CMPT 413/713: Natural Language Processing

Transformers and Self-Attention

Spring 2024

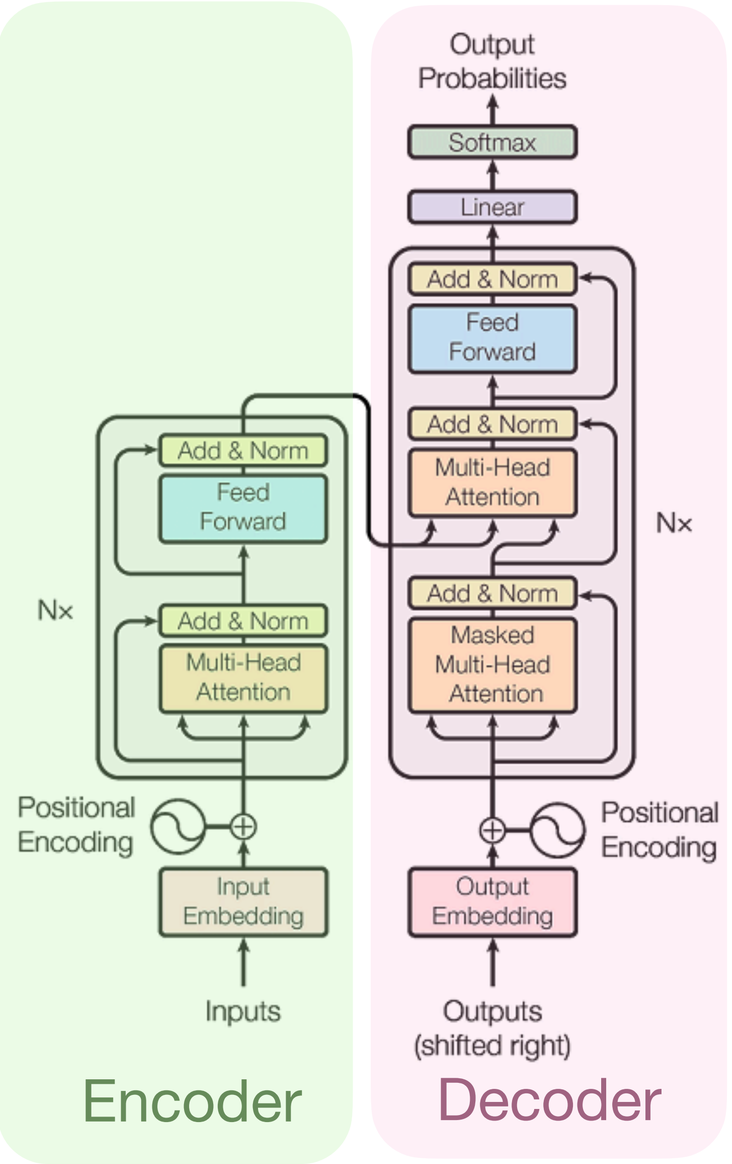
2024-02-07

Adapted from slides from Danqi Chen and Karthik Narasimhan
(with some content from slides from Chris Manning and Abigail See)

Transformers

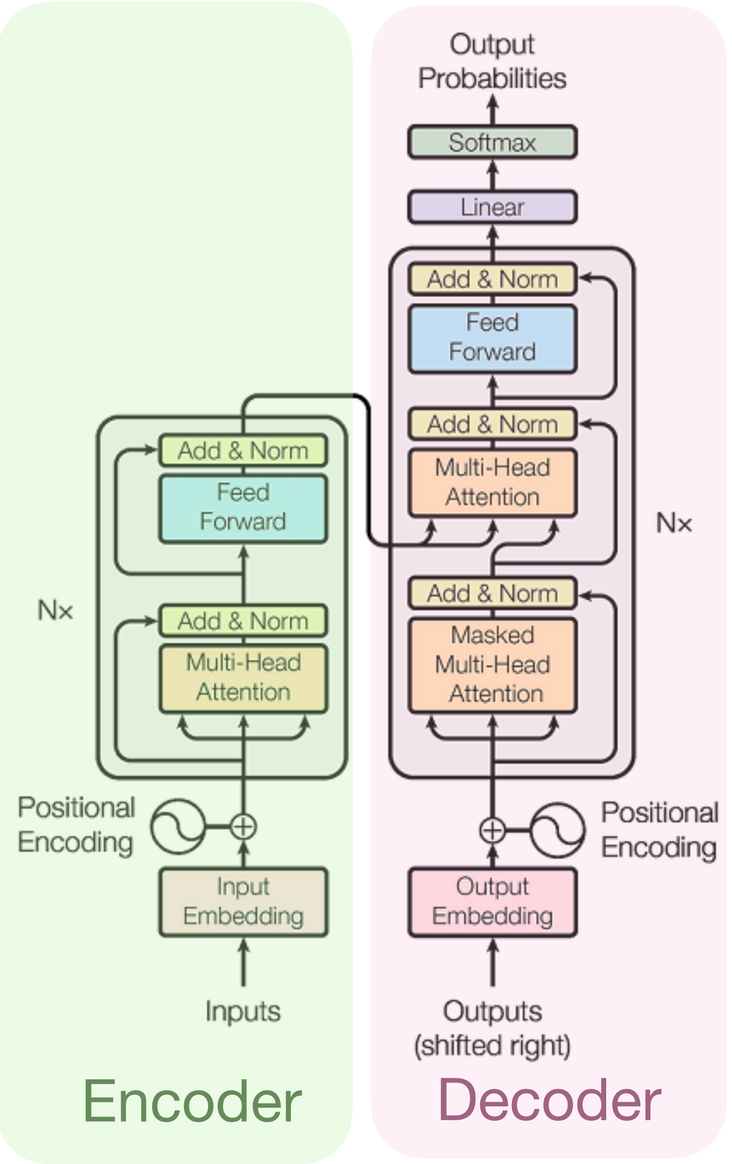
Transformers

- NIPS'17: Attention is All You Need
- Originally proposed for NMT (encoder-decoder framework)
- Used as the base model of BERT (encoder only)
- Key idea: **Multi-head self-attention**
- No recurrence structure any more so it trains much faster



Understanding transformers

- From attention to self-attention
- From self-attention to multi-headed self-attention
- Transformer encoder
- Transformer decoder
- Putting the pieces together



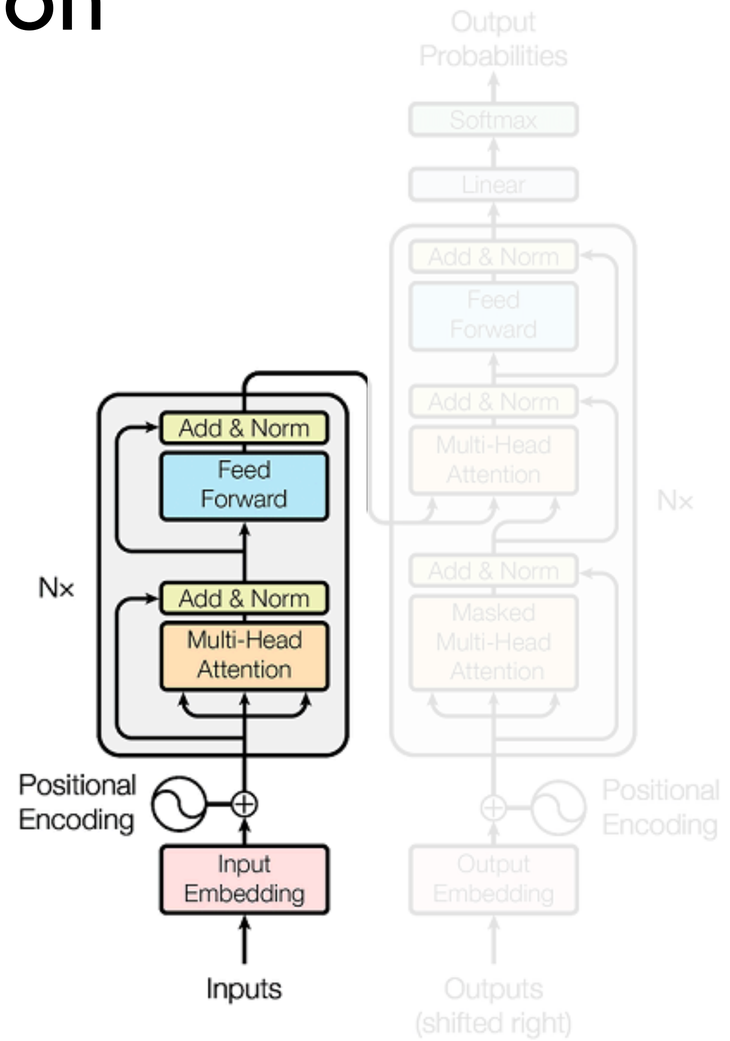
Multi-head self-attention

- Each Transformer block has two-sublayers
  - Multi-Head self-attention
  - 2 layer feedforward NN (with ReLU)

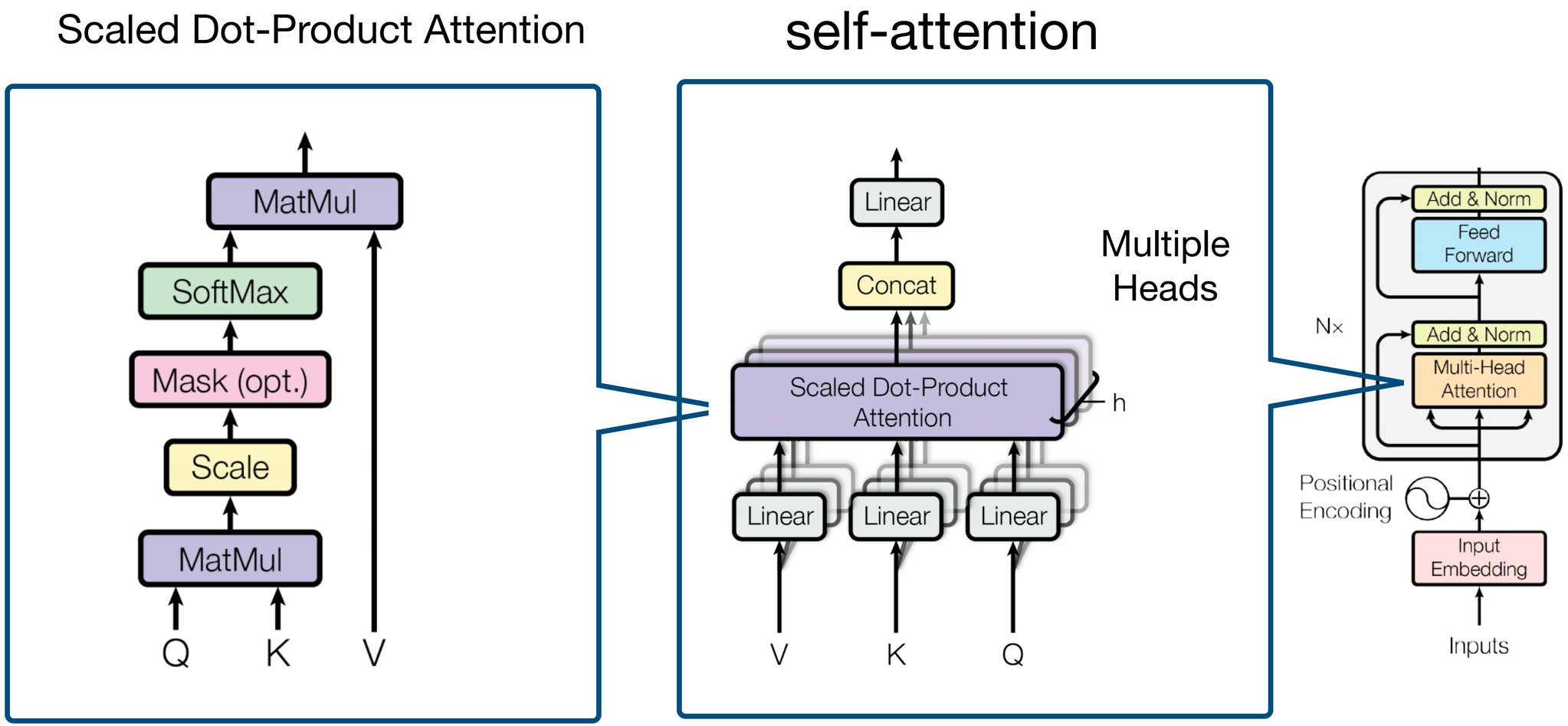
- Each sublayer has a residual connection and a layer normalization
  - LayerNorm(x+SubLayer(x))

Helps the training process!

- Input layer has a positional encoding



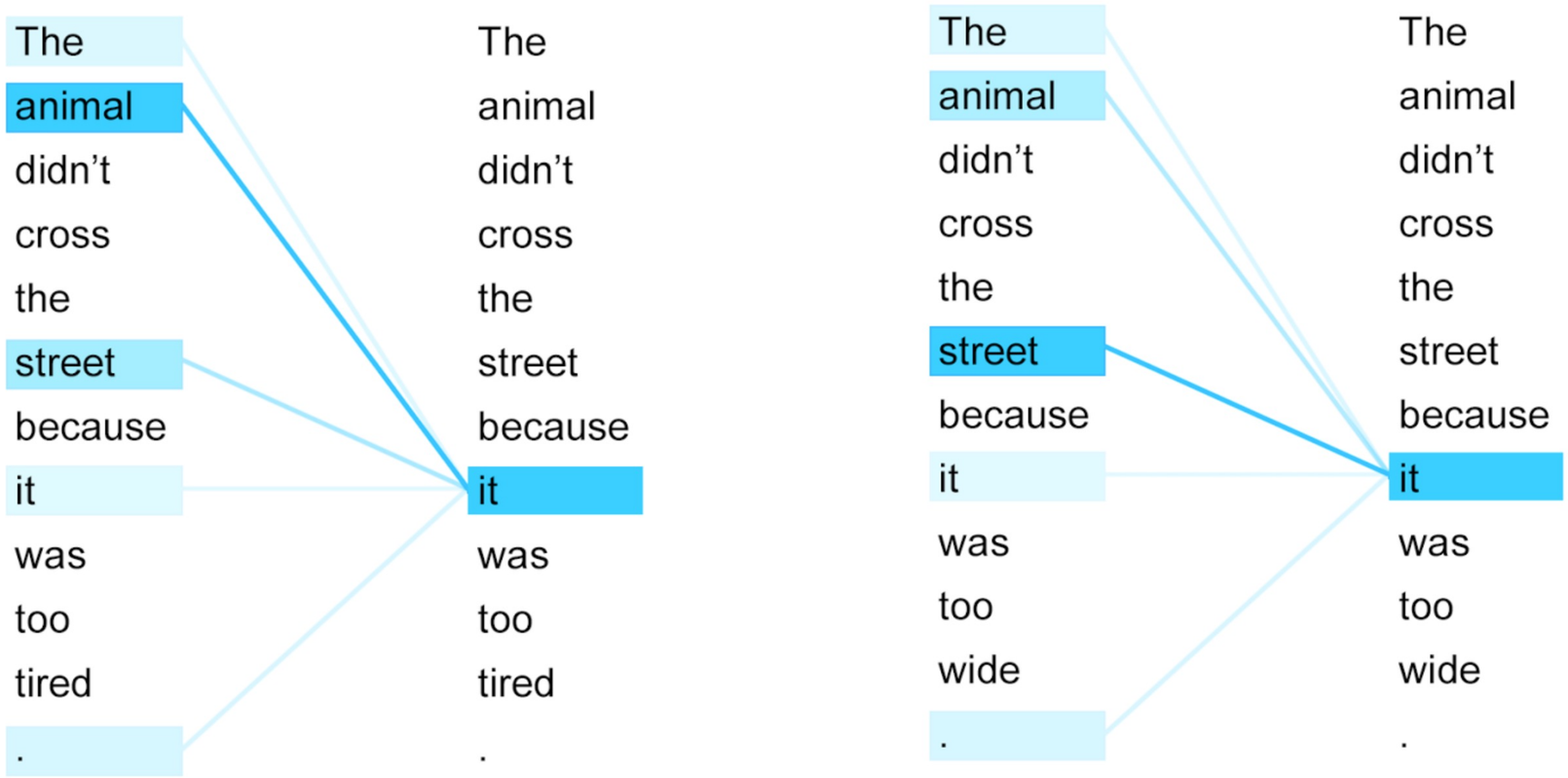
Multi-head self-attention



Attention Is All You Need https://arxiv.org/pdf/1706.03762.pdf

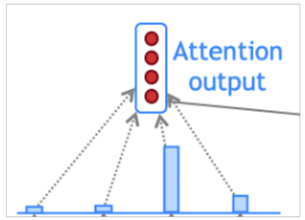
Self-attention

- Attention (correlation) with different parts of itself



- Transformers: modules with scaled dot-product self-attention

Attentive machine translation summary

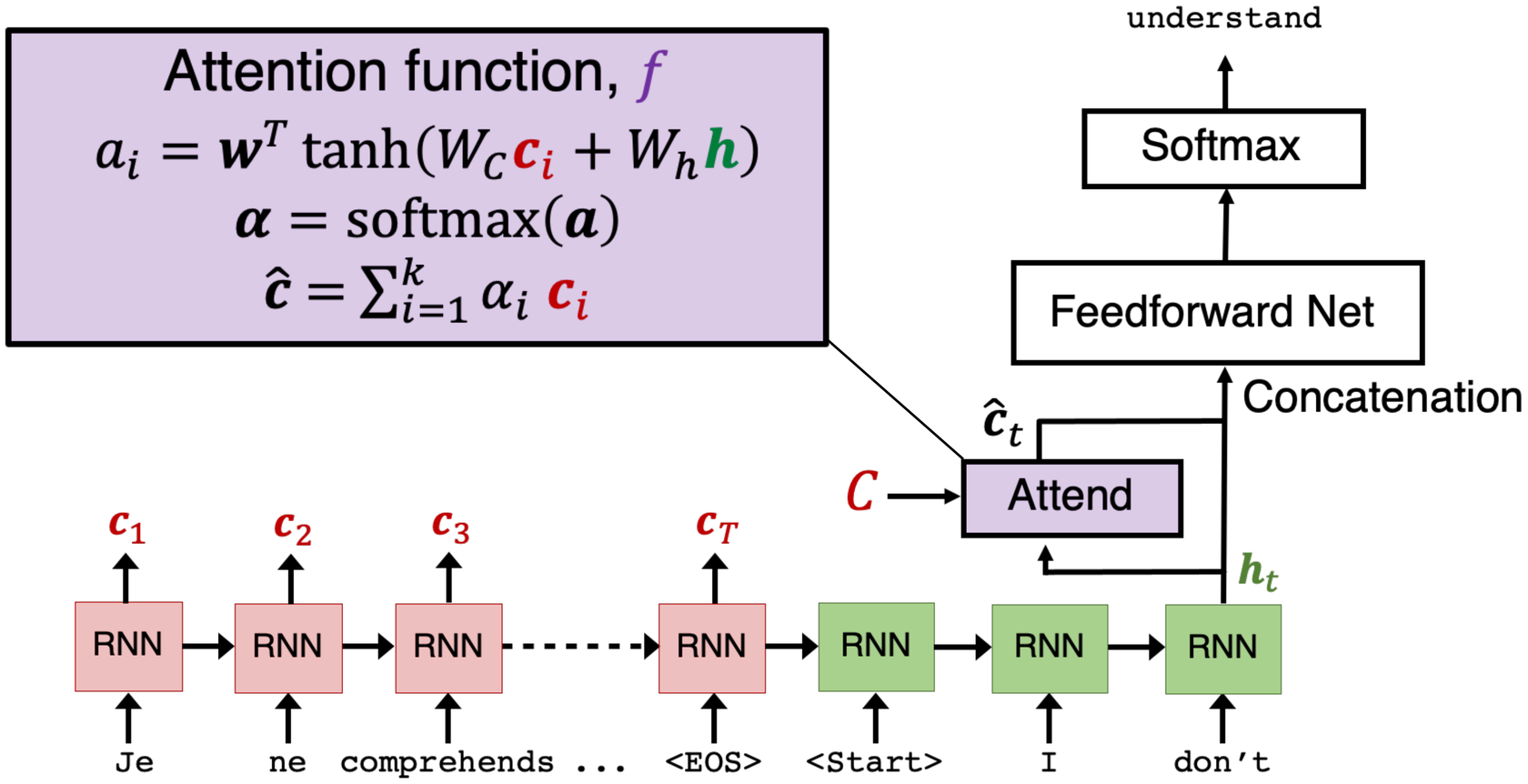


$\hat{c}_t$ is the weighted sum of encoder hidden states

$$C = \{c_1, \ldots, c_T\}$$

(slide credit: Peter Anderson)

Attentive machine translation summary



(slide credit: Peter Anderson)

Summary of attention

Attention function, f

$$a_i = g(\boldsymbol{c}_i, \boldsymbol{z})$$
$$\boldsymbol{\alpha} = \text{softmax}(\boldsymbol{a})$$
$$\hat{\boldsymbol{c}} = \sum_{i=1}^{k} \alpha_i \, \boldsymbol{c}_i$$

Attention scores: $\boldsymbol{a}$ (unnormalized)

Attention weights: $\boldsymbol{\alpha}$ (normalized)

Final attention output

Weighted sum of context features

Attention score $a_i = g(\boldsymbol{c}_i, \boldsymbol{z})$
how well does the attention candidate $\boldsymbol{c}_i$ match the query $\boldsymbol{z}$

- Dot-product attention:
  $g(c_i, z) = z^\top c_i$
- Neural network
  $g(c_i, z) = v^\top \tanh\left(W_1 c_i + W_2 z\right)$

Attention is a *general* deep learning technique

- Given a set of value vectors and a query vector, **attention** is a way to compute a **weighted sum** of the values dependent on the query.
  - The query determines what values to focus on,
    - We say: the query "*attends*" to the values
  - In NMT, each decoder hidden state (query) attends to all the encoder hidden state (values)
- Intuition
  - The weighted sum is a **selective summary** of the information found in the values.
  - It is a way to obtain a **fixed-sized representation** of an arbitrary set of representations (values) based on some other representation (the query)

Attention is always computed the same way

- Assume that we have a set of value $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^{d_v}$ and a query vector $\mathbf{q} \in \mathbb{R}^{d_q}$
- Computing attention consists of the following steps:
  - Compute the attention scores: $e_i = g(\mathbf{v}_i, \mathbf{q}), \mathbf{e} \in \mathbb{R}^n$
  - Take softmax to get the attention distribution

$$\alpha = \text{softmax}(\mathbf{e}) \in \mathbb{R}^n$$

  - Use attention distribution to take weighted sum of values

$$\mathbf{a} = \sum_{i-1}^{n} = \alpha_i \mathbf{v}_i \in \mathbb{R}^{d_v}$$

- A more general form: use a set of keys and values
  - The keys are used to compute the **attention scores**
  - The values are used to compute the **output vector**

General form of attention: key-value-query

- Assume that we have a set of key-value pairs $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^{d_v}$, $\mathbf{k}_1, \ldots, \mathbf{k}_n \in \mathbb{R}^{d_k}$ and a query vector $\mathbf{q} \in \mathbb{R}^{d_q}$
- Computing attention consists of the following steps:
  - Compute the attention scores: $e_i = g(\mathbf{k}_i, \mathbf{q}), \mathbf{e} \in \mathbb{R}^n$
  - Take softmax to get the attention distribution

$$\alpha = \text{softmax}(\mathbf{e}) \in \mathbb{R}^n$$

  - Use attention distribution to take weighted sum of values

$$\mathbf{a} = \sum_{i-1}^{n} = \alpha_i \mathbf{v}_i \in \mathbb{R}^{d_v}$$

General form of attention: key-value-query

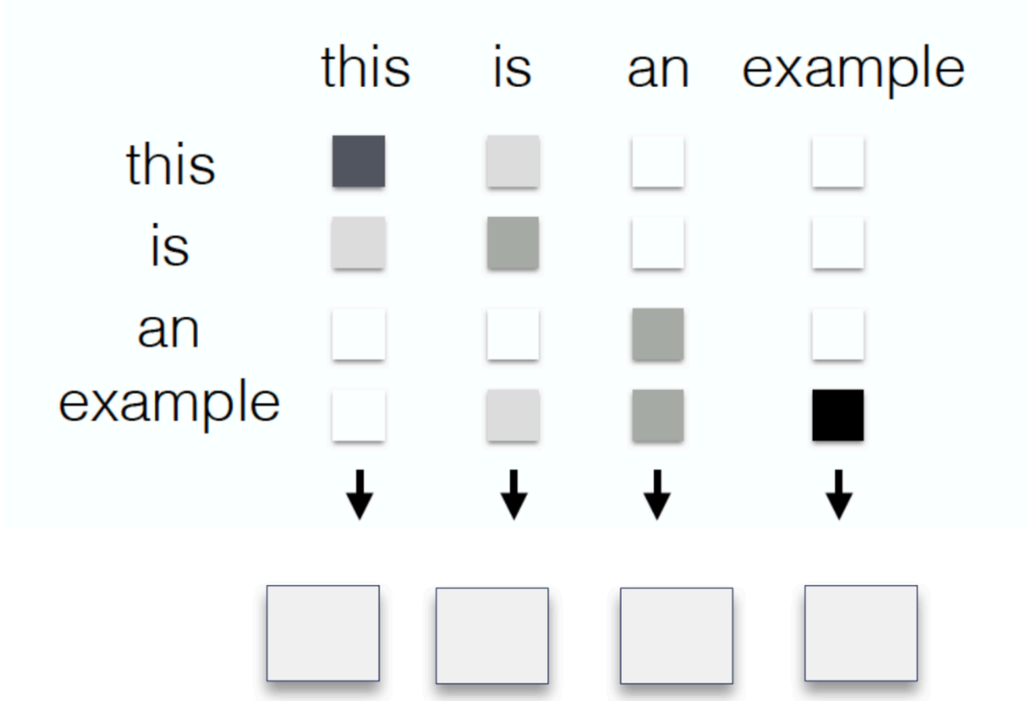
- **Attention** is a way to compute a **weighted sum** of the values dependent on the query and the corresponding keys.
  - All of these (key value query) are represented using **vectors**
    - The query and key are used for addressing (contains partial information). While the values provide more complete information
    - The weighted sum is a **selective summary** of the information found in the values.
    - It is a way to obtain a **fixed-sized representation** of an arbitrary set of representations (values) based on some other representation (the query)

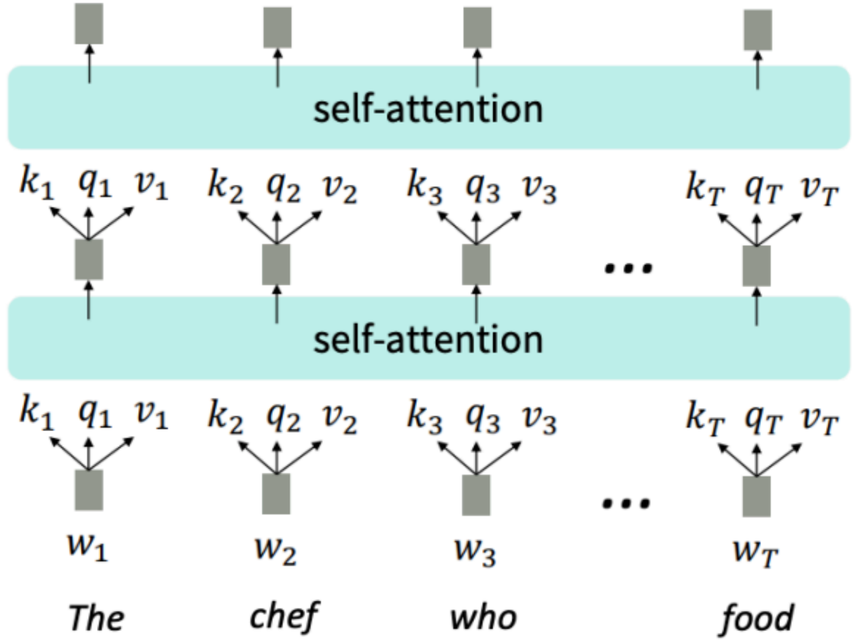
Self Attention

(also referred to as Intra-Attention)

- **Self-attention**: let's use each word as query and compute the attention with all the other words (other words are the keys and values)

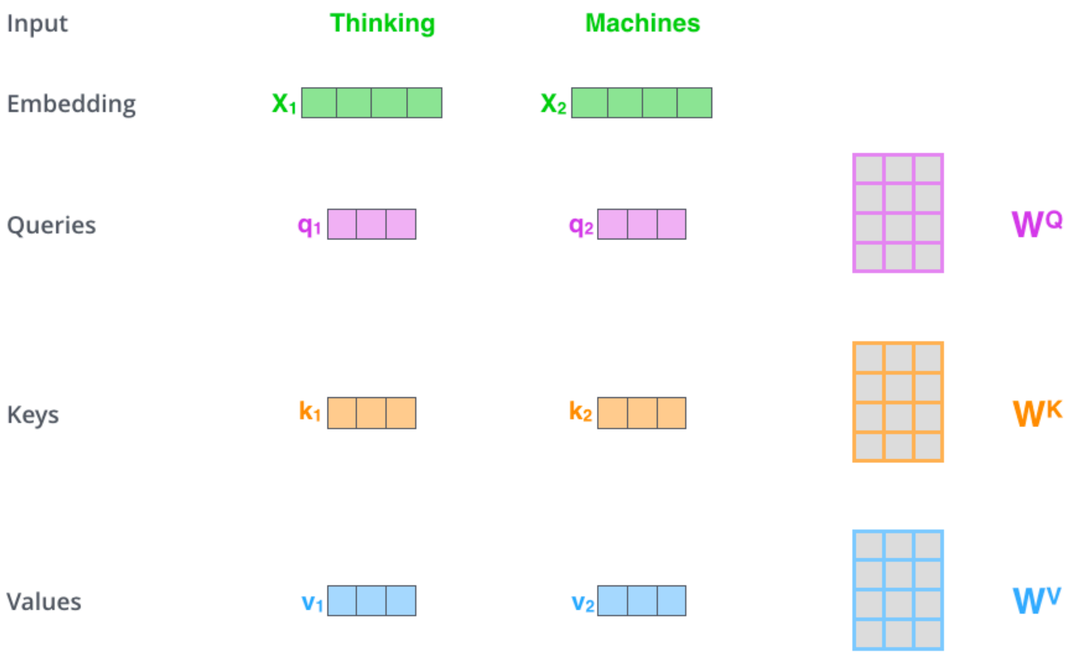
  = the word vectors themselves select each other



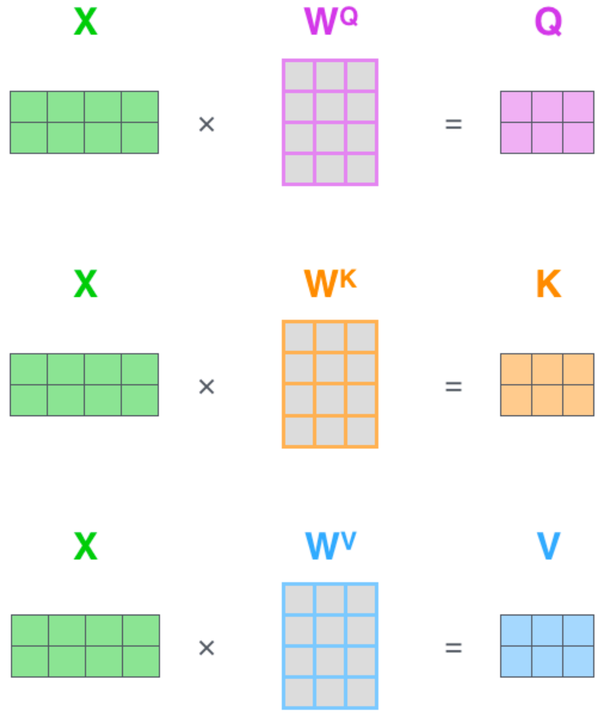


How to get key-value-query for each word?

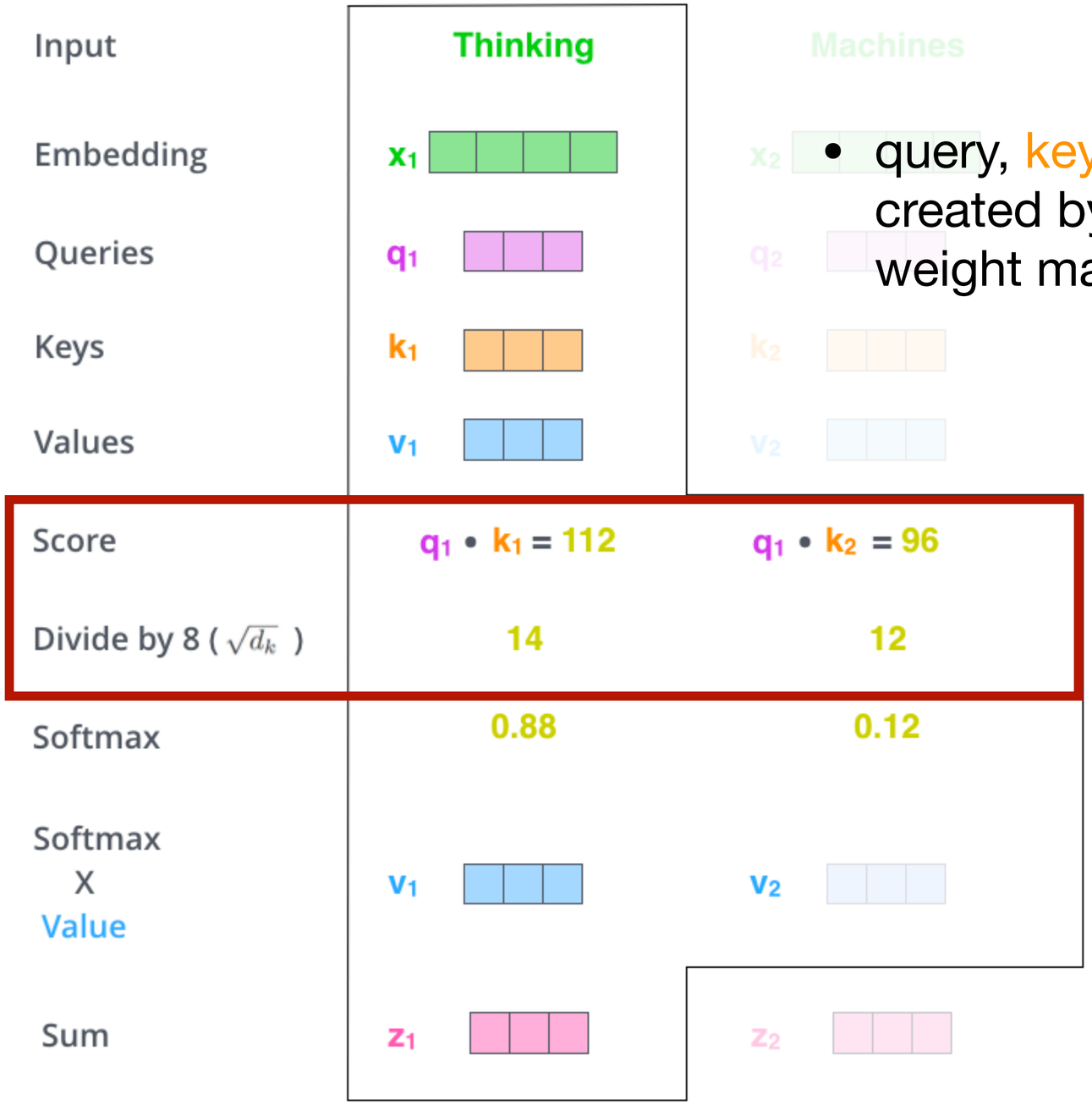
- For each word, we have vectors for the key-value-query
- These vectors are created by multiplying the word embedding by trained weight matrices



Stack into matrices and compute all at once!



(figure credit: Jay Alammar http://jalammar.github.io/illustrated-transformer/)



- query, key, and value vectors created by multiplying learned weight matrices with embedding

- Can be any kind of attention function
- For transformers, this is the scaled dot-product attention

(figure credit: Jay Alammar http://jalammar.github.io/illustrated-transformer/)

Recall: types of attention

- Assume keys $\mathbf{k}_1, \mathbf{k}_2, \ldots, \mathbf{k}_n$ and query $\mathbf{q}$

1. **Dot-product attention** (assumes equal dimensions for $\mathbf{k}_i$ and $\mathbf{q}$):

$$g(\mathbf{k}_i, \mathbf{q}) = \mathbf{q}^T \mathbf{k}_i \in \mathbb{R}$$

more efficient (matrix multiplication)

Simplest (no extra parameters)
Does not work well for large dimensions

2. **Bilinear / multiplicative attention:**

$$g(\mathbf{k}_i, \mathbf{q}) = \mathbf{k}^T \mathbf{W} \mathbf{k}_i \in \mathbb{R}, \text{ where } \mathbf{W} \text{ is a weight matrix}$$

More flexible than dot-product (W is trainable)

3. **Additive attention (essentially MLP):**

$$g(\mathbf{k}_i, \mathbf{q}) = \mathbf{w}^T \tanh\left(\mathbf{W}_1 \mathbf{k}_i + \mathbf{W}_2 \mathbf{q}\right) \in \mathbb{R}$$

where $\mathbf{W}_1, \mathbf{W}_2$ are weight matrices and $\mathbf{w}$ is a weight vector

Perform better for larger dimensions

Scaled dot-product attention

- Assume keys $\mathbf{k}_1, \mathbf{k}_2, \ldots, \mathbf{k}_n$ and query $\mathbf{q}$

1. **Dot-product attention** (assumes equal dimensions for $\mathbf{k}_i$ and $\mathbf{q}$):

$$g(\mathbf{k}_i, \mathbf{q}) = \mathbf{q}^T \mathbf{k}_i \in \mathbb{R}$$

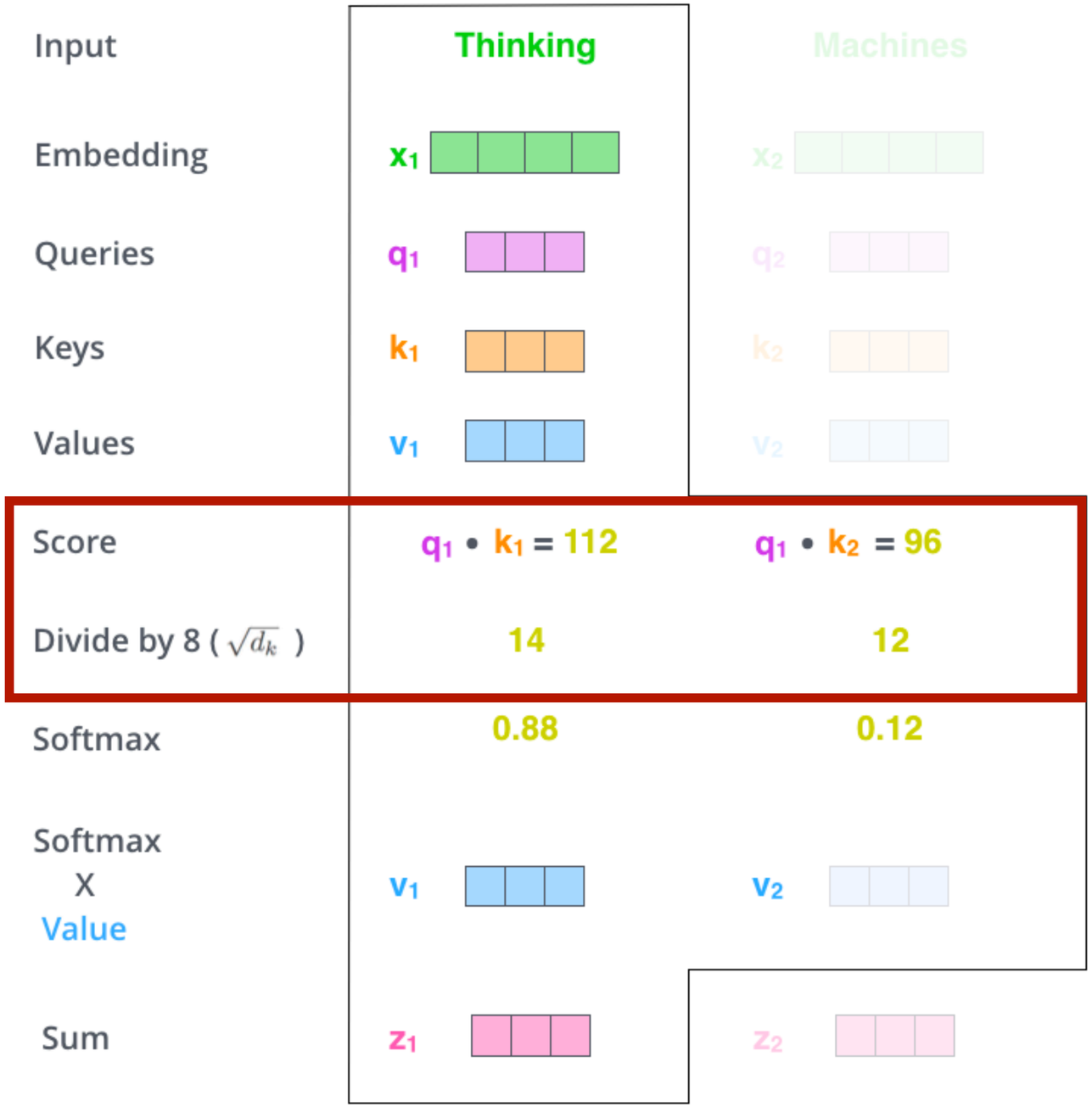
2. **Scaled dot-product attention:**

$$g(\mathbf{k}_i, \mathbf{q}) = \frac{\mathbf{q}^T \mathbf{k}_i}{\sqrt{d}} \in \mathbb{R}$$

Scale of dot product increases as dimension gets larger
Perform poorly for large *d*
Softmax has small gradient

Scaled dot product will perform well for larger dimensions

Scaling factor: d = dimension of hidden state



- Can be any kind of attention function
- For transformers, this is the scaled dot-product attention

- z_1 is the final vector of attended values for “Thinking” as the query

(figure credit: Jay Alammar http://jalammar.github.io/illustrated-transformer/)

Self-attention in equations

- A self-attention layer maps a sequence of input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^{d_1}$ to a sequence of n vectors: $\mathbf{y}_1, \ldots, \mathbf{y}_n \in \mathbb{R}^{d_2}$
  - Note: this is similar as an RNN layer and can be used to replace an RNN layer
- First, construct a set of queries, keys, and values:

$$\mathbf{q}_i = W^Q \mathbf{x}_i, W^Q \in \mathbb{R}^{d_q \times d_1}$$
$$\mathbf{k}_i = W^K \mathbf{x}_i, W^K \in \mathbb{R}^{d_k \times d_1}$$
$$\mathbf{v}_i = W^V \mathbf{x}_i, W^V \in \mathbb{R}^{d_v \times d_1}$$

- Second, for each $\mathbf{q}_i$, compute attentions scores and attention distribution

$$\alpha_{i,j} = \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}\right)$$

Scaled dot-product so $d_k = d_q$

- Finally, compute the weighted sum:

$$\mathbf{y}_i = \sum_{j=1}^{n} \alpha_{i,j} \mathbf{v}_j \in \mathbb{R}^{d_v} \qquad d_v = d_2$$

Self-attention: matrix notation

$$X \in \mathbb{R}^{n \times d_1}$$

$$Q = XW^Q, W^Q \in \mathbb{R}^{d_1 \times d_q}$$

$$K = XW^K, W^K \in \mathbb{R}^{d_1 \times d_k}$$

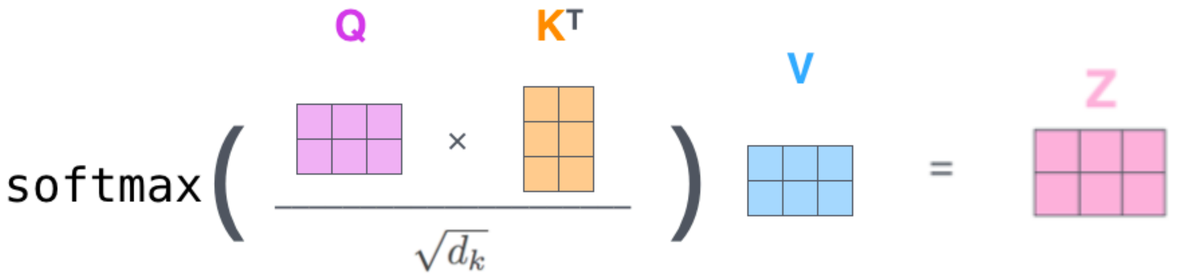
$$V = XW^V, W^V \in \mathbb{R}^{d_1 \times d_v}$$

Note: the notation on this slide are following the original paper
(= the transpose of the matrices in the previous slide)

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

$n \times d_q$ $d_k \times n$ $n \times d_v$

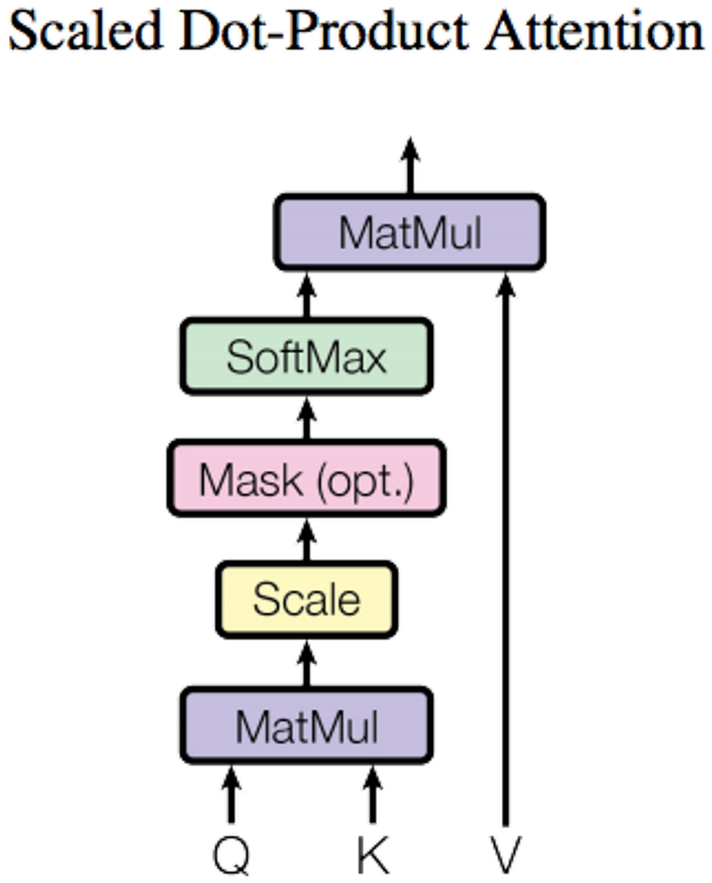
Be careful to make sure
the softmax is over the correct dimension



(figure credit: Jay Alammar
http://jalammar.github.io/illustrated-transformer/)

Scaled Dot Product Attention

Efficient, stable training



Let $Z \in \mathbb{R}^{M \times d_z}$ be a matrix of task context vectors to attend to

Let $C \in \mathbb{R}^{N \times d_C}$ be a matrix of input vectors to attend over

$\boldsymbol{SDPAttention(Z, C)}$:

$Q = W_Q Z^T$ $\quad W_Q \in \mathbb{R}^{d_q \times d_z}$ $\quad d_q = d_k$

$K = W_K C^T$ $\quad W_K \in \mathbb{R}^{d_k \times d_C}$

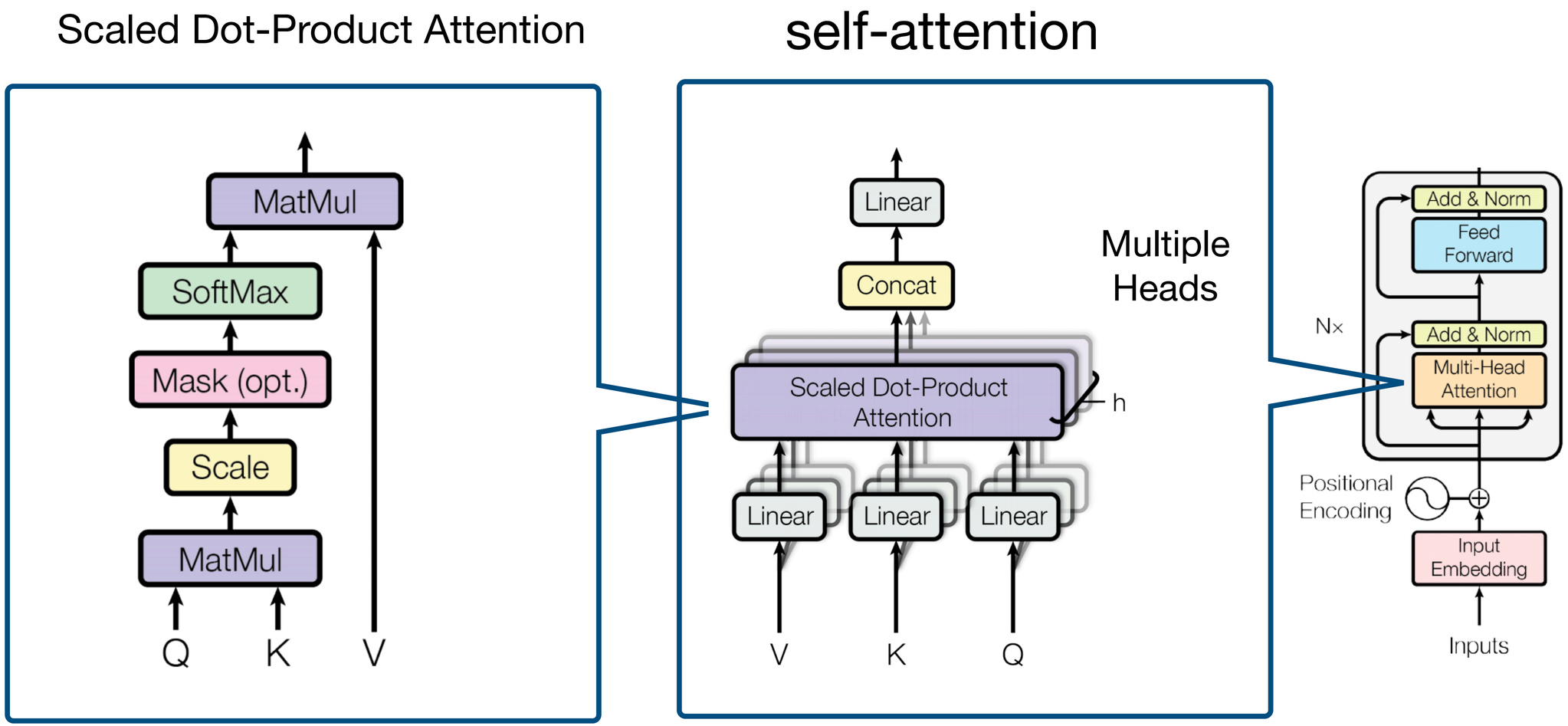
$V = W_V C^T$ $\quad W_V \in \mathbb{R}^{d_v \times d_C}$

Return $\hat{V} = softmax\left(\frac{Q^T K}{\sqrt{d_k}}\right) V$

$\hat{V} \in \mathbb{R}^{M \times d_v}$ be a matrix of attended values

Attention Is All You Need https://arxiv.org/pdf/1706.03762.pdf

Multi-head self-attention



Attention Is All You Need https://arxiv.org/pdf/1706.03762.pdf

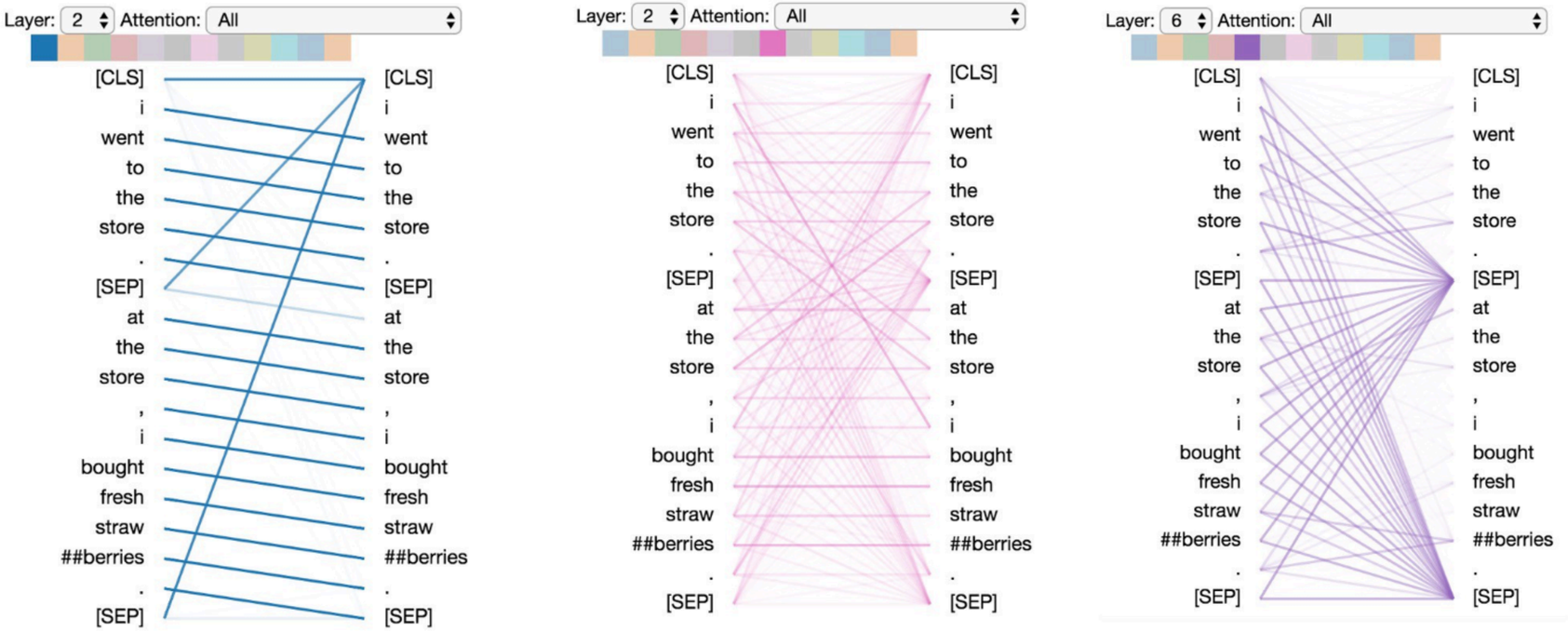
Multi-head self-attention

One head is not expressive enough. Let's have multiple heads!

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

$$\text{head}_i = A(XW_i^Q, XW_i^K, XW_i^V)$$

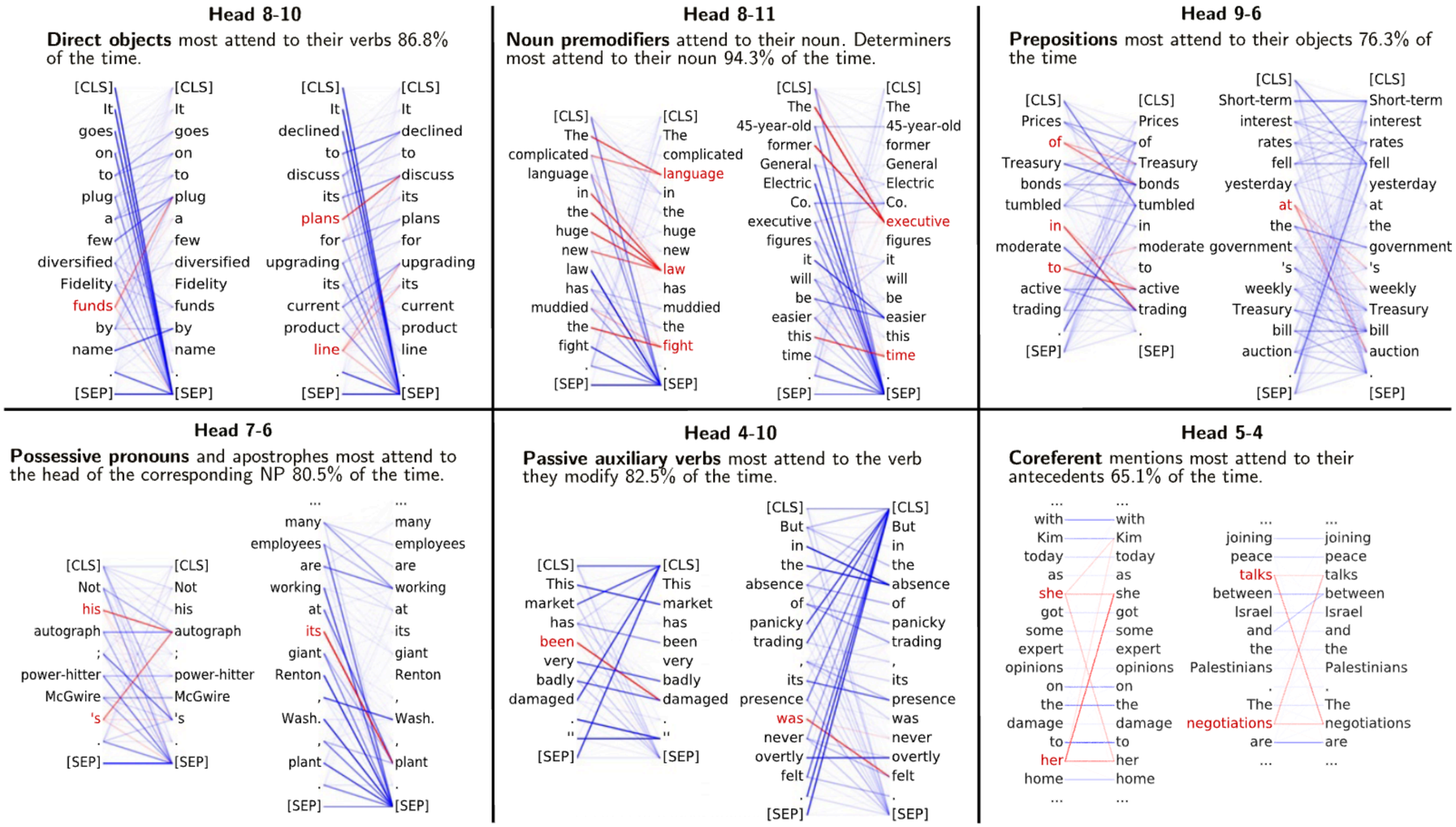
In practice, $h = 8$,
$d = d_{out}/h$, $W^O \in \mathbb{R}^{d_{out} \times d_{out}}$



https://github.com/jessevig/bertviz

Why different heads?

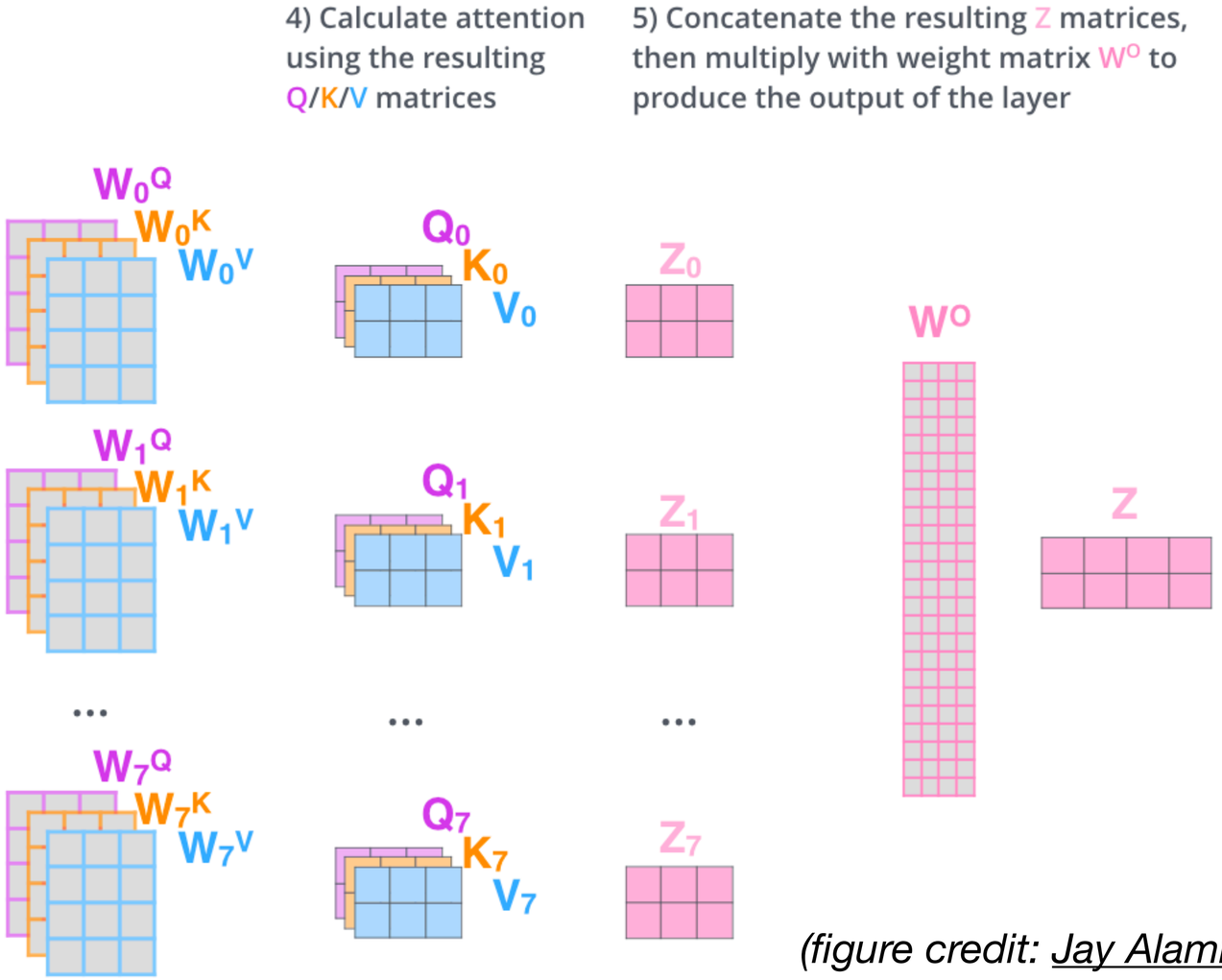
- Different heads learn to attend to different things



Emergent linguistic structure in artificial neural networks trained by self-supervision, Manning et al, PNAS 2019

Multiple heads

- Multiple (different) representations for each query, key, and values
- Different weight matrices —> different vectors
- Different ways for the words to interact with each other



(figure credit: Jay Alammar http://jalammar.github.io/illustrated-transformer/)

Multi-head attention

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

$$\text{head}_i = A(XW_i^Q, XW_i^K, XW_i^V)$$

- In practice, we use a reduced dimension for each head.

$$W_i^Q \in \mathbb{R}^{d_1 \times d_q}, W_i^K \in \mathbb{R}^{d_1 \times d_k}, W_i^V \in \mathbb{R}^{d_1 \times d_v}$$

$$d_q = d_k = d_v = d/h$$

d = hidden size, h = # of heads

$$W^O \in \mathbb{R}^{d \times d_2}$$

If we stack multiple layers, usually $d_1 = d_2 = d$

- The total computational cost is similar to that of single-head attention with full dimensionality

Adding nonlinearities

- There is no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors

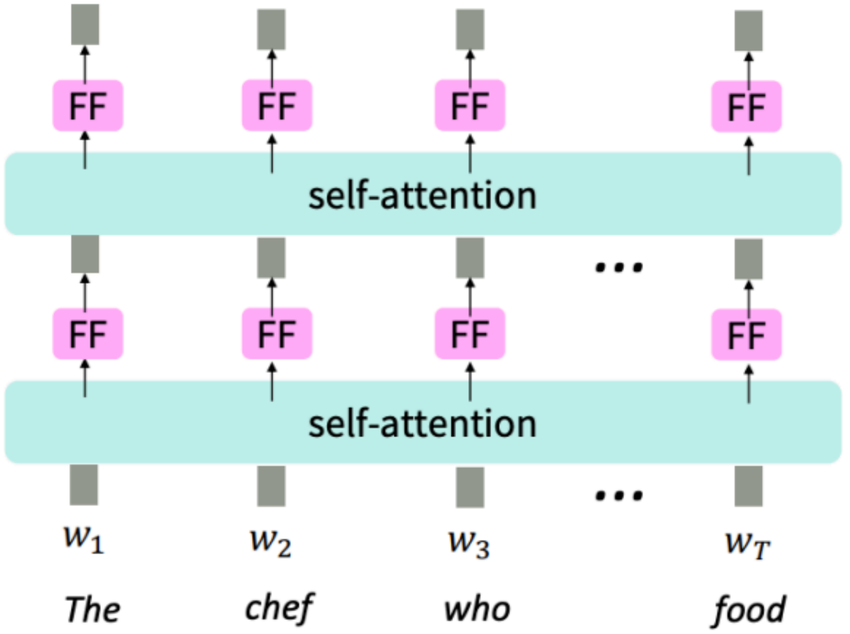
- Simple fix: add a feed-forward network to post-process each output vector

$$\text{FFN}(\mathbf{x}_i) = W_2 \text{ReLU}(W_1 \mathbf{x}_i + \mathbf{b}_1) + \mathbf{b}_2$$

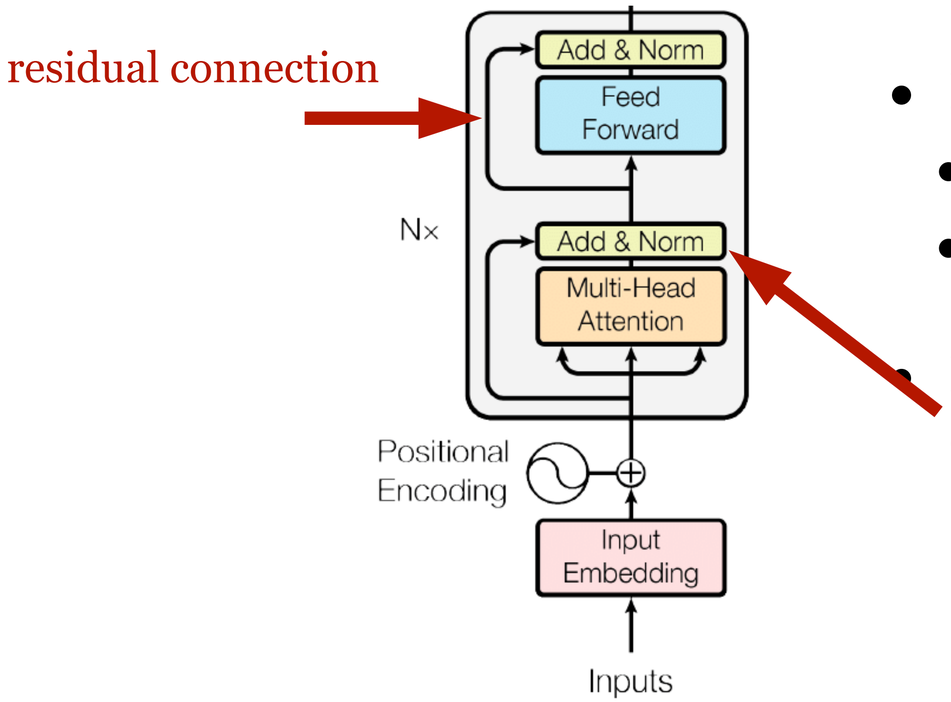
$$W_1 \in \mathbb{R}^{d_{ff} \times d}, \mathbf{b}_1 \in \mathbb{R}^{d_{ff}}$$

$$W_2 \in \mathbb{R}^{d \times d_{ff}}, \mathbf{b}_2 \in \mathbb{R}^{d}$$

In practice, they use $d_{ff} = 4d$



Transformer Encoder



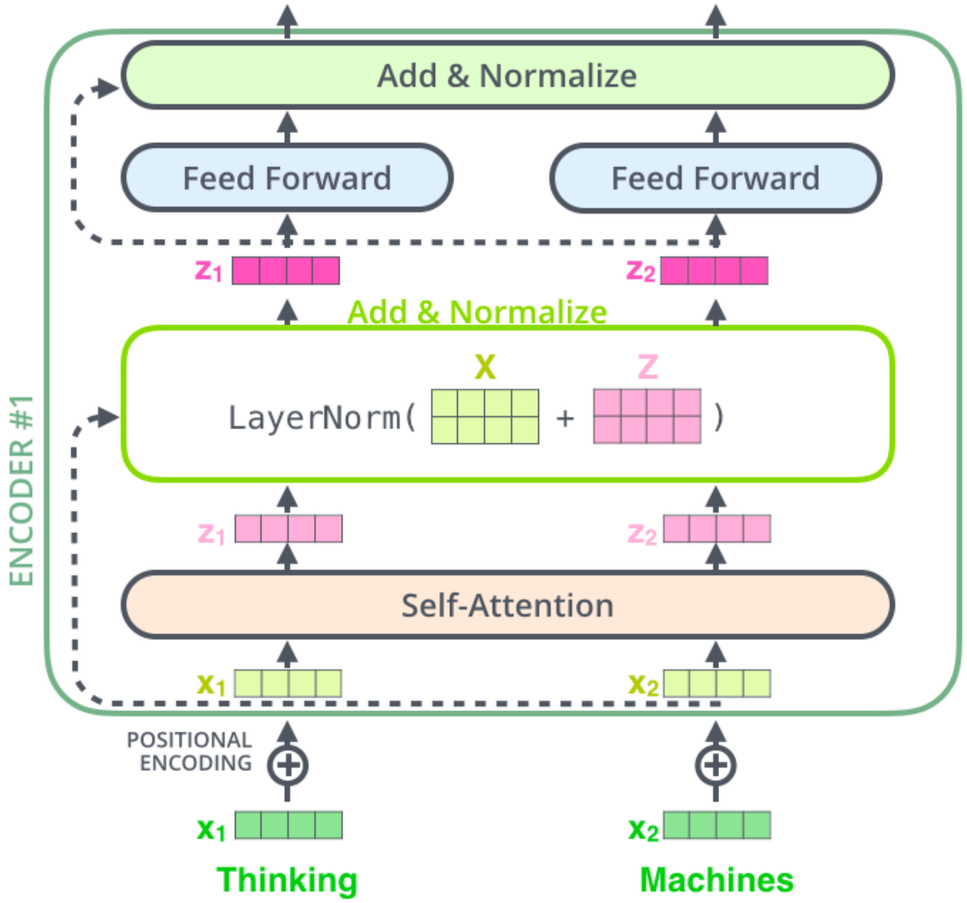
- Each Transformer block has two sub-layers
  - Multi-head attention
  - 2-layer feedforward NN (with ReLU)
- Each sublayer has a residual connection and a layer normalization

$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

(He et al, 2016): Residual connections

(Ba et al, 2016): Layer Normalization

Residual connections and Layer Normalization



(figure credit: Jay Alammar http://jalammar.github.io/illustrated-transformer/)

LayerNorm

- changes input features to have mean 0 and variance 1 per layer.
- Adds two more parameters

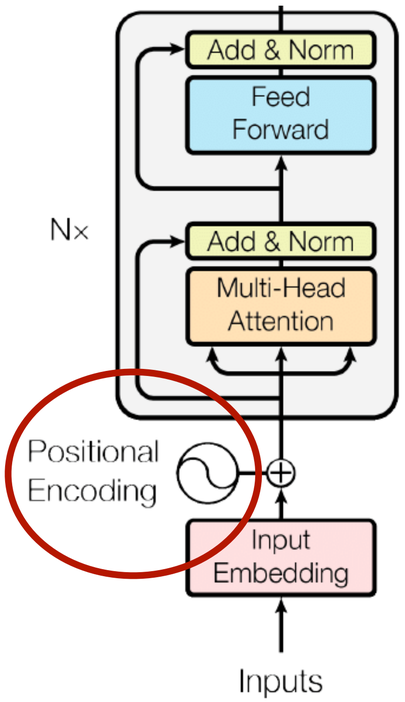
$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

$$h_i = \frac{g_i}{\sigma_i}\left(a_i - \mu_i\right) + b_i$$

- For more stable and efficient training

(Ba et al, 2016): Layer Normalization

Transformer Encoder



- Each Transformer block has two sub-layers
  - Multi-head attention
  - 2-layer feedforward NN (with ReLU)
- Each sublayer has a residual connection and a layer normalization

$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

- Input layer has a positional encoding

Necessary for the model to know the position of the token

(He et al, 2016): Residual connections

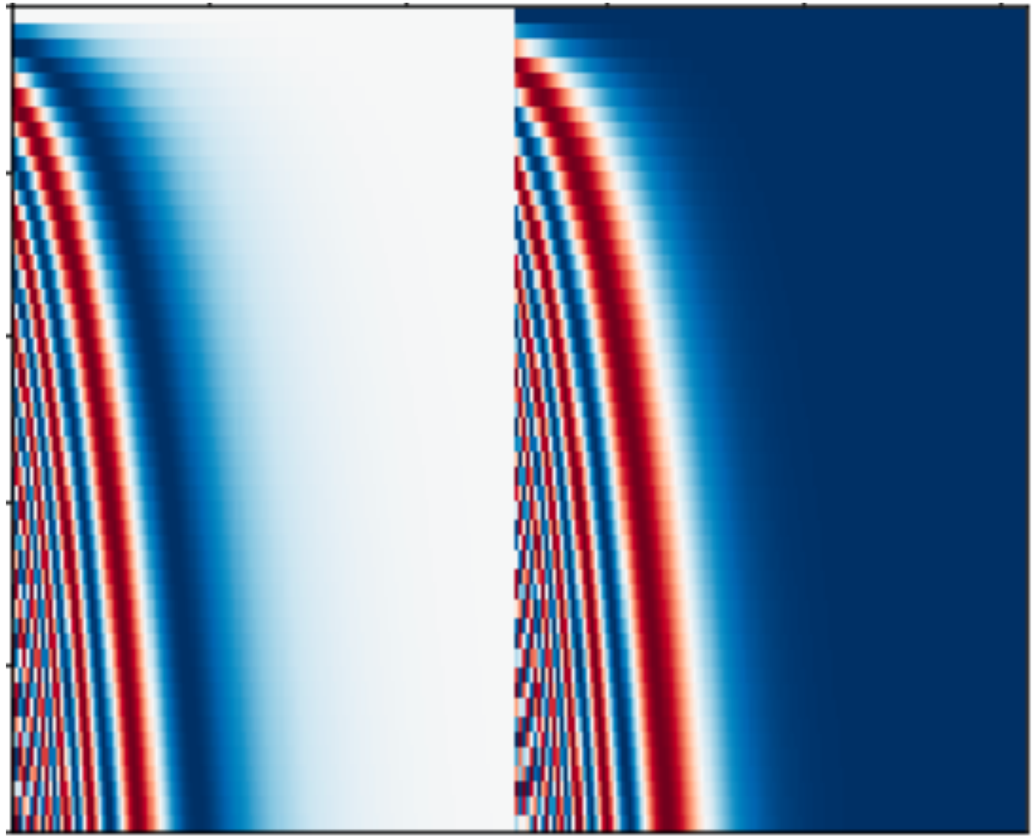
(Ba et al, 2016): Layer Normalization

Positional encoding

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

$$\omega_k = \frac{1}{10000^{2k/d}}$$

$$\overrightarrow{p_t} = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \\ \vdots \\ \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_{d \times 1}$$



Sine Cosine

t = position

d = embedding dimension

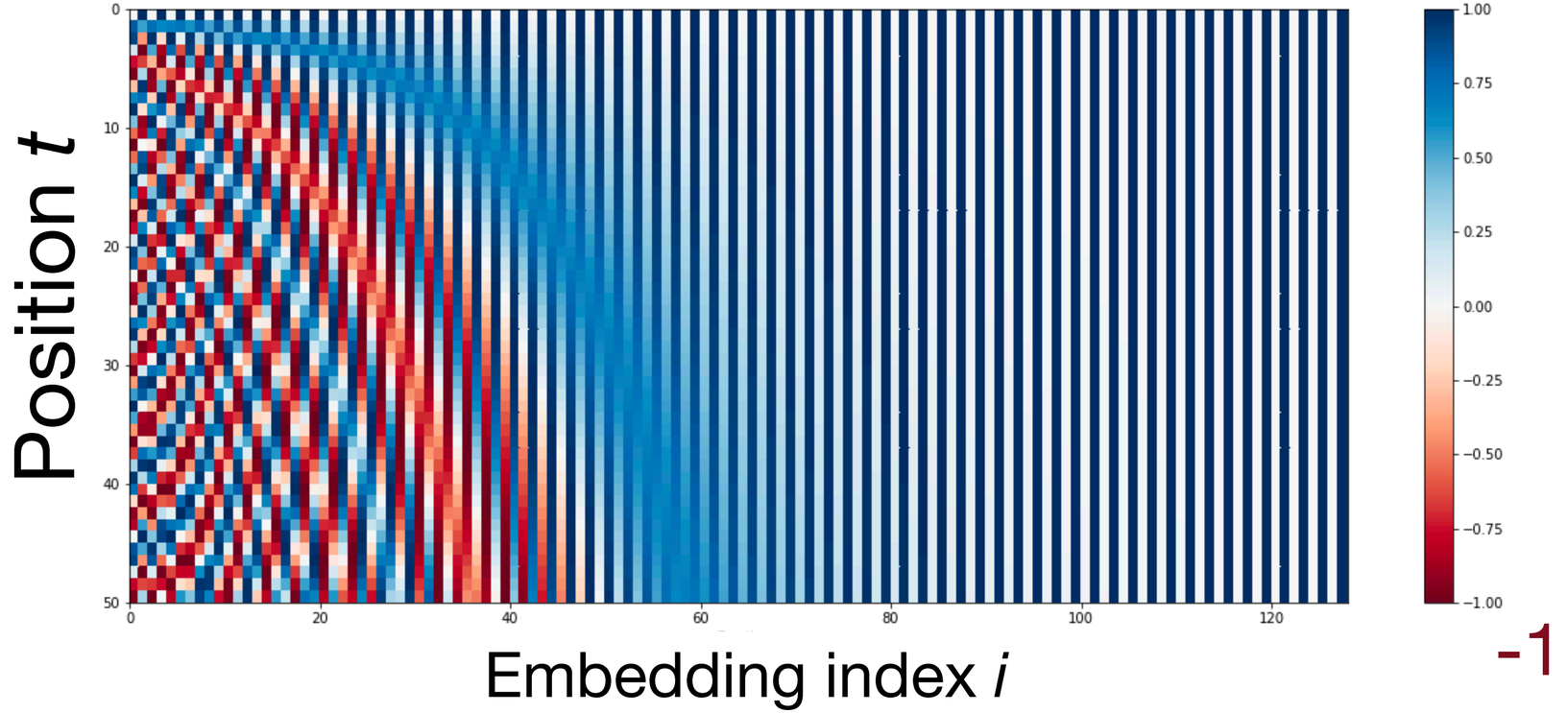
i = embedding index (0 to d-1)

Positional encoding

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

$$\omega_k = \frac{1}{10000^{2k/d}}$$

$$\overrightarrow{p_t} = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \\ \vdots \\ \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_{d \times 1}$$



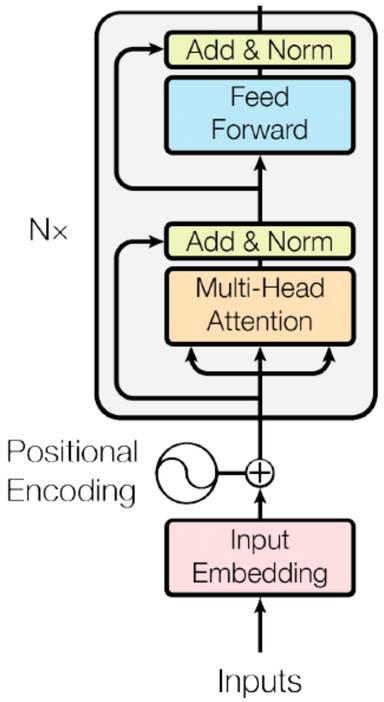
t = position

d = embedding dimension

i = embedding index (0 to d-1)

Transformer
Non-recurrent, deep model with attention

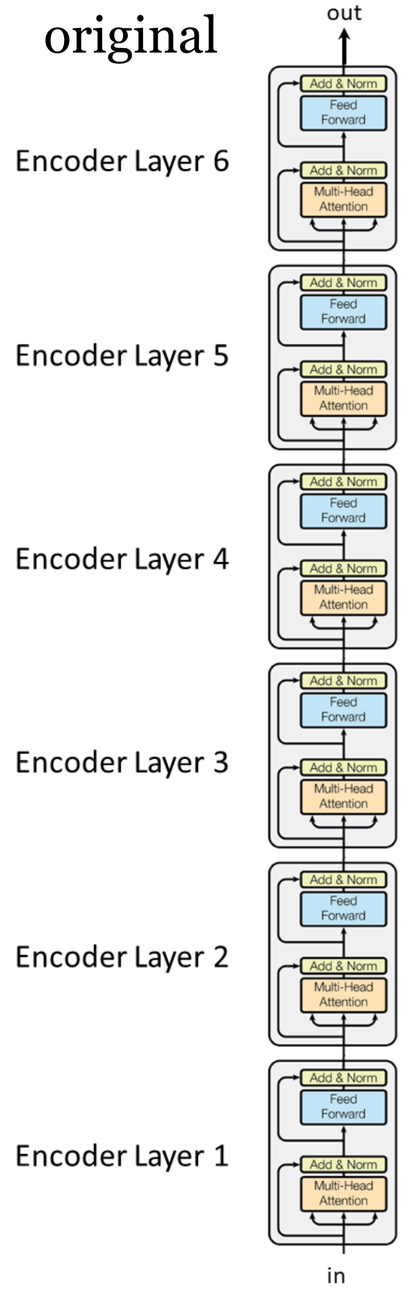
Transformer encoder



- Each Transformer block has two sub-layers
  - Multi-head attention
  - 2-layer feedforward NN (with ReLU)
- Each sublayer has a residual connection and a layer normalization

$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

- Input layer has a positional encoding
- Input embedding is byte pair encoding (BPE)

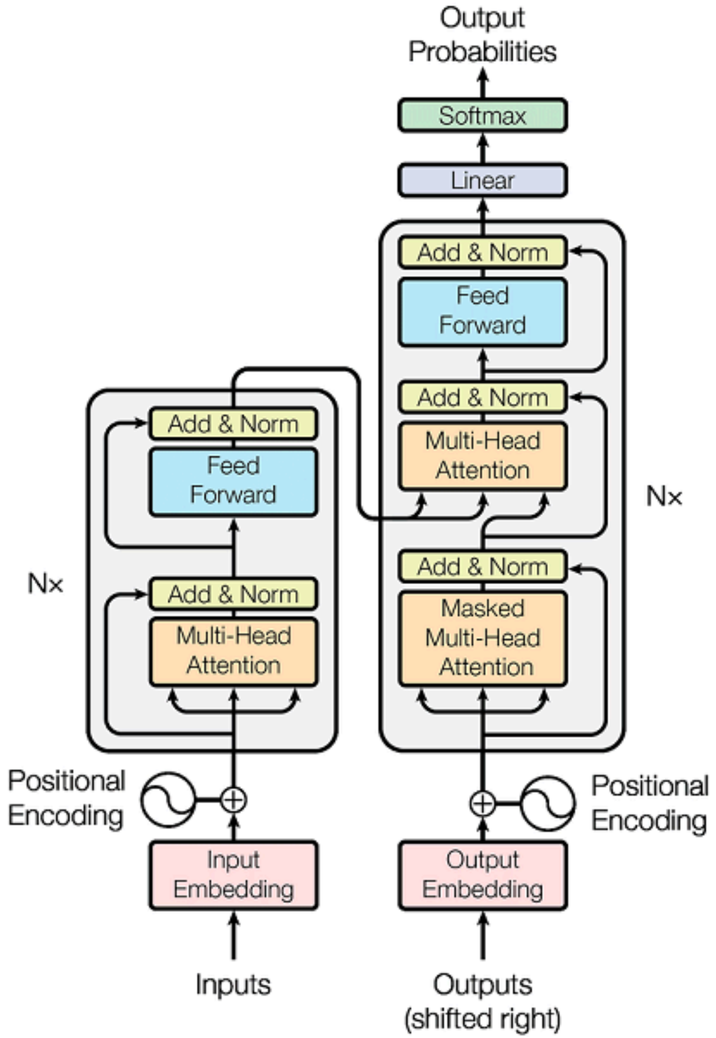


- BERT_base: 12 layers, 12 heads, hidden size = 768, 110M parameters
- BERT_large: 24 layers, 16 heads, hidden size = 1024, 340M parameters

(He et al, 2016): Residual connections

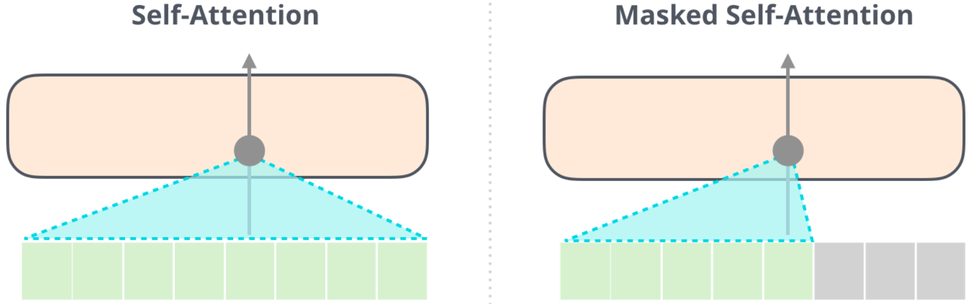
(Ba et al, 2016): Layer Normalization

Transformer decoder



- Encoder-Decoder Attention, where queries come from previous decoder layer and keys and values come from output of encoder

- Masked decoder self-attention on previously generated outputs



(figure credit: Jay Alammar http://jalammar.github.io/illustrated-gpt2/)

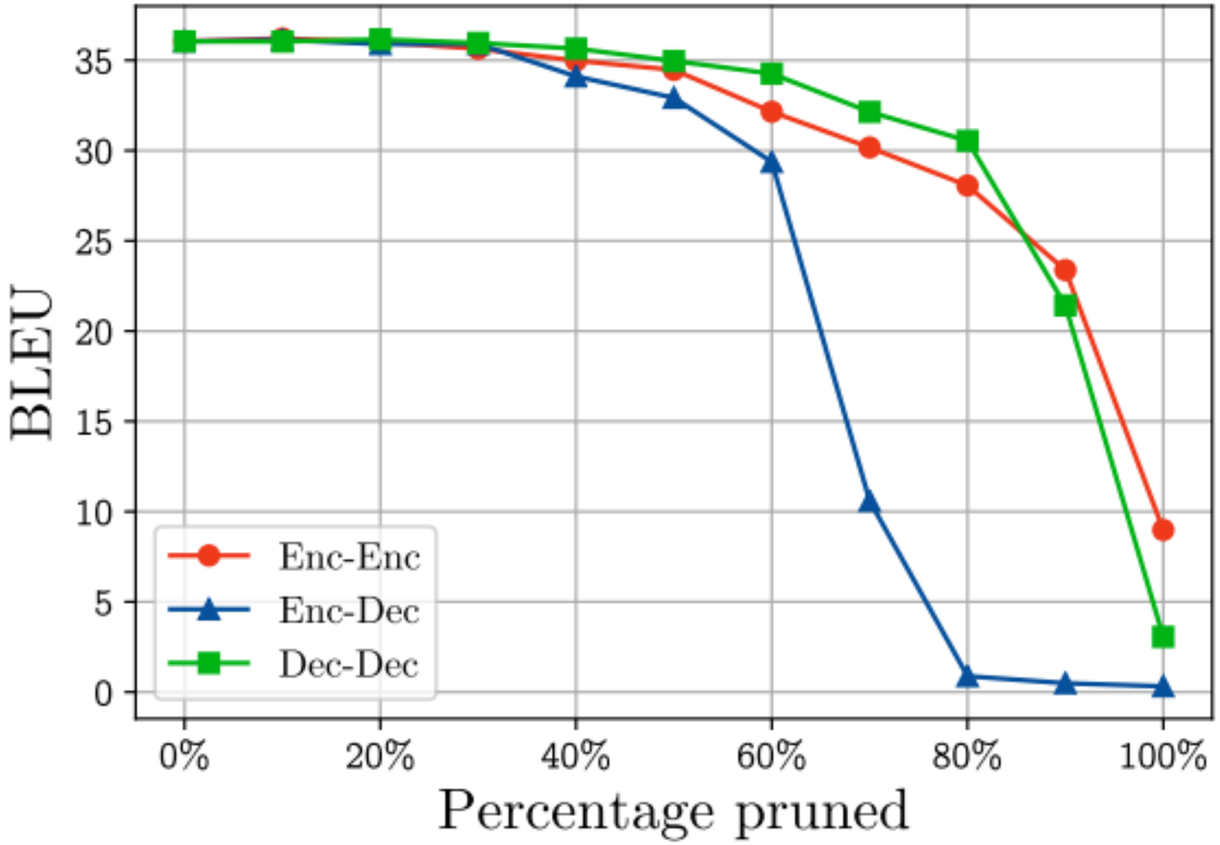
- also 6 layers (in original paper)

Do we need all these heads?

3 types of attention: Enc-Enc, Enc-Dec, Dec-Dec

6 layers, 16 heads each layer for each type

- Can we prune away some of the heads of a trained model during test time?



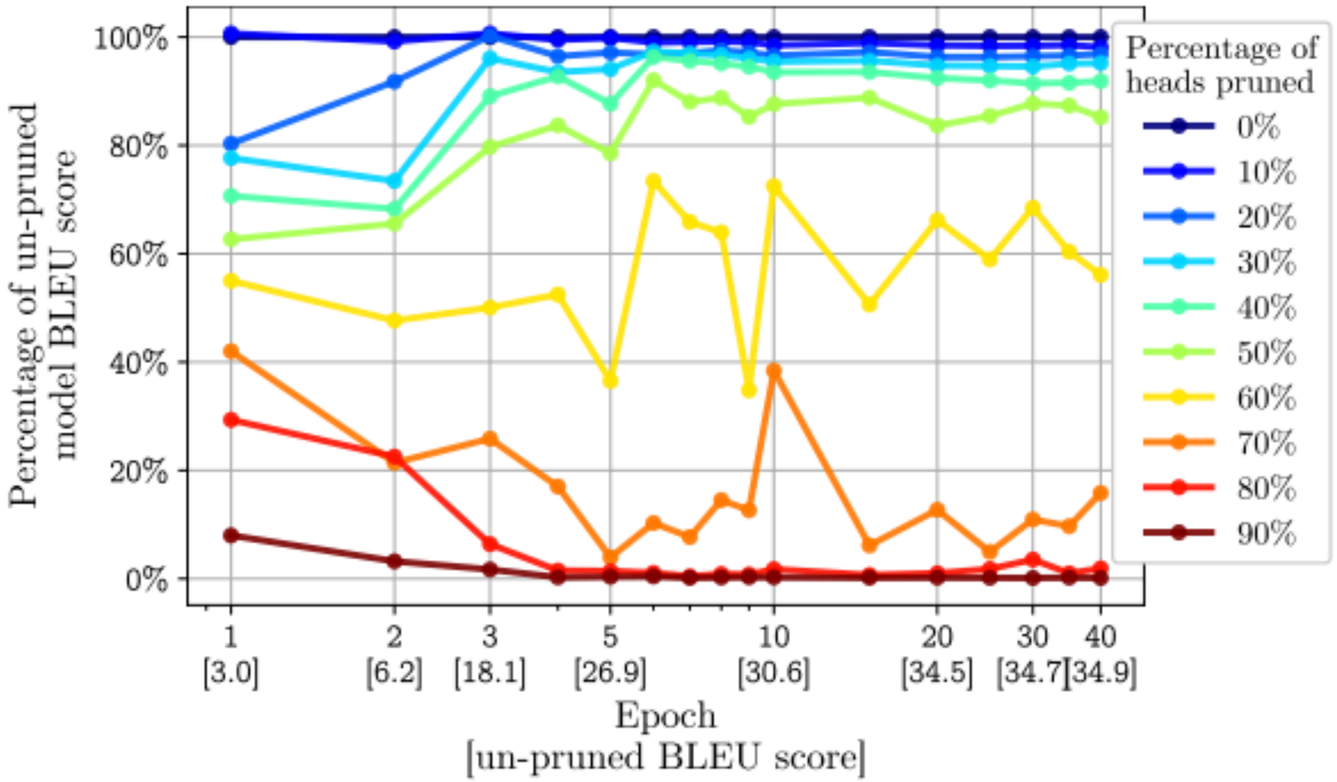
Are Sixteen Heads Really Better than One?
Michel, Levy, and Neubig, NeurIPS 2019

Do we need all these heads?

3 types of attention: Enc-Enc, Enc-Dec, Dec-Dec

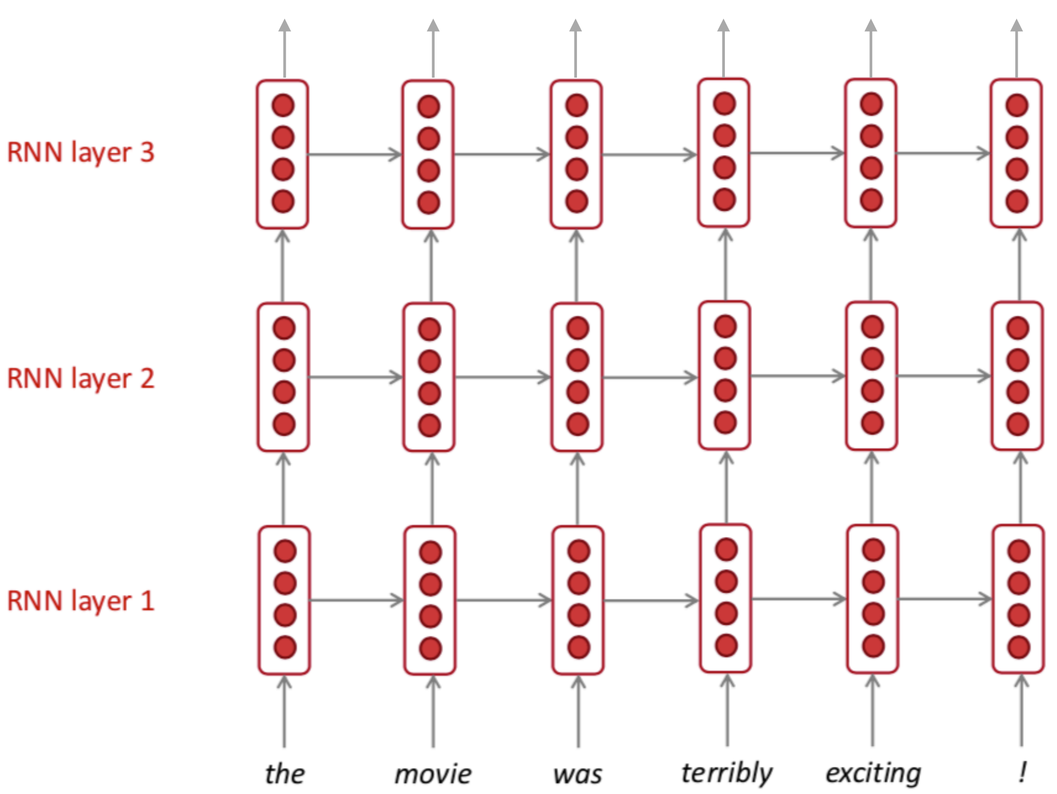
6 layers, 16 heads each layer for each type

- Can we train a good MT model with less heads?

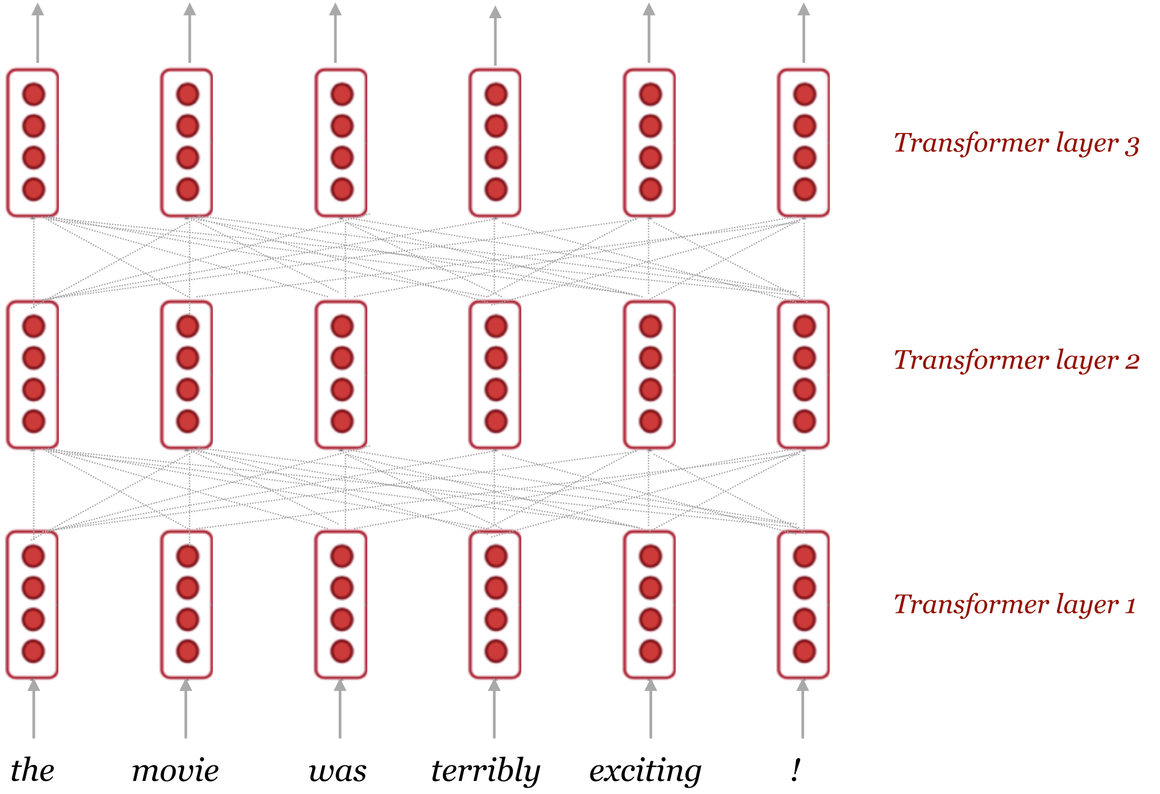


Are Sixteen Heads Really Better than One?
Michel, Levy, and Neubig, NeurIPS 2019

RNNs vs Transformers



RNN



Transformer

Useful Resources

Pytorch (https://pytorch.org/docs/stable/nn.html#transformer-layers)

nn.Transformer:

```
>>> transformer_model = nn.Transformer(nhead=16, num_encoder_layers=12)
>>> src = torch.rand((10, 32, 512))
>>> tgt = torch.rand((20, 32, 512))
>>> out = transformer_model(src, tgt)
```

nn.TransformerEncoder:

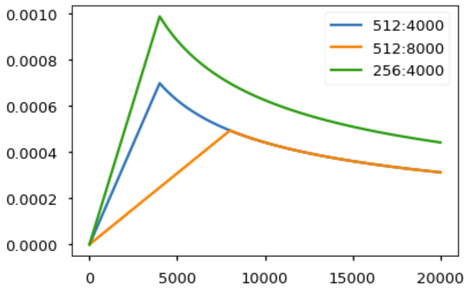
```
>>> encoder_layer = nn.TransformerEncoderLayer(d_model=512, nhead=8)
>>> transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=6)
>>> src = torch.rand(10, 32, 512)
>>> out = transformer_encoder(src)
```

🤗 **Transformers**

https://github.com/huggingface/transformers

Other details

- Learning rate with warmup and decay



- Label smoothing

The Annotated Transformer:

http://nlp.seas.harvard.edu/2018/04/03/attention.html

A Jupyter notebook which explains how Transformer works line by line in PyTorch!

Perfomance on machine translation

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

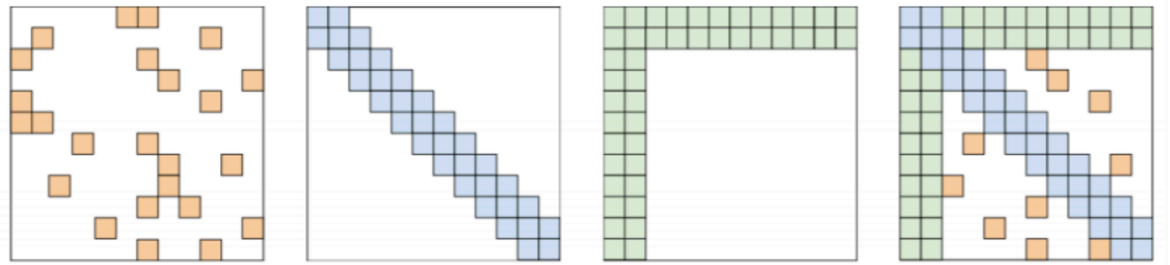
Attention is all you need
Vaswani et al, NeurIPS 2017

Transformer Pros and Cons

- Pros
  - Easier to capture dependencies: we draw attention between every pair of words
  - Easier to parallelize (matrix operations)

$$Q = XW^Q, W^Q \in \mathbb{R}^{d_1 \times d_q}$$
$$K = XW^K, W^K \in \mathbb{R}^{d_1 \times d_k}$$
$$V = XW^V, W^V \in \mathbb{R}^{d_1 \times d_v}$$

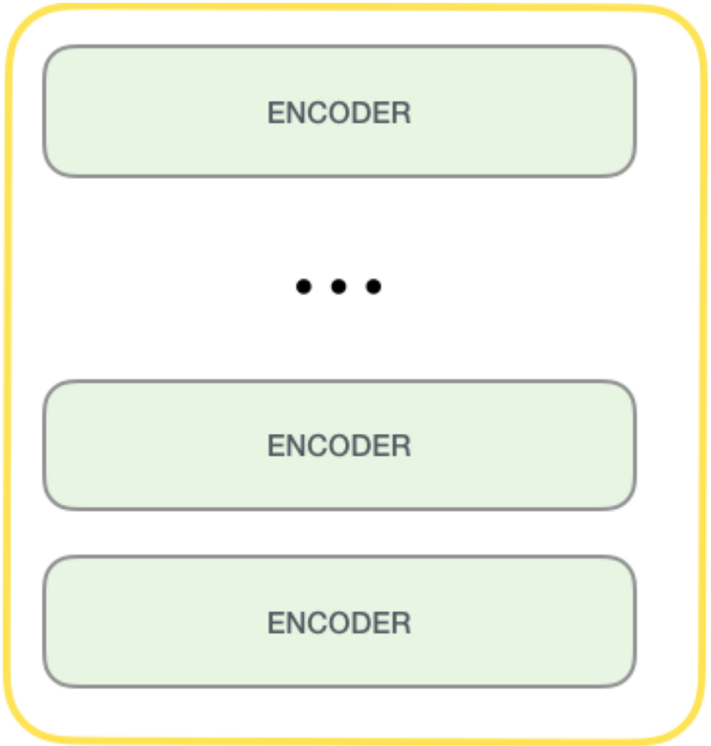
- Cons
  - Quadratic computation in self-attention
    - Can become very slow when the sequence length is large



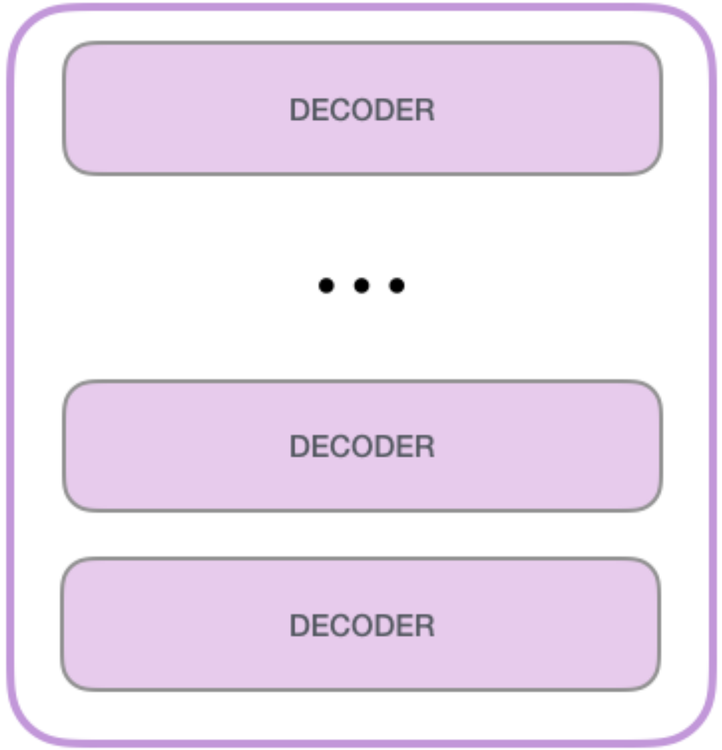
  - Are these positional representations enough to capture positional information?

Transformers blocks as building blocks

- BERT (built on Transformer encoders)



- GPT-2 (built on Transformer decoders)



(figure credit: Jay Alammar http://jalammar.github.io/illustrated-gpt2/)