

UNIVERSITY OF
TORONTO

Applied Deep Learning

Lecture 9. *Sequence Models and Transformers*

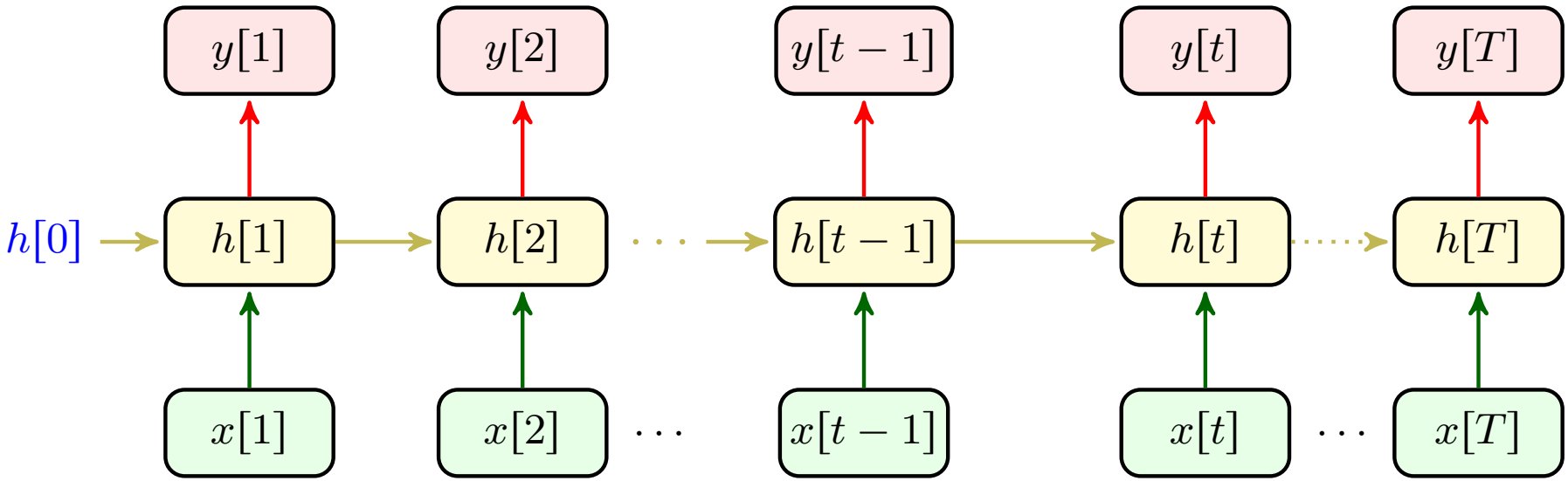
Ali Bereyhi
ECE Department, University of Toronto
Fall 2026



UNIVERSITY OF
TORONTO

Lecture Overview

Recap: *Sequence Processing by Recursion*



- Basic RNN has short local memory
- We can control this memory over a longer period via gating

Lecture 9. *Sequence Models and Transformers*

Today, we learn sequence-to-sequence models

1. Seq2Seq Models
   - *Different types of sequence problem*
   - *Basic language model*
   - *Encoder-decoder architecture as a general solution*
2. Machine Translation and Transformer
   - *Challenges of sequential processing*
   - *Attention mechanism and self-attention*
   - *Encoder-decoder design using pure attention*



UNIVERSITY OF
TORONTO

Sequence Models

Learning Sequence from Sequence

In many applications, our dataset contains data-points that of the form

$$(\mathbf{x}[t], \mathbf{v}[t])$$

This is still standard supervised learning where

- *input data is a sequence, and*
- *label is a sequence*

We have already seen some examples

- *We want to predict next words $\equiv$ language model*
  - ↳ *Dataset contains sequences of sentences with label being last word*
- *We want to caption an image*
  - ↳ *Dataset contains images with sequences of caption sentences*
- *We want to train a machine translator*
  - ↳ *Dataset contains sequences of German sentences with English translations*

Sequence-to-Sequence Problem

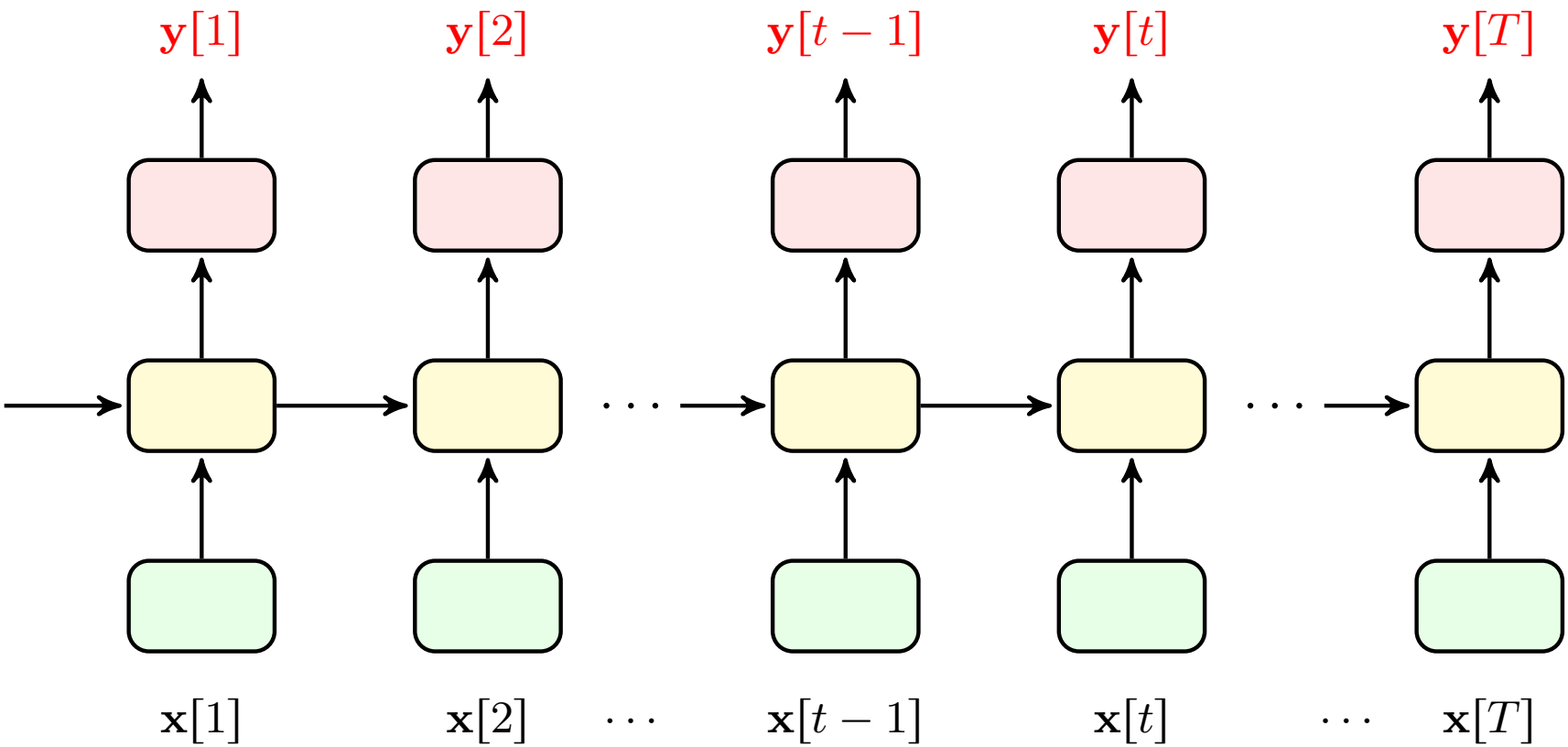
Sequence-to-Sequence (Seq2Seq) Models

A Seq2Seq model is a model, e.g., a NN, that takes a sequence as an input and returns an output sequence

+ *Then isn't RNN a Seq2Seq model?*
– Sure! Strictly speaking *even MLPs and CNNs are Seq2Seq models with sequences of length 1!*

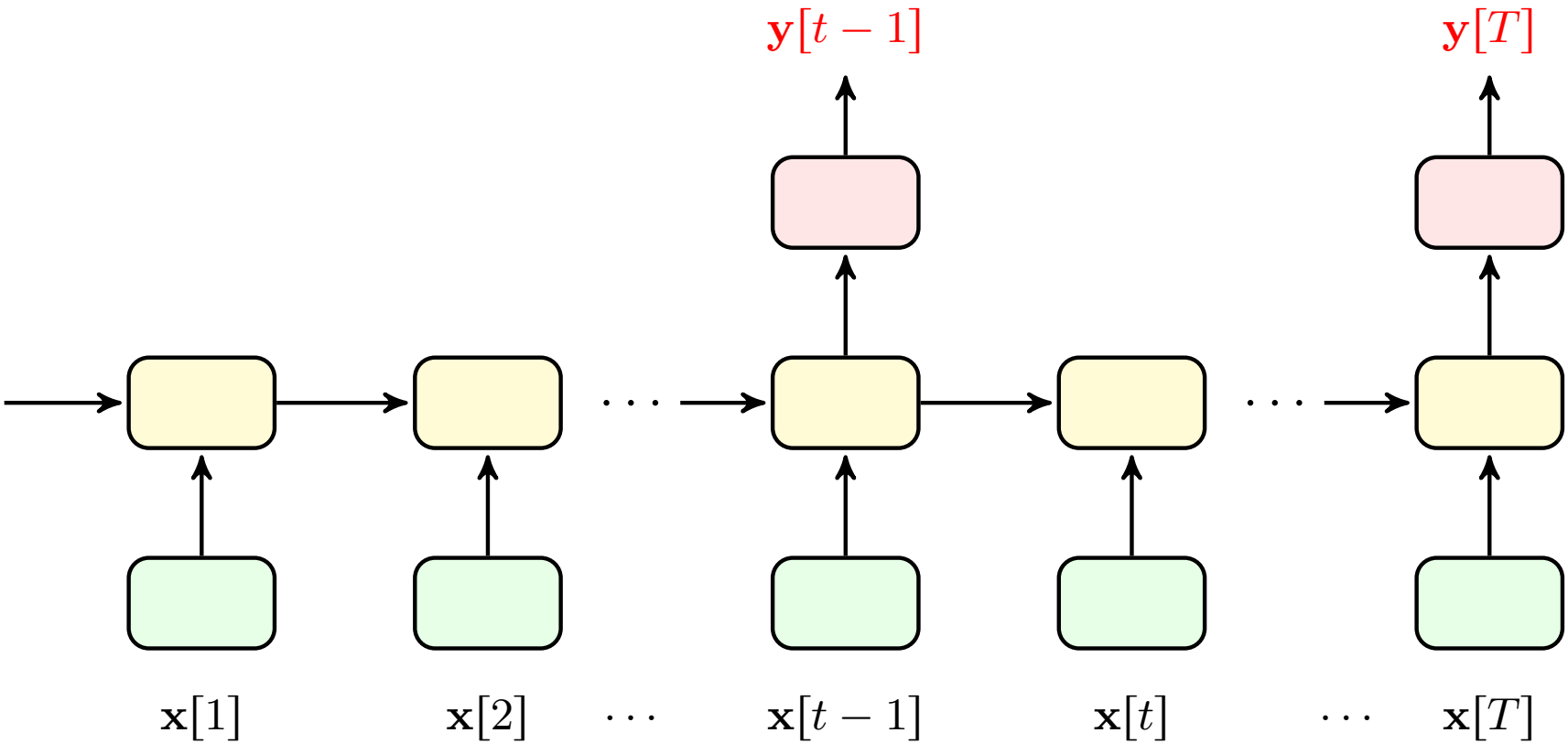
Despite this definition, when we talk about Seq2Seq models in practice, we mainly refer to architectures with encoder and decoder

Types of Sequence Problems: *Many-to-Many I*



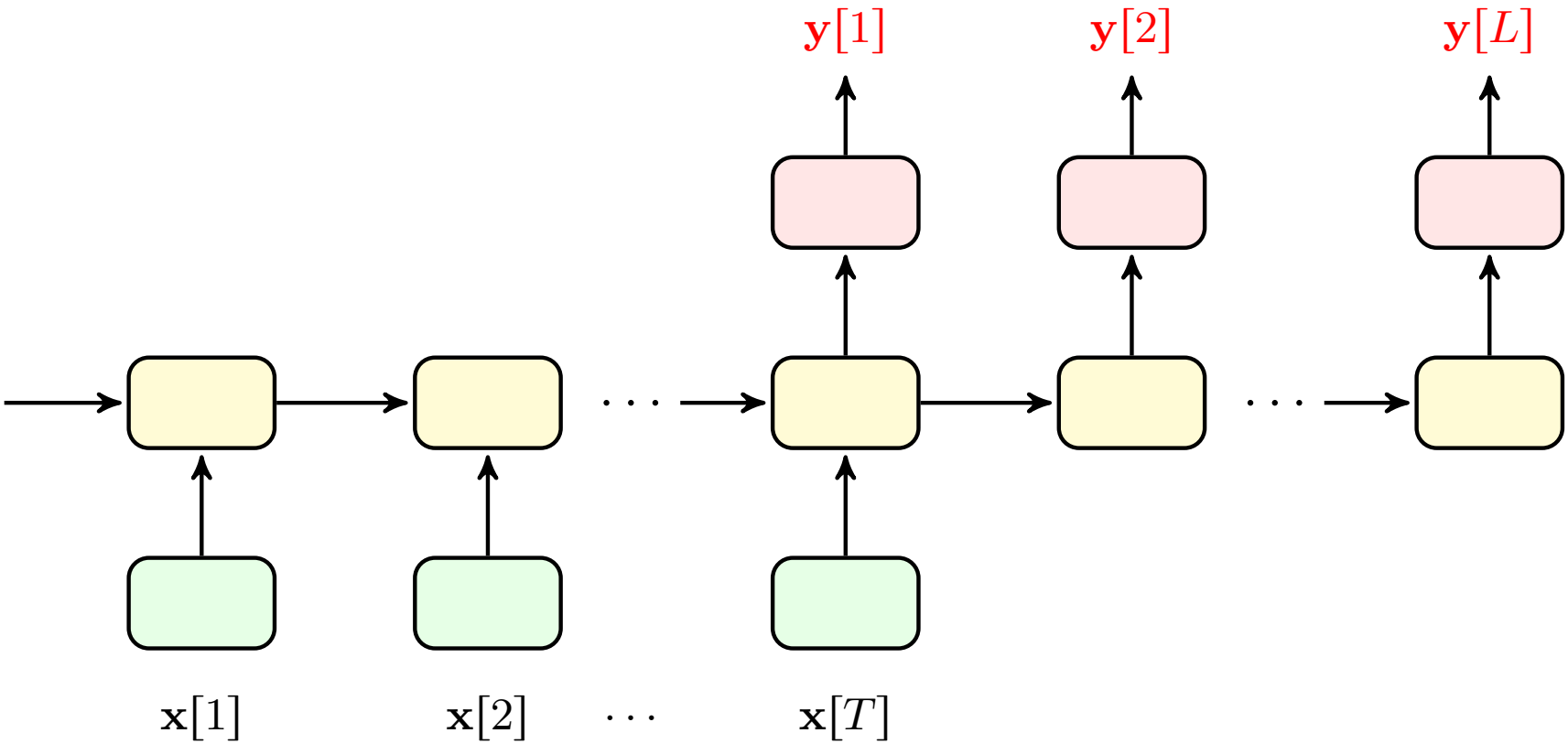
Speech tagging where $\mathbf{x}[t]$ is a part of speech and $\mathbf{y}[t]$ is its tag

Types of Sequence Problems: *Many-to-Many II*



Speech recognition where $\mathbf{x}[t]$ is audio frame and $\mathbf{y}[t]$ is topic

Types of Sequence Problems: *Many-to-Many III*



Language translation where $\mathbf{x}[1:T]$ is a sentence and $\mathbf{y}[1:L]$ is its translation

Types of Sequence Problems: *One-to-Many*

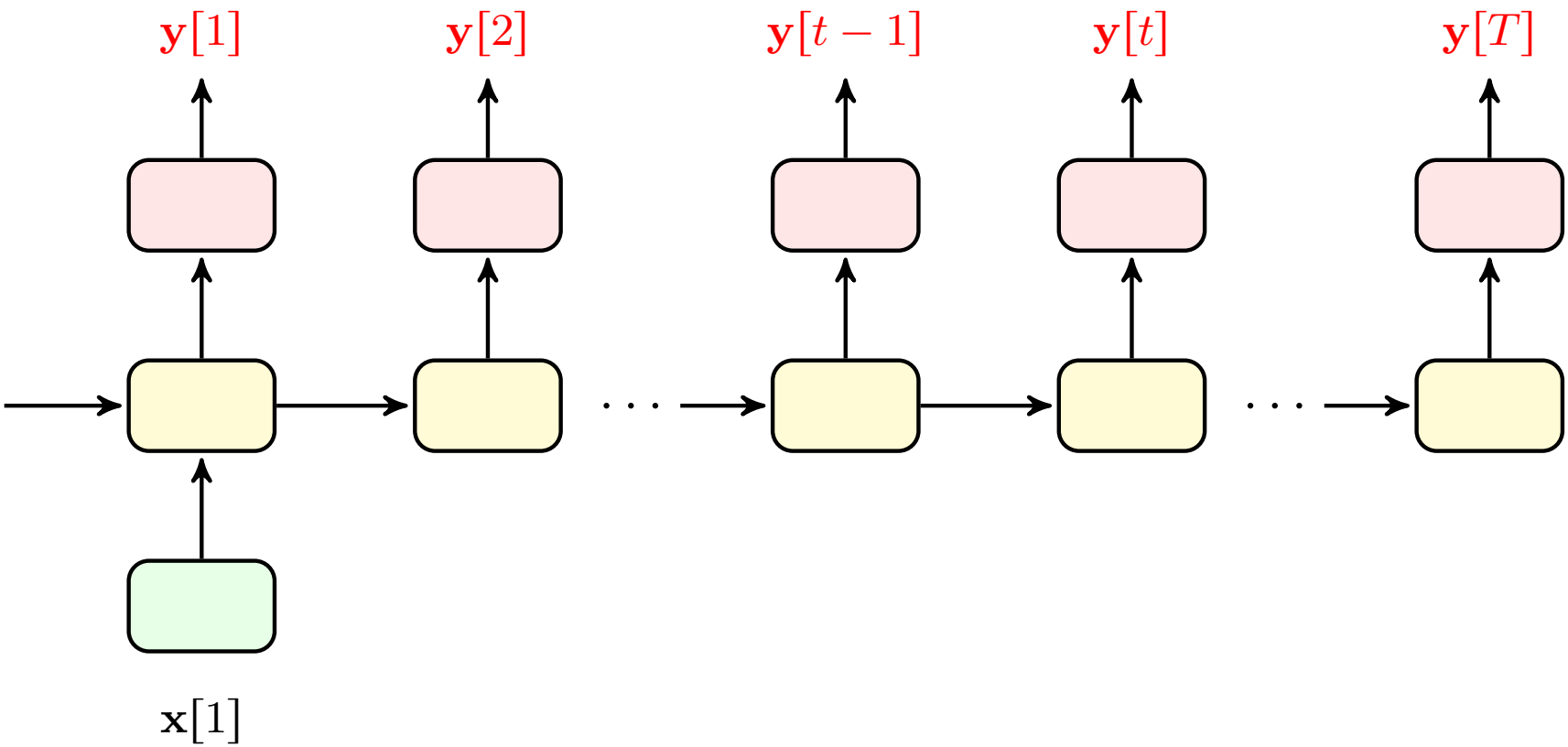
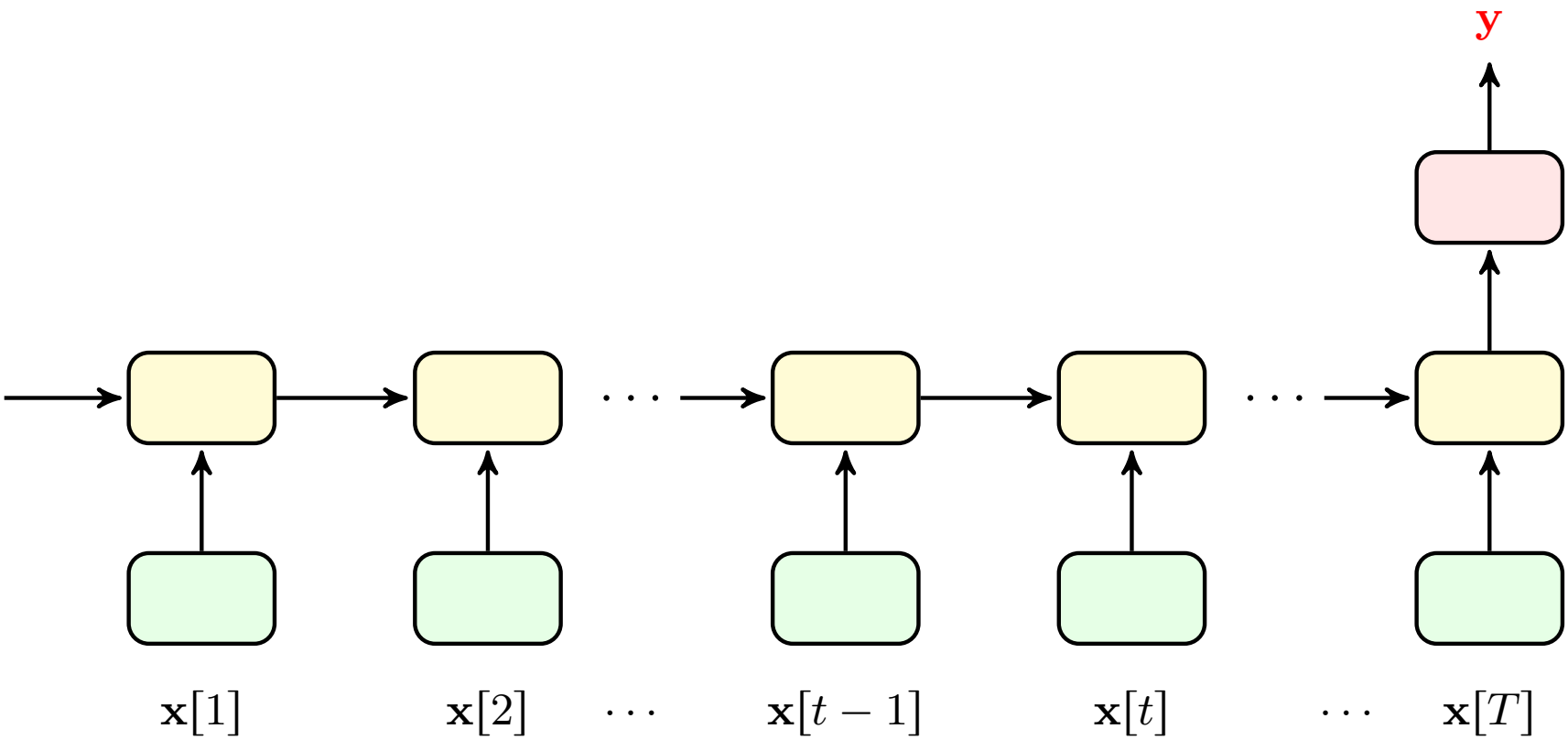


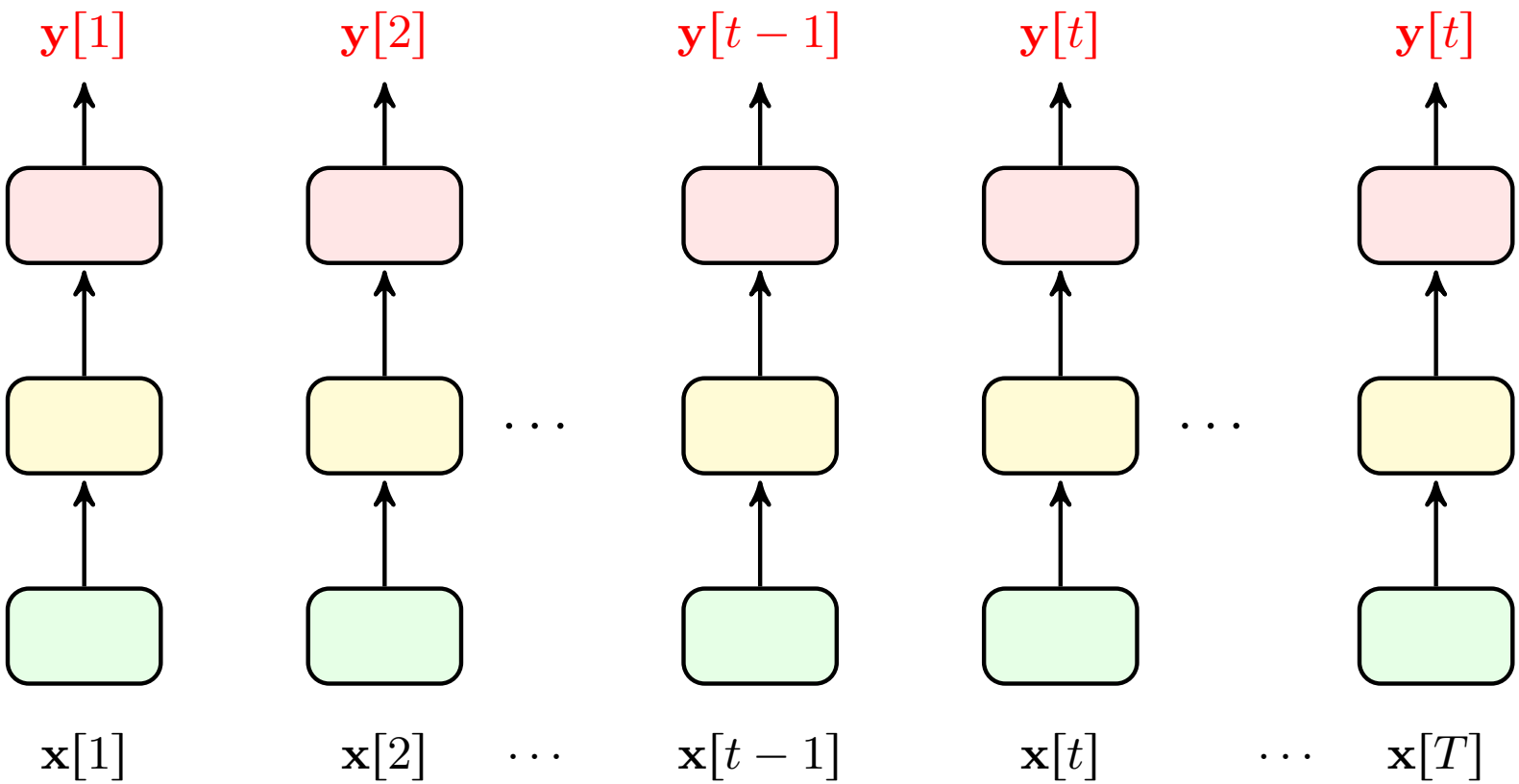
Image captioning where $\mathbf{x}[1]$ is an image and $\mathbf{y}[1:T]$ is a caption

Types of Sequence Problems: *Many-to-One*



Sentiment analysis where $\mathbf{x}[1:T]$ is a speech and $\mathbf{y}$ is its mood

Types of Sequence Problems: *One-to-One*



What we have had in the first part of course!

Basic Language Model

We intend to train a model that generates coherent sentences

this is a basic language model

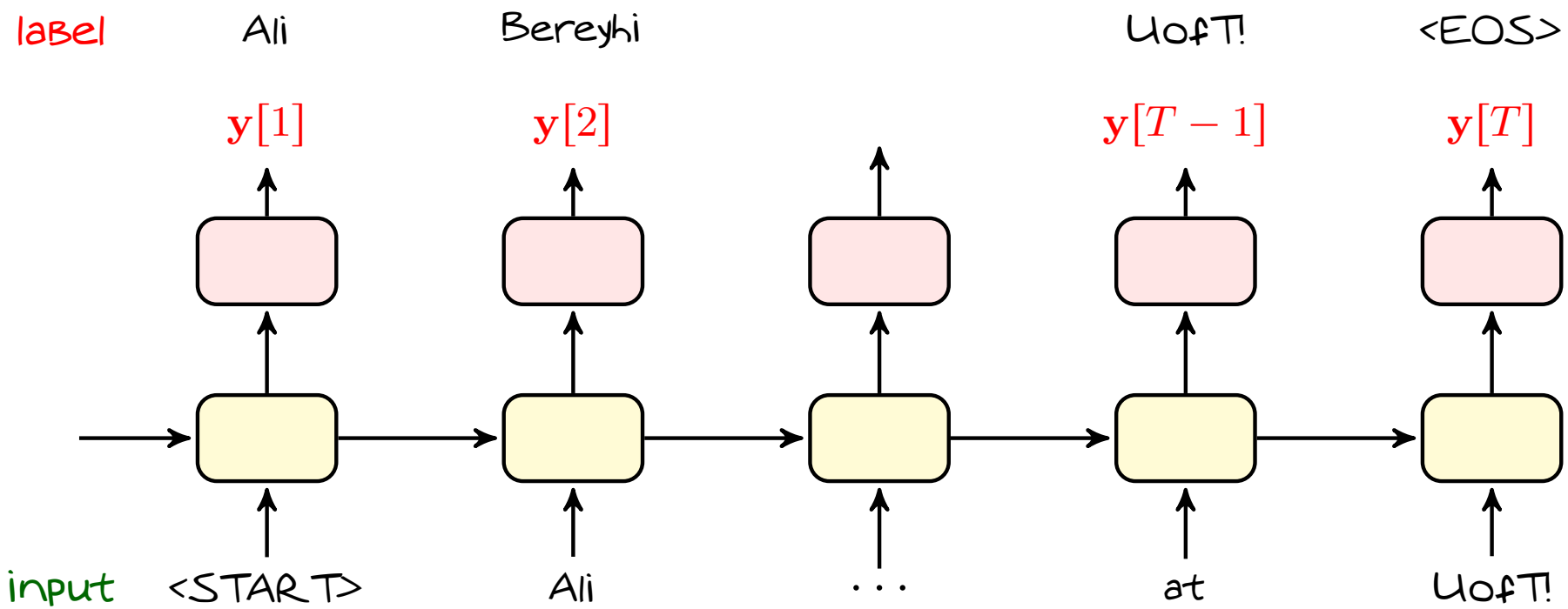
- If we give it an incomplete sentence $\rightsquigarrow$ *it completes*
- If we give it noting $\rightsquigarrow$ *it writes a valid English sentence*

Let's write a dataset first: it contains sequences of coherent English sentences

Ali Bereyhi is the coolest professor at UofT!

- *Sentences are of different lengths*
  - ↳ *Each word in a sentence is one input entry*
  - ↳ *We label each word with its next word in the sentence*

Basic Language Model: *Dataset*



To be able to predict first word and end of sentence we add two new words

- *We tag the beginning of sentence with* <START>
- *We label the end of sentence with* <EOS>

Basic Language Model: *Dataset*

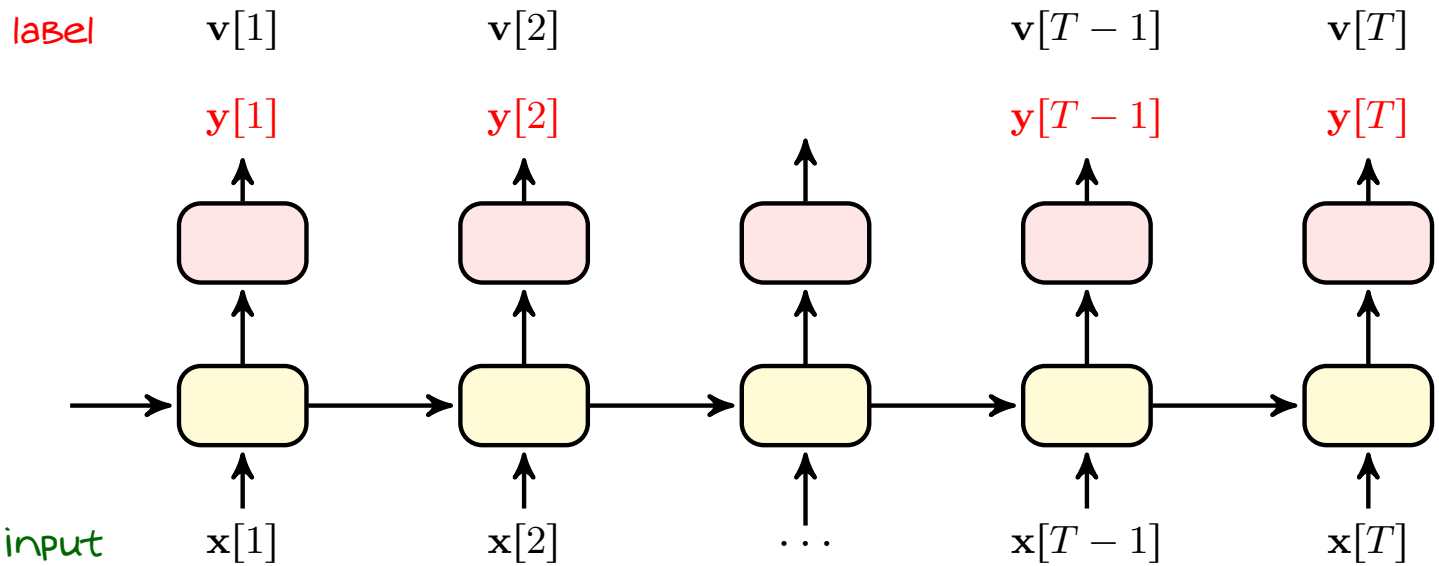
- ? *How can we feed those words to our RNN?*
- We convert them to *vectors*

The basic approach is to make a token for each word

- *We list all possible words and index them by* 1 *to* D
  - ! D *is in practice very large*
- *We add* <START> *with index* 0 *and* <EOS> *with* $D+1$
- *We show each character by its one-hot vector which is called token, e.g.,*

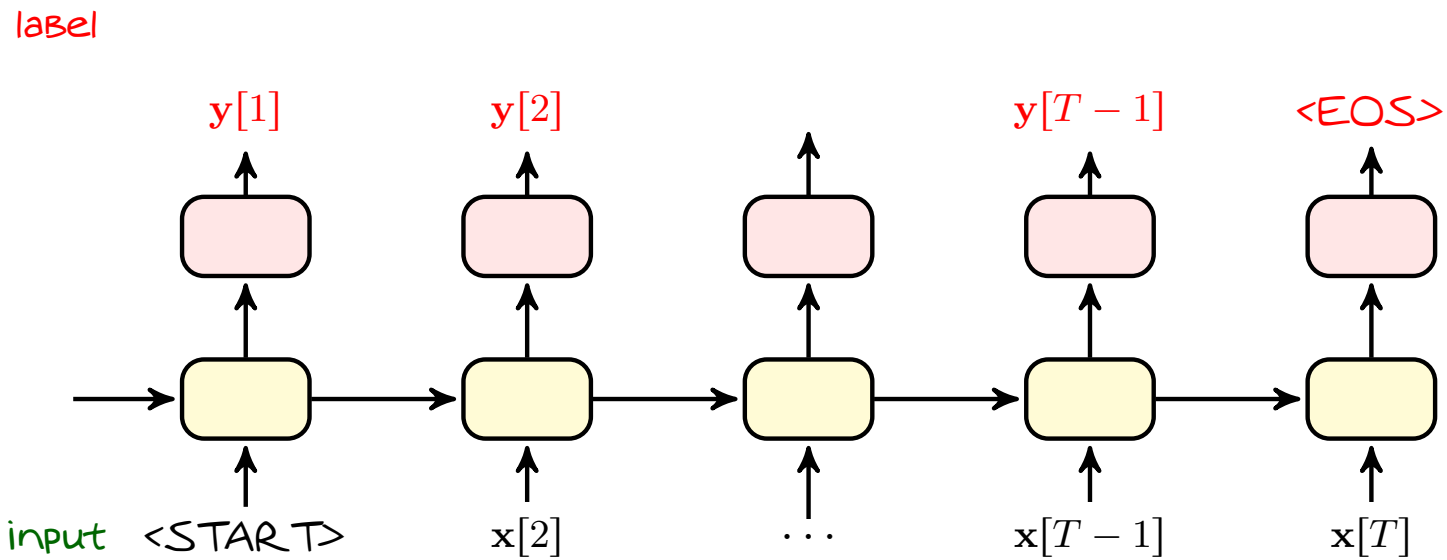
$$\texttt{<START>} \mapsto \begin{bmatrix} 1 \\ 0 \\ \vdots \end{bmatrix} \in \{0,1\}^{D+2} \qquad \texttt{<EOS>} \mapsto \begin{bmatrix} 0 \\ \vdots \\ 1 \end{bmatrix} \in \{0,1\}^{D+2}$$

Basic Language Model: *Training*



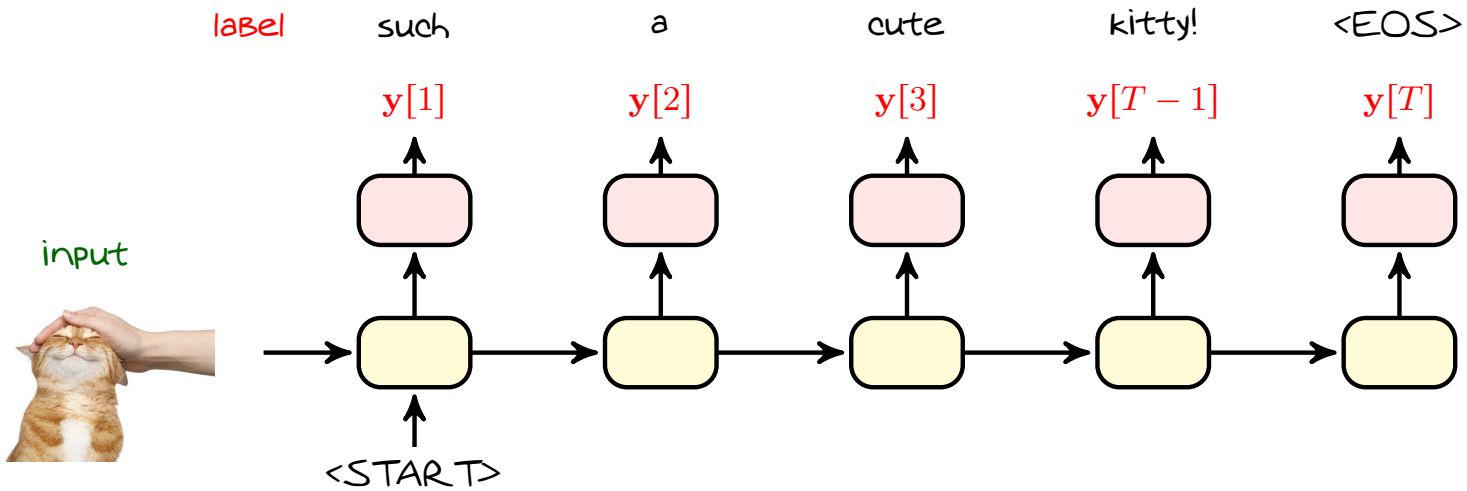
- We now train an RNN on this dataset embedded sentences
  - ↳ *For each data-point we pass first forward and then backward through time*
  - ↳ *We compute aggregated loss for each point and average over mini-batch*

Basic Language Model: *Inference*



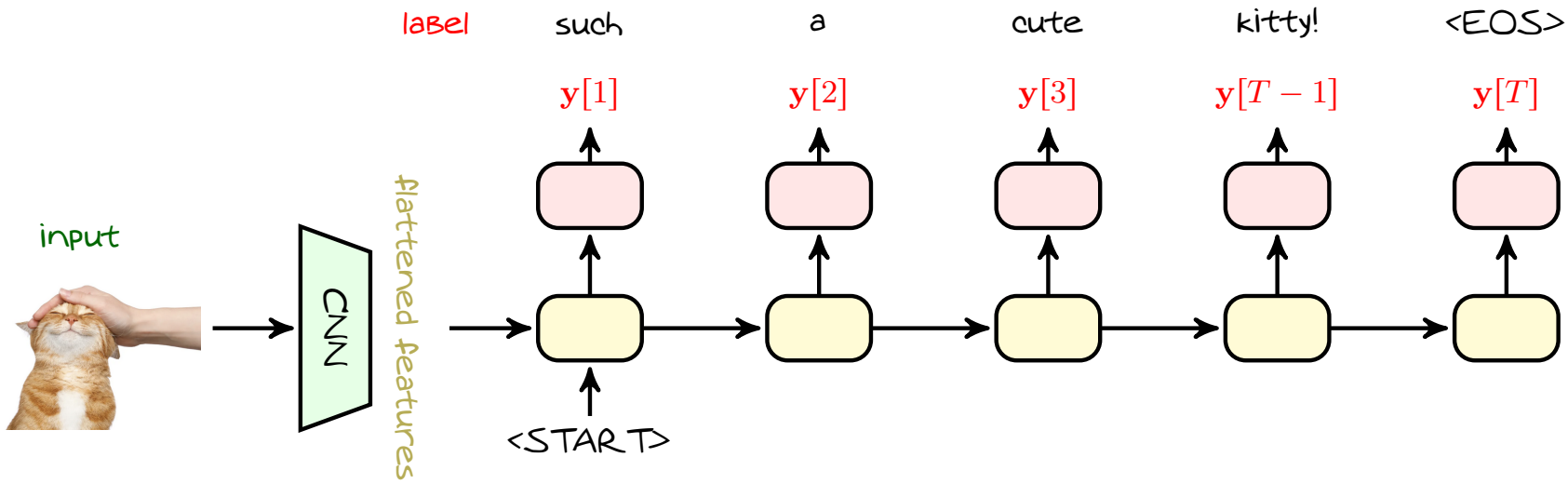
- If we want to complete the sentence, we give the initial part
  - ↳ *It keeps on generating till* <EOS>
- If we want to generate a new sentence, we can give <START>
  - ↳ *It generates a word in each time step*

Caption Generation



- *We can generate a meaningful sentence with our basic language model*
  - ↳ *The sentence is probably not relevant to the image*
  - ↳ *We need to make the model talk about an image*
- *Maybe, we could set initial state of the RNN depending on the image*
  - ↳ *We need to extract features of image*
  - ↳ *Those features are going to explain what is inside the image*

Caption Generation: *Encoder-Decoder*



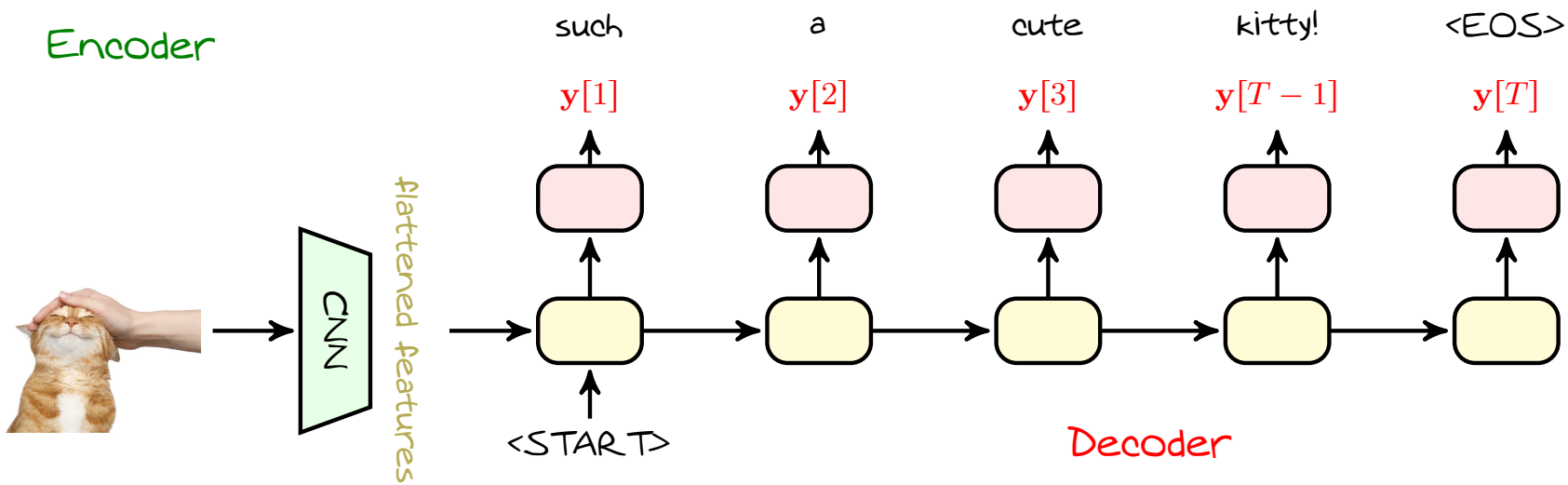
- *We can extract a lot of features from the image via a CNN*
  - ↳ *We flatten those features and give them as initial state to the RNN*

Encoder-Decoder Architecture

Sequence Models ≡ Encoder-Decoder Architectures

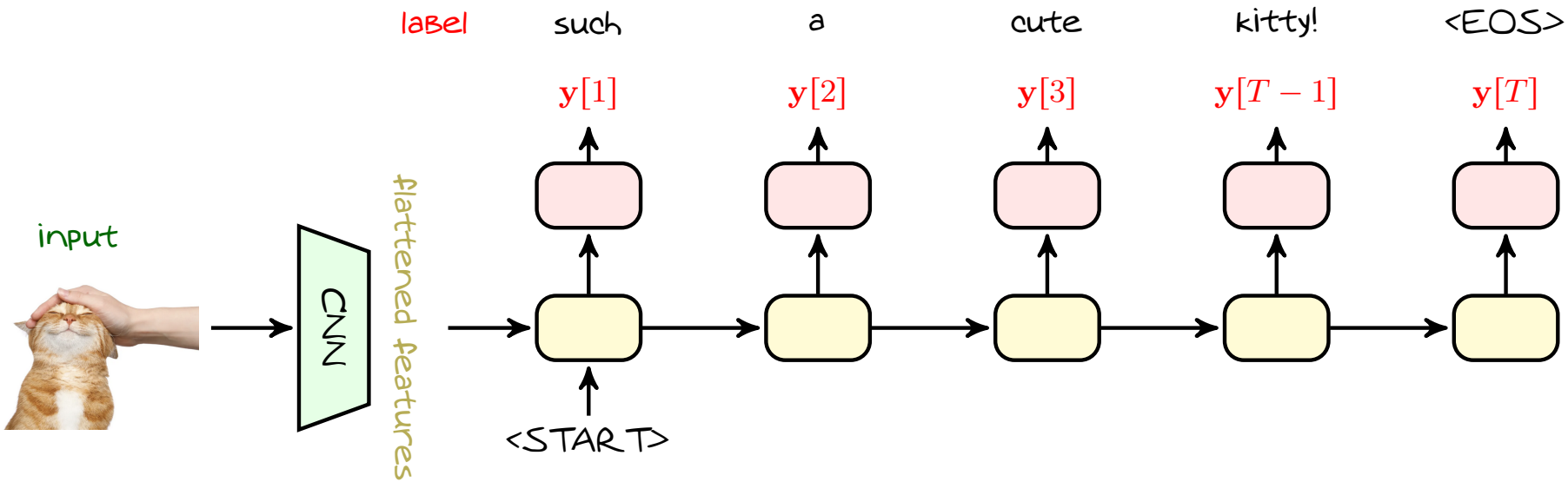
Sequence models comprise of two parts

1. *Encoder takes the input sequence and encodes it into vector of features*
2. *Decoder takes vector of features and decodes it into output sequence*



Back to *Caption Generation*

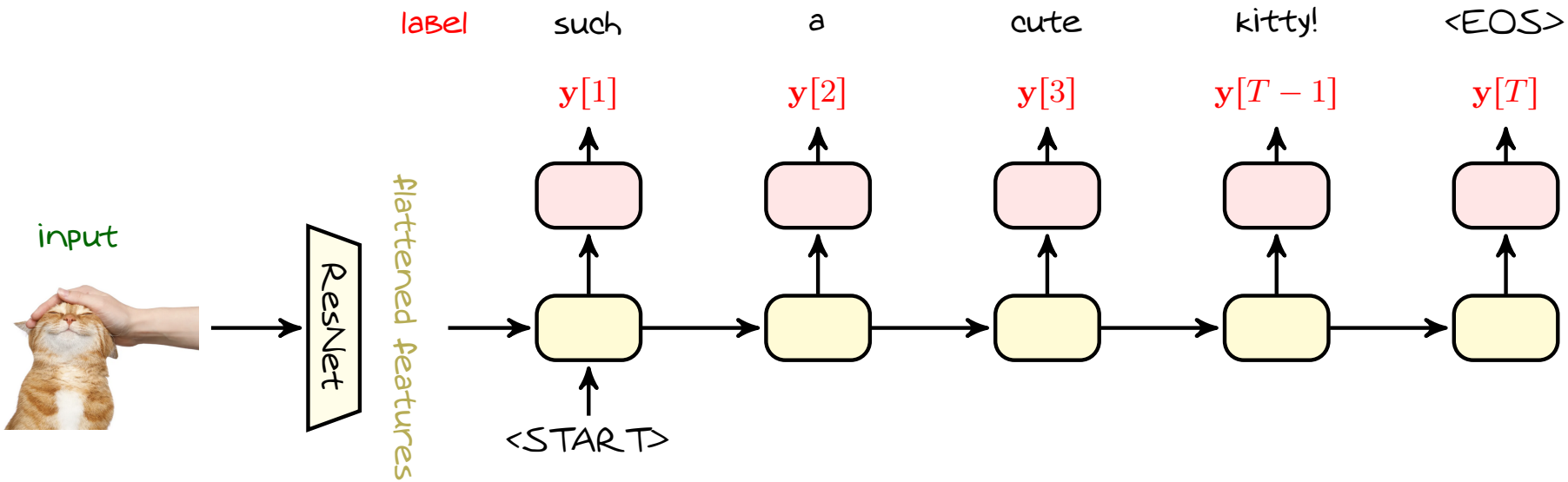
We used a CNN for encoding and an RNN for decoding



We can replace CNN with any other architecture the extracts features

Back to *Caption Generation*

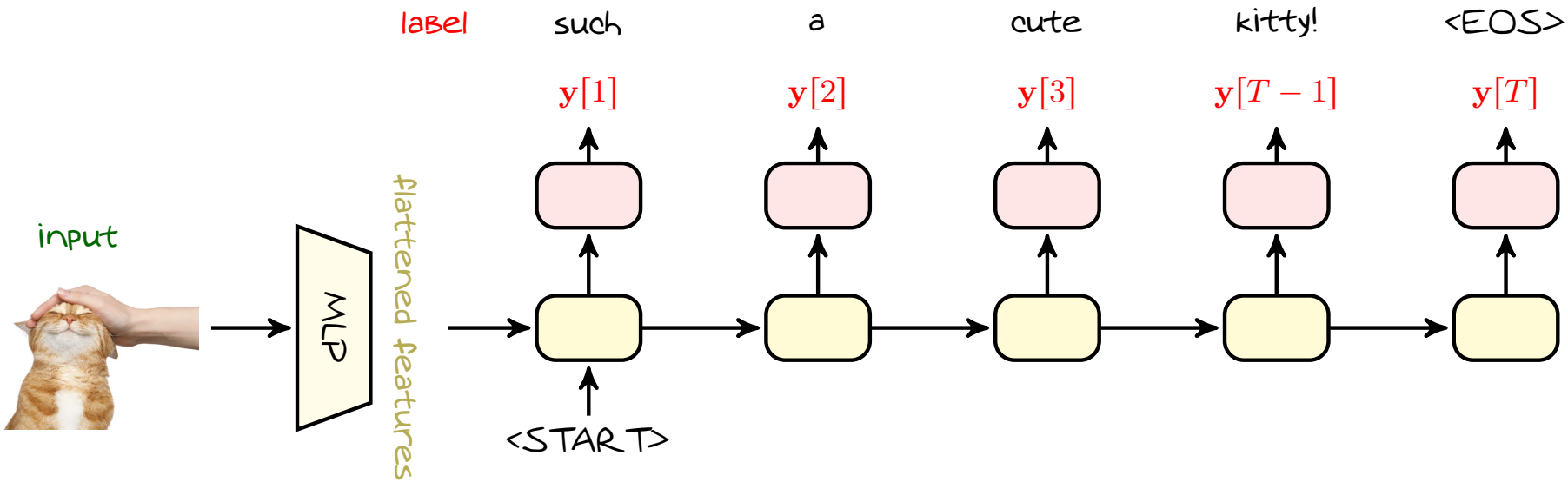
We used a CNN for encoding and an RNN for decoding



We can replace CNN with any other architecture the extracts features

Back to *Caption Generation*

We used a CNN for encoding and an RNN for decoding

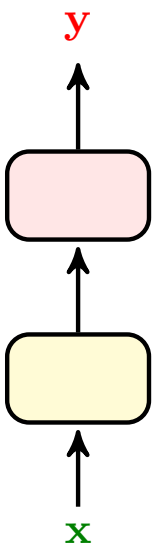


We can replace CNN with any other architecture the extracts features

RNN as Conditional Distribution

- ? *But, why should those features make RNN speak about cat?*
- It makes RNN to *sample random words conditional to input*

NN can be seen as a machine that computes distribution *and based on its input, it samples random outputs*



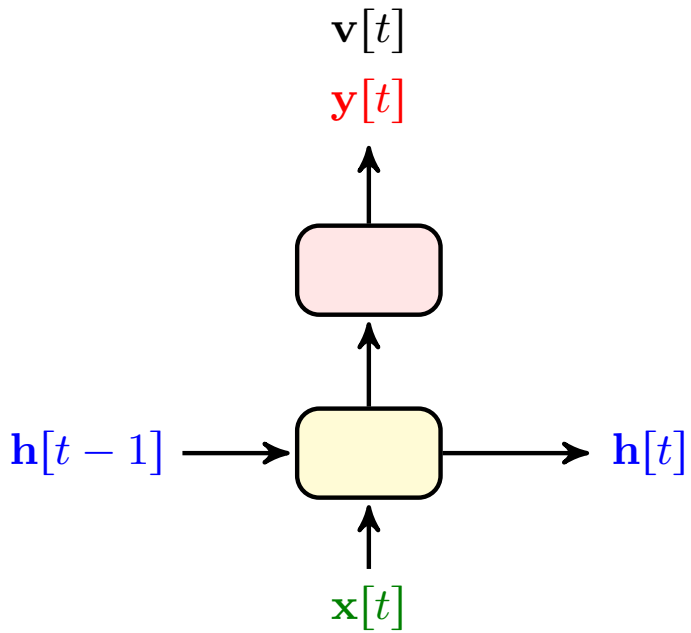
In classification $\mathbf{y}$ *is a vector if probability*

- *Its length equals to the number of classes*
- *Its entry* k *represents the probability of class* k
- *We can say that*

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \dots \\ y_K \end{bmatrix} \leftrightsquigarrow y_k \propto \Pr\{\text{label} = k | \mathbf{x}\}$$

- *We typically sample the most likely label*

RNN as Conditional Sampling



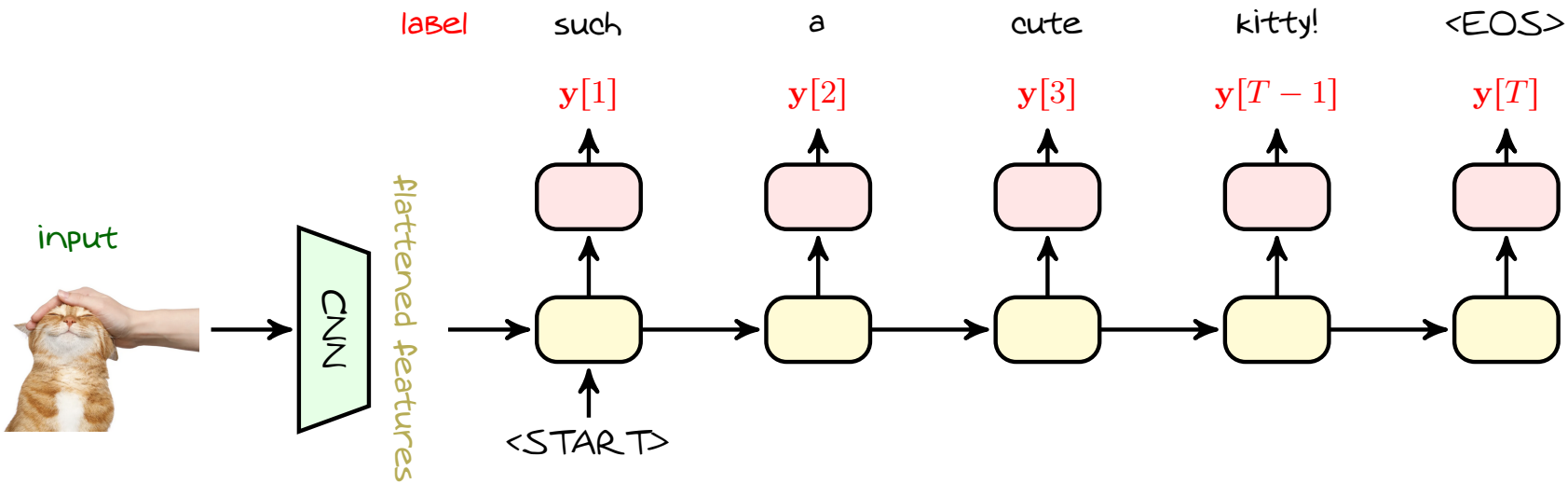
Similarly the output of RNN in each time can be seen as

$$y_k[t] \propto \Pr\{\text{label} = k | \mathbf{h}[t-1], \mathbf{x}[t]\} = p(\mathbf{v}[t] | \mathbf{h}[t-1], \mathbf{x}[t])$$

Since $\mathbf{h}[t-1] \propto \mathbf{x}[1:t-1]$, we could say

$$y_k[t] \propto p(\mathbf{v}[t] | \mathbf{x}[1:t])$$

Caption Generation as Conditional Sampling



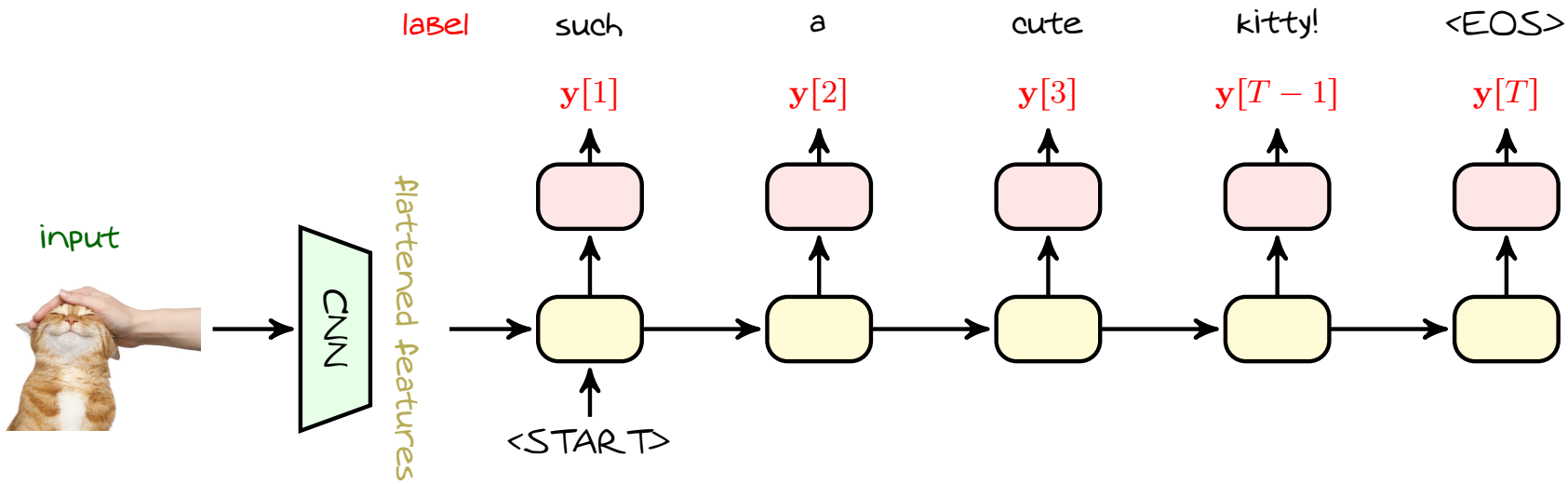
In caption generation, the first generated token can be seen as

$$\mathbf{y}[1] \equiv \Pr\{\mathbf{v}[1]|\mathbf{h}[0], \mathbf{x}[1]\} = p\left(\texttt{token} = \#|\texttt{cat}, \texttt{<start>}\right)$$

and we progress through time

$$\mathbf{y}[t] \equiv = p\left(\texttt{token} = \#|\texttt{cat}, \mathbf{x}[1:t]\right)$$

Caption Generation: *Training*



Note that we train the model end-to-end: *for each pair of image and caption*

- *We compute the loss between output and true caption*
  - ↳ *We can use the cross-entropy function*
- *We backpropagate through time till we arrive at the beginning of decoder*
- *We get* $\nabla_{\text{features}} \hat{R}$ *and keep backpropagateing through encoder*
- *We update all weights and go for the next round*



UNIVERSITY OF
TORONTO

Machine Translation and Transformers

Neural Machine Translation

Design Problem

We want to build a neural network that translates German to English

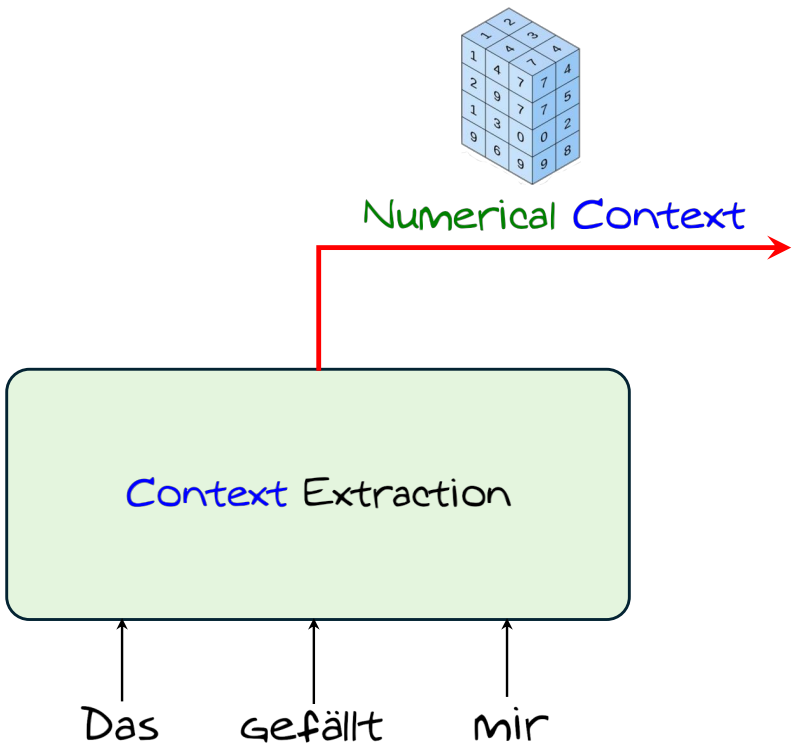


Key Observations

- *An intelligent translator*
  1. *comprehends the whole German sentence*
  2. *gives the corresponding English sentence*
- *Each English word attends to a couple of German words*

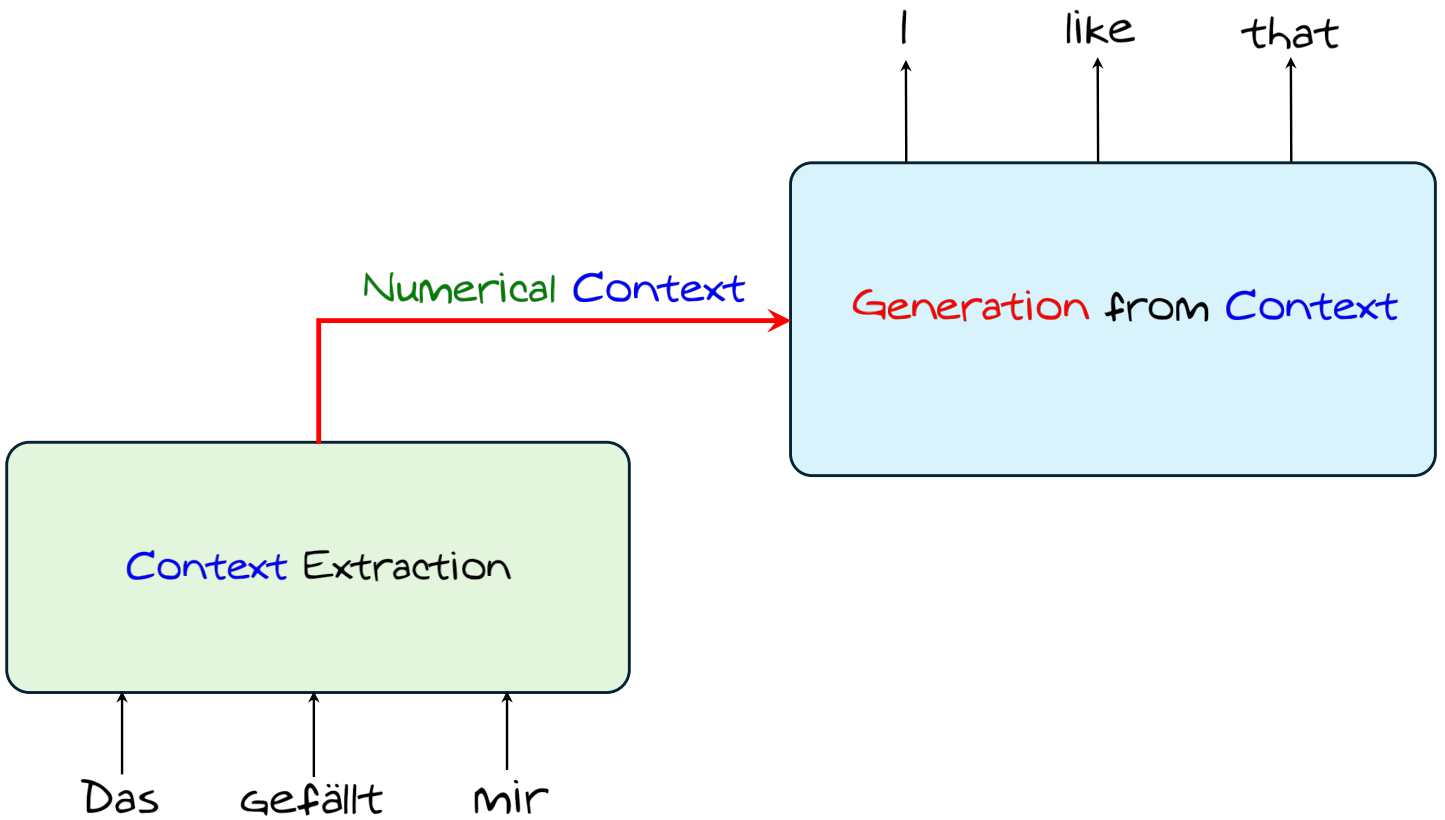
Deep Learning Solution: *Seq2Seq Model*

1. Build a model that extracts context as numerical features



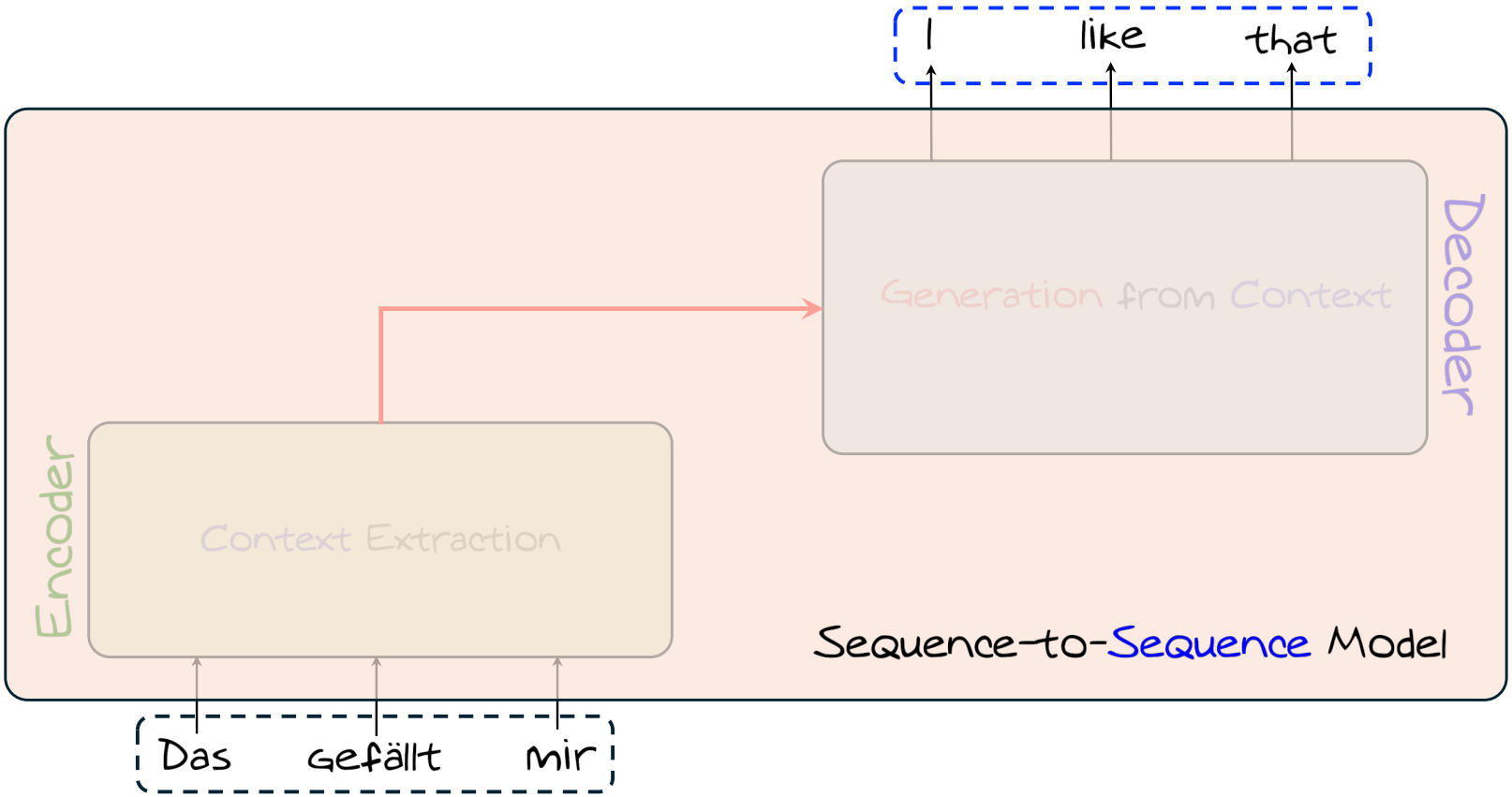
Deep Learning Solution: *Seq2Seq Model*

1. Build a model that extracts context as numerical features
2. Build a model that generates text from numerical context

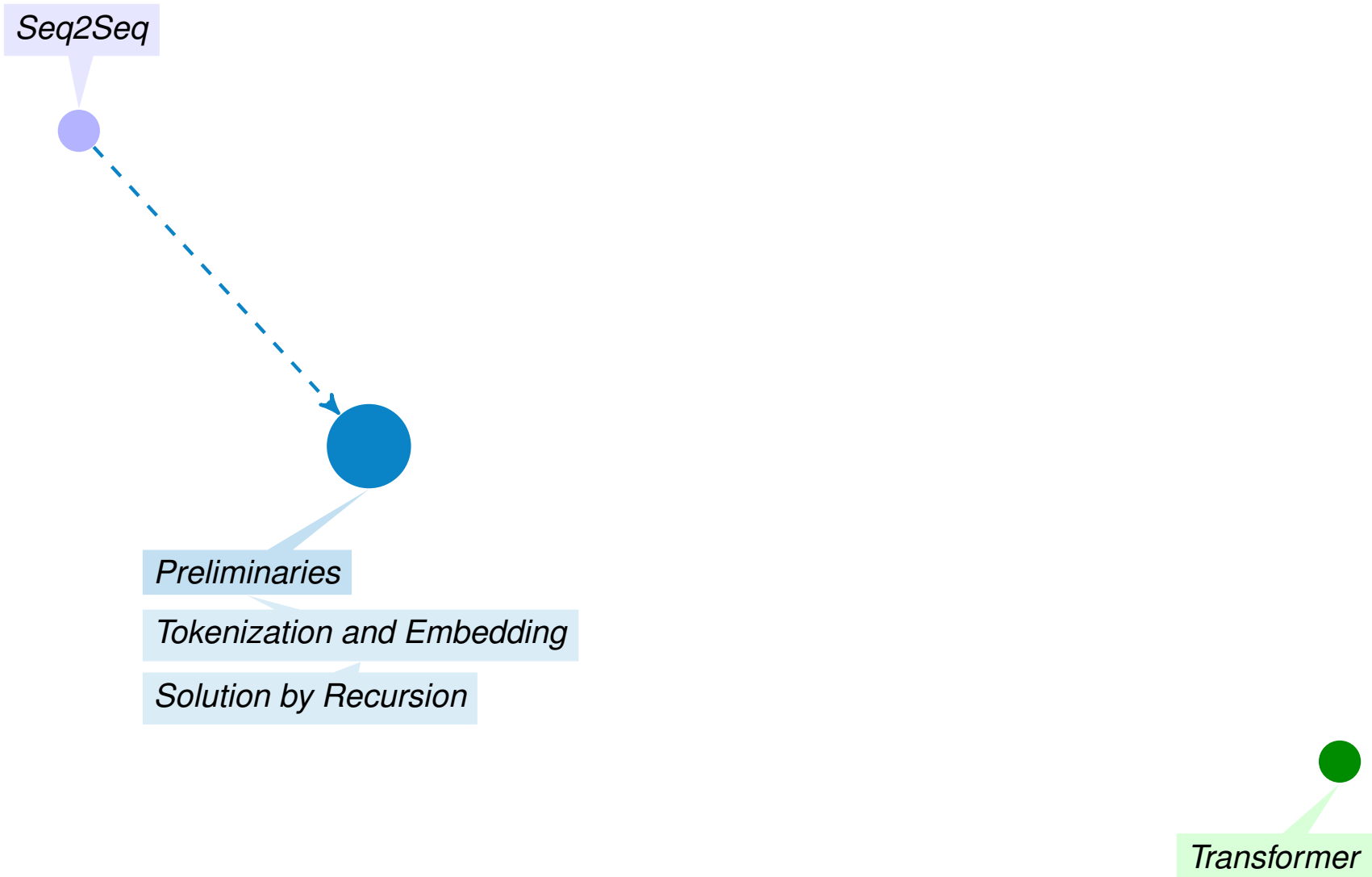


Deep Learning Solution: *Seq2Seq Model*

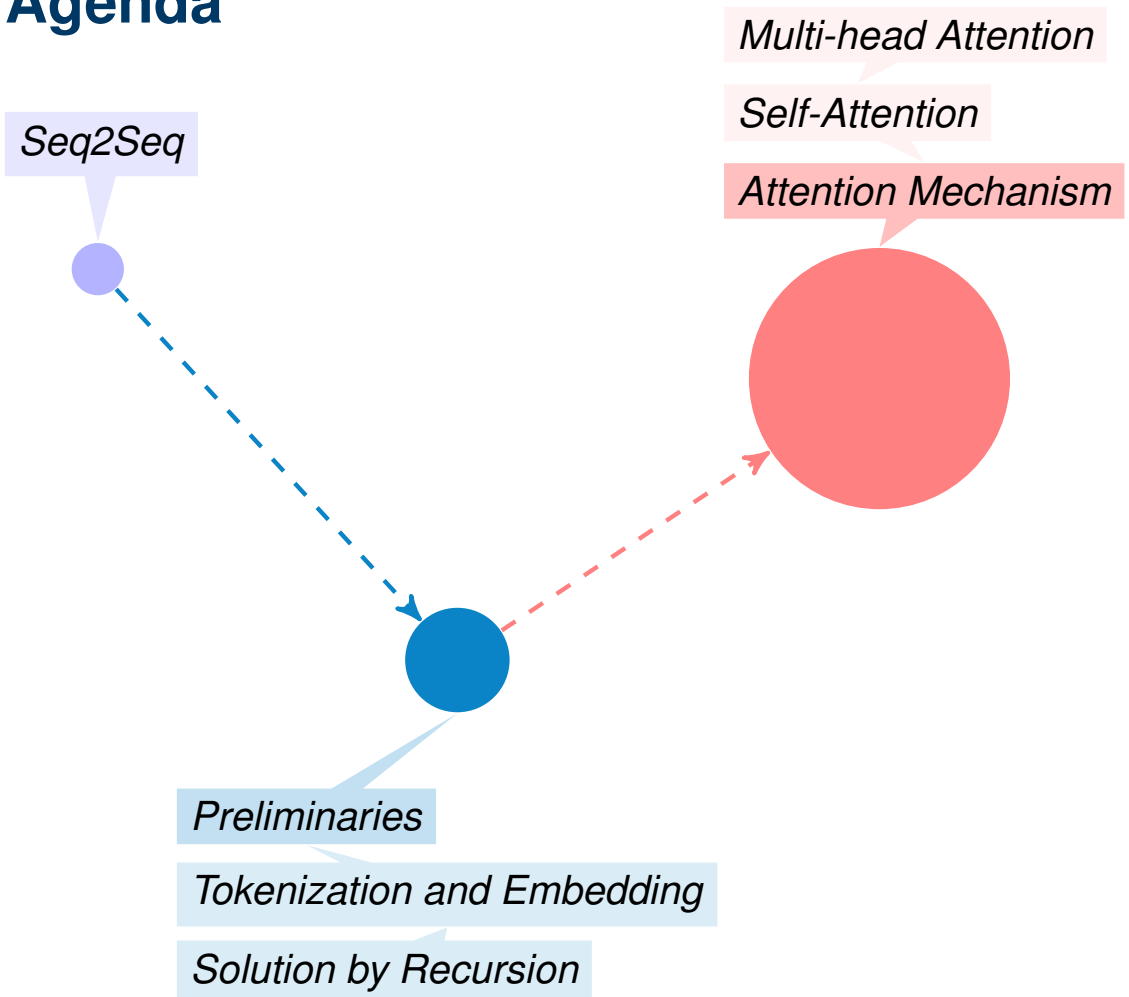
1. Build a model that extracts context as numerical features ⇝ *Encoder*
2. Build a model that generates text from numerical context ⇝ *Decoder*



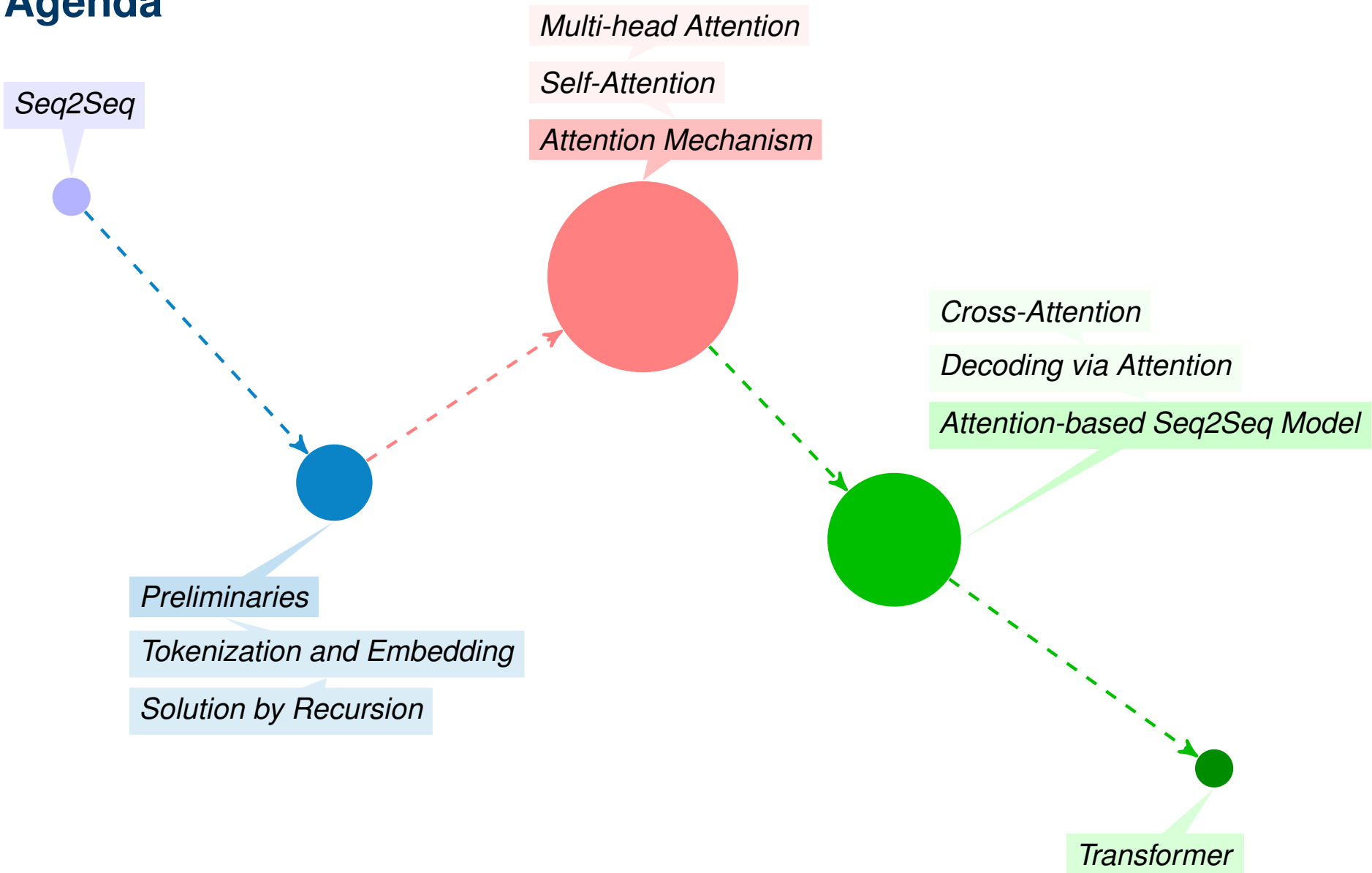
Agenda


Seq2Seq
Preliminaries
Tokenization and Embedding
Solution by Recursion
Transformer

Agenda


Multi-head Attention
Self-Attention
Attention Mechanism
Seq2Seq
Preliminaries
Tokenization and Embedding
Solution by Recursion
Transformer

Agenda


Seq2Seq
Multi-head Attention
Self-Attention
Attention Mechanism
Cross-Attention
Decoding via Attention
Attention-based Seq2Seq Model
Preliminaries
Tokenization and Embedding
Solution by Recursion
Transformer

Back to NMT: *Tokenization*

Data

We make a corpus of German-English text by collecting at least millions of pairs

? *Should we still represent each word by a one-hot vector?*

Step I: *Tokenization*

Represent sentences as sequences of small pieces called token, e.g., sub-words

Vocabulary

token	*index*
`the`	`0`
`ing`	`1`
`interest`	`2`
`<space>`	`3`
⋮	⋮
`;`	$n-1$

`the interesting ...` ⇝ `0, 3, 2, 1, ...`

Back to NMT: *Embedding*

Step II: *Embedding*

Assign each token in the vocabulary a fixed-size vector representation

This is like building a lookup table: *let* $\mathbf{E} \in \mathbb{R}^{E \times n}$ *be the embedding matrix*

Vocabulary

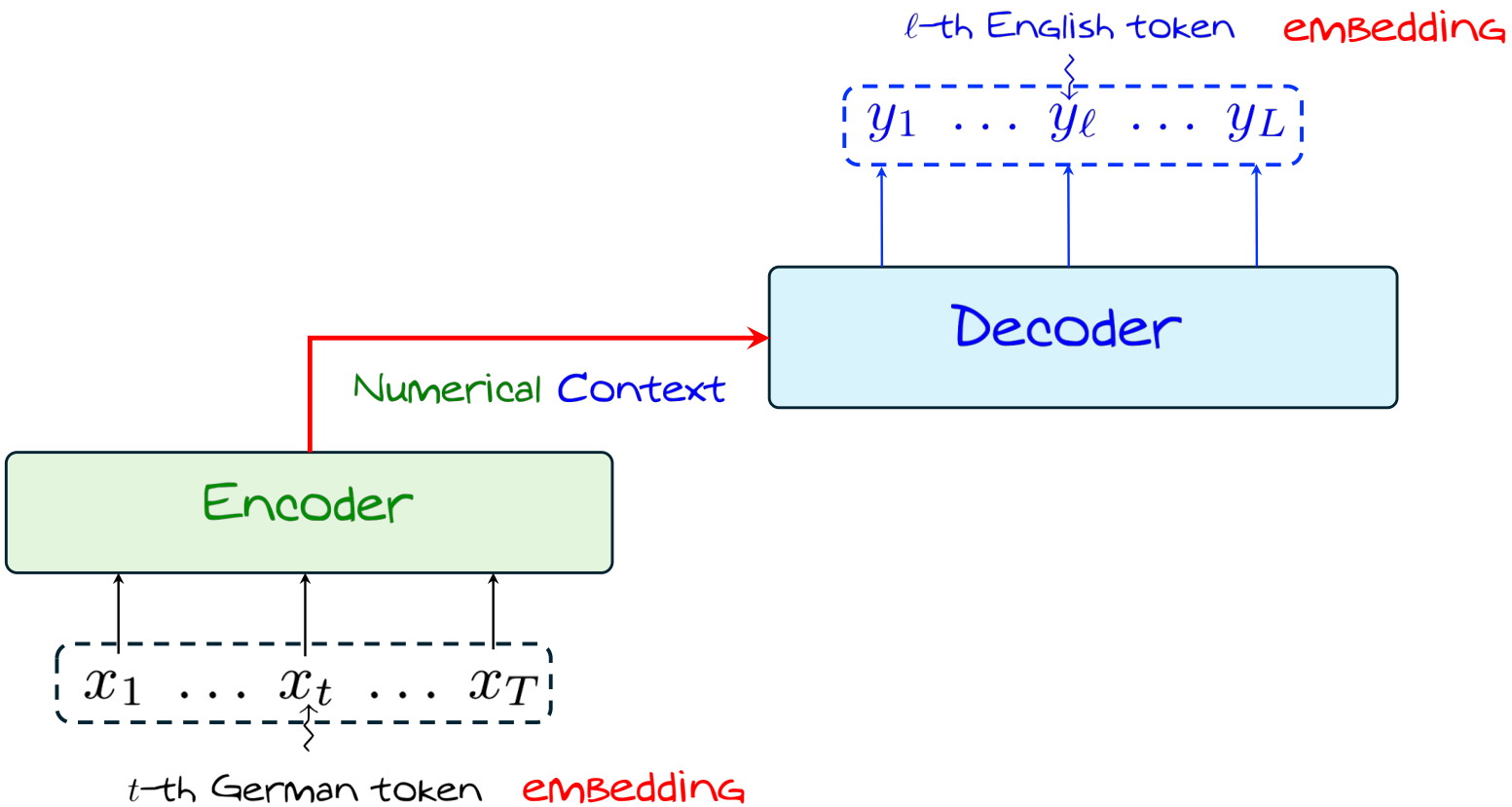
token	*index*
the	0
ing	1
interest	2
<space>	3
⋮	⋮
;	$n-1$

$$\mathbf{E} = \overset{E}{\updownarrow} \begin{bmatrix} \texttt{embd(the)} & \texttt{embd(ing)} & \dots & \texttt{embd(;)} \end{bmatrix}$$

$\texttt{the interesting ...} \rightsquigarrow \texttt{col}_0(\mathbf{E}), \texttt{col}_3(\mathbf{E}), \texttt{col}_2(\mathbf{E}), \texttt{col}_1(\mathbf{E}), \ldots$

Seq2Seq Model Revisited

- A sample sentence pair looks like $(x_1, \ldots, x_T \leftrightsquigarrow y_1, \ldots, y_L) \updownarrow^E$
- Encoder encodes $x_{1:T}$ to features and decoder decodes them to $y_{1:L}$

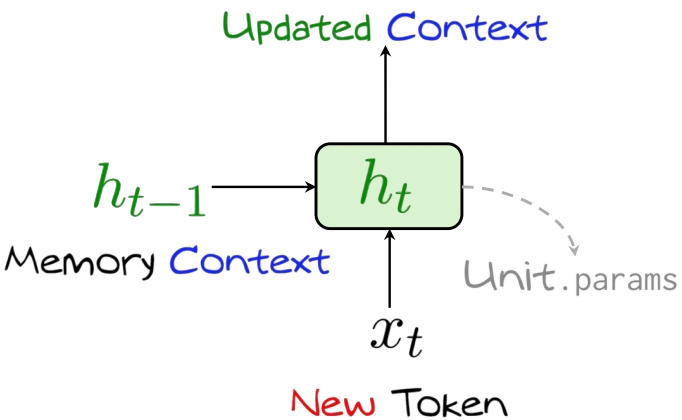


Old Design: *Recursion*

We can build the model by recursion

Say h_{t-1} is the context extracted from $x_{1:t-1}$. *A recurrent unit*

1. *takes memory context* h_{t-1} *and new token* x_t *as input*
2. *computes an updated context* h_t *that captures the context of* $x_{1:t}$



Recurrent Unit

The unit can be a shallow MLP

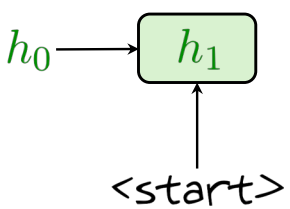
$$h_t = f\left(\mathbf{W}_{\mathsf{M}} h_{t-1} + \mathbf{W}_{\mathsf{in}} x_t\right)$$

or more complicated, e.g. LSTM or GRU

Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

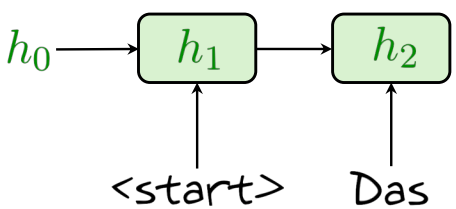
Das gefeällt mir ⇝ <start> Das gefeällt mir <end>



Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

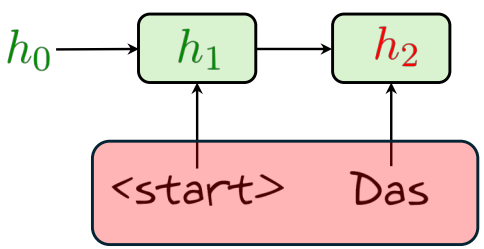
Das gefeällt mir ⇝ <start> Das gefeällt mir <end>



Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

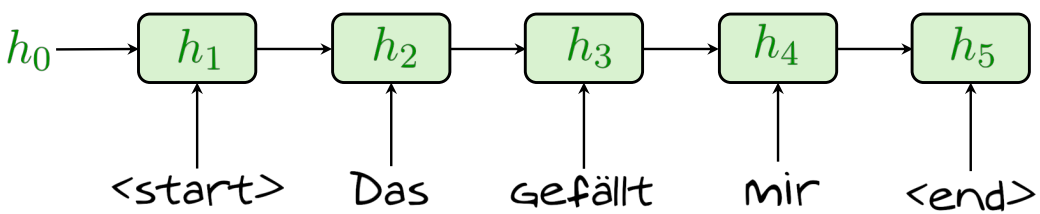
Das gefeällt mir ⇝ <start> Das gefeällt mir <end>



Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

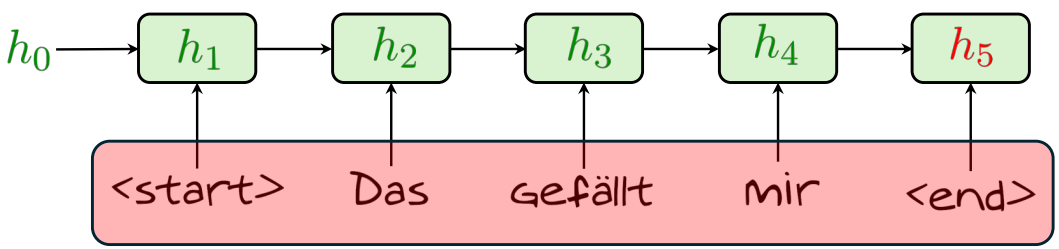
Das Gefällt mir ⇝ <start> Das Gefällt mir <end>



Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

Das Gefeällt mir $\rightsquigarrow$ <start> Das Gefeällt mir <end>



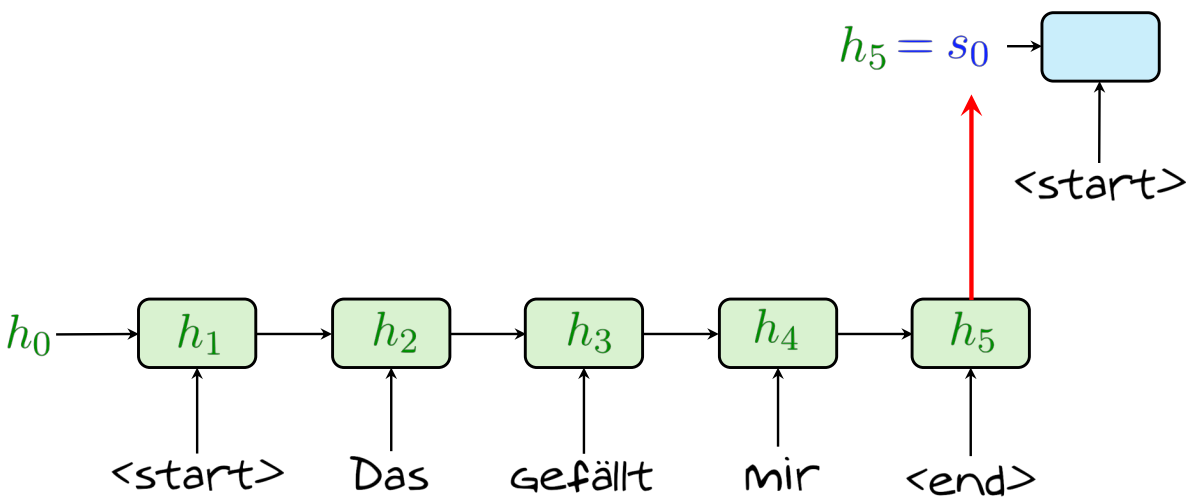
Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

  Das gefeällt mir ⇝ <start> Das gefeällt mir <end>

- *Decode sequentially by generating one token at a time*

  <start> I like that <end> ⇝ I like that



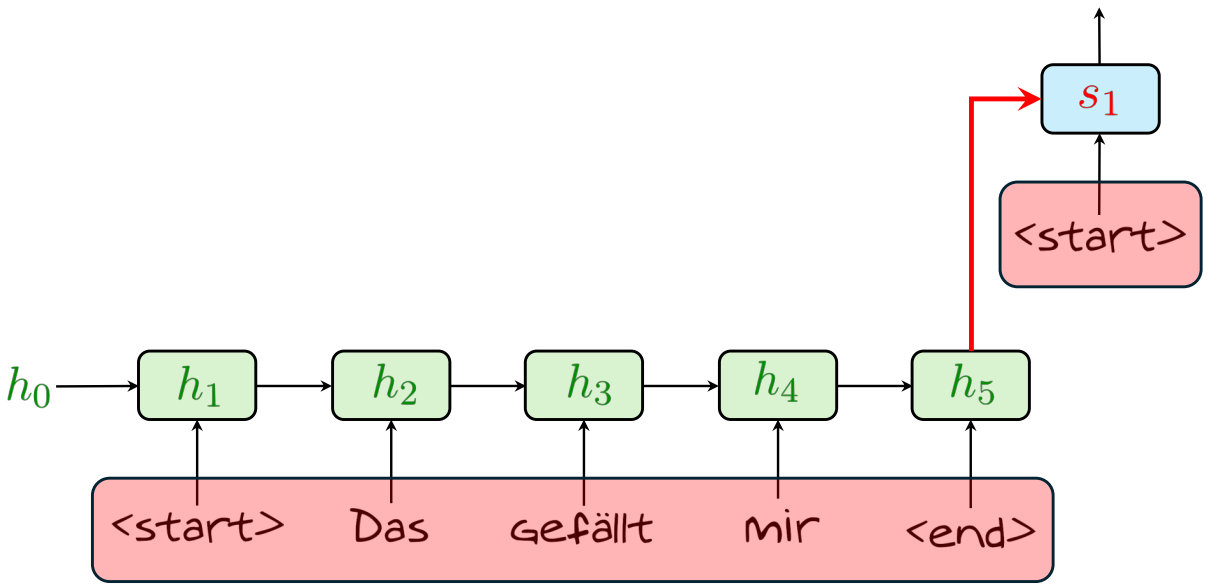
Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

  Das gefeällt mir ⇝ <start> Das gefeällt mir <end>

- *Decode sequentially by generating one token at a time*

  <start> I like that <end> ⇝ I like that



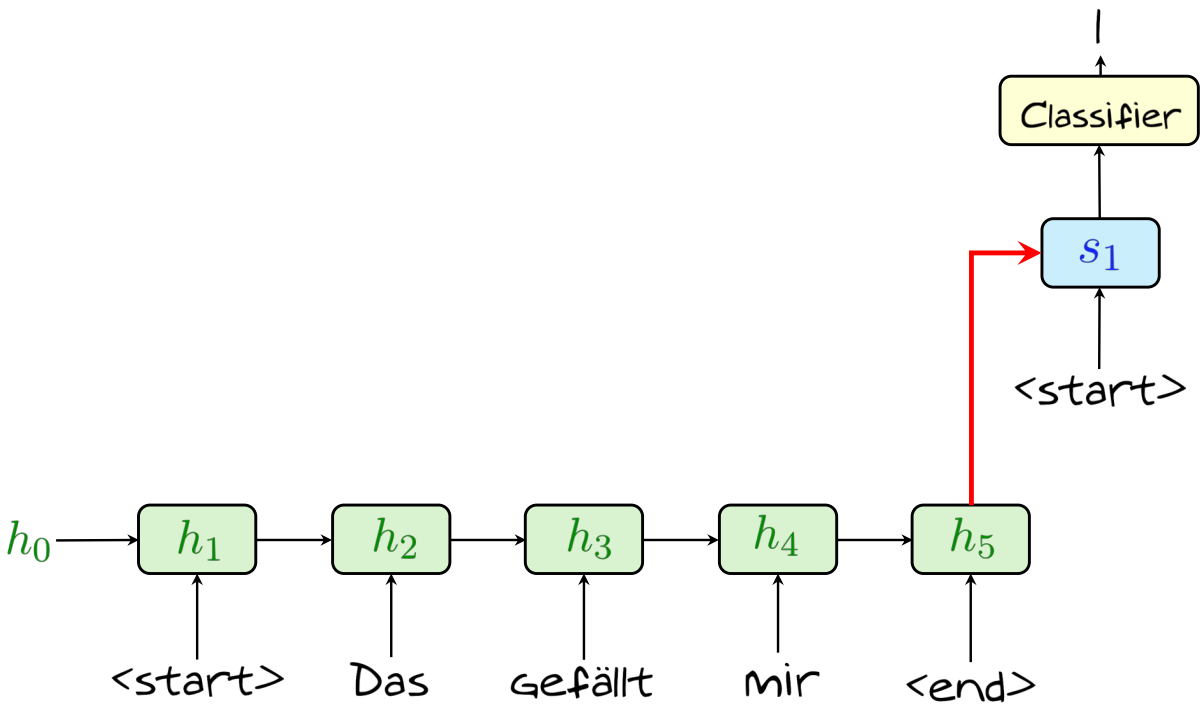
Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

  Das Gefeällt mir ⇝ <start> Das Gefeällt mir <end>

- *Decode sequentially by generating one token at a time*

  <start> I like that <end> ⇝ I like that



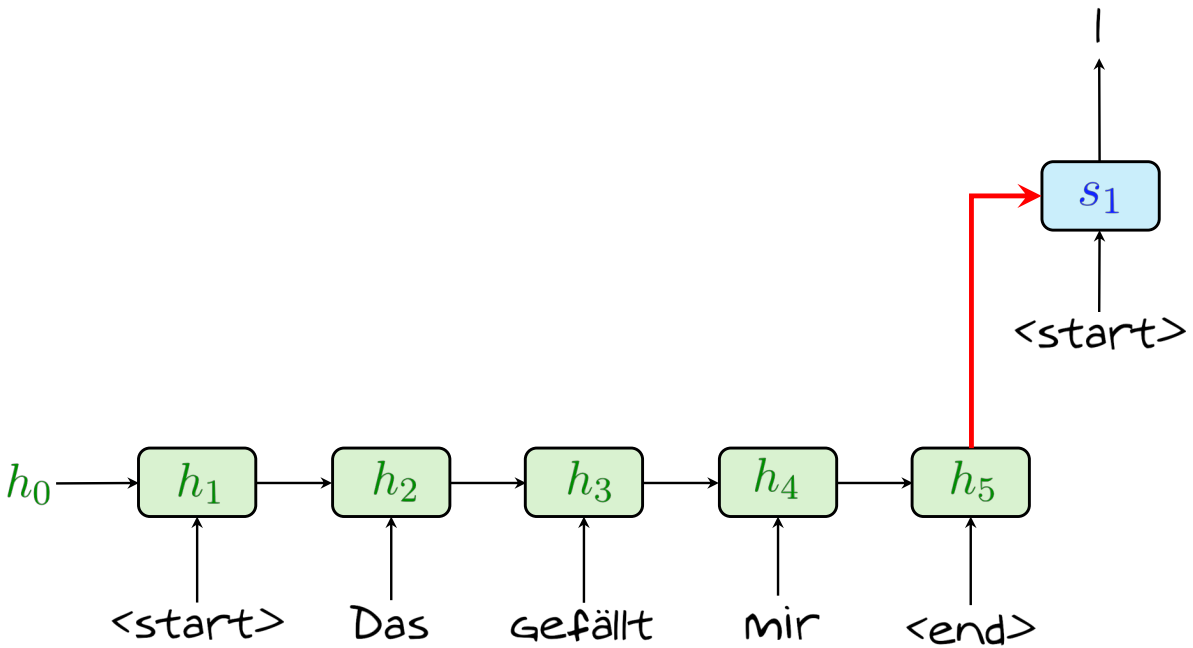
Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

  Das Gefeällt mir ⇝ <start> Das Gefeällt mir <end>

- *Decode sequentially by generating one token at a time*

  <start> I like that <end> ⇝ I like that



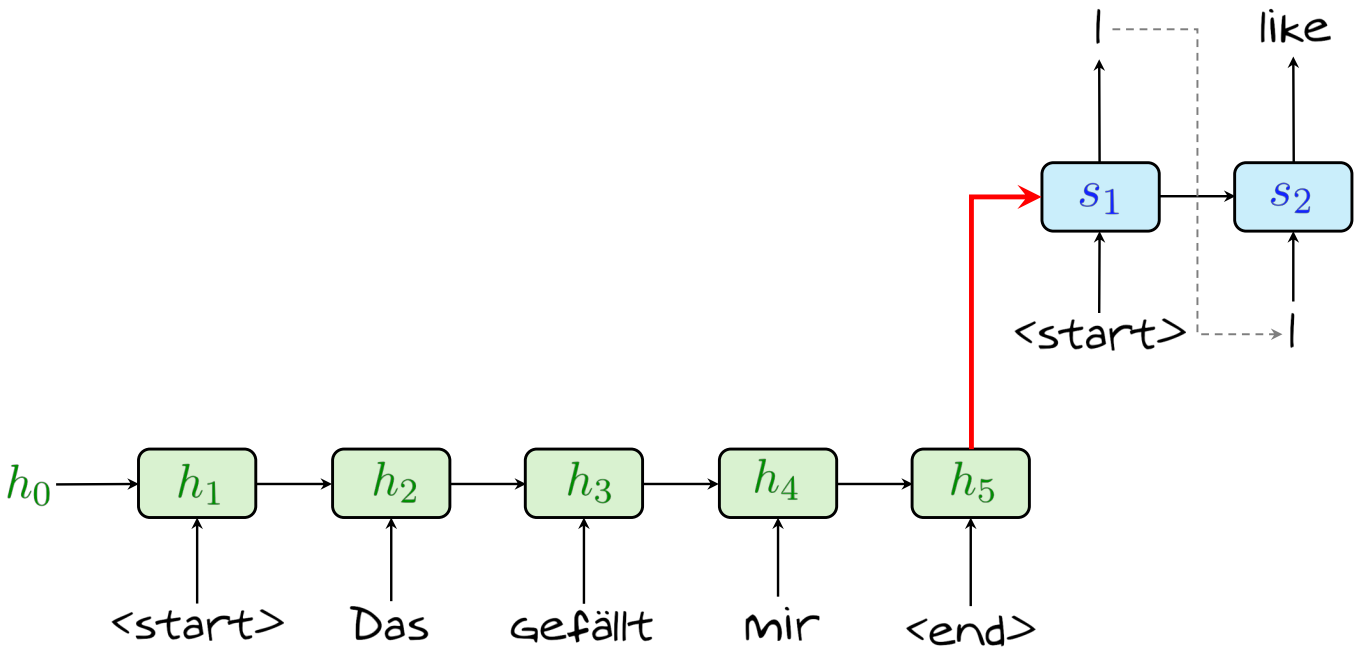
Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

  Das Gefeällt mir ⇝ <start> Das Gefeällt mir <end>

- *Decode sequentially by generating one token at a time*

  <start> I like that <end> ⇝ I like that



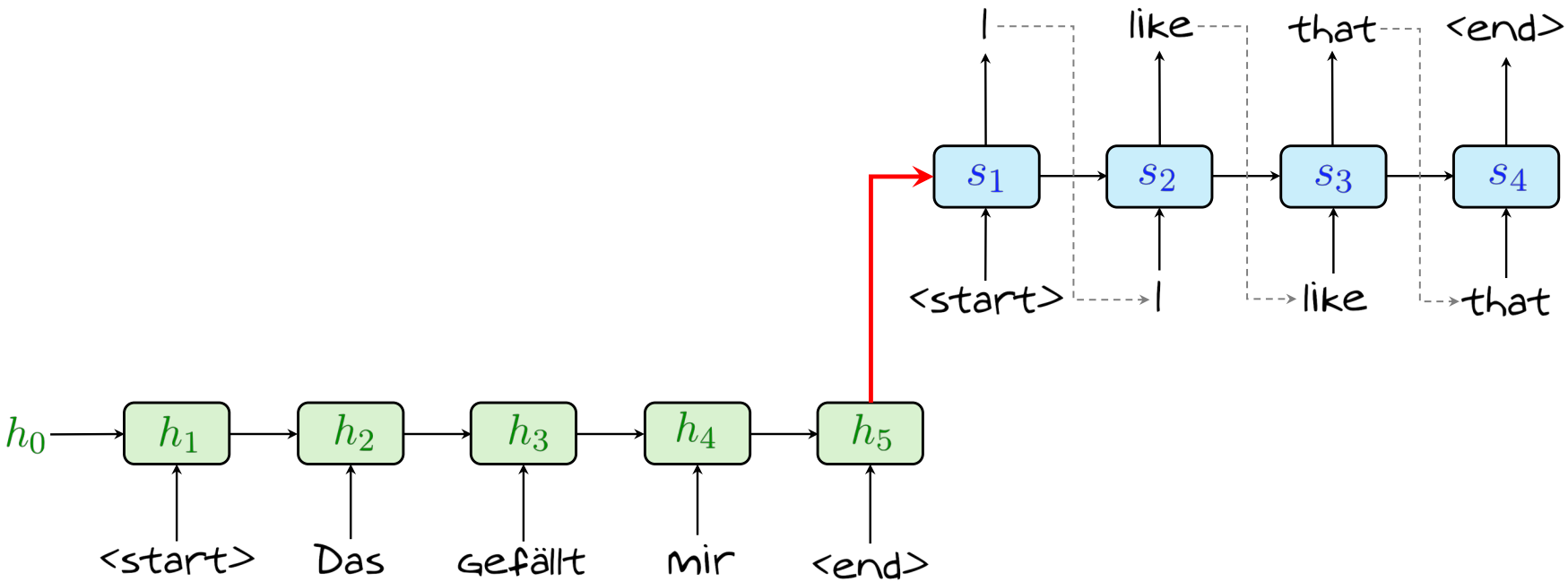
Seq2Seq Model by *Recursion*

- *Encode sequentially by processing one token at a time*

  Das Gefeällt mir ⇝ <start> Das Gefeällt mir <end>

- *Decode sequentially by generating one token at a time*

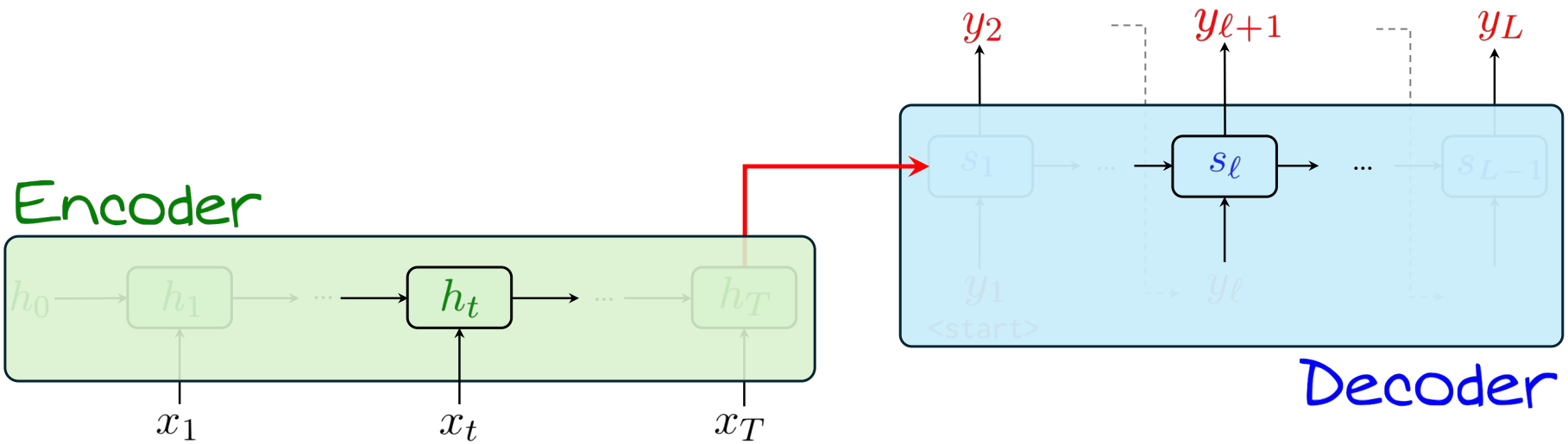
  <start> I like that <end> ⇝ I like that



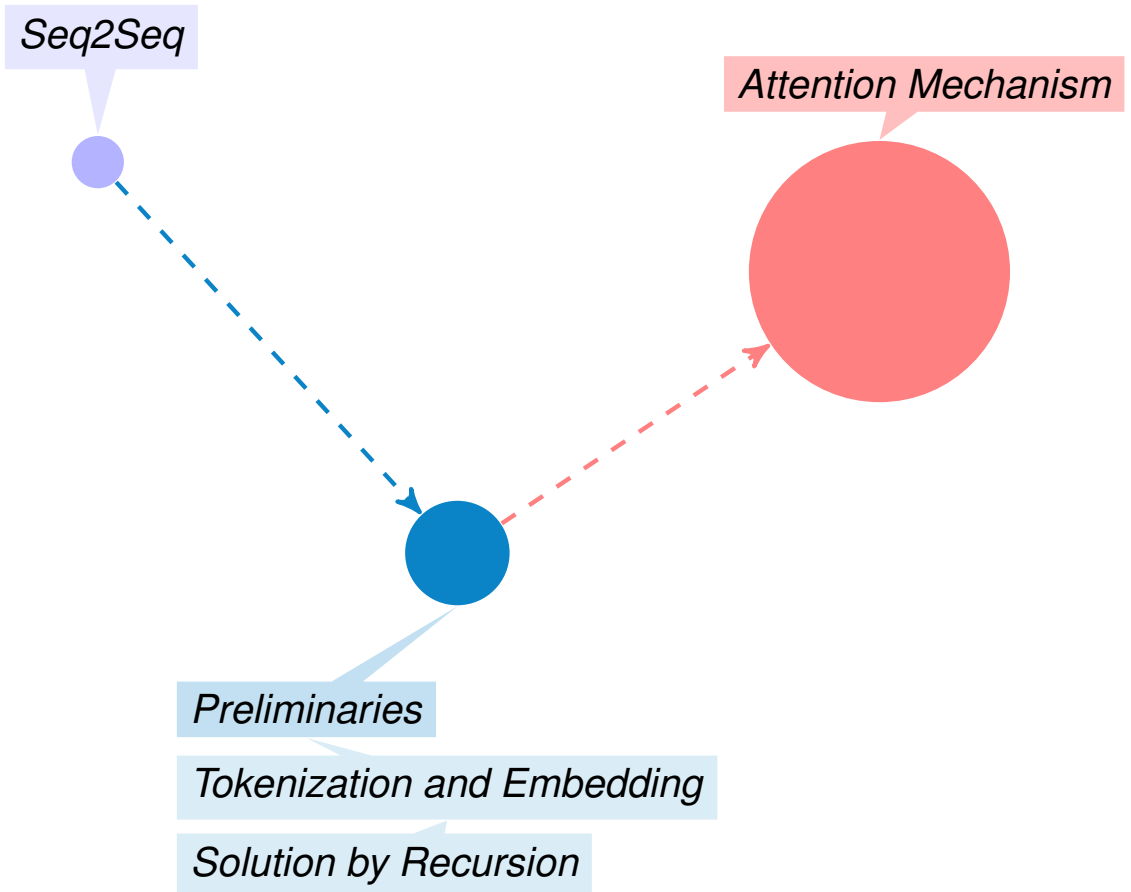
Recurrent NMT Model

Rec_NMT($x_{1:T}$):

1: Initialize encoder with some h_0
2: **for** $t = 1 : T$ **do** Update context as $h_t = \texttt{RUnit}_{\text{enc}}(h_{t-1}, x_t)$ **end for**
3: Initialize decoder as $s_0 = h_T$ and first token $y_1 \equiv$ <start>
4: **for** $\ell = 1 : L - 1$ **do**
5: Update context as $s_\ell = \texttt{RUnit}_{\text{dec}}(s_{\ell-1}, y_\ell)$
6: Compute next token via a classifier as $y_{\ell+1} = \texttt{Classifier}(s_\ell)$
7: **end for**



Agenda for Today



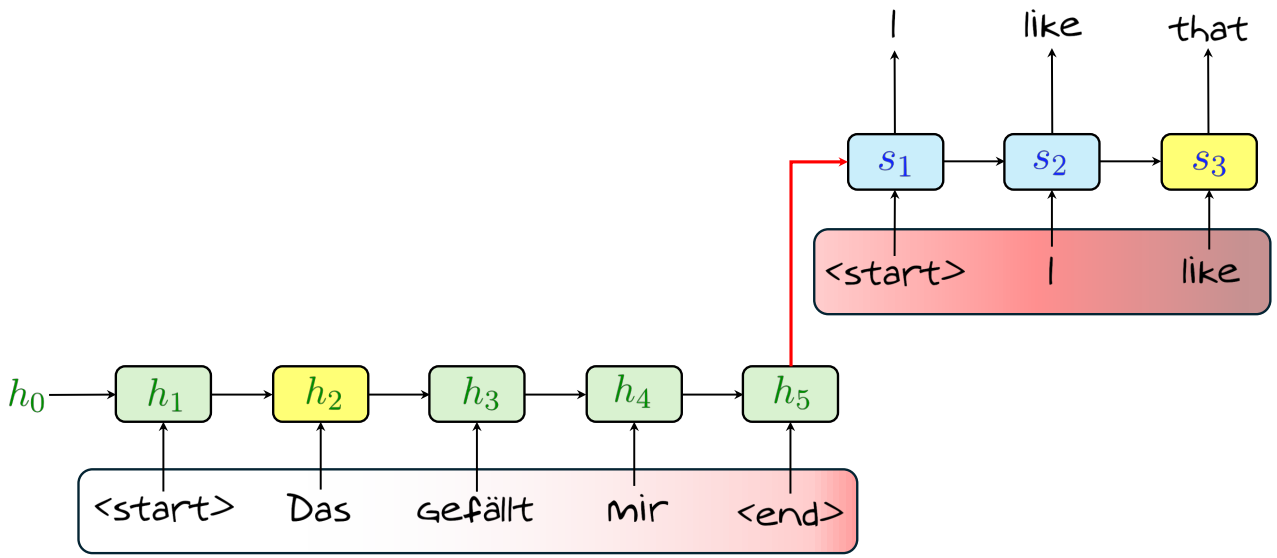
Recursion Drawback I: *Information Bottleneck*

Information Bottleneck

Encoded context is passed through a bottleneck

- *Each time context is updated older information is overwritten*
- *This limits long-term dependency*
- *As input length increases decoding becomes more unreliable*

At $\ell = 3$, decoder observes limited effect from the context encoded at $t = 2$



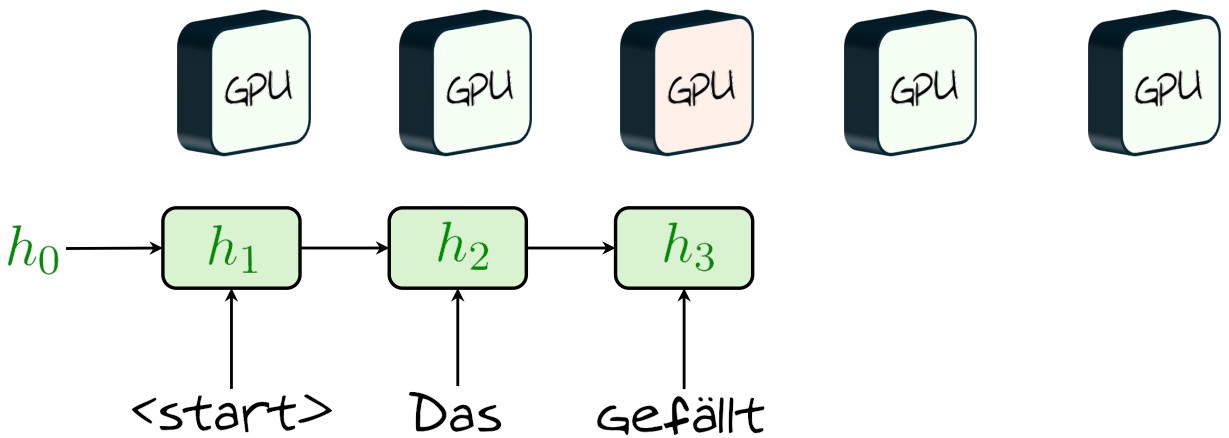
Recursion Drawback II: *Sequential Computation*

Sequential Computation

Encoding and *decoding* *cannot* be parallelized *across time*

- *To encode or decode a new token, the latest context is needed*
- ! *Processing time scales with text length regardless of computation capacity*

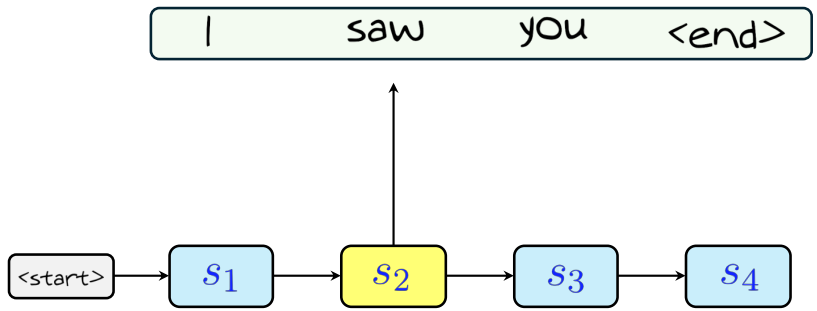
To compute h_3, we need to have h_2 already computed

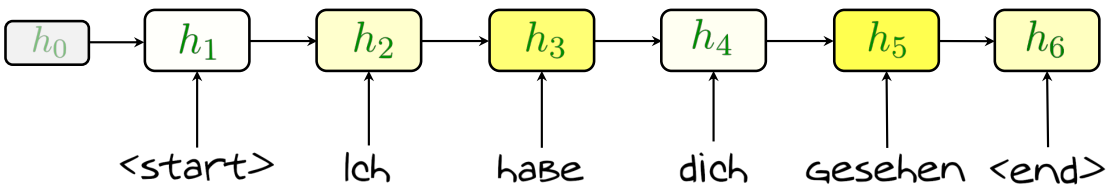


Removing Bottleneck via *Attention*

Thought Experiment. *What if we build a block that finds related encoded tokens?*

- ✔ *It can tell decoder to which tokens it should attend*

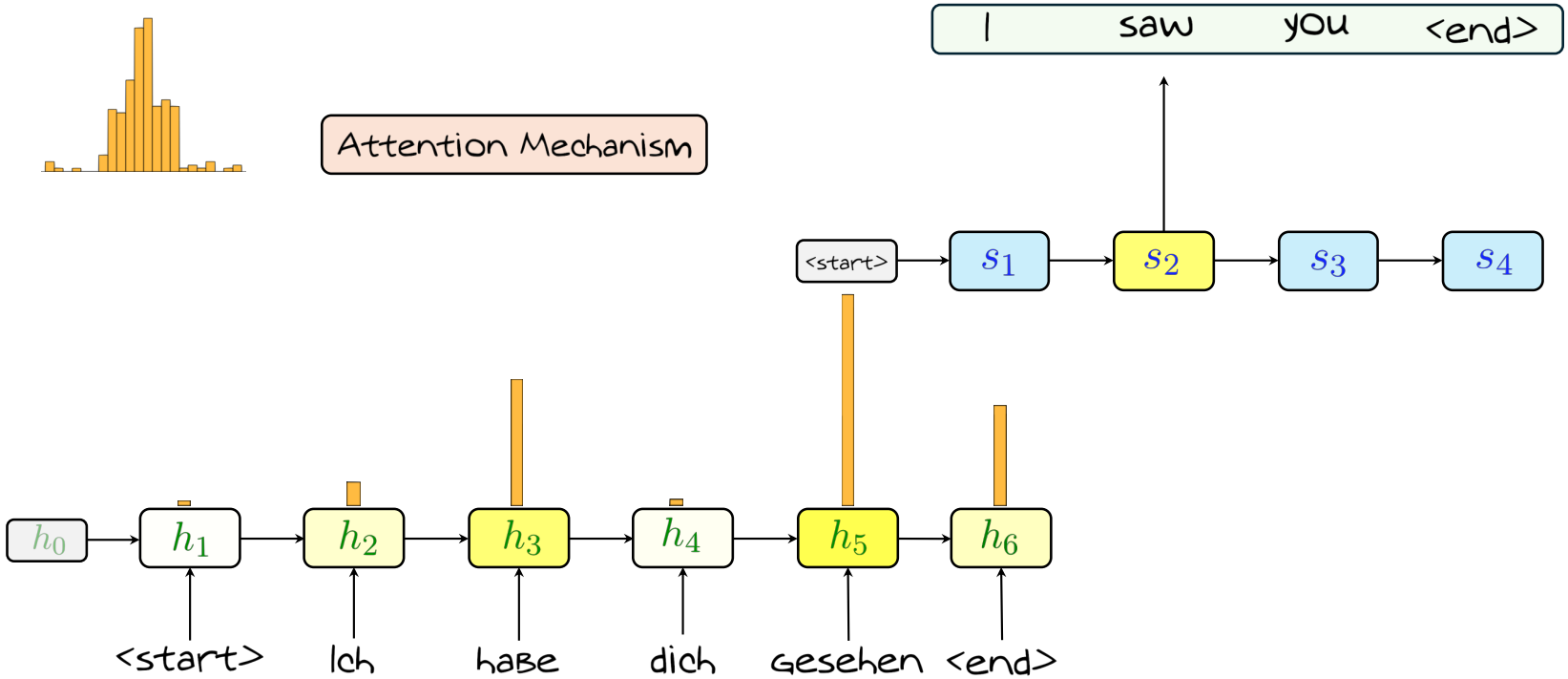




Removing Bottleneck via *Attention*

Thought Experiment. *What if we build a block that finds related encoded tokens?*

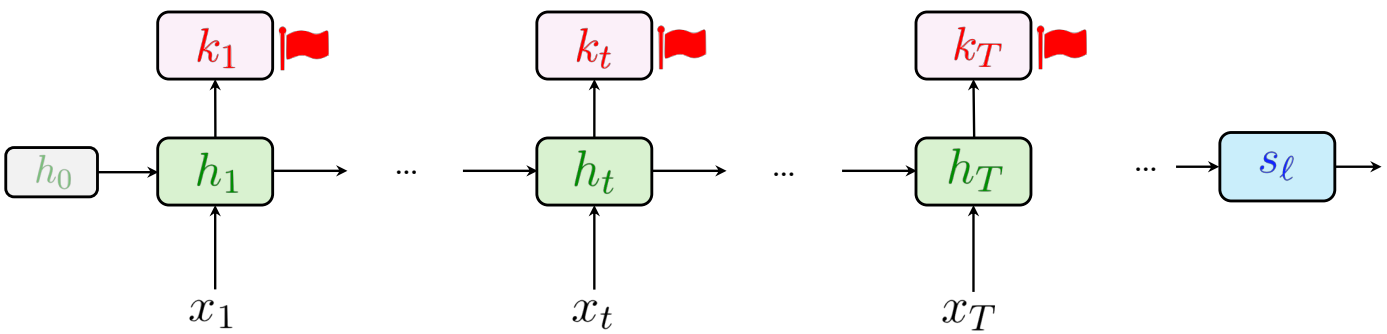
- ✔ *It can tell decoder to which tokens it should attend*
- 💡 *The block can return a distribution over encoded tokens*
  - ↳ *Entries of this distribution can be treated as attention coefficients*



Attention Mechanism: *Computational Model*

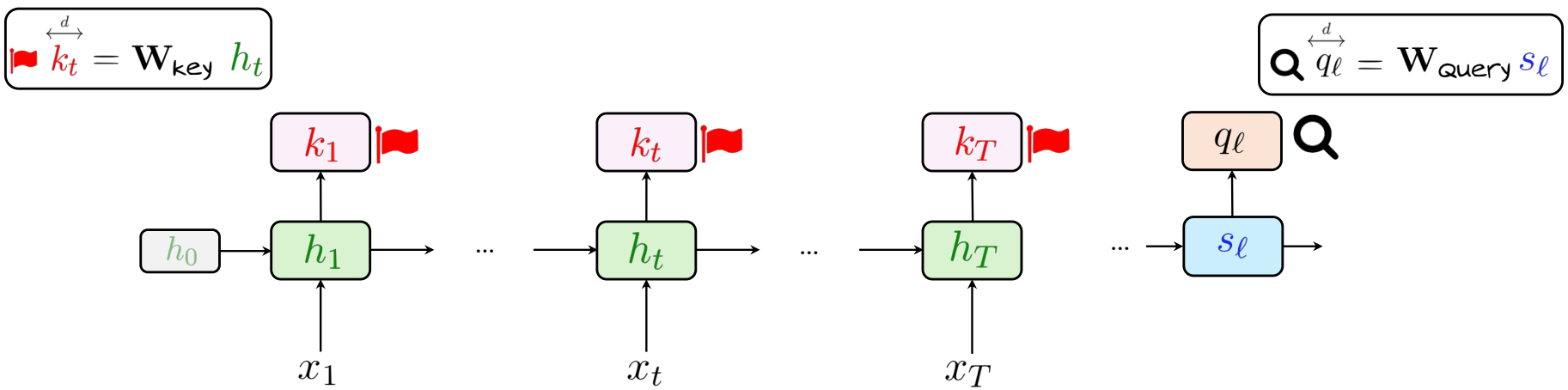
1. *Generate a key vector for each encoded token as a representing flag*

$$\overset{\longleftrightarrow}{k_t}^{d} = \mathbf{W}_{\text{key}}\ h_t$$



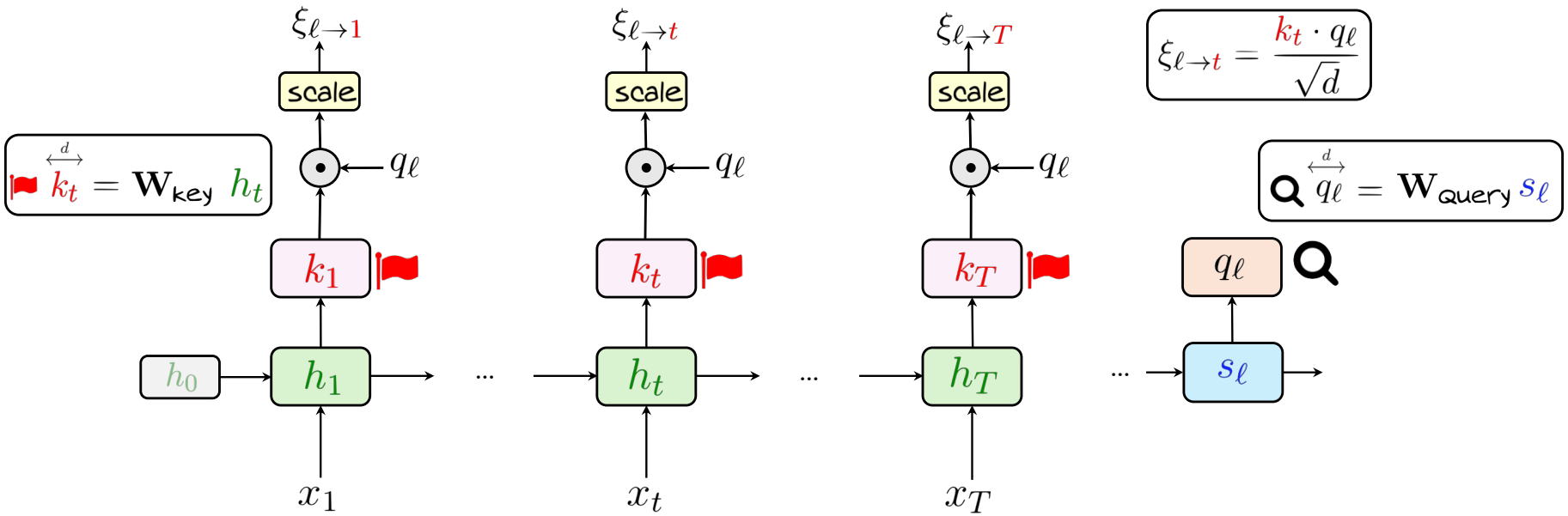
Attention Mechanism: *Computational Model*

1. *Generate a key vector for each encoded token as a representing flag*
2. *Generate a query vector at the decoder as a search request*



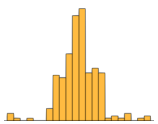
Attention Mechanism: *Computational Model*

1. *Generate a key vector for each encoded token as a representing flag*
2. *Generate a query vector at the decoder as a search request*
3. *Compare the query to all keys via scaled dot-product*



Attention Mechanism: *Computational Model*

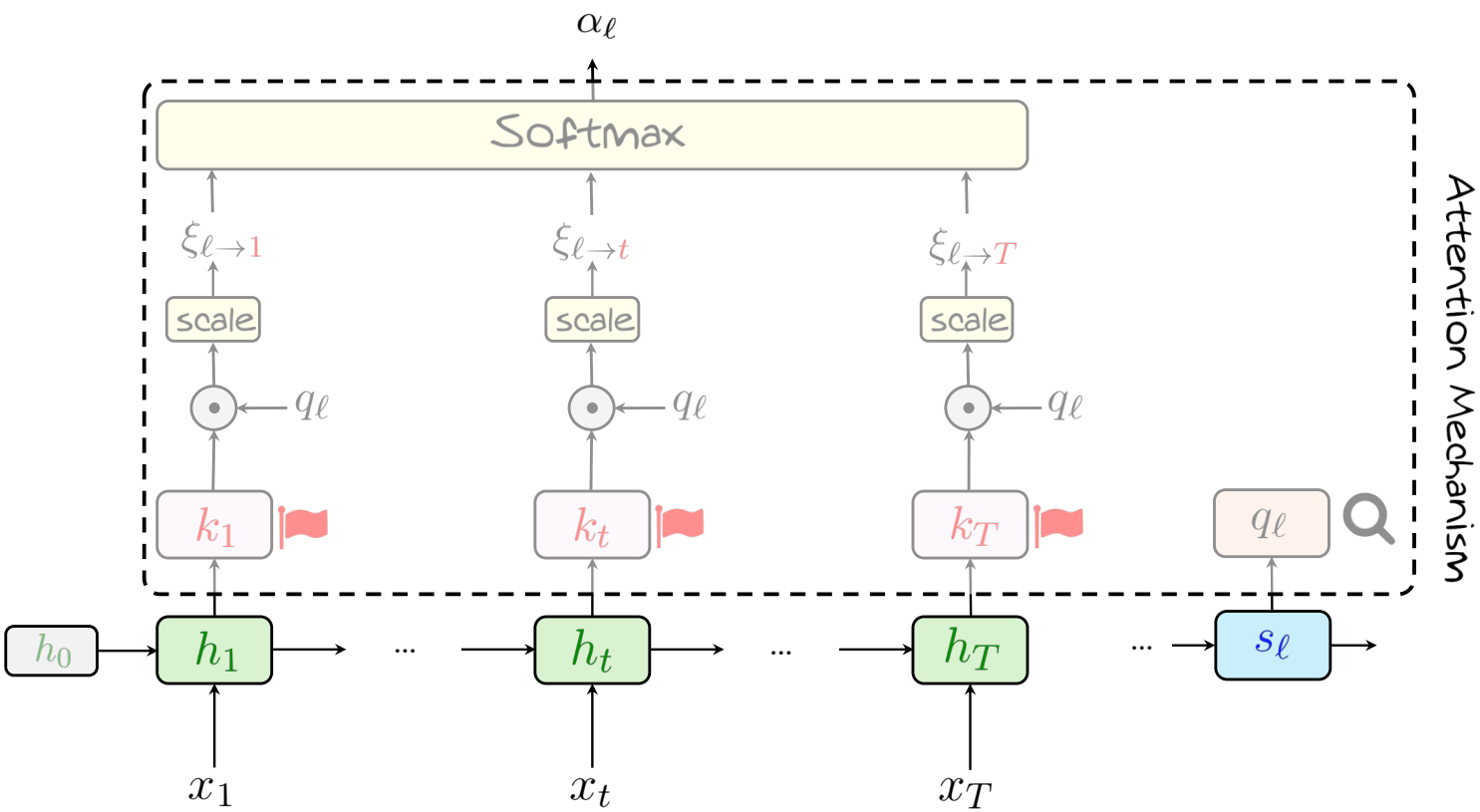
1. *Generate a key vector for each encoded token as a representing flag*
2. *Generate a query vector at the decoder as a search request*
3. *Compare the query to all keys via scaled dot-product*
4. *Compute a distribution using Softmax*



$$\alpha_\ell = \text{Softmax}(\xi_\ell)$$

Attention Mechanism: *Computational Model*

1. *Generate a key vector for each encoded token as a representing flag*
2. *Generate a query vector at the decoder as a search request*
3. *Compare the query to all keys via scaled dot-product*
4. *Compute a distribution using Softmax*



Computational Attention: *Notes*

❓ Why do we compute scaled dot product?

ℹ️ *The value of dot ptoduct can describe the correlation between tokens*

↳ *If the query is close to a key ⇝ their dot product is large*

ℹ️ *The scaling is to avoid high variance in dot product values*

🔍 *Do a quick research to find out why we scale with $\sqrt{d}$*

ℹ️ *Recall that Softmax computes a distribution proportional to raw scores*

$$\alpha_{\ell \to t} = [\text{Softmax}(\xi_\ell)]_t = \frac{e^{\xi_{\ell \to t}}}{\sum_j e^{\xi_{\ell \to j}}}$$

⬆️ *Large dot products lead to large coefficients*

⬇️ *Small dot products lead to small coefficients*

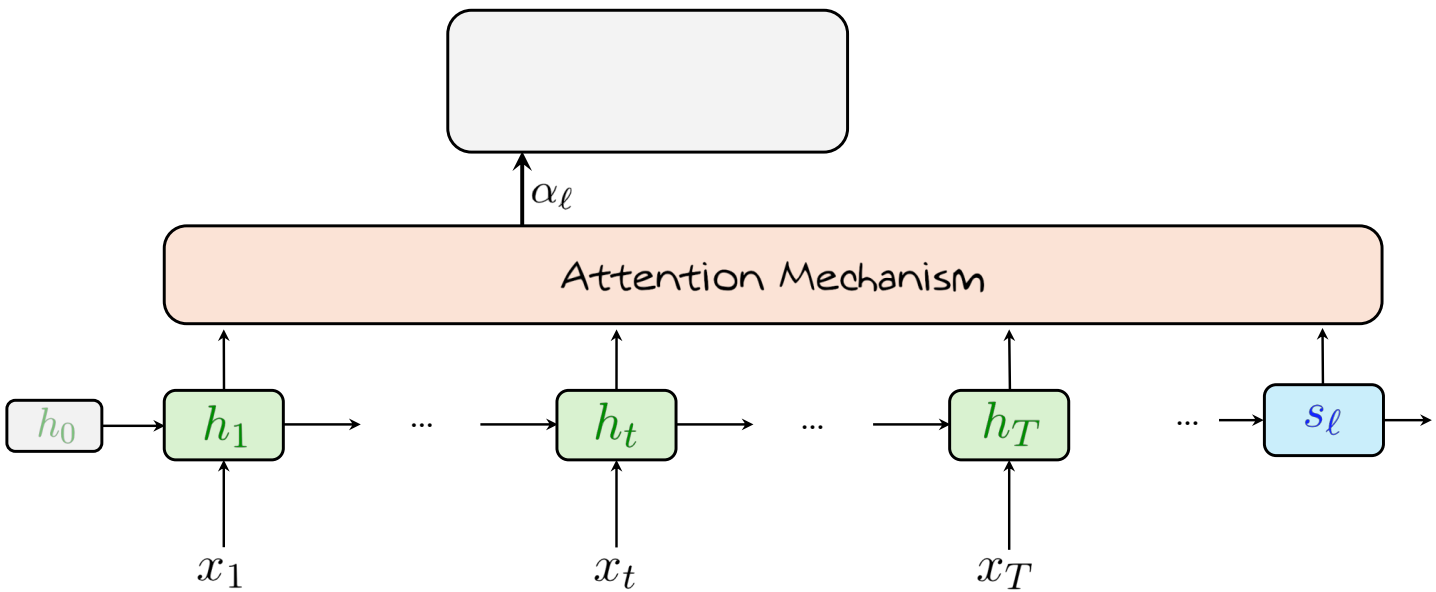
Computational Attention Block

We can now build an attention-based block to compute context at the decoder

Computational Attention Block

We can now build an attention-based block to compute context at the decoder

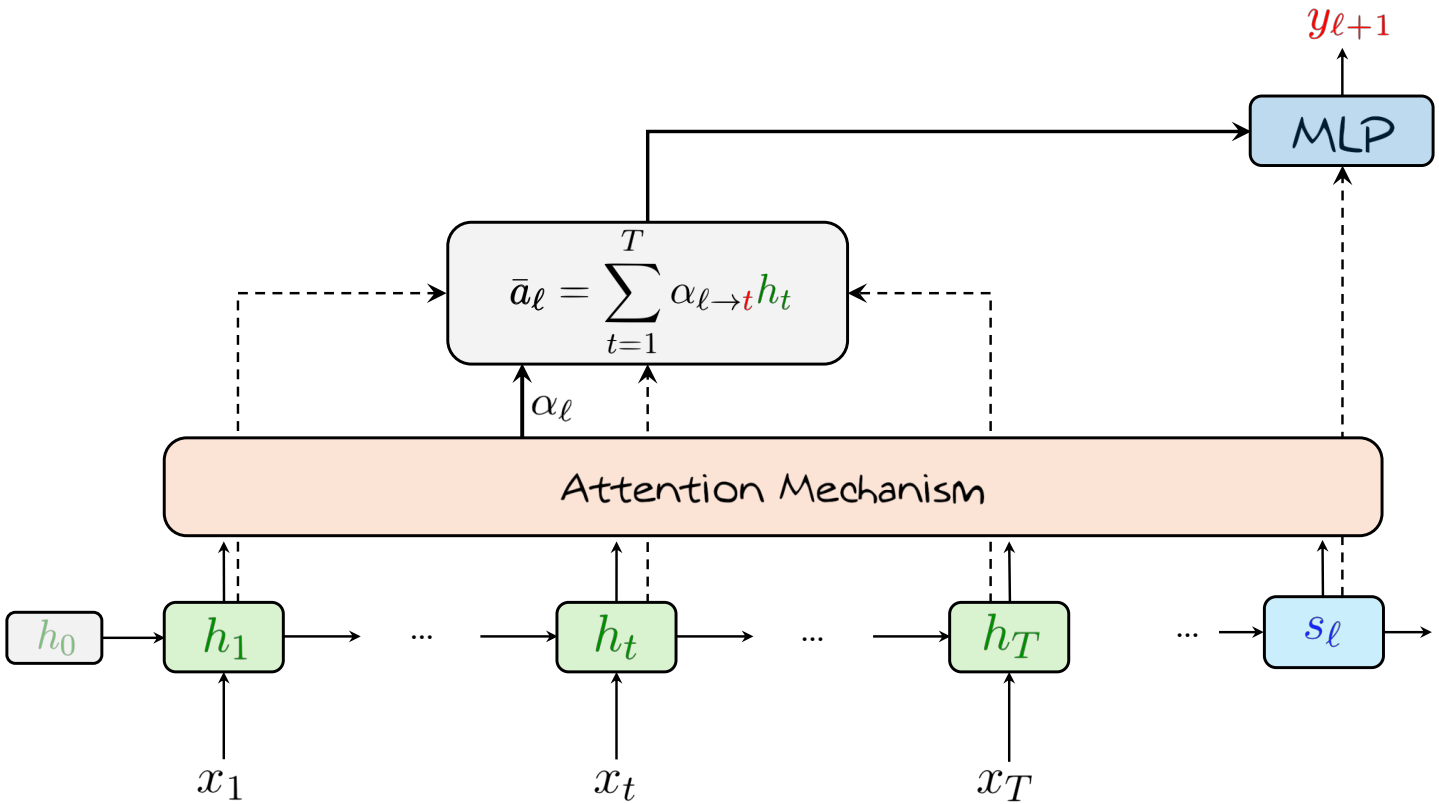
1. *Compute attention coefficients at each decoding time*



Computational Attention Block

We can now build an attention-based block to compute context at the decoder

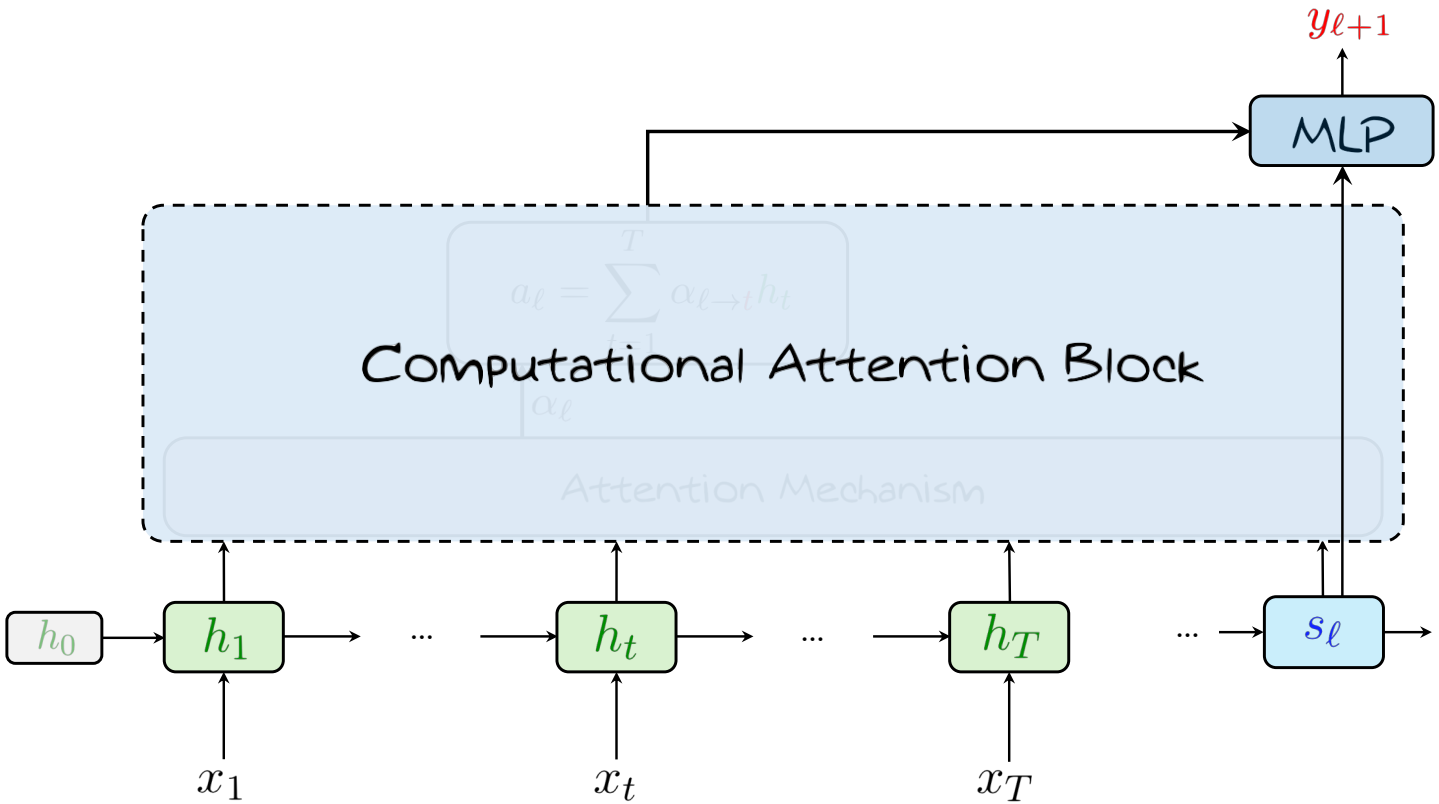
1. *Compute attention coefficients at each decoding time*
2. *Use attention coefficients to compute weighted average of encoded contexts*
   - ↳ *Encoded contexts with larger attention coefficients dominate the average*



Computational Attention Block

We can now build an attention-based block to compute context at the decoder

1. *Compute attention coefficients at each decoding time*
2. *Use attention coefficients to compute weighted average of encoded contexts*
   - ↳ *Encoded contexts with larger attention coefficients dominate the average*



Computational Attention Block: *Pseudo Code*

all vectors are column-vectors

- *Array of keys and the query are computed by matrix multiplication*

$$\mathbf{K} = [k_1 \dots k_T] = \mathbf{W}_{\text{key}} [h_1 \dots h_T] \qquad q_\ell = \mathbf{W}_{\text{query}} s_\ell$$

- *Array of dot products is computed by a matrix multiplication*

$$[q_\ell \cdot k_1 \dots q_\ell \cdot k_T] = \left[k_1^\top q_\ell \dots k_T^\top q_\ell\right] = \mathbf{K}^\top q_\ell$$

- *Weighted sum is computed by a matrix multiplication*

$$\bar{a}_\ell = \sum_t \alpha_{\ell \to t} h_t = [h_1 \dots h_T] \begin{bmatrix} \alpha_{\ell \to 1} \\ \vdots \\ \alpha_{\ell \to T} \end{bmatrix} = [h_1 \dots h_T] \alpha_\ell$$

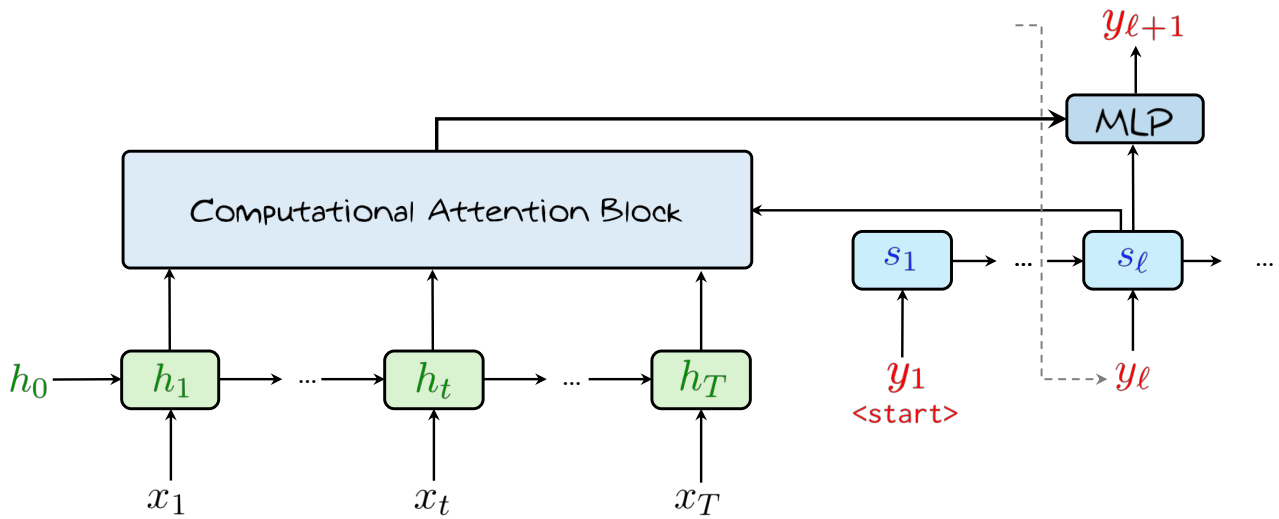
`AttenBlock`($h_{1:T}, s_\ell$):

1: Compute $\mathbf{K} = \mathbf{W}_{\text{key}} [h_1 \dots h_T]$ and $q_\ell = \mathbf{W}_{\text{query}} s_\ell$

2: Compute $\alpha_\ell = \text{softmax}\left(\mathbf{K}^\top q_\ell / \sqrt{d}\right)$

3: Compute $\bar{a}_\ell = [h_1 \dots h_T] \alpha_\ell$

Looking Back at Drawbacks



Recursion Drawbacks

- ✔ Information Bottleneck ⇝ *attention block removes need for bottleneck*
- ✖ Sequential Computation ⇝ *encoder and decoder still process sequentially*

Key Task: *Context Encoding via Attention*

? How can the context be encoded without recursion?

Properties of Encoded Features $h_{1:T}$

1. h_t *mainly reflects token* x_t *and summarizes tokens* $x_{i<t}$
2. *Value of* h_t *depends on the position of input tokens*
   ↳ *Swapping tokens at* $i, j < t$ *changes the context* h_t

`I am happy` *versus* `am I happy`

We can build features with these properties via self-attention mechanism

Self-Attention

Each token computes its context by attending to all tokens in the sequence

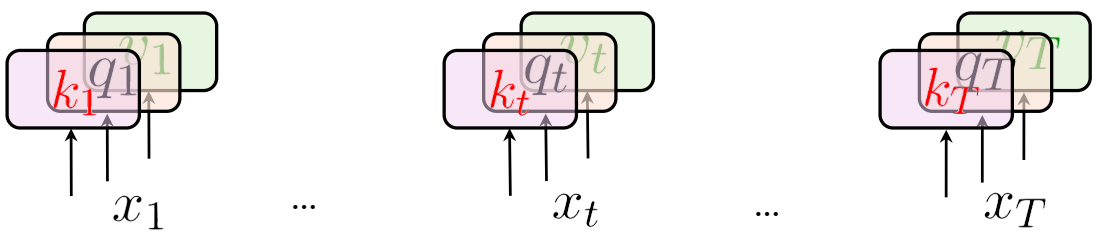
Self-Attention Mechanism

1: **for** $t = 1 : T$ **do** Compute key k_t, query q_t, and value v_t **end for** `#dim(`k_t`) = dim(`q_t`)`

$$\overleftrightarrow{k_t}^{d} = \mathbf{W}_{\text{key}} x_t \qquad \mathbf{K} = [k_1, \ldots, k_T]$$

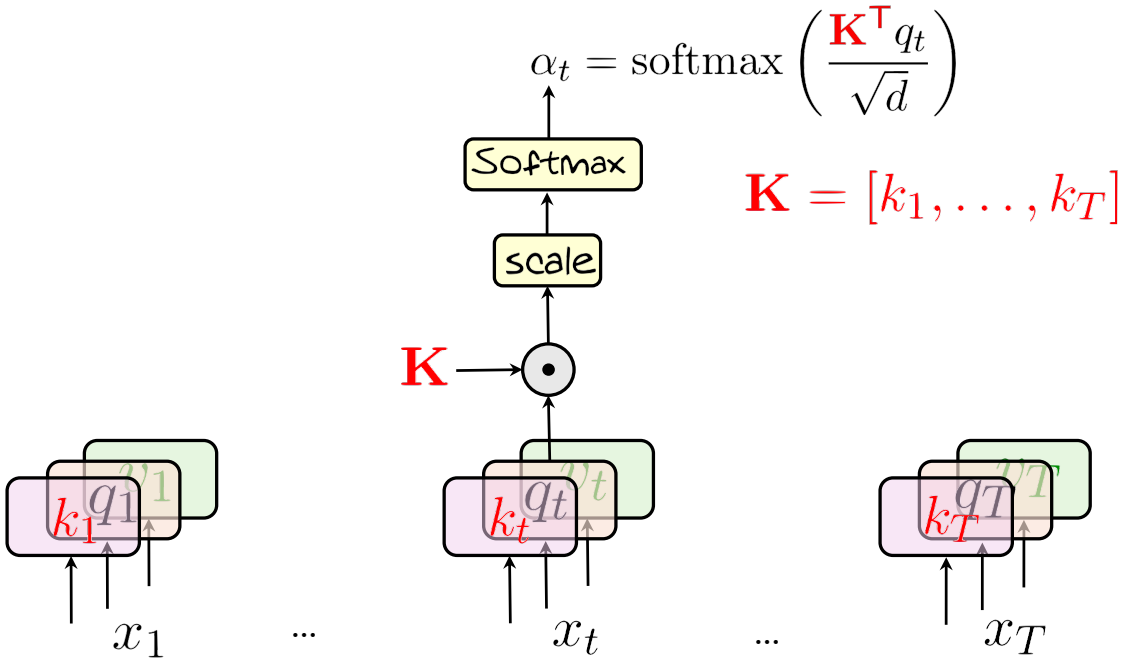
$$\overleftrightarrow{q_t}^{d} = \mathbf{W}_{\text{query}} x_t \qquad \mathbf{Q} = [q_1, \ldots, q_T]$$

$$v_t = \mathbf{W}_{\text{value}} x_t \qquad \mathbf{V} = [v_1, \ldots, v_T]$$



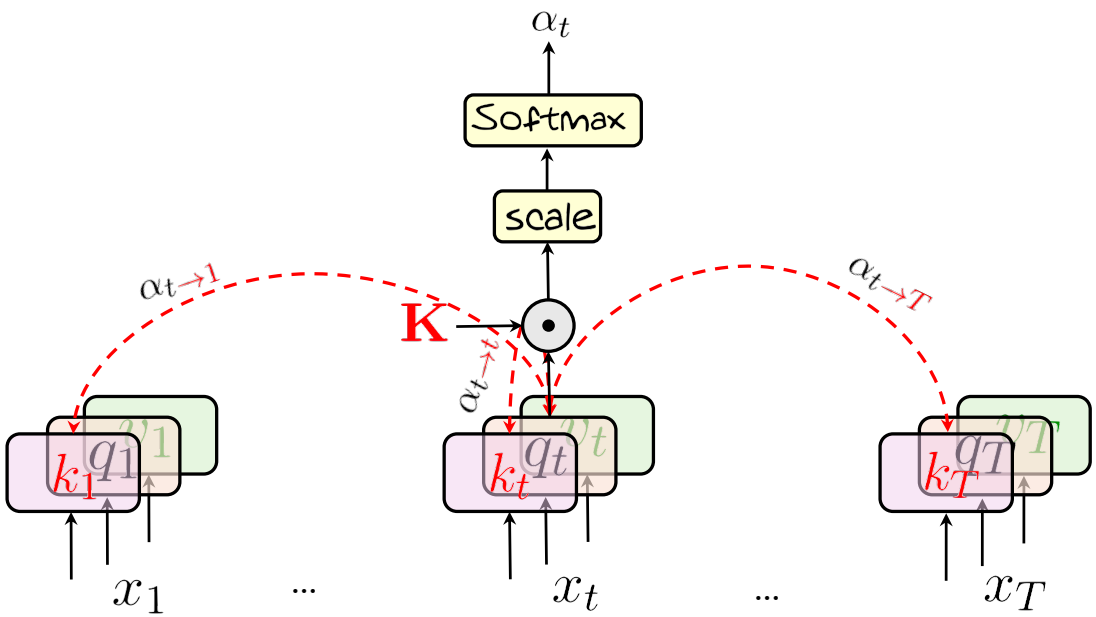
Self-Attention Mechanism

2: **for** $t = 1 : T$ **do** Compute attention coefficients α_t from q_t and $\mathbf{K} = [k_1 \dots k_T]$ **end for**



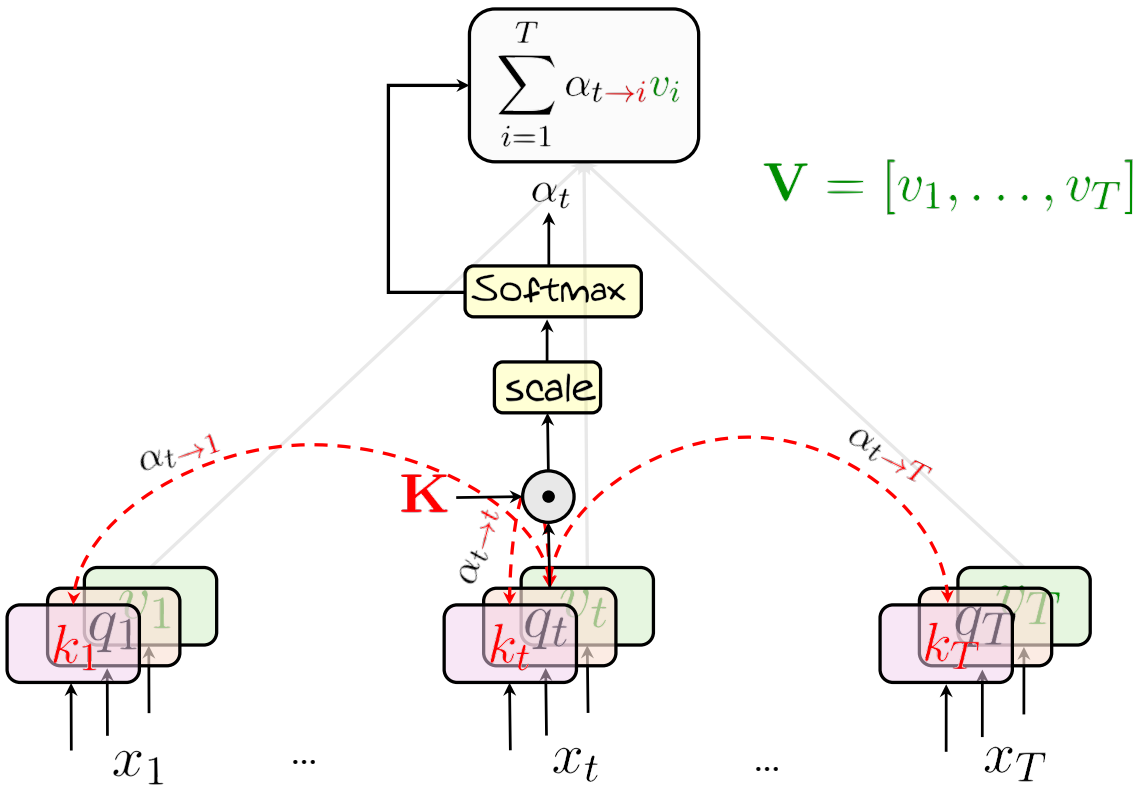
Self-Attention Mechanism

2: **for** $t = 1 : T$ **do** Compute attention coefficients α_t from q_t and $\mathbf{K} = [k_1 \dots k_T]$ **end for**



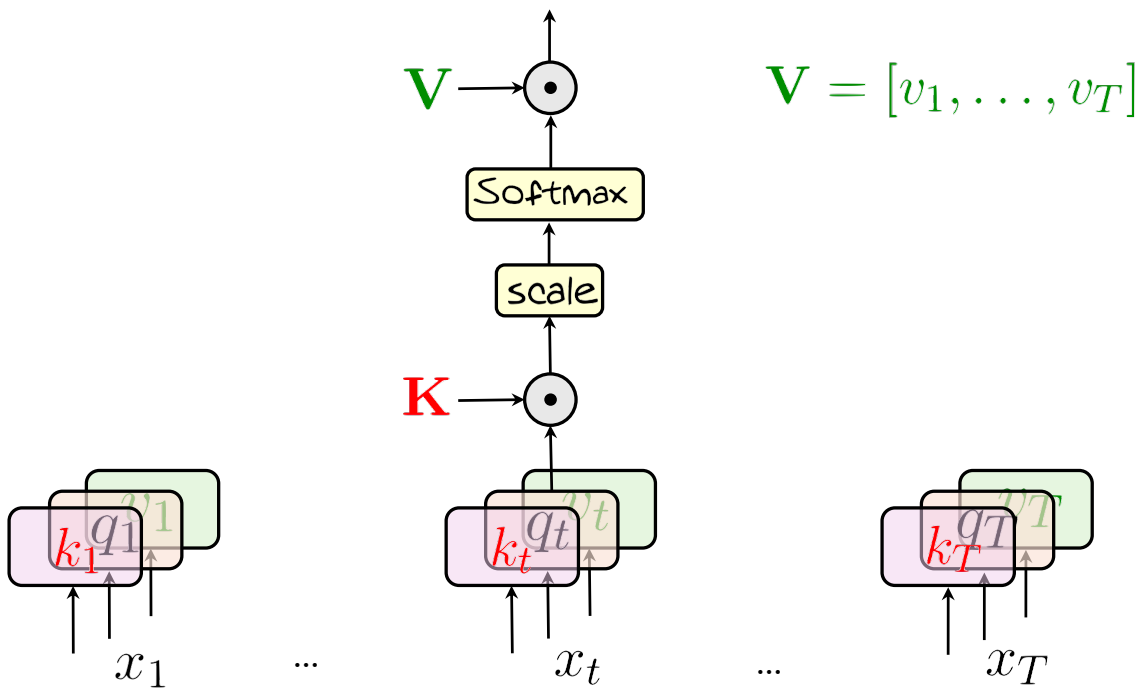
Self-Attention Mechanism

3: **for** $t = 1 : T$ **do** Compute $\mathbf{V}\alpha_t = [v_1 \dots v_T]\,\alpha_t$ as context **end for**



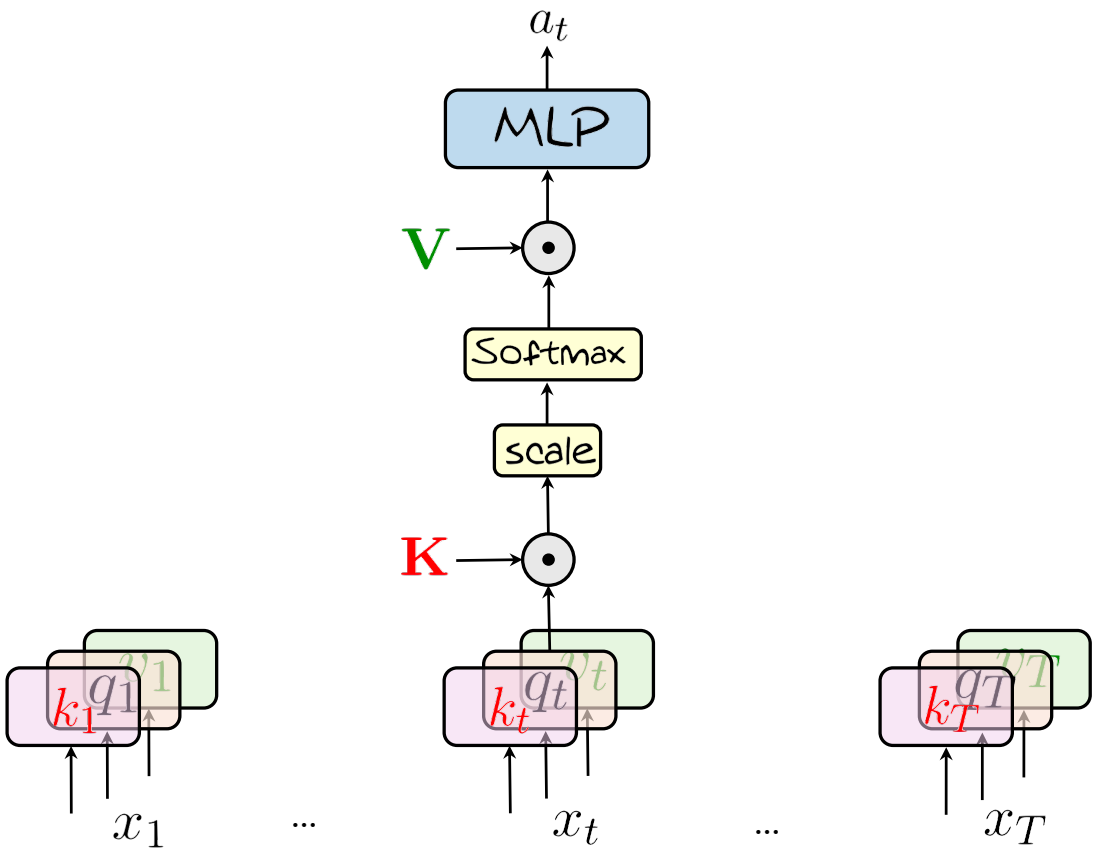
Self-Attention Mechanism

3: **for** $t = 1 : T$ **do** Compute $\mathbf{V}\alpha_t = [v_1 \dots v_T]\,\alpha_t$ as context **end for**



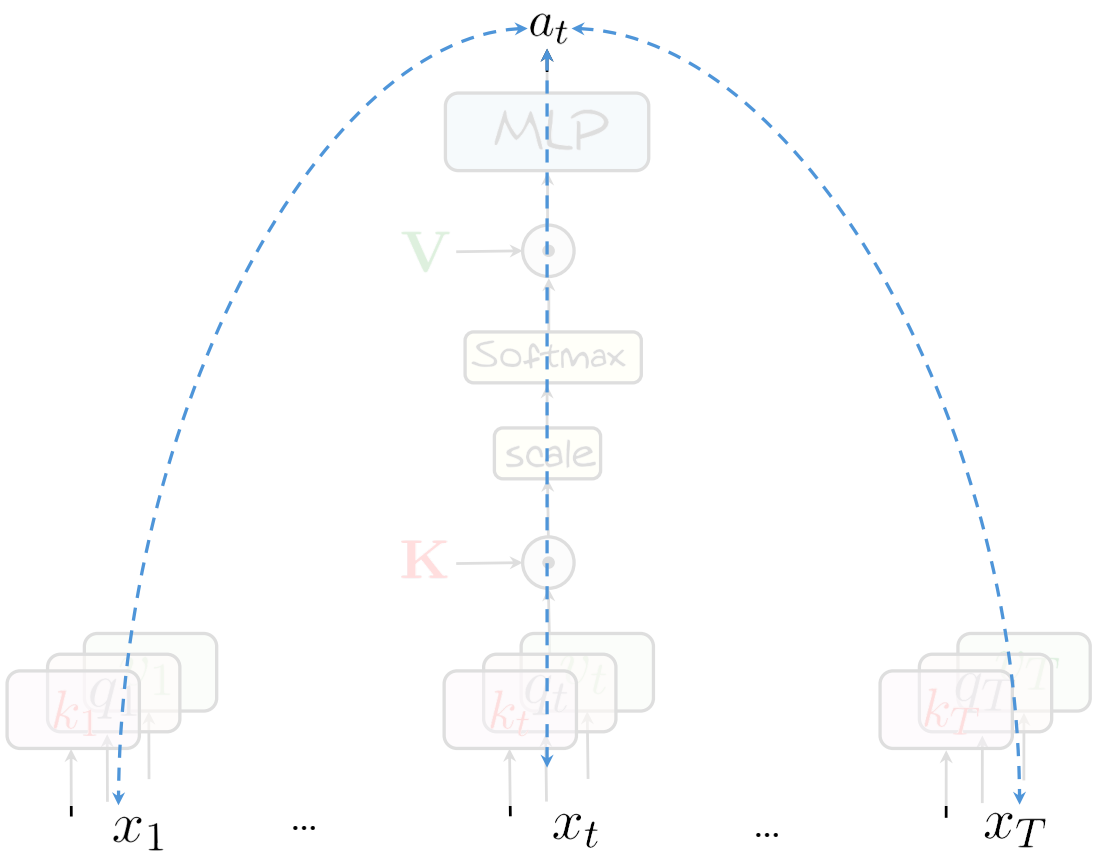
Self-Attention Mechanism

4: **for** $t = 1 : T$ **do** Compute attention-based context as $a_t = \texttt{MLP}\left(\mathbf{V}\alpha_t\right)$ **end for**



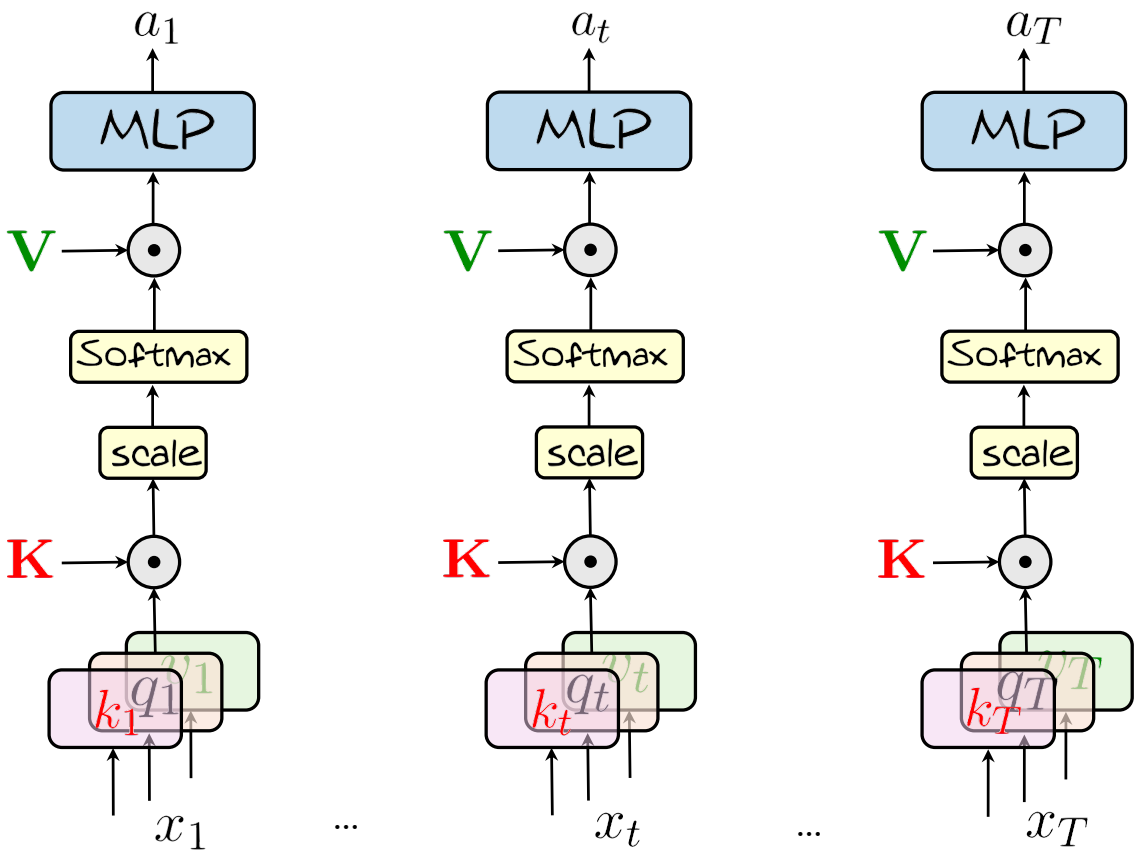
Self-Attention Mechanism

4: **for** $t = 1 : T$ **do** Compute attention-based context as $a_t = \texttt{MLP}\left(\mathbf{V}\alpha_t\right)$ **end for**



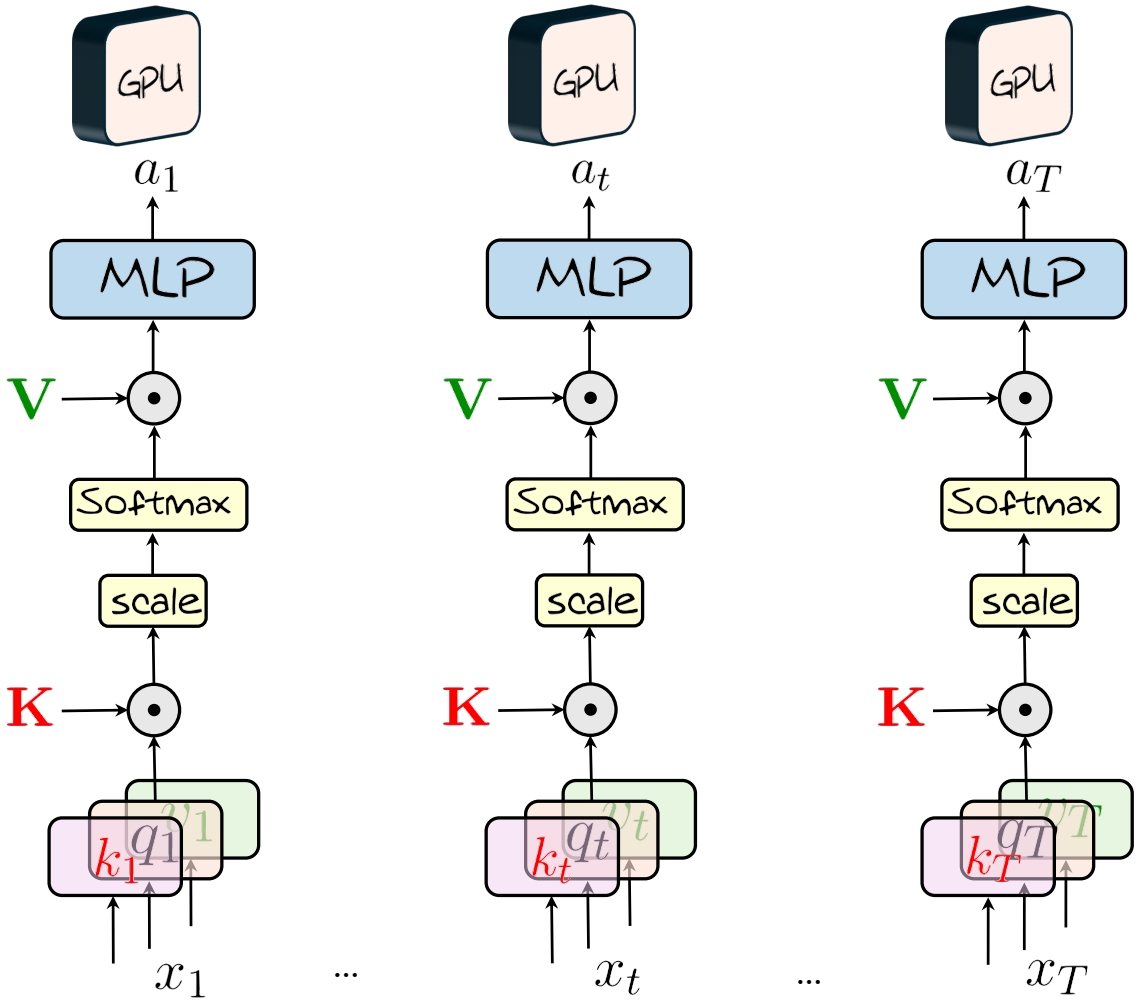
Self-Attention Mechanism

5: **return** $a_1 \dots a_T$



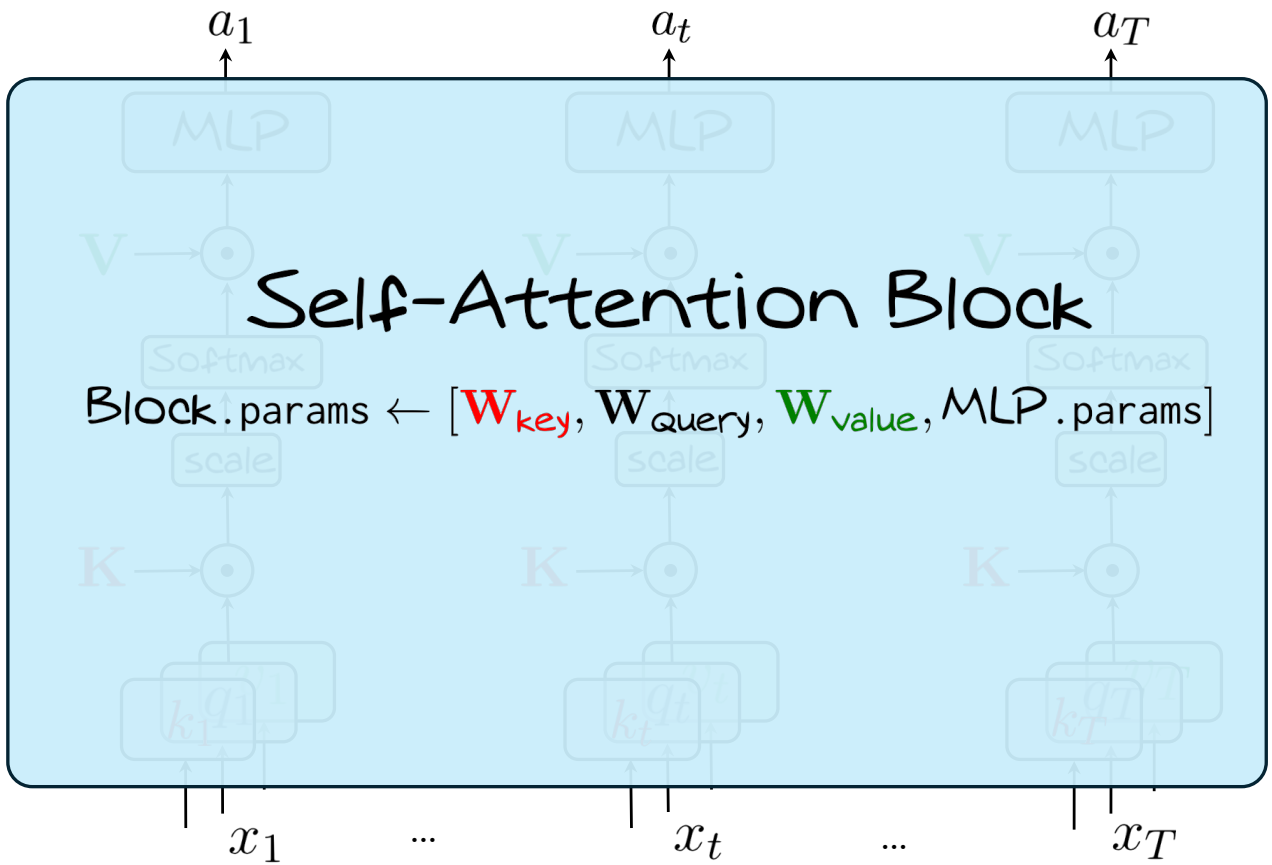
Self-Attention Mechanism

5: **return** $a_1 \dots a_T$



Self-Attention: *Computational Model*

Let's see how efficient this unit is implemented



Self-Attention: *Computational Model*

- *Arrays of keys, queries, and values are computed by matrix multiplication*

$$\mathbf{K} = \mathbf{W}_{\text{key}}[x_1 \ldots x_T] \quad \mathbf{Q} = \mathbf{W}_{\text{query}}[x_1 \ldots x_T] \quad \mathbf{V} = \mathbf{W}_{\text{value}}[x_1 \ldots x_T]$$

- *Array of all dot products is computed by*

$$\left[\mathbf{K}^\mathsf{T} q_1 \ldots \mathbf{K}^\mathsf{T} q_T\right] = \mathbf{K}^\mathsf{T}\mathbf{Q}$$

- *Array of attention coefficients is computed by column-wise Softmax*

$$[\alpha_1 \ldots \alpha_T] = \text{softmax}\left(\left[\frac{\mathbf{K}^\mathsf{T} q_1}{\sqrt{d}} \ldots \frac{\mathbf{K}^\mathsf{T} q_T}{\sqrt{d}}\right]\right) = \text{softmax}\left(\frac{\mathbf{K}^\mathsf{T}\mathbf{Q}}{\sqrt{d}}\right)$$

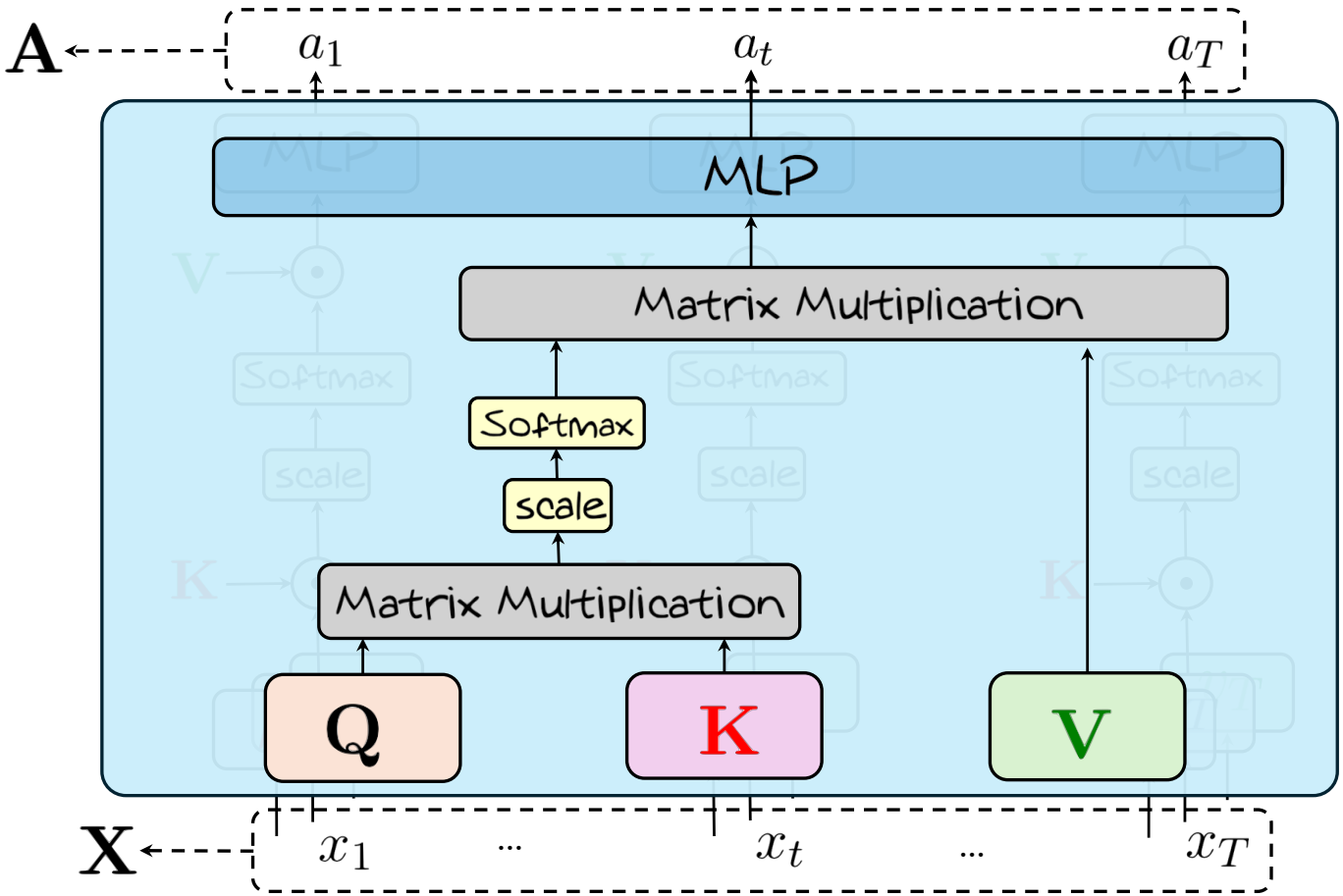
- *Array of all context vectors is computed by a matrix multiplication*

$$\left[\mathbf{V}\alpha_1 \ldots \mathbf{V}\alpha_T\right] = \mathbf{V}\left[\alpha_1 \ldots \alpha_T\right] = \mathbf{V}\,\text{softmax}\left(\frac{\mathbf{K}^\mathsf{T}\mathbf{Q}}{\sqrt{d}}\right)$$

all vectors are column-vectors

Self-Attention: *Computational Model*

The whole block is implemented very efficiently

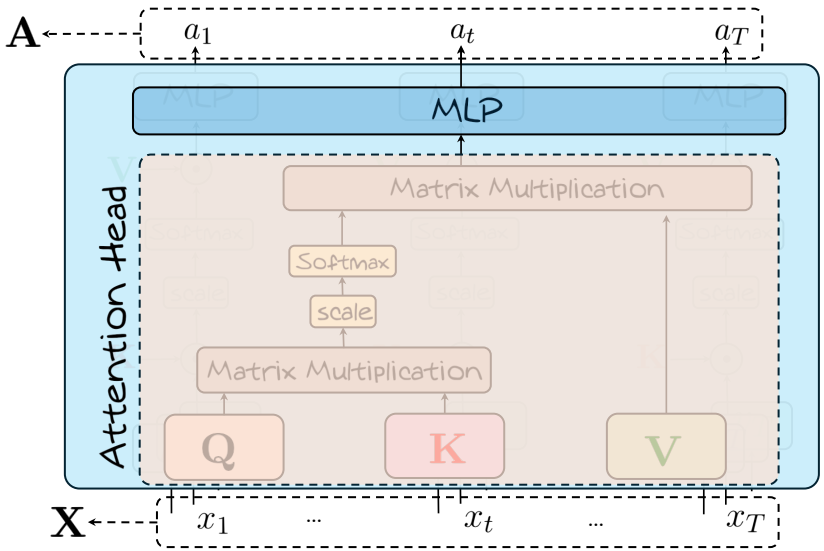


Self-Attention: *Pseudo Code*

`SelfAttenBlock`$(x_{1:T})$:

1: Compute $\bar{a}_{1:T} =$ `AttenHead`$(x_{1:T})$

2: **return** $a_{1:T} =$ `MLP`$(\bar{a}_{1:T})$



`AttenHead`$(x_{1:T})$:

1: Compute $\mathbf{K} = \mathbf{W}_{\text{key}}\left[x_{1:T}\right]$, $\mathbf{Q} = \mathbf{W}_{\text{que}}\left[x_{1:T}\right]$, and $\mathbf{V} = \mathbf{W}_{\text{val}}\left[x_{1:T}\right]$

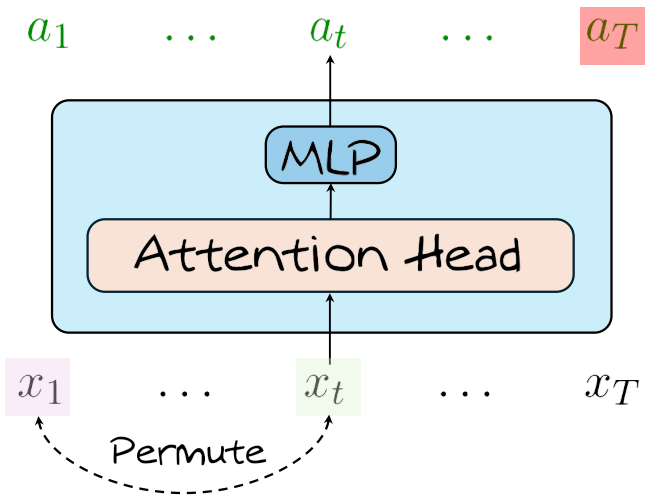
2: **return** $\bar{a}_{1:T} = \mathbf{V}\text{softmax}\left(\mathbf{K}^{\mathsf{T}}\mathbf{Q}/\sqrt{d}\right)$

Positional Dependency

Properties of Context Sequence

✔ *Context at t mainly reflects x_t and summarizes tokens $x_{i<t}$*

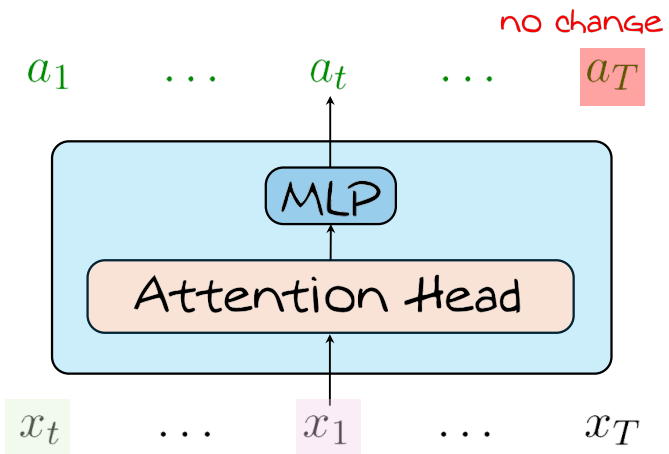
? *Context at t depends on the position of input tokens*



Positional Dependency

Properties of Context Sequence

- ✔ *Context at t mainly reflects x_t and summarizes tokens $x_{i<t}$*
- ✖ *Context at t depends on the position of input tokens*



Solution: *Positional Encoding*

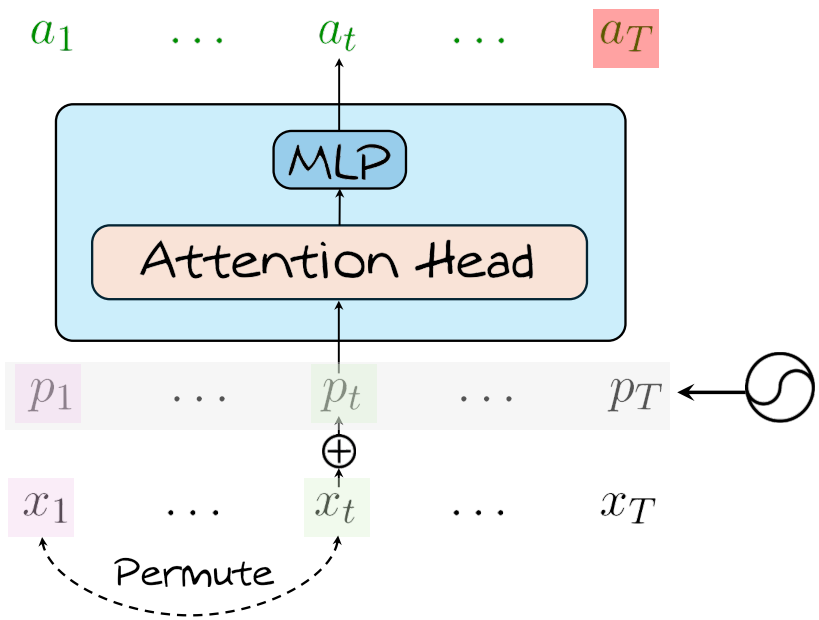
We can incorporate position information into the input sequence

Positional Encoding

Positional Encoding

Give a unique embedding p_t to position t and mix it with the token x_t

- ↳ p_t *has typically the same dimension as token embedding*
- ↳ *We can mix by adding the embeddings* $\mathsf{PosEnc}\left(x_t\right) = x_t + p_t$

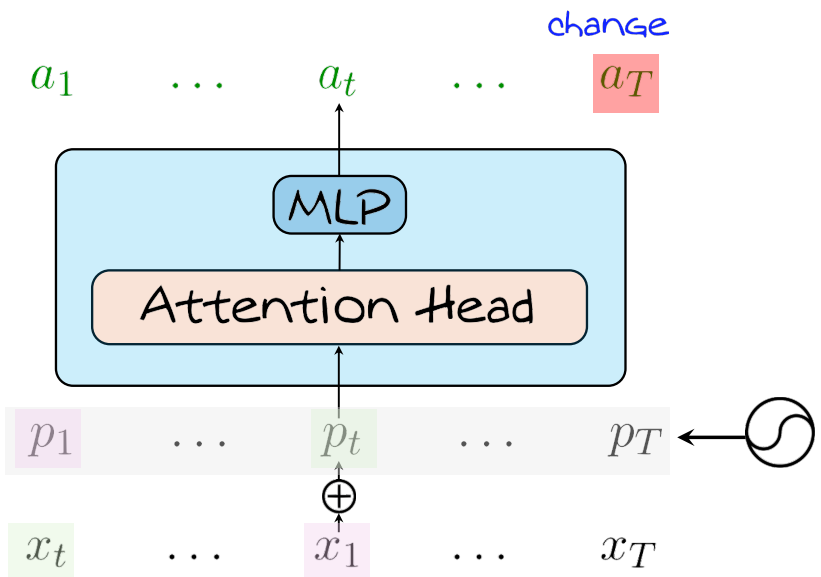


Positional Encoding

Positional Encoding

Give a unique embedding p_t to position t and mix it with the token x_t

- ↳ *p_t has typically the same dimension as token embedding*
- ↳ *We can mix by adding the embeddings* $\mathsf{PosEnc}\left(x_t\right) = x_t + p_t$



Example: *Sinusoidal Positional Embedding*

Example. *Unique positional embedding of length E is given at position t by*

$$p_t\left[2i+1\right] = \sin\left(\frac{t-1}{10000^{2i/E}}\right) \qquad p_t\left[2i+2\right] = \cos\left(\frac{t-1}{10000^{2i/E}}\right)$$

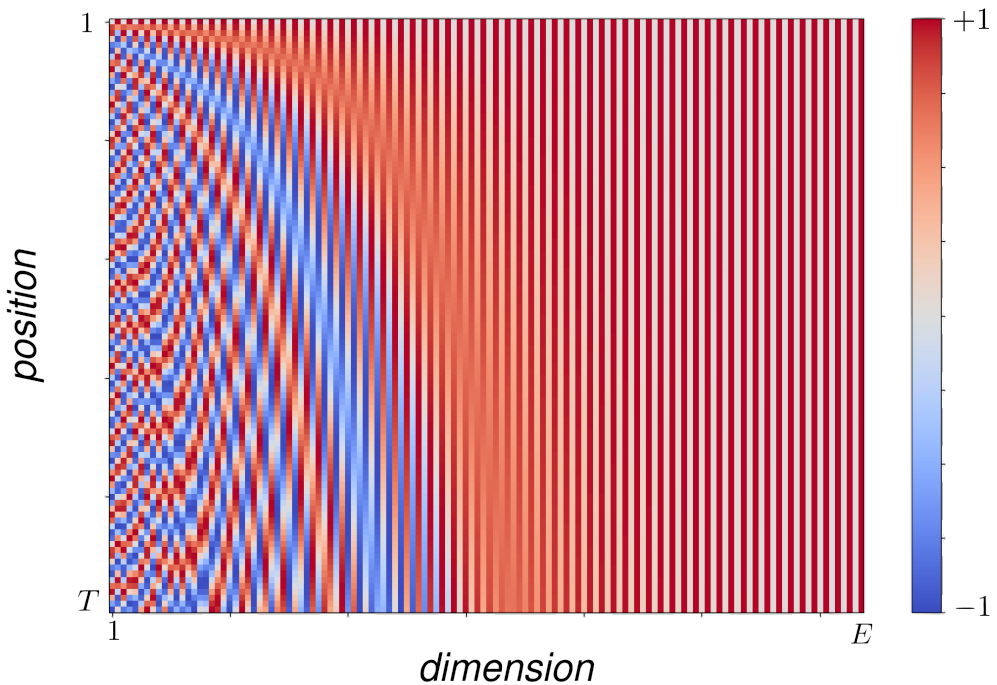
with i running from 0 to $E/2-1$

Example: *Sinusoidal Positional Embedding*

Example. *Unique positional embedding of length E is given at position t by*

$$p_t\left[2i+1\right] = \sin\left(\frac{t-1}{10000^{2i/E}}\right) \qquad p_t\left[2i+2\right] = \cos\left(\frac{t-1}{10000^{2i/E}}\right)$$

with i running from 0 to $E/2-1$

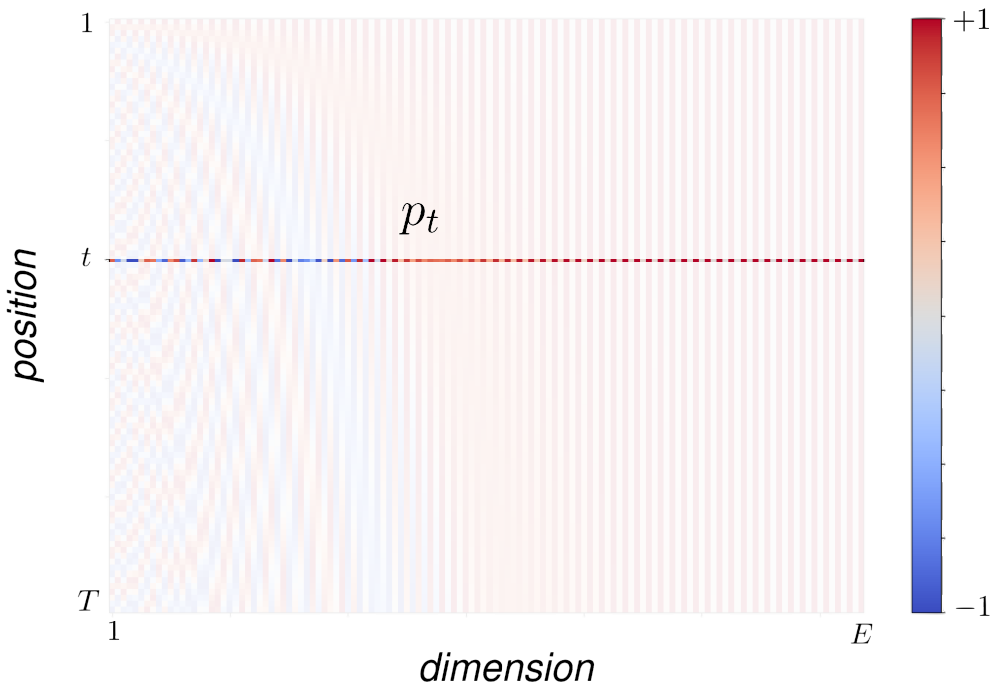


Example: *Sinusoidal Positional Embedding*

Example. *Unique positional embedding of length E is given at position t by*

$$p_t\left[2i+1\right] = \sin\left(\frac{t-1}{10000^{2i/E}}\right) \qquad p_t\left[2i+2\right] = \cos\left(\frac{t-1}{10000^{2i/E}}\right)$$

with i running from 0 to $E/2-1$



Increasing Complexity of Attention Block

We can easily incorporate positional encoding into the attention head

`AttenHead`$(x_{1:T})$:

1: Compute $\mathbf{K} = \mathbf{W}_{\text{key}}\left[x_{1:T}\right]$, $\mathbf{Q} = \mathbf{W}_{\text{que}}\left[x_{1:T}\right]$, and $\mathbf{V} = \mathbf{W}_{\text{val}}\left[x_{1:T}\right]$

2: **return** $\bar{a}_{1:T} = \mathbf{V}\text{softmax}\left(\mathbf{K}^{\mathsf{T}}\mathbf{Q}/\sqrt{d}\right)$

But the attention block remains relatively simple

`SelfAttenBlock`$(x_{1:T})$:

1: Encode position as $x_{1:T} \leftarrow x_{1:T} + p_{1:T}$

2: Compute $\bar{a}_{1:T} =$ `AttenHead`$(x_{1:T})$

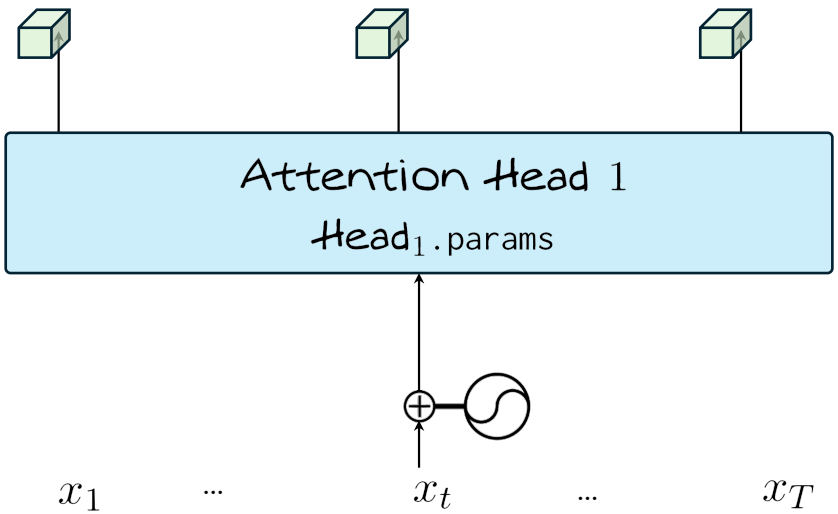
3: **return** $a_{1:T} =$ `MLP`$(\bar{a}_{1:T})$

The parameters are the weights of attention head and MLP

- *We can deploy more attention heads*

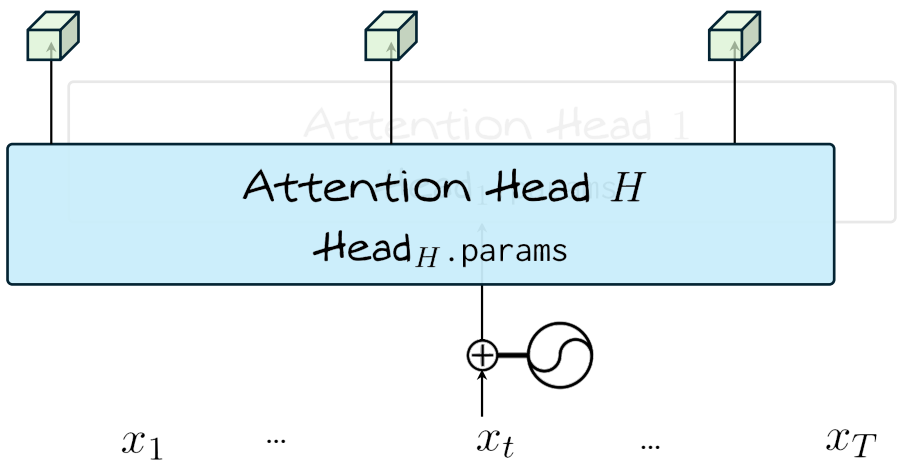
Multi-head Attention

We can deploy multiple attention heads



Multi-head Attention

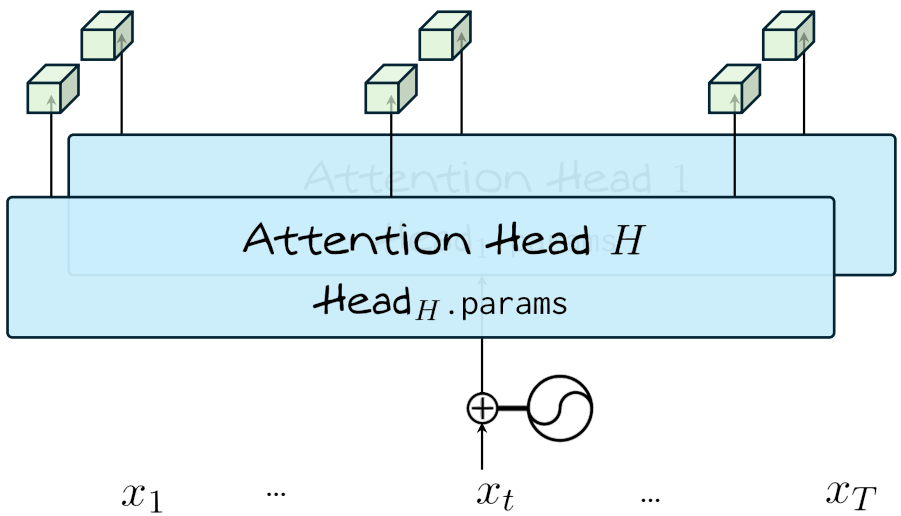
We can deploy multiple attention heads



Multi-head Attention

We can deploy multiple attention heads

↳ *each head looks at sequence from different perspective* ⇝ *different context*



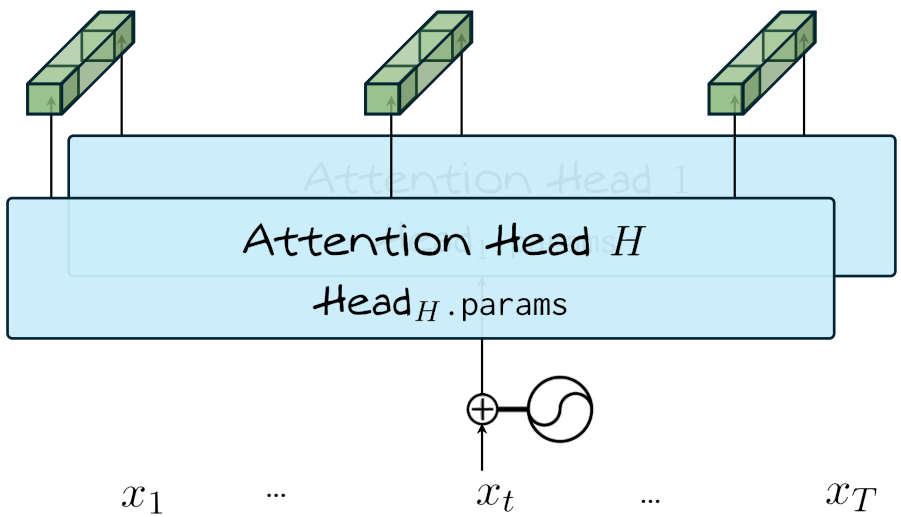
Multi-head Attention

We can deploy multiple attention heads

↳ *each head looks at sequence from different perspective* ⇝ *different context*

We then mix the head contexts at each time

↳ *mixing can be simply done by concatenation and a weighted layer*



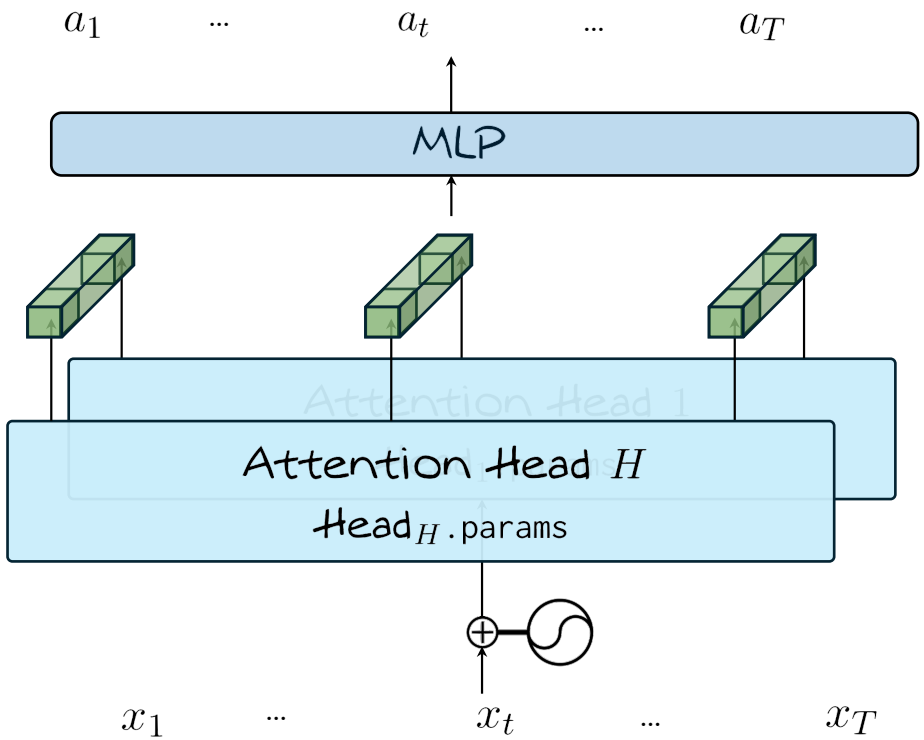
Multi-head Attention

We can deploy multiple attention heads

↳ *each head looks at sequence from different perspective* ⇝ *different context*

We then mix the head contexts at each time and process them by a shared MLP

↳ *mixing can be simply done by concatenation and a weighted layer*



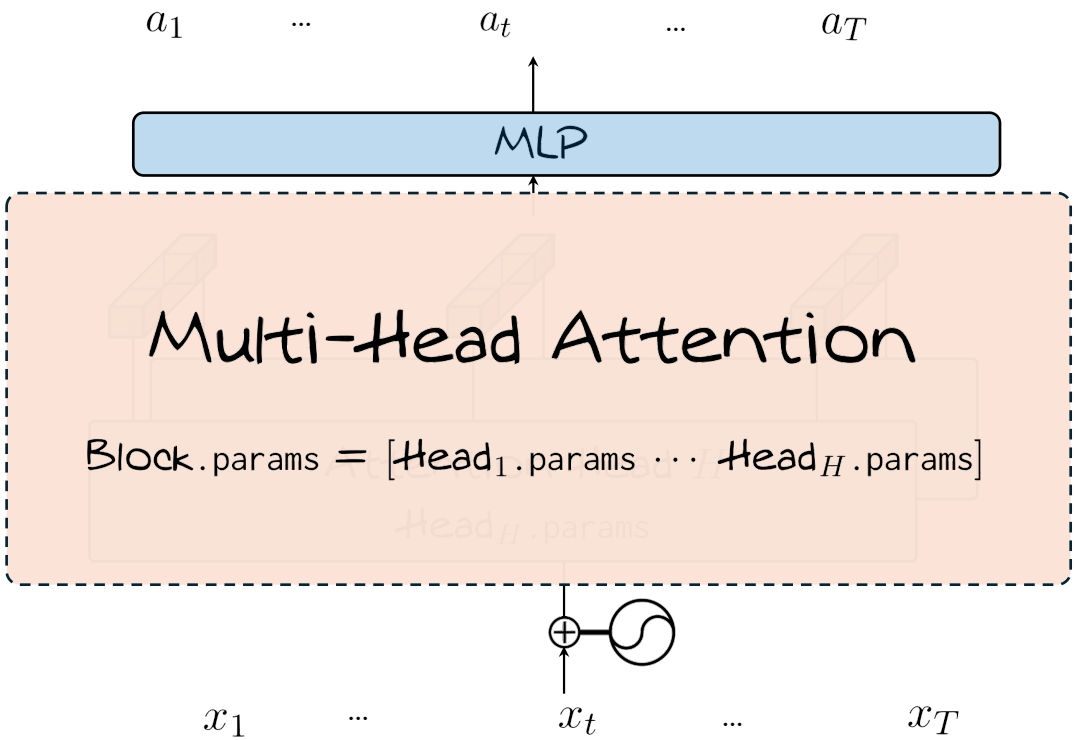
Multi-head Attention

We can deploy multiple attention heads

↳ *each head looks at sequence from different perspective* ⇝ *different context*

We then mix the head contexts at each time and process them by a shared MLP

↳ *mixing can be simply done by concatenation and a weighted layer*



Multi-head Attention: *Pseudo Code*

Each attention head is implemented exactly as before with its own parameters

`AttenHead`$^h(x_{1:T})$:

1: Compute $\mathbf{K}_h = \mathbf{W}^h_{\text{key}}\left[x_{1:T}\right]$, $\mathbf{Q}_h = \mathbf{W}^h_{\text{que}}\left[x_{1:T}\right]$, and $\mathbf{V}_h = \mathbf{W}^h_{\text{val}}\left[x_{1:T}\right]$

2: **return** $\bar{a}^h_{1:T} = \mathbf{V}_h \text{softmax}\left(\mathbf{K}_h^\intercal \mathbf{Q}_h / \sqrt{d}\right)$

The multi-head attention can significantly benefit from parallel computation

`mHeadAtten`$(x_{1:T})$:

1: Encode position as $x_{1:T} \leftarrow x_{1:T} + p_{1:T}$

2: **for** $h = 1 : H$ **do**

3: Compute $\bar{a}^h_{1:T} =$ `AttenHead`$^h(x_{1:T})$ `#run in parallel`

4: **end for**

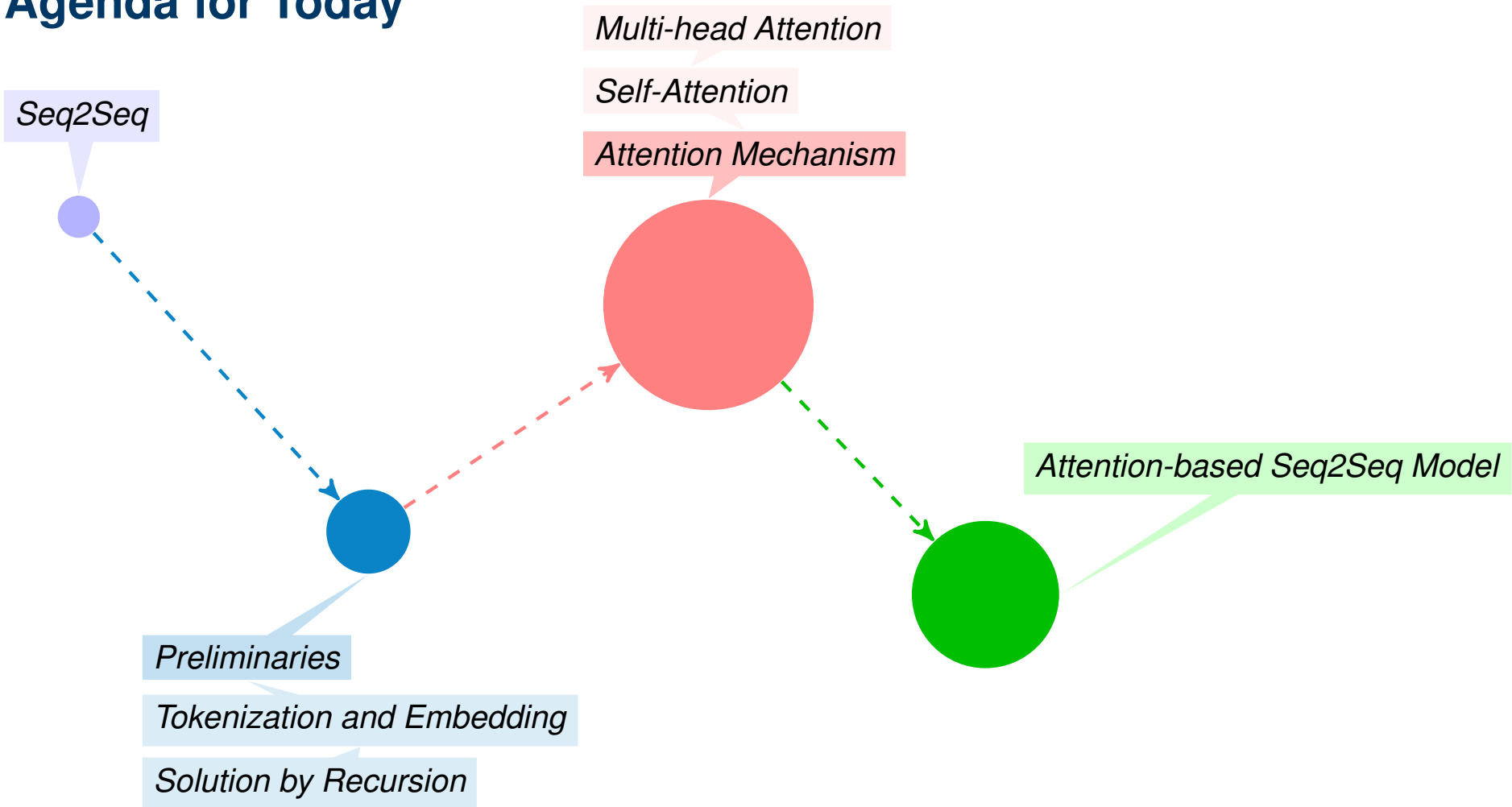
5: **return** $\bar{a}_t = \mathbf{W}_{\text{mix}} \text{concat}\left[\bar{a}^1_t \ldots \bar{a}^H_t\right]$ for $t = 1 : T$

The outputs of multi-head attention block can be further processed by an MLP

`mHeadAttenBlock`$(x_{1:T})$:

1: **return** $a_{1:T} =$ `MLP(mHeadAtten(`$x_{1:T}$`))`

Agenda for Today



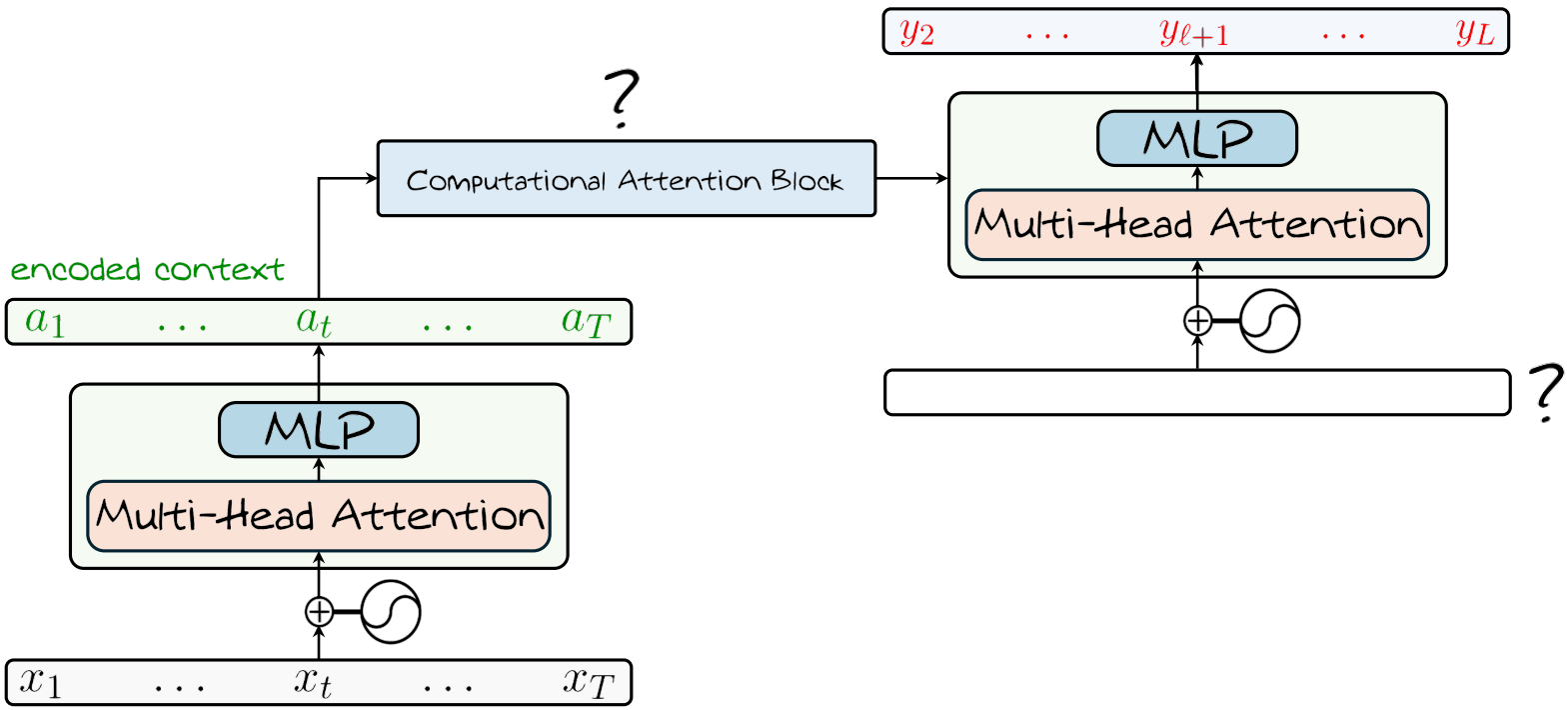
Transformer: *First Sketch*

Transformer in Nutshell

Generate context at the encoder and decoder via multi-head attention

↳ *Also attention mechanism injects encoded context to the decoder*

? *Encoding is easy, but decoding and cross-attention need more thoughts*



Decoding via Self-Attention: *Inference*

- **!** Decoding is sequential by nature
  - ↳ *We start with* $y_1 = \texttt{<start>}$*, then get* y_2*, and so on*

Decoding via Self-Attention: *Inference*

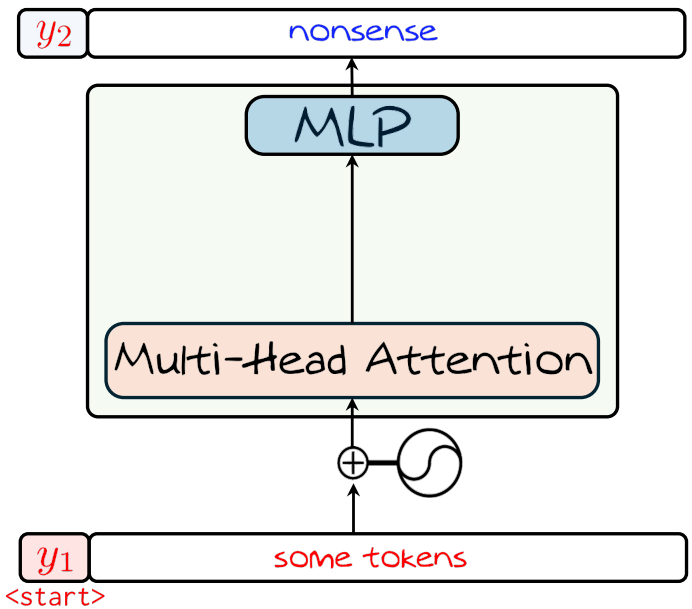
- Decoding is sequential by nature
  - ↳ *We start with $y_1 =$ <start>, then get y_2, and so on*
- We can use the attention block sequentially during inference

Decoding via Self-Attention: *Inference*

- Decoding is sequential by nature
  - ↳ *We start with $y_1 =$ <start>, then get y_2, and so on*
- We can use the attention block sequentially during inference

1. *Start with $y_1 =$ <start> and fill the remaining with some tokens*

↳ *Get the complete sequence but care only about first token*



Decoding via Self-Attention: *Inference*

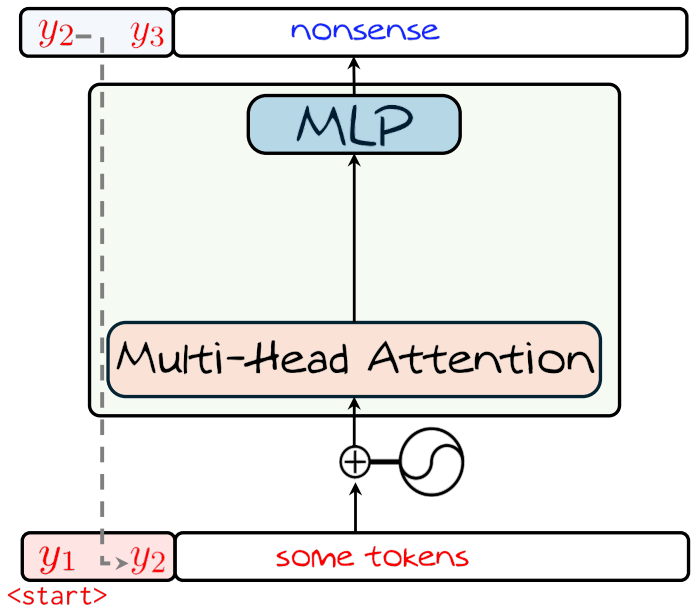
- Decoding is sequential by nature
  - ↳ *We start with $y_1 =$ <start>, then get y_2, and so on*
- We can use the attention block sequentially during inference

1. *Start with $y_1 =$ <start> and fill the remaining with some tokens*

↳ *Get the complete sequence but care only about first token*

2. *Input again y_1, y_2 and fill the remaining with some tokens*

↳ *Get the sequence but care only about first two tokens*



Decoding via Self-Attention: *Inference*

- Decoding is sequential by nature
  - ↳ *We start with $y_1 =$ <start>, then get y_2, and so on*
- We can use the attention block sequentially during inference

1. *Start with $y_1 =$ <start> and fill the remaining with some tokens*

↳ *Get the complete sequence but care only about first token*

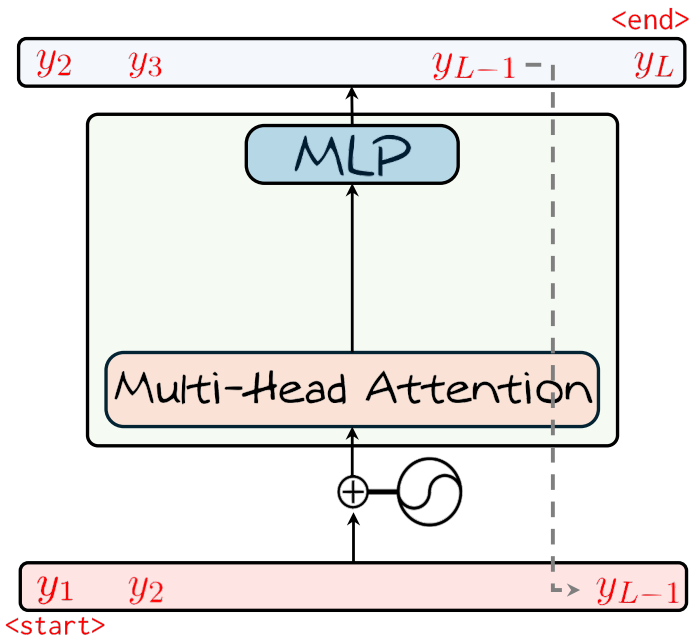
2. *Input again y_1, y_2 and fill the remaining with some tokens*

↳ *Get the sequence but care only about first two tokens*

⋮

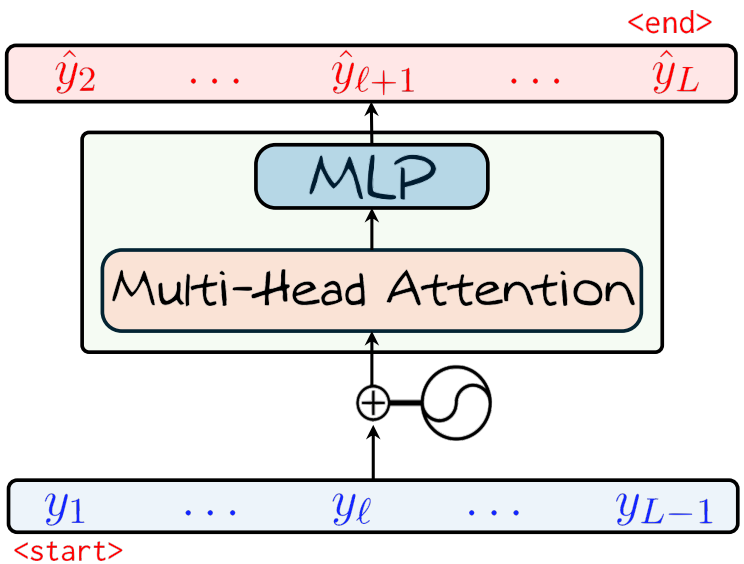
$(L-1)$. *Input $y_1, \ldots, y_{L-1}$*

↳ *Get sequence $y_2, \ldots, y_L$*



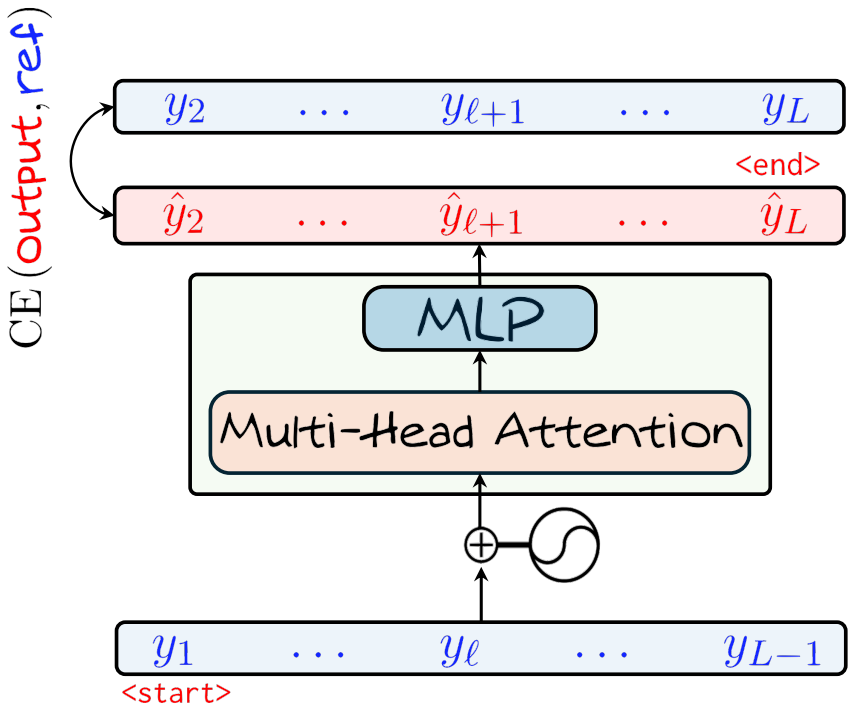
Decoding via Self-Attention: *Training*

- Unlike inference, training still benefits from parallel decoding



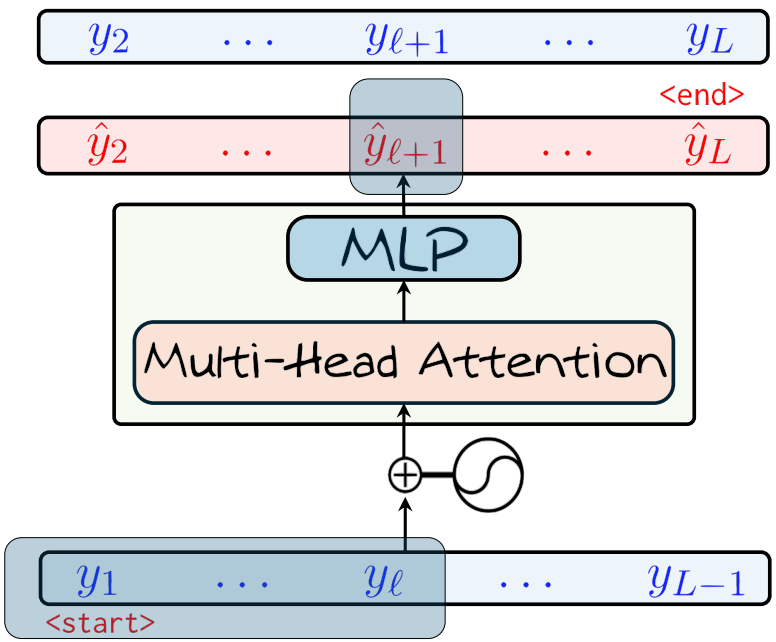
Decoding via Self-Attention: *Training*

- Unlike inference, training still benefits from parallel decoding



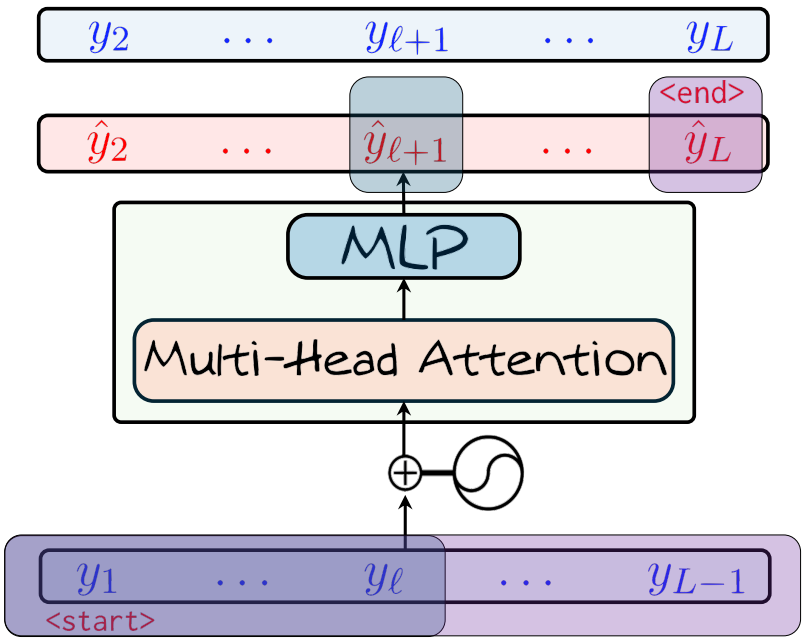
Decoding via Self-Attention: *Training*

- Unlike inference, training still benefits from parallel decoding
  - ↳ *In training, we know the true tokens*



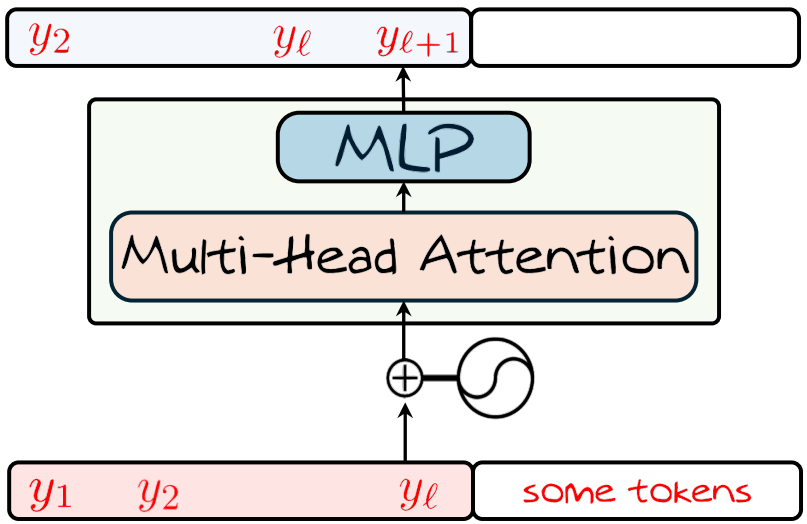
Decoding via Self-Attention: *Training*

- Unlike inference, training still benefits from parallel decoding
  - ↳ *In training, we know the true tokens*
  - *We can use "teacher forcing" to compute all next tokens at once*



Decoding via Self-Attention: *Causality*

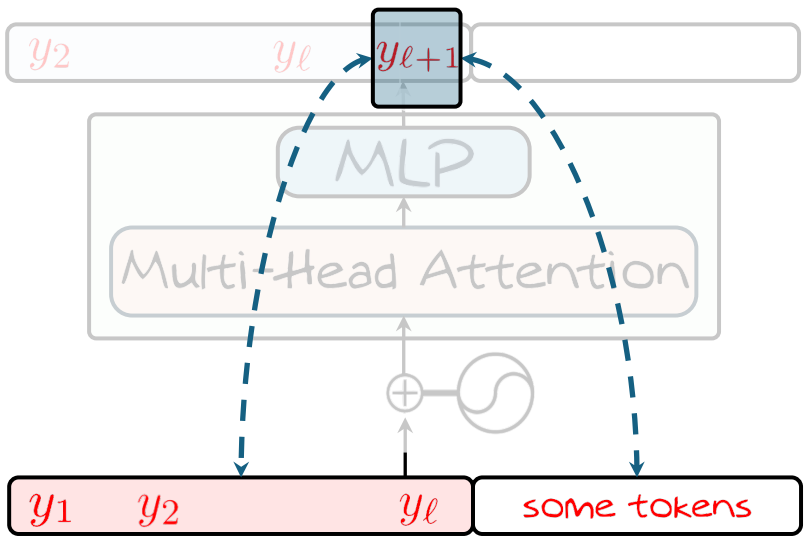
The decoding by self-attention is not causal



Decoding via Self-Attention: *Causality*

The decoding by self-attention is not causal

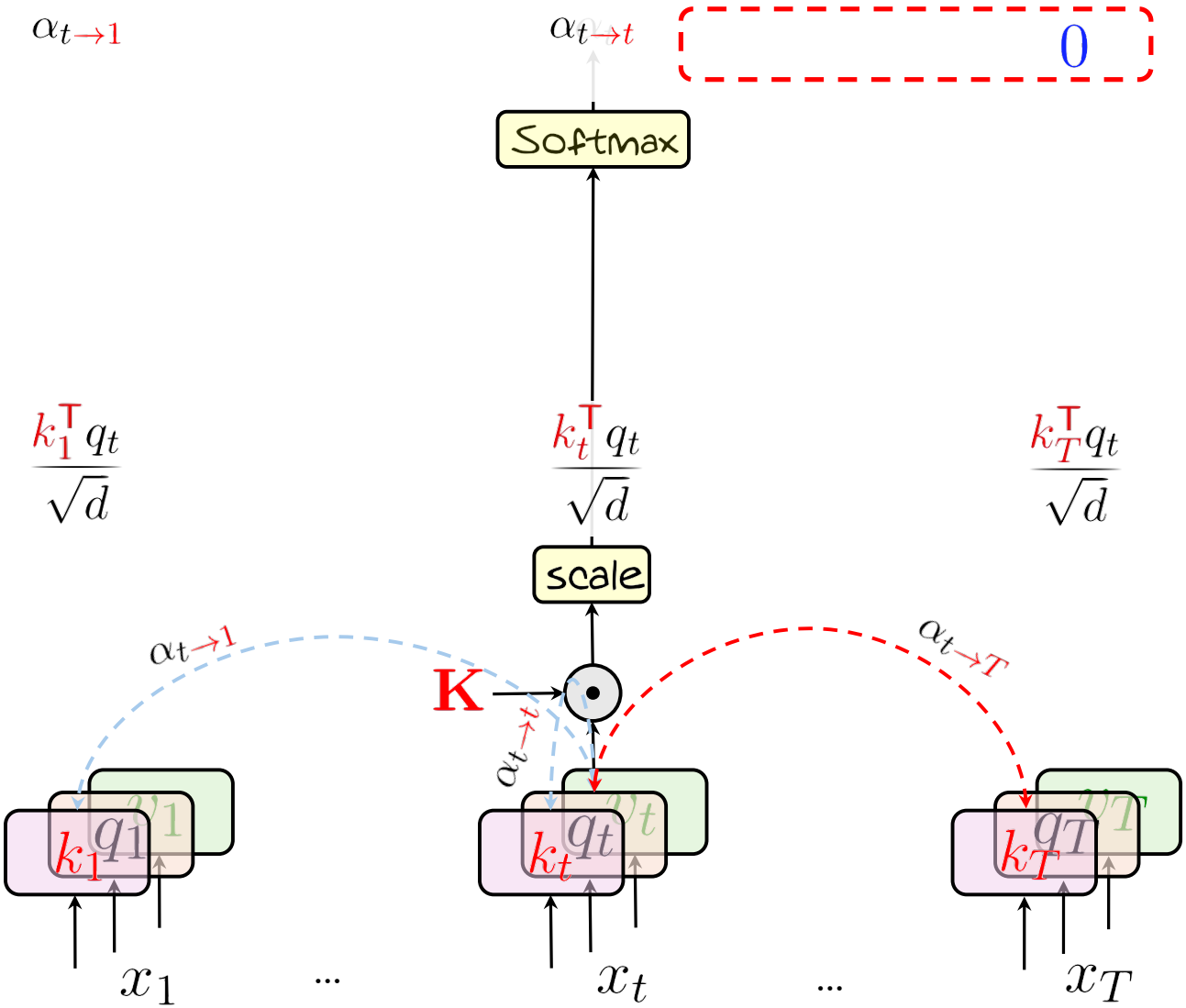
- ! At time ℓ, next token is predicted from complete input including future tokens
  - ↳ *Training* ⇝ *Next token is already revealed at the input!*



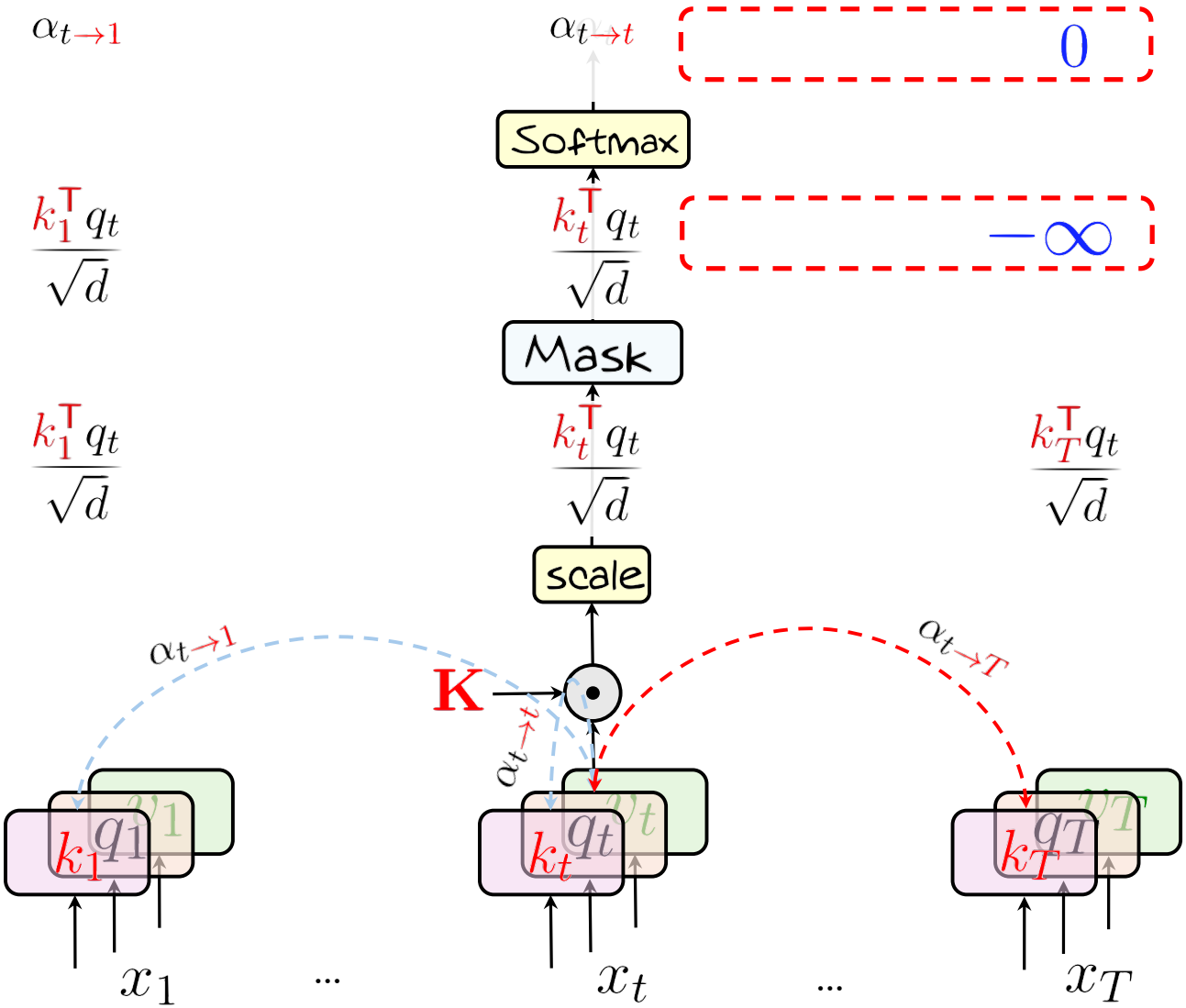
Solution: *Masked Decoding*

While computing attention outputs, we mask out the value of future tokens

Masked Decoding



Masked Decoding



Masked Decoding: *Formulation*

Let's look at the array of dot products

$$\mathbf{K}^\mathsf{T}\mathbf{Q} = \begin{bmatrix} k_1^\mathsf{T} q_1 & \dots & k_1^\mathsf{T} q_t & \dots & k_1^\mathsf{T} q_T \\ k_2^\mathsf{T} q_1 & \dots & k_2^\mathsf{T} q_t & \dots & k_2^\mathsf{T} q_T \\ & \ddots & & \ddots & \\ k_t^\mathsf{T} q_1 & \dots & k_t^\mathsf{T} q_t & \dots & k_t^\mathsf{T} q_T \\ & \ddots & & \ddots & \\ k_T^\mathsf{T} q_1 & \dots & k_T^\mathsf{T} q_t & \dots & k_T^\mathsf{T} q_T \end{bmatrix}$$

Masked Decoding: *Formulation*

Let's look at the array of dot products

$$\mathbf{K}^\mathsf{T}\mathbf{Q} = \begin{bmatrix} k_1^\mathsf{T} q_1 & \dots & k_1^\mathsf{T} q_t & \dots & k_1^\mathsf{T} q_T \\ & \dots & k_2^\mathsf{T} q_t & \dots & k_2^\mathsf{T} q_T \\ & \ddots & & \ddots & \\ -\infty & \dots & k_t^\mathsf{T} q_t & \dots & k_t^\mathsf{T} q_T \\ & \ddots & -\infty & \ddots & \\ & \dots & & \dots & k_T^\mathsf{T} q_T \end{bmatrix}$$

We can readily mask by

$$\texttt{masked}\left(\mathbf{K}^\mathsf{T}\mathbf{Q}\right) = \mathbf{K}^\mathsf{T}\mathbf{Q} + \mathbf{M}$$

with lower triangle matrix

$$\mathbf{M} = \begin{bmatrix} 0 & \dots & 0 & \dots & 0 \\ -\infty & \dots & 0 & \dots & 0 \\ & \ddots & & \ddots & \\ -\infty & \dots & -\infty & \dots & 0 \end{bmatrix}$$

Masked Multi-head Attention: *Pseudo Code*

We can add masking option to attention heads

`AttenHead`^{h}`(`$x_{1:T}$`,mask):`

1: Compute $\mathbf{K}_h = \mathbf{W}^h_{\text{key}}\left[x_{1:T}\right]$, $\mathbf{Q}_h = \mathbf{W}^h_{\text{que}}\left[x_{1:T}\right]$, and $\mathbf{V}_h = \mathbf{W}^h_{\text{val}}\left[x_{1:T}\right]$

2: **if** `mask = True` **do** compute $\mathbf{D}_h = \mathbf{K}_h^\mathsf{T}\mathbf{Q}_h + \mathbf{M}$ **else** compute $\mathbf{D}_h = \mathbf{K}_h^\mathsf{T}\mathbf{Q}_h$

3: **return** $\bar{a}^h_{1:T} = \mathbf{V}_h \text{softmax}\left(\mathbf{D}_h/\sqrt{d}\right)$

Masked multi-head attention uses masking in all heads

`mHeadAtten(`$x_{1:T}$`,mask):`

1: Encode position as $x_{1:T} \leftarrow x_{1:T} + p_{1:T}$

2: **for** $h = 1 : H$ **do**

3: Compute $\bar{a}^h_{1:T} =$ `AttenHead`^{h}`(`$x_{1:T}$`,mask)` `#run in parallel`

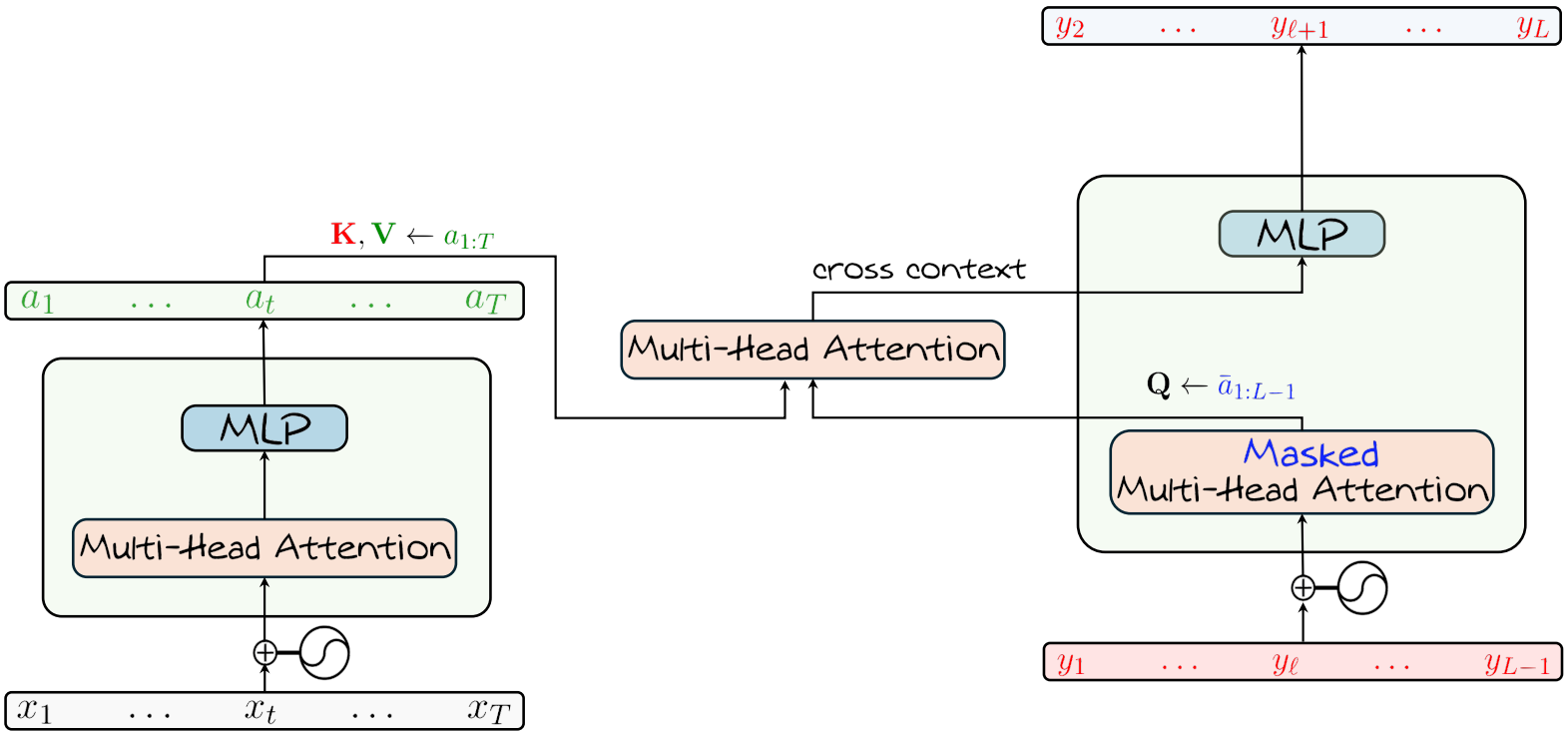
4: **end for**

5: **return** $\bar{a}_t = \mathbf{W}_{\text{mix}} \text{concat}\left[\bar{a}^1_t \ldots \bar{a}^H_t\right]$ for $t = 1 : T$

Cross Attention

For attending to the encoded context, we can use computational attention block

- *Treat encoder output and decoder masked output as context*
- *Generate keys and values from encoder output*
- *Generate queries from decoder output*
- *At time ℓ, the decoder attends to input tokens using multi-head attention*



Cross Attention: *Pseudo Code*

Let's implement the cross attention unit

$\texttt{CrAttenHead}^h(a_{1:T}, \bar{a}_{1:L-1})$:

1: Compute $\mathbf{K}_h = \mathbf{W}^h_{\text{key}}\left[a_{1:T}\right]$ and $\mathbf{V}_h = \mathbf{W}^h_{\text{val}}\left[a_{1:T}\right]$

2: Compute $\mathbf{Q}_h = \mathbf{W}^h_{\text{que}}\left[\bar{a}_{1:L-1}\right]$

3: **return** $c^h_{1:L-1} = \mathbf{V}_h \text{softmax}\left(\mathbf{K}_h^\mathsf{T}\mathbf{Q}_h/\sqrt{d}\right)$

The multi-head cross attention simply uses multiple heads

$\texttt{mHeadCrAtten}(a_{1:T}, \bar{a}_{1:L-1})$:

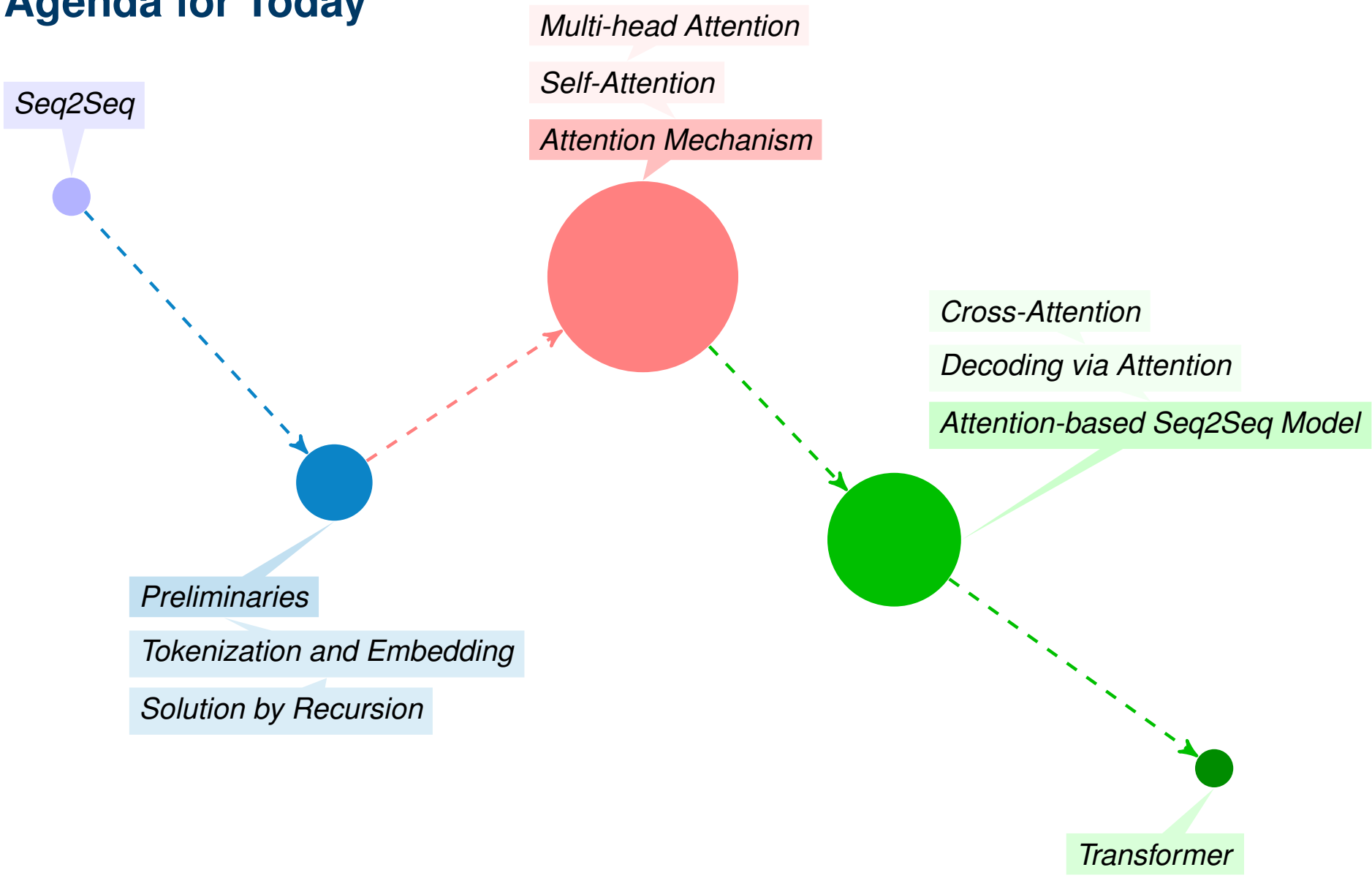
1: **for** $h = 1 : H$ **do**

2: Compute $c^h_{1:L-1} = \texttt{CrAttenHead}^h(a_{1:T}, \bar{a}_{1:L-1})$ `#run in parallel`

3: **end for**

4: **return** $c_\ell = \mathbf{W}_{\text{mix}}\text{concat}\left[c^1_\ell \ldots c^H_\ell\right]$ for $\ell = 1 : L-1$

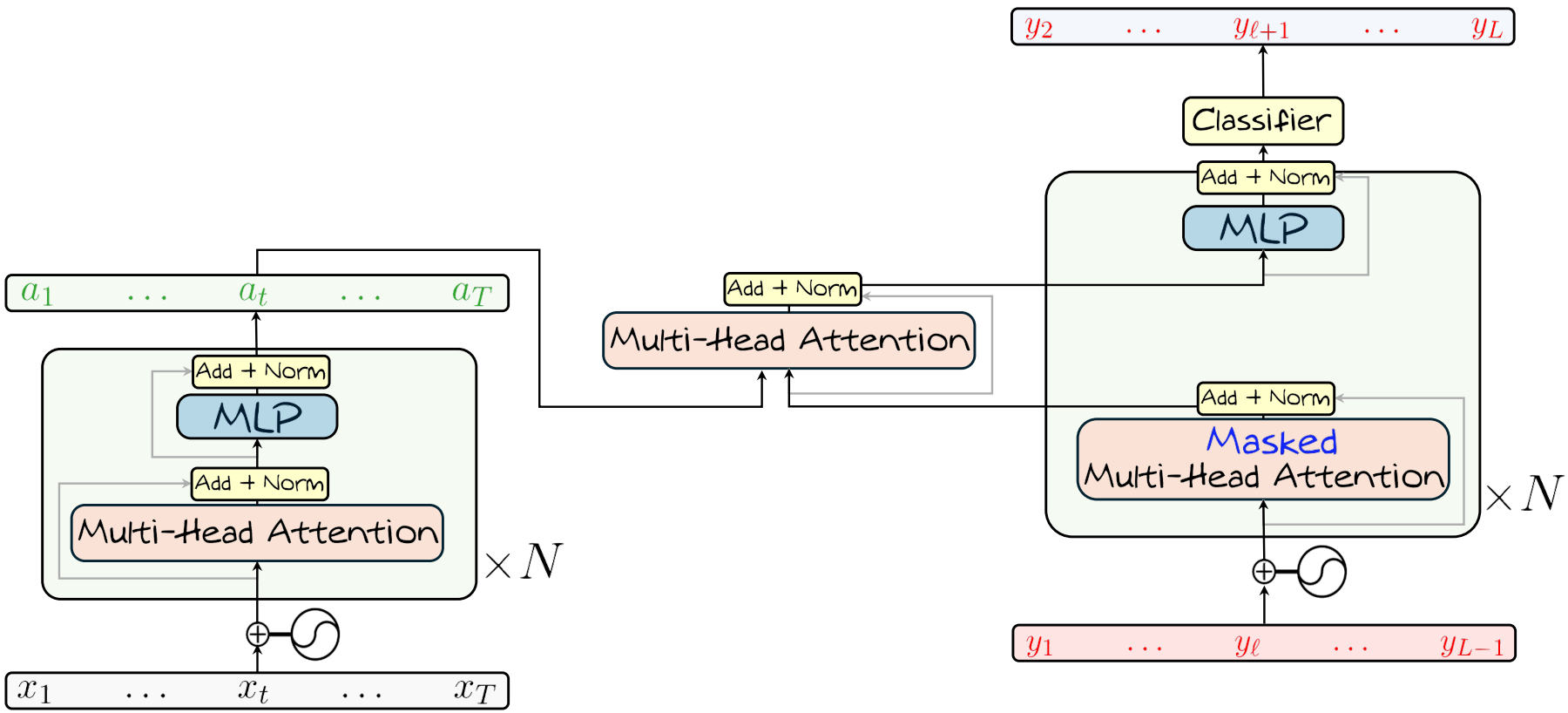
Agenda for Today



Transformer

This is a simple Transformer that maps $x_{1:T}$ to $y_{2:L}$

- *In real-world applications we use deep encoder and decoder*
- *Recall that the next token is given via a final classifier*
- *In practice we add skip connections and normalization for stable training*



Homework

1. Write pseudo-code for the full Transformer
   - ↳ *Include skip connections and normalization*
   - ↳ *Consider batch computation and think about tensor dimensions and ordering*
2. Consider the end-to-end training of the Transformer
   - ↳ *Explain how teacher forcing enables parallel computation in the decoder*
   - ↳ *Read Vaswani et al. to observe the order-of-magnitude speedup*



UNIVERSITY OF
TORONTO

Further Read

Further Read

Read More on These Topics

- Bishop & Bishop, *Deep Learning: Foundations and Concepts*
  - ↳ Chapter 12

Nice to ($\approx$ Must) Check

1. Attention Is All You Need
   - ↳ A. Vaswani et al. (2017)
2. Hands-on Large Language Models
   - ↳ J. Alammar and M. Grootendorst (2024)