Attention Models

CSE 849 Deep Learning
Spring 2025

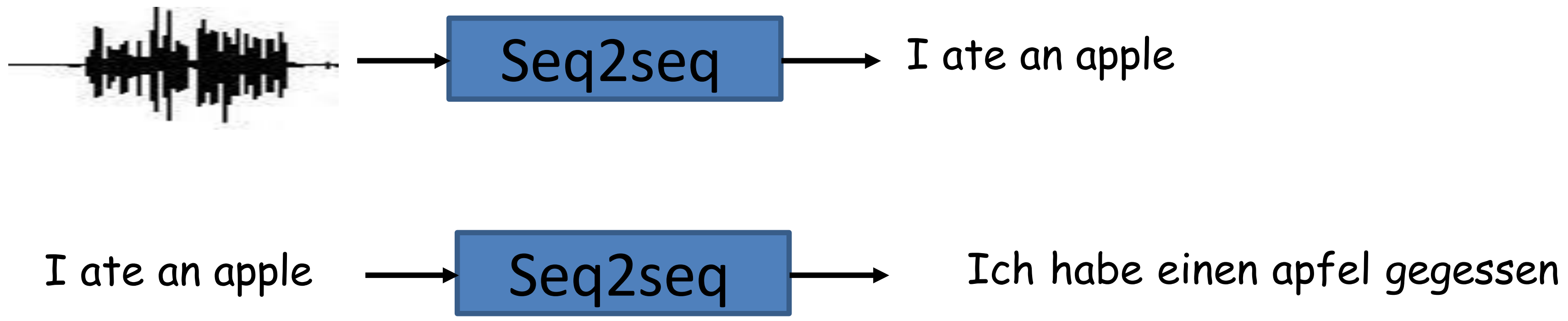
Zijun Cui

Partial slides are adapted from CMU 11-785 Introduction to Deep Learning by Bhiksha Raj

Outline

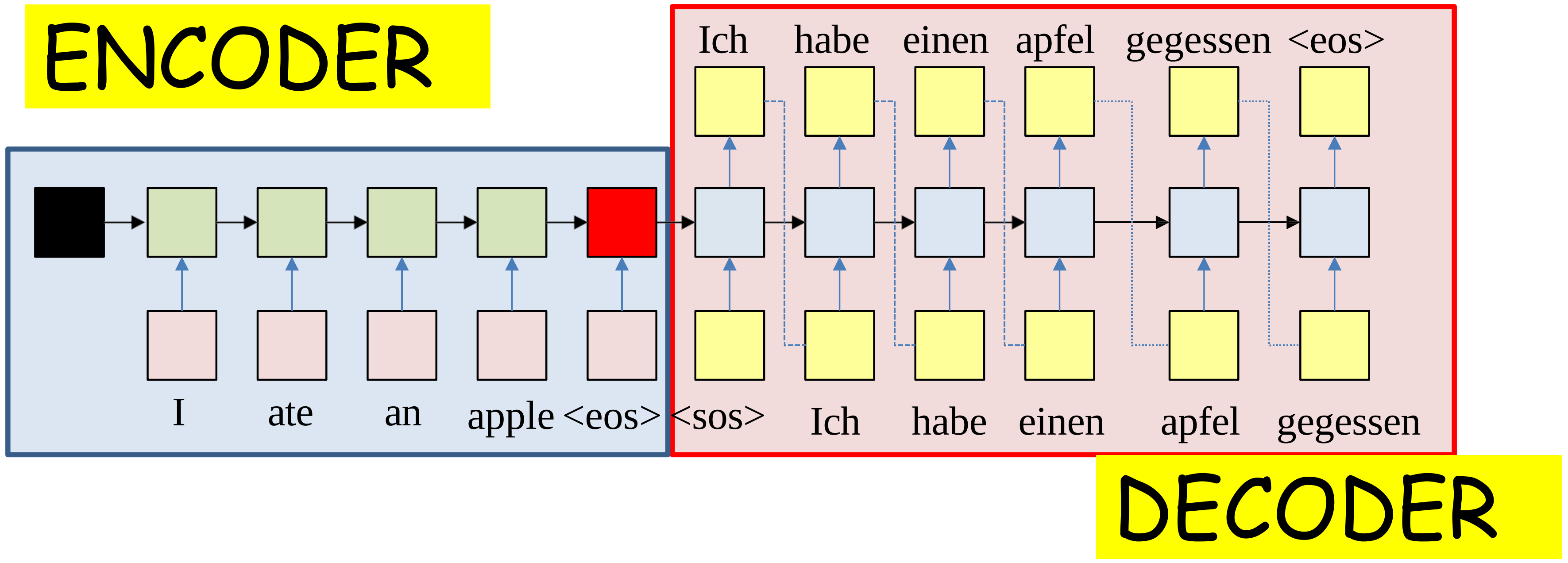
- What is attention
- Attention and Multihead Attention
- Self-attention
- Position Encoding

Sequence to sequence models



- Sequence goes in, sequence comes out

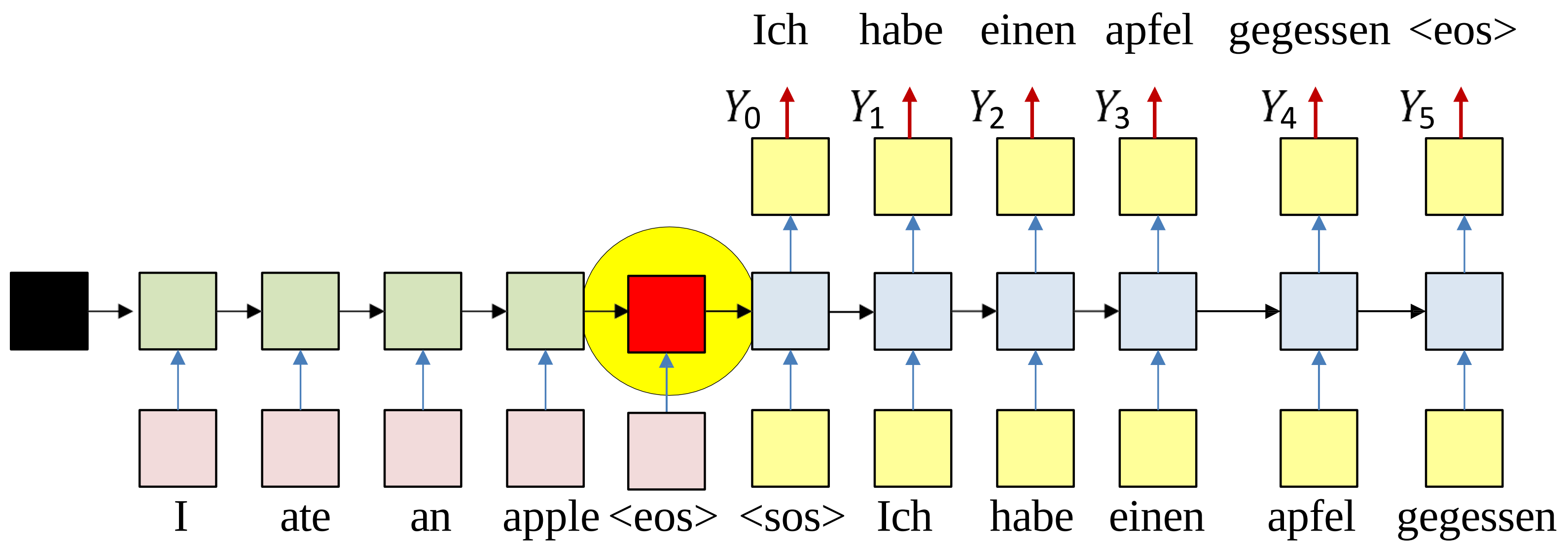
The "simple" translation model



- The recurrent structure that extracts the hidden representation from the input sequence is the *encoder*

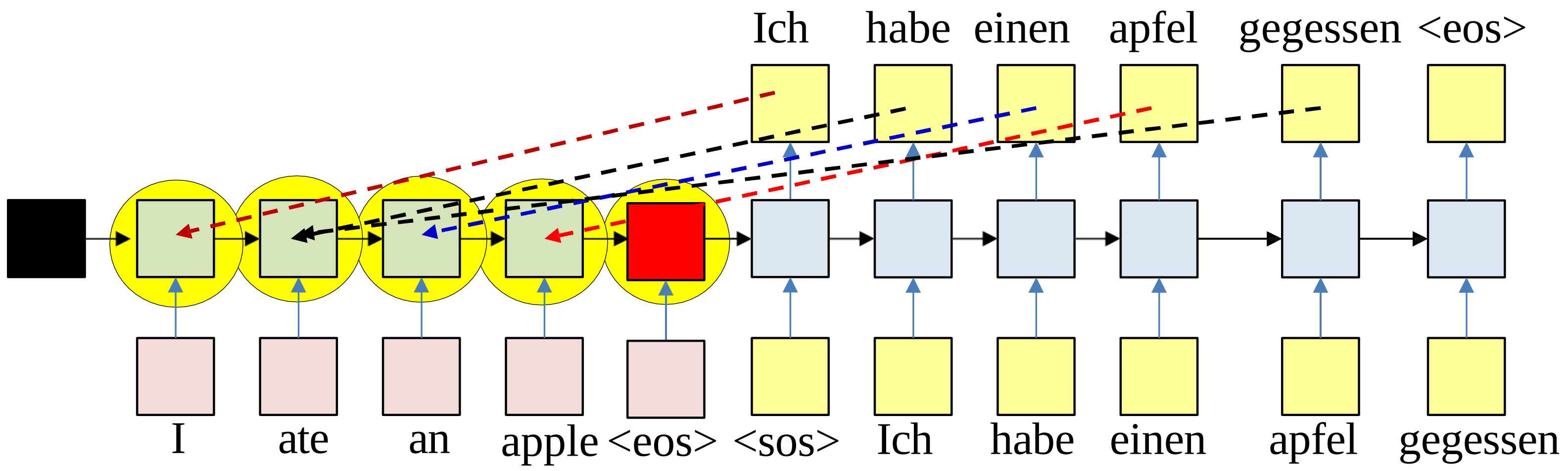
- The recurrent structure that utilizes this representation to produce the output sequence is the *decoder*

A problem with this framework



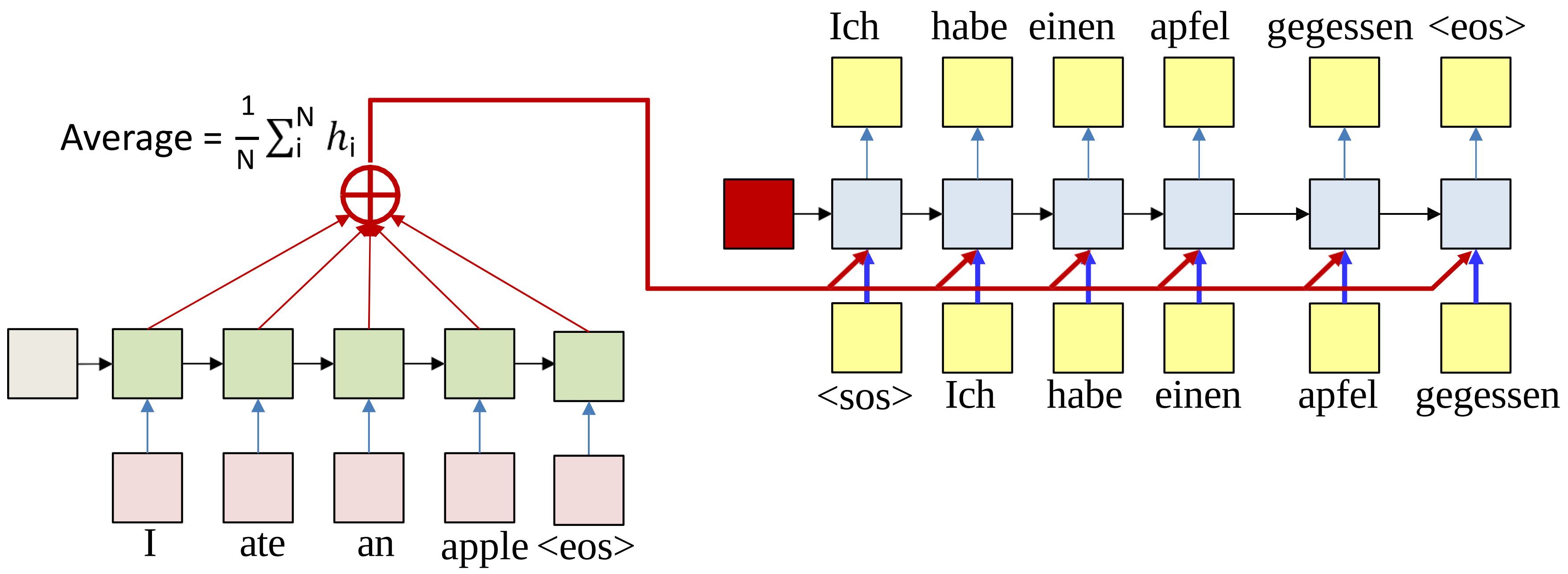
- *All* the information about the input sequence is embedded into a *single* vector
  - The "hidden" node layer at the end of the input sequence
  - This one node is "overloaded" with information
    - Particularly if the input is long

A problem with this framework



- In reality: *All* hidden values carry information
  - Some of which may be diluted by the time we get to the final state of the encoder
- *Every* output is related to the input directly
  - Not sufficient to have the encoder hidden state to *only* the initial state of the decoder
  - Misses the direct relation of the outputs to the inputs

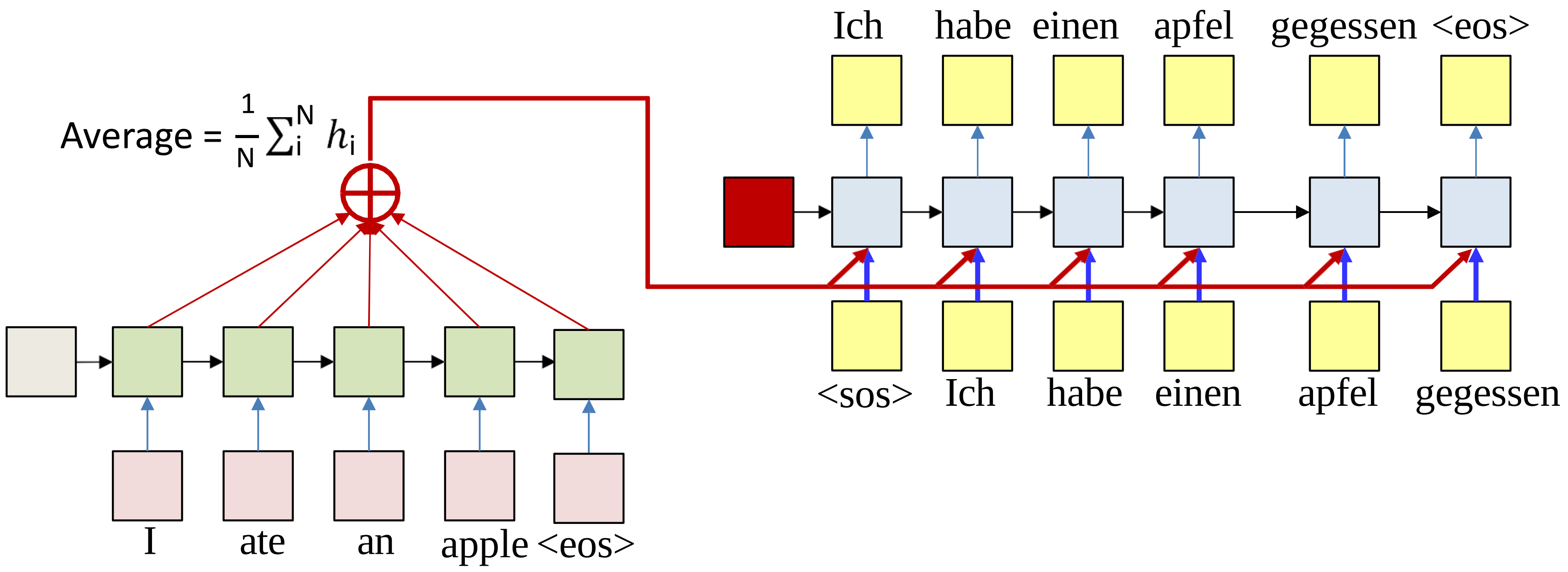
Using all input hidden states



Average = $\frac{1}{N}\sum_{i}^{N} h_i$
Ich habe einen apfel gegessen <eos>
<sos> Ich habe einen apfel gegessen
I ate an apple <eos>

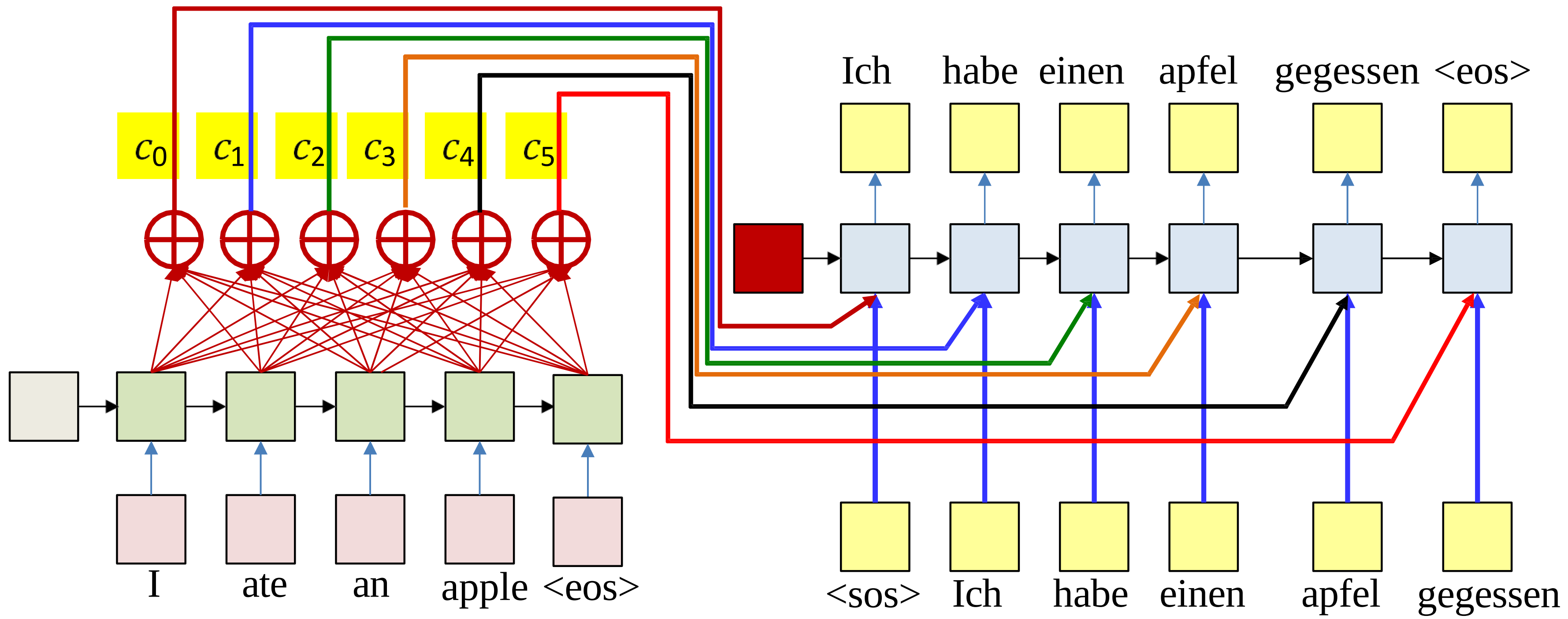
- Simple solution: Compute the average of all encoder hidden states
- Input this average to every stage of the decoder
- The initial decoder hidden state is now separate from the encoder
  - And may be a learnable parameter

Using all input hidden states



- **Problem:** The average applies the same weight to every input
- It supplies the same average to every output word
- In practice, different outputs may be related to different inputs
  – E.g. “Ich” is most related to “I”, and “habe” and “gegessen” are both most related to “ate”

Using all input hidden states

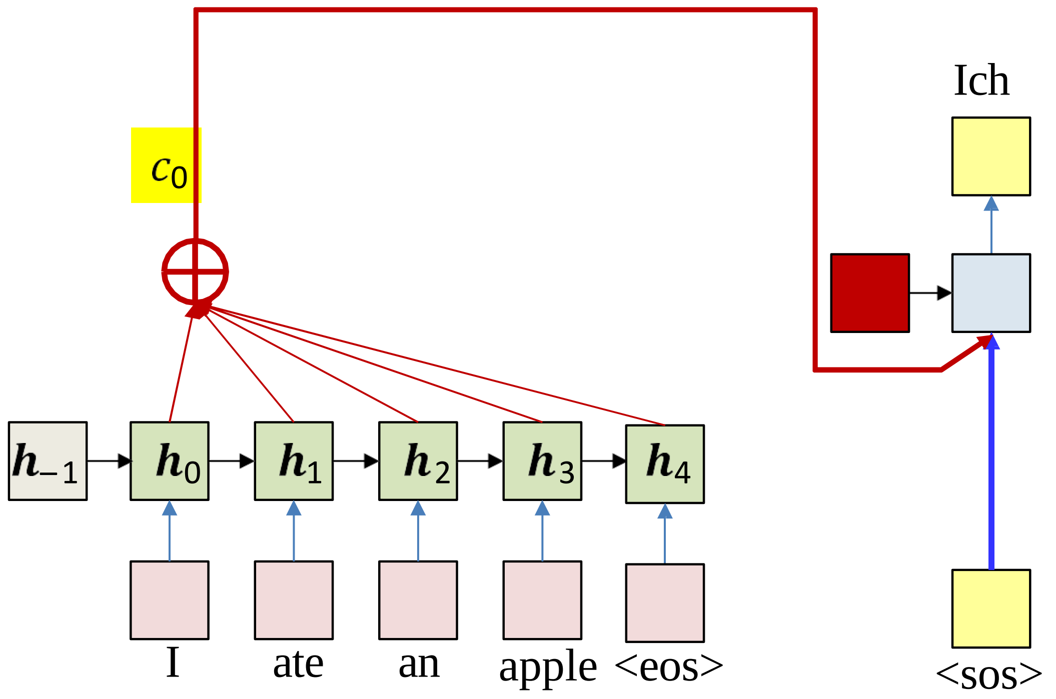


c_0
c_1
c_2
c_3
c_4
c_5
Ich habe einen apfel gegessen <eos>
I ate an apple <eos>
<sos> Ich habe einen apfel gegessen

- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

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

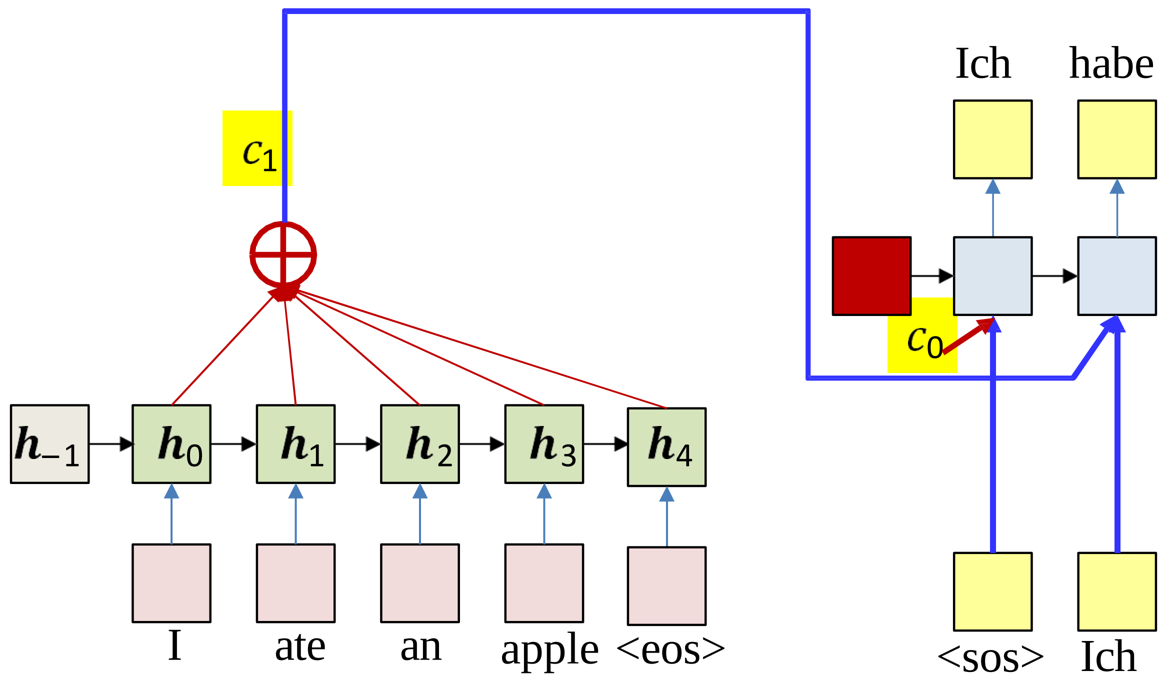
Using all input hidden states



- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

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

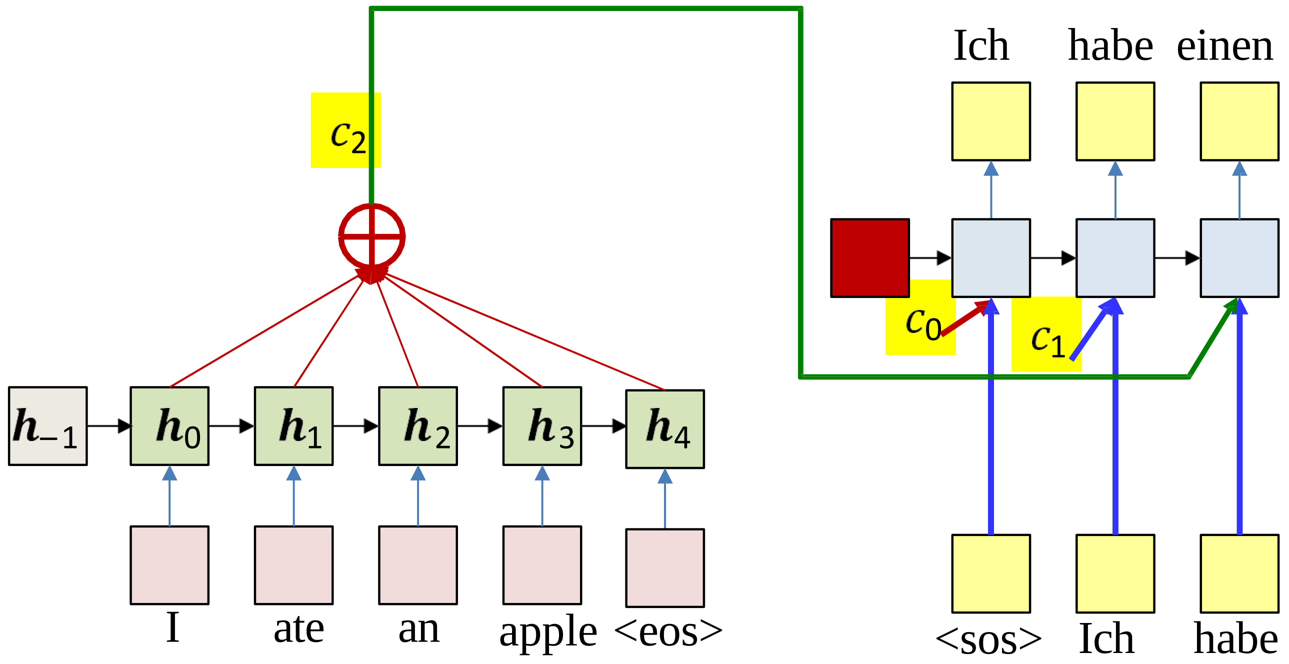
Using all input hidden states



- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

$$c_1 = \frac{1}{N}\sum_{\mathrm{i}}^{\mathrm{N}} w_{\mathrm{i}}(1)h_{\mathrm{i}}$$

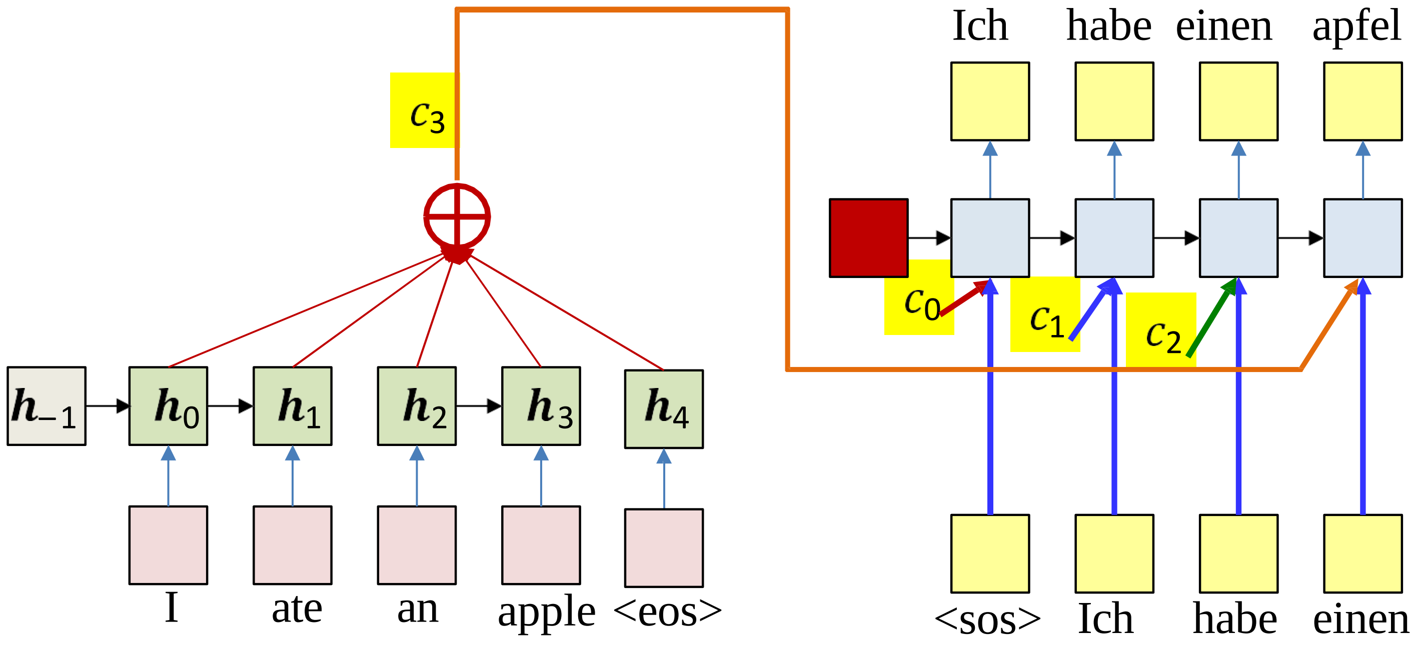
Using all input hidden states



- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

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

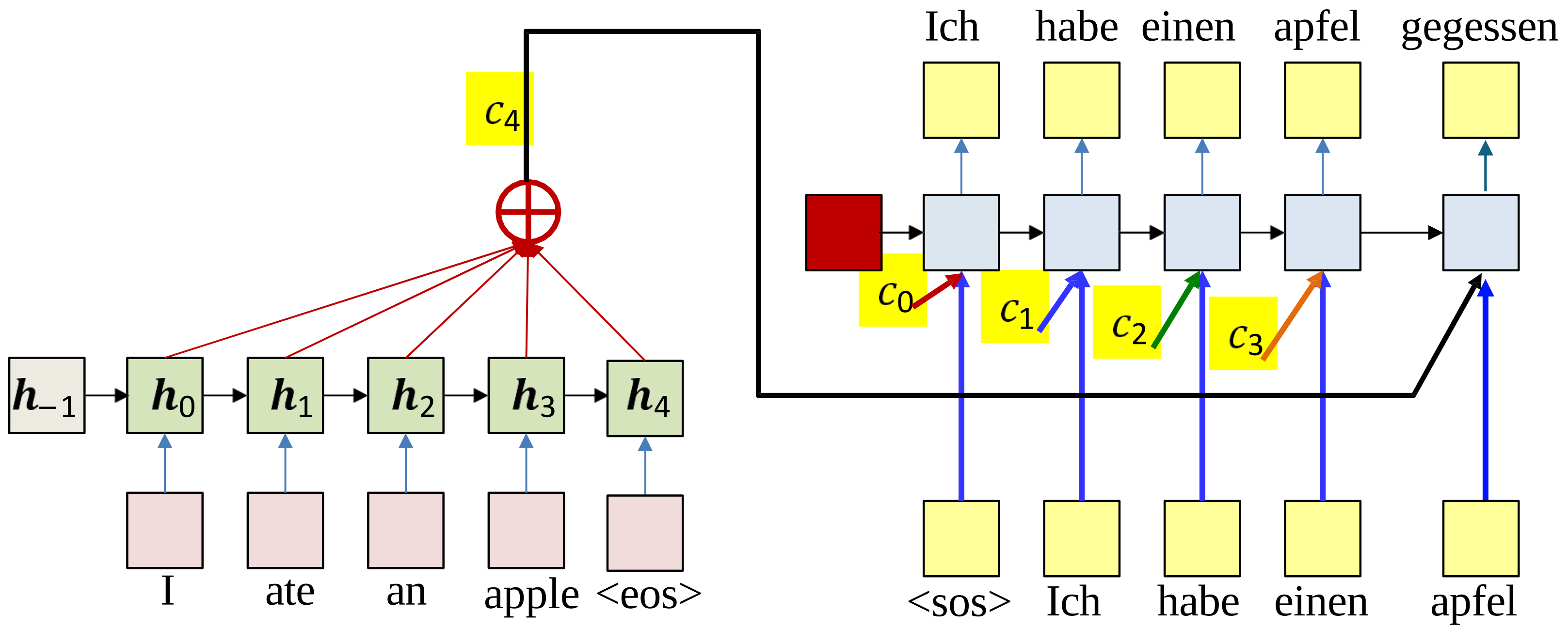
Using all input hidden states



- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

$$c_3 = \frac{1}{N}\sum_{i}^{N} w_i(3) h_i$$

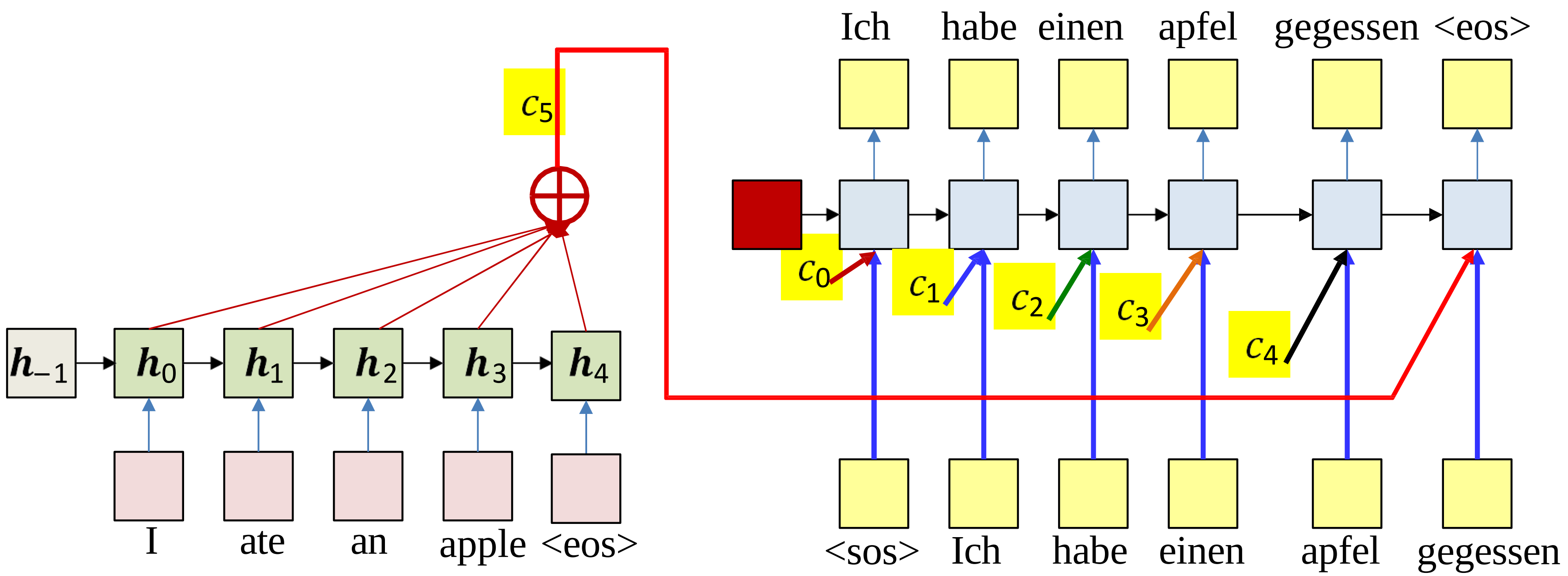
Using all input hidden states



- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

$$c_4 = \frac{1}{N}\sum_{i}^{N} w_i(4)h_i$$

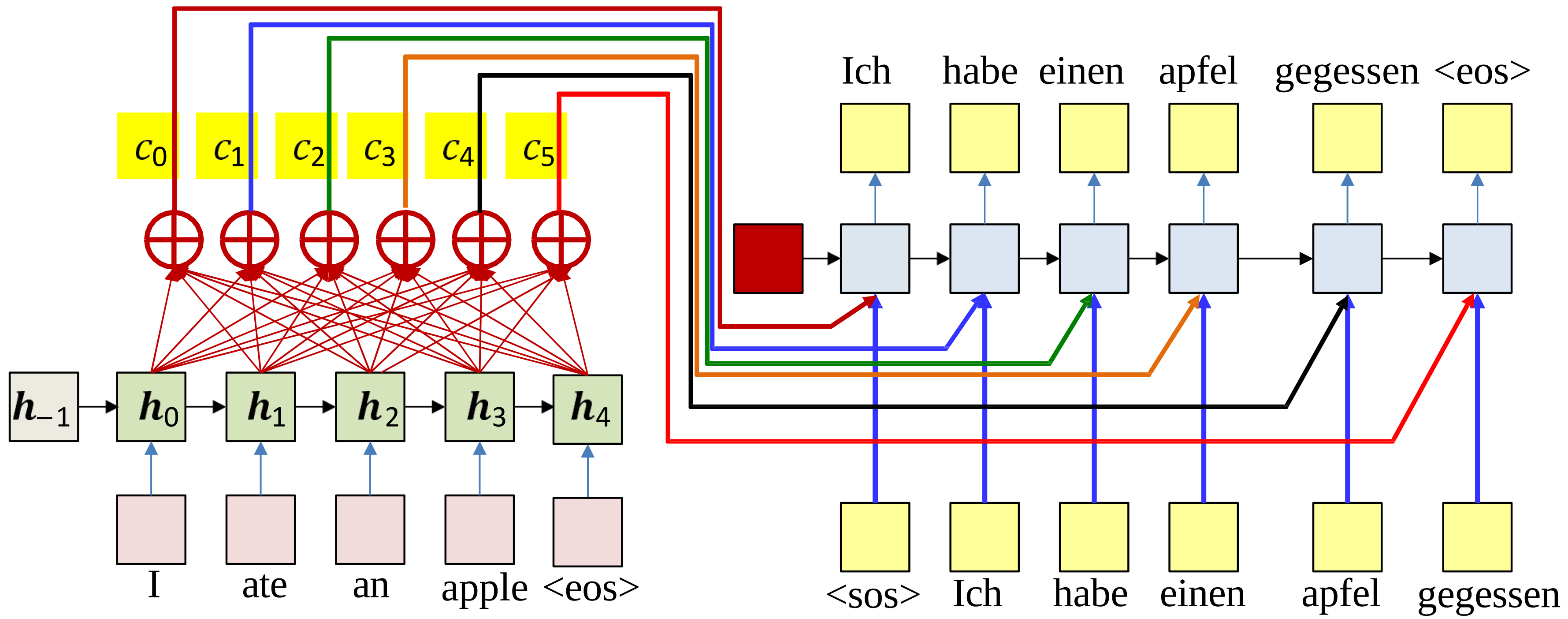
Using all input hidden states



- **Solution:** Use a *different* weighted average for each output word
  - The weighted average provided for the kth output word is:

$$c_5 = \frac{1}{N}\sum_{i}^{N} w_i(5)h_i$$

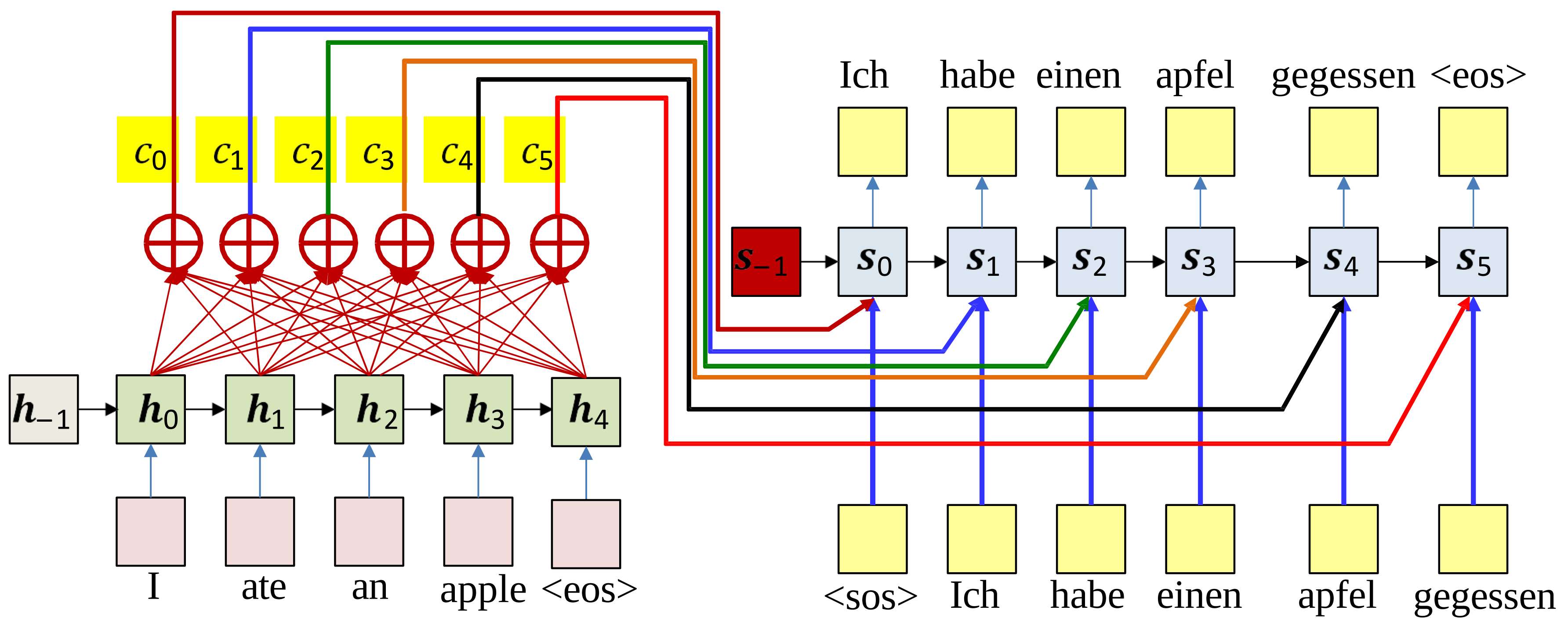
Using all input hidden states



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

- This solution will work if the weights w_{ki} can somehow be made to "focus" on the right input word
  - E.g., when predicting the word "apfel", $w_3(4)$, the weight for "apple" must be high while the rest must be low
- How do we generate such weights??

Attention Models

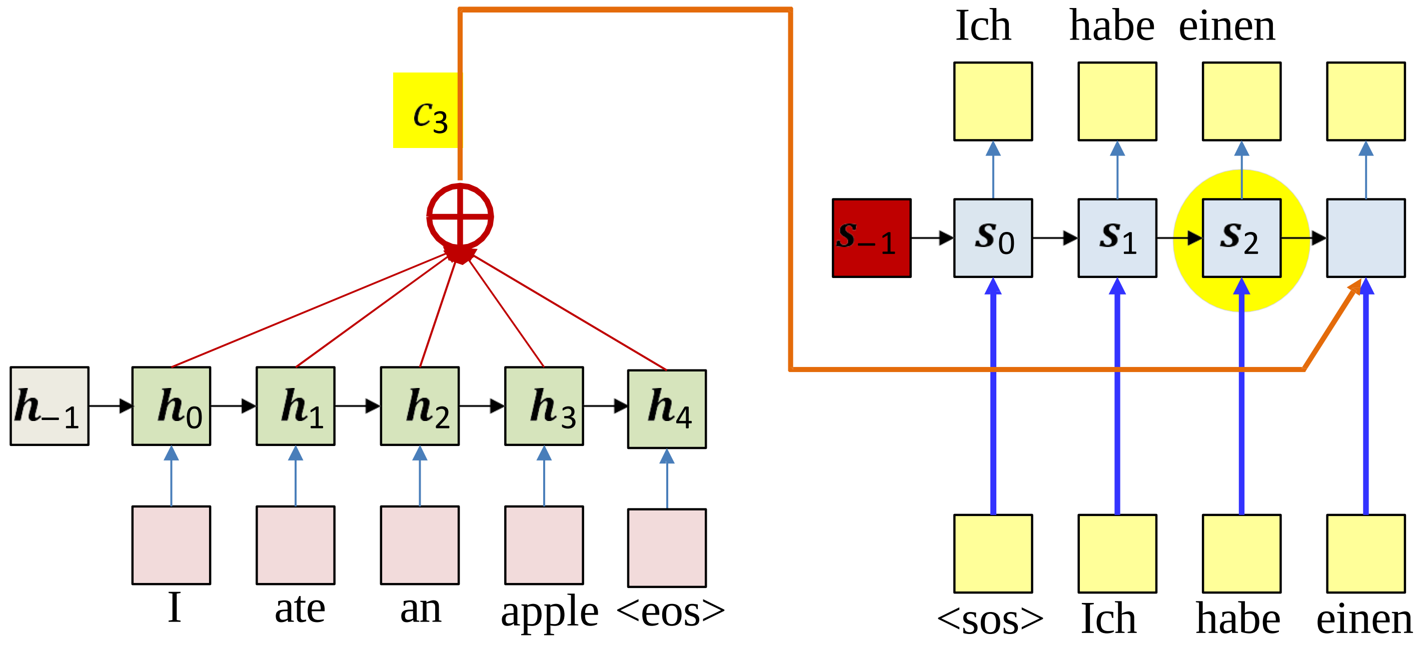


c_0 c_1 c_2 c_3 c_4 c_5
Ich habe einen apfel gegessen <eos>
$\boldsymbol{s}_{-1}$ $\boldsymbol{s}_0$ $\boldsymbol{s}_1$ $\boldsymbol{s}_2$ $\boldsymbol{s}_3$ $\boldsymbol{s}_4$ $\boldsymbol{s}_5$
$\boldsymbol{h}_{-1}$ $\boldsymbol{h}_0$ $\boldsymbol{h}_1$ $\boldsymbol{h}_2$ $\boldsymbol{h}_3$ $\boldsymbol{h}_4$
I ate an apple <eos>
<sos> Ich habe einen apfel gegessen

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

- **Attention weights:** The weights $w_i(t)$ are dynamically computed **as functions of decoder state**
  - Expectation: if the model is well-trained, this will automatically "highlight" the correct input

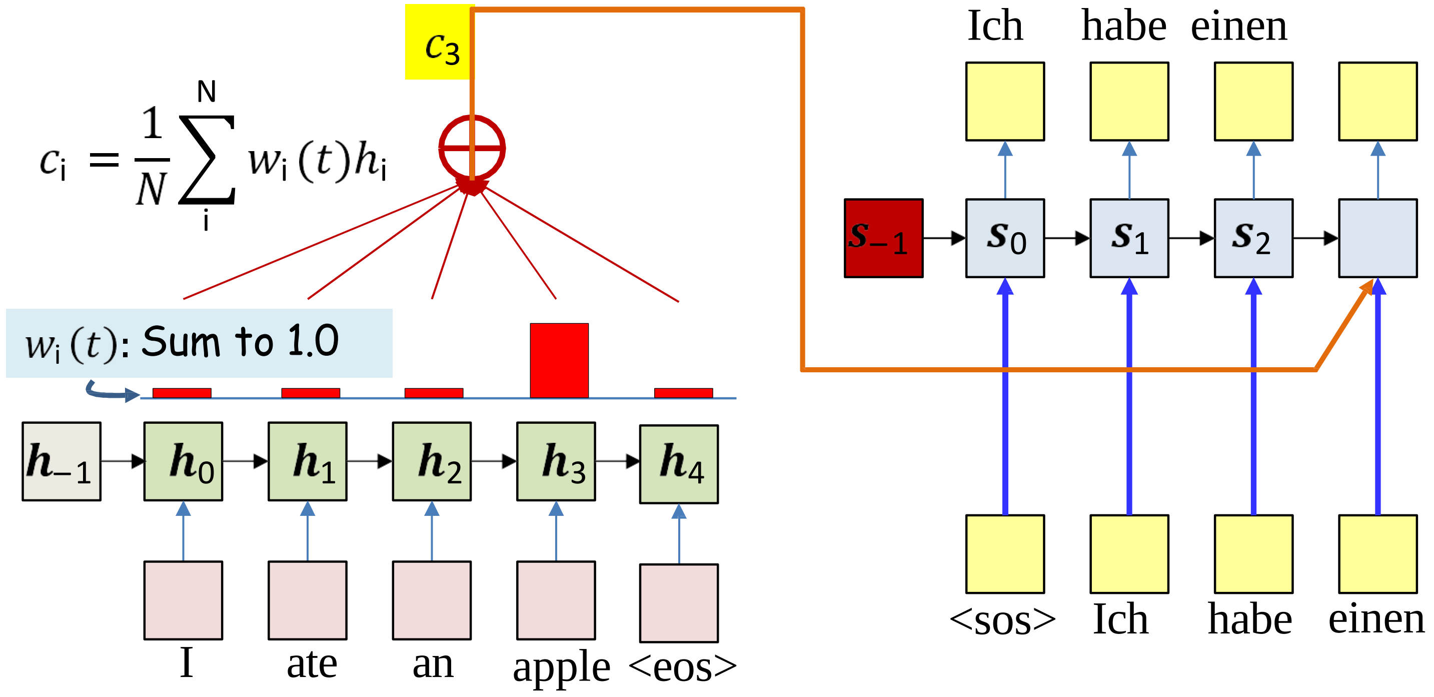
Attention weights at time *t*



- The “attention” weights $w_i(t)$ at time t must be computed from available information at time t
- The primary information is s_{t-1} (the state at time time $t-1$)
  - Also, the input word at time t, but generally not used for simplicity

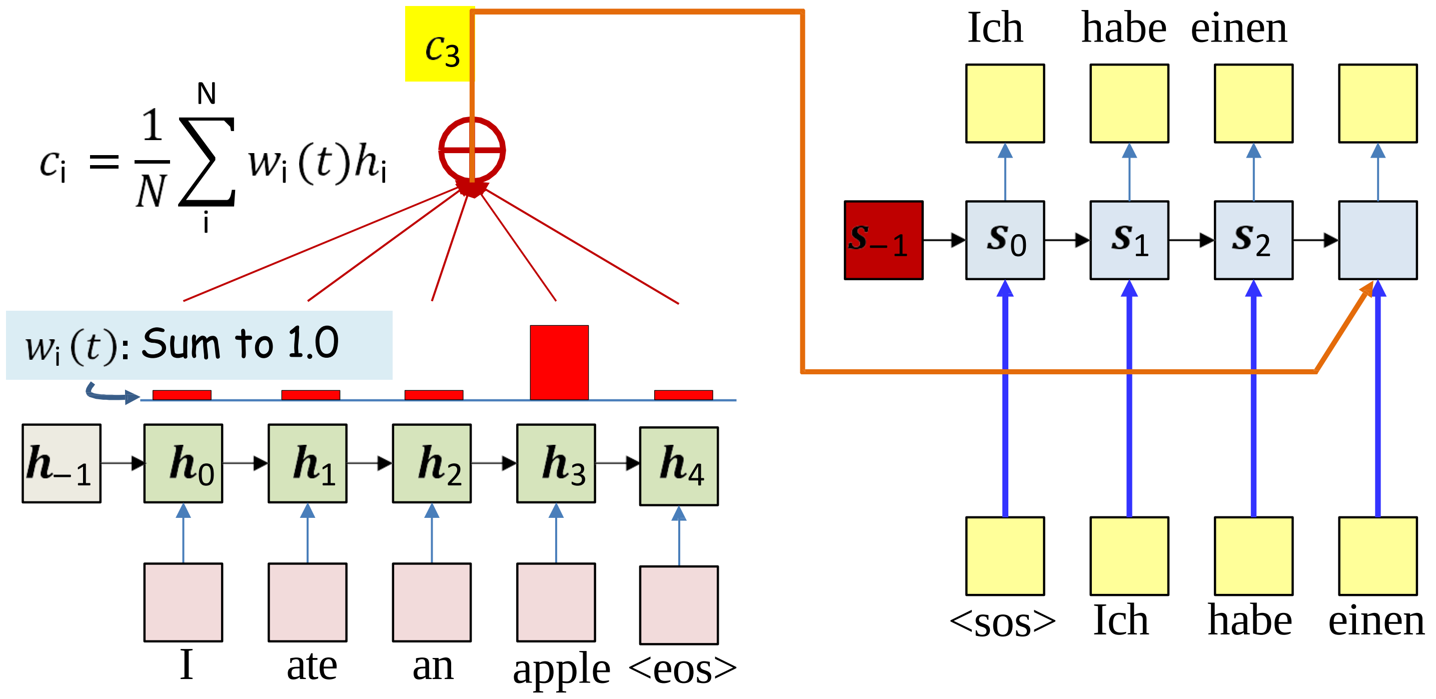
$$w_i(t) = a(\boldsymbol{h}_i, \boldsymbol{s}_{t-1})$$

Requirement on attention weights



- The weights $w_i(t)$ must be positive and sum to 1.0
  - I.e. be a distribution
  - Ideally, they must be high for the most relevant inputs for the ith output and low elsewhere

Requirement on attention weights

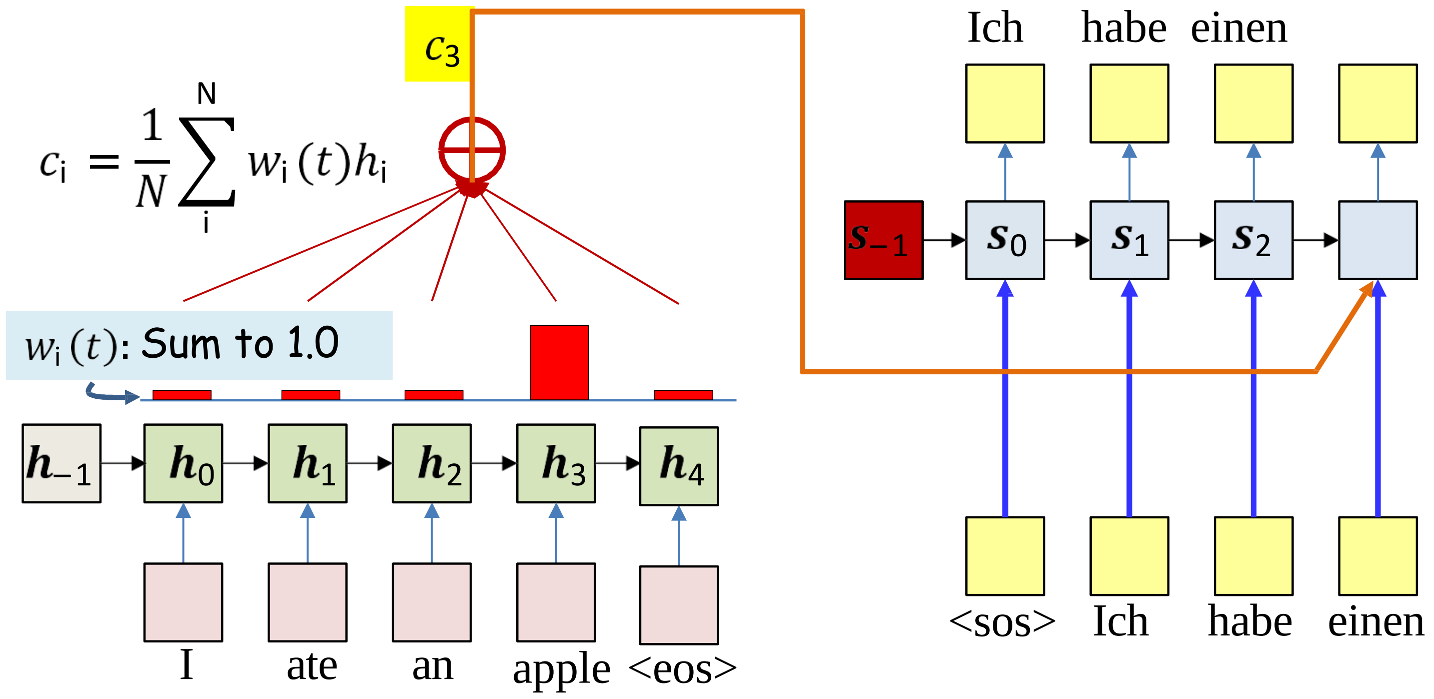


- Solution: A two step weight computation
  - First compute *raw* weights (which could be + or –)
  - Then softmax them to convert them to a distribution

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

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

Attention weights



- Typical options for $g()$ (**variables in red must learned**)

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

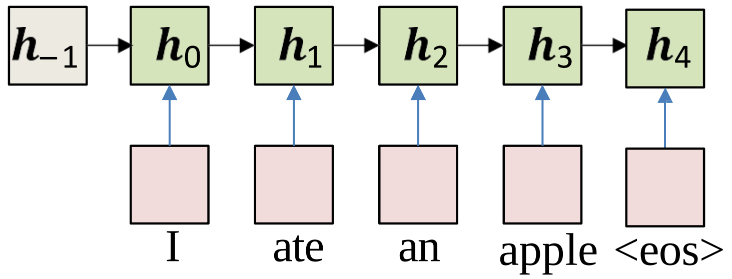
$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_{t-1}$$

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

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

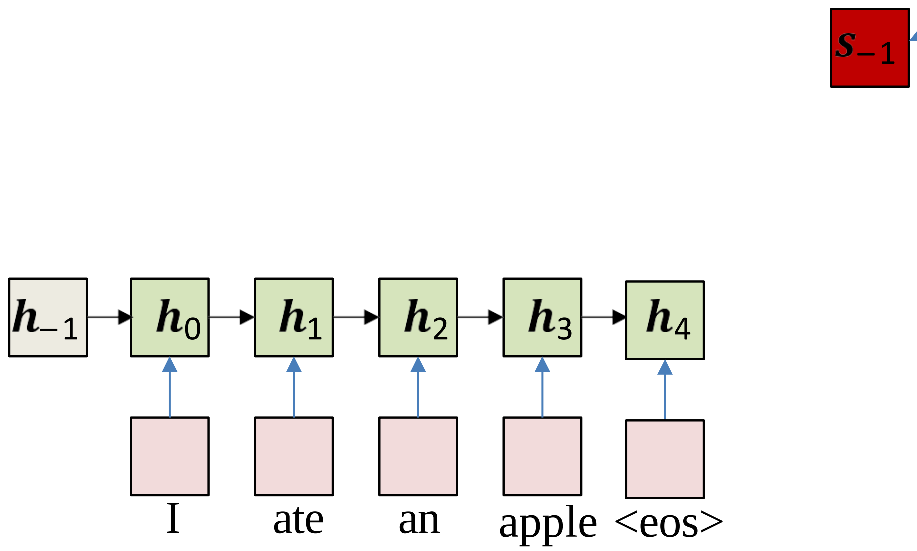
$$e_i(t) = g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1})$$

Converting an input: Inference



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

Converting an input: Inference

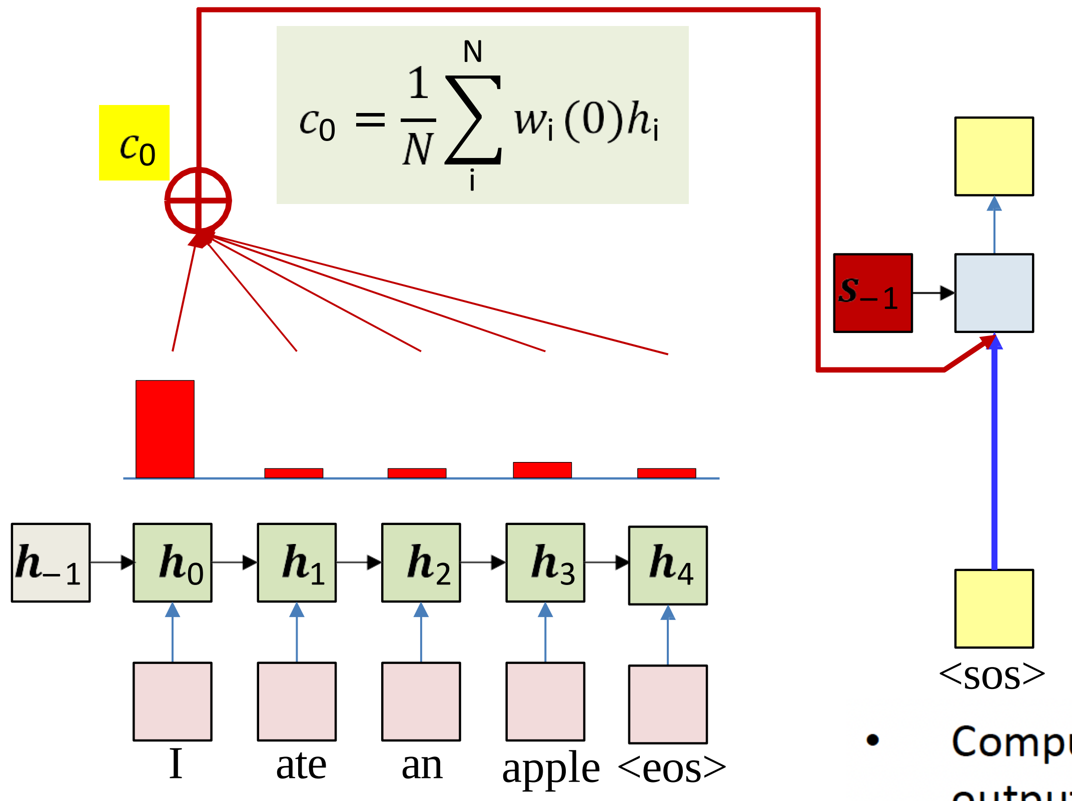


This may be

- a learned parameter, or
- Or just set to some fixed value, e.g. a vector of 1s or 0s, or
- Or the average of all the encoder embeddings: $mean(h_0, \ldots, h_4)$
- Or $W_{\text{init}}\ mean(h_0, \ldots, h_4)$ where W_{init} is a learned parameter

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

Converting an input: Inference



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

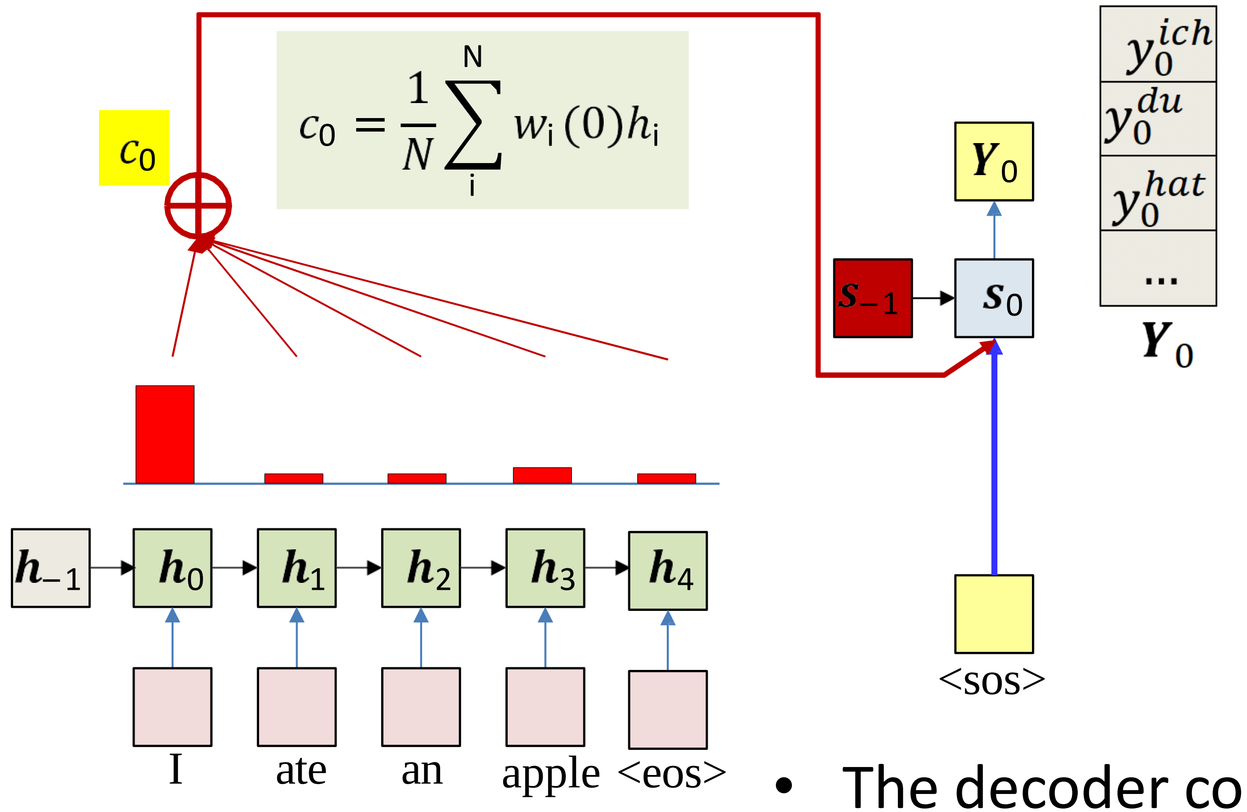
$$g(\boldsymbol{h}_i, \boldsymbol{s}_{-1}) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_{-1}$$

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

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

- 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>

Converting an input: Inference



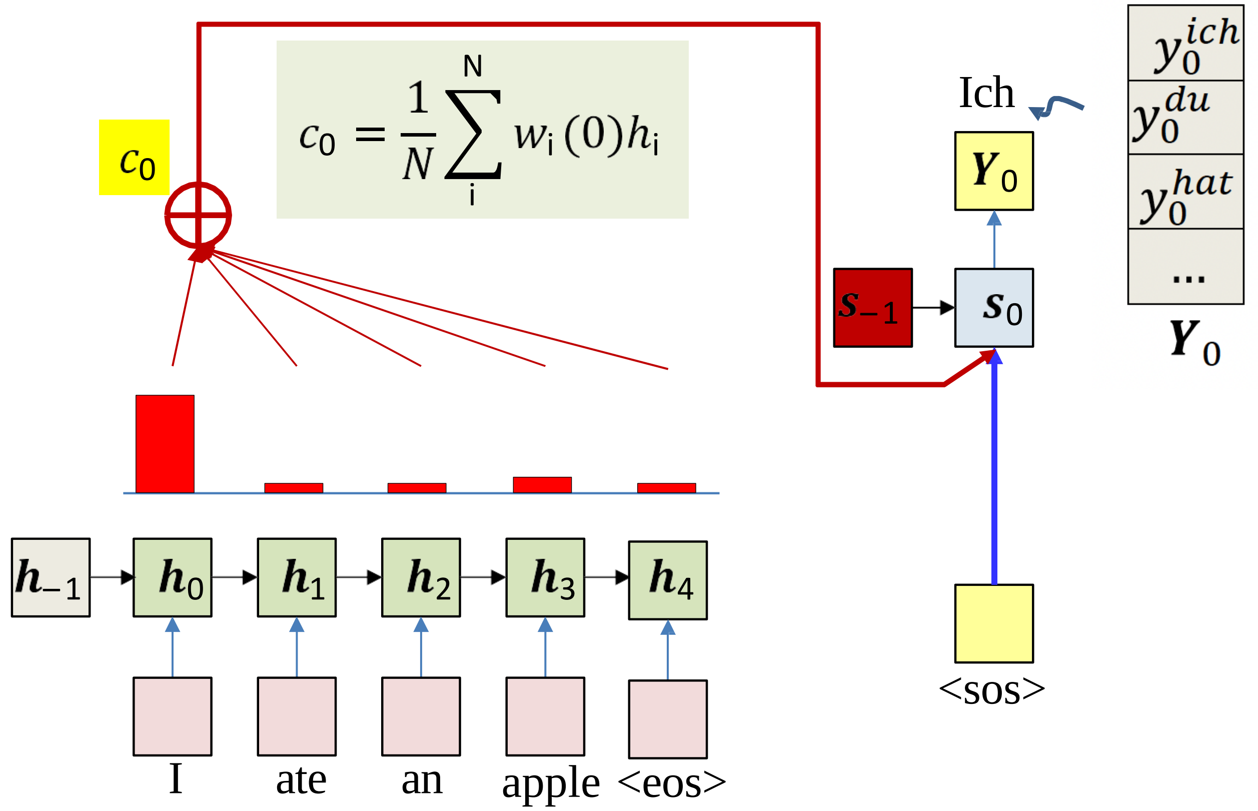
$$g(\boldsymbol{h}_i, \boldsymbol{s}_{-1}) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_{-1}$$

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

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

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

Converting an input: Inference



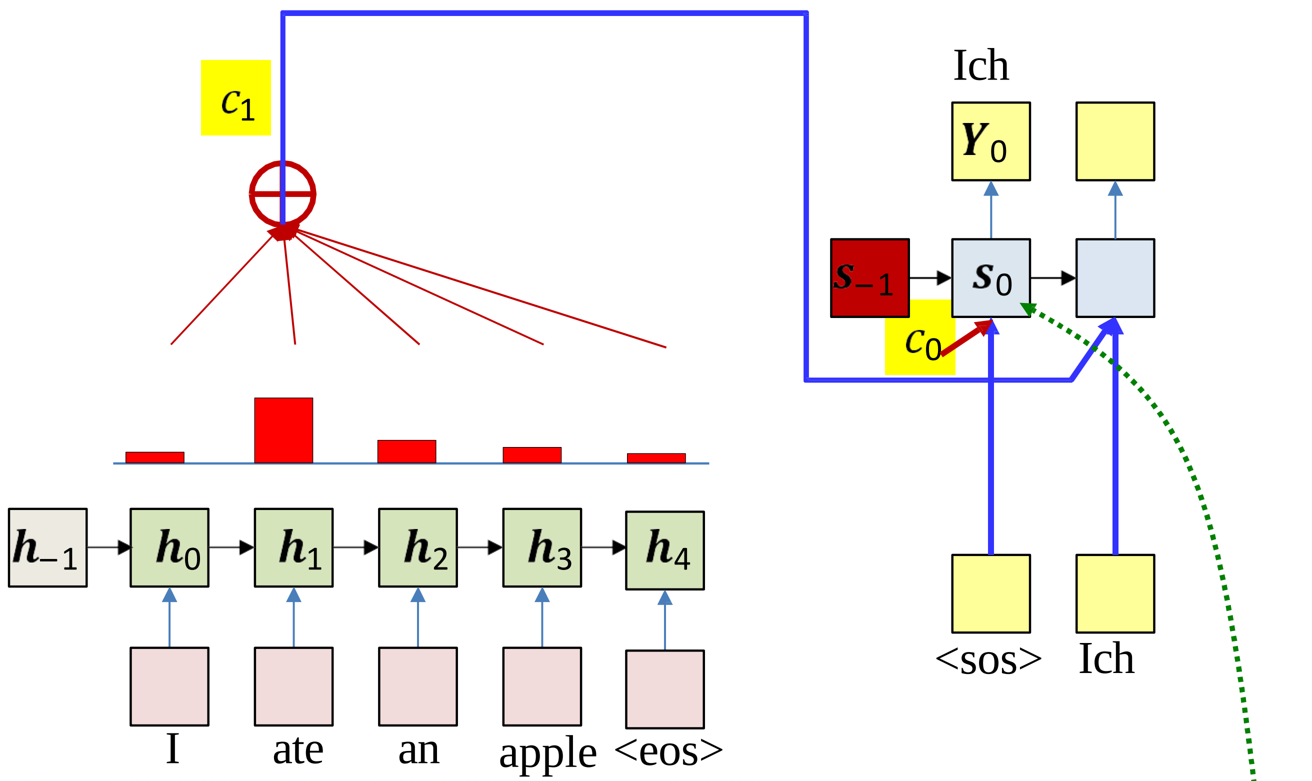
$$g(\boldsymbol{h}_i, \boldsymbol{s}_{-1}) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_{-1}$$

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

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

- Sample a word from the output distribution

Converting an input: Inference



- 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

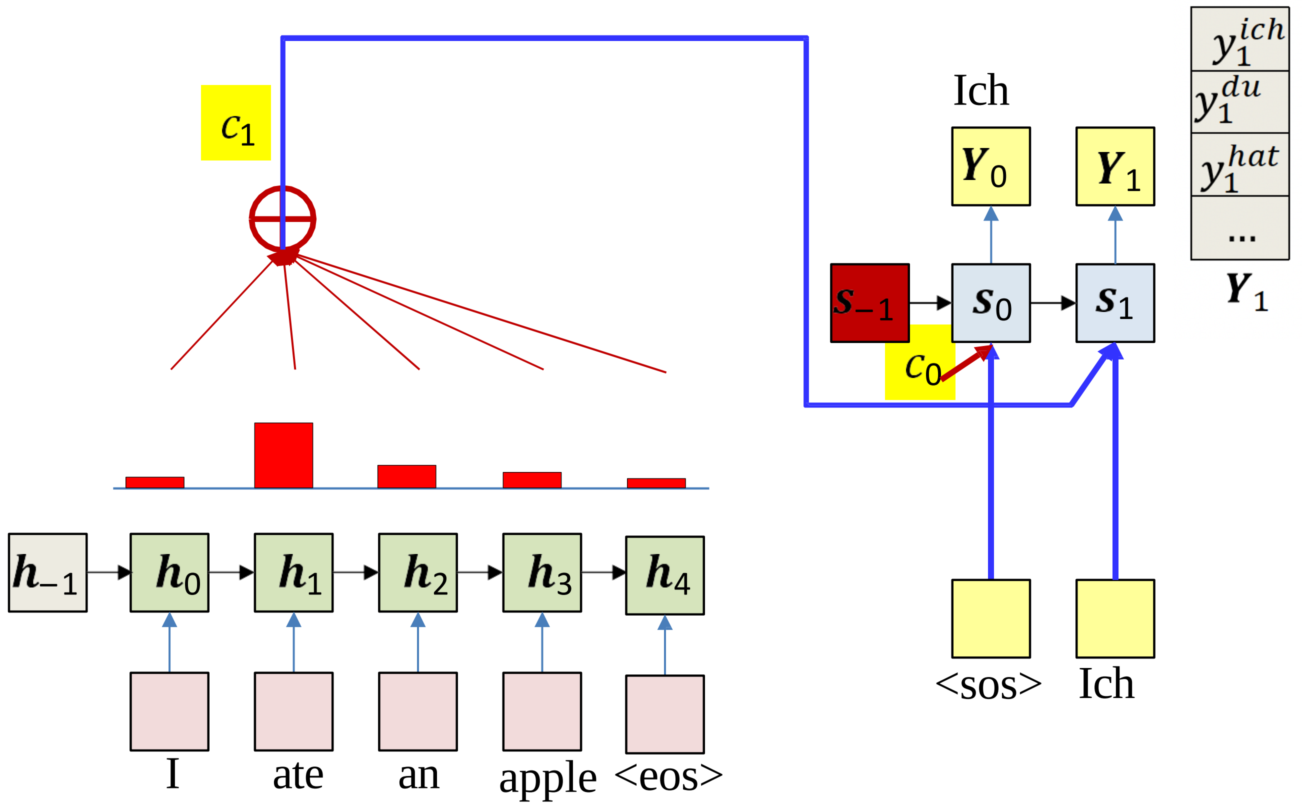
$$g(\boldsymbol{h}_i, \boldsymbol{s}_0) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_0$$

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

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

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

Converting an input: Inference



- The decoder computes
  - s_1
  - A probability distribution over the output vocabulary

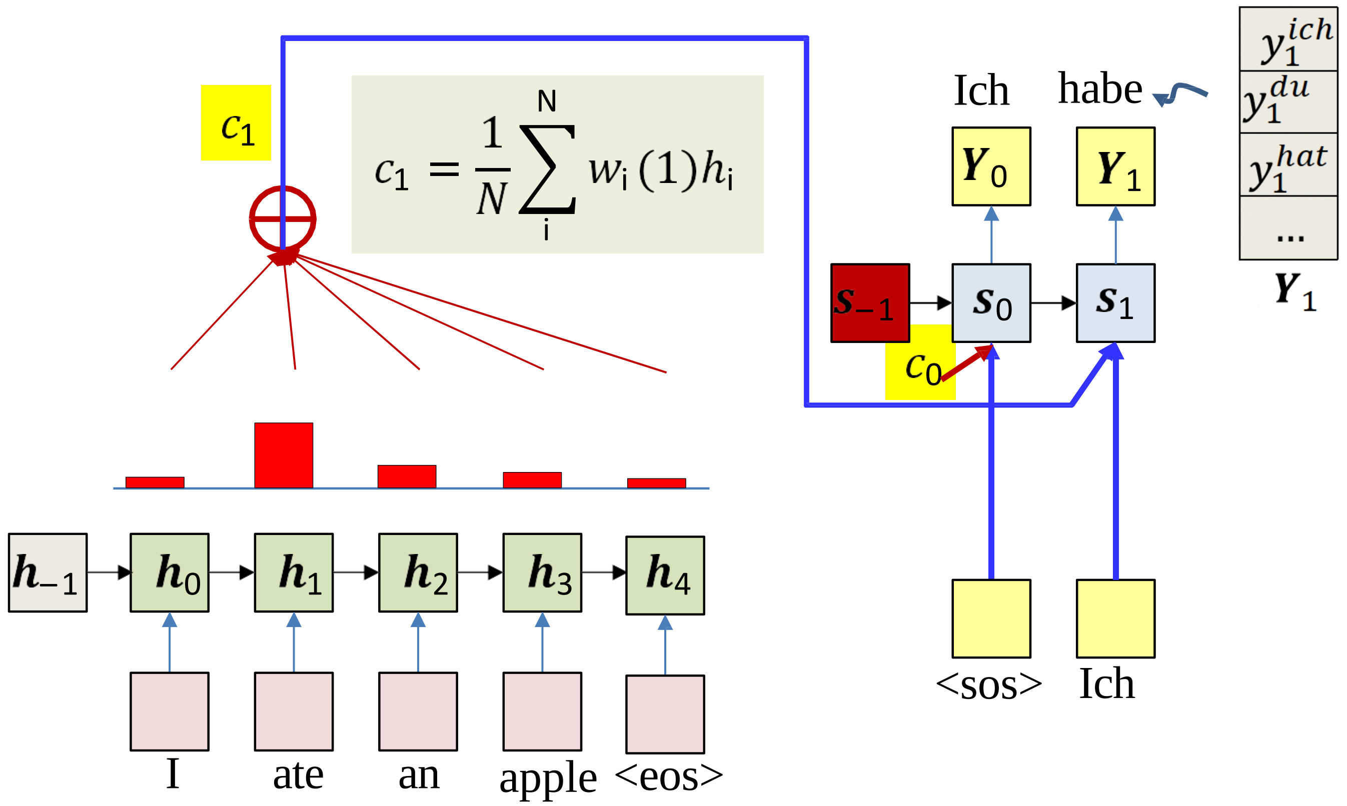
$$g(\boldsymbol{h}_i, \boldsymbol{s}_0) = \boldsymbol{h}_i^\top \boldsymbol{W}_g \boldsymbol{s}_0$$

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

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

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

Converting an input: Inference



- Sample the second word from the output distribution

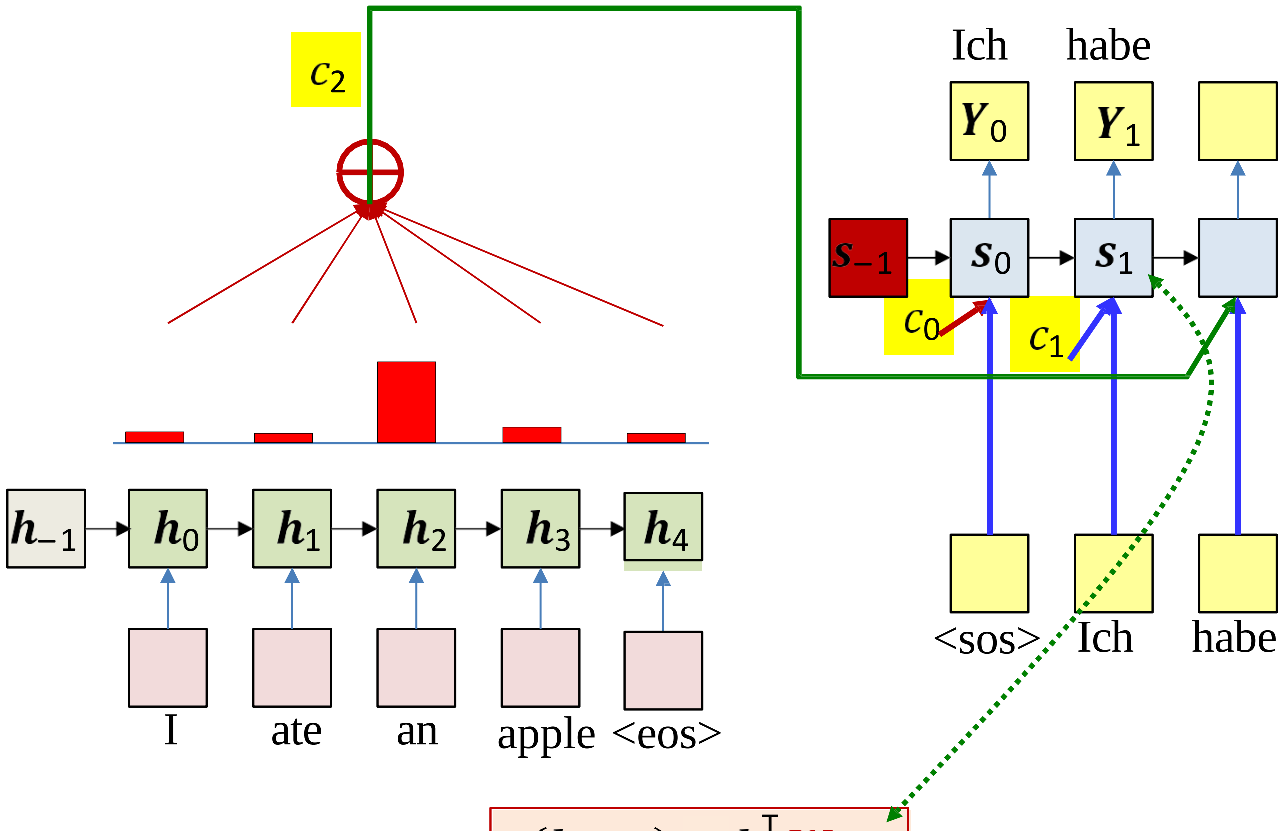
$$g(\mathbf{h}_i, \mathbf{s}_0) = \mathbf{h}_i^T \mathbf{W}_g \mathbf{s}_0$$

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

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

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

Converting an input: Inference



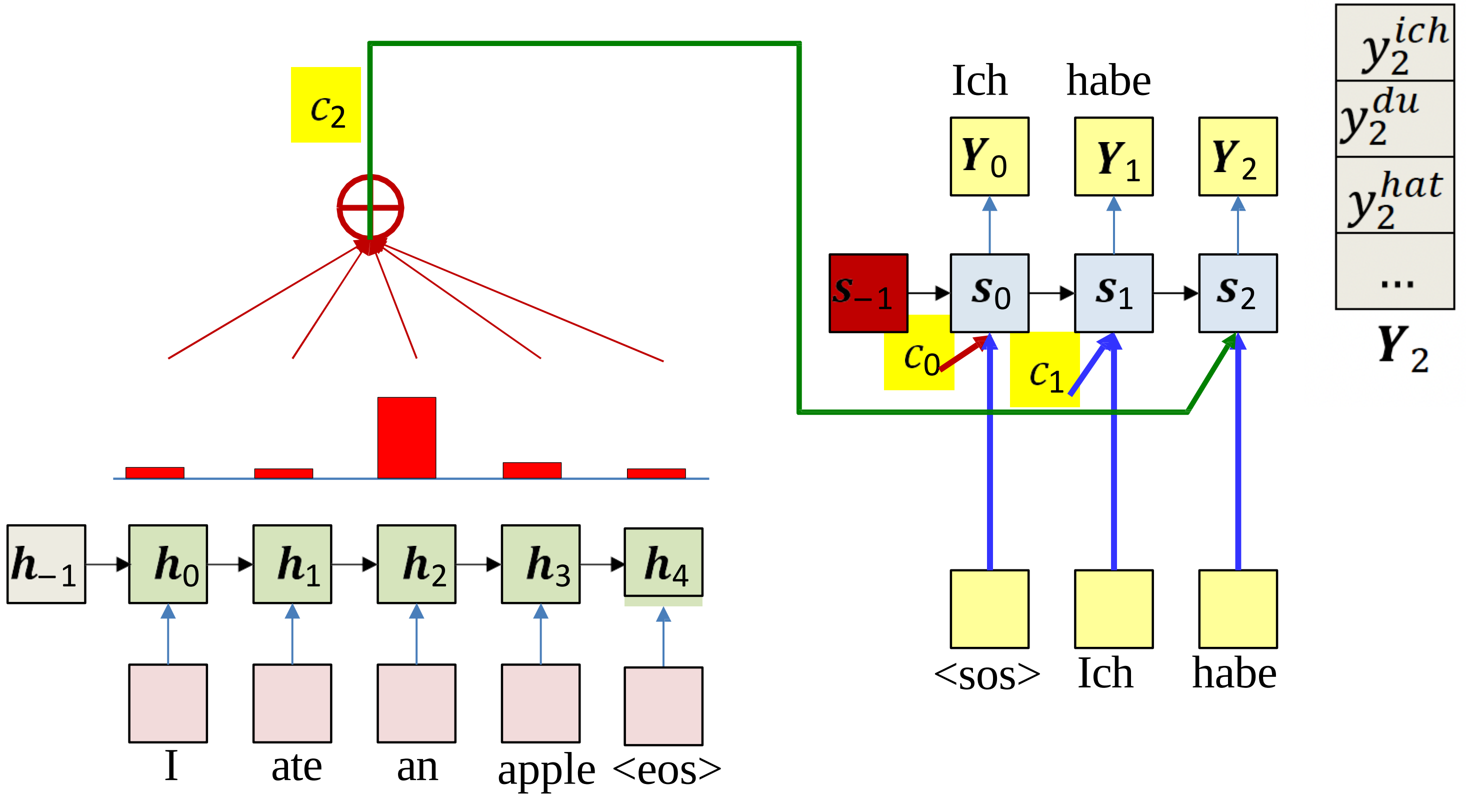
$$g(h_i, s_1) = h_i^\top W_g s_1$$

$$e_i(2) = g(h_i, s_1)$$

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

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

Converting an input: Inference



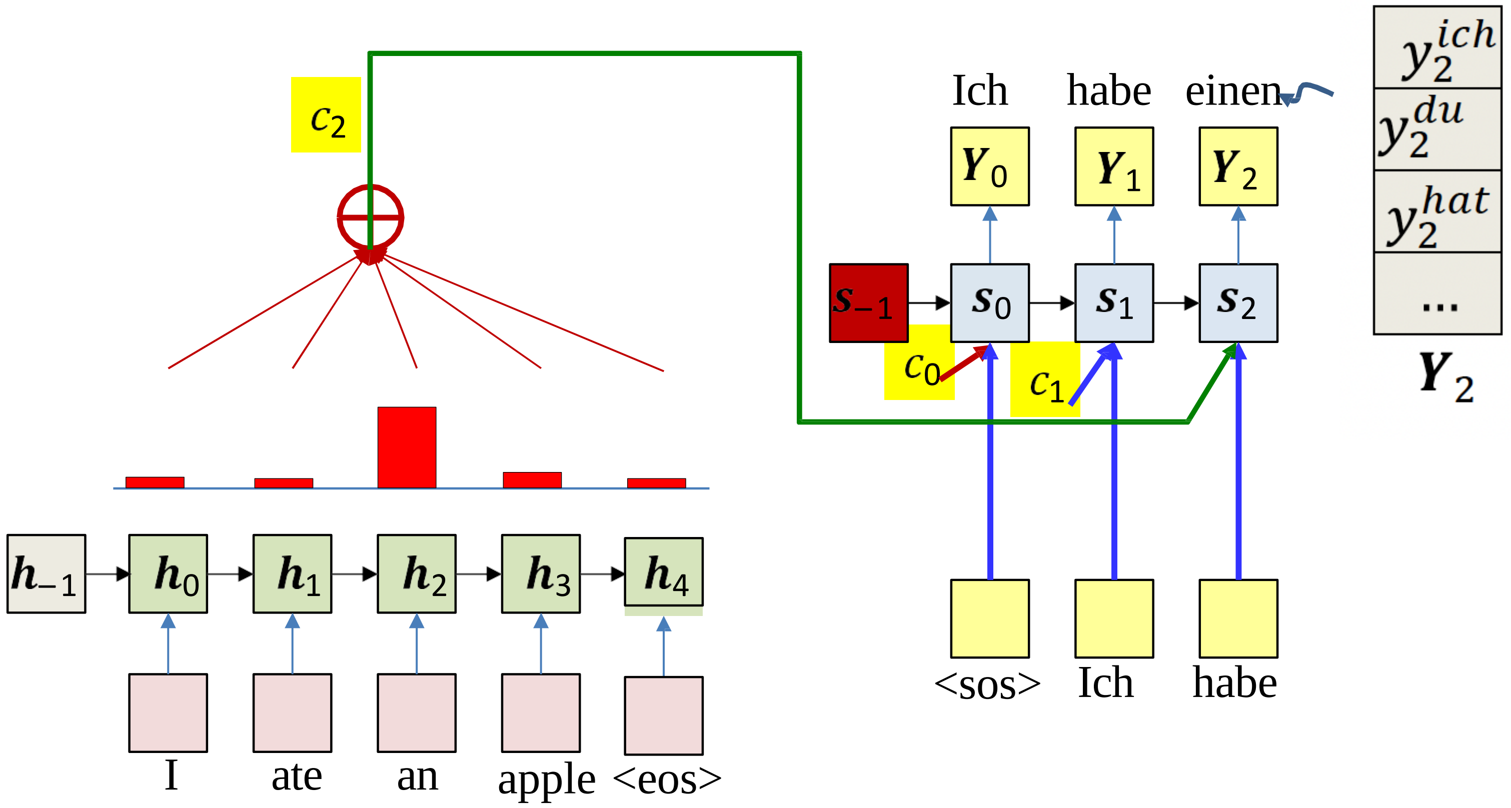
$$g(\boldsymbol{h}_i, \boldsymbol{s}_1) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_1$$

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

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

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

Converting an input: Inference



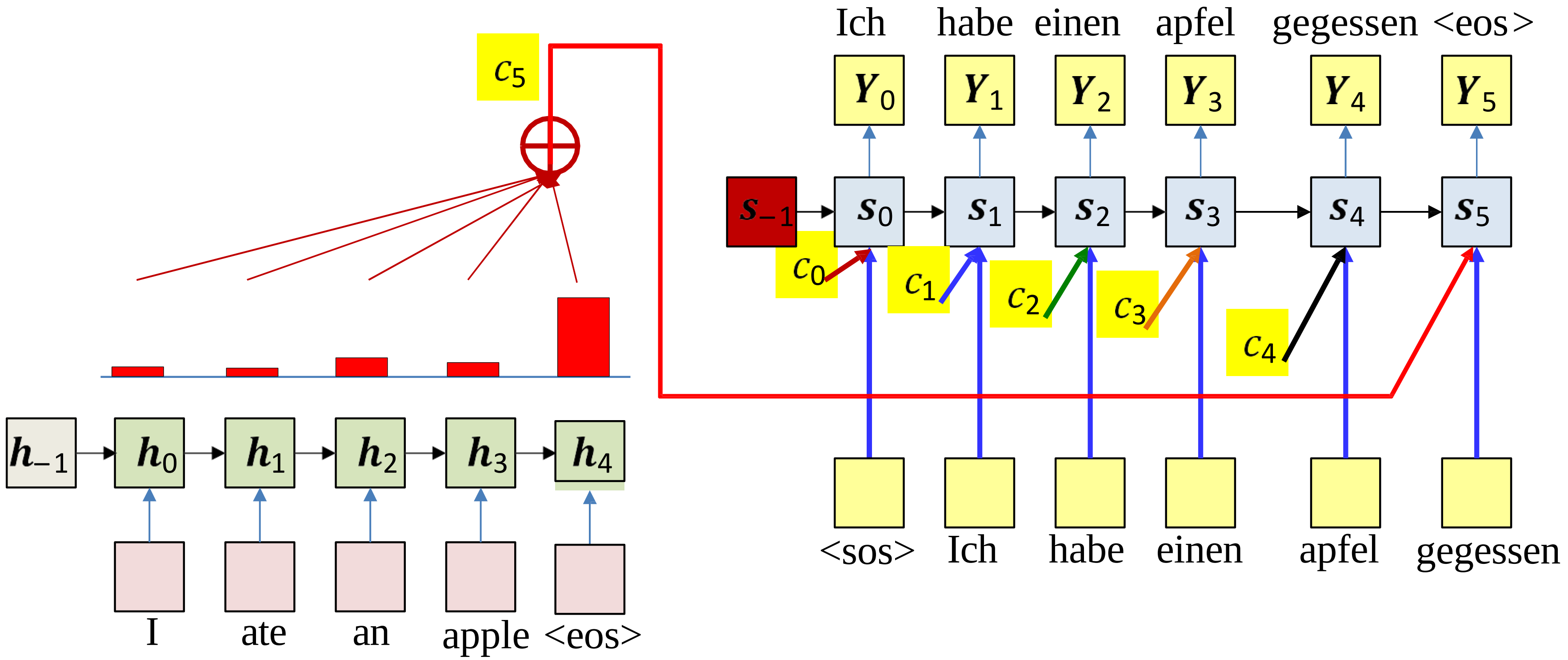
$$g(\boldsymbol{h}_i, \boldsymbol{s}_1) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_1$$

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

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

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

Converting an input: Inference



c_5
Ich habe einen apfel gegessen <eos>
Y_0 Y_1 Y_2 Y_3 Y_4 Y_5
s_{-1} s_0 s_1 s_2 s_3 s_4 s_5
c_0 c_1 c_2 c_3 c_4
<sos> Ich habe einen apfel gegessen
h_{-1} h_0 h_1 h_2 h_3 h_4
I ate an apple <eos>

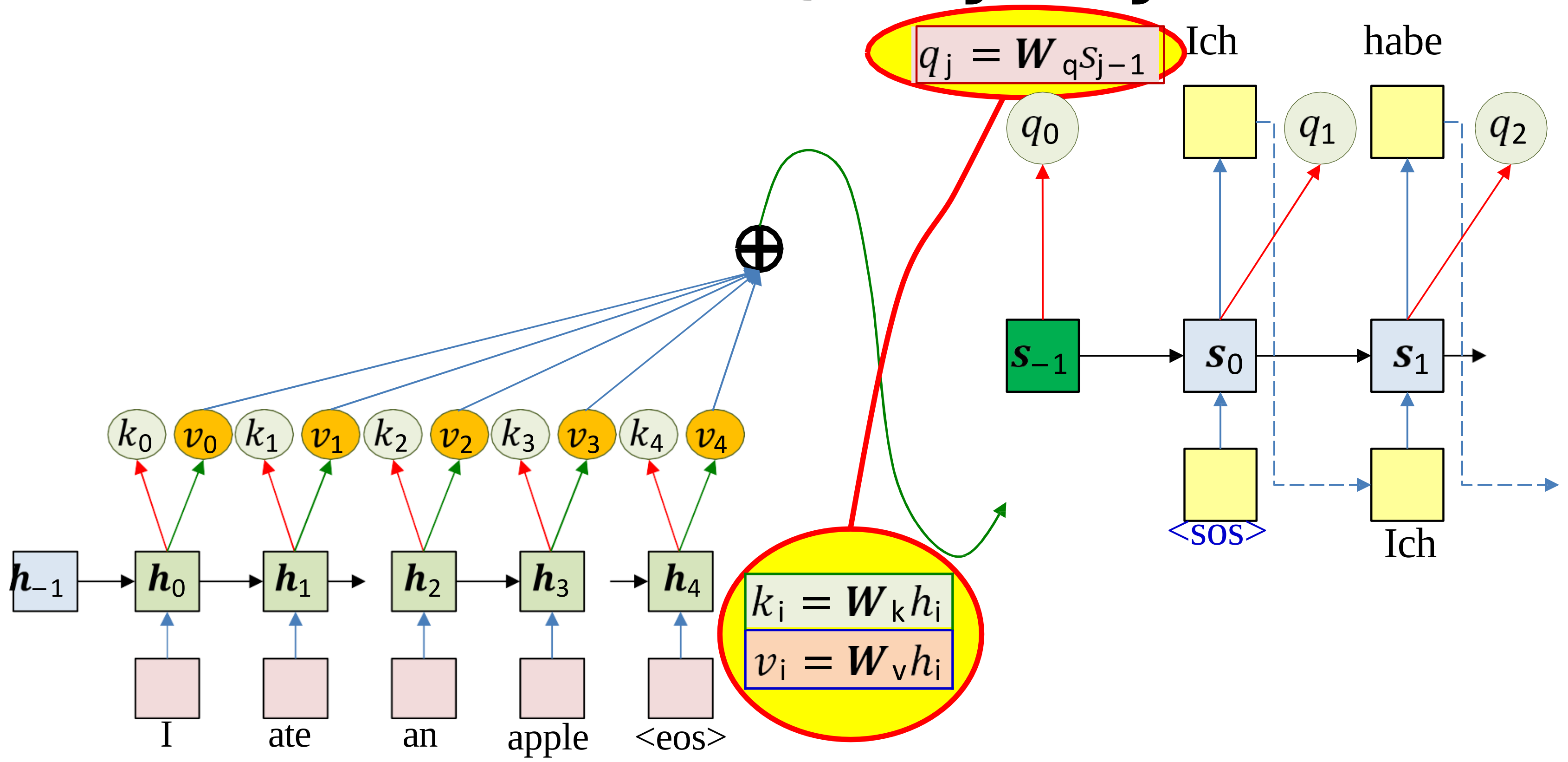
Continue this process until <eos> is drawn

$$e_i(5) = g(\boldsymbol{h}_i, \boldsymbol{s}_4)$$

$$w_i(5) = \frac{\exp(e_i(5))}{\sum_j \exp(e_j(5))}$$

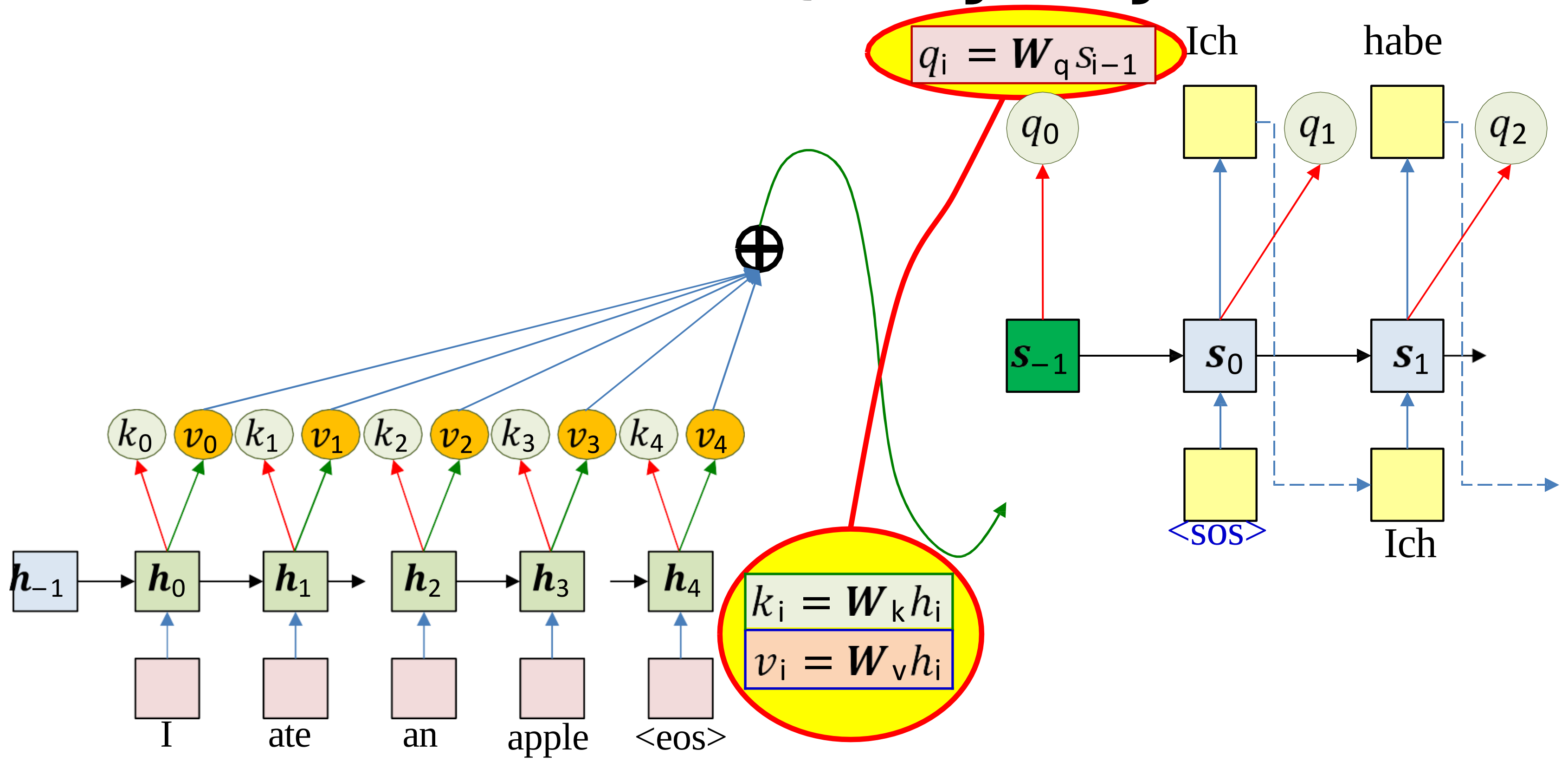
$$c_5 = \frac{1}{N}\sum_i^N w_i(5) h_i$$

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

Modification: Query key value



$q_i = \boldsymbol{W}_q s_{i-1}$
Ich
habe
q_0
q_1
q_2
$\boldsymbol{s}_{-1}$
$\boldsymbol{s}_0$
$\boldsymbol{s}_1$
<sos>
Ich
k_0 v_0 k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4
$\boldsymbol{h}_{-1}$ $\boldsymbol{h}_0$ $\boldsymbol{h}_1$ $\boldsymbol{h}_2$ $\boldsymbol{h}_3$ $\boldsymbol{h}_4$
I
ate
an
apple
<eos>
$k_i = \boldsymbol{W}_k h_i$
$v_i = \boldsymbol{W}_v h_i$

$$e\ (t) = g(\boldsymbol{k}\ ,\boldsymbol{q}\ \ )$$

$$w\ (t) = softmax(e\ (t))$$

Input to hidden decoder layer: $\sum_i w_i(t)\boldsymbol{v}_i$

Modification: Query key value

$$e\ (t) = g(\boldsymbol{k}\ , \boldsymbol{q}\ )$$

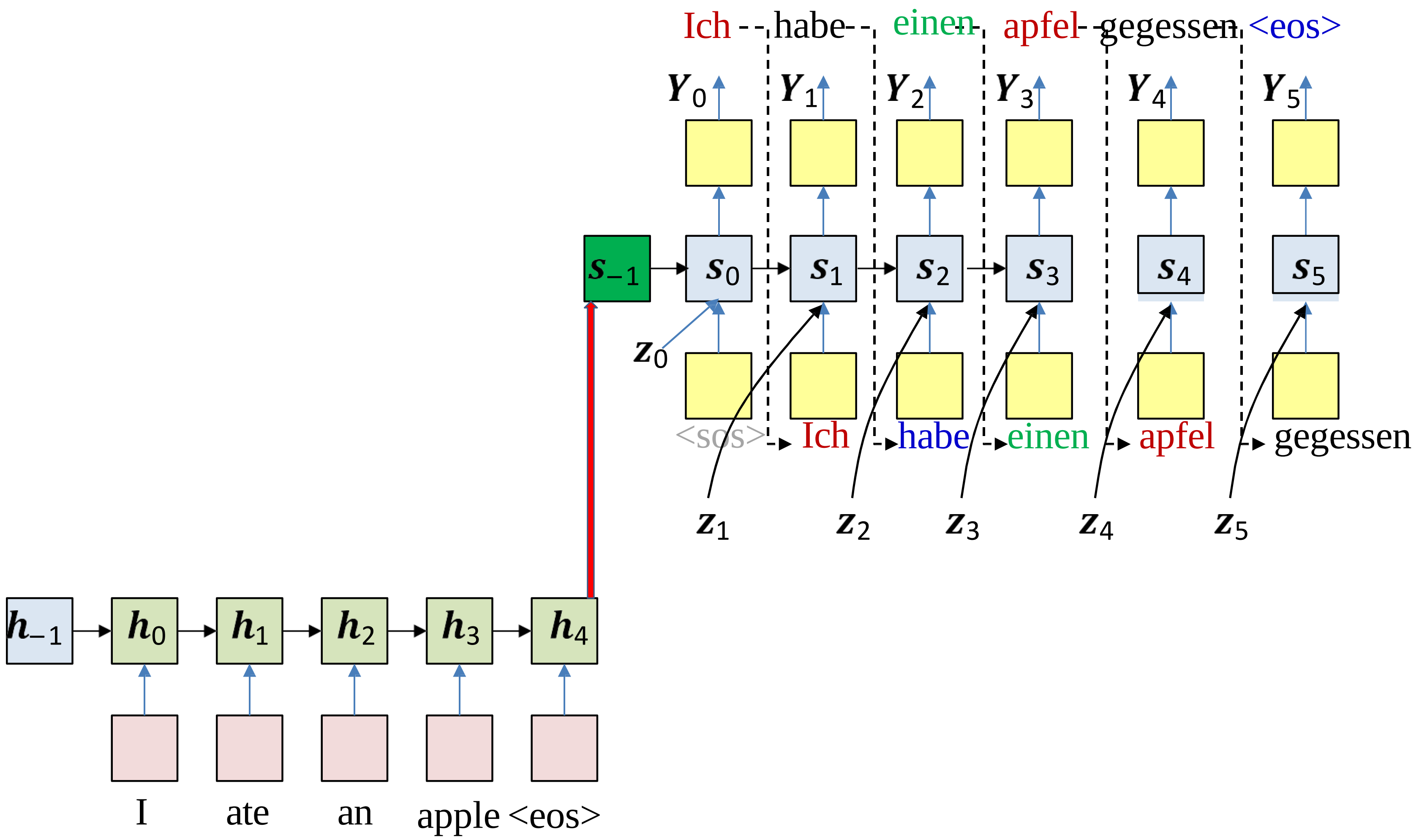
$$w\ (t) = softmax(e\ (t))$$

Input to hidden decoder layer: $\sum_i w_i\,(t)\boldsymbol{v}_i$

Special case: $k_i = v_i = h_i$

$q_t = s_{t-1}$

We will continue using this assumption in the following slides but in practice the query-key-value format is used

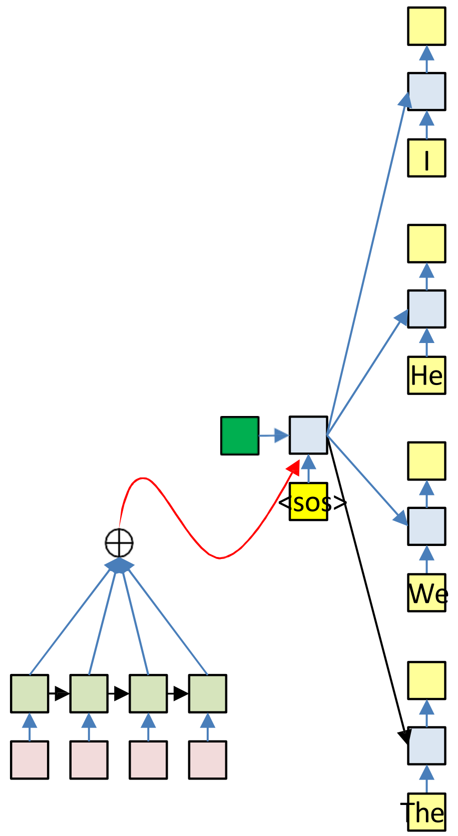


- As before, the objective of drawing: Produce the most likely output (that ends in an <eos>)

$$\underset{O_1,\ldots,O_L}{\text{argmax}}\, y_1^{O_1} y_1^{O_2} \ldots y_1^{O_L}$$

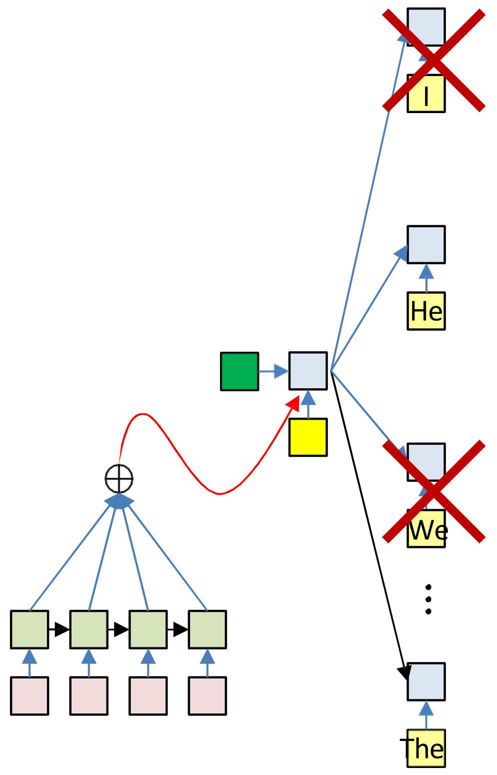
- Simply selecting the most likely symbol at each time may result in suboptimal output

Solution: Multiple choices



- Retain all choices and *fork* the network
  - With every possible word as input

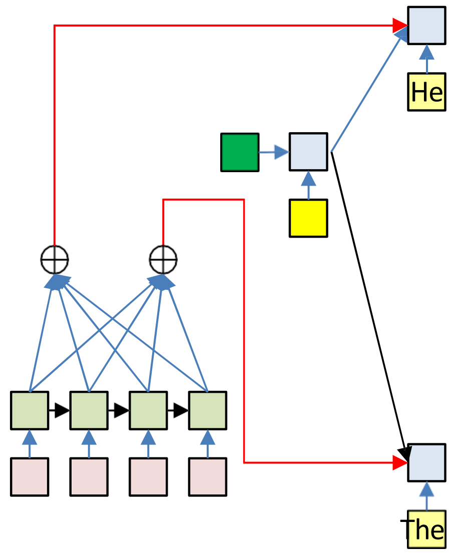
To prevent blowup: Prune



$Top_K \ P(O_1|I_1, \ldots, I_N)$

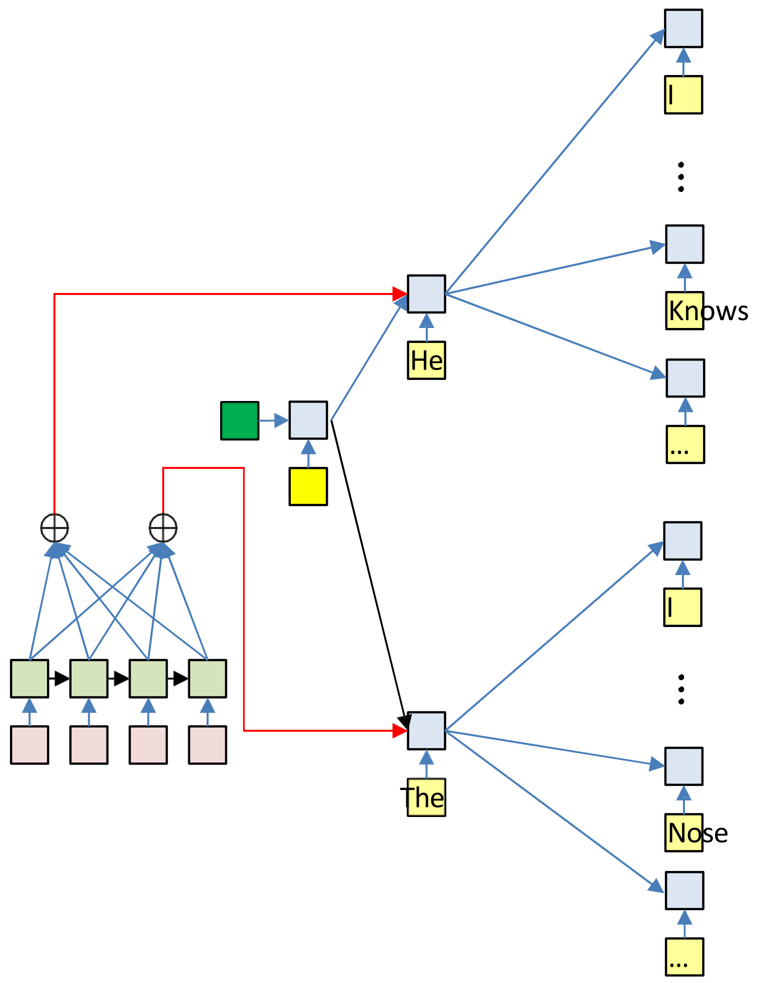
- **Prune**
  - At each time, retain only the top K scoring forks

Decoding



- At each time, retain only the top K scoring forks

Decoding



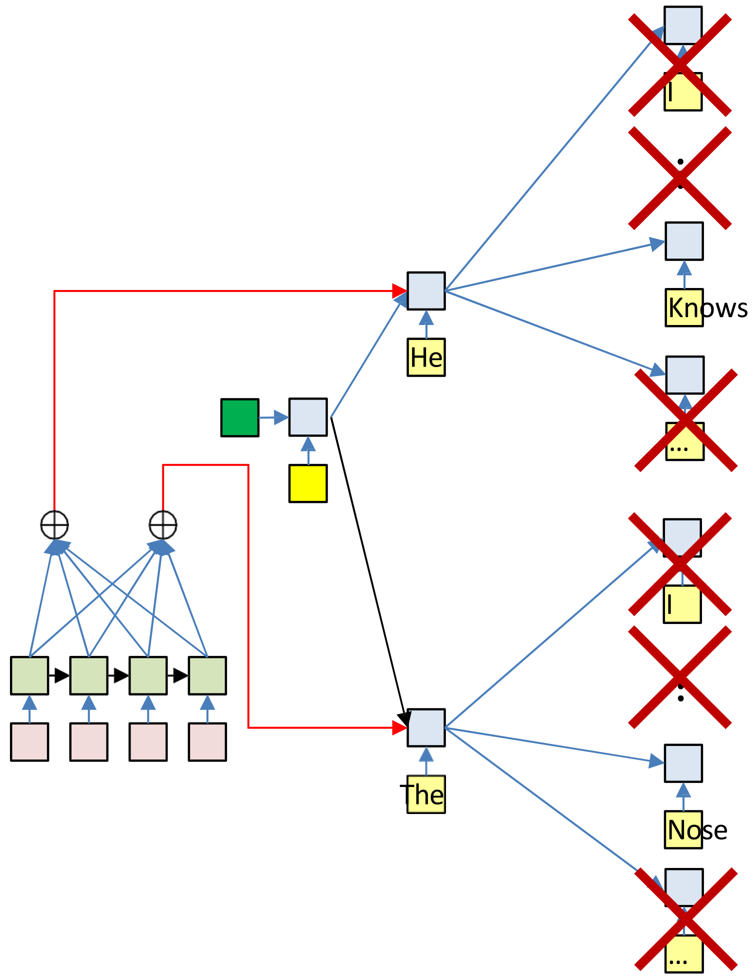
Note: based on product

$$Top_K \; P(O_2 O_1 | I_1, \dots, I_N)$$

$$= Top_K \; P(O_2 | O_1, I_1, \dots, I_N) P(O_1 | I_1, \dots, I_N)$$

- At each time, retain only the top K scoring forks

Decoding



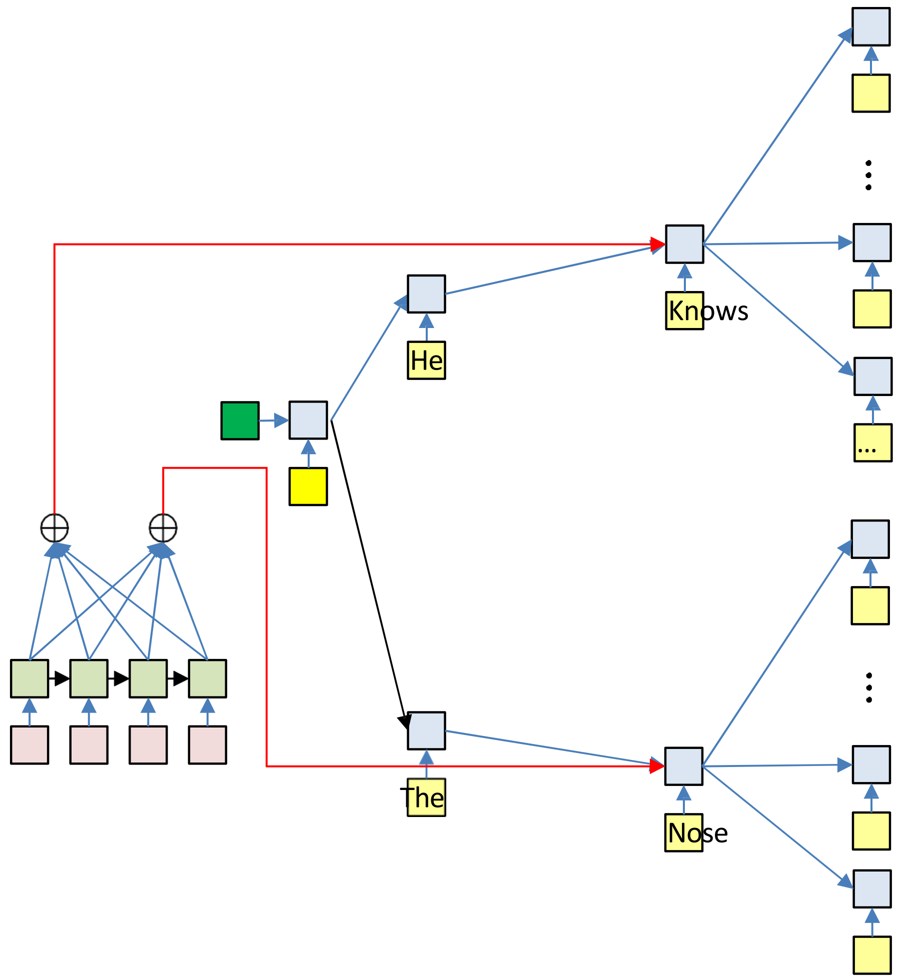
Note: based on product

$$Top_K\ P(O_2 O_1 | I_1, \dots, I_N)$$

$$= Top_K\ P(O_2 | O_1, I_1, \dots, I_N) P(O_1 | I_1, \dots, I_N)$$

- At each time, retain only the top K scoring forks

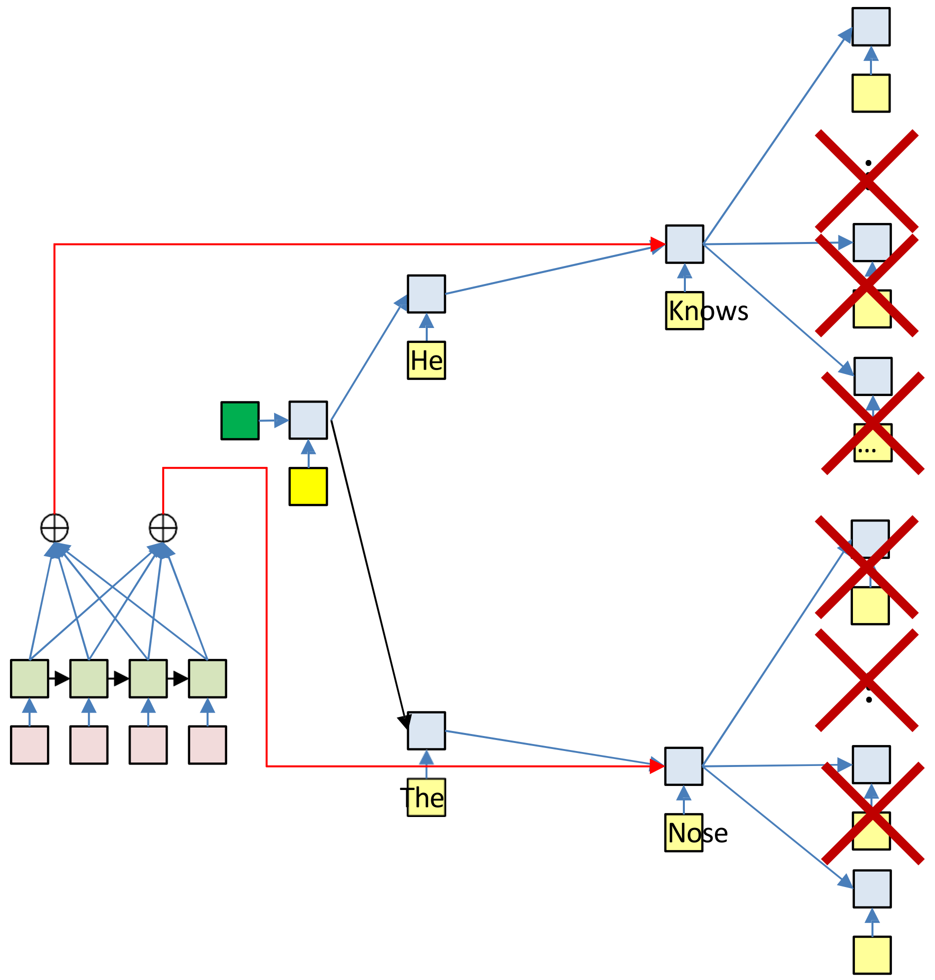
Decoding



$$= Top_K\, P(O_2|O_1, I_1, \ldots, I_N) \times$$
$$P(O_2|O_1, I_1, \ldots, I_N) \times$$
$$P(O_1|I_1, \ldots, I_N)$$

- At each time, retain only the top K scoring forks

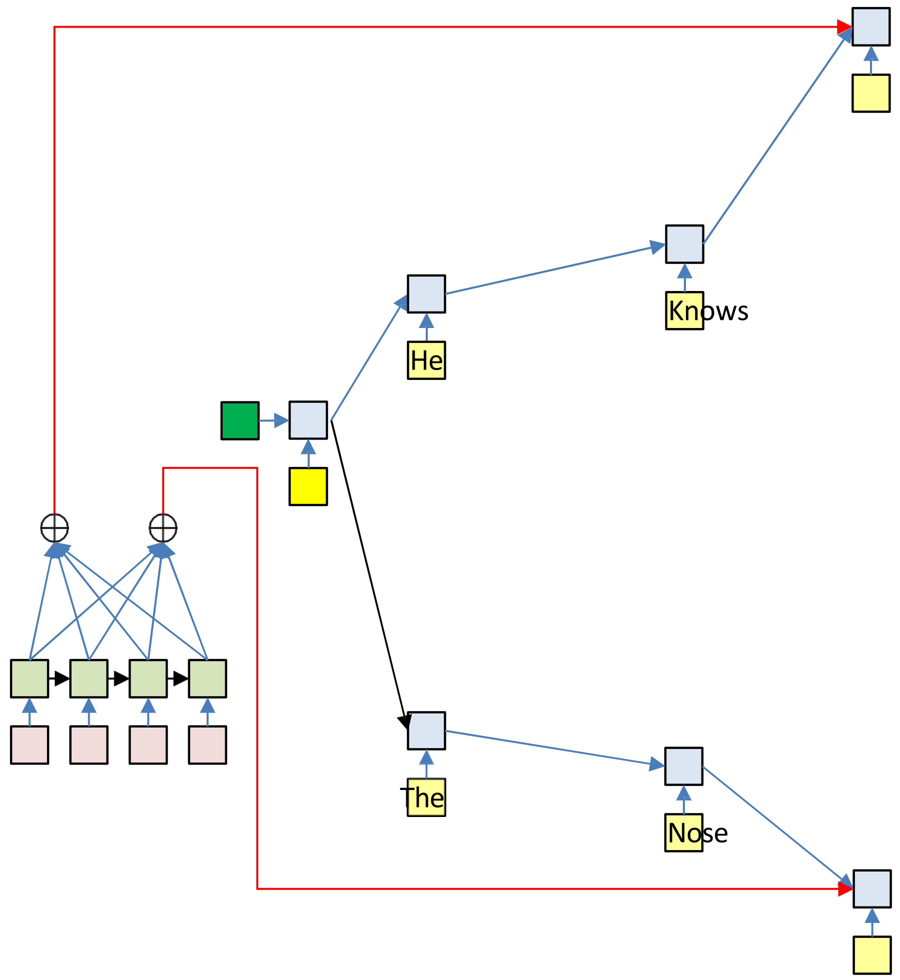
Decoding



$$= Top_K\, P(O_2|O_1, I_1, \dots, I_N) \times$$
$$P(O_2|O_1, I_1, \dots, I_N) \times$$
$$P(O_1|I_1, \dots, I_N)$$

- At each time, retain only the top K scoring forks

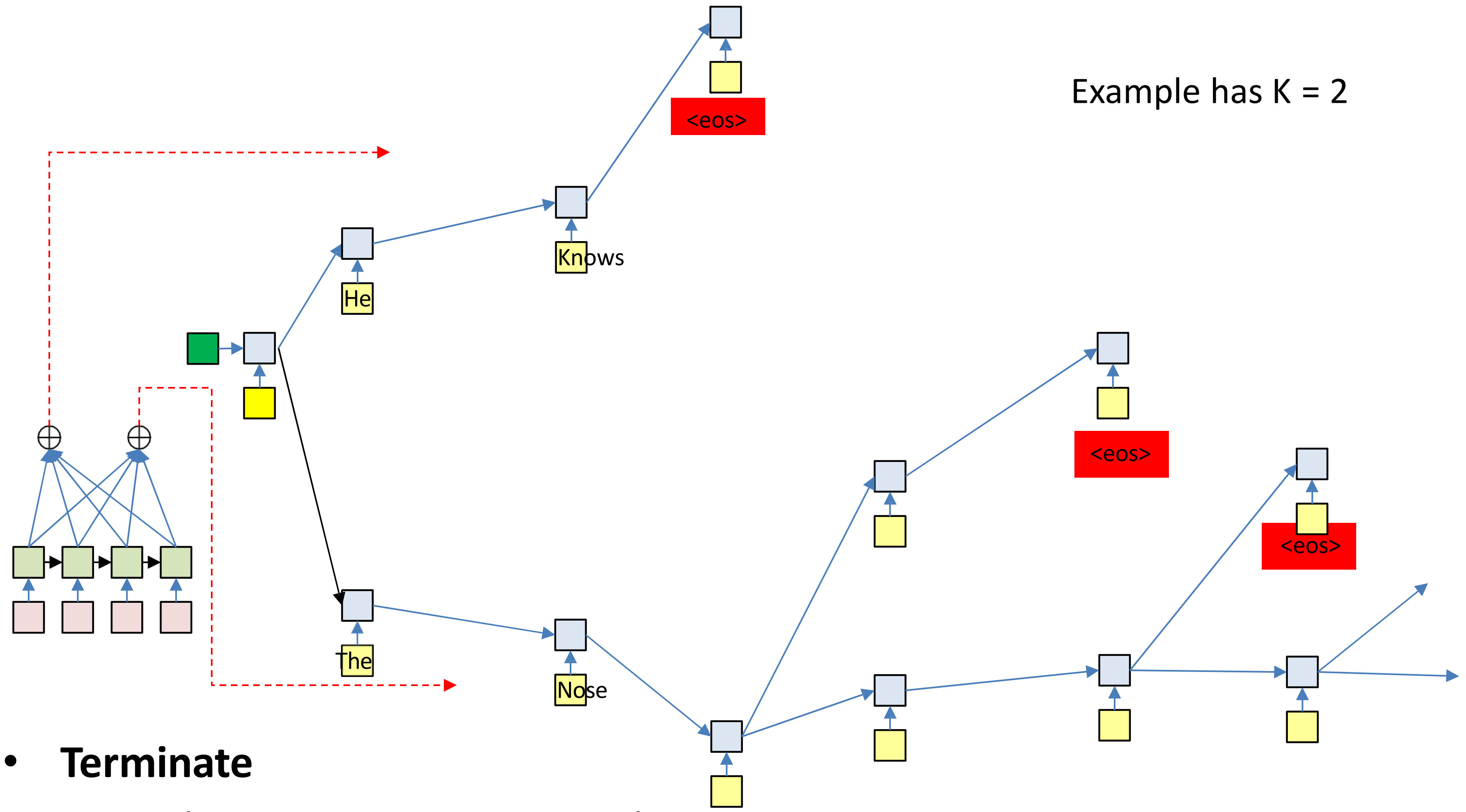
Decoding



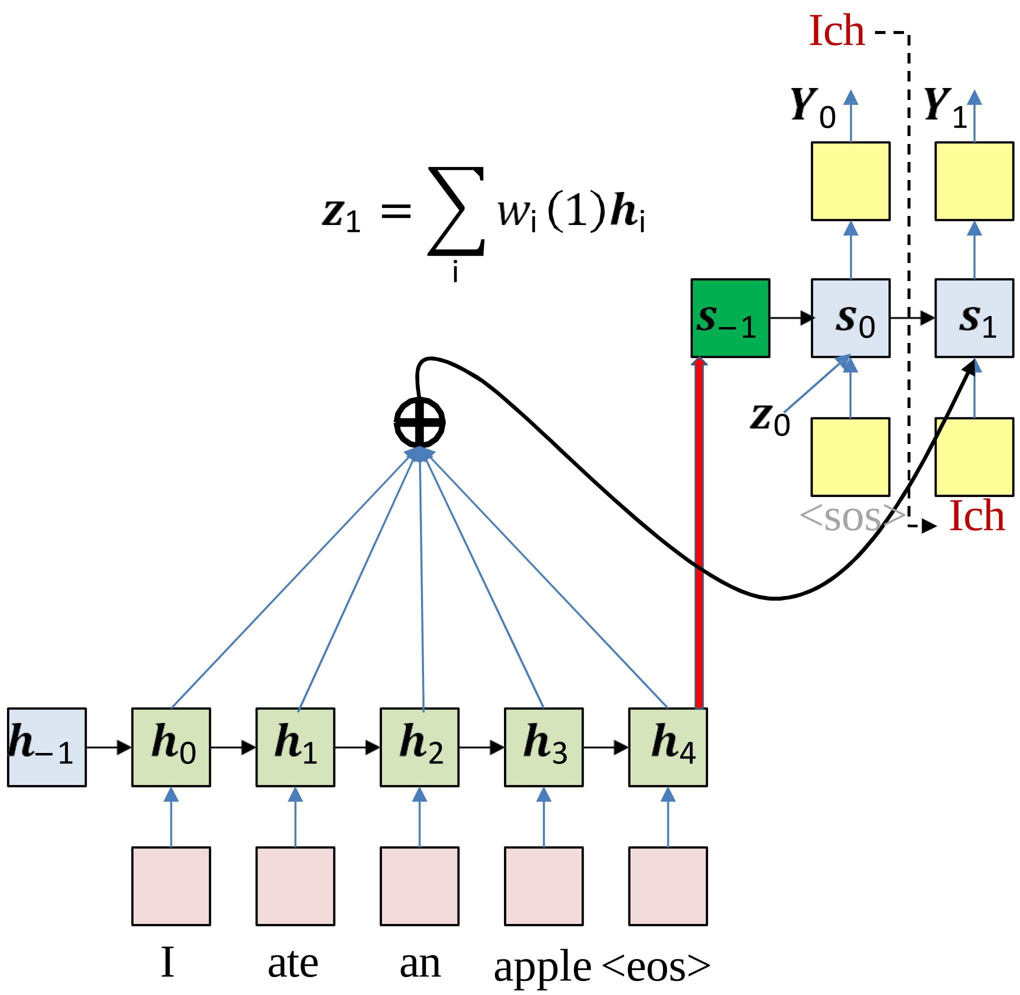
$$Top_K \prod_{t=1}^{n} P(O_n | O_1, \ldots, O_{n-1}, I_1, \ldots, I_N)$$

- At each time, retain only the top K scoring forks

Termination: <eos>



- **Terminate**
  - Paths cannot continue once the output an <eos>
    - So paths may be different lengths
      - Select the most likely sequence ending in <eos> across *all* terminating sequences



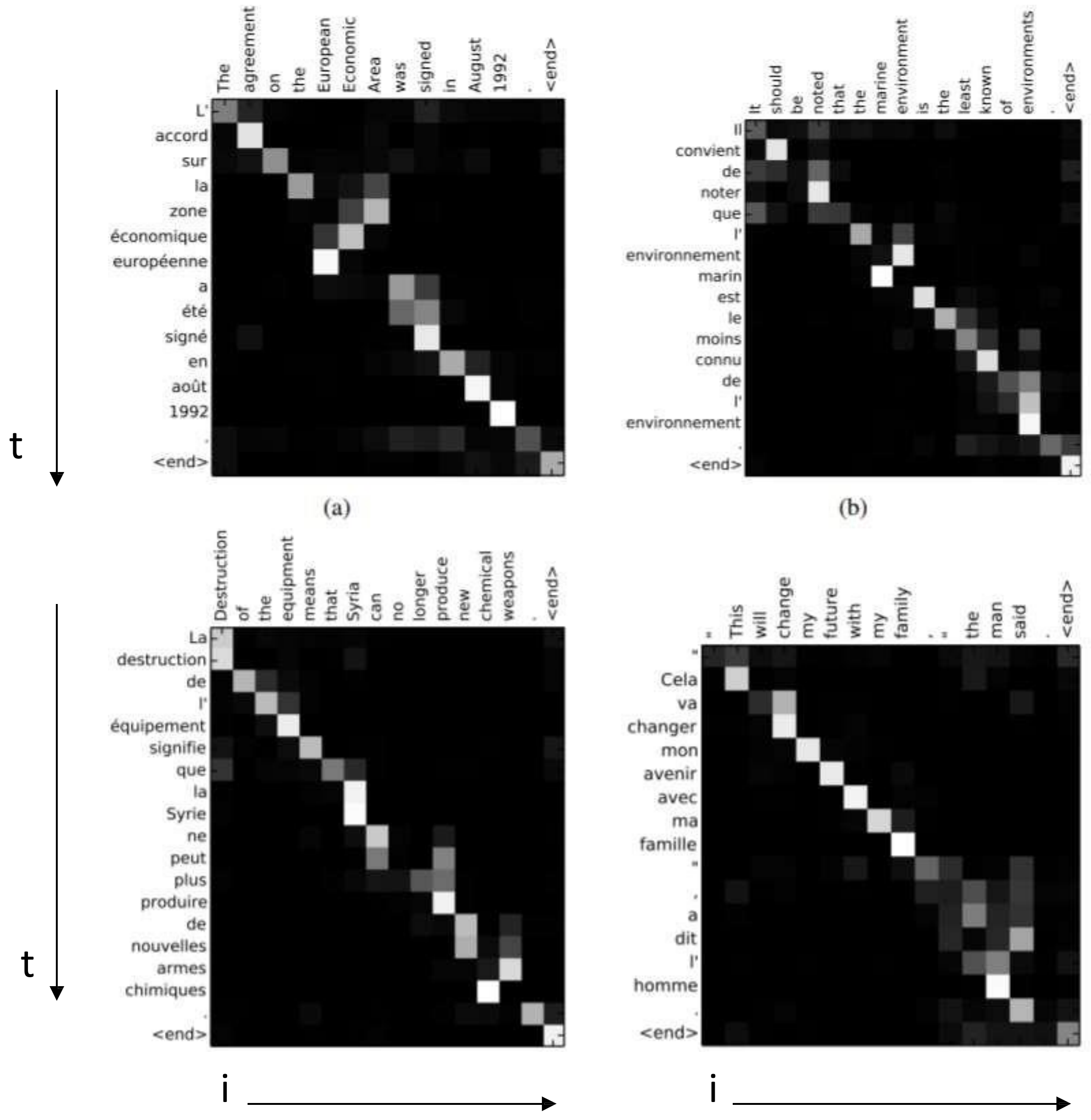
What does the attention learn?

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

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

- The key component of this model is the attention weight
  - It captures the relative importance of each position in the input to the current output

“Alignments” example: Bahdanau et al.



Plot of $w_i(t)$

Color shows value (white is larger)

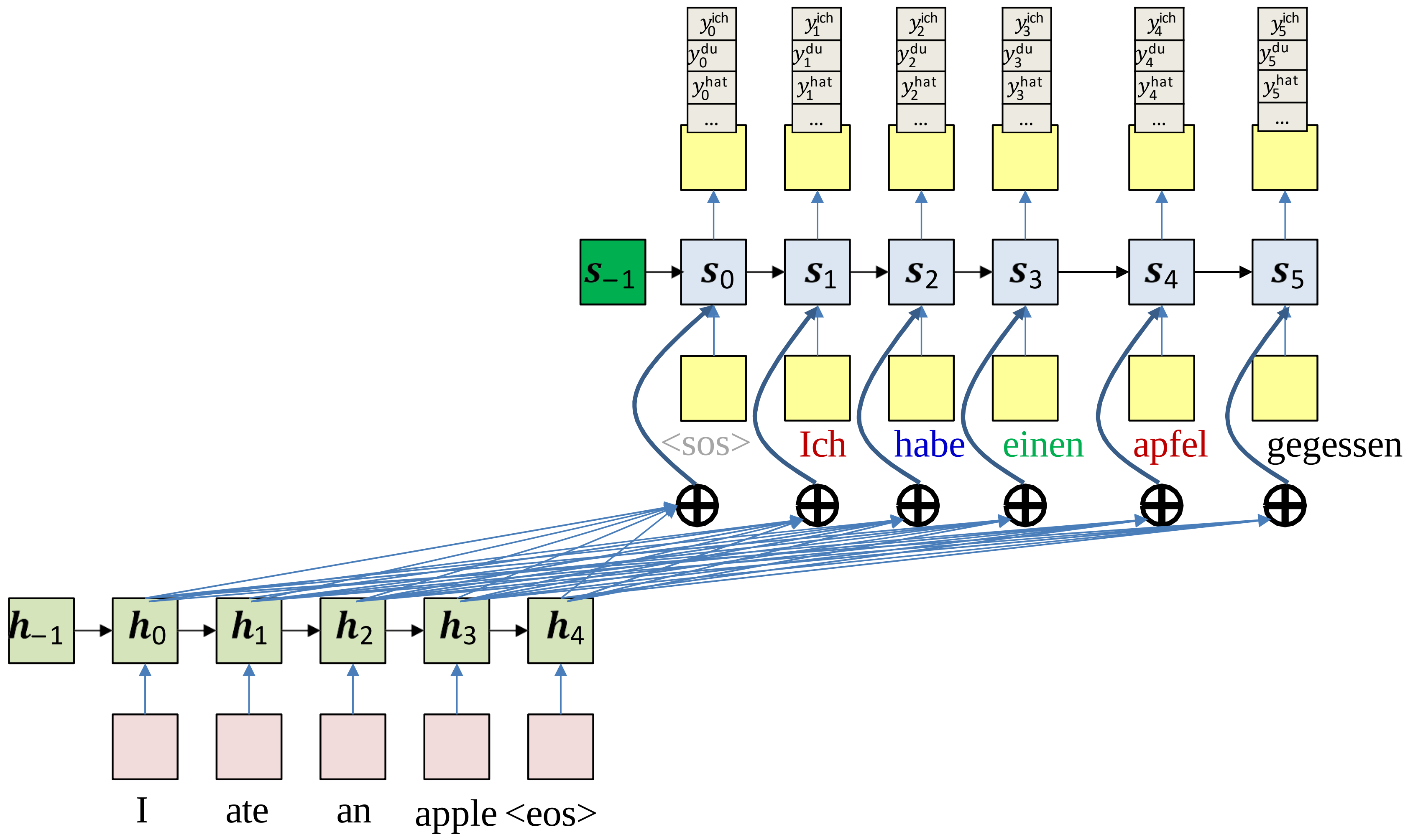
Note how most important input words for any output word get automatically highlighted

The general trend is somewhat linear because word order is roughly similar in both languages

Training the network

- We have seen how a trained network can be used to compute outputs
  - Convert one sequence to another

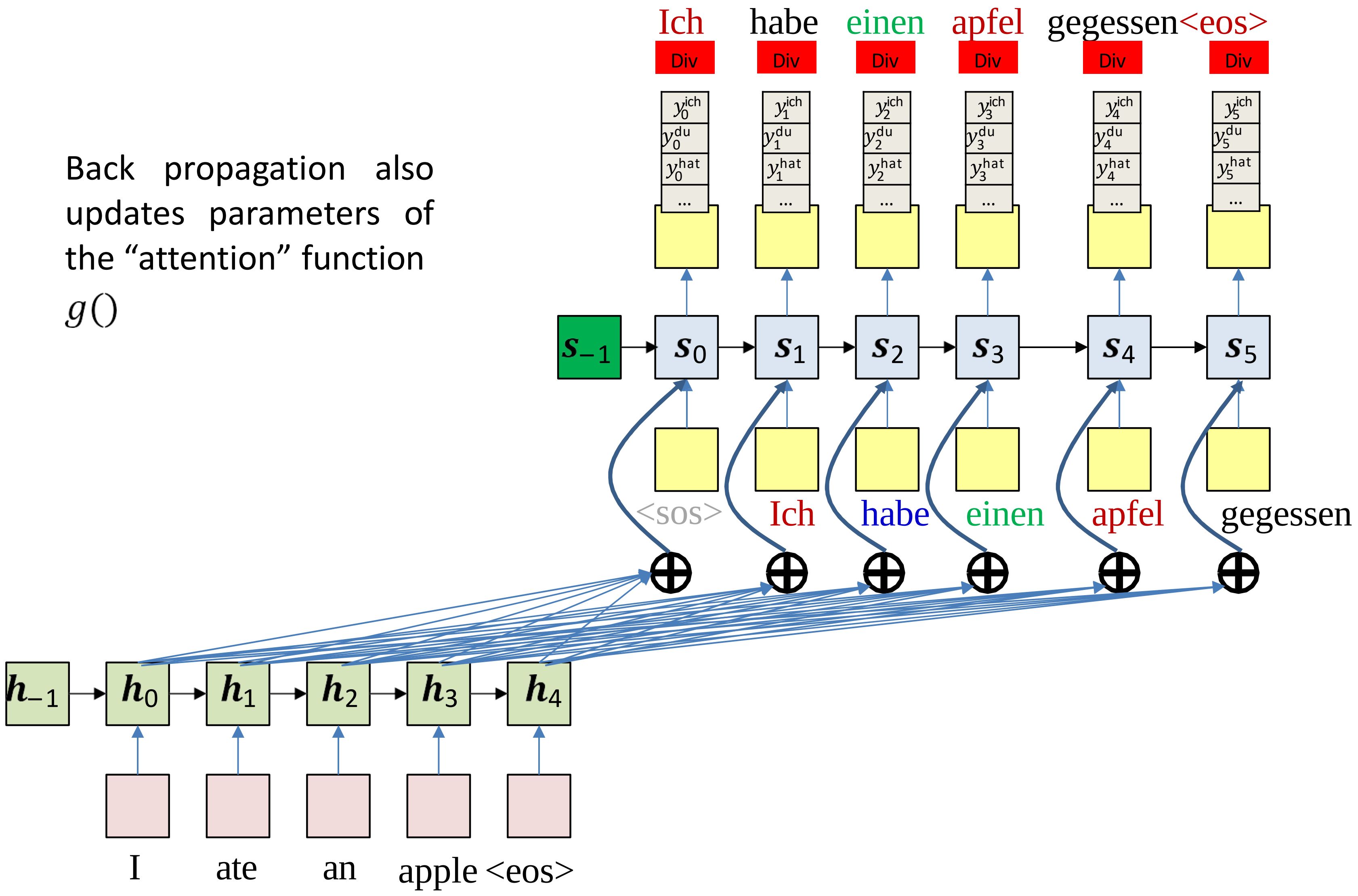
- Let’s consider training..



y_0^{ich} y_0^{du} y_0^{hat} ...
y_1^{ich} y_1^{du} y_1^{hat} ...
y_2^{ich} y_2^{du} y_2^{hat} ...
y_3^{ich} y_3^{du} y_3^{hat} ...
y_4^{ich} y_4^{du} y_4^{hat} ...
y_5^{ich} y_5^{du} y_5^{hat} ...
s_{-1}
s_0
s_1
s_2
s_3
s_4
s_5
<sos>
Ich
habe
einen
apfel
gegessen
h_{-1}
h_0
h_1
h_2
h_3
h_4
I
ate
an
apple
<eos>

- Given training input (source sequence, target sequence) pairs
- **Forward pass**: Pass the actual input sequence through the encoder
  - At each time the output is a probability distribution over words

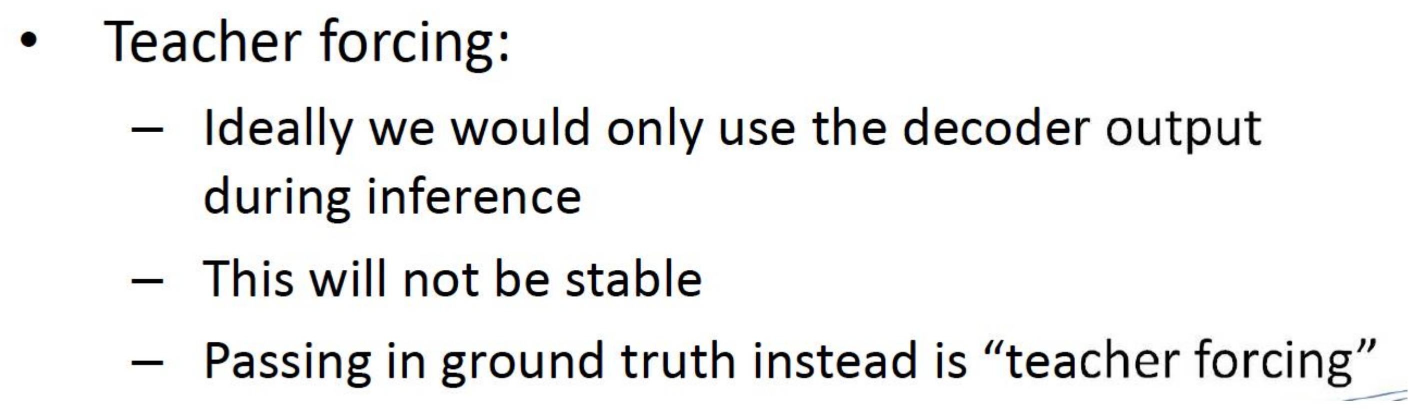
Back propagation also updates parameters of the “attention” function $g()$



Ich habe einen apfel gegessen <eos>
Div
y_0^{ich} y_0^{du} y_0^{hat} ...
y_1^{ich} y_1^{du} y_1^{hat} ...
y_2^{ich} y_2^{du} y_2^{hat} ...
y_3^{ich} y_3^{du} y_3^{hat} ...
y_4^{ich} y_4^{du} y_4^{hat} ...
y_5^{ich} y_5^{du} y_5^{hat} ...
s_{-1} s_0 s_1 s_2 s_3 s_4 s_5
<sos> Ich habe einen apfel gegessen
h_{-1} h_0 h_1 h_2 h_3 h_4
I ate an apple <eos>

- **Backward pass**: Compute a divergence between target output and output distributions
  - Backpropagate derivatives through the network

Tricks of the trade…



- Teacher forcing:
  - Ideally we would only use the decoder output during inference
  - This will not be stable
  - Passing in ground truth instead is "teacher forcing"

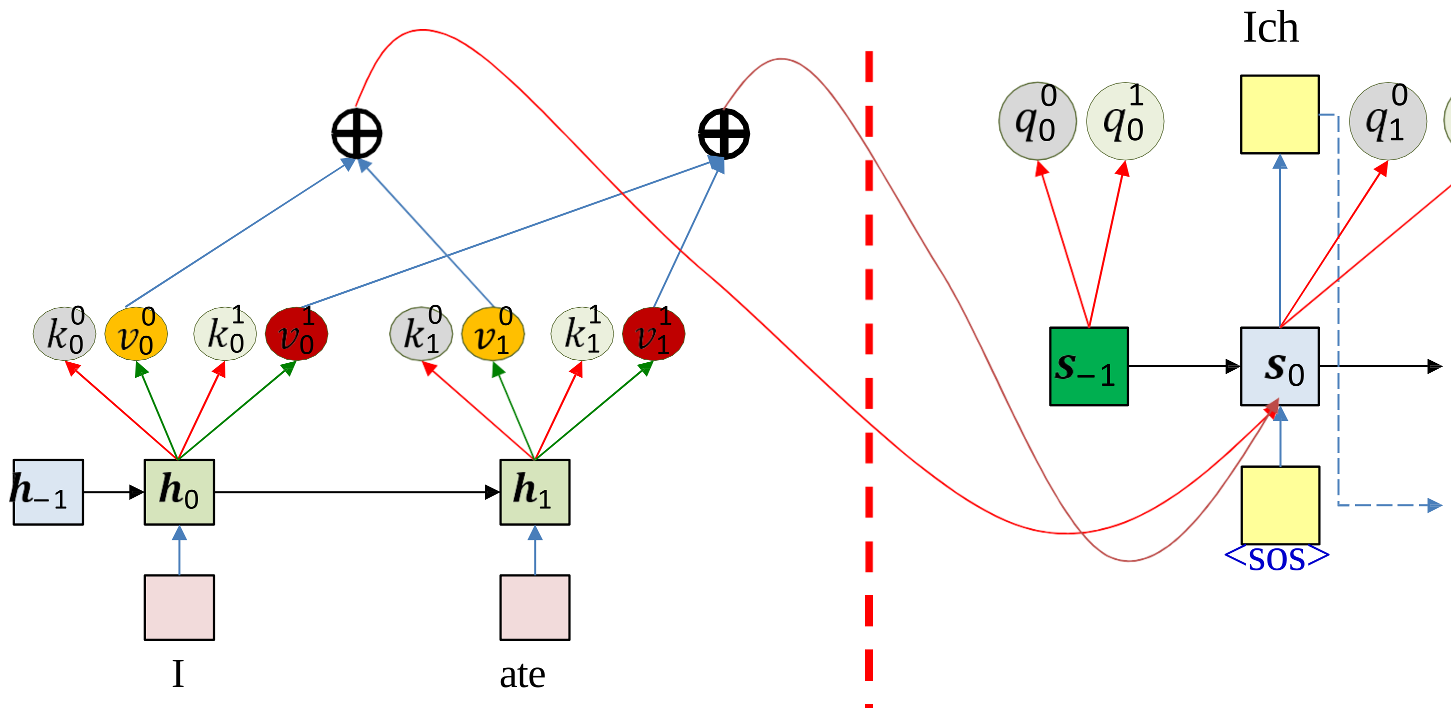
- Sampling the output:
  - Sample the system output and
  - as input during training for only some of the time

- The "Gumbel noise" trick:
  - Sampling is not differentiable, and gradients cannot be passed through it
  - The "Gumbel noise" approach recasts sampling as computing the argmax of a Gumbel distribution, with the network output as parameters
  - The "argmax" can be replaced by a "softmax", making the process differentiable w.r.t. network outputs

Various extensions

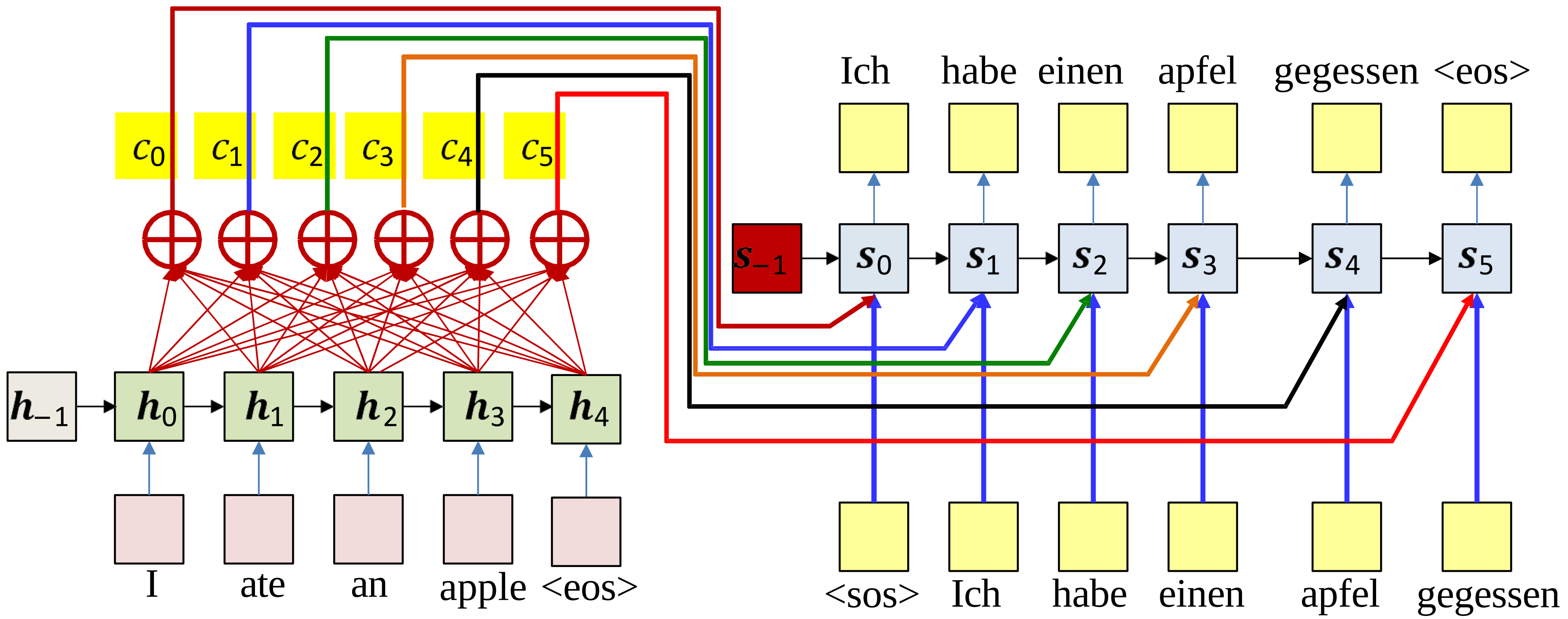
- Bidirectional processing of input sequence
  - Bidirectional networks in encoder
  - E.g. “Neural Machine Translation by Jointly Learning to Align and Translate”, Bahdanau et al. 2016
- Attention: Local attention vs global attention
  - E.g. “Effective Approaches to Attention-based Neural Machine Translation”, Luong et al., 2015
  - Other variants

Extensions: Multihead attention



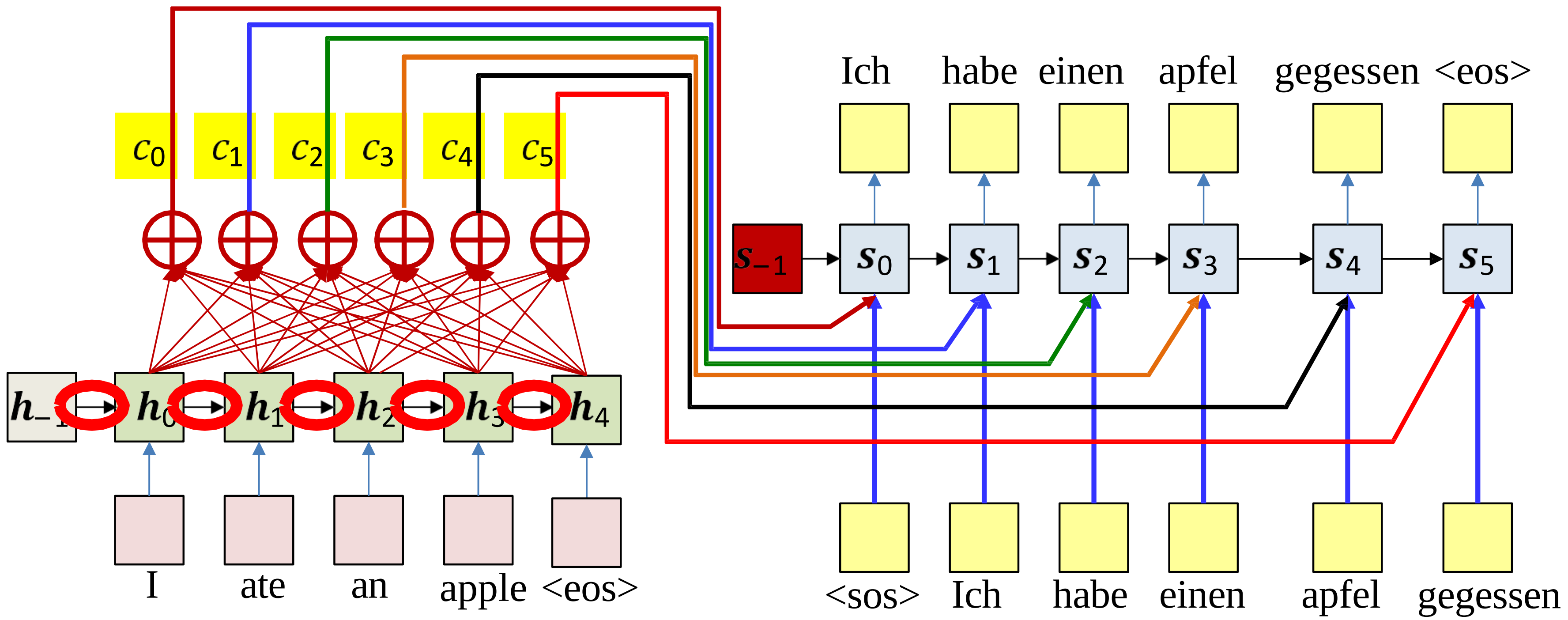
- Have multiple query/key/value sets.
  - Each attention "head" uses one of these sets
  - The combined contexts from all heads are passed to the decoder
- Each "attender" focuses on a different aspect of the input that's important for the decode

Recap: Attention Models



- Encoder recurrently produces hidden representations of input word sequence
- Decoder recurrently generates output word sequence
  - For each output word the decoder uses a weighted average of the hidden input representations as input “context”, along with the recurrent hidden state and the previous output word

Recap: Attention Models

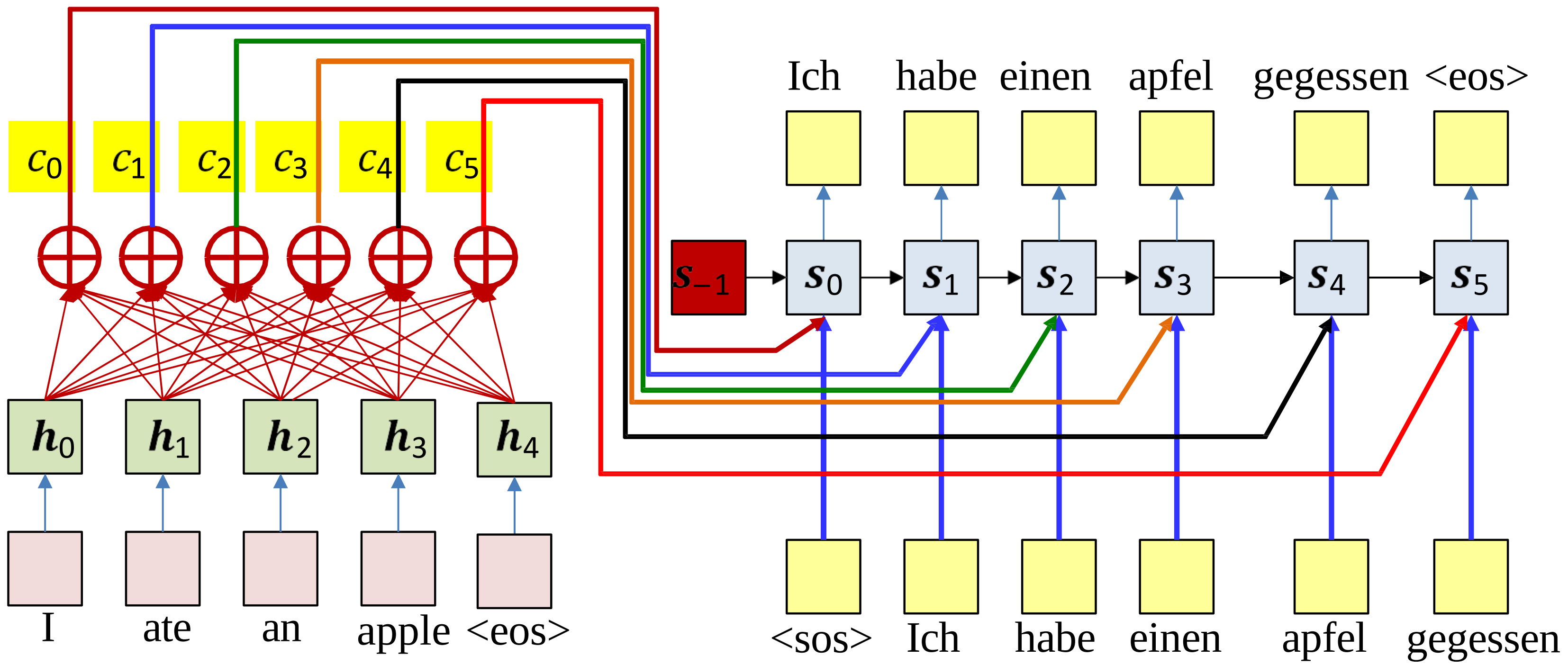


c_0 c_1 c_2 c_3 c_4 c_5
h_{-1} h_0 h_1 h_2 h_3 h_4
I ate an apple <eos>
s_{-1} s_0 s_1 s_2 s_3 s_4 s_5
Ich habe einen apfel gegessen <eos>
<sos> Ich habe einen apfel gegessen

- Problem: Because of the recurrence, the hidden representation for any word is also influenced by *all* preceding words
  - **The decoder is actually paying attention to the sequence, and not just the word**

- If the decoder is automatically figuring out which words of the input to attend to at each time, is recurrence in the input even necessary?

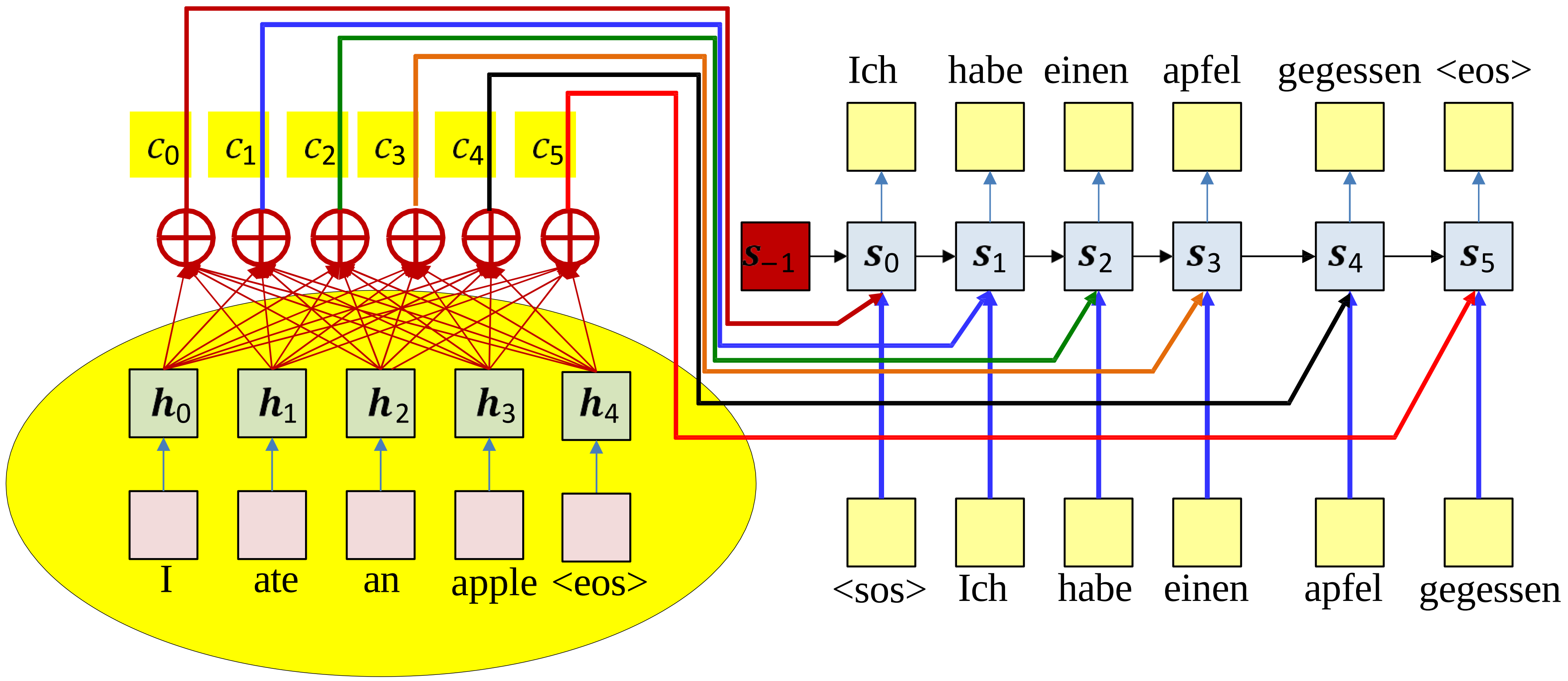
Non-recurrent encoder



c_0 c_1 c_2 c_3 c_4 c_5
h_0 h_1 h_2 h_3 h_4
I ate an apple <eos>
s_{-1} s_0 s_1 s_2 s_3 s_4 s_5
Ich habe einen apfel gegessen <eos>
<sos> Ich habe einen apfel gegessen

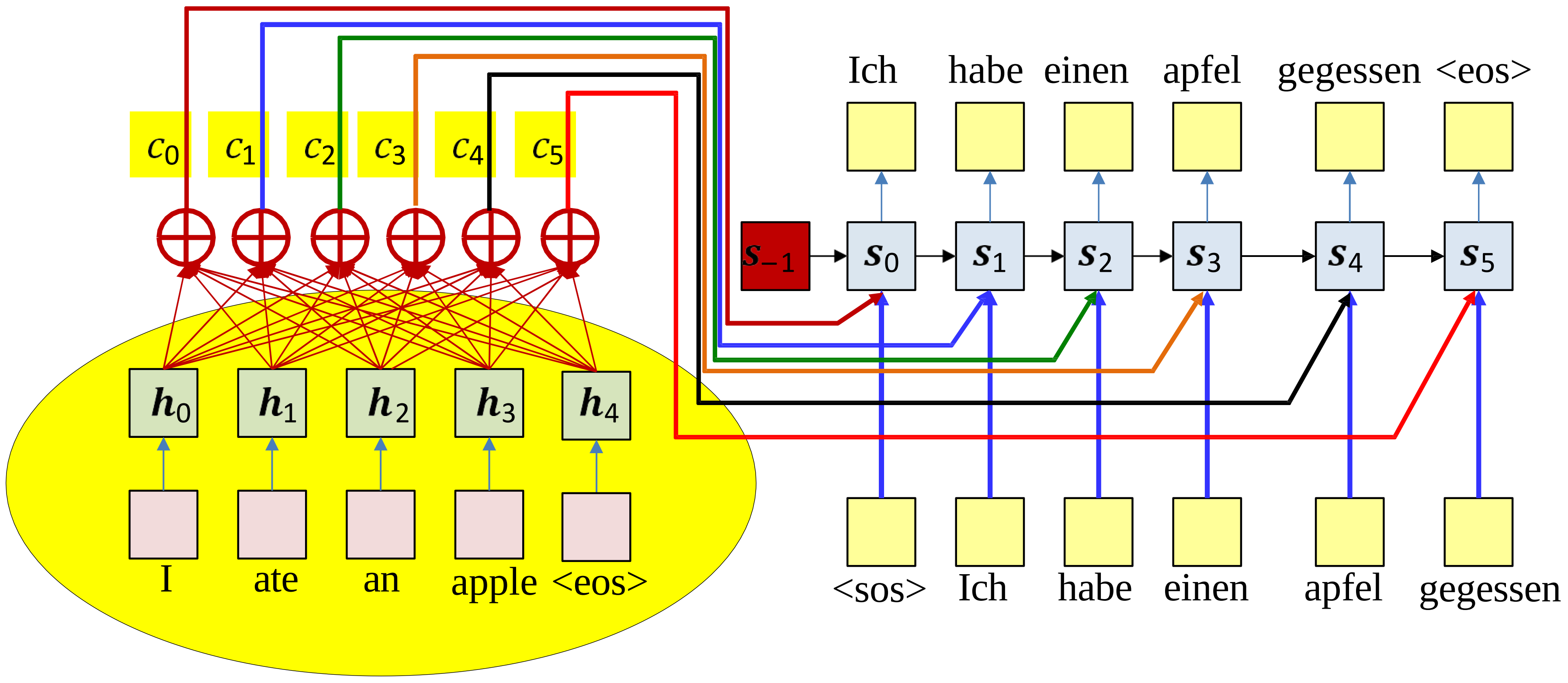
- Modification: Let us eliminate the recurrence in the encoder

Non-recurrent encoder



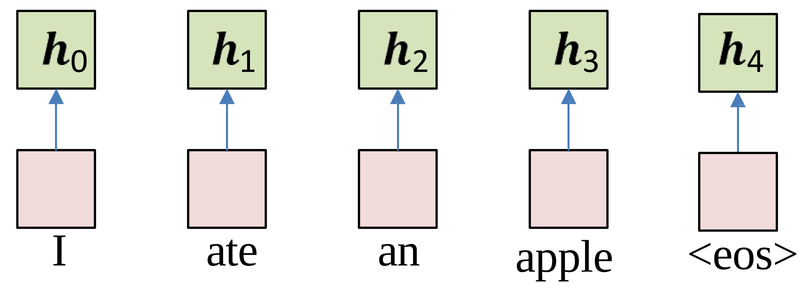
- But this will eliminate *context-specificity* in the encoder embeddings
  - The embedding for "an" must really depend on the remaining words
    - It could be translated to "ein", "einer", or "eines" depending on the context.
- Solution: Use the attention framework itself to introduce context- specificity in embeddings

Non-recurrent encoder



- The encoder in a sequence-to-sequence model can be composed without recurrence.
- Use the attention framework itself to introduce context-specificity in embeddings
  - "Self" attention

Self attention



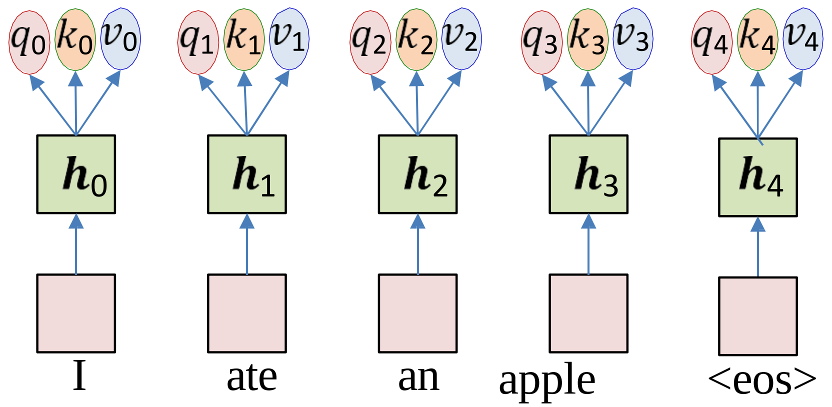
- First, for every word in the input sequence we compute an initial representation
  - E.g. using a single MLP layer

Self attention

$q_i = \boldsymbol{W}_q h_i$

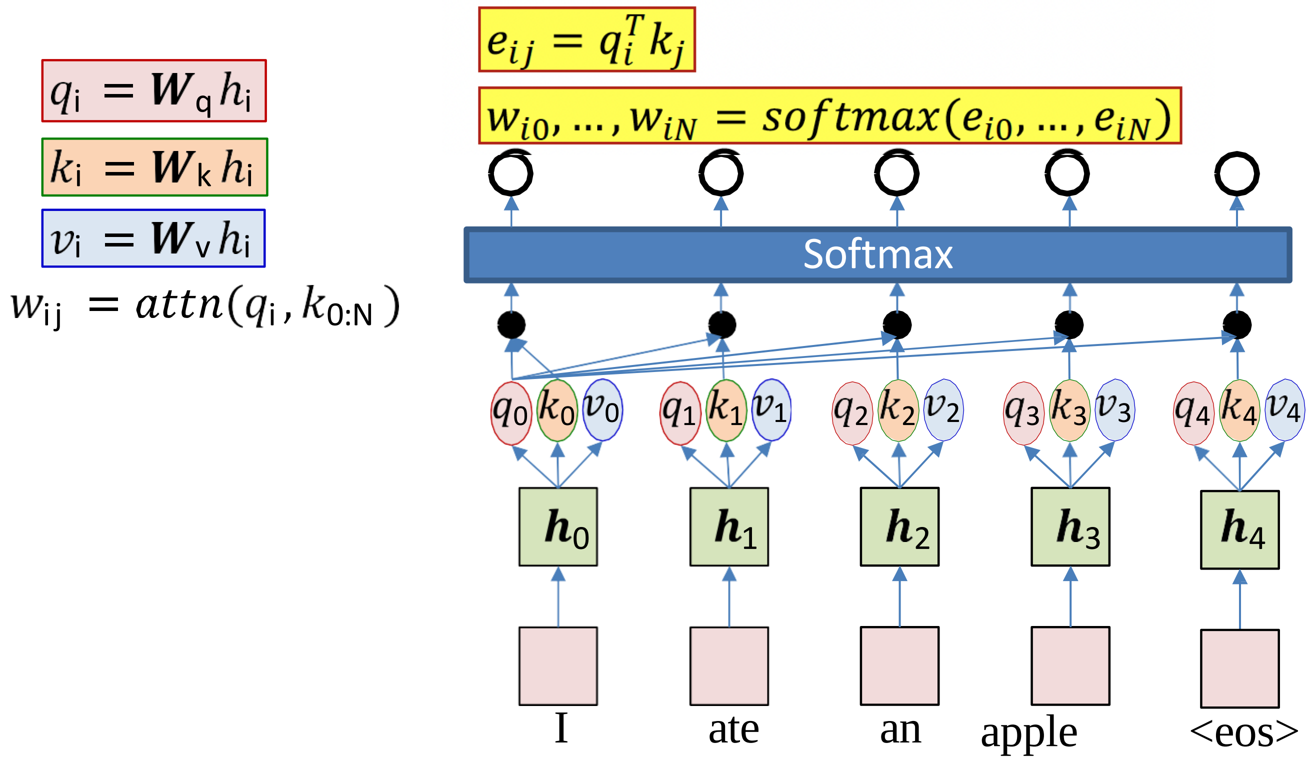
$k_i = \boldsymbol{W}_k h_i$

$v_i = \boldsymbol{W}_v h_i$



- Then, from each of the hidden representations, we compute a query, a key, and a value.
  - Using separate linear transforms
  - The weight matrices W_q, W_k and W_v are learnable parameters

Self Attention



$e_{ij} = q_i^T k_j$
$w_{i0}, \dots, w_{iN} = softmax(e_{i0}, \dots, e_{iN})$
$q_i = W_q h_i$
$k_i = W_k h_i$
$v_i = W_v h_i$
$w_{ij} = attn(q_i, k_{0:N})$
Softmax
q_0 k_0 v_0 q_1 k_1 v_1 q_2 k_2 v_2 q_3 k_3 v_3 q_4 k_4 v_4
h_0 h_1 h_2 h_3 h_4
I
ate
an
apple
<eos>

- For each word, we compute an attention weight between that word and all other words
  - The raw attention of the ith word to the jth word is a function of query q_i and key k_j
  - The raw attention values are put through a softmax to get the final attention weights

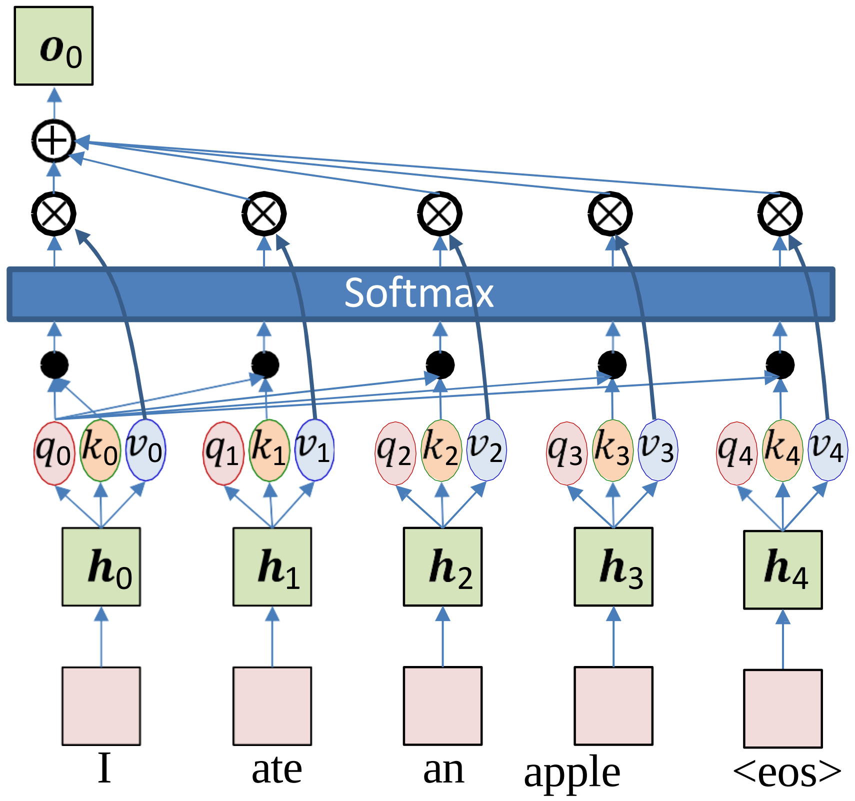
$$\boxed{q_i = \boldsymbol{W}_q h_i}$$

$$\boxed{k_i = \boldsymbol{W}_k h_i}$$

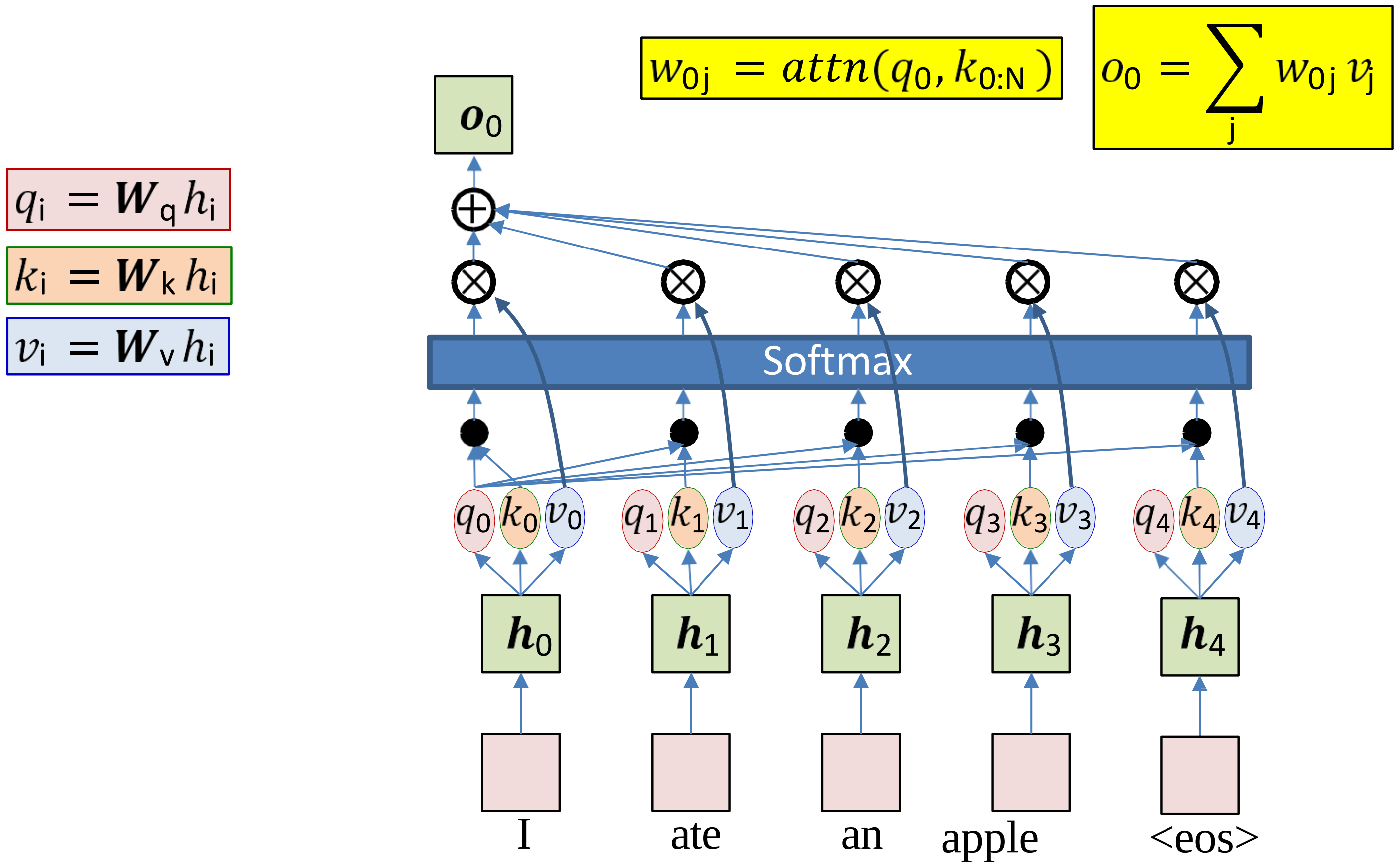
$$\boxed{v_i = \boldsymbol{W}_v h_i}$$

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

$$\boxed{o_i = \sum_j w_{ij} v_j}$$



- The updated representation for the word is the attention-weighted sum of the values for all words
  - Including itself



- Compute query-key-value sets for every word
- For each word
  - Using the query for that word, compute attention weights for all words using their keys
  - Compute updated representation for the word as attention-weighted sum of values of all words

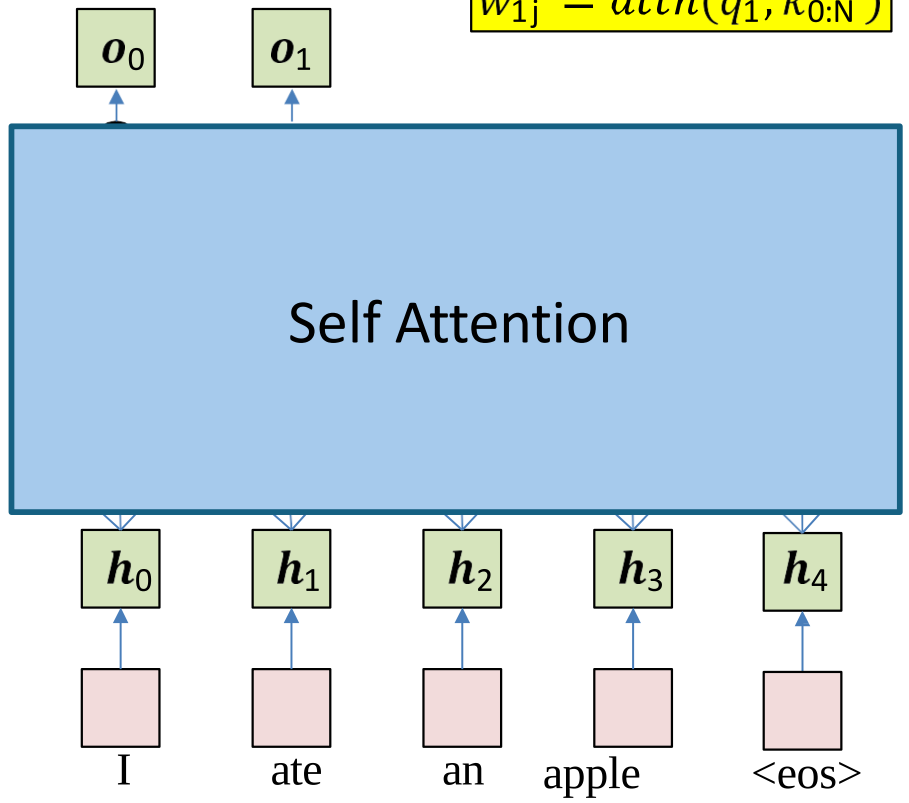
$q_i = \boldsymbol{W}_q h_i$

$k_i = \boldsymbol{W}_k h_i$

$v_i = \boldsymbol{W}_v h_i$

$w_{1j} = attn(q_1, k_{0:N})$

$o_1 = \sum_j w_{1j} v_j$



- Compute query-key-value sets for every word
- For each word
  - Using the query for that word, compute attention weights for all words using their keys
  - Compute updated representation for the word as attention-weighted sum of values of all words

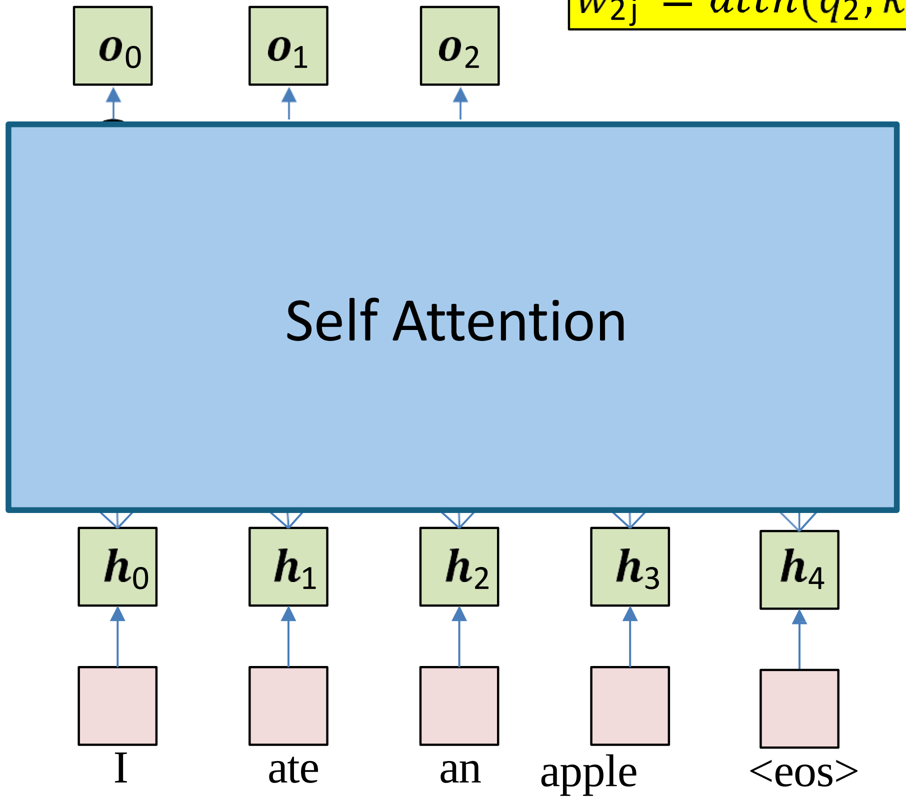
$w_{2j} = attn(q_2, k_{0:N})$

$o_2 = \sum_j w_{2j} v_j$

$q_i = \boldsymbol{W}_q h_i$

$k_i = \boldsymbol{W}_k h_i$

$v_i = \boldsymbol{W}_v h_i$

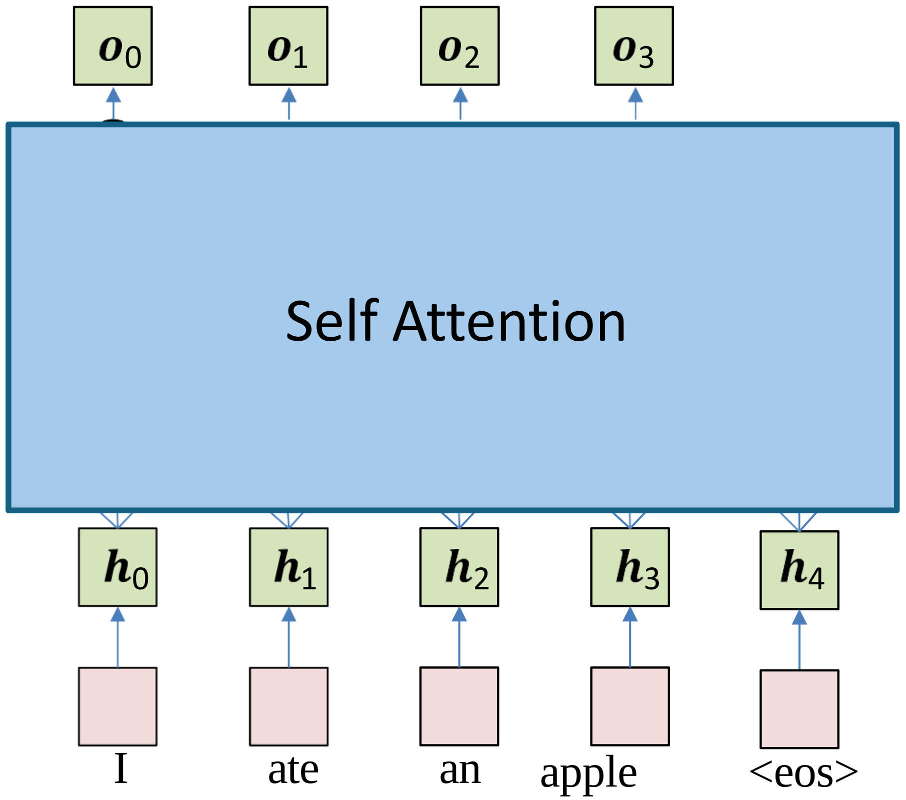


- Compute query-key-value sets for every word
- For each word
  - Using the query for that word, compute attention weights for all words using their keys
  - Compute updated representation for the word as attention-weighted sum of values of all words

$q_i = \boldsymbol{W}_q h_i$

$k_i = \boldsymbol{W}_k h_i$

$v_i = \boldsymbol{W}_v h_i$

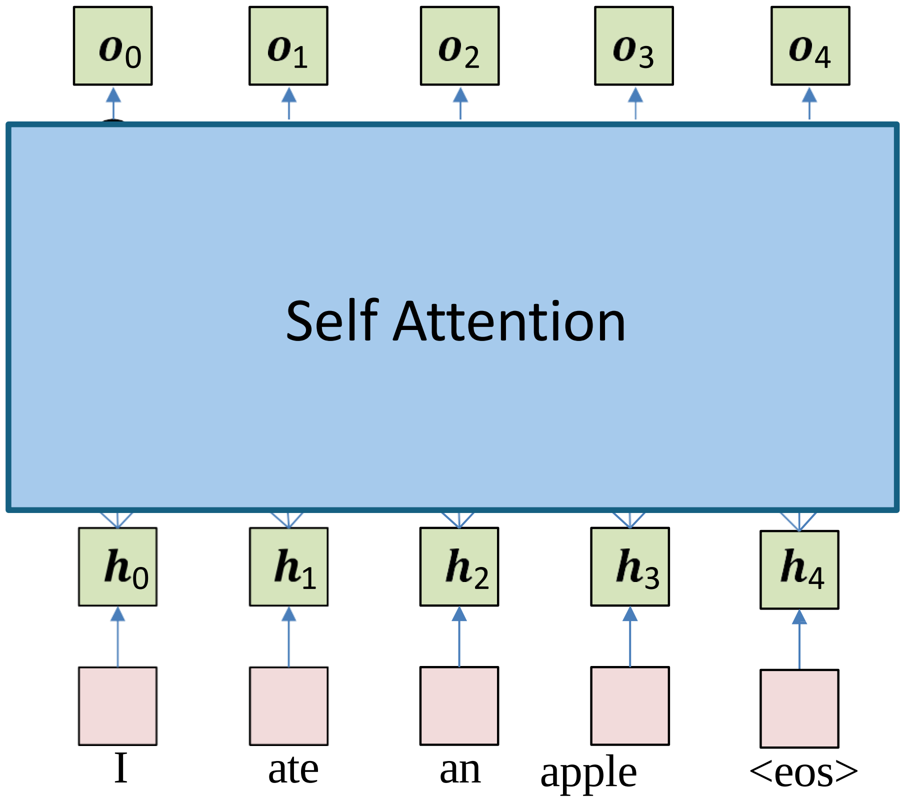


- Compute query-key-value sets for every word
- For each word
  - Using the query for that word, compute attention weights for all words using their keys
  - Compute updated representation for the word as attention-weighted sum of values of all words

$q_i = \boldsymbol{W}_q h_i$

$k_i = \boldsymbol{W}_k h_i$

$v_i = \boldsymbol{W}_v h_i$



- Compute query-key-value sets for every word
- For each word
  - Using the query for that word, compute attention weights for all words using their keys
  - Compute updated representation for the word as attention-weighted sum of values of all words

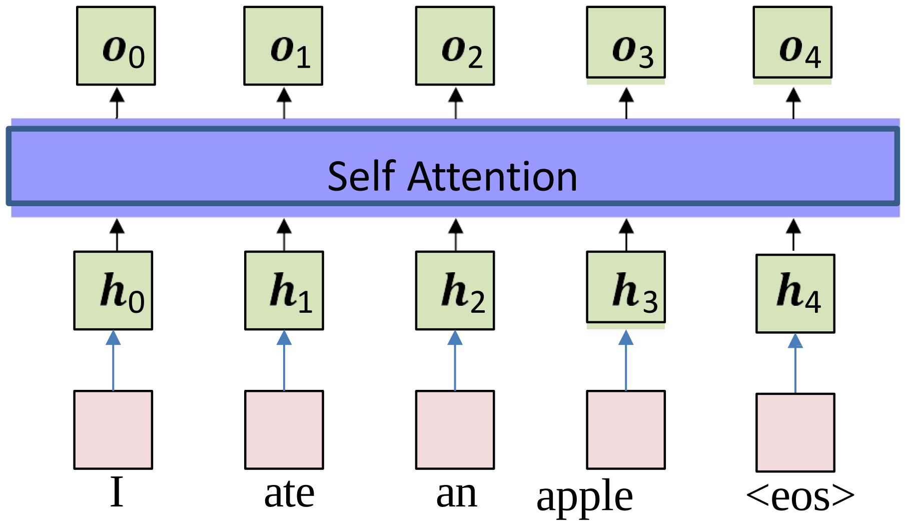
$q_i = \boldsymbol{W}_q h_i$

$k_i = \boldsymbol{W}_k h_i$

$v_i = \boldsymbol{W}_v h_i$

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

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



This is a “single-head” self-attention block

$q_i^a = \boldsymbol{W}_q^a h_i$

$k_i^a = \boldsymbol{W}_k^a h_i$

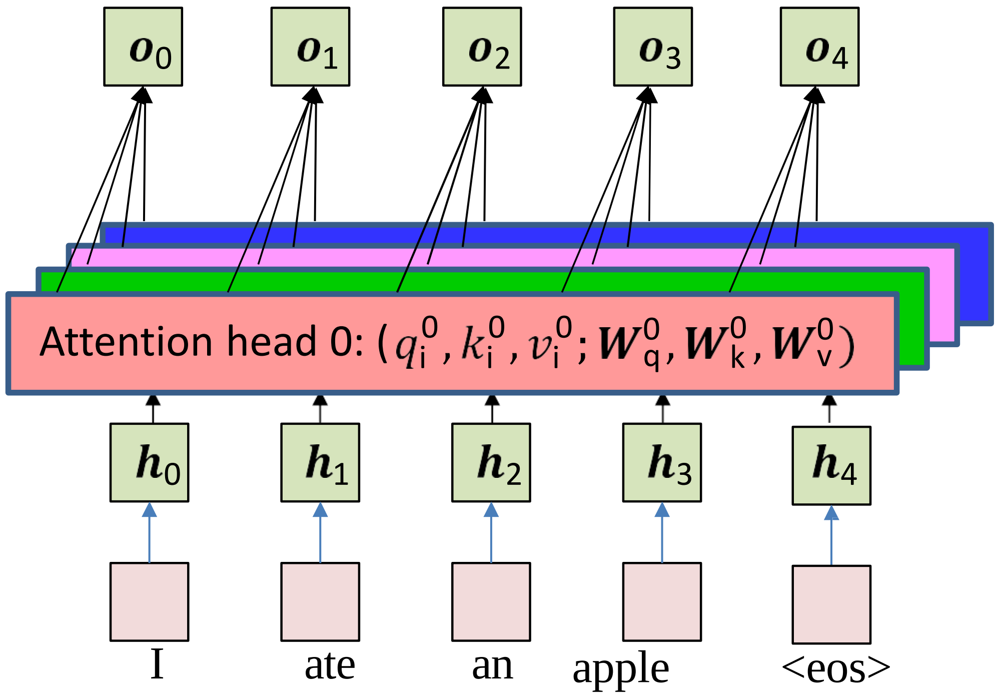
$v_i^a = \boldsymbol{W}_v^a h_i$

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

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

Concatenate

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



- We can have *multiple* such attention "heads"
  - Each will have an independent set of queries, keys and values
  - Each will obtain an independent set of attention weights
    - Potentially focusing on a different aspect of the input than other heads
  - Each computes an independent output
- The final output is the concatenation of the outputs of these attention heads
- "MULTI-HEAD ATTENTION" (actually Multi-head *self* attention)

$$\boxed{q_i^a = \mathbf{W}_q^a h_i}$$

$$\boxed{k_i^a = \mathbf{W}_k^a h_i}$$

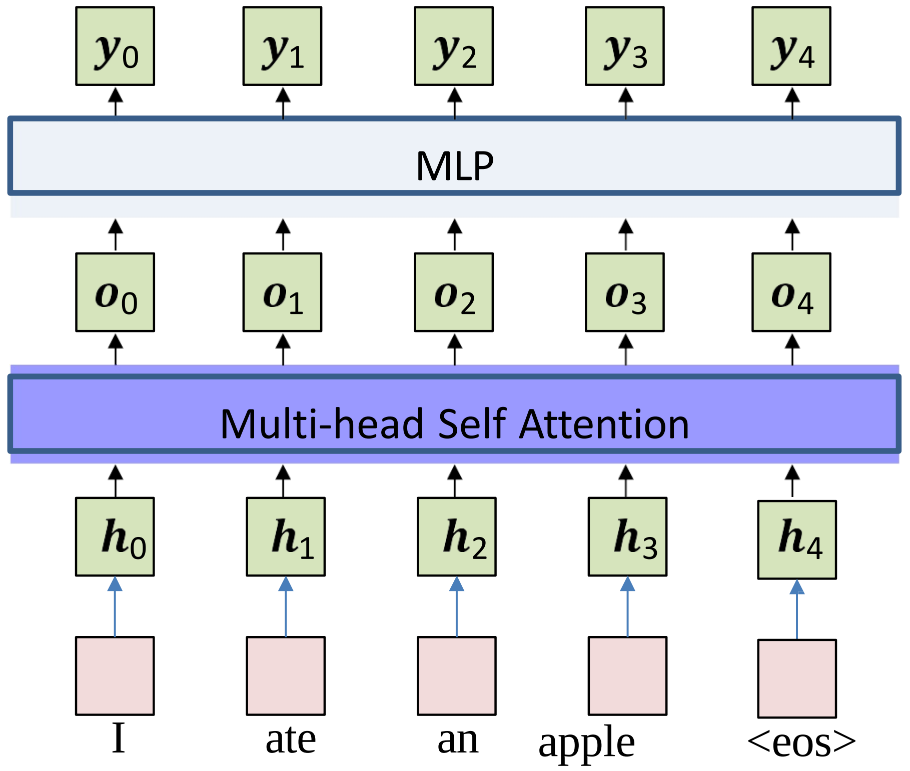
$$\boxed{v_i^a = \mathbf{W}_v^a h_i}$$

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

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

$$\boxed{o_i = [o_i^1; o_i^2; o_i^3; \dots; o_i^H]}$$

$$\boxed{y_i = MLP(o_i)}$$



y_0 y_1 y_2 y_3 y_4
MLP
o_0 o_1 o_2 o_3 o_4
Multi-head Self Attention
h_0 h_1 h_2 h_3 h_4
I
ate
an
apple
<eos>

- Typically, the output of the multi-head self attention is passed through one or more regular feedforward layers
  - Affine layer followed by a non-linear activation such as ReLU

$$q_i^a = \mathbf{W}_q^a h_i$$

$$k_i^a = \mathbf{W}_k^a h_i$$

$$v_i^a = \mathbf{W}_v^a h_i$$

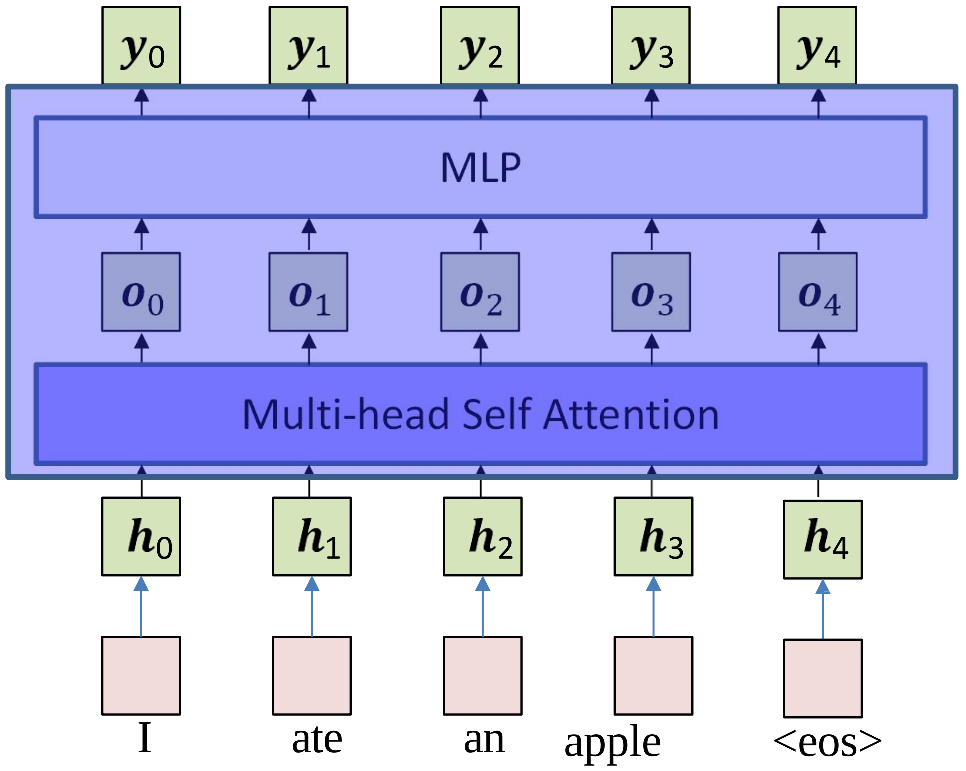
$$w_{ij}^a = attn(q_i^a, k_{0:N}^a)$$

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

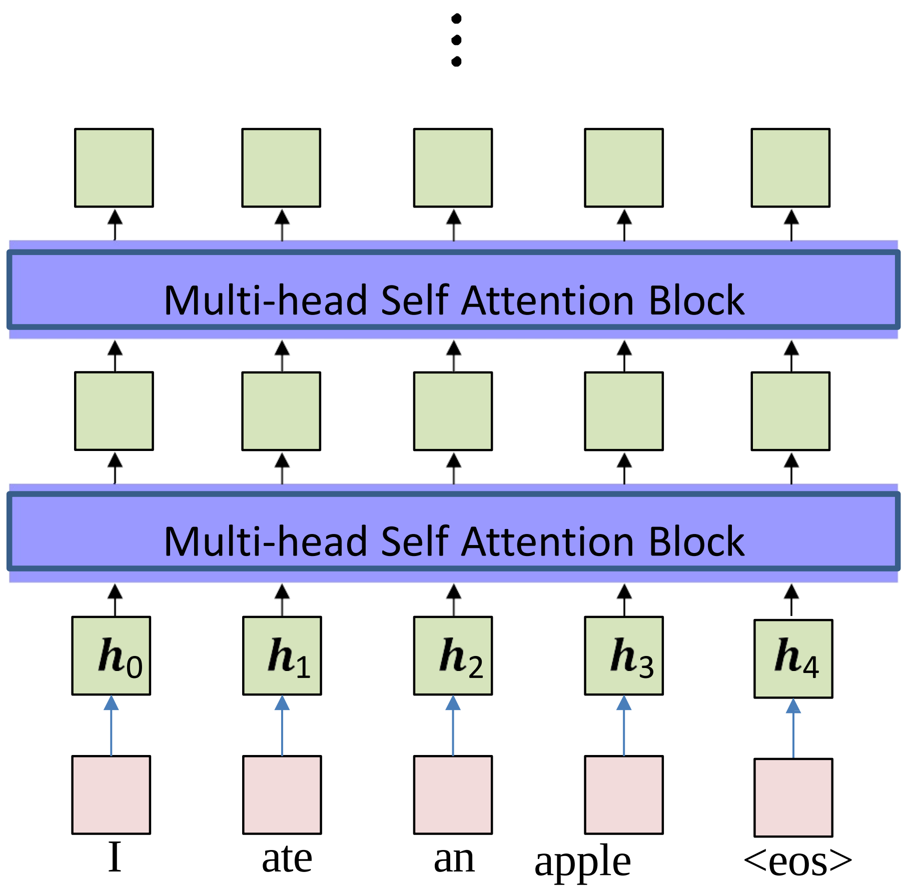
$$o_i = [o_i^1; o_i^2; o_i^3; \dots; o_i^H]$$

$$y_i = MLP(o_i)$$

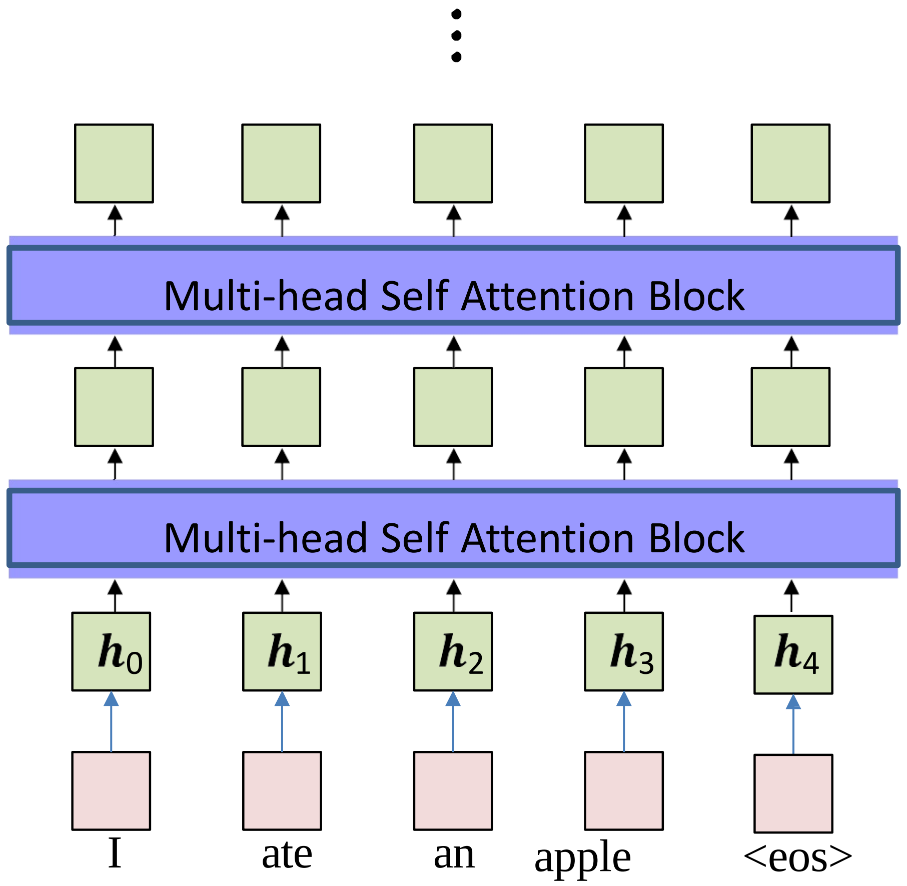
MULTI-HEAD SELF ATTENTION BLOCK



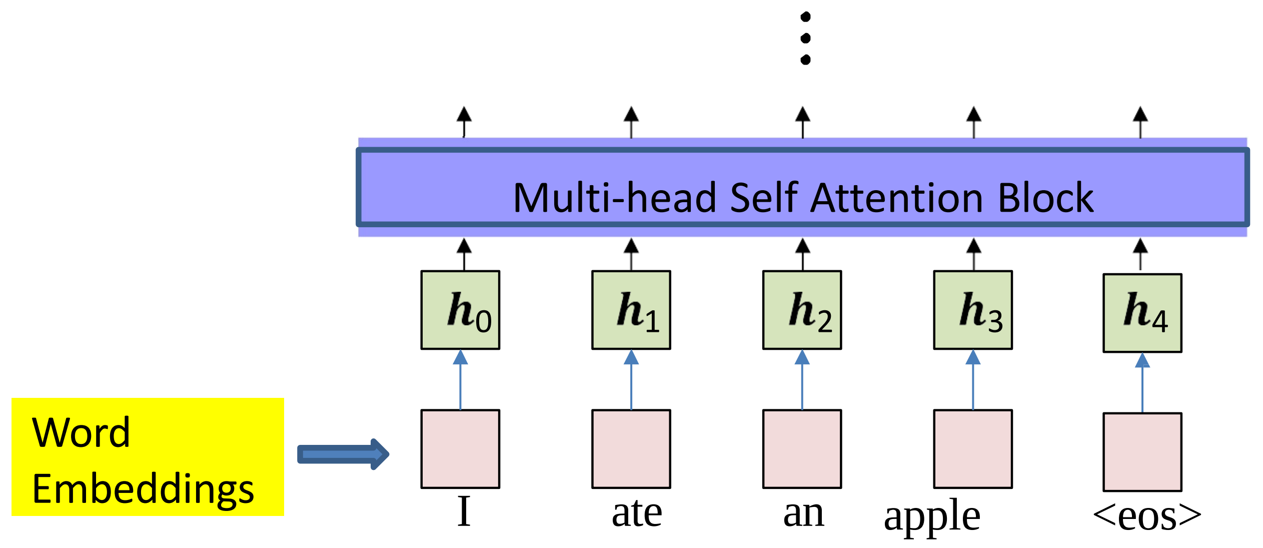
The entire unit, including multi-head self-attention module followed by MLP is a ***multi- head self-attention block***


Multi-head Self Attention Block
Multi-head Self Attention Block
h_0
h_1
h_2
h_3
h_4
I
ate
an
apple
<eos>

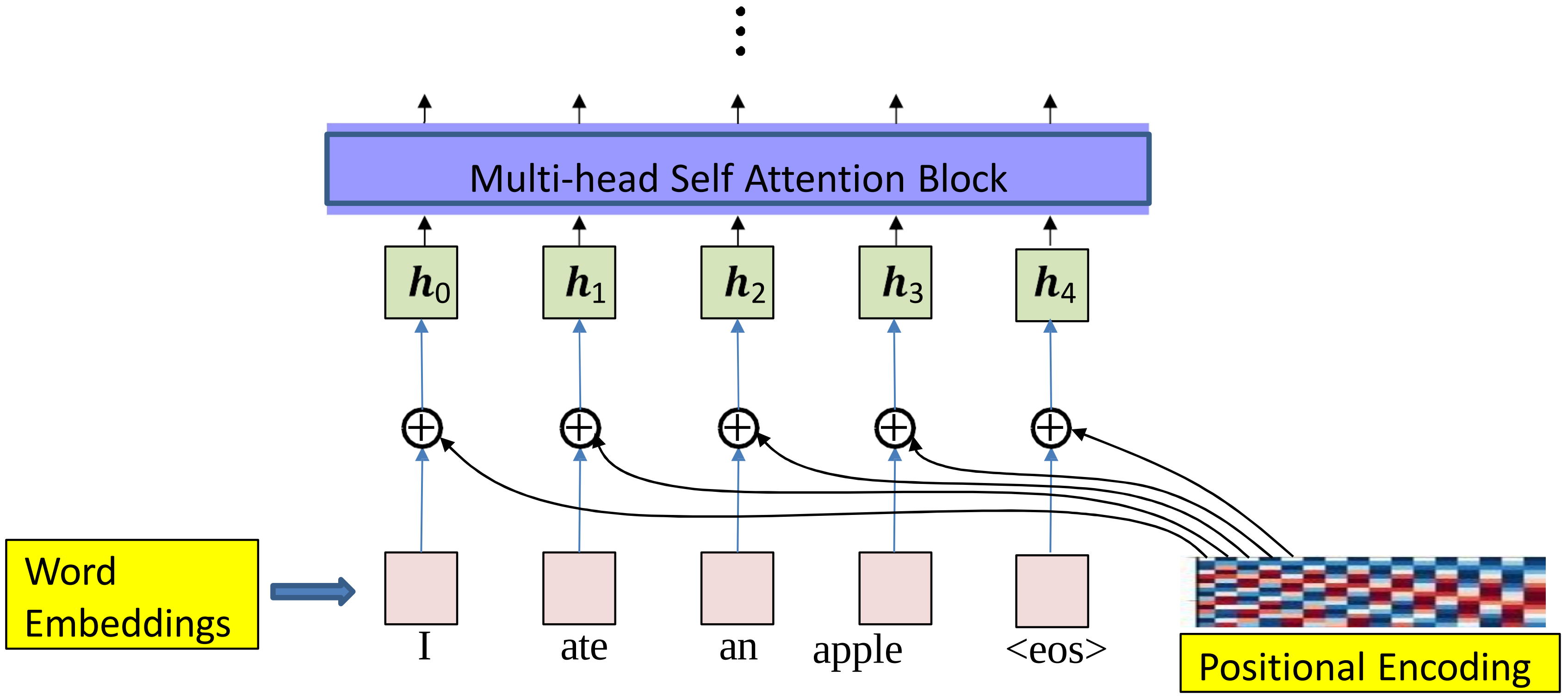
- The encoder can include many layers of such blocks
- No need for recurrence…


Multi-head Self Attention Block
Multi-head Self Attention Block
h_0
h_1
h_2
h_3
h_4
I
ate
an
apple
<eos>

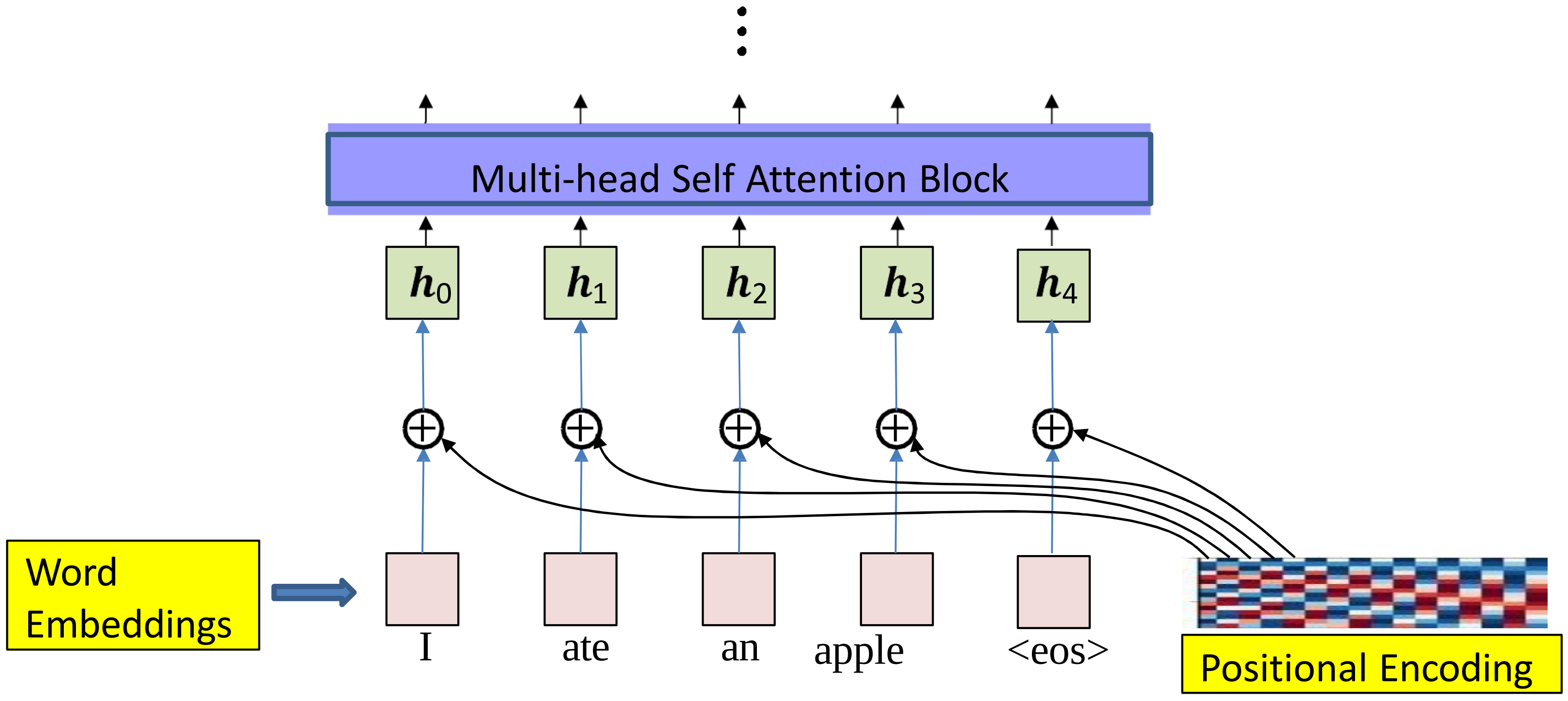
- The encoder in a sequence-to-sequence model can replace recurrence through a series of “multi-head self attention” blocks
- But this still ignores *relative position*
  - A context word one word away is different from one 10 words away
  - The attention framework does not take distance into consideration



- Note that the inputs are actually word *embeddings*



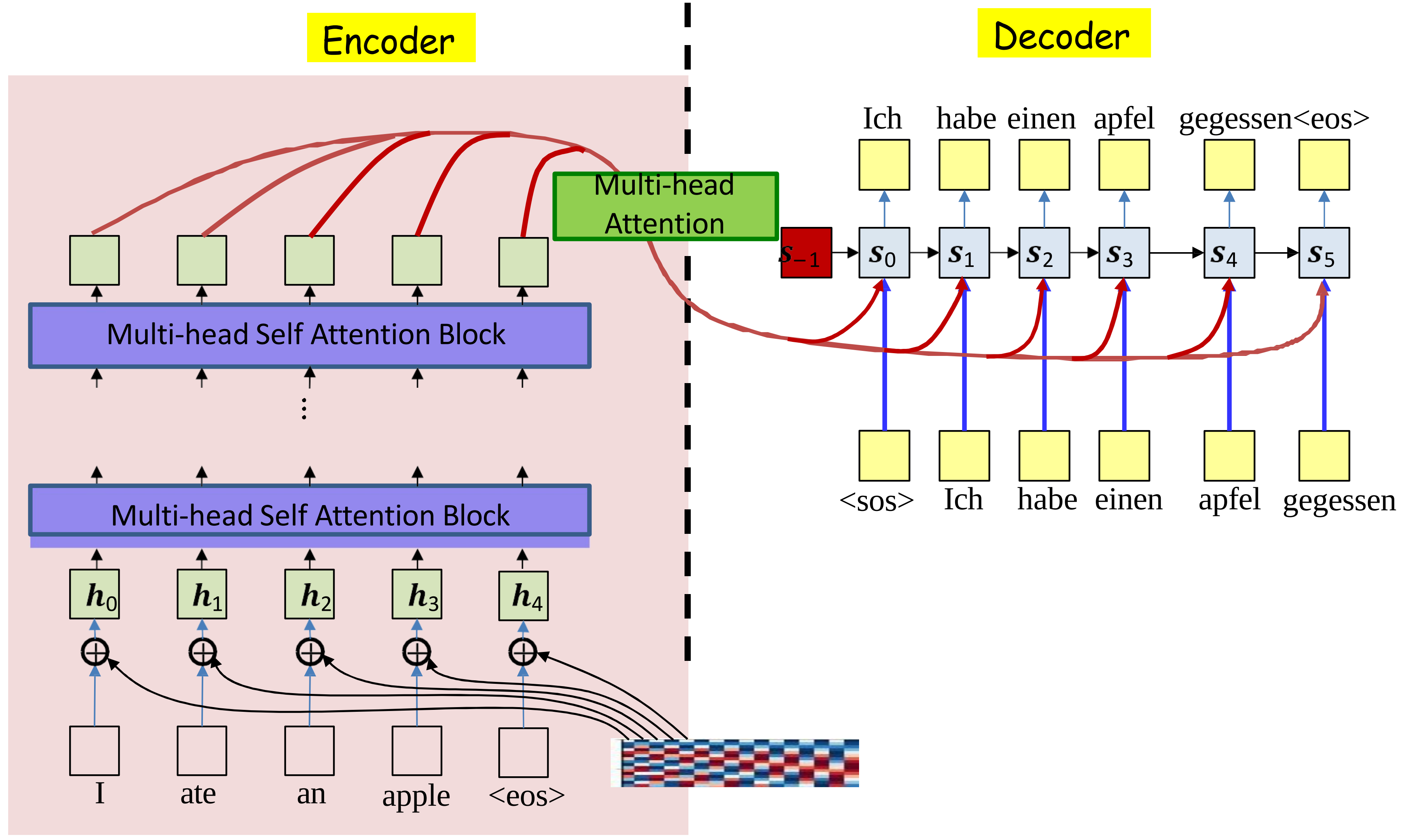
- We add a “positional” encoding to them to capture the relative distance from one another



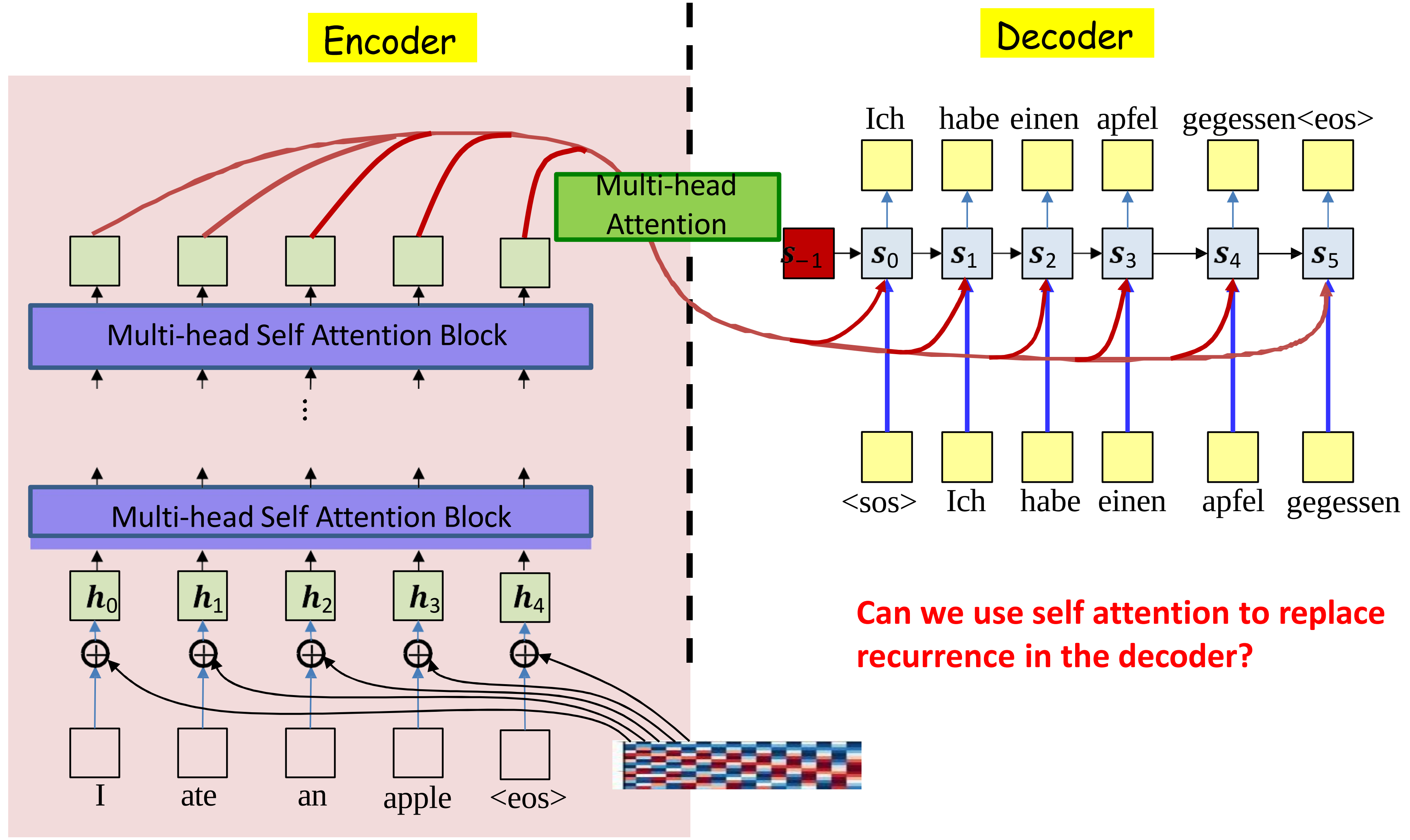
- **Positional Encoding:** A sequence of vectors $P_0, \ldots, P_N$, to encode position
  - Every vector is unique (and uniquely represents time)
  - Relationship between P_t and P_{t+c} only depends on the distance between them

$$P_{t+c} = M_c P_t$$

- The linear relationship between P_t and P_{t+c} enables the net to learn shift-invariant "gap" dependent relationships



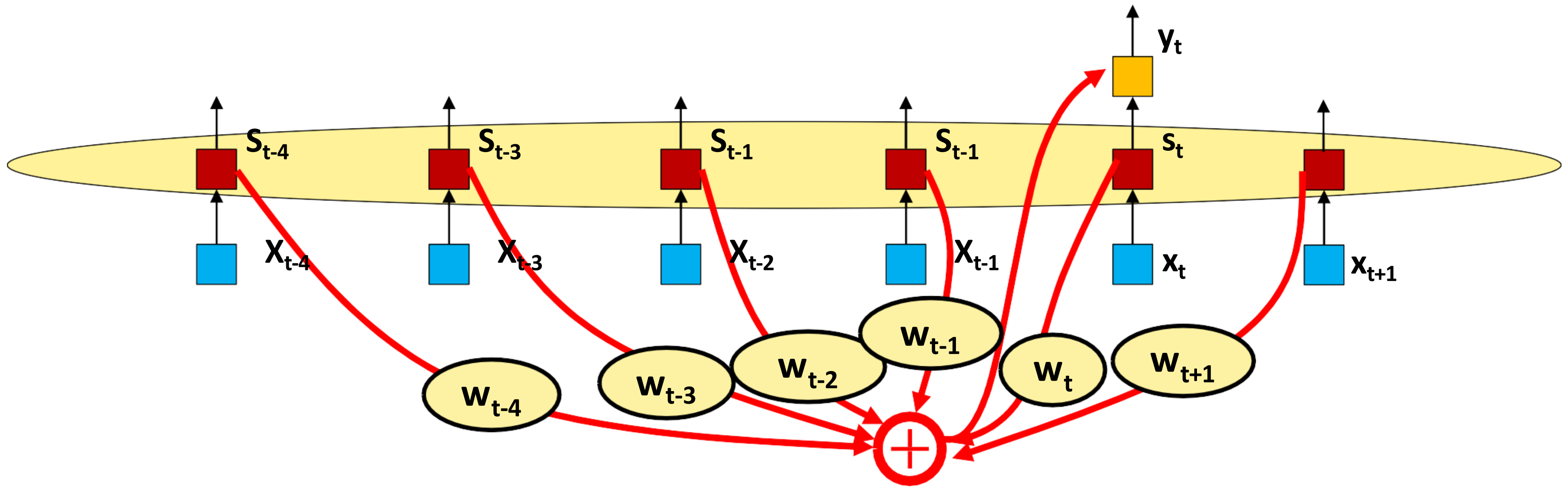
- The self-attending encoder


Encoder
Decoder
Multi-head Attention
Multi-head Self Attention Block
Multi-head Self Attention Block
h_0 h_1 h_2 h_3 h_4
I ate an apple <eos>
s_{-1} s_0 s_1 s_2 s_3 s_4 s_5
Ich habe einen apfel gegessen <eos>
<sos> Ich habe einen apfel gegessen

Can we use self attention to replace recurrence in the decoder?

- The self-attending encoder

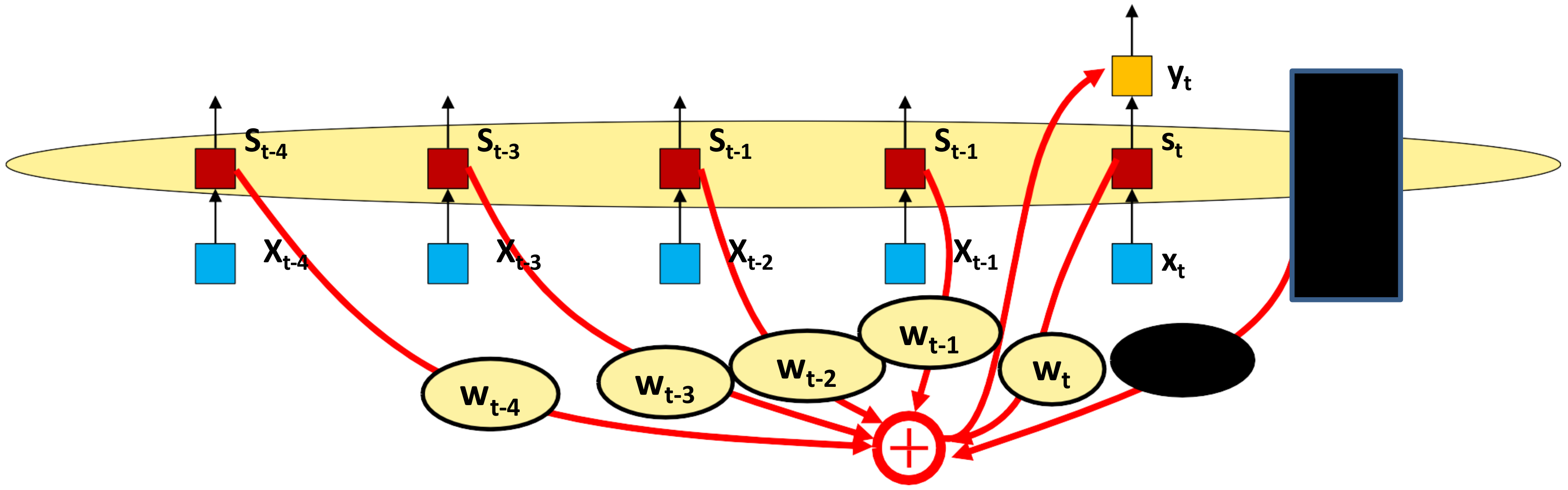
Self attention and masked self attention



- **Self attention in encoder:** Can use input embedding at time t+1 and further to compute output at time t, because all inputs are available

Self attention and masked self attention

- **Self attention in decoder:** Decoder is sequential
  - Each word is produced using the previous word as input
  - Only embeddings until time t are available to compute the output at time t
- The attention will have to be "masked", forcing attention weights for t+1 and later to 0



Masked self-attention block

$q_i = W_q h_i$

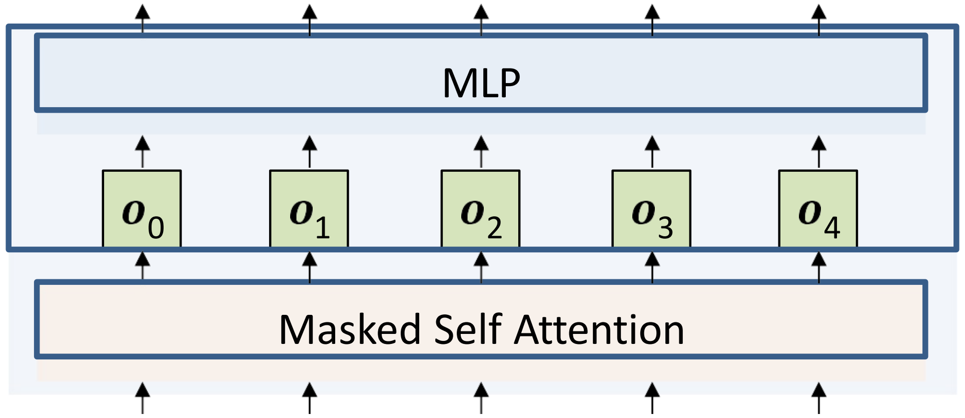
$k_i = W_k h_i$

$v_i = W_v h_i$

$e_{ij} = q_i^T k_j$

$w_{i0}, \ldots, w_{ii} = softmax(e_{i0}, \ldots, e_{ii})$

$$o_i = \sum_{j=0}^{i-1} w_{ij} v_j$$



- The "masked self attention ***block***" includes an MLP after the masked self attention
  - Like in the encoder

Masked self-attention block

$q_i = \boldsymbol{W}_q h_i$

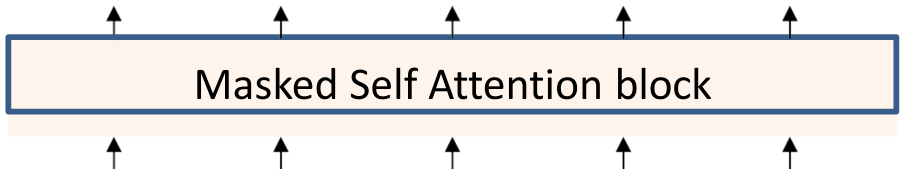
$k_i = \boldsymbol{W}_k h_i$

$v_i = \boldsymbol{W}_v h_i$

$e_{ij} = q_i^T k_j$

$w_{i0}, \dots, w_{ii} = softmax(e_{i0}, \dots, e_{ii})$

$$o_i = \sum_{j=0}^{i-1} w_{ij} v_j$$



- The "masked self attention ***block***" sequentially computes outputs begin to end
  - Sequential nature of decoding prevents outputs from being computed in parallel
  - Unlike in an encoder

Masked multi-head self-attention block

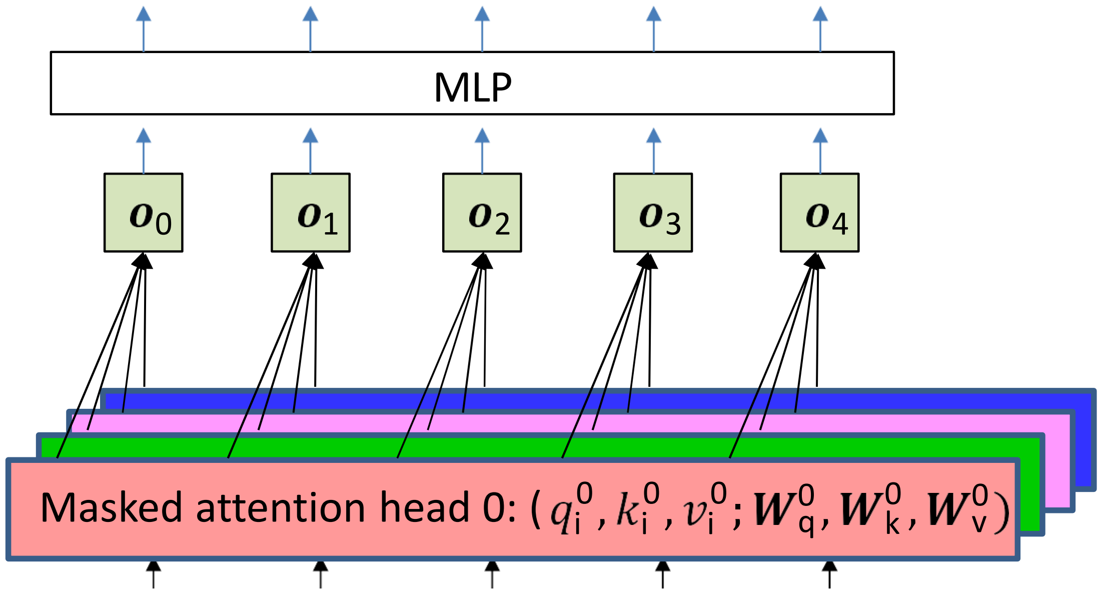
$q_i^a = \boldsymbol{W}_q^a h_i$

$k_i^a = \boldsymbol{W}_k^a h_i$

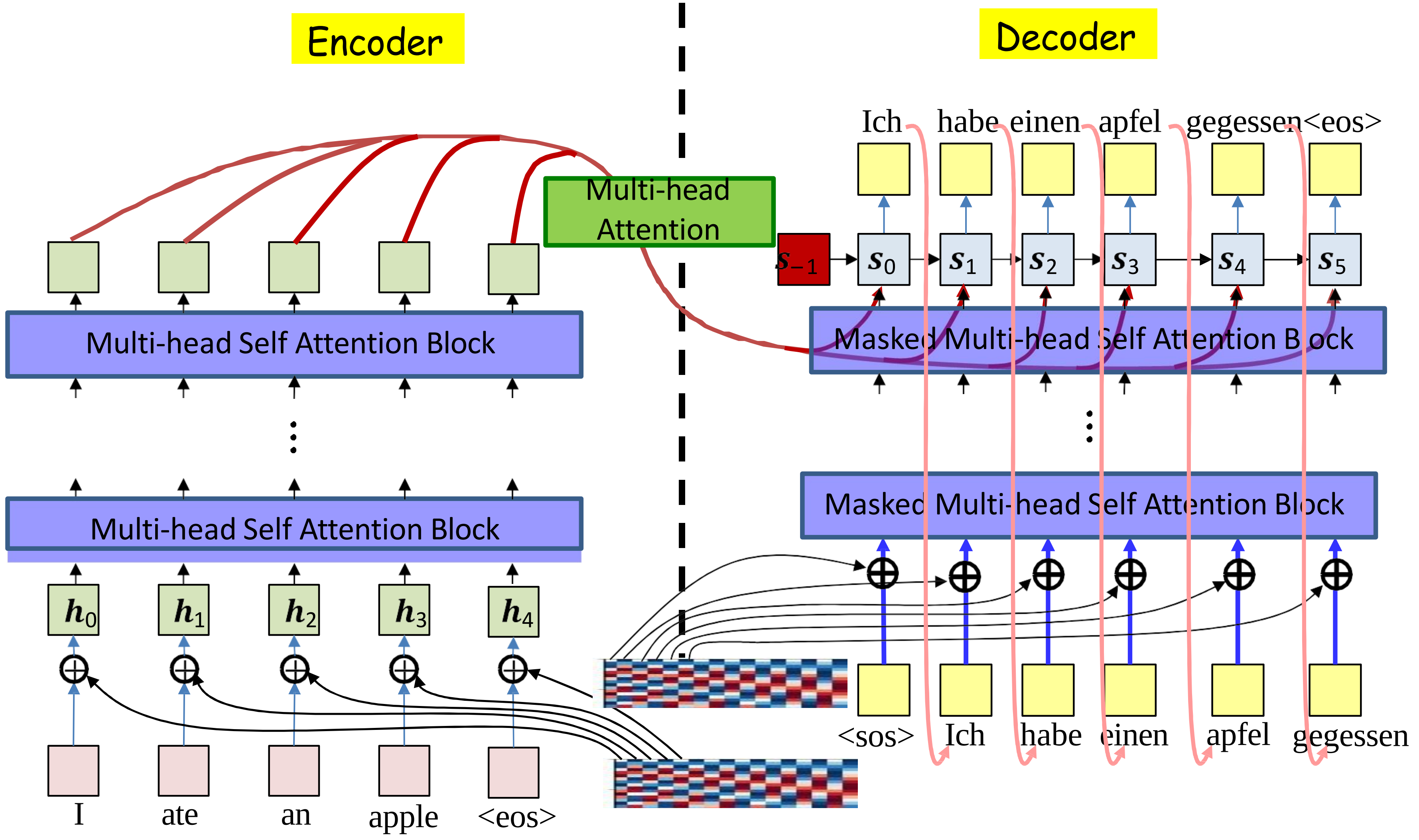
$v_i^a = \boldsymbol{W}_v^a h_i$

$w_{ij}^a = attn(q_i^a, k_{0:i-1}^a)$

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

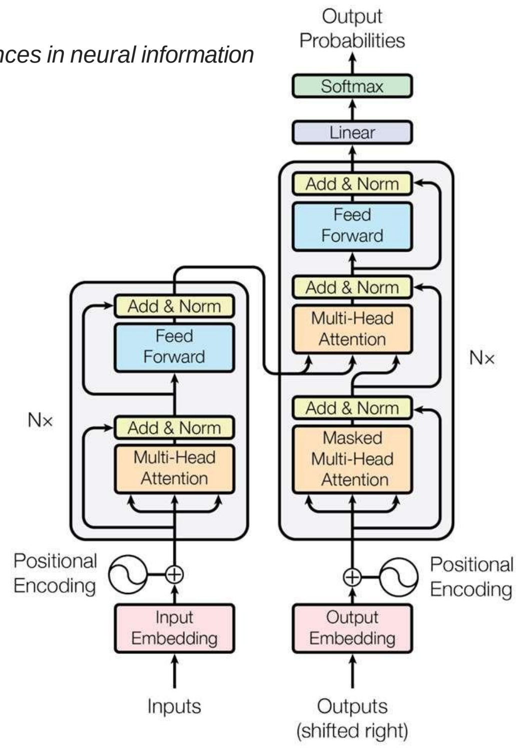

MLP
$\boldsymbol{o}_0$
$\boldsymbol{o}_1$
$\boldsymbol{o}_2$
$\boldsymbol{o}_3$
$\boldsymbol{o}_4$
Masked attention head 0: $(q_i^0, k_i^0, v_i^0; \boldsymbol{W}_q^0, \boldsymbol{W}_k^0, \boldsymbol{W}_v^0)$

- The "masked ***multi-head*** self attention ***block***" includes multiple masked attention heads
  - Like in the encoder


Encoder
Decoder
Multi-head Attention
Multi-head Self Attention Block
Multi-head Self Attention Block
h_0 h_1 h_2 h_3 h_4
I ate an apple <eos>
Ich habe einen apfel gegessen <eos>
s_{-1} s_0 s_1 s_2 s_3 s_4 s_5
Masked Multi-head Self Attention Block
Masked Multi-head Self Attention Block
<sos> Ich habe einen apfel gegessen

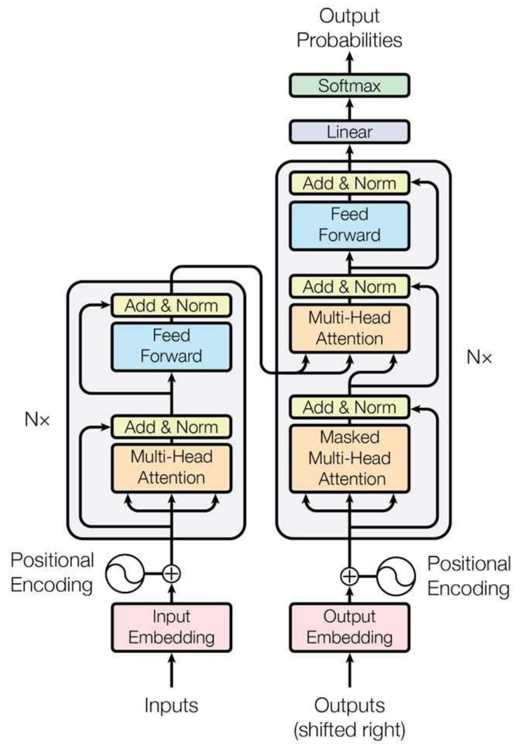
Transformer: Attention is all you need

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems*. 2017.



- Transformer: A sequence-to-sequence model that replaces recurrence with positional encoding and multi-head self attention
  - "Attention is all you need"

Transformer



From “Attention is all you need”

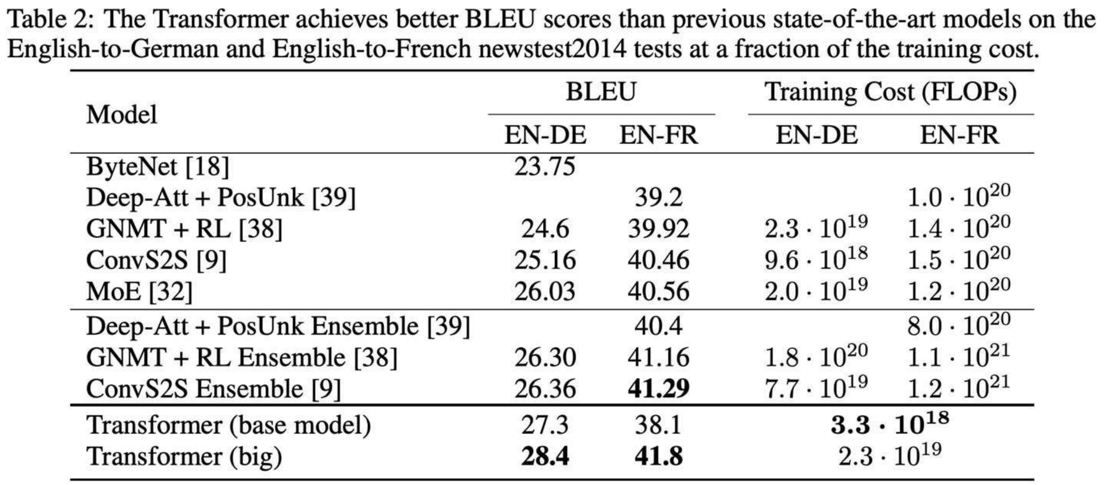
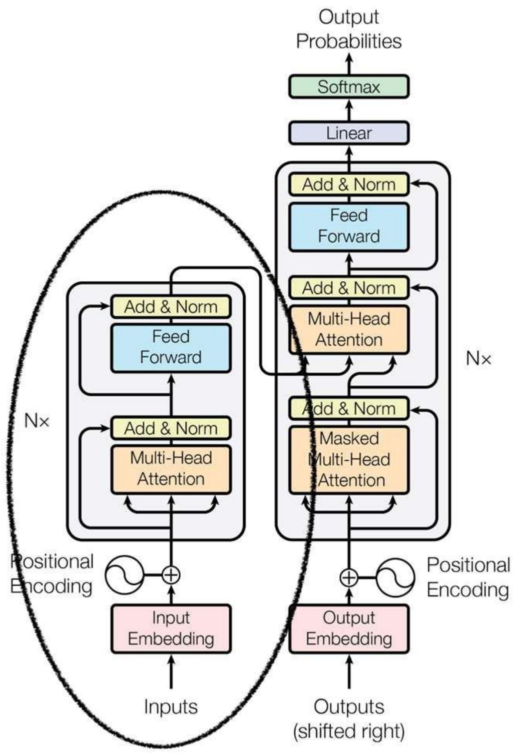


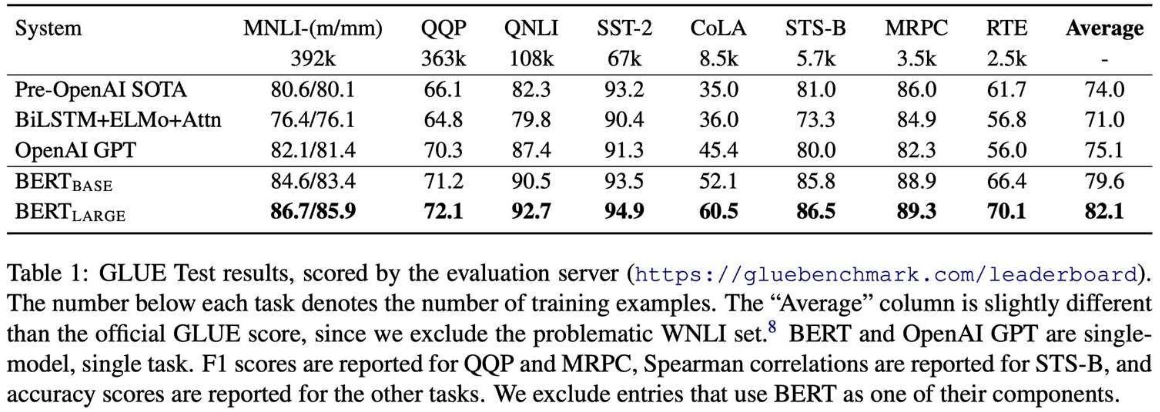
Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [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	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

- Transformer: tremendous decrease in model computation for similar performance as state-of-art translation models
- The last row in the table shows transformer performance
- The final two columns show computational cost.

BERT



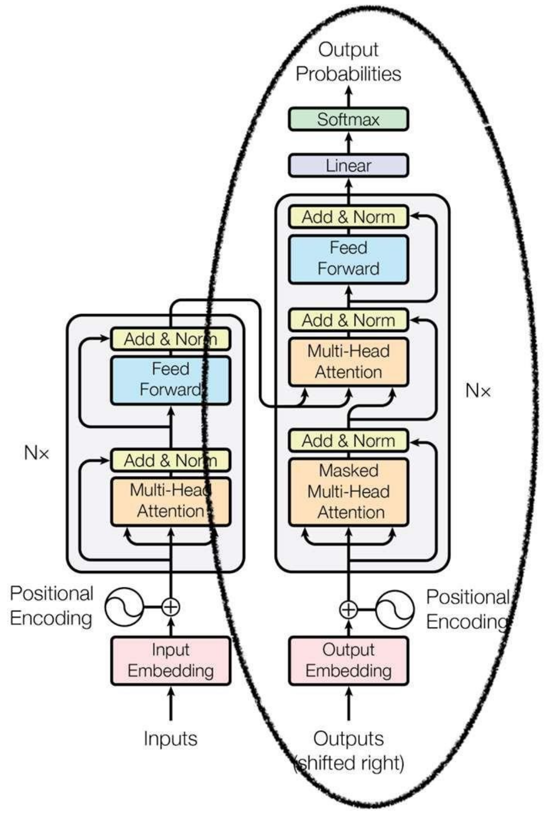


System	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	**Average**
	392k	363k	108k	67k	8.5k	5.7k	3.5k	2.5k	-
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
BERT$_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
BERT$_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

Table 1: GLUE Test results, scored by the evaluation server (`https://gluebenchmark.com/leaderboard`). The number below each task denotes the number of training examples. The "Average" column is slightly different than the official GLUE score, since we exclude the problematic WNLI set.[8] BERT and OpenAI GPT are single-model, single task. F1 scores are reported for QQP and MRPC, Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks. We exclude entries that use BERT as one of their components.

- Bert: Only uses encoder of transformer to derive word and sentence embeddings
- Trained to "fill in the blanks"
- This is *representation learning*

GPT



Alec Radford et. al., Improving Language Understanding by Generative Pre-Training

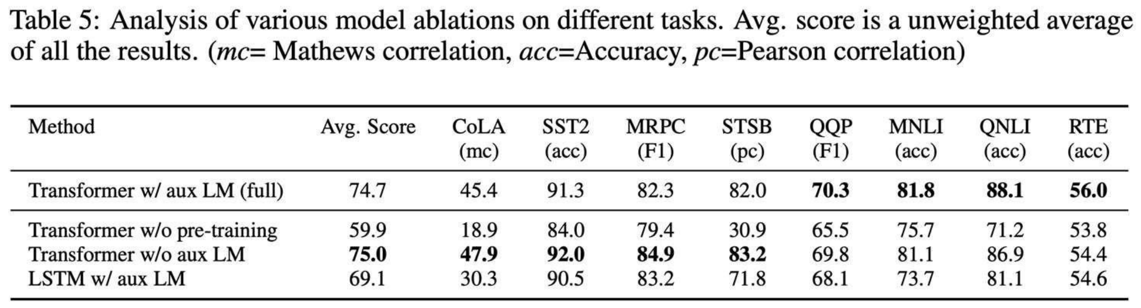


Table 5: Analysis of various model ablations on different tasks. Avg. score is a unweighted average of all the results. (*mc*= Mathews correlation, *acc*=Accuracy, *pc*=Pearson correlation)

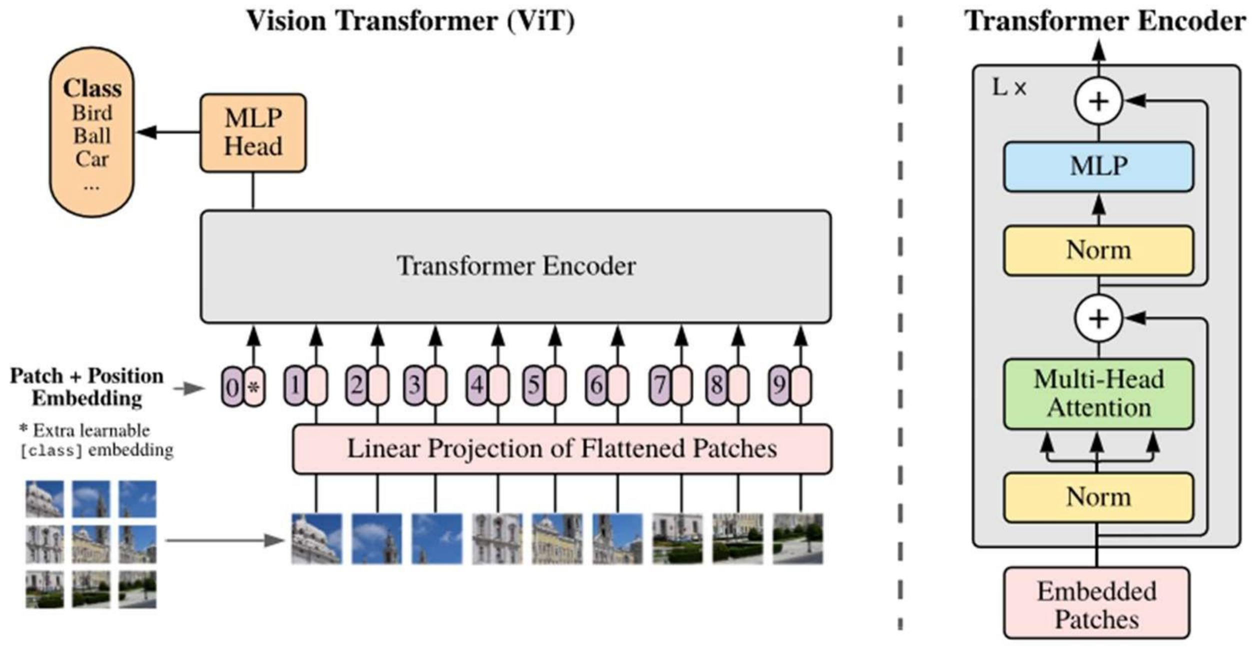
Method	Avg. Score	CoLA (mc)	SST2 (acc)	MRPC (F1)	STSB (pc)	QQP (F1)	MNLI (acc)	QNLI (acc)	RTE (acc)
Transformer w/ aux LM (full)	74.7	45.4	91.3	82.3	82.0	**70.3**	**81.8**	**88.1**	**56.0**
Transformer w/o pre-training	59.9	18.9	84.0	79.4	30.9	65.5	75.7	71.2	53.8
Transformer w/o aux LM	**75.0**	**47.9**	**92.0**	**84.9**	**83.2**	69.8	81.1	86.9	54.4
LSTM w/ aux LM	69.1	30.3	90.5	83.2	71.8	68.1	73.7	81.1	54.6

- GPT uses only the decoder of the transformer as an LM
  - "Transformer w/o aux LM"
- Large performance improvement in many tasks

Attention is all you need

- Self-attention can effectively replace recurrence in sequence-to-sequence models
  - "Transformers"
  - Requires "positional encoding" to capture positional information
- Can also be used in regular sequence analysis settings as a substitute for recurrence
- Currently *the* state of the art in most sequence analysis/prediction… and even computer vison problems!

Vision Transformers



Dosovitskiy et al, *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, 2020

- Divide your image in patches with pos. encodings
- Apply Self-Attention!
- Sequential and image problems are similar when using transformers