Transformers and sequence-to-sequence learning

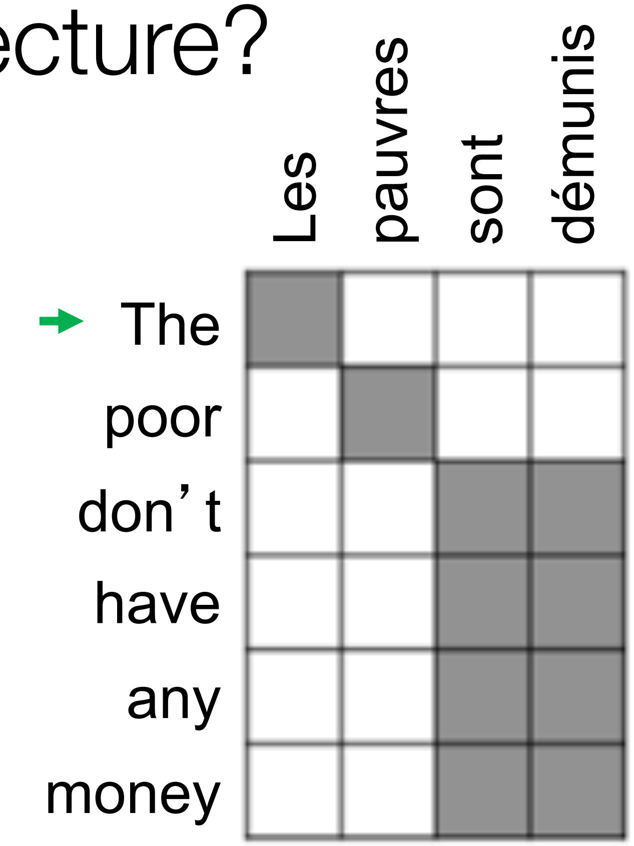
CS 685, Fall 2021

Mohit Iyyer
College of Information and Computer Sciences
University of Massachusetts Amherst

some slides from Emma Strubell

From last time

- Project proposals now due 10/1!
- Quiz 2 due Friday
- Can TAs record zoom office hours? Maybe
- How did we get this grid in the previous lecture? Will explain in today's class.
- Final proj. reports due Dec. 16th



iPad

sequence-to-sequence learning

Used when inputs and outputs are both sequences of words (e.g., machine translation, summarization)

- we'll use French (*f*) to English (*e*) as a running example
- **goal**: given French sentence *f* with tokens f_1, f_2, … f_n produce English translation *e* with tokens e_1, e_2, … e_m
- **real goal**: compute $\arg\max_e p(e \mid f)$

This is an instance of *conditional language modeling*

$$p(e \mid f) = p(e_1, e_2, \ldots, e_m \mid f)$$

$$= p(e_1 \mid f) \cdot p(e_2 \mid e_1, f) \cdot p(e_3 \mid e_2, e_1, f) \cdot \ldots$$

$$= \prod_{i=1}^{m} p(e_i \mid e_1, \ldots, e_{i-1}, f)$$

Just like we've seen before, except we additionally condition our prediction of the next word on some other input (here, the French sentence)

seq2seq models

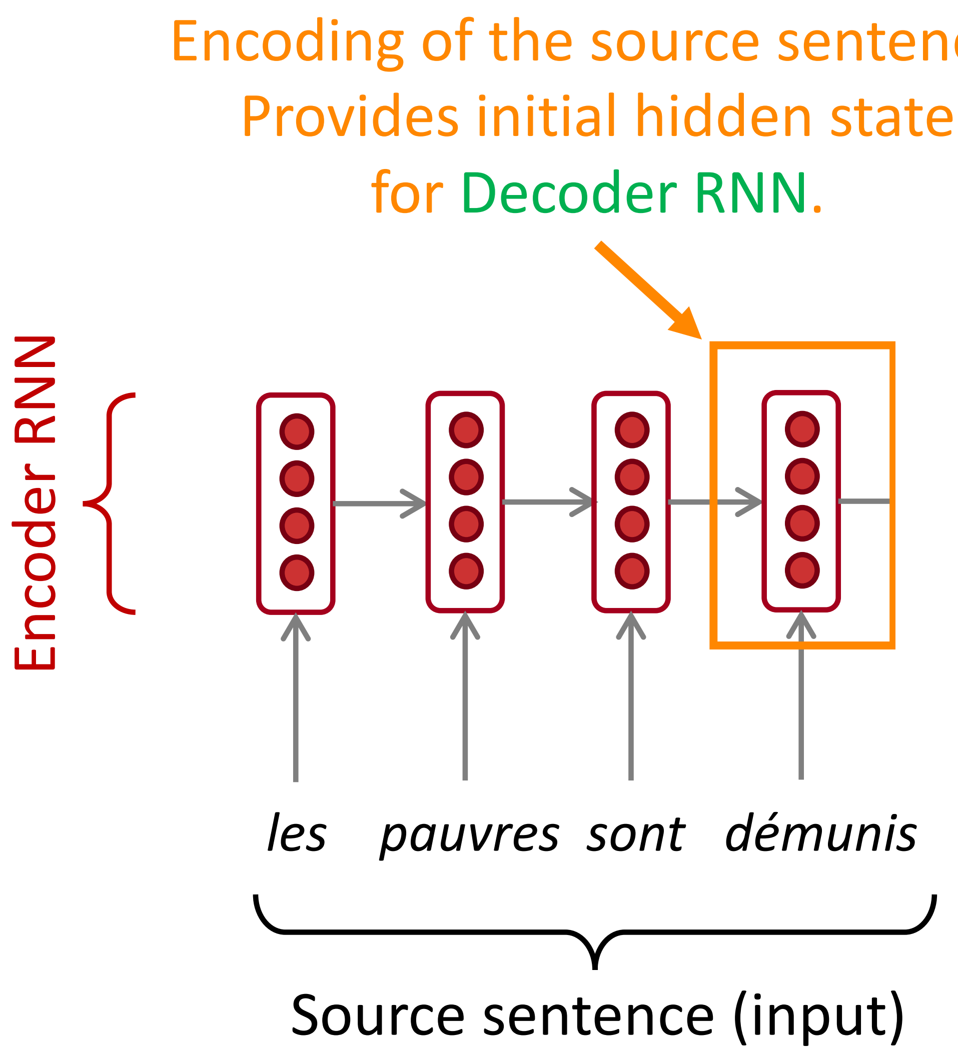
- use two different neural networks to model

$$\prod_{i=1}^{L} p(e_i \,|\, e_1, \ldots, e_{i-1}, f)$$

- first we have the *encoder*, which encodes the French sentence *f*
- then, we have the *decoder,* which produces the English sentence *e*

Neural Machine Translation (NMT)

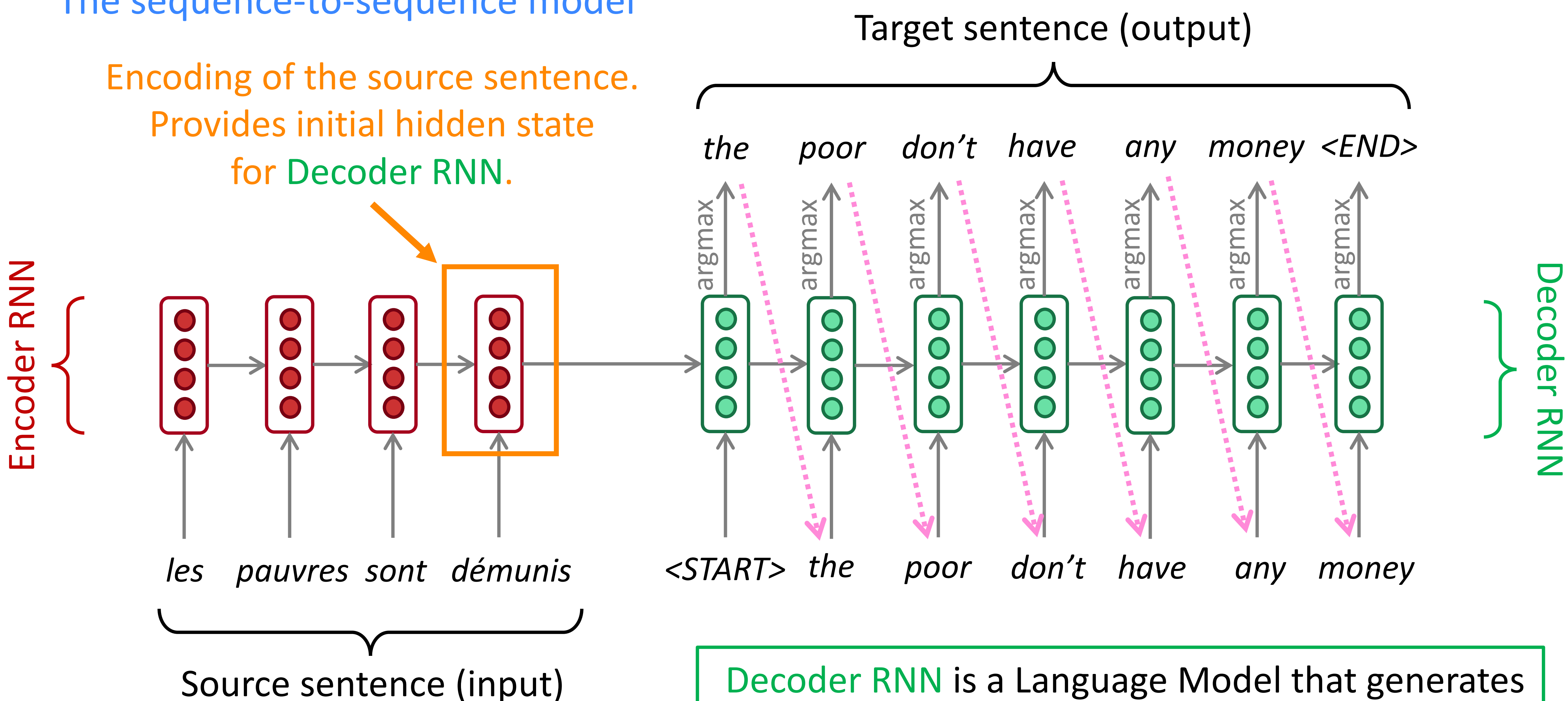
The sequence-to-sequence model



Encoder RNN produces an encoding of the source sentence.

Neural Machine Translation (NMT)

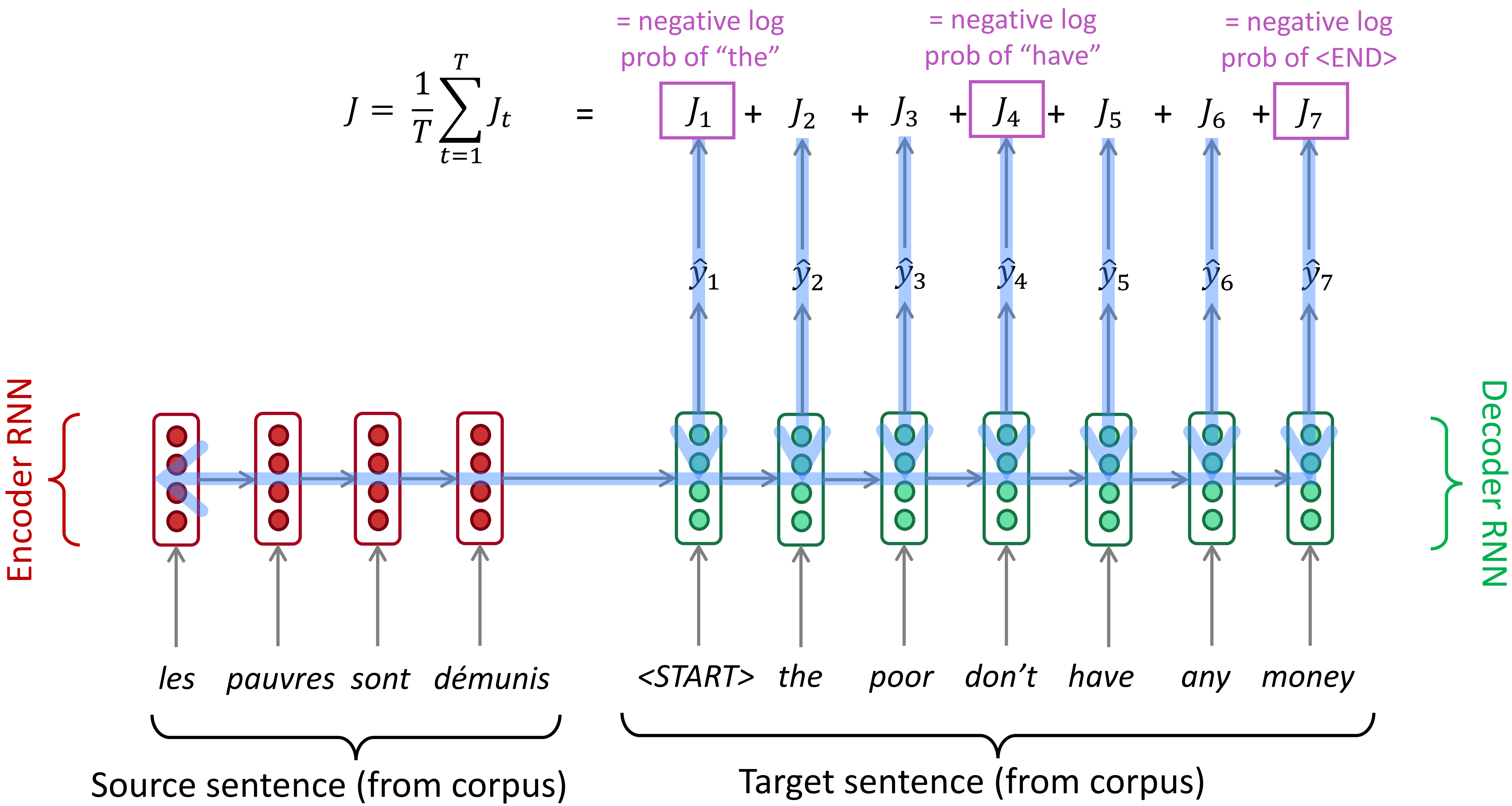
The sequence-to-sequence model



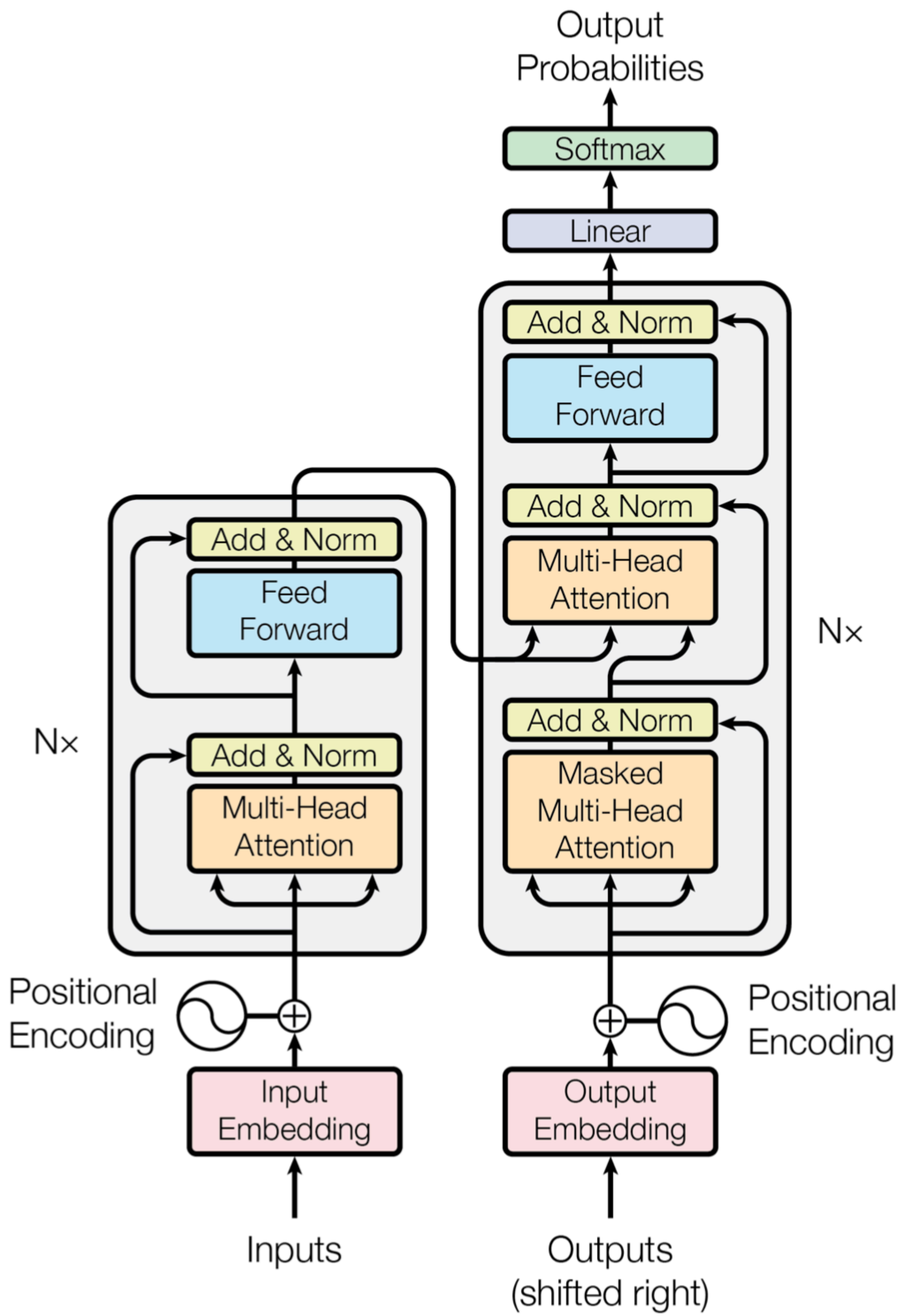
Decoder RNN is a Language Model that generates target sentence conditioned on encoding.

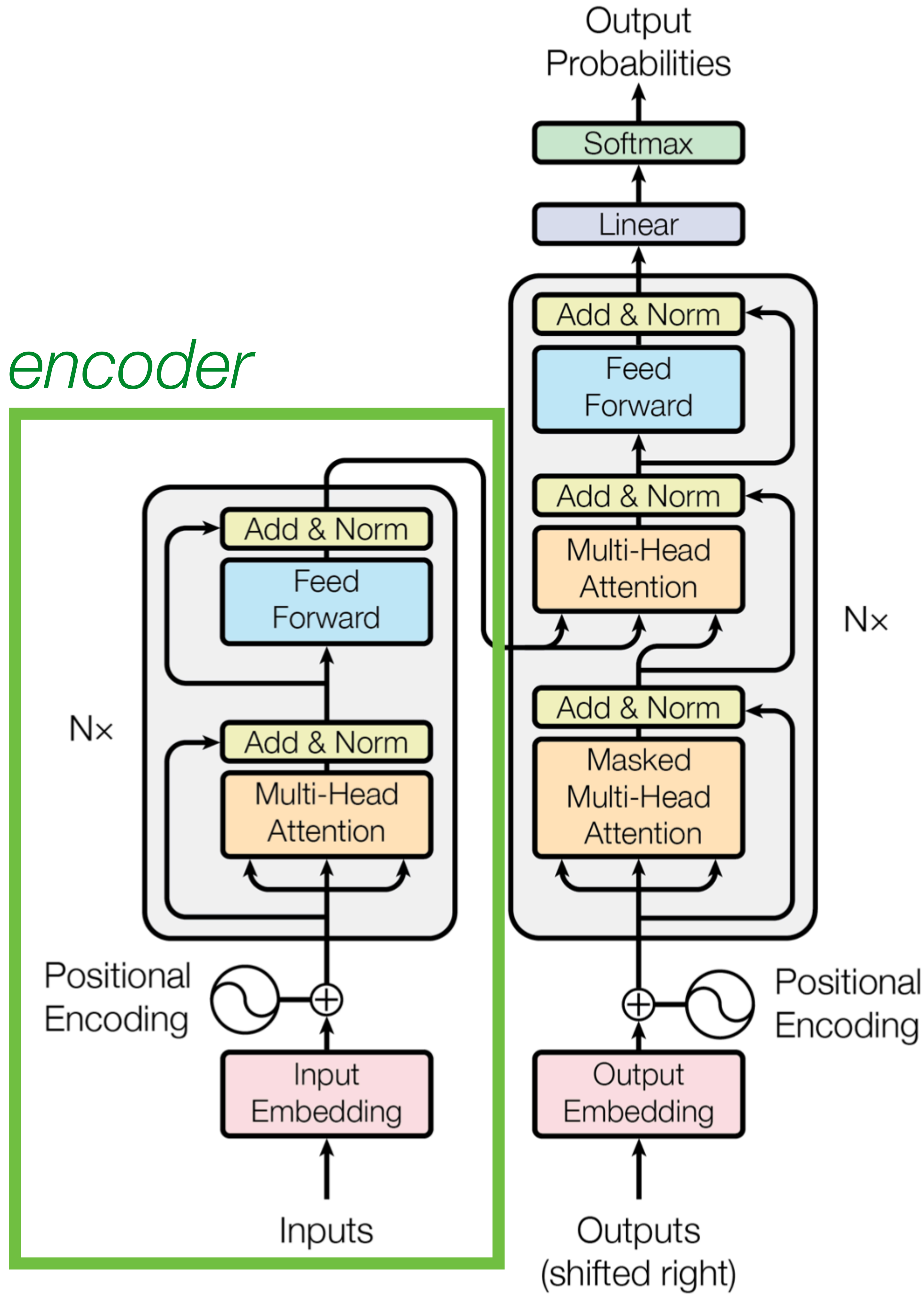
Encoder RNN produces an encoding of the source sentence.

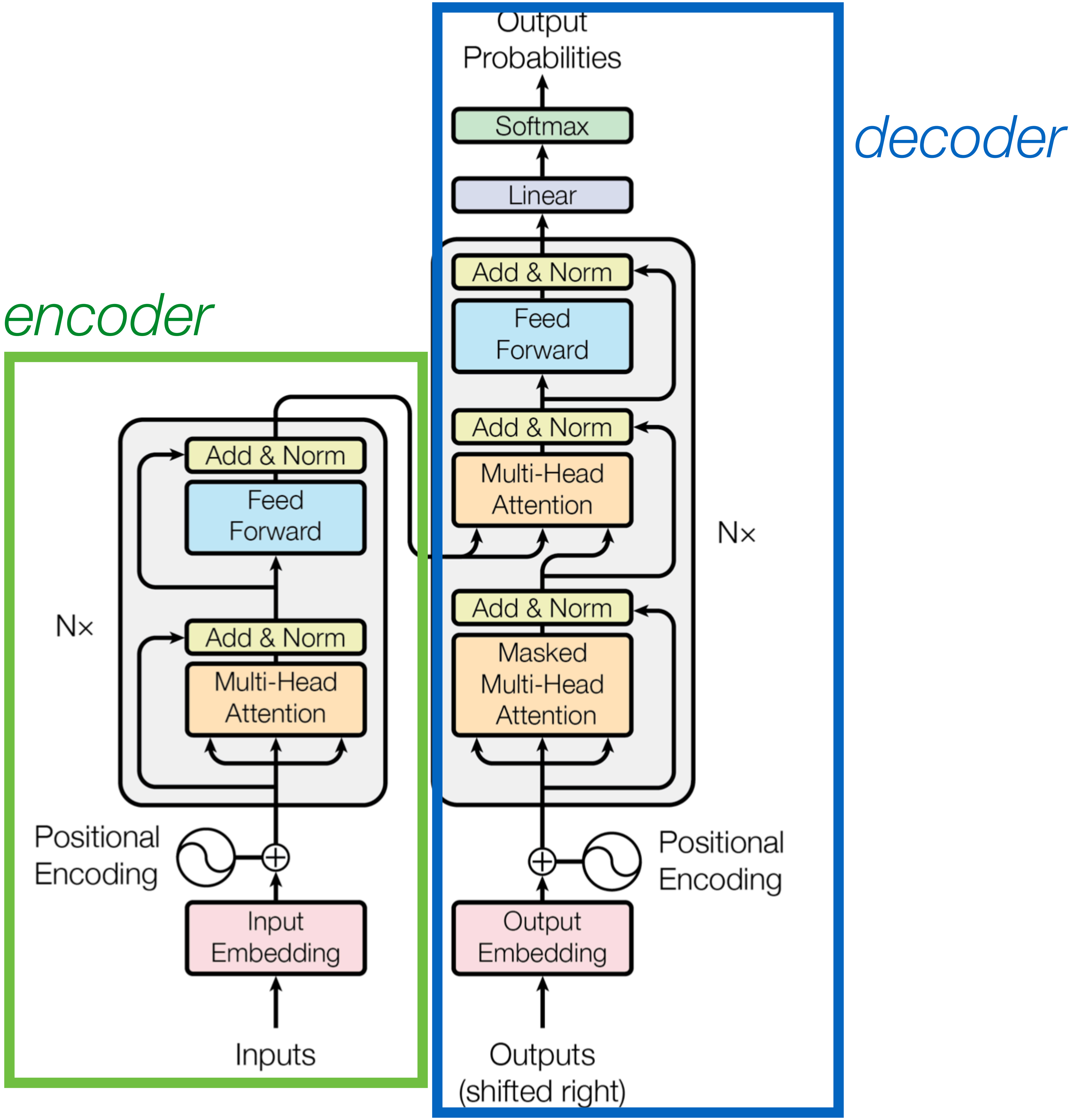
Training a Neural Machine Translation system



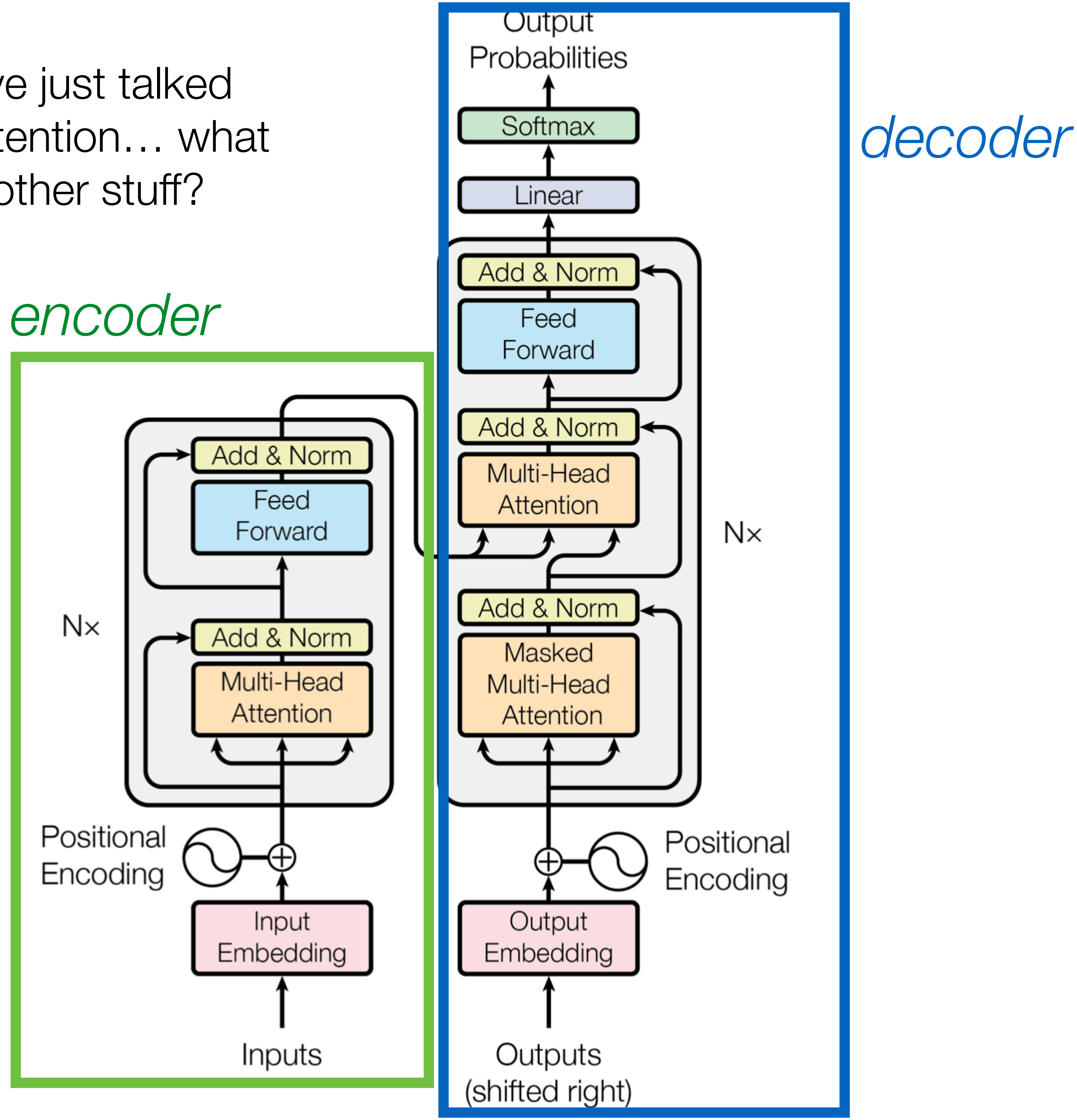
We’ll talk much more about machine translation / other seq2seq problems later… but for now, let’s go back to the Transformer


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)


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

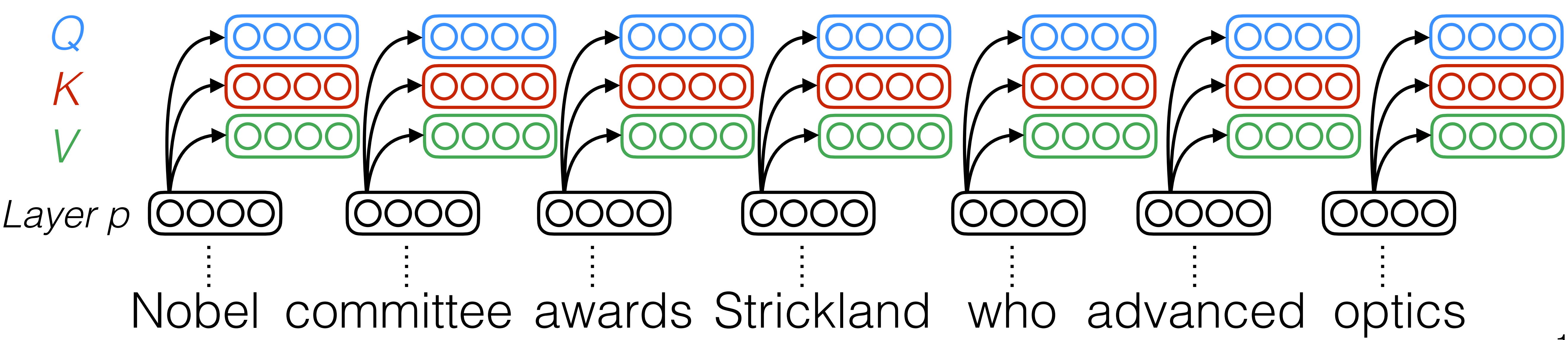

encoder
decoder
Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted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

So far we've just talked about self-attention… what is all this other stuff?



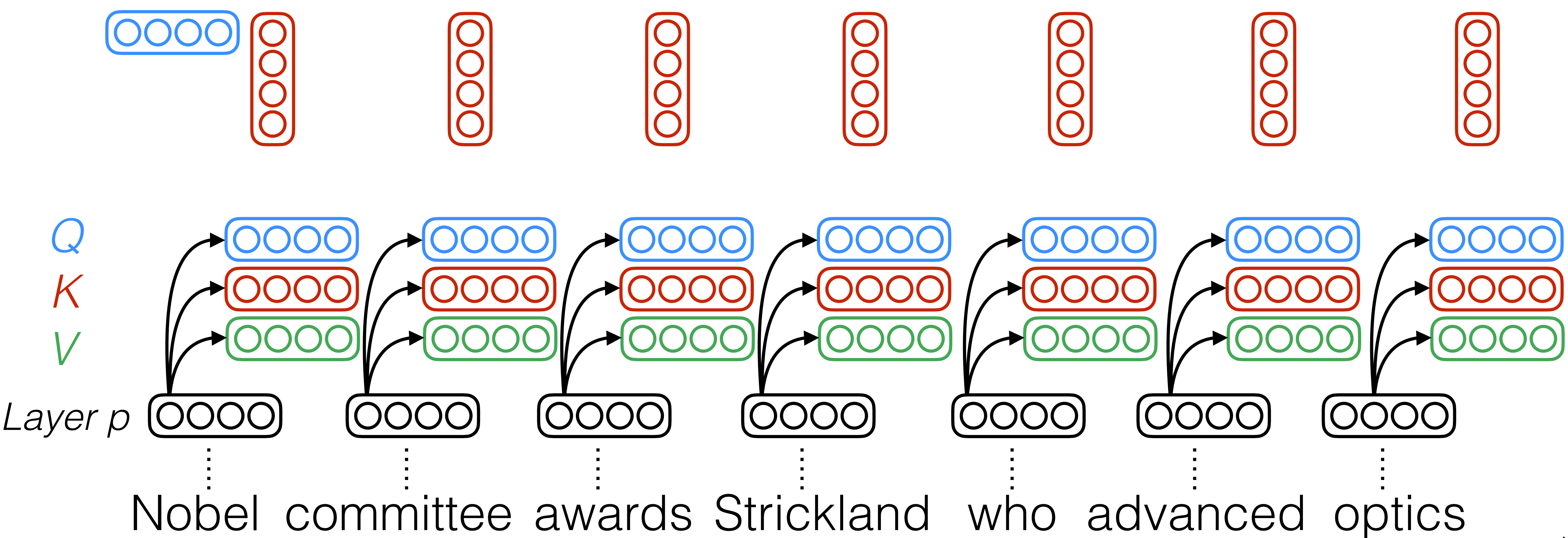
[Vaswani et al. 2017]

Self-attention (in encoder)



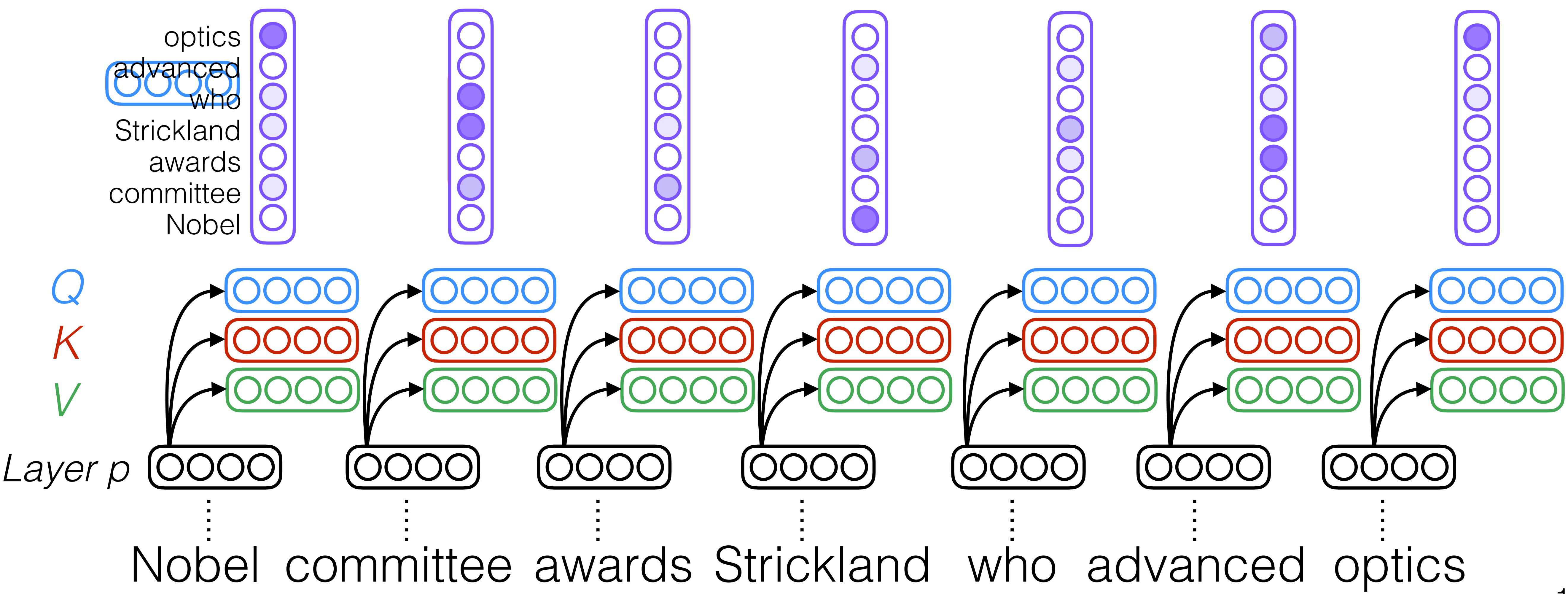
[Vaswani et al. 2017]

Self-attention (in encoder)



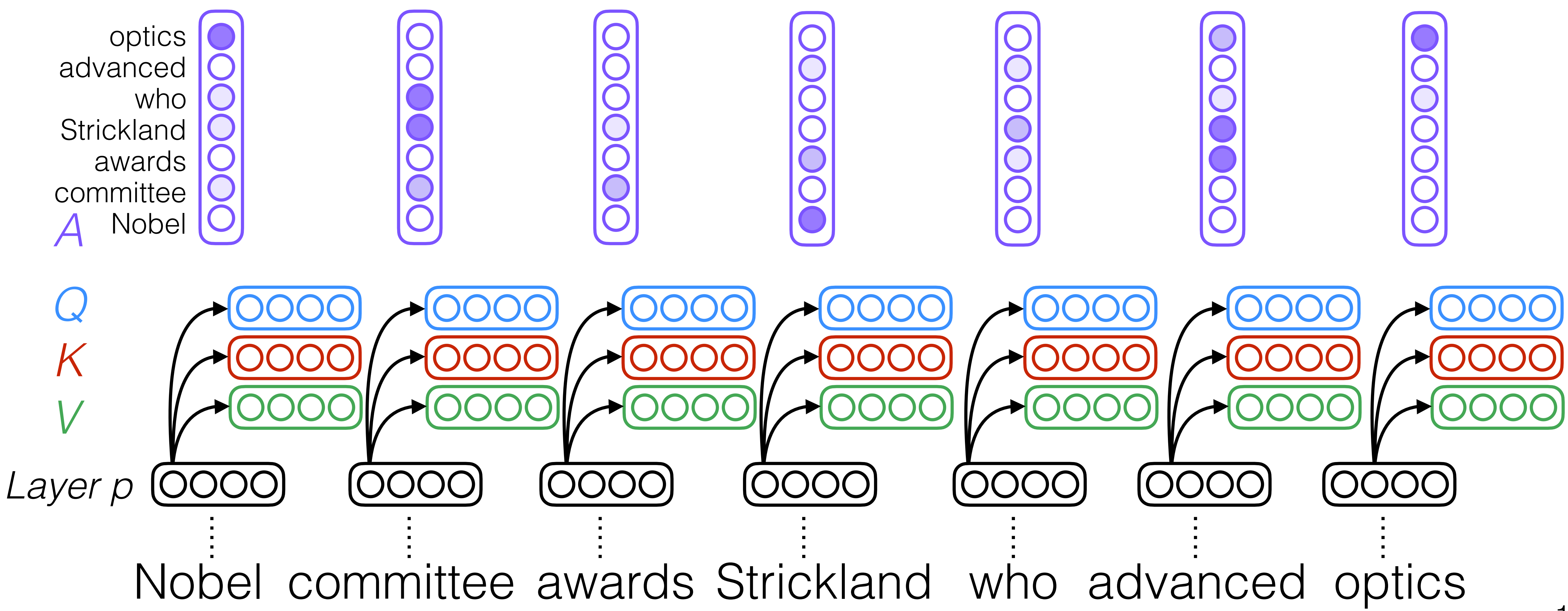
[Vaswani et al. 2017]

Self-attention (in encoder)



[Vaswani et al. 2017]

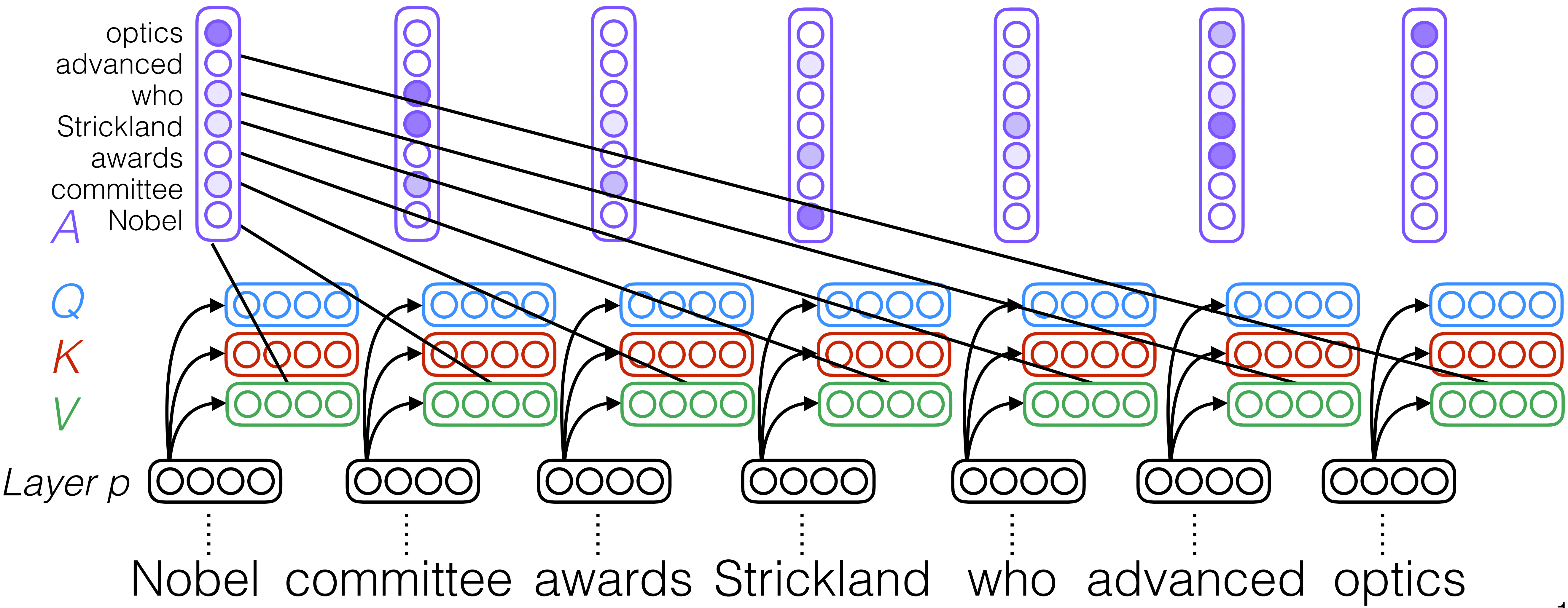
Self-attention (in encoder)



Slides by Emma Strubell!

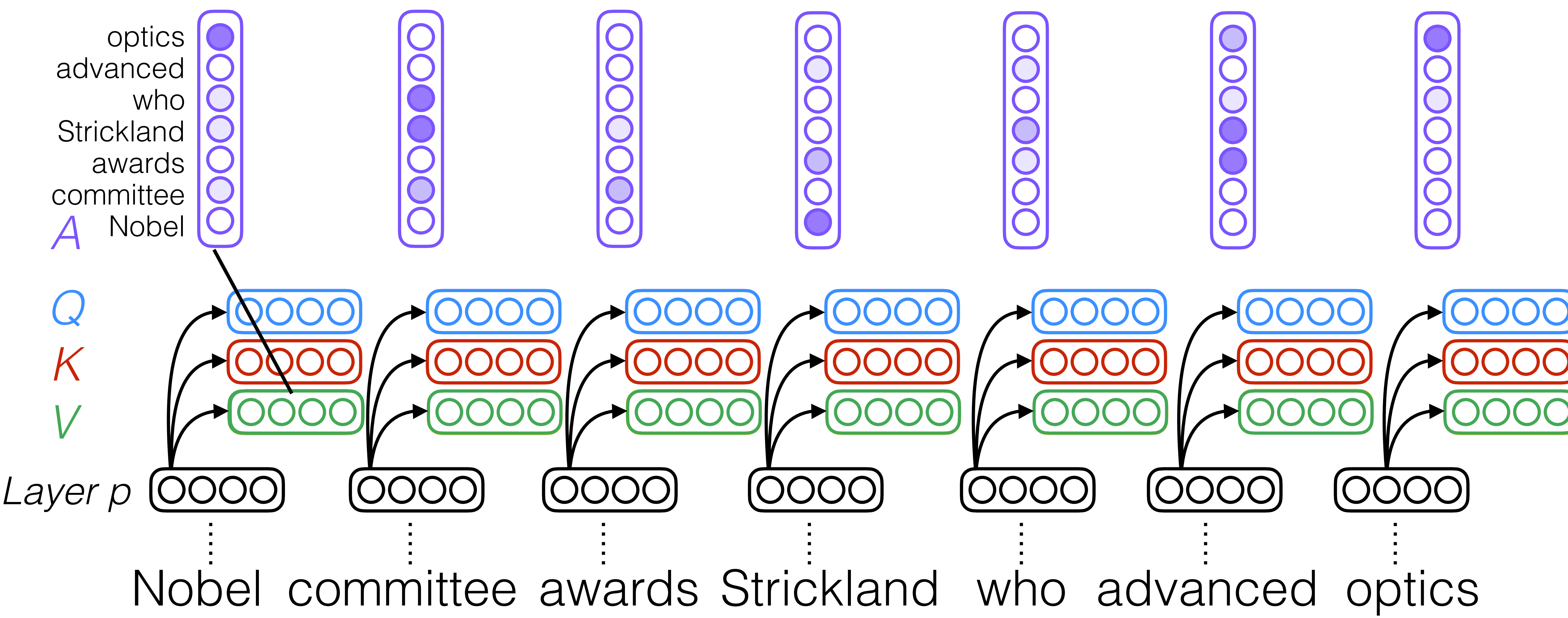
[Vaswani et al. 2017]

Self-attention (in encoder)



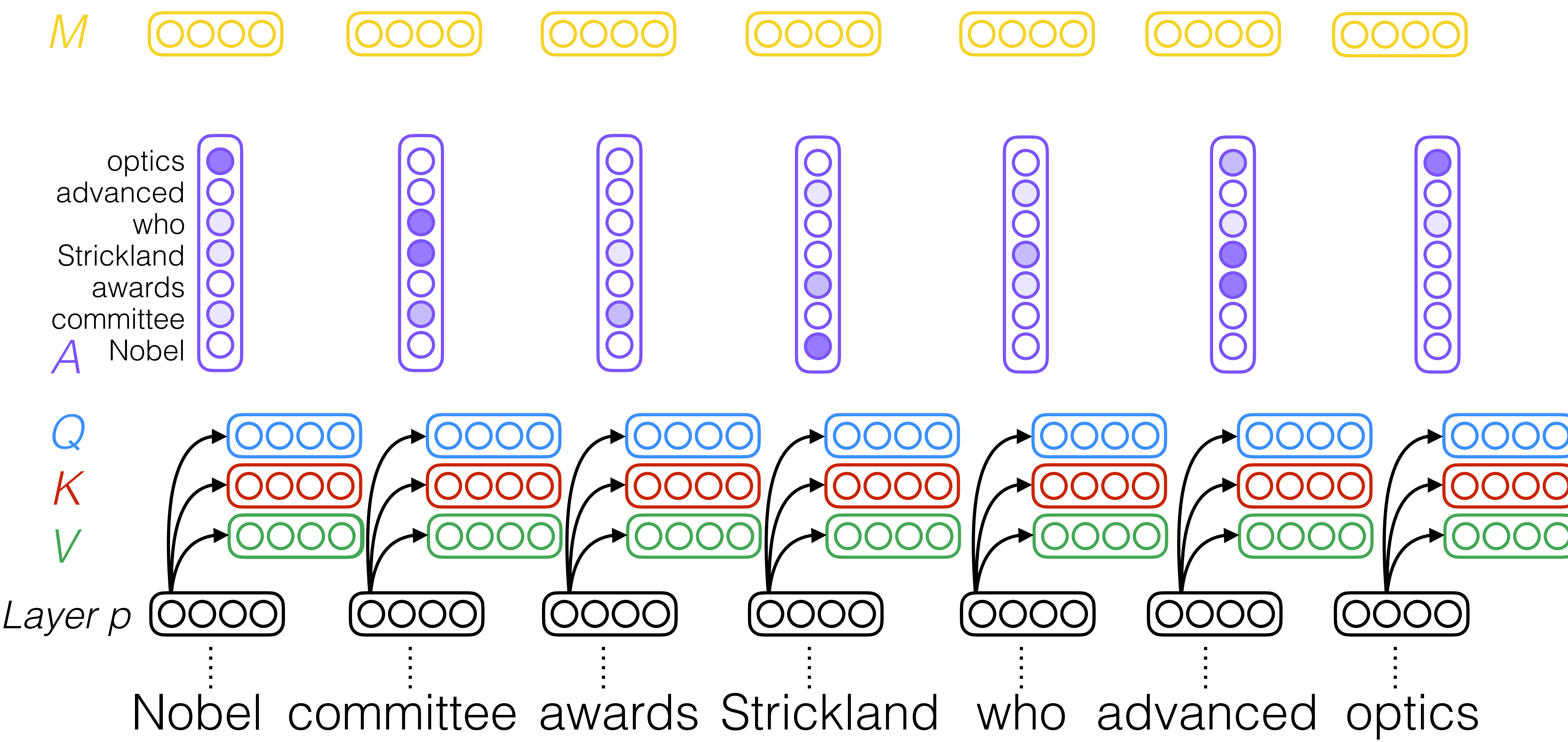
[Vaswani et al. 2017]

Self-attention (in encoder)



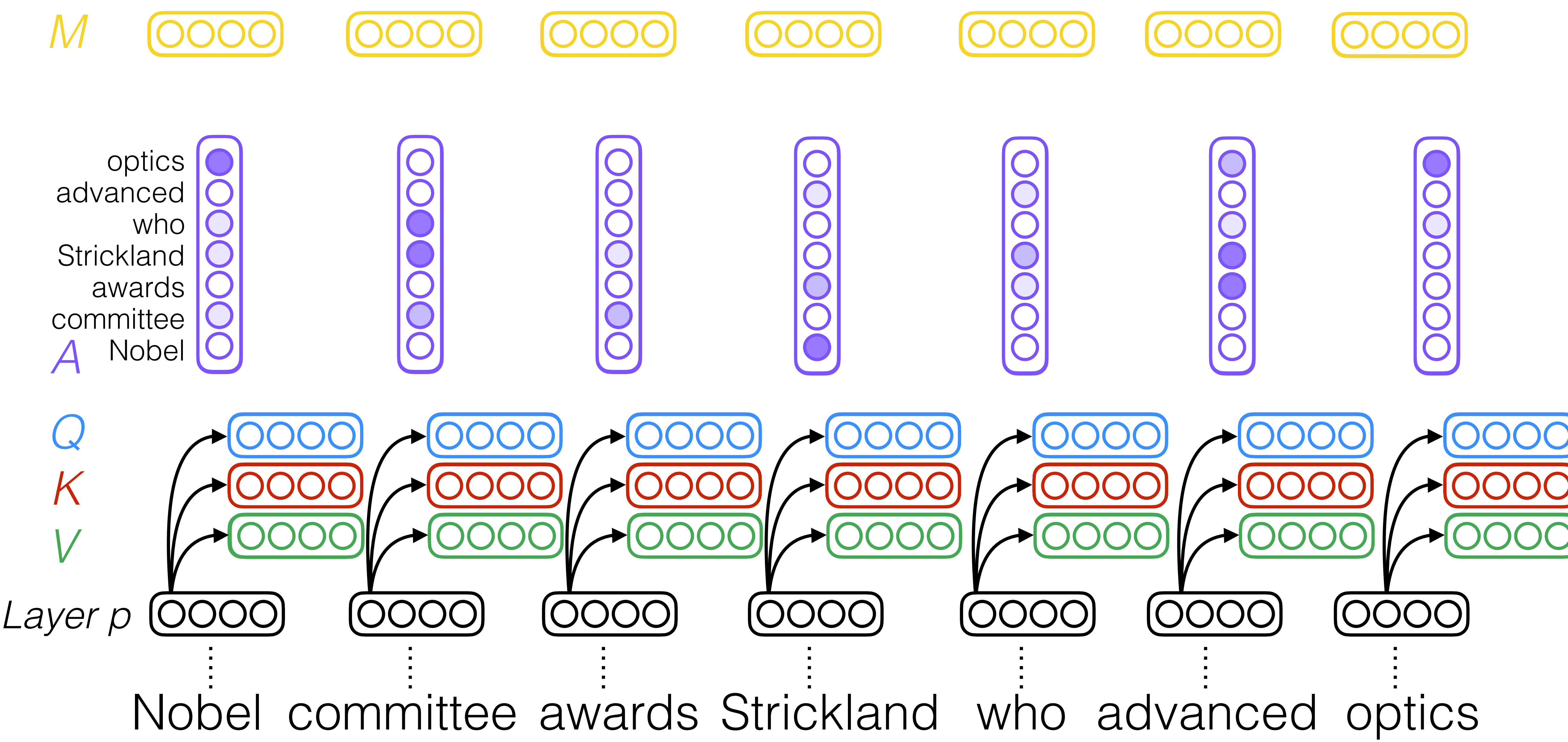
[Vaswani et al. 2017]

Self-attention (in encoder)



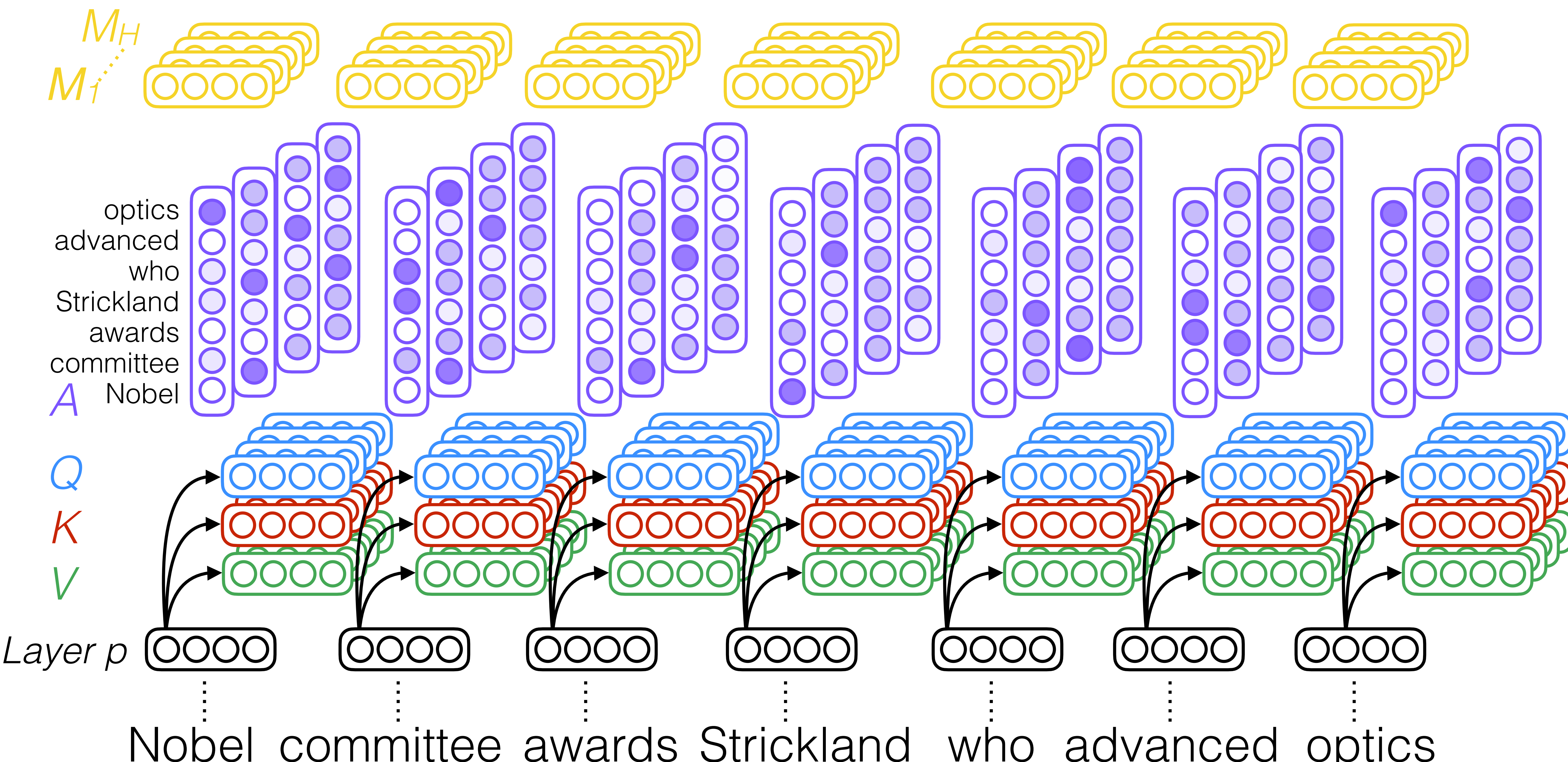
[Vaswani et al. 2017]

Self-attention (in encoder)



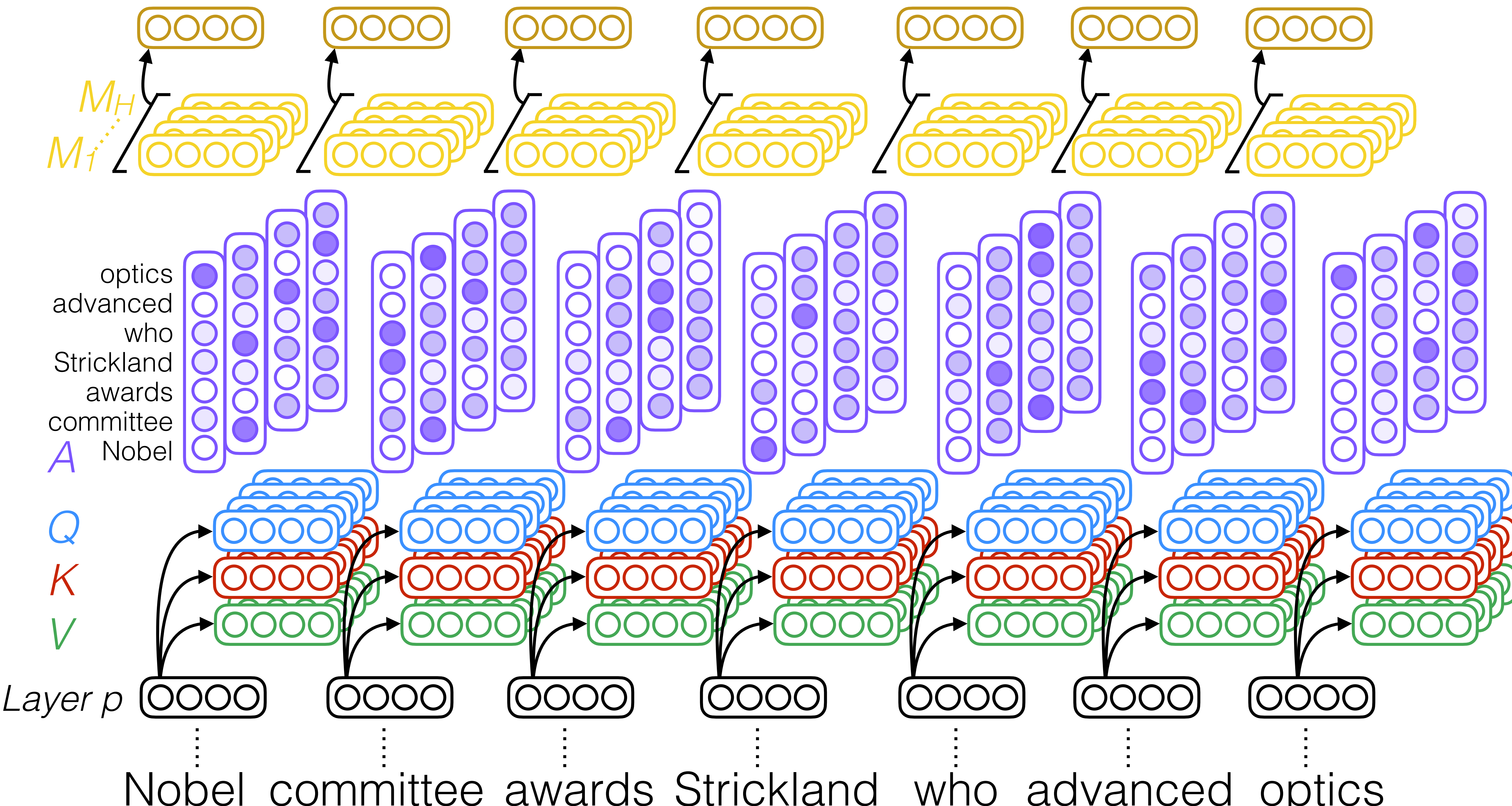
[Vaswani et al. 2017]

Multi-head self-attention



[Vaswani et al. 2017]

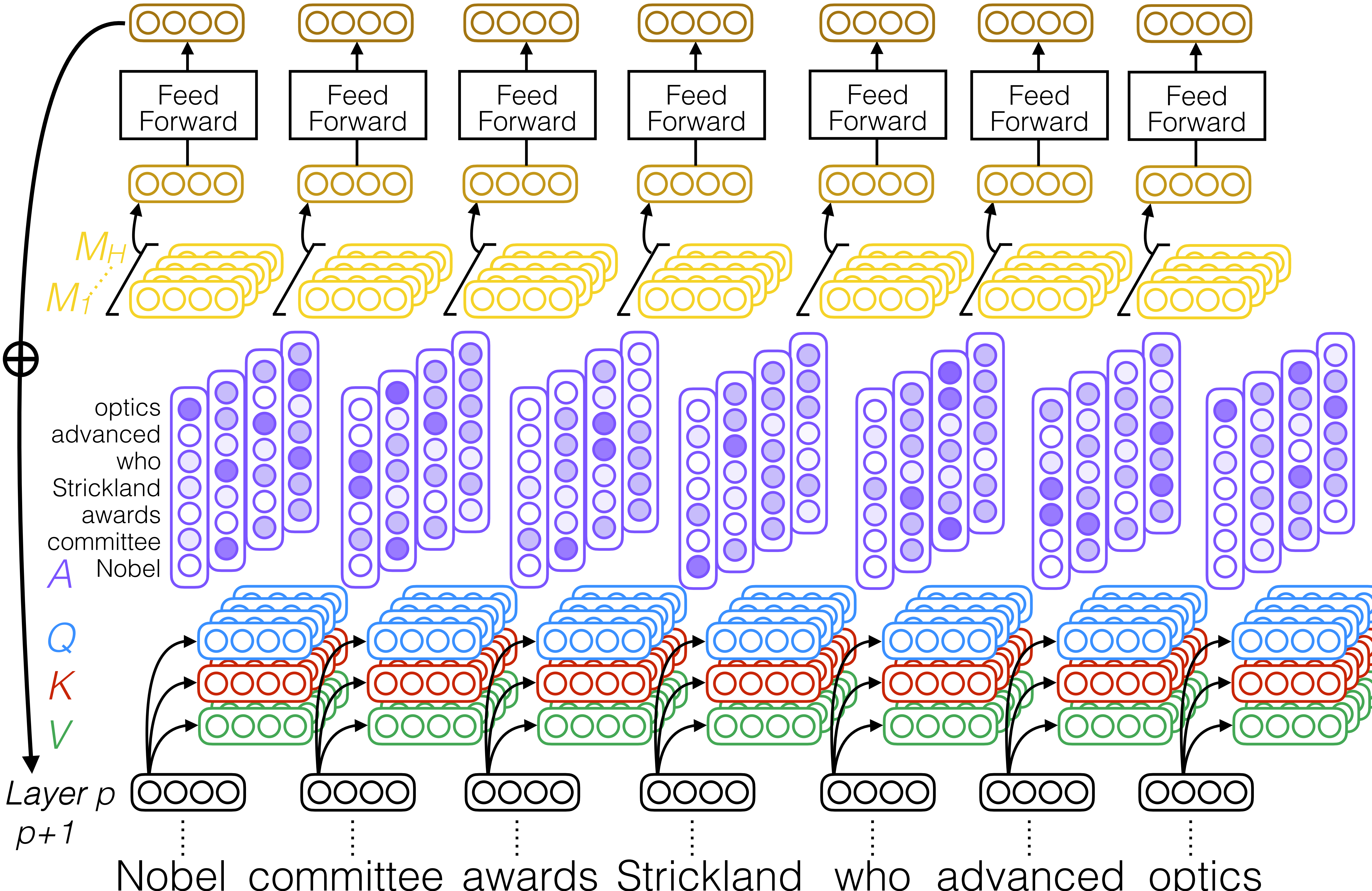
Multi-head self-attention



Slides by Emma Strubell!

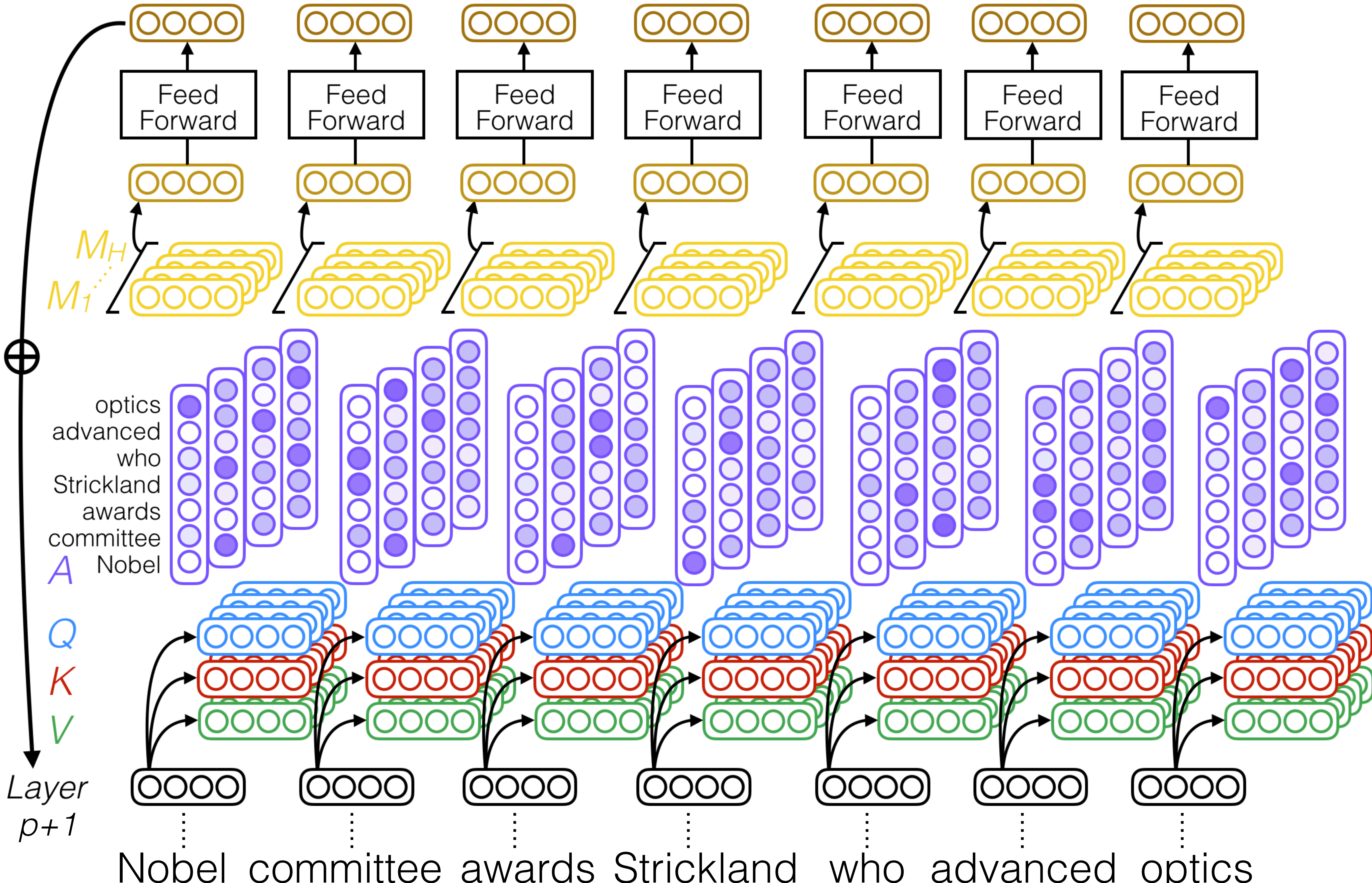
[Vaswani et al. 2017]

Multi-head self-attention



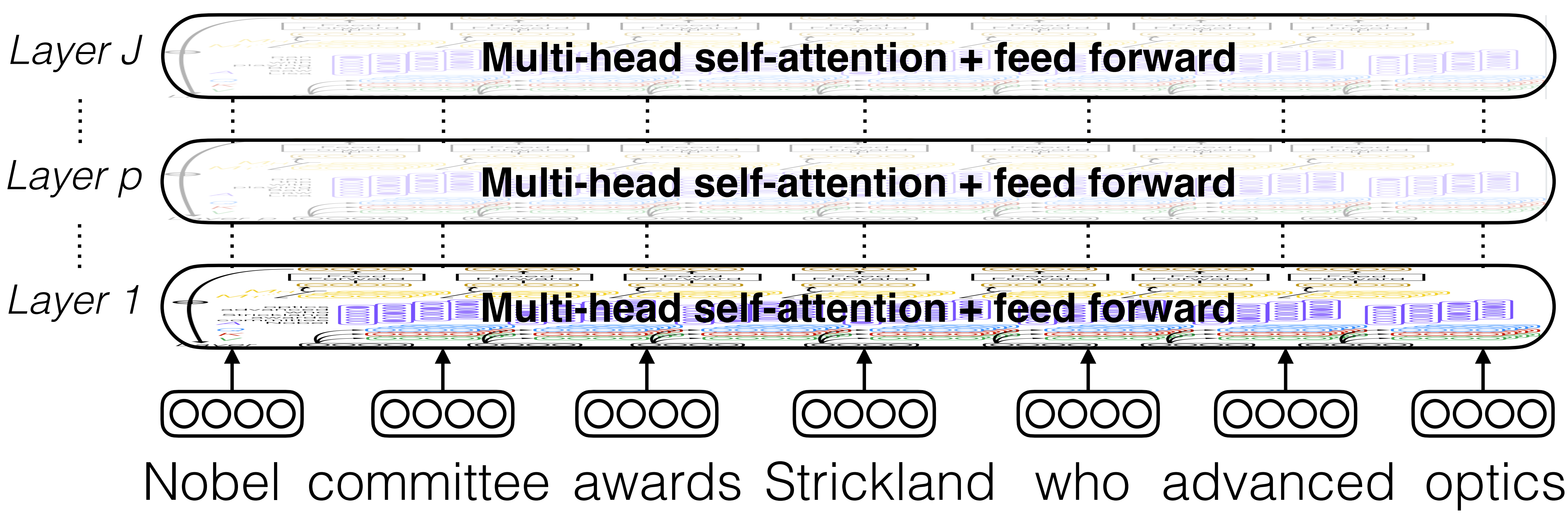
[Vaswani et al. 2017]

Multi-head self-attention

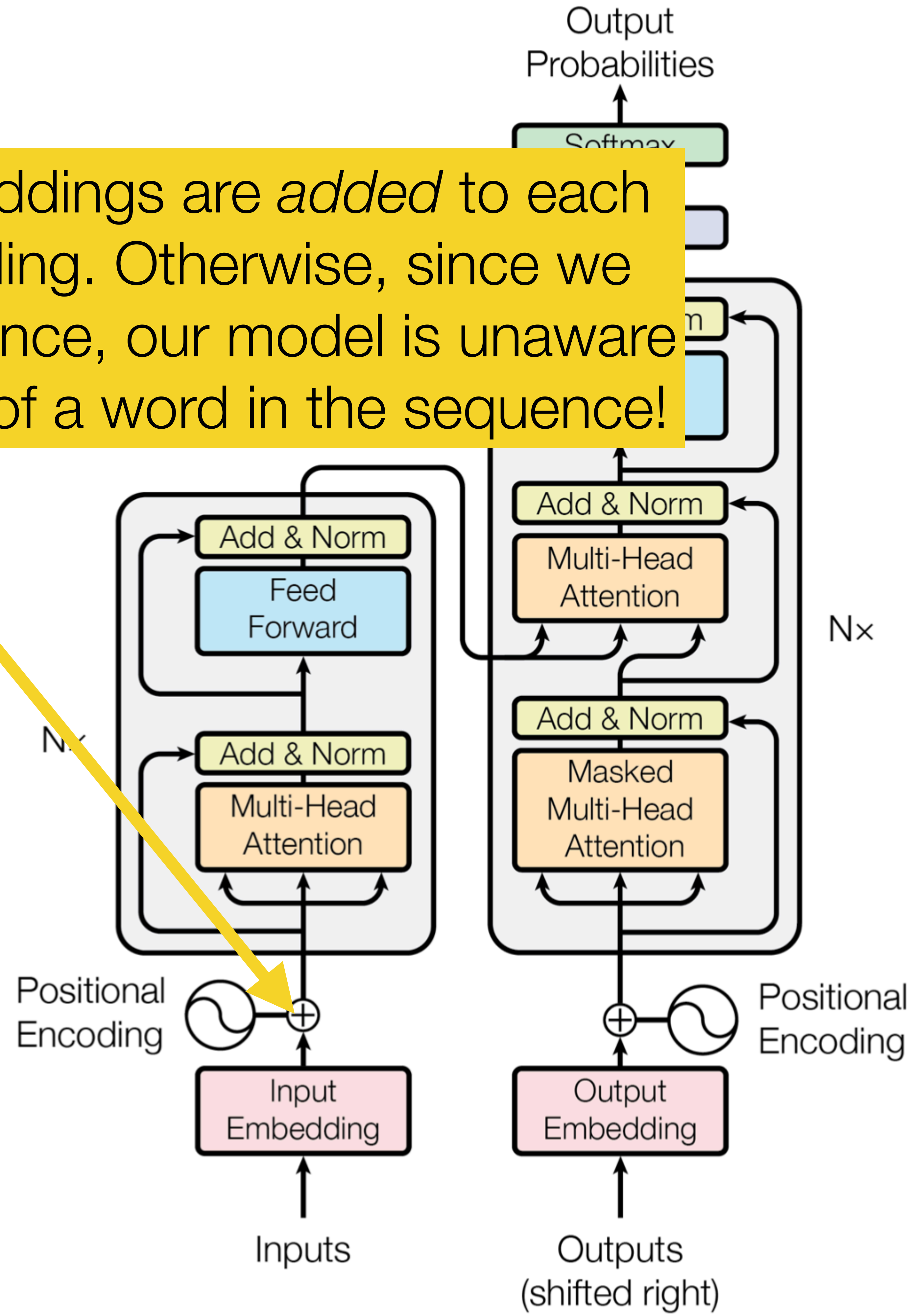


[Vaswani et al. 2017]

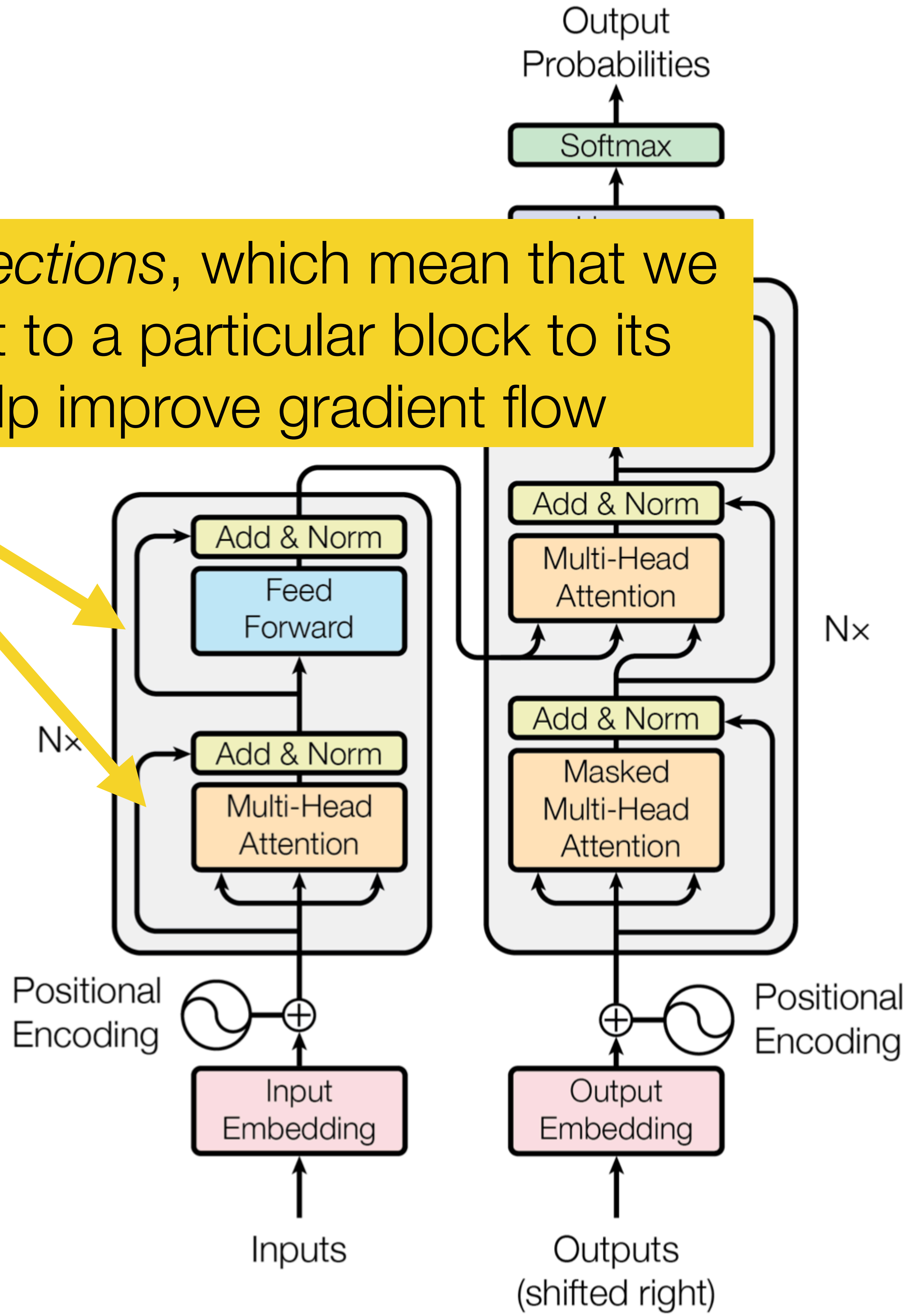
Multi-head self-attention



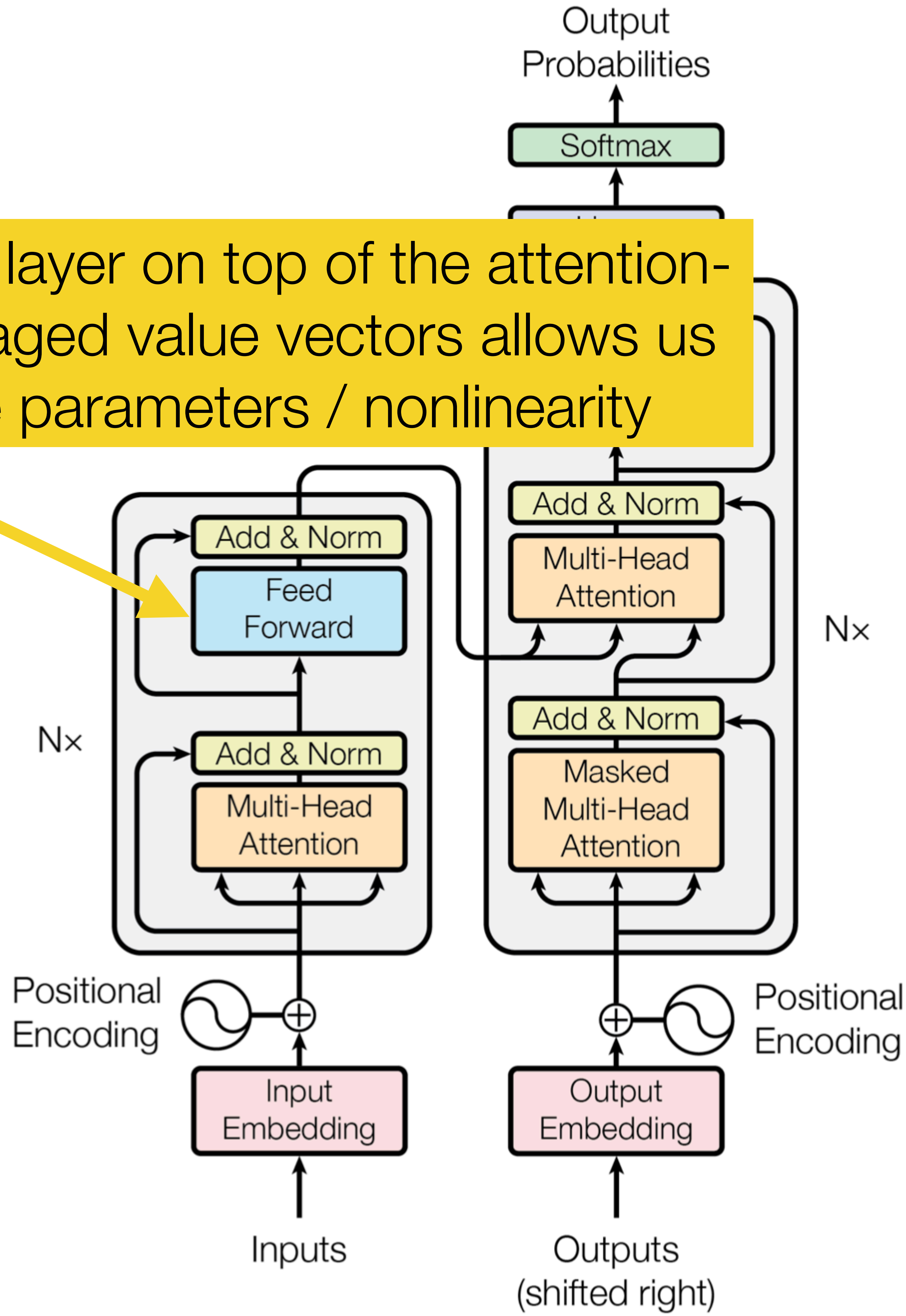
Position embeddings are *added* to each word embedding. Otherwise, since we have no recurrence, our model is unaware of the position of a word in the sequence!



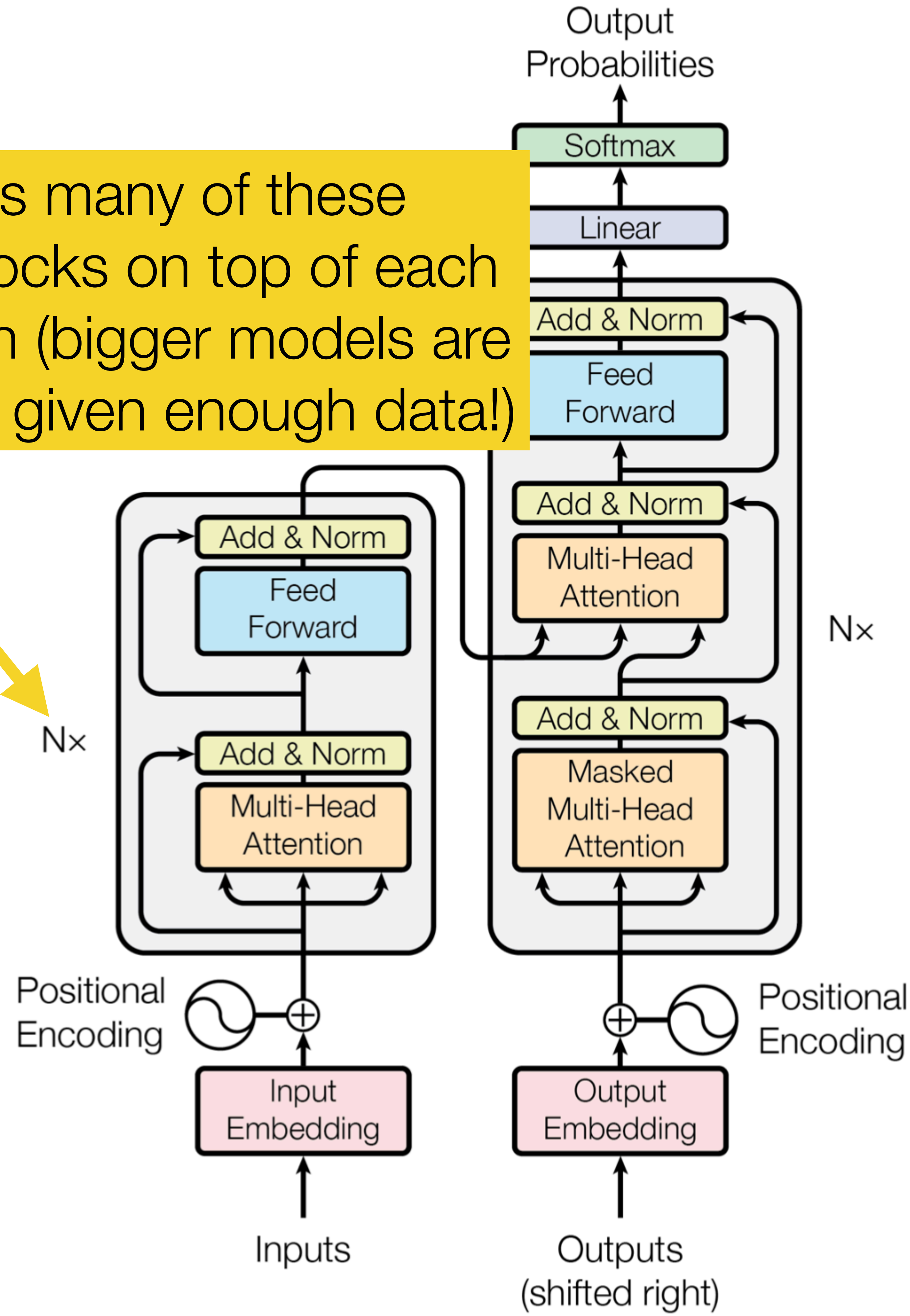
Residual connections, which mean that we add the input to a particular block to its output, help improve gradient flow



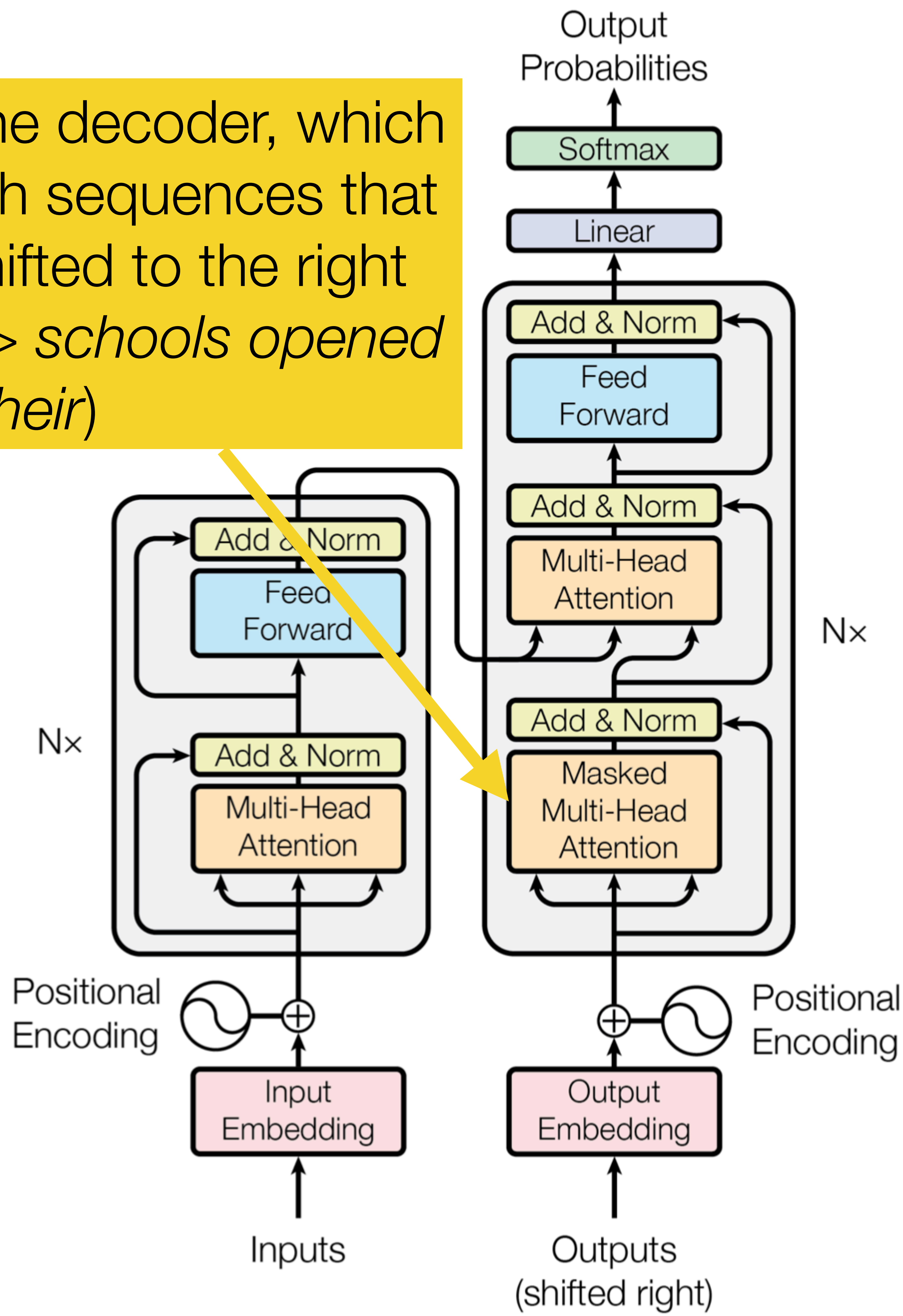
A feed-forward layer on top of the attention-weighted averaged value vectors allows us to add more parameters / nonlinearity



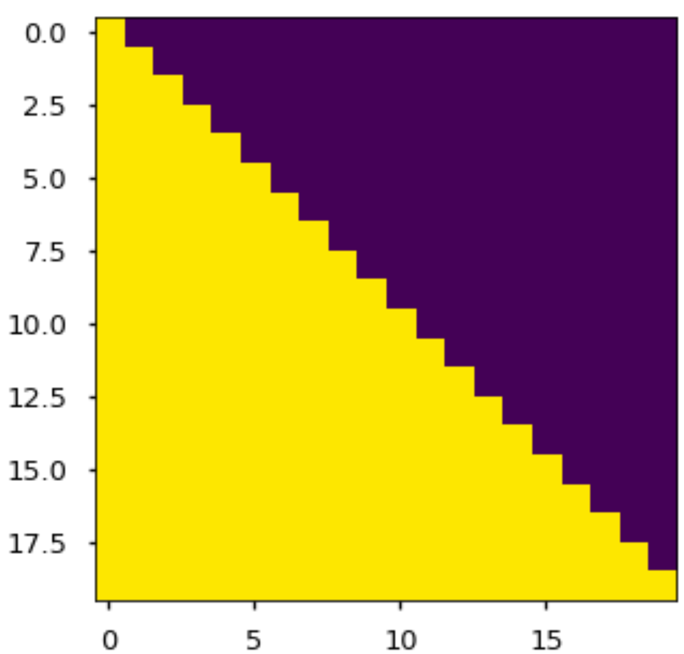
We stack as many of these *Transformer* blocks on top of each other as we can (bigger models are generally better given enough data!)

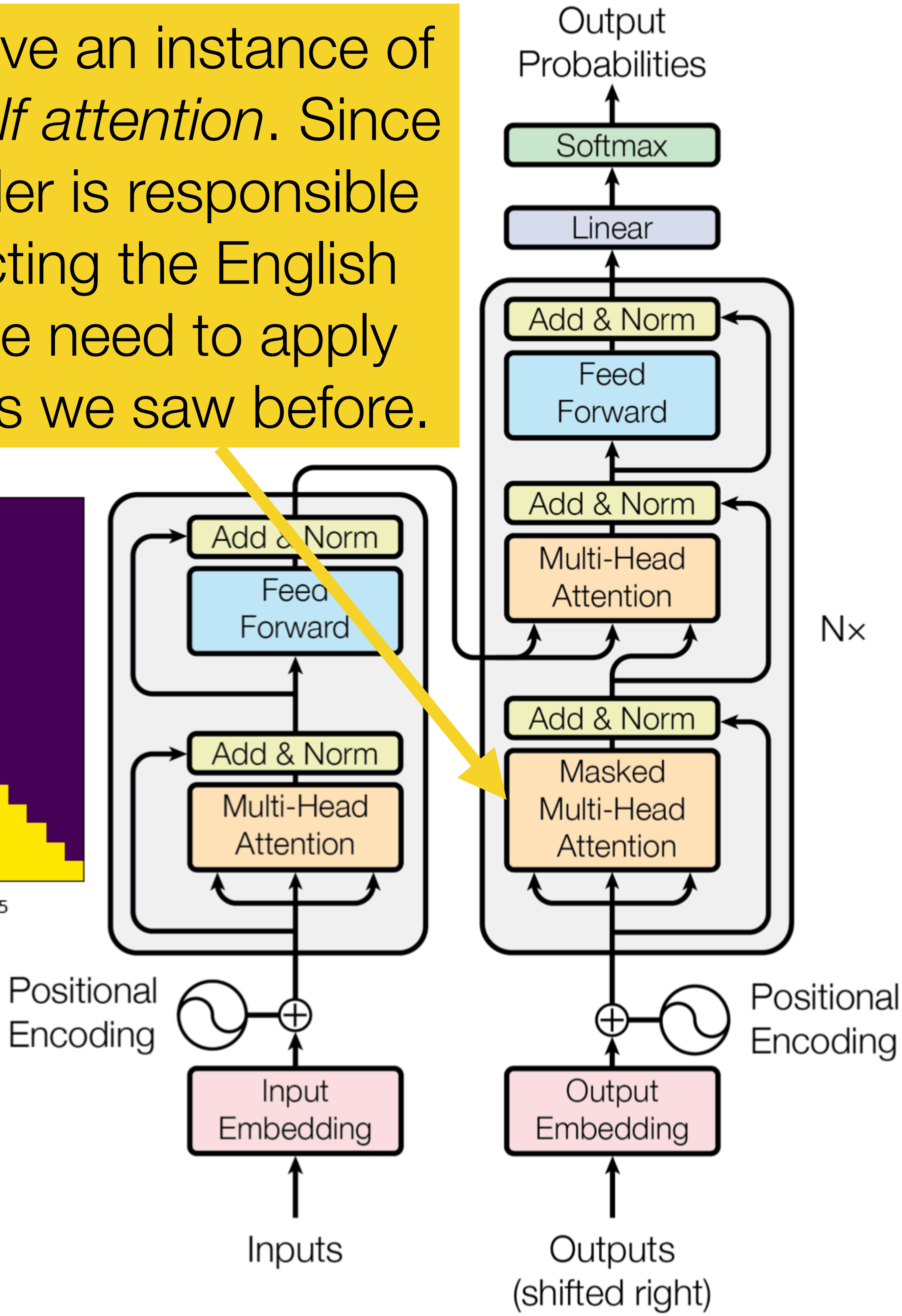


Moving onto the decoder, which takes in English sequences that have been shifted to the right (e.g., *<START> schools opened their*)



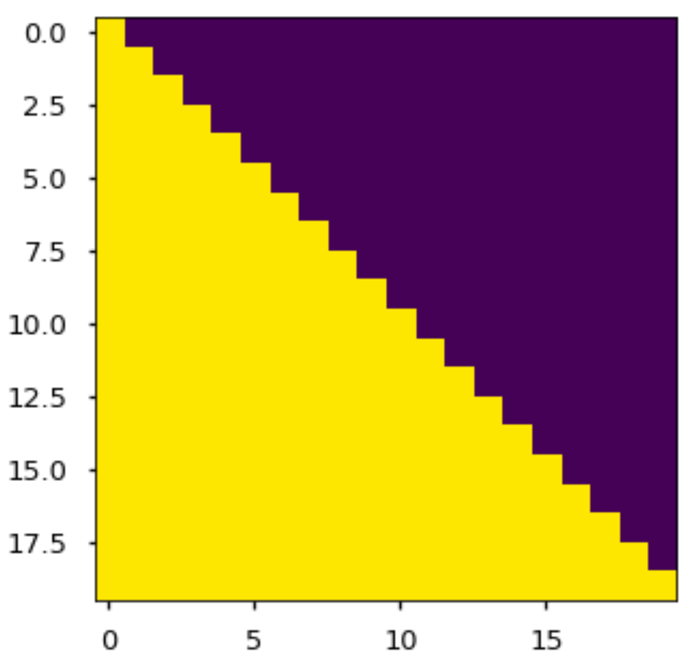
We first have an instance of *masked self attention*. Since the decoder is responsible for predicting the English words, we need to apply masking as we saw before.

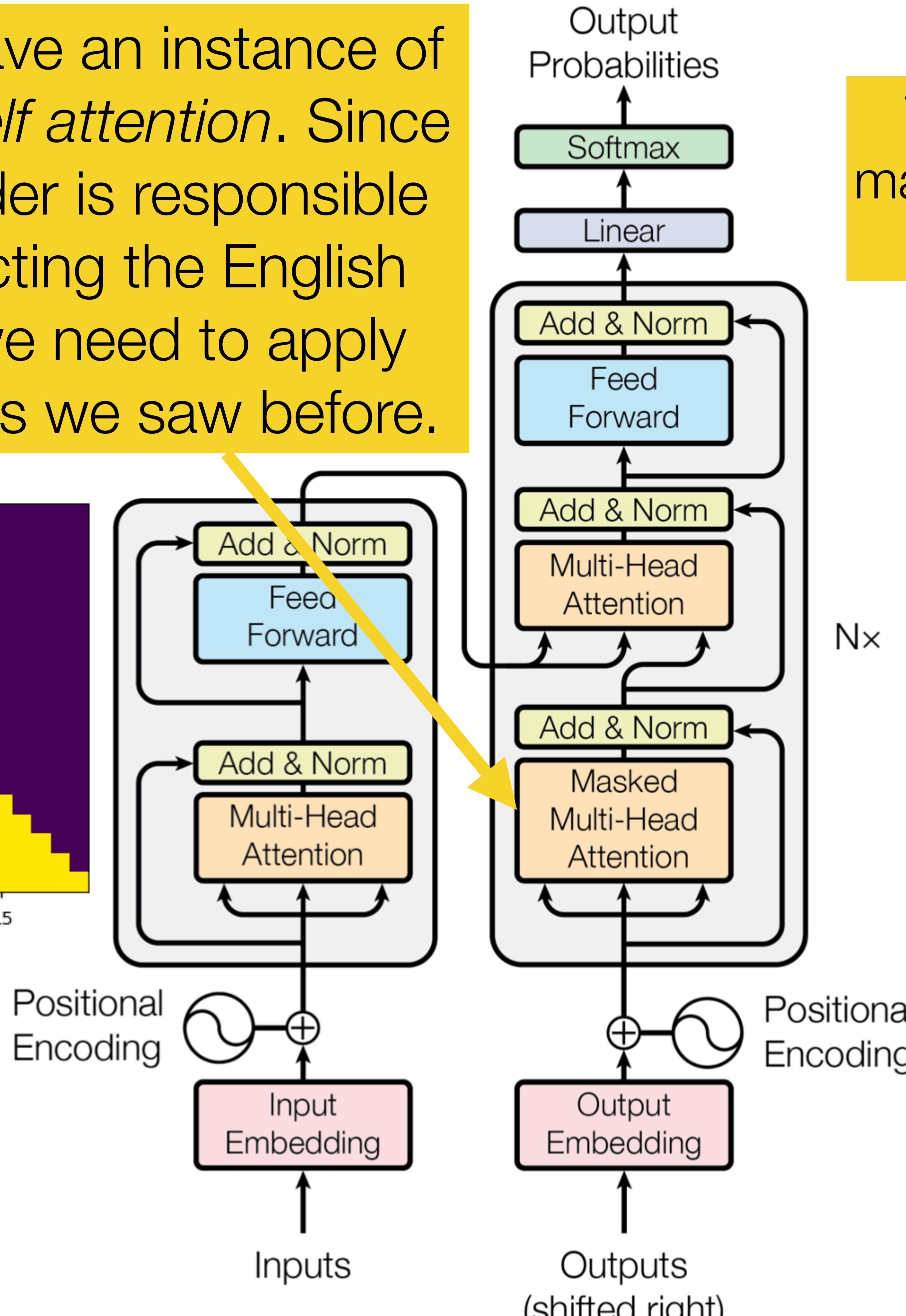




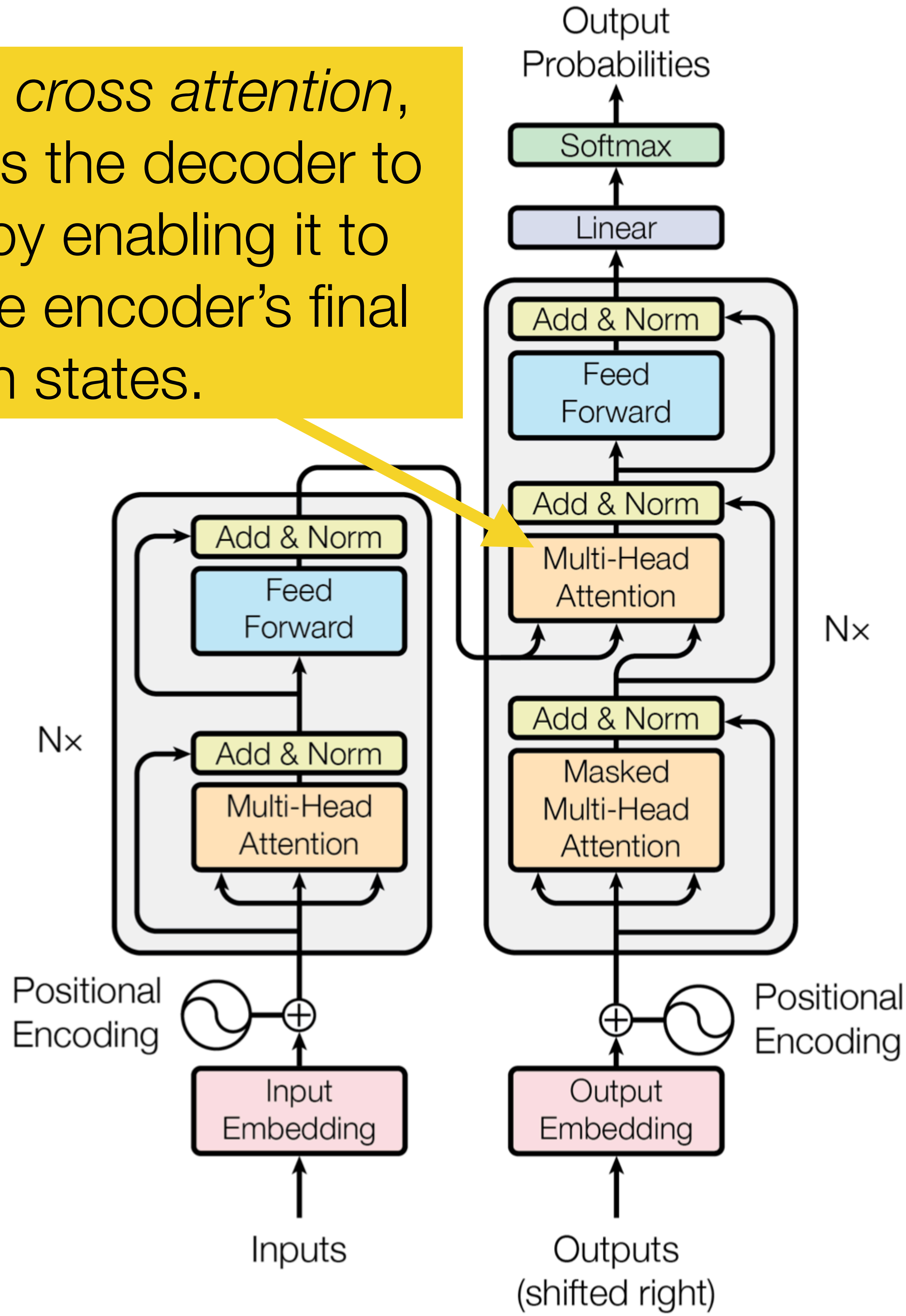
We first have an instance of *masked self attention*. Since the decoder is responsible for predicting the English words, we need to apply masking as we saw before.

Why don't we do masked self-attention in the encoder?

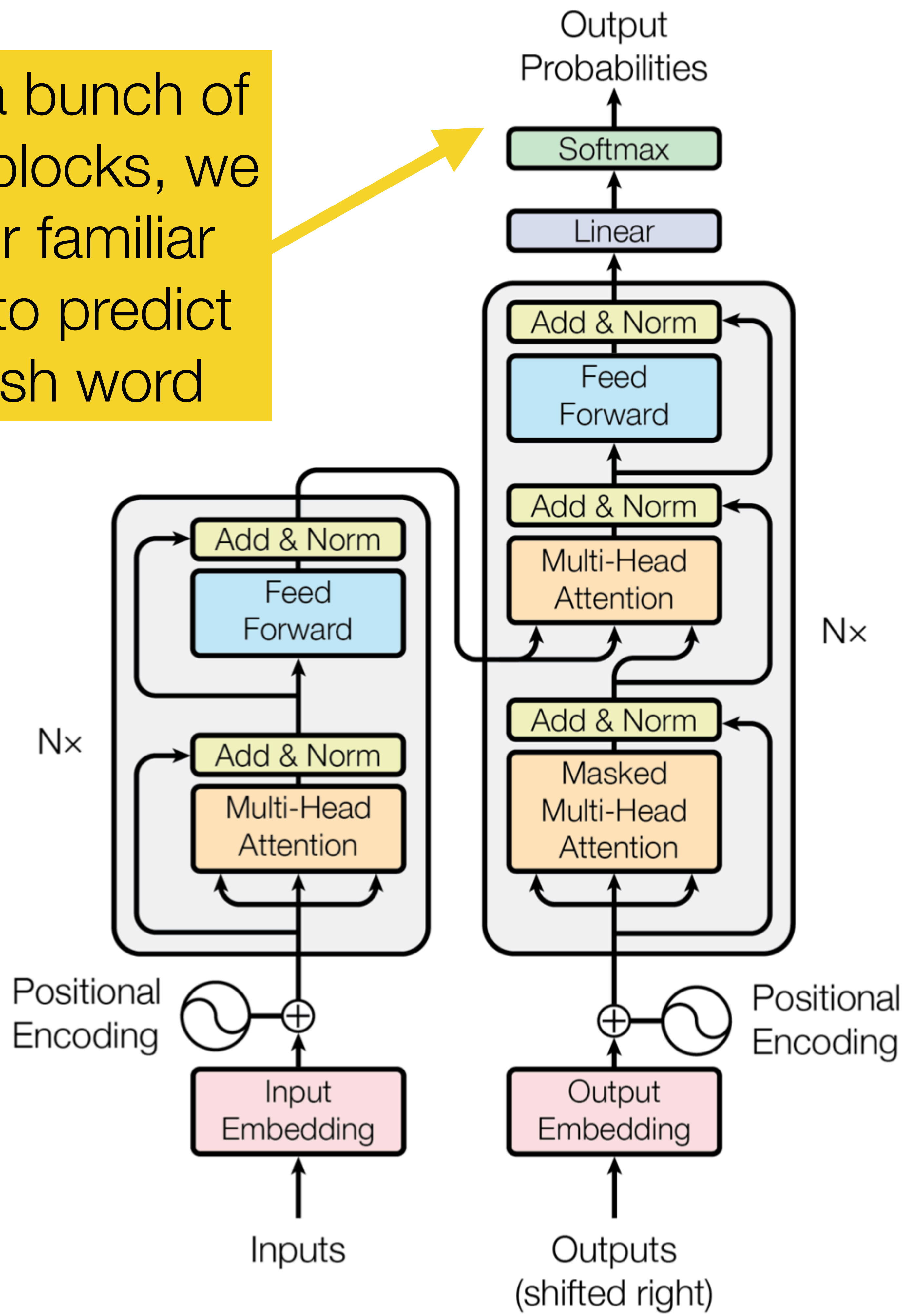




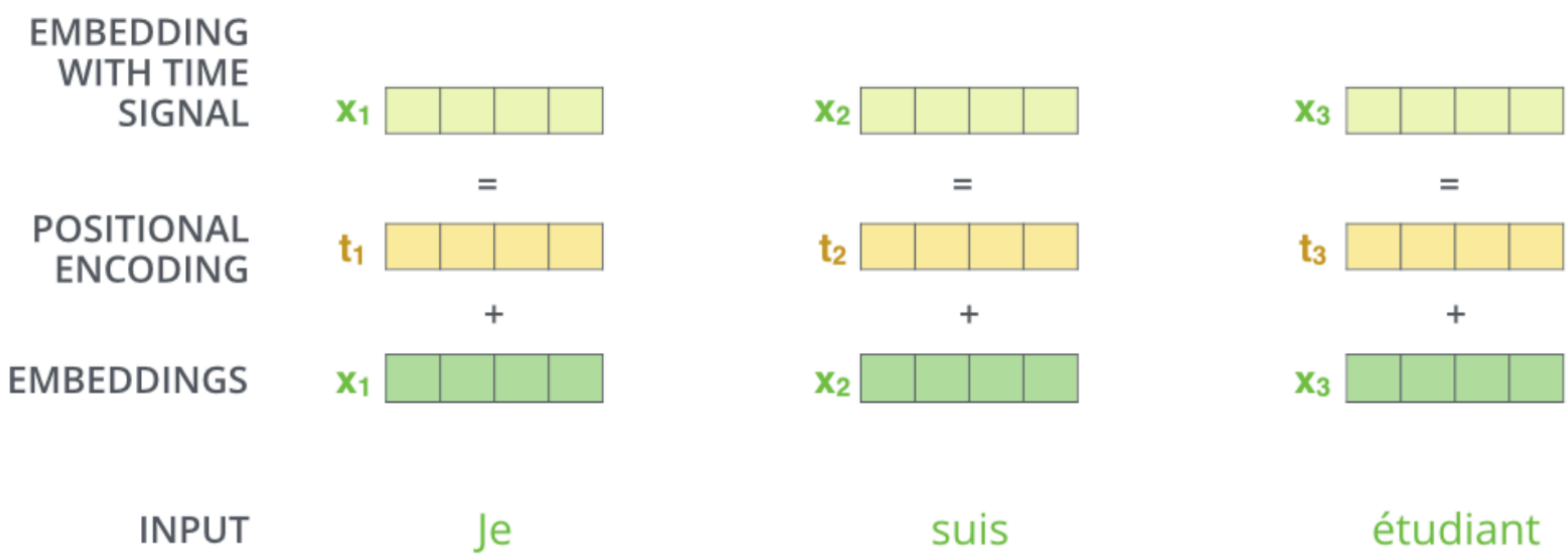
Now, we have *cross attention*, which connects the decoder to the encoder by enabling it to attend over the encoder's final hidden states.



After stacking a bunch of these decoder blocks, we finally have our familiar Softmax layer to predict the next English word



Positional encoding



Creating positional encodings?

- We could just concatenate a fixed value to each time step (e.g., 1, 2, 3, … 1000) that corresponds to its position, but then what happens if we get a sequence with 5000 words at test time?

- We want something that can generalize to arbitrary sequence lengths. We also may want to make attending to *relative positions* (e.g., tokens in a local window to the current token) easier.

- Distance between two positions should be consistent with variable-length inputs

Intuitive example

0 :	0	0	0	0	8 :	1	0	0	0
1 :	0	0	0	1	9 :	1	0	0	1
2 :	0	0	1	0	10 :	1	0	1	0
3 :	0	0	1	1	11 :	1	0	1	1
4 :	0	1	0	0	12 :	1	1	0	0
5 :	0	1	0	1	13 :	1	1	0	1
6 :	0	1	1	0	14 :	1	1	1	0
7 :	0	1	1	1	15 :	1	1	1	1

https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

Transformer positional encoding

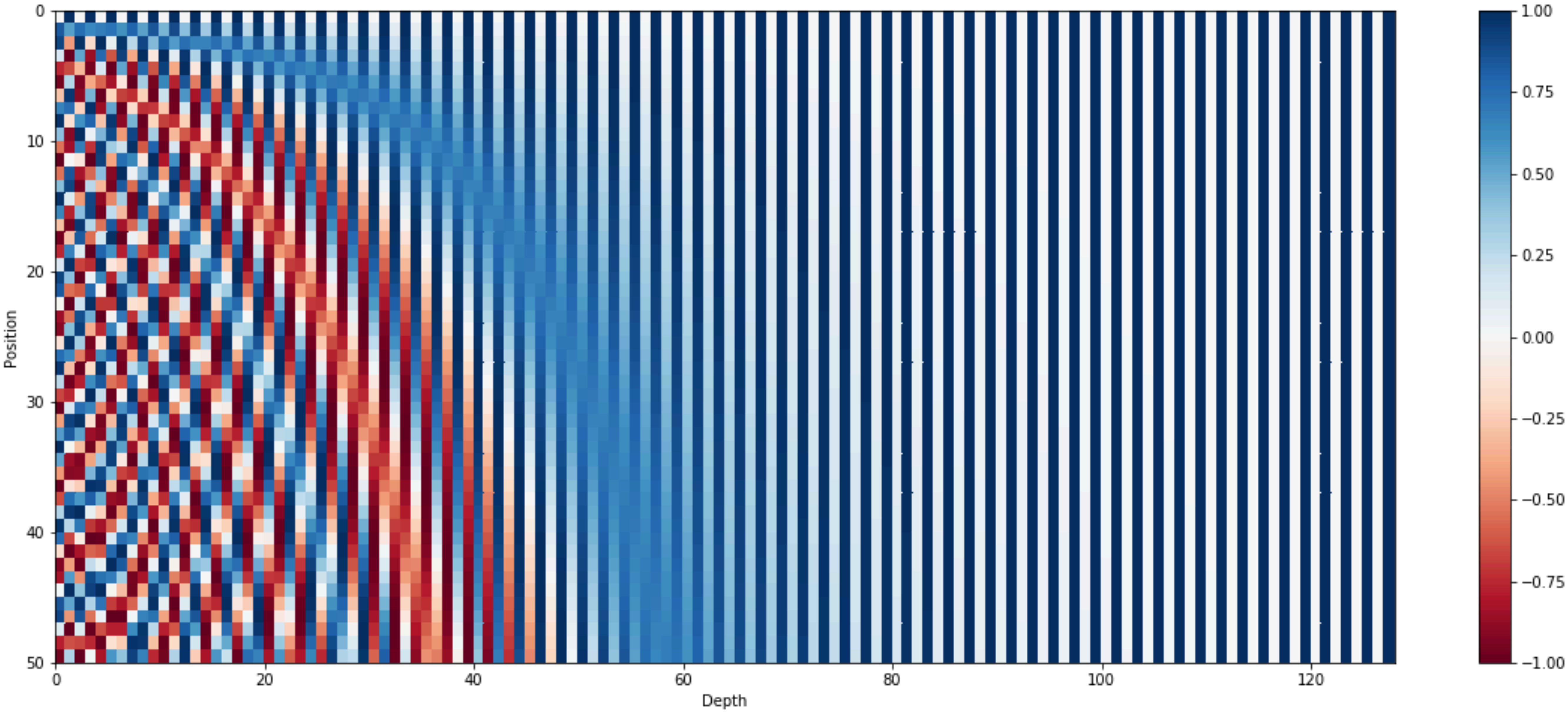
$$PE_{(pos,2i)} = \sin(\frac{pos}{10000^{2i/d_{model}}})$$

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

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

What does this look like?

(each row is the pos. emb. of a 50-word sentence)



https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

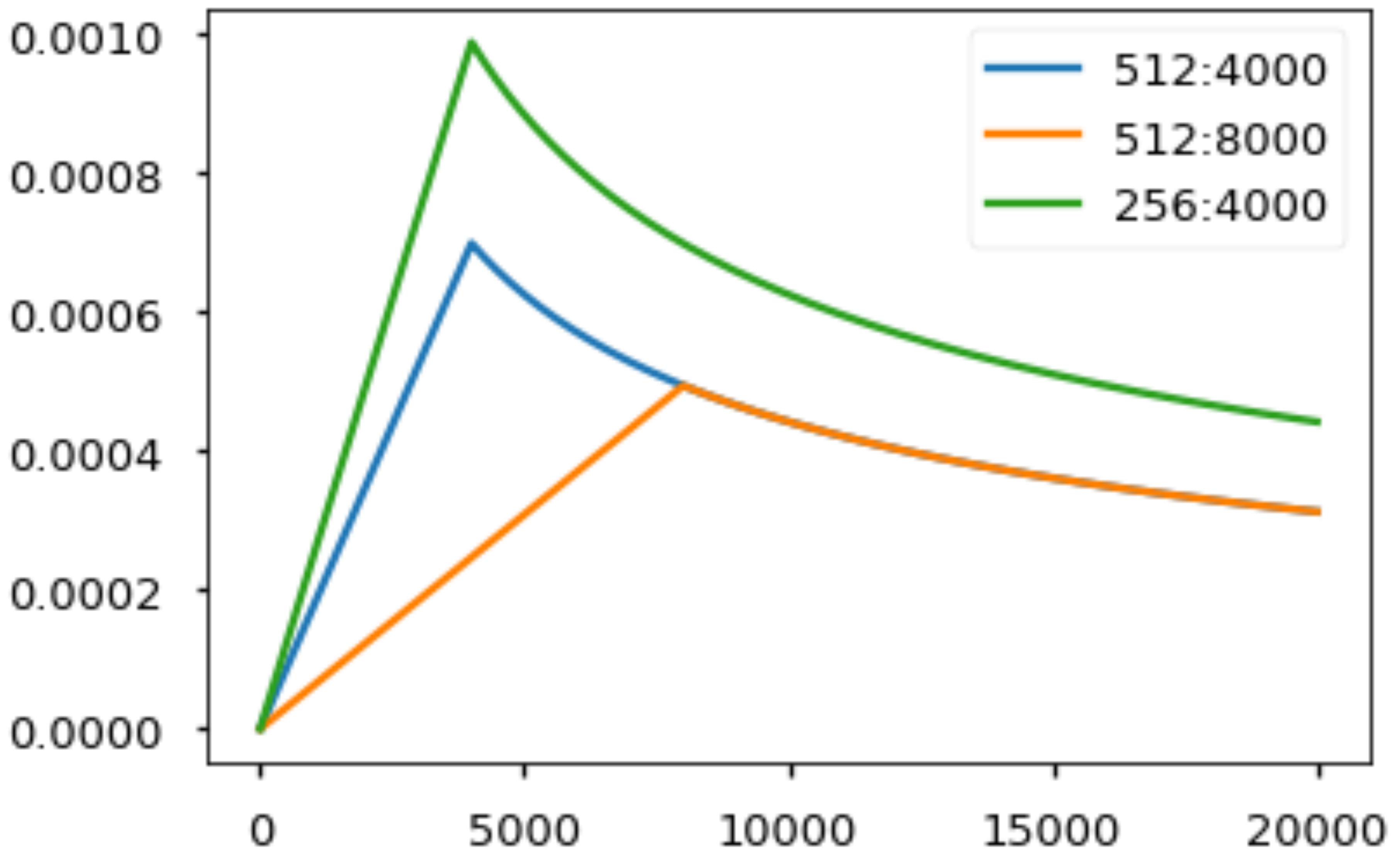
Despite the intuitive flaws, many models these days use *learned positional embeddings* (i.e., they cannot generalize to longer sequences, but this isn’t a big deal for their use cases)

Hacks to make Transformers work

Optimizer

We used the Adam optimizer (cite) with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We varied the learning rate over the course of training, according to the formula: $lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$ This corresponds to increasing the learning rate linearly for the first $warmup_steps$ training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used $warmup_steps = 4000$.

> *Note: This part is very important. Need to train with this setup of the model.*



Label Smoothing

During training, we employed label smoothing of value $\epsilon_{ls} = 0.1$ (cite). This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score.

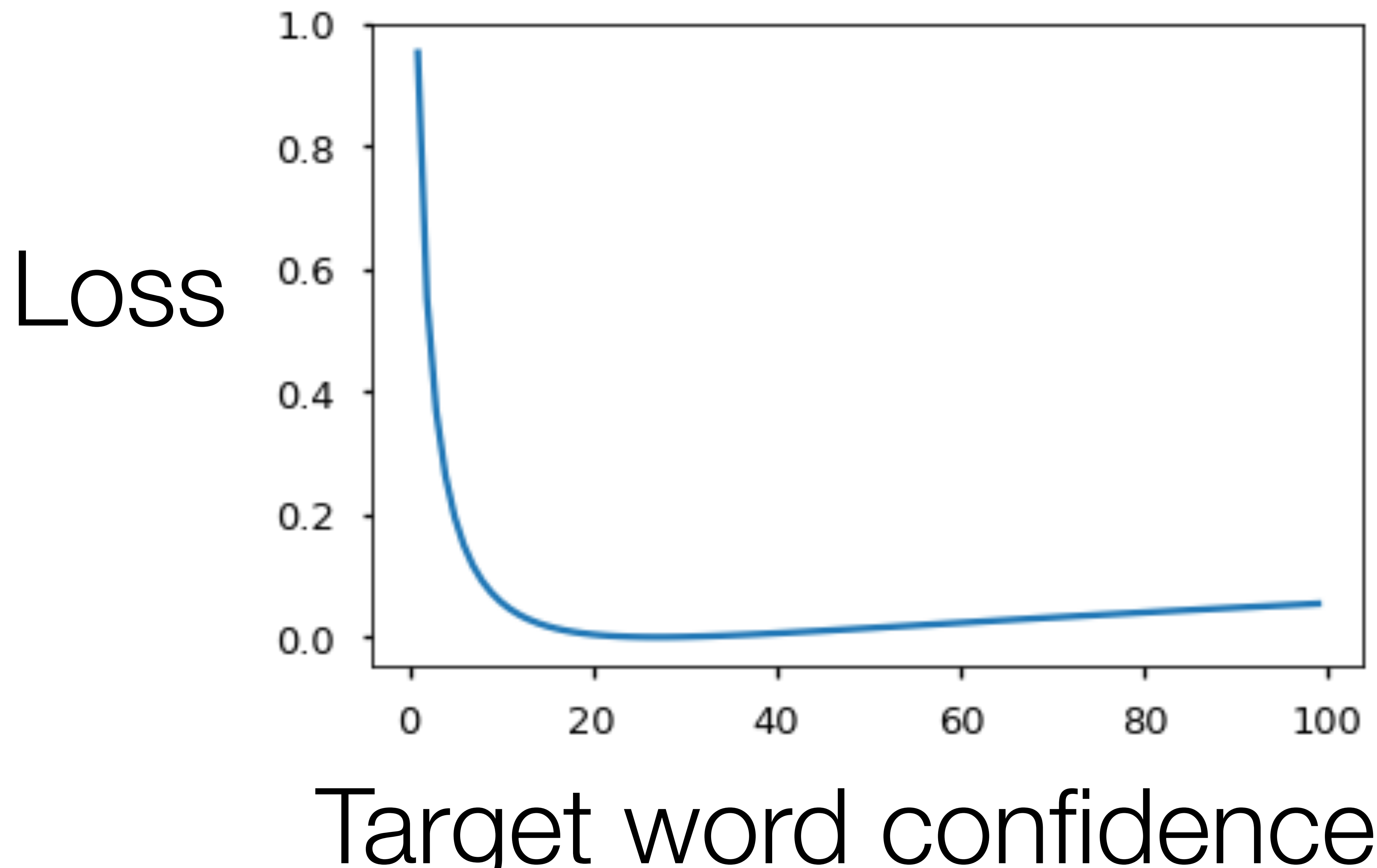
> *We implement label smoothing using the KL div loss. Instead of using a one-hot target distribution, we create a distribution that has `confidence` of the correct word and the rest of the `smoothing` mass distributed throughout the vocabulary.*

I went to class and took ___

cats	TV	notes	took	sofa
0	0	1	0	0
0.025	0.025	0.9	0.025	0.025

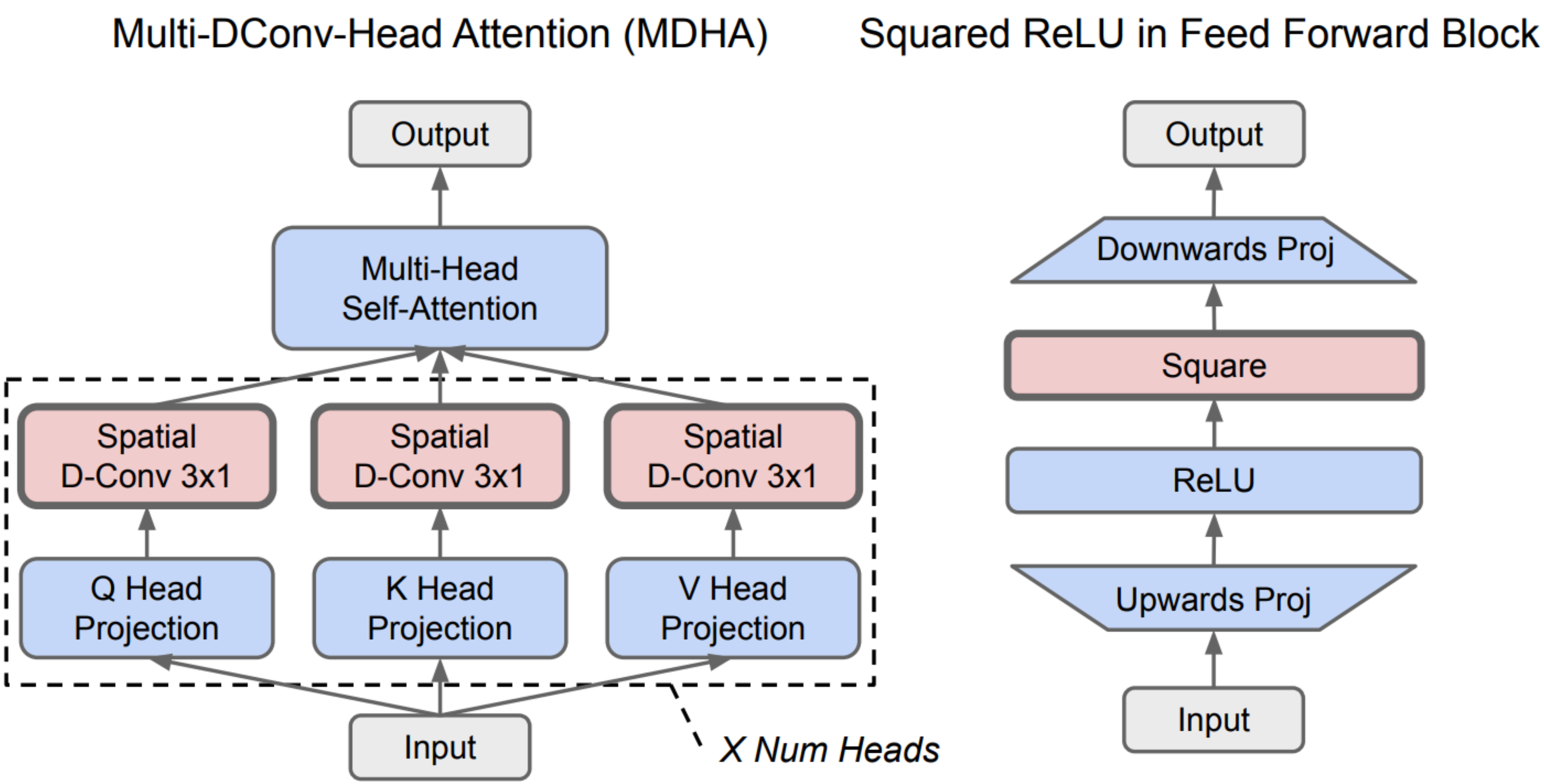
with label smoothing

Get penalized for overconfidence!



Why these decisions?

Unsatisfying answer: they empirically worked well.
Neural architecture search finds even better Transformer variants:



Primer: Searching for efficient Transformer architectures… So et al., Sep. 2021

OpenAI’s Transformer LMs

- GPT (Jun 2018): 117 million parameters, trained on 13GB of data (~1 billion tokens)
- GPT2 (Feb 2019): 1.5 billion parameters, trained on 40GB of data
- GPT3 (July 2020): 175 billion parameters, ~500GB data (300 billion tokens)

Coming up!

- Transfer learning via Transformer models like BERT
- Tokenization (word vs subword vs character/byte)
- Prompt-based learning
- Efficient / long-range Transformers
- Downstream tasks