot k_j}}{\sum_i e^{q \cdot k_i}} v_j$$

- Stacking *multiple* queries: (and scaling)

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

Scaled Dot-Product Attention

- Recall:

$$\alpha_j = q \cdot k_j$$

$$e_j = e^{\alpha_j}/\Sigma_j e^{\alpha_j}$$

$$c = \Sigma_j e_j v_j$$

- Putting it together: (keys/values in matrices)

$$\text{Attention}(q, K, V) = \sum_j \frac{e^{q \cdot k_j}}{\sum_i e^{q \cdot k_i}} v_j$$

- Stacking *multiple* queries: (and scaling)

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

Scaled Dot-Product Attention

- Recall:

$$\alpha_j = q \cdot k_j$$

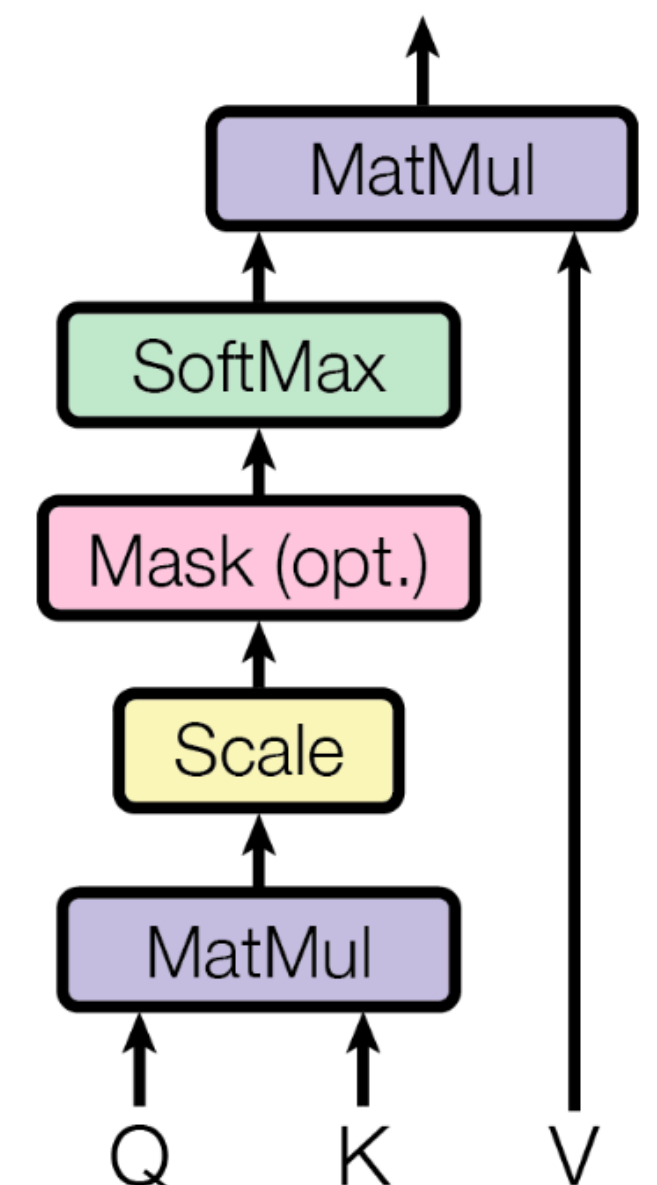$$e_j = e^{\alpha_j} / \Sigma_j e^{\alpha_j}$$

$$c = \Sigma_j e_j v_j$$

- Putting it together: (keys/values in matrices)

$$\text{Attention}(q, K, V) = \sum_j \frac{e^{q \cdot k_j}}{\sum_i e^{q \cdot k_i}} v_j$$

- Stacking *multiple* queries: (and scaling)

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

MatMul

SoftMax

Mask (opt.)

Scale

MatMul

Q   K   V

Why multiple queries?

Why multiple queries?

- seq2seq: single decoder token attends to *all* encoder states

Why multiple queries?

- seq2seq: single decoder token attends to *all* encoder states

- Transformer: *self*-attention

  - Every (token) position attends to every other position [including self!]

  - Caveat: in the encoder, and only by default

    - Mask in decoder to attend only to previous positions [next time]

      - Used for generation [NMT, LM, etc]

Why multiple queries?

- seq2seq: single decoder token attends to *all* encoder states

- Transformer: *self*-attention

  - Every (token) position attends to every other position [including self!]

  - Caveat: in the encoder, and only by default

    - Mask in decoder to attend only to previous positions [next time]

      - Used for generation [NMT, LM, etc]

- So vector at each position is a query

  - And a key, and a value

  - Linearly transformed, to be different "views"

Self-Attention, Details

- Every token attends to every other token

- X: [seq_len, embedding_dim]

  - XW_q: queries

  - XW_k: keys

  - XW_v: values

  - Each W is [embedding_dim, embedding_dim] learned matrix

Self-Attention: Details

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

- Q = XW_q, K = XW_k, V = XW_v
  - K^T: [embedding_dim, seq_len]
  - QK^T: [seq_len, seq_len]
    - Dot-product of rows of Q with columns of K
  - $(QK^T)_{ij} = q_i \cdot k_j$
- Scaled by sq-rt of hidden dimension [see paper for motivation]

- Softmax: along *rows*, gets the weights

Self-Attention: Details

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

- Softmax output: each row has weights

  - How much q_i should pay attention to each v_j

- Matrix multiplication with V: output is [seq_len, embedding_dim]

  - Each row: weighted average of the v_j (rows of V)

  - Each row: the weight sum attention value for each query (each input token)

- [NB: a more explicit notation, if you like: https://namedtensor.github.io/]

Multi-headed Attention

- So far: a *single* attention mechanism.

- Could be a bottleneck: need to pay attention to different vectors *for different reasons*

- Multi-headed: several attention mechanisms in parallel

Multi-headed Attention

- So far: a *single* attention mechanism.

- Could be a bottleneck: need to pay attention to different vectors *for different reasons*

- Multi-headed: several attention mechanisms in parallel

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\textbf{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Multi-headed Attention

- So far: a *single* attention mechanism.

- Could be a bottleneck: need to pay attention to different vectors *for different reasons*

- Multi-headed: several attention mechanisms in parallel

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

$$\textbf{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$
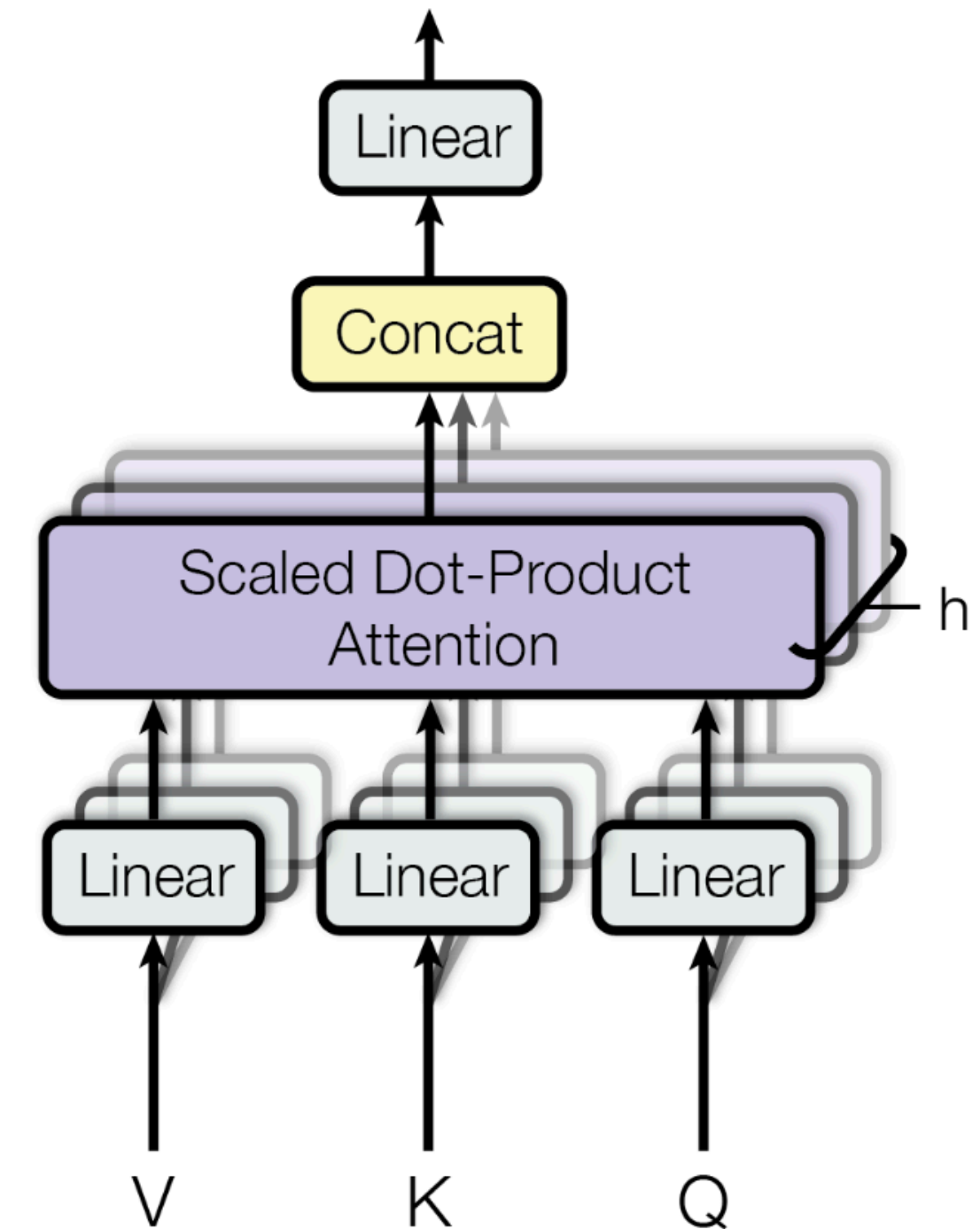
Problem With Self-Attention

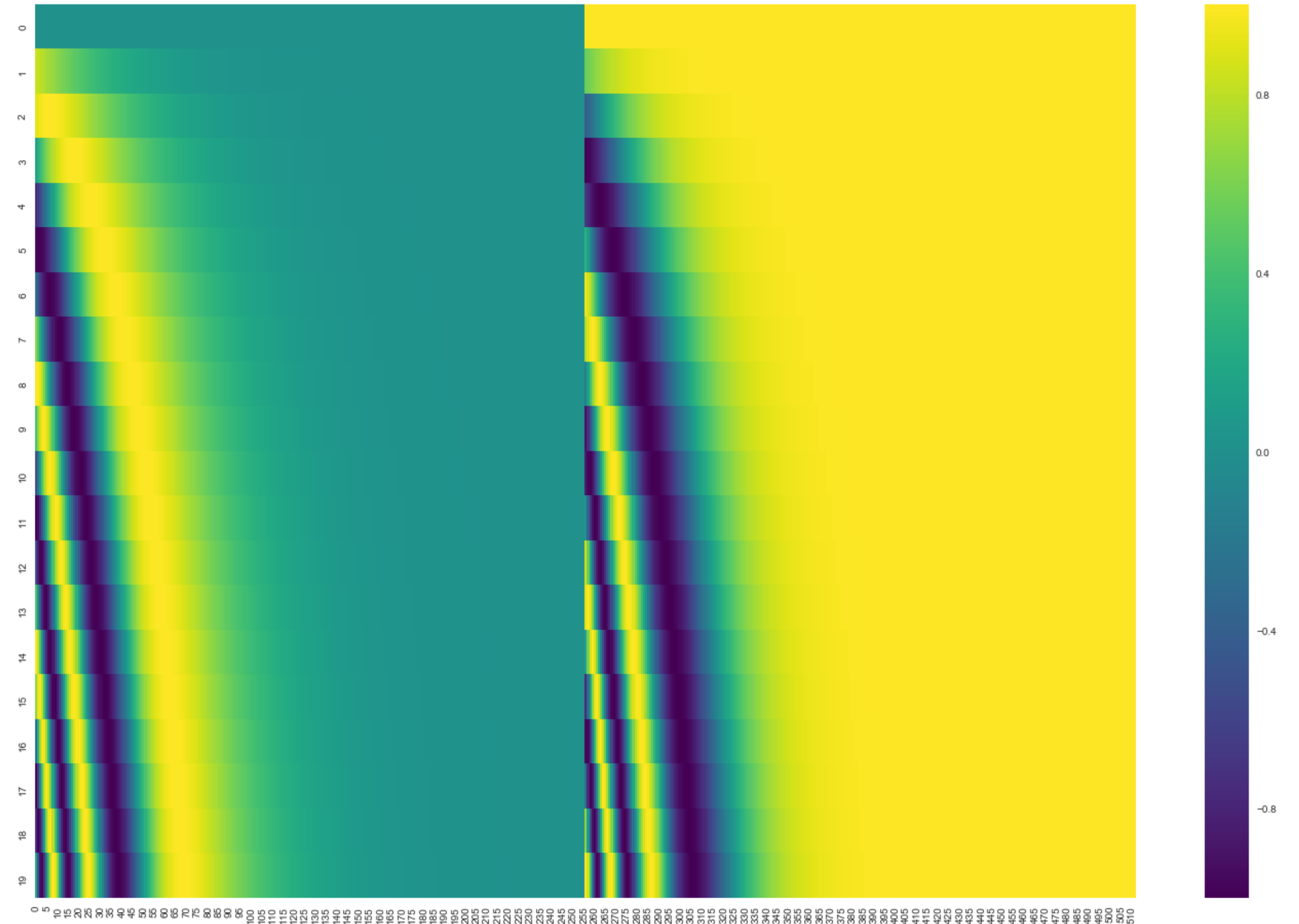- Attention is order-independent
  - If we shuffle Q, K, V, we get the same output!

Representing Order

Representing Order

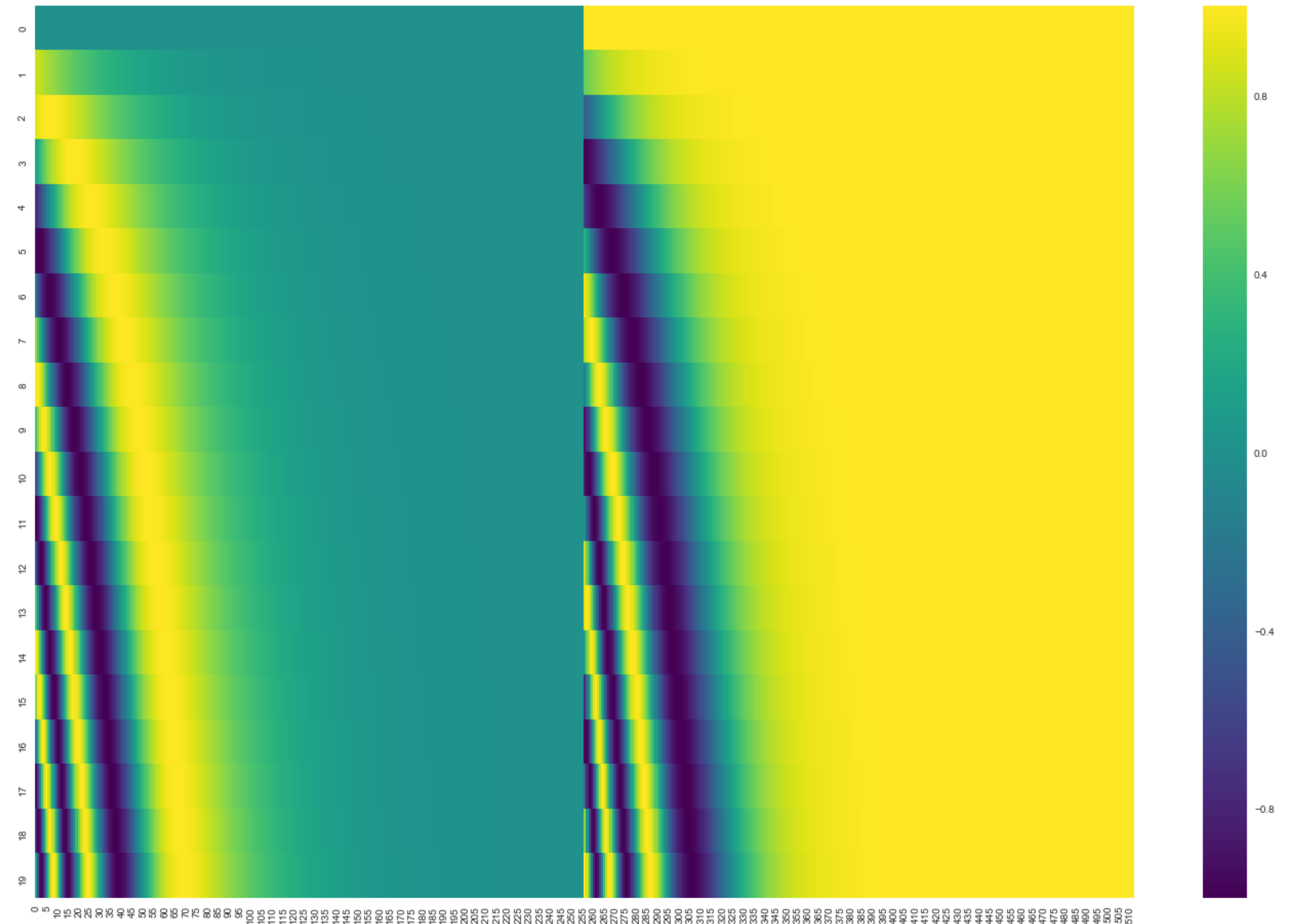- Represented via *positional* encodings.

Representing Order

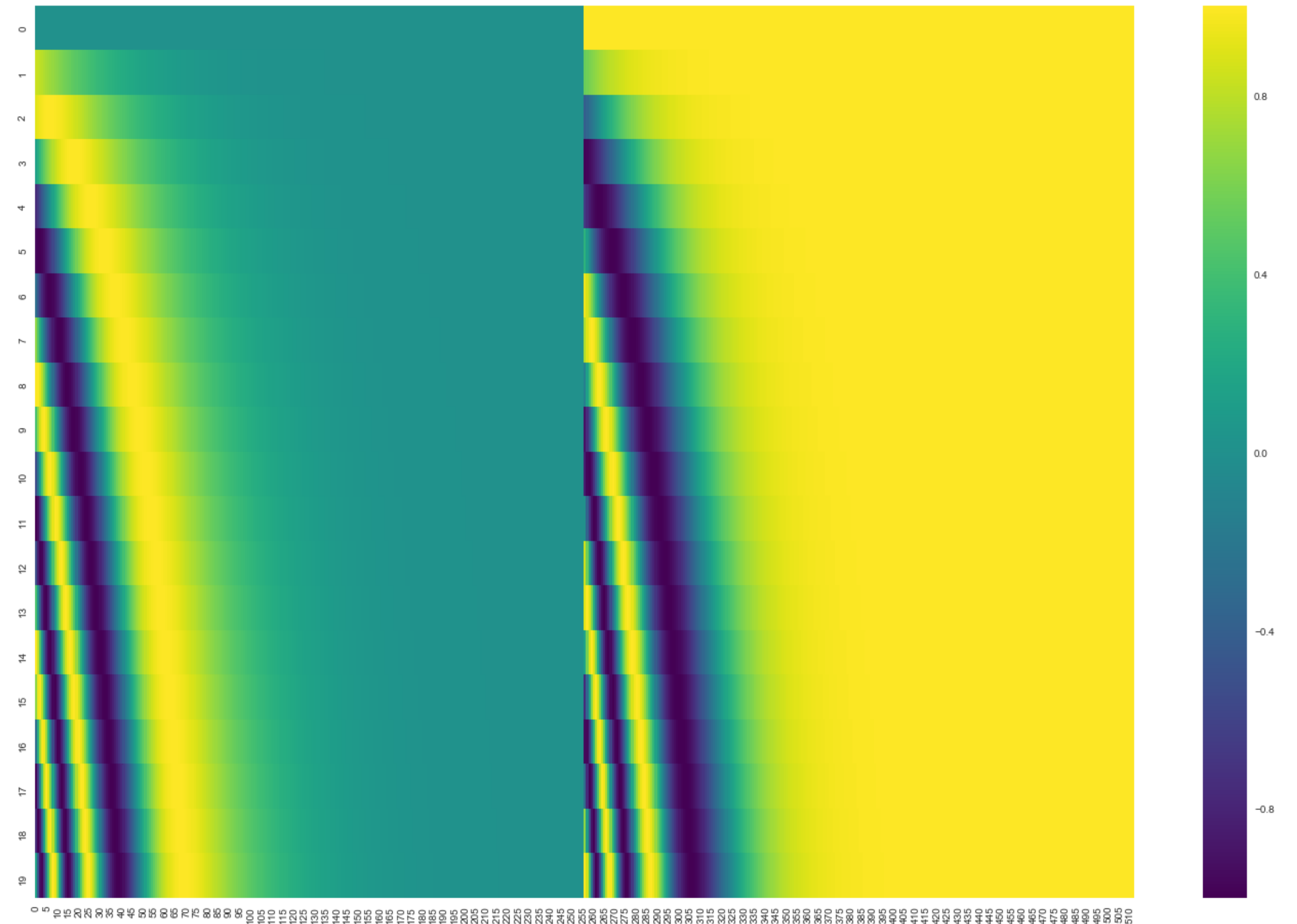- Represented via *positional* encodings.

Representing Order

- Represented via *positional* encodings.

- *P:* [seq_len, embedding_dim]
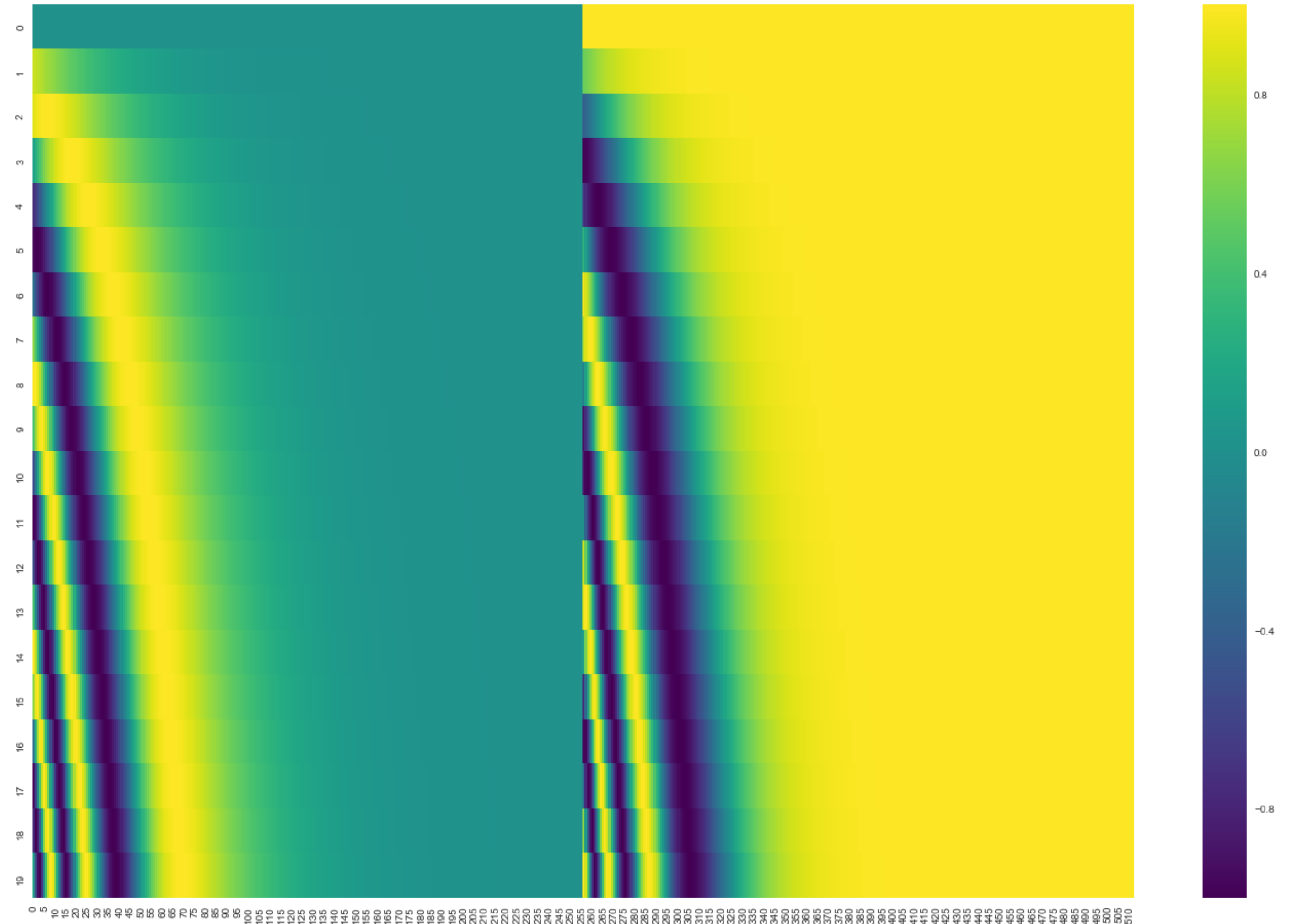

source

Representing Order

- Represented via *positional* encodings.

- *P:* [seq_len, embedding_dim]

  - Each row *i* represents that position in the sequence
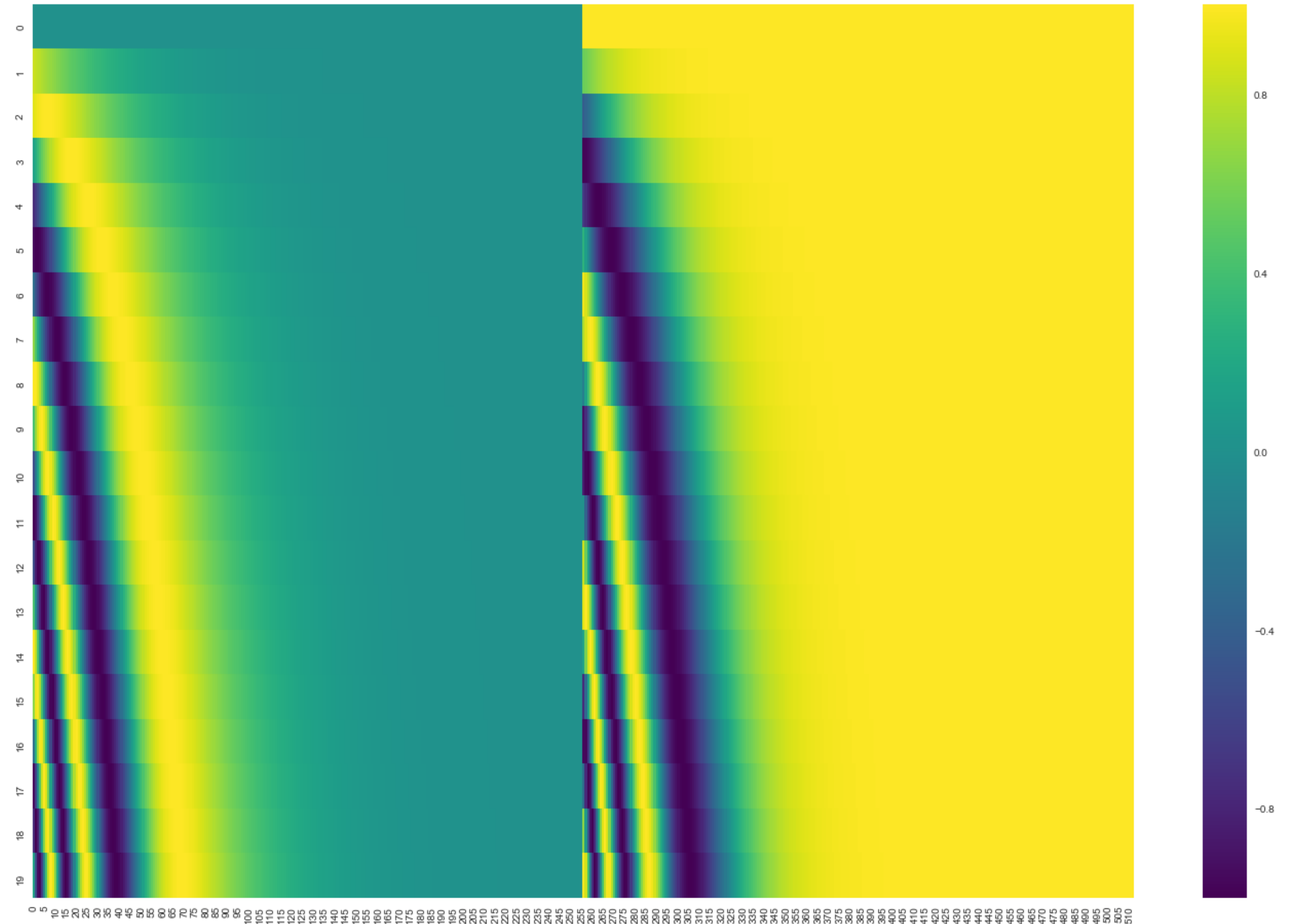

source

Representing Order

- Represented via *positional* encodings.

- *P:* [seq_len, embedding_dim]

  - Each row *i* represents that position in the sequence

  - Add to word embeddings at input layer:

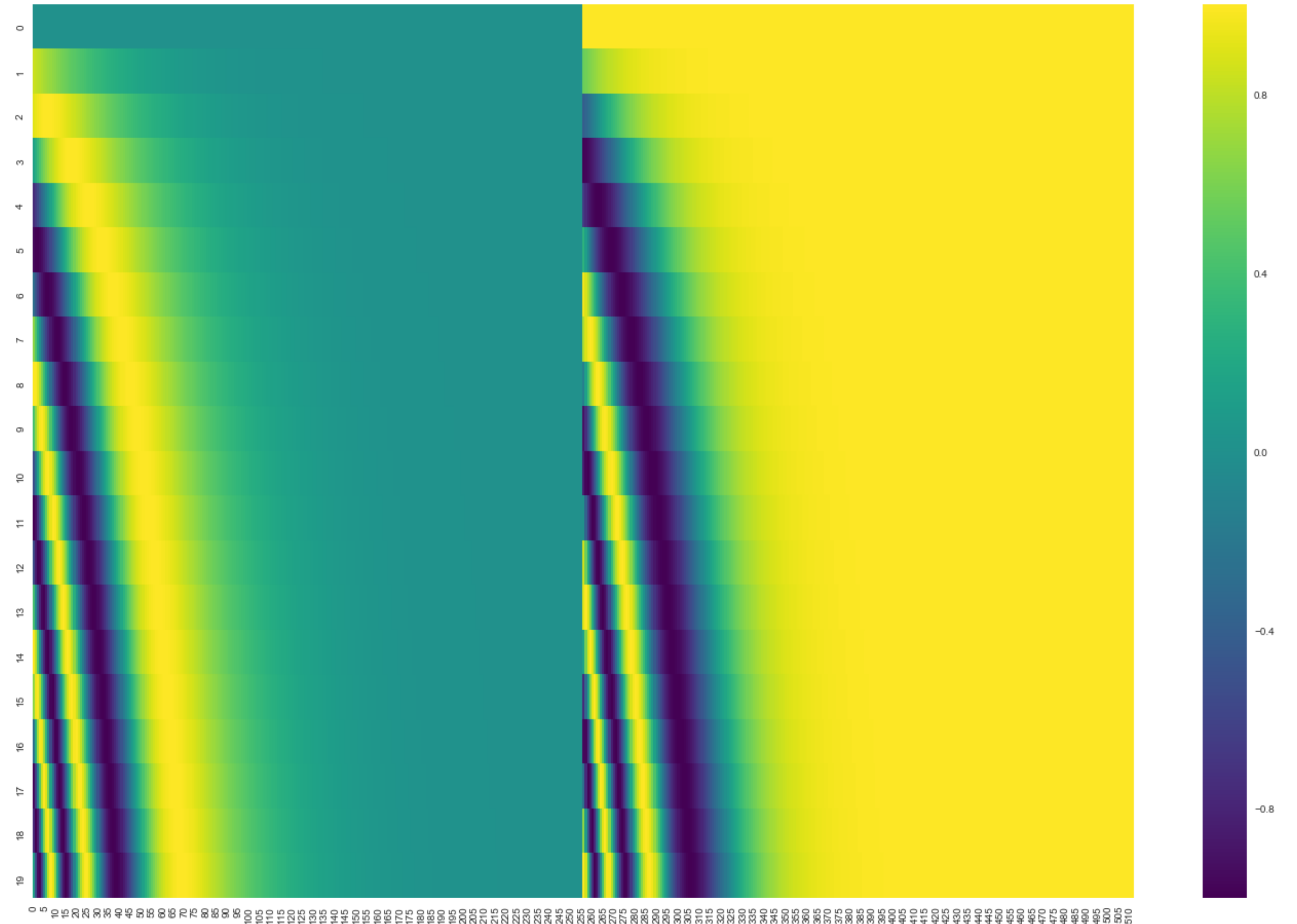

source

Representing Order

- Represented via *positional* encodings.

- *P:* [seq_len, embedding_dim]

  - Each row *i* represents that position in the sequence

  - Add to word embeddings at input layer:

    - $x_i = E_{w_i} + P_i$

Representing Order

- Represented via *positional* encodings.

- *P:* [seq_len, embedding_dim]

  - Each row *i* represents that position in the sequence

  - Add to word embeddings at input layer:

    - $x_i = E_{w_i} + P_i$

- Can be fixed/pre-defined [see right] or entirely learned

source

Fixed vs Learned Positional Encoding

- Fixed:
  - No need to be learned
  - Guaranteed to be unique to position
  - Generalizes to longer sequence lengths (in theory at least)

- Learned:
  - Might learn more useful encodings of position than e.g. sinusoidal
  - Can't extrapolate to longer sequence lengths
  - [This has become the default/norm]

- Fancier ways of representing positional info: rotary embeddings, learned bias of distance, <u>fixed bias of distance</u> (ALiBi)

Basic Transformer Encoder Block
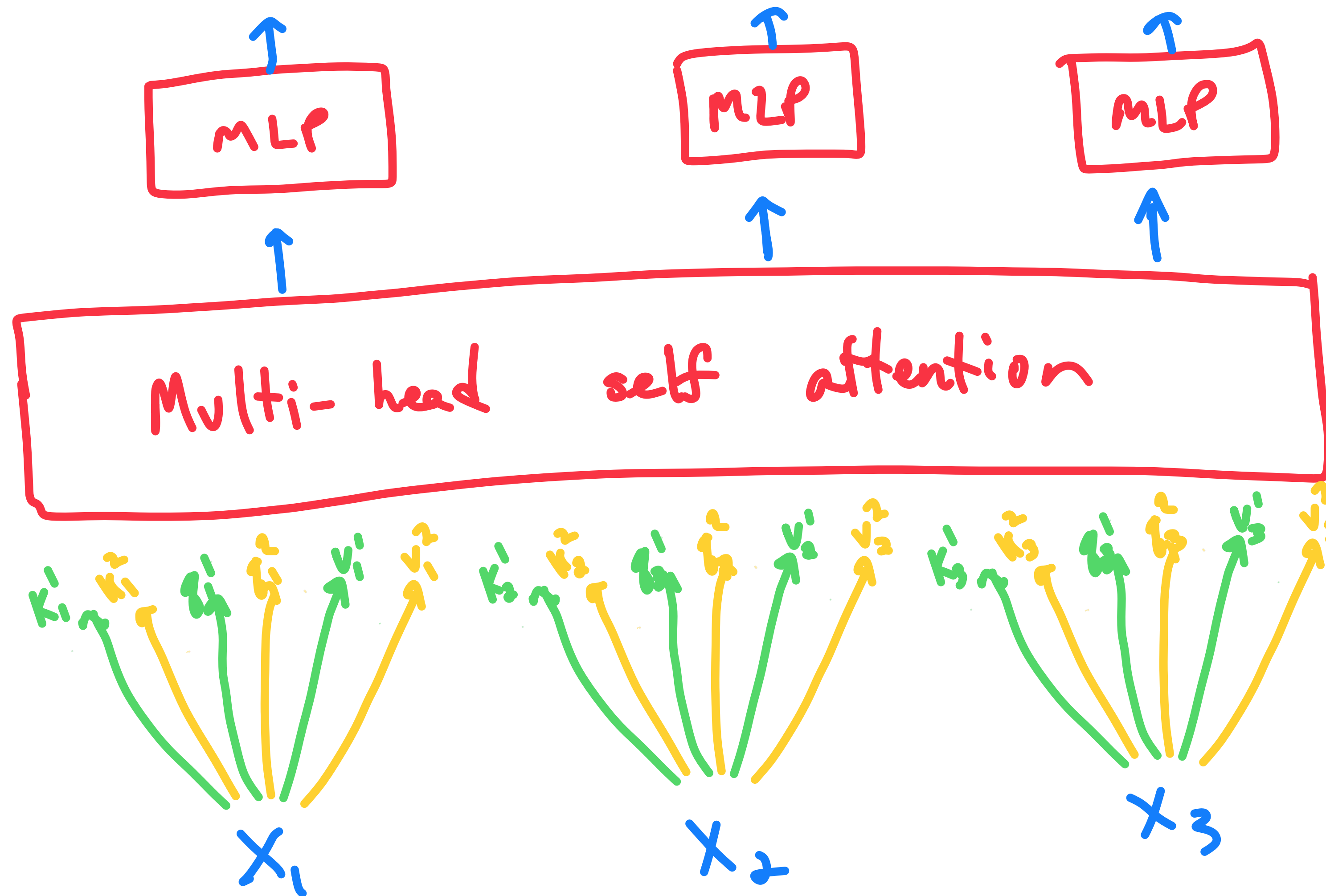
same MLP
applied at
each position

shared
W_q^i, W_k^i, W_v^i

Final Ingredients: <u>Residual Connections</u>

- Core idea: add a "skip" connection around neural building blocks

- Replace $f(x)$ with $x + f(x)$

- Makes training work much better, by smoothing out loss surface

- In Transformer: residual connection around both self-attention and feed-forward blocks

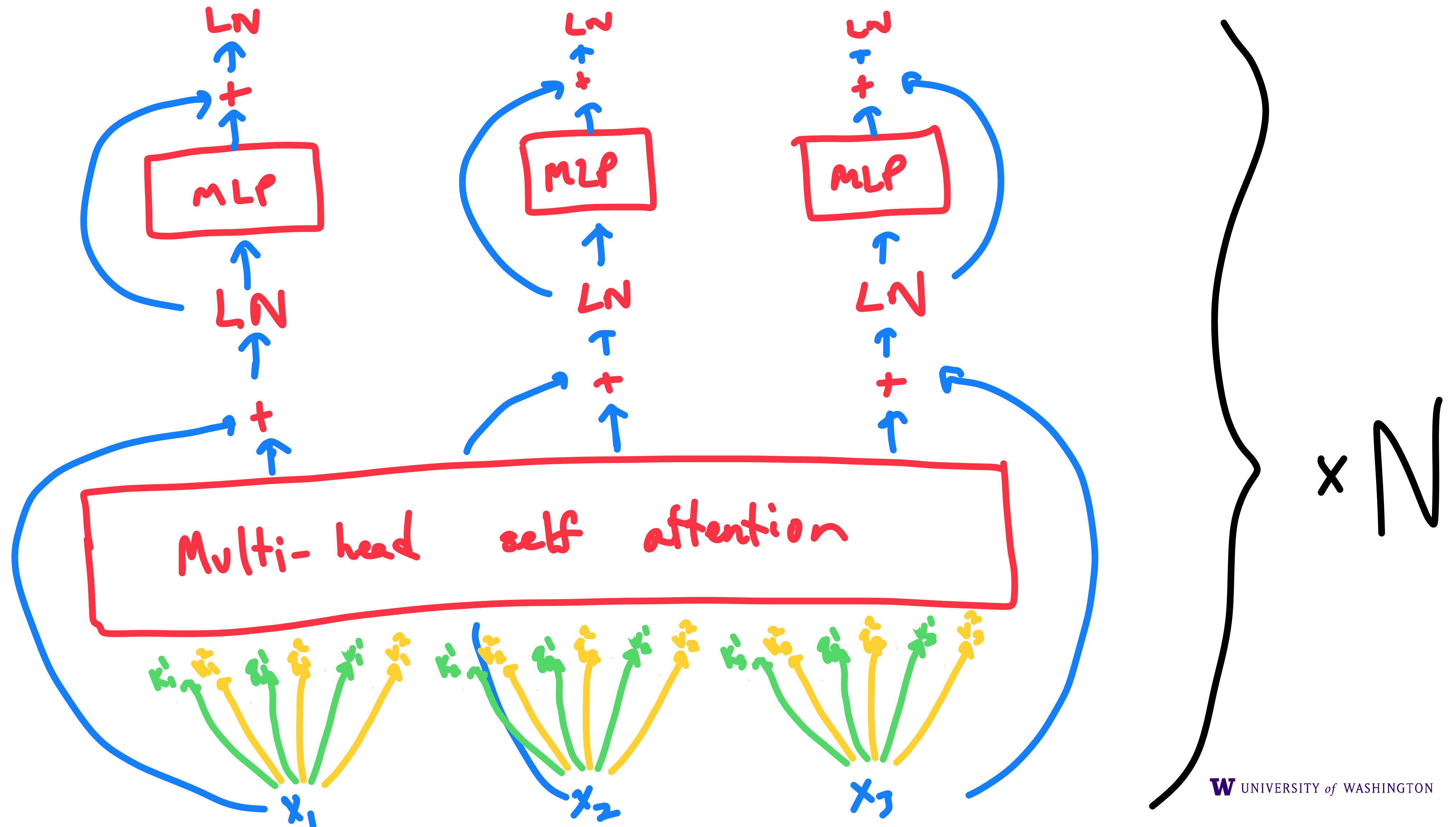- Used widely now: FFNNs, CNNs, RNNs, Transformers, ...

(a) without skip connections   (b) with skip connections
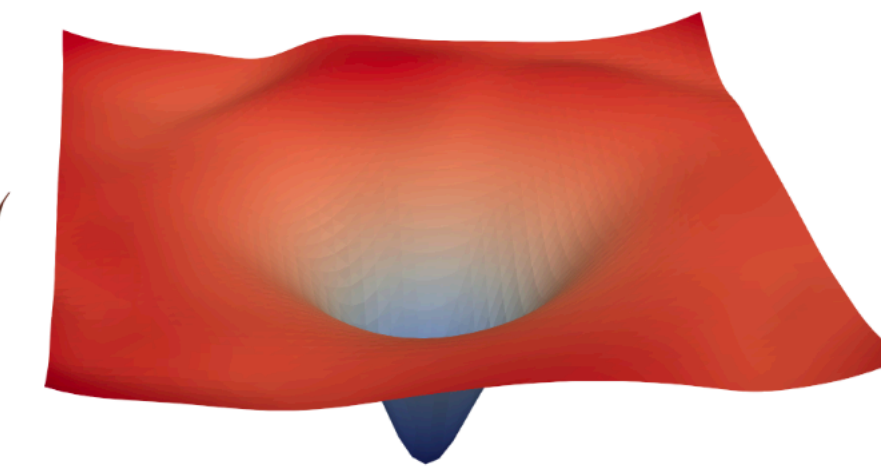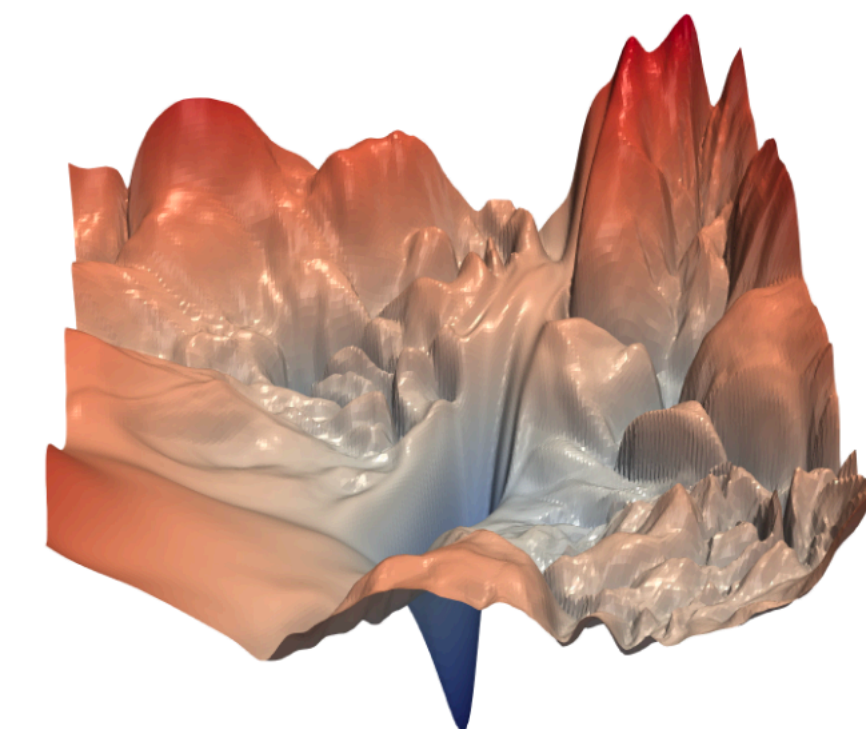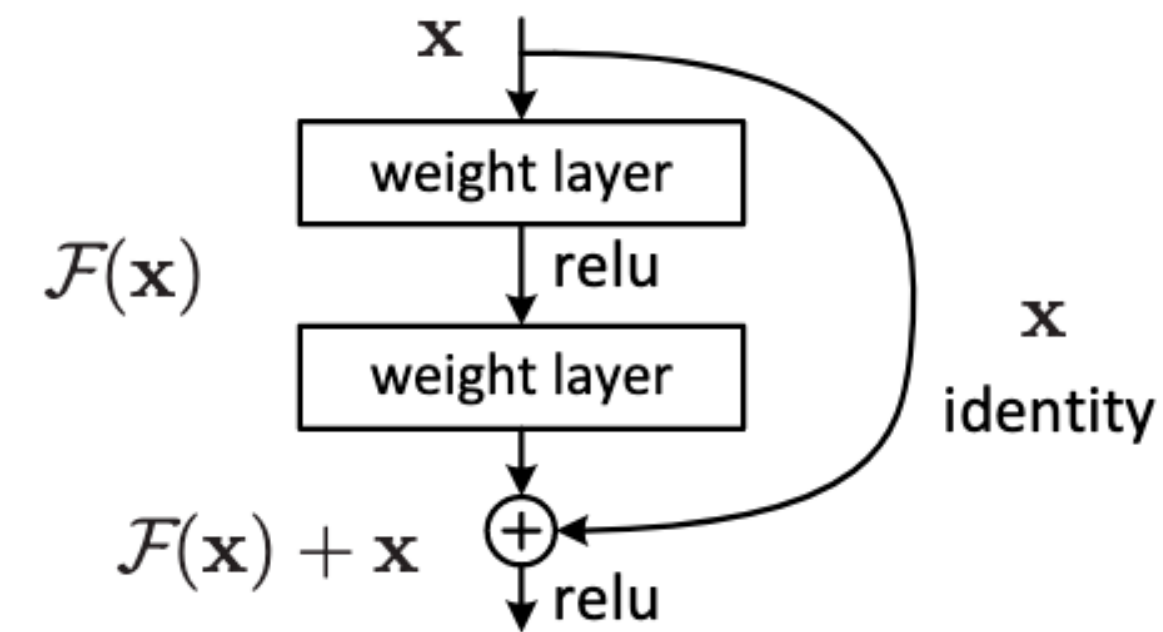
<u>source</u>

Final Ingredients: <u>Layer Normalization</u>

- Normalizing inputs: subtract mean, divide by standard deviation

  - Makes new mean 0, new standard deviation 1

  - Widely used in many kinds of statistical modeling [e.g. z-scoring predictors in linear regression], including in NNs

- Layer norm: to each row x of a matrix [a batch]: $LN(x) = \dfrac{x - \mu}{\sigma + \epsilon}\gamma + \beta$

  - Where μ is mean, σ is std dev

  - γ, β are learned scaling parameters [but often omitted entirely]

Full Transformer Encoder Block

$\times N$

Initial WMT Results

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	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**	**41.0**	$2.3 \cdot 10^{19}$	

Initial WMT Results

| Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	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**	**41.0**	$2.3 \cdot 10^{19}$	

More on why
important later

Attention Visualization: Coreference?
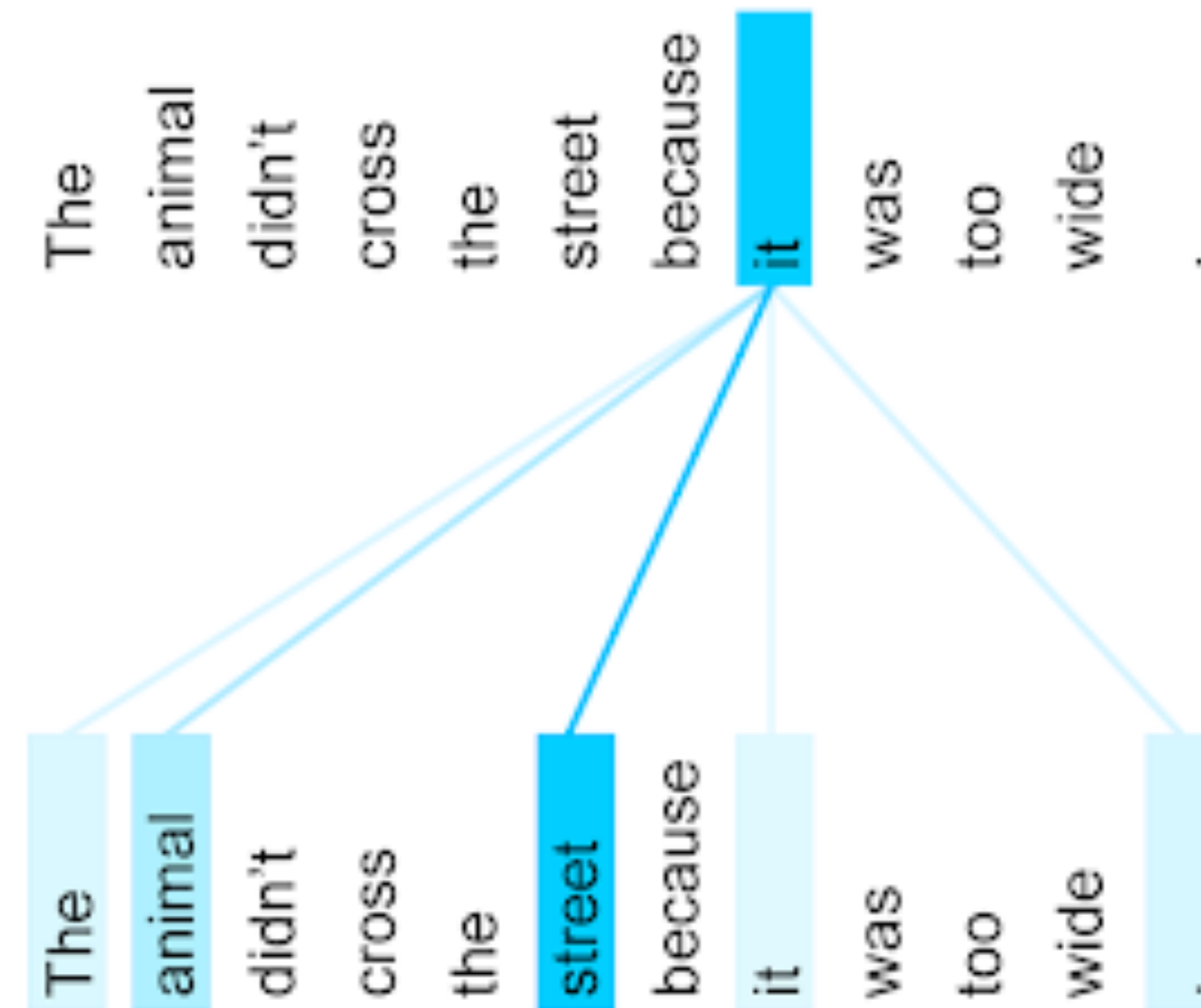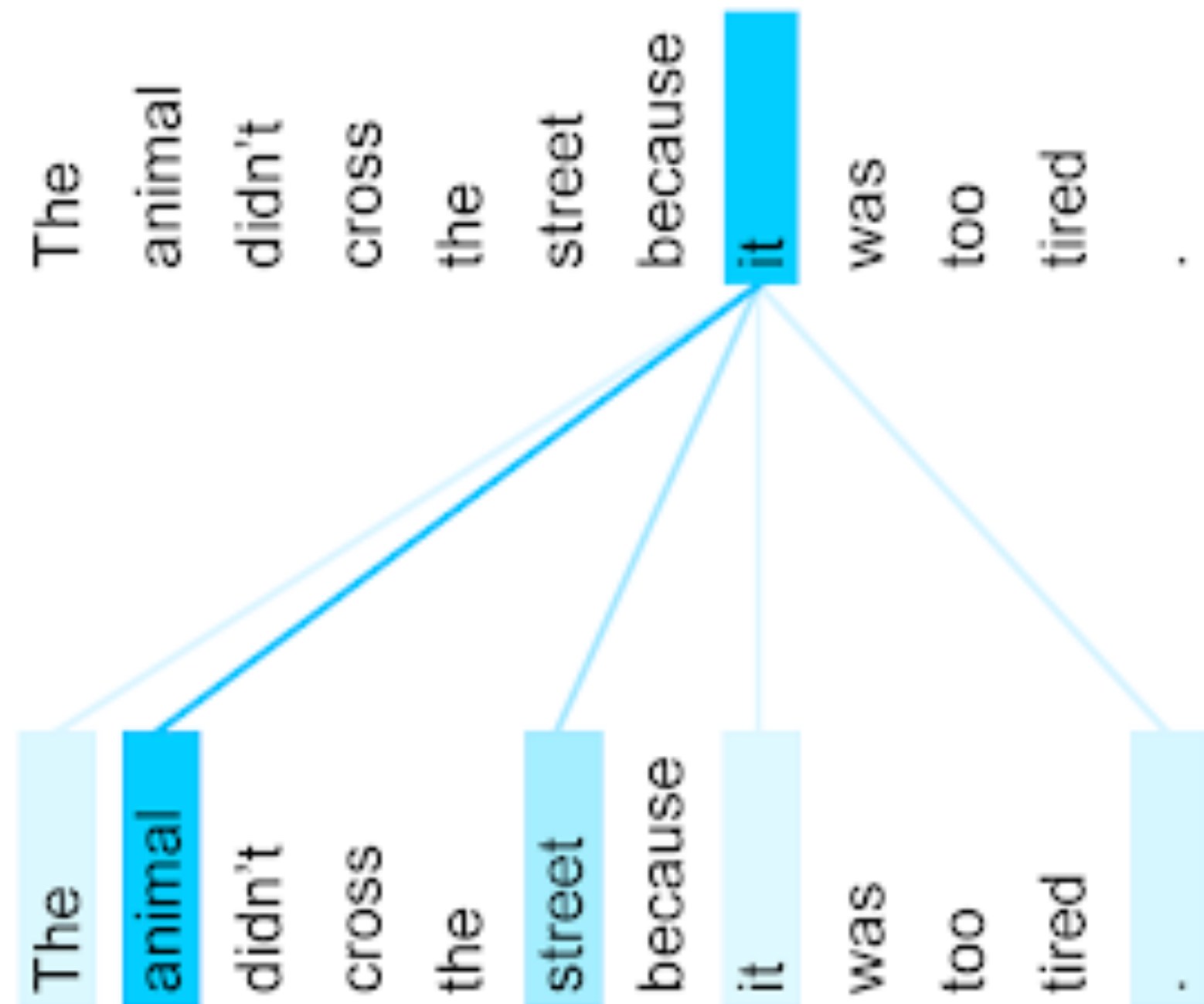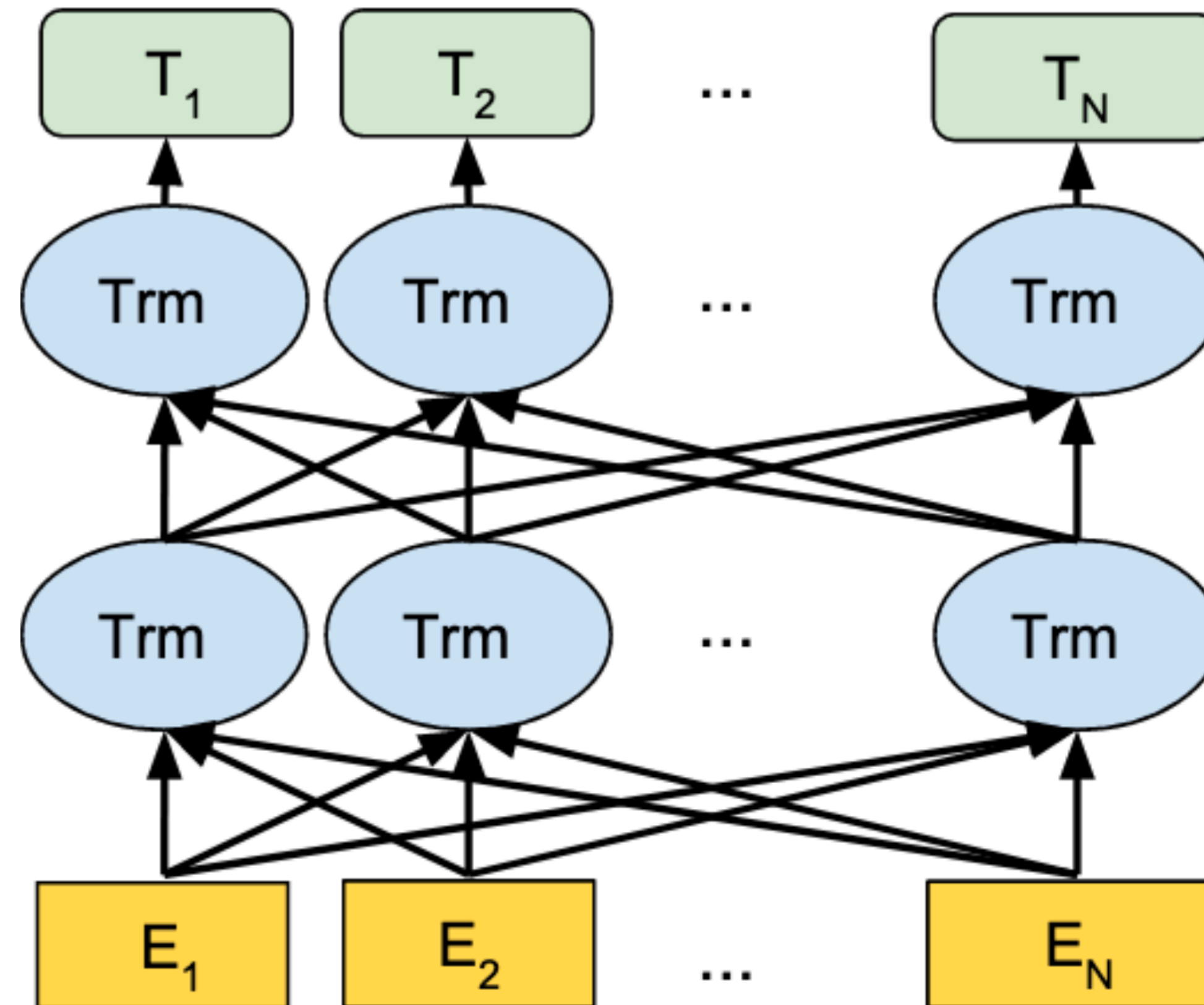
Transformer: Path Lengths + Parallelism
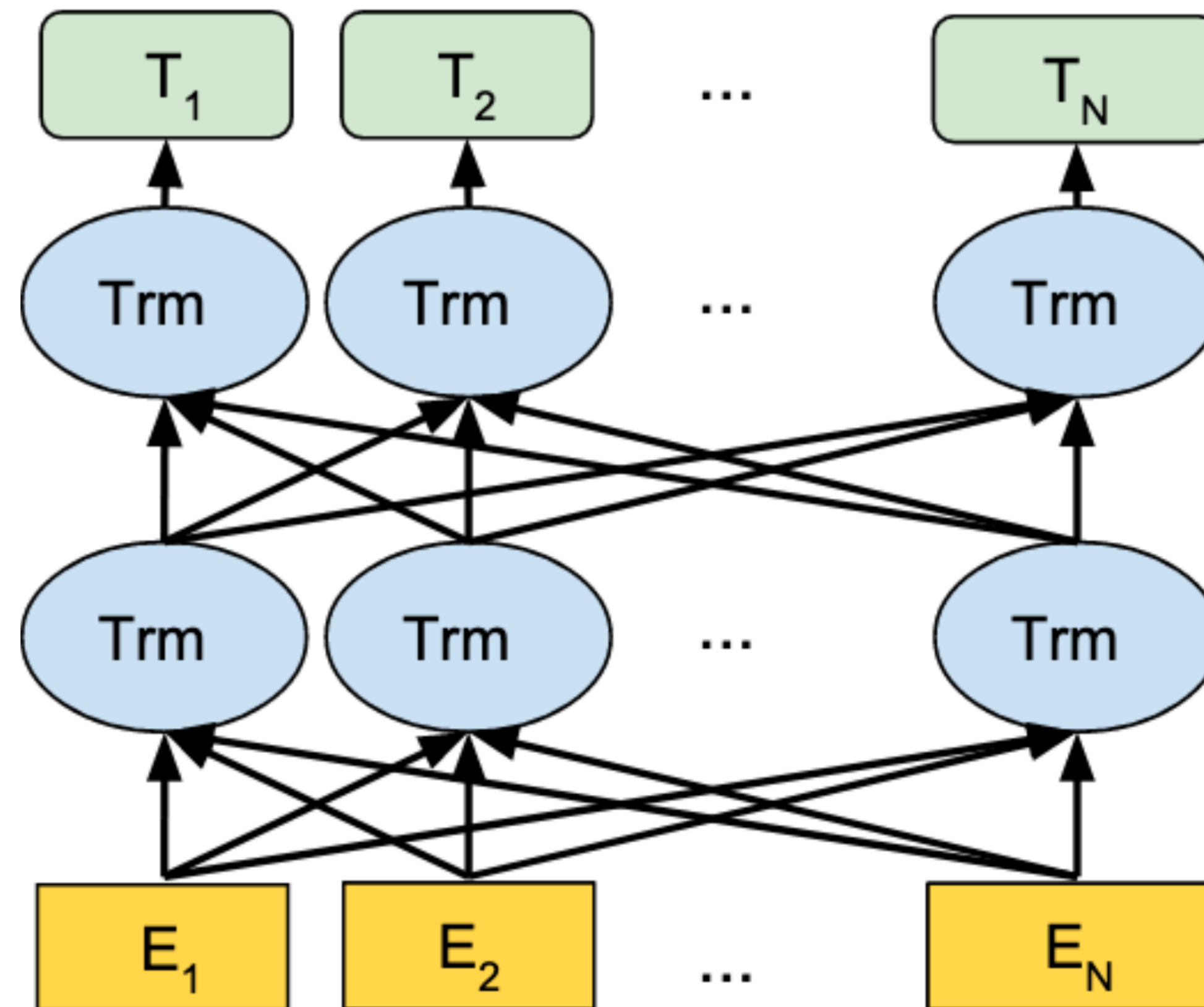
Transformer: Path Lengths + Parallelism

Path lengths between tokens: 1
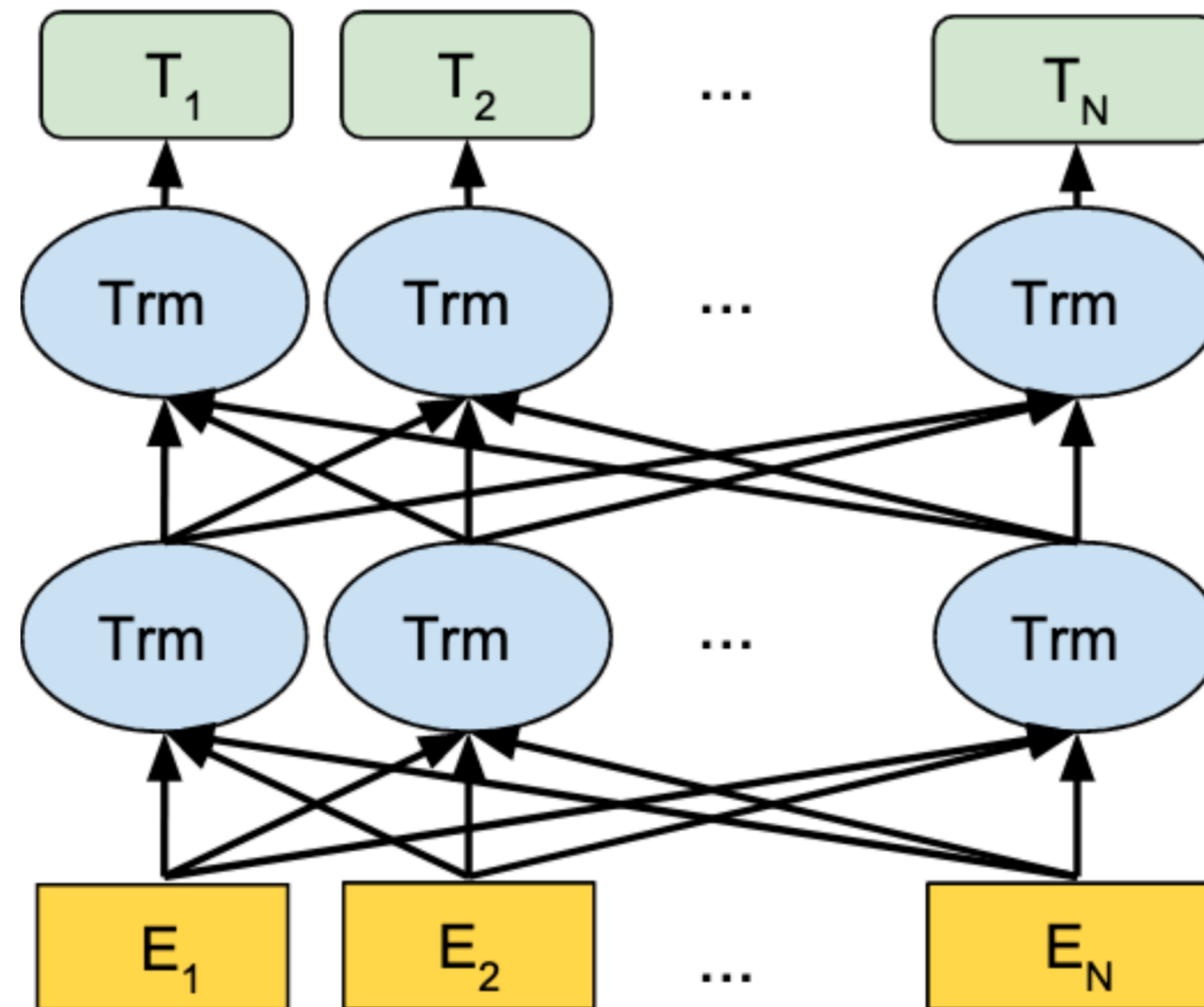[constant, not linear]

Transformer: Path Lengths + Parallelism

Computation order:

Entire second layer: 1

Entire first layer: 0

Also not linear in sequence length! Can be parallelized.

Path lengths between tokens: 1 [constant, not linear]

Transformer: Summary

- *Entirely* feed-forward

  - Therefore massively parallelizable

  - RNNs are inherently sequential, a parallelization bottleneck

- (Self-)attention everywhere

- Long-term dependencies:

  - LSTM: has to maintain representation of early item

  - Transformer: very short "path-lengths"

Next Time

- A deeper look at the *decoder* block of a Transformer

  - Attention masks

- Subword tokenization