香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

COMP 4901B

Large Language Models

Recurrent Neural Networks, Transformers

Junxian He

Sep 12, 2025

Recap: Autoregressive Language Models

$$
\begin{aligned}
p(\text{the, mouse, ate, the, cheese}) = \; & p(\text{the}) \\
& p(\text{mouse} \mid \text{the}) \\
& p(\text{ate} \mid \text{the, mouse}) \\
& p(\text{the} \mid \text{the, mouse, ate}) \\
& p(\text{cheese} \mid \text{the, mouse, ate, the}).
\end{aligned}
$$

$P(x_i | x_{1:i-1})$

from

NN

$$p(x_1, x_2, \ldots, x_I) = \prod_{i=1}^{I} p(x_i \mid x_{1:i-1})$$

Next Word

Context

Recap: Neural Language Models

Recap: Neural Language Models

Neural language models are typically autoregressive

Recap: Neural Language Models

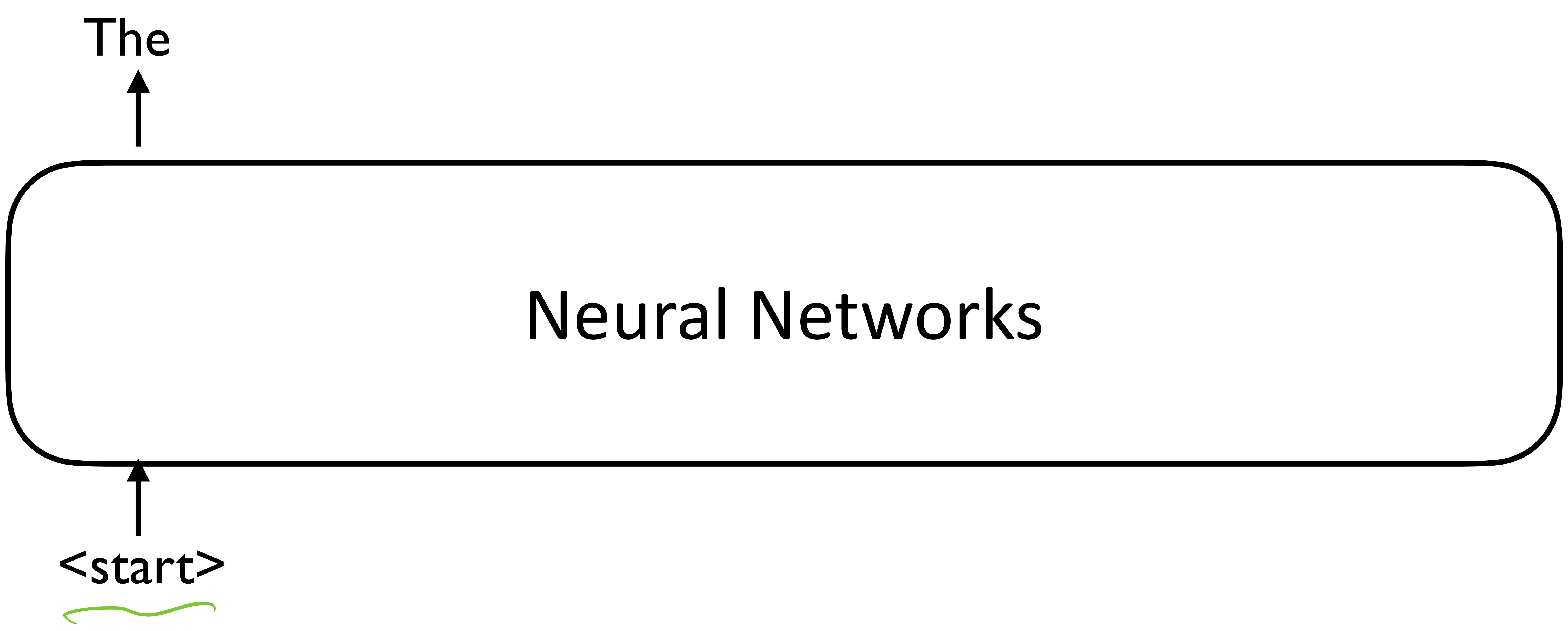
Neural language models are typically autoregressive

Data: "The mouse ate the cheese ."

Recap: Neural Language Models

Neural language models are typically autoregressive

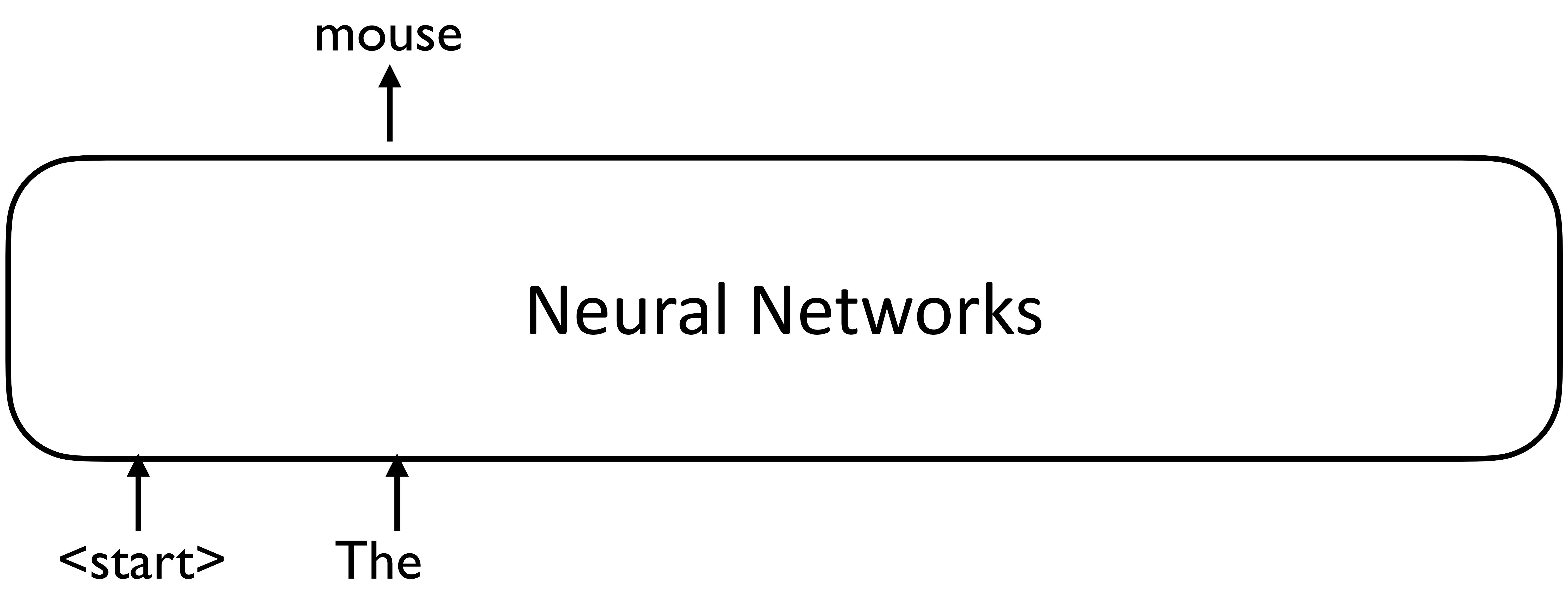
Data: "The mouse ate the cheese ."



Recap: Neural Language Models

Neural language models are typically autoregressive

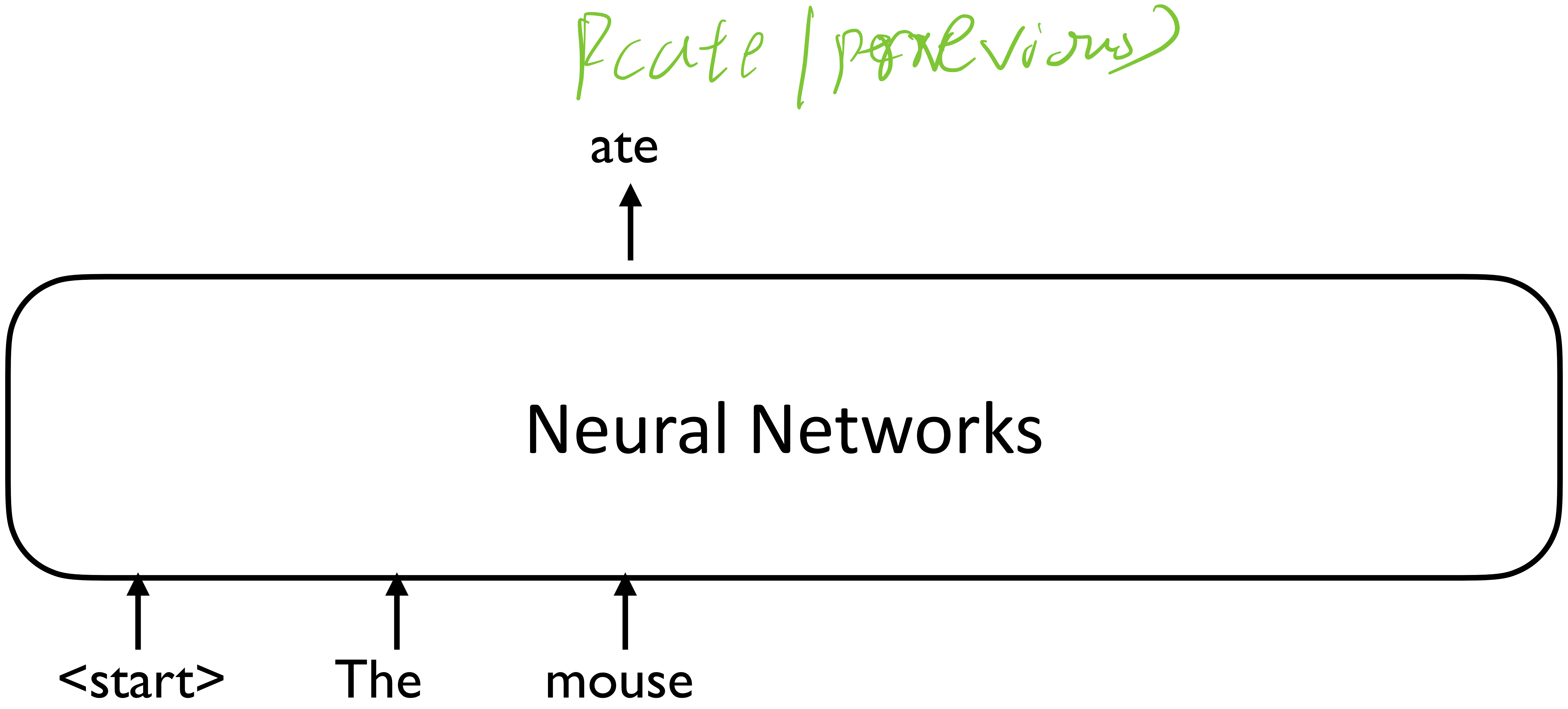
Data: "The mouse ate the cheese ."



Recap: Neural Language Models

Neural language models are typically autoregressive

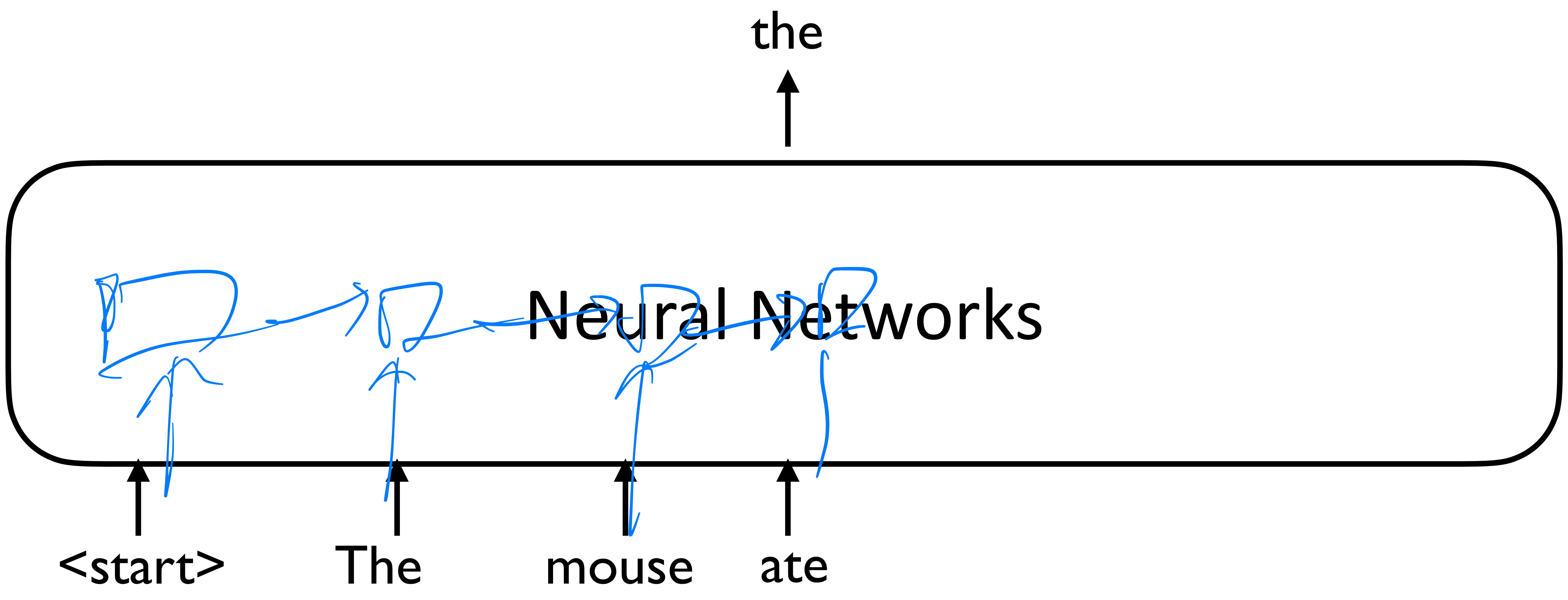
Data: "The mouse ate the cheese ."



Recap: Neural Language Models

Neural language models are typically autoregressive

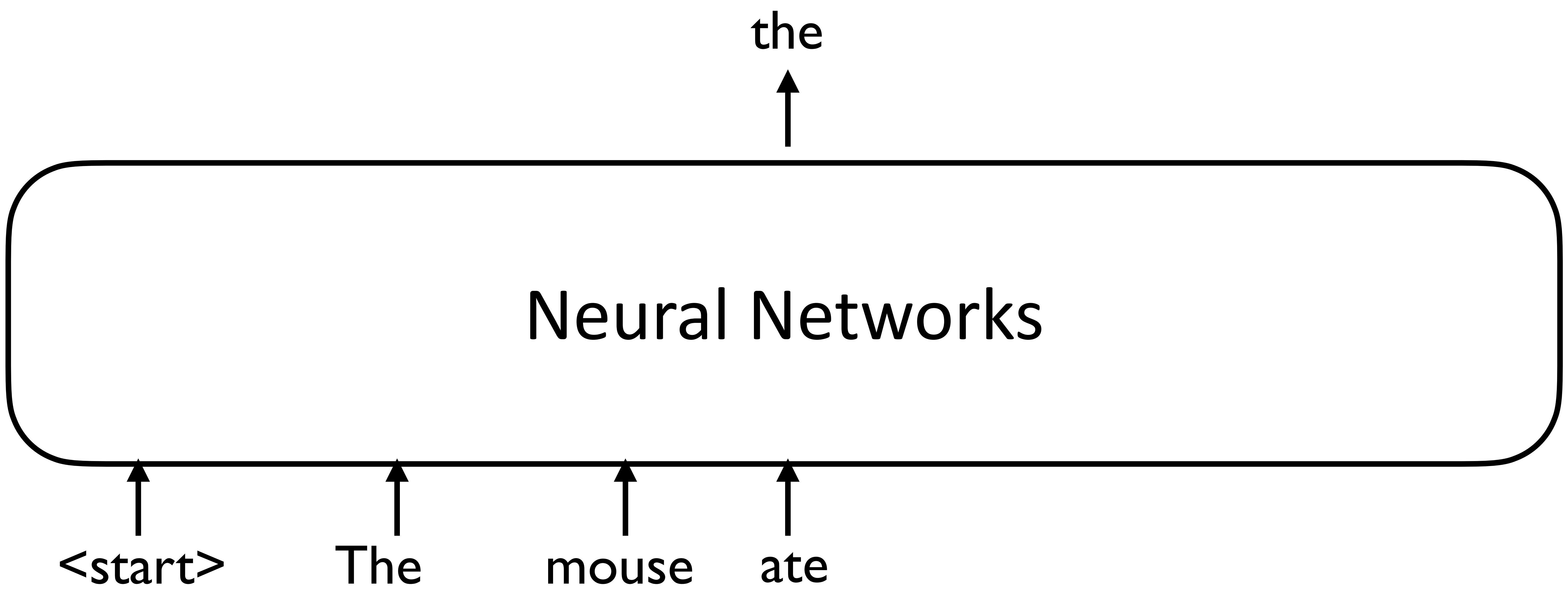
Data: "The mouse ate the cheese ."



Recap: Neural Language Models

Neural language models are typically autoregressive

Data: "The mouse ate the cheese ."

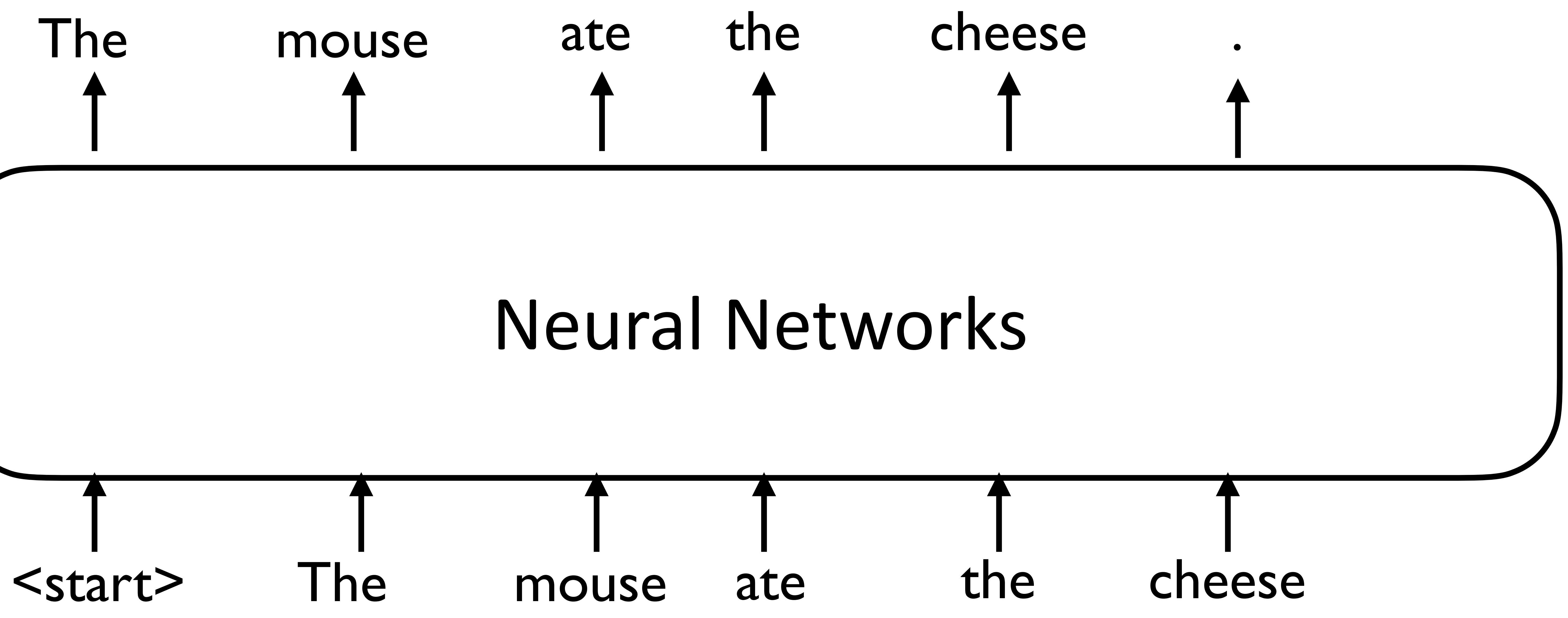


We can compute the loss on every token in parallel

Recap: Neural Language Models

Neural language models are typically autoregressive

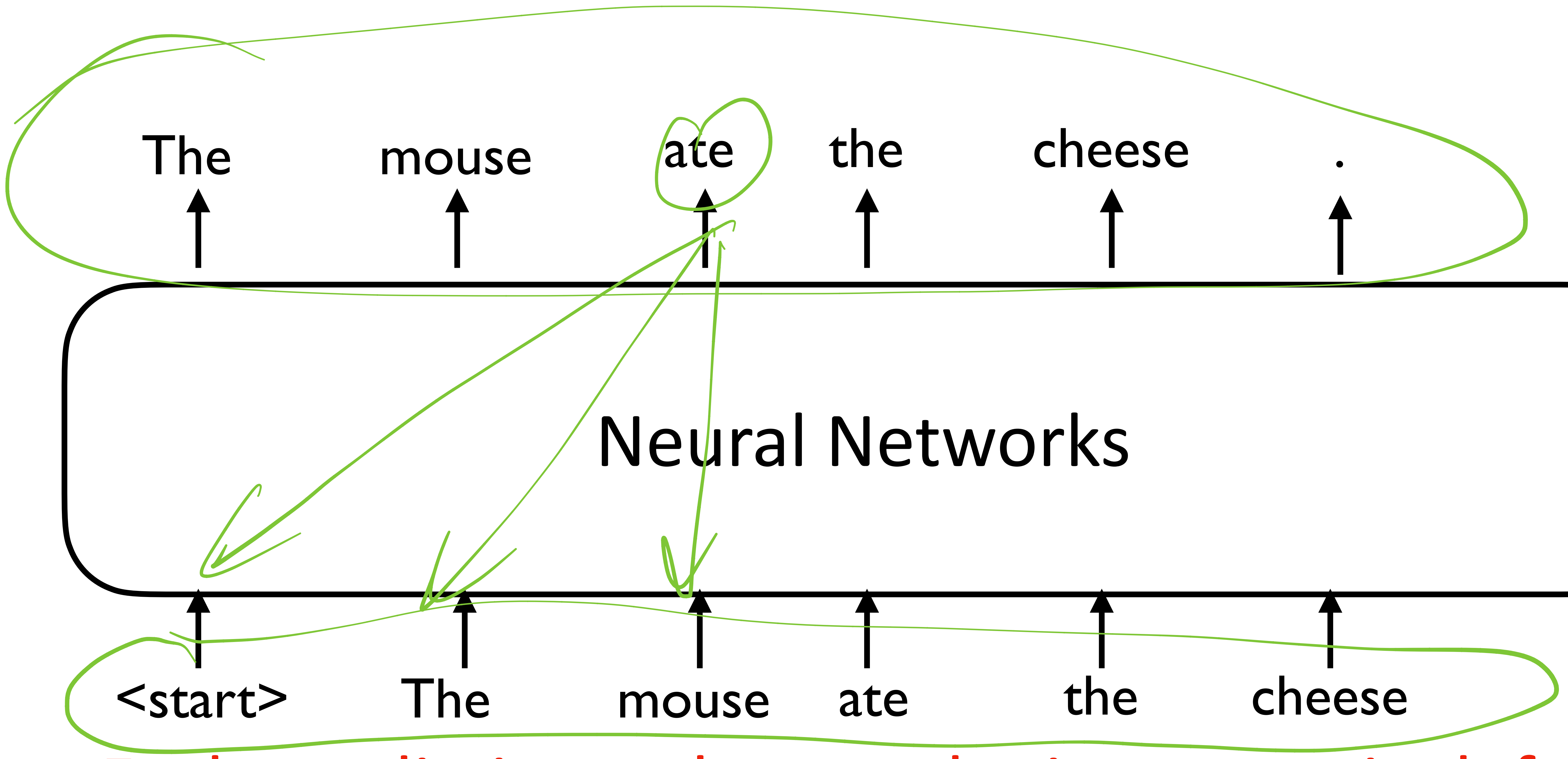
Data: "The mouse ate the cheese ."



Recap: Neural Language Models

Neural language models are typically autoregressive

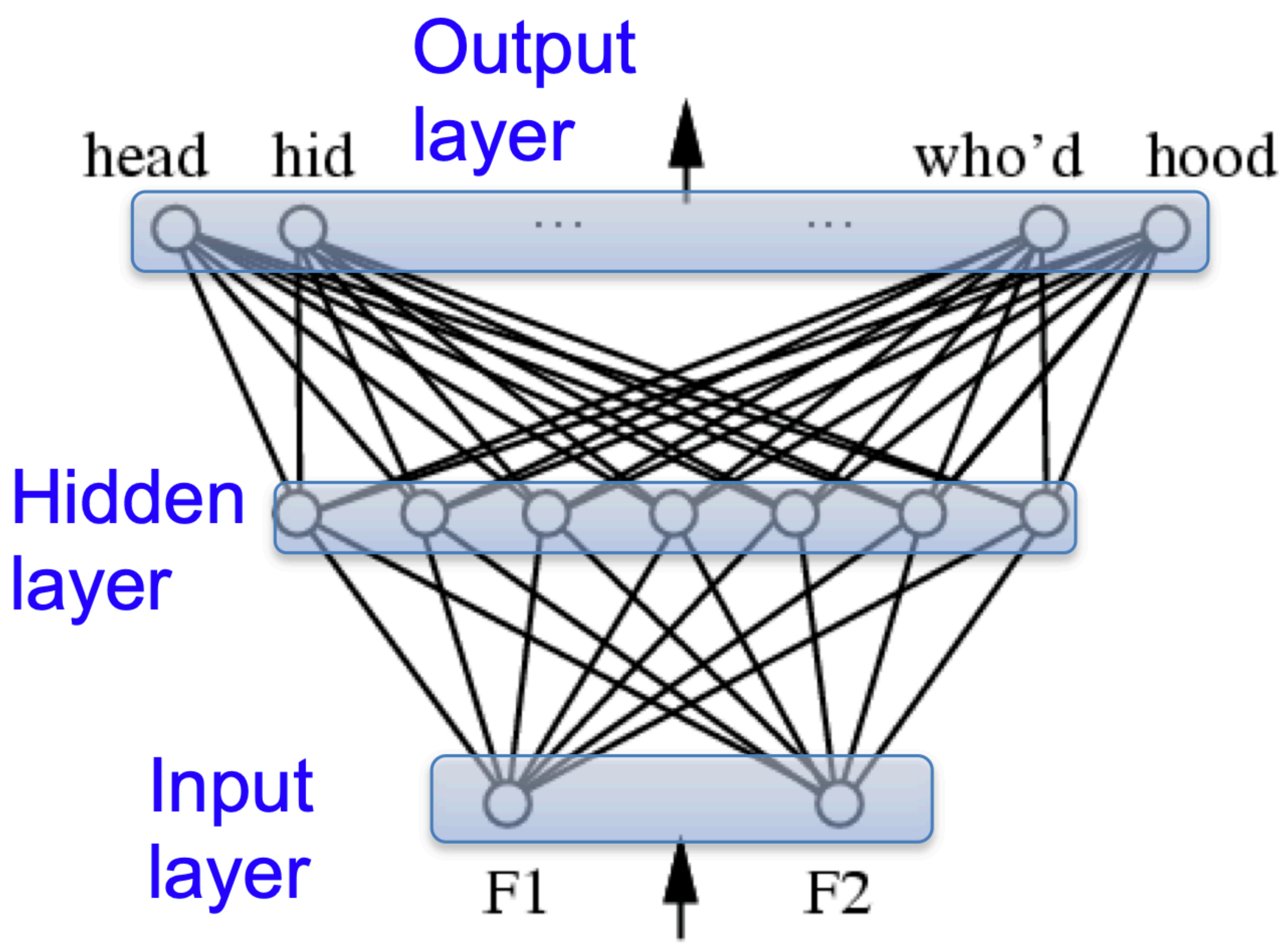
Data: "The mouse ate the cheese ."



Each prediction only sees the inputs on its left

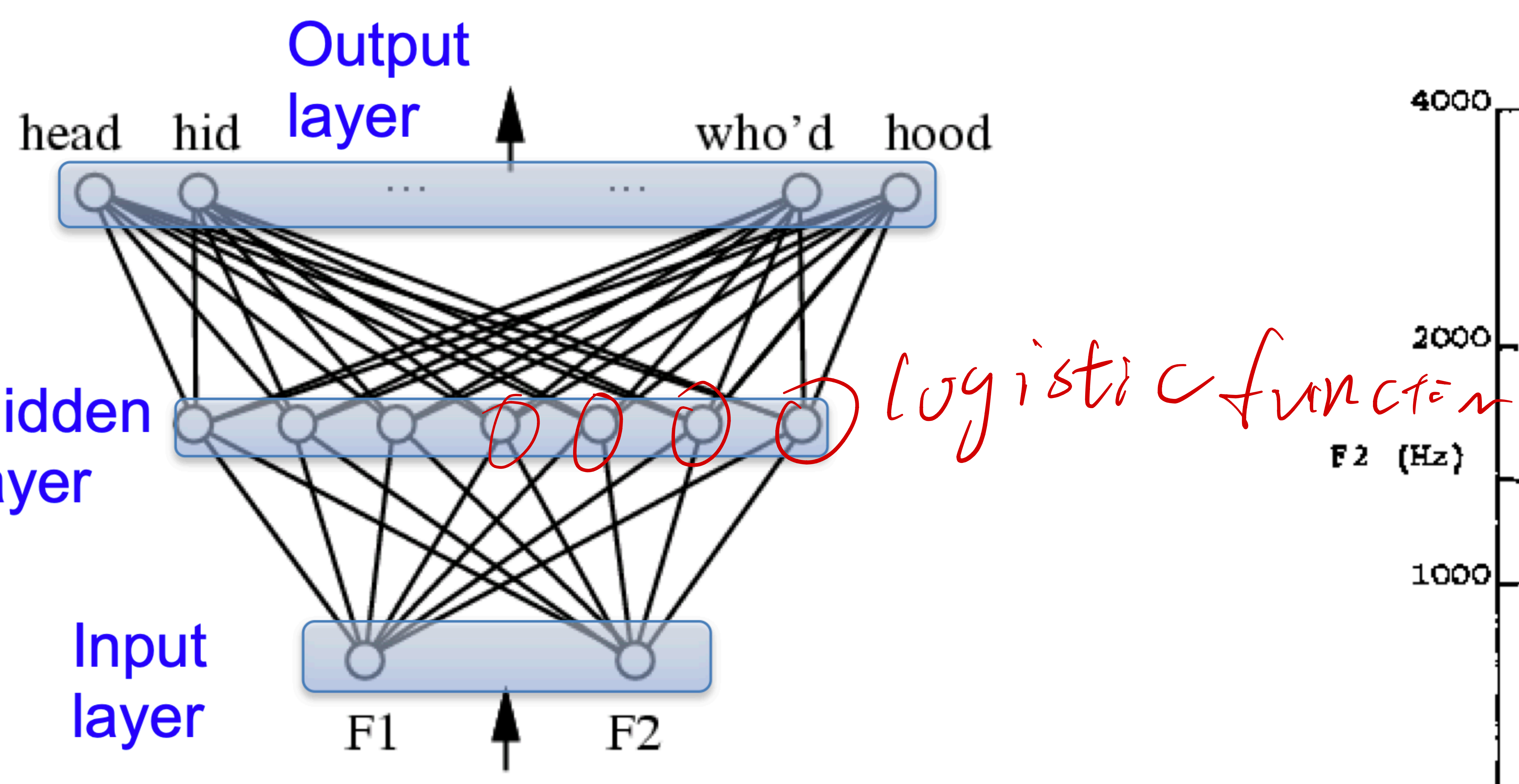
Recap: Multilayer Networks of Sigmoid Units

Recap: Multilayer Networks of Sigmoid Units

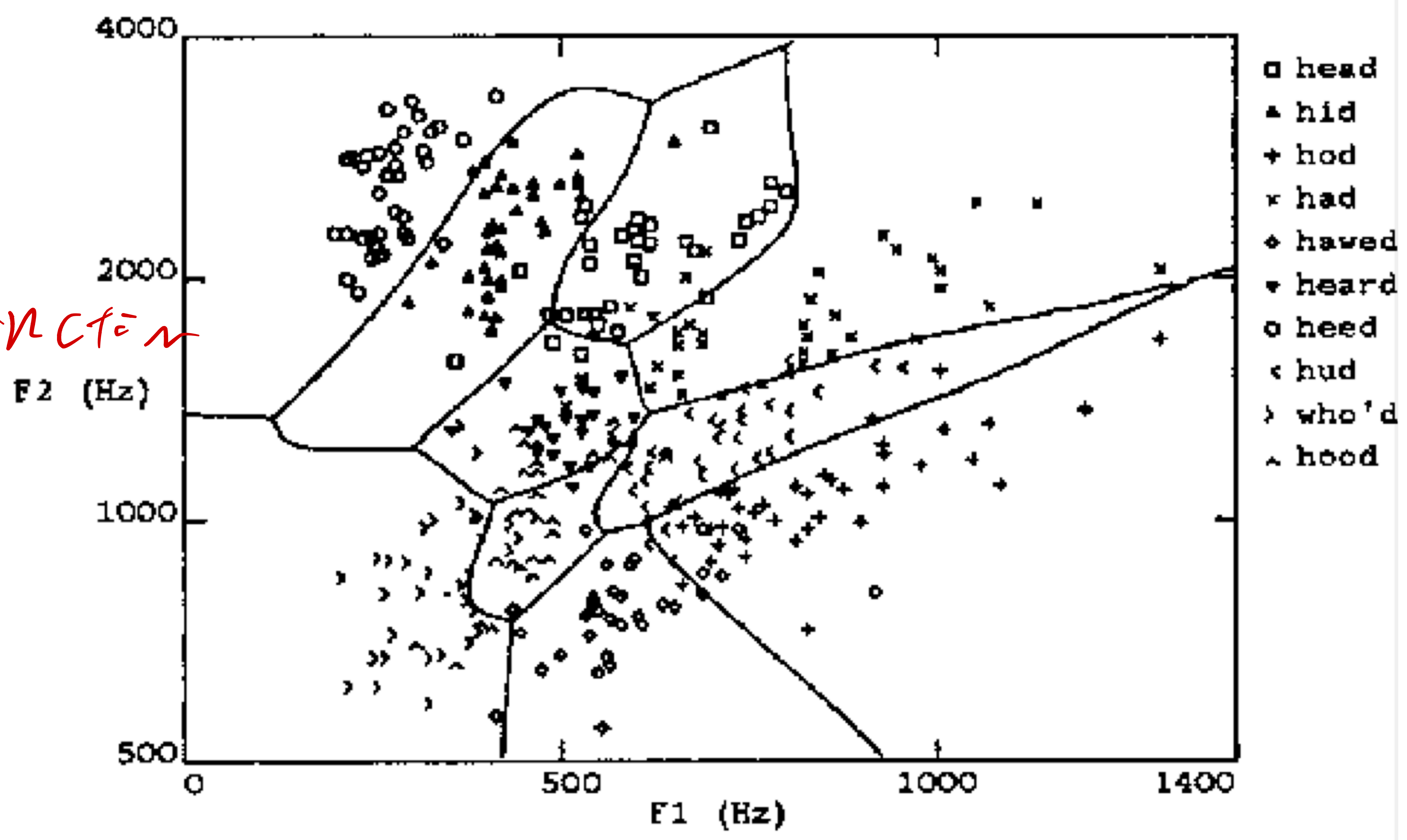


Two layers of logistic units

Recap: Multilayer Networks of Sigmoid Units



Two layers of logistic units



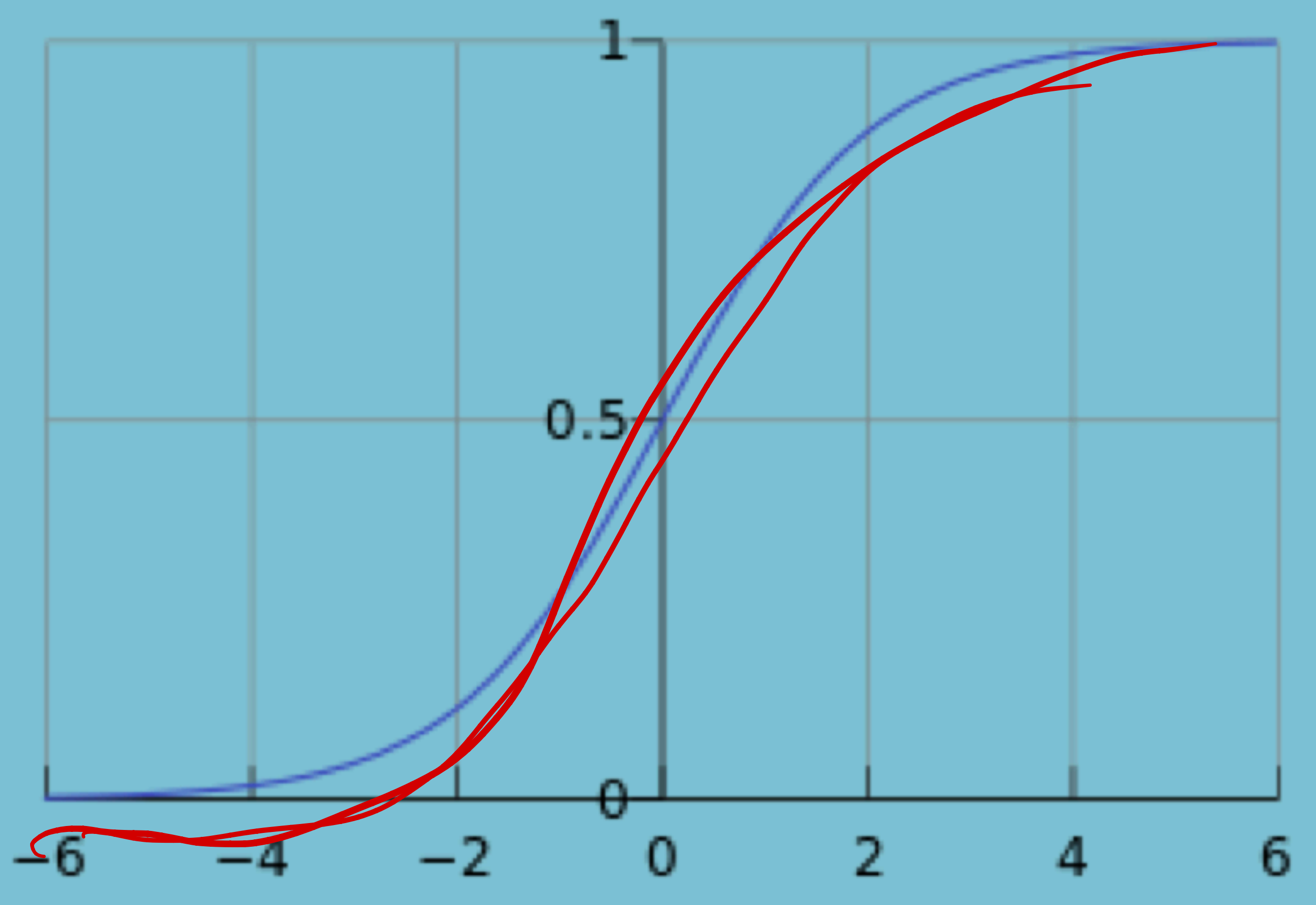
Highly non-linear decision surface

Activation Functions

x → Ax → BAx → CBAx

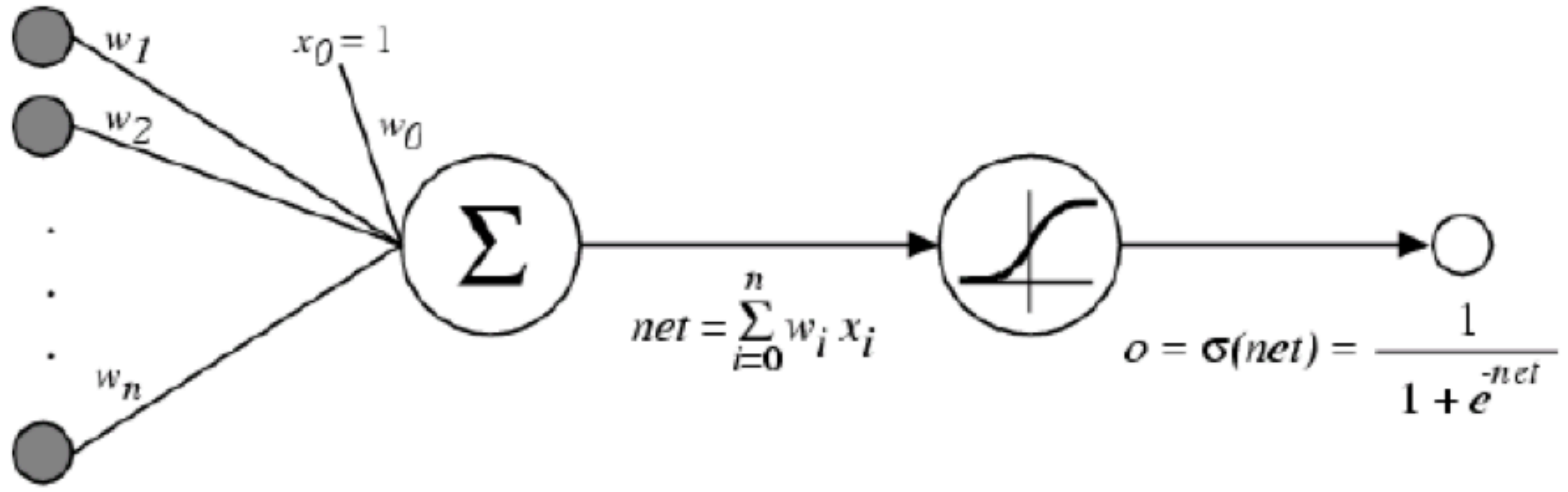
Sigmoid / Logistic Function

$$\text{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$



So far, we've assumed that the activation function (nonlinearity) is always the sigmoid function…

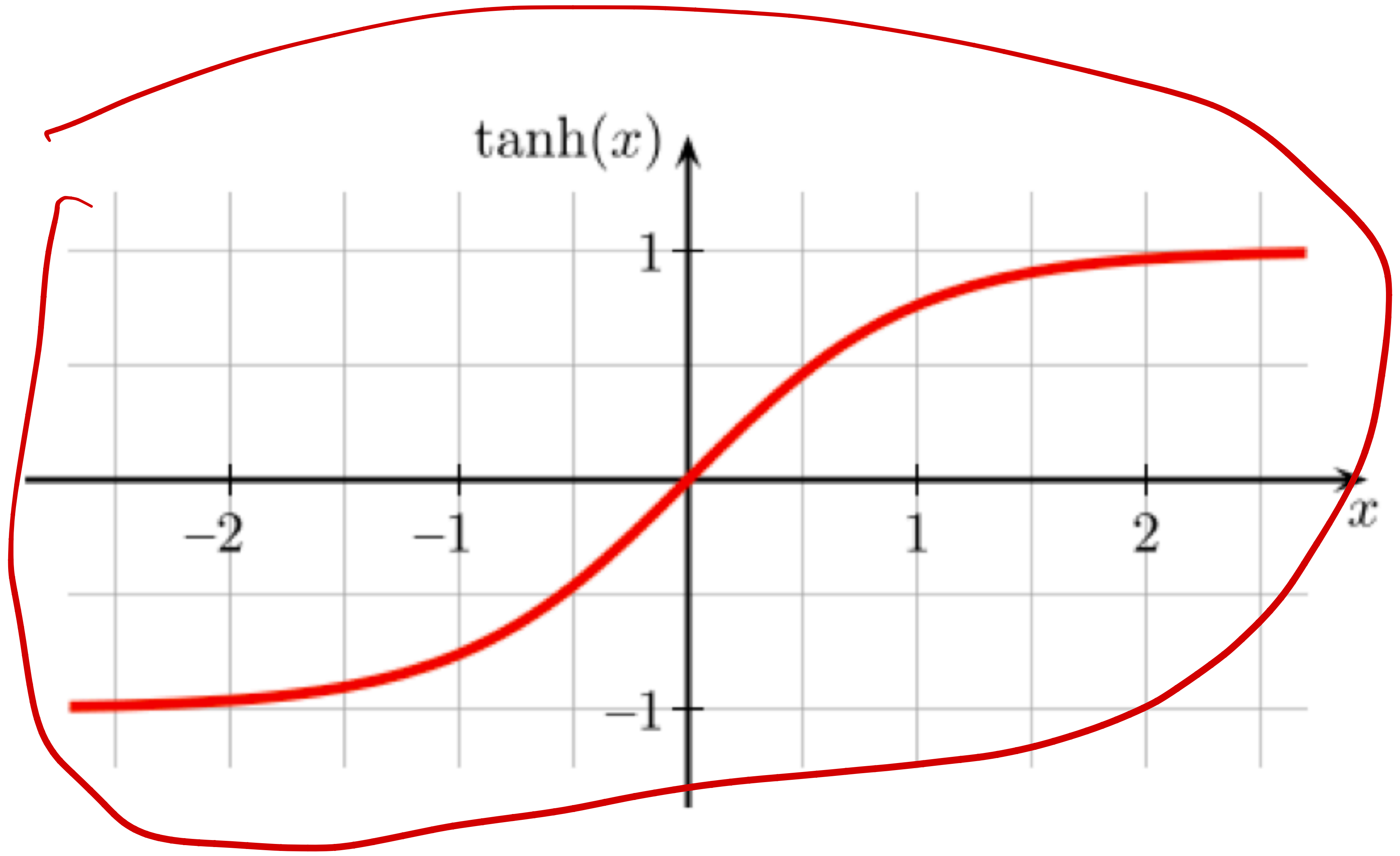
CBA
=
Mx



Tanh

- A new change: modifying the nonlinearity
  - The logistic is not widely used in modern ANNs

negative



Alternate 1: tanh

Like logistic function but shifted to range [-1, +1]

Activation Function

Understanding the difficulty of training deep feedforward neural networks

AI Stats 2010

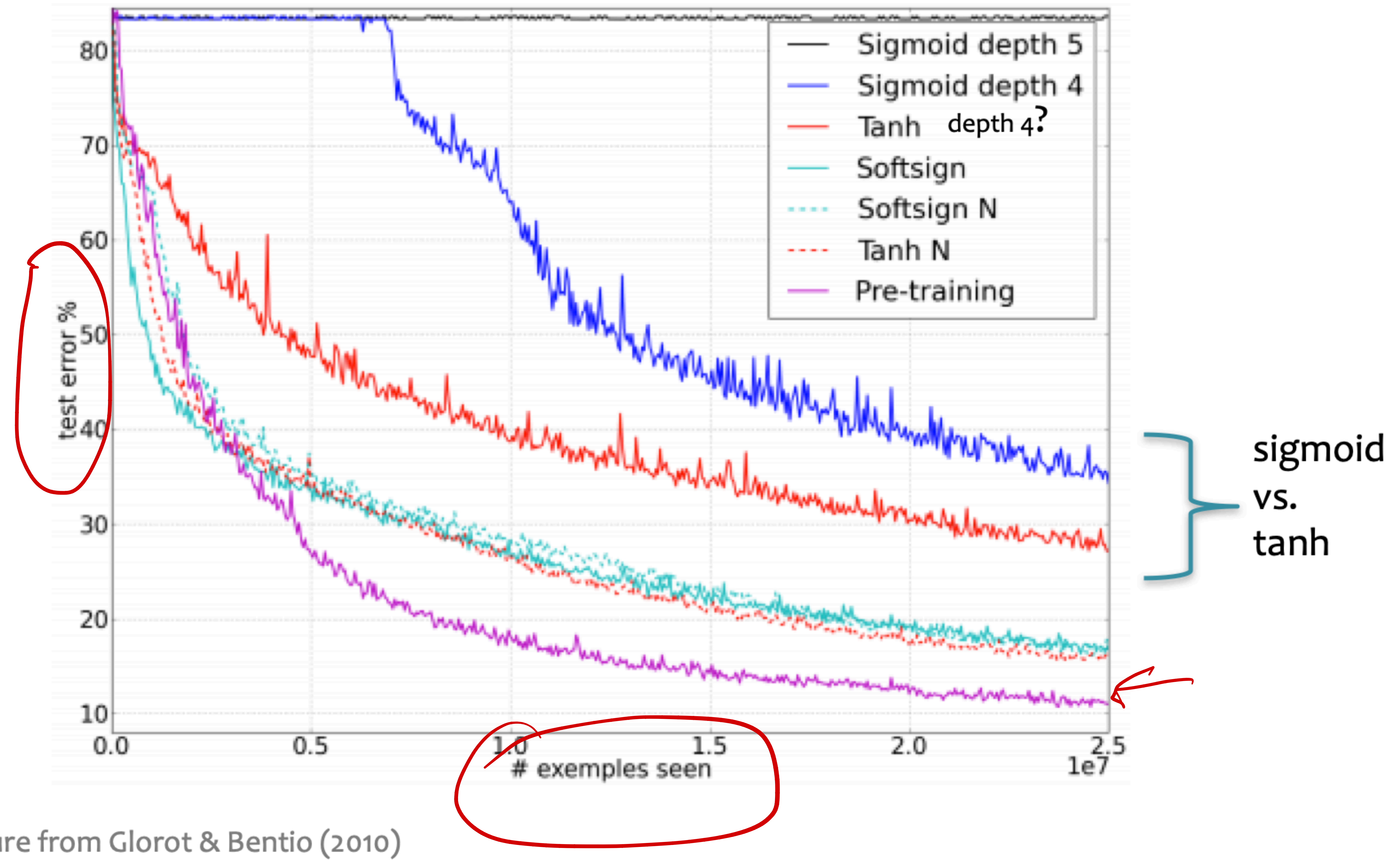
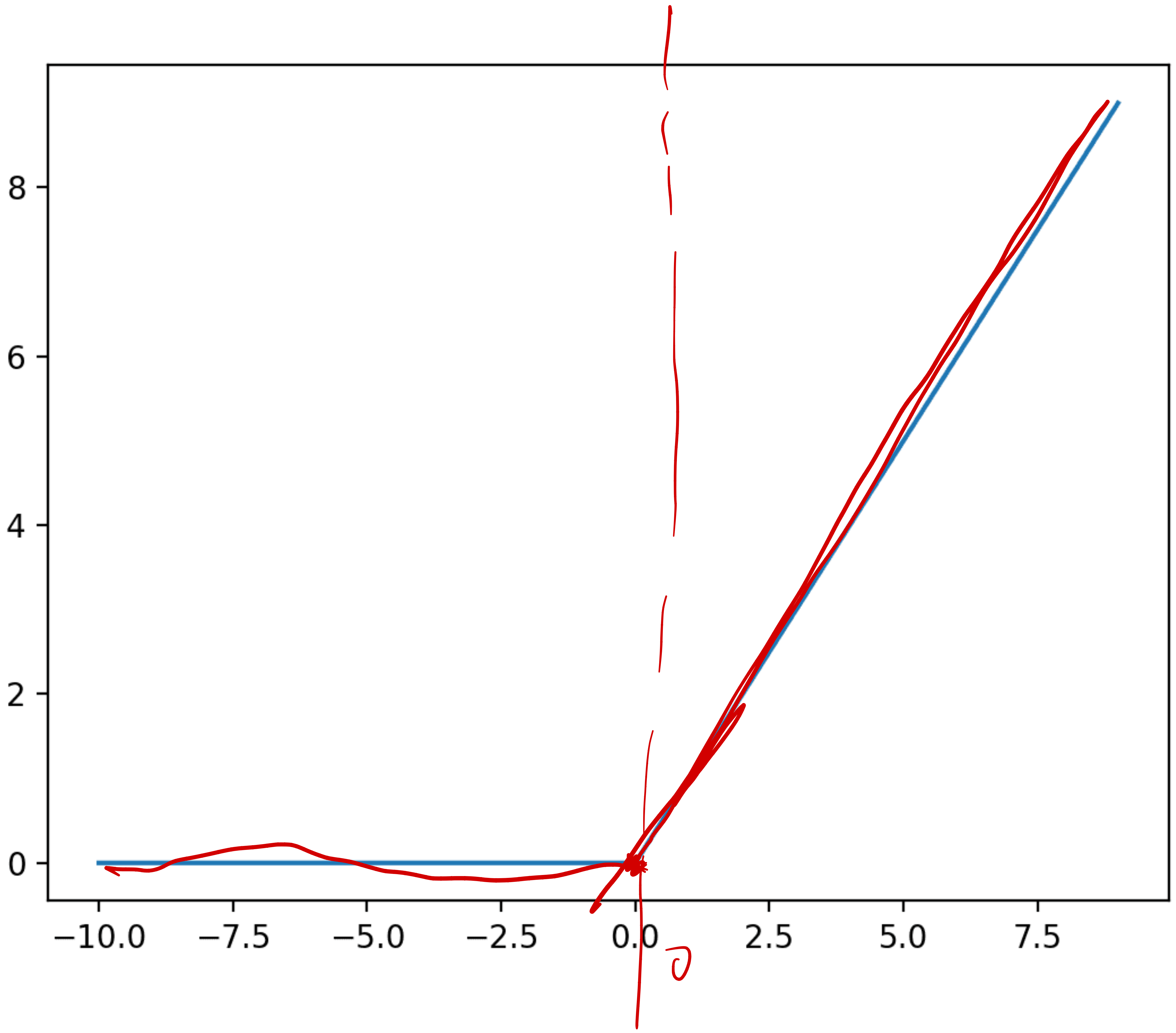


Figure from Glorot & Bentio (2010)

ReLU

y = max(0, x)

Relu(x) = max(0, x)



Other Activation Functions

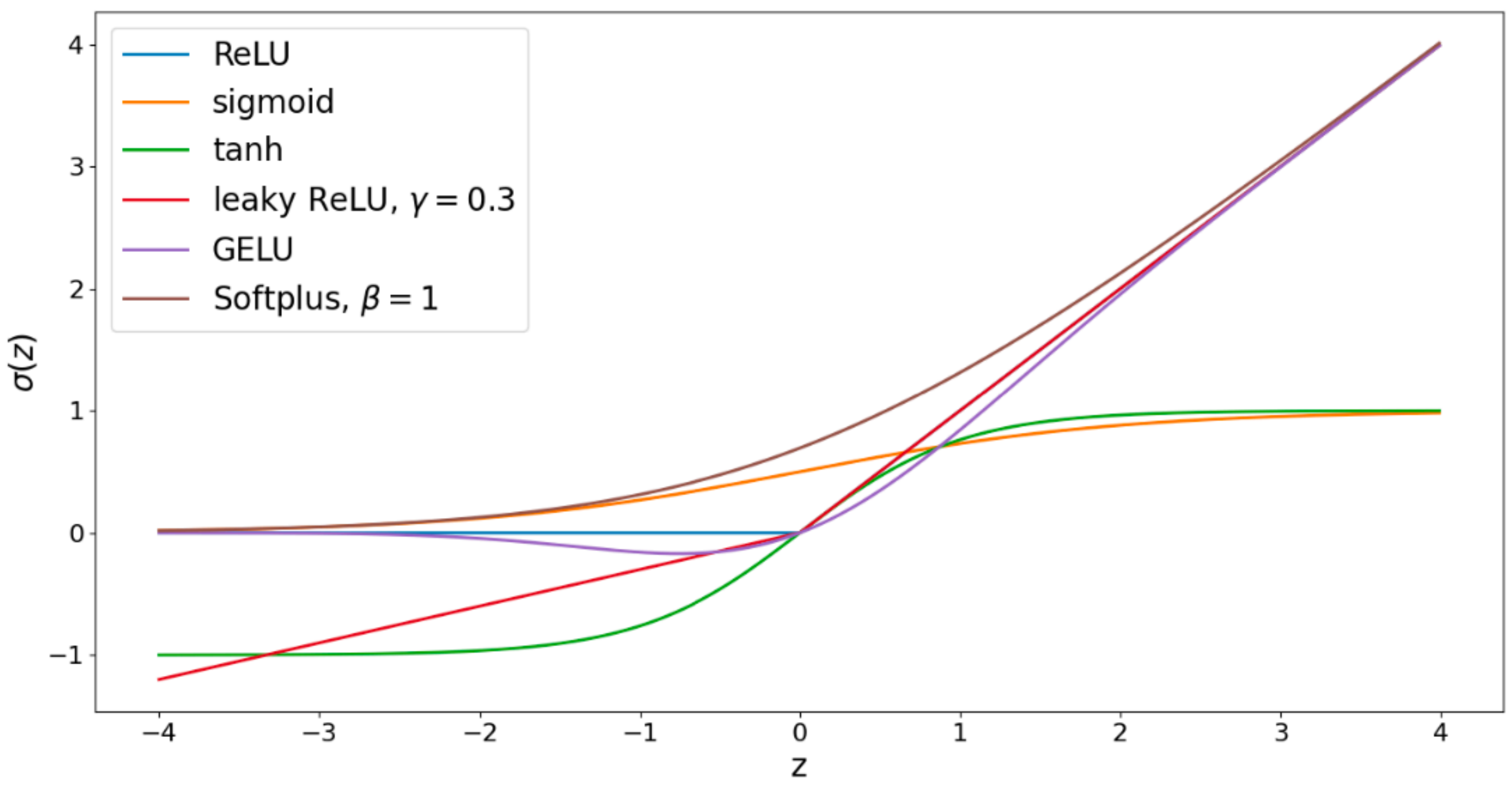
$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{(sigmoid)}$$

$$\sigma(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \qquad \text{(tanh)}$$

$$\sigma(z) = \max\{z, \gamma z\}, \gamma \in (0,1) \qquad \text{(leaky ReLU)}$$

$$\sigma(z) = \frac{z}{2}\left[1+\operatorname{erf}\left(\frac{z}{\sqrt{2}}\right)\right] \qquad \text{(GELU)}$$

$$\sigma(z) = \frac{1}{\beta}\log(1+\exp(\beta z)), \beta > 0 \qquad \text{(Softplus)}$$



Multilayer Perceptron Neural Networks (MLP)

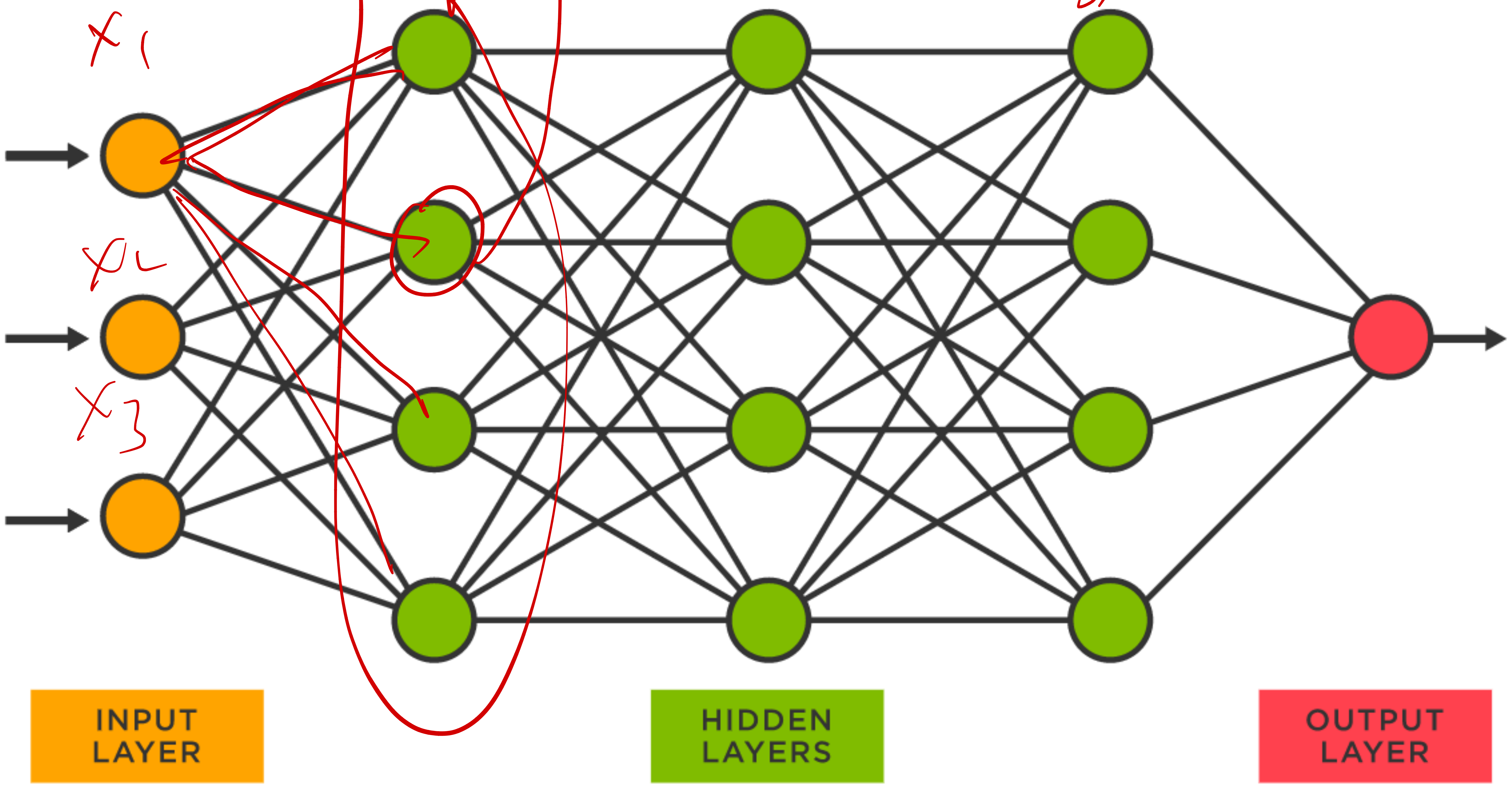
$\vec{x} = (x_1, x_2, x_3)$

$W\vec{x}$

$\sigma(w_1 x_1 + w_2 x_2 + w_3 x_3)$

$\sigma(w_1' x_1 + w_2' x_2 + w_3' x_3)$

Fully connected



Residual Connection

We want deeper and deeper NNs, but going deep is difficult

Residual Connection

We want deeper and deeper NNs, but going deep is difficult

- Troubles accumulate w/ more layers
- Signals get distorted when propagated
- in forward and backward passes

Residual Connection

We want deeper and deeper NNs, but going deep is difficult

- Troubles accumulate w/ more layers
- Signals get distorted when propagated
- in forward and backward passes

Commonly used techniques to train "Deep" NNs:

Weight initialization

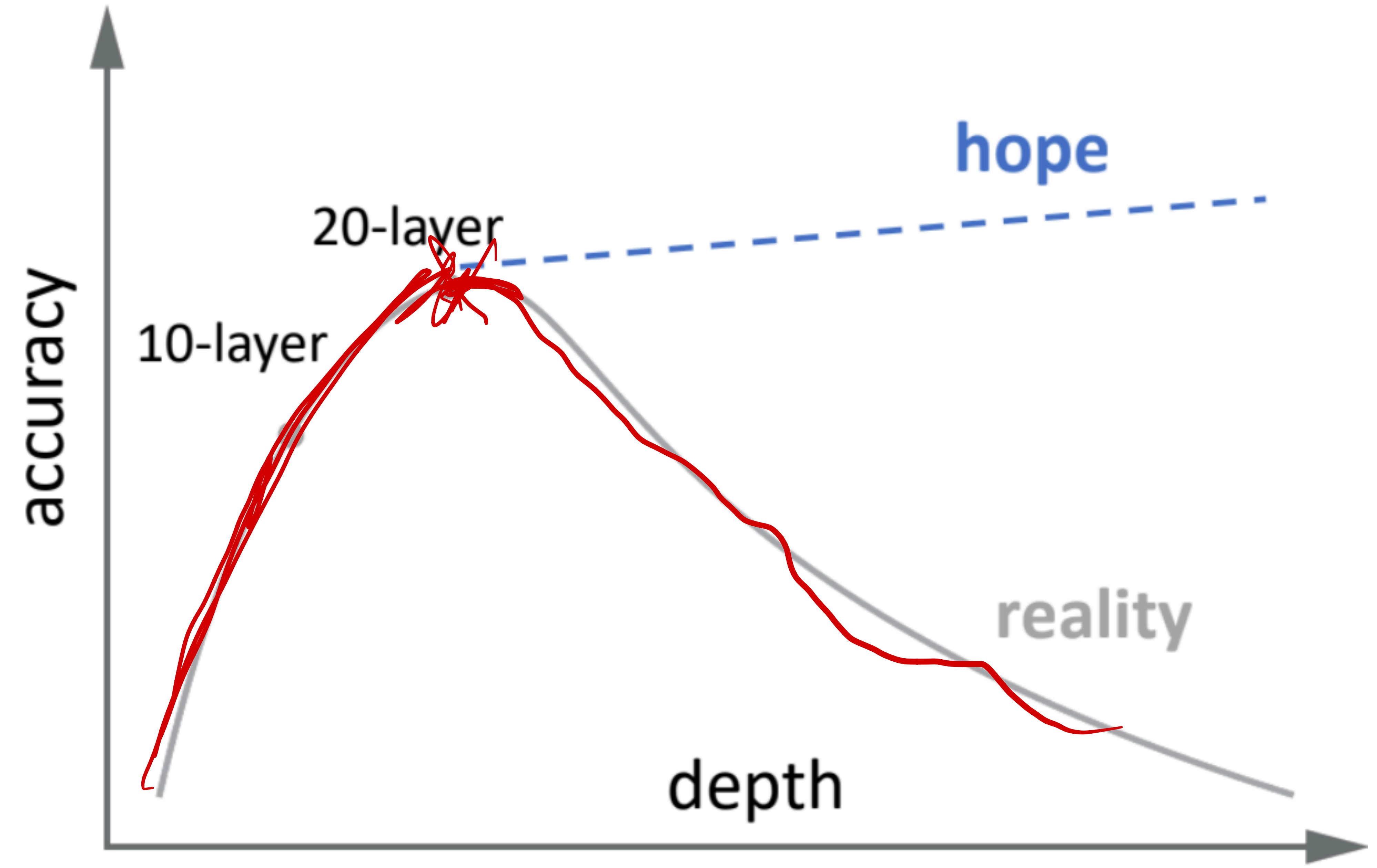
Normalization modules

Deep residual learning

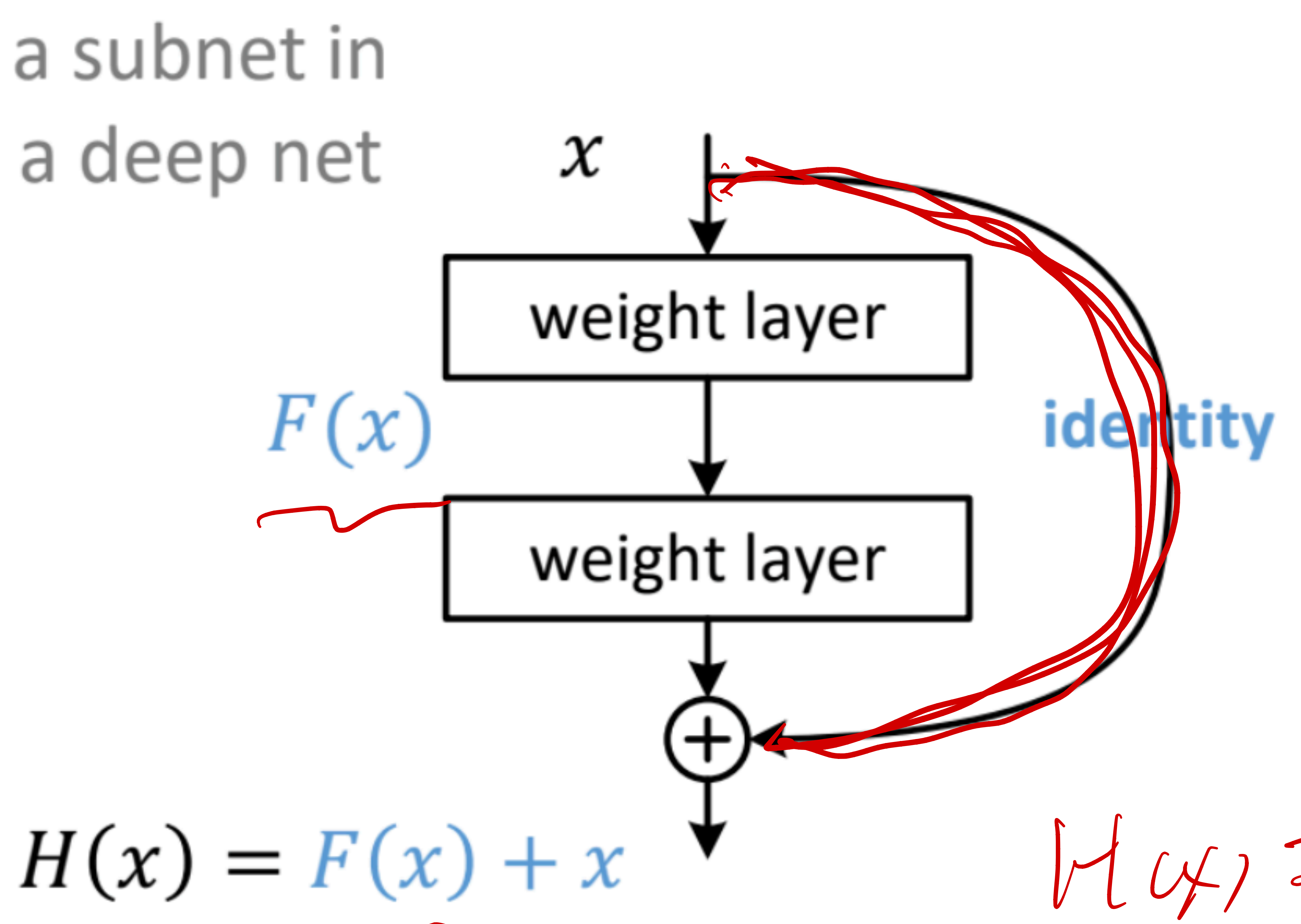
The Degradation Problem

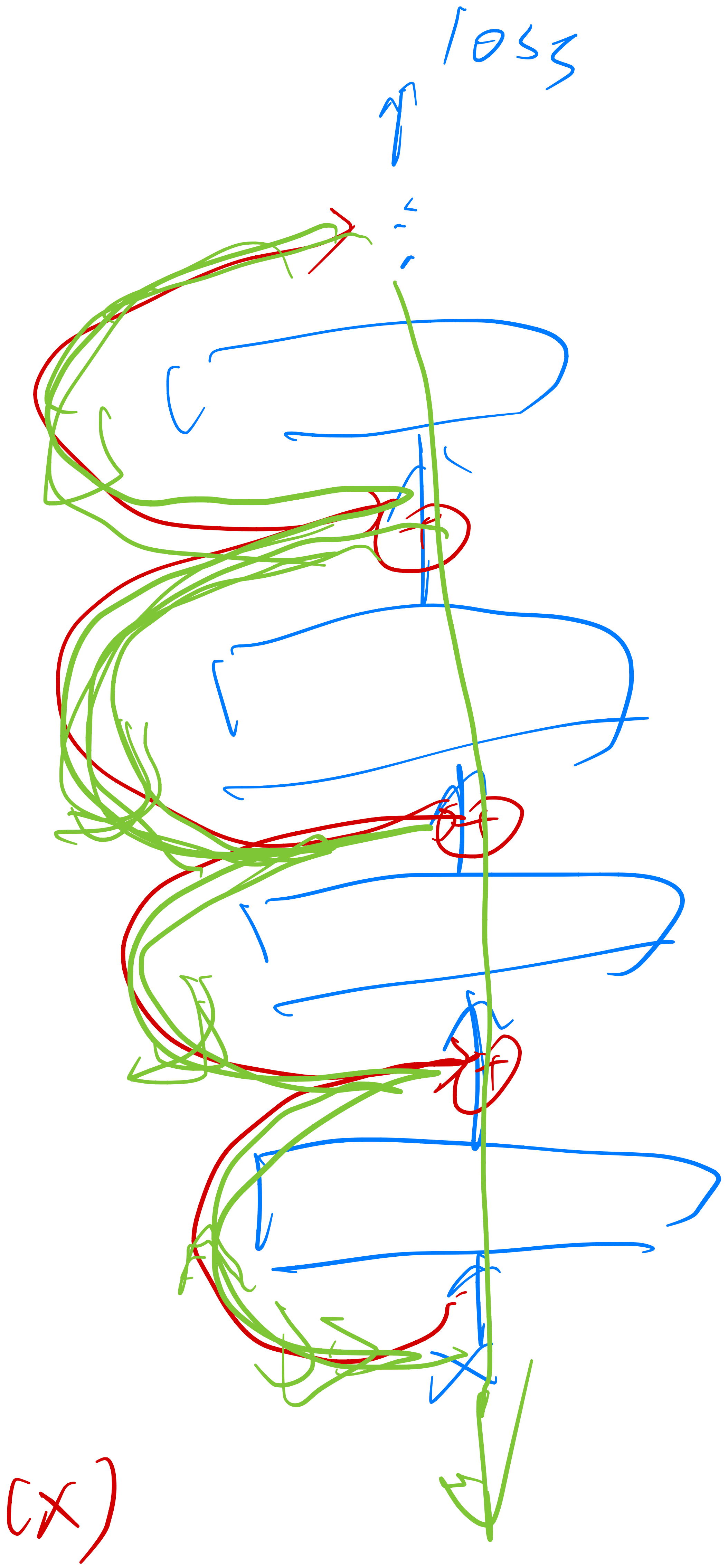
- Good init + norm enable training deeper models
- Simply stacking more layers?

- Degrade after ~20 layers
- Not overfitting
- Difficult to train



Deep Residual Learning

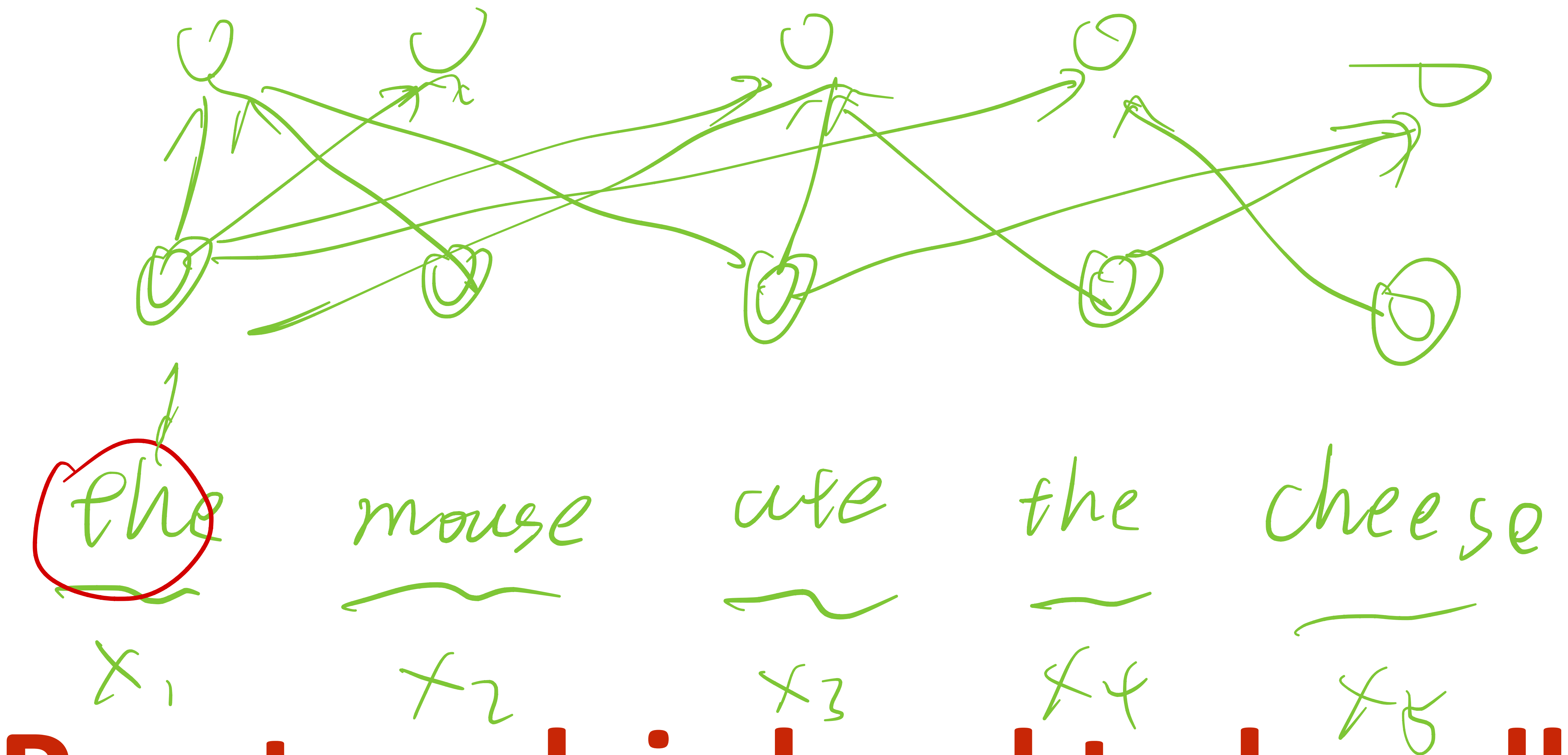




$H(x) = F(x)$

$H(x) = F(x) + x$

ResNet

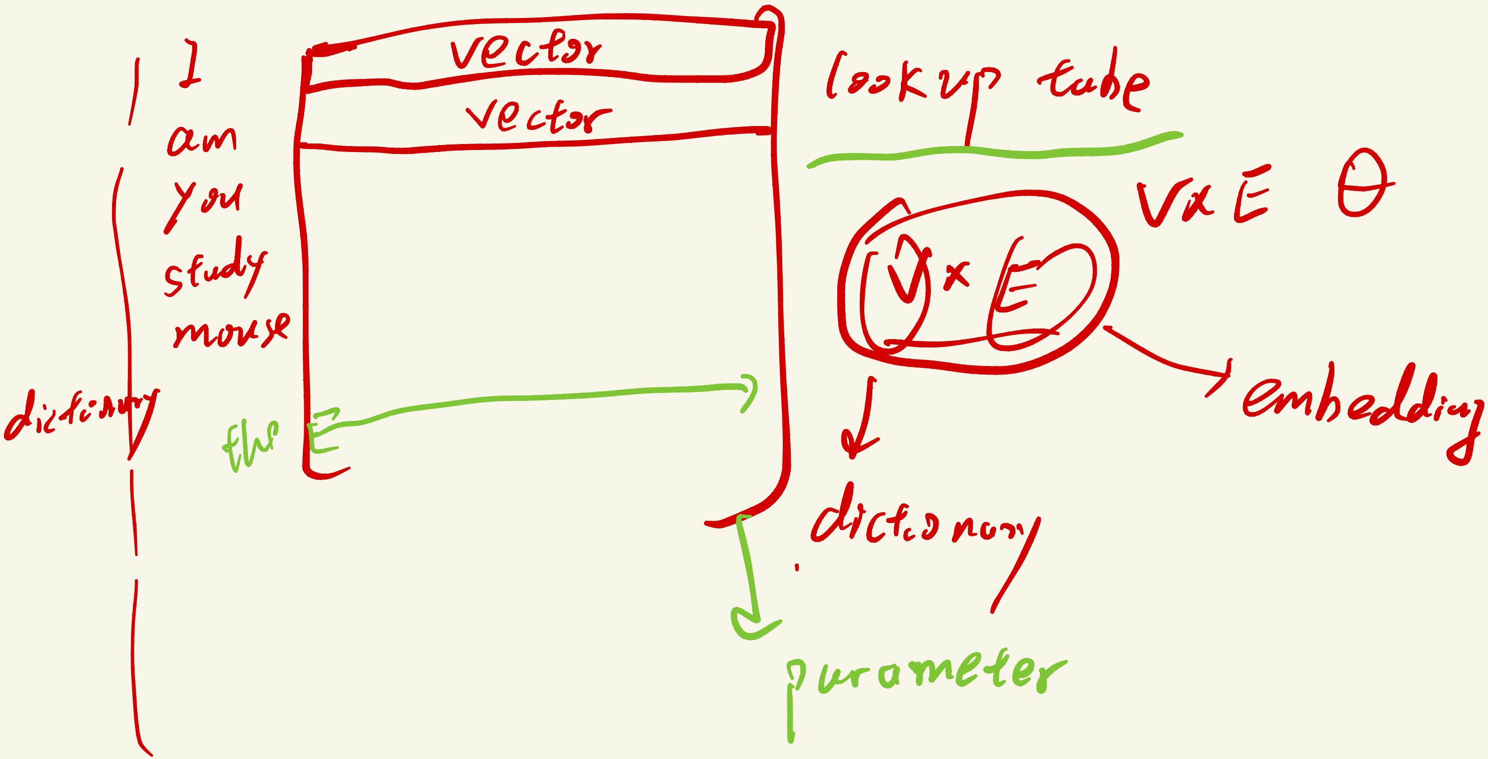


MLP network is hard to handle sequence data with varying length

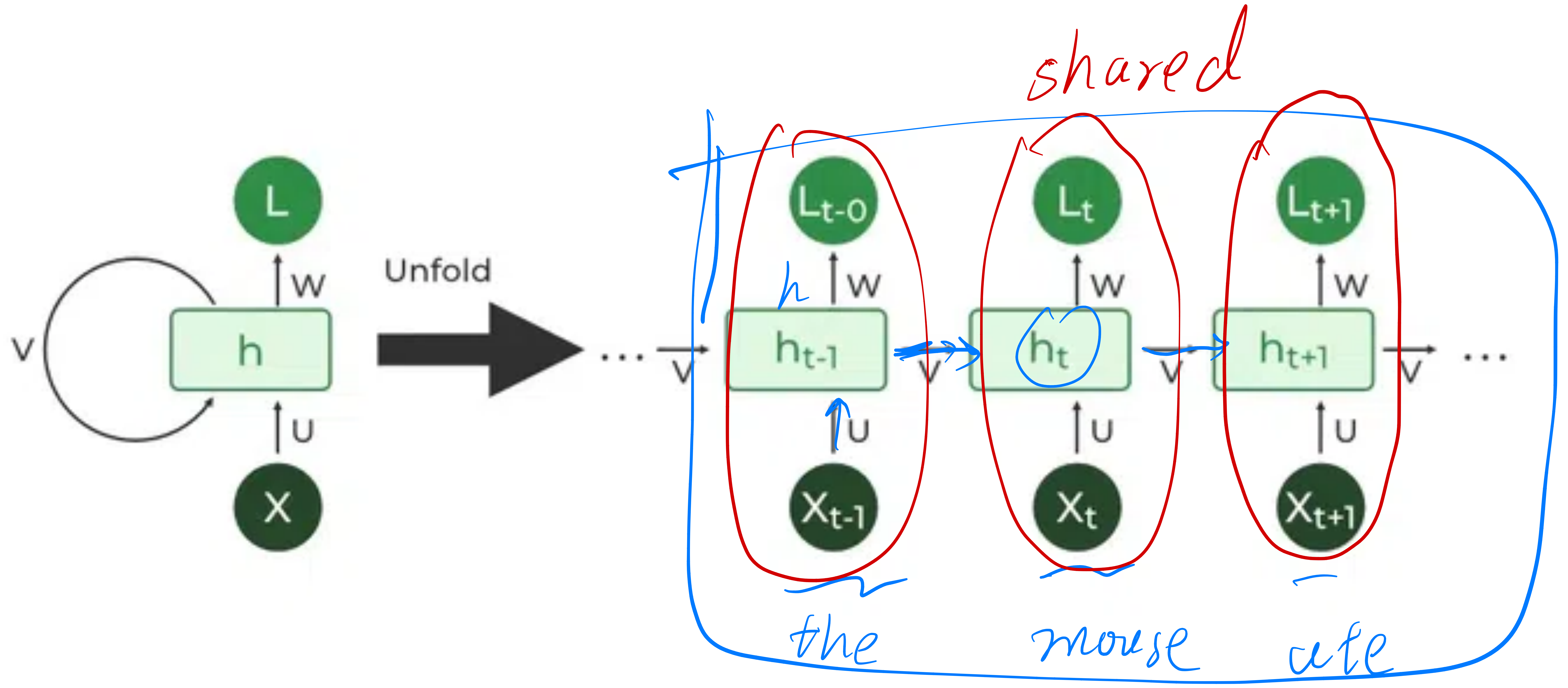
word emb

parameter size grows when sequence → longer

word embedding:



Recurrent Neural Networks (RNNs)

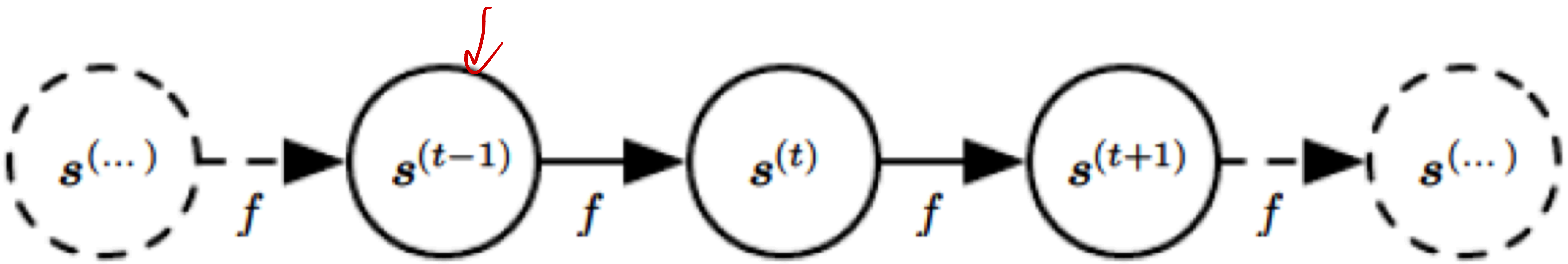


Recurrent Neural Networks

- Dates back to (Rumelhart *et al.*, 1986)
- A family of neural networks for handling sequential data, which involves variable length inputs or outputs

- Especially, for natural language processing (NLP)

Computation Graph



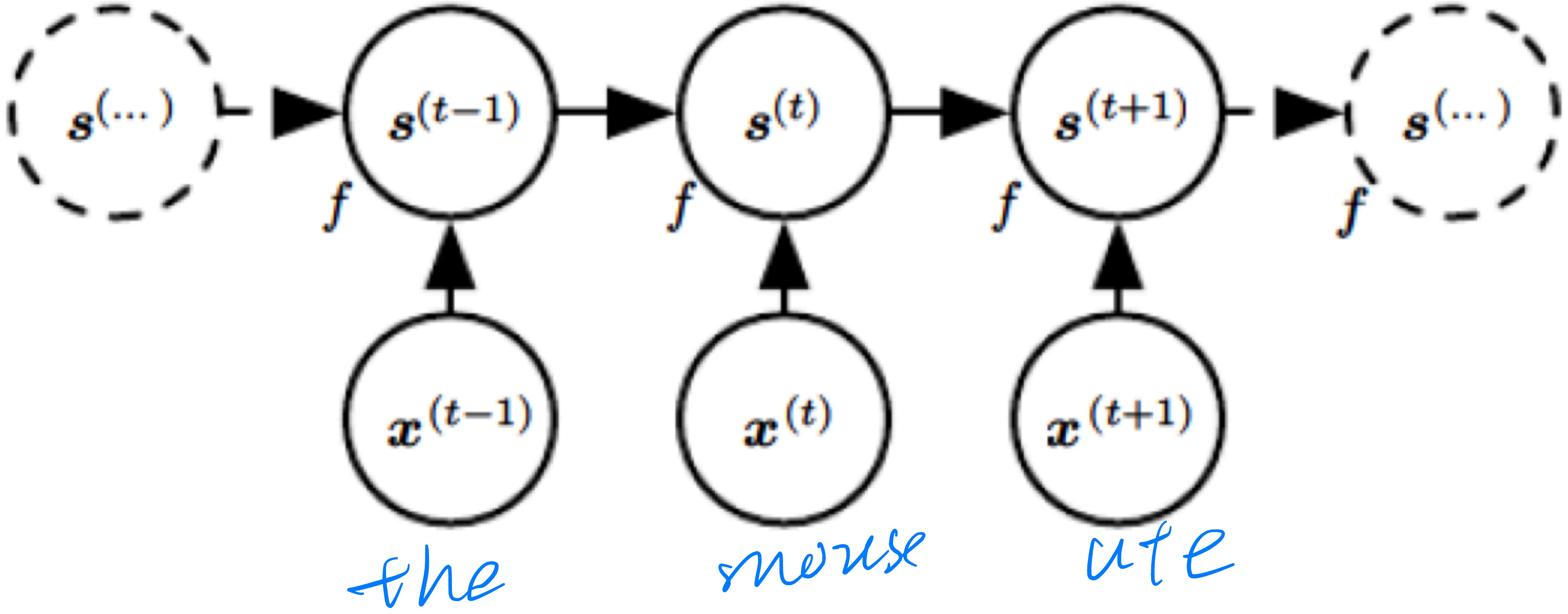
$$s^{(t+1)} = f(s^{(t)}; \theta)$$

next timestep = next word

θ parameter

Figure from *Deep Learning*,
Goodfellow, Bengio and Courville

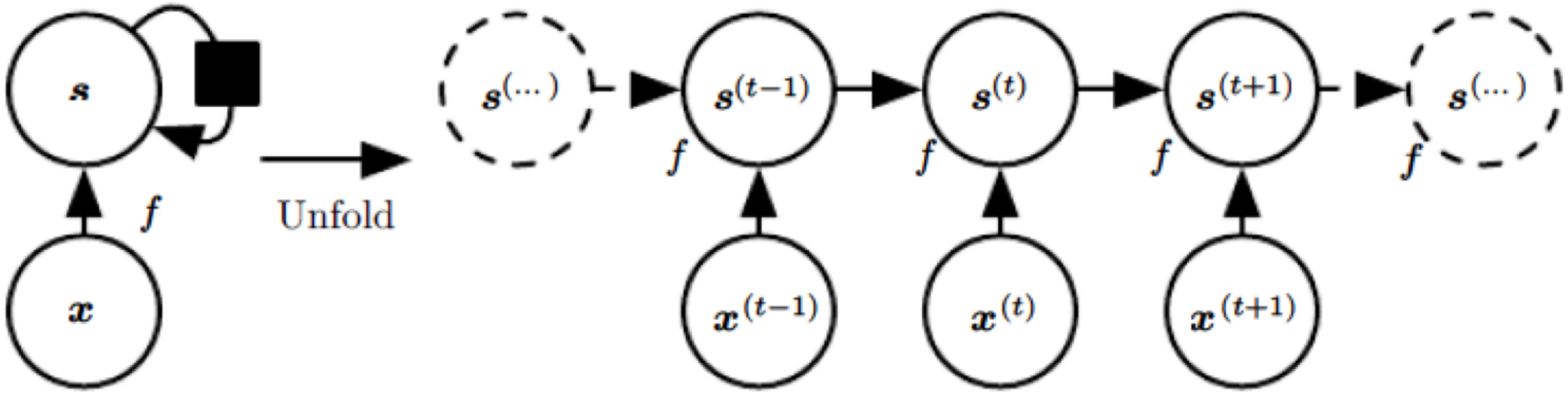
Computation Graph



$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

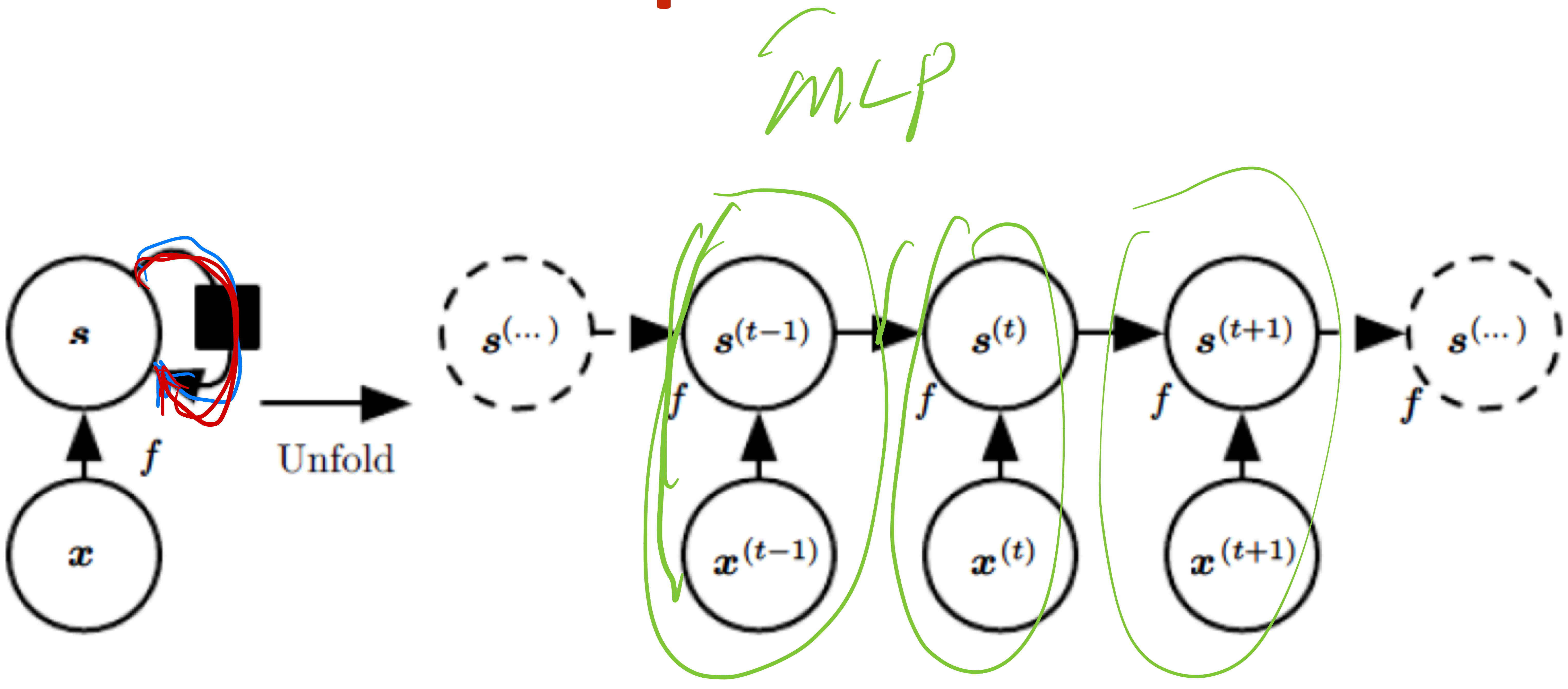
MLP

Compact view



$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Compact view

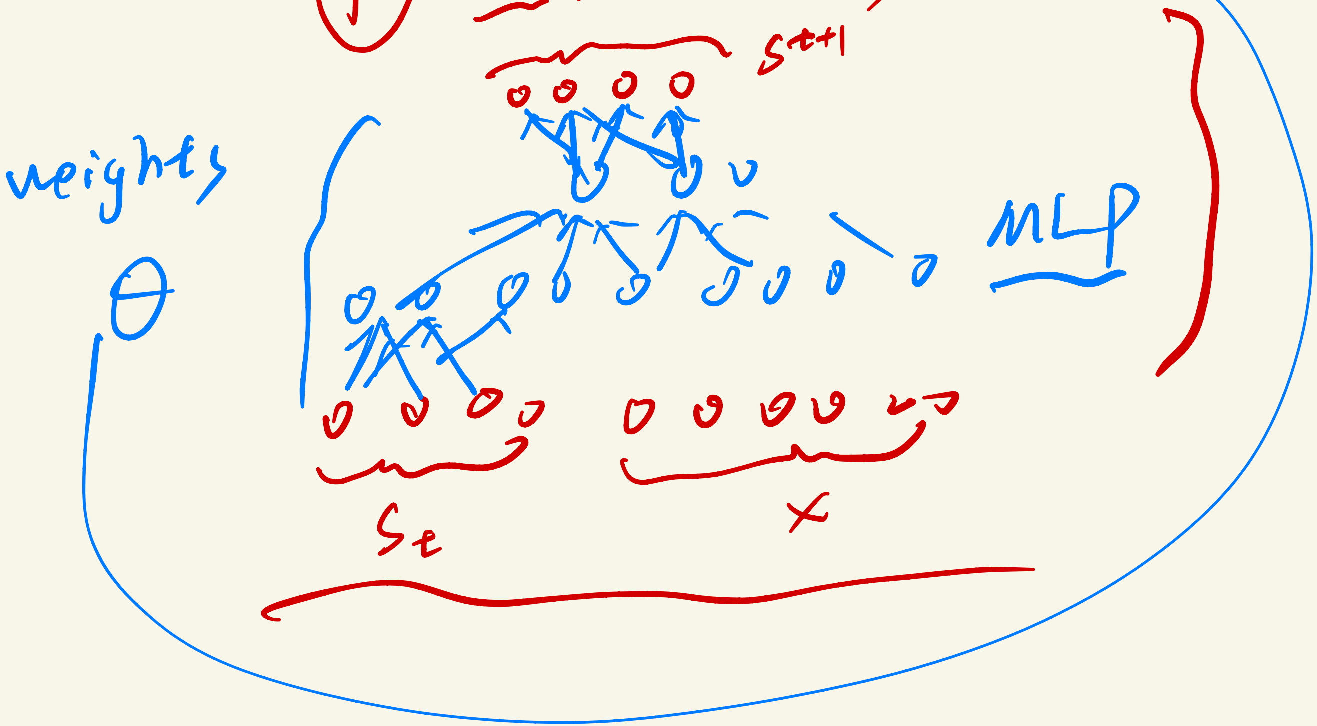


$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Key: the same f and θ for all time steps

parameter size doesn't grow

$$S^{t+1} = f(S^t, X^t; \theta)$$



Recurrent Neural Networks

Recurrent Neural Networks

- Use the same computational function and parameters across different time steps of the sequence

Recurrent Neural Networks

- Use the same computational function and parameters across different time steps of the sequence

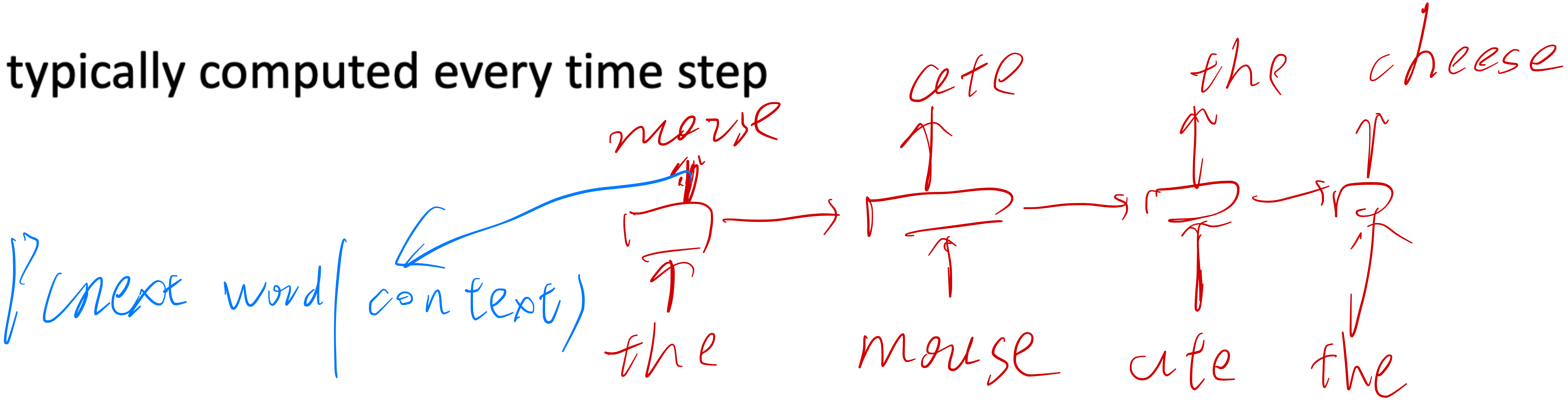
- Each time step: takes the input entry and the previous hidden state to compute the output entry

Recurrent Neural Networks

- Use the same computational function and parameters across different time steps of the sequence

- Each time step: takes the input entry and the previous hidden state to compute the output entry

- Loss: typically computed every time step



Recurrent Neural Networks

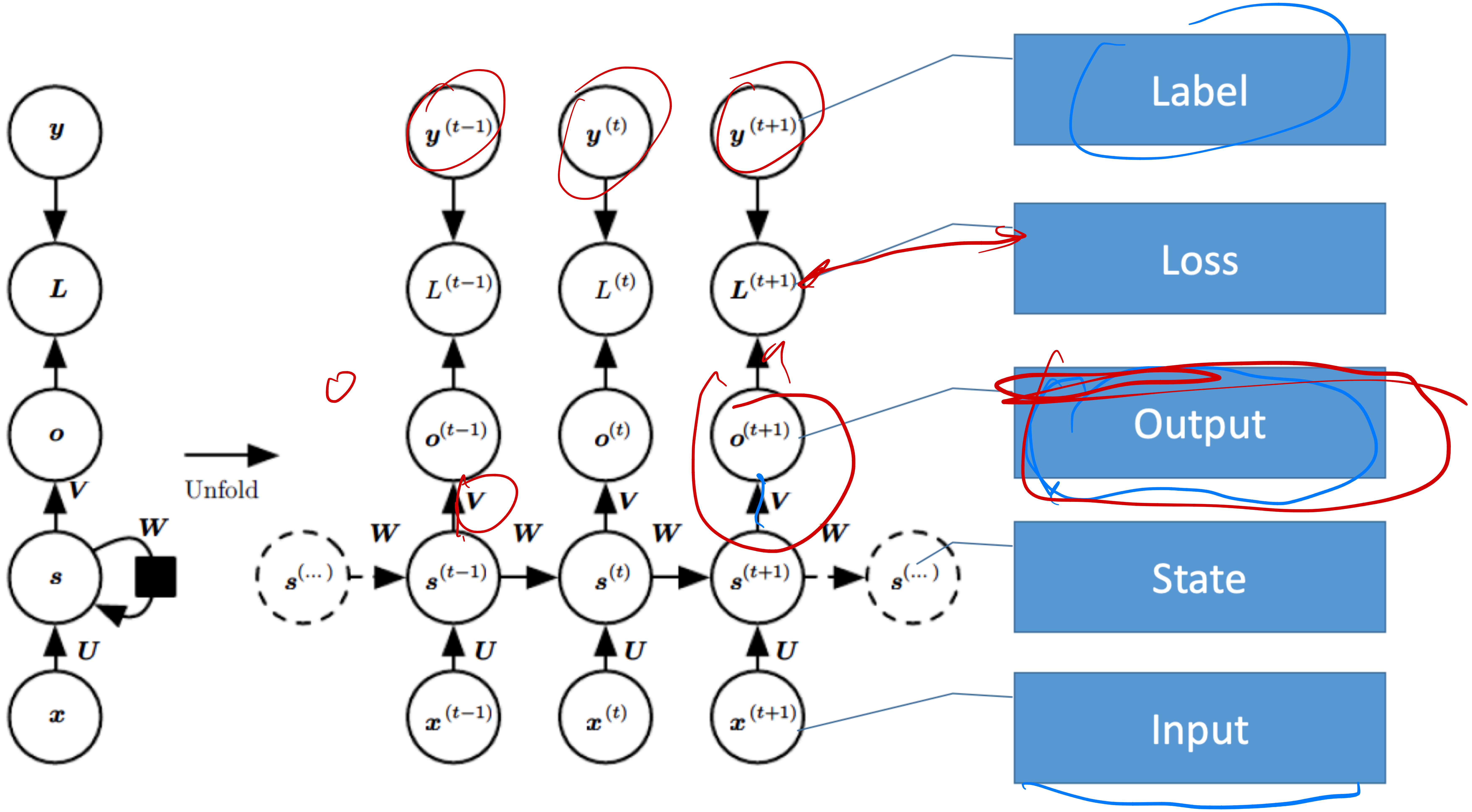


Figure from *Deep Learning*, by Goodfellow, Bengio and Courville

$\vec{s}$ dim 1024   PC

$\vec{s}$ dim [0 0 0 0]

1024

mouse

P(anyword | the)

softmax([V]·$\vec{s}$)

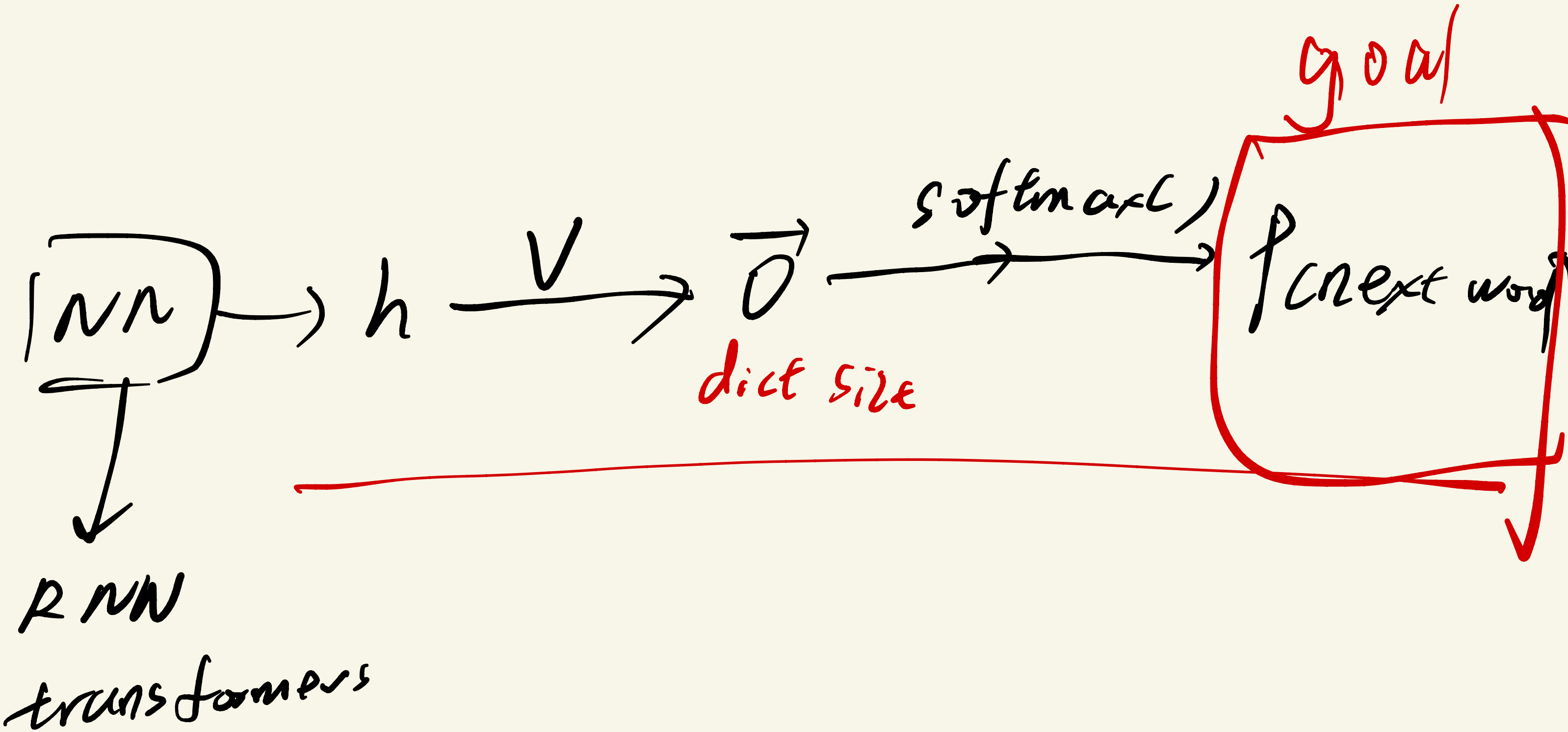
[0, 1]

[V]·$\vec{s}$

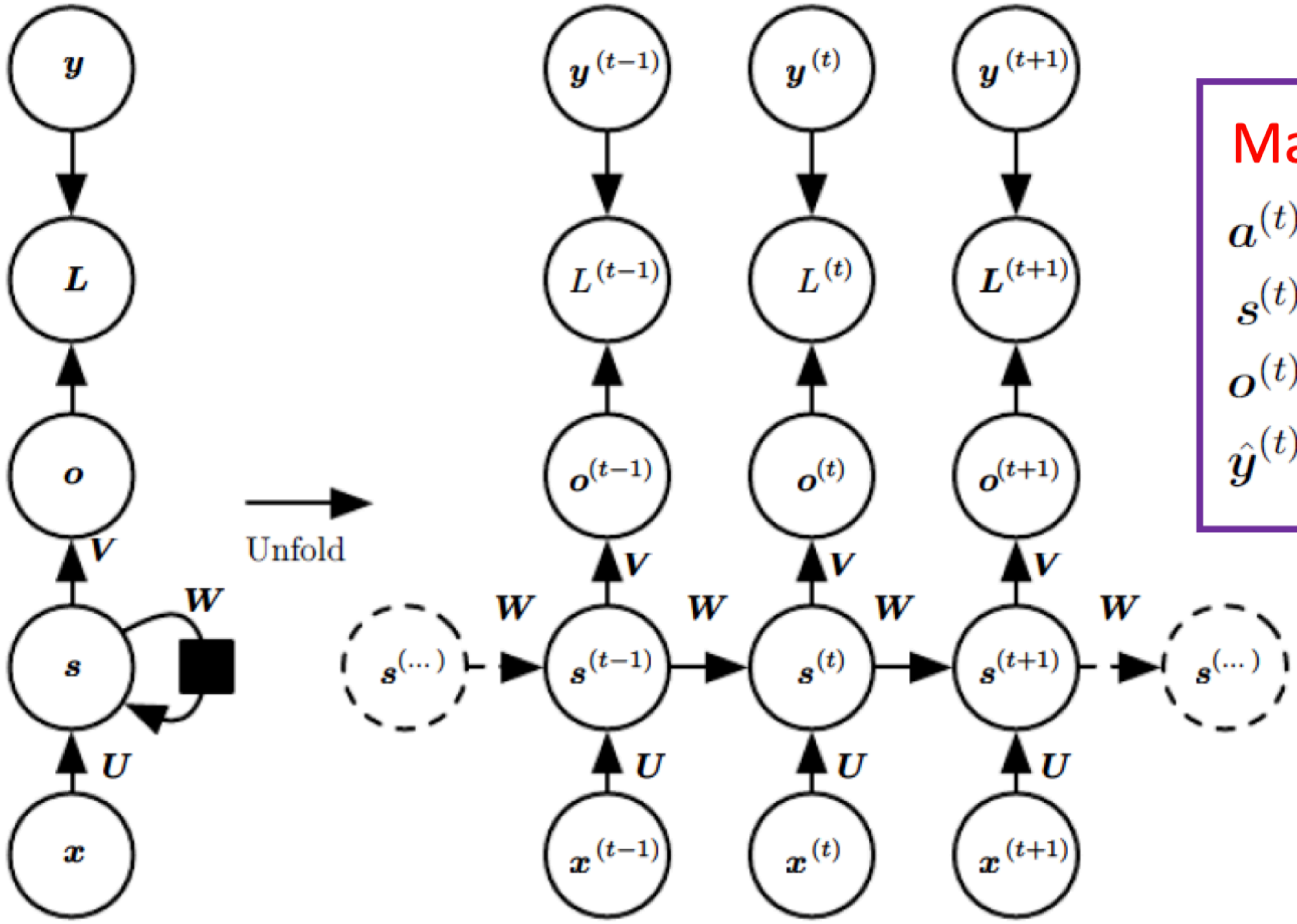
dict size × 1024

dim = dict size

vector of dict size


NN
h
V
$\vec{o}$
dict size
softmax( )
p(next word
goal
RNN
transformers

Recurrent Neural Networks

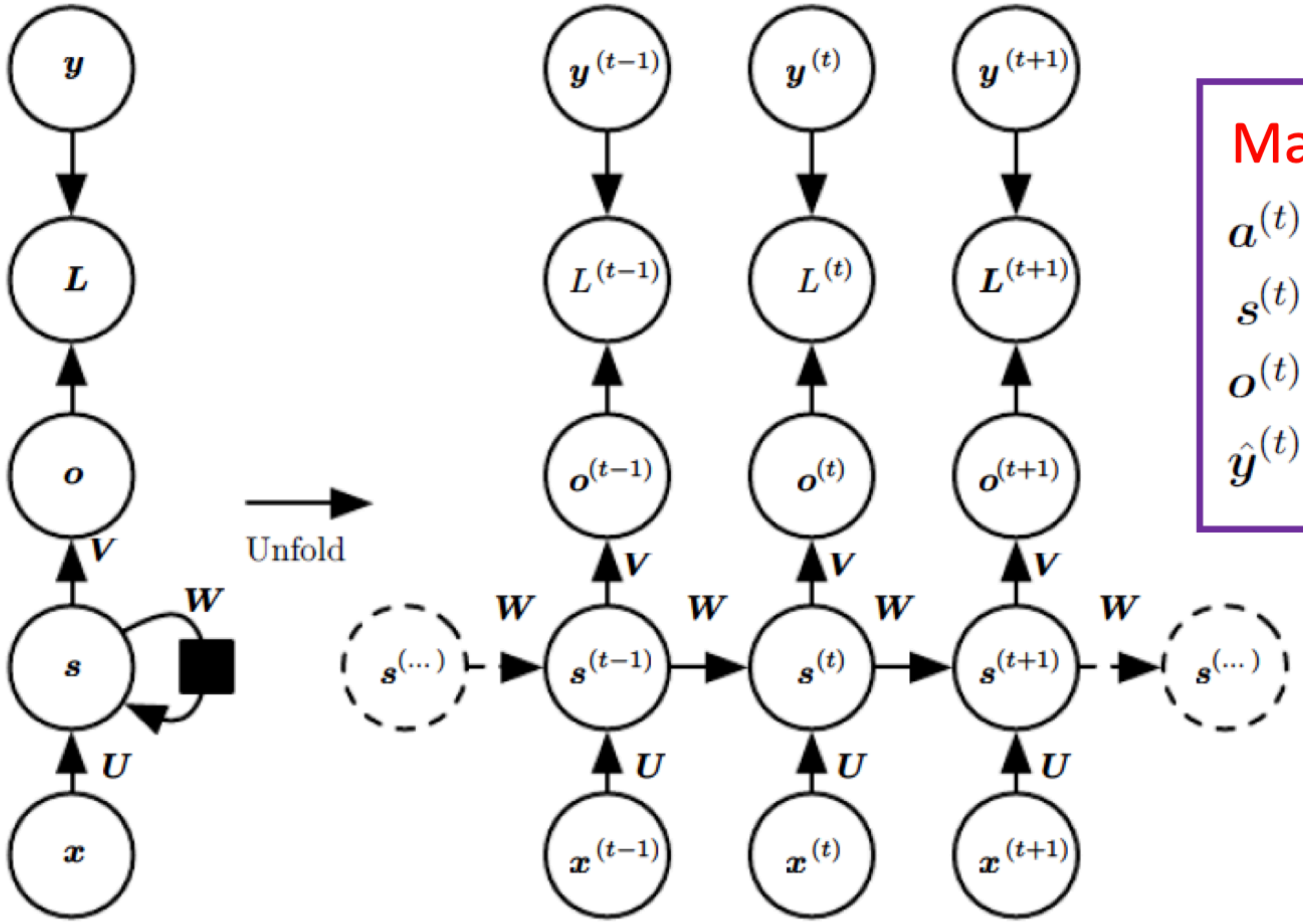


Math formula:

$$\begin{aligned} \boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\ \boldsymbol{s}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\ \boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}^{(t)} \\ \hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)}) \end{aligned}$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Recurrent Neural Networks

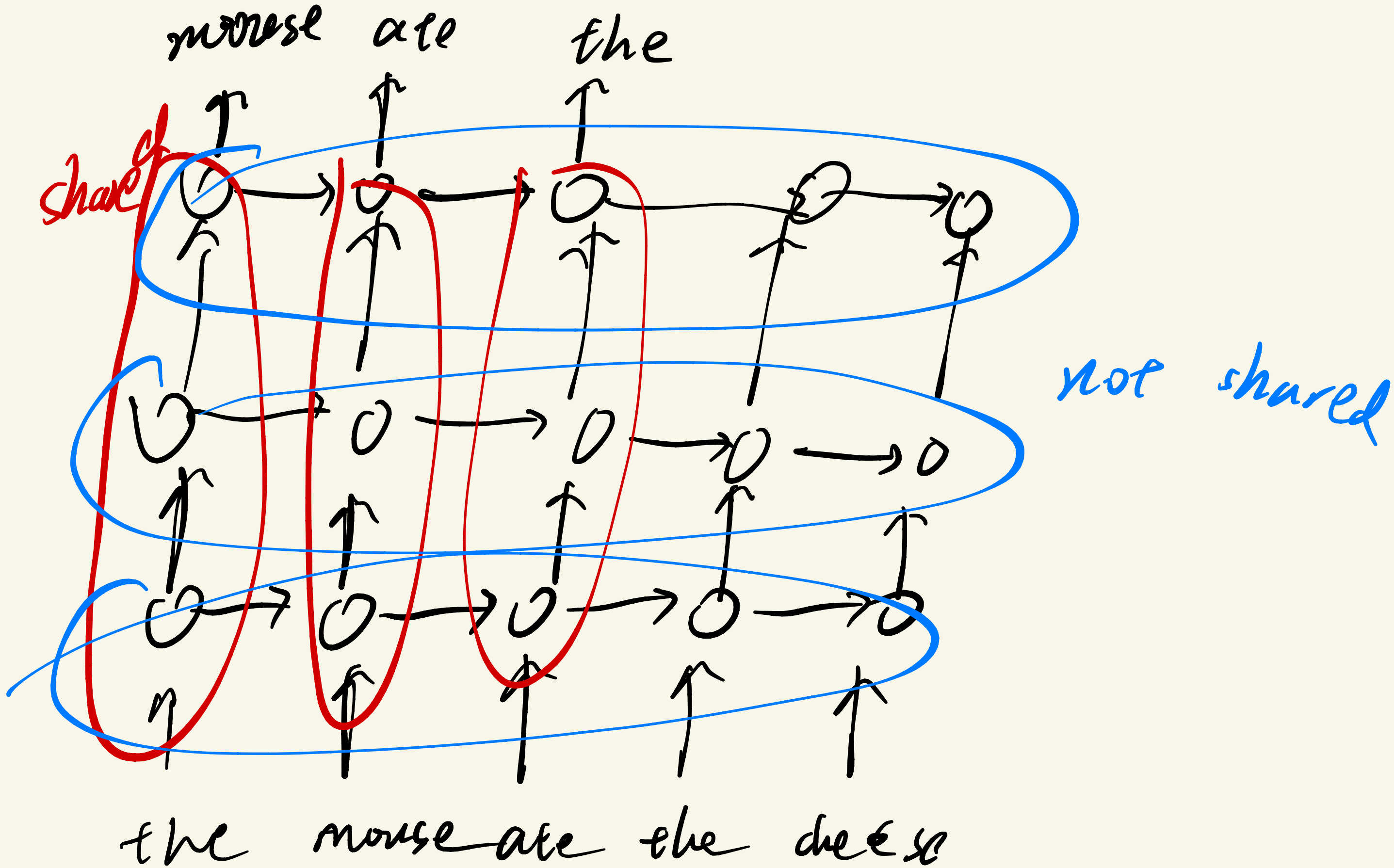


Math formula:

$$\begin{aligned} \boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\ \boldsymbol{s}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\ \boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}^{(t)} \\ \hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)}) \end{aligned}$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

There are many variants of RNNs since the functional form to compute $s^{(t)}$ can vary, e.g., LSTM



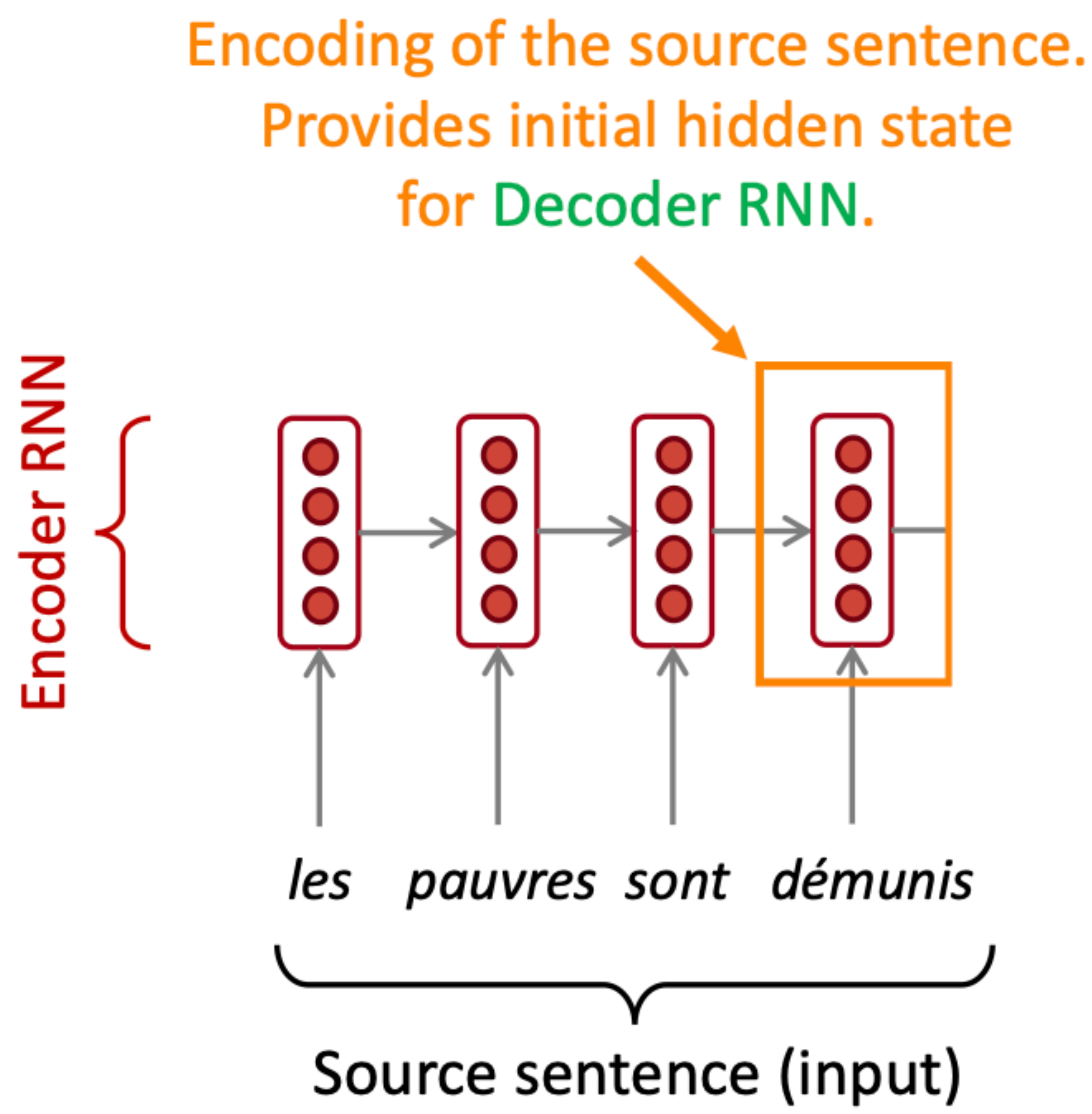
mouse ate the
shared
not shared
the mouse ate the cheese

Sequence-to-Sequence Learning

Example of Neural Machine Translation

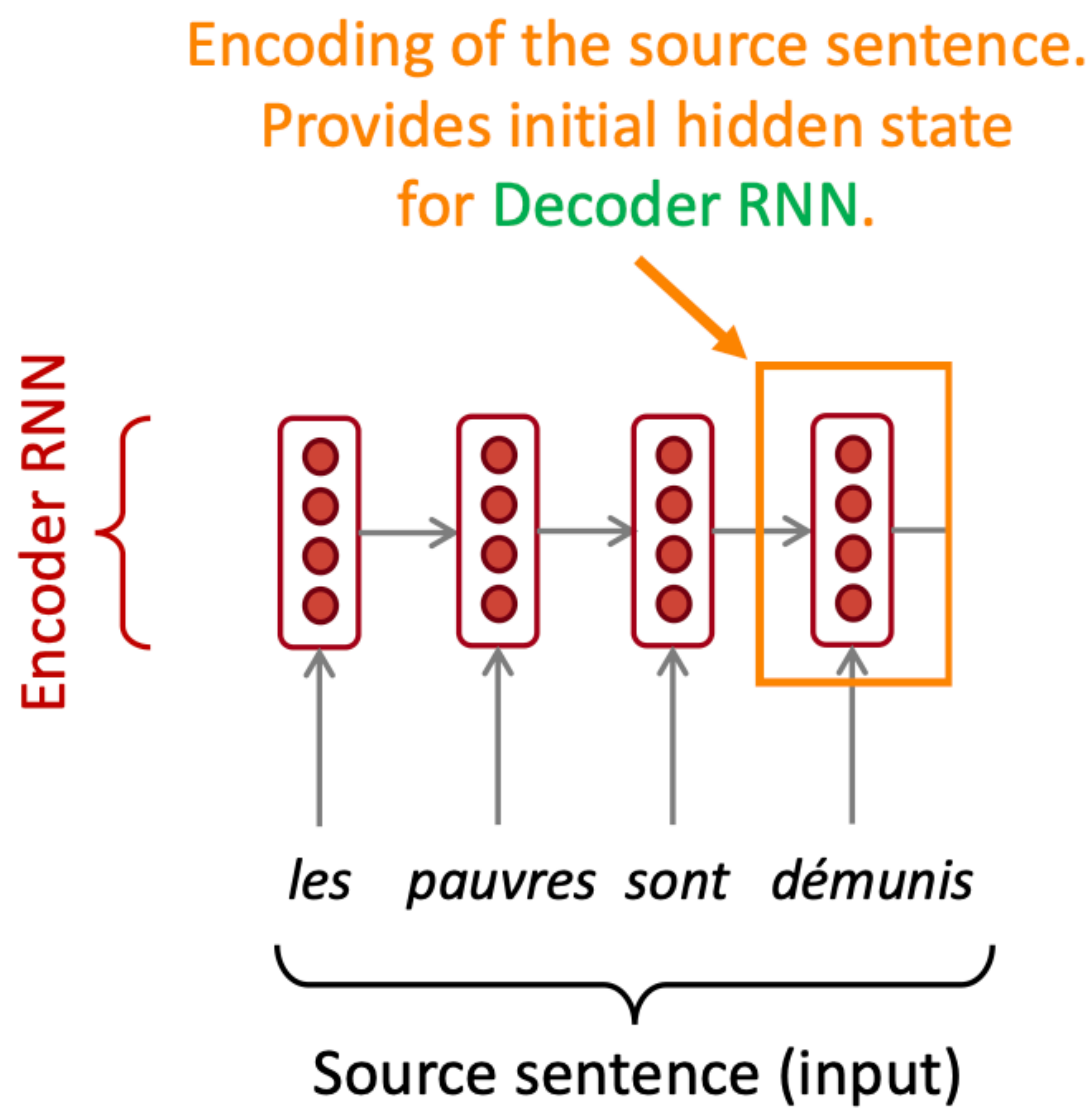
Sequence-to-Sequence Learning

Example of Neural Machine Translation



Sequence-to-Sequence Learning

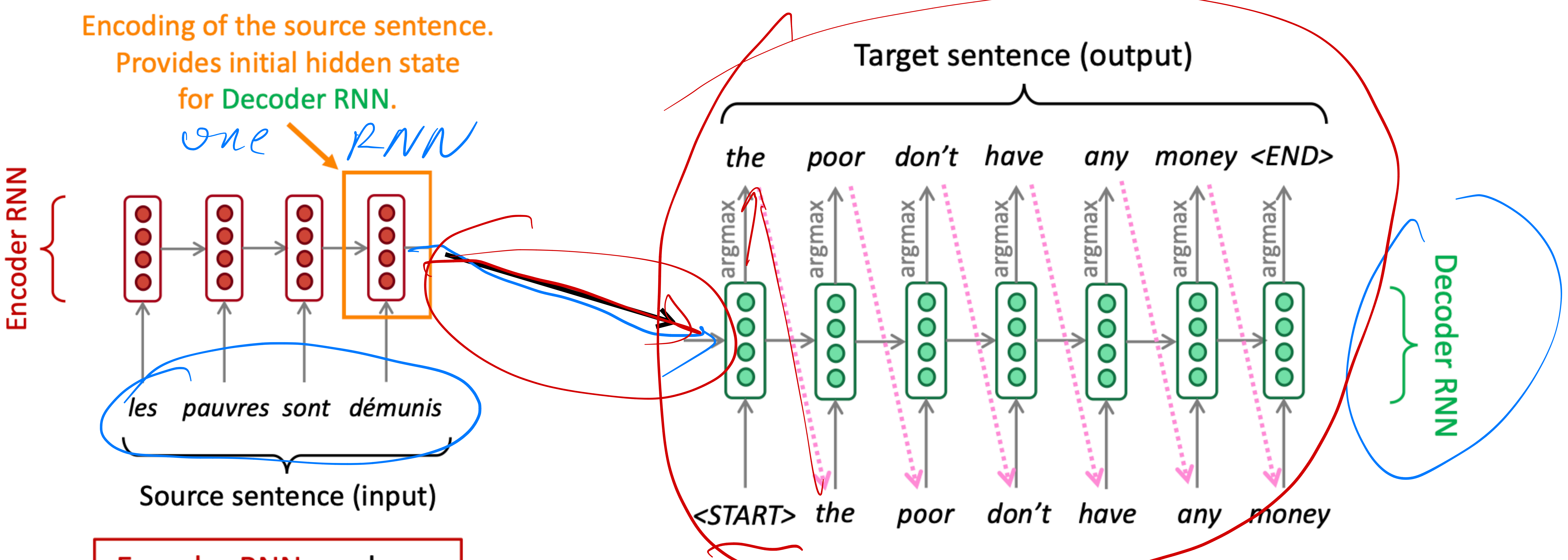
Example of Neural Machine Translation



Encoder RNN produces an encoding of the source sentence.

Sequence-to-Sequence Learning

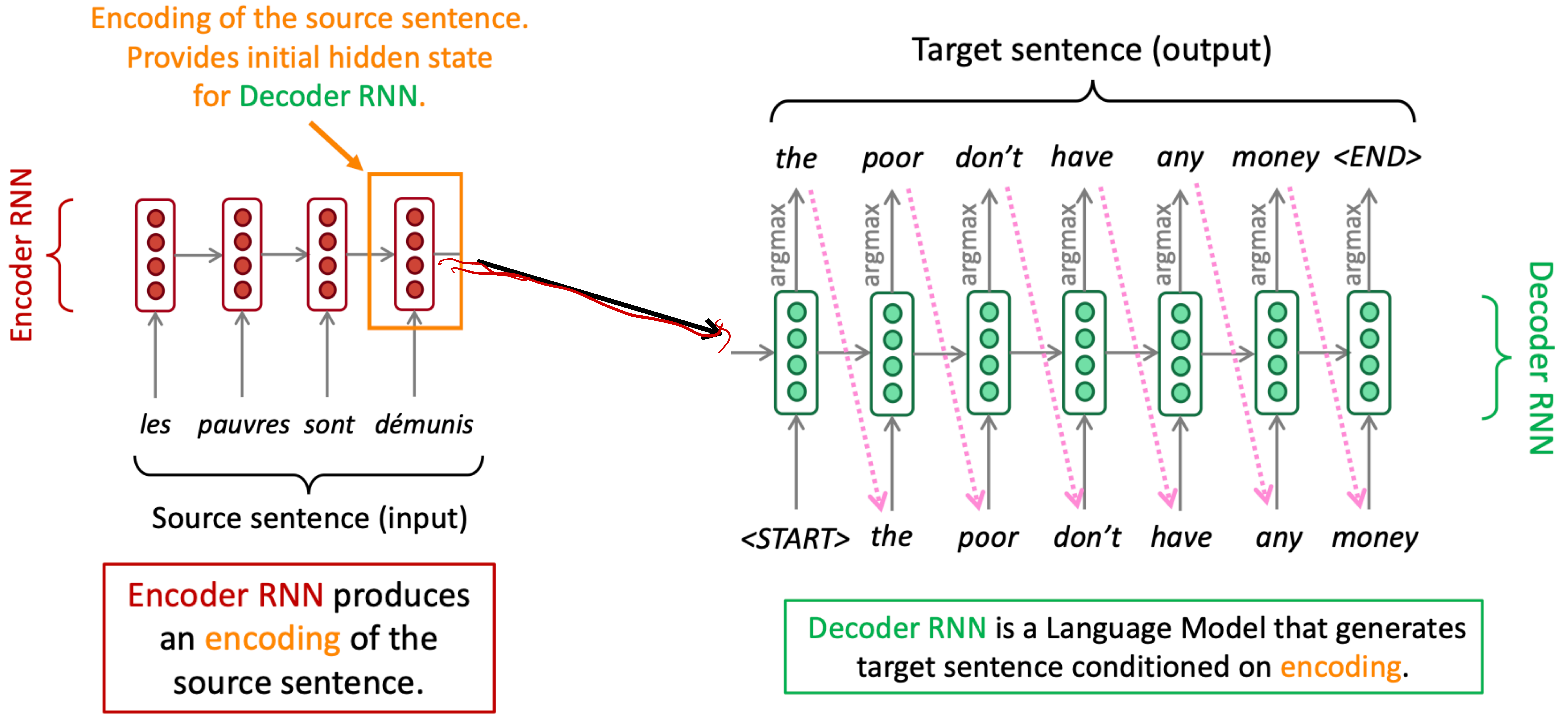
Example of Neural Machine Translation

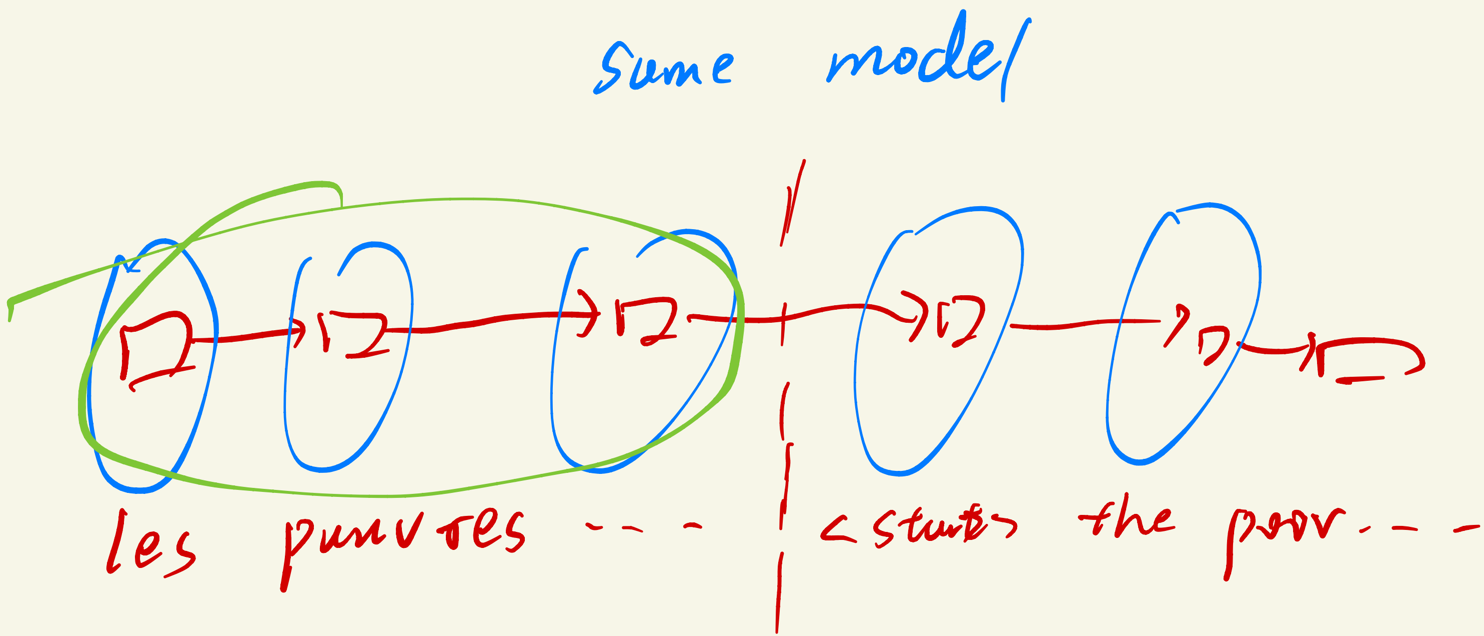


Encoder RNN produces an encoding of the source sentence.

Sequence-to-Sequence Learning

Example of Neural Machine Translation

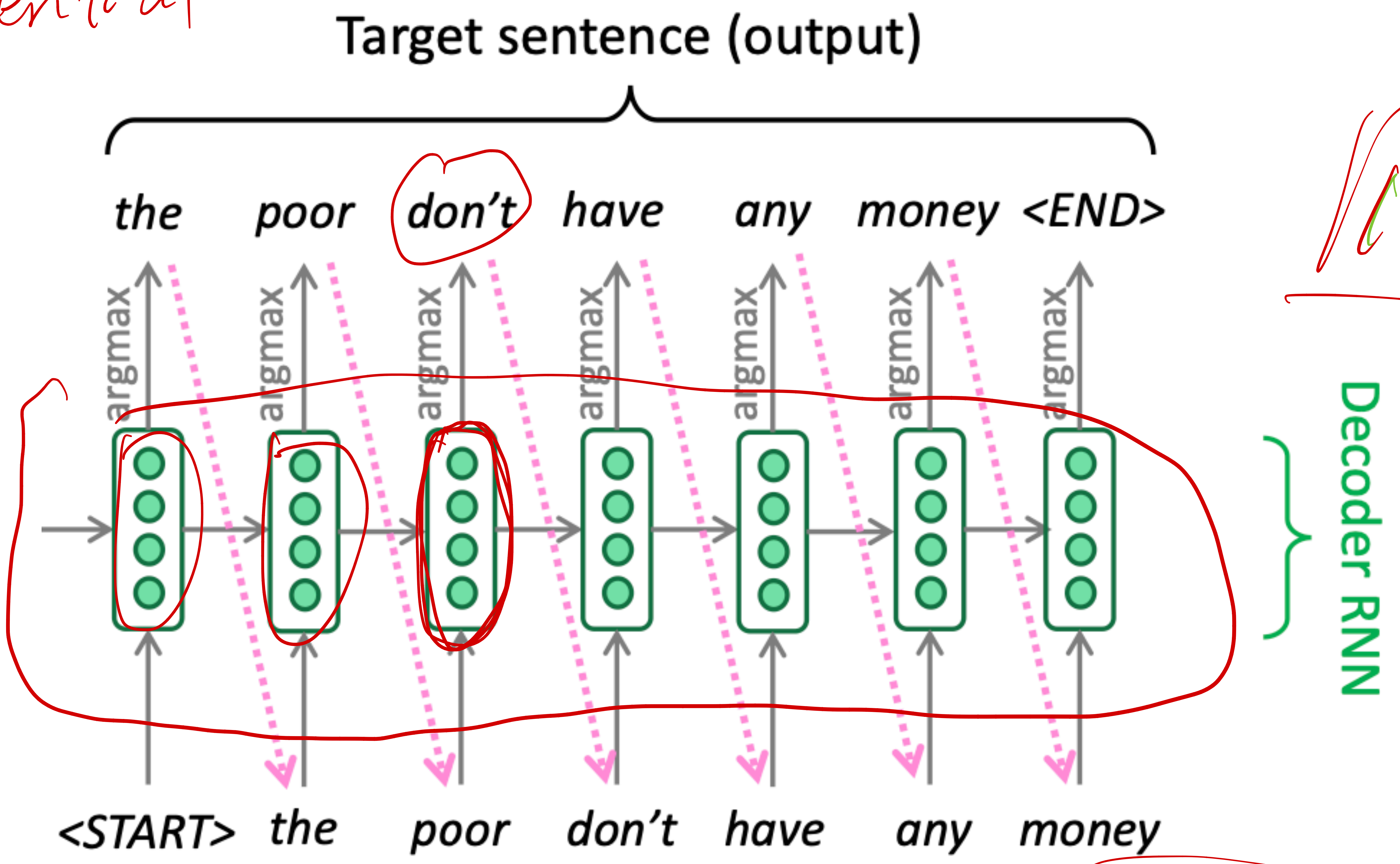




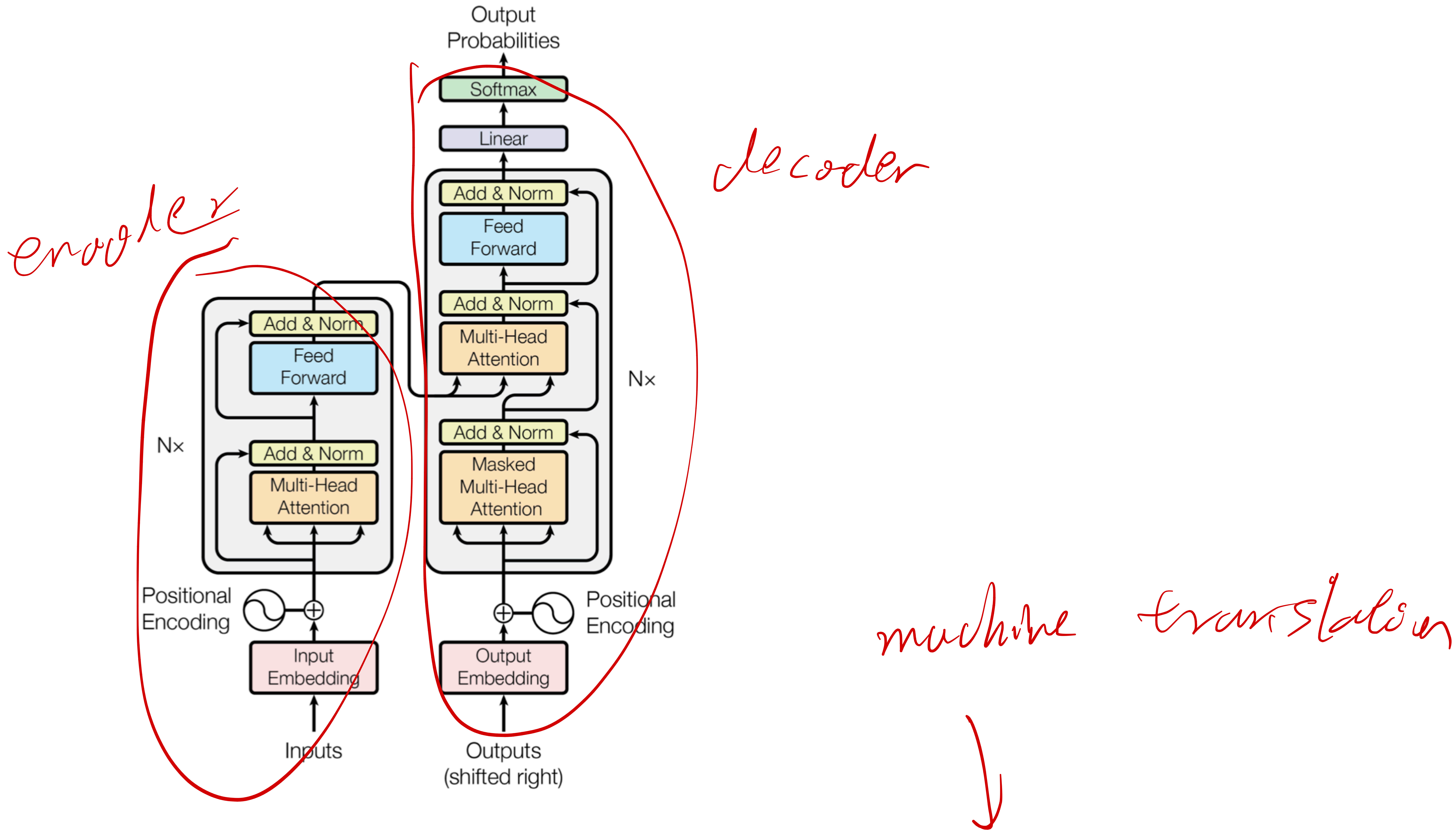
same model
les pauvres ...
<start> the poor ...

RNN Language Model

RNN Language Model

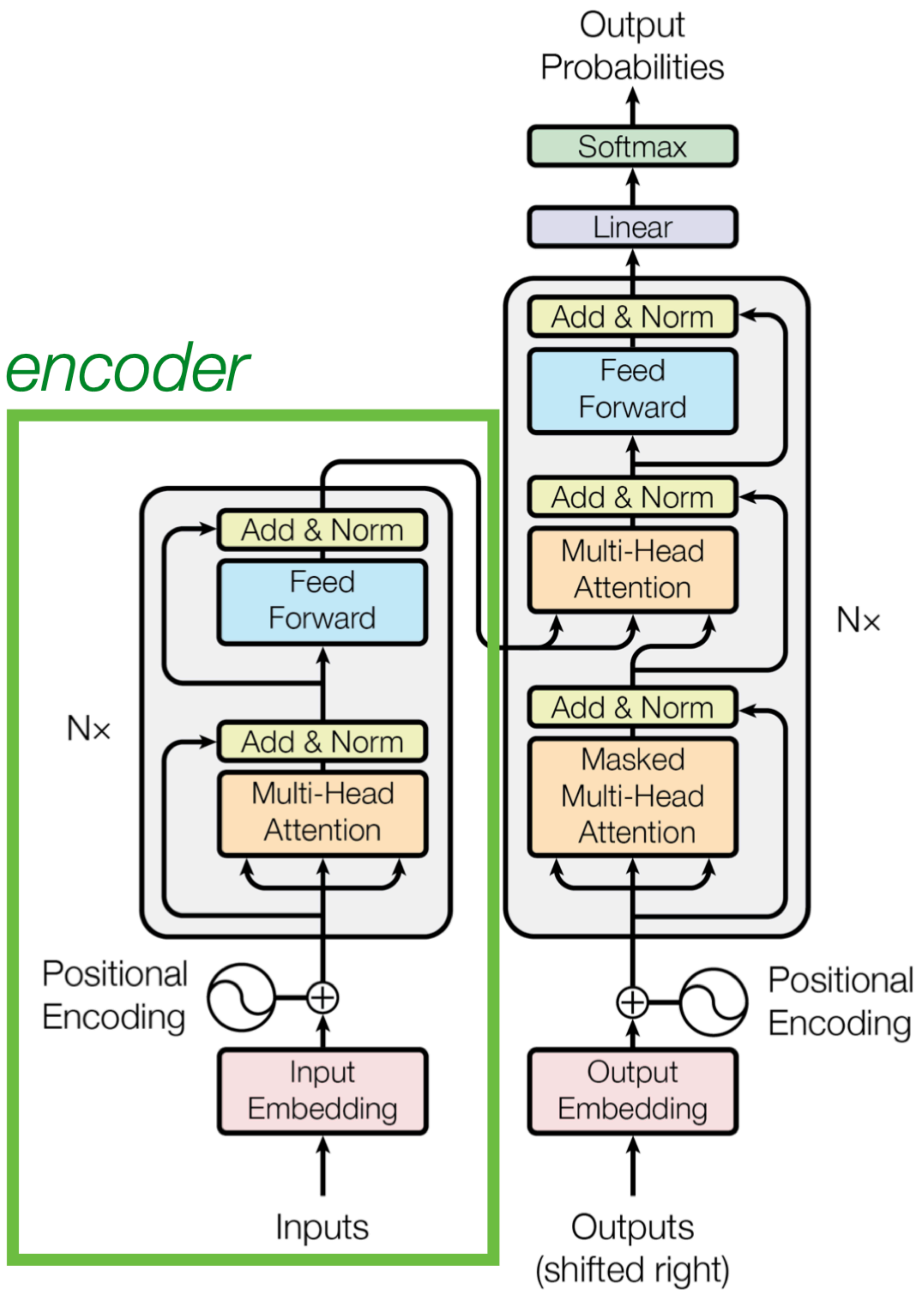


Transformer



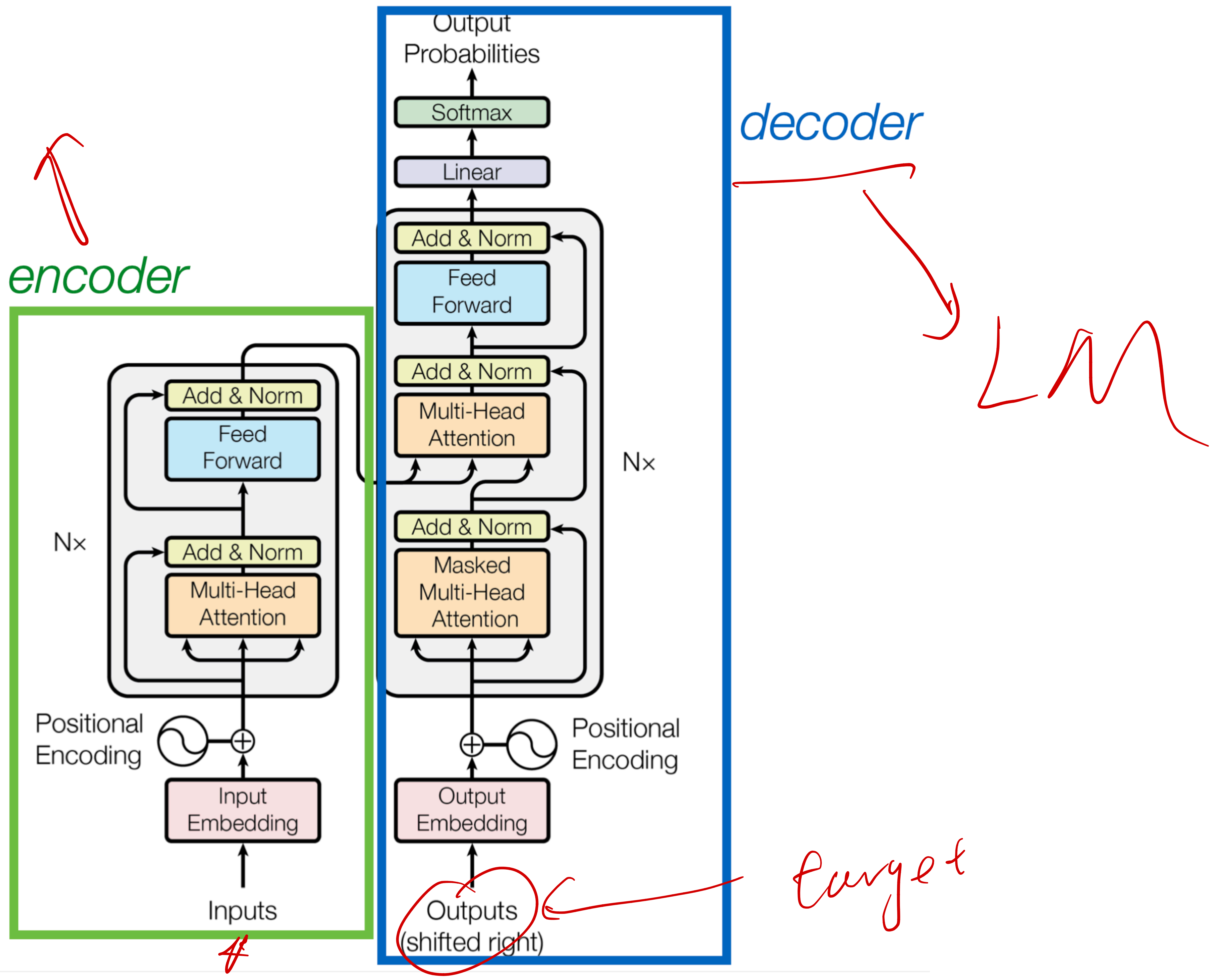
Vaswani et al. Attention is All You Need. NeurIPS 2017.

Encoder

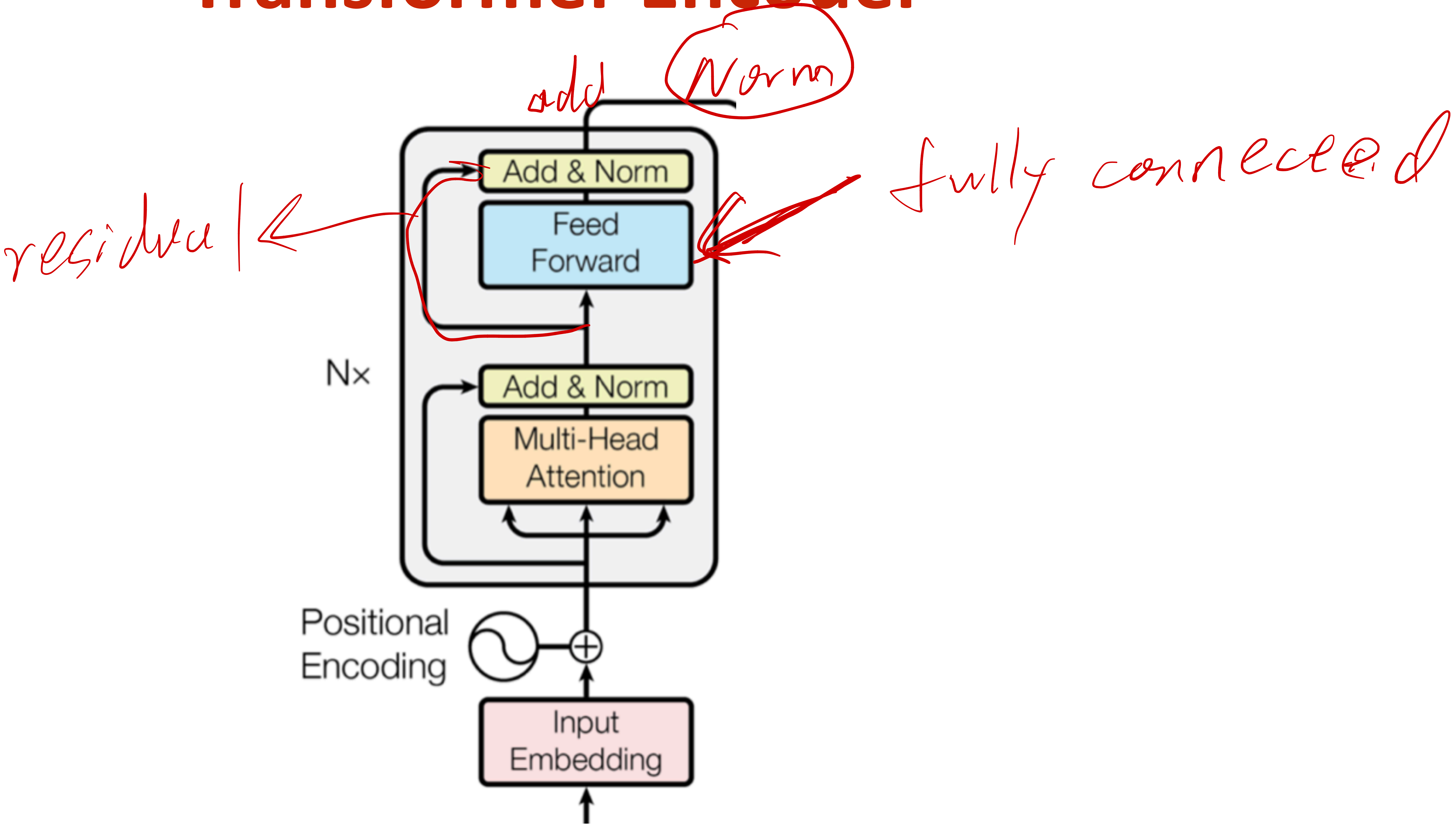


encoder
Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

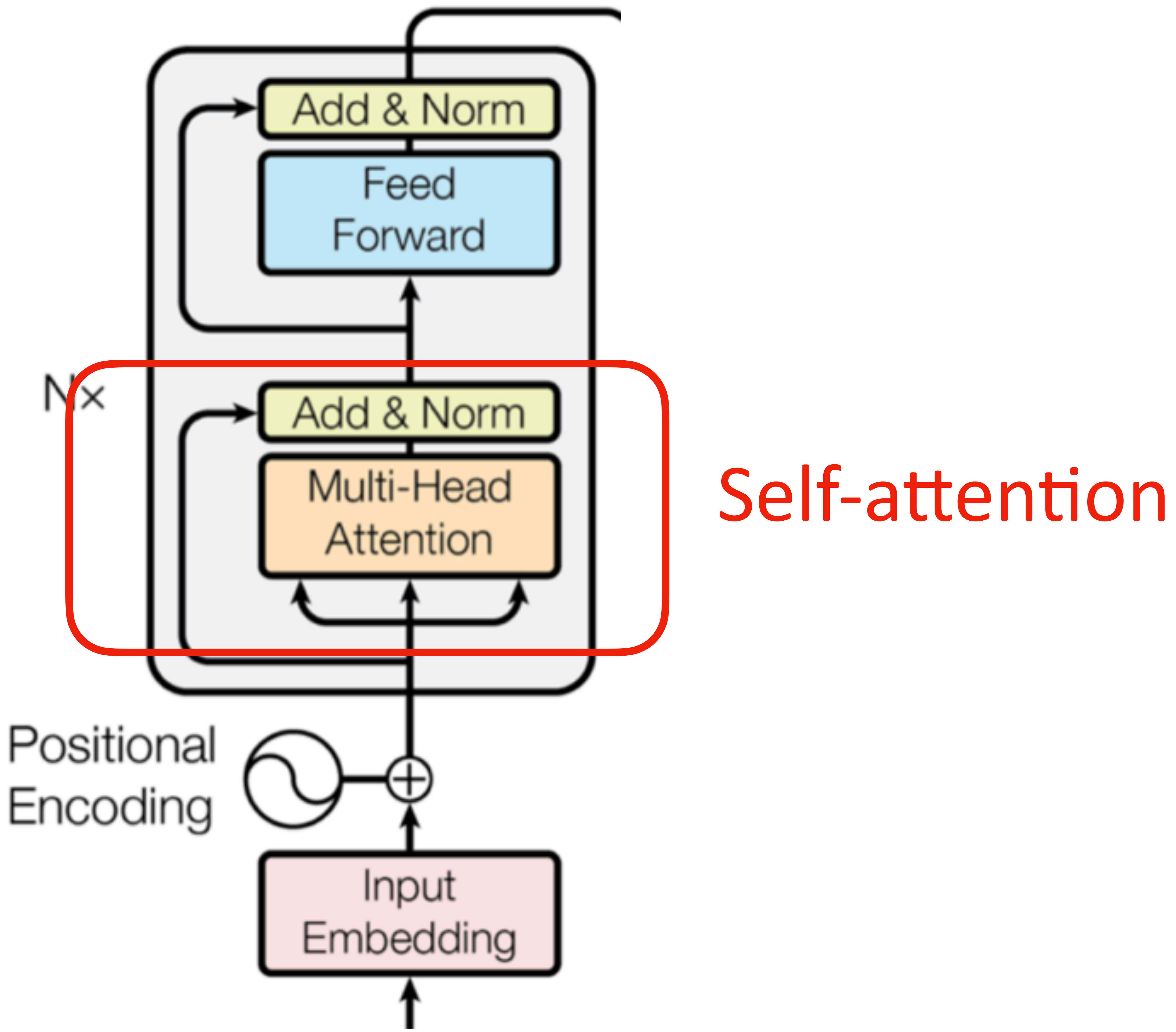
Decoder


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

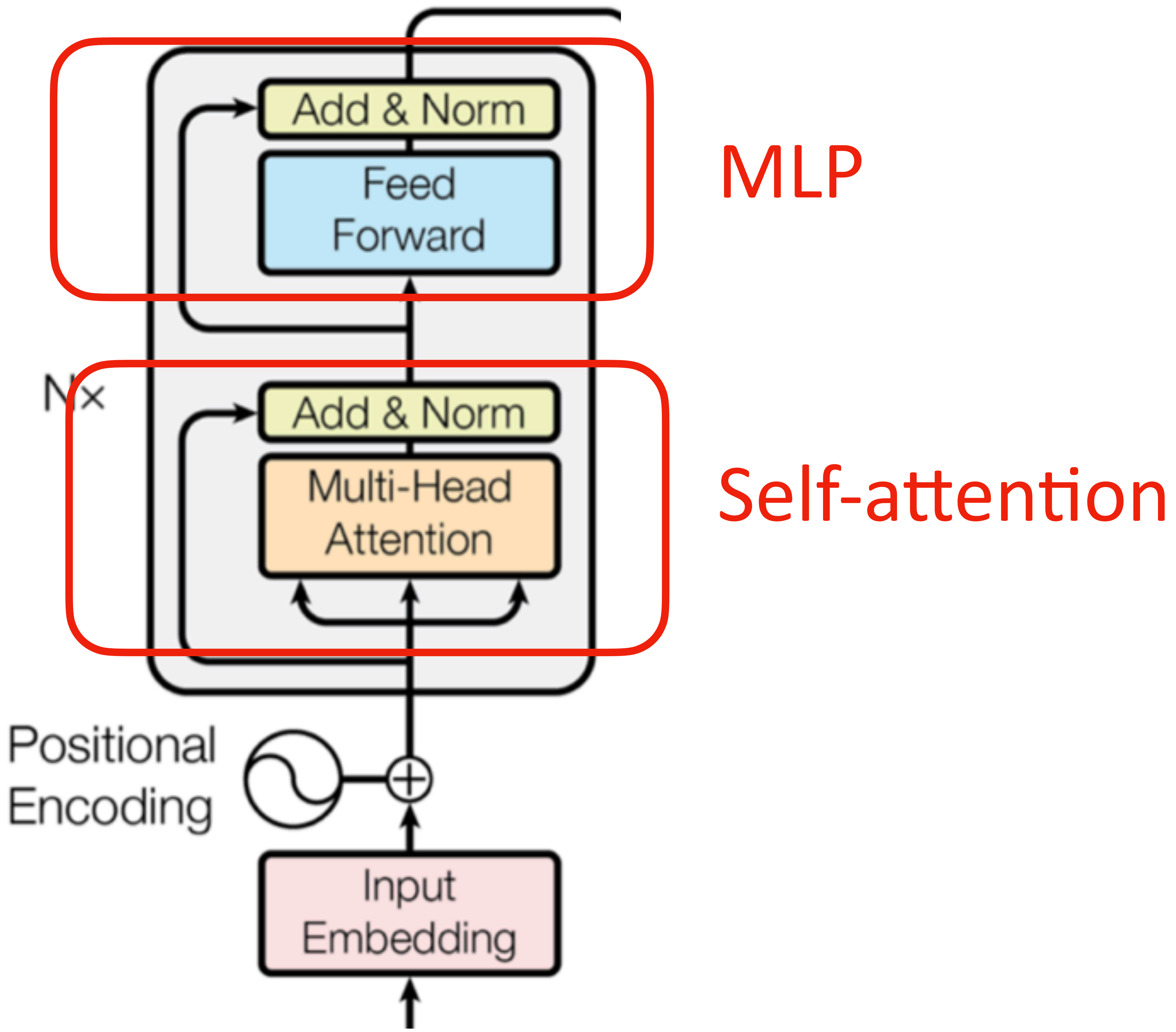
Transformer Encoder



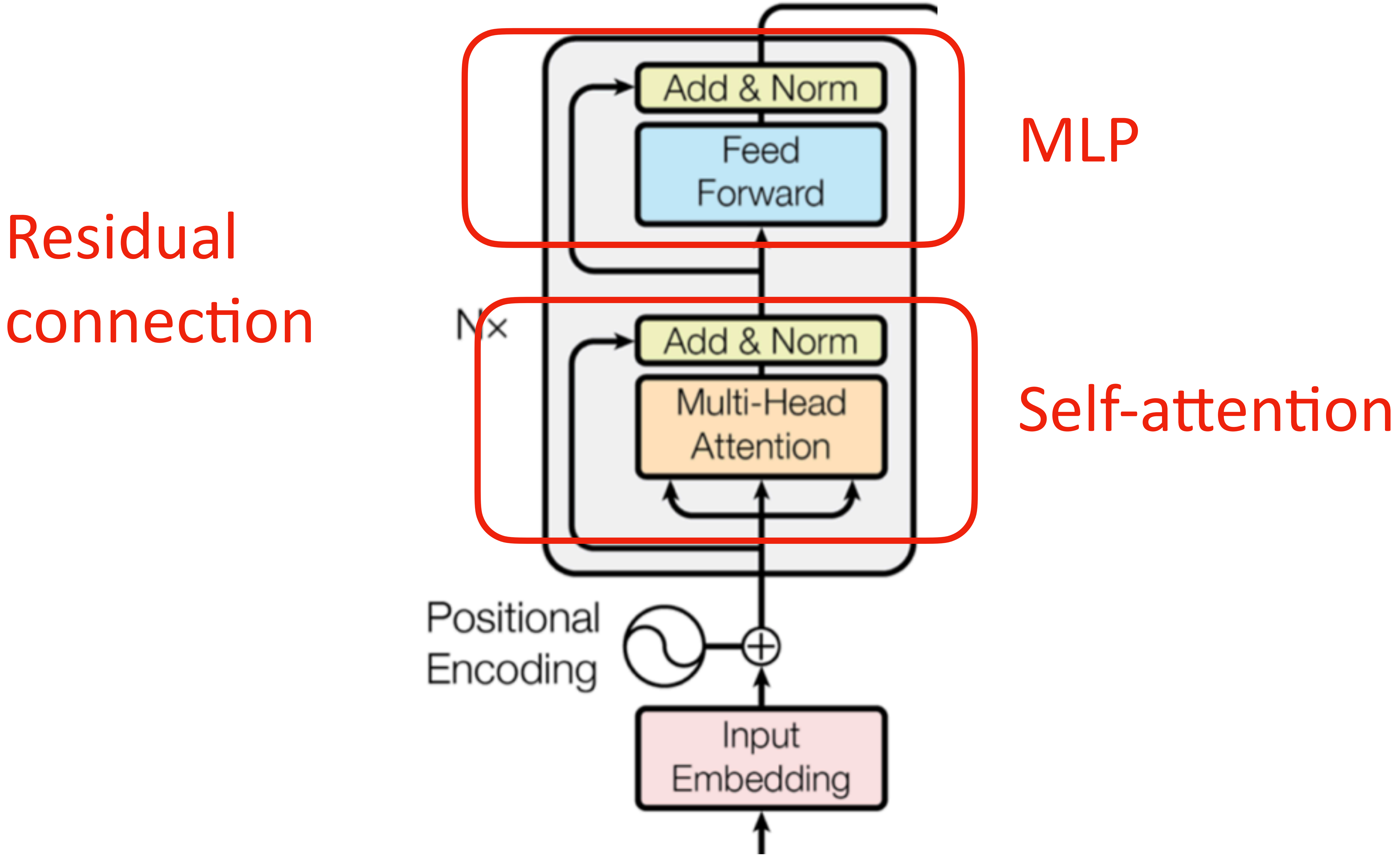
Transformer Encoder



Transformer Encoder

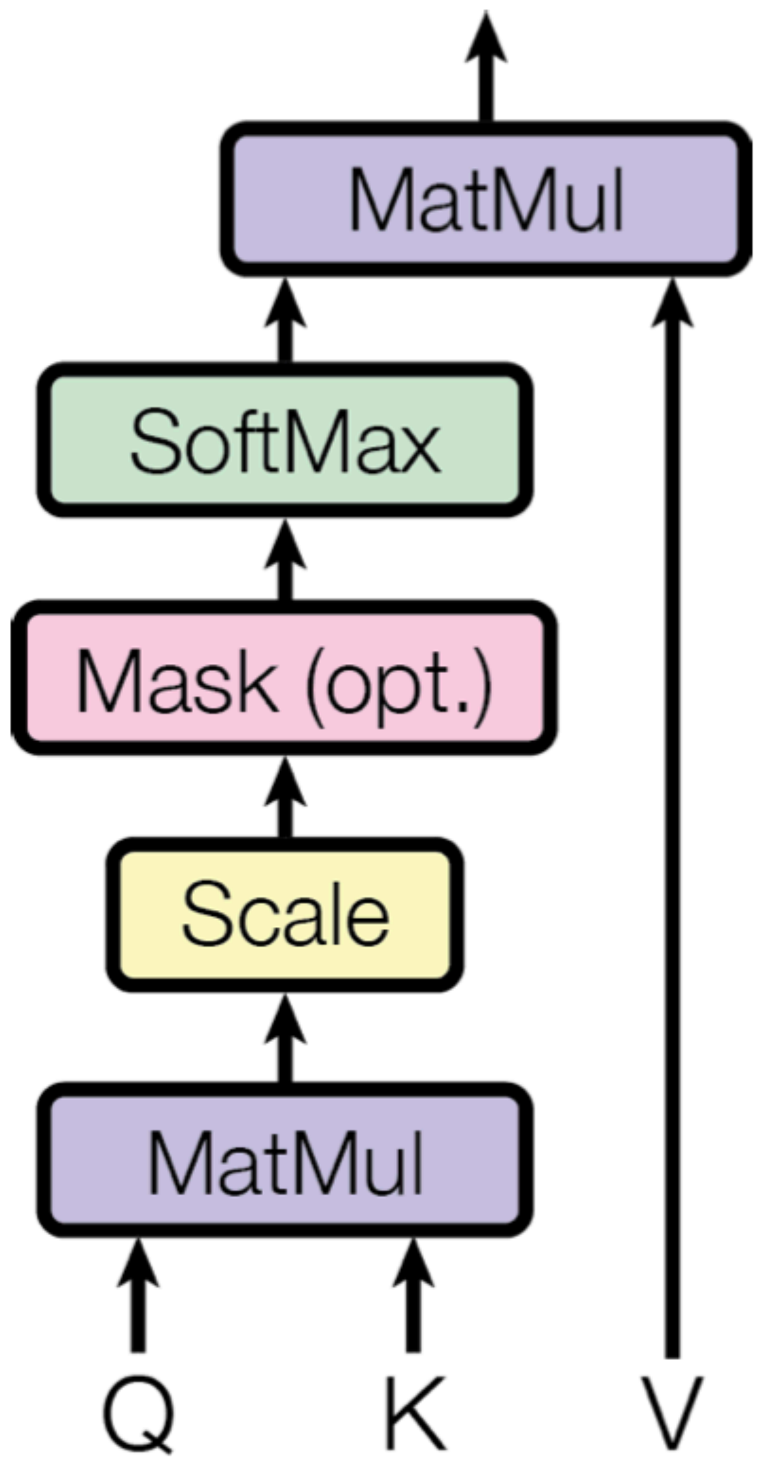


Transformer Encoder

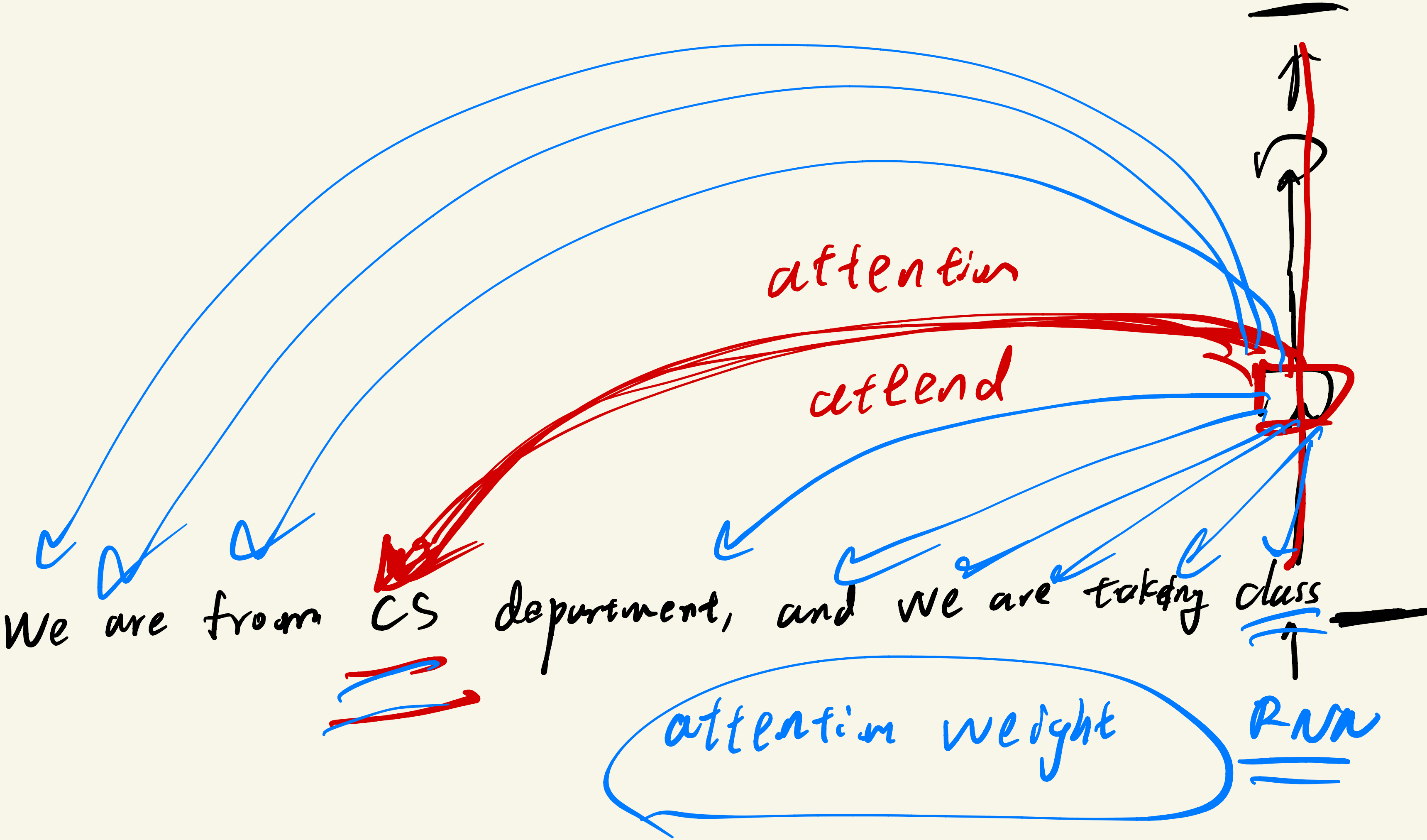


What is Attention

Scaled Dot-Product Attention



Q: Query
K: key
V: value



attention
attend
We are from CS department, and we are taking class
attention weight
RNN

attention weight dot product

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum x_i y_i$$

query(class) · key(cs) → weight

How much attend

query
key
value

query
key
value

query
key
value

key(deparment)

We are from cs department and we are taking class

[weight] ← [0, 1]

weight (class, cs)

weigh (class, depart)

weight (class, we)

)

⋮

softmax (   ) = attention weight

weight [0, 1]

probability "class"

attend "cs)

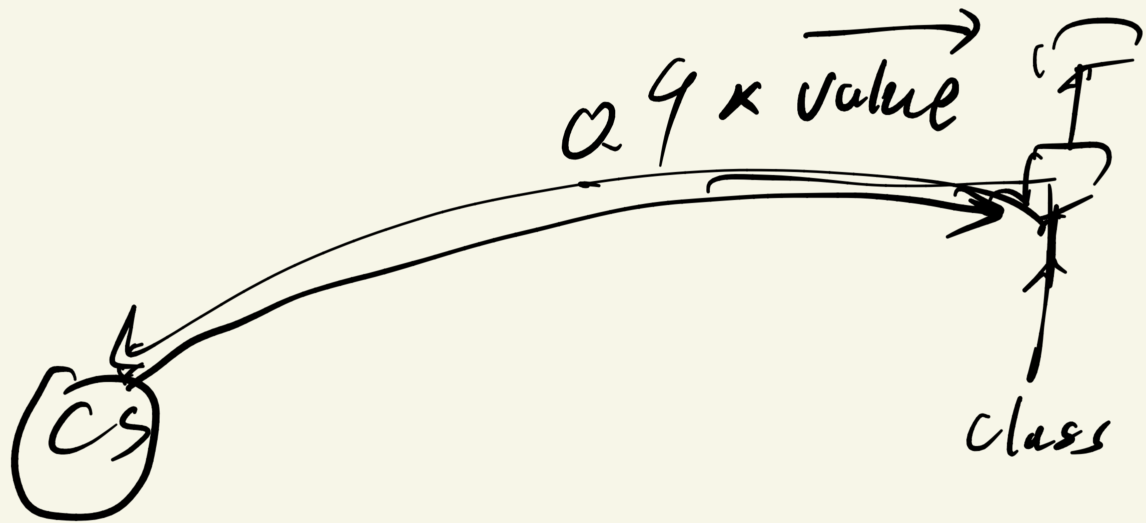
influence:

~~cls~~

attn weight for each word

w_i

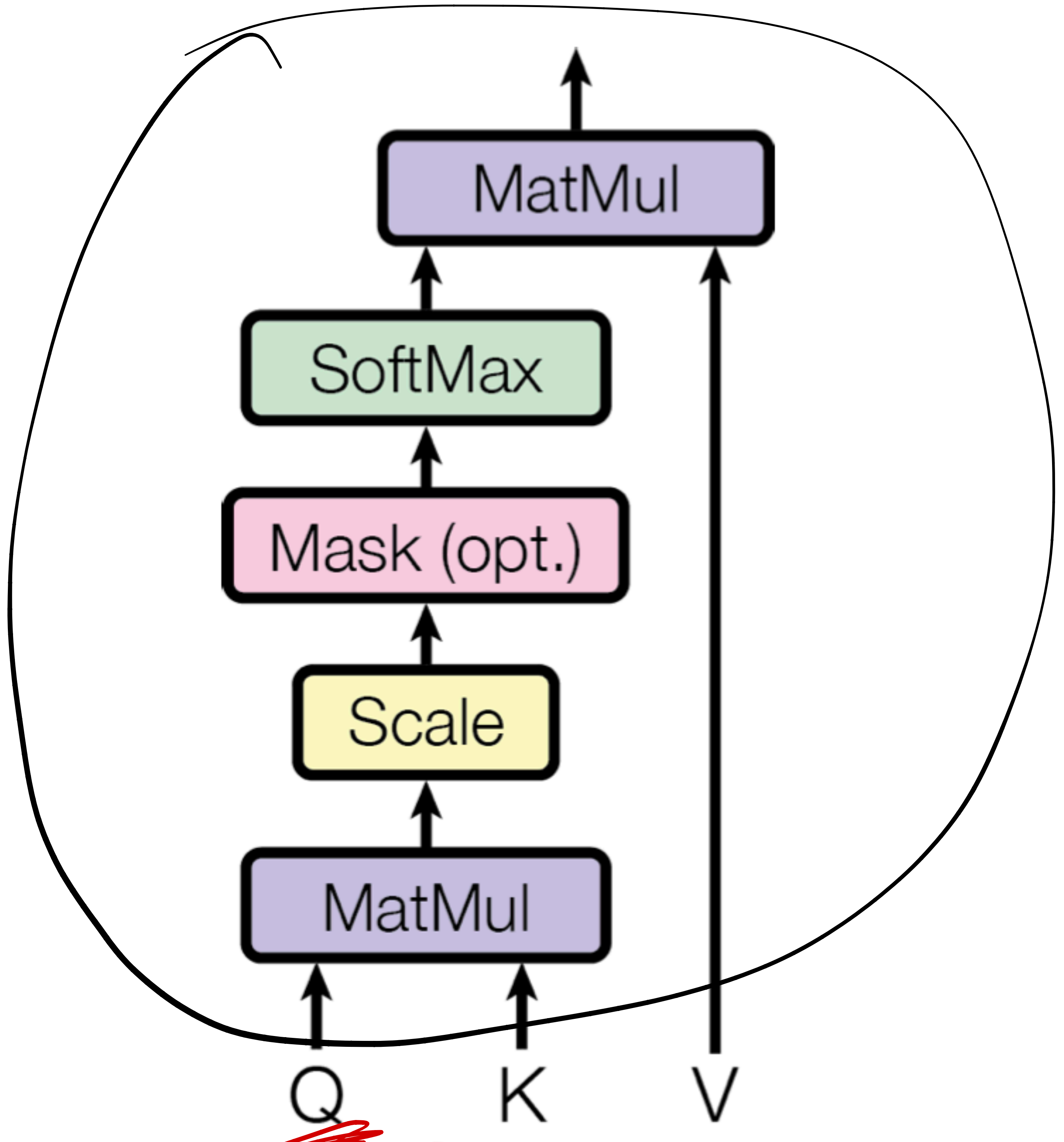
$\sum_i w_i v_i$ —influence→



What is Attention

$Q \in R^{n \times d}$ $K \in R^{m \times d}$ $V \in R^{m \times d}$

Scaled Dot-Product Attention



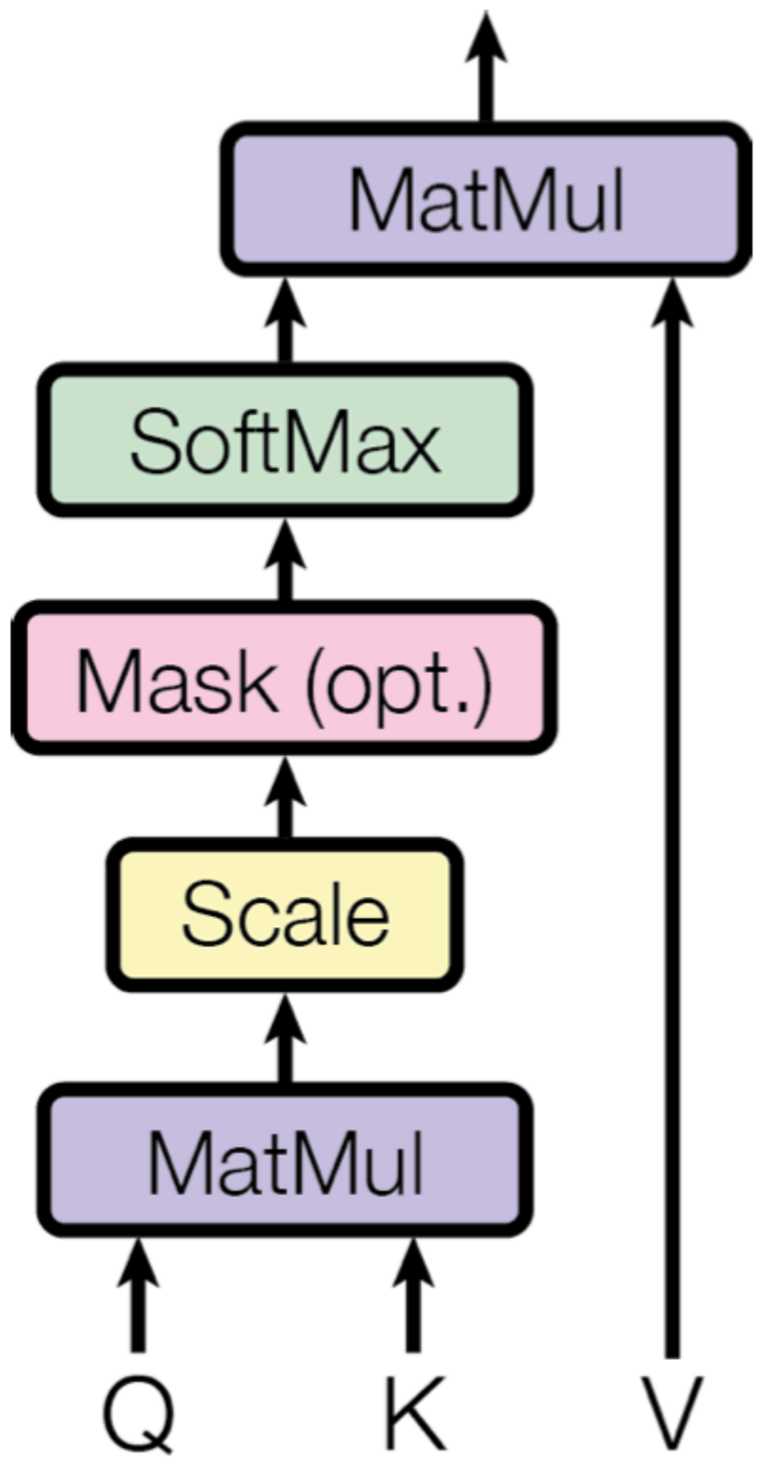
Q: Query
K: key
V: value

What is Attention

$$Q \in R^{n \times d} \qquad K \in R^{m \times d} \qquad V \in R^{m \times d}$$

We have n queries, m (key, value) pairs

Scaled Dot-Product Attention



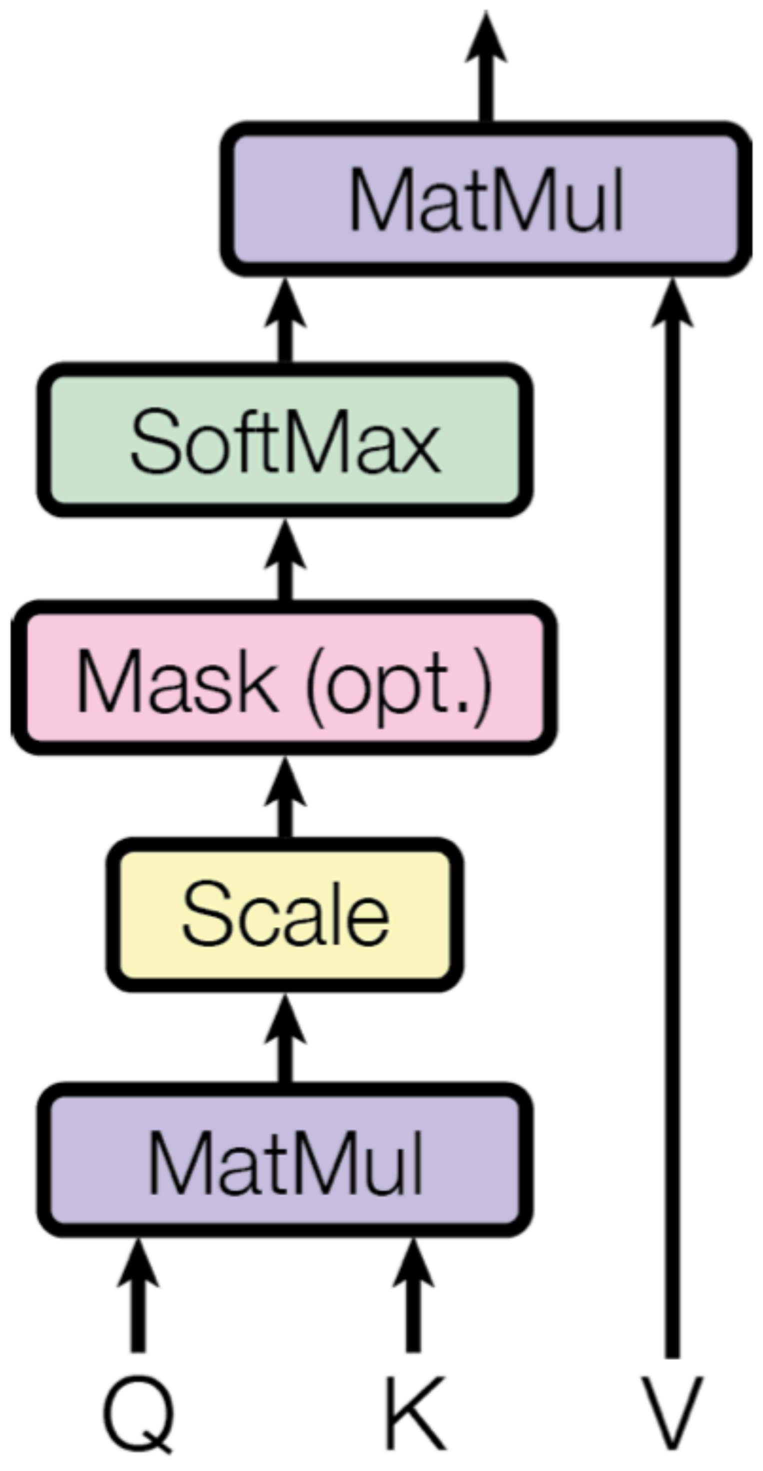
Q: Query
K: key
V: value

What is Attention

$Q \in R^{n \times d}$ $K \in R^{m \times d}$ $V \in R^{m \times d}$

We have n queries, m (key, value) pairs

Scaled Dot-Product Attention



Attention weight = softmax(QK^T)

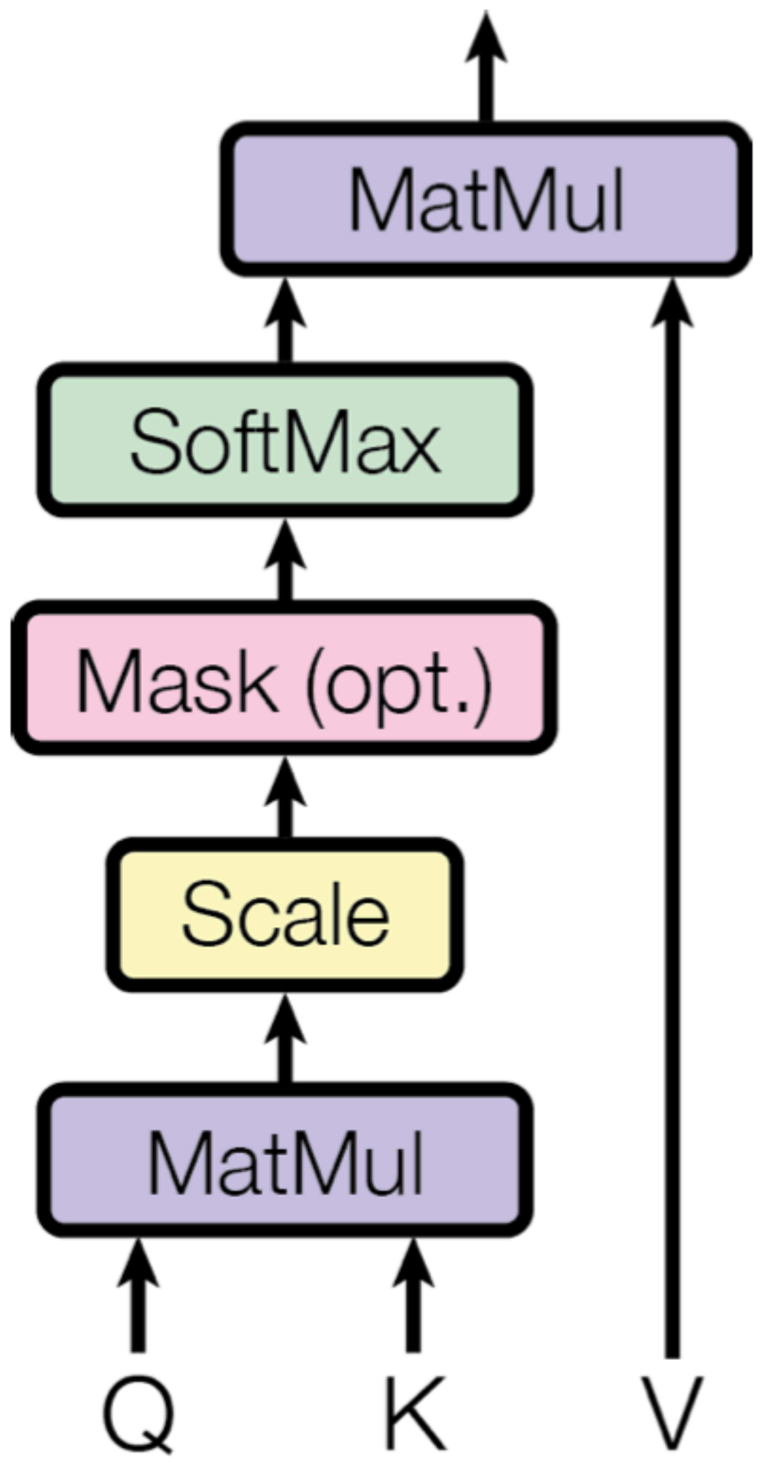
Q: Query
K: key
V: value

What is Attention

$Q \in R^{n \times d}$ $K \in R^{m \times d}$ $V \in R^{m \times d}$

We have n queries, m (key, value) pairs

Scaled Dot-Product Attention



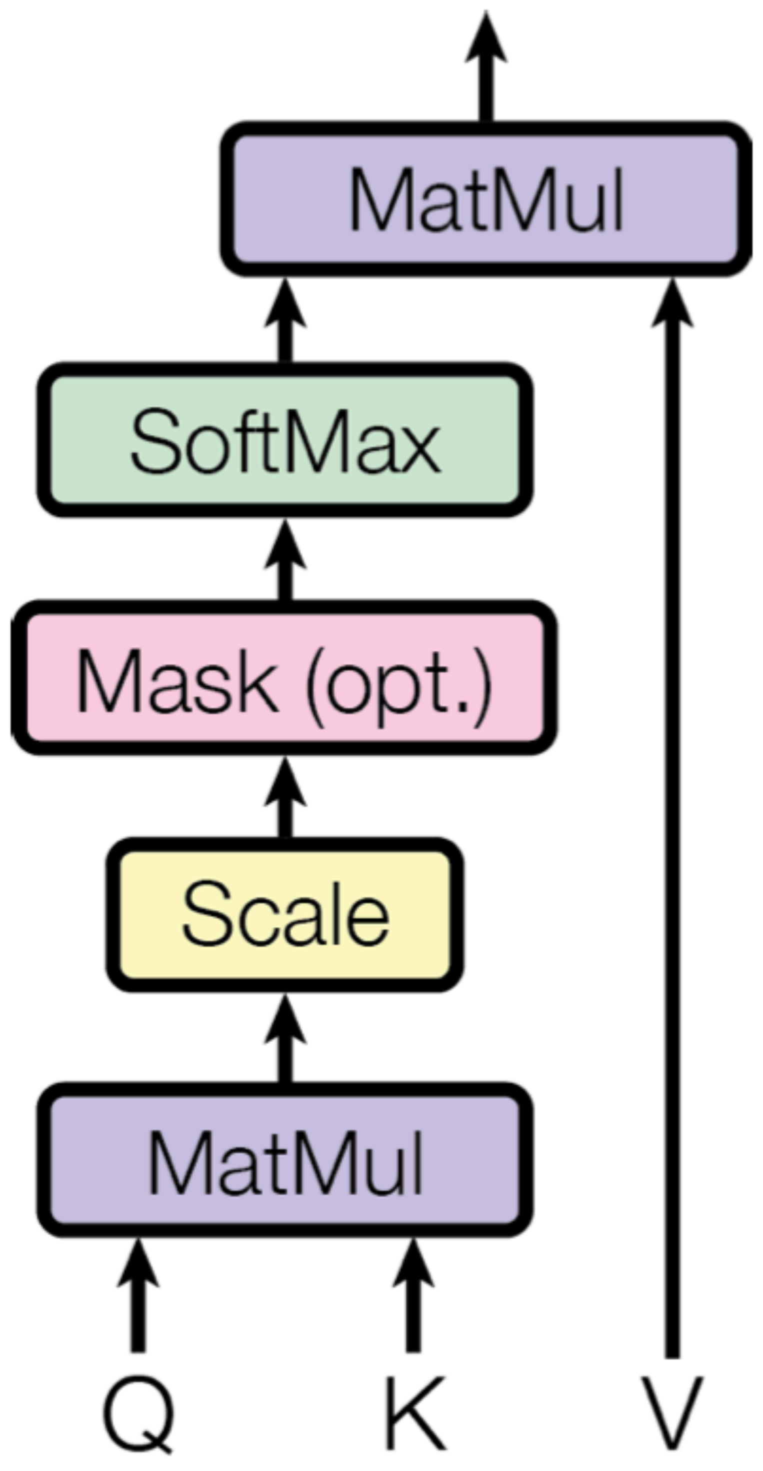
Attention weight = softmax(QK^T)

Dot-products grow large in magnitude

Q: Query
K: key
V: value

What is Attention

Scaled Dot-Product Attention



Q: Query
K: key
V: value

$Q \in R^{n \times d}$ $K \in R^{m \times d}$ $V \in R^{m \times d}$

We have n queries, m (key, value) pairs

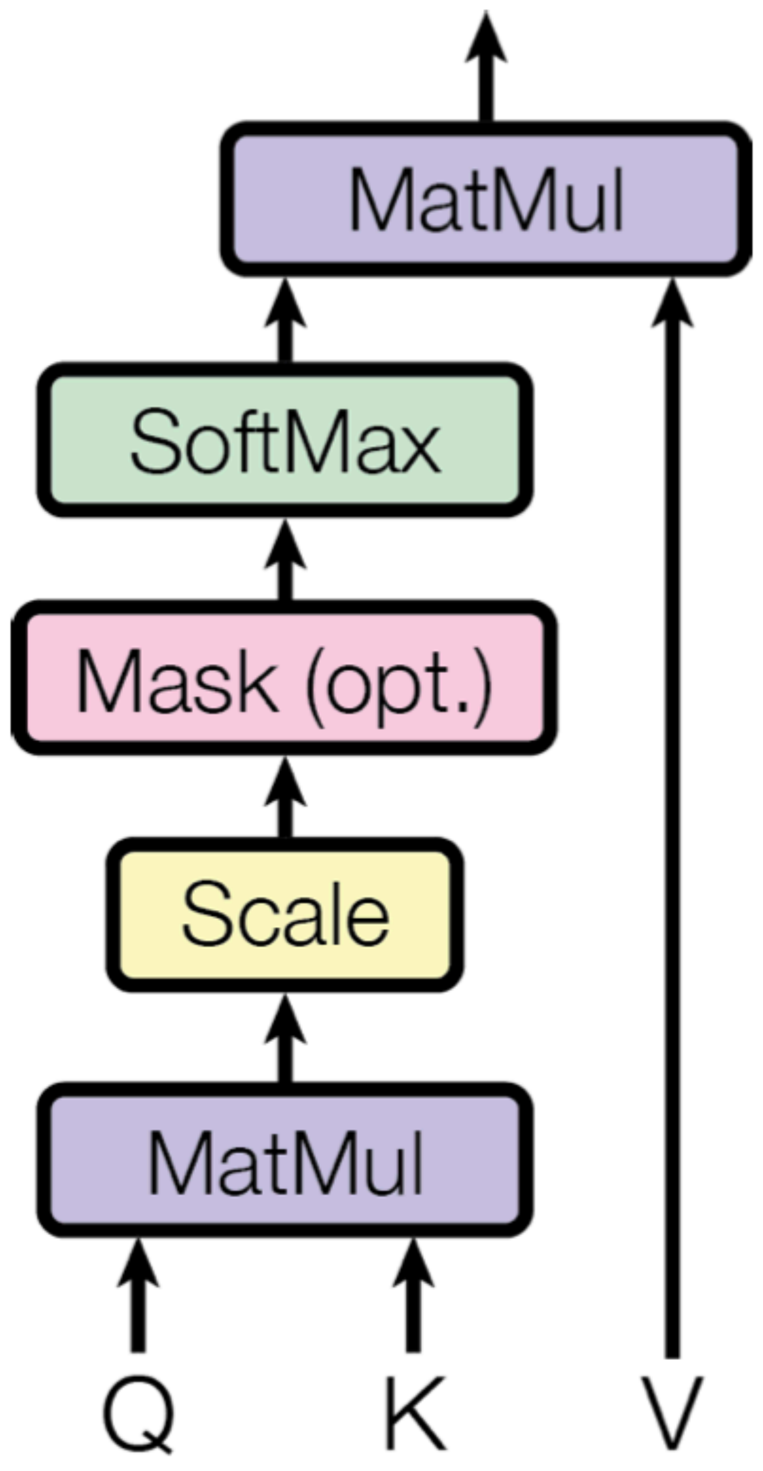
Attention weight = softmax(QK^T)

Dot-products grow large in magnitude

Scaled Attention weight = softmax$(\frac{QK^T}{\sqrt{d_k}})$

What is Attention

Scaled Dot-Product Attention



Q: Query
K: key
V: value

$Q \in R^{n \times d}$ $K \in R^{m \times d}$ $V \in R^{m \times d}$

We have n queries, m (key, value) pairs

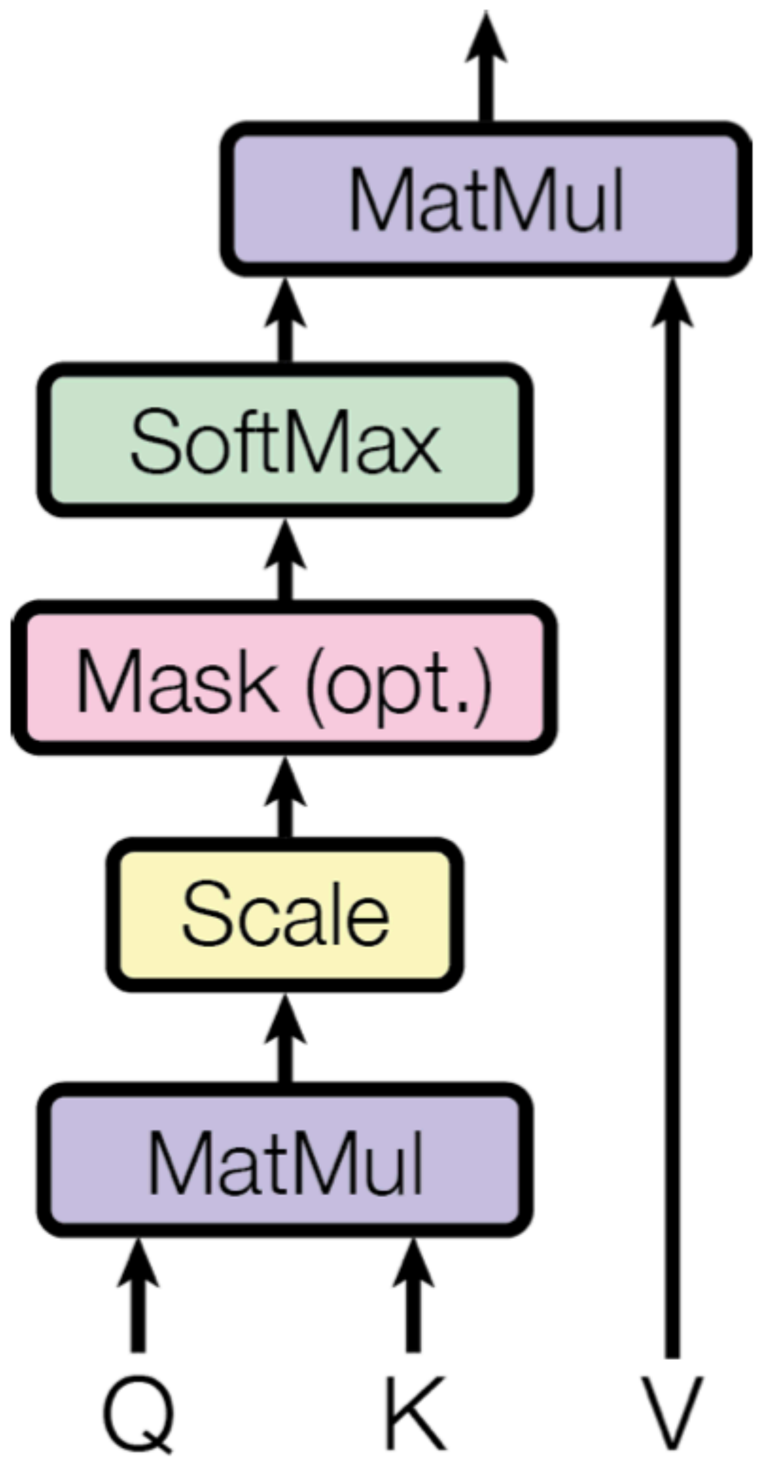
Attention weight = softmax(QK^T)

Dot-products grow large in magnitude

Scaled Attention weight = softmax$(\frac{QK^T}{\sqrt{d_k}})$ Shape is mxn

What is Attention

Scaled Dot-Product Attention



Q: Query

K: key

V: value

$$Q \in R^{n \times d} \qquad K \in R^{m \times d} \qquad V \in R^{m \times d}$$

We have n queries, m (key, value) pairs

Attention weight = softmax(QK^T)

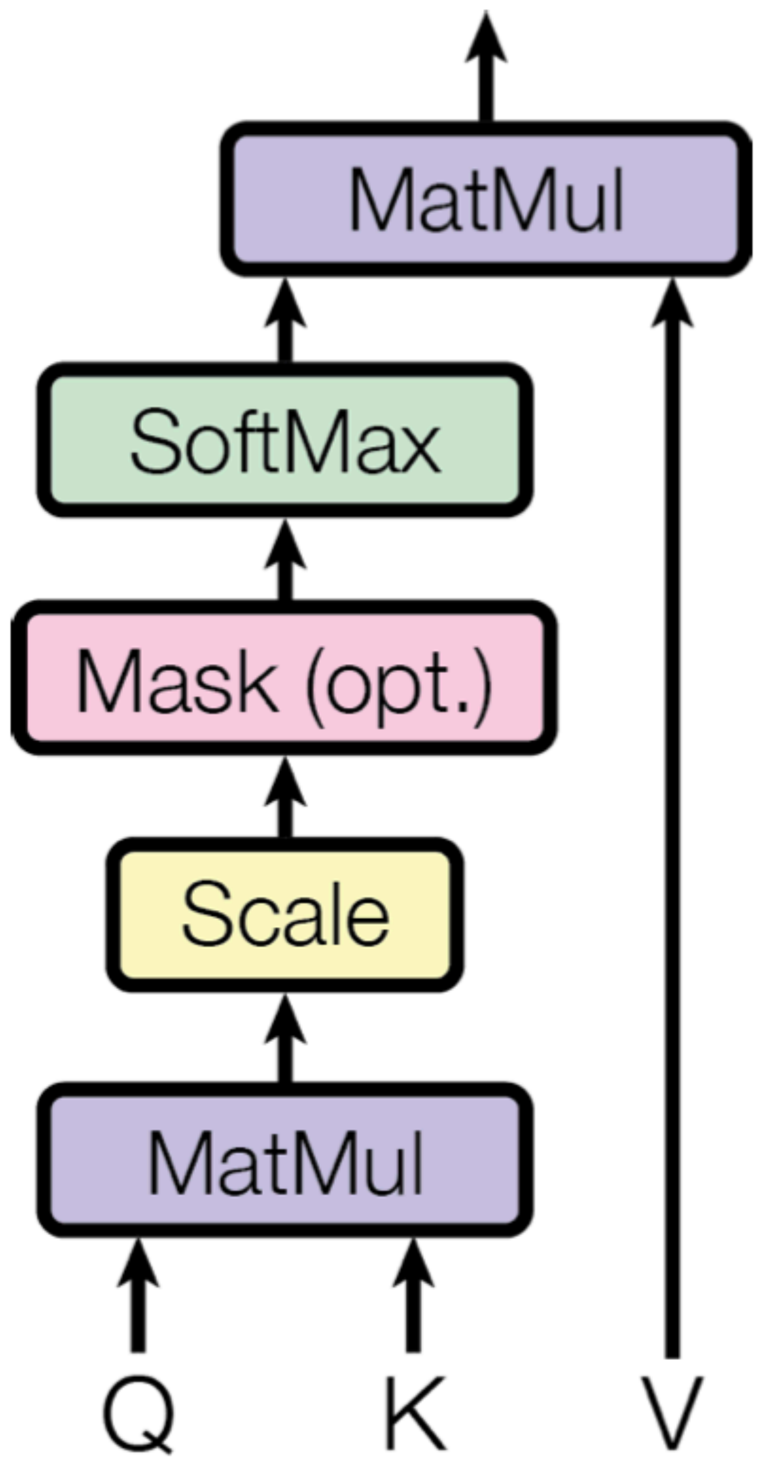
Dot-products grow large in magnitude

Scaled Attention weight = softmax$\left(\frac{QK^T}{\sqrt{d_k}}\right)$ Shape is mxn

Attention weight represents the strength to “attend” values V

What is Attention

Scaled Dot-Product Attention



Q: Query
K: key
V: value

$Q \in R^{n \times d}$ $K \in R^{m \times d}$ $V \in R^{m \times d}$

We have n queries, m (key, value) pairs

Attention weight = softmax(QK^T)

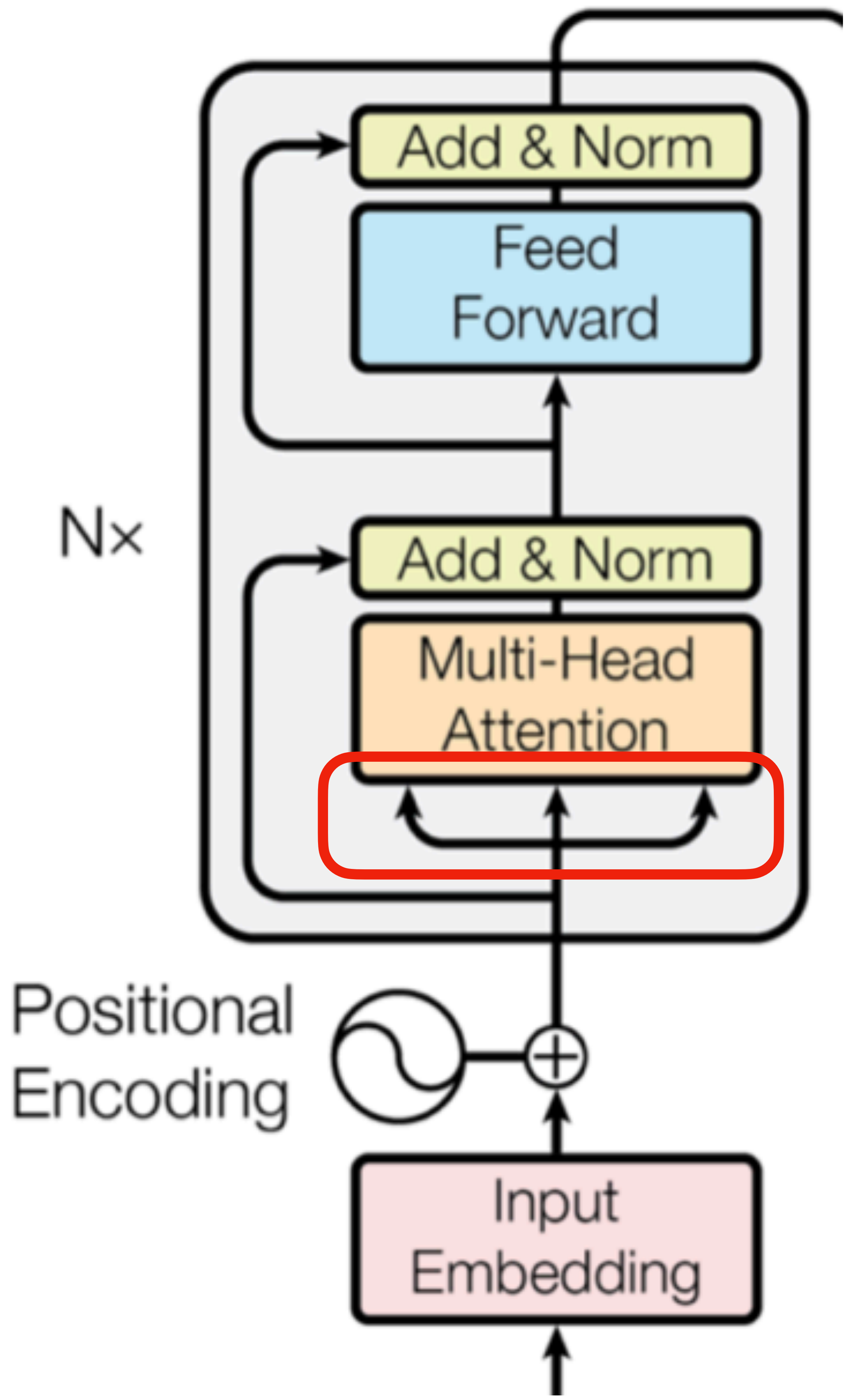
Dot-products grow large in magnitude

Scaled Attention weight = softmax$(\frac{QK^T}{\sqrt{d_k}})$ Shape is mxn

Attention weight represents the strength to "attend" values V

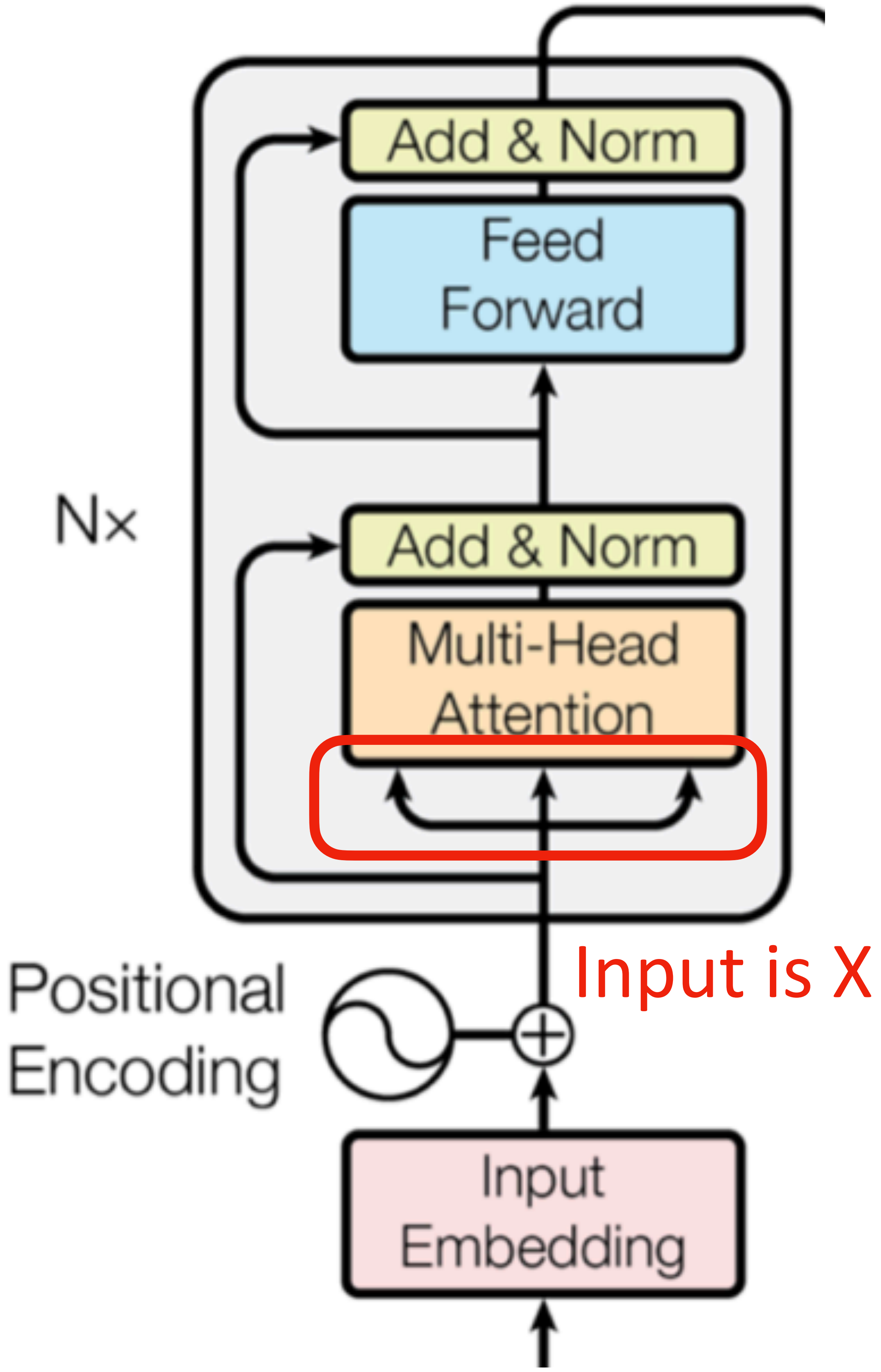
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

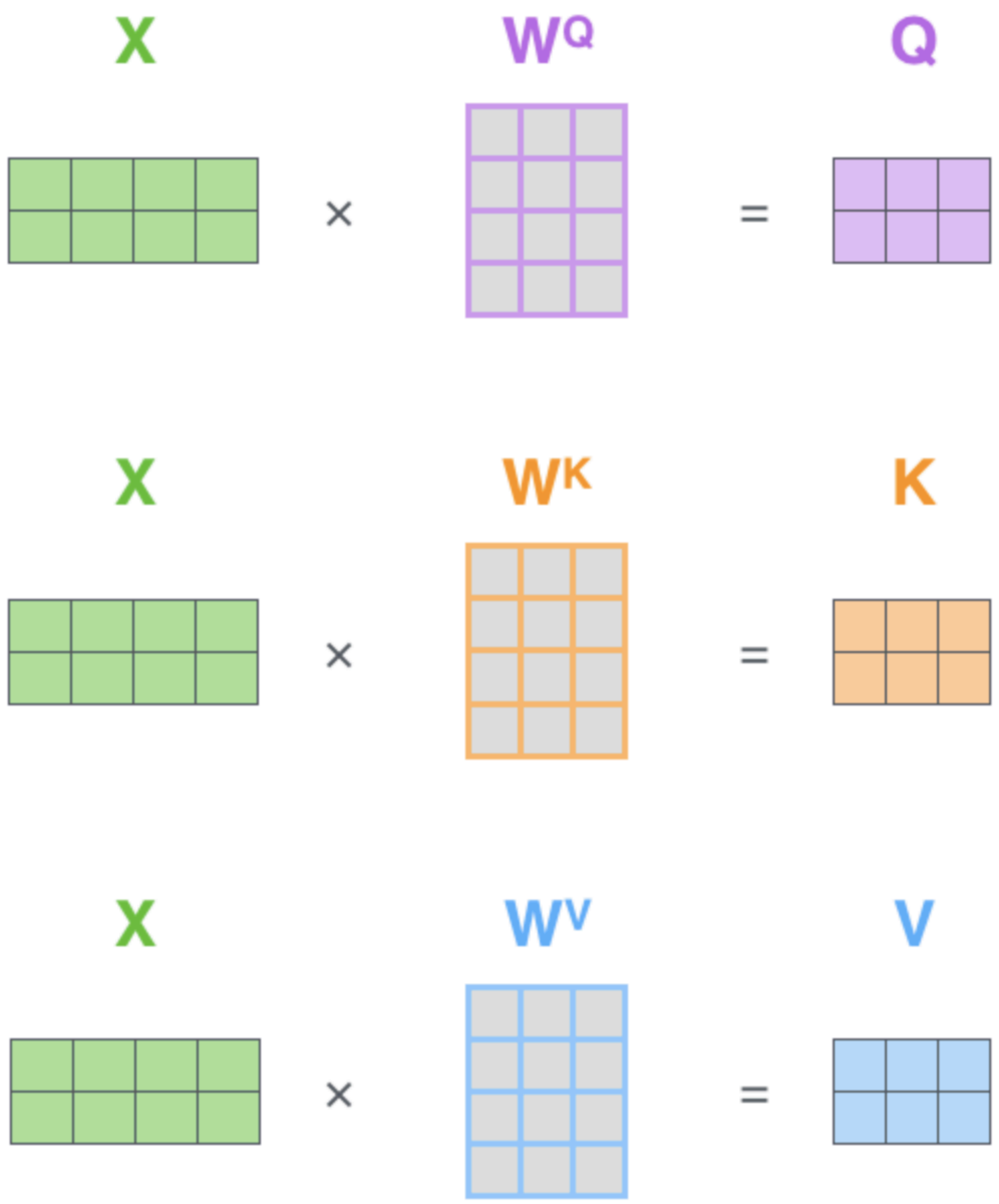
Q, K, V



What are Q, K, V in the transformer

Self-Attention

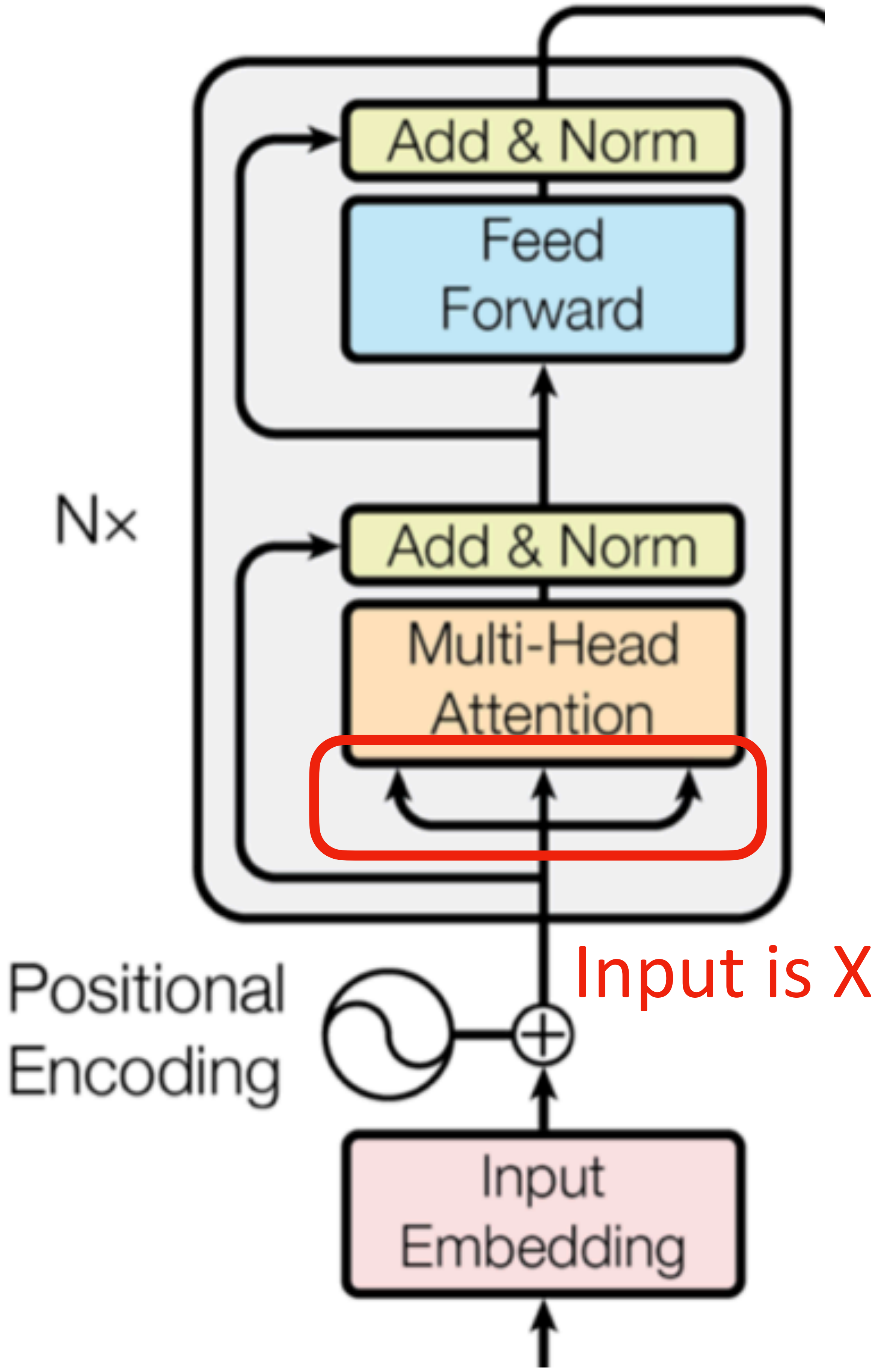


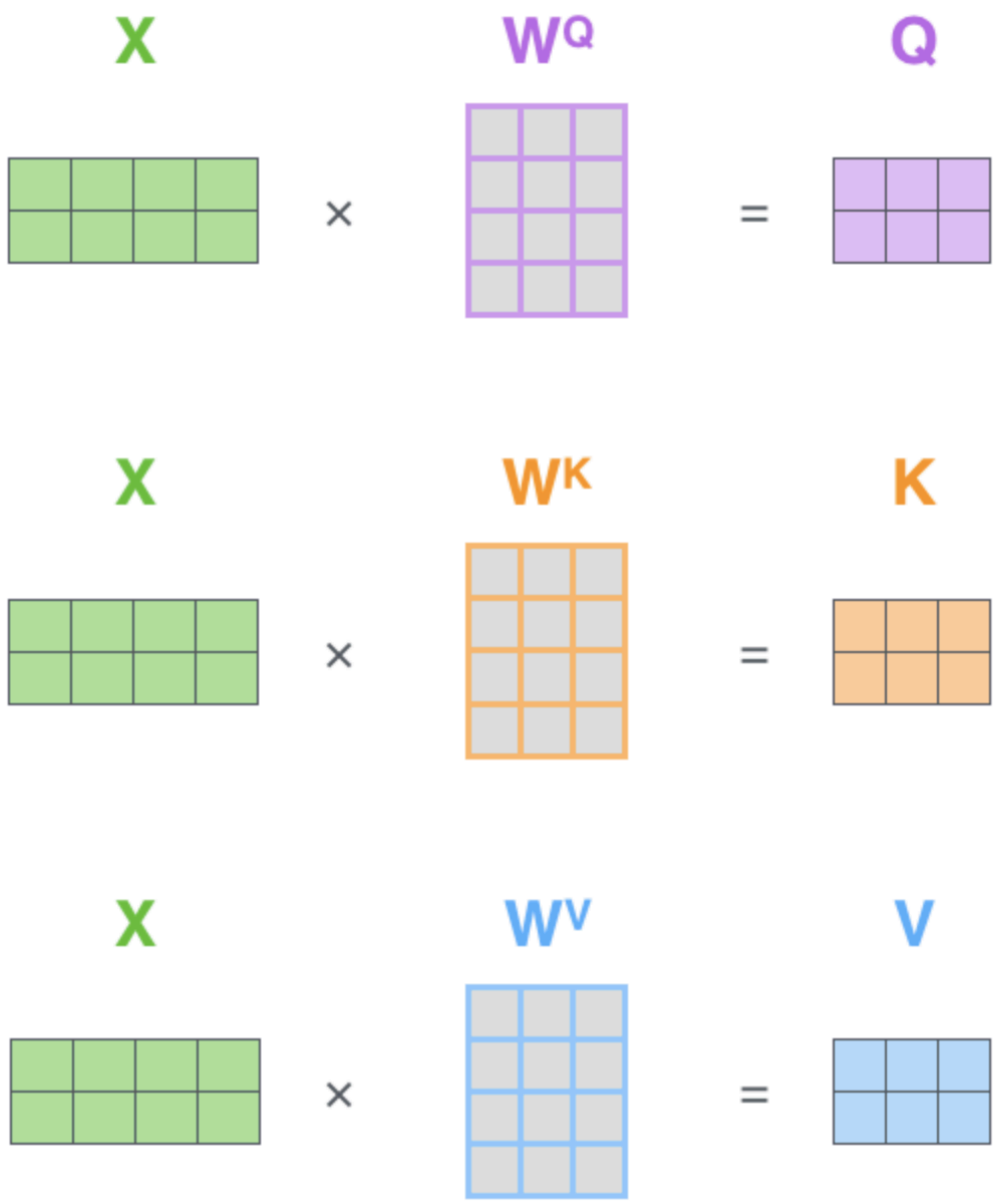


35

Jay Alammar. The Illustrated Transformer.

Self-Attention



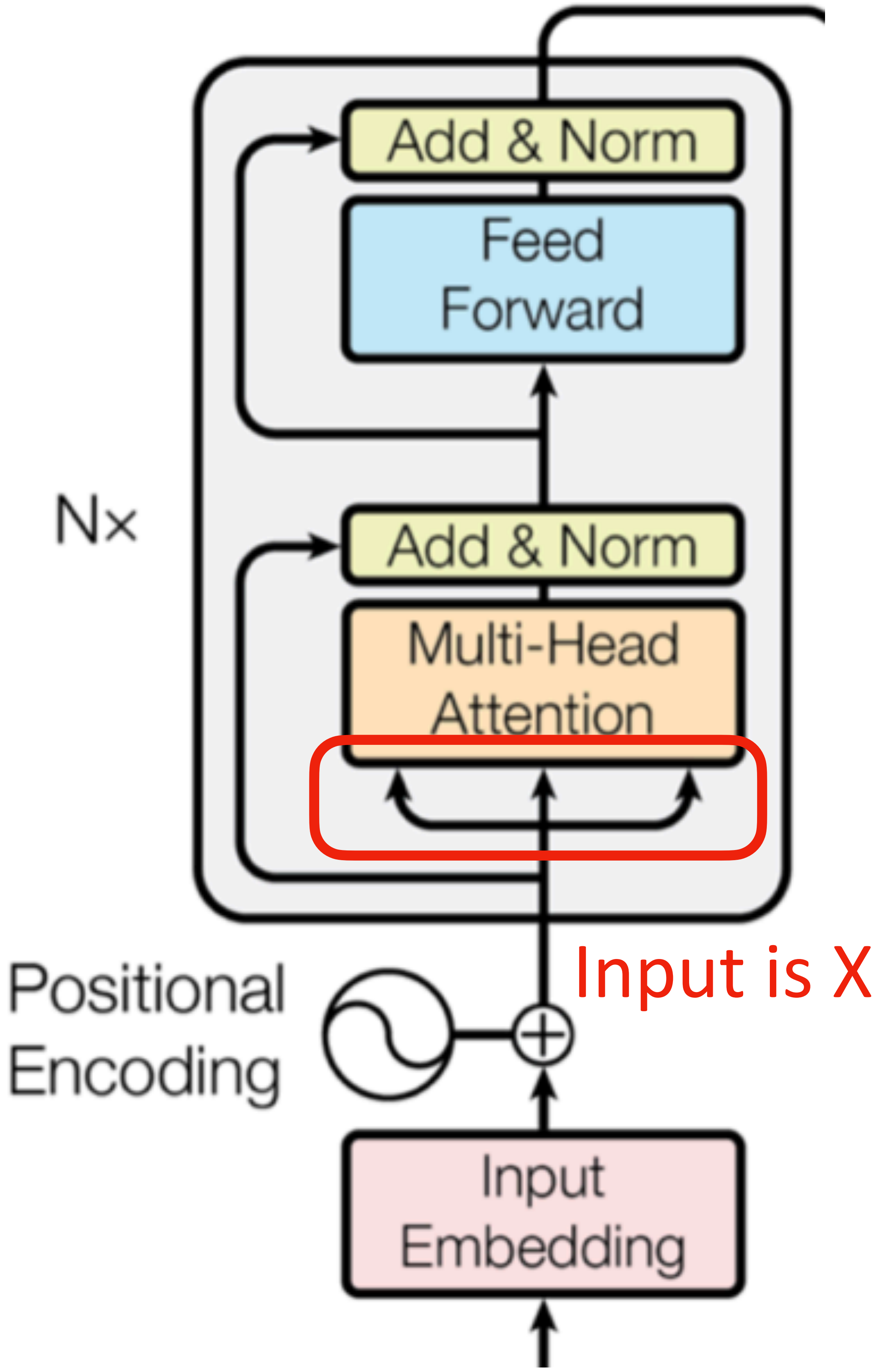


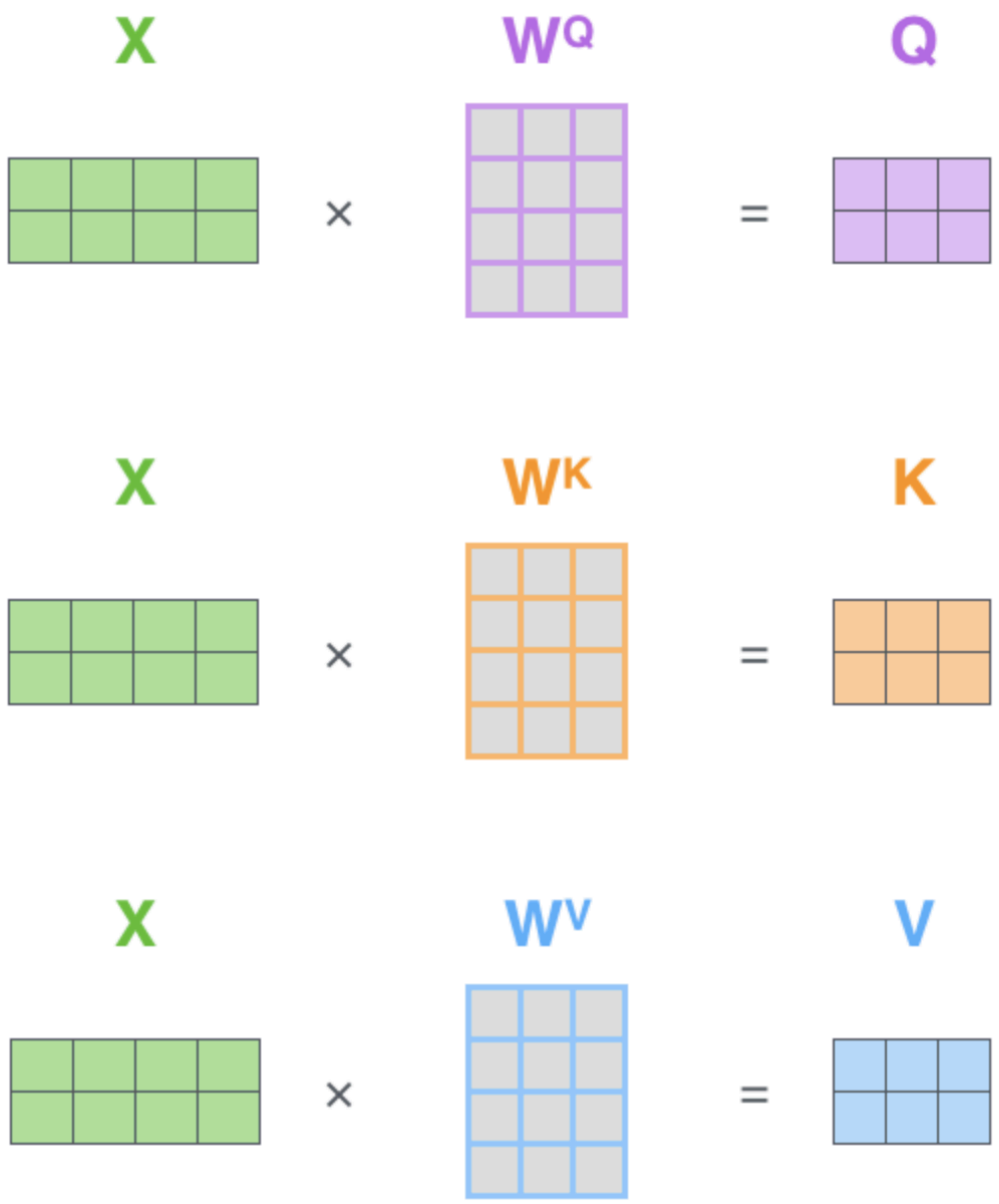
Query, key, and value are from the same input, thus it is called “self”-attention

35

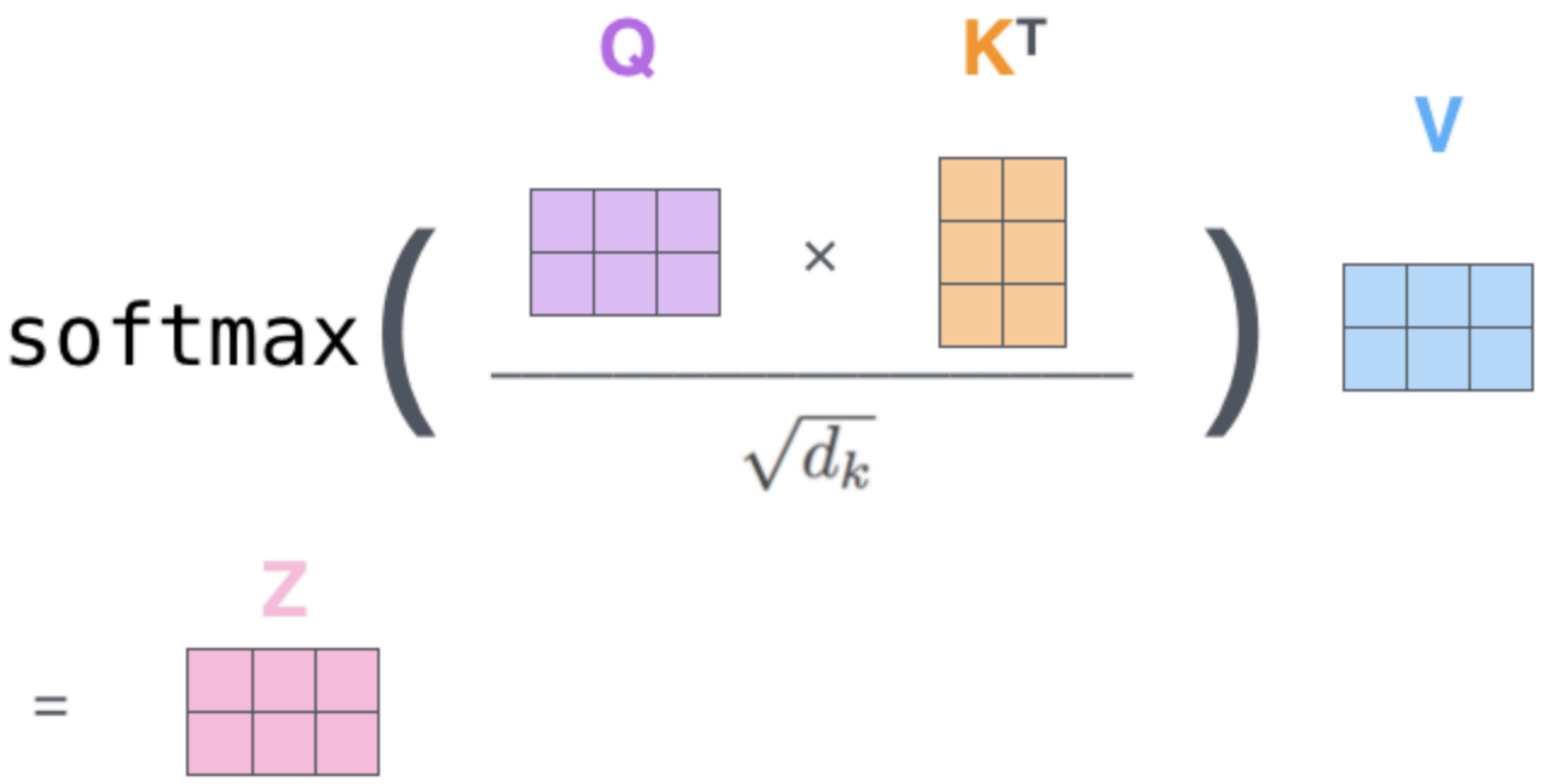
Jay Alammar. The Illustrated Transformer.

Self-Attention





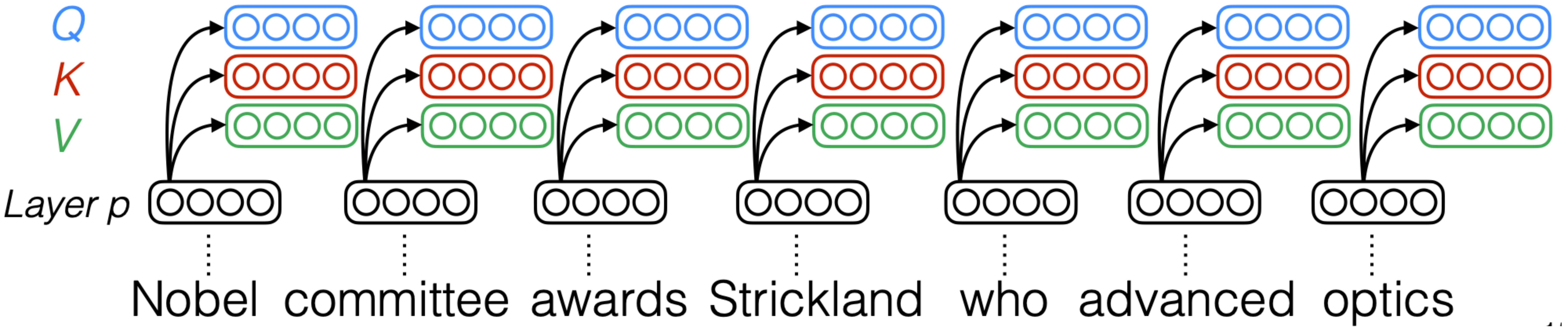
Query, key, and value are from the same input, thus it is called “self”-attention



Jay Alammar. The Illustrated Transformer.

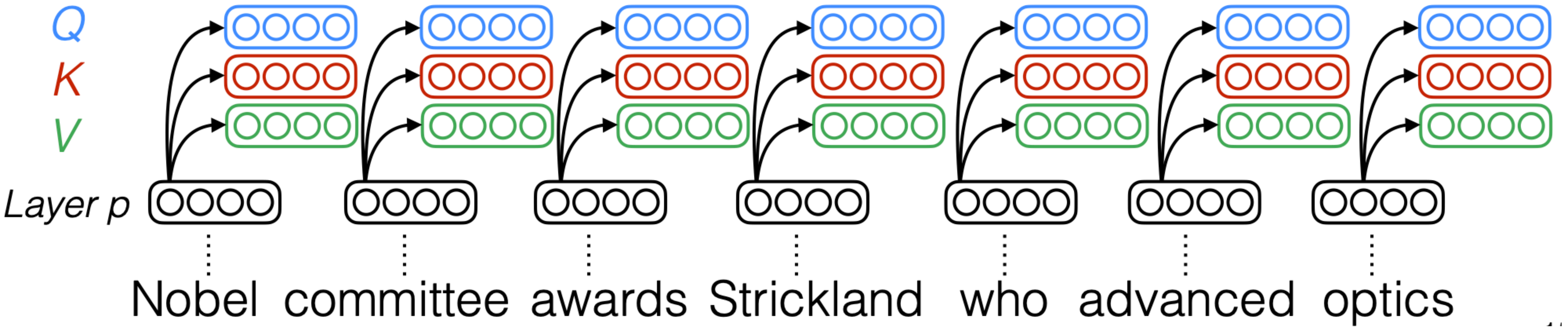
35

Self-Attention

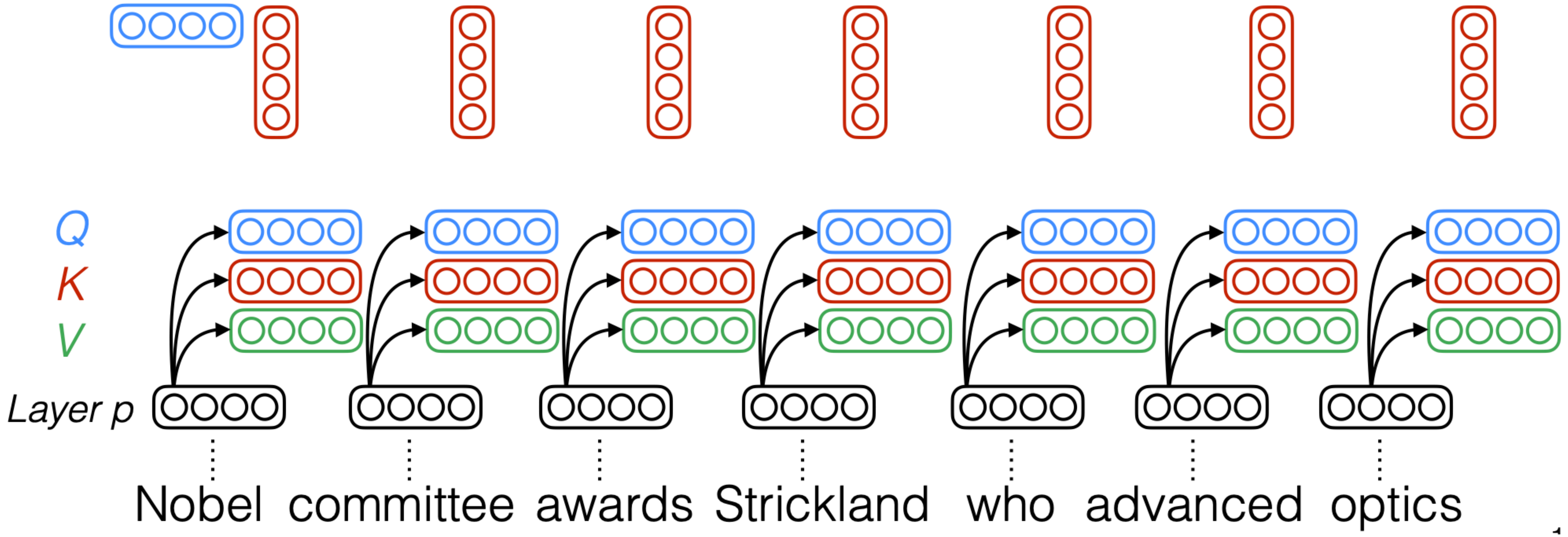


Self-Attention

At each step, the attention computation attends to all steps in the input example

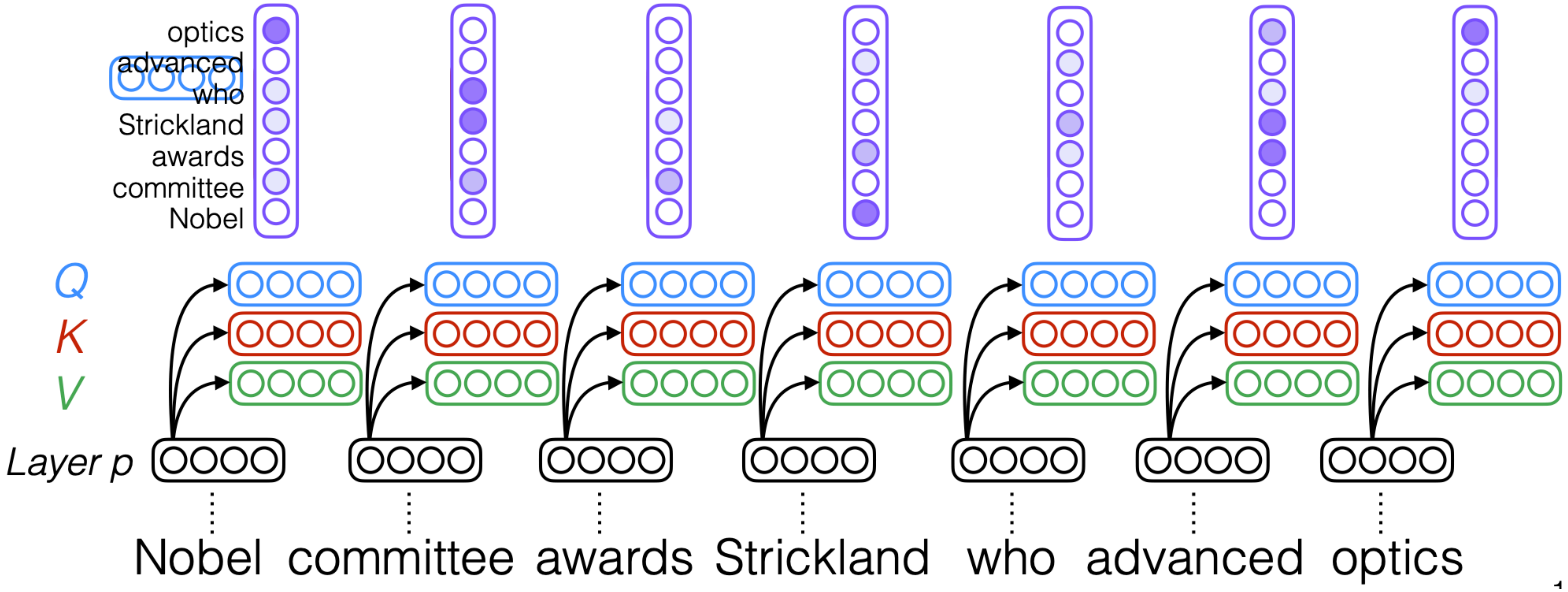


Self-Attention

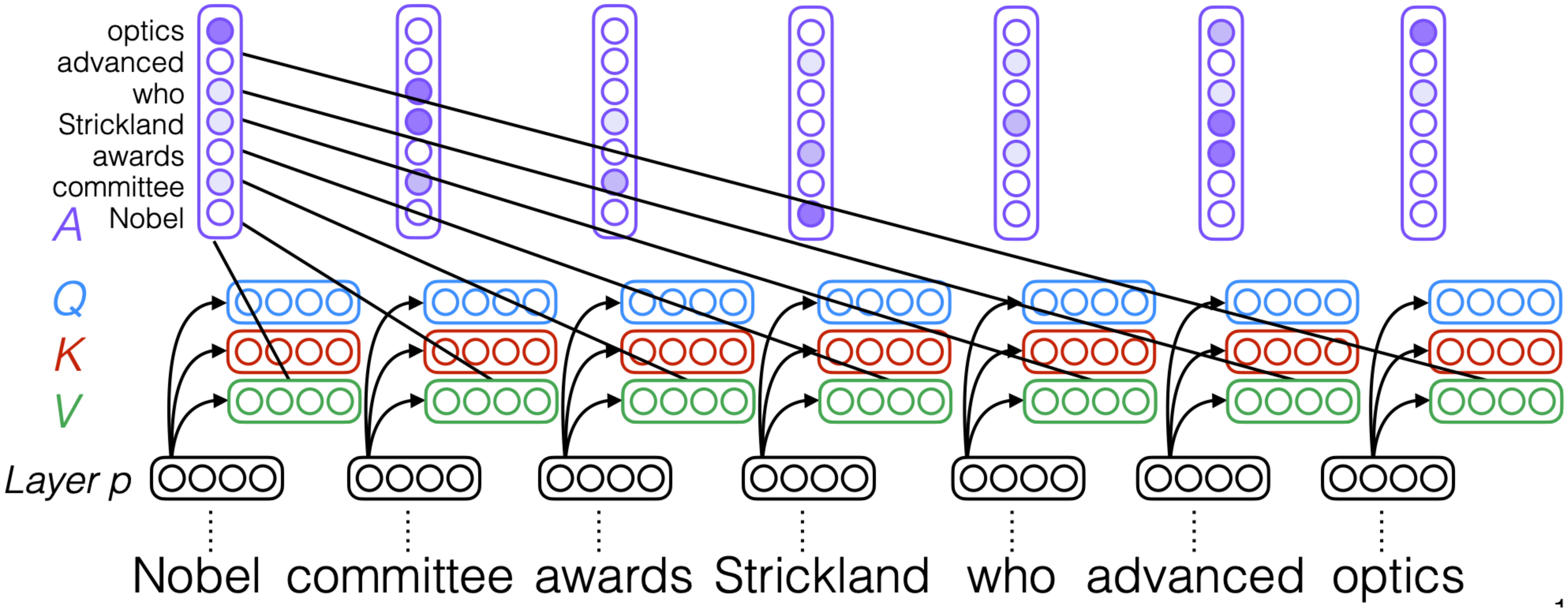


Self-Attention

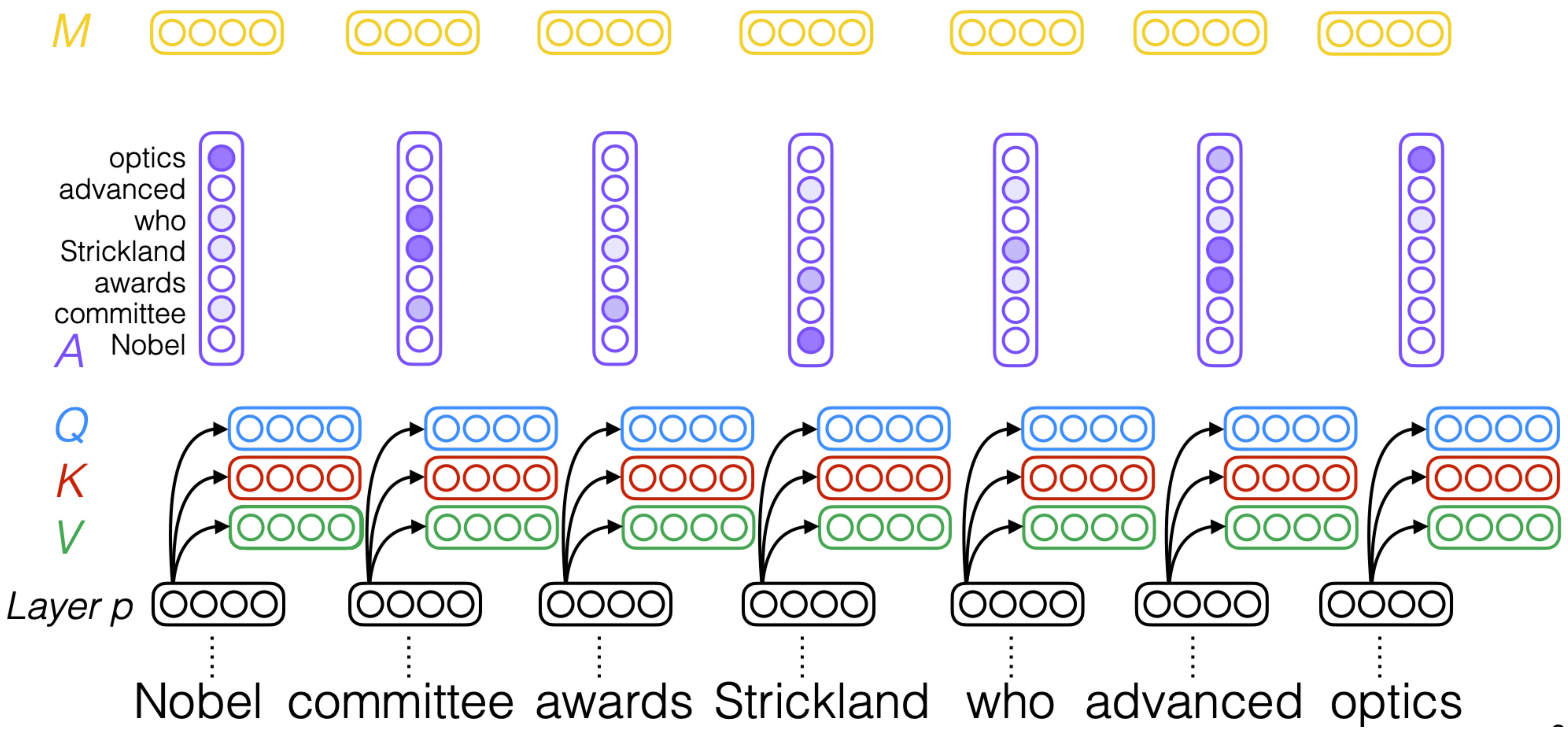
Attention weight on every word in the sequence



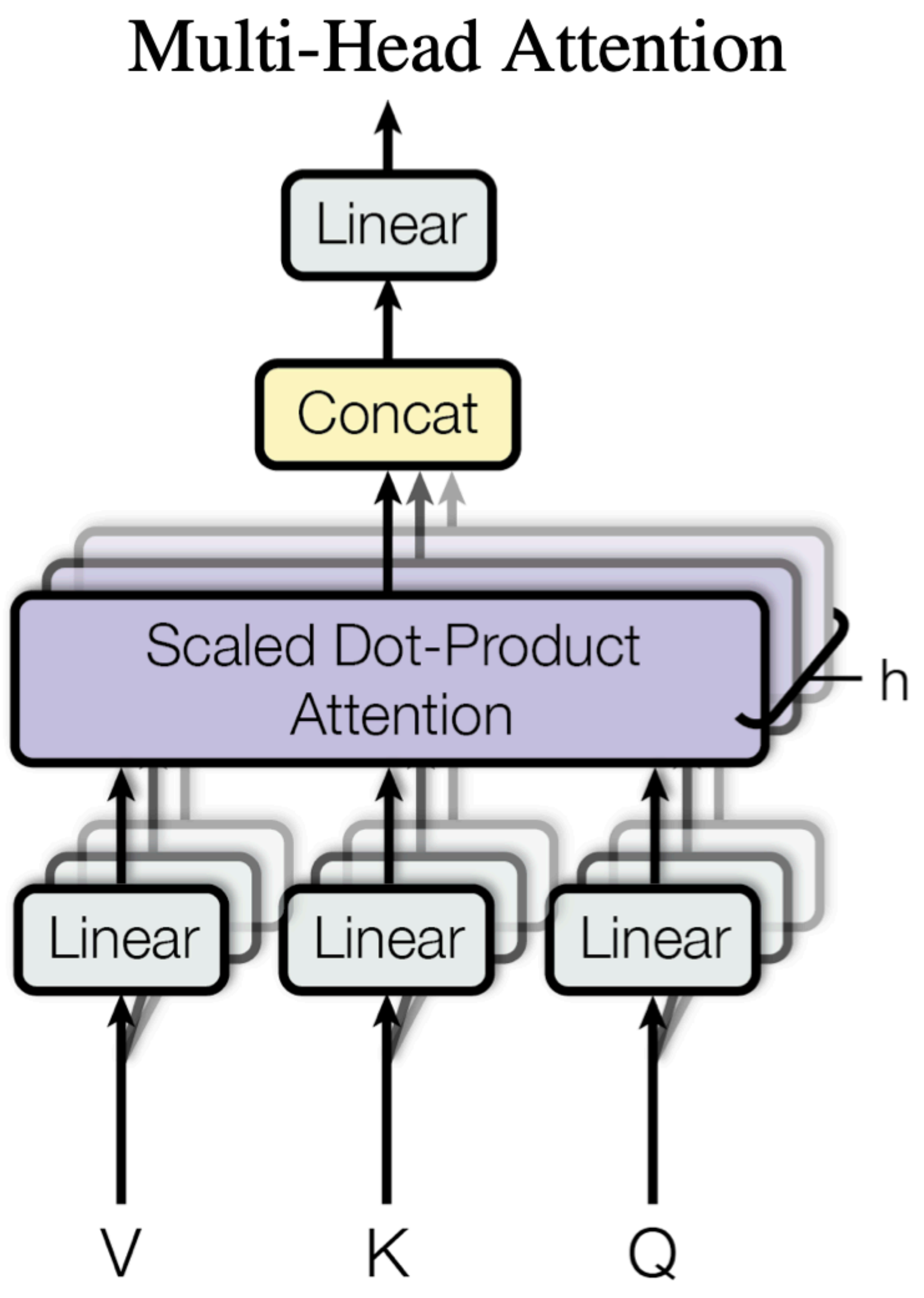
Self-Attention



Self-Attention



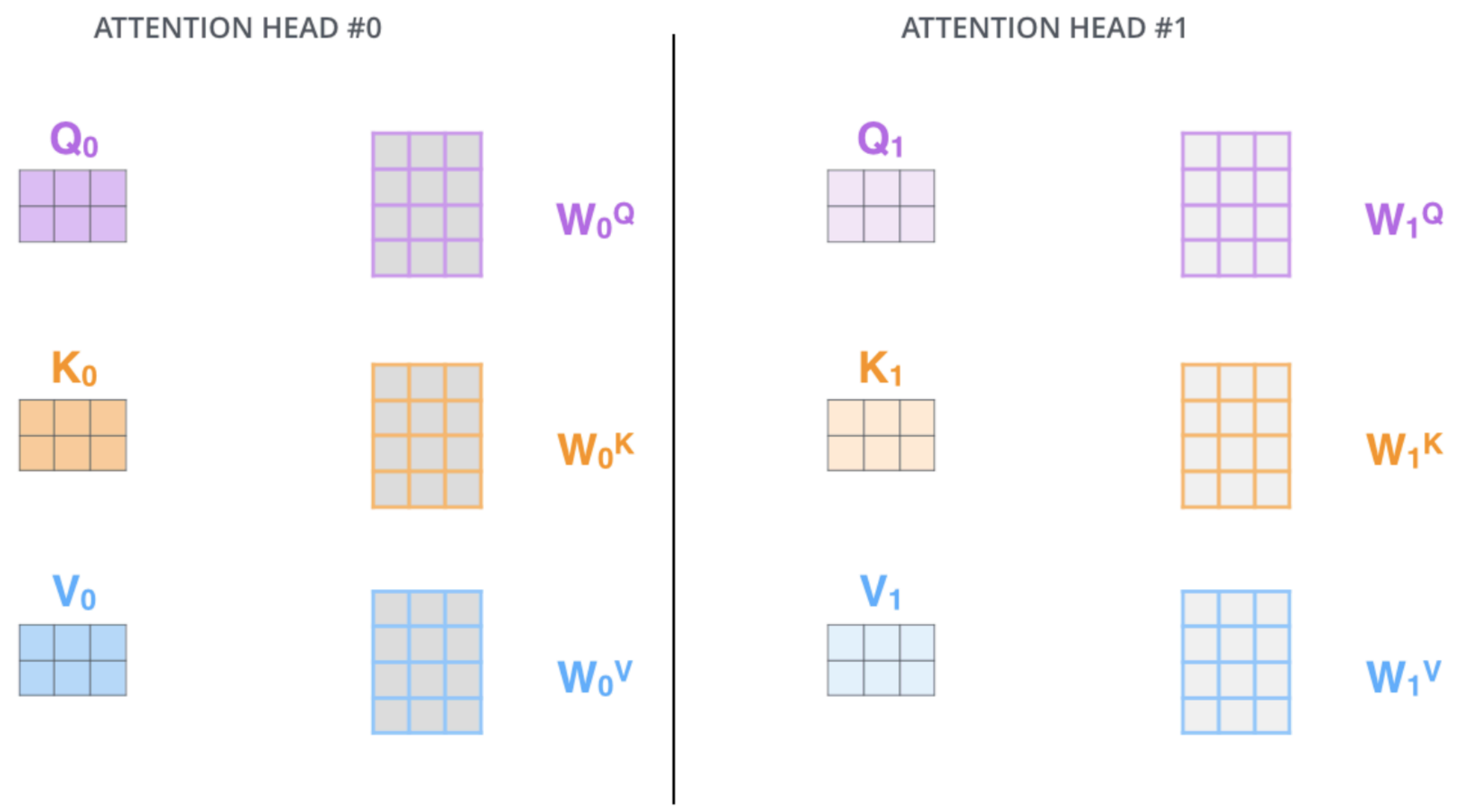
Multi-Head Attention



Multi-Head Self-Attention

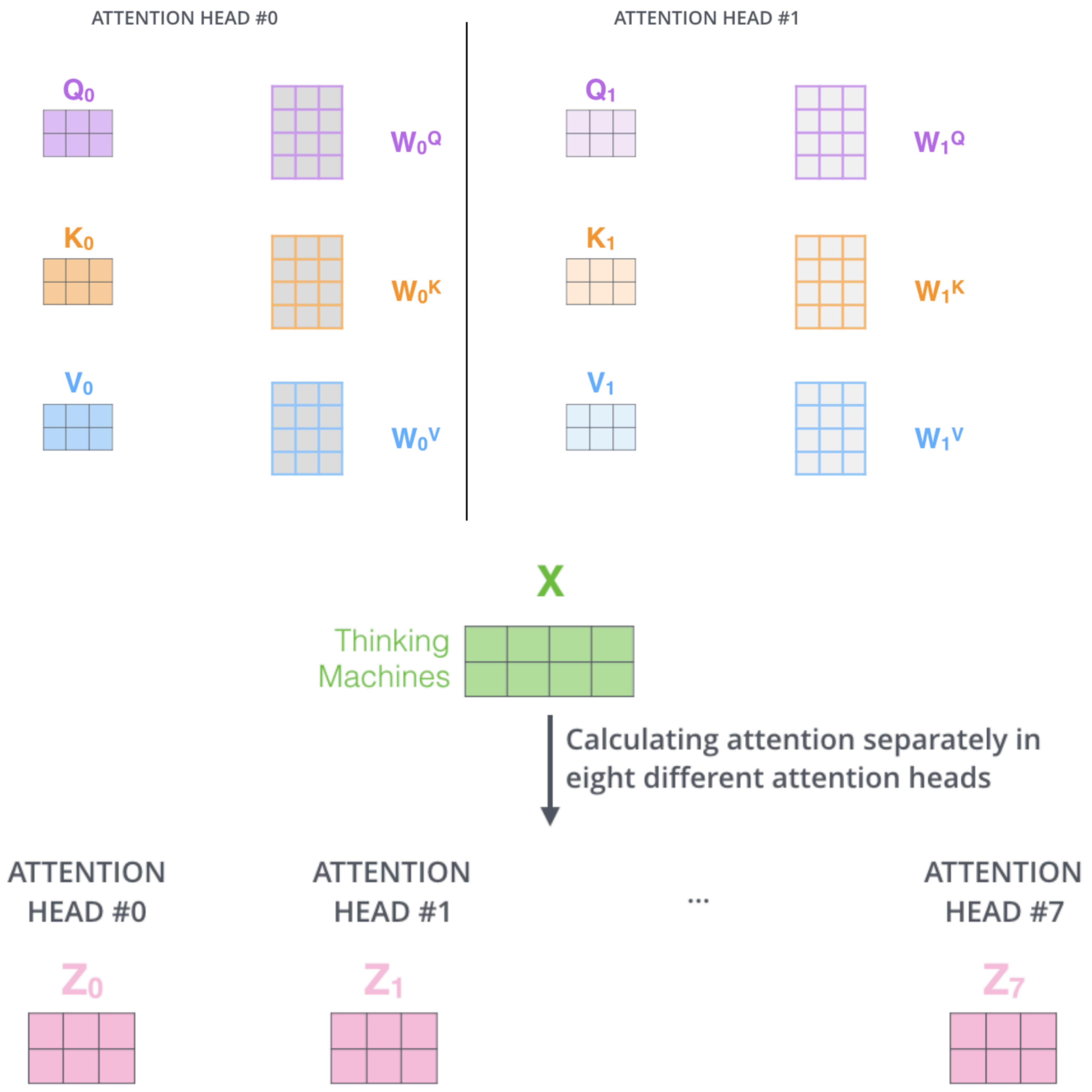
Jay Alammar. The Illustrated Transformer.

Multi-Head Self-Attention



Jay Alammar. The Illustrated Transformer.

Multi-Head Self-Attention



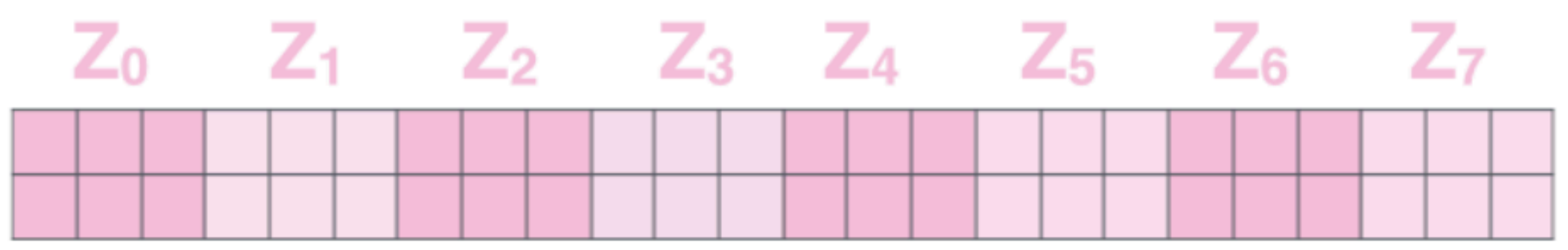
Jay Alammar. The Illustrated Transformer.

Multi-Head Self-Attention

Jay Alammar. The Illustrated Transformer.

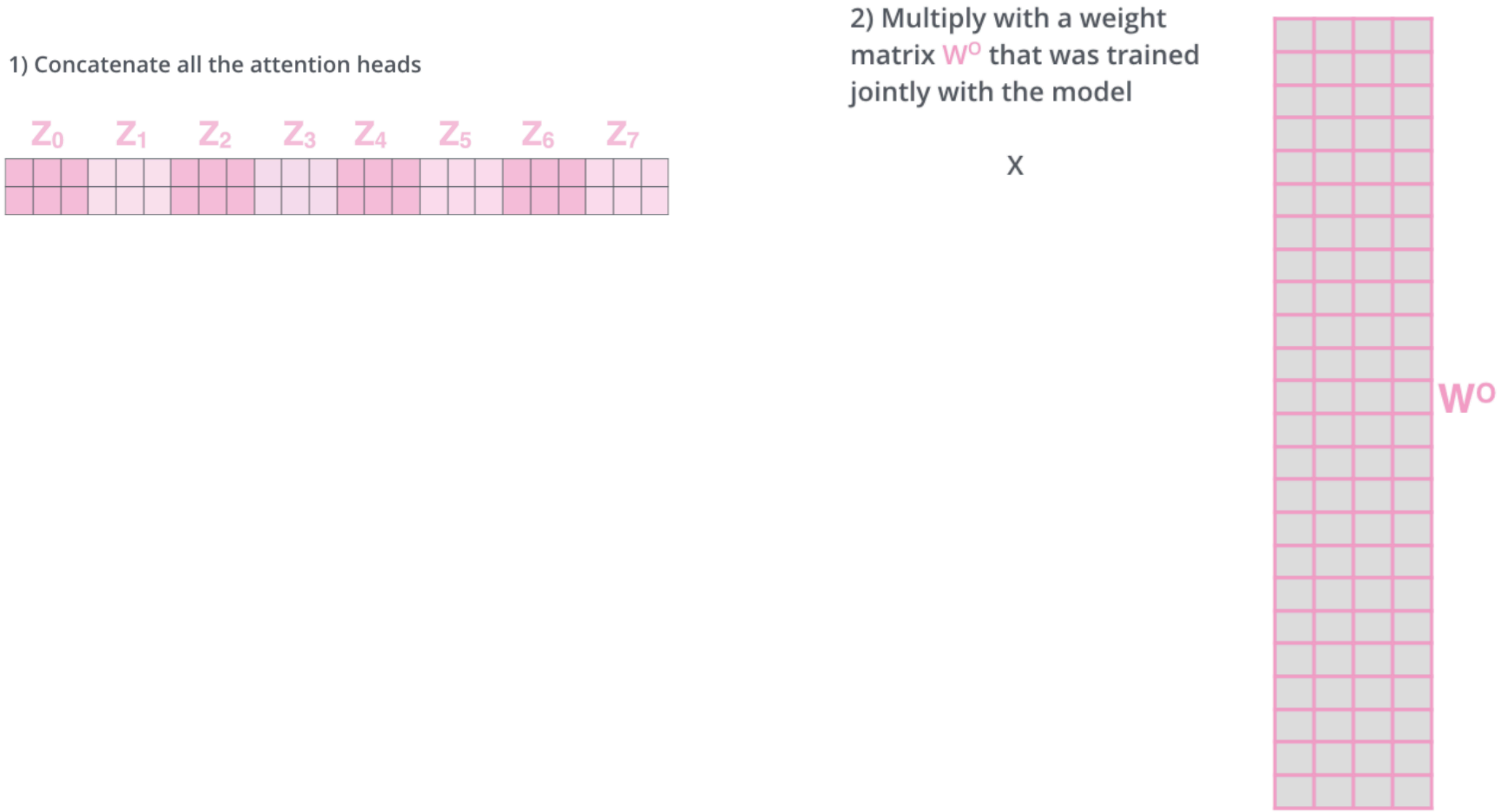
Multi-Head Self-Attention

1) Concatenate all the attention heads



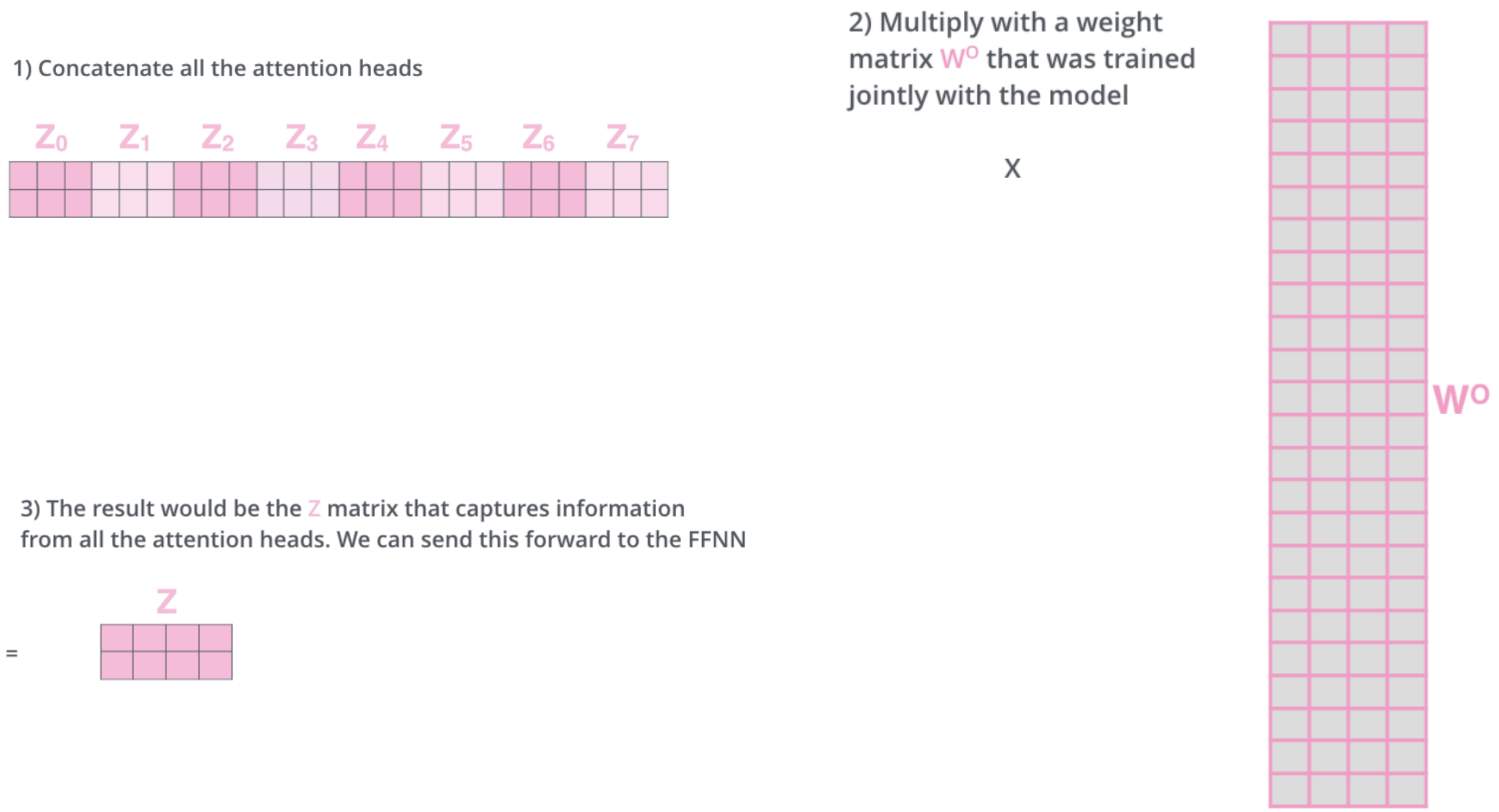
Jay Alammar. The Illustrated Transformer.

Multi-Head Self-Attention



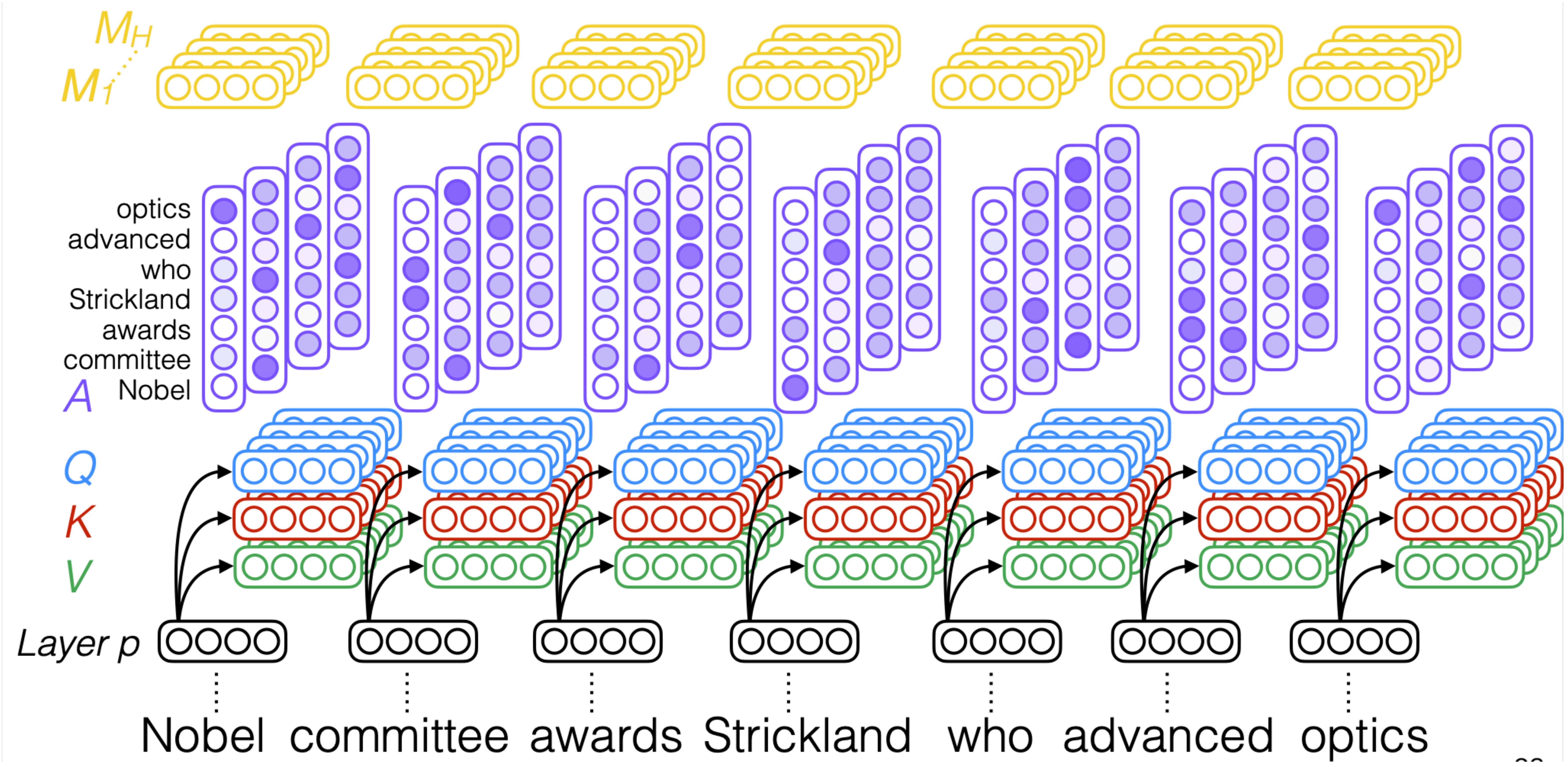
Jay Alammar. The Illustrated Transformer.

Multi-Head Self-Attention



Jay Alammar. The Illustrated Transformer.

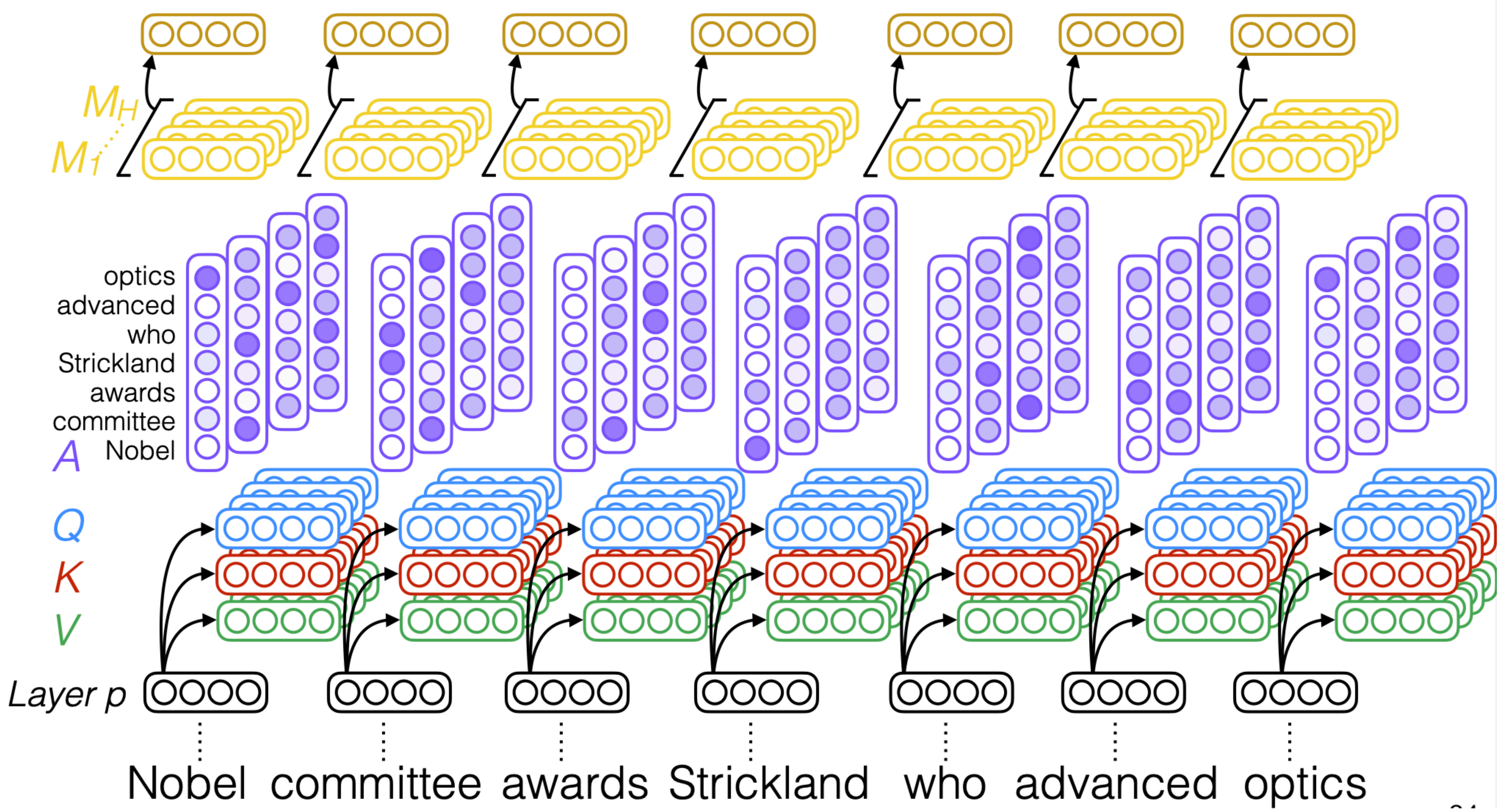
Multi-head Self-Attention



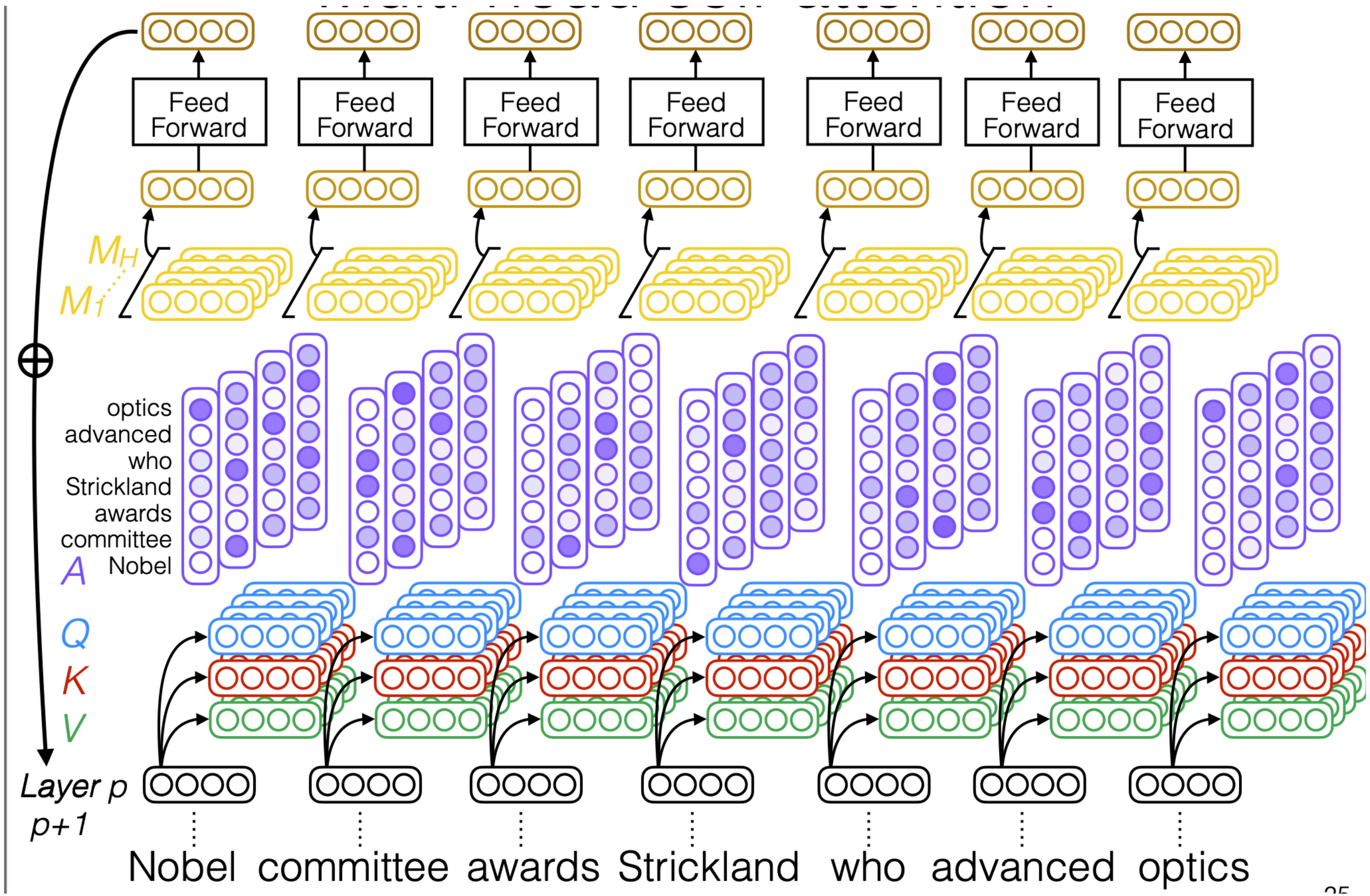
Slides by Emma Strubell

Multi-head Self-Attention

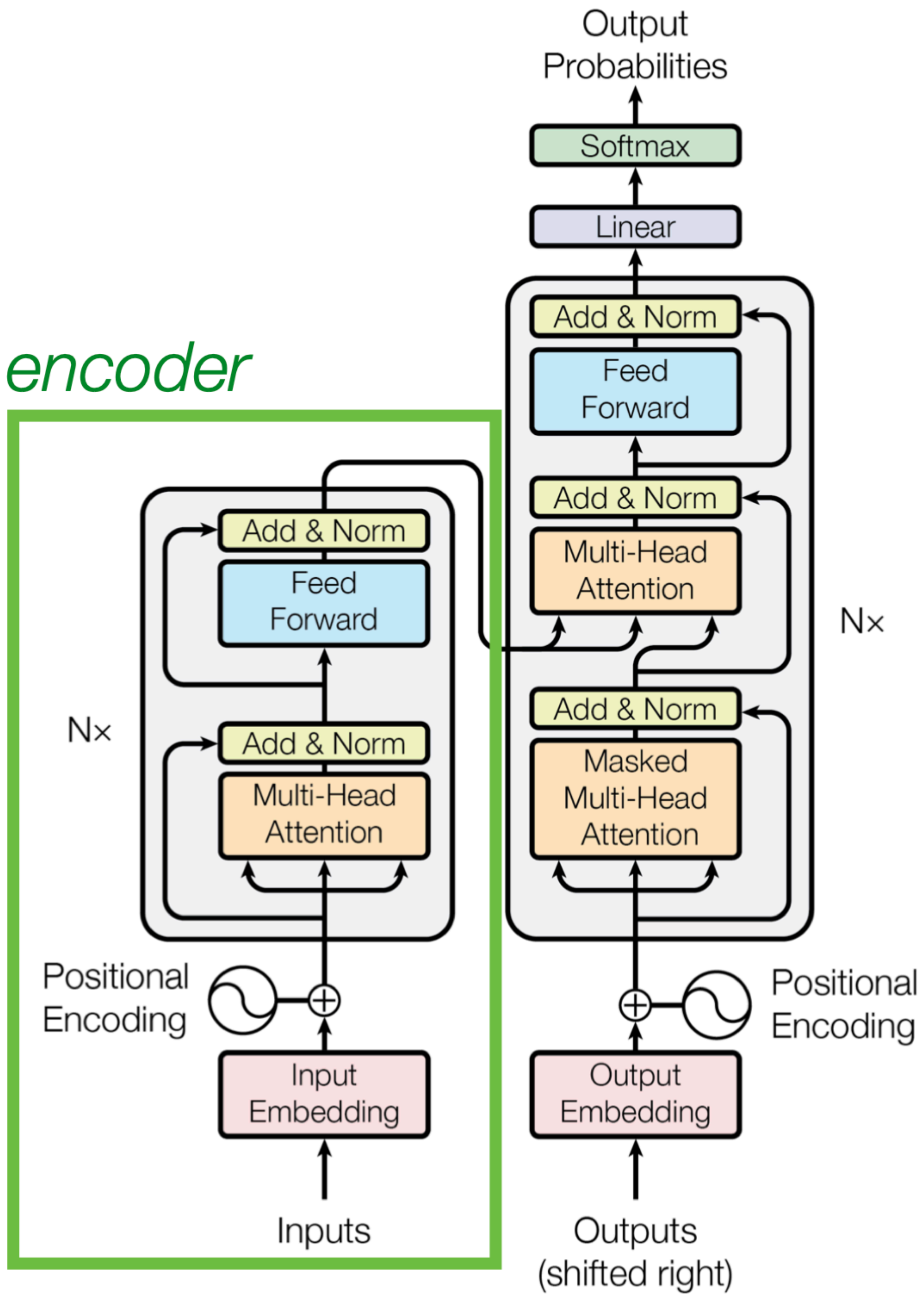
Concat and output projection



Multi-head Self-Attention + FFN

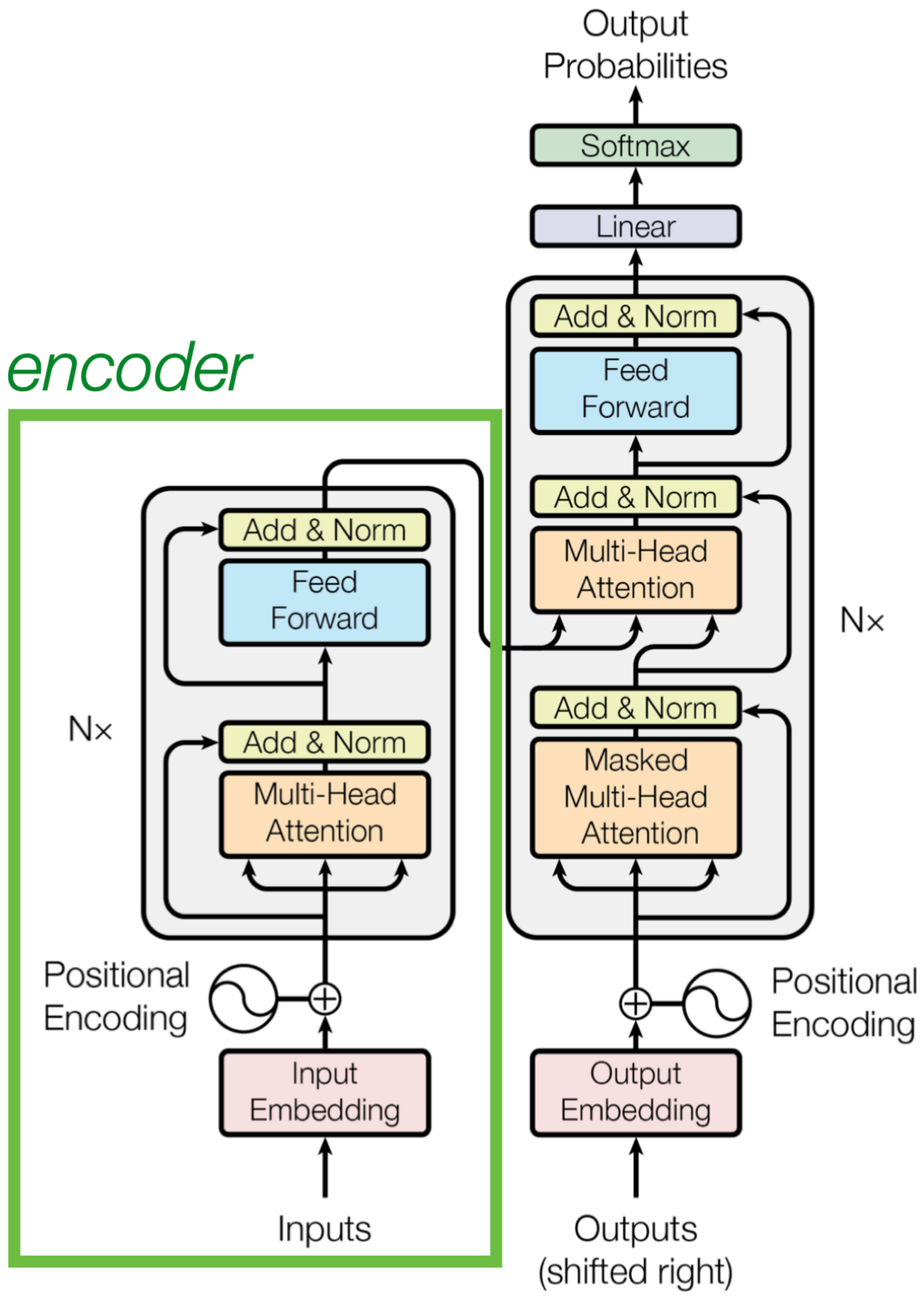


Transformer Encoder

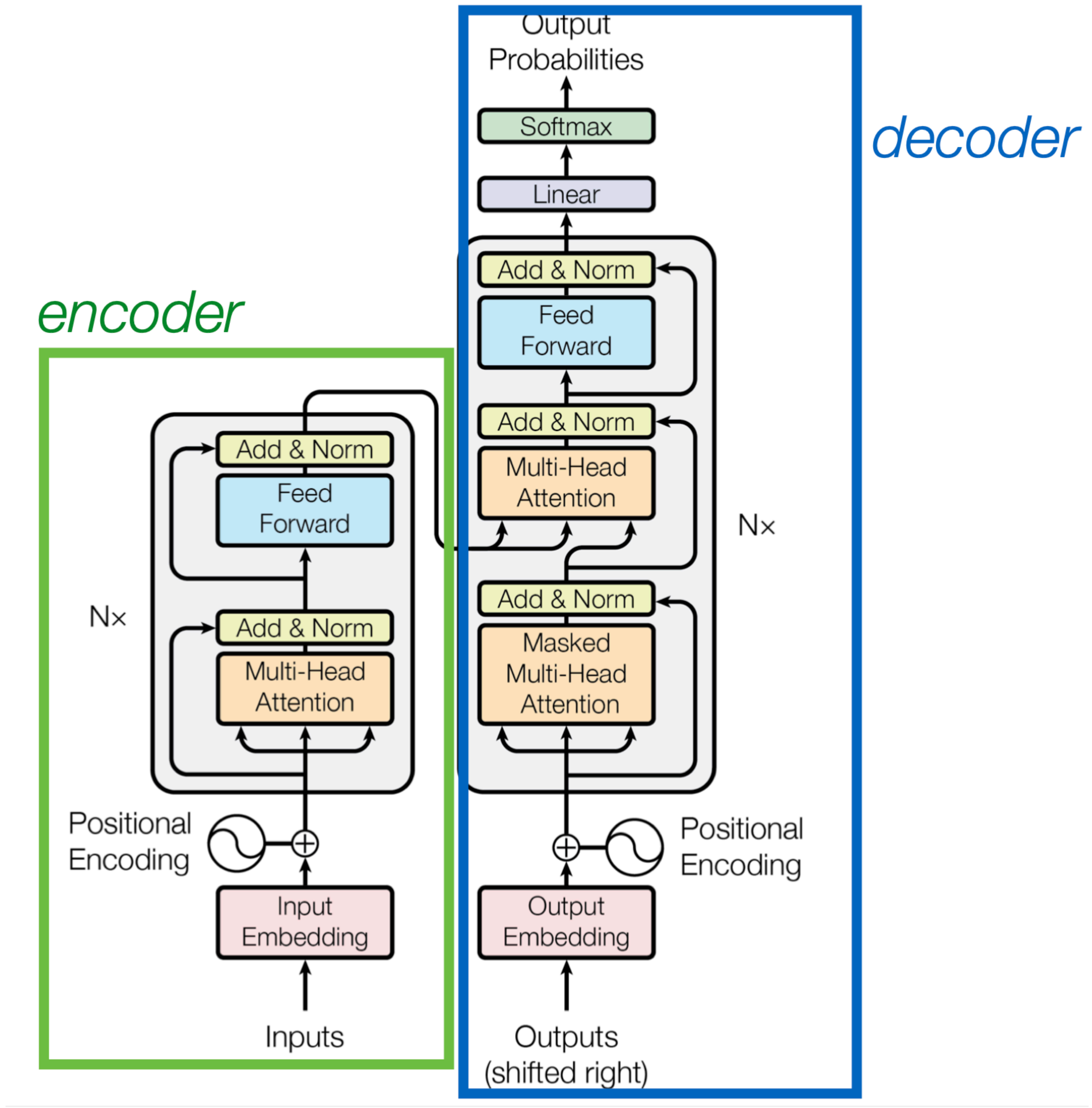


Currently we only cover the encoder side

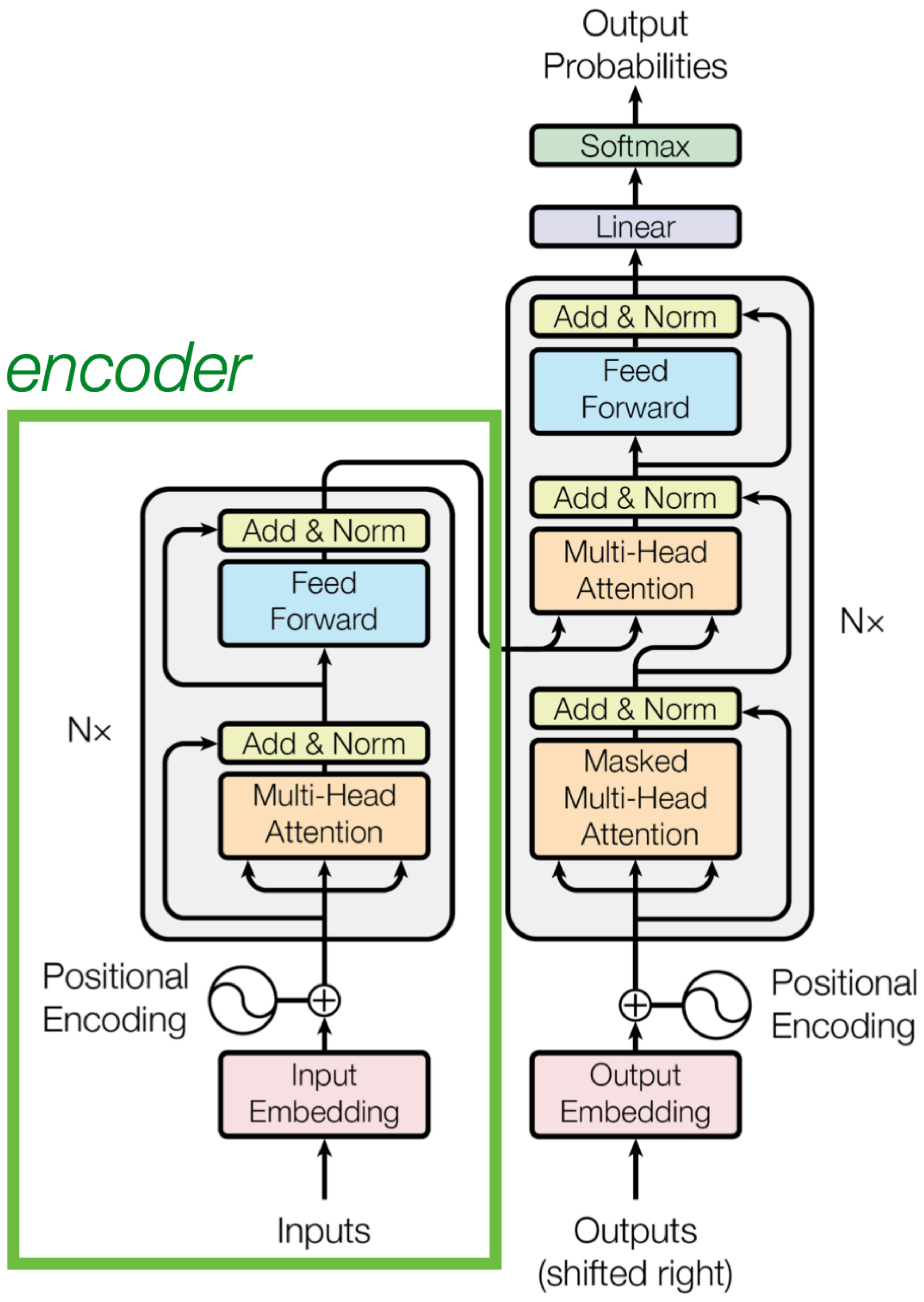
Transformer Encoder



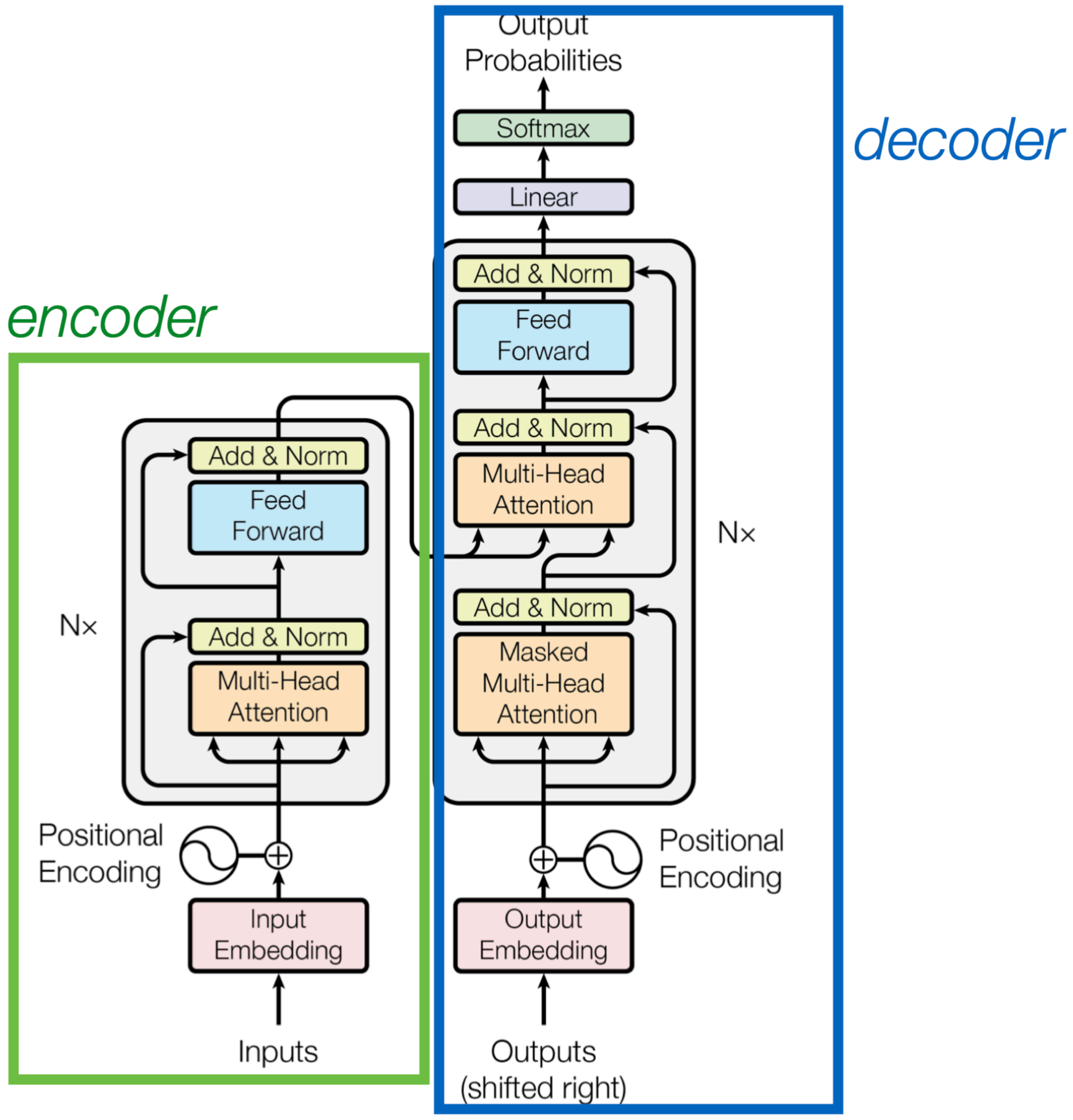
Currently we only cover the encoder side



Transformer Encoder

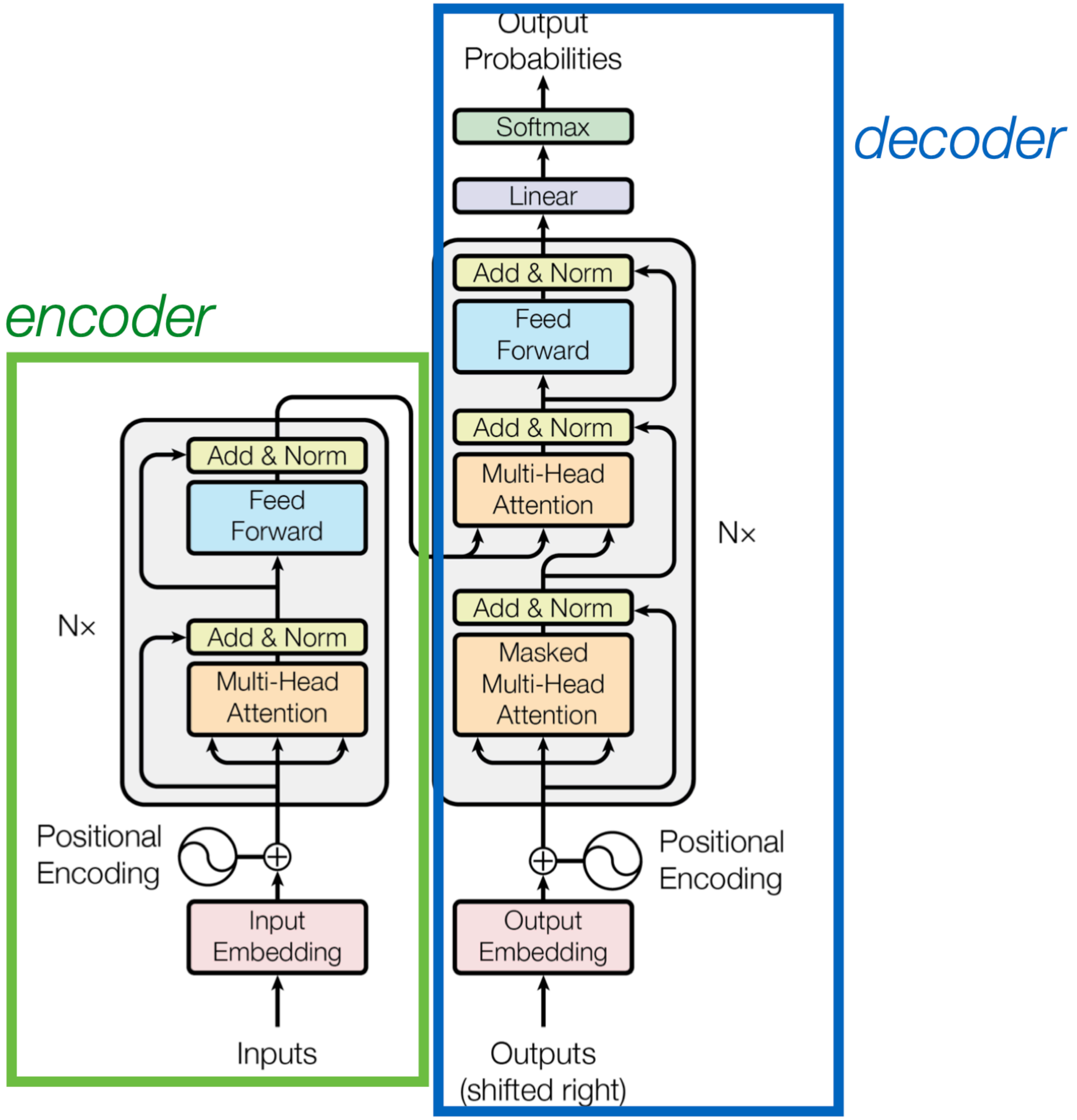


Currently we only cover the encoder side

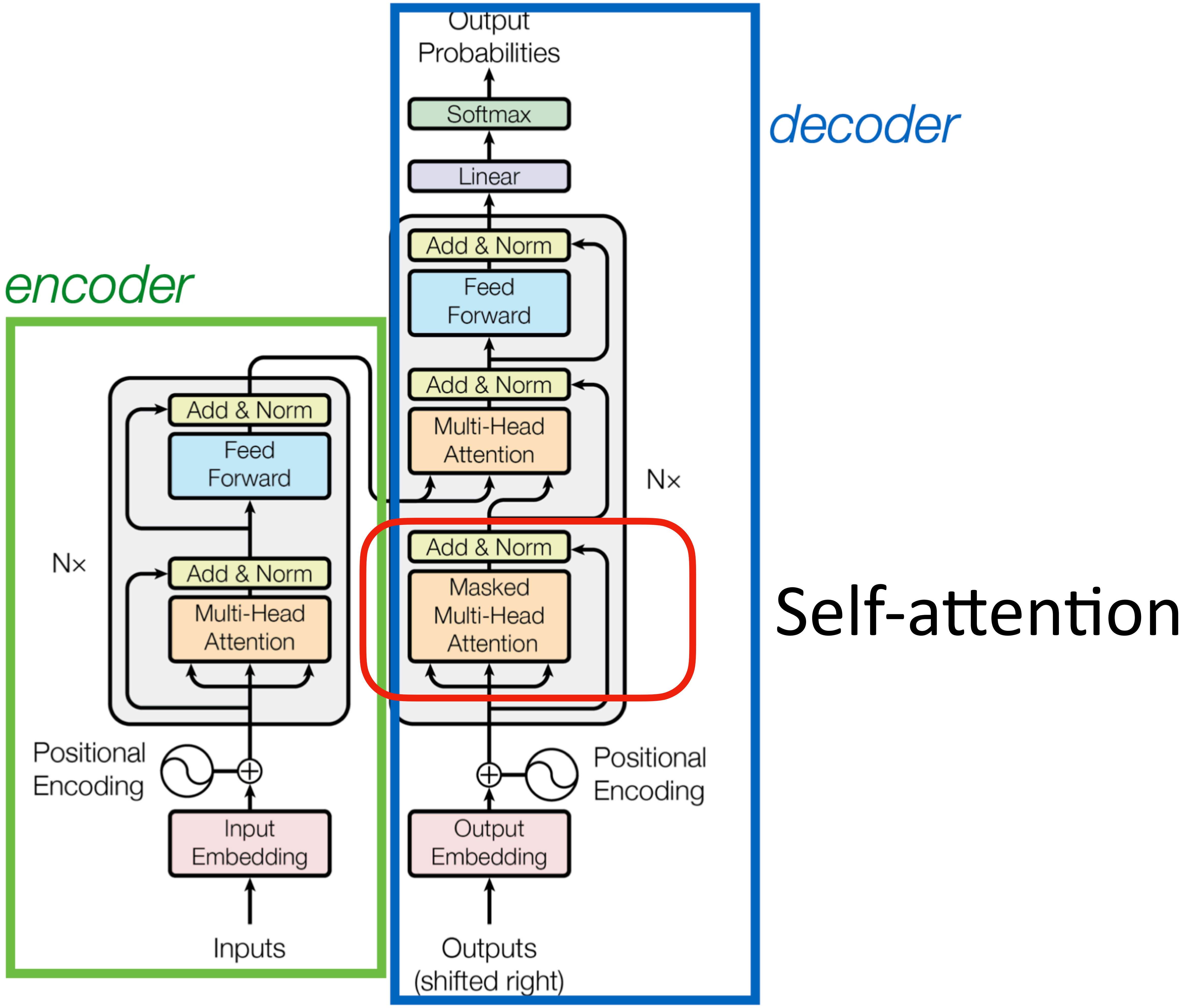


This encoder-decoder arch is originally proposed as a seq2seq arch, for classification tasks, often only encoder is used. And language models often only have a decoder

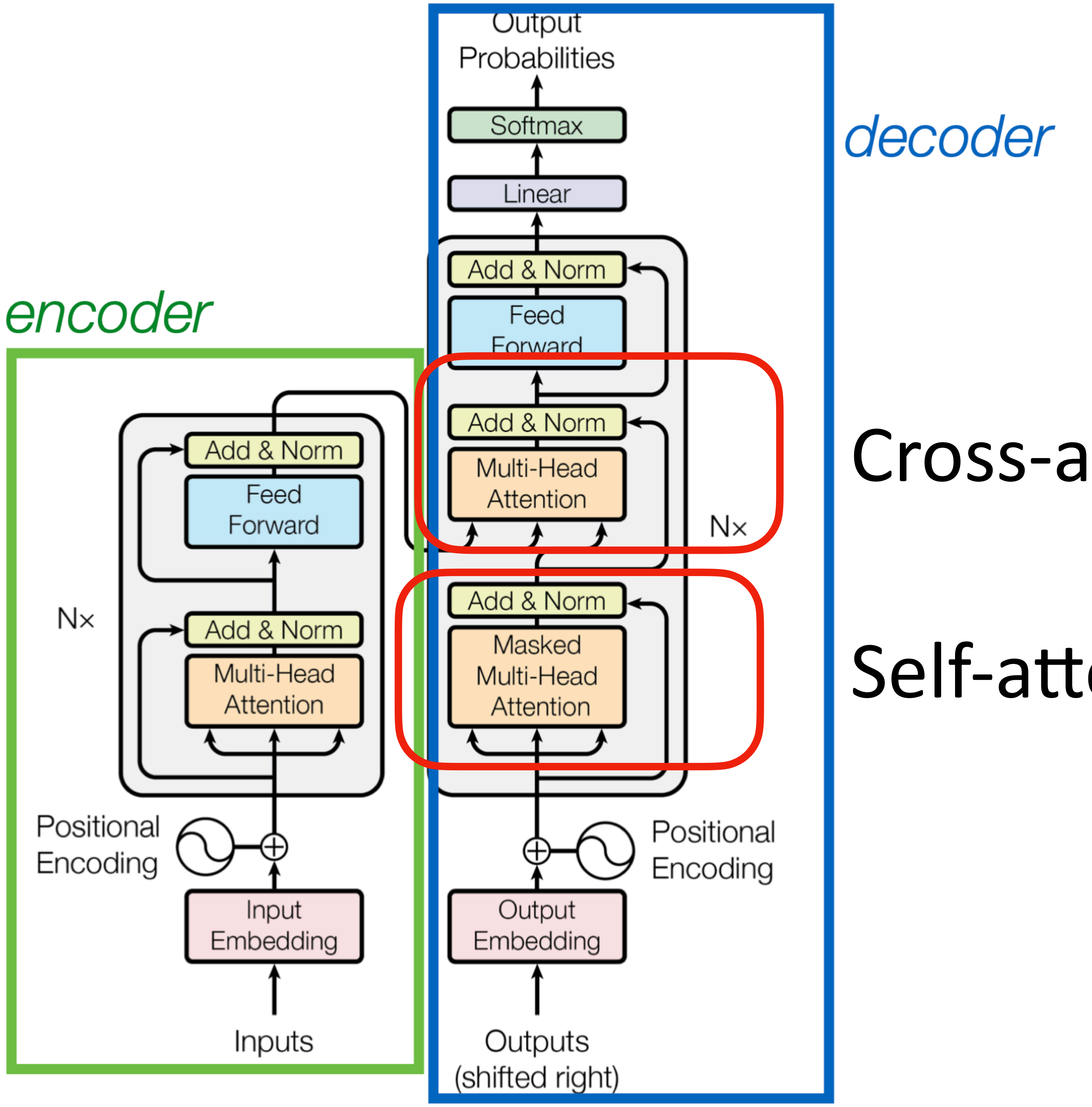
Transformer Decoder in Seq2Seq



Transformer Decoder in Seq2Seq



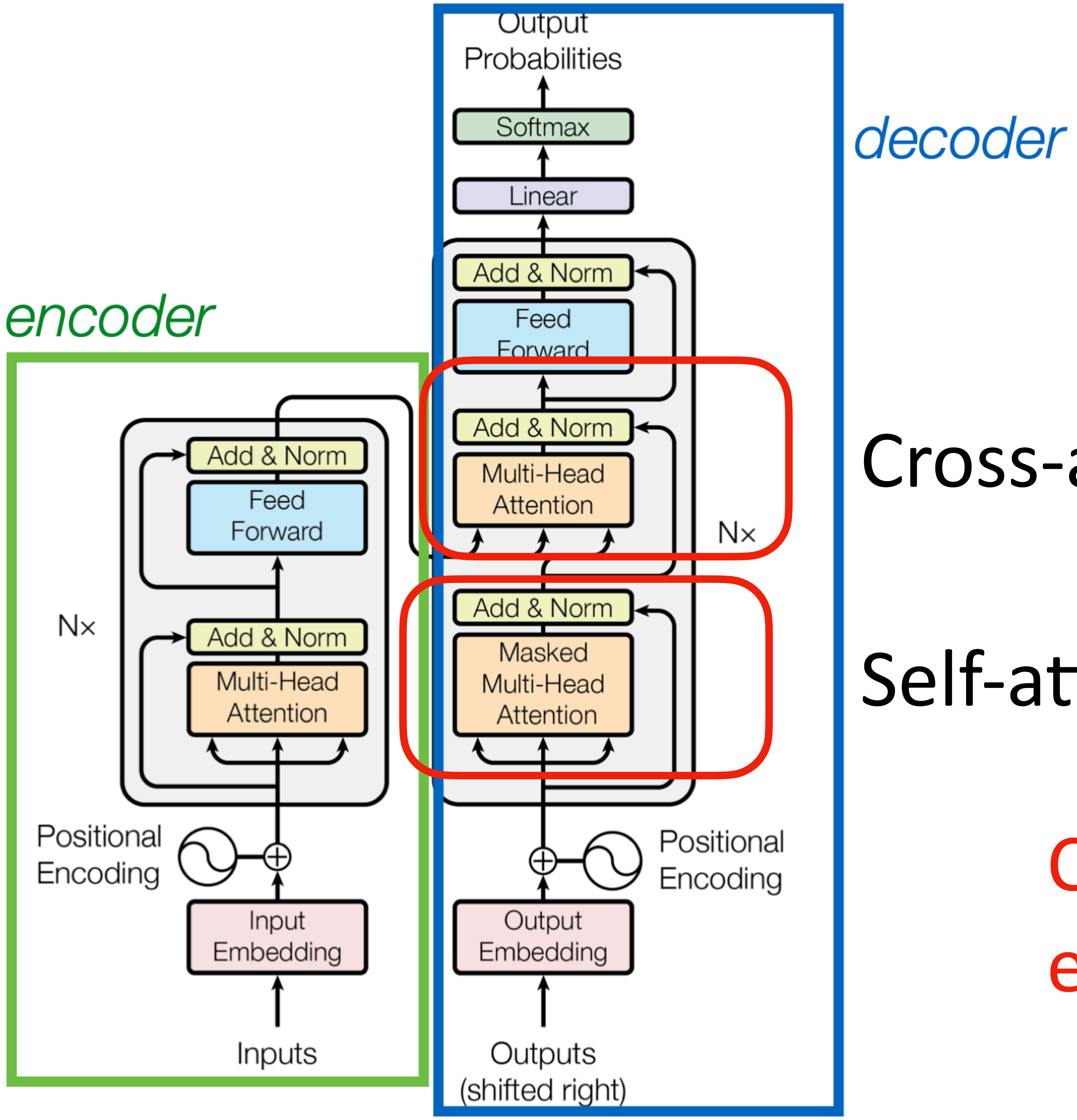
Transformer Decoder in Seq2Seq



Cross-attention

Self-attention

Transformer Decoder in Seq2Seq

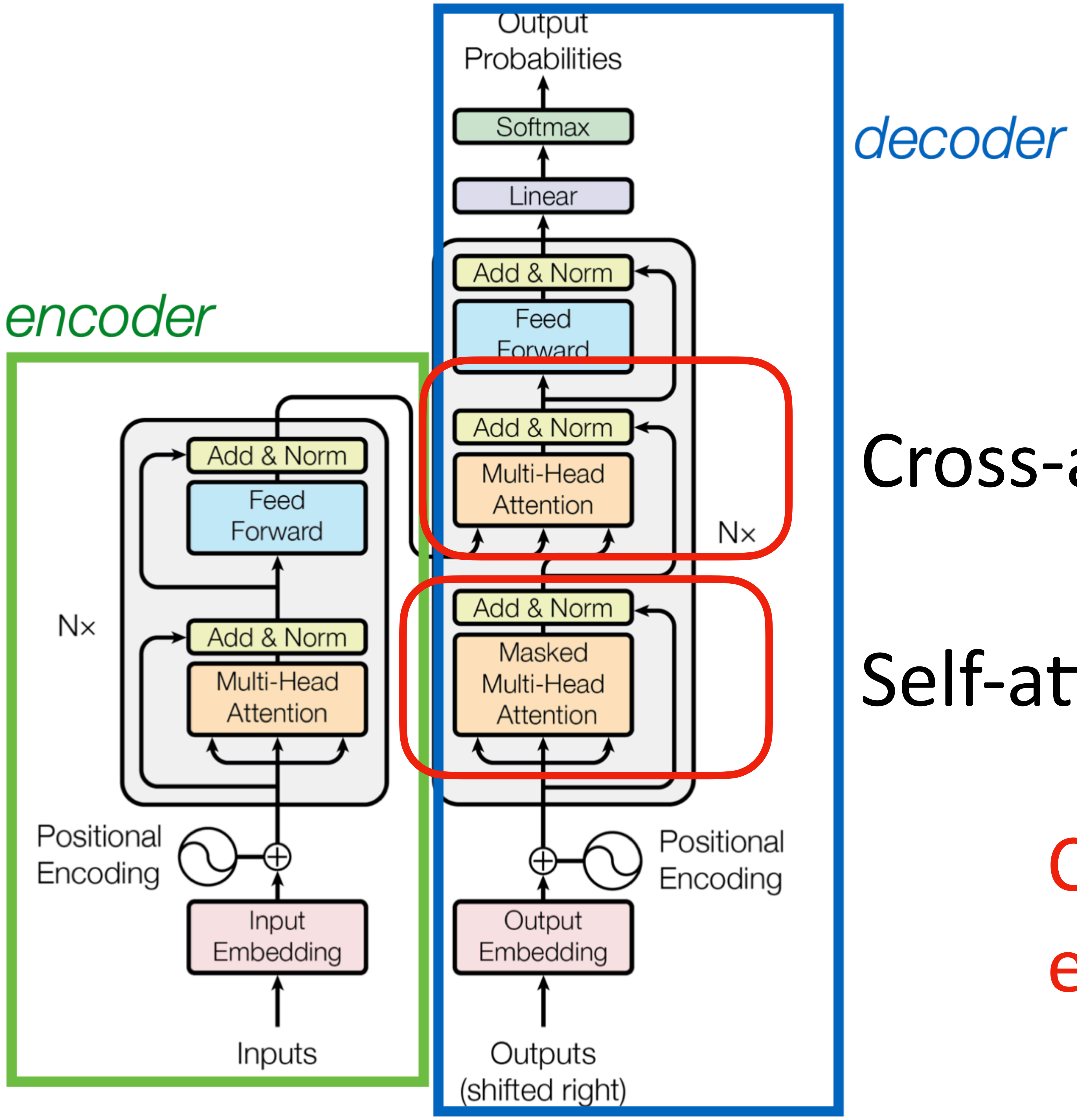


Cross-attention

Self-attention

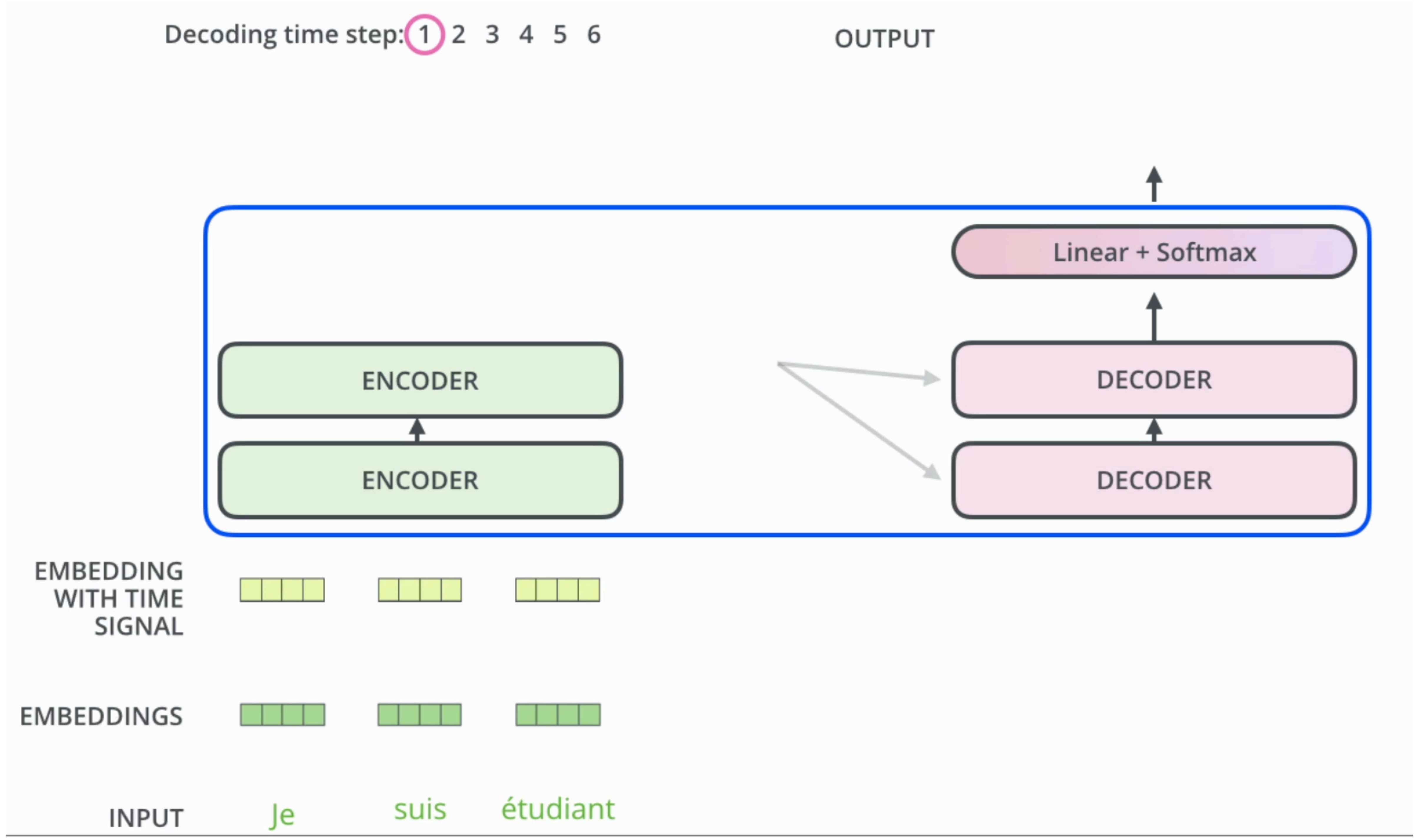
Cross-attention uses the output of encoder as input

Transformer Decoder in Seq2Seq



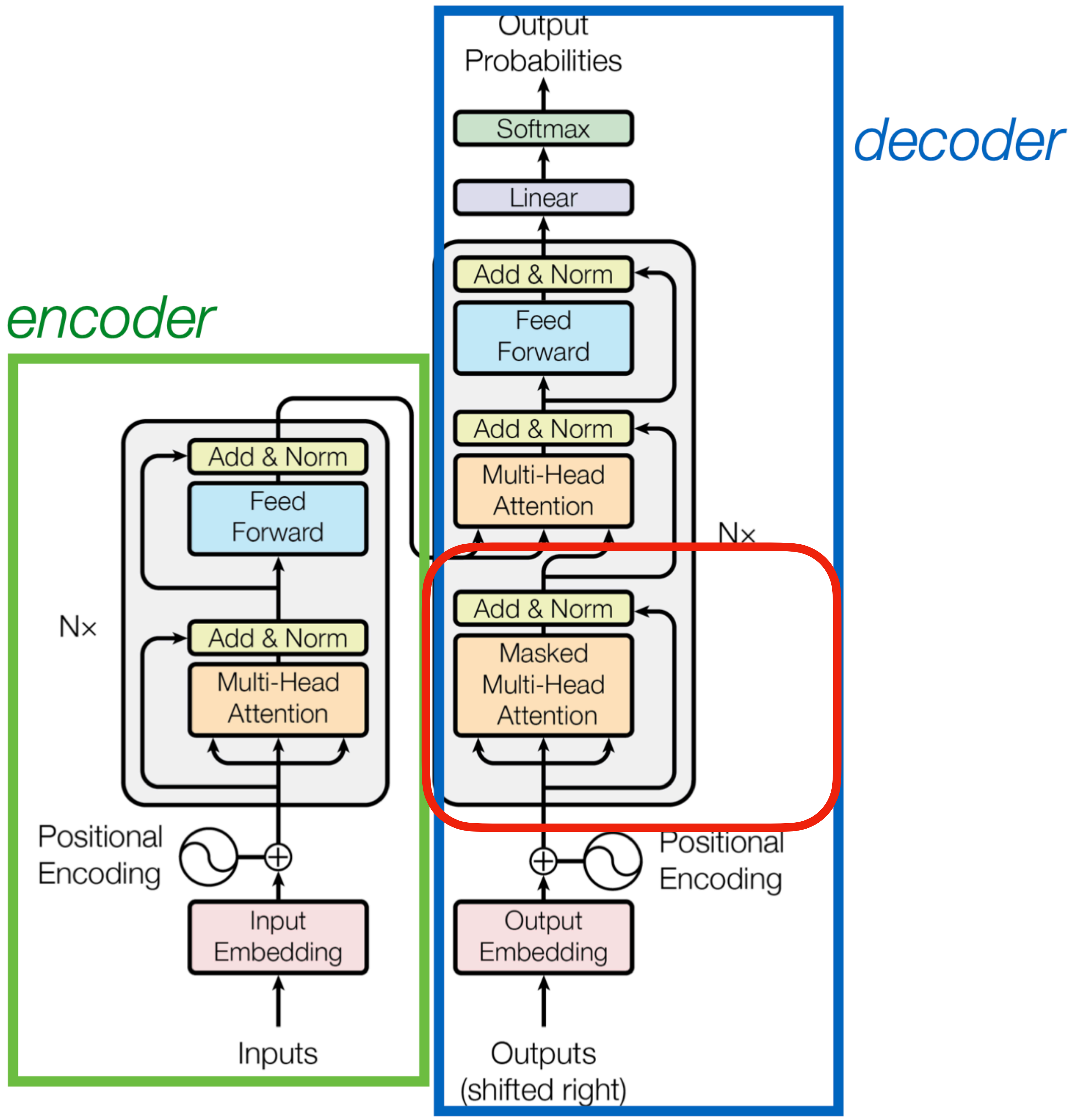
Cross-attention

Self-attention

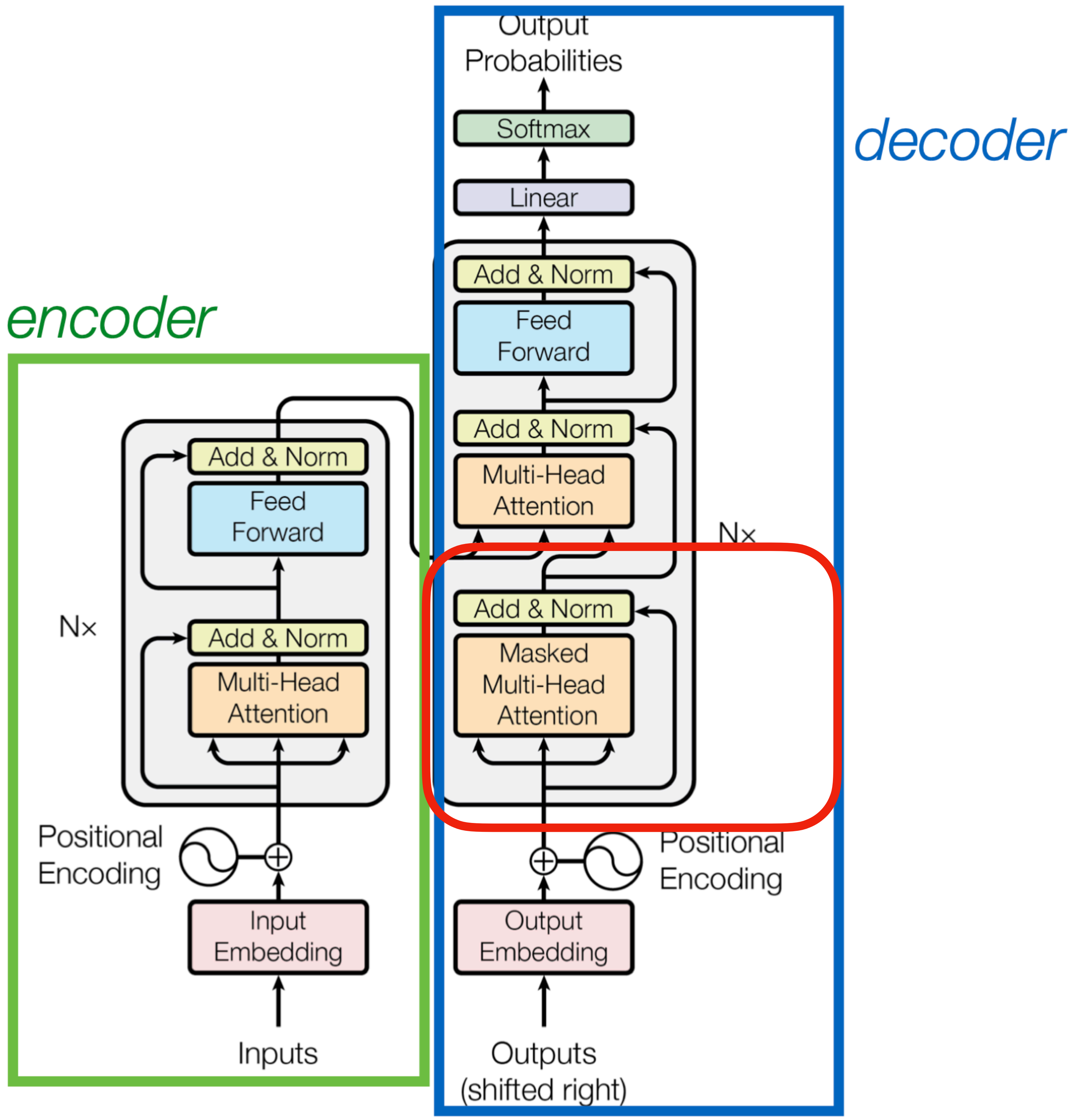


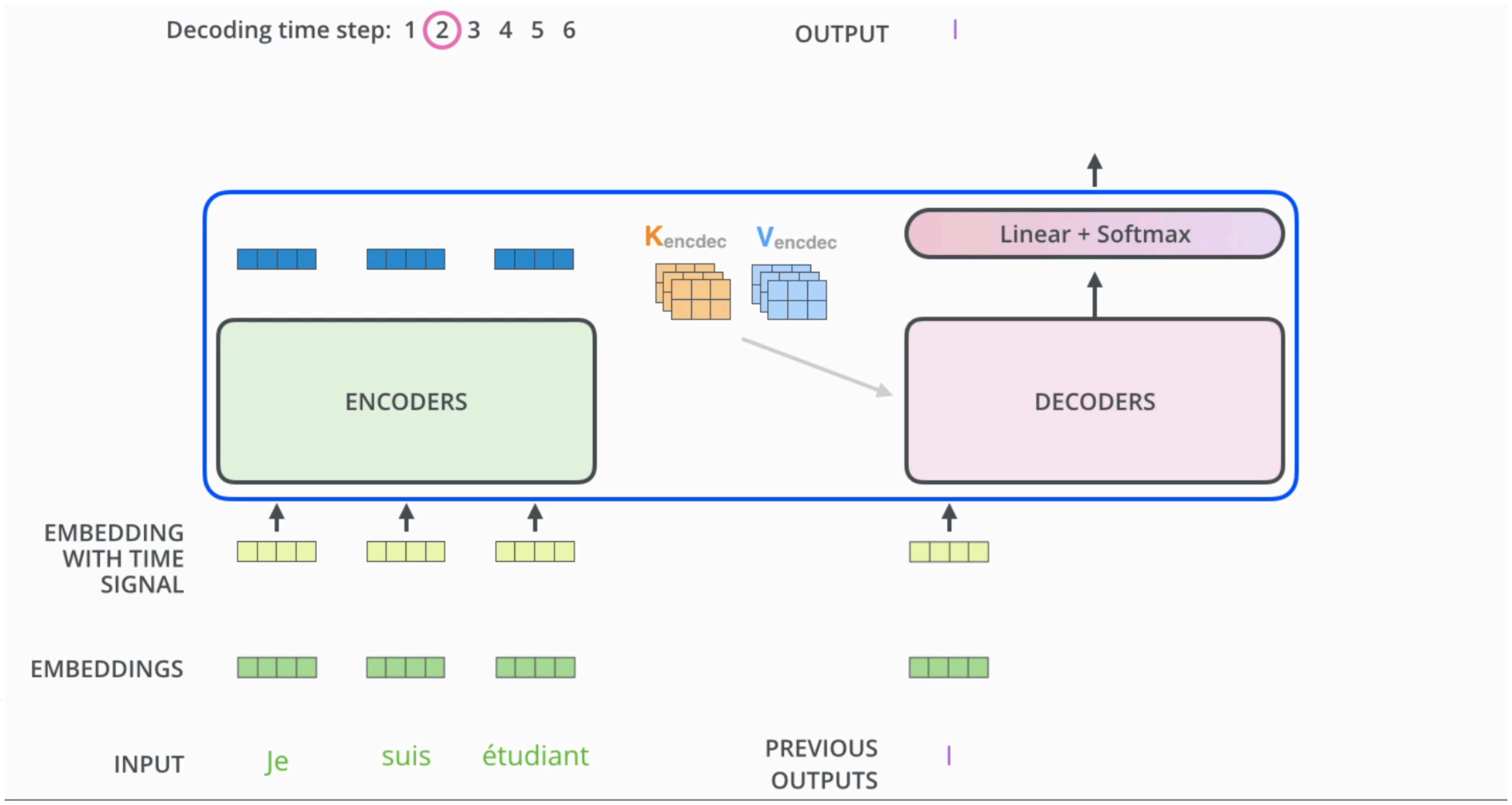
Cross-attention uses the output of encoder as input

Masked Attention



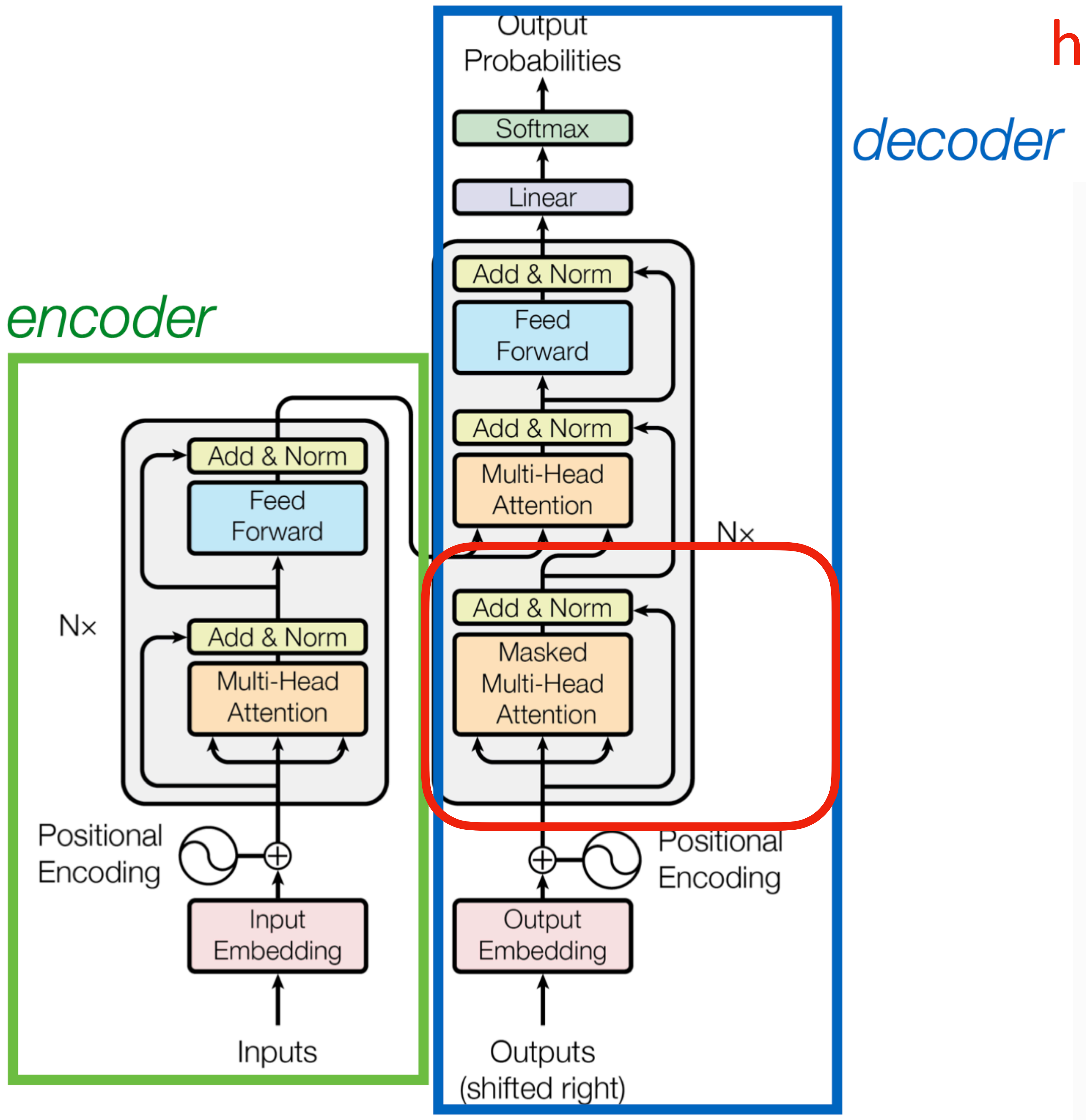
Masked Attention

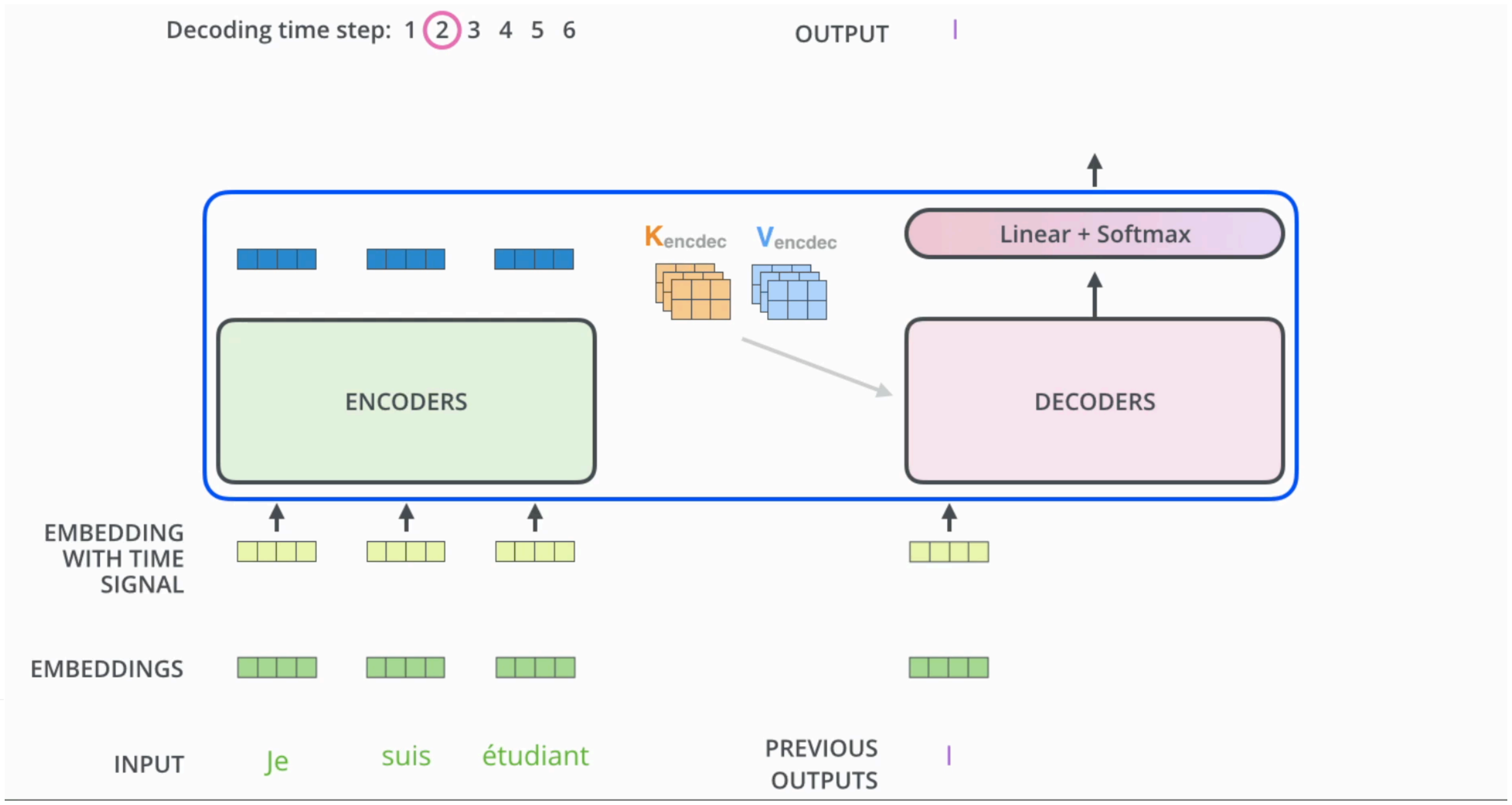




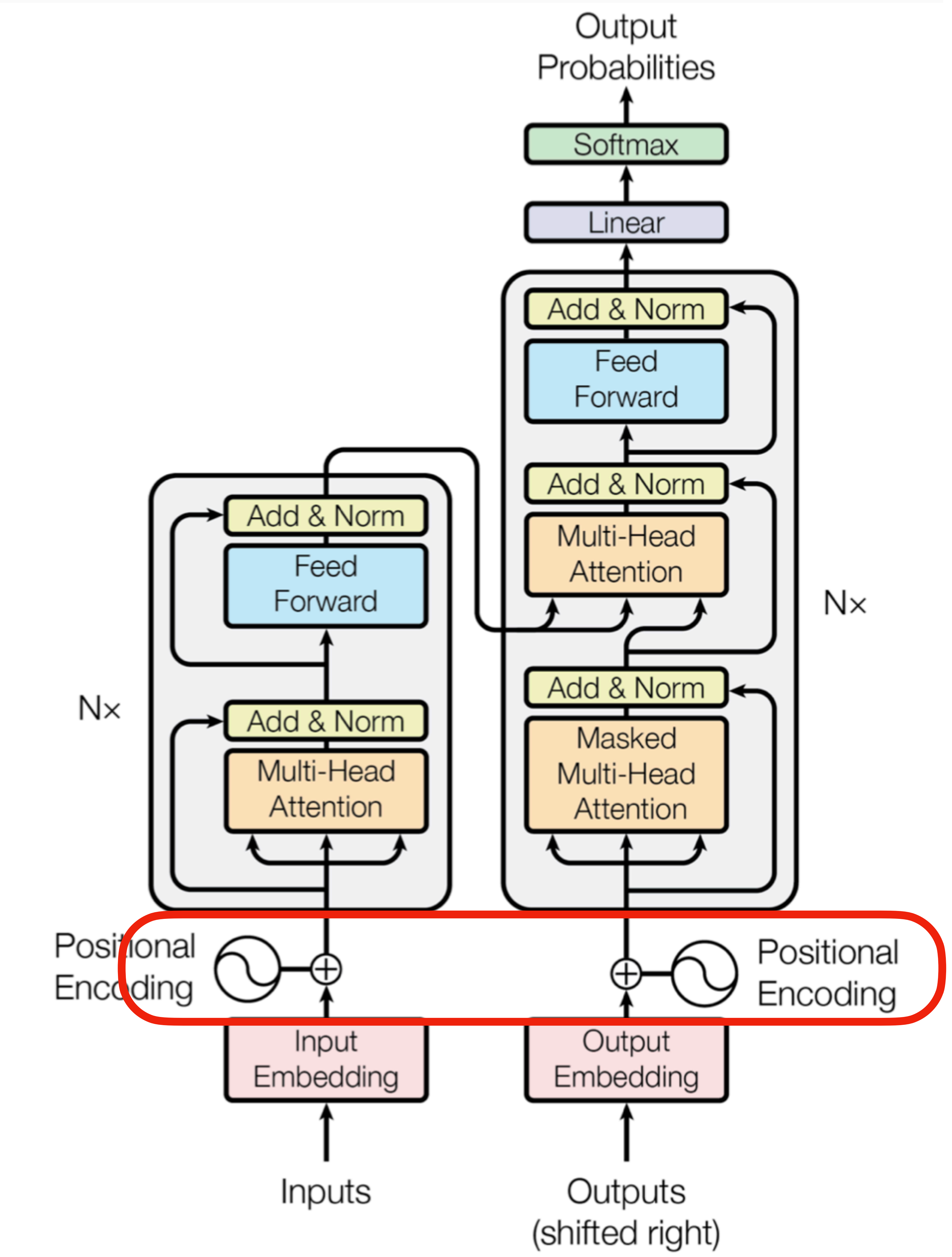
Masked Attention

Typical attention attends to the entire sequence, while masked attention only attends to the ones on the left because future words have not been generated

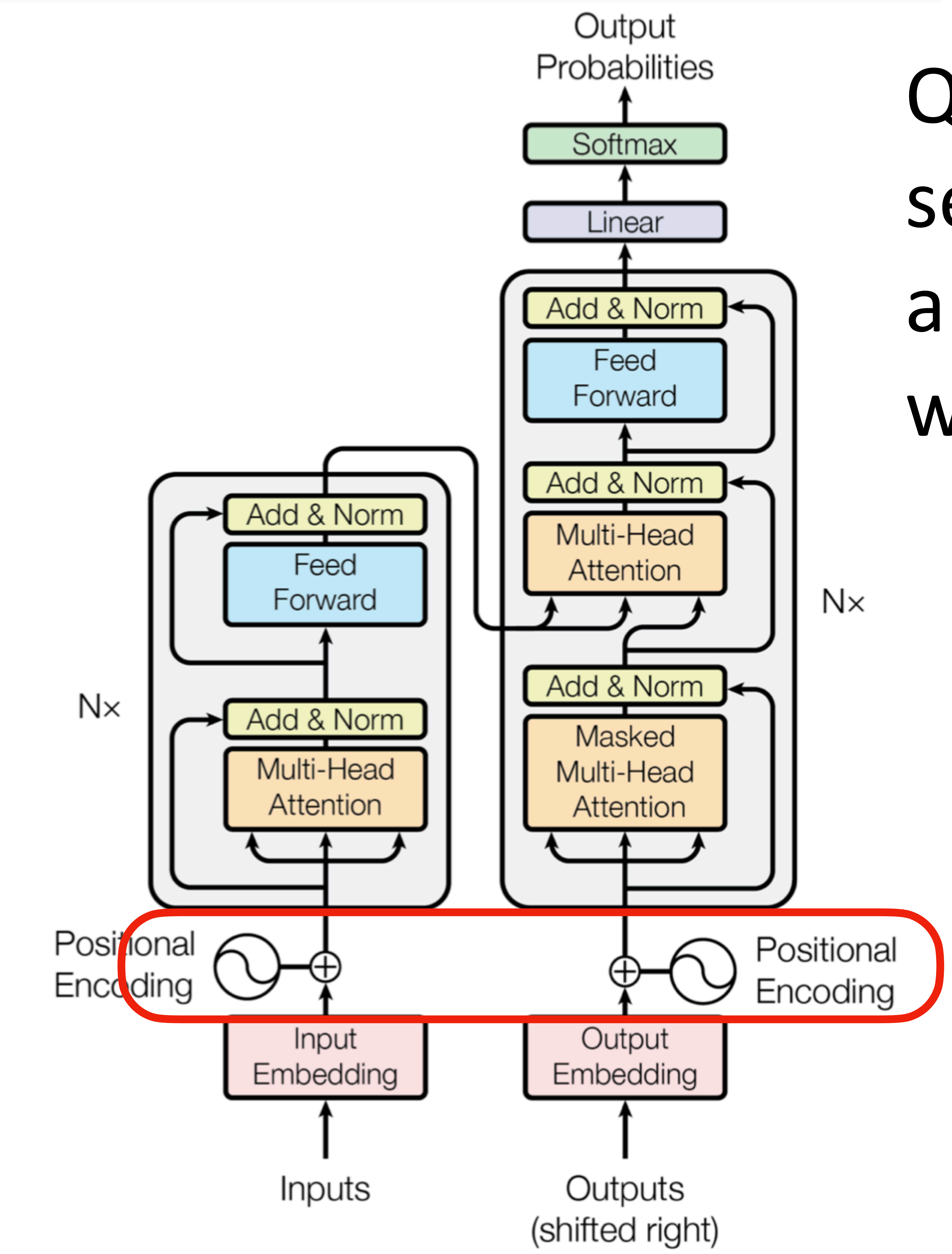




Position Embeddings

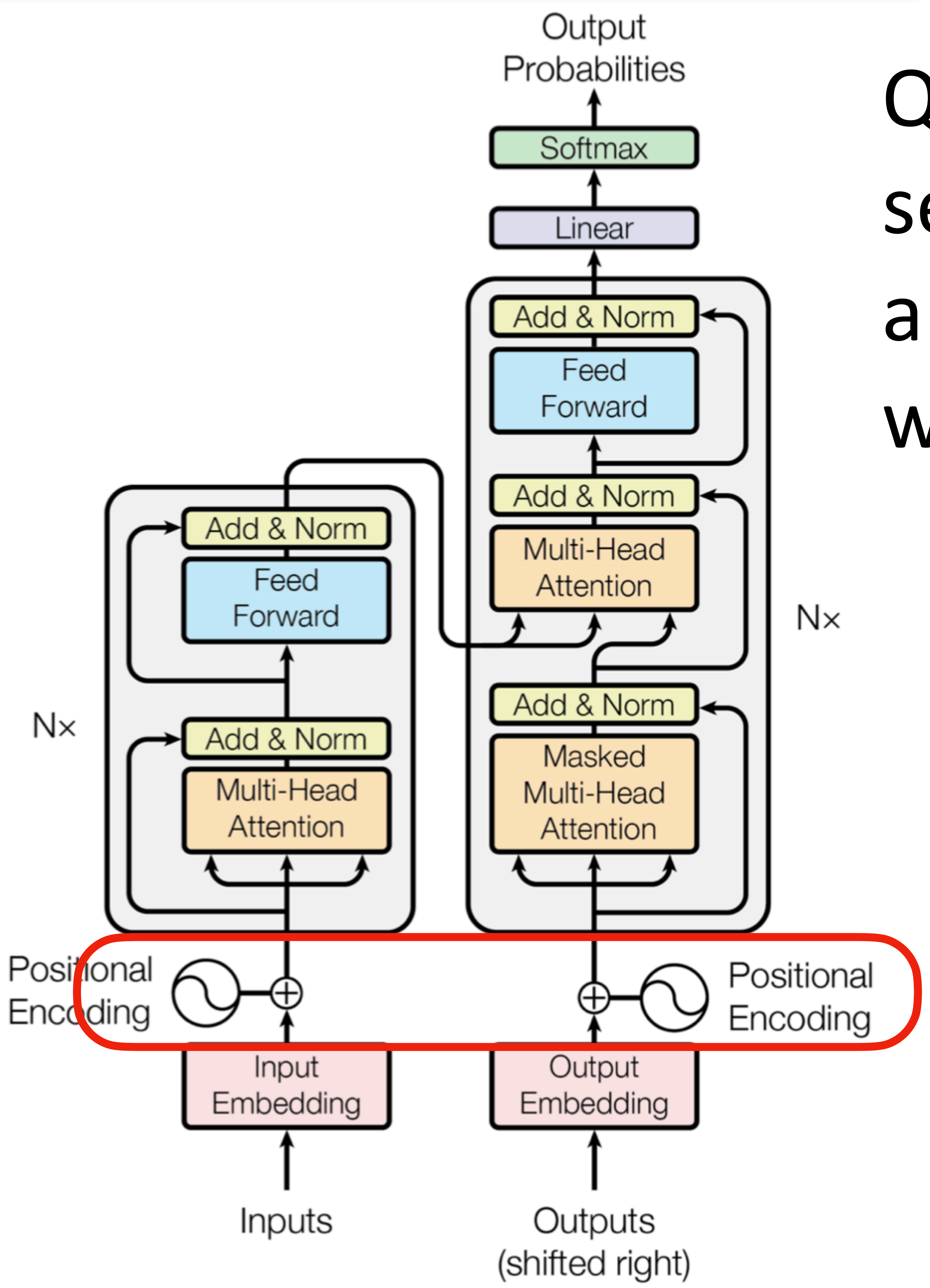


Position Embeddings



Question: If we shuffle the order of words in the sequence, will that change the attention output and feed forward output of the corresponding word?

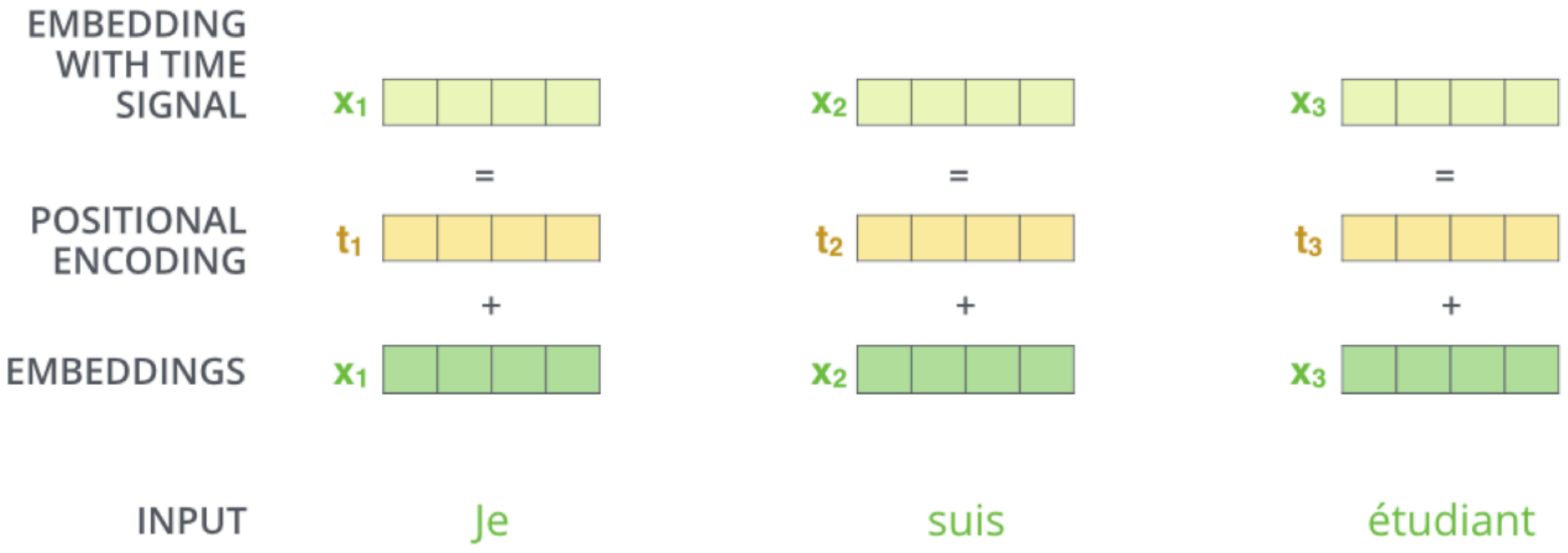
Position Embeddings



Question: If we shuffle the order of words in the sequence, will that change the attention output and feed forward output of the corresponding word?

Position embeddings are added to each word embedding, otherwise our model is unaware of the position of a word

Positional Encoding



Transformer Positional Encoding

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

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

Positional encoding is a 512d vector
i = a particular dimension of this vector
pos = dimension of the word
d_model = 512

Complexity

Layer Type	Complexity per Layer	Sequential Operations
Self-Attention	$O(n^2 \cdot d)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$

n is sequence length, d is embedding dimension.

Complexity

Layer Type	Complexity per Layer	Sequential Operations
Self-Attention	$O(n^2 \cdot d)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$

n is sequence length, d is embedding dimension.

Restricted self-attention means not attending all words in the sequence, but only a restricted field

Complexity

Layer Type	Complexity per Layer	Sequential Operations
Self-Attention	$O(n^2 \cdot d)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$

n is sequence length, d is embedding dimension.

Restricted self-attention means not attending all words in the sequence, but only a restricted field

Square complexity of sequence length is a major issue for transformers to deal with long sequence

Thank You!