Transformer

Wei-Ta Chu

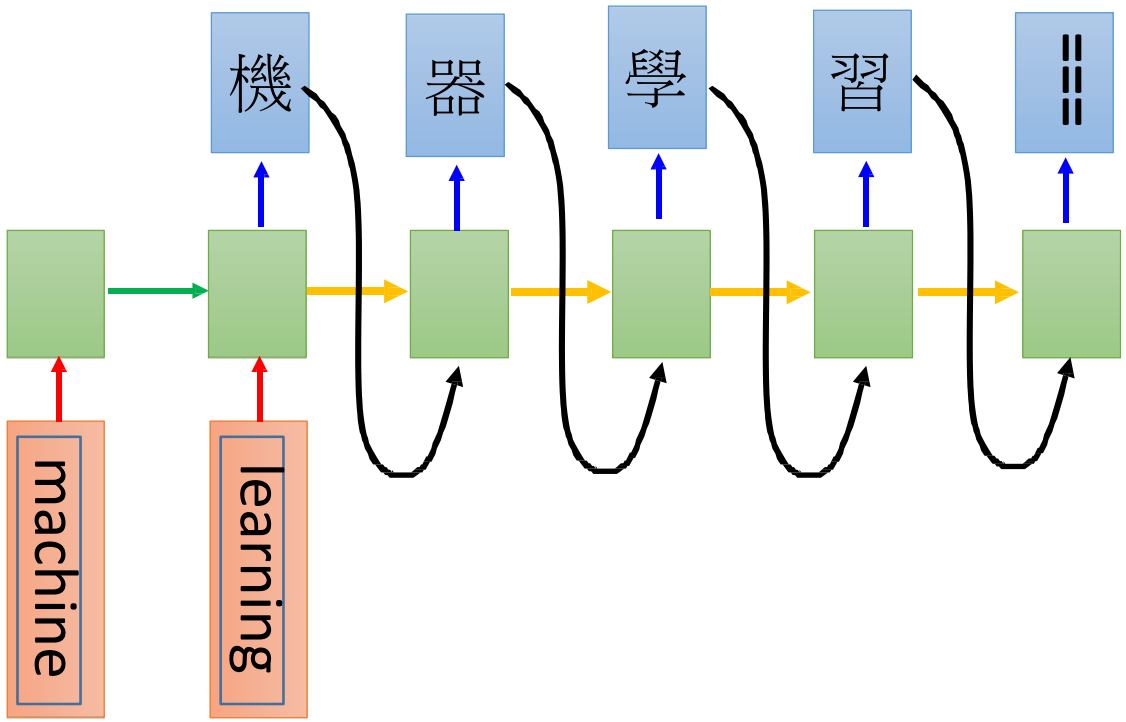
Attention Mechanism

Wei-Ta Chu

Dzmitry Bahdanau, KyungHyun Cho, and Yoshua Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” ICLR 2015.

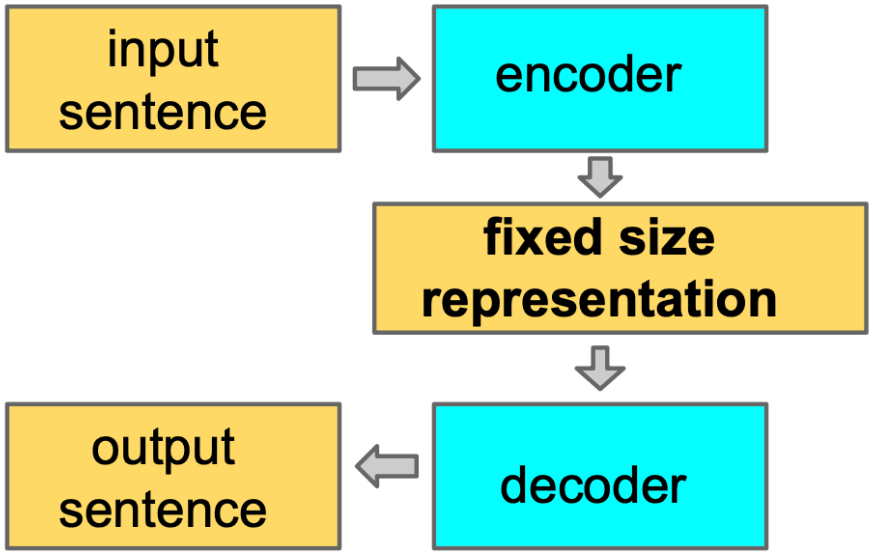
Introduction

- Machine translation – a sequence to sequence task
- Problem of RNN: For a long sequence, information of the beginnings of the sequence may miss gradually

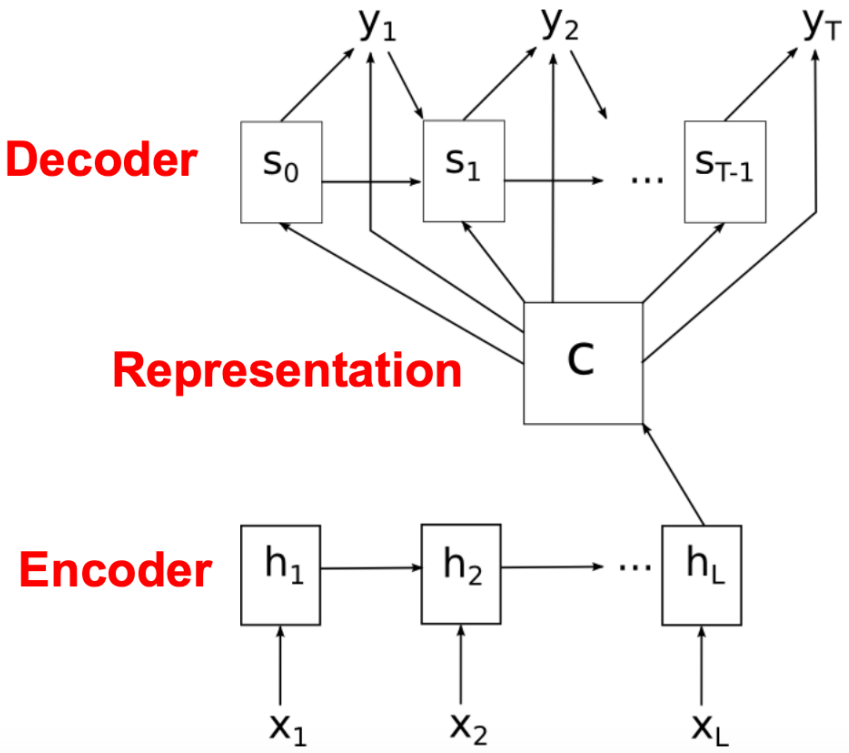


Introduction

- Encoder-decoder approach



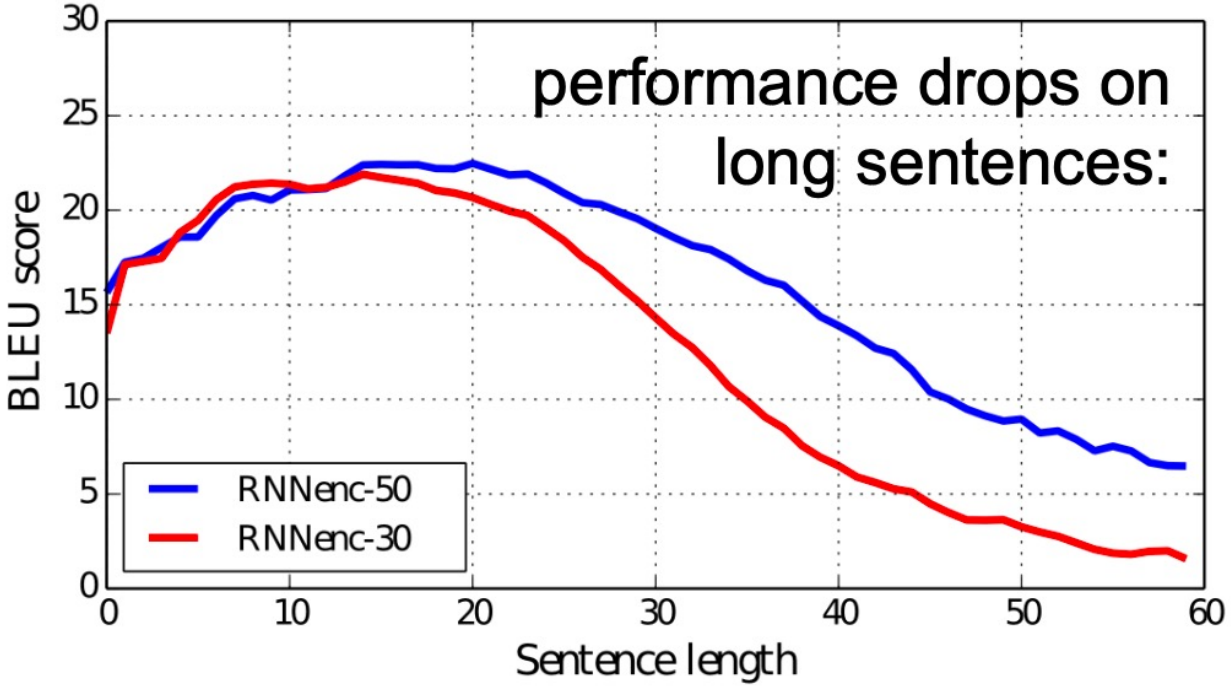
RNN Encoder-Decoder (Cho et al. 2014):



https://reurl.cc/ezW3ZQ

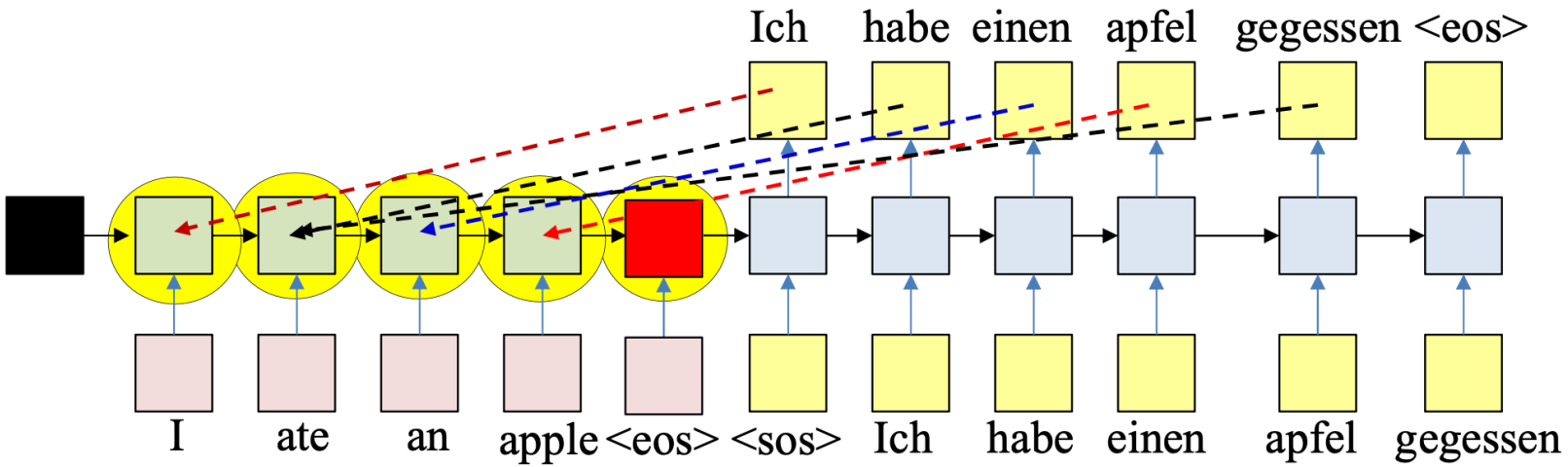
Issues

- has to remember the whole sentence
- fixed size representation can be the bottleneck
- humans do it differently



Ideas

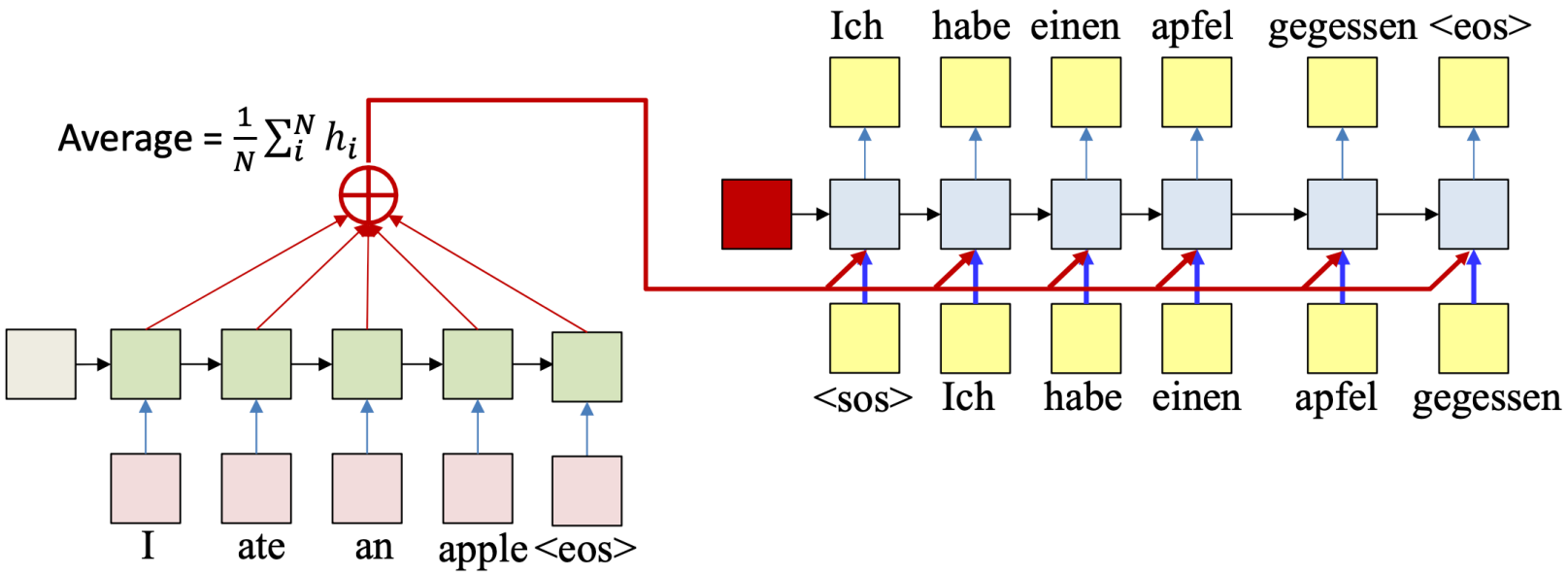
- (1) Refer to different source words' information when generating each target word
- (2) Refer more to more correlated words when generating a target word



https://reurl.cc/LWXmqL

Refer to All Words

- (1) Simple solution: average the hidden information
  - Problem: The average applies the same weight to every input
  - In practice, different outputs may be related to different inputs → (2)

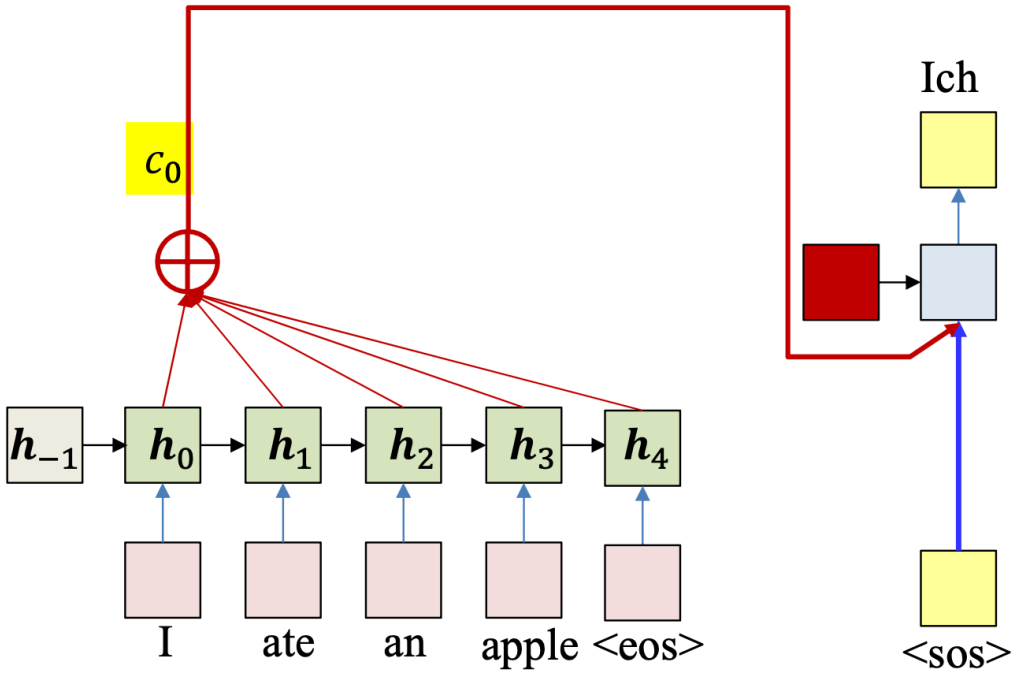


https://reurl.cc/LWXmqL

Weighted Reference

- (2) The weighted average provided for the *k*th output word is:

$$c_0 = \frac{1}{N}\sum_{i}^{N} w_i(0) h_i$$

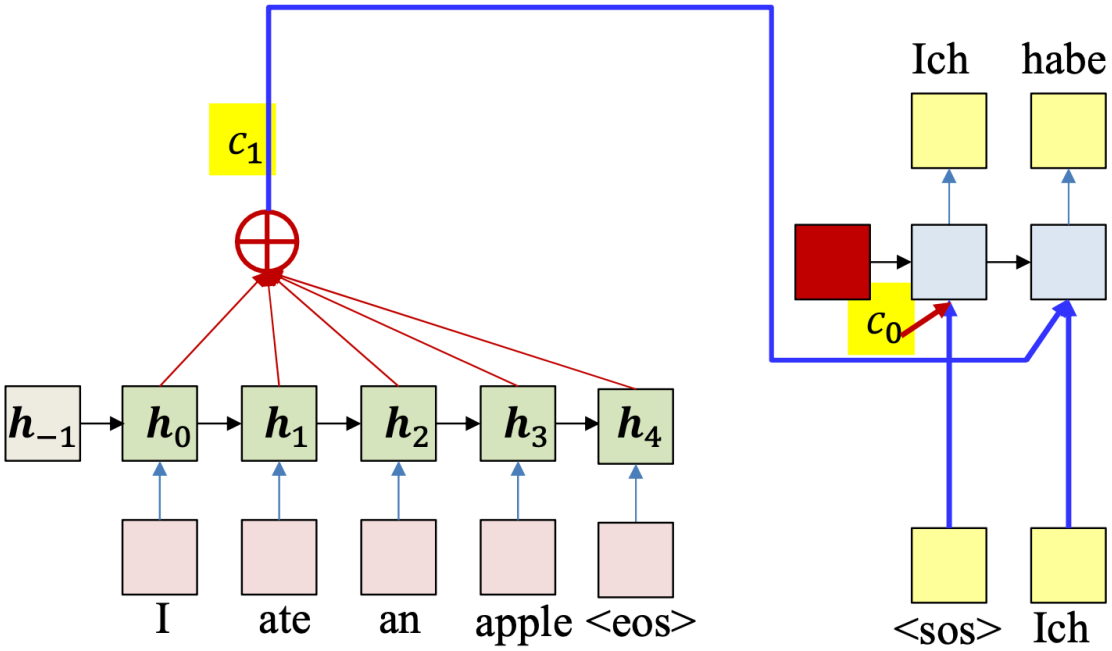


https://reurl.cc/LWXmqL

Weighted Reference

- (2) The weighted average provided for the *k*th output word is:

$$c_1 = \frac{1}{N}\sum_{i}^{N} w_i(1)h_i$$

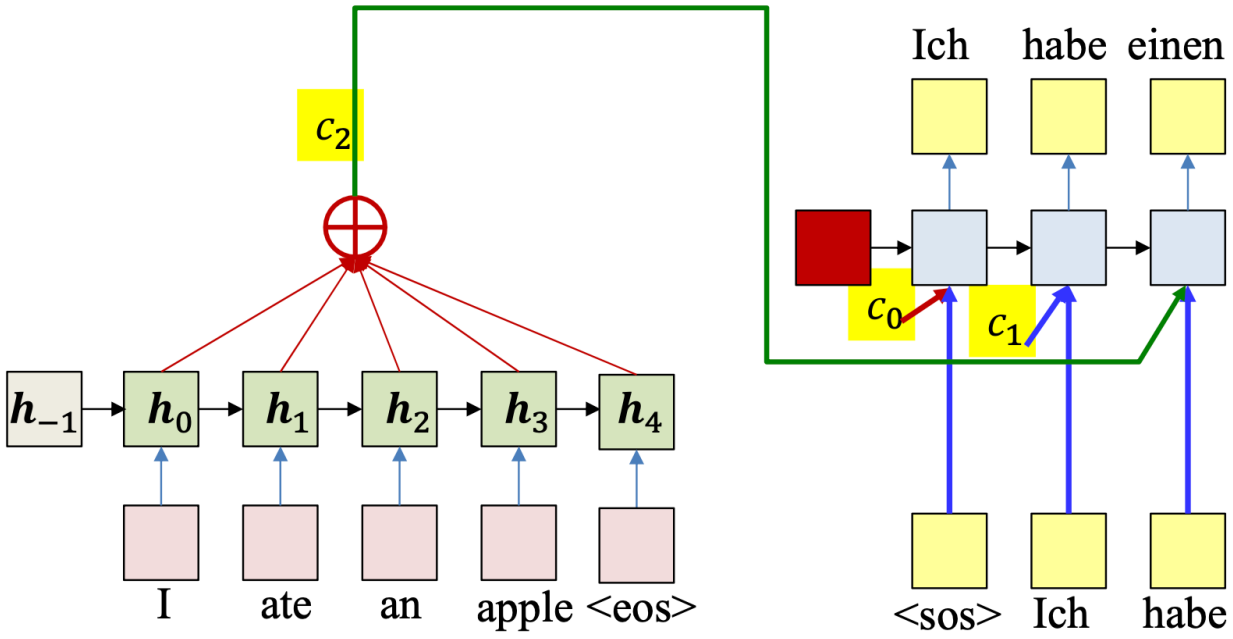


https://reurl.cc/LWXmqL

Weighted Reference

- (2) The weighted average provided for the *k*th output word is:

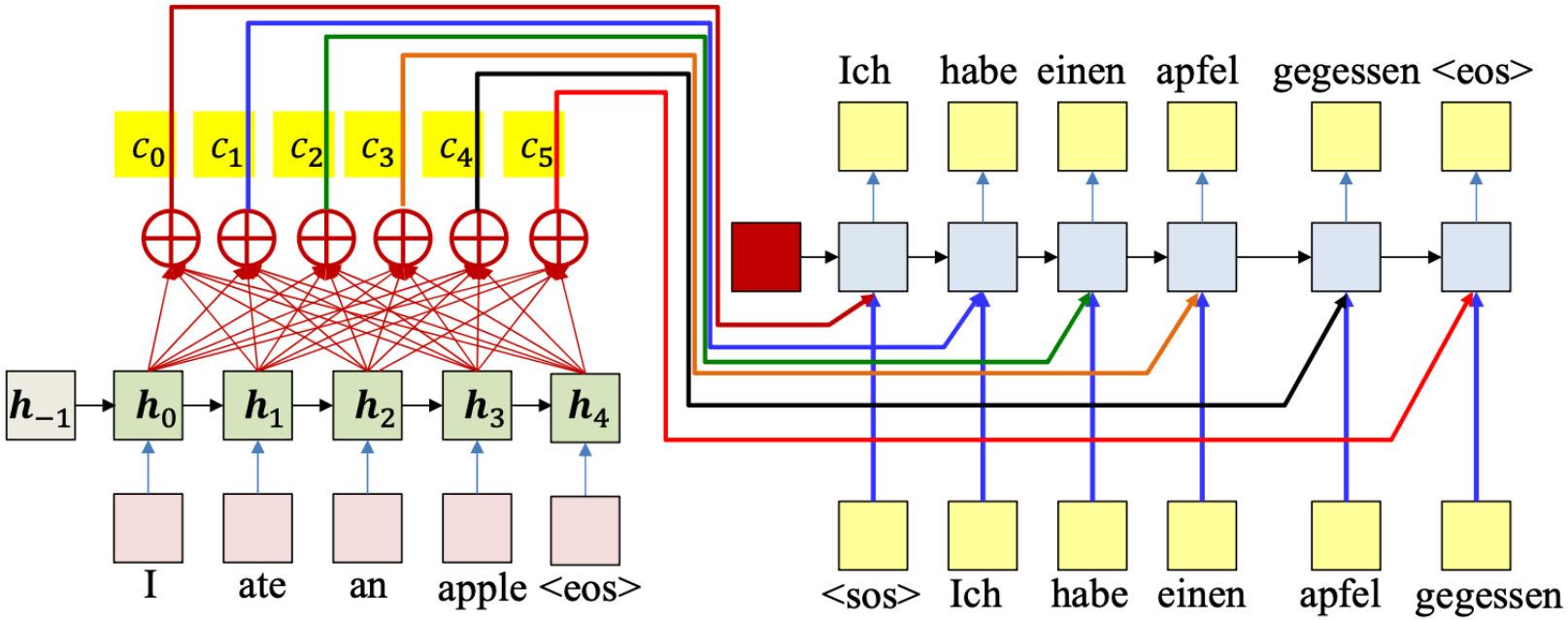
$$c_2 = \frac{1}{N}\sum_{i}^{N} w_i(2)h_i$$



https://reurl.cc/LWXmqL

Weighted Reference

- (2) This solution will work if the weights can somehow be made to "focus" on the right input word.
  - → How to determine the weights?



$$c_t = \frac{1}{N}\sum_{i}^{N} w_i(t) h_i$$

https://reurl.cc/LWXmqL

Attention Model

- Weights should be positive and sum to 1
- They must be high for the most relevant inputs for the *i*th output and low elsewhere

- A two step weight computation
  - First compute raw weights (which could be +ve or –ve)
  - Then softmax them to convert them to a distribution

<1> $$e_i(t) = g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1})$$ What is this function?

<2> $$w_i(t) = \frac{\exp(e_i(t))}{\sum_j \exp(e_j(t))}$$

https://reurl.cc/LWXmqL

Attention Model

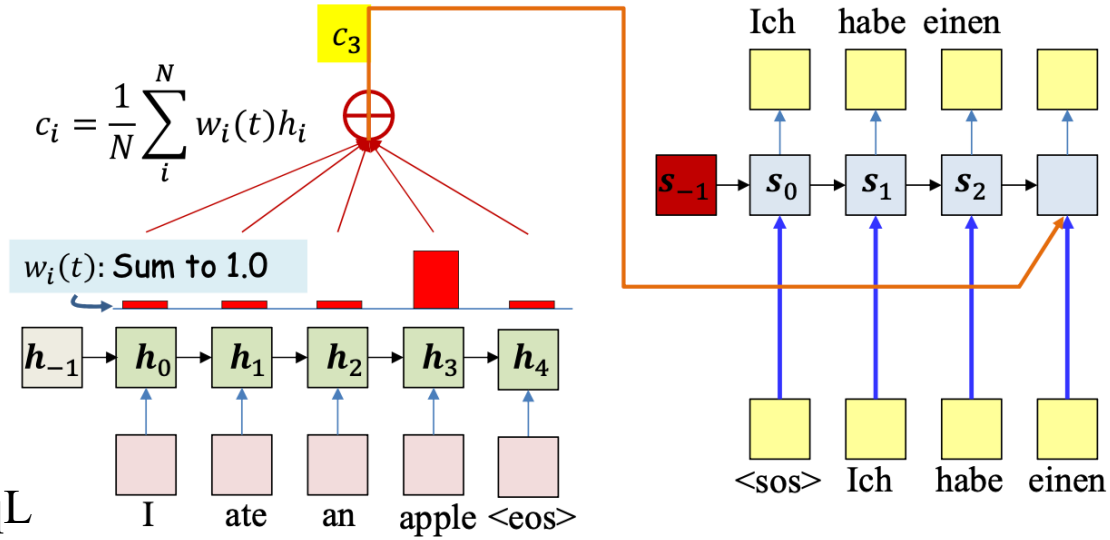
- Typical options for $g()$. Variables in read must be learned.

$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{h}_i^T \boldsymbol{s}_{t-1}$$

$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_{t-1}$$

$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{v}_g^T \tanh\left(\boldsymbol{W}_g \begin{bmatrix} \boldsymbol{h}_i \\ \boldsymbol{s}_{t-1} \end{bmatrix}\right)$$

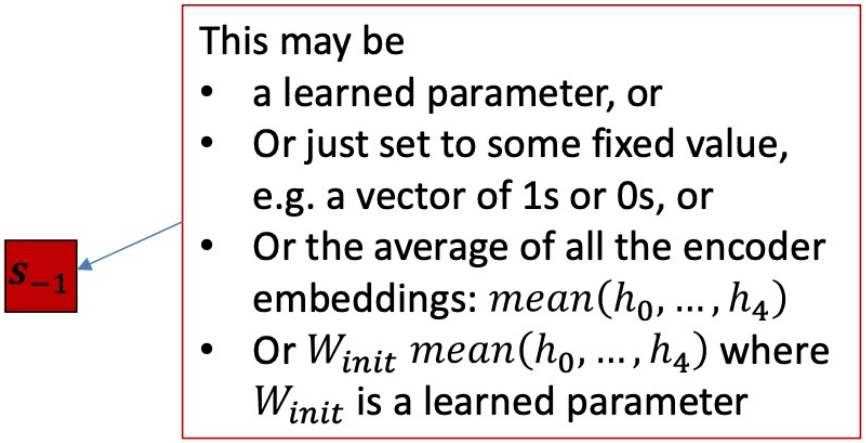
$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = MLP([\boldsymbol{h}_i, \boldsymbol{s}_{t-1}])$$

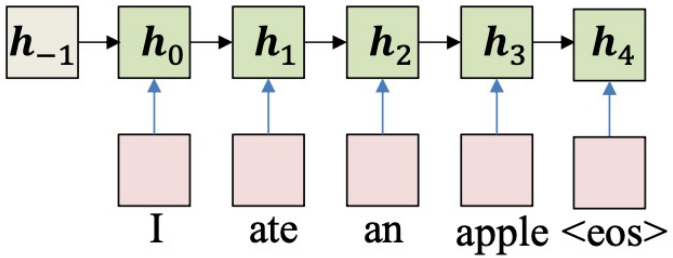


https://reurl.cc/LWXmqL

Attention Model

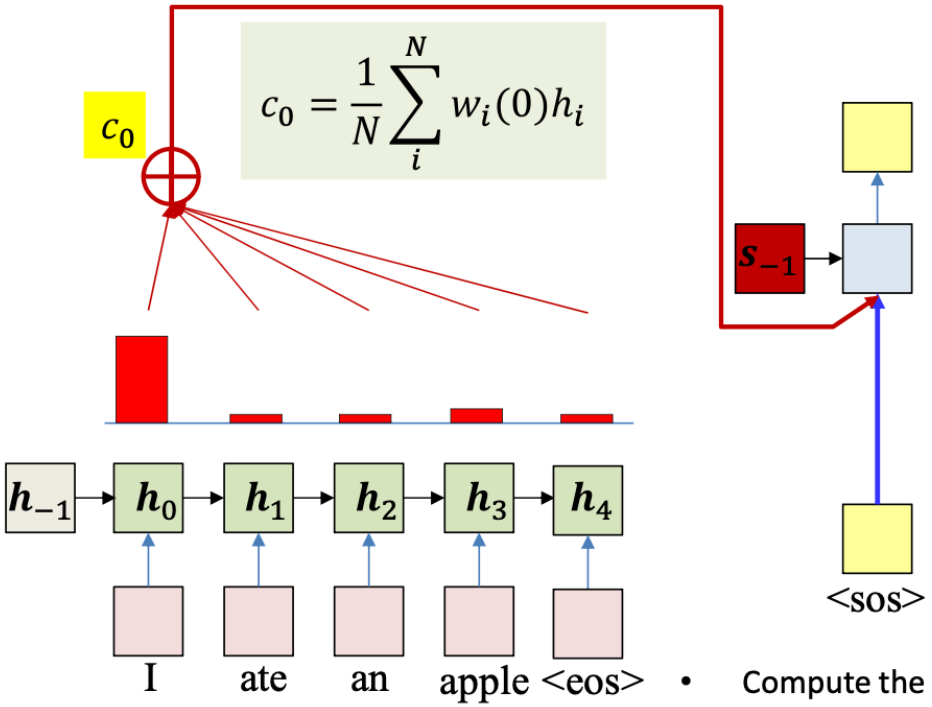
- Pass the input through the encoder to produce hidden representations $\boldsymbol{h}_i$





https://reurl.cc/LWXmqL

Attention Model



$$g(\boldsymbol{h}_i, \boldsymbol{s}_{-1}) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_{-1}$$

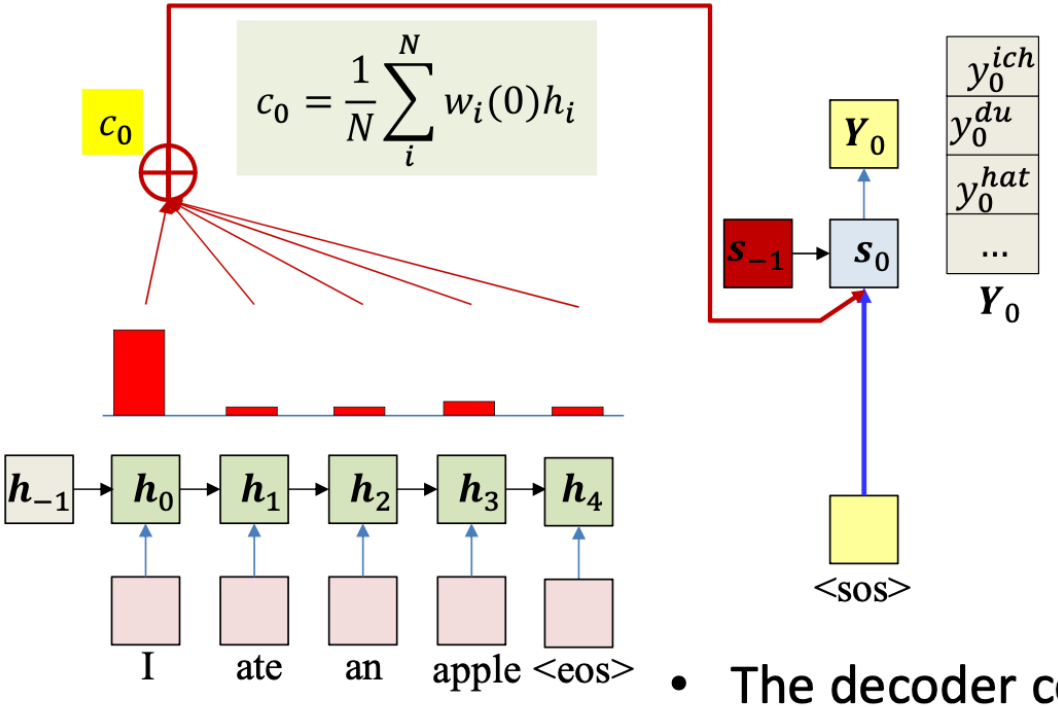
$$e_i(0) = g(\boldsymbol{h}_i, \boldsymbol{s}_{-1})$$

$$w_i(0) = \frac{\exp(e_i(0))}{\sum_j \exp(e_j(0))}$$

- Compute the attention weights $w_i(0)$ for the first output using s_{-1}
  - Will be a distribution over the input words
- Compute "context" c_0
  - Weighted sum of input word hidden states
- Input c_0 and <sos> to the decoder at time 0
  - <sos> because we are starting a new sequence
  - In practice we will enter the *embedding* of <sos>

https://reurl.cc/LWXmqL

Attention Model



$$g(\boldsymbol{h}_i, \boldsymbol{s}_{-1}) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_{-1}$$

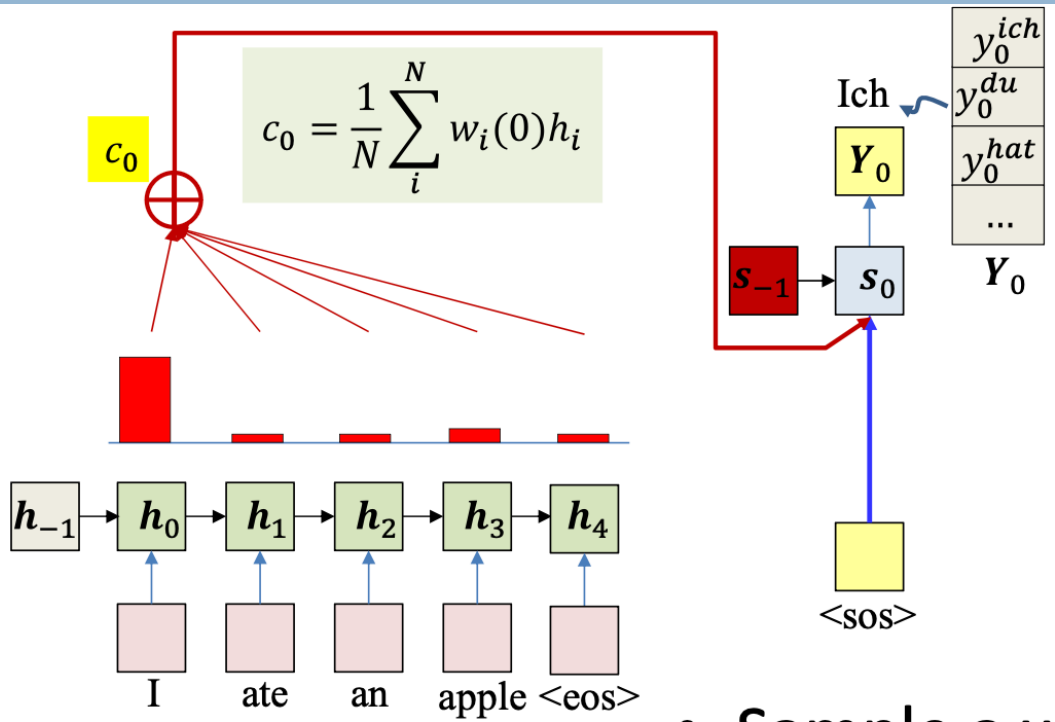
$$e_i(0) = g(\boldsymbol{h}_i, \boldsymbol{s}_{-1})$$

$$w_i(0) = \frac{\exp(e_i(0))}{\sum_j \exp(e_j(0))}$$

- The decoder computes
  - s_0
  - A probability distribution over the output vocabulary
    - Output of softmax output layer

https://reurl.cc/LWXmqL

Attention Model



$$g(\boldsymbol{h}_i, \boldsymbol{s}_{-1}) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_{-1}$$

$$e_i(0) = g(\boldsymbol{h}_i, \boldsymbol{s}_{-1})$$

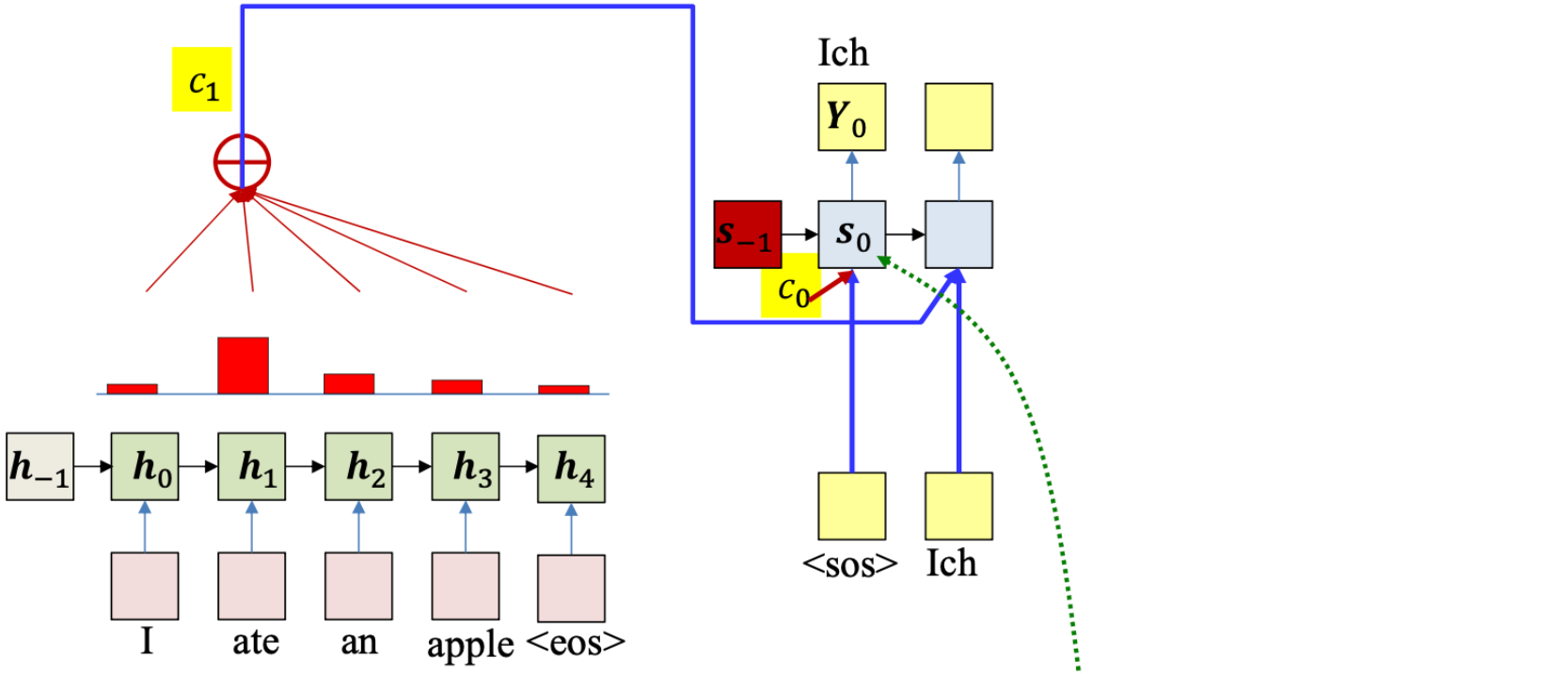
$$w_i(0) = \frac{\exp(e_i(0))}{\sum_j \exp(e_j(0))}$$

- Sample a word from the output distribution

https://reurl.cc/LWXmqL

Attention Model

https://reurl.cc/LWXmqL



- Compute the attention weights $w_i(1)$ over all inputs for the *second* output using s_0
  - Compute raw weights, followed by softmax
- Compute "context" c_1
  - Weighted sum of input hidden representations
- Input c_1 and first output word to the decoder
  - In practice we enter the *embedding* of the word

$$g(\boldsymbol{h}_i, \boldsymbol{s}_0) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_0$$

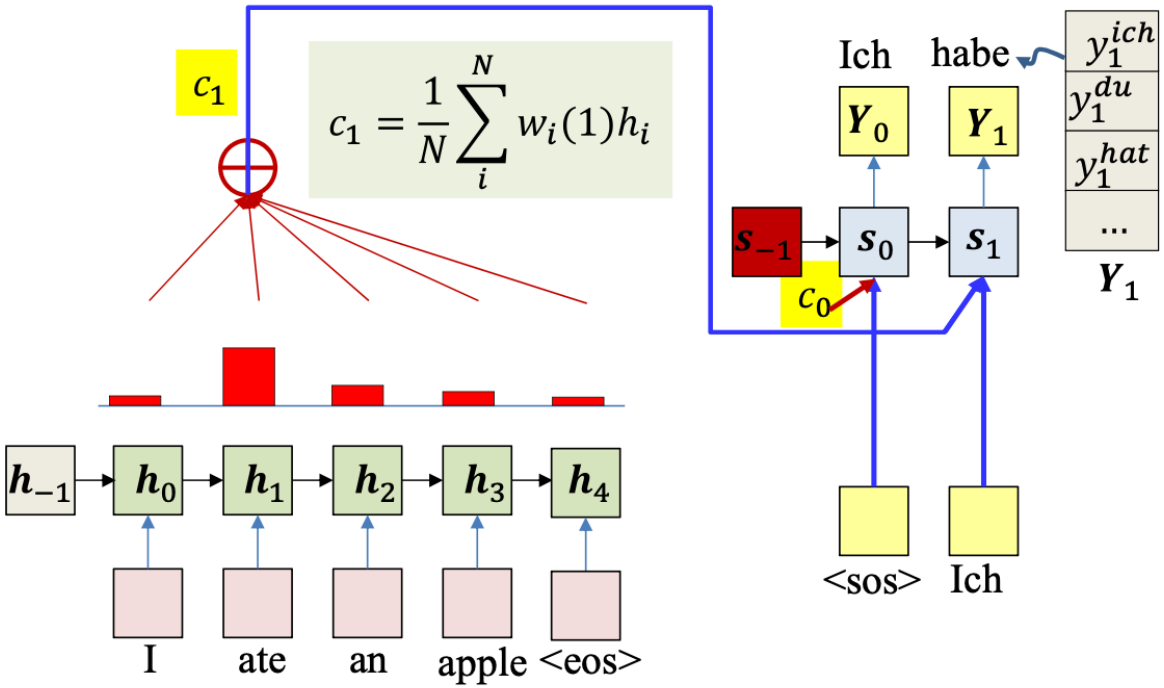
$$e_i(1) = g(\boldsymbol{h}_i, \boldsymbol{s}_0)$$

$$w_i(1) = \frac{\exp(e_i(1))}{\sum_j \exp(e_j(1))}$$

$$c_1 = \frac{1}{N}\sum_i^N w_i(1) h_i$$

Attention Model

https://reurl.cc/LWXmqL



- Sample the second word from the output distribution

$$g(\boldsymbol{h}_i, \boldsymbol{s}_0) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_0$$

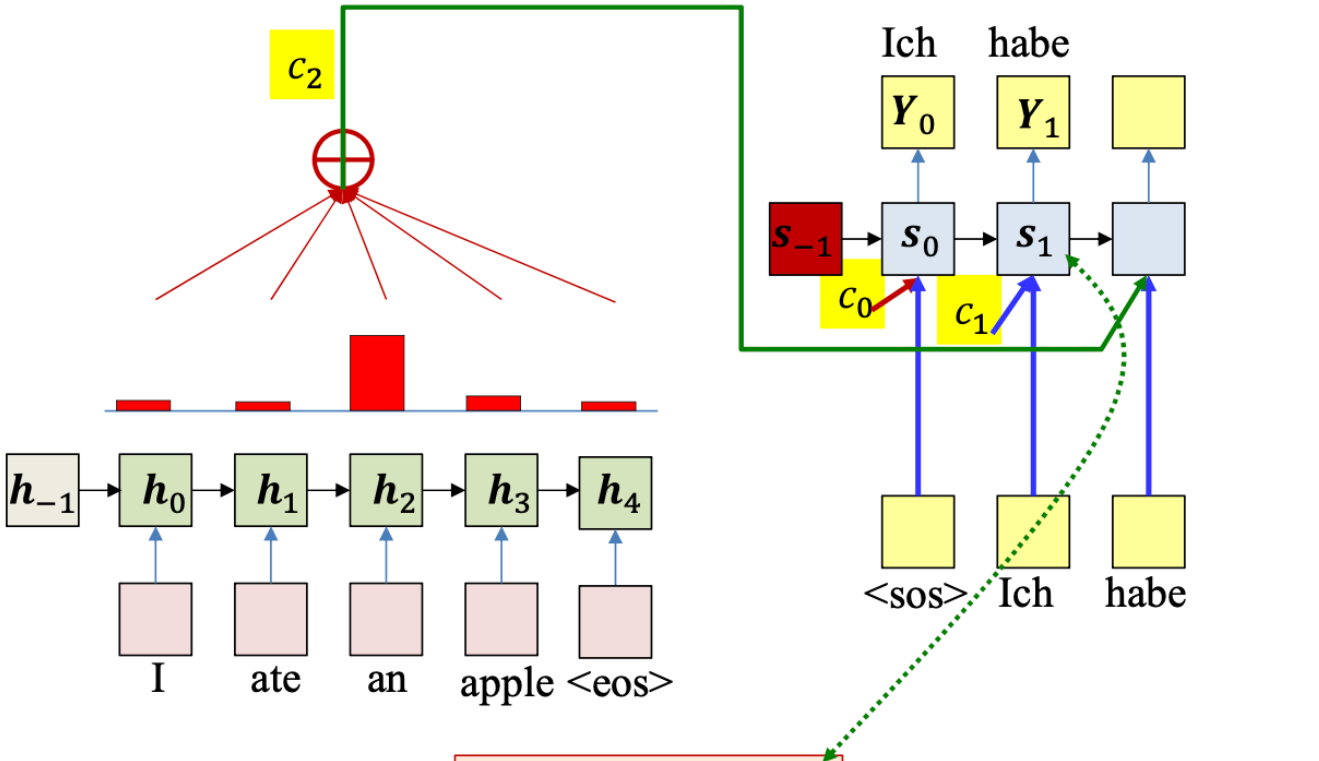
$$e_i(1) = g(\boldsymbol{h}_i, \boldsymbol{s}_0)$$

$$w_i(1) = \frac{\exp(e_i(1))}{\sum_j \exp(e_j(1))}$$

$$c_1 = \frac{1}{N}\sum_i^N w_i(1)h_i$$

Attention Model

https://reurl.cc/LWXmqL



$$g(\boldsymbol{h}_i, \boldsymbol{s}_1) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_1$$

$$e_i(2) = g(\boldsymbol{h}_i, \boldsymbol{s}_1)$$

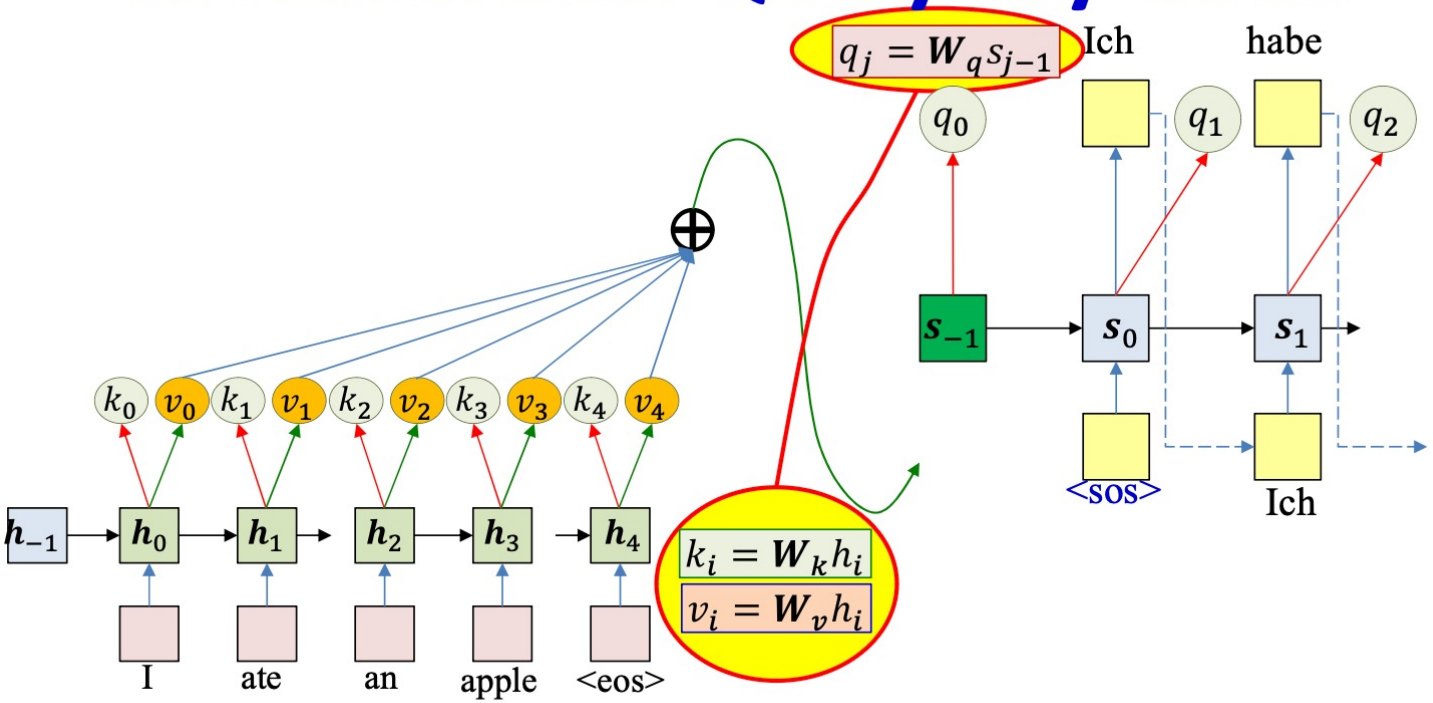
$$w_i(2) = \frac{\exp(e_i(2))}{\sum_j \exp(e_j(2))}$$

$$c_2 = \frac{1}{N}\sum_i^N w_i(2) h_i$$

Attention Model

https://reurl.cc/LWXmqL

Modification: Query key value

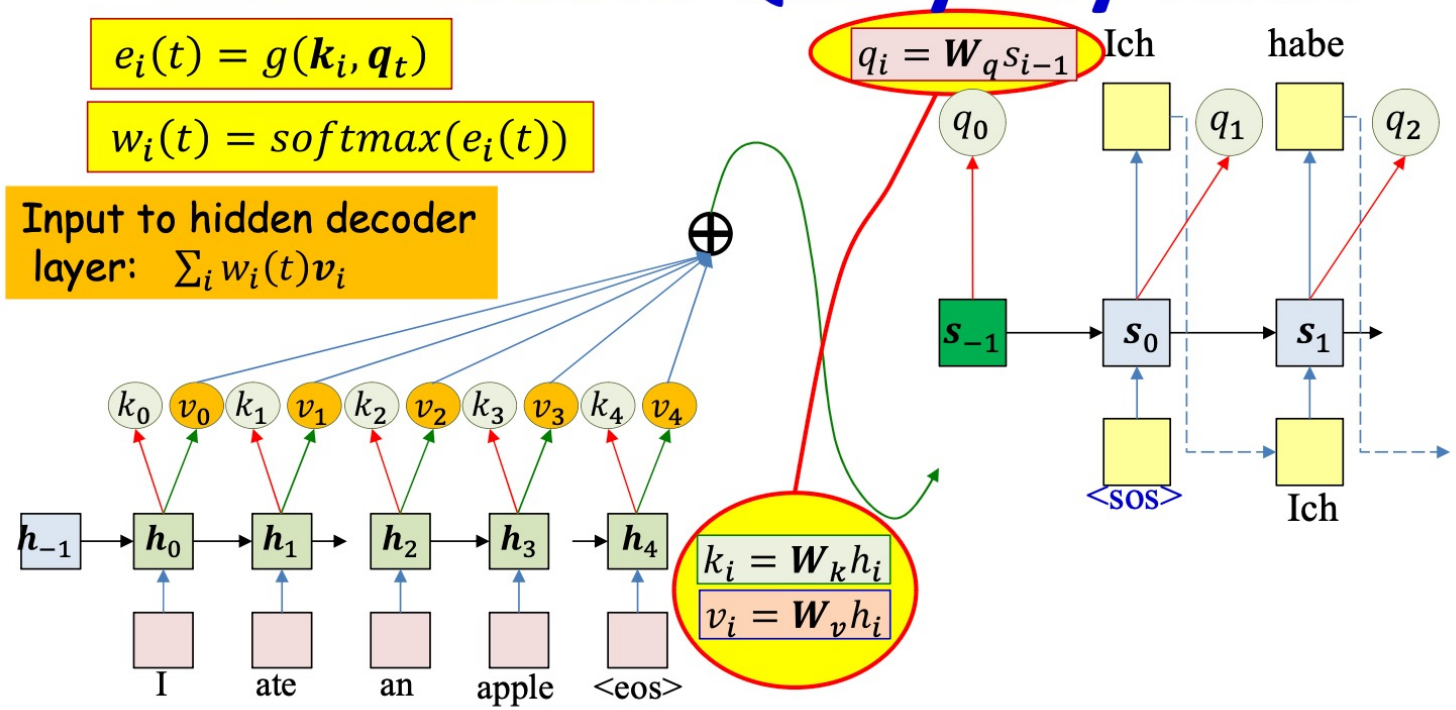


- Encoder outputs an explicit "key" and "value" at each input time
  - Key is used to evaluate the importance of the input at that time, for a given output
- Decoder outputs an explicit "query" at each output time
  - Query is used to evaluate which inputs to pay attention to
- The weight is a function of key and query
- The actual context is a weighted sum of value

Attention Model

https://reurl.cc/LWXmqL

Modification: Query key value



- Encoder outputs an explicit "key" and "value" at each input time
  - Key is used to evaluate the importance of the input at that time, for a given output
- Decoder outputs an explicit "query" at each output time
  - Query is used to evaluate which inputs to pay attention to
- The weight is a function of key and query
- The actual context is a weighted sum of value

Attention Is All You Need

Wei-Ta Chu

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, “Attention Is All You Need,” NeurIPS, 2017.

Introduction

https://reurl.cc/vamK0e

- Self-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.
- The Transformer relies entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution.

Introduction

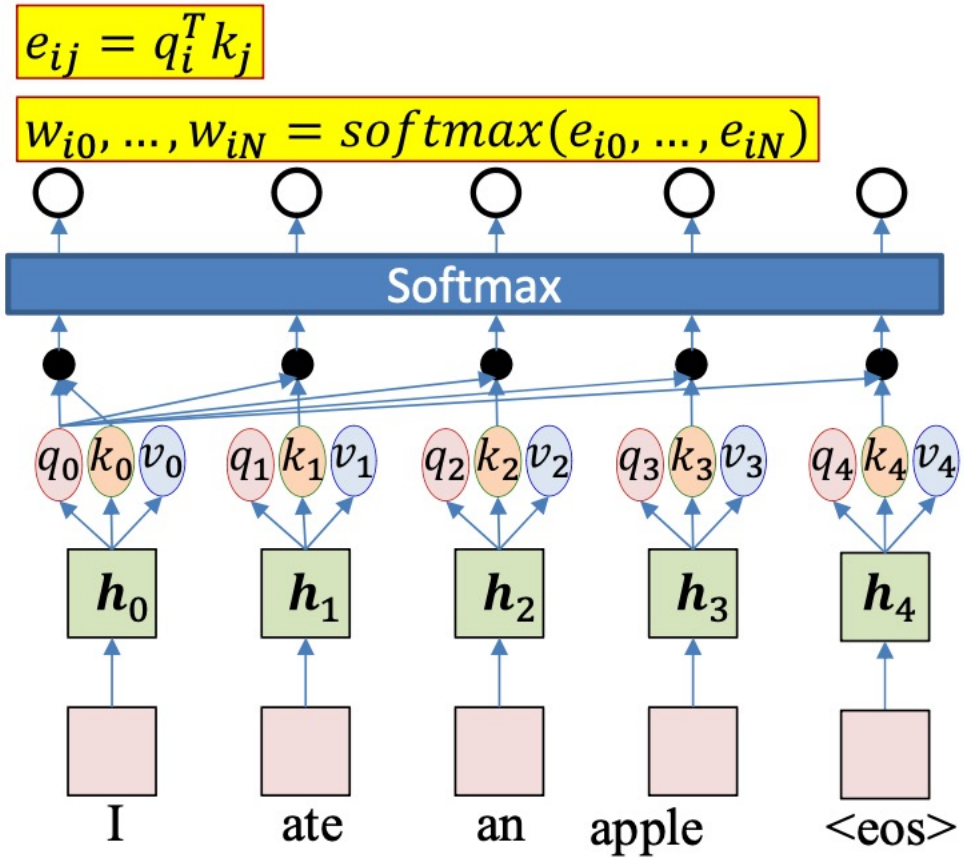
https://reurl.cc/LWXmqL

$q_i = \boldsymbol{W}_q h_i$

$k_i = \boldsymbol{W}_k h_i$

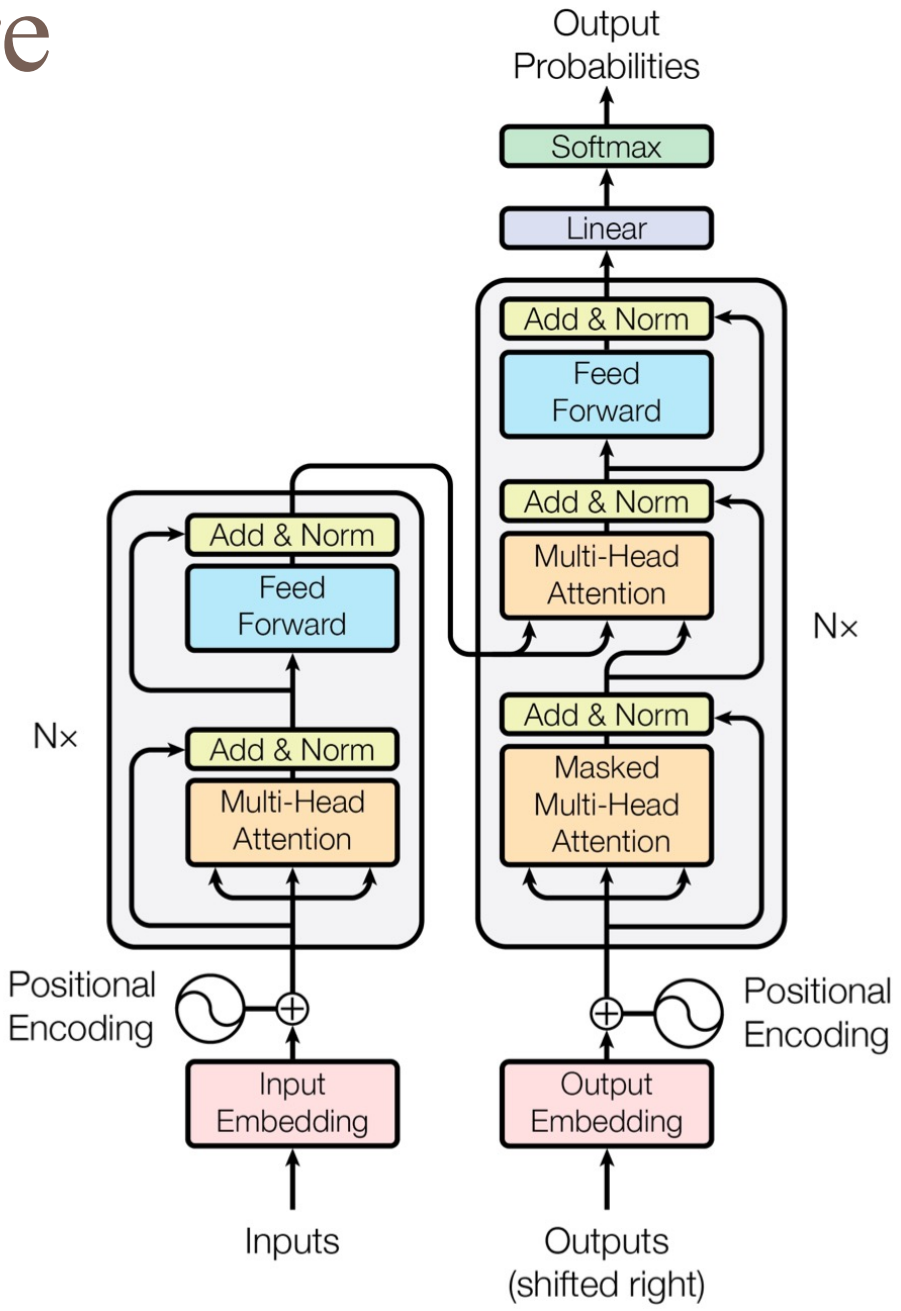
$v_i = \boldsymbol{W}_v h_i$

$w_{ij} = attn(q_i, k_{0:N})$



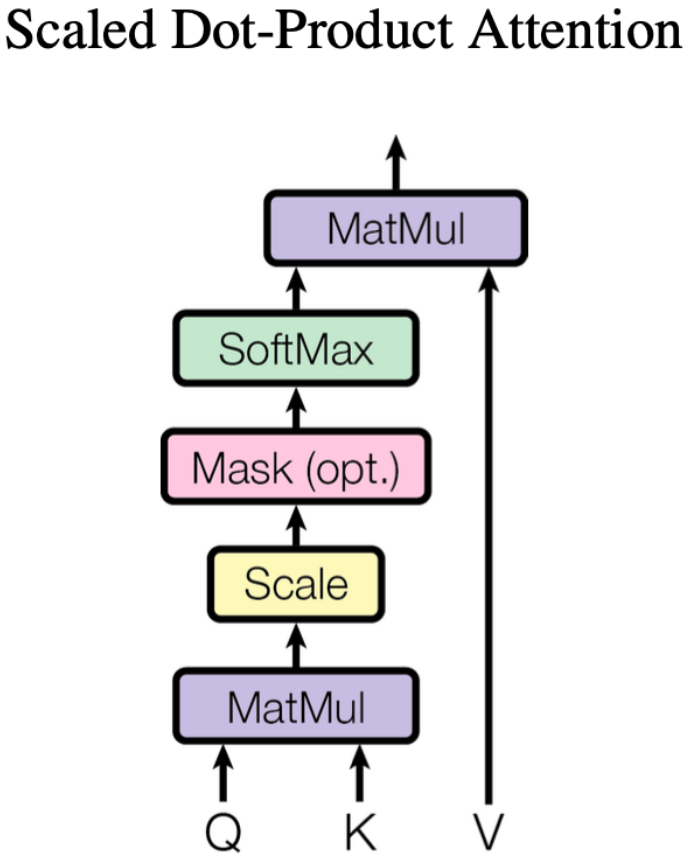
Model Architecture

- Encoder-decoder structure
- The encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, ..., z_n)$.
- Given $\mathbf{z}$, the decoder then generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time.



Self-Attention

- Scaled Dot-Product Attention
- The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by values $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the



Self-Attention

- In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V.

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

- We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\sqrt{d_k}$

Self-Attention

- In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V.

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

- We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\sqrt{d_k}$

Self-Attention

https://reurl.cc/VzRmey

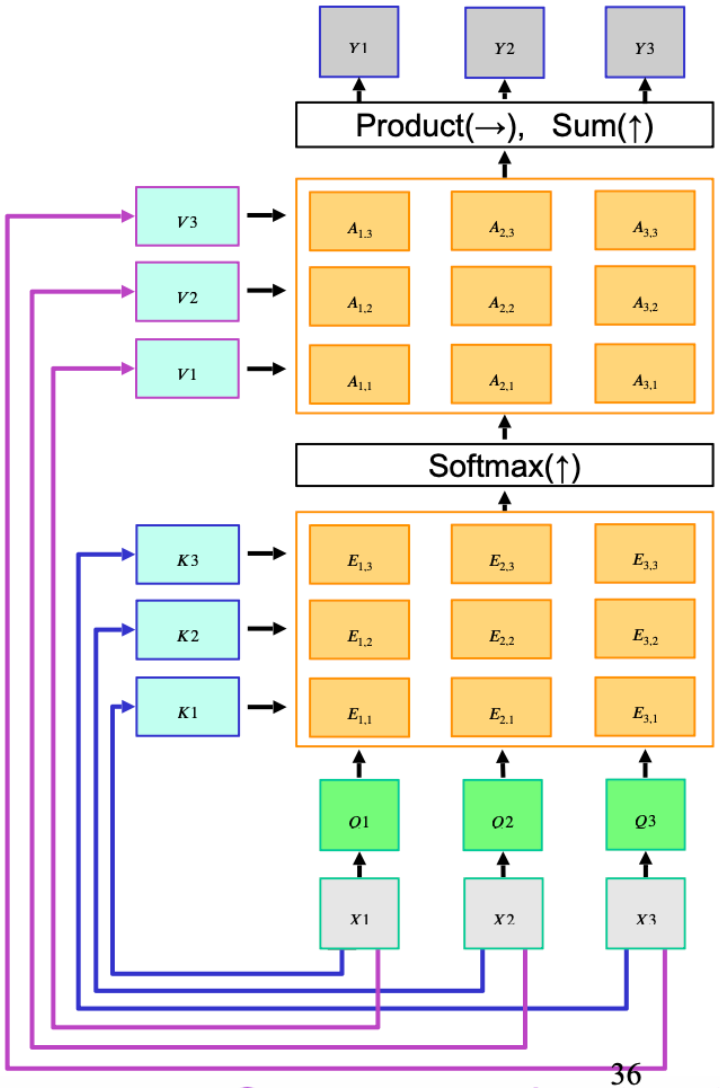
- Query vectors: $Q = XW_Q$
- Key vectors: $K = XW_K$
- Value vectors: $V = XW_V$
- Similarities: *scaled dot-product attention*

$$E_{i,j} = \frac{(Q_i \cdot Kj)}{\sqrt{D}} \text{ or } E = QK^T/\sqrt{D}$$

(D is the dimensionality of the keys)

- Attn. weights: $A = \text{softmax}(E, \text{dim} = 1)$
- Output vectors:

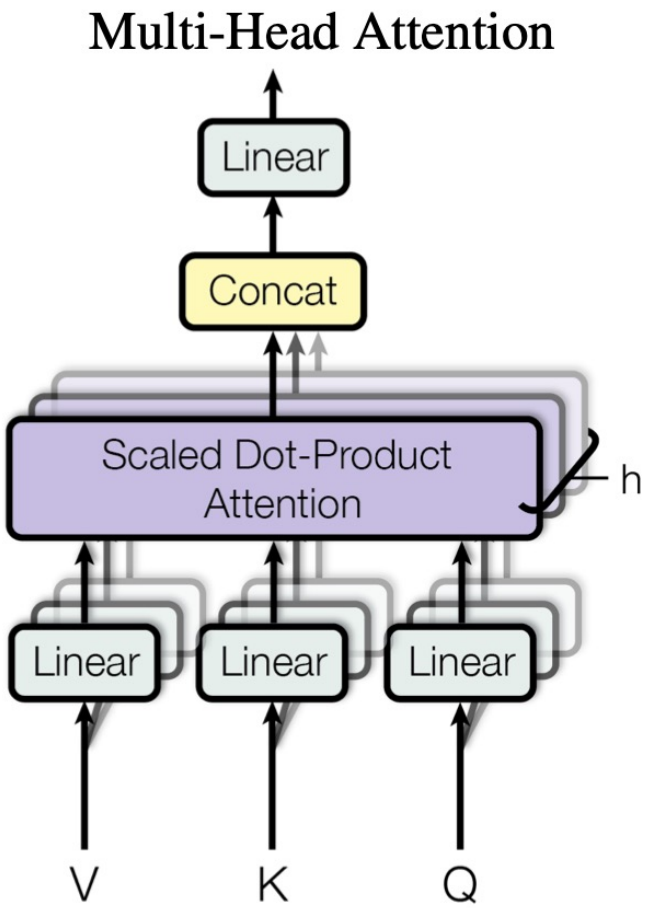
$$Y_i = \sum_j A_{i,j} V_j \text{ or } Y = AV$$



One query per input vector

Multi-Head Attention

- Multi-Head Attention
- We found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions respectively.
- On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional outpu values.



Multi-Head Attention

- Multi-head attention allows the model to jointly attend to information from **different representation subspaces** at different positions.

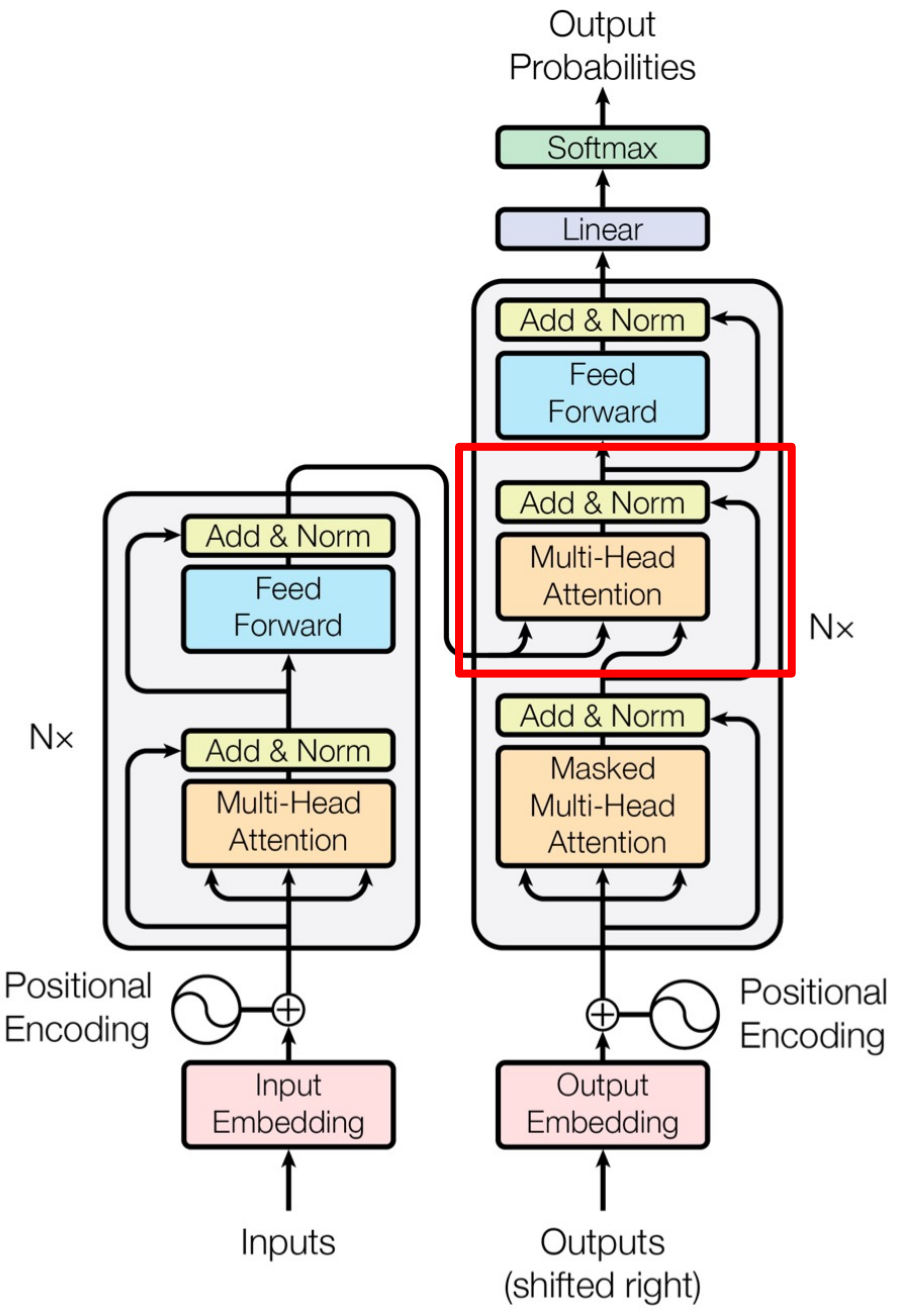
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\textbf{where } \text{head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- In this work we employ $h = 8$ parallel attention layers, or heads. For each of these we use $d_k = d_v = d_{model}/h = 64$.
- Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

Applications

- In "encoder-decoder attention" layers, the **queries** come from the previous decoder layer, and the memory **keys** and **values** come from the output of the encoder.
- This allows every position in the decoder to attend over all positions in the input sequence.



Model Architecture

- Encoder: A stack of $N = 6$ identical layers.
- Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network.
- We employ a **residual connection** around each of the two sub-layers, followed by layer normalization.
- To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{model} = 512$.

Model Architecture

https://reurl.cc/LWXmqL

$$q_i^a = W_q^a h_i$$

$$k_i^a = W_k^a h_i$$

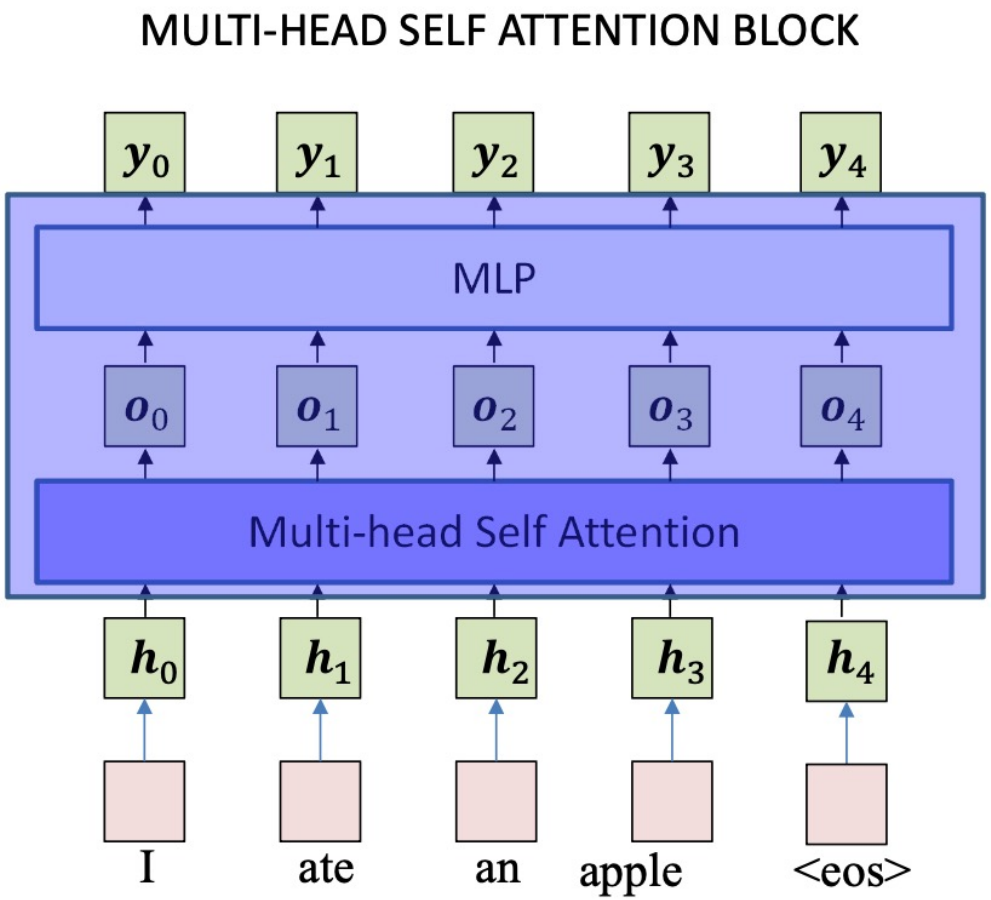
$$v_i^a = W_v^a h_i$$

$$w_{ij}^a = attn(q_i^a, k_{0:N}^a)$$

$$o_i^a = \sum_j w_{ij}^a v_j^a$$

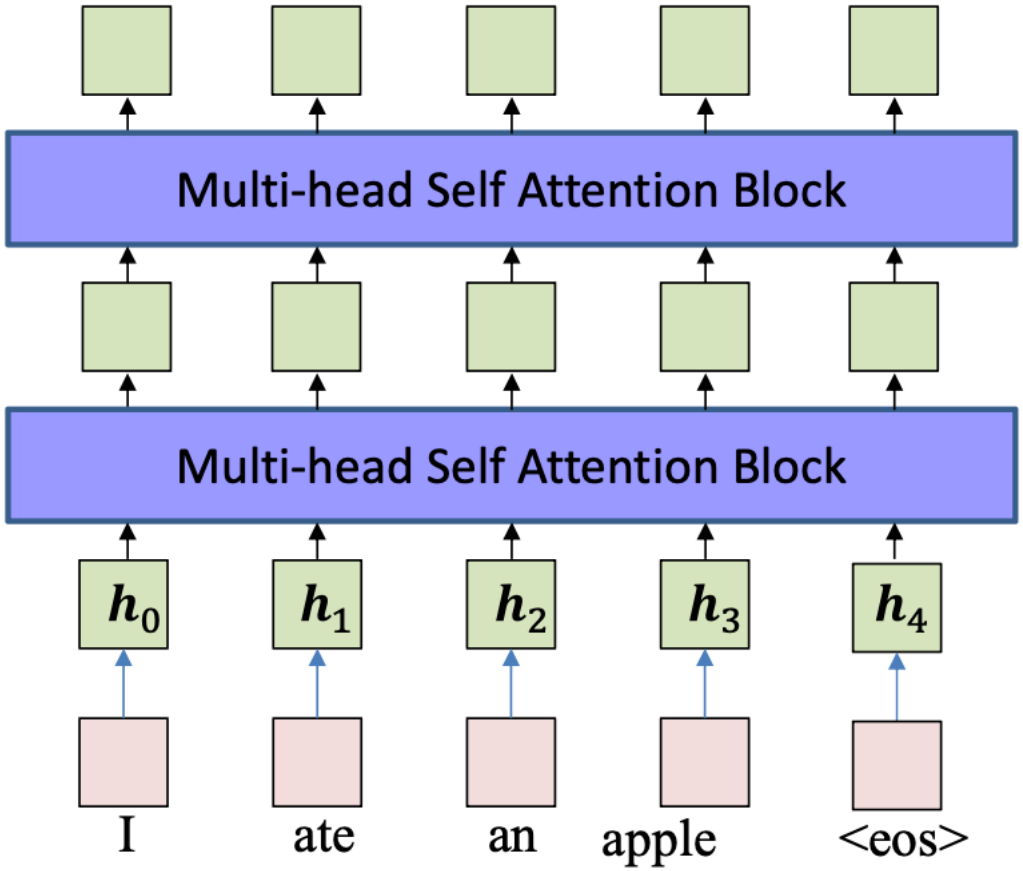
$$o_i = [o_i^1; o_i^2; o_i^3; \ldots; o_i^H]$$

$$y_i = MLP(o_i)$$



Model Architecture

https://reurl.cc/LWXmqL



Model Architecture

- Decoder: A stack of $N = 6$ identical layers.
- In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack.
- We modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Positional Encoding

- Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence.
- We add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed.

Positional Encoding

- We use sine and cosine functions of different frequencies

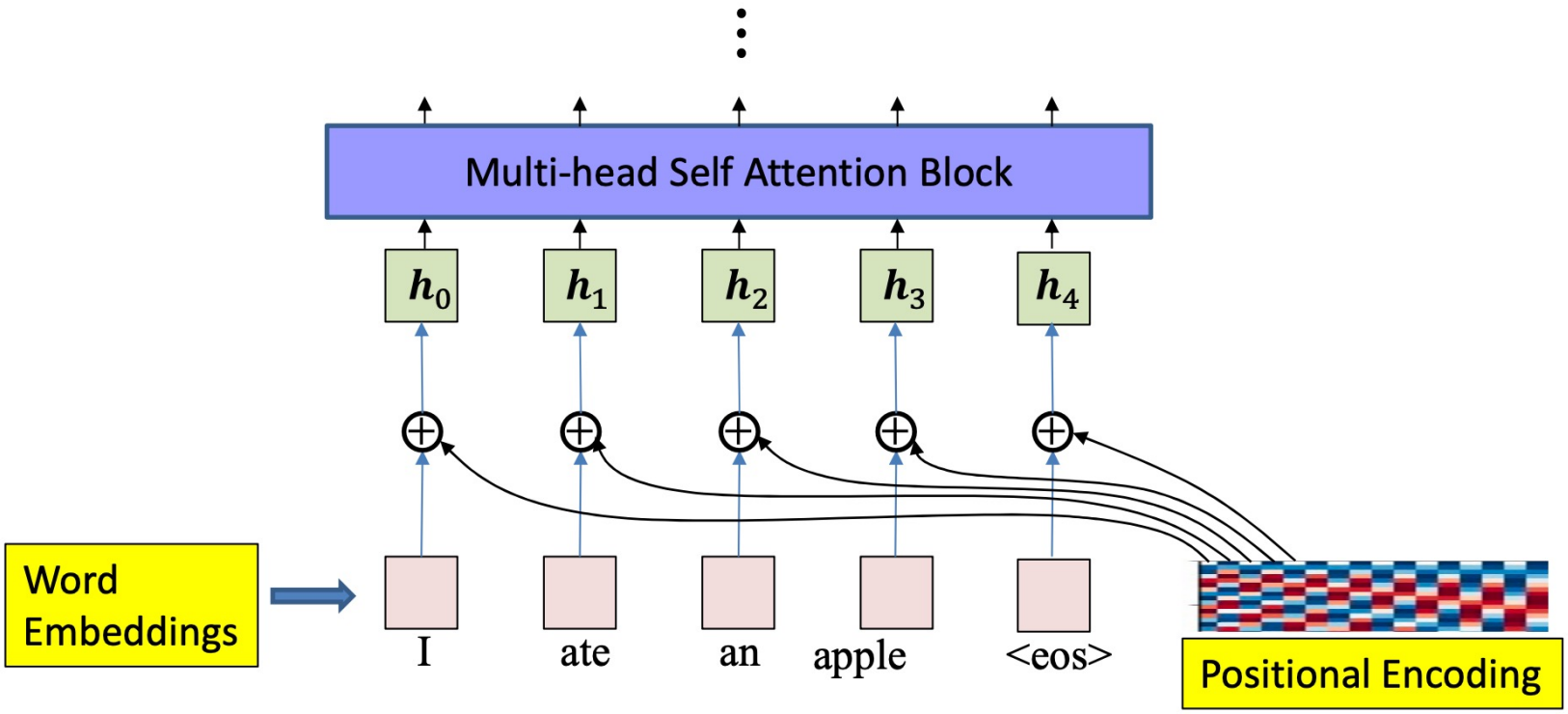
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

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

where *pos* is the position and *i* is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset *k*, PE_{pos+k} can be represented as a linear function of PE_{pos}.

Positional Encoding

https://reurl.cc/LWXmqL



Training

- We trained on the standard WMT 2014 English-German dataset consisting of about 4.5 million sentence pairs.
- For English-French, we used the significantly larger WMT 2014 English-French dataset consisting of 36M sentences and split tokens into a 32000 word-piece vocabulary. Sentence pairs were batched together by approximate sequence length. Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

Training

- We used the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\varepsilon = 10^{-9}$.
- We varied the learning rate over the course of training

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

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.

Results

- Machine translation
- On the WMT 2014 English-to-German translation task, the big transformer model outperforms the best previously reported models by more than 2.0 BLEU.

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	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**	**41.0**	$2.3 \cdot 10^{19}$	

Results

- Machine translation
- On the WMT 2014 English-to-French translation task, our big model achieves a BLEU score of 41.0, outperforming all of the previously published single models, at less than 1/4 the training cost of the previous state-of-the-art model.

Results

- Model variations

Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to per-word perplexities.

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

Results

- In Table 3 rows (A), while single-head attention is 0.9 BLEU worse than the best setting, quality also drops off with too many heads.
- In Table 3 rows (B), we observe that reducing the attention key size d_k hurts model quality.
- We further observe in rows (C) and (D) that bigger models are better, and dropout is very helpful in avoiding over-fitting.
- In row (E) we replace our sinusoidal positional encoding with learned positional embeddings, and observe nearly identical results to the base model.