

UNIVERSITY OF
TORONTO

Applied Deep Learning

Lecture 10. *Multimodal Transformers and LMs*

Ali Bereyhi
ECE Department, University of Toronto
Fall 2026

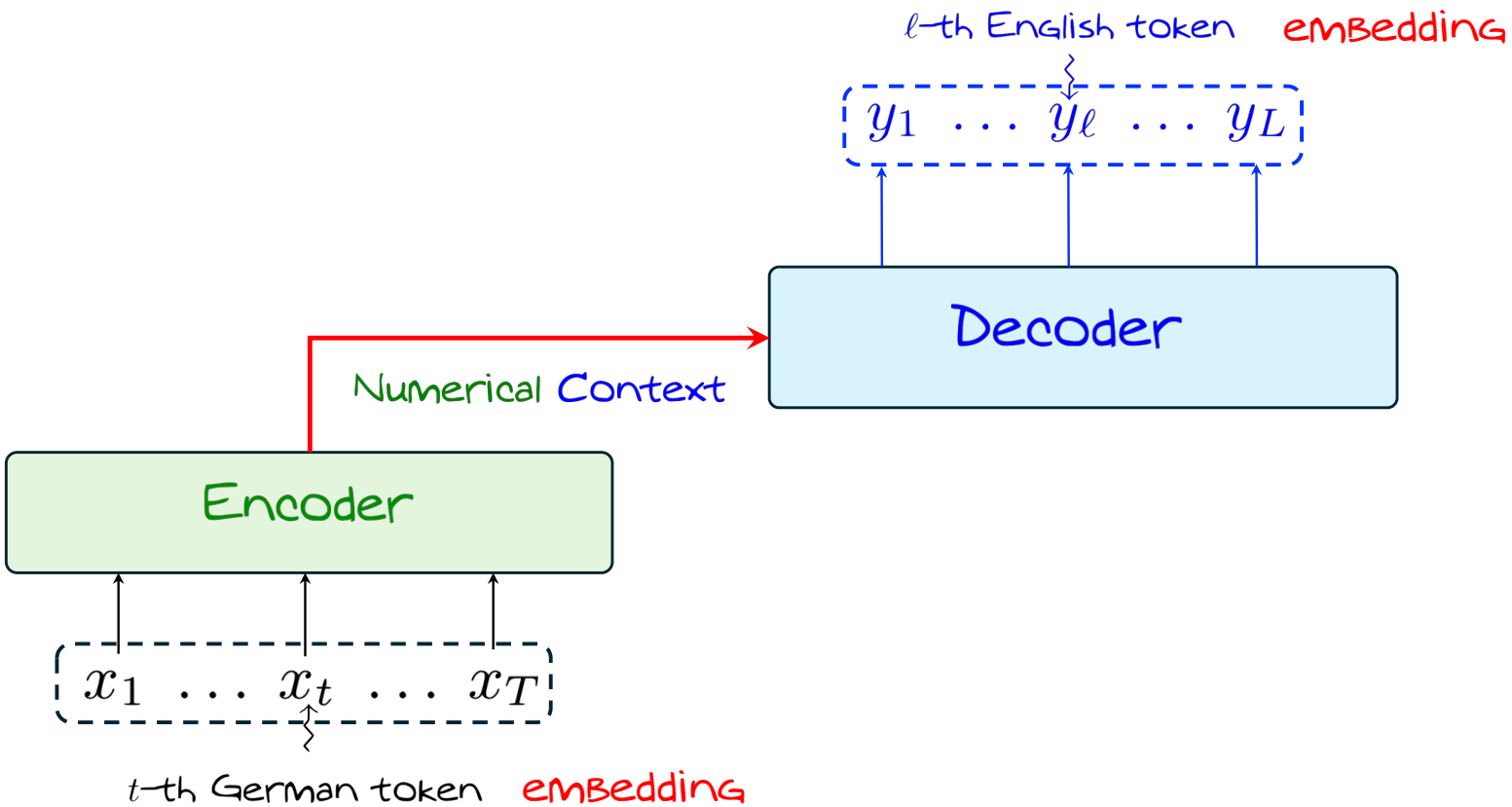


UNIVERSITY OF
TORONTO

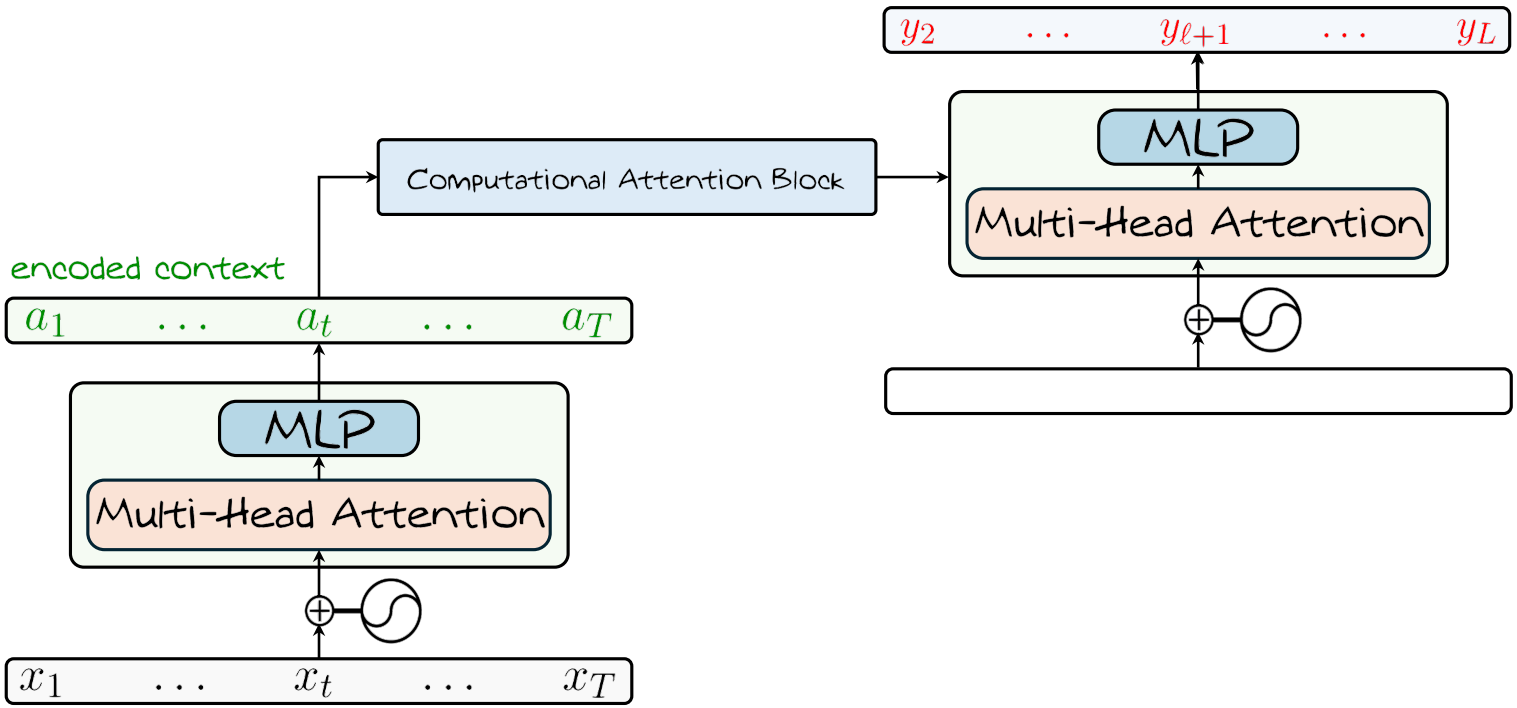
Lecture Overview

NMT: *Seq2Seq Model*

- 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 $y_{1:L}$



Transformer: *Attention-Only Seq2Seq*



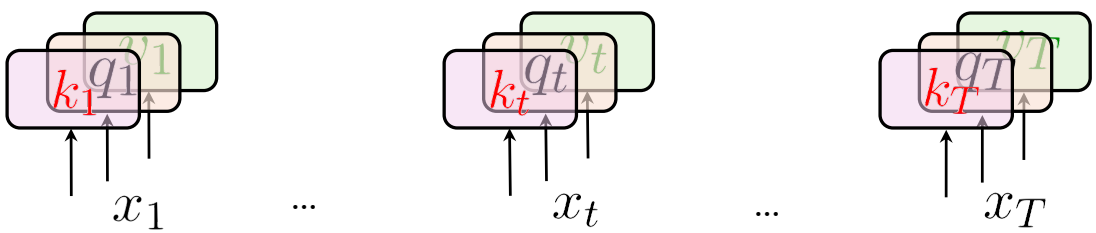
Self-Attention Mechanism

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`)`

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

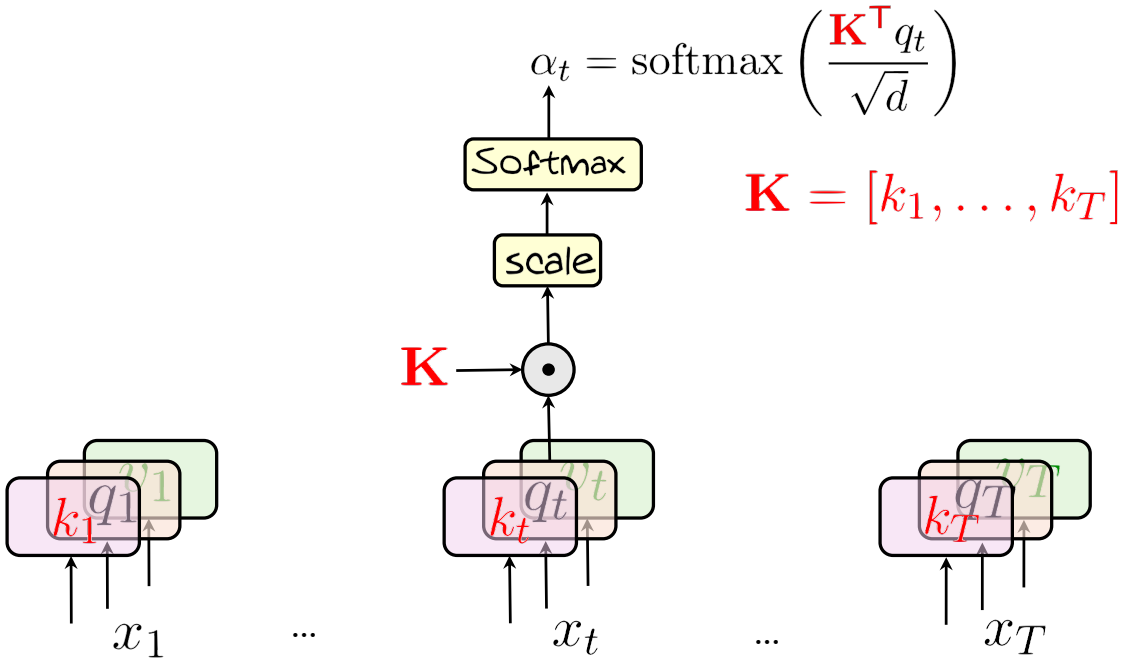
$$\overleftrightarrow{q_t}^{d} = \mathbf{W}_{\text{query}} x_t \qquad \mathbf{Q} = [q_1, \ldots, q_T]$$

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



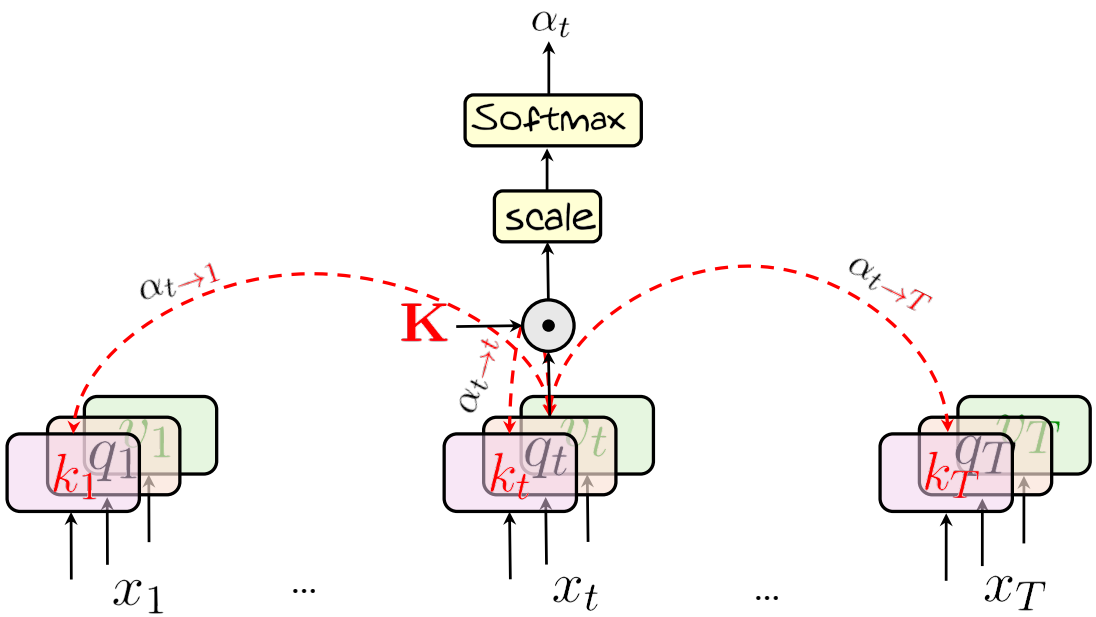
Self-Attention Mechanism

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



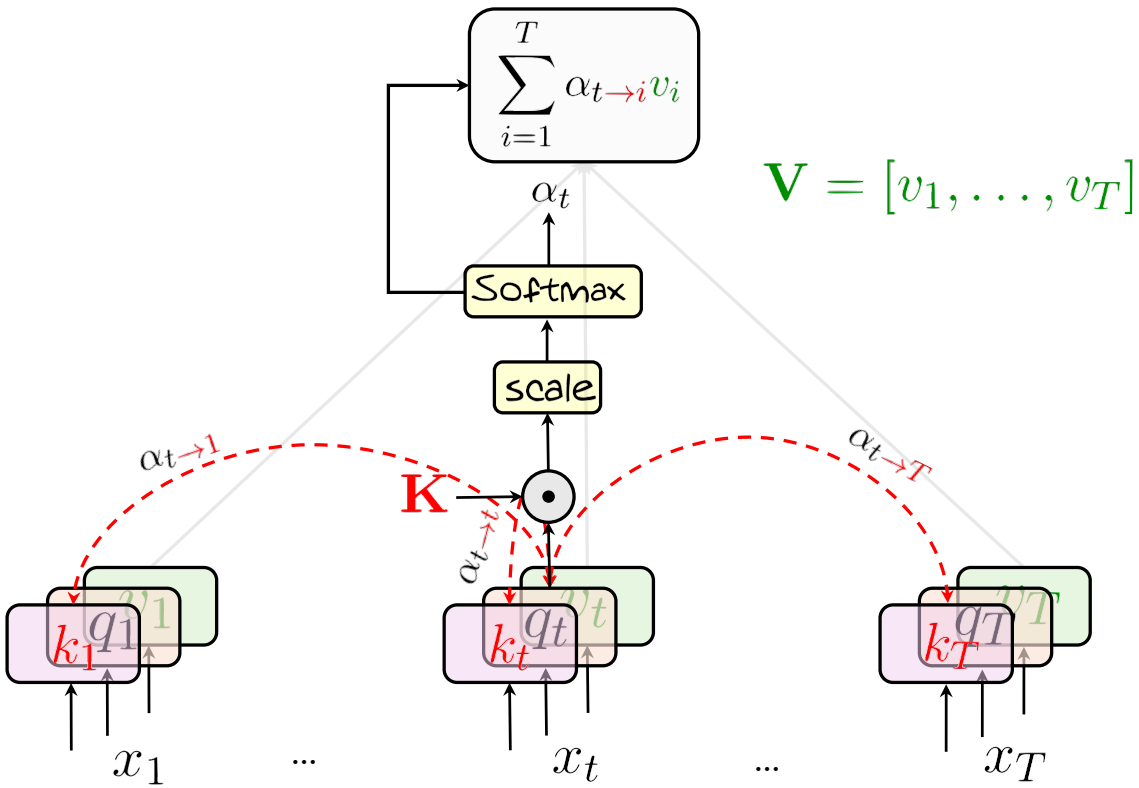
Self-Attention Mechanism

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



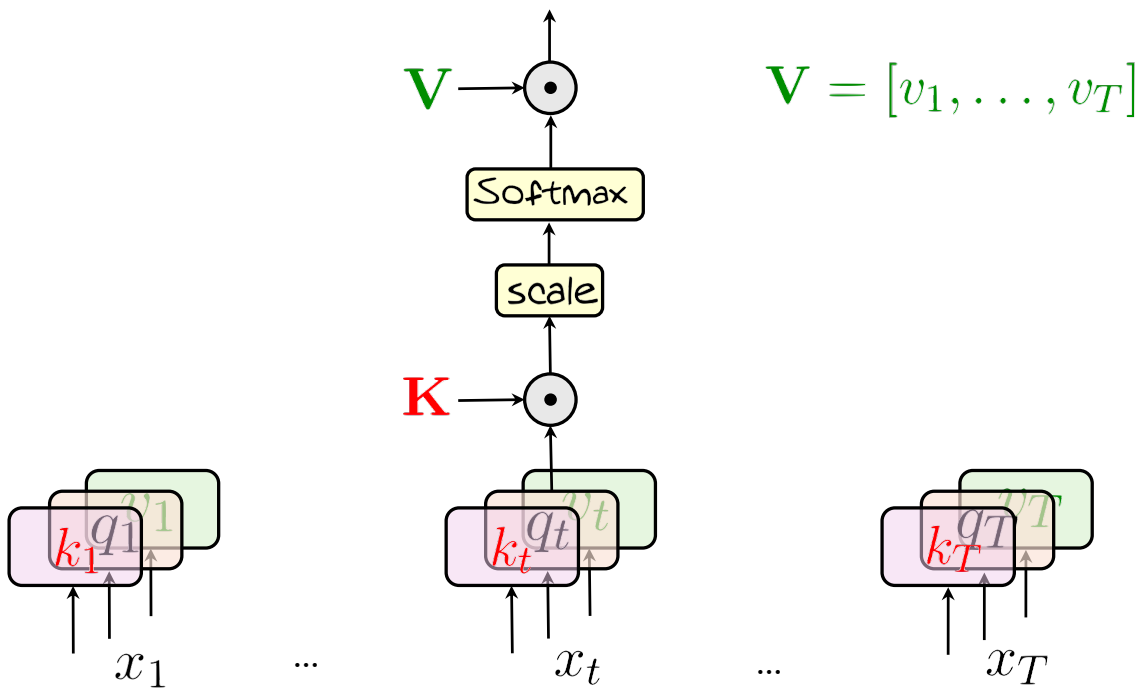
Self-Attention Mechanism

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



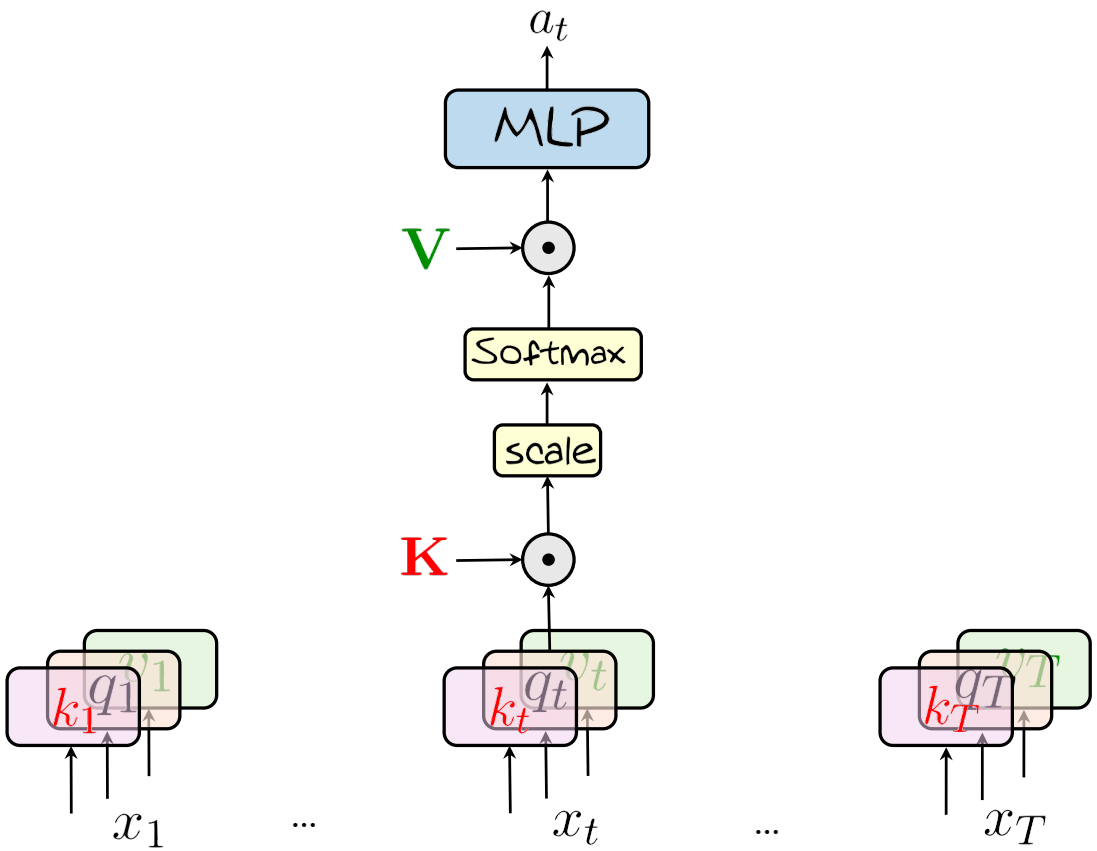
Self-Attention Mechanism

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



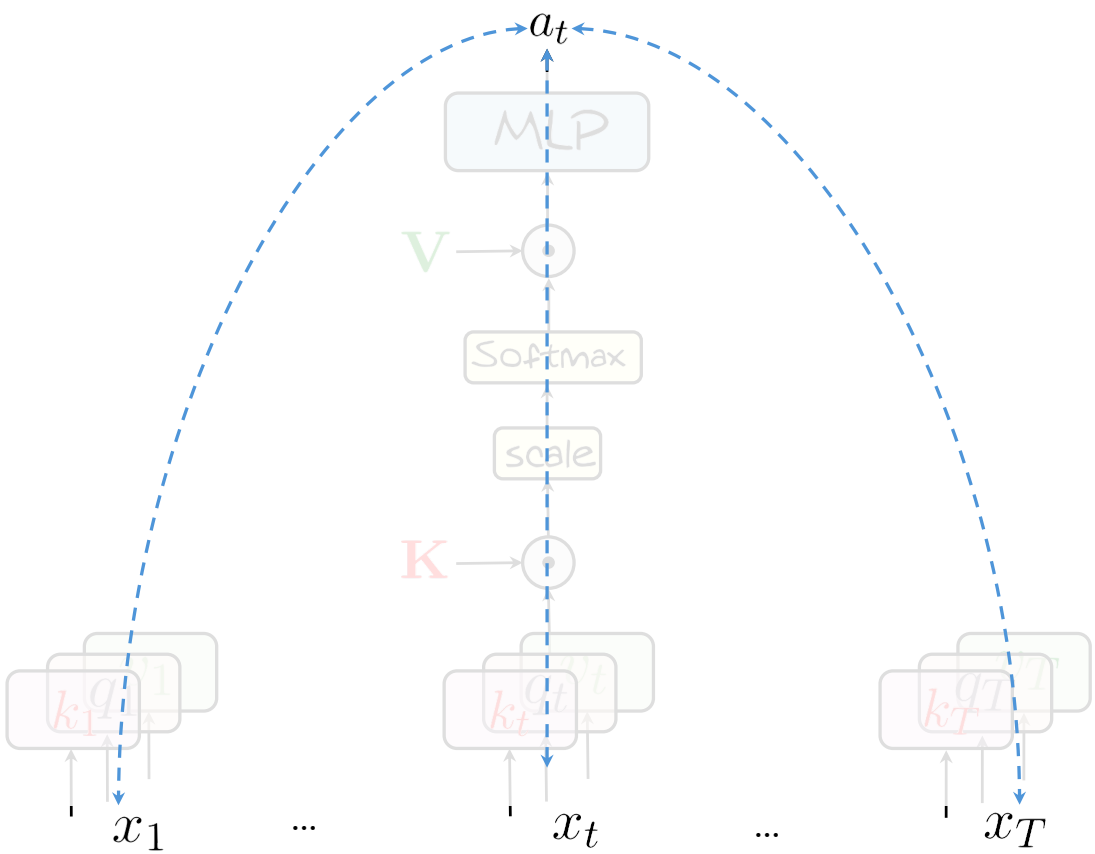
Self-Attention Mechanism

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



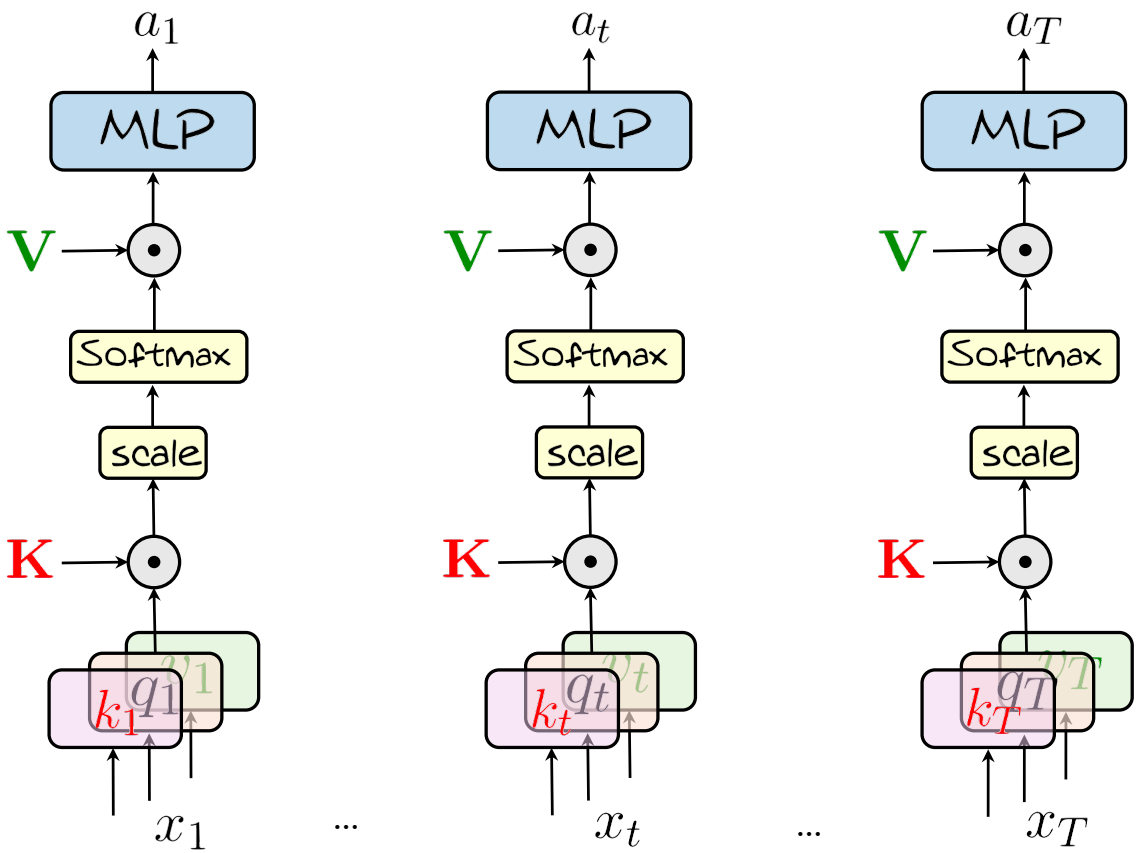
Self-Attention Mechanism

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



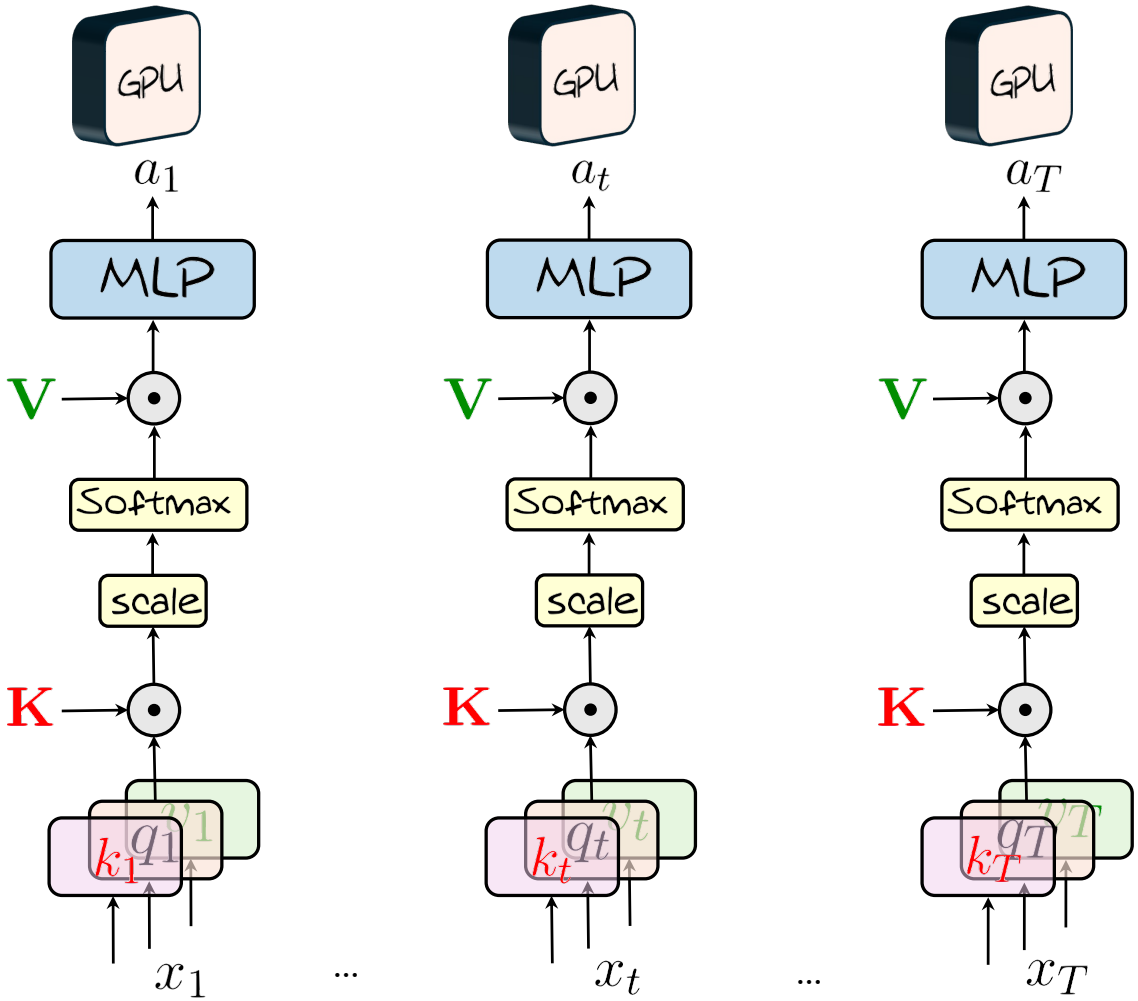
Self-Attention Mechanism

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



Self-Attention Mechanism

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

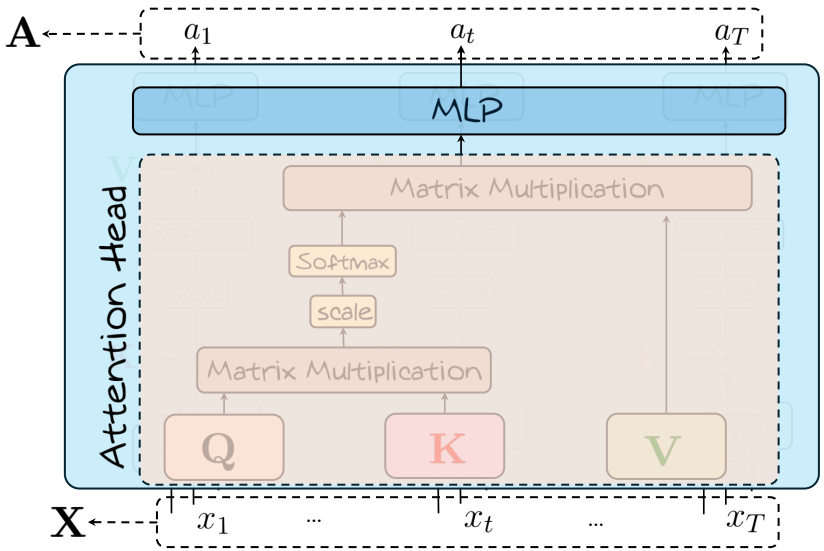


Self-Attention: *Pseudo Code*

`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 $\mathbf{V} = \mathbf{W}_{\text{val}}\left[x_{1:T}\right]$

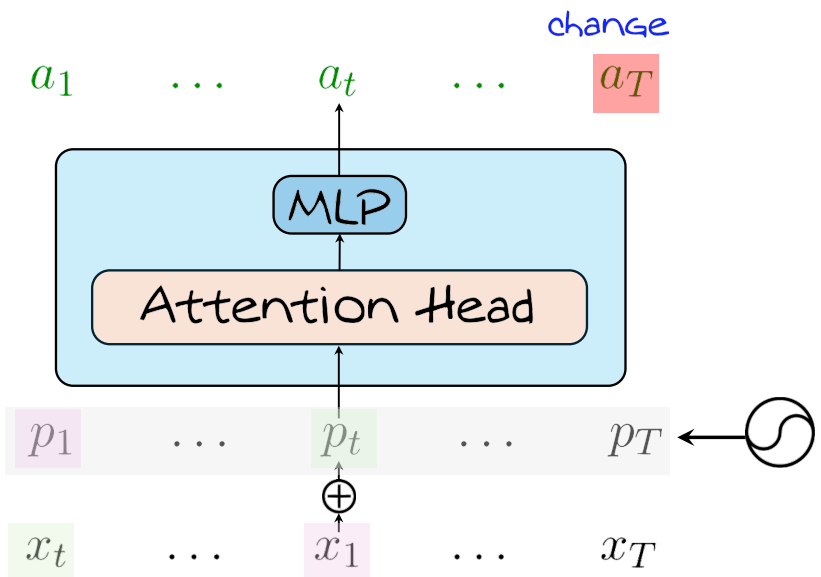
2: **return** $\bar{a}_{1:T} = \mathbf{V}\text{softmax}\left(\mathbf{K}^{\mathsf{T}}\mathbf{Q}/\sqrt{d}\right)$

Positional 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* $\texttt{PosEnc}\left(x_t\right) = x_t + p_t$



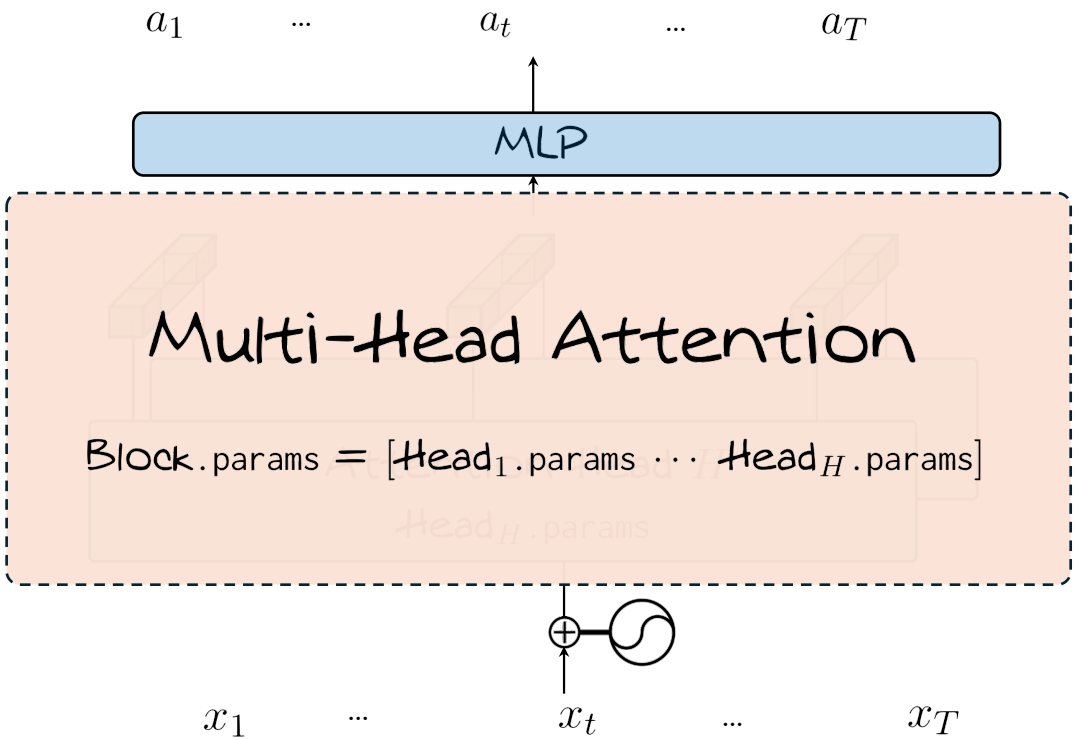
Multi-head Attention

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

Lecture 10. *Multimodal Transformers and LMs*

Today, we learn how to build any sequence-to-sequence model via Transformers

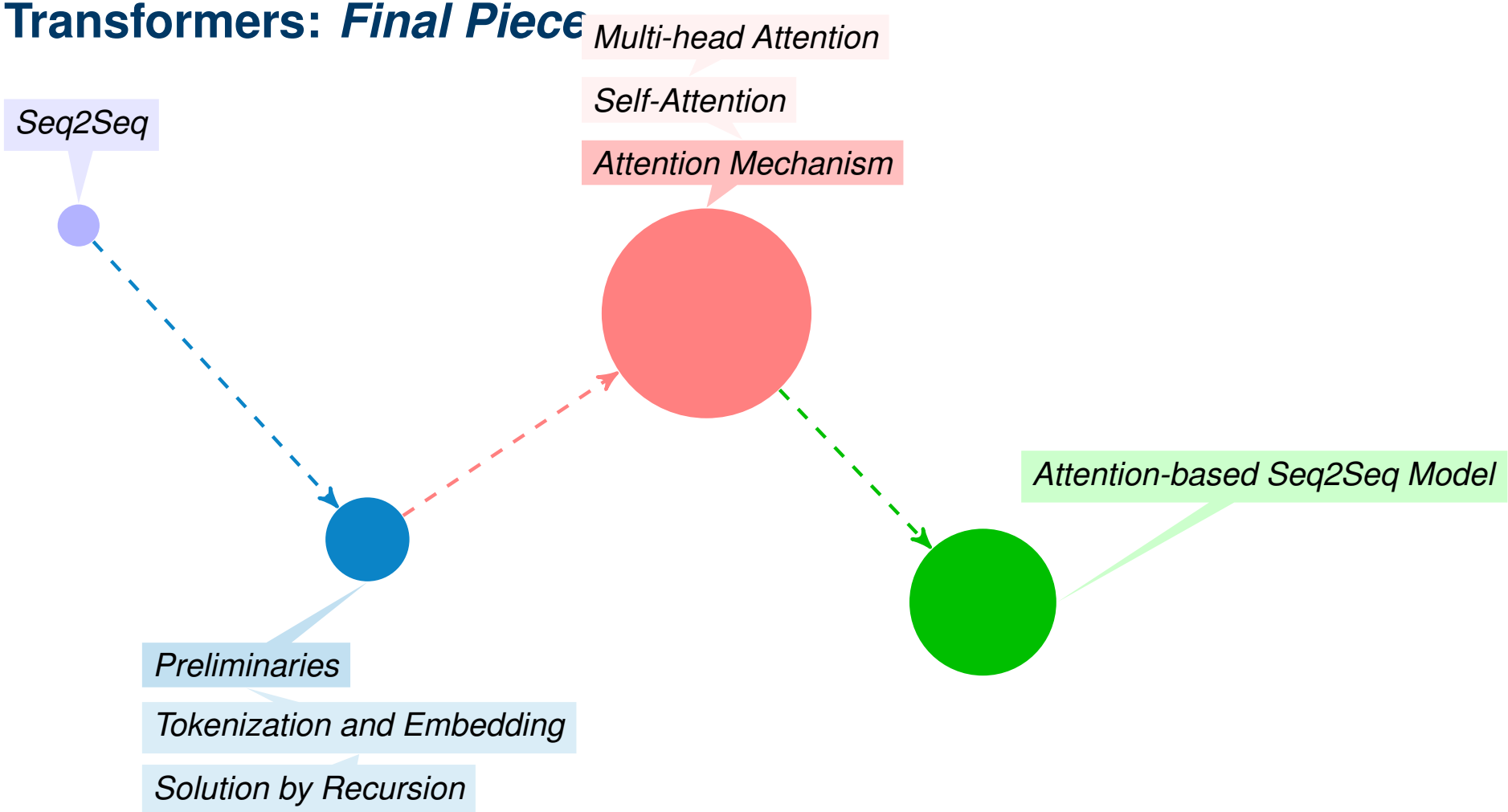
1. Transformers
   - *Cross-attention and encoding decoding via Transformers*
2. LLMs
   - *Decoder-only Transformer as LMs*
   - *Large LMs*
3. Vision Transformers
   - *Represent other modalities as sequence of tokens*
   - *General sequence-to-sequence modelling via Transformers*



UNIVERSITY OF
TORONTO

Transformers

Transformers: *Final Piece*



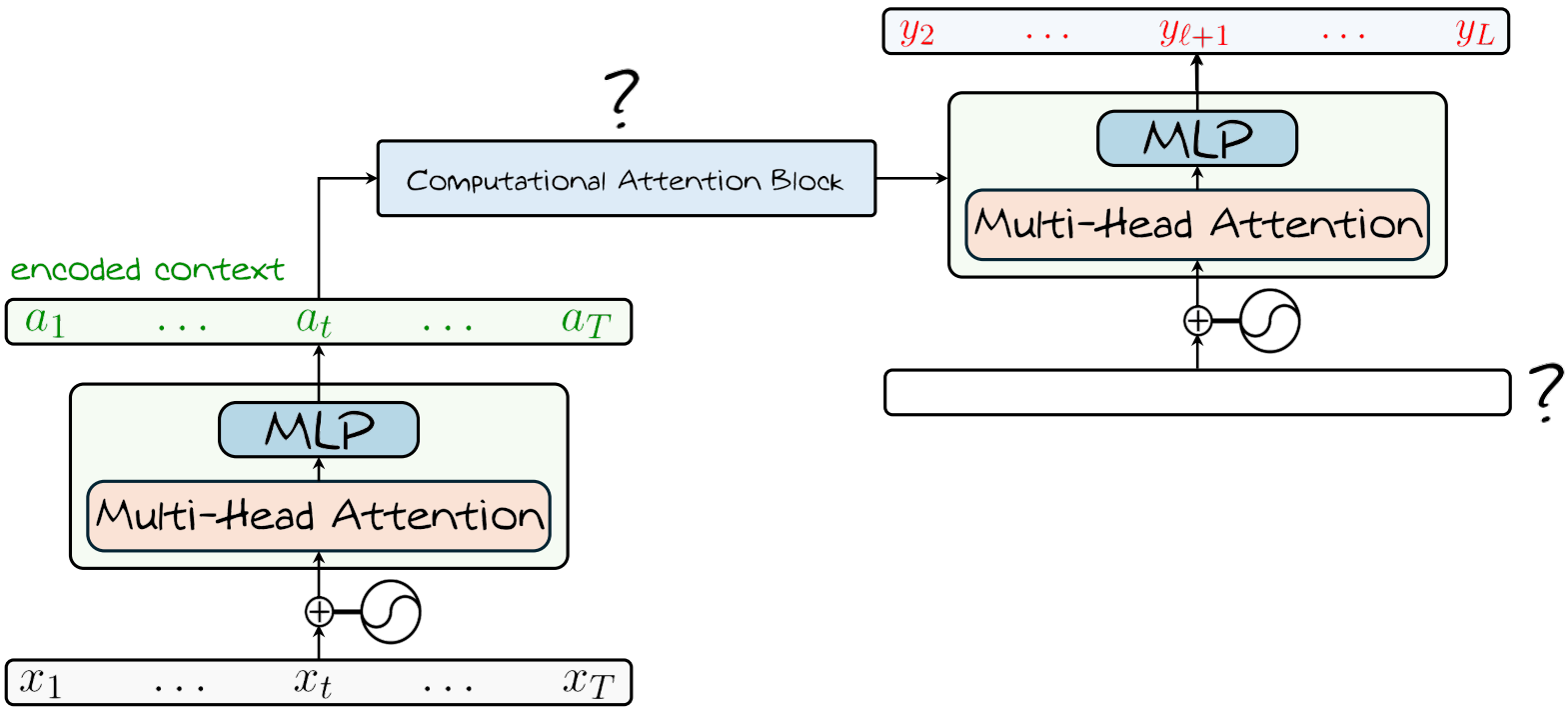
Transformer: *First Sketch*

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*

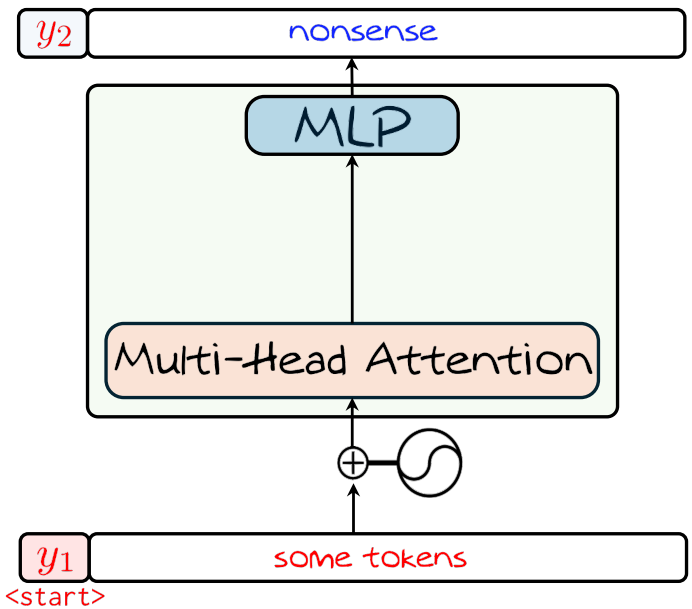
- Decoding is sequential by nature
  - *We start with $y_1 =$ <start>, then get y_2, and so on*
- We can use the attention block sequentially during inference

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*

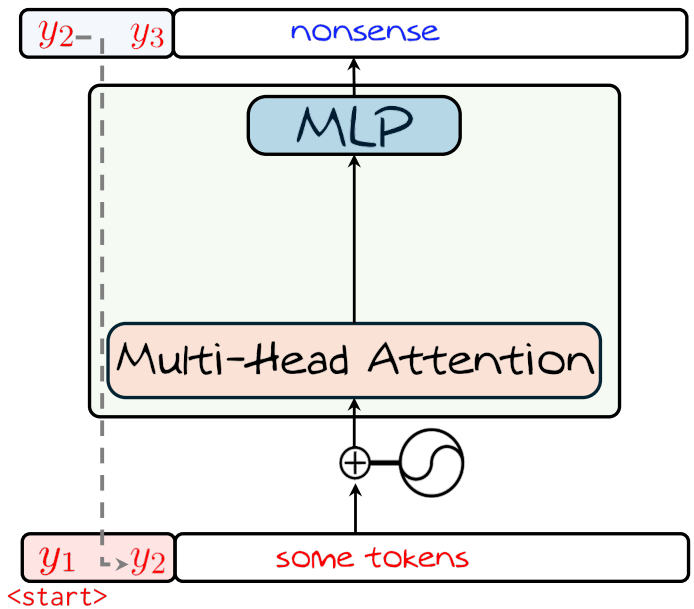
- Decoding is sequential by nature
  - ↳ *We start with $y_1 =$ <start>, then get y_2, and so on*
- We can use the attention block sequentially during inference

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*

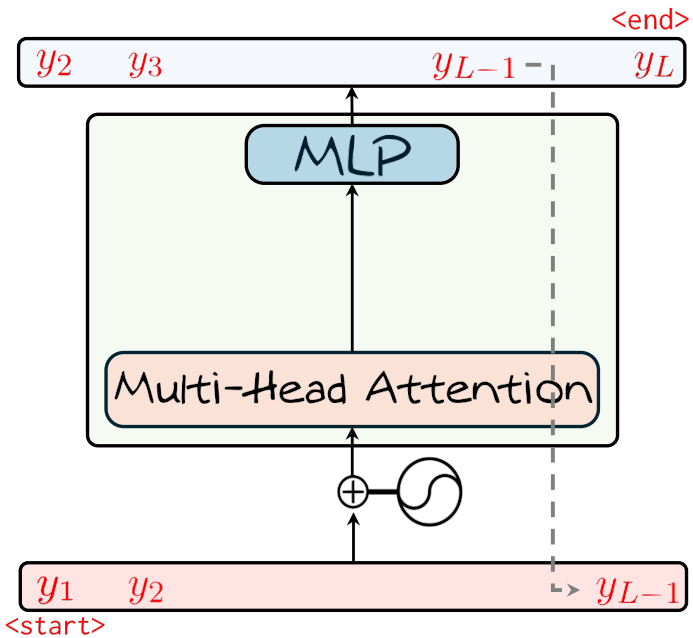
2. *Input again y_1, y_2 and fill the remaining with some tokens*

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

⋮

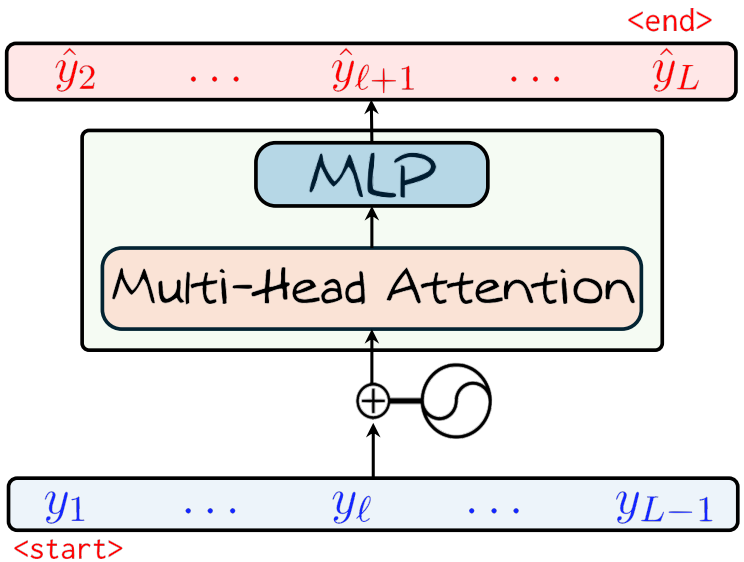
$(L-1)$. *Input $y_1, \ldots, y_{L-1}$*

↳ *Get sequence $y_2, \ldots, y_L$*



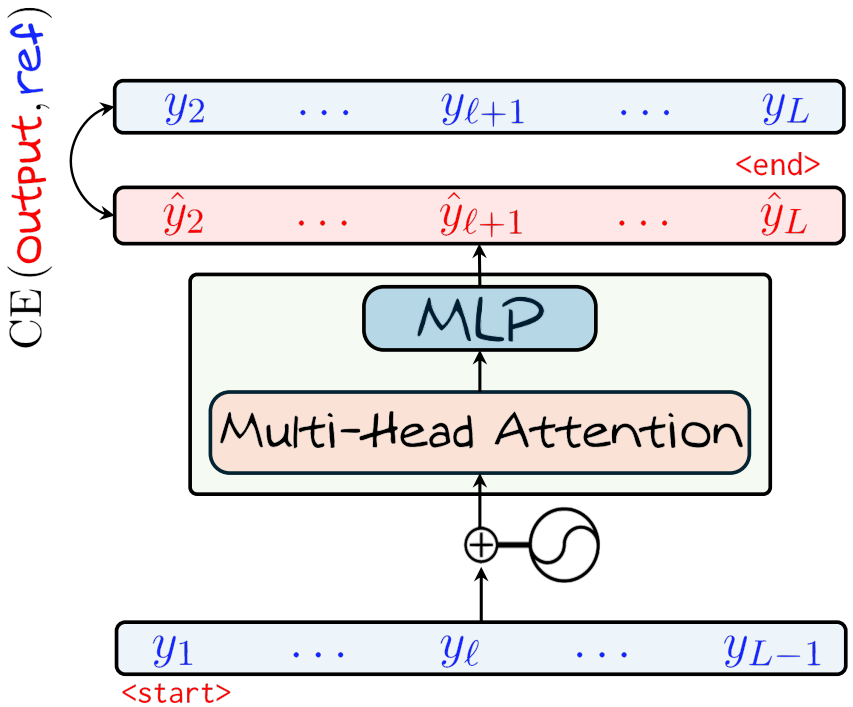
Decoding via Self-Attention: *Training*

- Unlike inference, training still benefits from parallel decoding



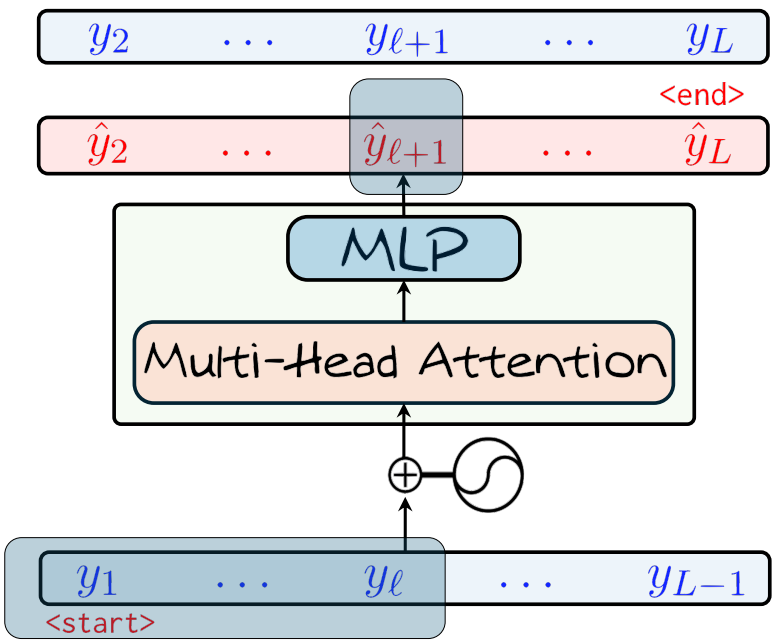
Decoding via Self-Attention: *Training*

- Unlike inference, training still benefits from parallel decoding



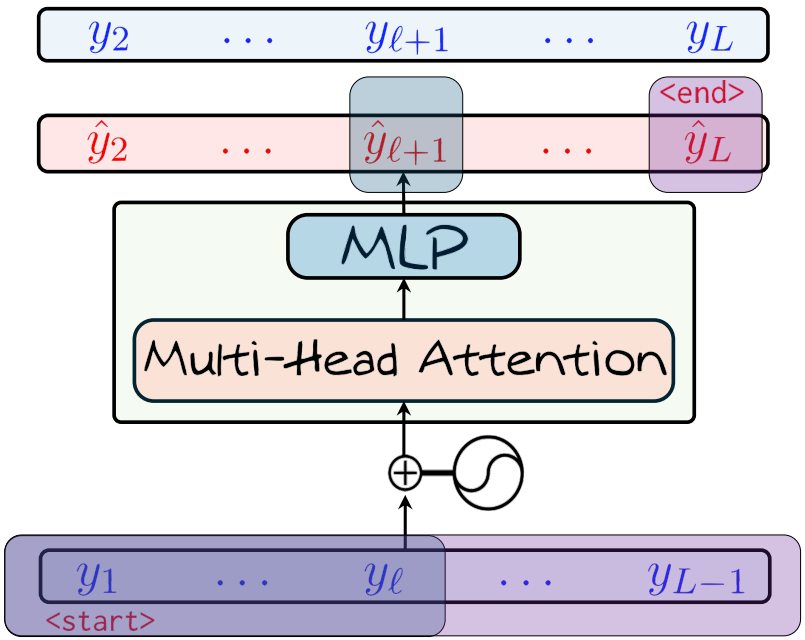
Decoding via Self-Attention: *Training*

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



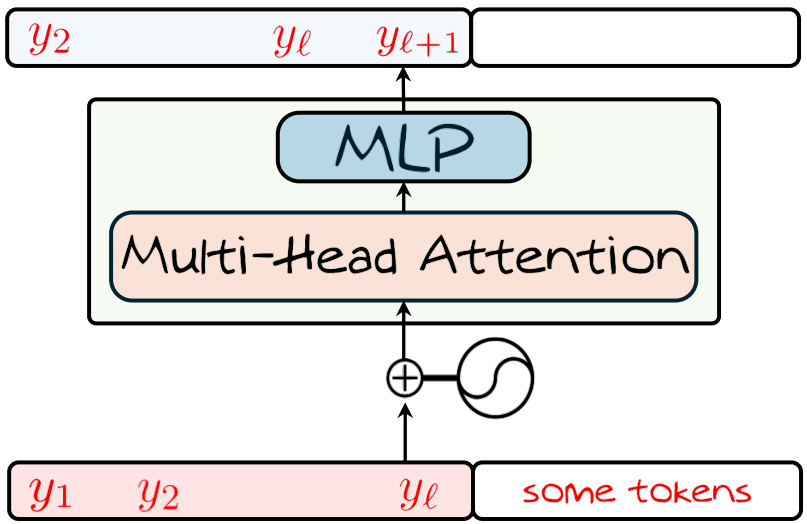
Decoding via Self-Attention: *Training*

- 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



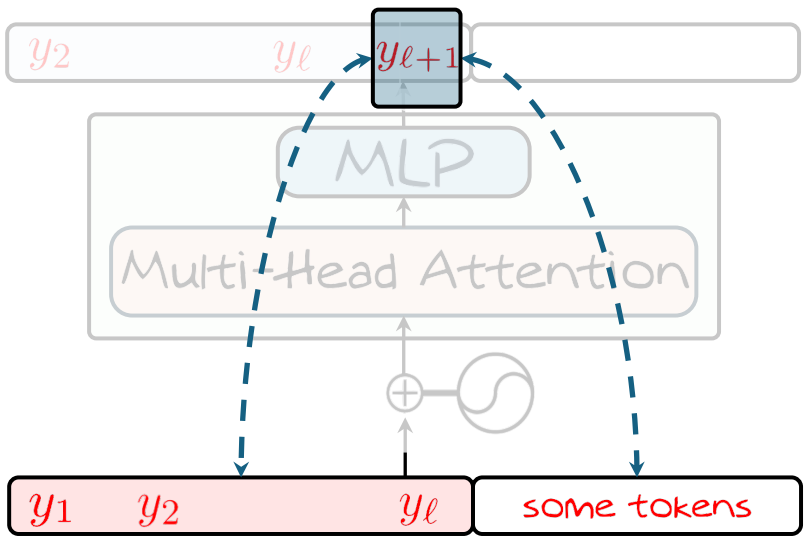
Solution: *Masked Decoding*

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

Decoding via Self-Attention: *Causality*

The decoding by self-attention is not causal

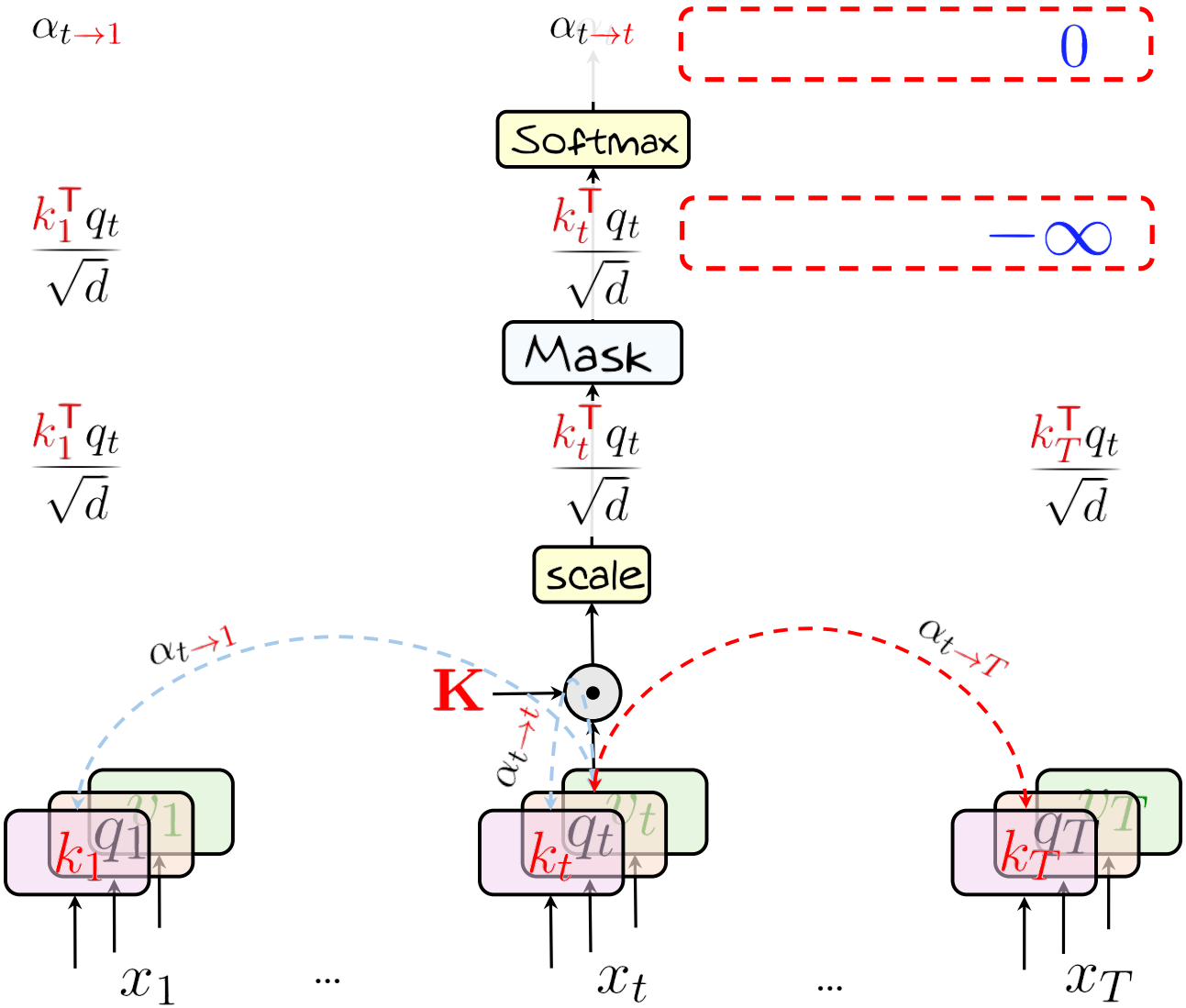
- ! At time ℓ, next token is predicted from complete input including future tokens
  - ↳ *Training* ⇝ *Next token is already revealed at the input!*



Solution: *Masked Decoding*

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

Masked Decoding



Masked Decoding: *Formulation*

Let's look at the array of dot products

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

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`

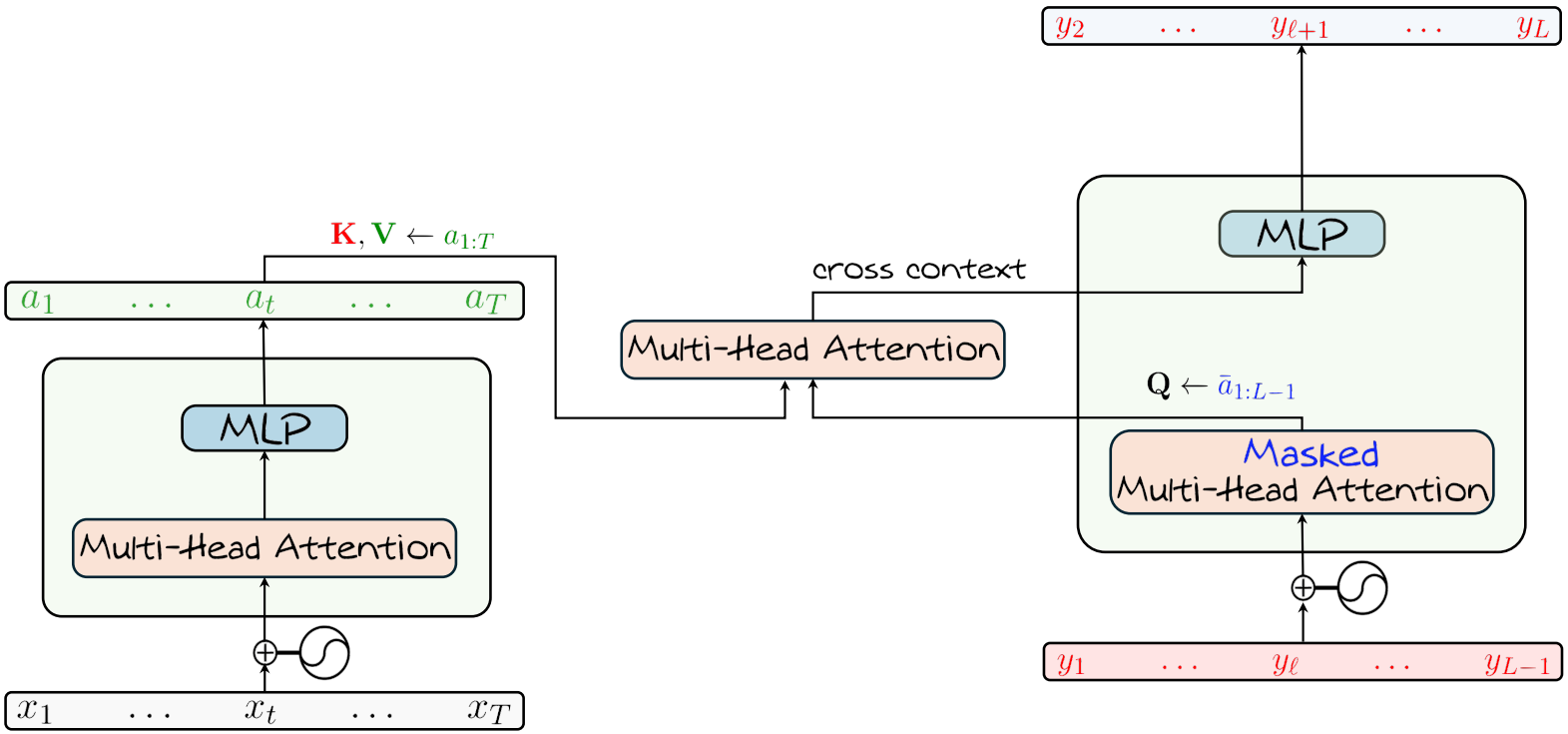
4: **end for**

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

The multi-head cross attention simply uses multiple heads

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

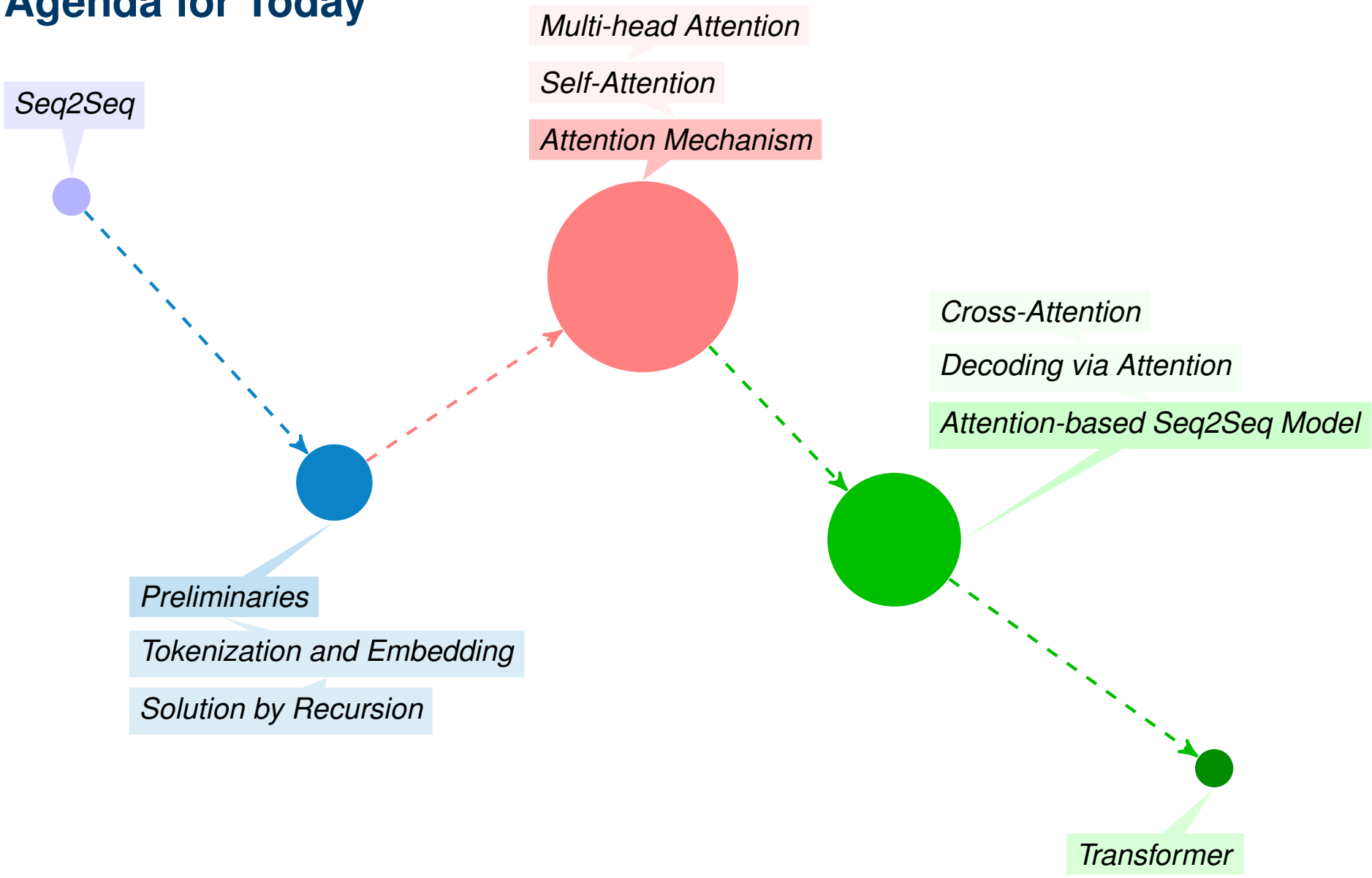
1: **for** $h = 1 : H$ **do**

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 $\ell = 1 : L-1$

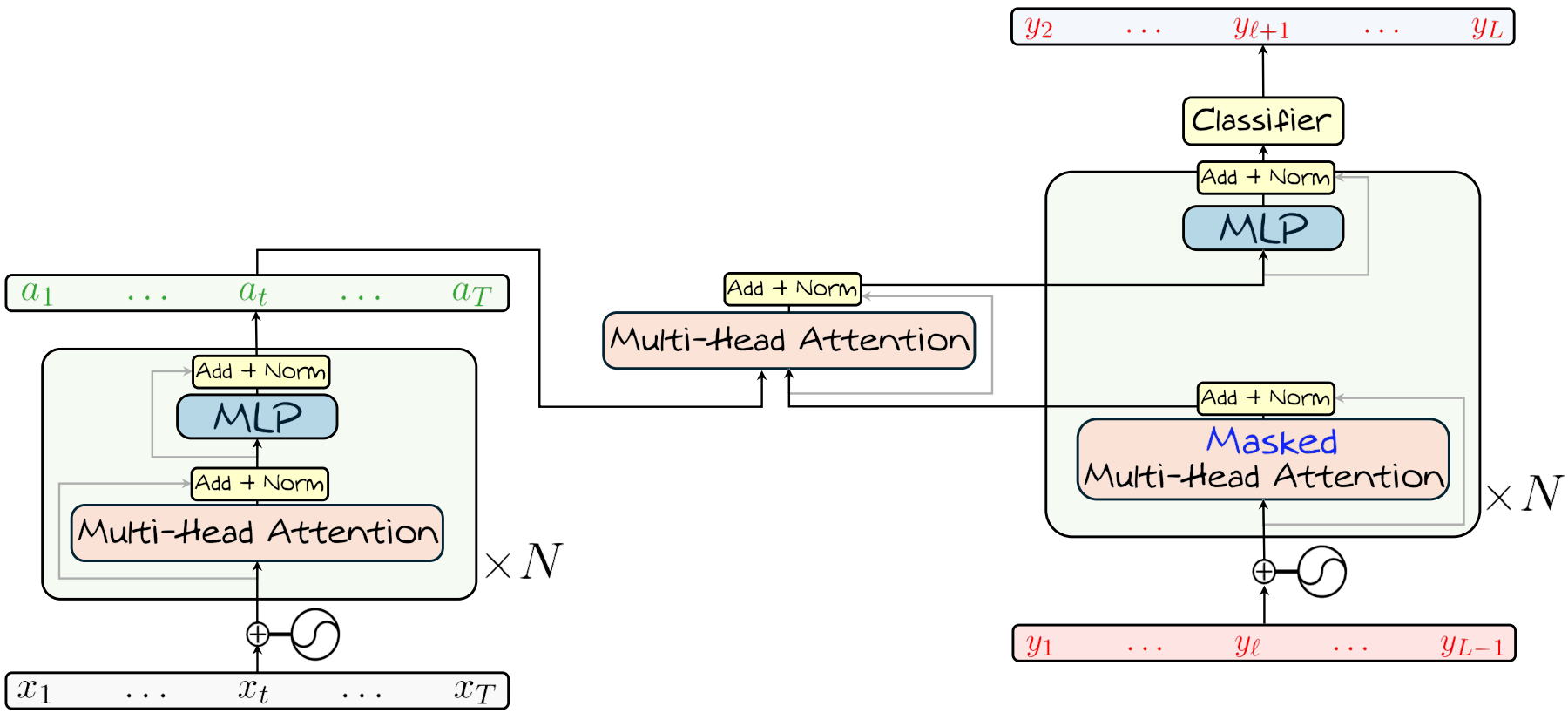
Agenda for Today



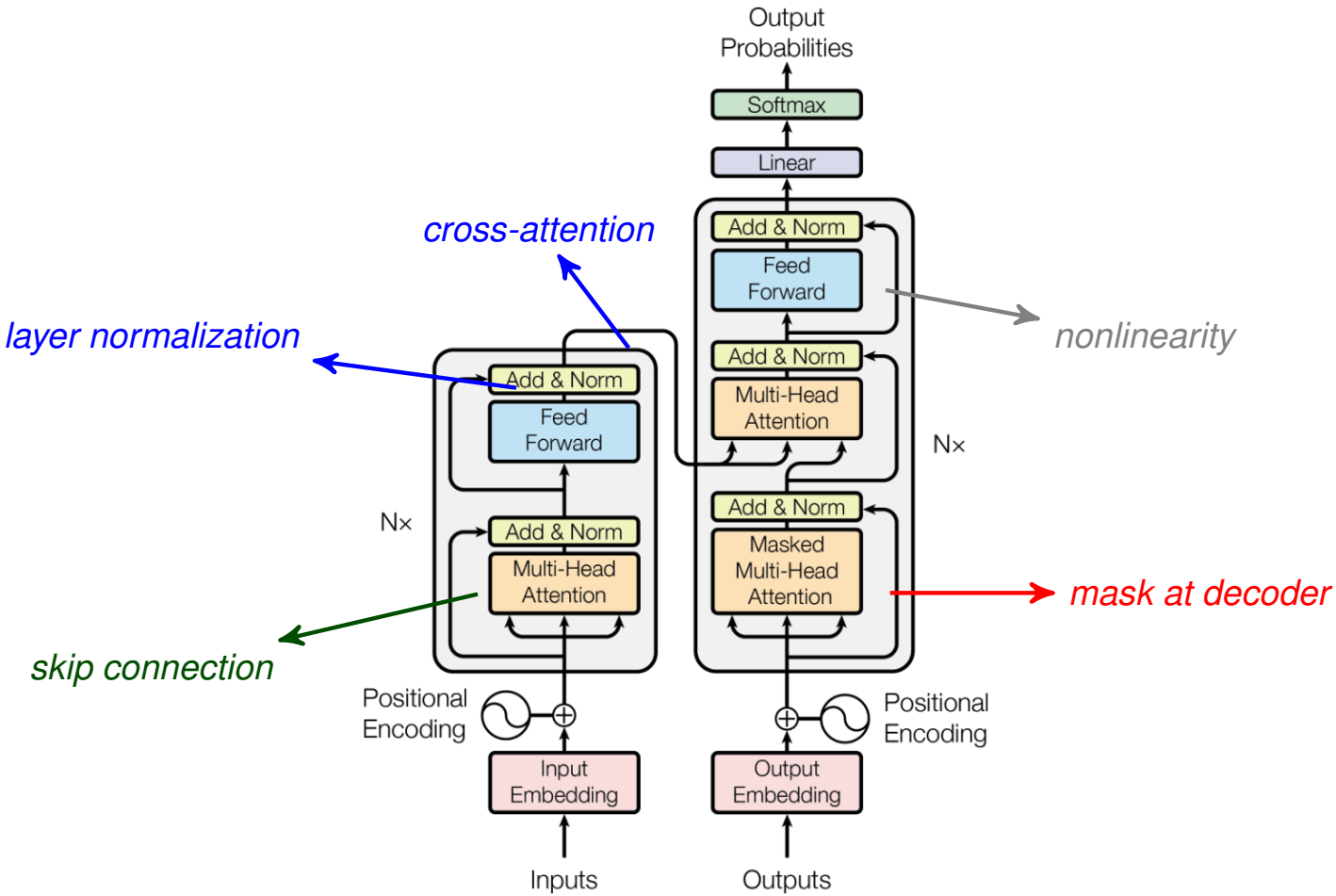
Transformer

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*



A Classic Transformer



Why Transformers?

+ *Transformers seem to be more challenging to implement!*
– That's right! But they come with some benefits!

In general transformers

- *process sequences in parallel ⇝ room for parallel computing*
- *preserve much better long-term connections*
- *can be deepened much more straightforward*

But, it's true that

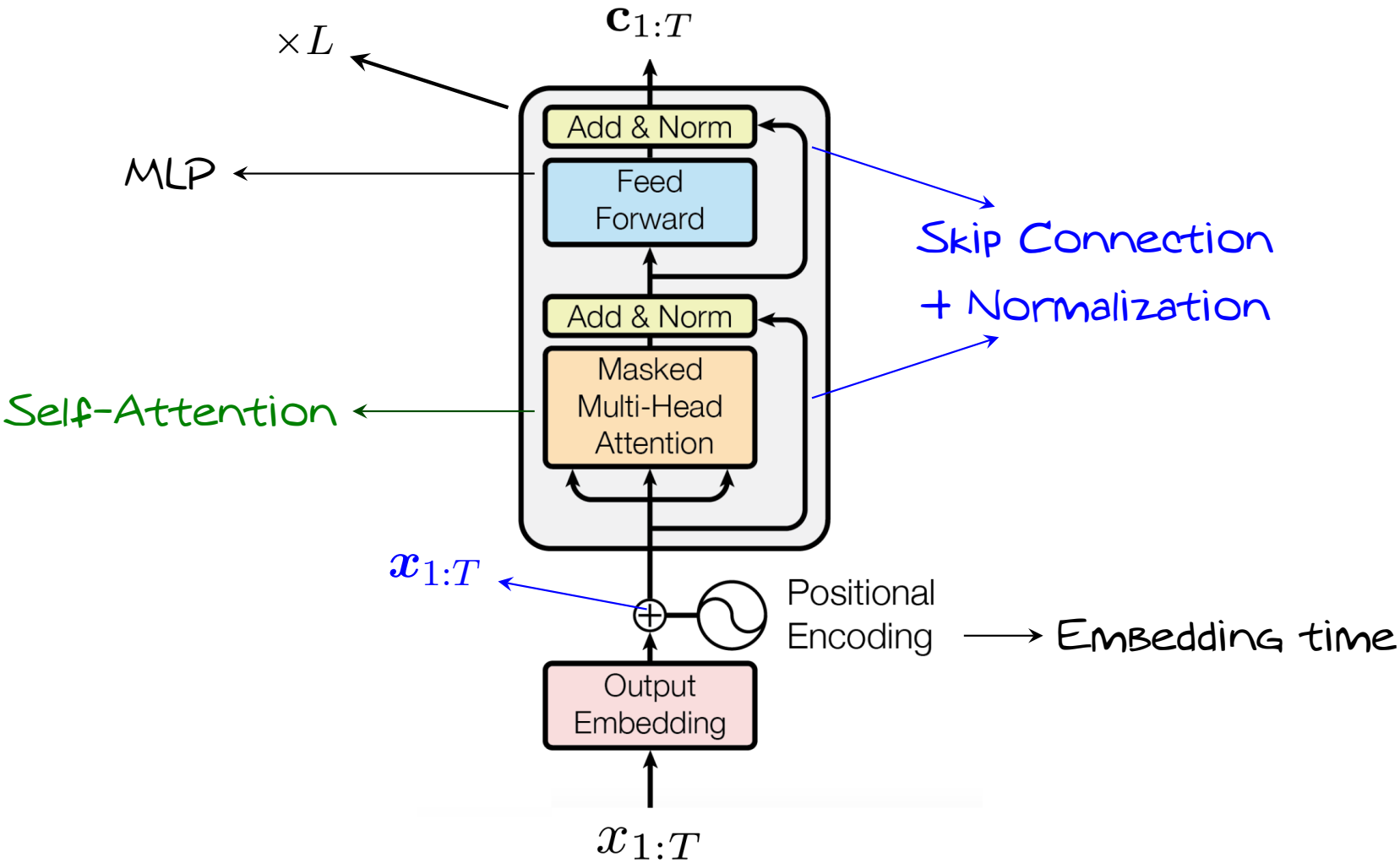
- *they pose more complexity ⇝ we can handle it nowadays*



UNIVERSITY OF
TORONTO

LMs: *Decoder-only Architectures*

Transformer-Based LM: *Overview*



Transformer-based LM: *Context Extractor*

- ? *How does the classification head look like?*
- We can add a final MLP with Softmax

`TransformerLM(`$x_{1:T}$`,` H`:# of heads,` L`:depth):`

1: $\mathbf{c}_{1:T} \leftarrow$ `TransformerDec(`$x_{1:T}, H, L$`)`
2: **for** $t = 1 : T$ **do**
3: $\quad \mathbf{p}_{t+1} \leftarrow$ `MLP(`$\mathbf{c}_t$`)` $\qquad \#\mathbf{p}_{t+1} \in [0, 1]^I$
4: **end for**
5: **return** $\mathbf{p}_{t+1}$ for $t = 1 : T$

Training Transformer-based LM

It's good to imagine how we can train this model

`TransformerLM_Train(`$\mathbb{D}$`:train_set):`

1: **for** *multiple epochs* **do**
2: Sample a batch $\mathbb{B} = \{(x_{1:T+1})_b\}$ `#samples of` T `tokens`
3: **for** $b = 1 : B$ **do**
4: $\mathbf{p}_{2:T+1,b} \leftarrow$ `TransformerLM(`$x_{1:T,b}, H, L$`)`
5: Compute $\nabla_b \leftarrow \sum_t \nabla \mathrm{CE}(\mathbf{p}_{t+1,b}; x_{t+1,b})$ `#backpropagation`
6: **end for**
7: Update $\mathbf{w} \leftarrow \mathbf{w} - \eta$`mean`$\{\nabla_b\}$ `#average batch`
8: **end for**

Important

Training loop in this case can be extensively parallelized!

Generation via Transformer-based LMs

It's good to think how a this LM generates the whole new text

`TransformerLM_Generation(`$x_{1:t}$`:input_text):`

1: **for** $i = t : T-1$ **do**
2: $\quad x_{1:T+1} = x_{1:i}, 0, \ldots, 0$ `#add` $\varnothing : 0$
3: $\quad \mathbf{p}_{2:T+1} \leftarrow$ `TransformerLM(`$x_{1:T-1}, H, L$`)` `#masked decode`
4: $\quad x_{i+1} \leftarrow$ `sample(`$\mathbf{p}_{i+1}$`, #smpl=1)` `#sample next token`
5: **end for**
6: **return** $\{$`token`$[x_t]$ *for* $t = 1 : T\}$ `#completed text`

Large LMs

LLMs are deep Transformer LMs, trained on massive data

LLMs: *Example of GPT*

For instance the generative pre-trained model (GPT) has evolved as

1. *GPT-1 had $L = 12$ layers of multi-head SA $\rightsquigarrow \approx$ 117 million parameters*
2. *GPT-2 got $L = 48$ layers $\rightsquigarrow \approx$ 1.5 billions parameters*
3. *GPT-3 got $L = 96$ layers $\rightsquigarrow \approx$ 175 billions parameters*
4. *GPT-4+ $\rightsquigarrow$?*

The larger they got, the more concerns raised $\rightsquigarrow$ publicized details got reduced

	Model	**Data**	**Code**
GPT-1	✔	✔	✔
GPT-2	✔	✔	✔
GPT-3	✔	⚠	✖
GPT-4+	✖	✖	✖

Data for LLMs

The most common resources to collect data for training LLMs are

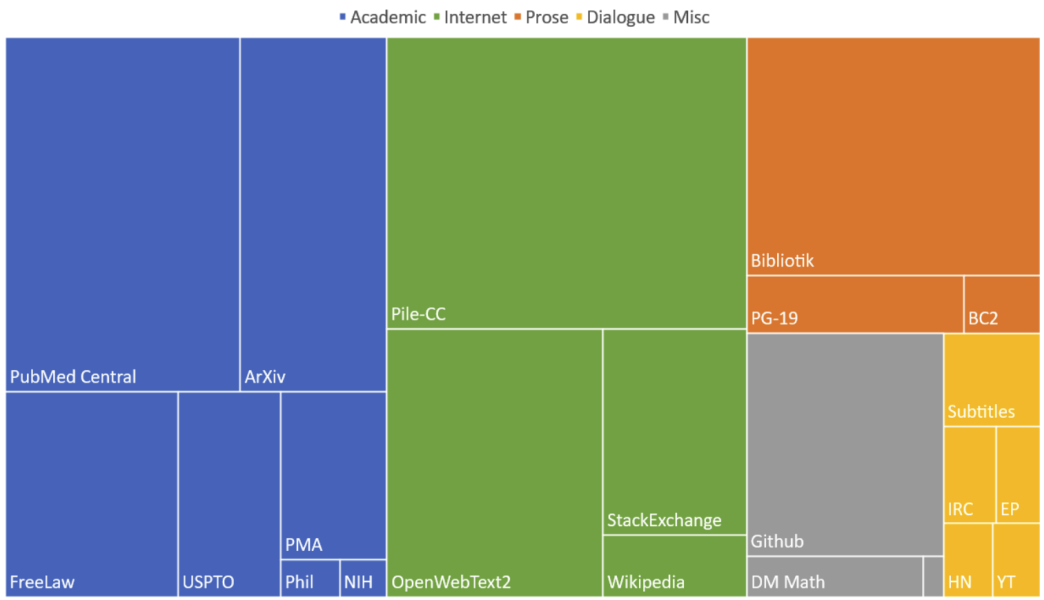
- ***Web-Text*** *which contains high quality texts from internet*
  - ↳ *Sources like Reddit and Quora*
- ***Wikipedia*** *articles*
- ***Coding-Corpora*** *collected from various online resources*
  - ↳ *Sources like GitHub and StackOverflow*
- ***Book-Corpora*** *which is made by including texts from reliable books*
- ***Common Crawl*** *that is a service providing web crawling*
  - ↳ *Crawls need to be filtered and cleaned*
  - ↳ *They need to be filtered due to safety and ethical issues*

Data: *Practical Aspects*

Getting access to fair data is crucial in training LLMs

The Pile

The Pile is a large public dataset for LLM training



Data: *Ethical Challenges*

⚠ Attention

Data collection for LLM training faces various legal and ethical challenges

- ! *We should take care of individuals' privacy*
- ! *We must filter raw data to ensure training on ethically sound content*
- ? *How can we make sure about safety aspects of trained model*
- ! *We should make sure there is no copyright violations*
- ? *How can we guarantee rights of intellectual properties*

And believe it or not . . .

there are examples for each of these aspects[1]

[1] Just take a look at further reads on course page

Pre- and Post-Training

? *So, you mean that by training these giant models for a long time, they will start to do what ChatGPT does?*

! Well, not immediately! We need a few further steps

The trained LLMs are indeed only text completer

> How can I get rid of Ali Bereyhi ⇜ $x_{1:t}$
>
> ... who works at the ECE department in UofT and is kind of annoying, especially in the Applied Deep Learning course where he talks about backpropagation for ever! ⇜ $x_{t+1:T}$

This is why we call these models pre-trained models

Pre-training

Pre-training

The process of training a LM on a large language dataset is called pre-training

A pre-trained LM is able to sample from the *generic language distribution*

$$p\left(x_{t+1:T} \middle| x_{1:t}\right)$$

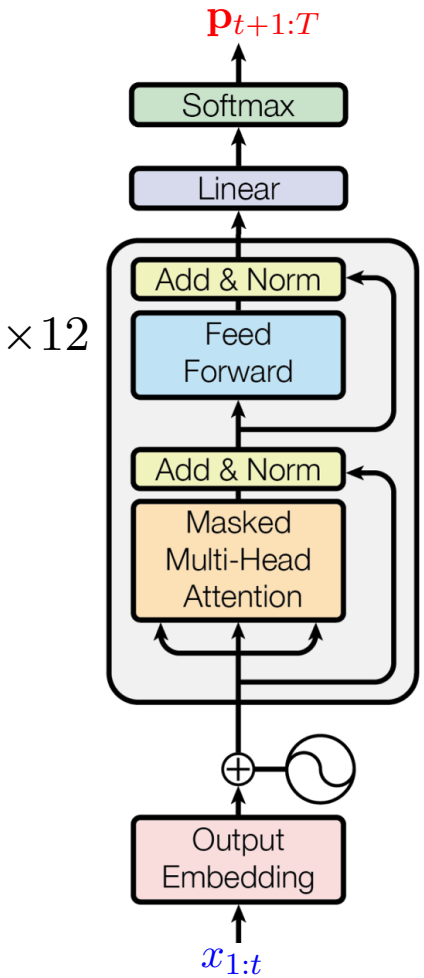
But, we often want them to sample from a *task-specific distribution*

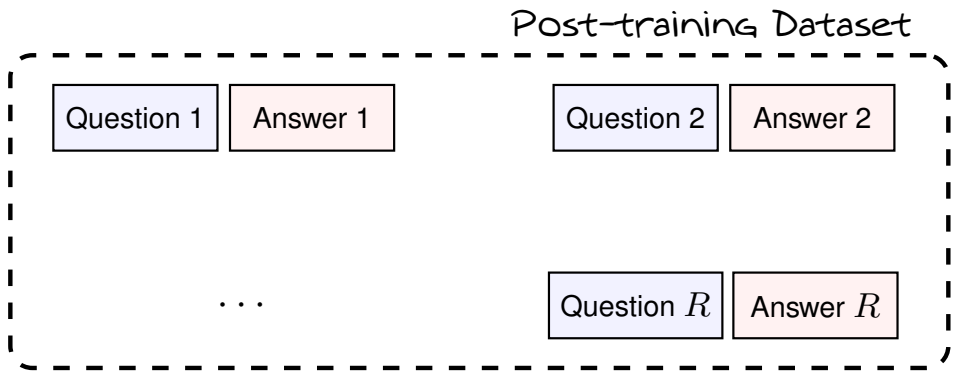
$$p\left(y_{1:L} \middle| x_{1:t}\right)$$

Example: *To learn to give answer to a question*

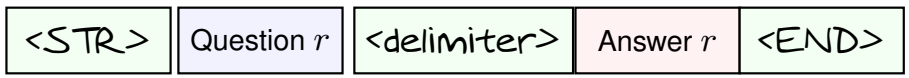
How can I get rid of Ali Bereyhi	$x_{1:t}$
Just ask questions!	$y_{1:L}$

Fine-Tuning *GPT-1 for Q&A*





Post-training Samples:



<STR> starting tokens

<delimiter> delimiter tokens indicating end of question

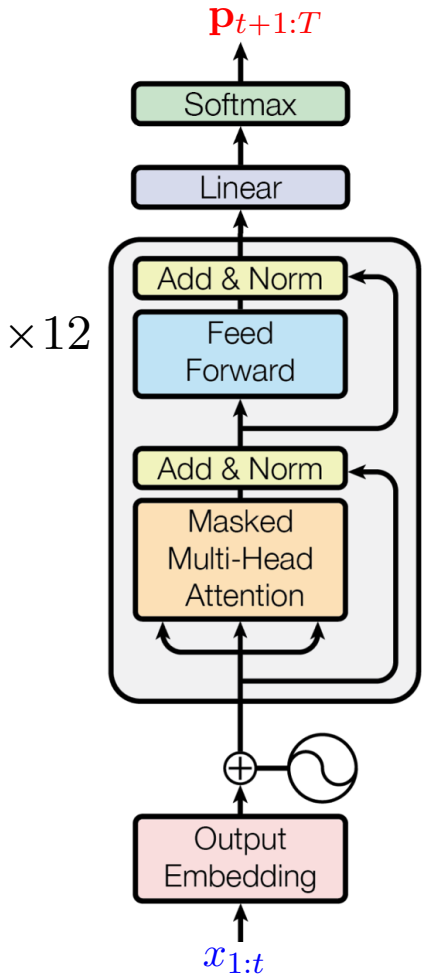
<END> end-of-sample tokens

Fine-Tuning *GPT-1 for Q&A*

Staring from the pre-trained GPT-1, we train the model for a few epochs on the post-training dataset made out of sample questions and answers

- *Post-training data size and duration are significantly less than pre-training*
  - ↳ *For Question Answering, GPT-1 used RACE dataset with 100K questions*
  - ↳ *The fine-tuning took only 3 epochs*

Sampling *Answers* from Fine-Tuned *GPT-1*



Question ⋆ | ??

Set $x_{1:t}$ to

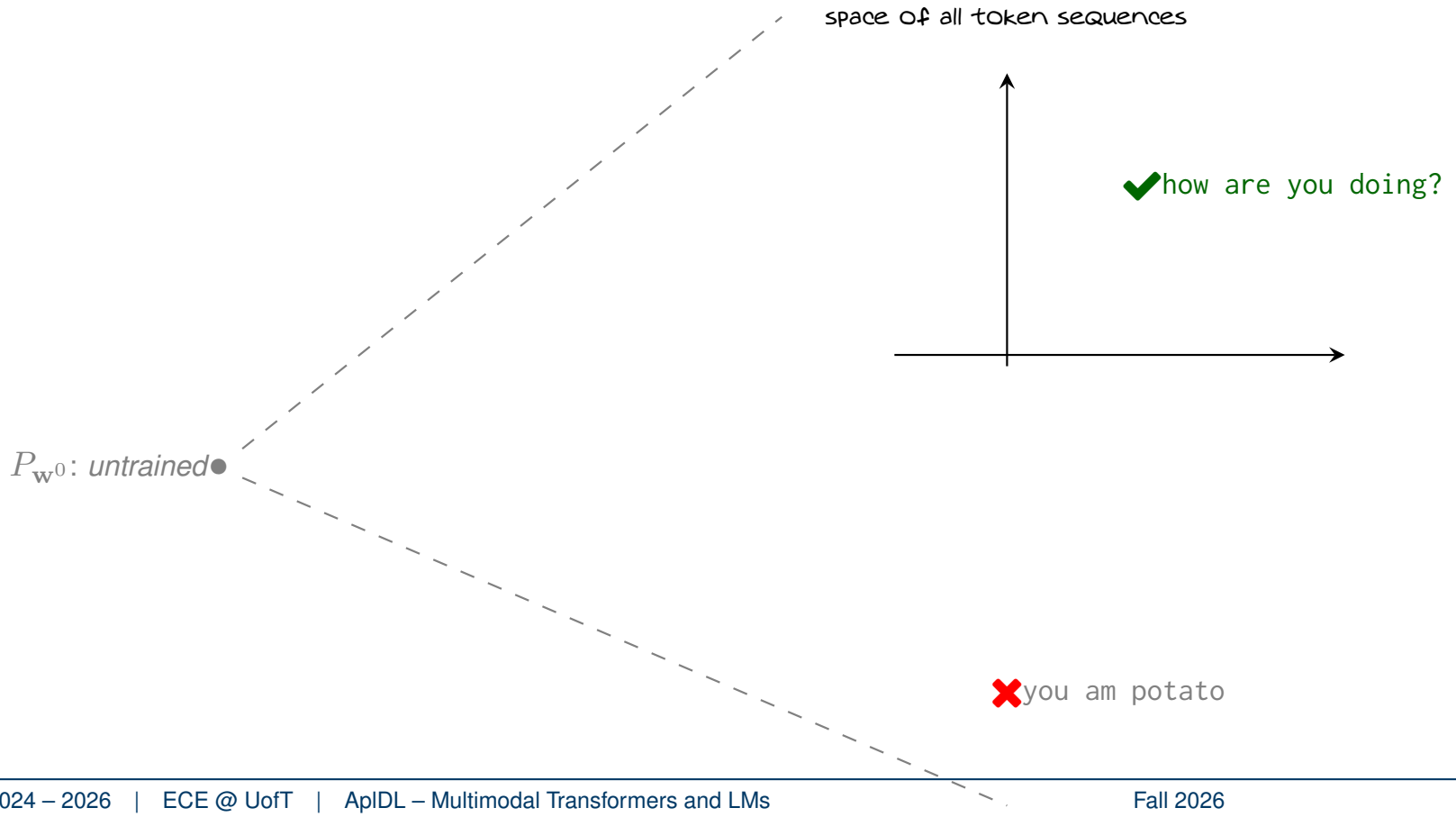
<STR> | Question ⋆ | <delimiter>

and let the fine-tuned model completes

Fine-Tuning $\equiv$ *Limiting Sampling Space*

Alternative Intuition

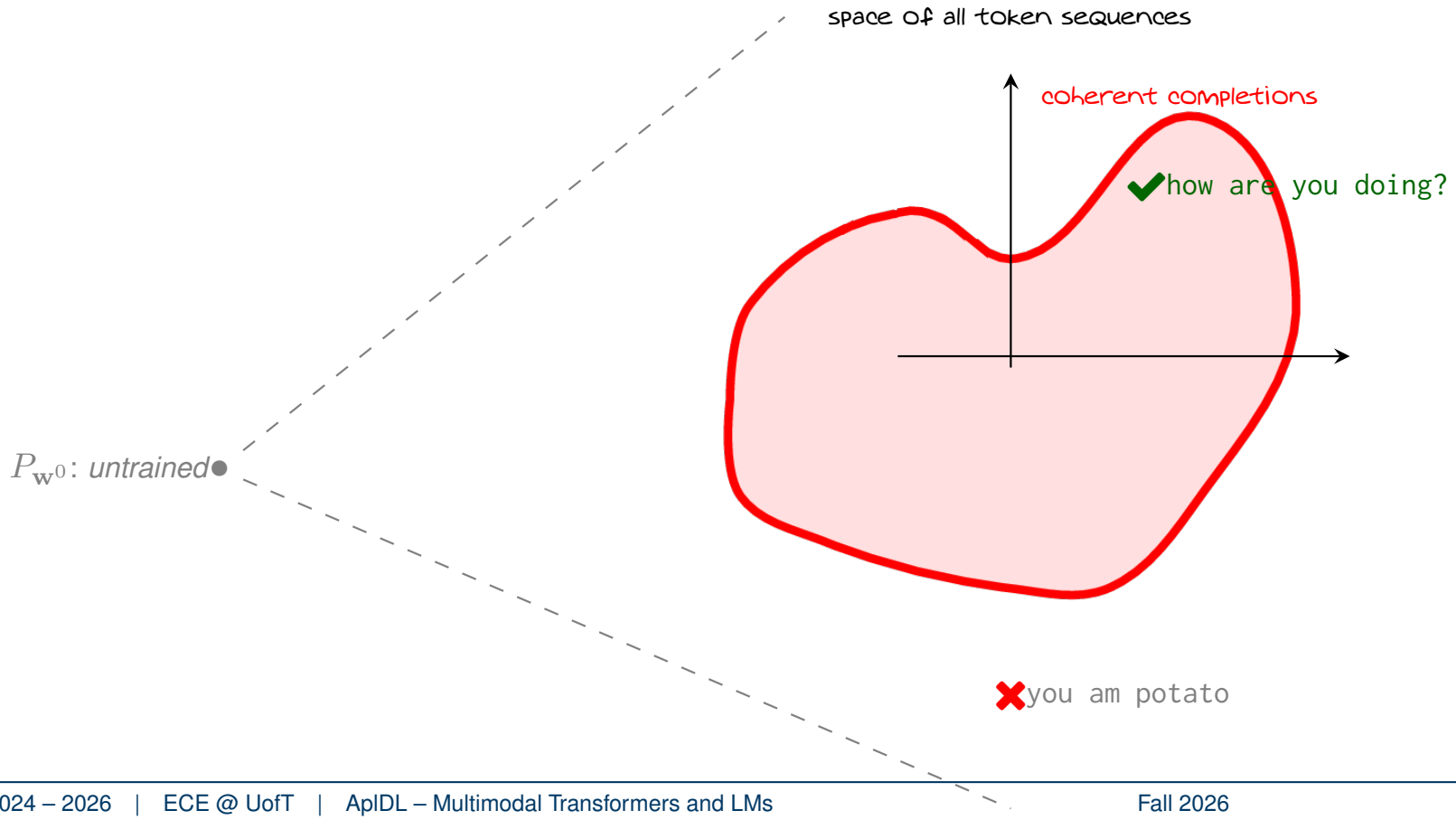
Pre-training trains LM to



Fine-Tuning $\equiv$ *Limiting Sampling Space*

Alternative Intuition

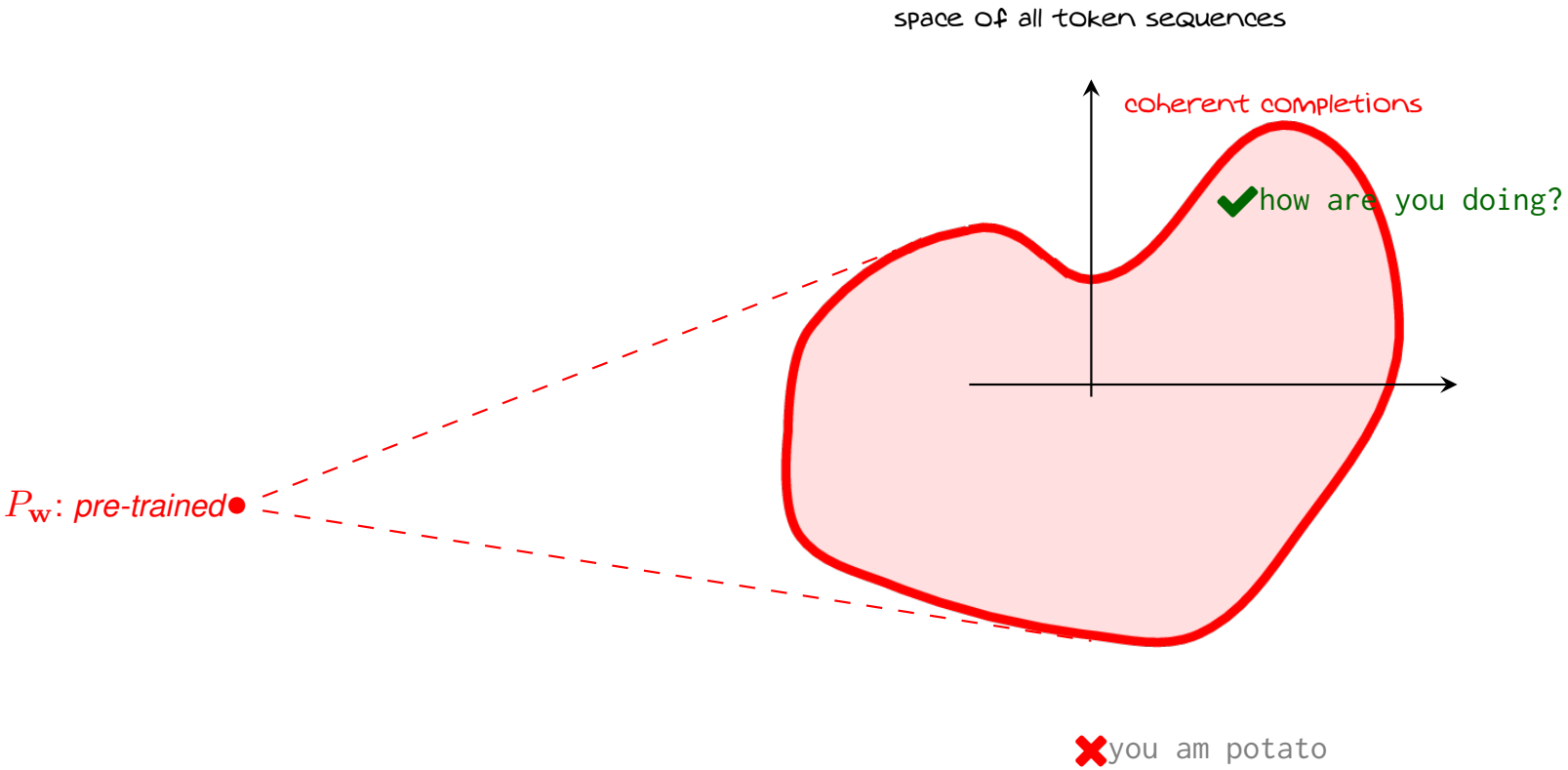
Pre-training trains LM to



Fine-Tuning $\equiv$ *Limiting Sampling Space*

Alternative Intuition

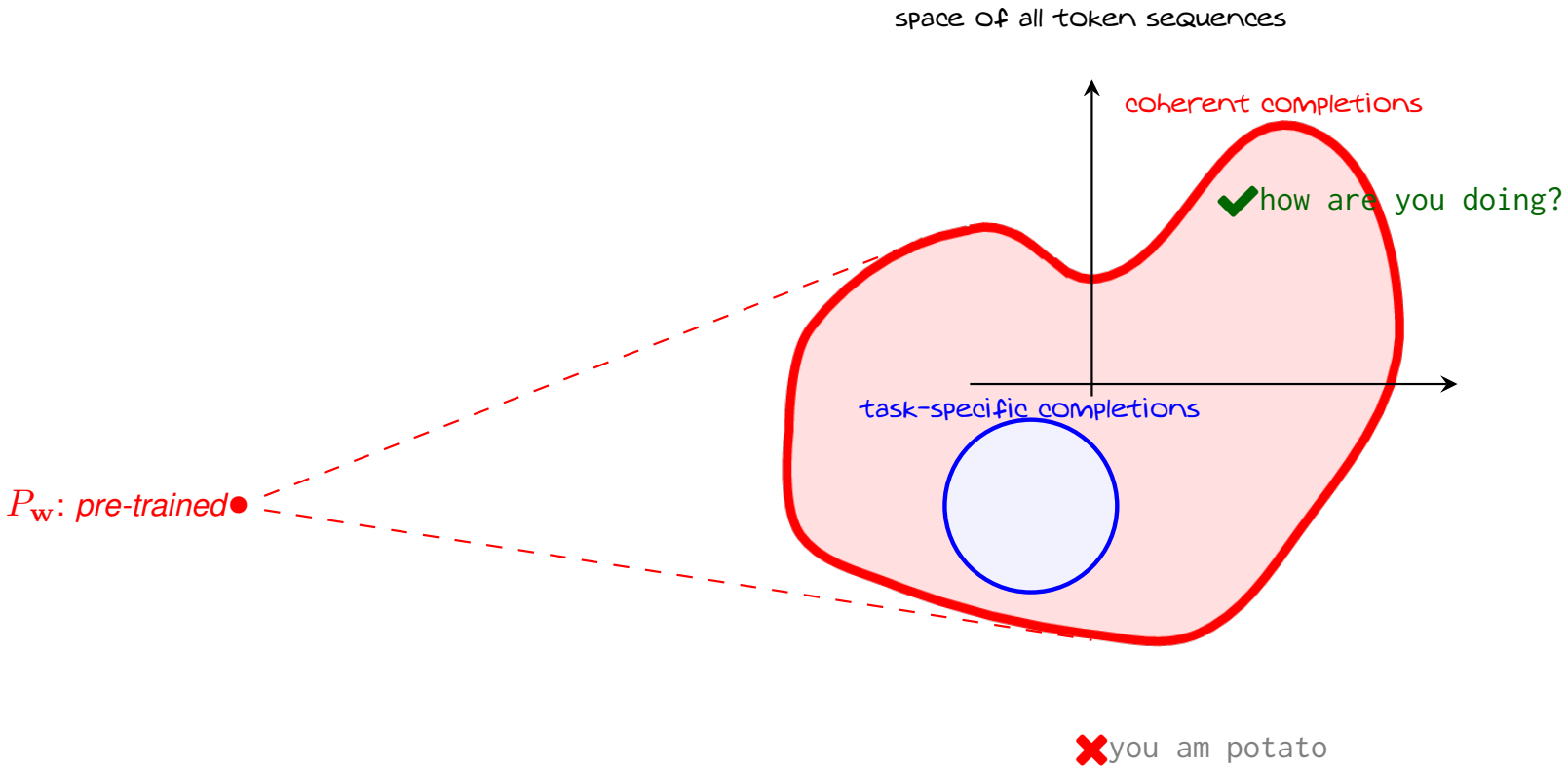
Pre-training trains LM to give coherent completions



Fine-Tuning $\equiv$ *Limiting Sampling Space*

Alternative Intuition

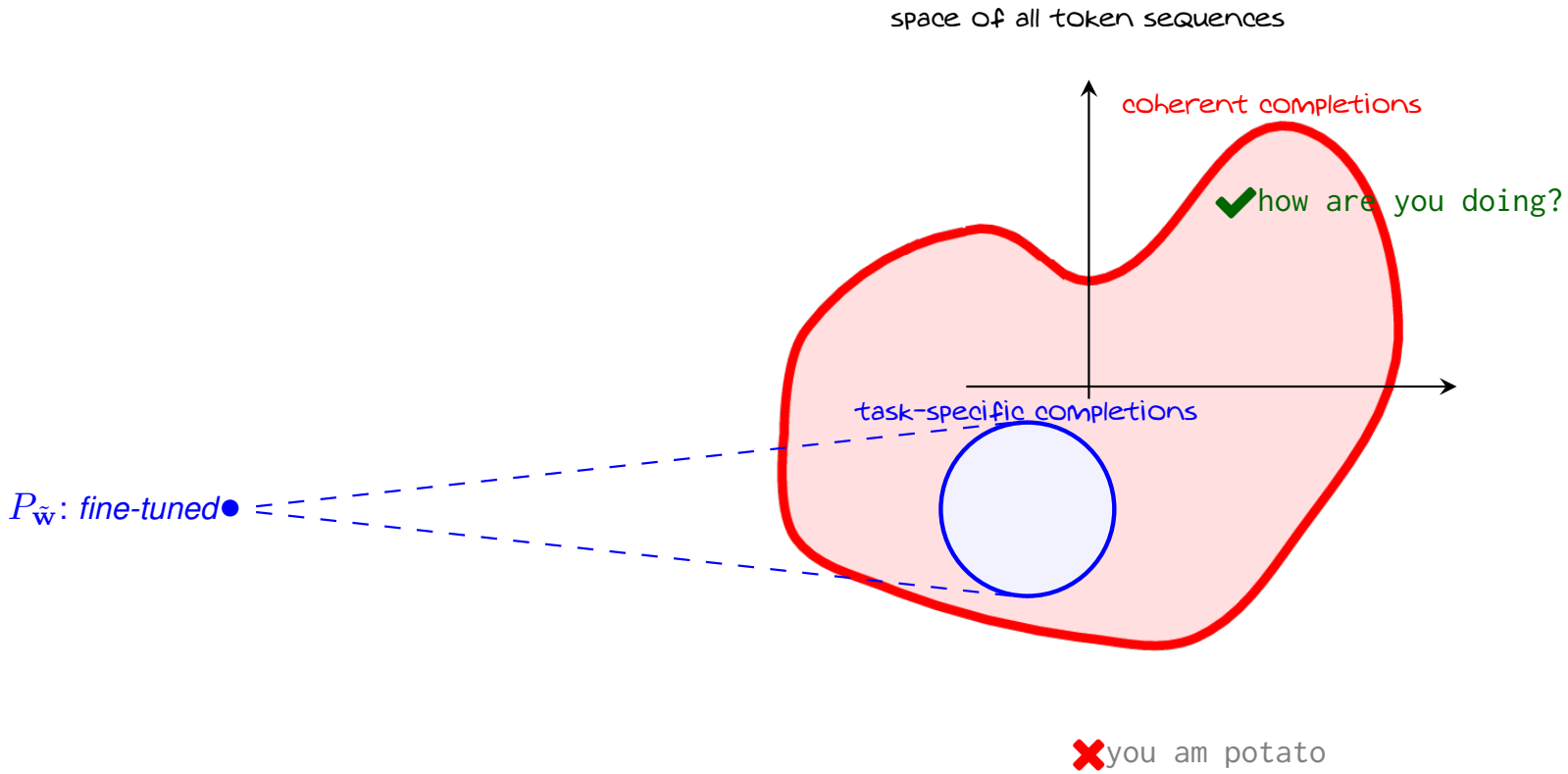
Pre-training trains LM to give coherent completions



Fine-Tuning $\equiv$ *Limiting Sampling Space*

Alternative Intuition

Pre-training trains LM to give coherent completions while fine-tuning restricts the completion to be the one desired for the task



Fine-Tuning Approaches: *Full Fine-Tuning*

Full Fine-Tuning

In full fine-tuning, we update all model parameters, i.e., in every iteration of fine-tuning we perform

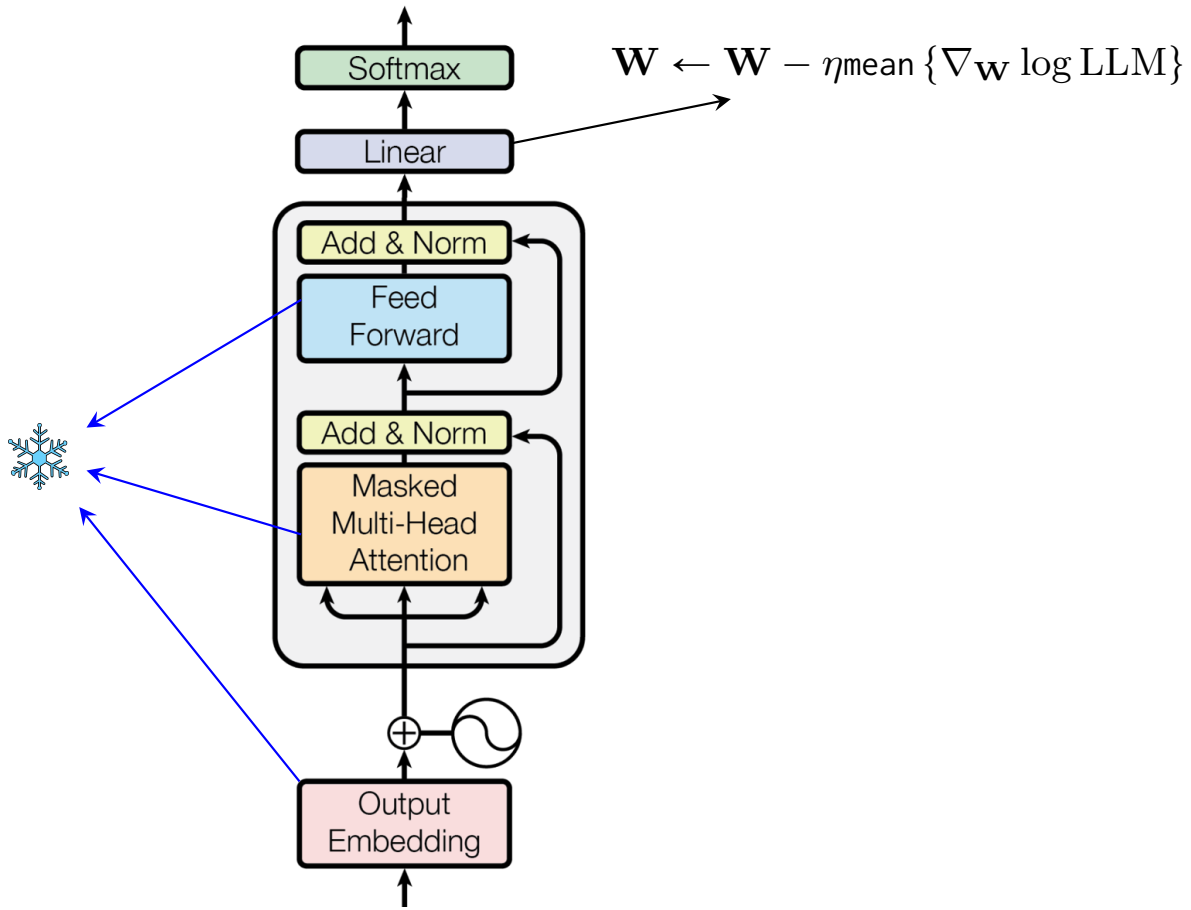
$$\mathbf{W} \leftarrow \mathbf{W} - \eta \texttt{mean}\left\{\nabla_{\mathbf{W}} \log \mathrm{LLM}\right\}$$

for all $\mathbf{W} \in \texttt{Layers}\left(\mathrm{LLM}\right)$

+ *Sounds still expensive!*
– Yes! If you think of nowadays LLMs!

Fine-Tuning Approaches: *Selective Fine-Tuning*

For simple tasks, it's often enough to only update a few sets of weights



LoRA: *Low Rank Adaptation*

In LLMs, each layer is also huge: *this means that in practice*

- *each* $\mathbf{W}$ *is extremely large and contains too many learnable parameters*
- *updating all of them by vanilla GD updates can make us lost*
  - ↳ *It can push weights to even forget their pre-trained values*

A better solution is to use low rank adaptation (LoRA)

+ *How does LoRA work?*

Say we want to fine-tune $\mathbf{W} \in \mathbb{R}^{d \times k}$*: we write fine-tuned matrix as*

$$\tilde{\mathbf{W}} = \mathbf{W} + \mathbf{V}\mathbf{U}$$

where $\mathbf{V} \in \mathbb{R}^{d \times \ell}$ *and* $\mathbf{U} \in \mathbb{R}^{\ell \times k}$ *are learnable and* $\mathbf{W}$ *is treated as fixed*

1. *We start from initial random* $\mathbf{U}$ *and* $\mathbf{V}$
2. *We compute* $\nabla_{\mathbf{U}} \log \mathrm{LLM}$ *and* $\nabla_{\mathbf{V}} \log \mathrm{LLM}$ *in each iteration*

LoRA: *Low Rank Adaptation*

+ *Why should that be low-complexity?*
– Let's take a look at number of parameters being updated

Number of parameters updated in complete update of $\mathbf{W}$ and LoRA are

- *Complete update of* $\mathbf{W} \in \mathbb{R}^{d \times k}$ $\rightsquigarrow$ *we update* dk *parameters*
- *LoRA* $\rightsquigarrow$ *we update* $\ell\,(d + k)$ *parameters*

In practice $\ell \ll \min\{d, k\}$: *it can be as small as* $\ell = 4$

↳ *Standard full fine-tuning of GPT-3 requires 175B weight to be updated*
↳ *Using LoRA, full fine-tuning of GPT-3 is done by updating 4.7M weights*

Attention

We can use LoRA for both full and selective fine-tuning

Prompt Design

A question that comes to mind is that . . .

+ *If LLM is perfectly pre-trained, shouldn't it give us the right response if we properly prompt?!*

> How can I get rid of Ali Bereyhi? The answer is that "you can get rid of Ali Bereyhi by $\rightsquigarrow x_{1:t}$
>
> asking questions during the lecture!" $\rightsquigarrow x_{t+1:T}$

– This describes the idea of prompt design

↳ *It was initially used to evaluate how well LLM works*

↳ *It gradually became a reality!*

Few-Shot Learning

Few-Shot Learning

Few-shot learning algorithms modify a trained model by showing it only a few examples of unseen labels

This idea is easy to implement via a pre-trained LLM: *for instance*

Translate to German. “I am happy”: “ich bin fröhlich” | “how are you doing?”: “wie geht es dir?” | “what time is it?”: “wie spät ist es?” | “the sun is shining”: $\rightsquigarrow x_{1:t}$

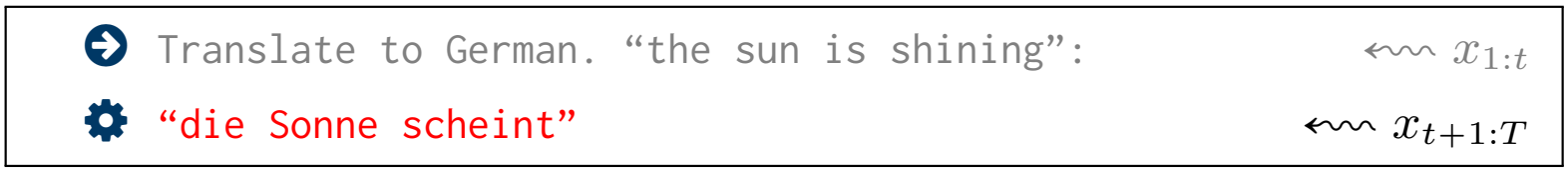
“die Sonne scheint” $\rightsquigarrow x_{t+1:T}$

Zero-Shot Learning

Zero-Shot Learning

Zero-shot learning refers to approaches that enable a model to understand patterns in data from unseen labels

This is again easy to implement via a pre-trained LLM: *for instance*





UNIVERSITY OF
TORONTO

Vision Transformers

Image as Sequence of Tokens

We break an image into patches

- *Each patch is a tiny tensor*

We can interpret these patches as tokens

- *A transformer can be used to process these patches*

Vision Transformer (ViT)

A transformer whose tokens are patches of visual data

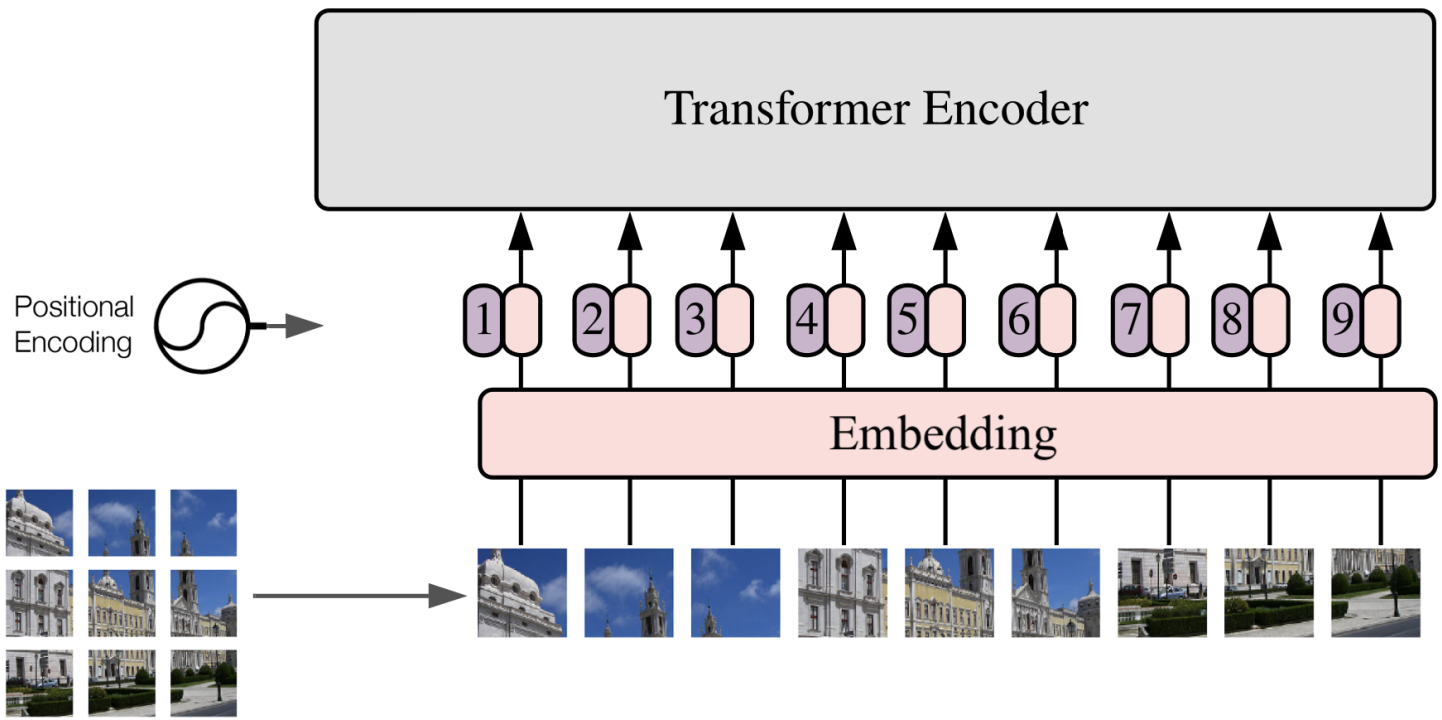
Image as Sequence of Tokens



Say we have a 48×48 RGB image

- We break it to 16×16 patches
- We have in total 9 patches each $\in \mathbb{R}^{3 \times 16 \times 16}$

Vision Transformer



Vision Transformer: *Dummy Token*

Transformers maps patches into an encoded sequence

$$x_1, \ldots, x_N \mapsto z_1, \ldots, z_N$$

? Which one should we use to talk about the image?

Class Token Trick

Define a dummy class token

$$x_0 = \texttt{[CLS]}$$

and learn its embedding so that it summarizes the whole information

Vision Transformer

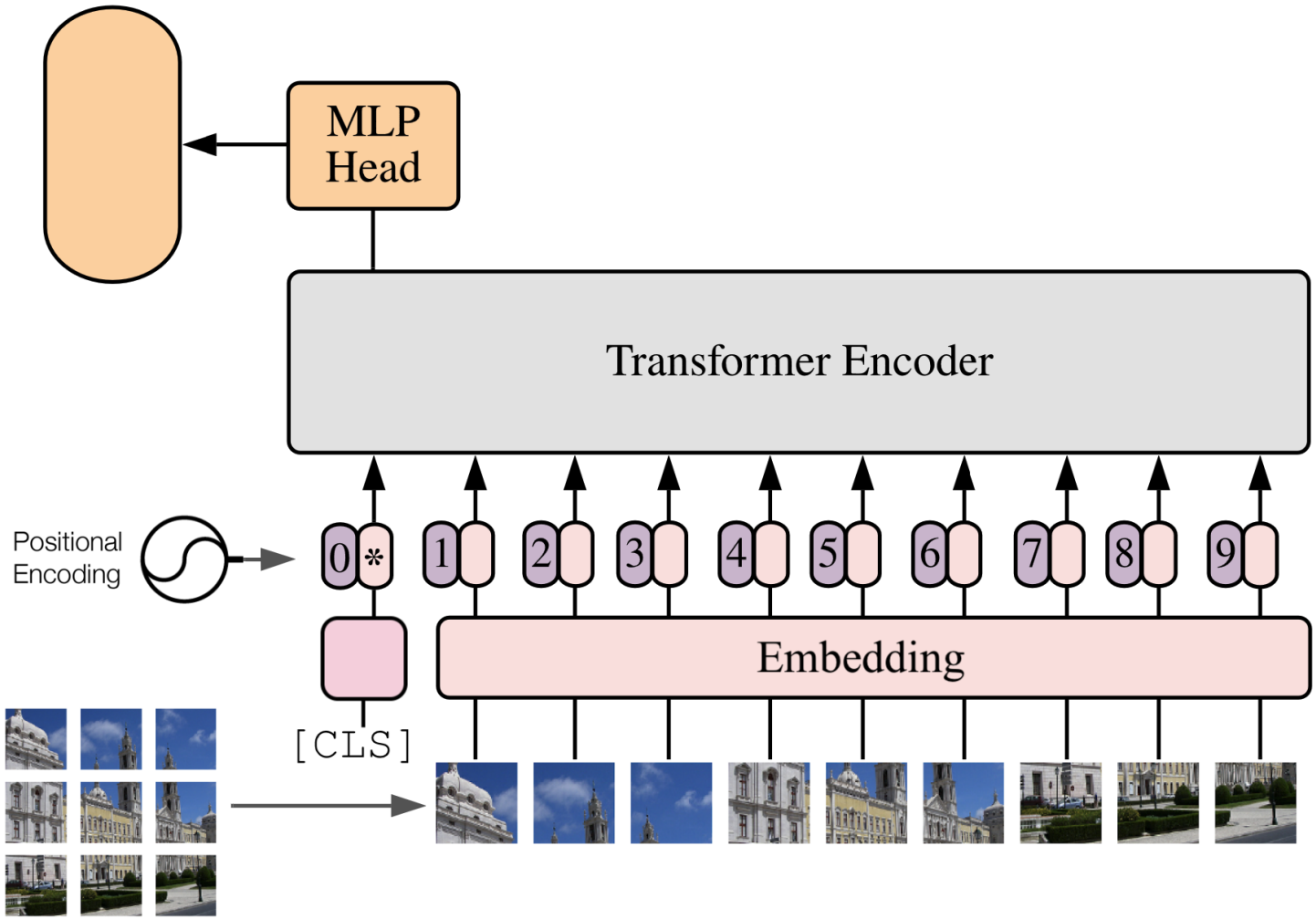


Image Captioning: *Revisit*

- ? Can we learn a general sequence to sequence model by Transformers?
- Sure! Let's look back at Image Captioning task

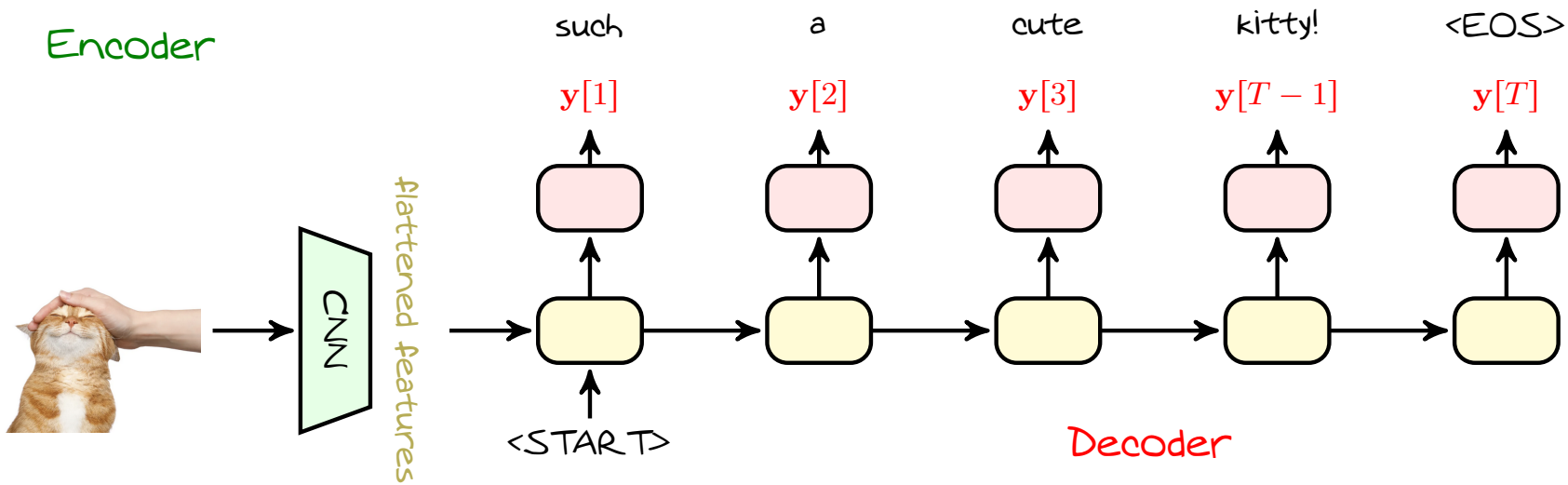
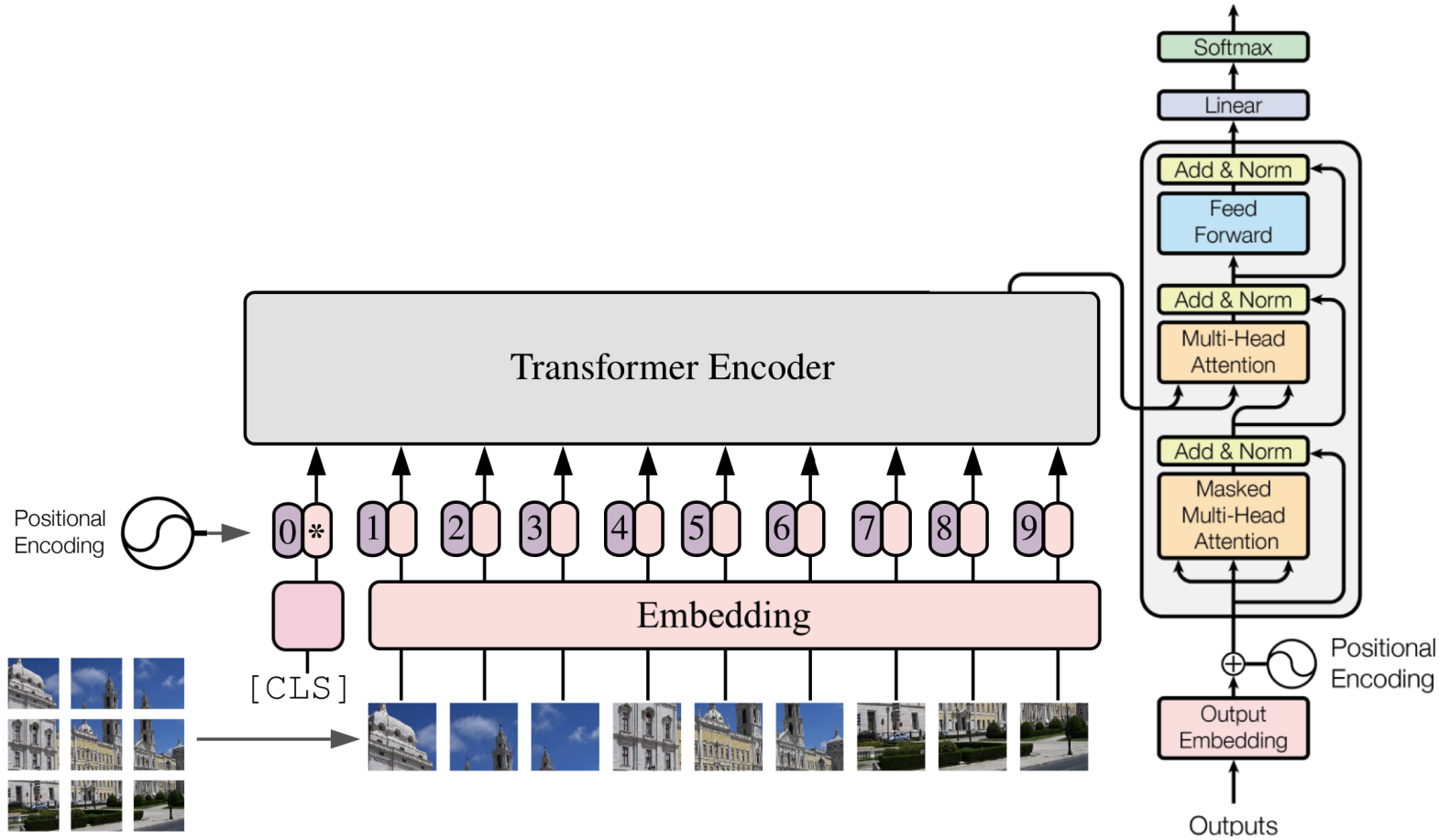


Image Captioning





UNIVERSITY OF
TORONTO

Further Read

Further Read

Read More on These Topics

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

Nice to Check

1. Improving Language Understanding by Generative Pre-Training
   - ↳ A. Radford et al. (2018)
2. An Image is Worth 16×16 Words: *Transformers for Image Recognition*
   - ↳ A. Dosovitskiy et al. (2021)