Transformers

some slides from Emma Strubell, M Iyyer , Hedu Ai, C Manning

Announcement

- Quiz 2: 29th March (syllabus: Minor 2 syllabus)
- Assignment 3 will be floated in the first week of April
- Quiz 3: in April
- Bonus Quiz in April

- For paper reading, please follow the sequence, not the dates.
- Two papers per class

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*

Whiteboard

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

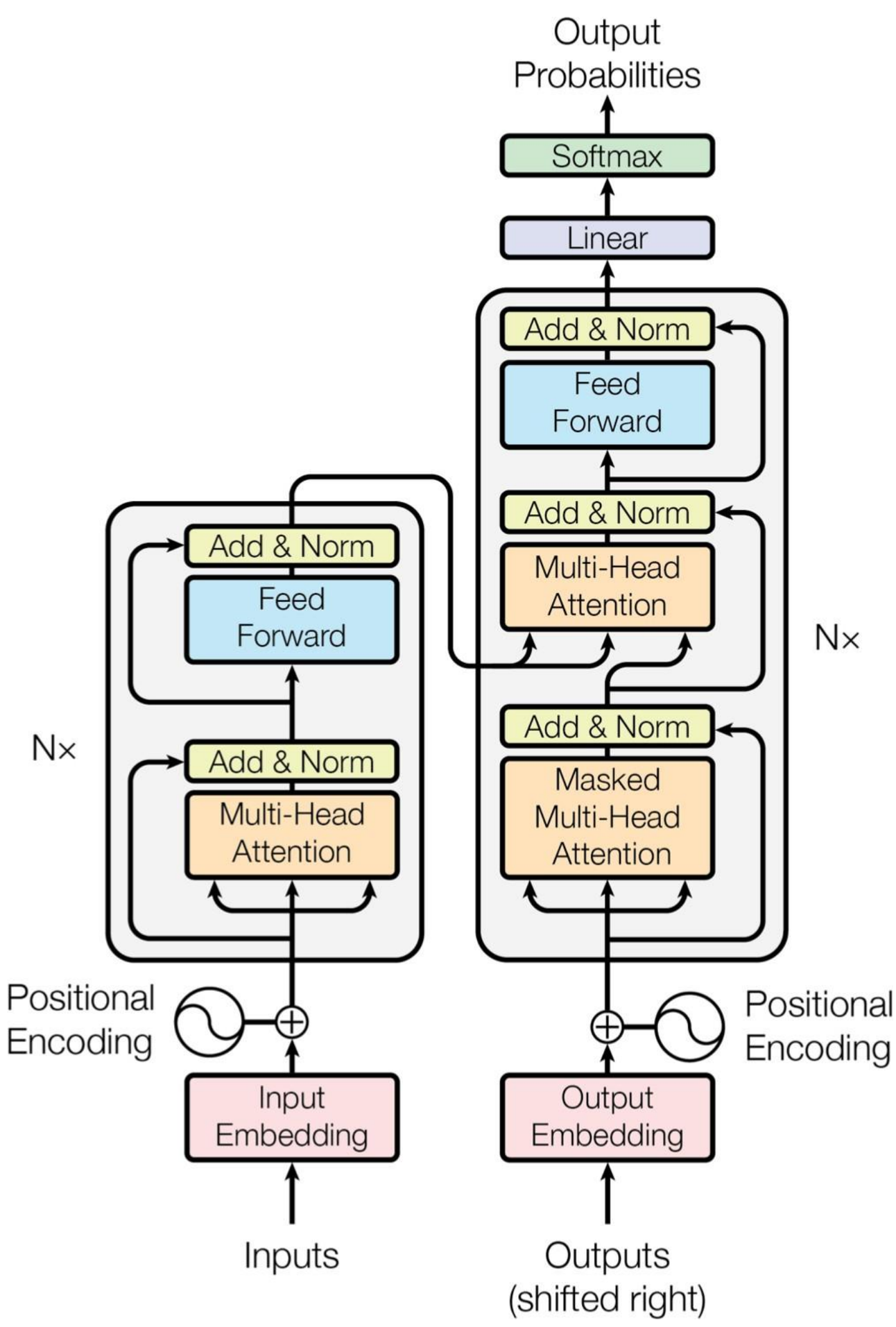
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

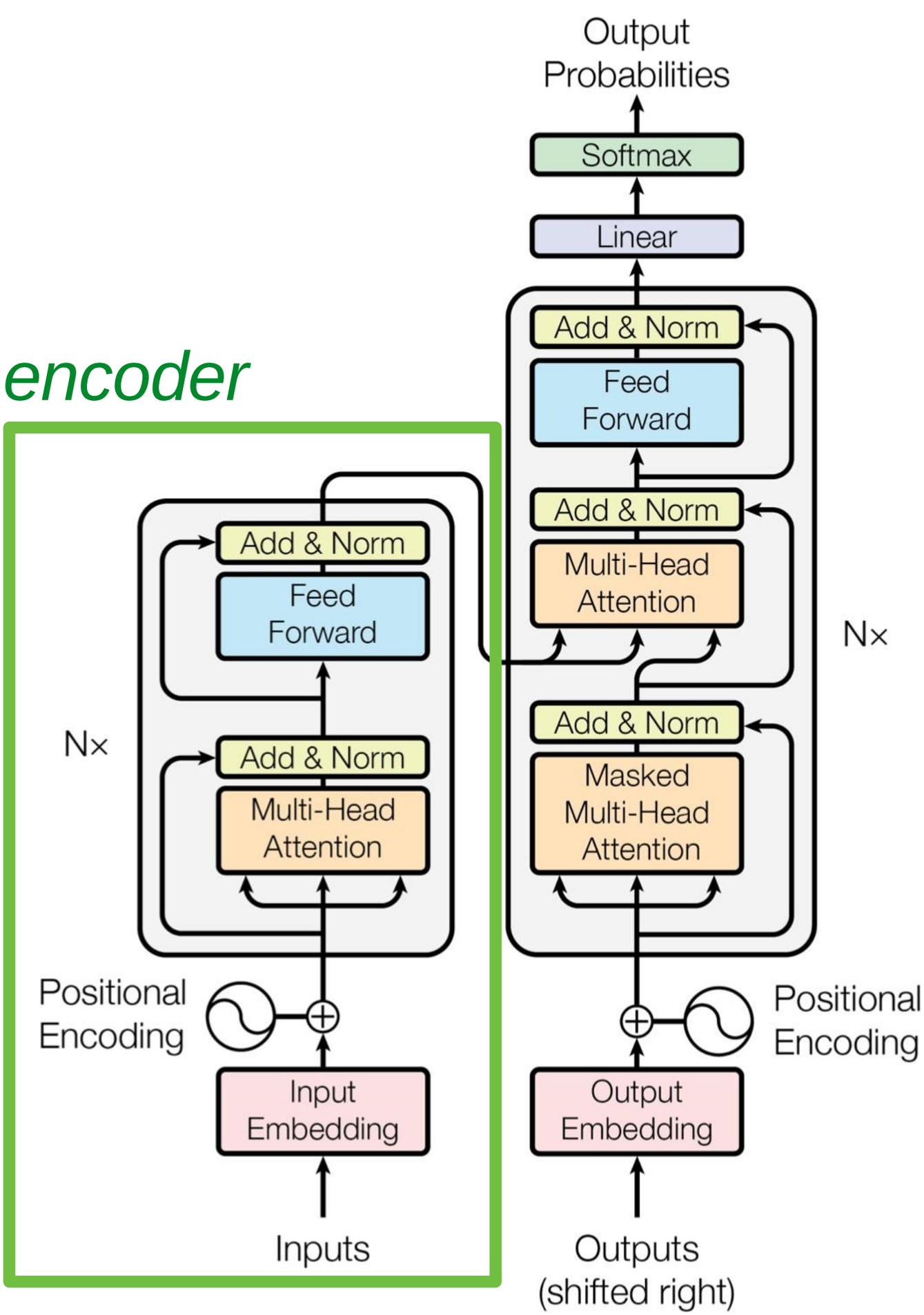
Attention is all you need

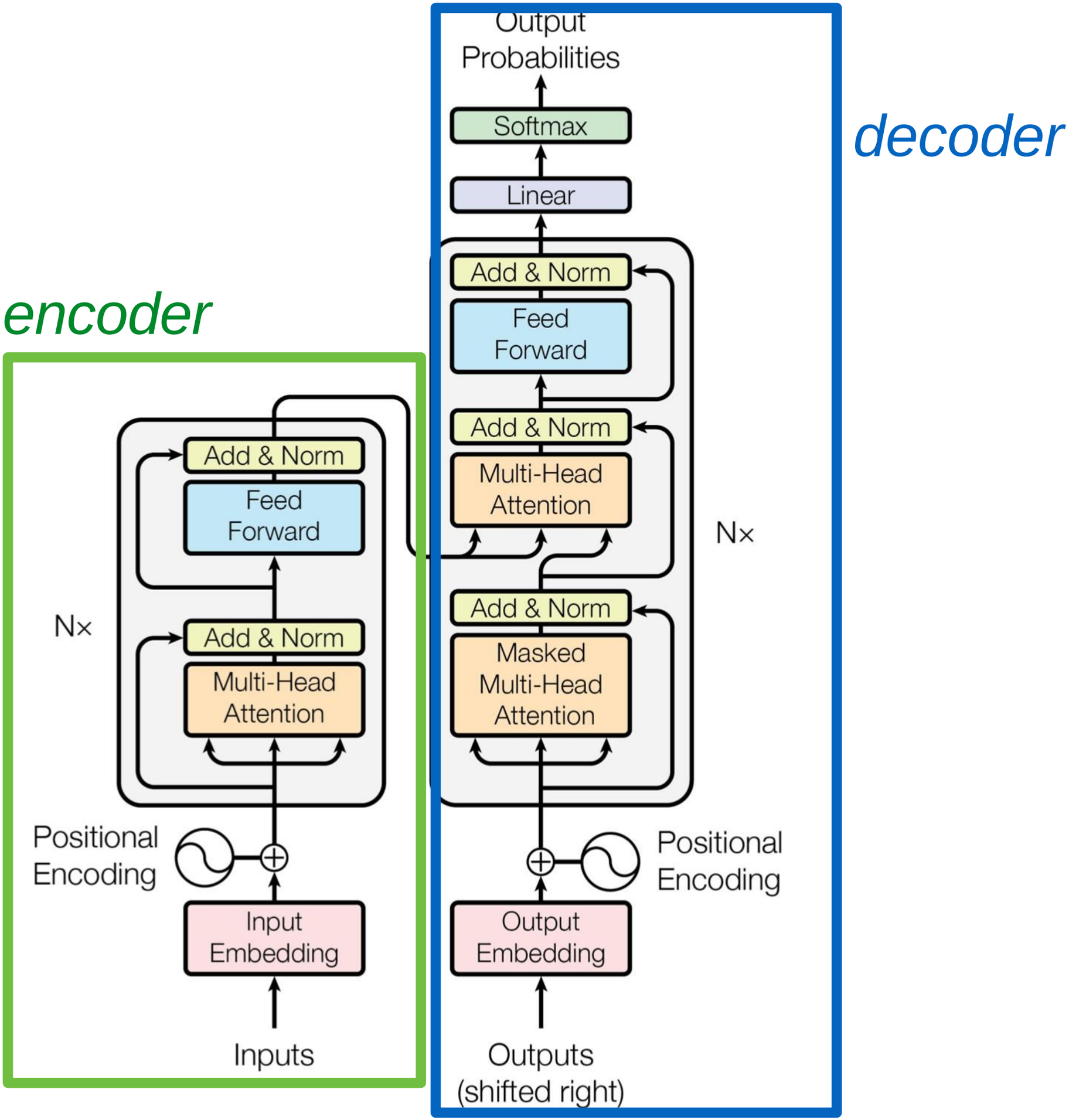
A Vaswani, N Shazeer, N Parmar... - Advances in neural ..., 2017 - proceedings.neurips.cc

The dominant sequence transduction models are based on complex recurrent orconvolutional neural networks in an encoder and decoder configuration. The best performing such models also connect the encoder and decoder through an attentionm echanisms. We propose a novel, simple network architecture based solely onan attention mechanism, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superiorin quality while being more ...

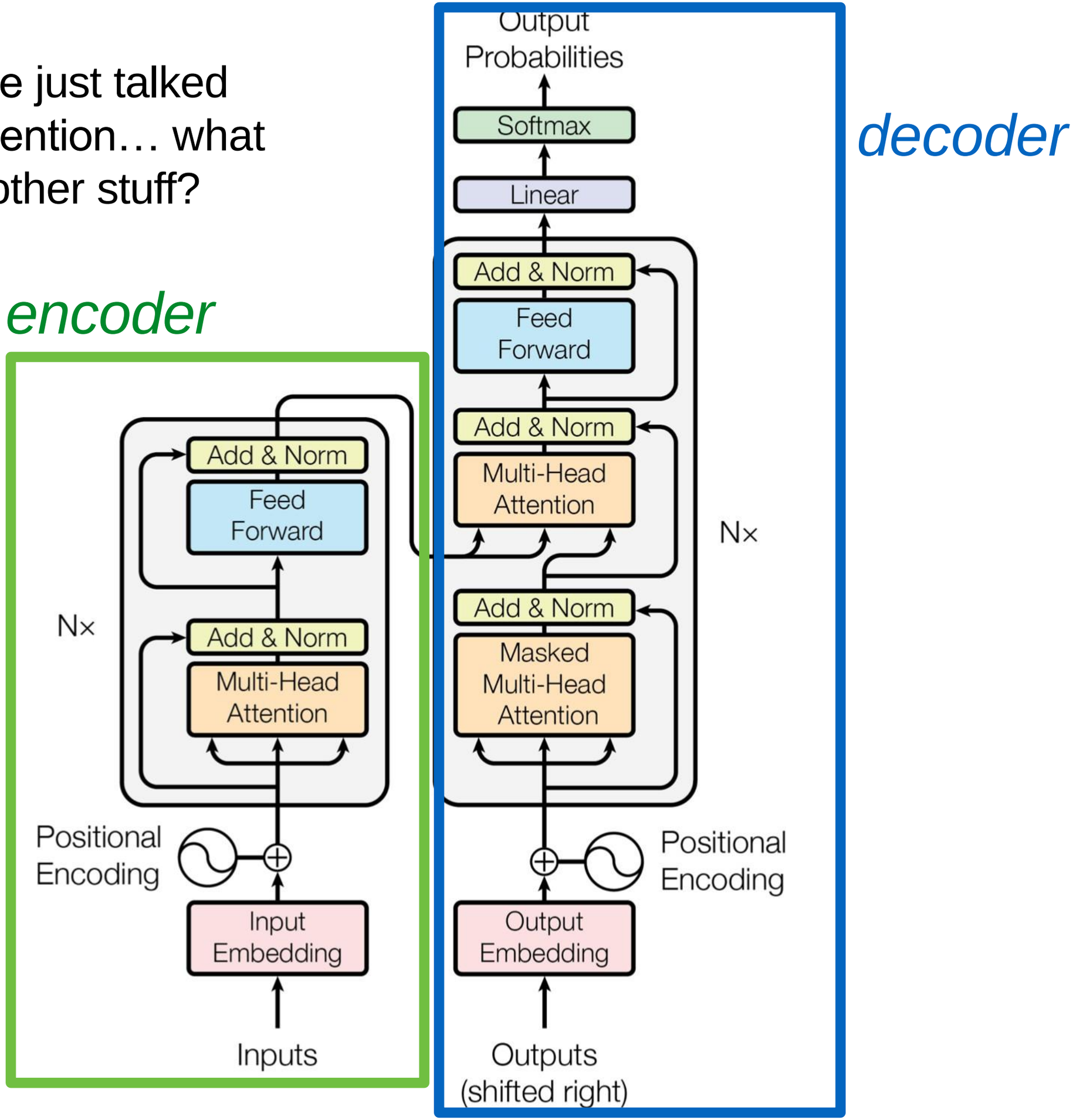
☆ Save 99 Cite Cited by 69087 Related articles All 46 versions »


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
Inputs
Positional Encoding
Output Embedding
Outputs (shifted right)


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)


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

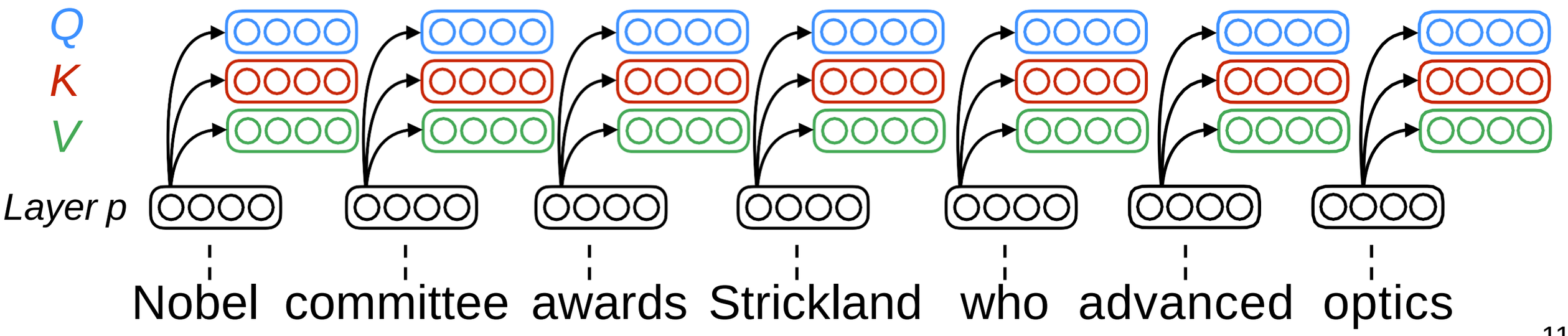
So far we've just talked about self-attention… what is all this other stuff?



Slides by Emma Strubell!

[Vaswani et al. 2017]

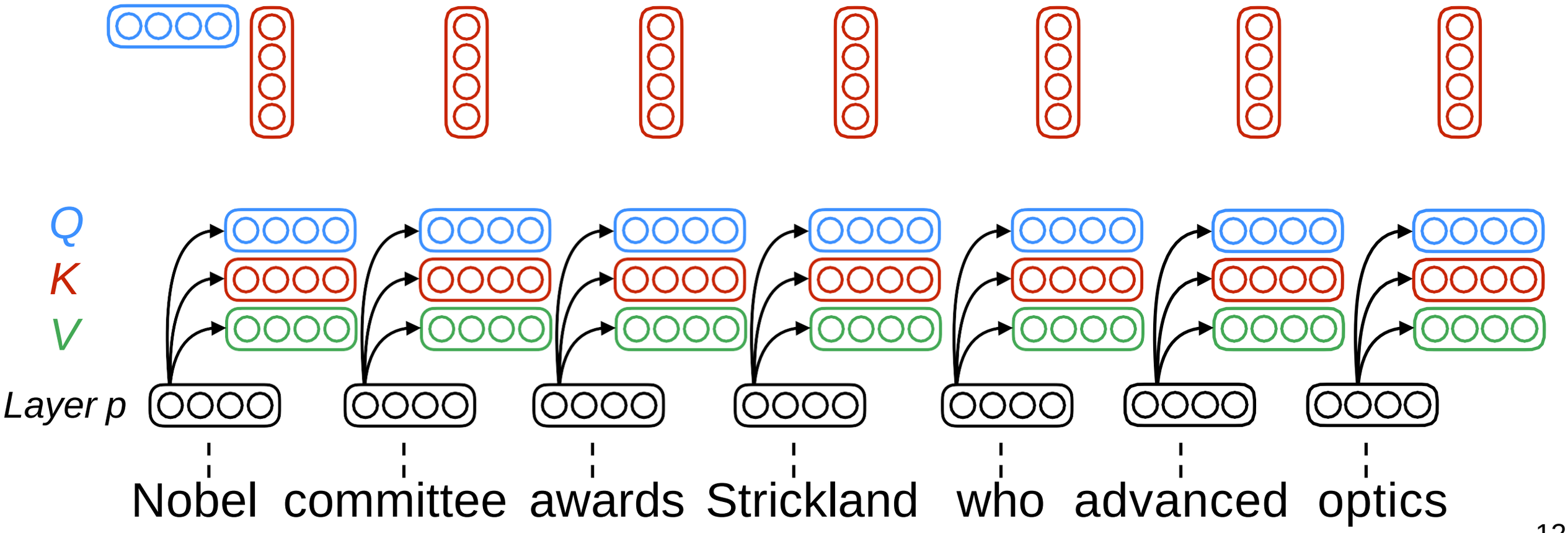
Self-attention (in encoder)



Slides by Emma Strubell!

[Vaswani et al. 2017]

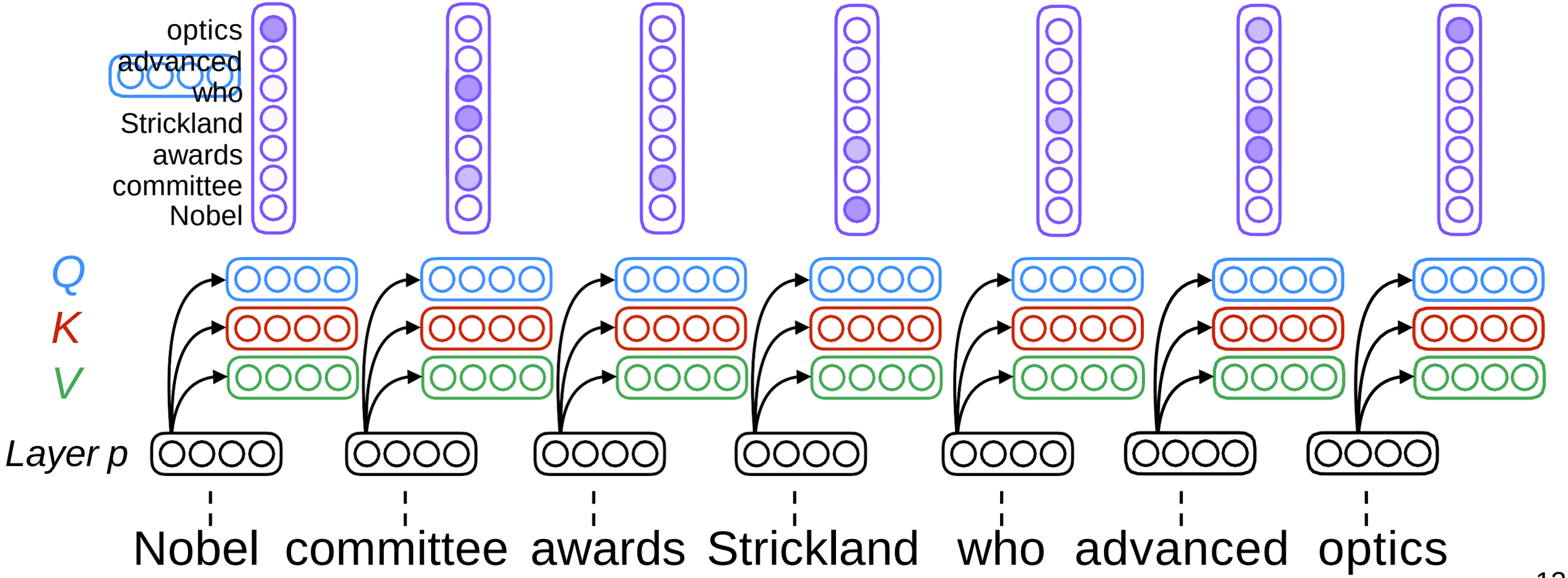
Self-attention (in encoder)



Slides by Emma Strubell!

[Vaswani et al. 2017]

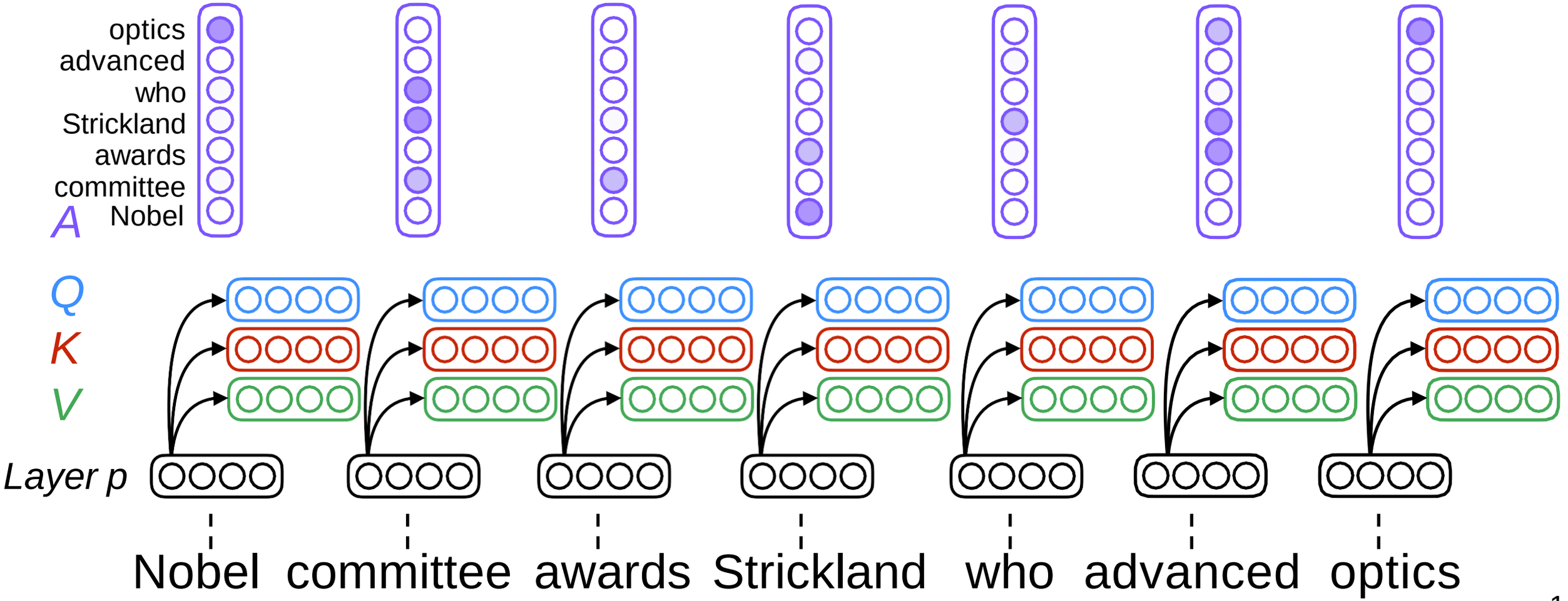
Self-attention (in encoder)



Slides by Emma Strubell!

[Vaswani et al. 2017]

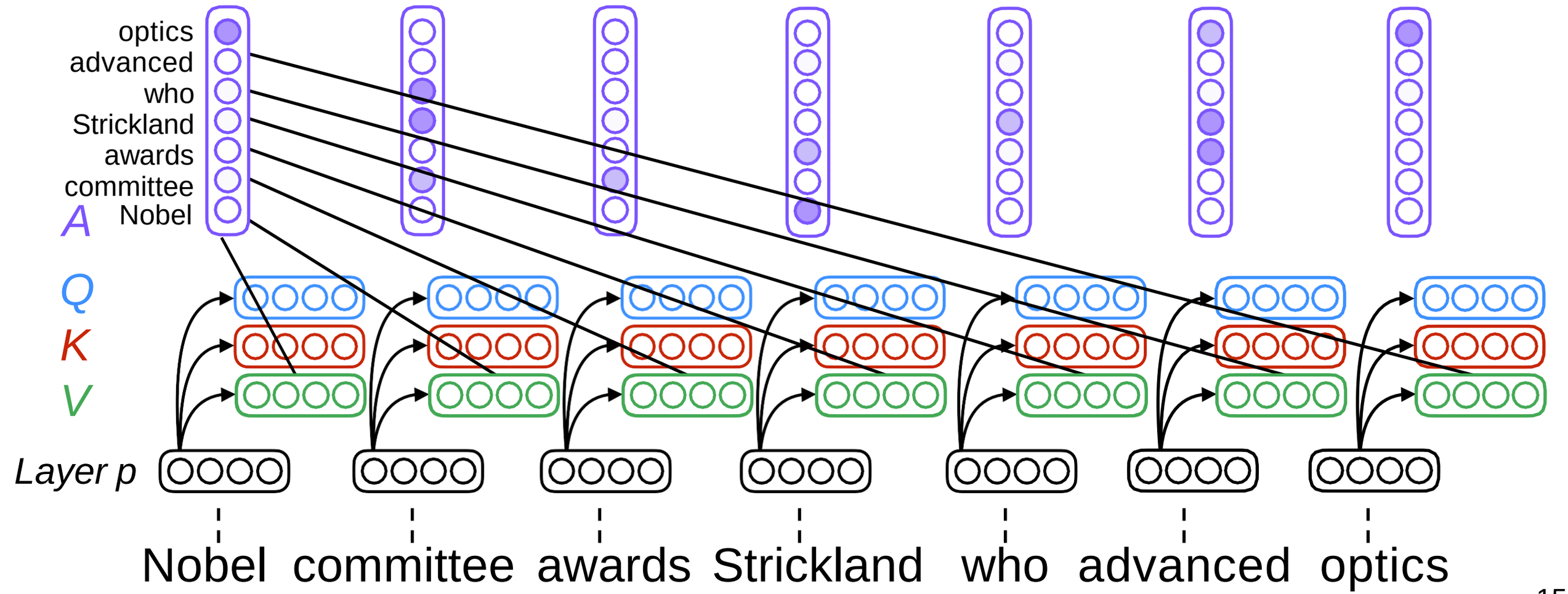
Self-attention (in encoder)



Slides by Emma Strubell!

[Vaswani et al. 2017]

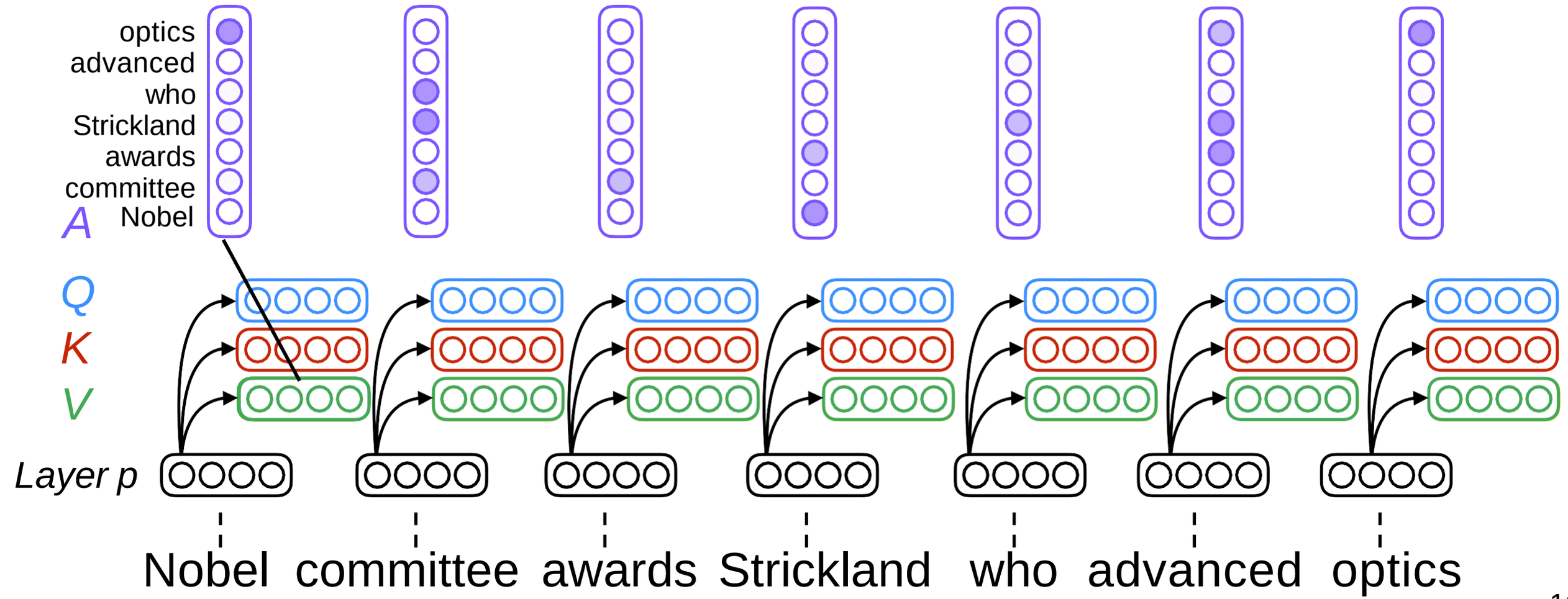
Self-attention (in encoder)



Slides by Emma Strubell!

[Vaswani et al. 2017]

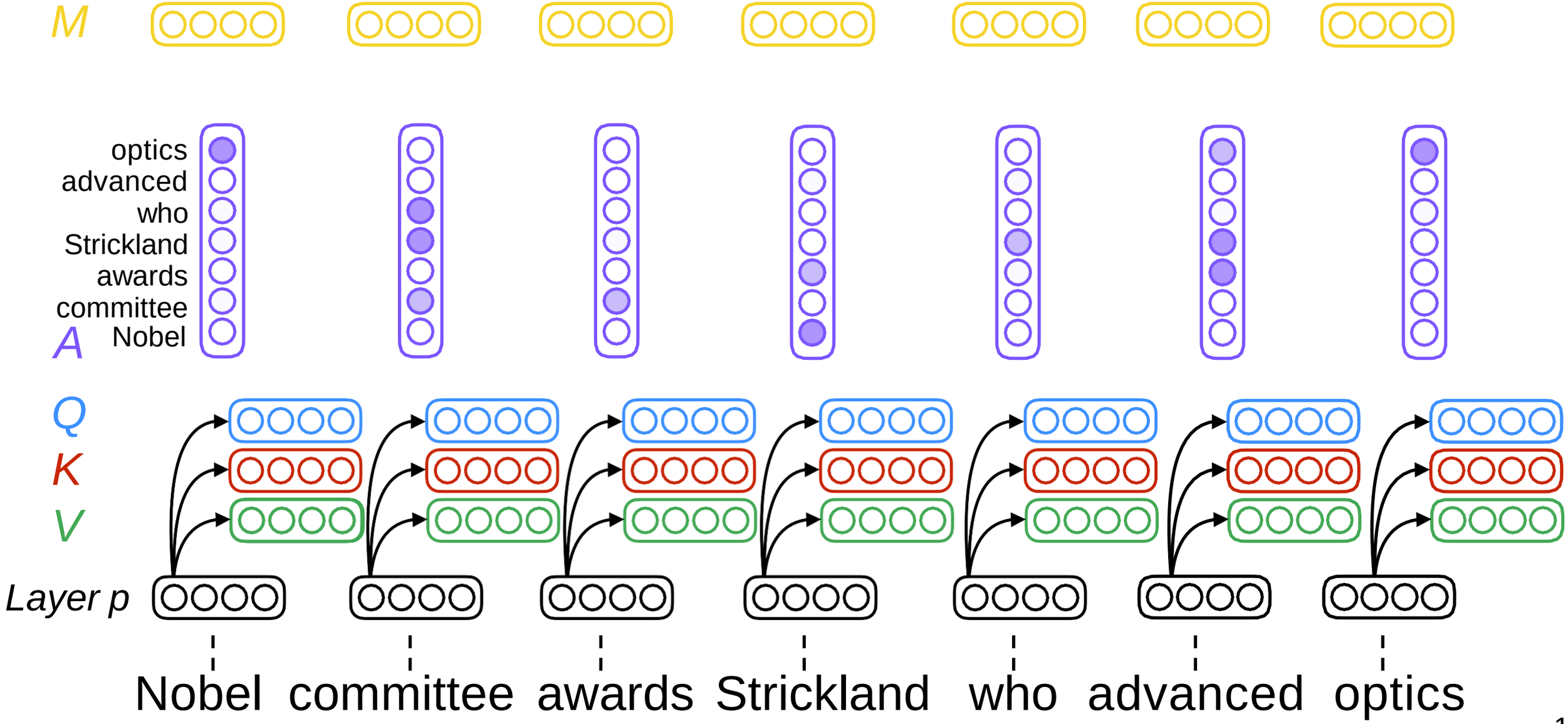
Self-attention (in encoder)



Slides by Emma Strubell!

[Vaswani et al. 2017]

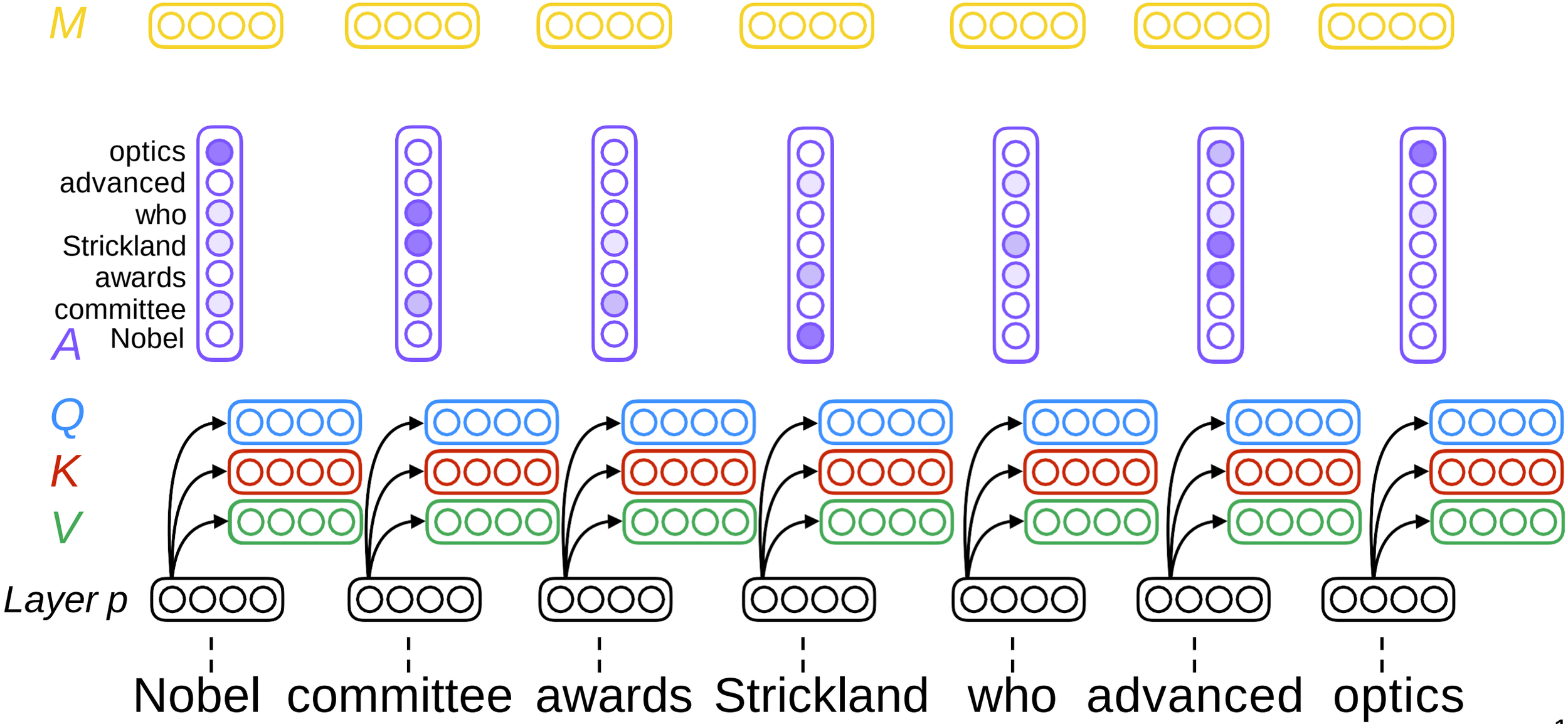
Self-attention (in encoder)



Slides by Emma Strubell!

[Vaswani et al. 2017]

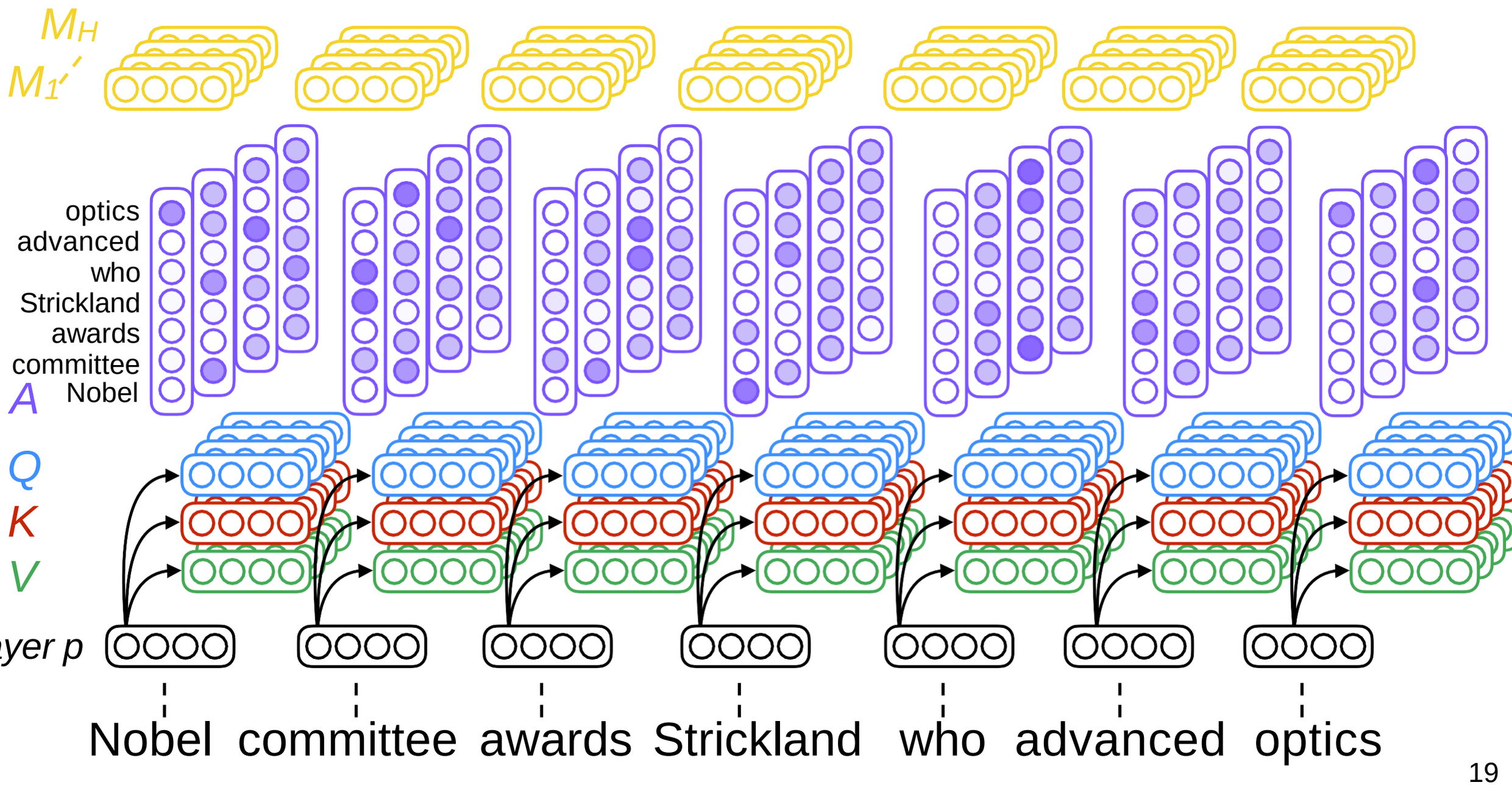
Self-attention (in encoder)



Slides by Emma Strubell!

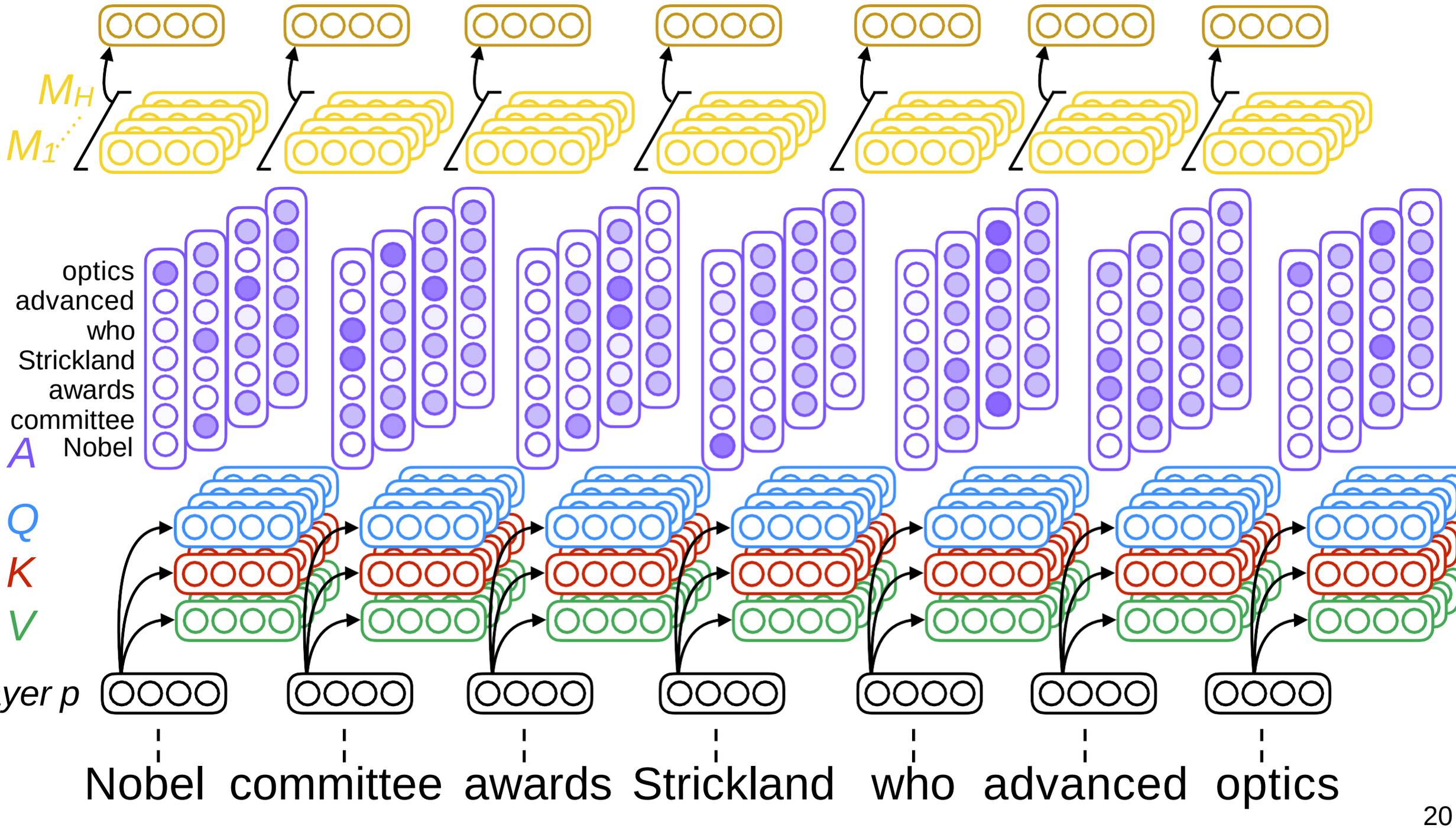
[Vaswani et al. 2017]

Multi-head self-attention



[Vaswani et al. 2017]

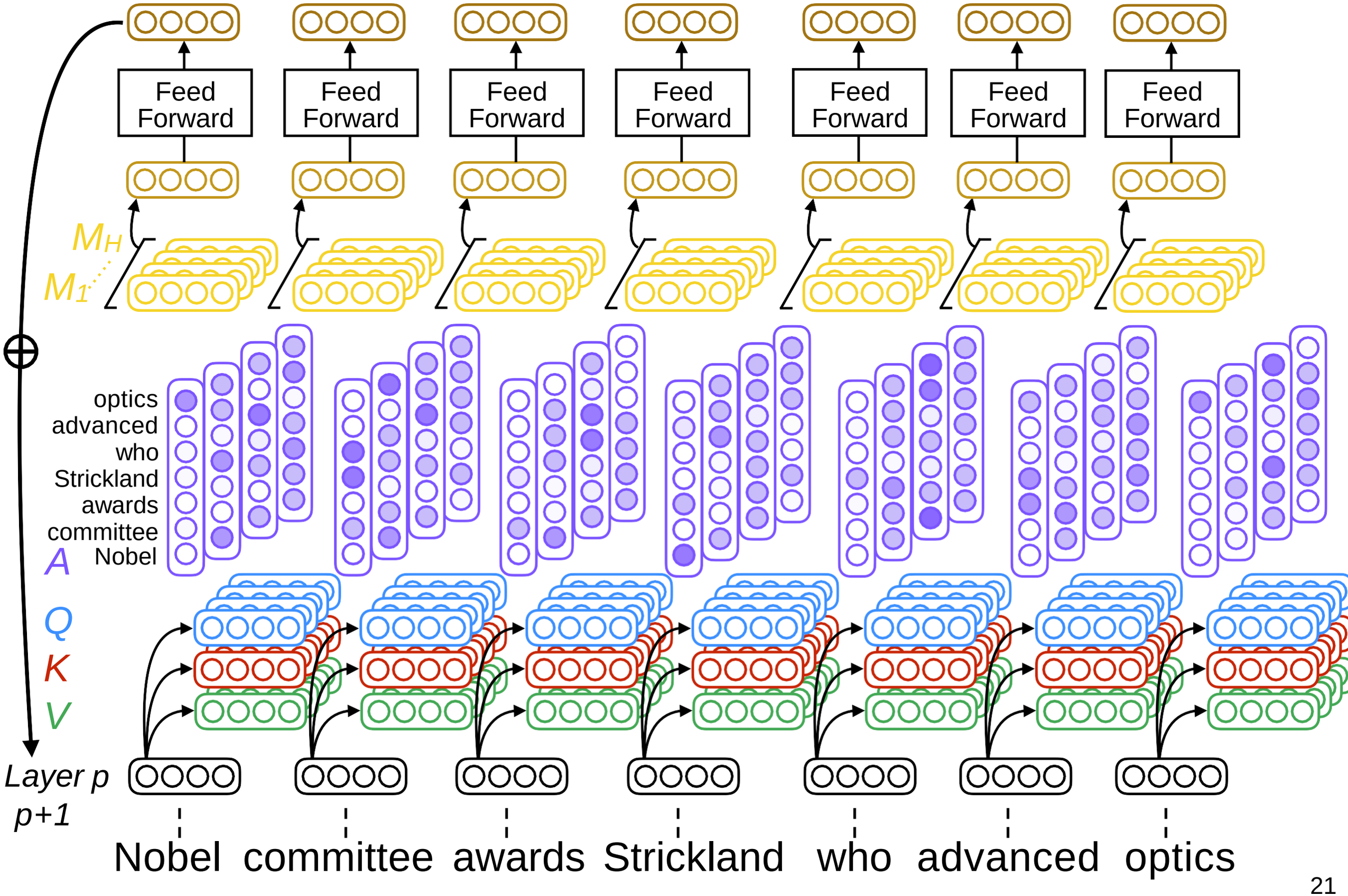
Multi-head self-attention



Slides by Emma Strubell!

[Vaswani et al. 2017]

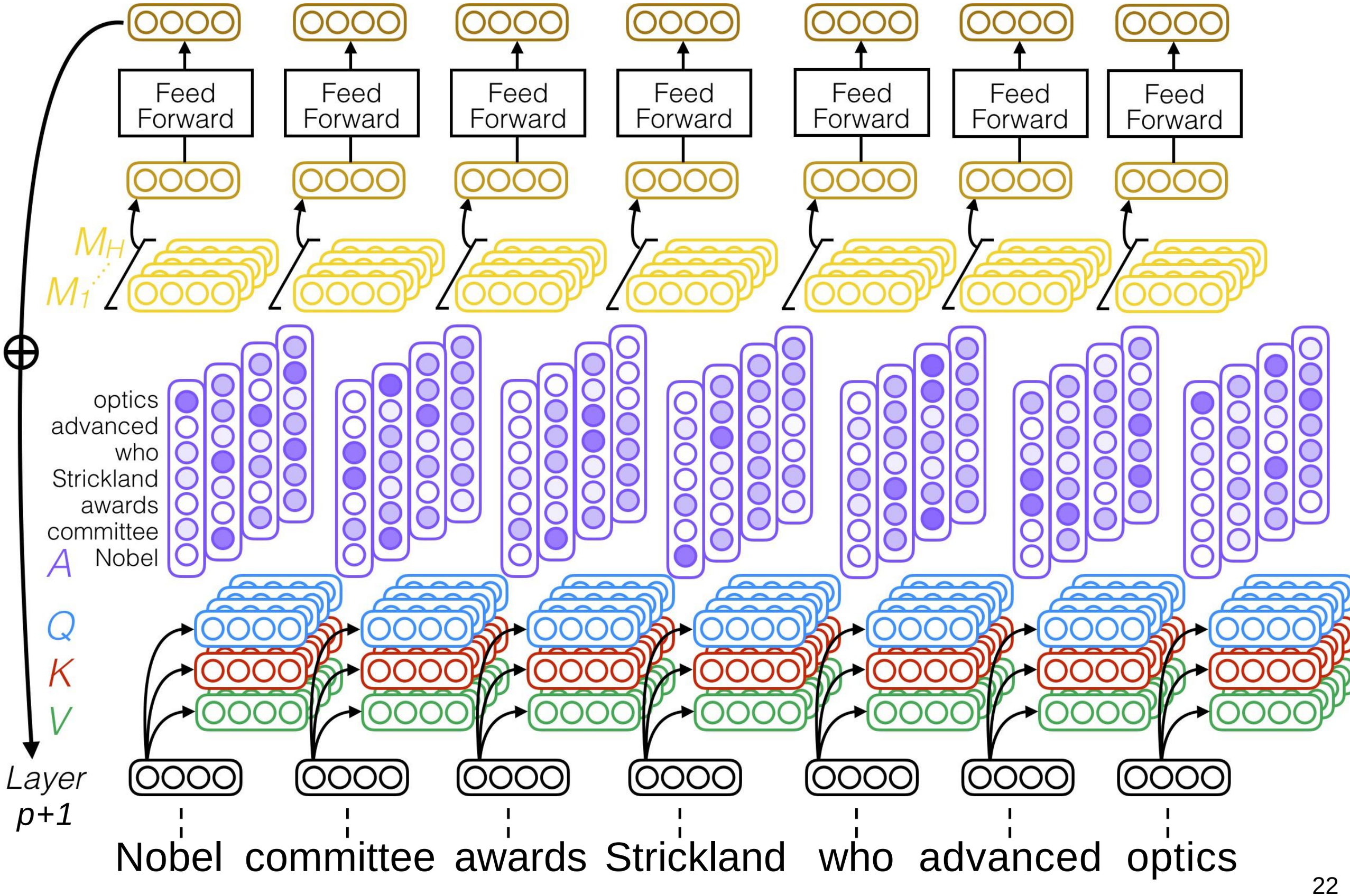
Multi-head self-attention



Slides by Emma Strubell!

[Vaswani et al. 2017]

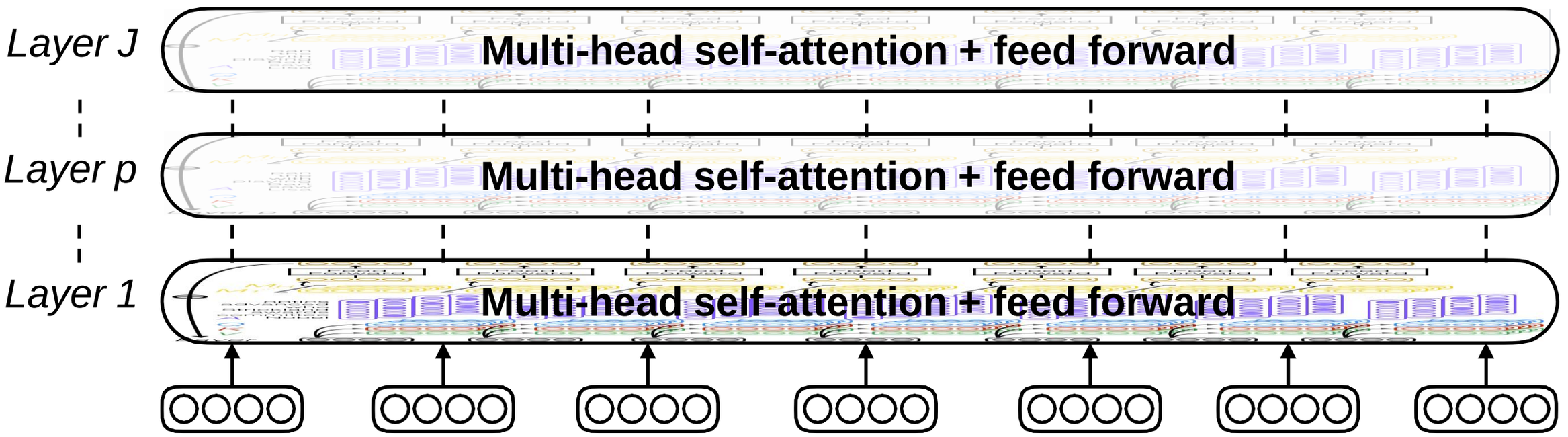
Multi-head self-attention



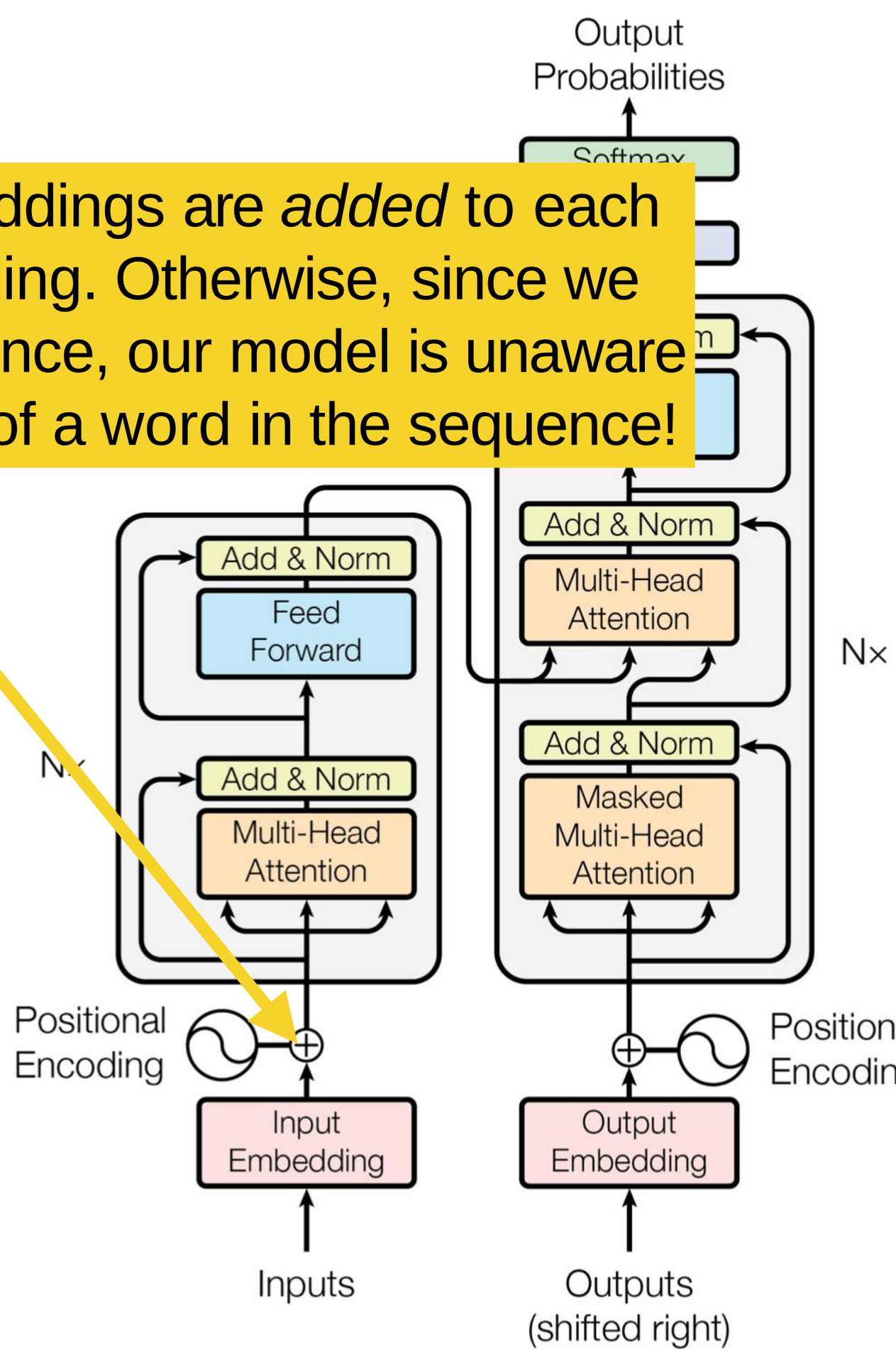
Slides by Emma Strubell!

[Vaswani et al. 2017]

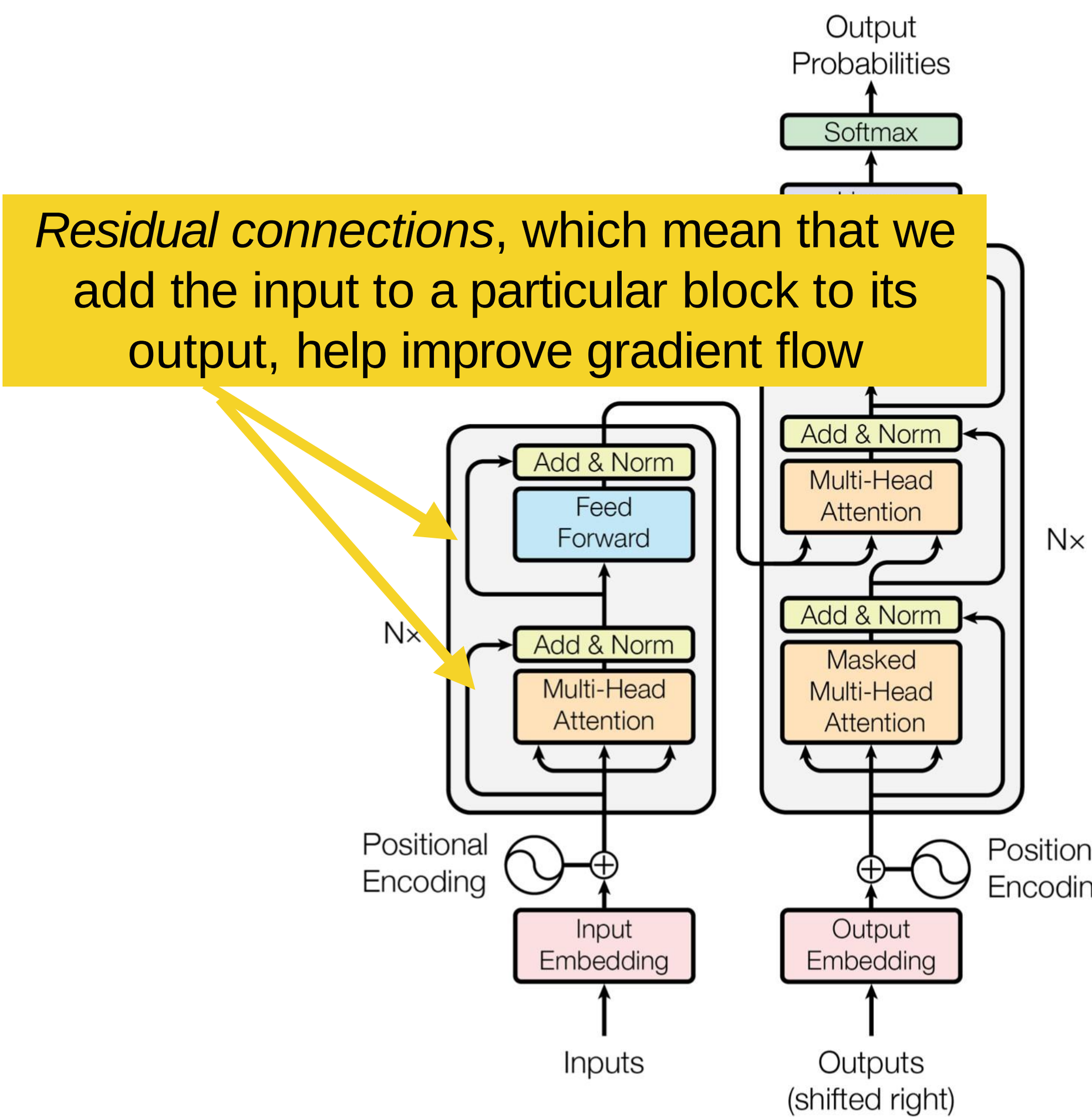
Multi-head self-attention



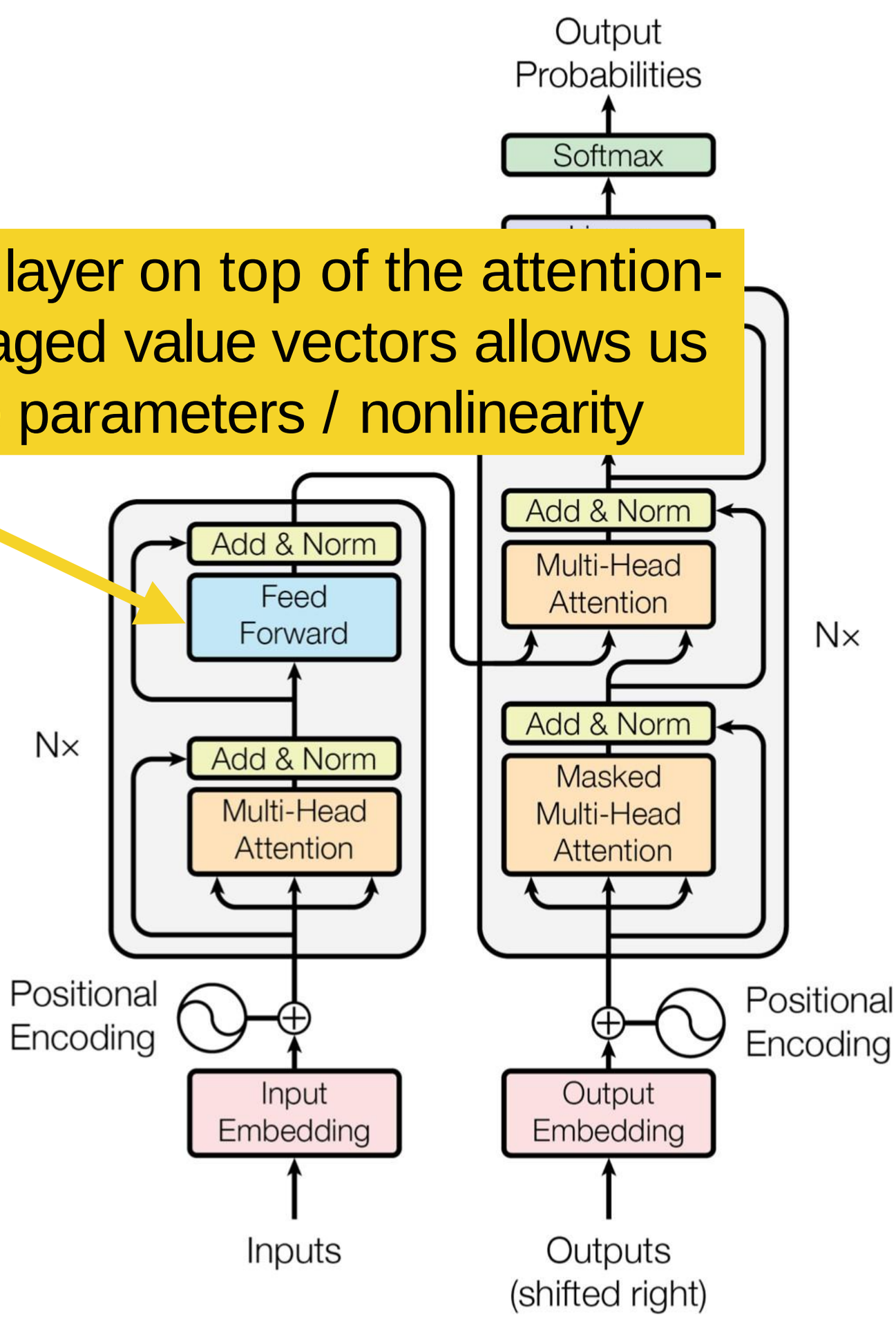
Position embeddings are *added* to each word embedding. Otherwise, since we have no recurrence, our model is unaware of the position of a word in the sequence!



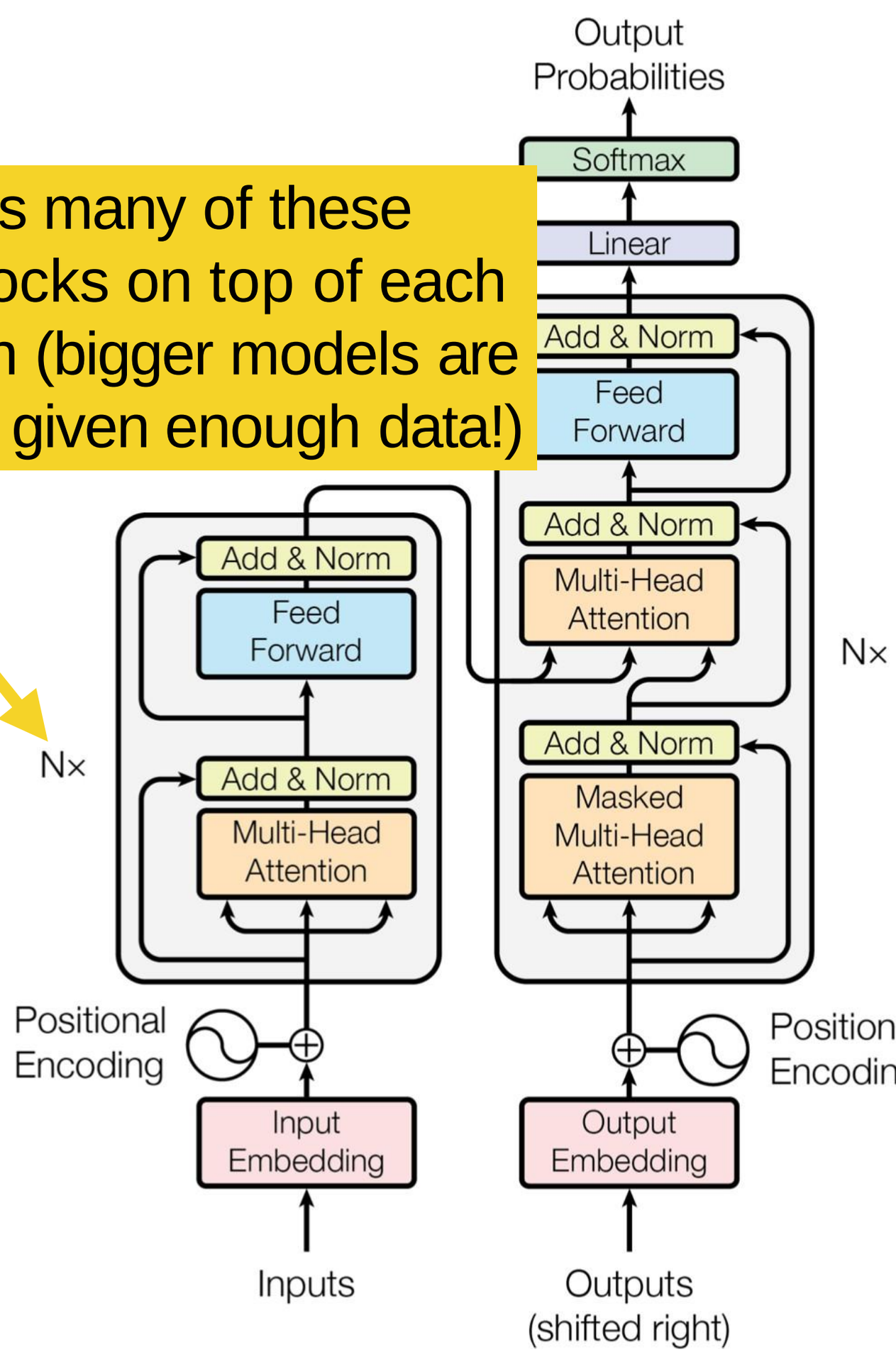
Residual connections, which mean that we add the input to a particular block to its output, help improve gradient flow



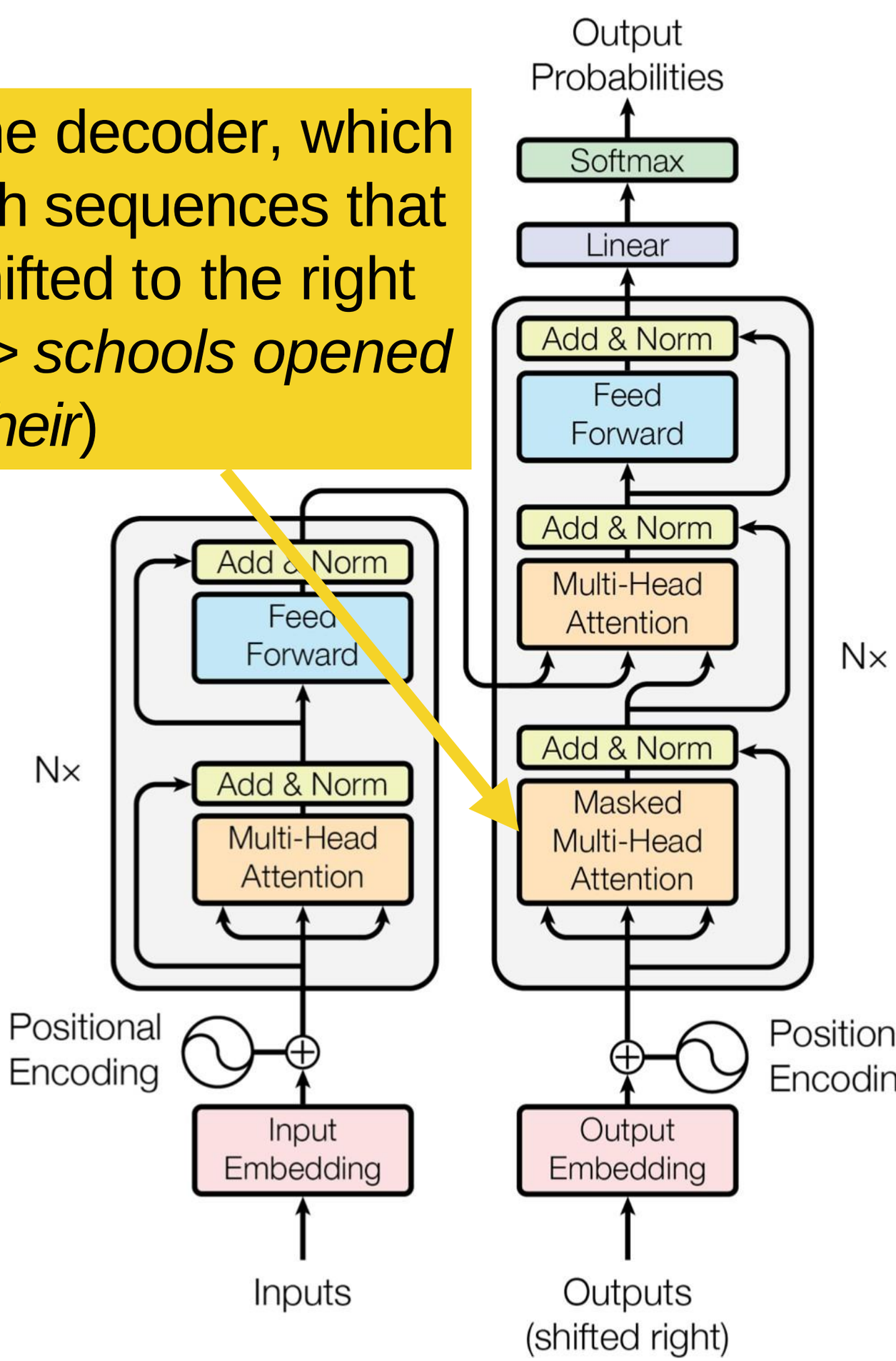
A feed-forward layer on top of the attention-weighted averaged value vectors allows us to add more parameters / nonlinearity



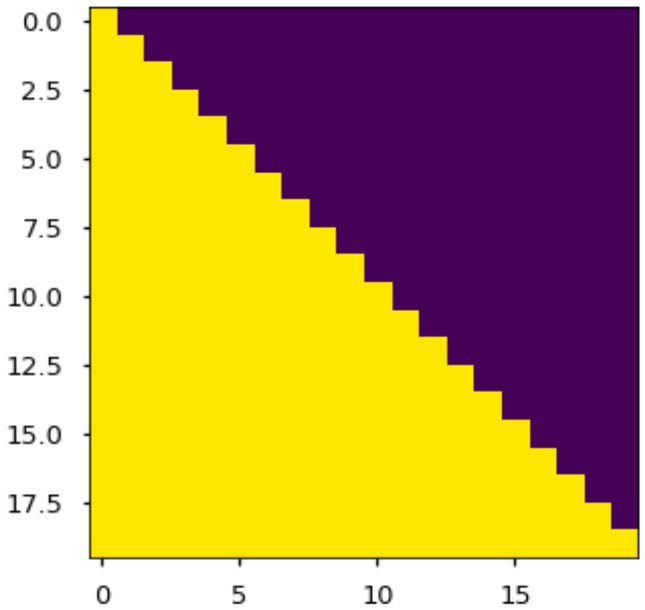
We stack as many of these *Transformer* blocks on top of each other as we can (bigger models are generally better given enough data!)

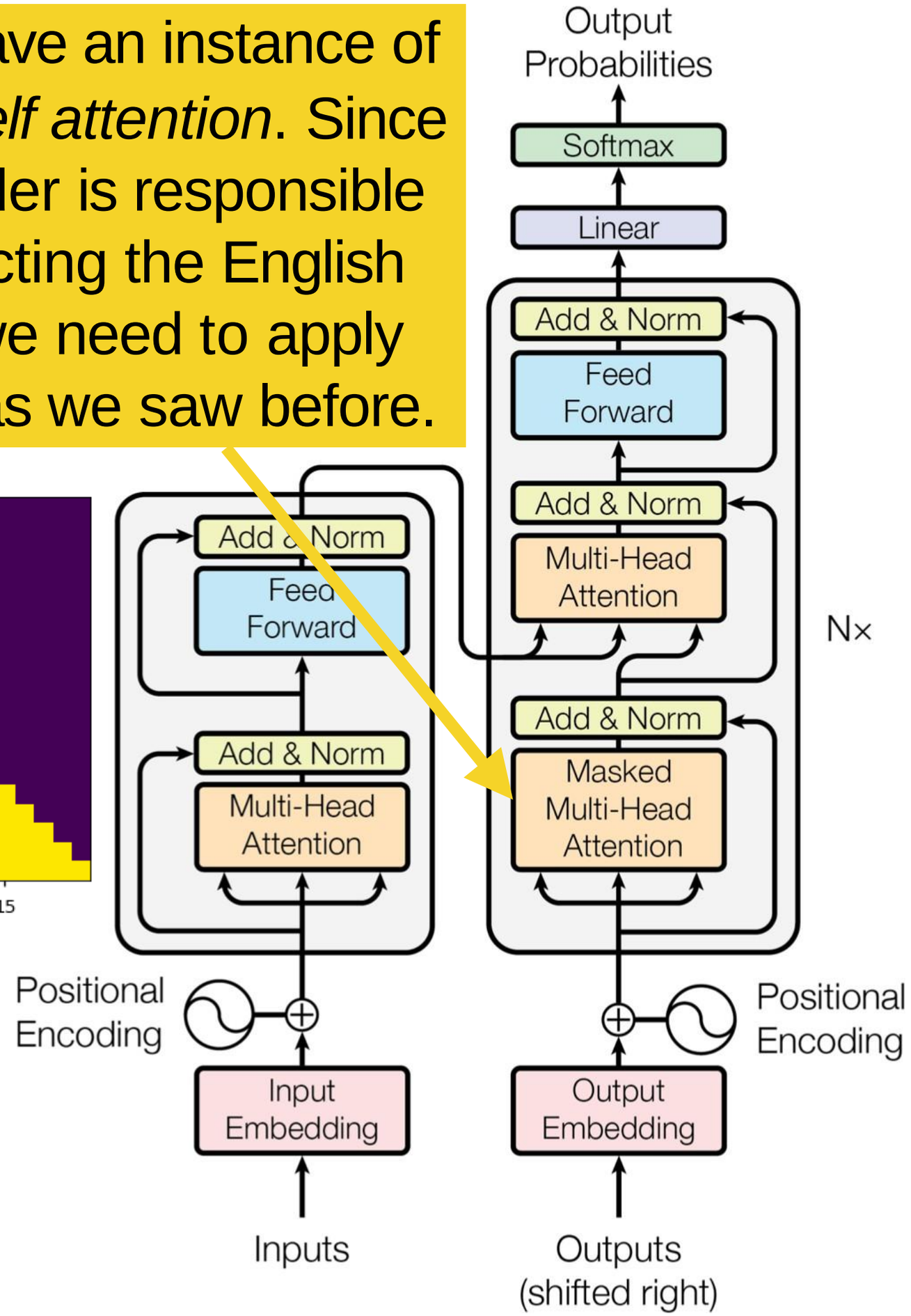


Moving onto the decoder, which takes in English sequences that have been shifted to the right (e.g., *<START> schools opened their*)



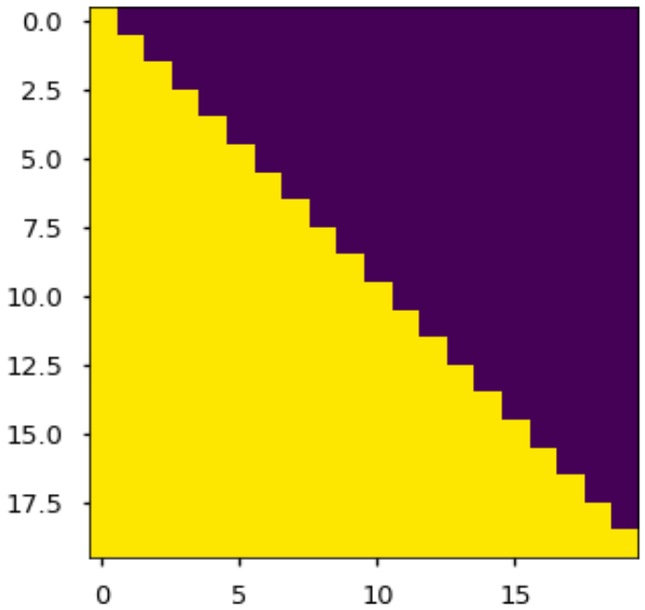
We first have an instance of *masked self attention*. Since the decoder is responsible for predicting the English words, we need to apply masking as we saw before.

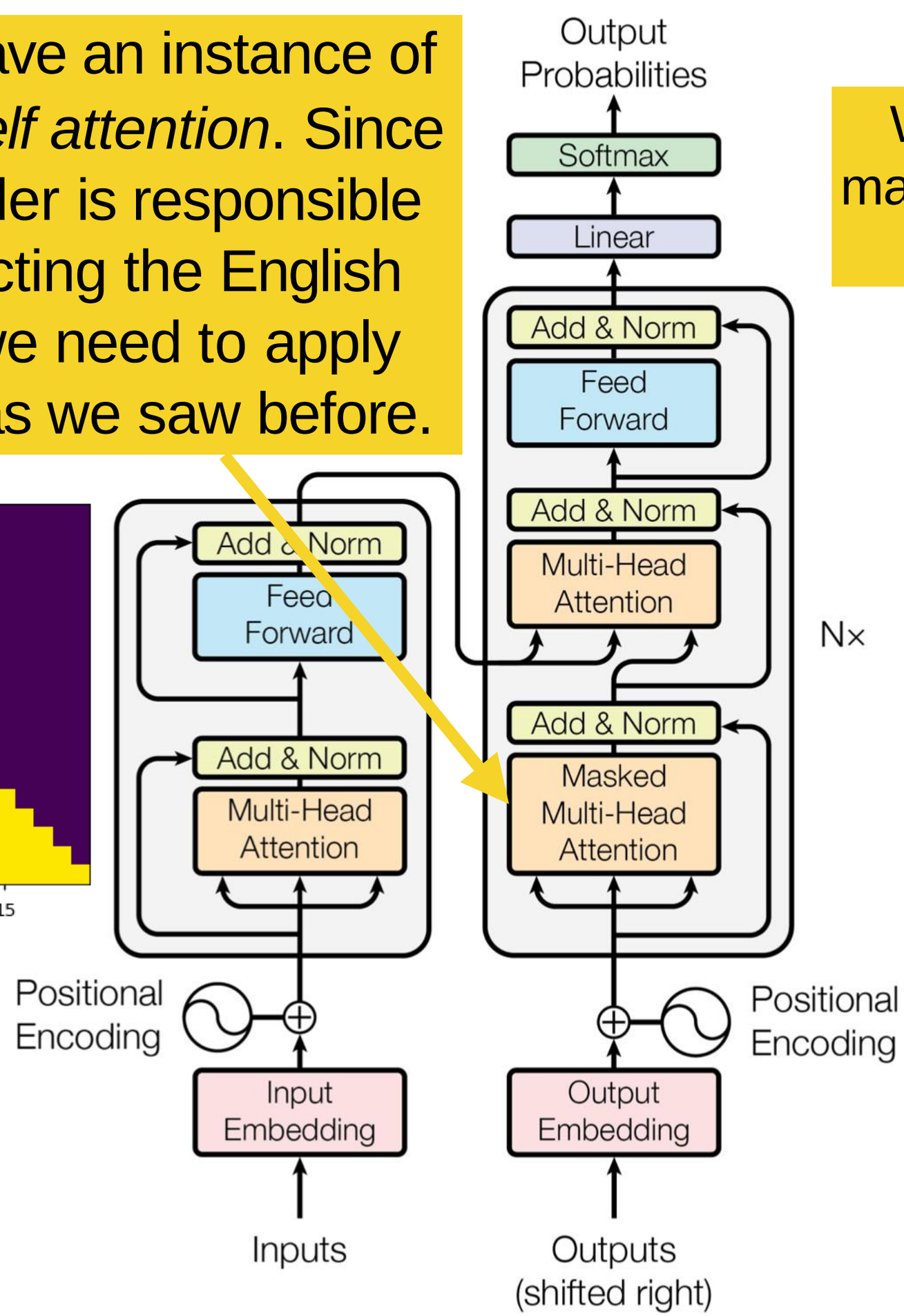




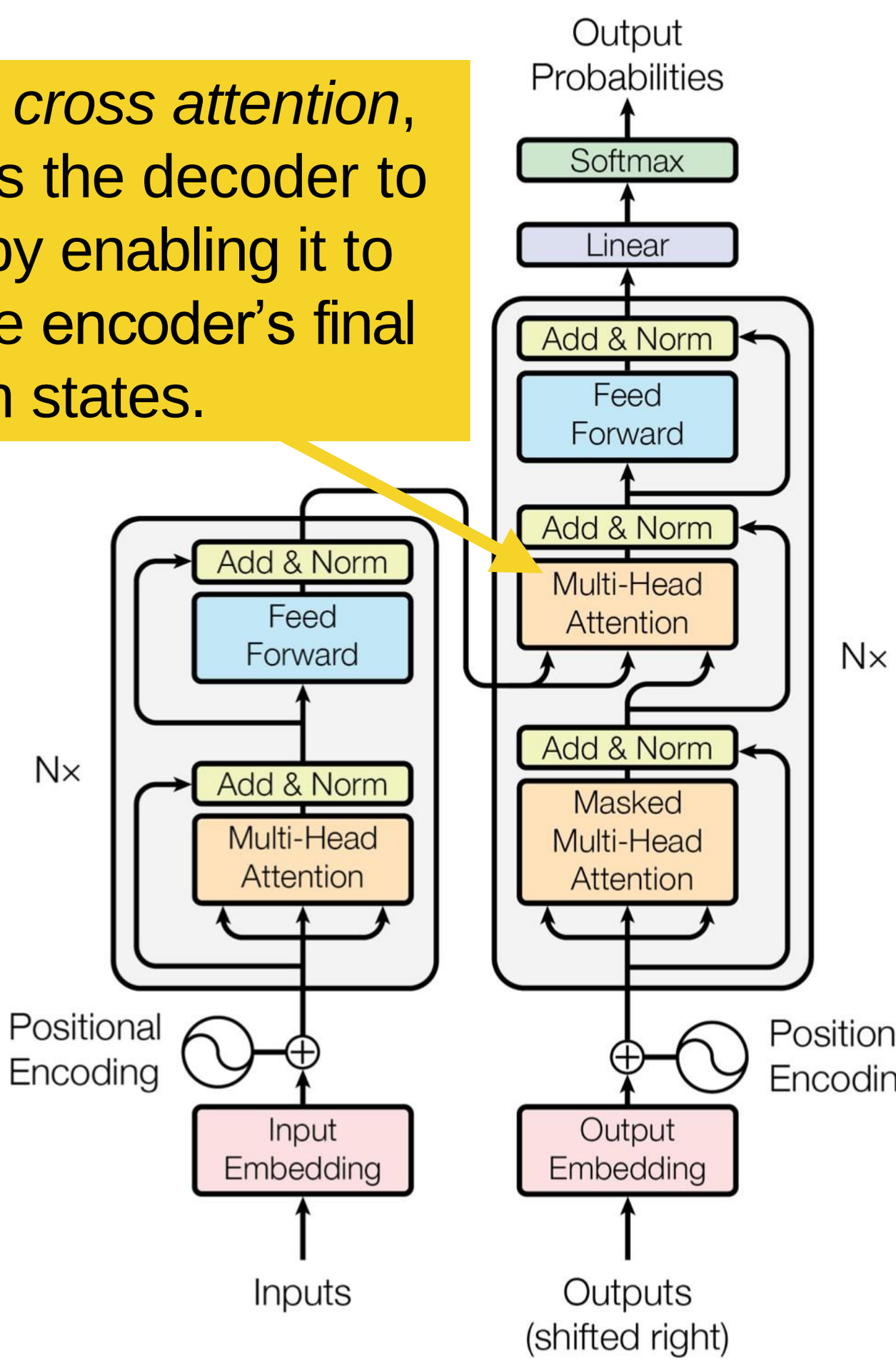
We first have an instance of *masked self attention*. Since the decoder is responsible for predicting the English words, we need to apply masking as we saw before.

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

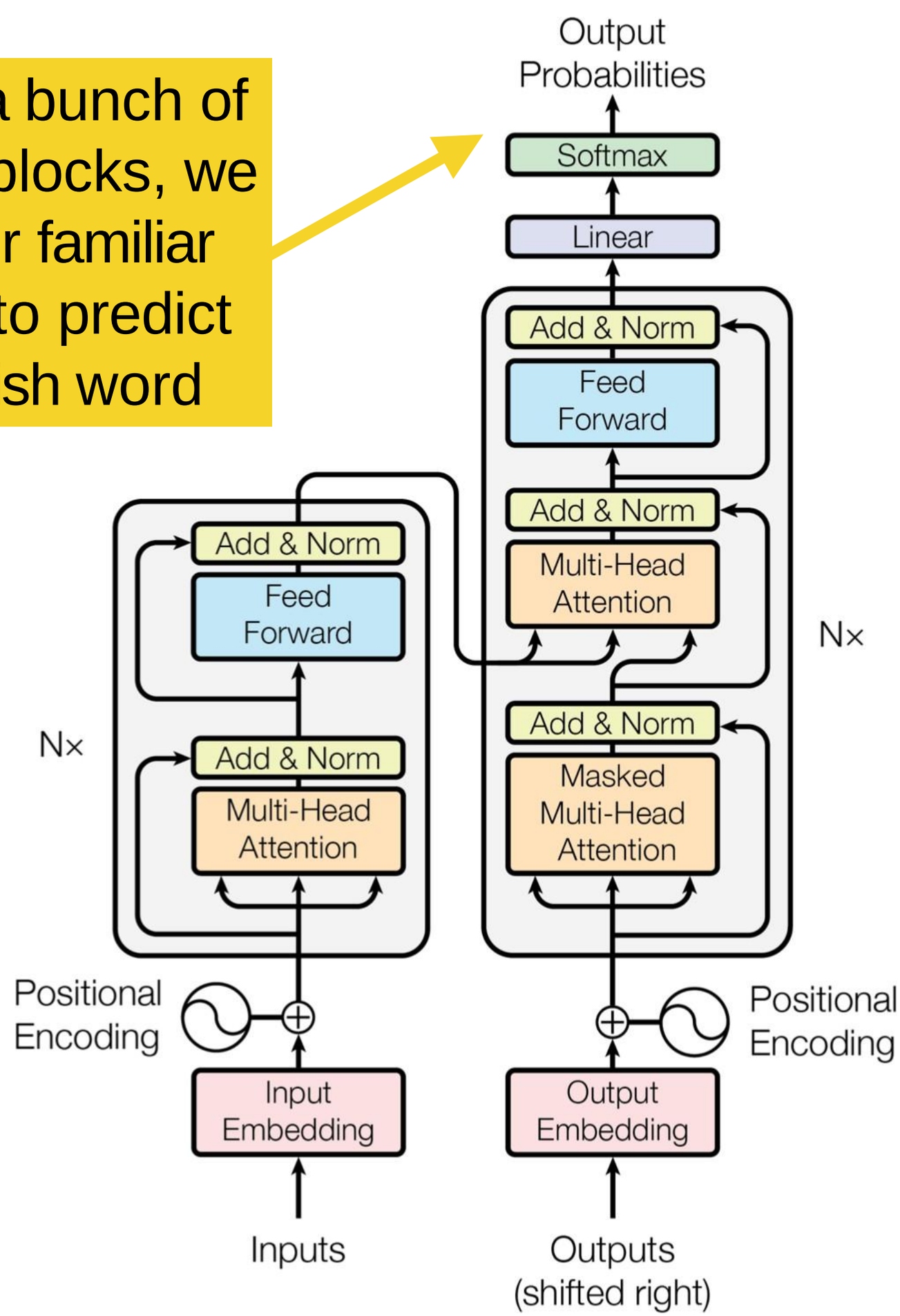




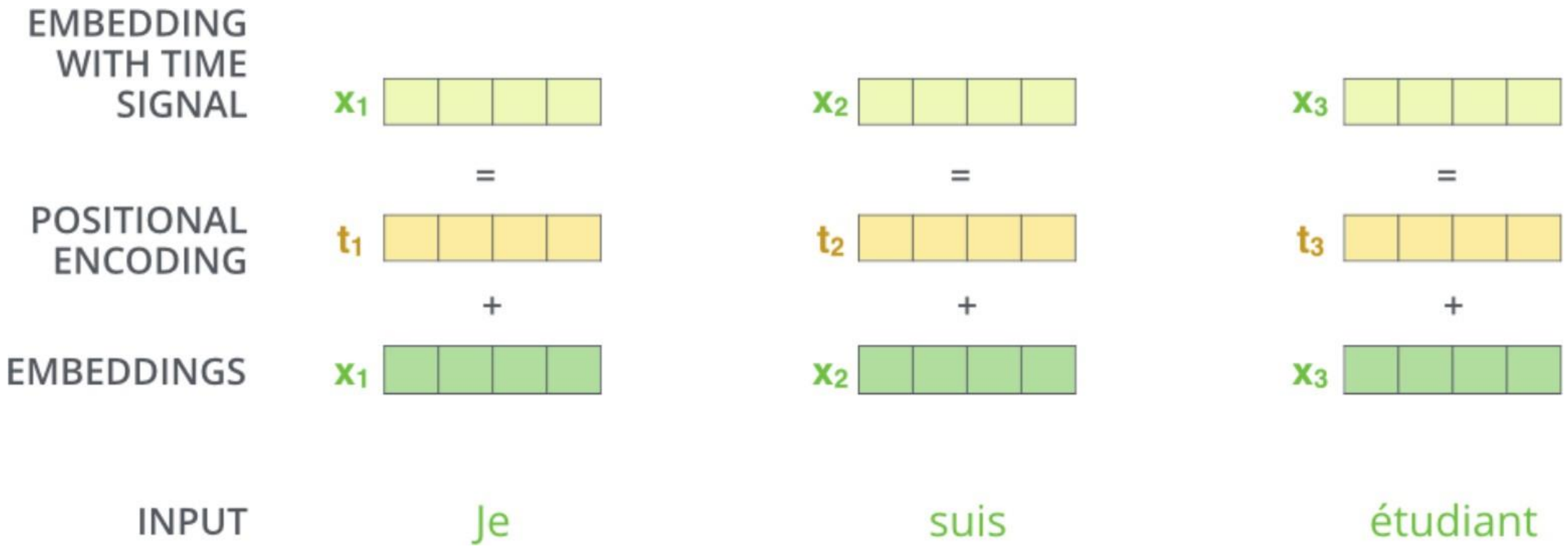
Now, we have *cross attention*, which connects the decoder to the encoder by enabling it to attend over the encoder's final hidden states.



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



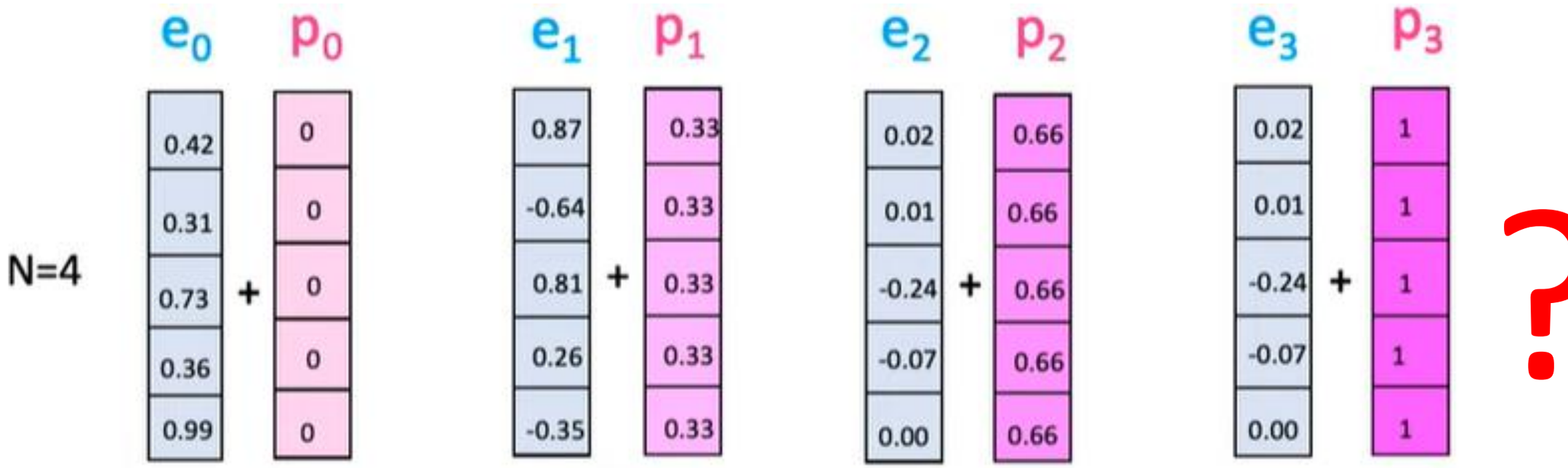
Positional encoding



Option 1


e0 p0
0.42 0 0.31 0 0.73 0 0.36 0 0.99 0
+
e1 p1
0.87 1 -0.64 1 0.81 1 0.26 1 -0.35 1
+
e2 p2
0.02 2 0.01 2 -0.24 2 -0.07 2 0.00 2
+
e3 p3
0.02 3 0.01 3 -0.24 3 -0.07 3 0.00 3
?
e0 p0
0.42 0 0.31 0 0.73 0 0.36 0 0.99 0
+
e1 p1
0.87 1 -0.64 1 0.81 1 0.26 1 -0.35 1
+
• • •
e30 p30
0.02 30 0.01 30 -0.24 30 -0.07 30 0.00 30
+

Option 2



$0 \times \frac{1}{3}$ $1 \times \frac{1}{3}$ $2 \times \frac{1}{3}$ $3 \times \frac{1}{3}$

$$\frac{1}{N-1} = \frac{1}{3}$$

Option 2

The positional embedding vector at a given position should remain the same irrespective of the length of the sequence

Sentence 1

N=4

e_0	p_0	e_1	p_1	e_2	p_2	e_3	p_3
0.42	0	0.87	0.33	0.02	0.66	0.02	1
0.31	0	-0.64	0.33	0.01	0.66	0.01	1
0.73	0	0.81	0.33	-0.24	0.66	-0.24	1
0.36	0	0.26	0.33	-0.07	0.66	-0.07	1
0.99	0	-0.35	0.33	0.00	0.66	0.00	1

Sentence 2

N=5

e_0	p_0	e_1	p_1	e_2	p_2	...	e_4	p_4
0.42	0	0.87	0.20	0.02	0.40		0.02	1
0.31	0	-0.64	0.20	0.01	0.40		0.01	1
0.73	0	0.81	0.20	-0.24	0.40		-0.24	1
0.36	0	0.26	0.20	-0.07	0.40		-0.07	1
0.99	0	-0.35	0.20	0.00	0.40		0.00	1

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

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

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

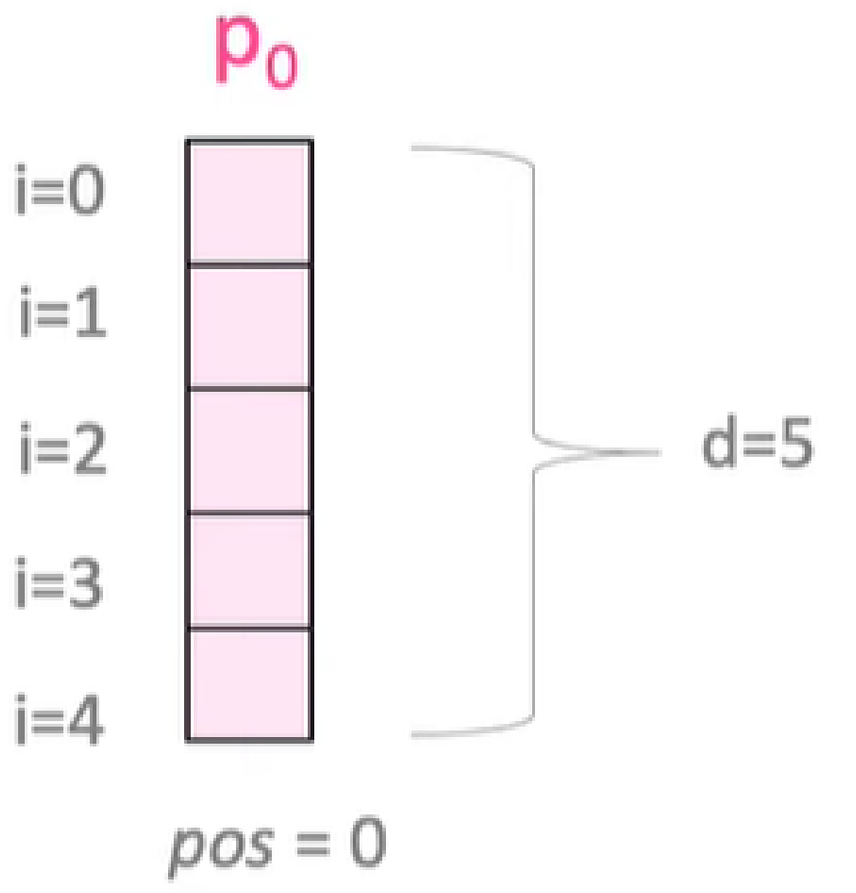
Positional encoding is a 512d vector

i = a particular dimension of this vector

pos = dimension of the word

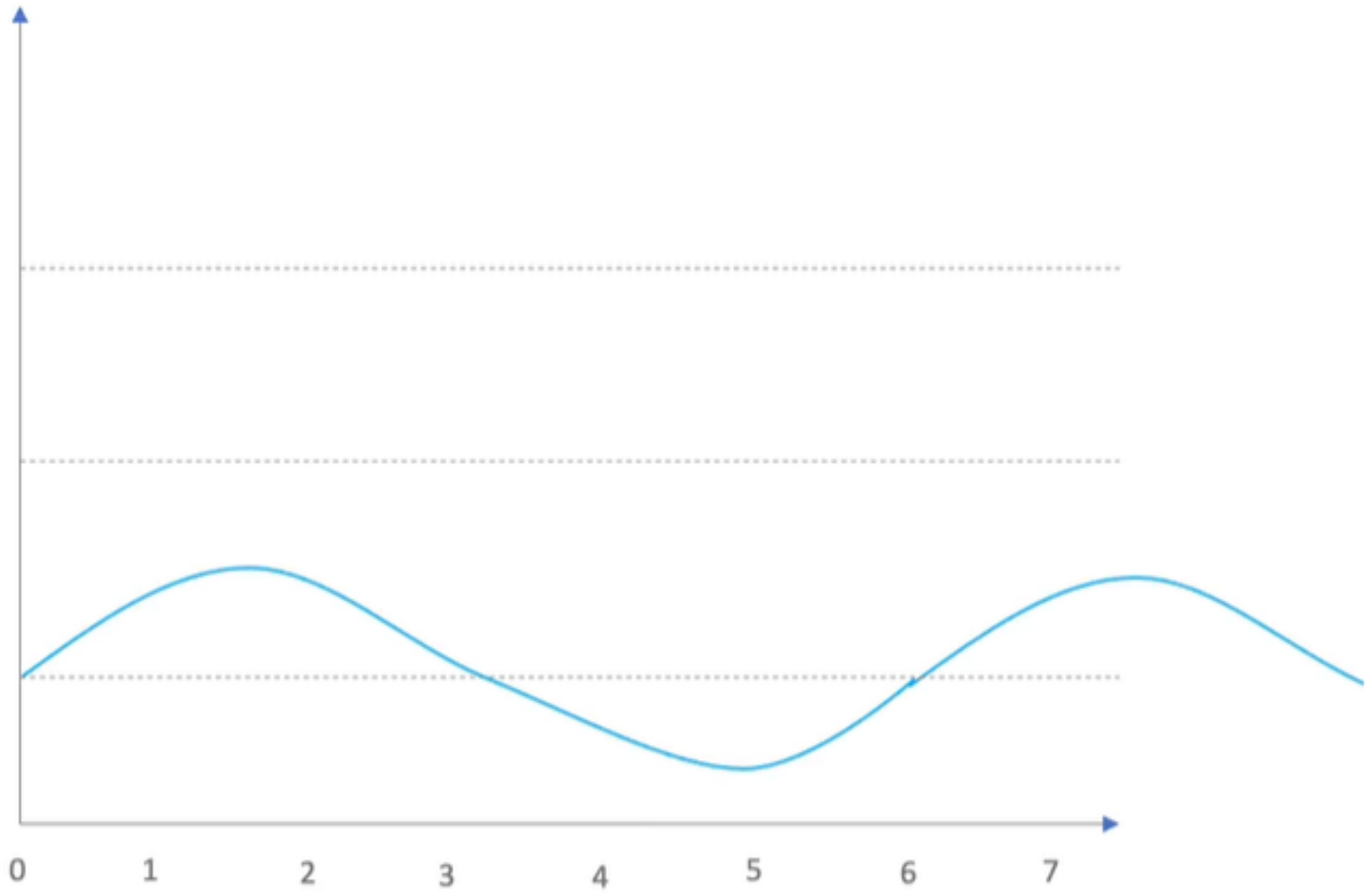
d_model = 512

Transformer positional encoding



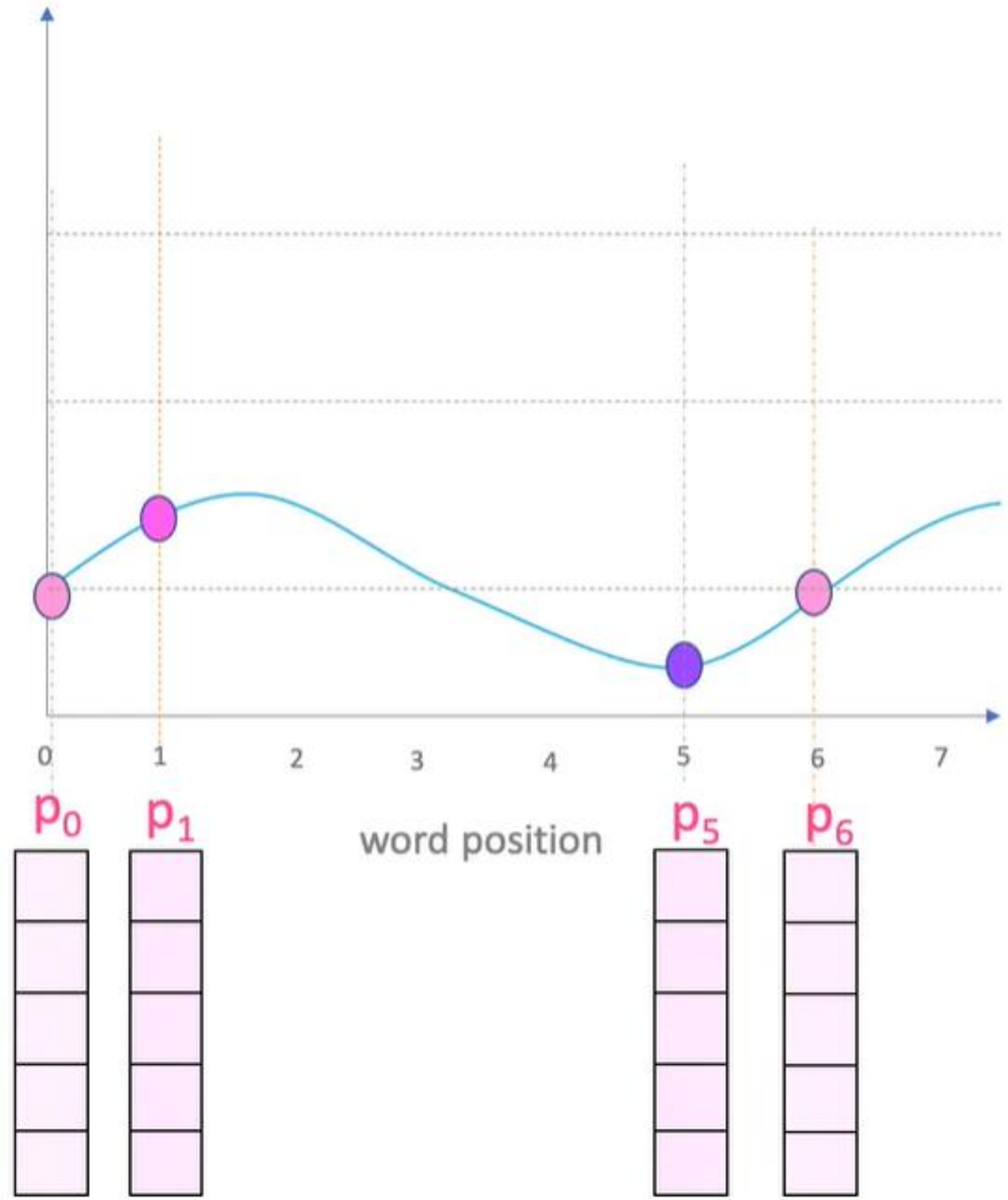
$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

Only varying the position



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

Only varying the position



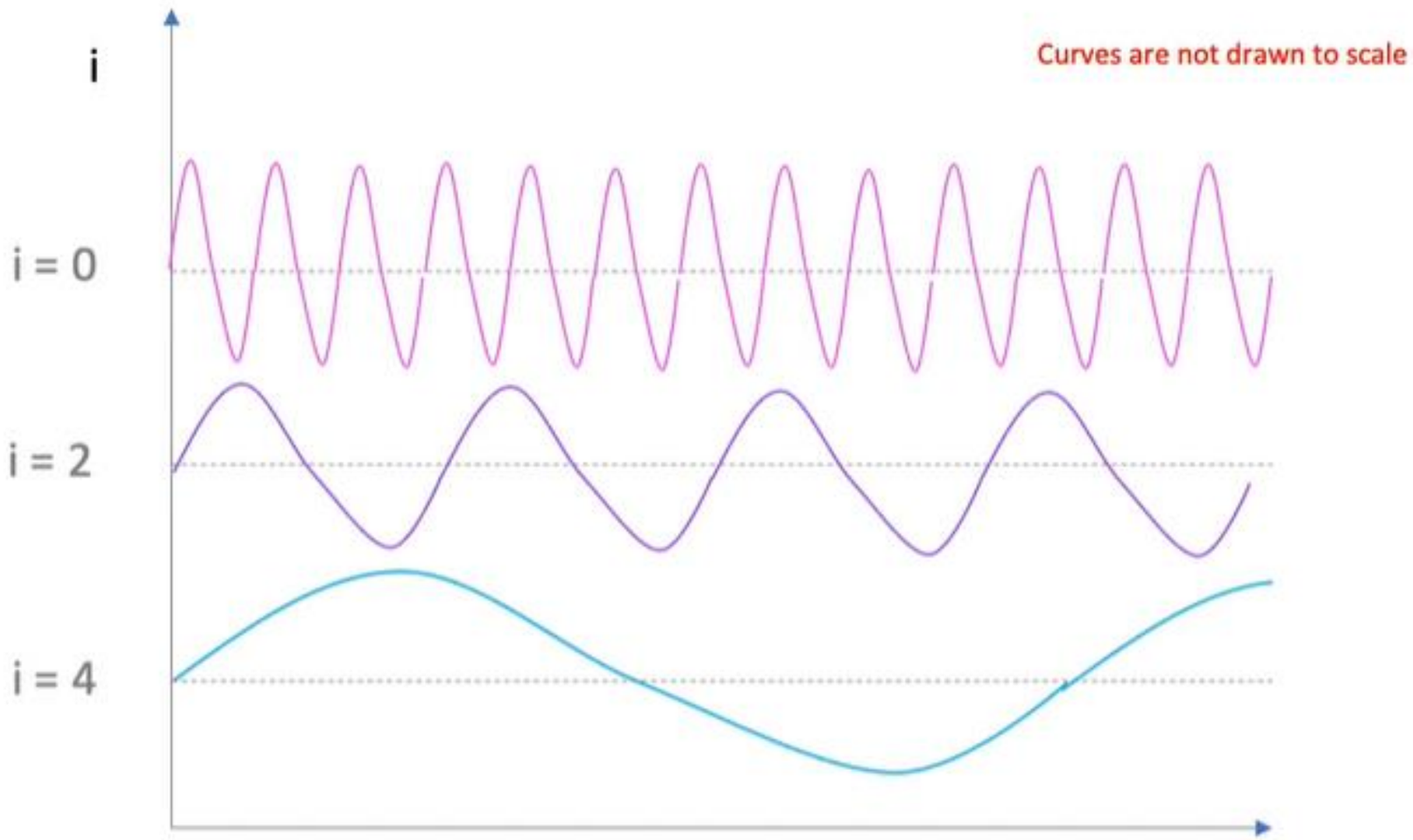
Pros

- It is independent of the length of the sequence

Cons

- P0 and P6 have same positional embeddings

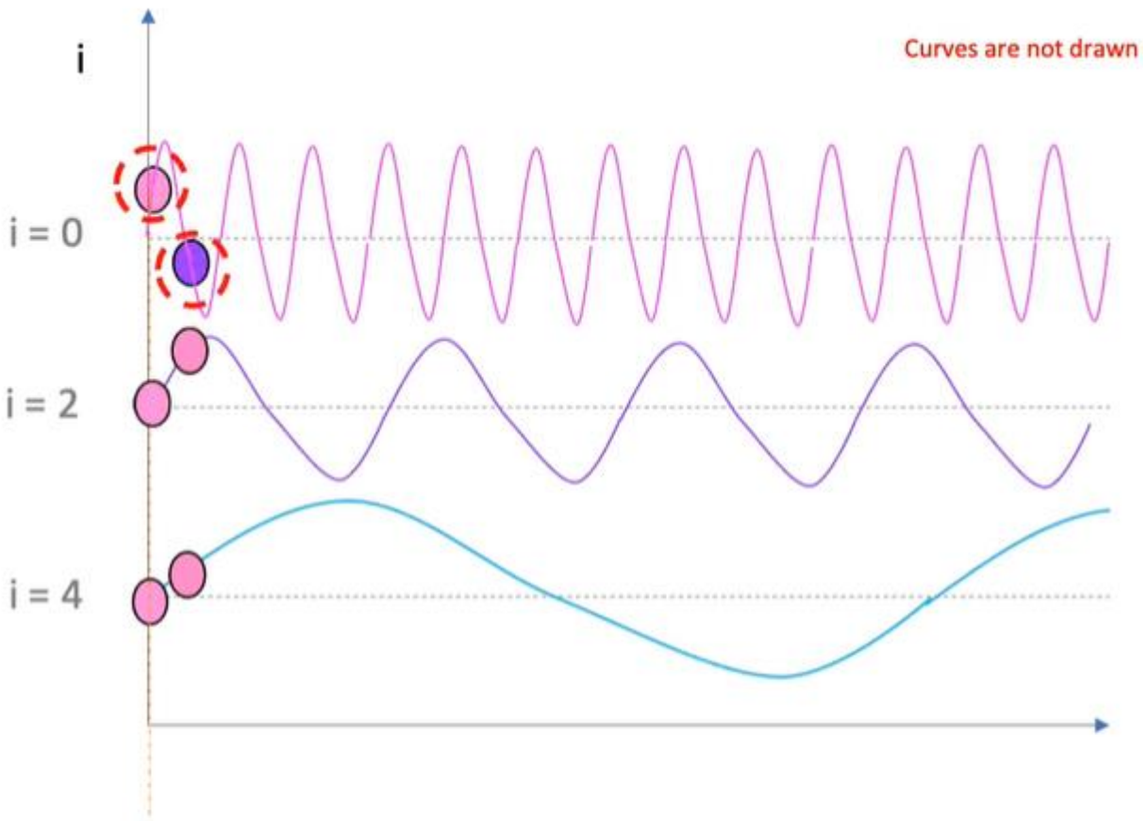
Varying both position and *i*



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

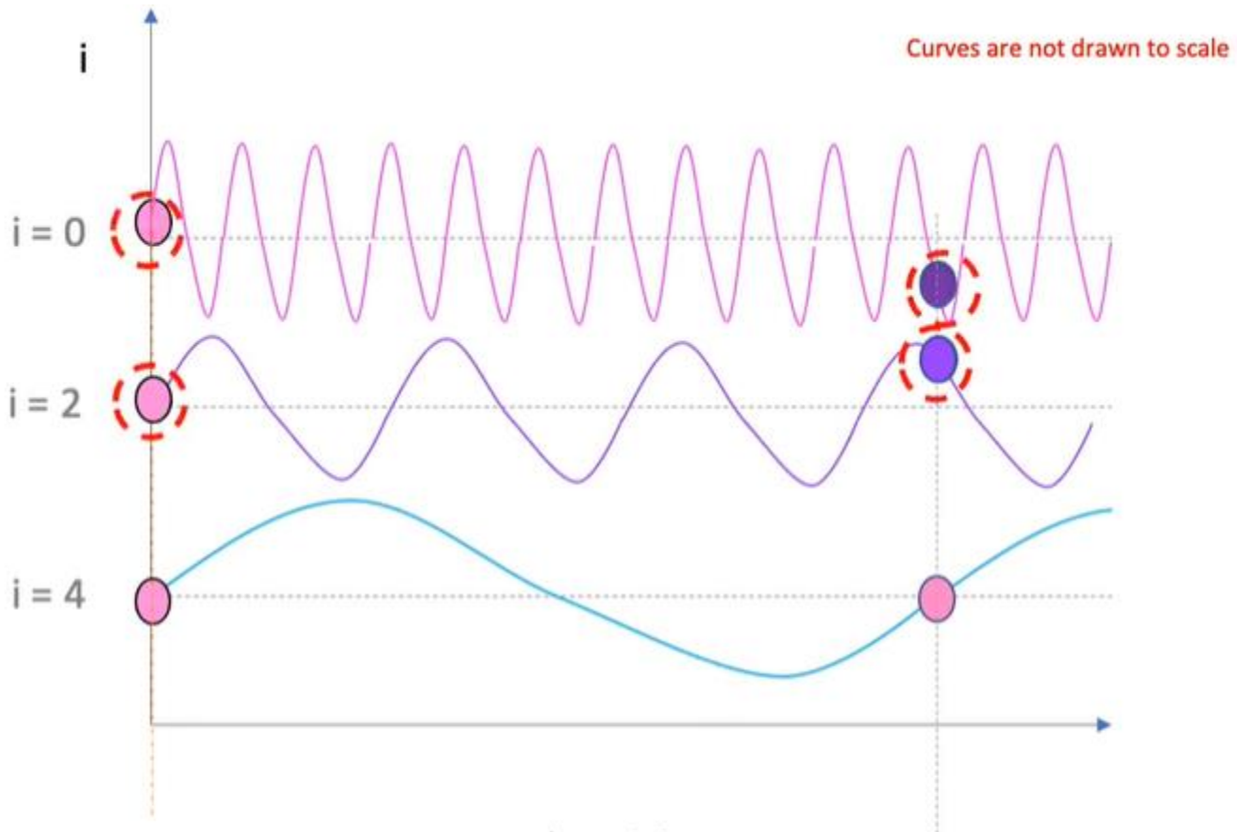
i

Varying both position and *i*



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

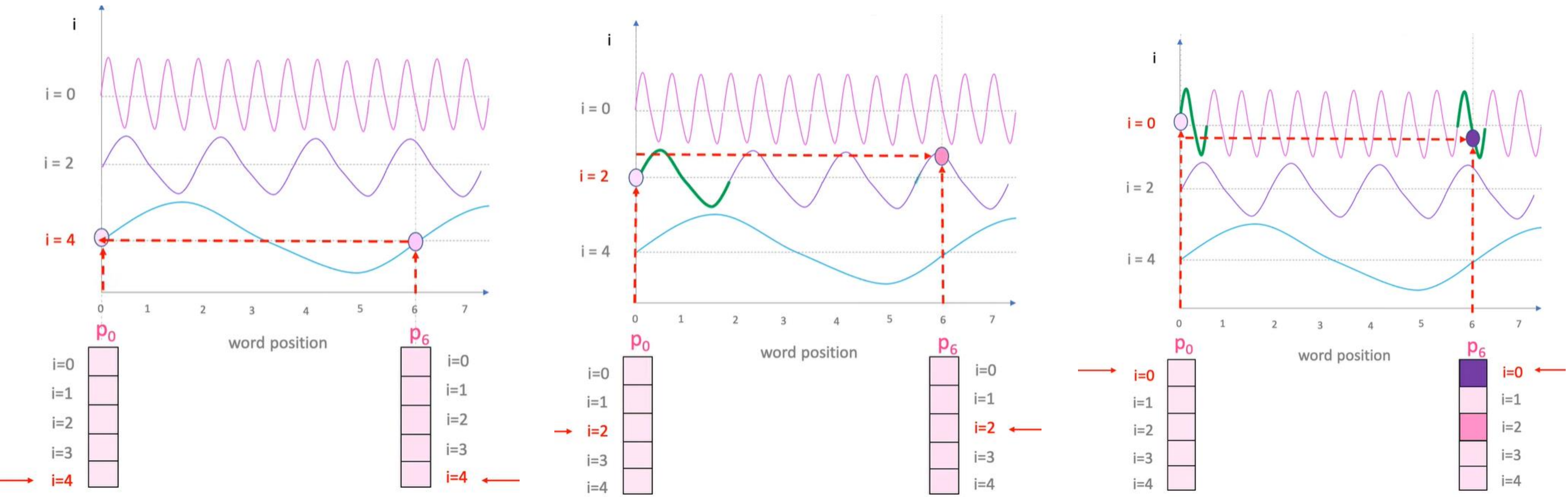
If two points are closeby on the curve, they will remain identical at high frequencies too. It is at the high frequency where their y-axis values differ and we may be able to take them apart.



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

For points which are far apart, we will see them falling apart on the y-axis quite early on

Varying both position and *i:* Example



At lower frequency, the value is exactly the same

This will start to differ significantly as we move to high frequencies

Example

For example, for word w at position $pos \in [0, L-1]$ in the input sequence $\boldsymbol{w} = (w_0, \cdots, w_{L-1})$, with 4-dimensional embedding e_w, and $d_{model} = 4$, the operation would be

$$
\begin{aligned}
e'_w &= e_w + \left[sin\left(\frac{pos}{10000^0}\right), cos\left(\frac{pos}{10000^0}\right), sin\left(\frac{pos}{10000^{2/4}}\right), cos\left(\frac{pos}{10000^{2/4}}\right) \right] \\
&= e_w + \left[sin\left(pos\right), cos\left(pos\right), sin\left(\frac{pos}{100}\right), cos\left(\frac{pos}{100}\right) \right]
\end{aligned}
$$

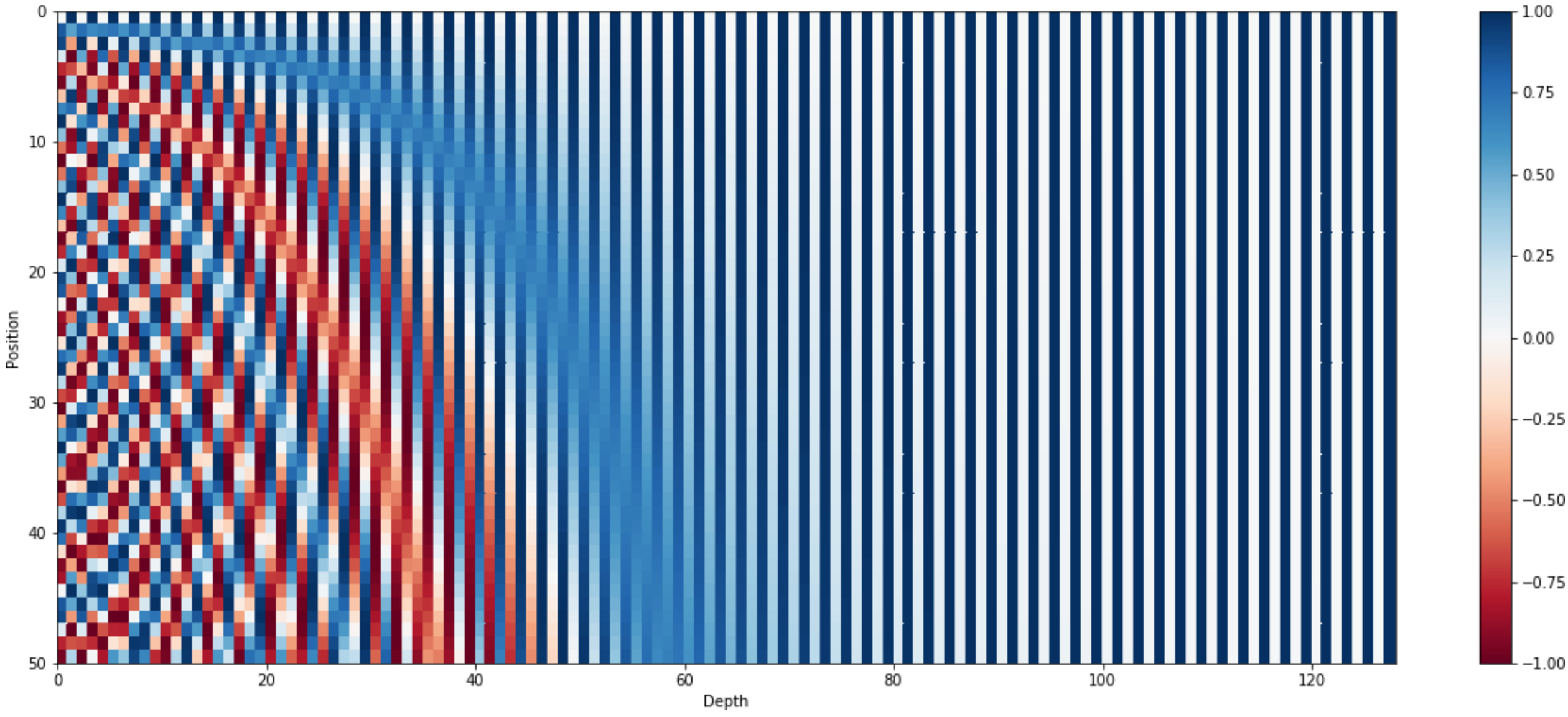
where the formula for positional encoding is as follows

$$
\begin{aligned}
\mathrm{PE}(pos, 2i) &= sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \\
\mathrm{PE}(pos, 2i+1) &= cos\left(\frac{pos}{10000^{2i/d_{model}}}\right).
\end{aligned}
$$

https://datascience.stackexchange.com/questions/51065/what-is-the-positional-encoding-in-the-transformer-model

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)

Batch and layer normalization

https://www.pinecone.io/learn/batch-layer-normalization/

Normalization

Example: student loans with the age of the student and the tuition as two input features

- two values are on totally *different* scales.
  - the age of a student will have a median value in the range 18 to 25 years
  - the tuition could take on values in the range $20K - $50K for a given academic year.

Normalization works by mapping all values of a feature to be in the range [0,1] using the transformation

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Suppose a particular input feature `x` has values in the range `[x_min, x_max]`. When `x` is equal to `x_min`, `x_norm` is equal to 0 and when `x` is equal to `x_max`, `x_norm` is equal to 1. So for all values of `x` between `x_min` and `x_max`, `x_norm` maps to a value between 0 and 1.

Standardization

Example: student loans with the age of the student and the tuition as two input features

- two values are on totally *different* scales.
  - the age of a student will have a median value in the range 18 to 25 years
  - the tuition could take on values in the range $20K - $50K for a given academic year.

Standardization transforms the input values such that they follow a distribution with zero mean and unit variance.

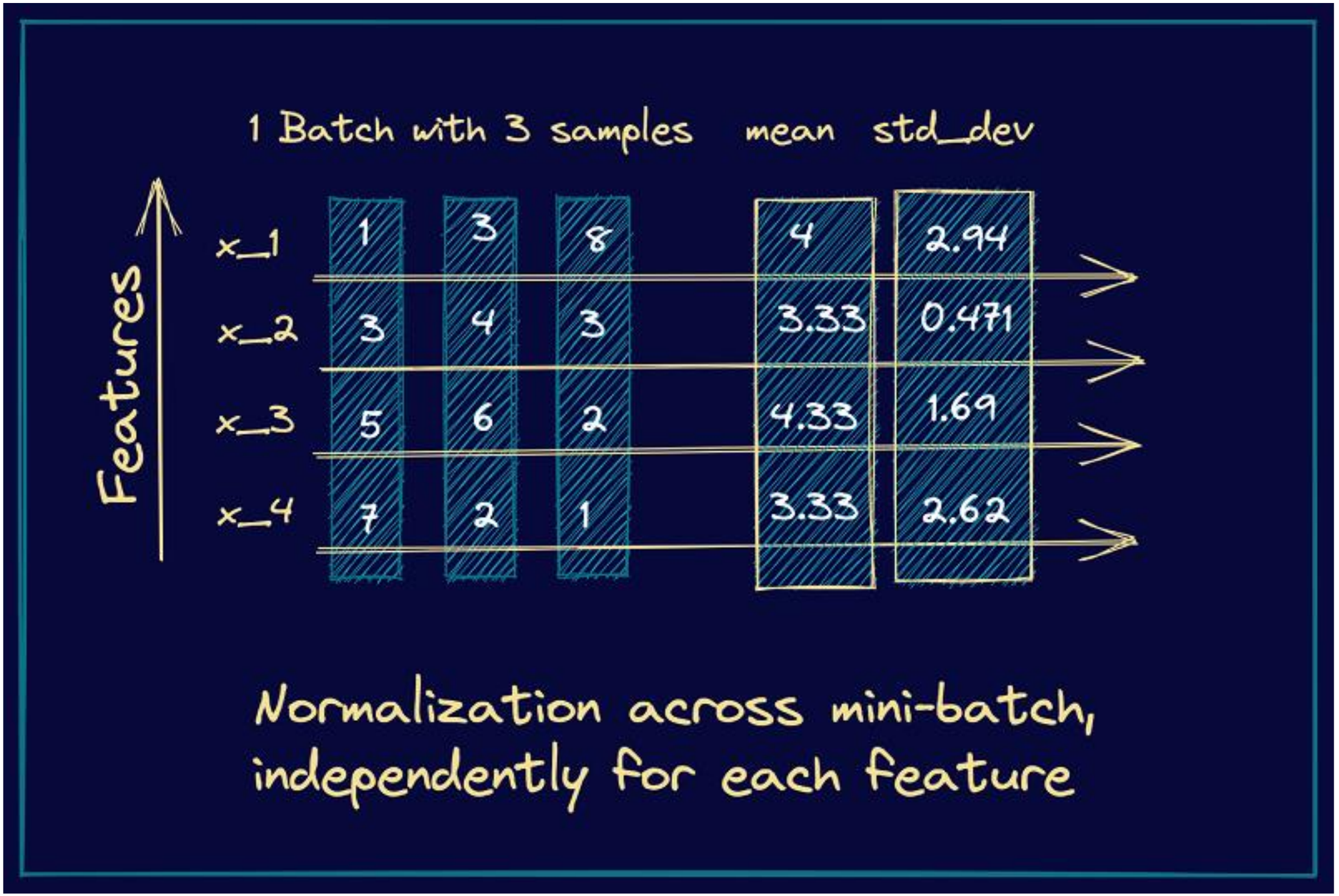
$$x_{std} = \frac{x - \mu}{\sigma}$$

In practice, this process of *standardization* is also referred to as *normalization*

Batch normalization

- For a network with hidden layers, the output of layer k-1 serves as the input to layer k
- Split the dataset into multiple batches and run the mini-batch gradient descent.
- The mini-batch gradient descent algorithm optimizes the parameters of the neural network by batch-wise processing of the dataset, one batch at a time.
- The input distribution at a particular layer keeps changing across batches.
- ***Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*** refers to this change in distribution of the input to a particular layer across batches as internal covariate shift.
- For instance, if the distribution of data at the input of layer *K* keeps changing across batches, the network will take longer to train.

Batch normalization



Batch normalization

- Forcing all the pre-activations to be zero and unit standard deviation across all batches can be too restrictive.
- It may be the case that the fluctuant distributions are necessary for the network to learn certain classes better.

$$\mu_b = \frac{1}{B}\sum_{i=1}^{B} x_i \ (1)$$

$$\sigma_b^2 = \frac{1}{B}\sum_{i=1}^{B}(x_i - \mu_b)^2 \ (2)$$

$$\hat{x}_i = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2}} (3)$$

$$or\ \hat{x}_i = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}} \ (3)$$

$$Adding\ \epsilon\ helps\ when\ \sigma_b^2\ is\ small$$

$$y_i = \mathcal{BN}(x_i) = \gamma . x_i + \beta \ (4)$$

Trainable parameters

Batch normalization

Limitations of Batch Normalization

- In batch normalization, we use the *batch statistics*: the mean and standard deviation corresponding to the current mini-batch. However, when the batch size is small, the sample mean and sample standard deviation are not representative enough of the actual distribution and the network cannot learn anything meaningful.
- As batch normalization depends on batch statistics for normalization, it is less suited for sequence models. This is because, in sequence models, we may have sequences of potentially different lengths and smaller batch sizes corresponding to longer sequences.

Layer normalization

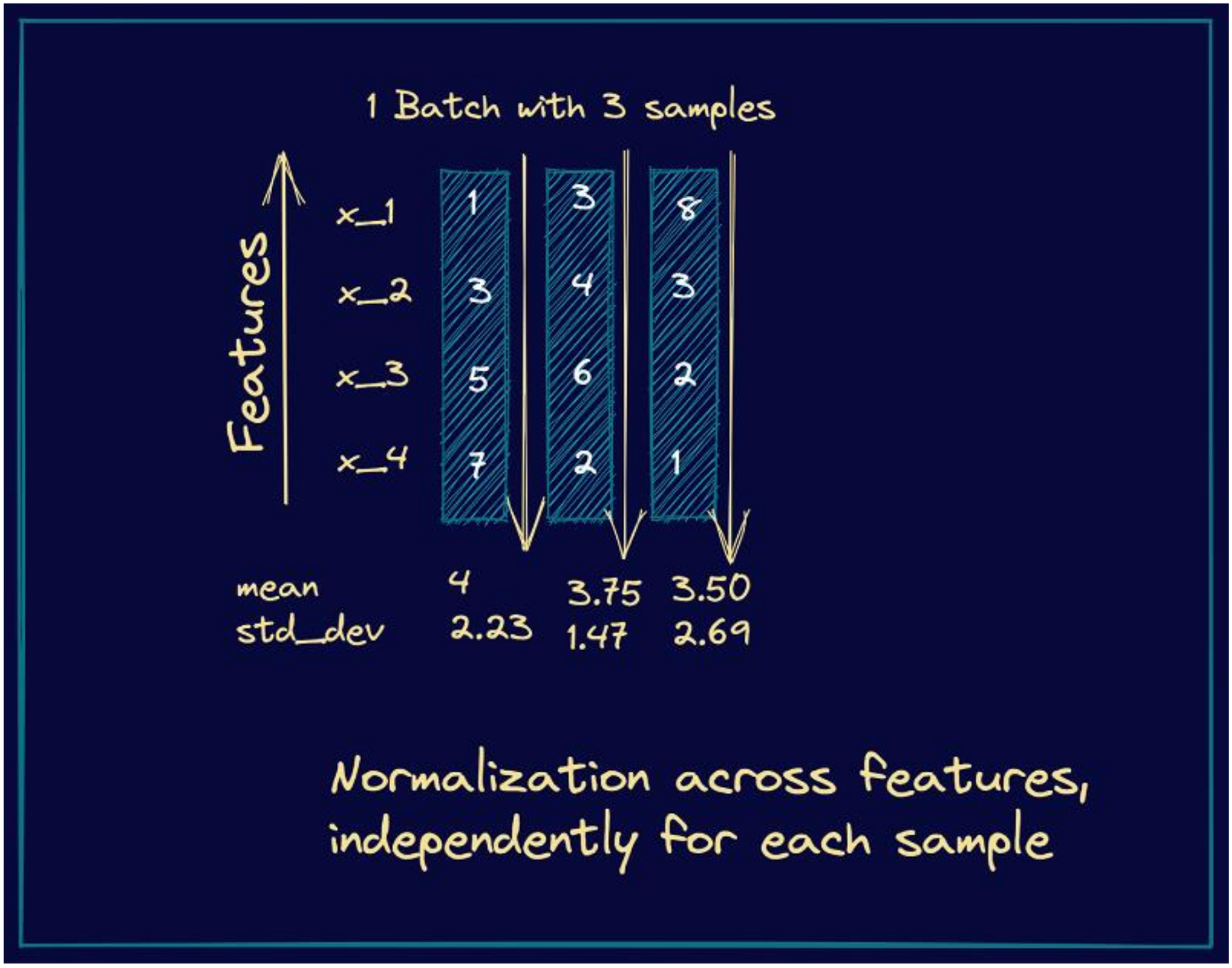
All neurons in a particular layer effectively have the same distribution across all features for a given input

If each input has d features, it's a d-dimensional vector. If there are B elements in a batch, the normalization is done along the length of the d-dimensional vector and not across the batch of size B.

- Normalizing *across all features* but for each of the inputs to a specific layer removes the dependence on batches.

- This makes layer normalization well suited for sequence models such as transformers and recurrent neural networks (RNNs).

Layer normalization



Layer normalization

$$\mu_l = \frac{1}{d}\sum_{i=1}^{d} x_i \ (1)$$

$$\sigma_l^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu_l)^2 \ (2)$$

$$\hat{x}_i = \frac{x_i - \mu_l}{\sqrt{\sigma_l^2}} \ (3)$$

$$or\ \hat{x}_i = \frac{x_i - \mu_l}{\sqrt{\sigma_l^2 + \epsilon}} \ (3)$$

$$Adding\ \epsilon\ helps\ when\ \sigma_l^2\ is\ small$$

$$y_i = \mathcal{LN}(x_i) = \gamma . x_i + \beta \ (4)$$

Trainable parameters

Batch Normalization vs Layer Normalization

So far, we learned how batch and layer normalization work. Let's summarize the key differences between the two techniques.

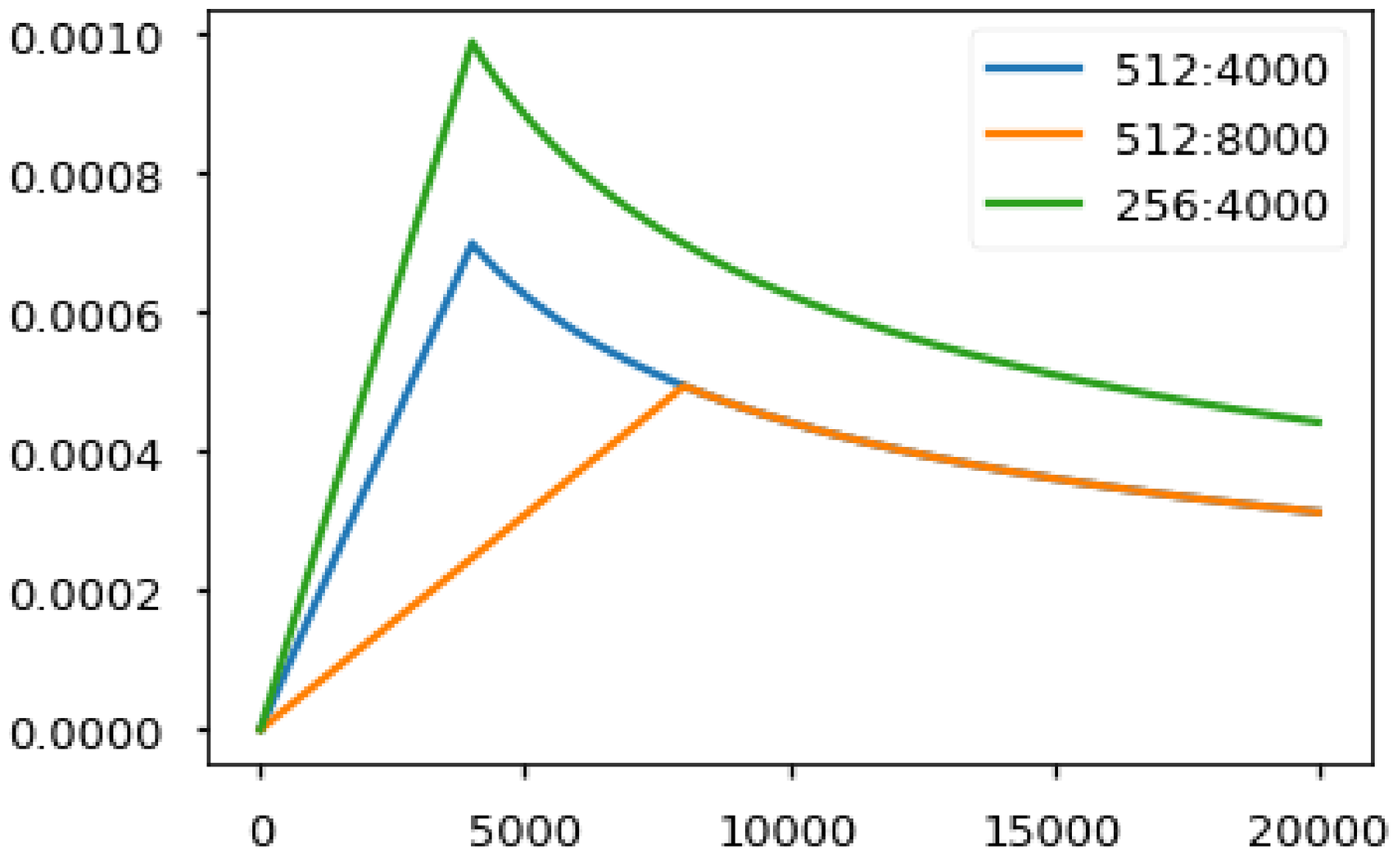
- Batch normalization normalizes each feature independently across the mini-batch. Layer normalization normalizes each of the inputs in the batch independently across all features.
- As batch normalization is dependent on batch size, it's not effective for small batch sizes. Layer normalization is independent of the batch size, so it can be applied to batches with smaller sizes as well.
- Batch normalization requires different processing at training and inference times. As layer normalization is done along the length of input to a specific layer, the same set of operations can be used at both training and inference times.

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.*

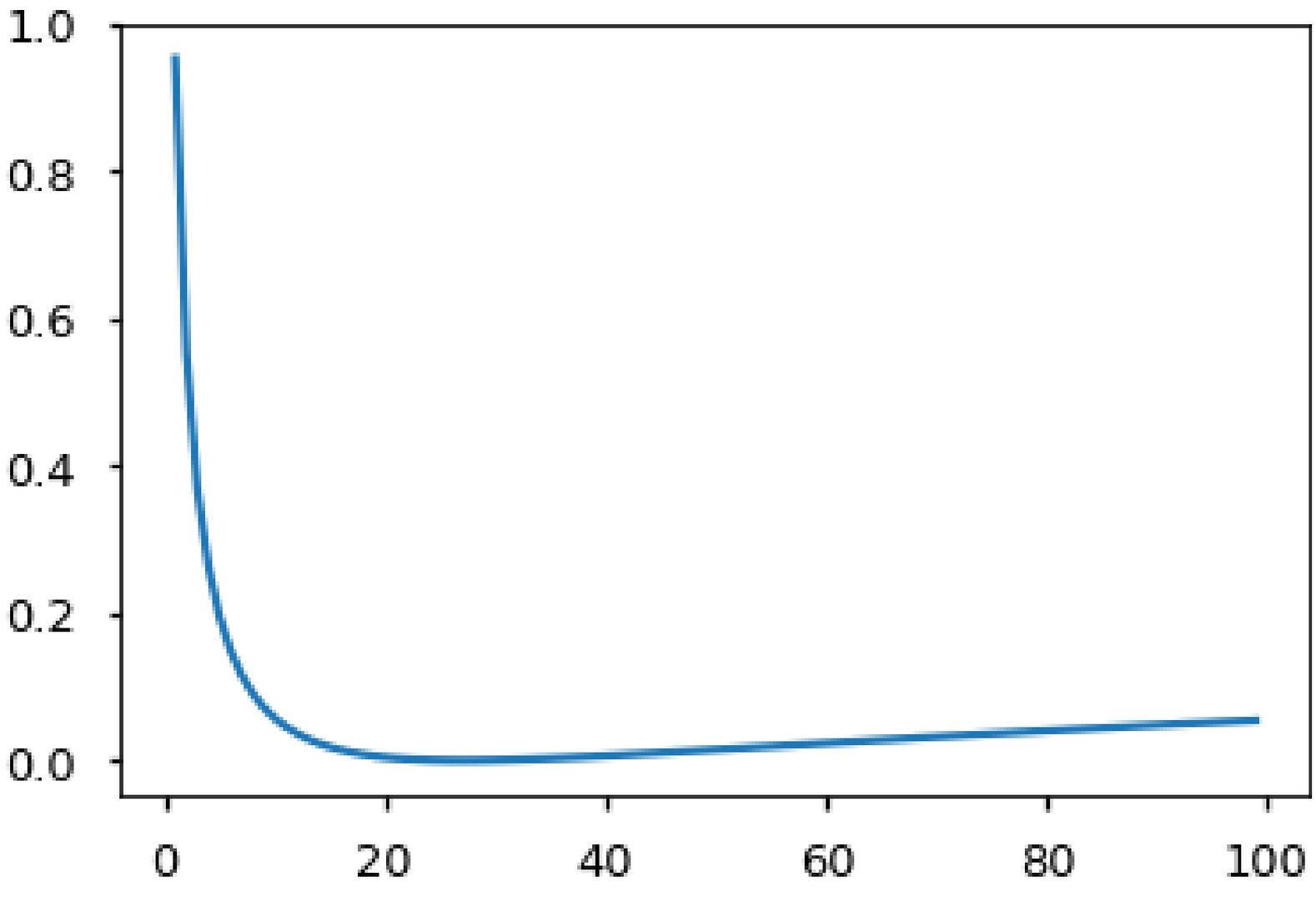
I went to class and took

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!

Loss



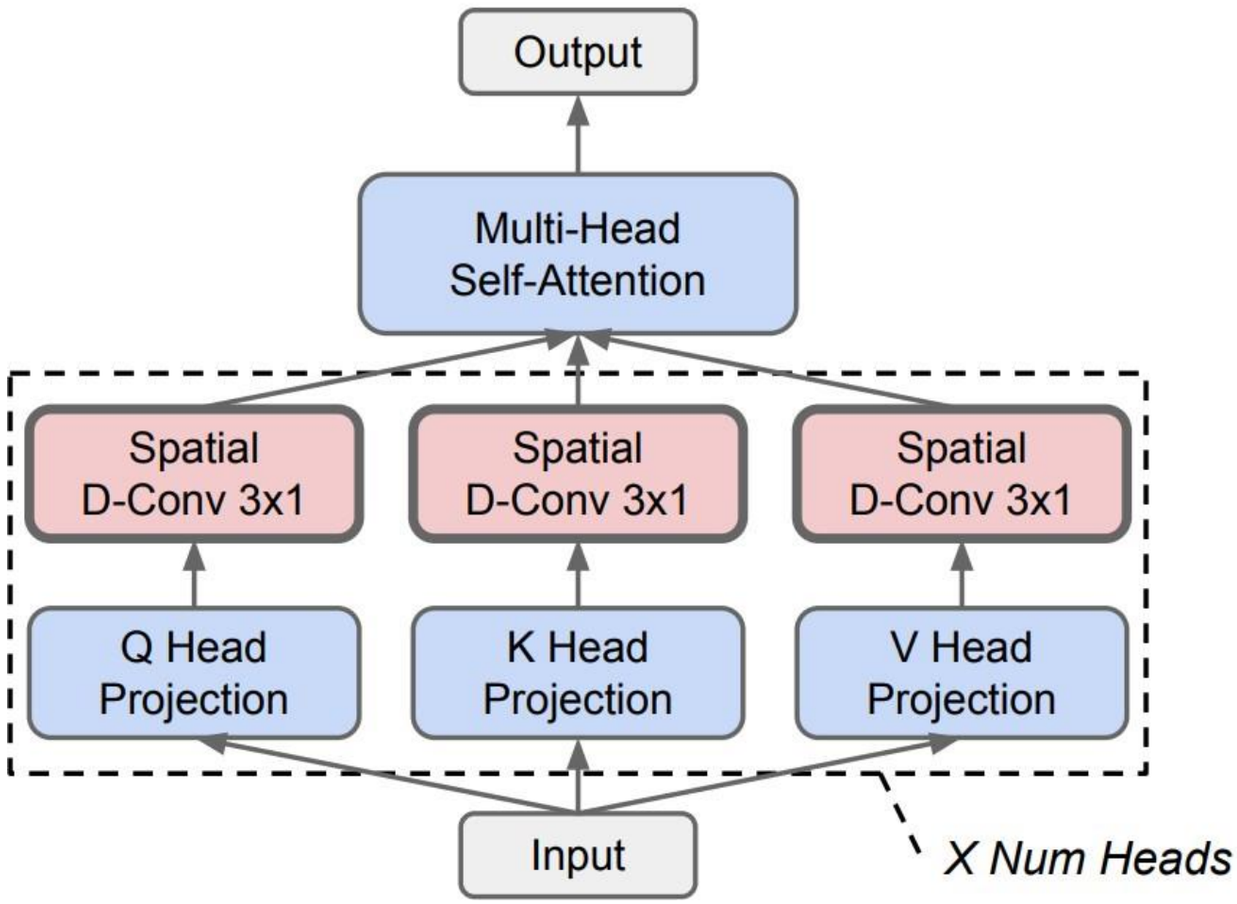
Target word confidence

Why these decisions?

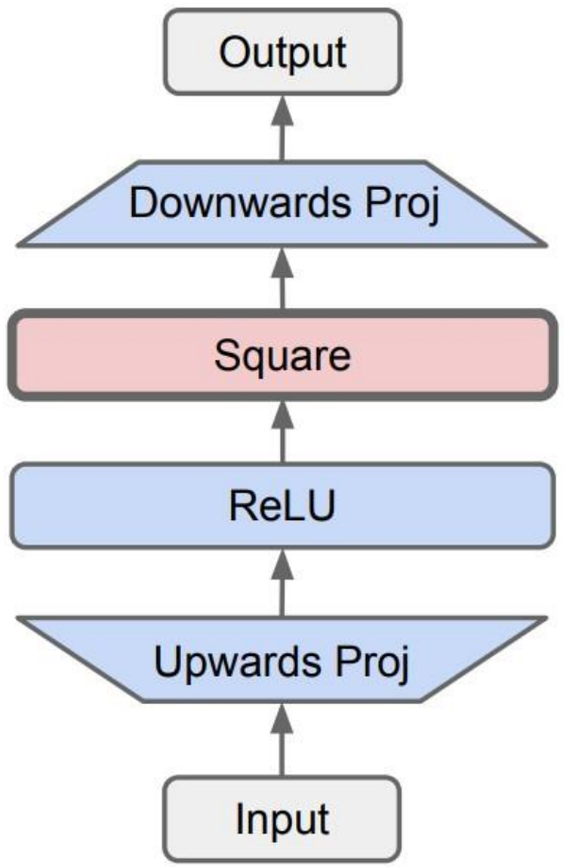
Unsatisfying answer: they empirically worked well.

Neural architecture search finds even better Transformer variants:

Multi-DConv-Head Attention (MDHA)



Squared ReLU in Feed Forward Block



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)