Summer semester 2022

344.063 KV Special Topic:

Natural Language Processing with Deep Learning

Transformers

Navid Rekab-saz

navid.rekabsaz@jku.at

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computational Perception

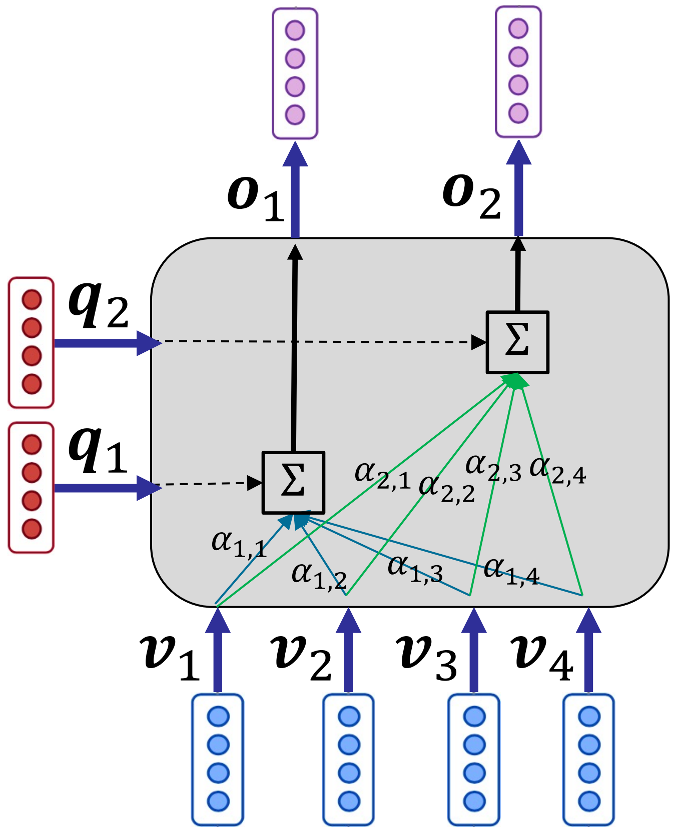
Agenda

- Transformers
  - Transformer encoder
  - Transformer decoder
- seq2seq with Transformers

Agenda

- **Transformers**
  - **Transformer encoder**
  - **Transformer decoder**
- seq2seq with Transformers

Attentions! – recap



$\alpha_{i,j}$ is the attention score of query $\boldsymbol{q}_i$ on value $\boldsymbol{v}_j$

$\boldsymbol{\alpha}_i$ is the vector of attentions of query $\boldsymbol{q}_i$ over value vectors $\boldsymbol{V}$ which forms a <u>probability distribution</u>

Attention Networks – recap

- Given query vector $\boldsymbol{q}_i$, an attention network uses the attention similarity function f to assign a non-normalized attention score $\tilde{\alpha}_{i,j}$ to value vector $\boldsymbol{v}_j$:

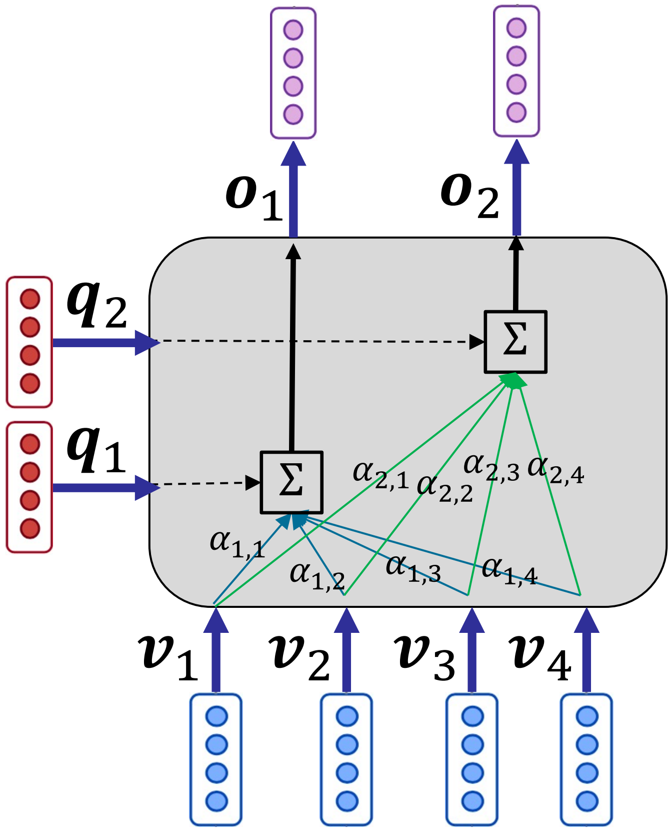
$$\tilde{\alpha}_{i,j} = f(\boldsymbol{q}_i, \boldsymbol{v}_j)$$

- Then, the attention scores over values are turned to a probability distribution using softmax:

$$\boldsymbol{\alpha}_i = \text{softmax}(\tilde{\boldsymbol{\alpha}}_i), \qquad \sum_{j=1}^{|\boldsymbol{V}|} \alpha_{i,j} = 1$$

- Finally, output vector $\boldsymbol{o}_i$ regarding query $\boldsymbol{q}_i$ is defined as the sum of the value vectors weighted by their corresponding attentions:

$$\boldsymbol{o}_i = \sum_{j=1}^{|\boldsymbol{V}|} \alpha_{i,j} \boldsymbol{v}_j$$

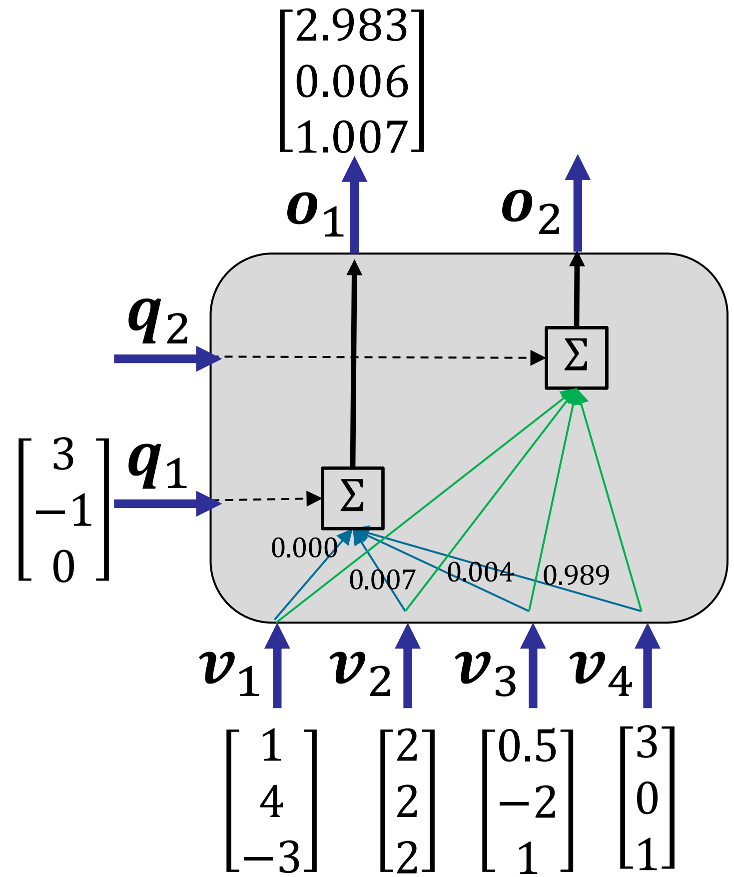


Example – recap

$$\tilde{\boldsymbol{\alpha}}_1 = \begin{bmatrix} \boldsymbol{q}_1\boldsymbol{v}_1^{\mathrm{T}} = -1 \\ \boldsymbol{q}_1\boldsymbol{v}_2^{\mathrm{T}} = 4 \\ \boldsymbol{q}_1\boldsymbol{v}_3^{\mathrm{T}} = 3.5 \\ \boldsymbol{q}_1\boldsymbol{v}_4^{\mathrm{T}} = 9 \end{bmatrix} \rightarrow \boldsymbol{\alpha}_1 = \begin{bmatrix} 0.000 \\ 0.007 \\ 0.004 \\ 0.989 \end{bmatrix}$$

$$\boldsymbol{o}_1 = 0.000 \begin{bmatrix} 1 \\ 4 \\ -3 \end{bmatrix} + 0.007 \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} + 0.004 \begin{bmatrix} 0.5 \\ -2 \\ 1 \end{bmatrix} + 0.989 \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}$$

$$\boldsymbol{o}_1 = \begin{bmatrix} 2.983 \\ 0.006 \\ 1.007 \end{bmatrix}$$

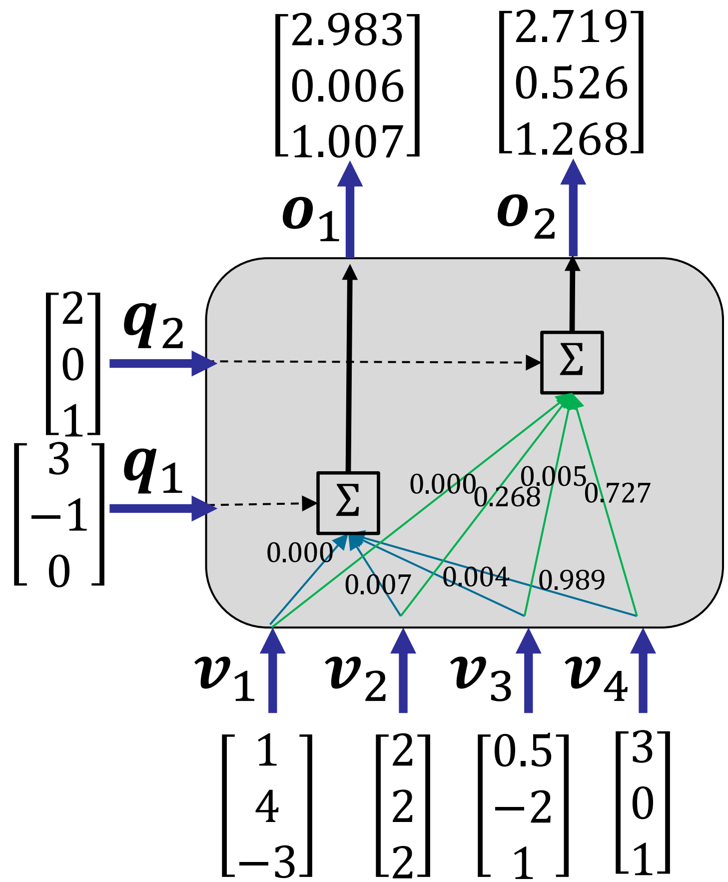


Example – recap

$$\tilde{\boldsymbol{\alpha}}_2 = \begin{bmatrix} \boldsymbol{q}_2\boldsymbol{v}_1^{\mathrm{T}} = -1 \\ \boldsymbol{q}_2\boldsymbol{v}_2^{\mathrm{T}} = 6 \\ \boldsymbol{q}_2\boldsymbol{v}_3^{\mathrm{T}} = 2 \\ \boldsymbol{q}_2\boldsymbol{v}_4^{\mathrm{T}} = 7 \end{bmatrix} \rightarrow \quad \boldsymbol{\alpha}_2 = \begin{bmatrix} 0.000 \\ 0.268 \\ 0.005 \\ 0.727 \end{bmatrix}$$

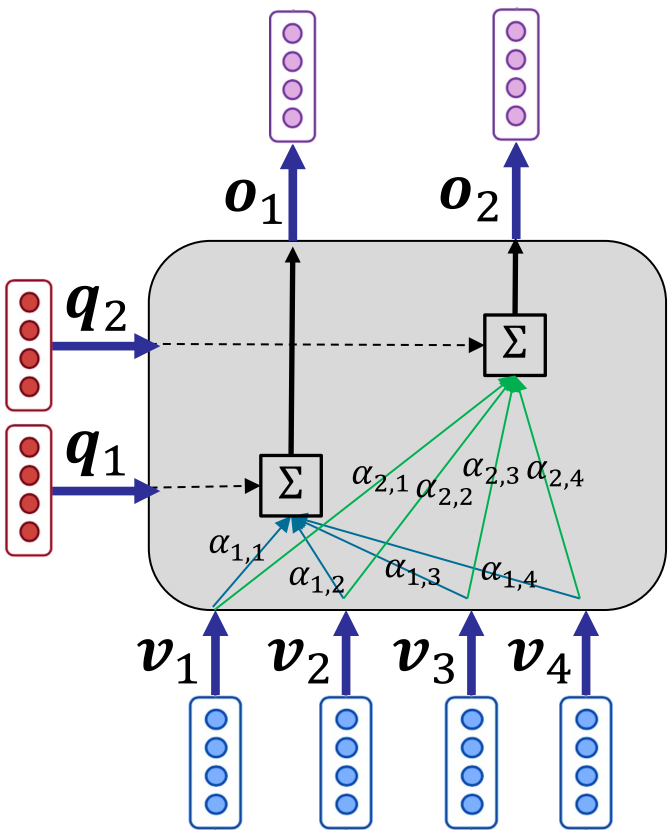
$$\boldsymbol{o}_2 = 0.000 \begin{bmatrix} 1 \\ 4 \\ -3 \end{bmatrix} + 0.268 \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} + 0.005 \begin{bmatrix} 0.5 \\ -2 \\ 1 \end{bmatrix} + 0.727 \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}$$

$$\boldsymbol{o}_2 = \begin{bmatrix} 2.719 \\ 0.526 \\ 1.268 \end{bmatrix}$$



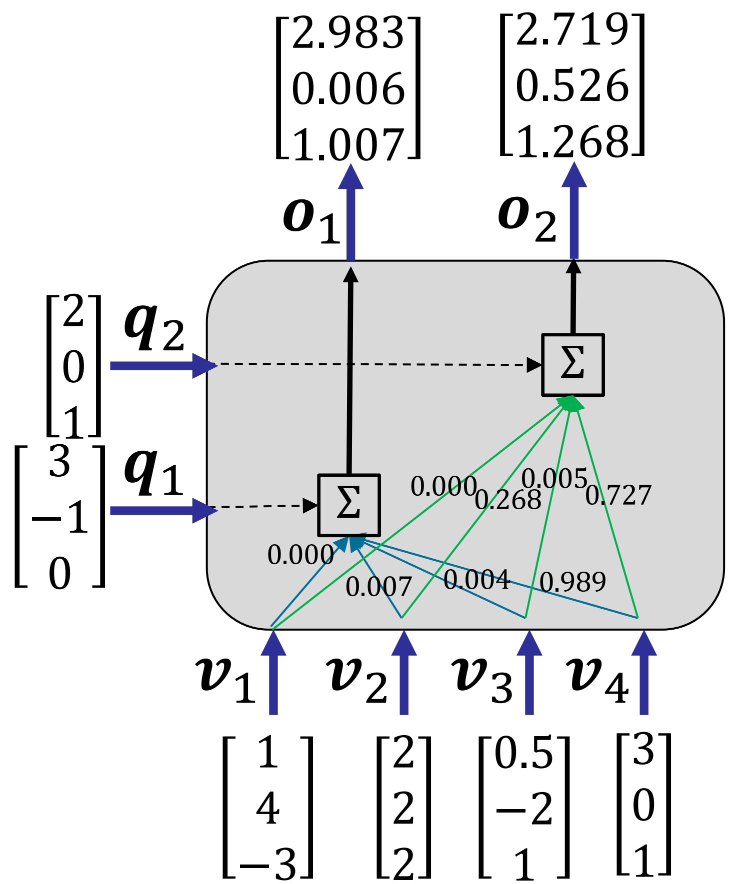
Attention table

	$\boldsymbol{v}_1$	$\boldsymbol{v}_2$	$\boldsymbol{v}_3$	$\boldsymbol{v}_4$
$\boldsymbol{q}_1$	$\alpha_{1,1}$	$\alpha_{1,2}$	$\alpha_{1,3}$	$\alpha_{1,4}$
$\boldsymbol{q}_2$	$\alpha_{2,1}$	$\alpha_{2,2}$	$\alpha_{2,3}$	$\alpha_{2,4}$



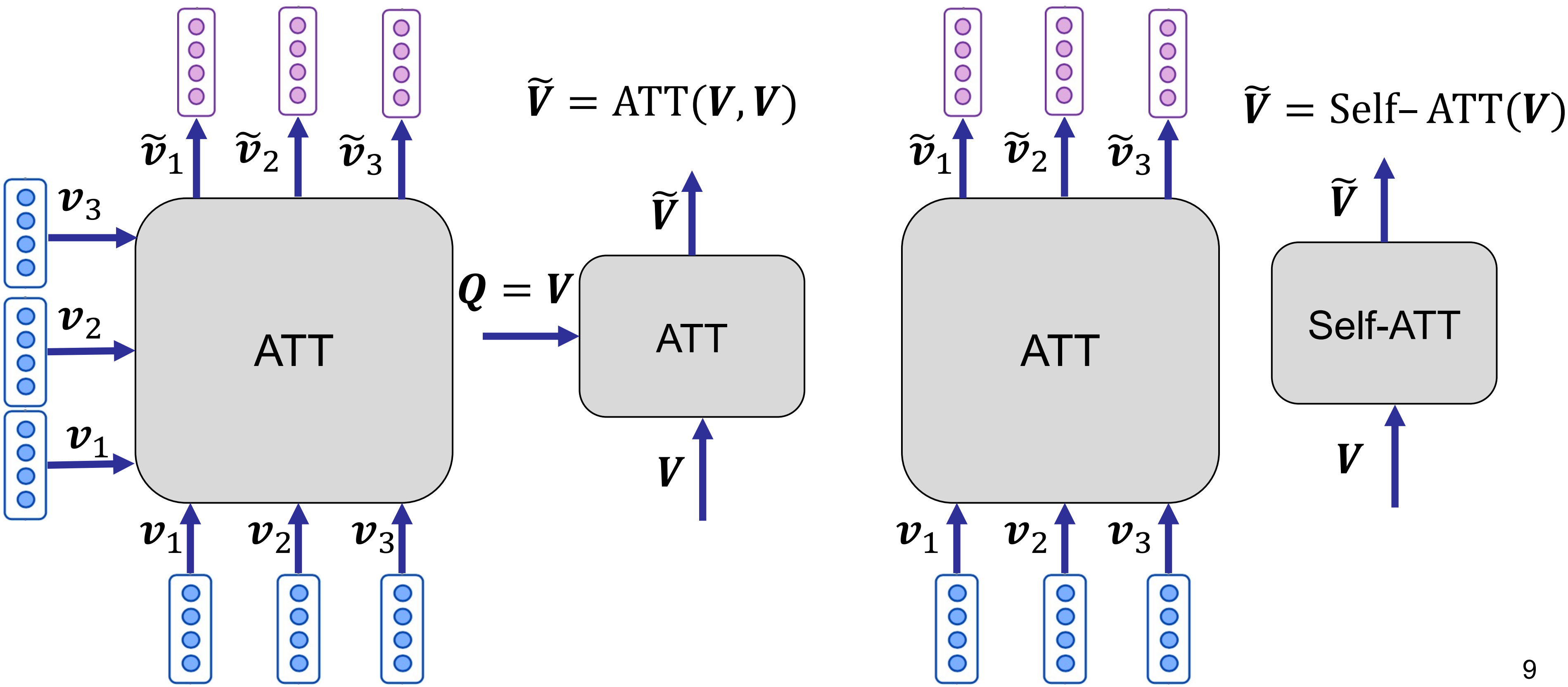
In the example:

	$\boldsymbol{v}_1$	$\boldsymbol{v}_2$	$\boldsymbol{v}_3$	$\boldsymbol{v}_4$
$\boldsymbol{q}_1$	0.000	0.007	0.004	0.989
$\boldsymbol{q}_2$	0.000	0.268	0.005	0.727



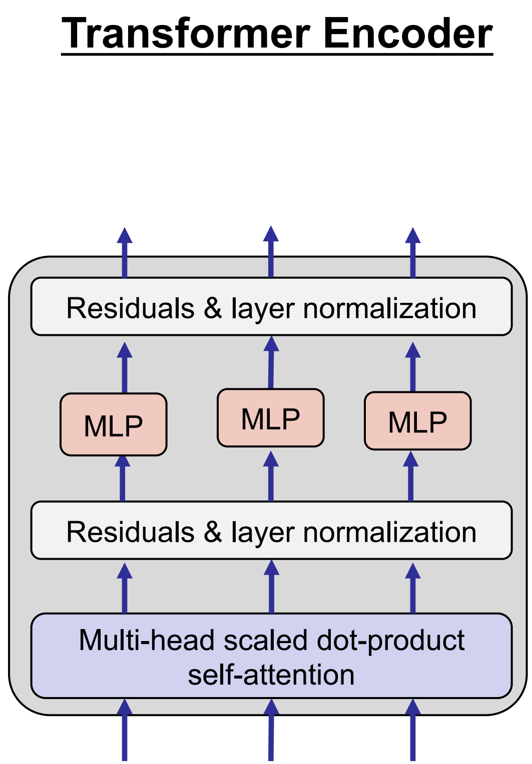
Self-attention

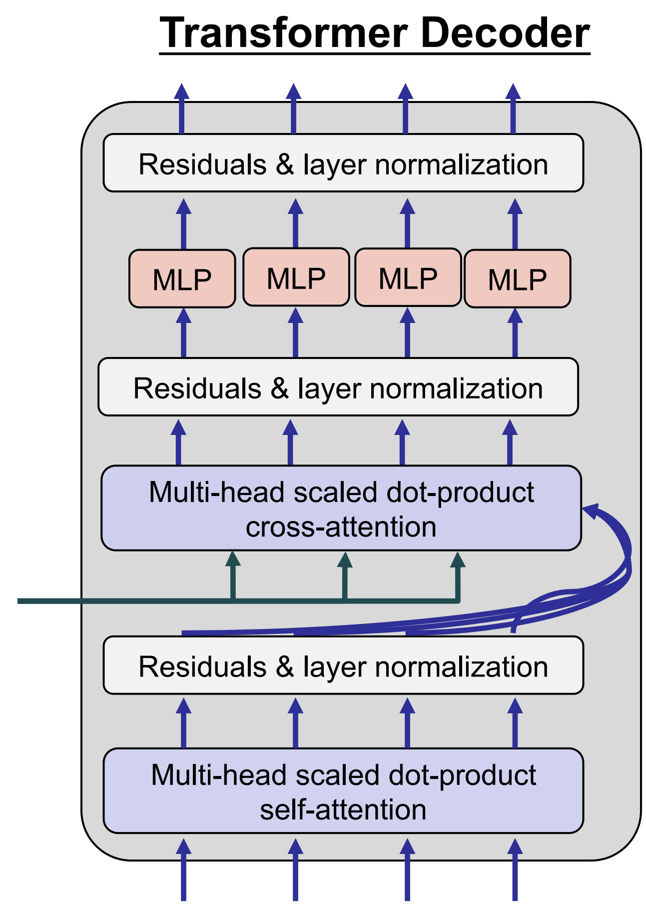
- Self-attention is when the values are also given as the queries: $\boldsymbol{Q} = \boldsymbol{V}$
- Self-attention encodes a sequence $\boldsymbol{V}$ to a contextualized sequence $\widetilde{\boldsymbol{V}}$
  - In self-attention, each input vector $\boldsymbol{v}_i$ attends to all other input vectors $\boldsymbol{V}$, and outputs $\widetilde{\boldsymbol{v}}_i$ as a composition of input vectors
  - Output vector $\widetilde{\boldsymbol{v}}_i$ is the contextual embedding of the input vector $\boldsymbol{v}_i$



Transformers

- Attention network with DL best practices!
  - Originally introduced in the context of machine translation and is now widely adopted for sequence encoding and decoding

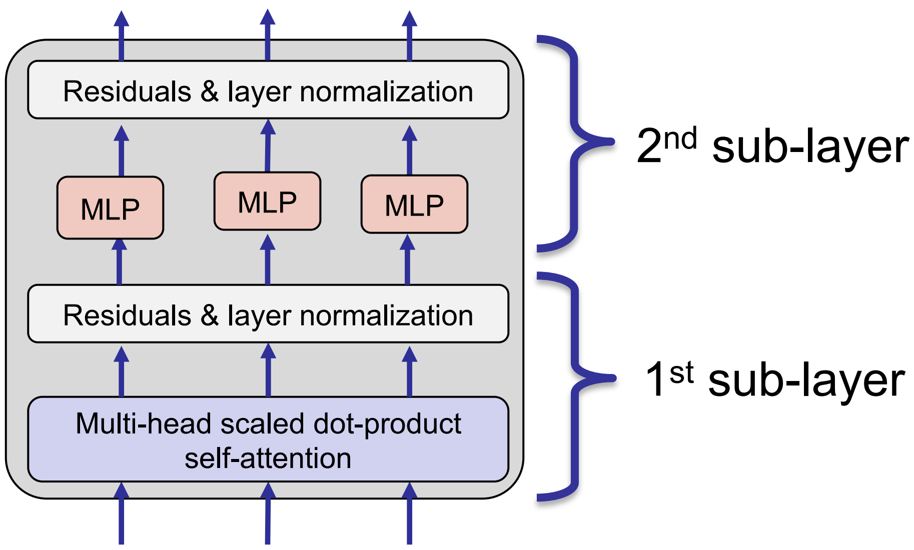




Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

Transformer Encoder

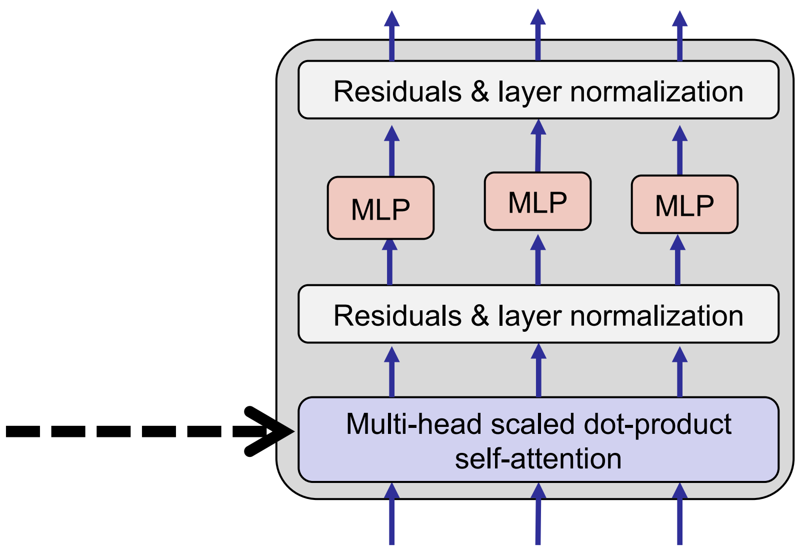
- Transformer Encoder consists of two sub-layers:
  - 1st : Multi-head scaled dot-product self-attention
  - 2nd : Position-wise multi-layer perceptron (feed forward)
- Each sub-layer is followed by residual networks and layer normalization
  - Drop-outs are applied after each computation



Transformer Encoder

Let's start from multi-head scaled dot-product self-attention:

1. Scaled dot-product attention
2. Multi-head attention
3. self-attention

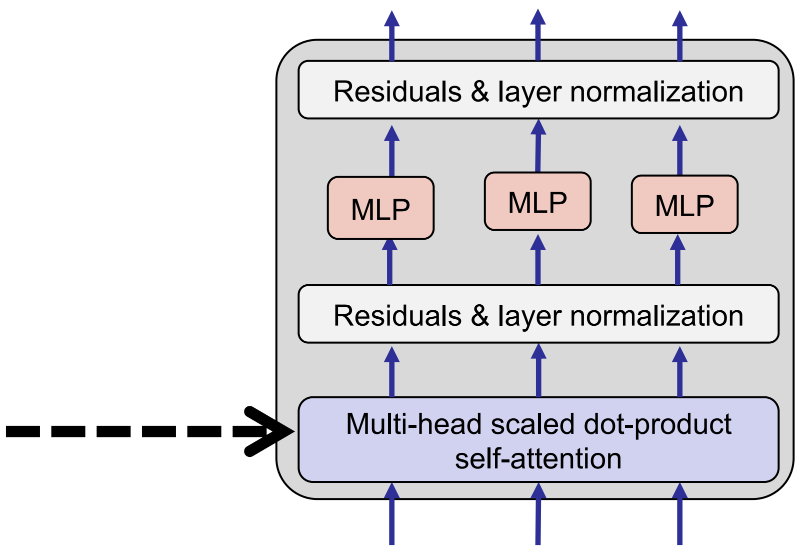


Transformer Encoder

Transformer Encoder

Let's start from multi-head scaled dot-product self-attention:

1. **Scaled dot-product attention**
2. Multi-head attention
3. self-attention



Transformer Encoder

Basic dot-product attention – recap

- Non-normalized attention scores:

$$\tilde{\alpha}_{i,j} = f(\boldsymbol{q}_i, \boldsymbol{v}_j)$$

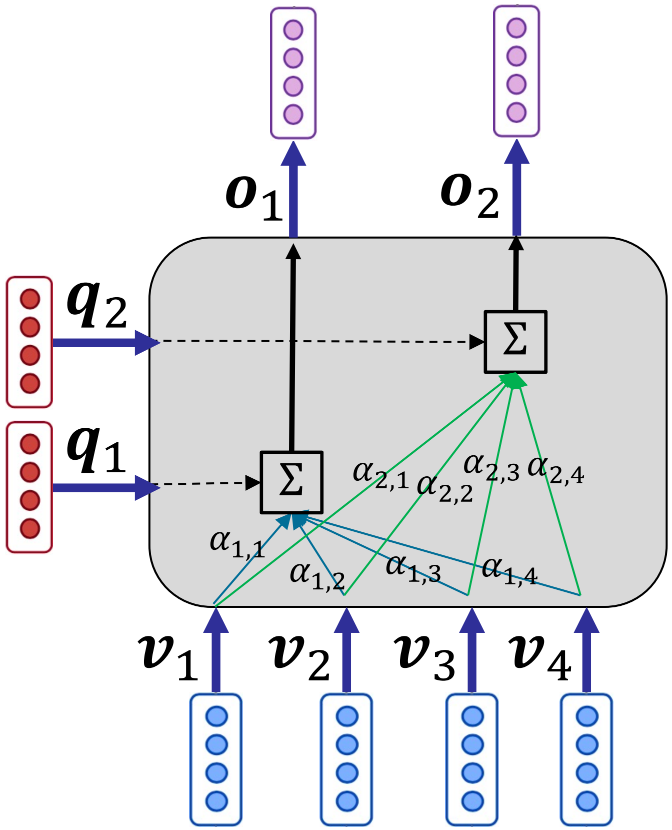
$$\tilde{\alpha}_{i,j} = \boldsymbol{q}_i \boldsymbol{v}_i^{\mathrm{T}}$$

  - In this case, $d_q = d_v$
  - Attention network has no parameter to learn!

- Softmax over value vectors:

$$\boldsymbol{\alpha}_i = \mathrm{softmax}(\tilde{\boldsymbol{\alpha}}_i)$$

- Output (weighted sum): $\boldsymbol{o}_i = \sum_{j=1}^{|\boldsymbol{V}|} \alpha_{i,j} \boldsymbol{v}_j$



Scaled dot-product attention

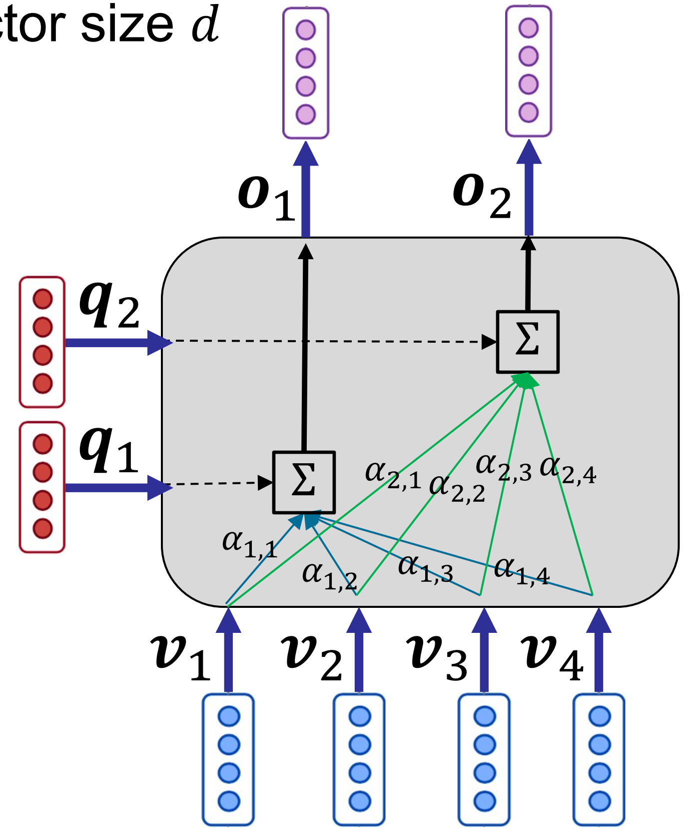
- Problem with basic dot-product attention:
  - As d gets large, the variance of $\tilde{\alpha}_{i,j}$ increases …
  - … this makes softmax very peaked for some values of $\tilde{\boldsymbol{\alpha}}_i$ …
  - … and hence its gradient gets smaller
- One approach: normalize/scale $\tilde{\alpha}_{i,j}$ by vector size d

<u>**Scaled dot-product attention**</u>

- Non-normalized attention scores:

$$\tilde{\alpha}_{i,j} = \frac{\boldsymbol{q}_i \boldsymbol{v}_j^{\mathrm{T}}}{\sqrt{d}}$$

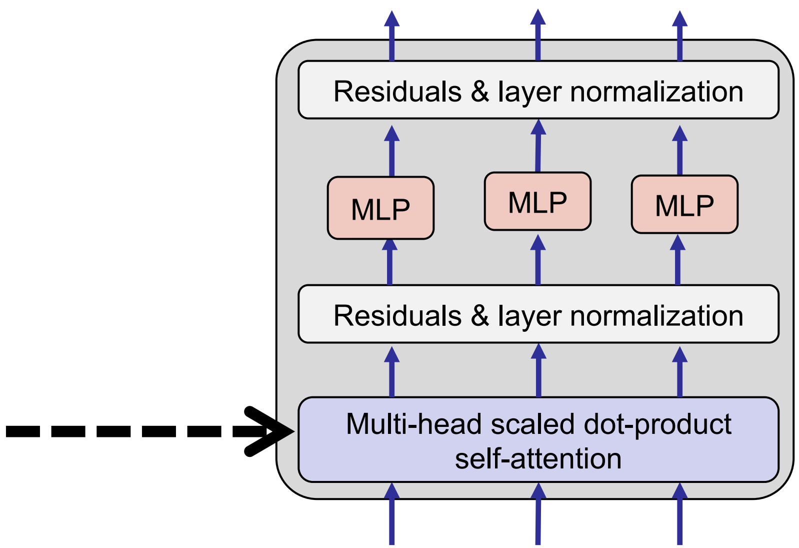
- Softmax over values: $\boldsymbol{\alpha}_i = \mathrm{softmax}(\tilde{\boldsymbol{\alpha}}_i)$
- Output: $\boldsymbol{o}_i = \sum_{j=1}^{|V|} \alpha_{i,j} \boldsymbol{v}_j$



Transformer Encoder

Let's start from multi-head scaled dot-product self-attention:

1. Scaled dot-product attention
2. **Multi-head attention**
3. self-attention



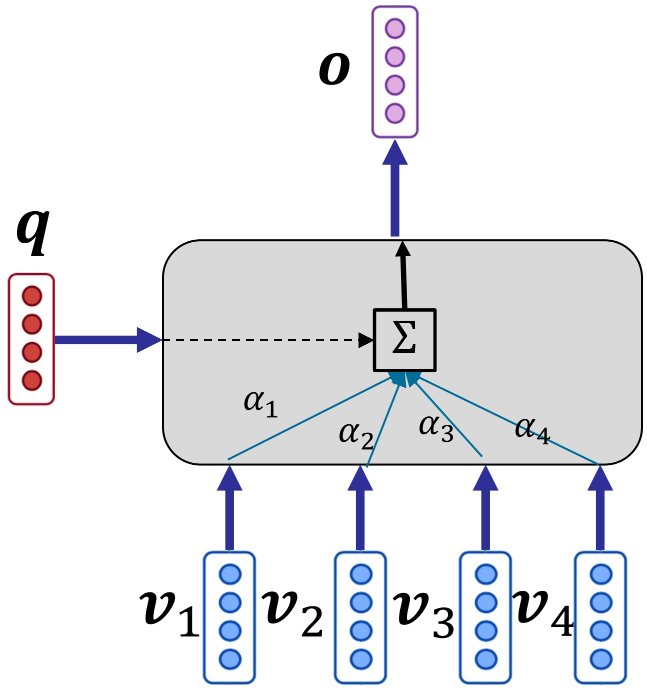
Transformer Encoder

Softmax bottleneck!

- Softmax is applied to non-normalized attention vectors
  - Recall: softmax makes the maximum value much higher than the other

$$\boldsymbol{z} = [1 \quad 2 \quad 5 \quad 6] \rightarrow \text{softmax}(\boldsymbol{z}) = [0.004 \quad 0.013 \quad 0.264 \quad 0.717]$$

- Common in language, a word may be related to several other words in a sequence, each through a specific concept
  - Like the relations of a verb to its subject and object
- However, normal (single-head) attention network aggregates all concepts in one set
- In this case, due to softmax, value vectors must compete for the attention of query vector → softmax bottleneck



Multi-head attention

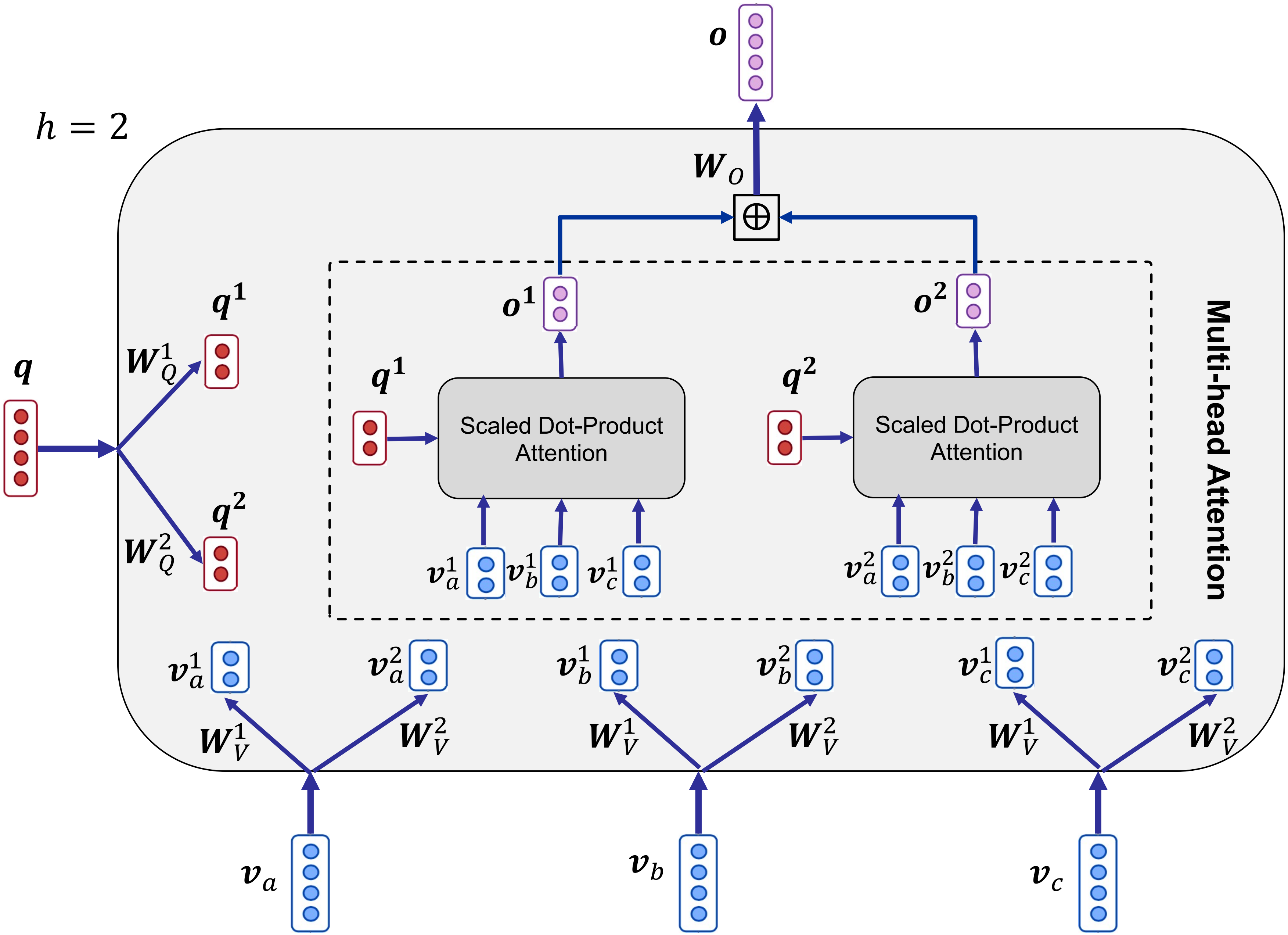
- Multi-head attention approaches *softmax bottleneck* by calculating multiple sets of attentions between a query and values

Multi-head attention:

1. Transfer each query/value vector to h query/value subspaces, each called a head
2. In each subspace, apply a normal (single-head) attention network using the queries and values transferred to the subspace to achieve the output vectors of that head
3. Concatenate the output vectors of all heads in respect to a query to achieve the final output of the query

- In multi-head attention, each head (and each subspace) can specialize on capturing a specific kind of relation

Multi-head attention



Multi-head attention – formulation

- Transfer every query $\boldsymbol{q}_i$ to h vectors, each with size ${}^{d}/_{h}$:

size: ${}^{d}/_{h}$

$$\boldsymbol{q}_i^1 = \boldsymbol{q}_i \boldsymbol{W}_Q^1 \quad \dots \quad \boldsymbol{q}_i^h = \boldsymbol{q}_i \boldsymbol{W}_Q^h$$

Matrix size: $d \times {}^{d}/_{h}$

- Transfer every value $\boldsymbol{v}_j$ to h vectors, each with size ${}^{d}/_{h}$:

size: ${}^{d}/_{h}$

$$\boldsymbol{v}_j^1 = \boldsymbol{v}_j \boldsymbol{W}_V^1 \quad \dots \quad \boldsymbol{v}_j^h = \boldsymbol{v}_j \boldsymbol{W}_V^h$$

Matrix size: $d \times {}^{d}/_{h}$

- Calculate outputs of subspaces corresponding to $\boldsymbol{q}_i$:

size: ${}^{d}/_{h}$

$$\boldsymbol{o}_i^1 = \mathrm{ATT}\left(\boldsymbol{q}_i^1, \boldsymbol{V}^1\right) \quad \dots \quad \boldsymbol{o}_i^h = \mathrm{ATT}\left(\boldsymbol{q}_i^h, \boldsymbol{V}^h\right)$$

- Concatenate outputs of subspaces for $\boldsymbol{q}_i$ as its final output:

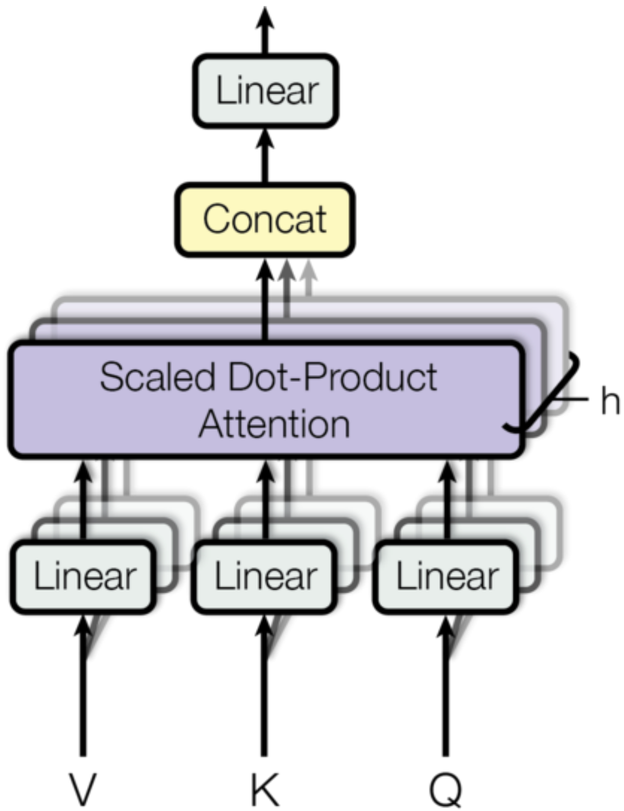
size: d

$$\boldsymbol{o}_i = \boldsymbol{W}_O \left[\boldsymbol{o}_i^1; \dots; \boldsymbol{o}_i^h\right]$$

Size: $d \times d$
This matrix linearly combines the dimensions of the concatenated vectors

Parameters are shown in red

Multi-head attention – graphic in original paper



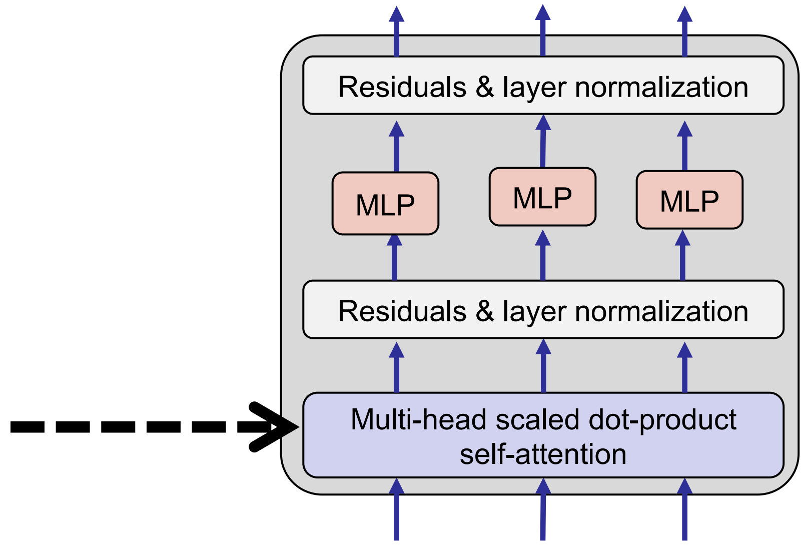
- Default number of heads in Transformers: $h = 8$
- Recall: Attentions (and Transformers) in fact have three inputs (not two), namely queries, keys, and values.
  - Keys are used to calculate attentions
  - Values are used to produce outputs

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

Transformer Encoder

Let's start from multi-head scaled dot-product self-attention:

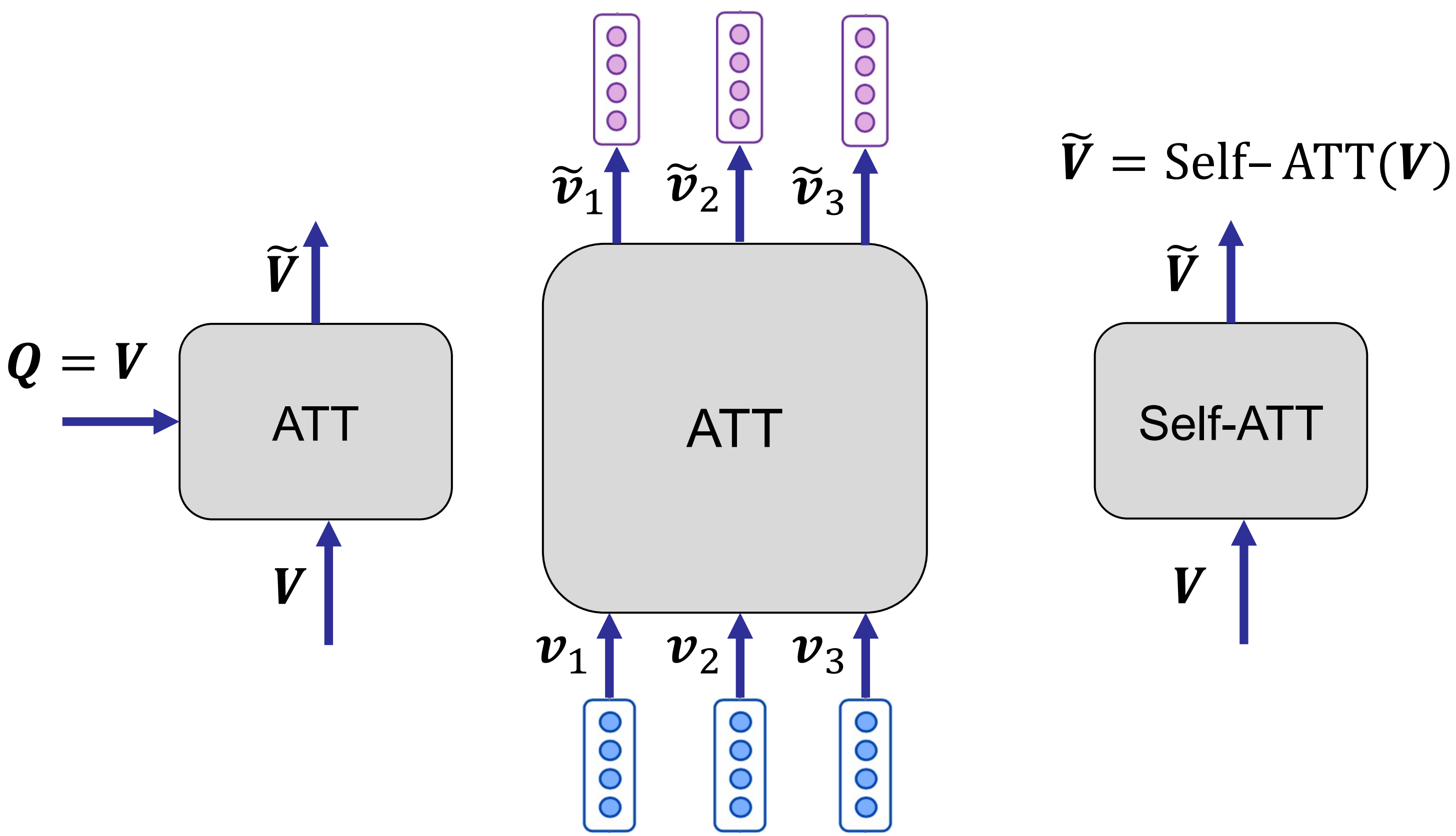
1. Scaled dot-product attention
2. Multi-head attention
3. **Self-attention**



Transformer Encoder

Self-attention (recap)

- Values are the same as queries
- Each output vector is the contextual embedding of the corresponding input vector
  - $\tilde{\boldsymbol{v}}_i$ is the contextual embedding of $\boldsymbol{v}_i$

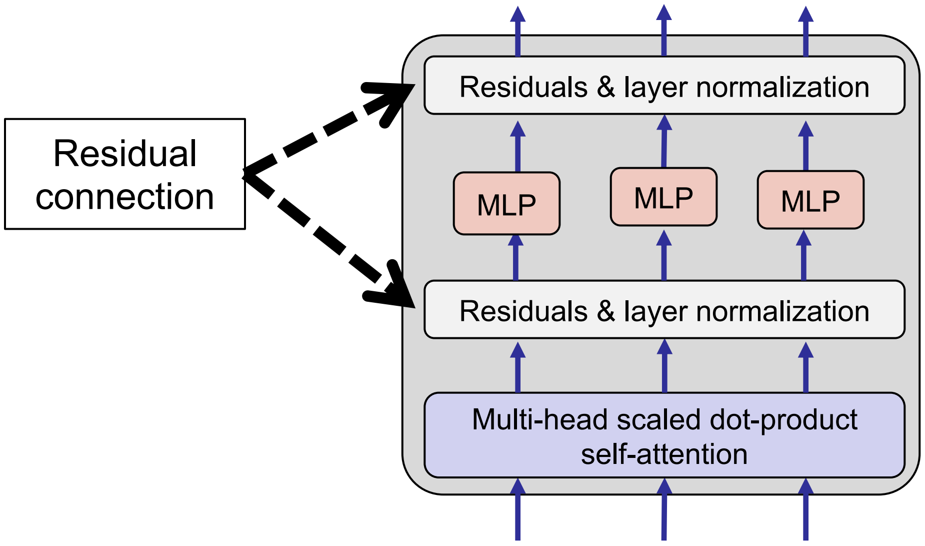


Residuals

- Residual (short-cut) connection:

$$\text{output} = f(x) + x$$

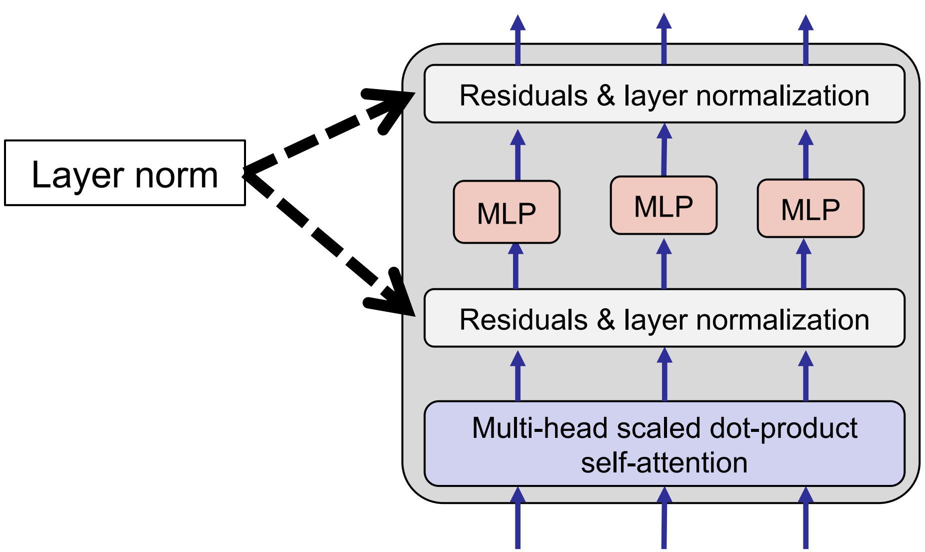
- Learn in detail:
  - He, Kaiming; Zhang, Xiangyu; Ren, Shaoqing; Sun, Jian (2016). "Deep Residual Learning for Image Recognition" . In proc. of CVPR
  - Srivastava, Rupesh Kumar; Greff, Klaus; Schmidhuber, Jürgen (2015). "Highway Networks". https://arxiv.org/pdf/1505.00387.pdf



Transformer Encoder

Layer normalization

- Layer normalization changes the activations of each vector to have mean 0 and variance 1 …
  - … and learns two parameters per layer to shift the mean and variance

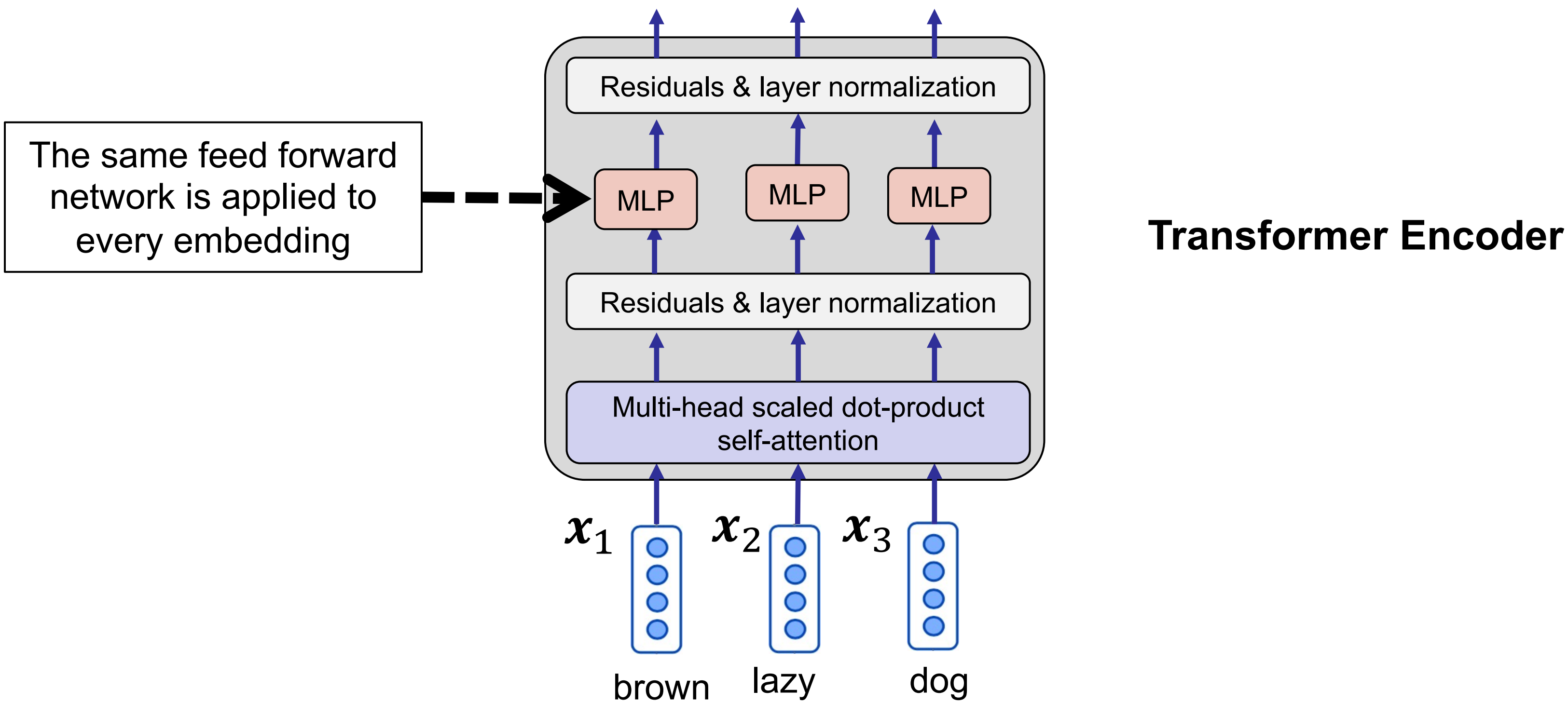


Transformer Encoder

Paper: https://arxiv.org/pdf/1607.06450.pdf

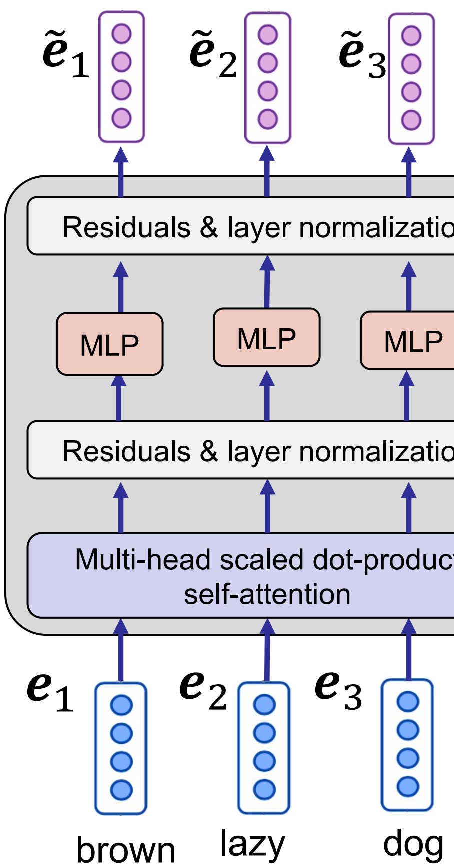
Multi-layer perceptron on embedding

- A two-layer multi-layer perceptron (with ReLU) is applied to each output embedding
  - This layer provides the capacity for a non-linear transformation over each (contextualized) embedding



Transformer Encoder – all together

- Transformer Encoder receive input embeddings and outputs the corresponding contextualized embeddings
  - Processing all inputs happen at the same time → non auto-regressive



Transformer Encoder – summary

- A self-attention model using
  - multi-head scaled dot-product attention
  - followed by the same feed-forward layer applied to each embedding
  - all packed with residuals, layer norms, and dropouts

Transformers as in attentions …

- do not have locality (position) bias
  - A long-distance context has "equal opportunity"
- process all the input together with a single computation per each layer
  - Friendly with parallel computations in GPU

Learn more and study the PyTorch implementation: http://nlp.seas.harvard.edu/2018/04/03/attention.html

Position embeddings

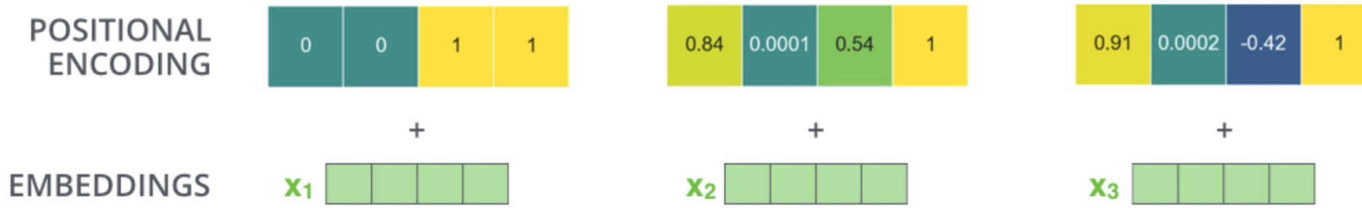
- Transformers are agnostic to the position of tokens
  - A context token in long-distance has the same effect as the one in short-distance (no *locality bias*)
- However, the positions of tokens in a sequence might be informative and important in some tasks

Position embeddings – a common approach in Transformers:

- Create embeddings representing positions in a sequence, and add the values of the position embeddings to the token embeddings at corresponding positions
  - Position embedding is usually created using a sine/cosine function
    - It can also be learned end-to-end with the model parameters
  - Using position embeddings, the same token at different positions of a sequence will have different final representations

Position embeddings – examples

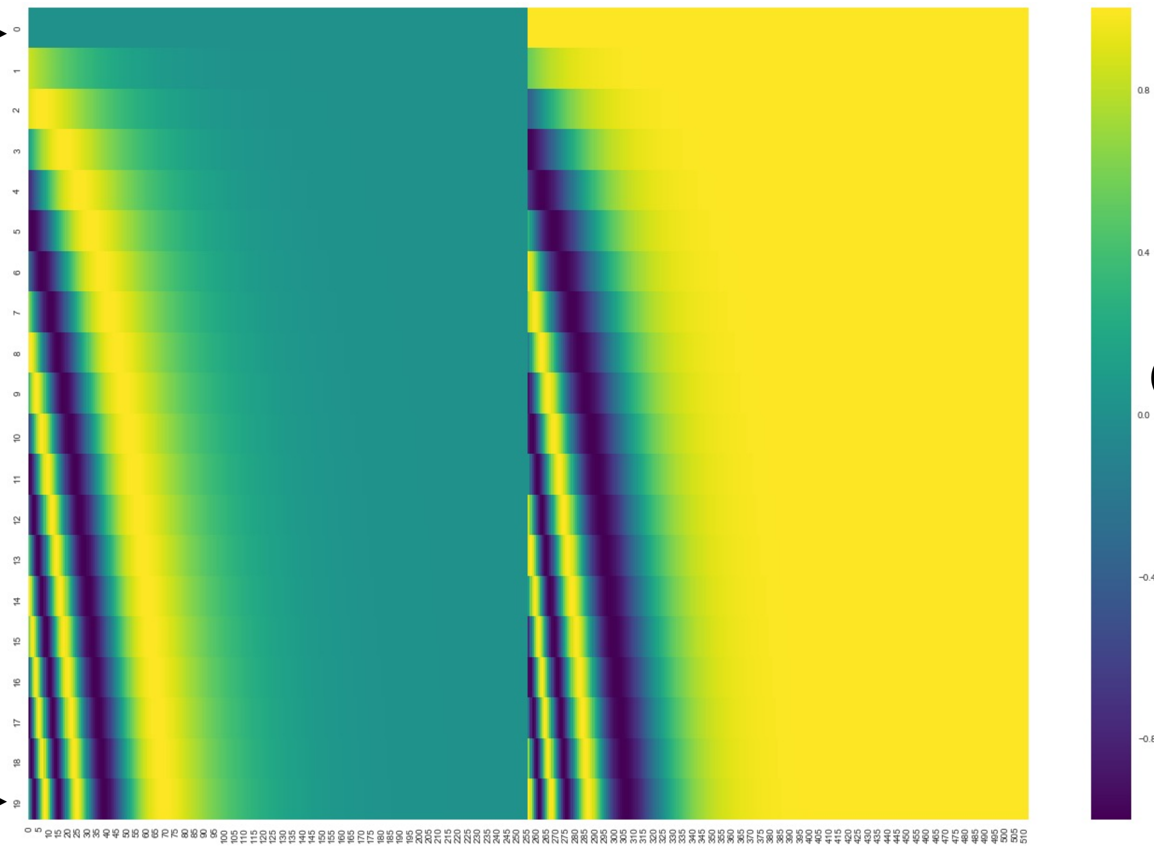
An example of embeddings with four dimensions:



Position embedding for location 0

Position embeddings

Position embedding for location 20

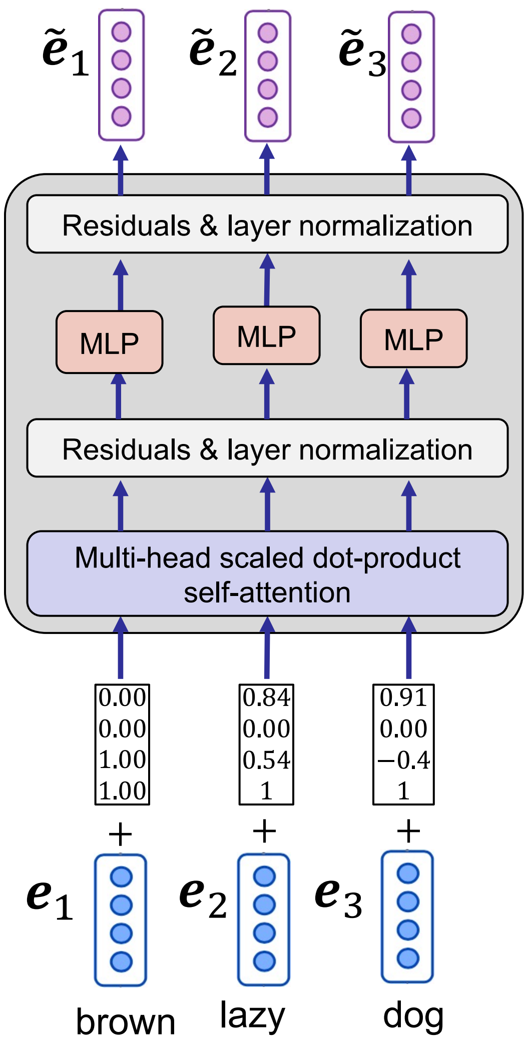


Values from -1 (dark) to +1 (light)

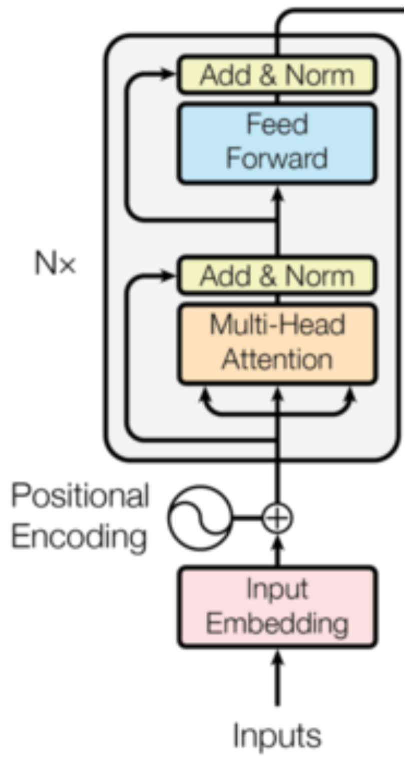
Dimensions (512)

Source: http://jalammar.github.io/illustrated-transformer/

Transformer Encoder with position embedding



Transformer Encoder with position embedding

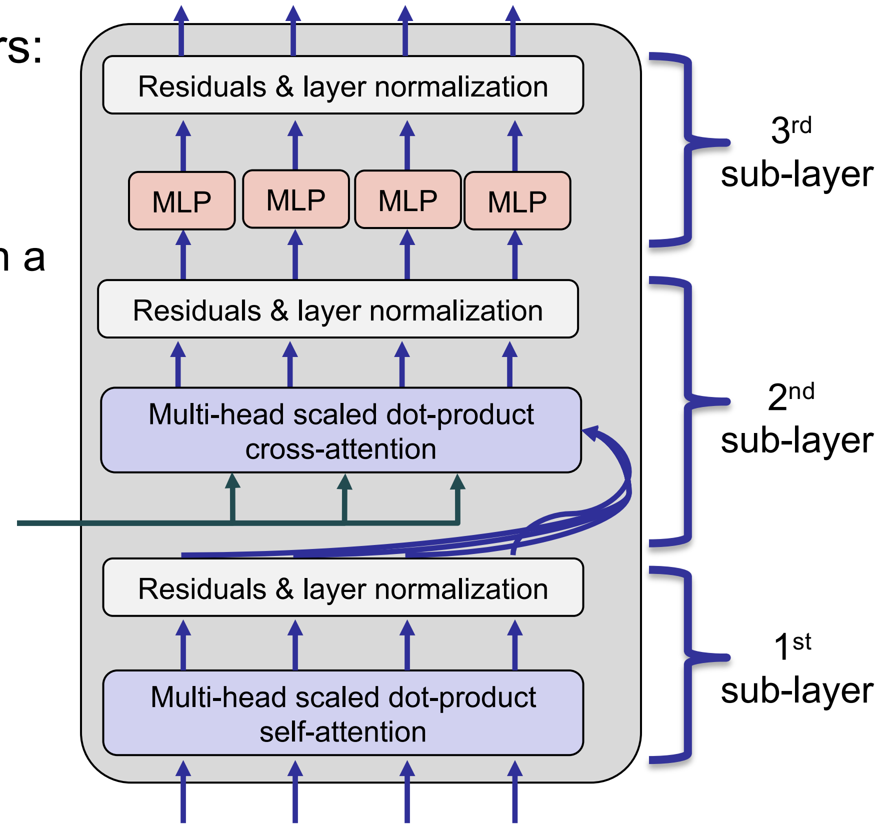


Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

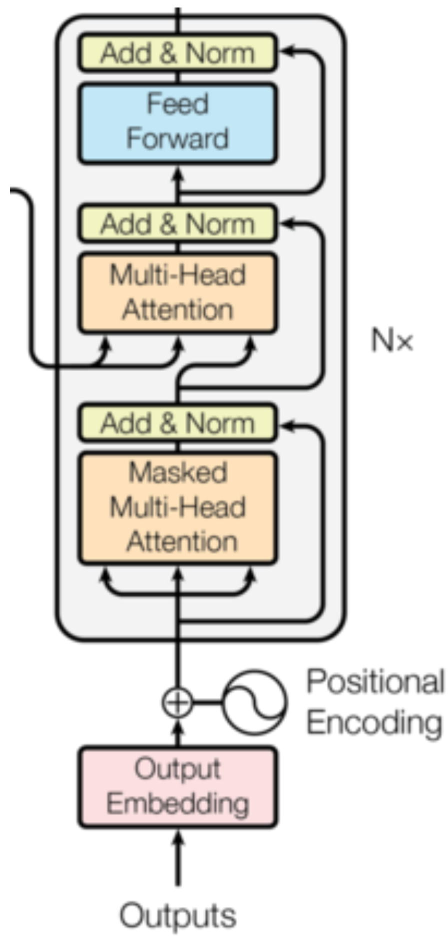
Transformer Decoder

Transformer Decoder consists of three sub-layers:

- 1^{st} : Masked multi-head self-attention
  - Exactly like Transformer Encoder but also with a *masking* functionality
- 2^{nd} : Multi-head cross attention
  - Values are given from outside
    - Like from the outputs of a Transformer Encoder
  - Queries are the outputs of the 1^{st} sub-layer
- 3^{rd} : Position-wise multi-layer perceptron
  - Exactly like Transformer Encoder



Transformer Decoder with position embedding



Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

Agenda

- Transformers
  - Transformer encoder
  - Transformer decoder
- **seq2seq with Transformers**

Sequence-to-sequence modeling – recap

- Given the source sequence $X = \{x^{(1)}, x^{(2)}, \dots, x^{(L)}\}$, …
- generate the target sequence $Y = \{y^{(1)}, y^{(2)}, \dots, y^{(T)}\}$

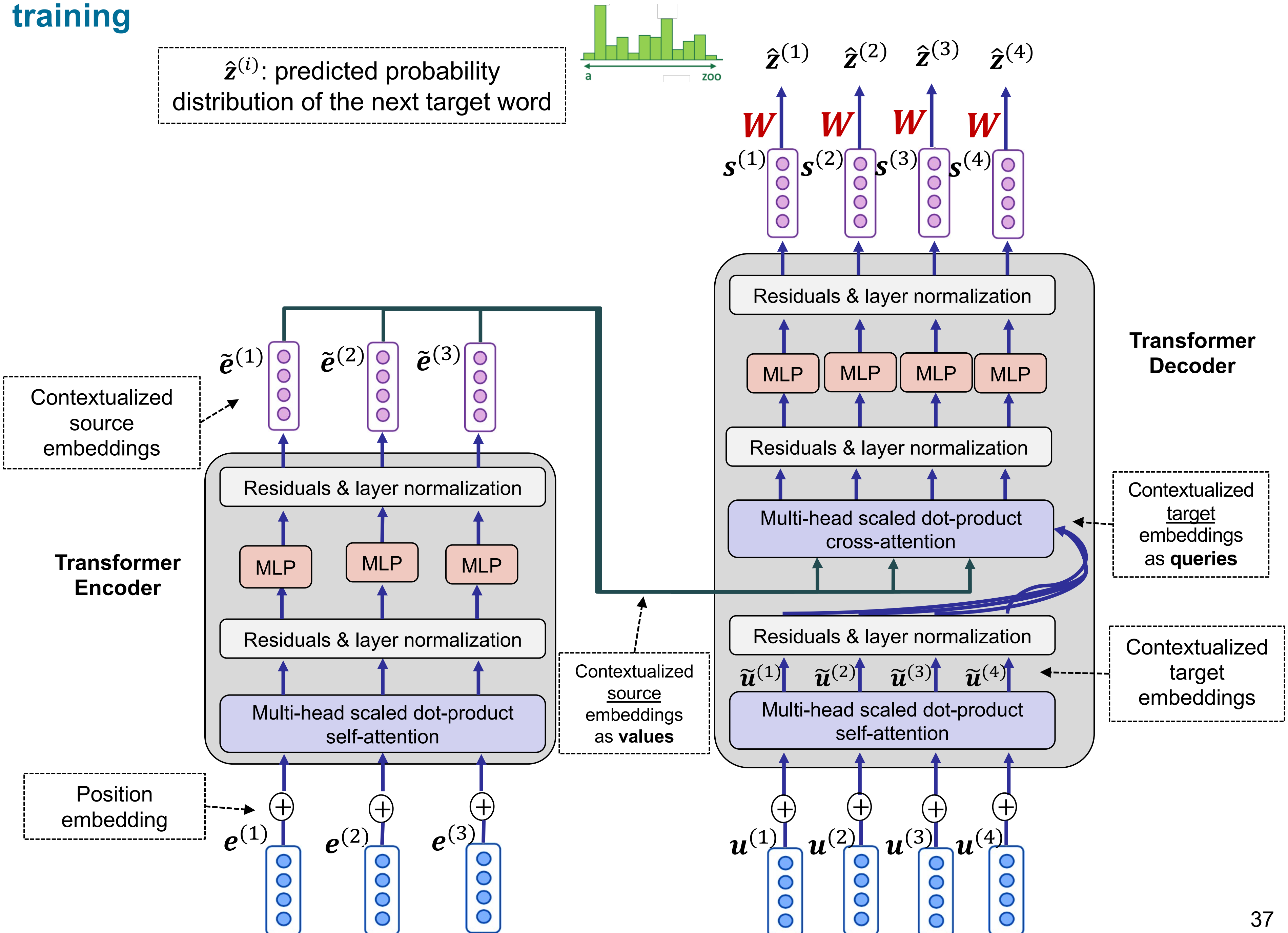
- A seq2seq model estimates the conditional probability:

$$P(Y|X)$$

- and at inference time, it generates a new sequence Y^* such that:

$$Y^* = \underset{Y}{\operatorname{argmax}}\, P(Y|X)$$

Seq2seq with Transformers – training

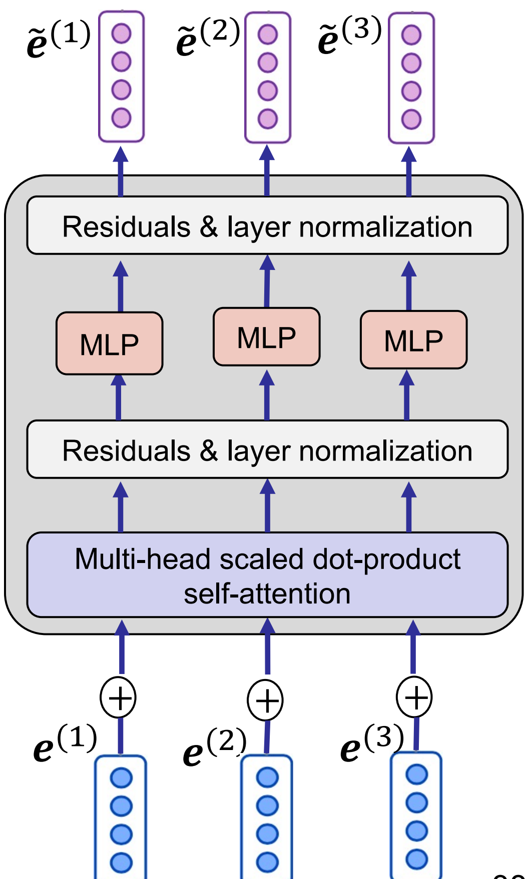


Seq2seq with Transformers – training

- Two sets of vocabularies
  - $\mathbb{V}_e$ is the set of vocabularies for source sequences
  - $\mathbb{V}_d$ is the set of vocabularies for target sequences
- Source sequence X and target sequence Y
  - Both are typically started/ended with $<\mathrm{bos}>/<\mathrm{eos}>$

<u>**Encoder**</u>

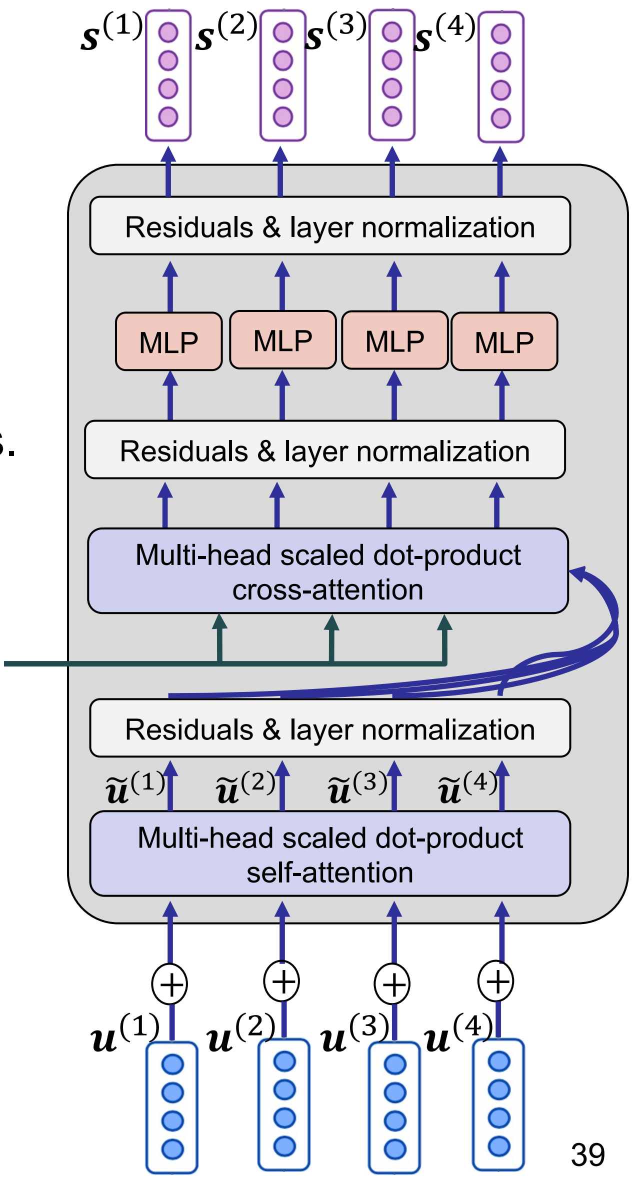
- Transformer encoder
  - passes source embeddings $\left[\boldsymbol{e}^{(1)}, \ldots, \boldsymbol{e}^{(L)}\right]$ and creates contextualized source embeddings: $\left[\tilde{\boldsymbol{e}}^{(1)}, \ldots, \tilde{\boldsymbol{e}}^{(L)}\right]$



Seq2seq with Transformers – training

Decoder

- Transformer Decoder self-attention layer
  - passes target embeddings $[\boldsymbol{u}^{(1)}, \dots, \boldsymbol{u}^{(T)}]$ and creates contextualized target embeddings: $[\tilde{\boldsymbol{u}}^{(1)}, \dots, \tilde{\boldsymbol{u}}^{(T)}]$
- Transformer Decoder cross-attention layer
  - applies attention with $[\tilde{\boldsymbol{u}}^{(1)}, \dots, \tilde{\boldsymbol{u}}^{(T)}]$ as queries. and $[\tilde{\boldsymbol{e}}^{(1)}, \dots, \tilde{\boldsymbol{e}}^{(L)}]$ as values (and keys)
- Transformer Decoder output
  - A set of vectors $[\boldsymbol{s}^{(1)}, \dots, \boldsymbol{s}^{(T)}]$



Incomplete version!

Seq2seq with Transformers – training

Decoder (cont.)

- Decoder output prediction
  - uses $\left[\boldsymbol{s}^{(1)}, \dots, \boldsymbol{s}^{(T)}\right]$ to calculate $\left[\hat{\boldsymbol{z}}^{(1)}, \dots, \hat{\boldsymbol{z}}^{(T)}\right]$, the vectors of the predicted probability distribution at the next position:

$$\hat{\boldsymbol{z}}^{(t)} = \mathrm{softmax}\left(\boldsymbol{W}[\boldsymbol{s}^{(t)}] + \boldsymbol{b}\right) \in \mathbb{R}^{|\mathbb{V}_d|}$$

- Training loss for each position t
  - NLL of the predicted probability of the next target word $y^{(t+1)}$

$$\mathcal{L}^{(t)} = -\log \hat{z}^{(t)}_{y^{(t+1)}}$$

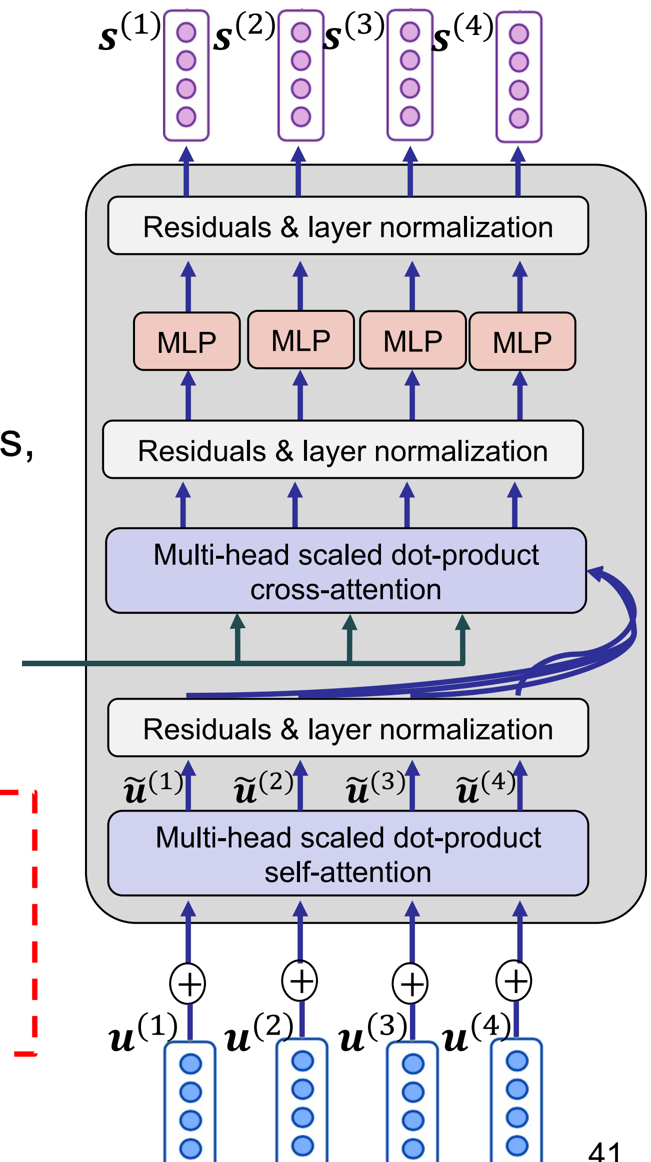
- Overall loss is the average of loss values over the target sequence:

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T}\mathcal{L}^{(t)}$$

Let's revisit the decoder!

Decoder

- Transformer Decoder self-attention layer
  - passes target embeddings $[\boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(T)}]$ and creates contextualized target embeddings: $[\tilde{\boldsymbol{u}}^{(1)}, \ldots, \tilde{\boldsymbol{u}}^{(T)}]$
- Transformer Decoder cross-attention layer
  - applies attention with $[\tilde{\boldsymbol{u}}^{(1)}, \ldots, \tilde{\boldsymbol{u}}^{(T)}]$ as queries, and $[\tilde{\boldsymbol{e}}^{(1)}, \ldots, \tilde{\boldsymbol{e}}^{(L)}]$ as values (and keys)
- Transformer Decoder output
  - A set of vectors $[\boldsymbol{s}^{(1)}, \ldots, \boldsymbol{s}^{(T)}]$



Problem: in self-attention part, every token looks at all other tokens, namely the previous ones <u>but also the next tokens</u>!

- Every token has access to what it suppose to predict!

Incomplete version!

Masking attentions

- In seq2seq with Transformers, we mask the attentions to every future token according to the self-attentions table of the Transformer Decoder

Example

- Non-normalized self-attention scores of Transformer Decoder:

attends to …
other target embeddings

Each target embedding	$\boldsymbol{u}^{(1)}$	$\boldsymbol{u}^{(2)}$	$\boldsymbol{u}^{(3)}$	$\boldsymbol{u}^{(4)}$
$\boldsymbol{u}^{(1)}$	5	3	1	-4
$\boldsymbol{u}^{(2)}$	1	4	-2	3
$\boldsymbol{u}^{(3)}$	0	2	2	-3
$\boldsymbol{u}^{(4)}$	3	-1	1	4

Non-normalized self-attention scores

	$\boldsymbol{u}^{(1)}$	$\boldsymbol{u}^{(2)}$	$\boldsymbol{u}^{(3)}$	$\boldsymbol{u}^{(4)}$
$\boldsymbol{u}^{(1)}$	5	3	1	-4
$\boldsymbol{u}^{(2)}$	1	4	-2	3
$\boldsymbol{u}^{(3)}$	0	-2	2	-3
$\boldsymbol{u}^{(4)}$	3	-1	1	4

attentions masks

	$\boldsymbol{u}^{(1)}$	$\boldsymbol{u}^{(2)}$	$\boldsymbol{u}^{(3)}$	$\boldsymbol{u}^{(4)}$
$\boldsymbol{u}^{(1)}$	1	0	0	0
$\boldsymbol{u}^{(2)}$	1	1	0	0
$\boldsymbol{u}^{(3)}$	1	1	1	0
$\boldsymbol{u}^{(4)}$	1	1	1	1

Applying masks to attention scores

- adds $-\infty$ for every mask value 0
- adds 0 for every mask value 1

	$\boldsymbol{u}^{(1)}$	$\boldsymbol{u}^{(2)}$	$\boldsymbol{u}^{(3)}$	$\boldsymbol{u}^{(4)}$
$\boldsymbol{u}^{(1)}$	5	$-\infty$	$-\infty$	$-\infty$
$\boldsymbol{u}^{(2)}$	1	4	$-\infty$	$-\infty$
$\boldsymbol{u}^{(3)}$	0	-2	2	$-\infty$
$\boldsymbol{u}^{(4)}$	3	-1	1	4

softmax

Final self-attention scores

	$\boldsymbol{u}^{(1)}$	$\boldsymbol{u}^{(2)}$	$\boldsymbol{u}^{(3)}$	$\boldsymbol{u}^{(4)}$
$\boldsymbol{u}^{(1)}$	1.00	0.00	0.00	0.00
$\boldsymbol{u}^{(2)}$	0.04	0.96	0.00	0.00
$\boldsymbol{u}^{(3)}$	0.11	0.01	0.86	0.00
$\boldsymbol{u}^{(4)}$	0.25	0.01	0.34	0.70

☞ In Transformers, there are h times of such attention matrices. The same masking is applied to each of them.

Seq2seq with Transformers – training

Decoder

- Transformer Decoder self-attention layer
  - passes target embeddings $\left[\boldsymbol{u}^{(1)}, \ldots, \boldsymbol{u}^{(T)}\right]$ and creates contextualized target embeddings: $\left[\widetilde{\boldsymbol{u}}^{(1)}, \ldots, \widetilde{\boldsymbol{u}}^{(T)}\right]$ **while masking future tokens**
- Transformer Decoder cross-attention layer
  - applies attention with $\left[\widetilde{\boldsymbol{u}}^{(1)}, \ldots, \widetilde{\boldsymbol{u}}^{(T)}\right]$ as queries and $\left[\tilde{\boldsymbol{e}}^{(1)}, \ldots, \tilde{\boldsymbol{e}}^{(L)}\right]$ as values (and keys)
- Transformer Decoder output
  - A set of vectors $\left[\boldsymbol{s}^{(1)}, \ldots, \boldsymbol{s}^{(T)}\right]$

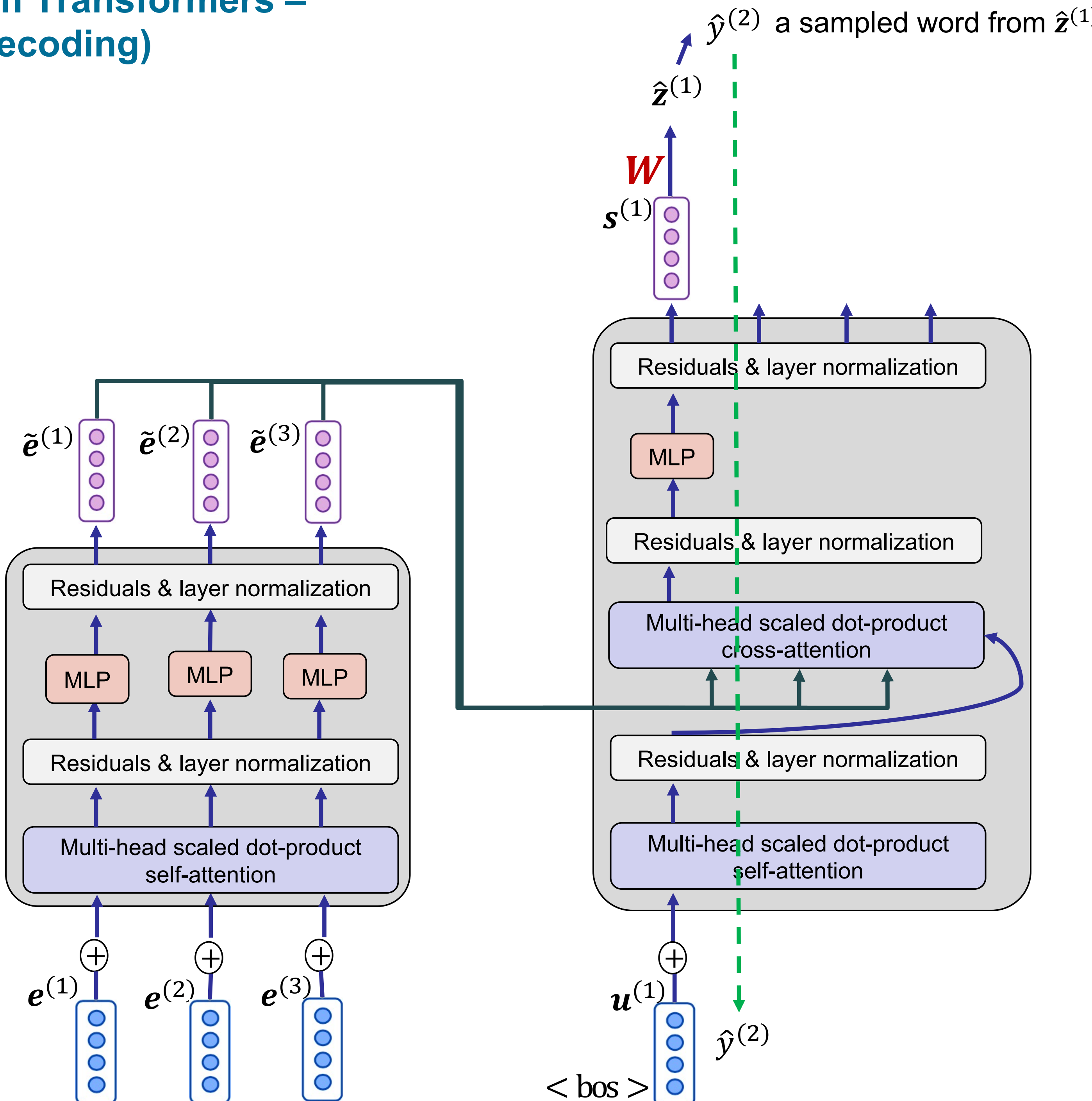
Complete version!

Inference (decoding)

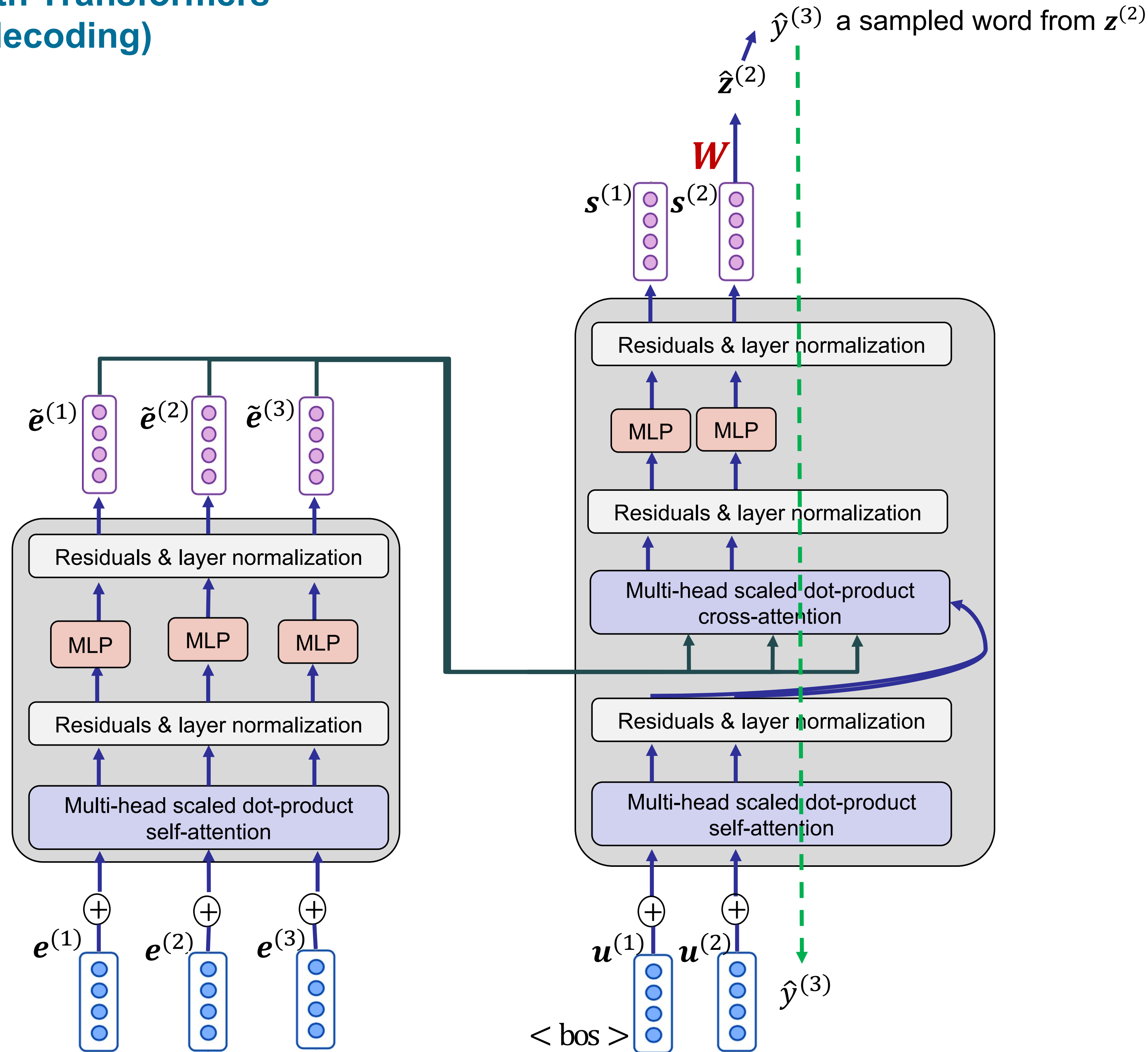
- During inference, as in training, the encoding of input sequence is done with a single computation (non-autoregressive)

- However, as in seq2seq with RNNs, decoding of seq2seq with Transformers is done in autoregressive fashion (one token after each other):
  - Pass the 1st target token ($<\mathrm{bos}>$), generate the 2nd token
  - Pass the 1st token + the 2nd generated target tokens, generate the 3rd token
  - Pass the 1st token + the 2nd and 3rd generated target tokens, generate the 4th token
  - …

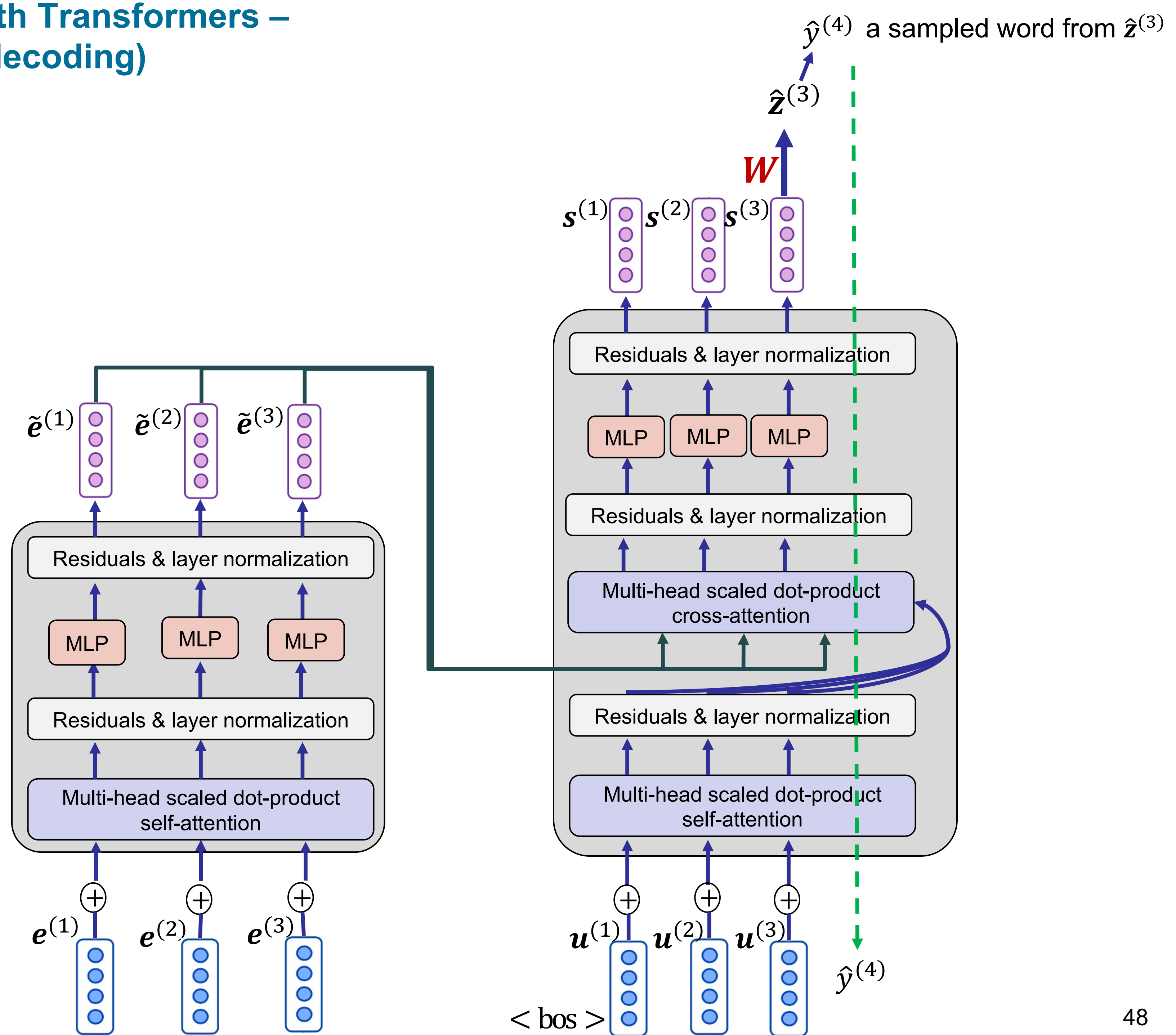
Seq2seq with Transformers – inference (decoding)



Seq2seq with Transformers – inference (decoding)



Seq2seq with Transformers – inference (decoding)



Seq2seq with Transformers – code

- Each Transformer encoder/decoder is a block. You can stack them several times and make the network deep!

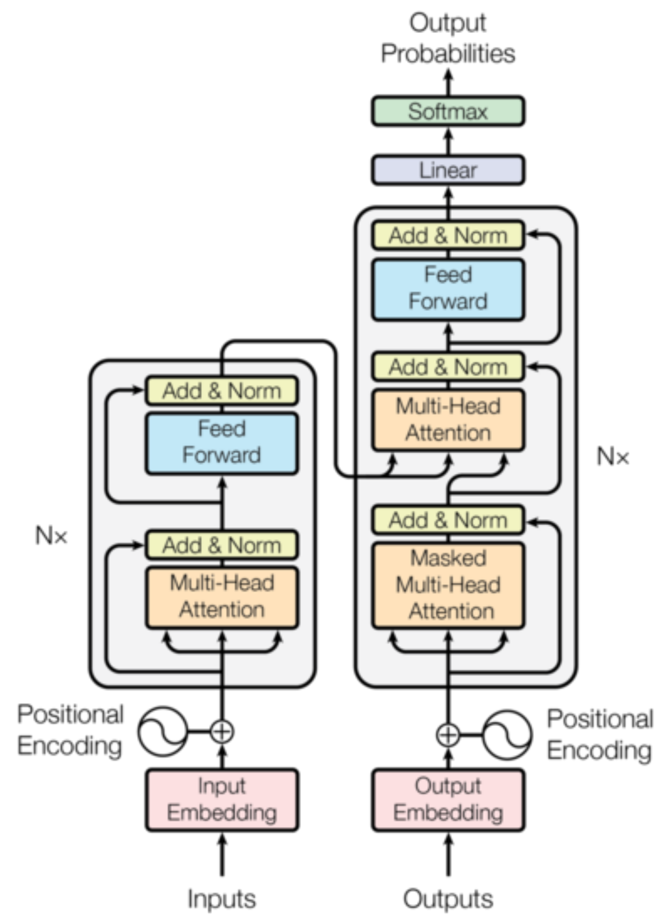
CLASS `torch.nn.TransformerEncoder(encoder_layer, num_layers, norm=None)` [SOURCE]

CLASS `torch.nn.TransformerEncoderLayer(d_model, nhead, dim_feedforward=2048, dropout=0.1, activation='relu')` [SOURCE]

CLASS `torch.nn.TransformerDecoder(decoder_layer, num_layers, norm=None)` [SOURCE]

CLASS `torch.nn.TransformerDecoderLayer(d_model, nhead, dim_feedforward=2048, dropout=0.1, activation='relu')` [SOURCE]

`forward(tgt, memory, tgt_mask=None, memory_mask=None, tgt_key_padding_mask=None, memory_key_padding_mask=None)` [SOURCE]



Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.