Multimodal Generative AI 2025
Transformers & Vision Transformers

VISTA
École Polytechnique

Today's lecture

Part I: Sequence to sequence

Part II: Attention

Part III: Transformers

Some slides adapted from various resources: [Intro to Large Language Models, Andrej Karpathy, Fei-Fei Li, Xi Wang, VGG Oxford, Executive Education Polytechnique, Udemy, Deeplearning.ai, Stanford University CS231n, Financial Times, New York Times, Justin Johnson]

Today's lecture

Part I: Sequence to sequence

Part II: Attention

Part III: Transformers

Some slides adapted from various resources: [Intro to Large Language Models, Andrej Karpathy, Fei-Fei Li, Xi Wang, VGG Oxford, Executive Education Polytechnique, Udemy, Deeplearning.ai, Stanford University CS231n, Financial Times, New York Times, Justin Johnson]

Outline: Part I

- **Sequence-to-Sequence**
  - Sequence-to-Sequence with RNNs
  - Sequence-to-Sequence with RNNs and Attention
  - Image captioning

Outline: Part I

- **Sequence-to-Sequence**
  - **Sequence-to-Sequence with RNNs**
  - Sequence-to-Sequence with RNNs and Attention
  - Image captioning

Recurrent Neural Networks

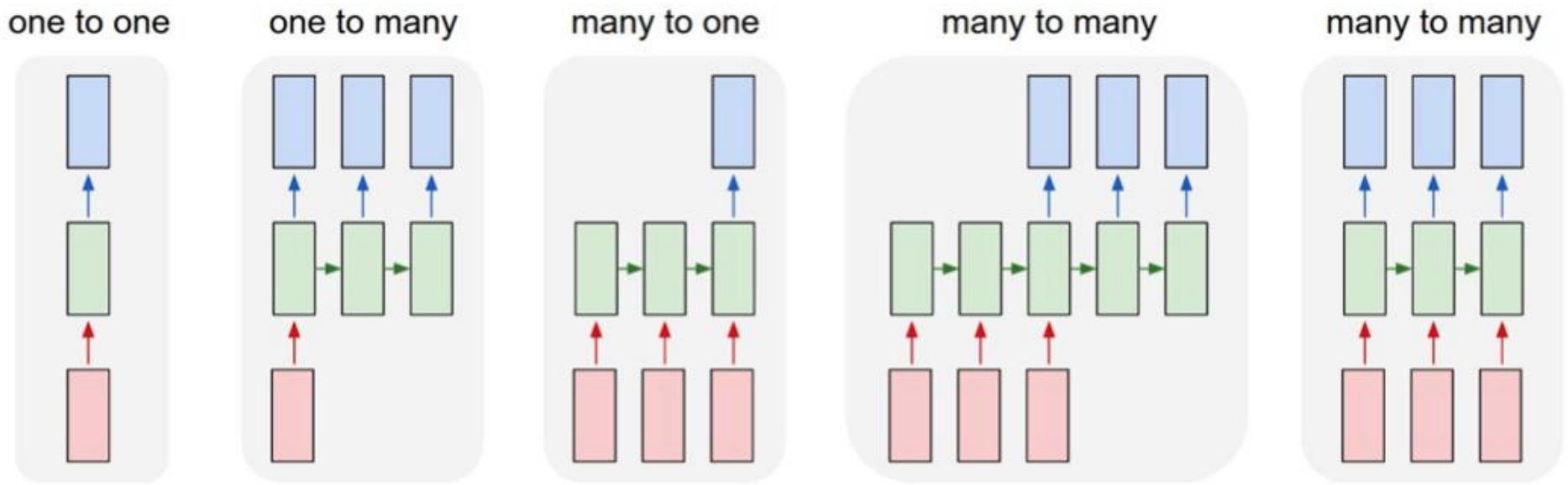
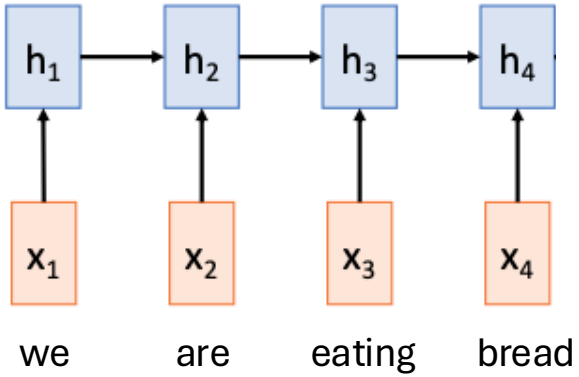


Image → set: image to bounding box
set → set: point clouds to bounding box

Sequence-to-Sequence with RNNs

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

Encoder: $h_t = f_w(x_t, h_{t-1})$

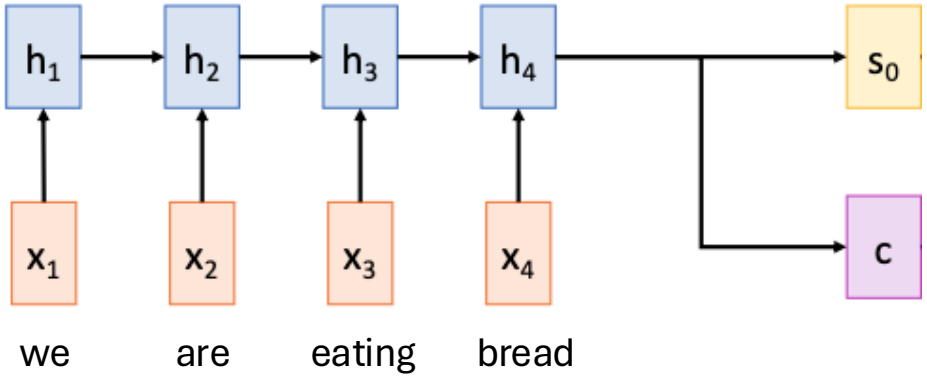


Sequence-to-Sequence with RNNs

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

Encoder: $h_t = f_w(x_t, h_{t-1})$

Predict:
Initial decoder state s_0 (often MLP)
Context vector c (often $c=h_T$)



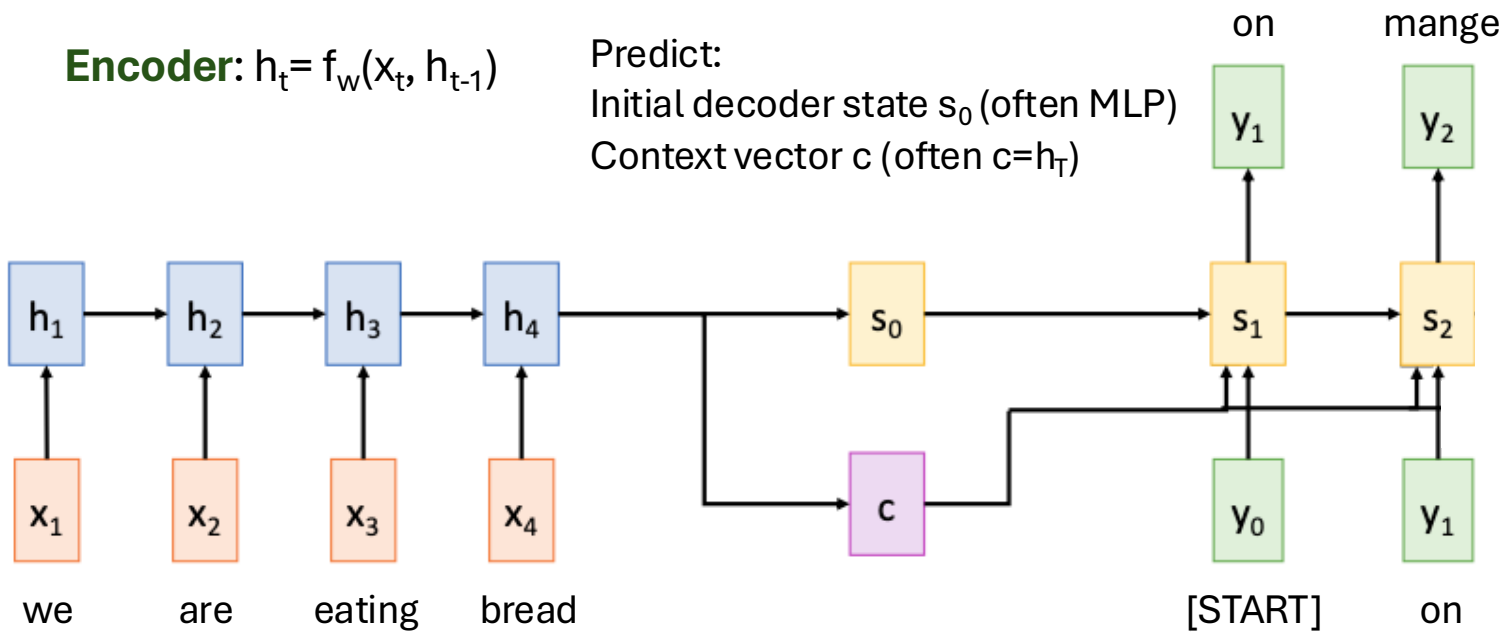
Sequence-to-Sequence with RNNs

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

Encoder: $h_t = f_w(x_t, h_{t-1})$

Predict:
Initial decoder state s_0 (often MLP)
Context vector c (often $c=h_T$)

Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$



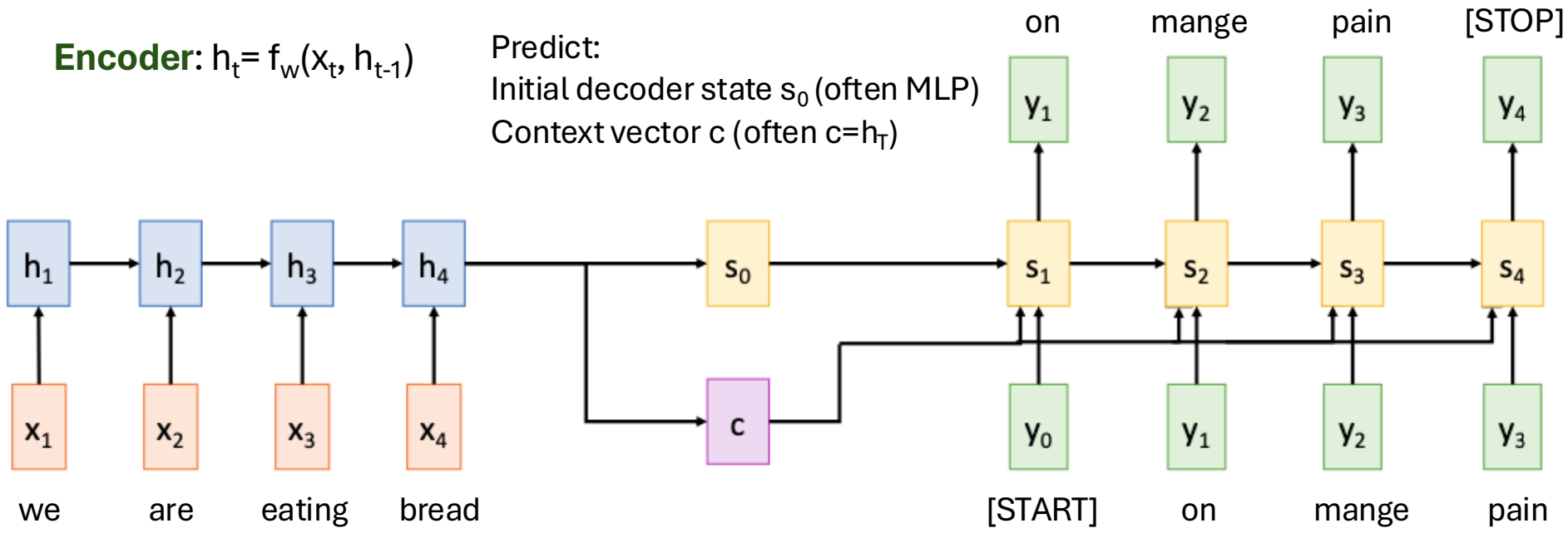
Sequence-to-Sequence with RNNs

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

Encoder: $h_t = f_w(x_t, h_{t-1})$

Predict:
Initial decoder state s_0 (often MLP)
Context vector c (often $c=h_T$)

Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$



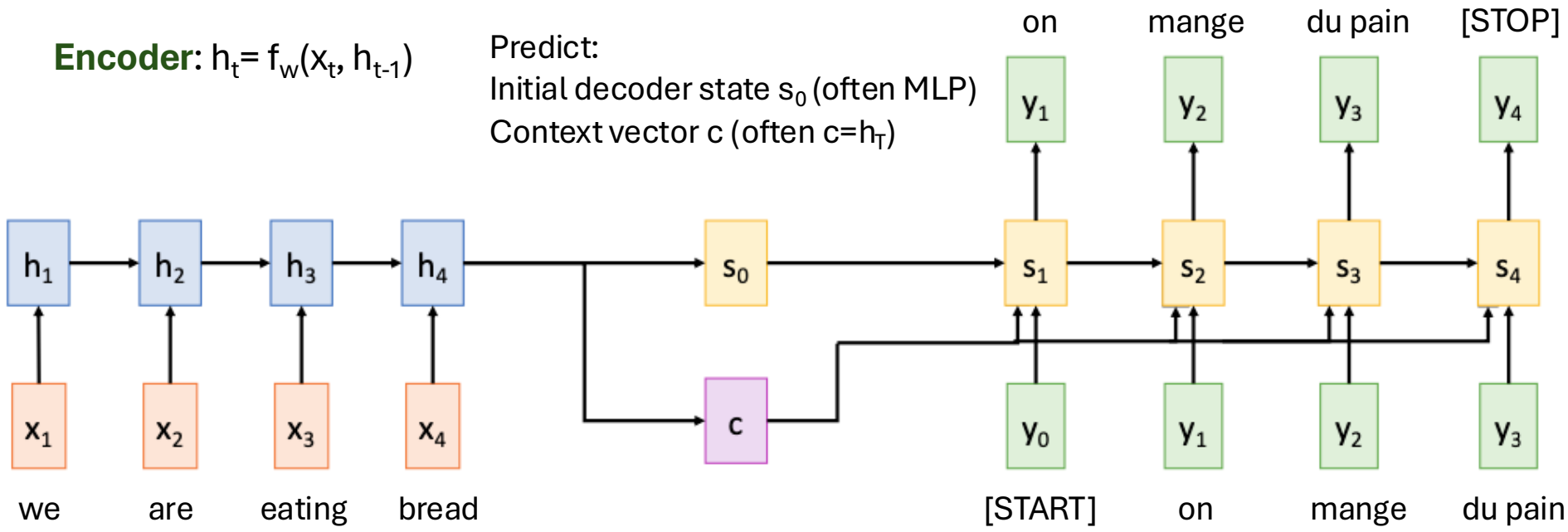
Pop Quiz

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

Encoder: $h_t = f_w(x_t, h_{t-1})$

Predict:
Initial decoder state s_0 (often MLP)
Context vector c (often $c=h_T$)

Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$



What is the problem with this architecture?

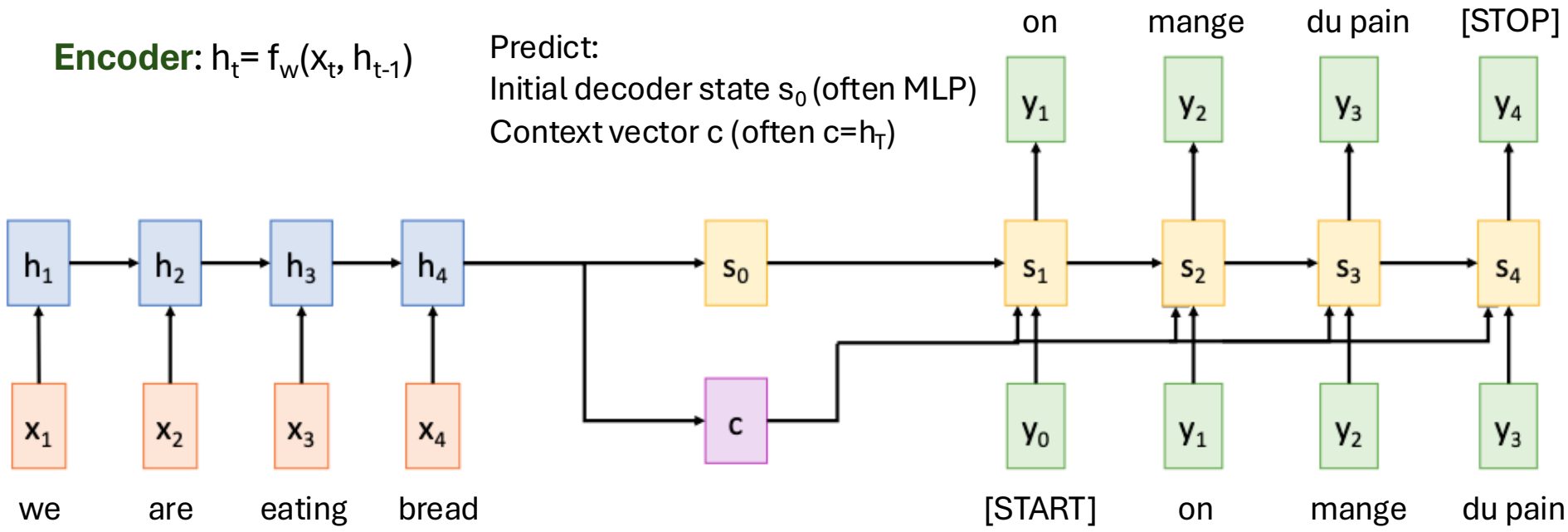
Pop Quiz - Hint

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

Encoder: $h_t = f_w(x_t, h_{t-1})$

Predict:
Initial decoder state s_0 (often MLP)
Context vector c (often $c=h_T$)

Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$



What is the problem with this architecture?

Hint: What if T=1000?

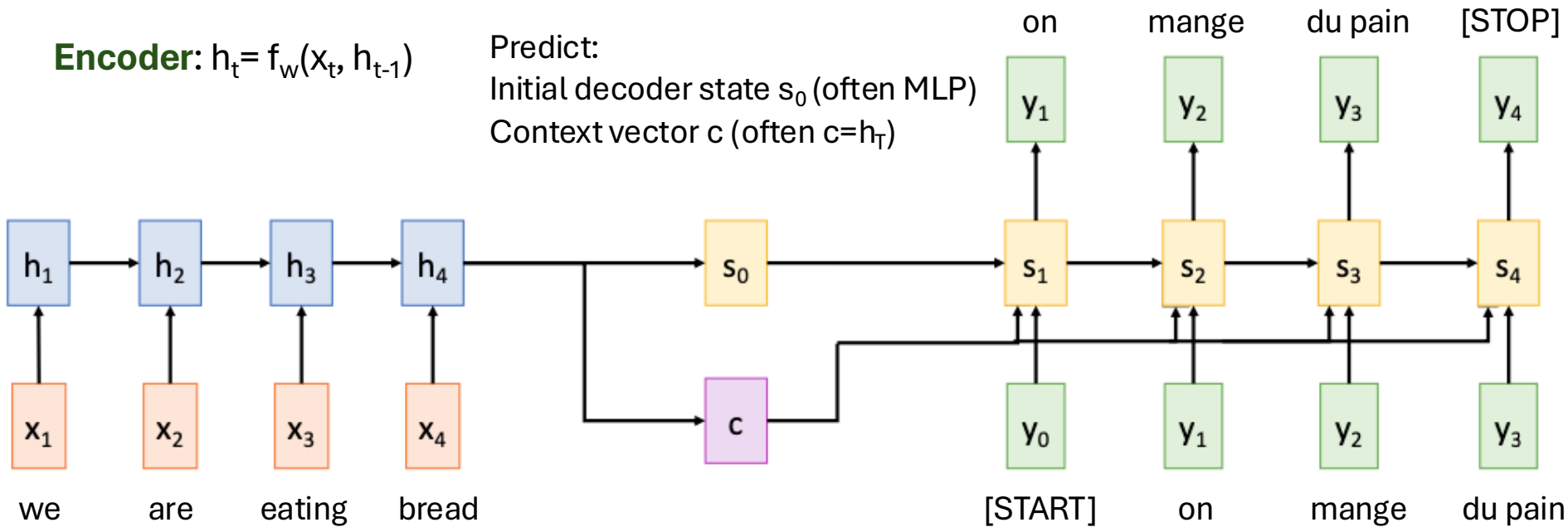
Pop Quiz Answer

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

Encoder: $h_t = f_w(x_t, h_{t-1})$

Predict:
Initial decoder state s_0 (often MLP)
Context vector c (often $c=h_T$)

Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$



Problem: Input sequence bottlenecked through fixed-sized vector

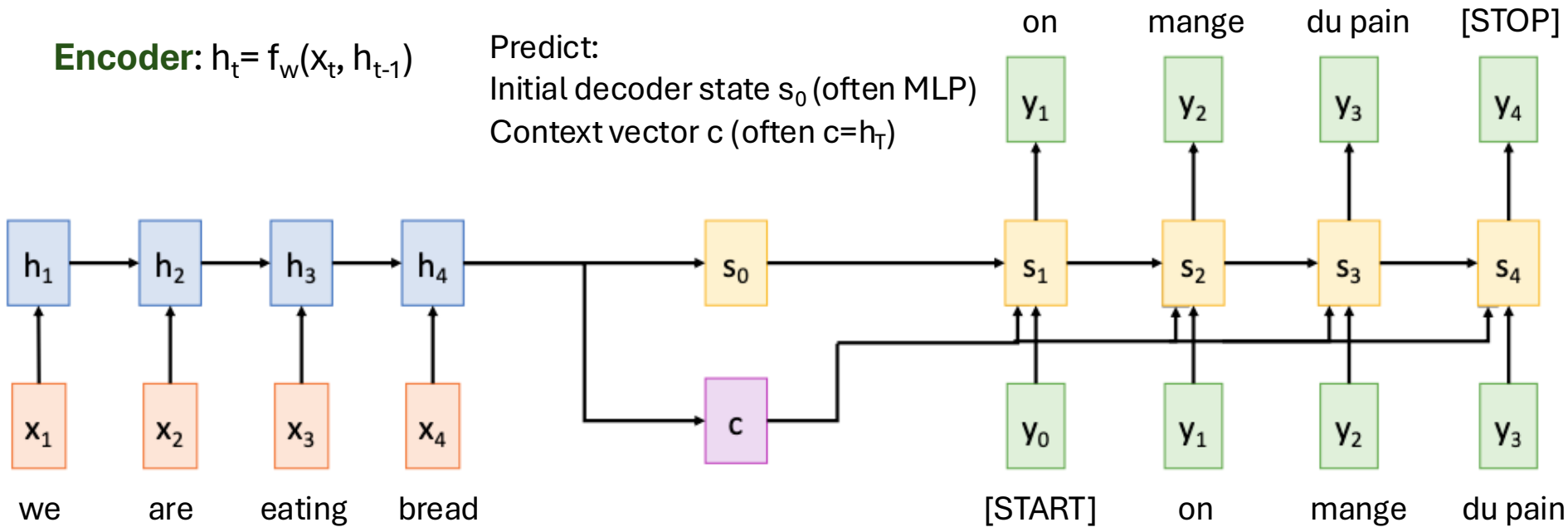
Pop Quiz

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

Encoder: $h_t = f_w(x_t, h_{t-1})$

Predict:
Initial decoder state s_0 (often MLP)
Context vector c (often $c=h_T$)

Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$



Problem: Input sequence bottlenecked through fixed-sized vector

How to solve?

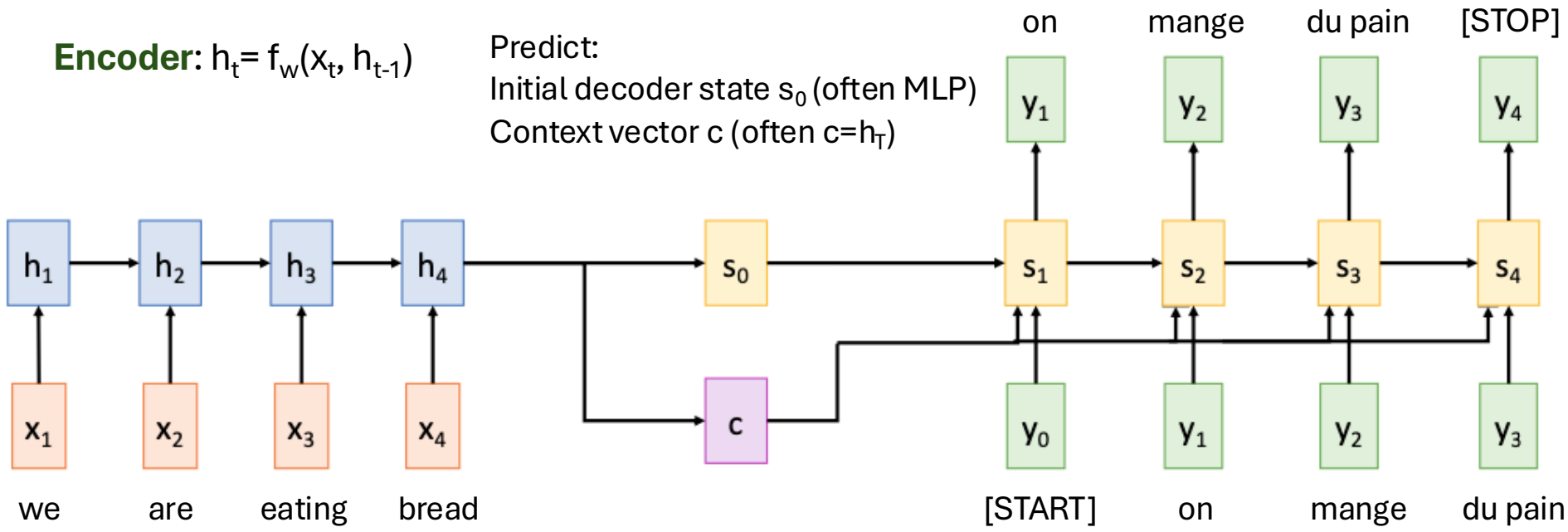
Pop Quiz Answer

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

Encoder: $h_t = f_w(x_t, h_{t-1})$

Predict:
Initial decoder state s_0 (often MLP)
Context vector c (often $c=h_T$)

Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$



Problem: Input sequence bottlenecked through fixed-sized vector

Solution: Use new context vector at each step of the decoder

Outline: Part I

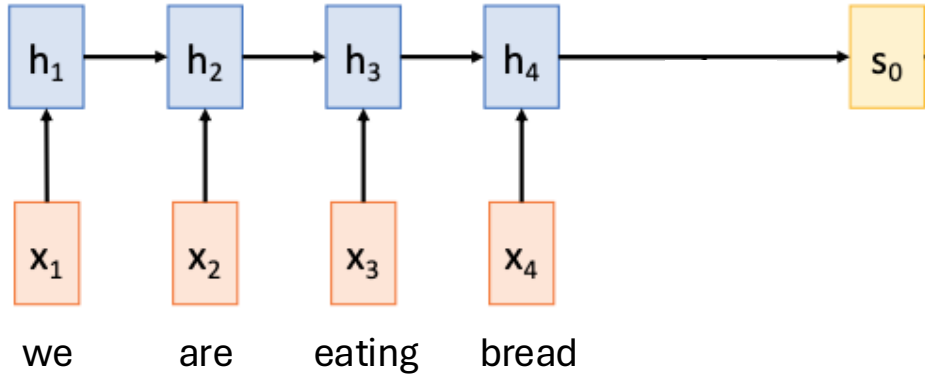
- **Sequence-to-Sequence**
  - Sequence-to-Sequence with RNNs
  - **Sequence-to-Sequence with RNNs and Attention**
  - Image captioning

Sequence-to-Sequence with RNNs and Attention

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

Encoder: $h_t = f_w(x_t, h_{t-1})$

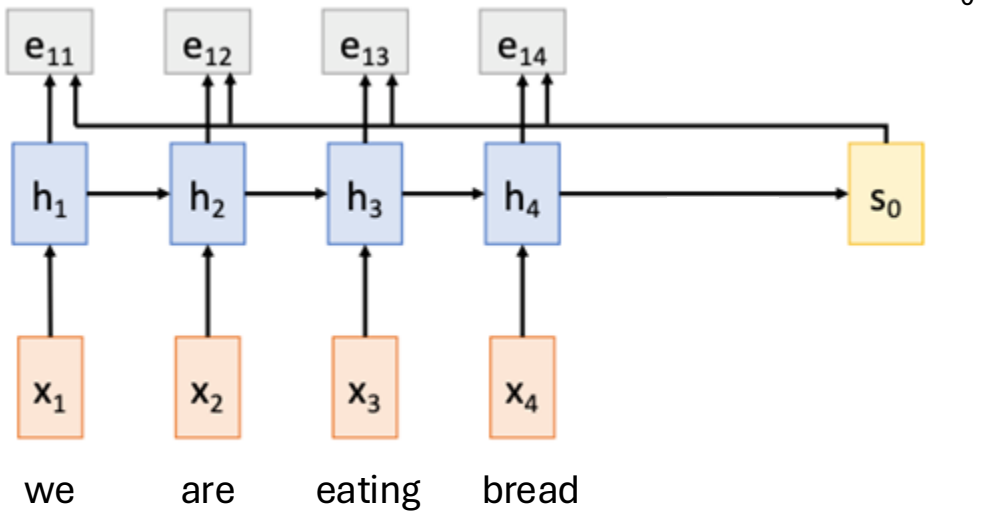
Predict:
Initial decoder state s_0



Sequence-to-Sequence with RNNs and Attention

Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$

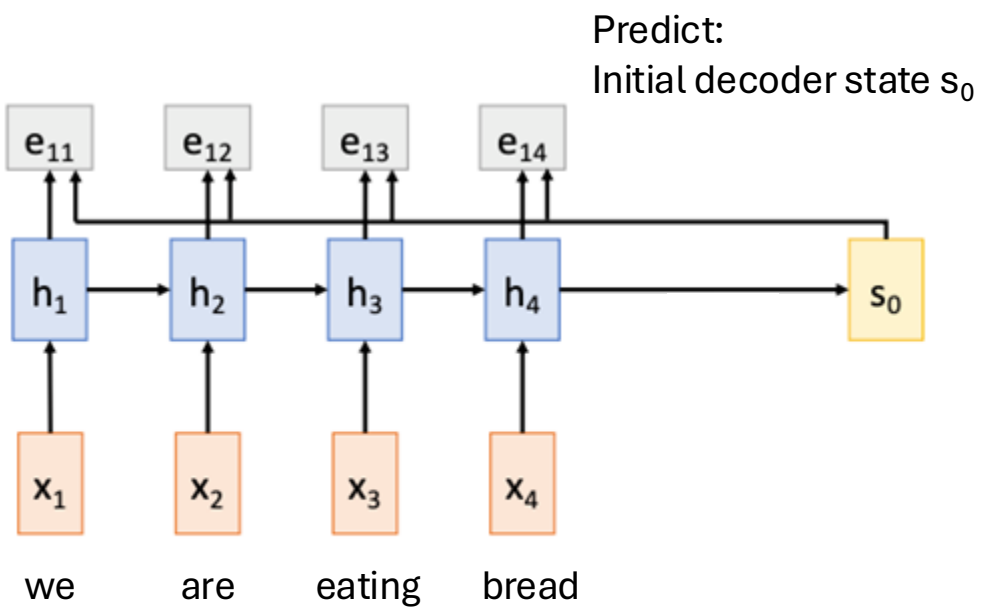
Predict:
Initial decoder state s_0



Compute (scalar) alignment scores
$e_{t,i}=f_{att}(s_{t-1},h_i)$, where f_{att} is a MLP

Sequence-to-Sequence with RNNs and Attention

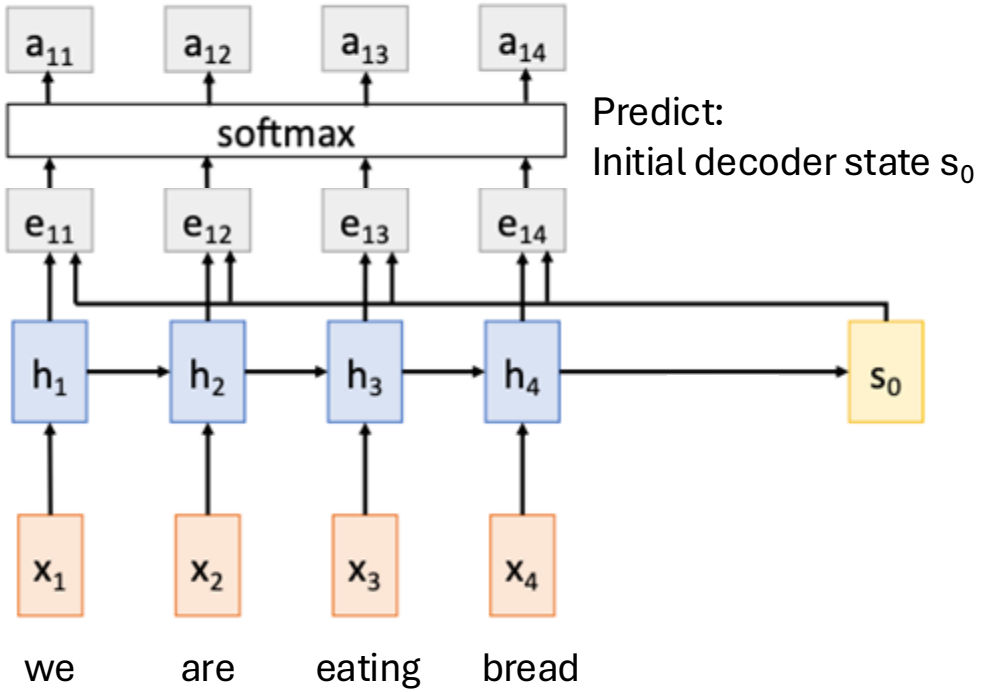
Input: Sequence $x_1,\ldots,x_T$
Output: Sequence $y_1,\ldots,y_{T'}$



Compute (scalar) alignment scores $e_{t,i}=f_{att}(s_{t-1},h_i)$, where f_{att} is a MLP

how much should we attend to each hidden state of the encoder given the current hidden state of the decoder

Sequence-to-Sequence with RNNs and Attention



Predict:
Initial decoder state s_0

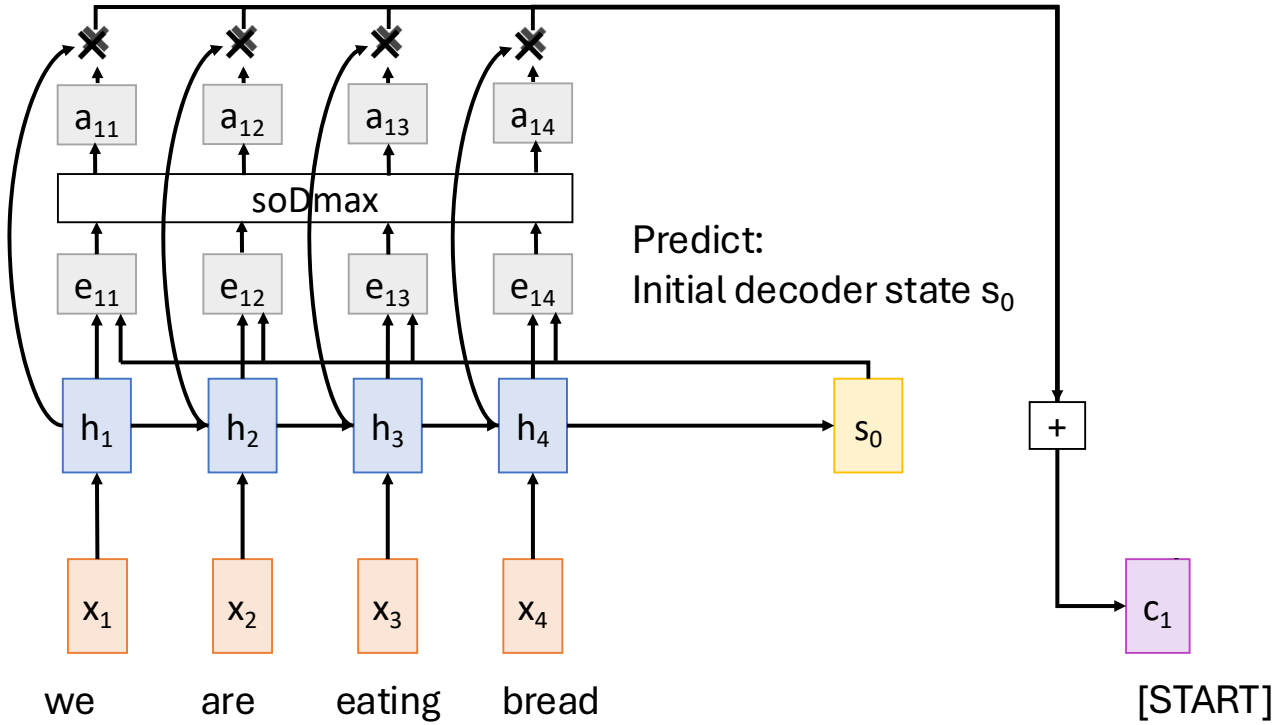
Compute (scalar) alignment scores
$e_{t,i}=f_{att}(s_{t-1},h_i)$, where f_{att} is a MLP

Normalize alignment scores to get
Attention weights:

$$0 < a_{t,i} < 1,$$

$$\sum_i a_{t,i} = 1$$

Sequence-to-Sequence with RNNs and Attention



Compute (scalar) alignment scores $e_{t,i}=f_{att}(s_{t-1},h_i)$, where f_{att} is a MLP

Normalize alignment scores to get Attention weights:

$$0 < a_{t,i} < 1,$$

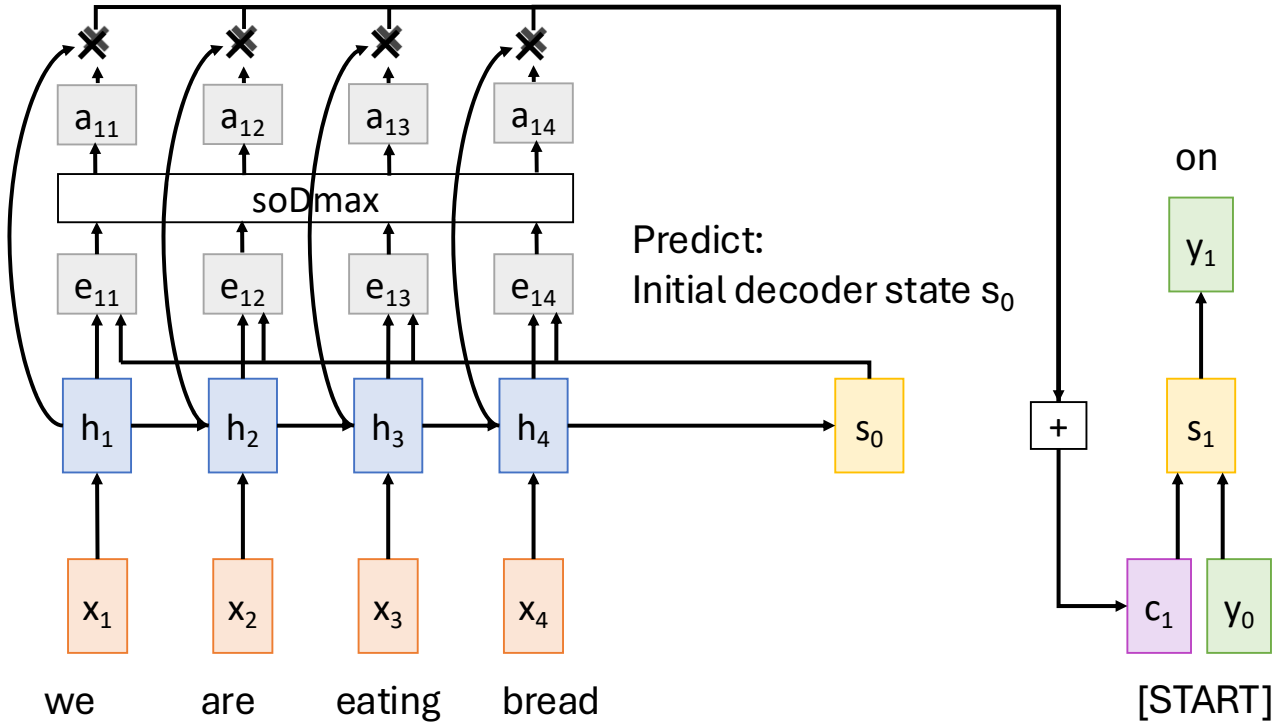
$$\sum_i a_{t,i} = 1$$

Compute context vectors as linear combination of hidden states

Attention weights:

$$Ct = \sum_i a_{t,i} h_i$$

Sequence-to-Sequence with RNNs and Attention



Compute (scalar) alignment scores $e_{t,i}=f_{att}(s_{t-1},h_i)$, where f_{att} is a MLP

Normalize alignment scores to get Attention weights:

$$0 < a_{t,i} < 1,$$

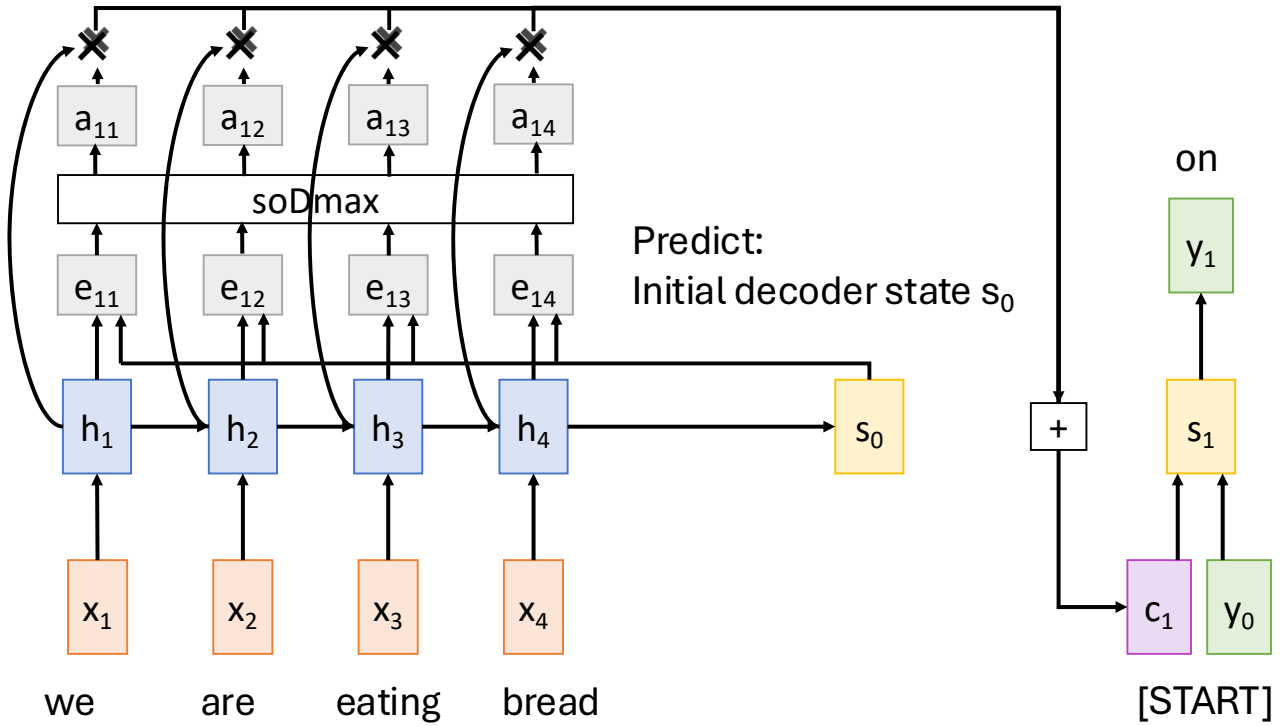
$$\sum_i a_{t,i} = 1$$

Compute context vectors as linear combination of hidden states

Attention weights:

$$Ct = \sum_i a_{t,i} h_i$$

Sequence-to-Sequence with RNNs and Attention



Compute (scalar) alignment scores $e_{t,i}=f_{att}(s_{t-1},h_i)$, where f_{att} is a MLP

Normalize alignment scores to get Attention weights:

$$0 < a_{t,i} < 1,$$

$$\sum_i a_{t,i} = 1$$

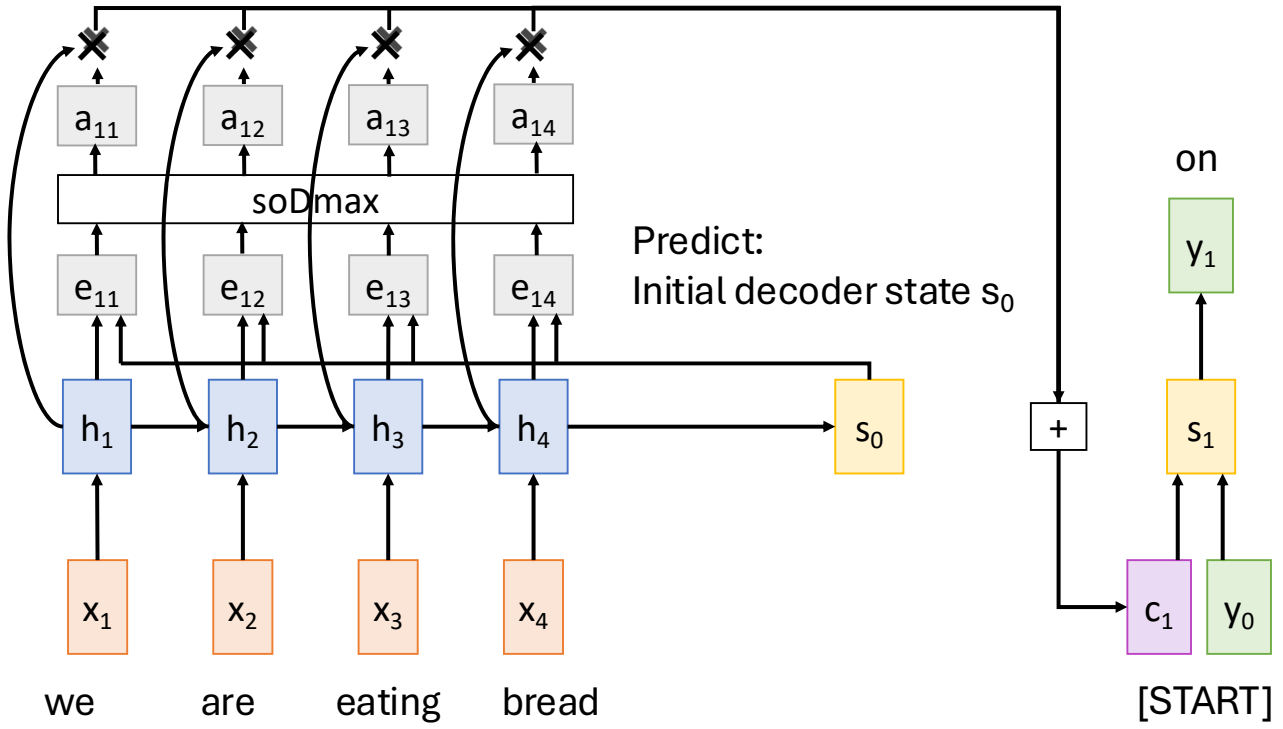
Compute context vectors as linear combination of hidden states

Attention weights:

$$Ct = \sum_i a_{t,i} h_i$$

Intuition: context vector → attend the relevant part of the input sequence

Sequence-to-Sequence with RNNs and Attention



Compute (scalar) alignment scores $e_{t,i}=f_{att}(s_{t-1},h_i)$, where f_{att} is a MLP

Normalize alignment scores to get Attention weights:

$$0 < a_{t,i} < 1,$$

$$\sum_i a_{t,i} = 1$$

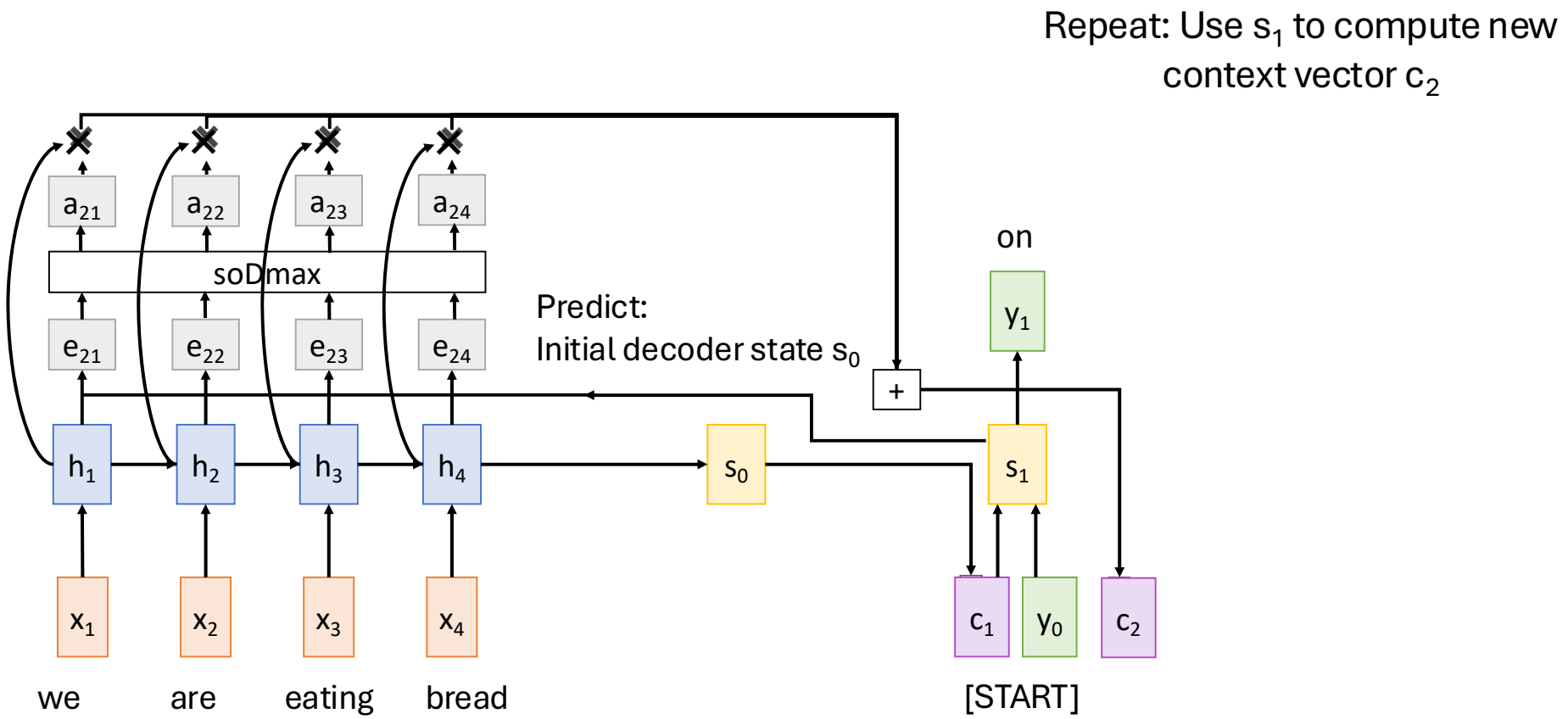
Compute context vectors as linear combination of hidden states

Attention weights:

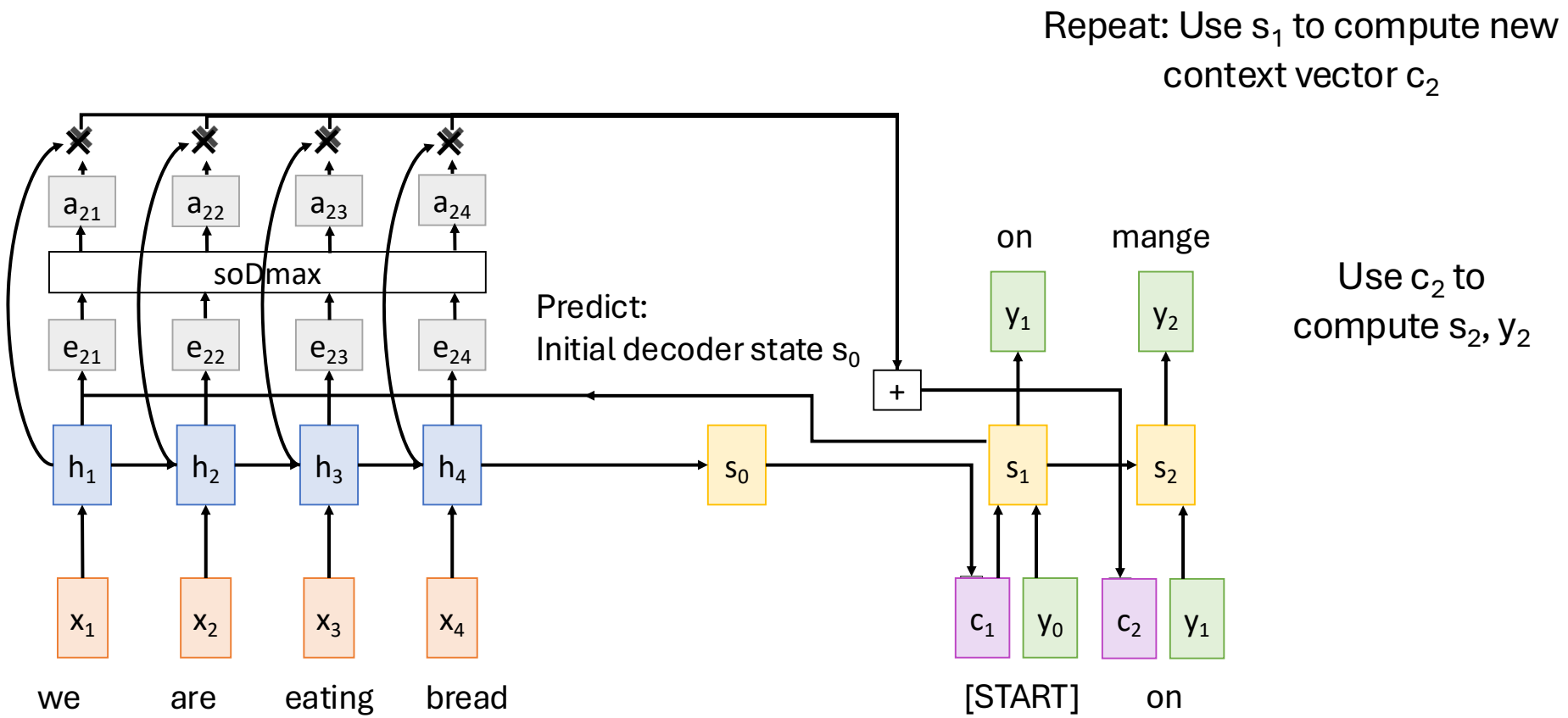
$$Ct = \sum_i a_{t,i} h_i$$

Intuition: context vector → attend the relevant part of the input sequence:
"are eating" → "mange"

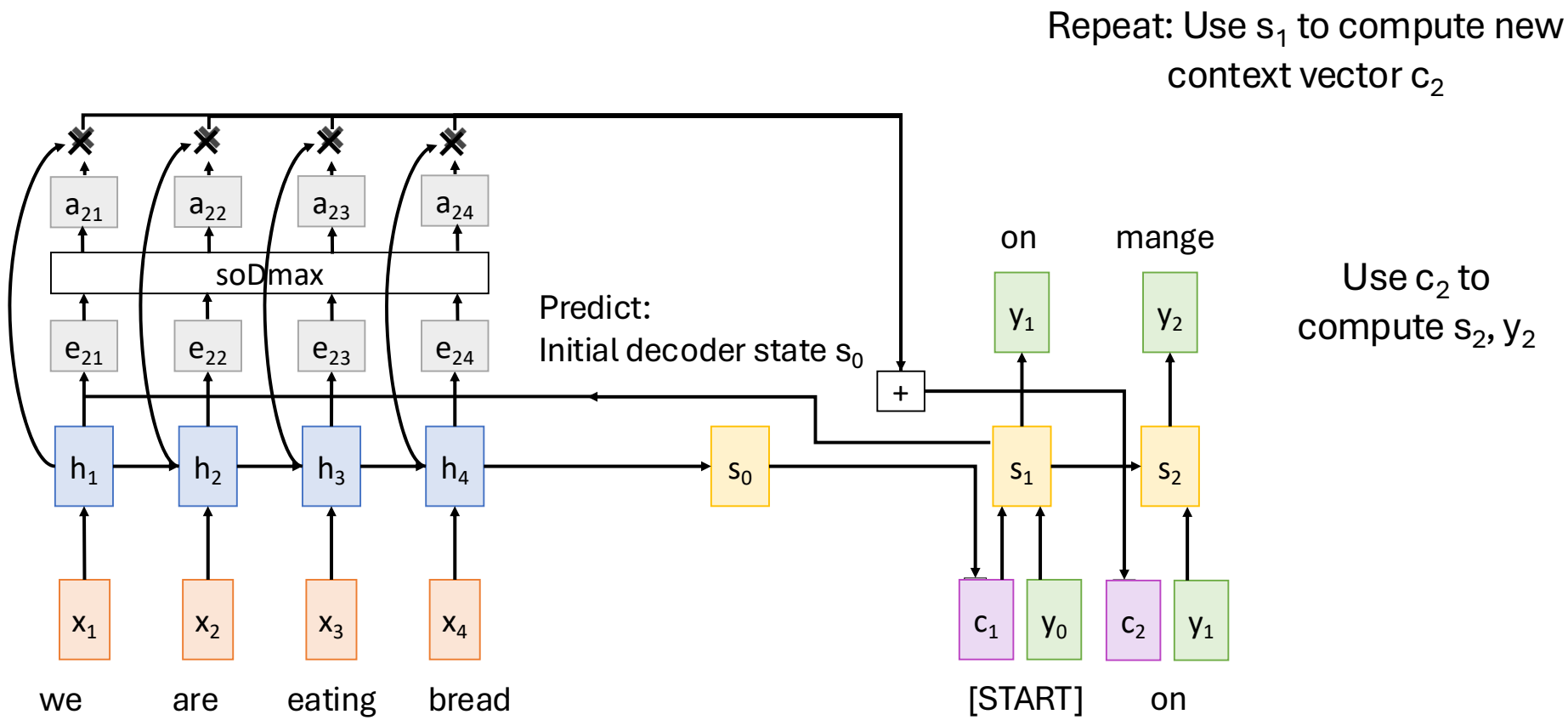
Sequence-to-Sequence with RNNs and Attention



Sequence-to-Sequence with RNNs and Attention



Sequence-to-Sequence with RNNs and Attention

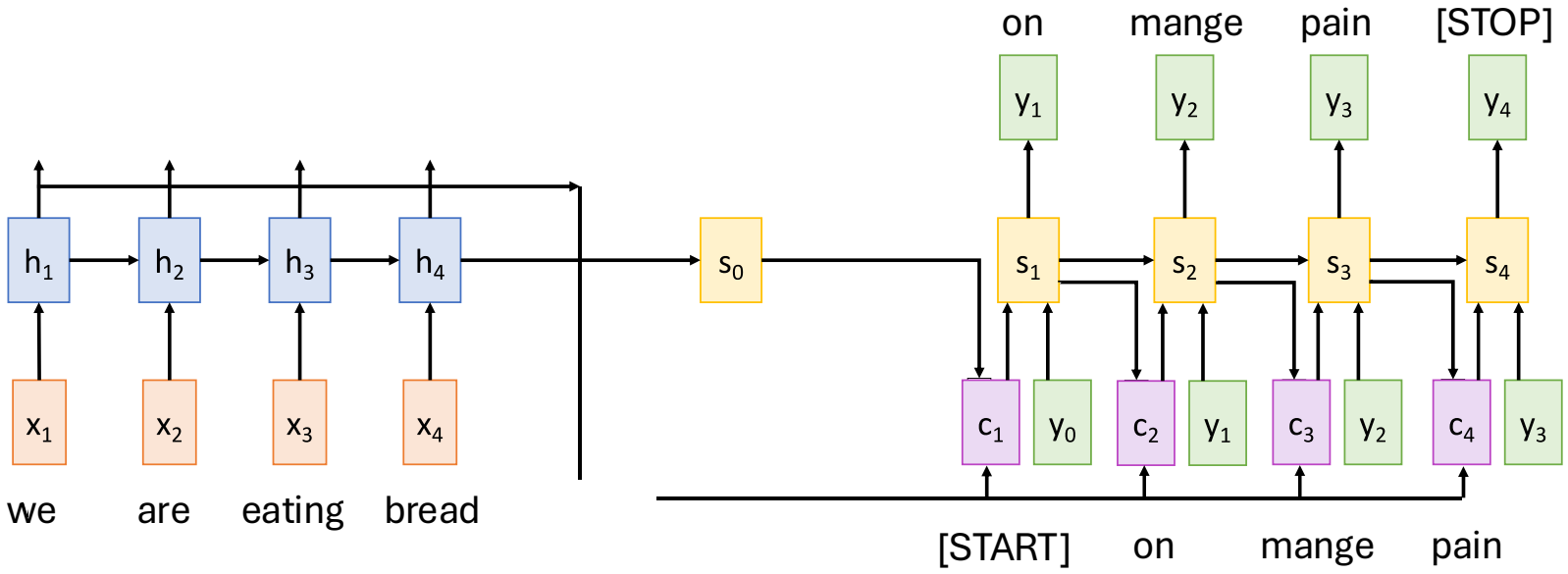


Intuition: context vector → attend the relevant part of the input sequence: "bread" → "pain"

Sequence-to-Sequence with RNNs and Attention

Use a different context vector in each timestep of decoder

- Input sequence not bottlenecked through single vector
- At each timestep of decoder, context vector "looks at" different parts of the input sequence



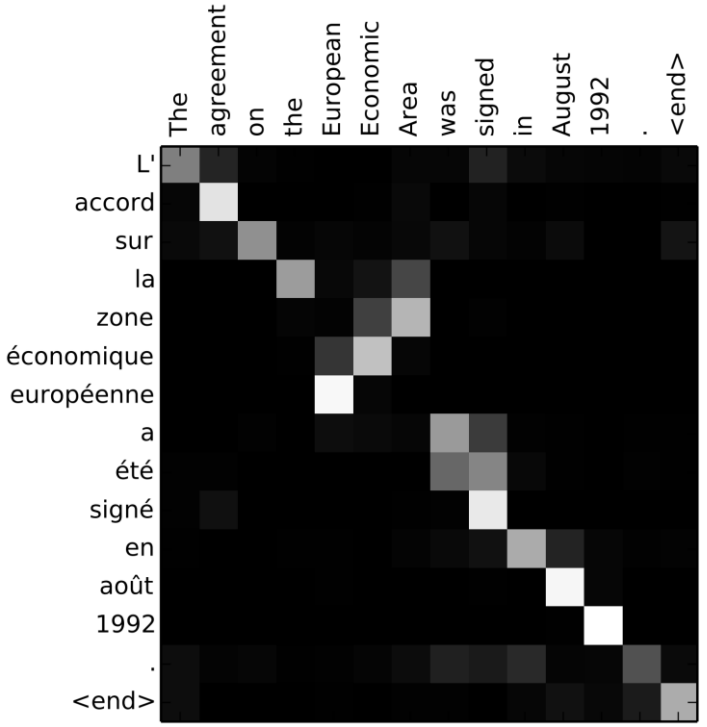
Sequence-to-Sequence with RNNs and Attention

Example: English to French translation

Input: “The agreement on the European Economic Area was signed in August 1992.”

Output: “L’accord sur la zone économique européenne a été signé en août 1992.”

Visualize attention weights $a_{t,i}$



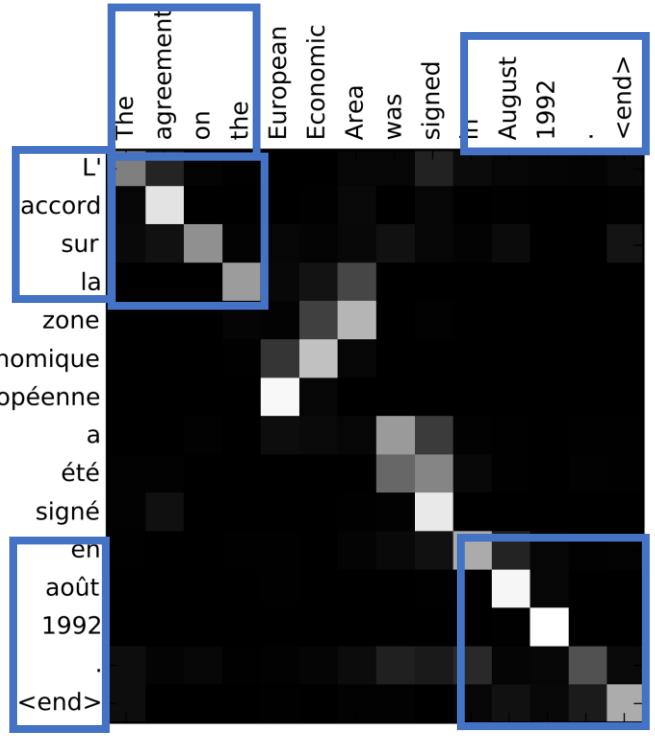
Sequence-to-Sequence with RNNs and Attention

Example: English to French transla<on

Input: "**The agreement on the** European Economic Area was signed **in August 1992**."

Output: "**L'accord sur la** zone économique européenne a été signé **en août 1992**."

Visualize a2 en<on weights $a_{t,i}$



Diagonal a) en+on means words correspond in order

Diagonal a) en+on means words correspond in order

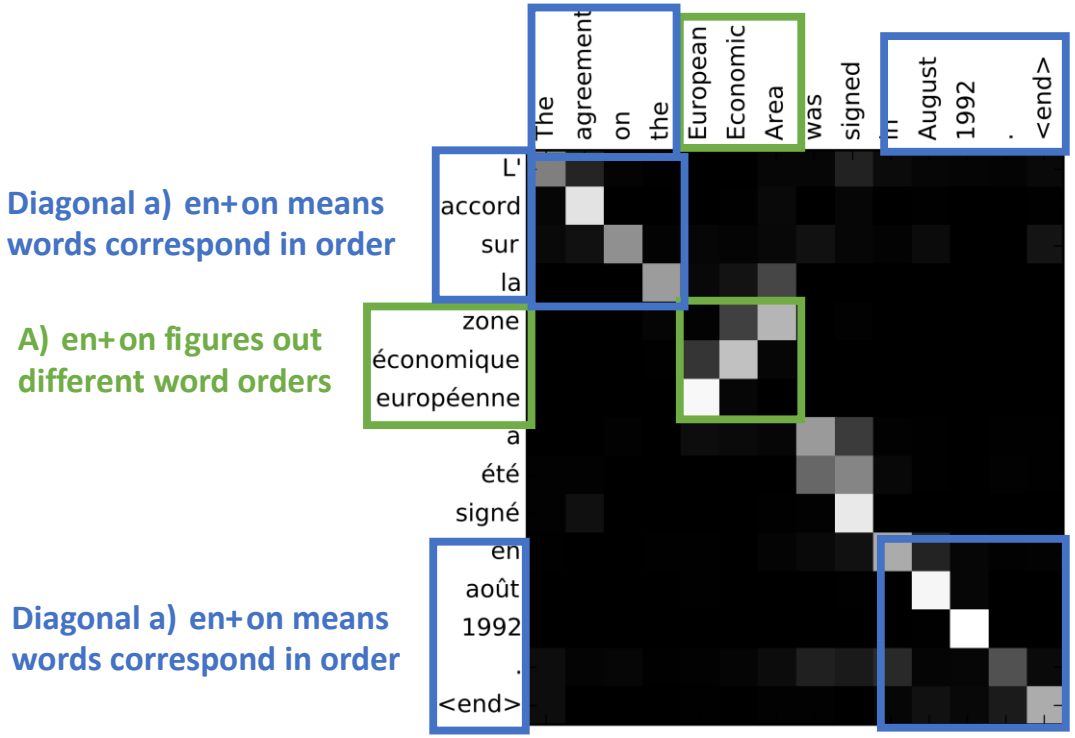
Sequence-to-Sequence with RNNs and Attention

Example: English to French transla<on

Input: "**The agreement on the European Economic Area** was signed **in August 1992**."

Output: "**L'accord sur la zone économique européenne** a été signé **en août 1992**."

Visualize a2 en<on weights $a_{t,i}$



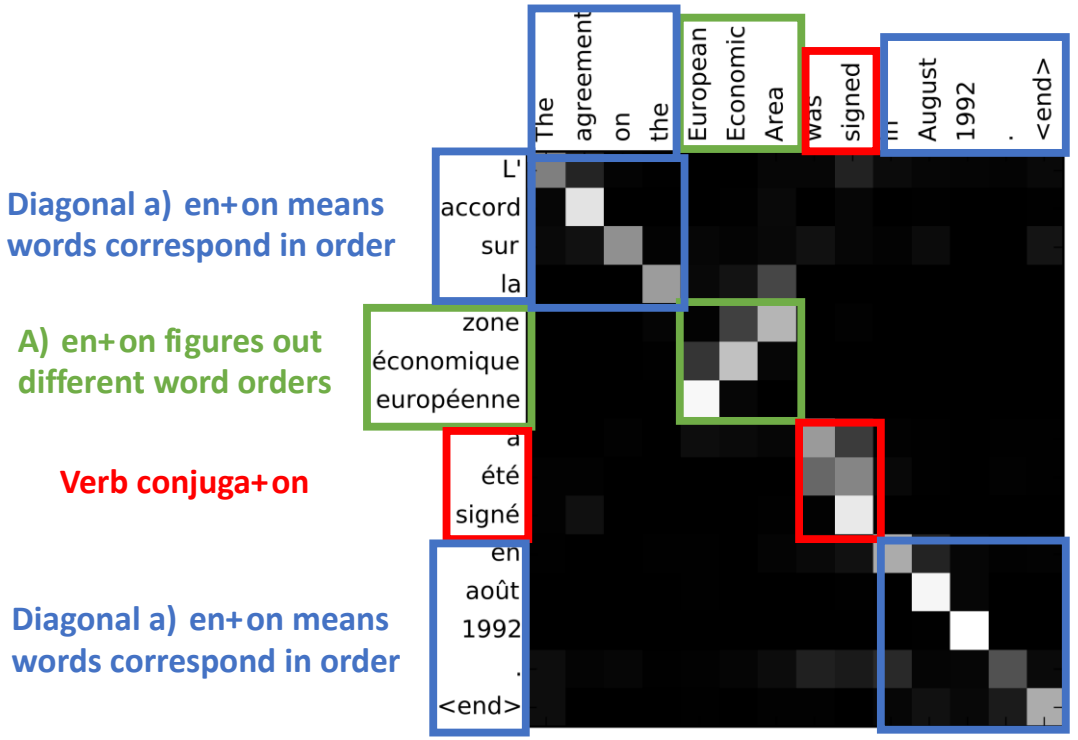
Sequence-to-Sequence with RNNs and Attention

Example: English to French transla<on

Input: "The agreement on the European Economic Area was signed in August 1992."

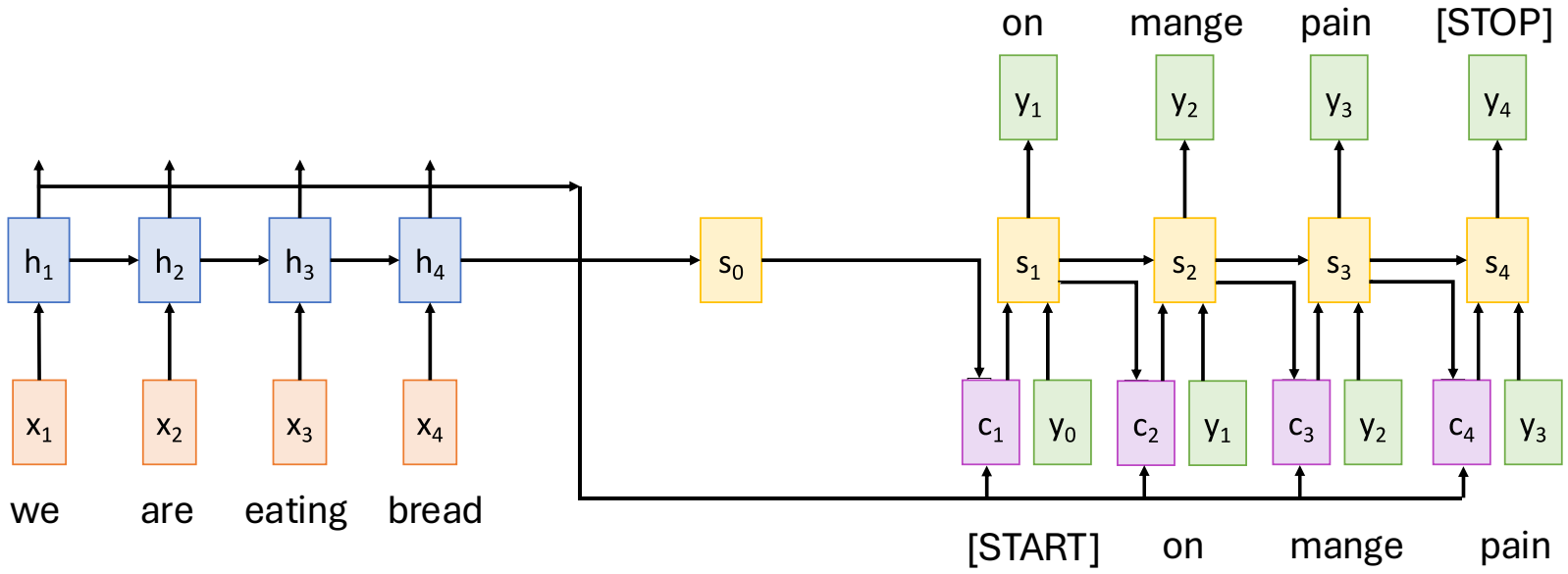
Output: "L'accord sur la zone économique européenne a été signé en août 1992."

Visualize a2 en<on weights $a_{t,i}$



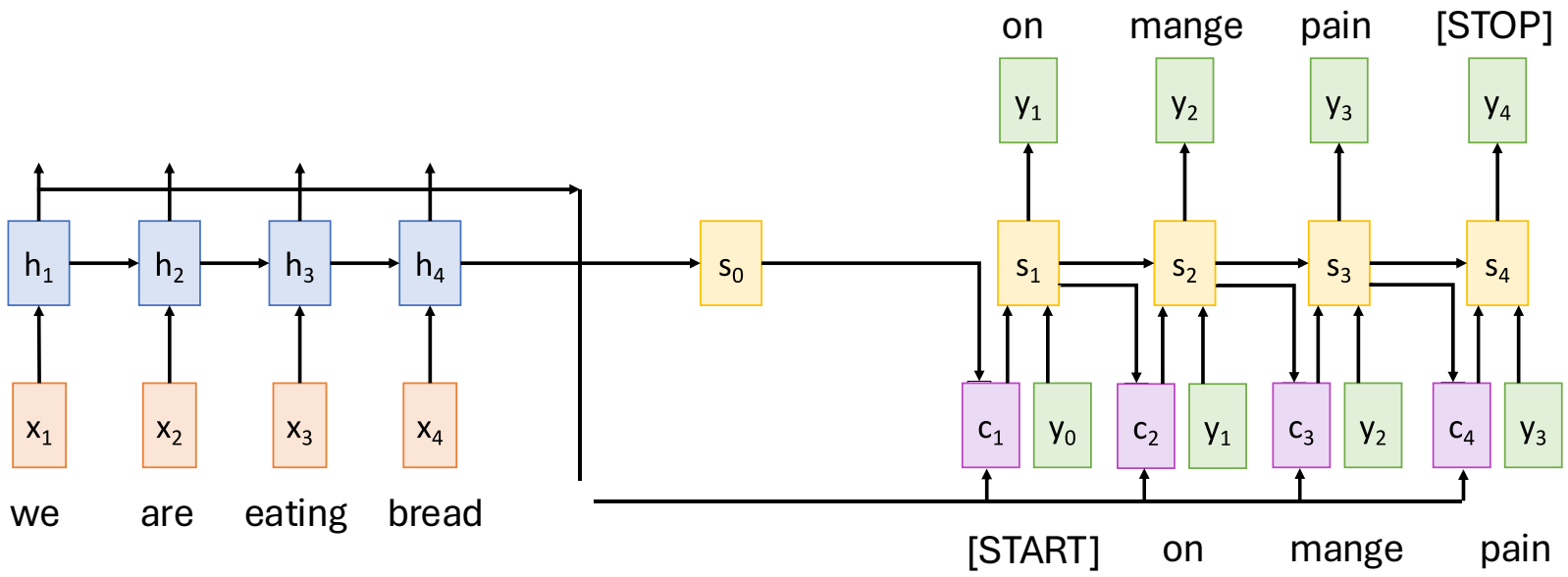
Pop Quiz

What can you notice from this architecture?



Pop Quiz Answer

The decoder doesn't use the fact that h_i form an ordered sequence
– it just treats them as an unordered set $\{h_i\}$



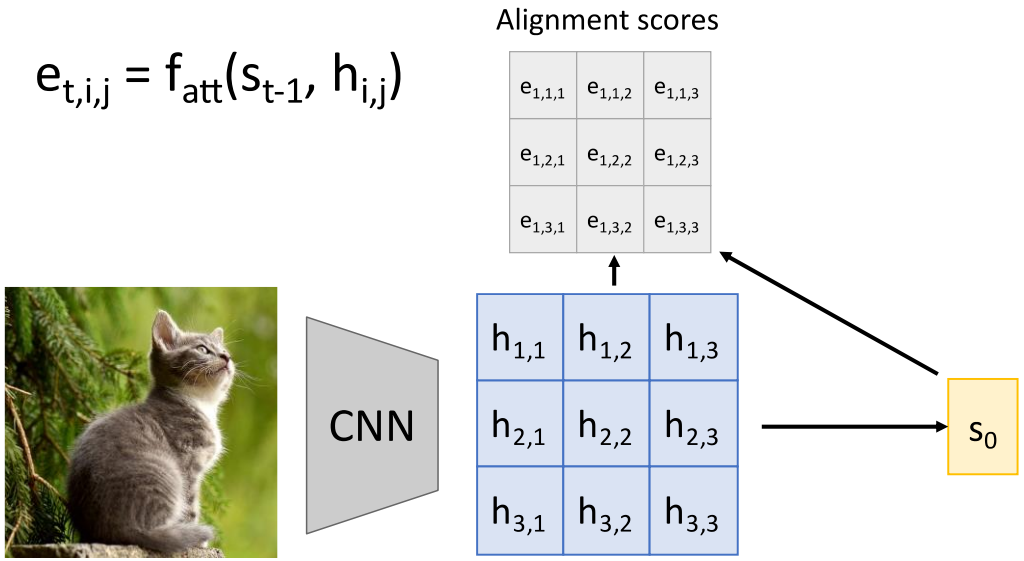
Can use similar architecture given any set of input hidden vectors $\{h_i\}$!

Outline: Part I

- **Sequence-to-Sequence**
  - Sequence-to-Sequence with RNNs
  - Sequence-to-Sequence with RNNs and Attention
  - **Image captioning**

Image Captioning

$e_{t,i,j} = f_{att}(s_{t-1}, h_{i,j})$



Use a CNN to compute a grid of features for an image

Image Captioning

$e_{t,i,j} = f_{att}(s_{t-1}, h_{i,j})$

$a_{t,:,:} = \text{softmax}(e_{t,:,:})$

$c_t = \sum_{i,j} a_{t,i,j} h_{i,j}$

CNN

Use a CNN to compute a grid of features for an image

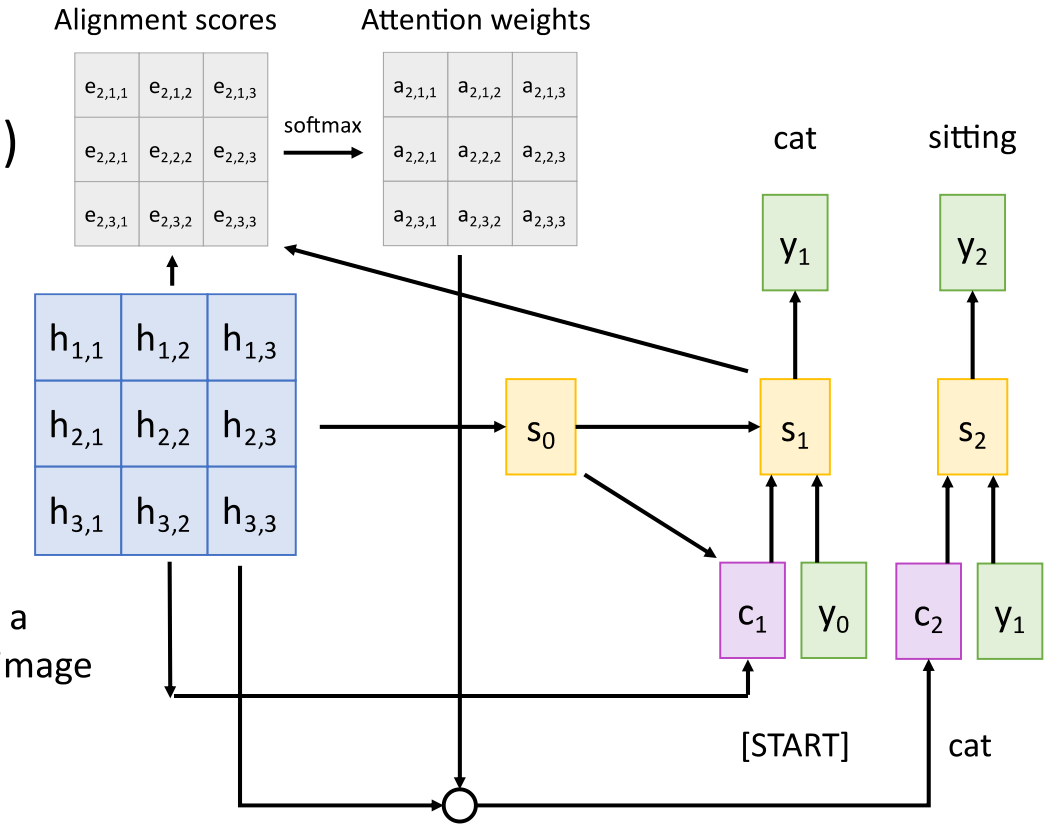


Image Captioning

$e_{t,i,j} = f_{att}(s_{t-1}, h_{i,j})$
$a_{t,:,:} = \text{softmax}(e_{t,:,:})$
$c_t = \sum_{i,j} a_{t,i,j} h_{i,j}$

Each timestep of decoder uses a different context vector that looks at different parts of the input image

CNN

$h_{1,1}$ $h_{1,2}$ $h_{1,3}$
$h_{2,1}$ $h_{2,2}$ $h_{2,3}$
$h_{3,1}$ $h_{3,2}$ $h_{3,3}$

Use a CNN to compute a grid of features for an image

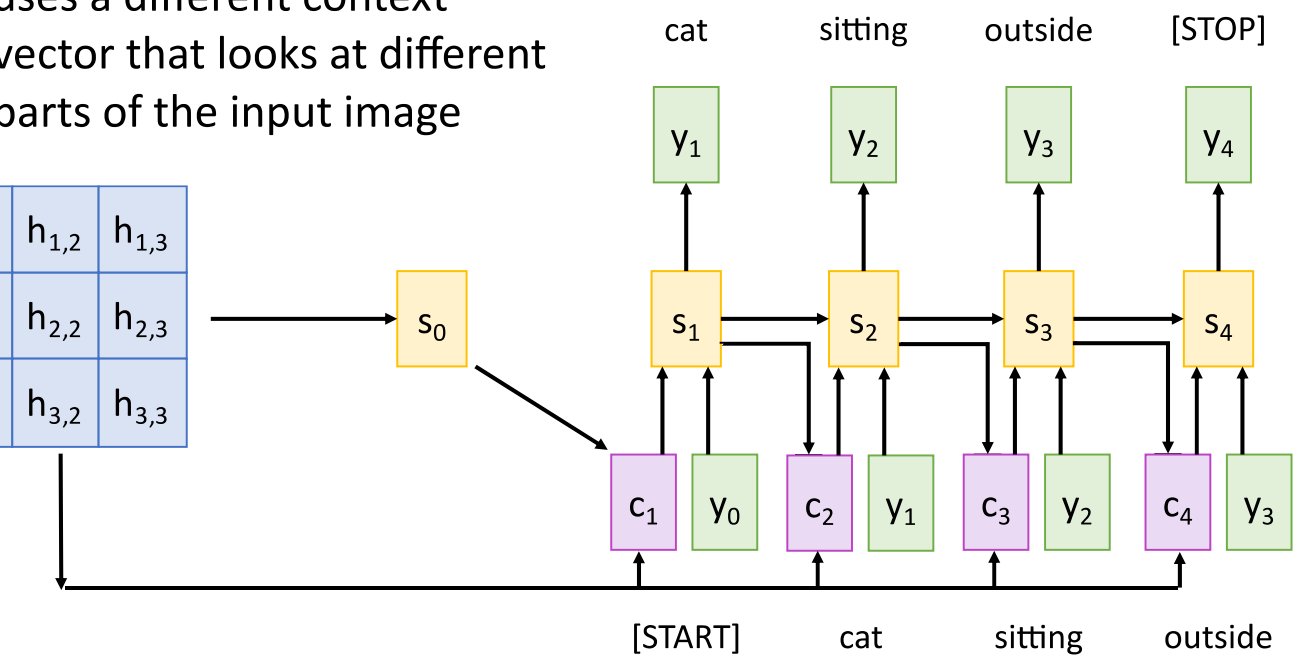


Image Captioning

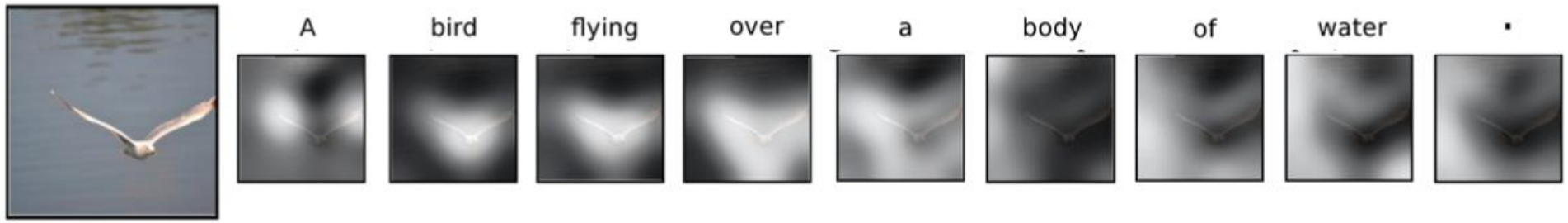


Image Captioning

A dog is standing on a hardwood floor.

A stop sign is on a road with a mountain in the background.

A group of people sitting on a boat in the water.

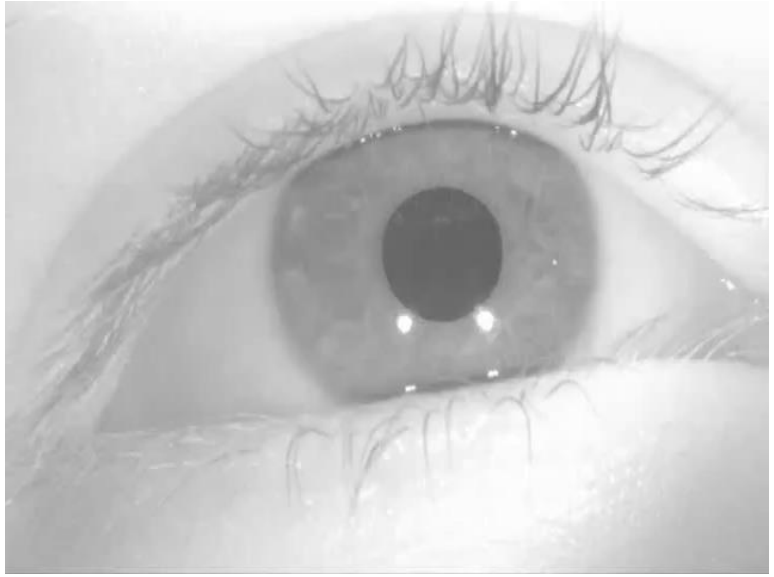
A giraffe standing in a forest with trees in the background.

Motivation

- Why do we do this?

Human Vision: Saccades

Human eyes are constantly moving so we don't no%ce



The **fovea** is a %ny region of the re%na that can see with high acuity

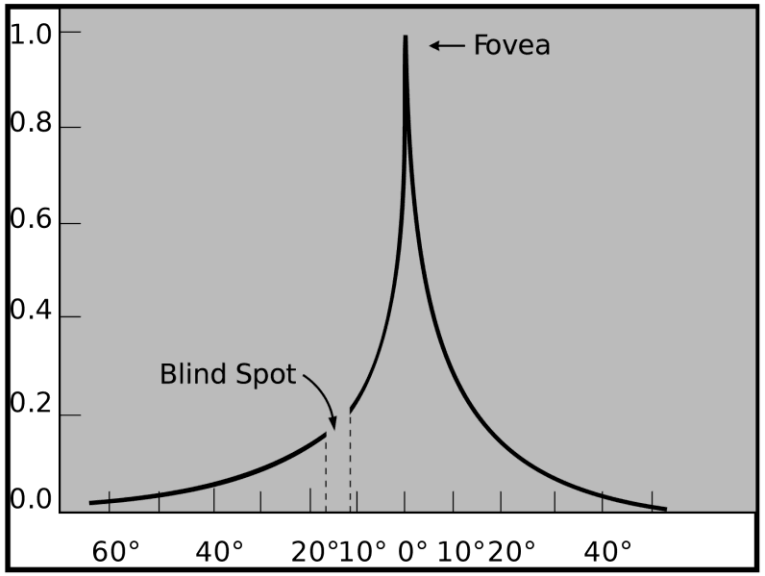
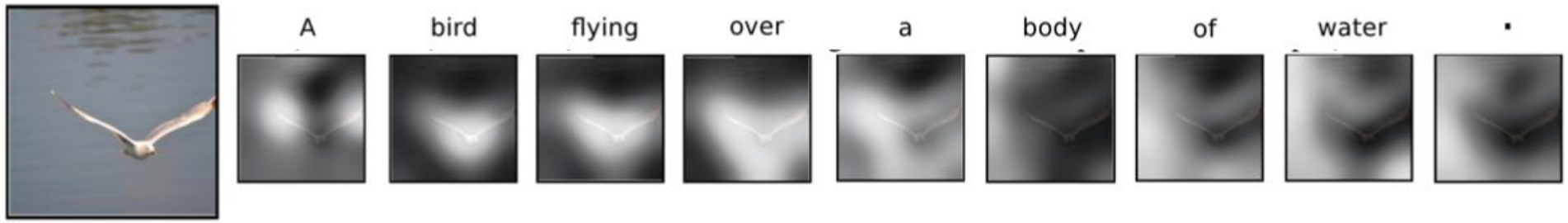
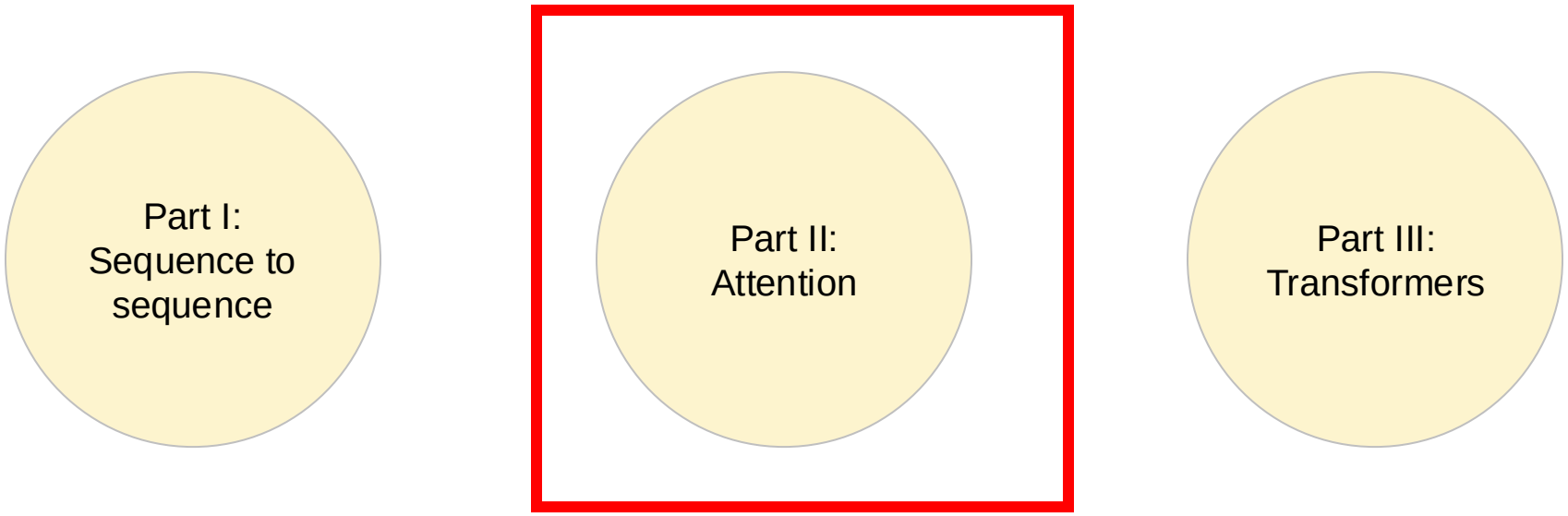


Image Captioning



Today's lecture



Some slides adapted from various resources: [Intro to Large Language Models, Andrej Karpathy, Fei-Fei Li, Xi Wang, VGG Oxford, Executive Education Polytechnique, Udemy, Deeplearning.ai, Stanford University CS231n, Financial Times, New York Times, Justin Johnson]

Outline: Part II

- **Attention and Self-attention**
  - Attention Layer
  - Self-attention layer
  - Properties of self-attention
  - Self-attention variations
  - Examples of CNNs with self-attention

Outline: Part II

- **Attention and Self-attention**
  - **Attention Layer**
  - Self-attention layer
  - Properties of self-attention
  - Self-attention variations
  - Examples of CNNs with self-attention

Attention Layer

Hidden states of output/decoder s_{t-1}

Inputs:

Query vector: q (Shape: D_Q)

Input vectors X (Shape: N_X x D_X)

Similarity function: f_{att}

Want to attend over hidden state of encoder h_i

Computation:

Similarities: e (Shape: N_X) e = f (q, X)

Attention weights: a = softmax(e) (Shape: N_X)

Output vector: $y = \sum_i a_i X_i$ (Shape: DX)

Attention Layer

Inputs:

Query vector: q (Shape: D_Q)

Input vectors X (Shape: N_X x D_X)

~~Similarity function: f_{att}~~

Similarity function: dot product

Computation:

~~Similarities: e (Shape: N_X) e = f (q, X)~~

Similarities: e (Shape: N_X) $e = q \cdot X$

Attention weights: a = softmax(e) (Shape: N_X)

Output vector: $y = \sum_i a_i X_i$ (Shape: D_X)

Attention Layer

Inputs:

Query vector: q (Shape: D_Q)

Input vectors X (Shape: N_X x D_X)

~~Similarity function: f_{att}~~

~~**Similarity** function: dot product~~

Similarity function: scaled dot product

Computation:

~~Similarities: e (Shape: N_x) e = f (q, X)~~

~~**Similarities**: e (Shape: N_x) e = q · X~~

Similarities: e (Shape: N_X) **e = q · X / sqrt (D_Q)**

Attention weights: a = softmax(e) (Shape: N_X)

Output vector: y = $\sum_i a_i X_i$ (Shape: D_X)

Pop Quiz

Inputs:

Query vector: q (Shape: D_Q)

Input vectors X (Shape: N_X x D_X)

~~Similarity function: f_{att}~~

~~**Similarity** function: dot product~~

Similarity function: scaled dot product

Computation:

~~Similarities: e (Shape: N_x) e = f (q, X)~~

~~**Similarities**: e (Shape: N_x) e = q · X~~

Similarities: e (Shape: N_X) **e = q · X / sqrt (D_Q)**

Attention weights: a = softmax(e) (Shape: N_X)

Output vector: y = $\sum_i a_i X_i$ (Shape: D_X)

Why do we do this scaling?

Pop Quiz - Hint

Inputs:

Query vector: q (Shape: D_Q)

Input vectors X (Shape: N_X x D_X)

~~Similarity function: f_{att}~~

~~**Similarity** function: dot product~~

Similarity function: scaled dot product

Computation:

~~Similarities: e (Shape: N_X) e = f (q, X)~~

~~**Similarities**: e (Shape: N_X) e = q · X~~

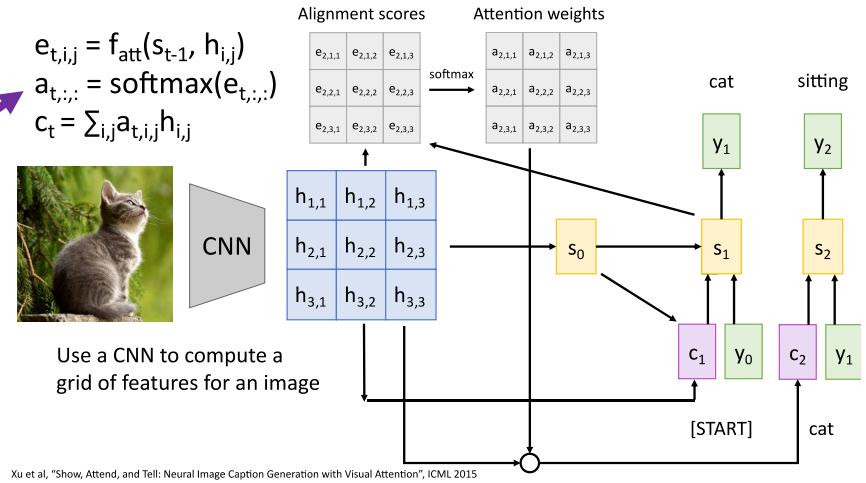
Similarities: e (Shape: N_X) **e = q · X / sqrt (D_Q)**

Attention weights: a = softmax(e) (Shape: N_X)

Output vector: y = $\sum_i a_i X_i$ (Shape: D_X)

Why do we do this scaling?

Hint:
Think of what follows!



Pop Quiz Answers

Inputs:

Query vector: q (Shape: D_Q)

Input vectors X (Shape: N_X x D_X)

~~Similarity function: f_{att}~~

~~**Similarity** function: dot product~~

Similarity function: scaled dot product

Computation:

~~Similarities: e (Shape: N_X) e = f (q, X)~~

~~**Similarities**: e (Shape: N_X) e = q · X~~

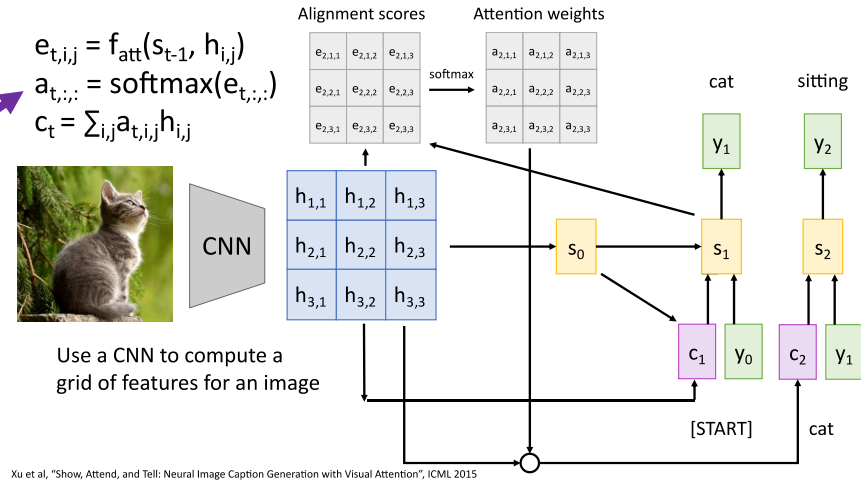
Similarities: e (Shape: N_X) **e = q · X / sqrt (D_Q)**

Attention weights: a = softmax(e) (Shape: N_X)

Output vector: y = $\sum_i a_i X_i$ (Shape: D_X)

Why do we do this scaling?

Hint:
Think of what follows!



Softmax follows: →

- Large elements → vanishing gradients
- One large element → highly peaked softmax distribution

Attention Layer

Inputs:

~~**Query** vector: q (Shape: D_Q)~~

Query vectors: Q (Shape: N_Q x D_Q)

Input vectors X (Shape: N_X x D_X)

~~Similarity function: f_{att}~~

~~**Similarity** function: dot product~~

Similarity function: scaled dot product

Computation:

~~Similarities: e (Shape: N_X) e = f (q, X)~~

~~**Similarities**: e (Shape: N_X) e = q · X~~

Similarities: e (Shape: N_X) **e = q · X / sqrt (D_Q)**

Attention weights: a = softmax(e) (Shape: N_X)

Output vector: y = $\sum_i a_i X_i$ (Shape: D_X)

Single query vector: at each time step of decoder

→

Multiple query vectors

Attention Layer

Inputs:

~~**Query** vector: q (Shape: D_Q)~~

Query vectors: Q (Shape: N_Q x D_Q)

Input vectors X (Shape: N_X x D_X)

~~Similarity function: f_{att}~~

~~**Similarity** function: dot product~~

Similarity function: **scaled dot product**

Computation:

~~Similarities: e (Shape: N_X) $e = f(q, X)$~~

~~**Similarities**: e (Shape: N_X) $e = q \cdot X$~~

~~**Similarities**: e (Shape: N_X) **$e = q \cdot X / \text{sqrt}(D_Q)$**~~

Similarities: e (Shape:N_Q x N_X) **$E = Q \cdot X^T$**

$\Rightarrow E_{i,j} = Q_i \cdot X_j / \text{sqrt}(D_Q)$

Attention weights: a = softmax(e) (Shape: N_X)

Output vector: $y = \sum_i a_i X_i$ (Shape: D_X)

For each query vector: probability distribution over input vectors

→

similarity between each query and each input

→

compute all similarity scores simultaneously with single matrix multiplication

Attention Layer

Inputs:

~~**Query** vector: q (Shape: D_Q)~~

Query vectors: Q (Shape: N_Q x D_Q)

Input vectors X (Shape: N_X x D_X)

~~Similarity function: f_{att}~~

~~**Similarity** function: dot product~~

Similarity function: **scaled dot product**

Computation:

~~Similarities: e (Shape: N_X) e = f (q, X)~~

~~**Similarities**: e (Shape: N_X) e = q · X~~

~~**Similarities**: e (Shape: N_X) **e = q · X / sqrt (D_Q)**~~

Similarities: e (Shape:N_Q x N_X) **E =** Q · X^T

=> $E_{i,j} = Q_i \cdot X_j / \text{sqrt}(D_Q)$

~~**Attention weights**: a = softmax(e) (Shape: N_X)~~

Attention weights: a = softmax(E, dim=1) (Shape: N_Q x N_X)

Output vector: y = $\sum_i a_i X_i$ (Shape: D_X)

For each query vector: compute distribution over input vectors

→

softmax over output attention scores E over only of dimension

Attention Layer

Inputs:

~~**Query** vector: q (Shape: D_Q)~~

Query vectors: Q (Shape: N_Q x D_Q)

Input vectors X (Shape: N_X x D_X)

~~Similarity function: f_{att}~~

~~**Similarity** function: dot product~~

Similarity function: **scaled dot product**

Computation:

~~Similarities: e (Shape: N_X) e = f (q, X)~~

~~**Similarities**: e (Shape: N_X) e = q · X~~

~~**Similarities**: e (Shape: N_X) **e = q · X / sqrt (D_Q)**~~

Similarities: e (Shape:N_Q x N_X) **E =** Q · X^T

=> $E_{i,j}$ = Q_i · X_j / sqrt (D_Q)

~~**Attention weights**: a = softmax(e) (Shape: N_X)~~

Attention weights: A = softmax(E, dim=1)

(Shape: N_Q x N_X)

~~**Output** vector: y = $\sum_i a_i X_i$ (Shape: D_X)~~

Output vector: Y =AX, (Shape: N_Q x D_X)

$Y_i = \sum_j A_{i,j} X_j$

One output vector for each query vector

→

compute linear combinations simultaneously (single matrix multiplication)

Attention Layer

Inputs:

Query vectors: Q (Shape: N_Q x D_Q)

***Input** vectors X (Shape: N_X x D_X)*

Similarity function: **scaled dot product**

Computation:

Similarities: e (Shape:N_Q x N_X) **E =** Q · X^T

=> $E_{i,j}$ = Q_i · X_j / sqrt (D_Q)

Attention weights: A = softmax(E, dim=1)

(Shape: N_Q x N_X)

Output vector: Y =AX, (Shape: N_Q x D_X)

=> $Y_i = \sum_j A_{i,j} X_j$

Split input vectors X

→

1. compute attention weights with every Q
2. produce the output Y

Attention Layer

Inputs:

Query vectors: Q (Shape: N_Q x D_Q)

Input vectors X (Shape: N_X x D_X)

Key Matrix: W_K (Shape: D_X x D_Q)

Value Matrix: W_V (Shape: D_X x D_V)

Split input vectors X

→

1. keys K for similarity scores
2. values V for output scores

Computation:

Key vectors: $K = XW_K$ (Shape: N_X x D_Q)

Value vectors: $V = XW_V$ (Shape: N_X x D_V)

~~**Similarities**: (Shape:N_Q x N_X) **E =** $Q \cdot X^T$~~

~~$\Rightarrow E_{i,j} = Q_i \cdot X_j$ / sqrt (D_Q)~~

Similarities: (Shape:N_X x N_X) **E =** $Q \cdot K^T$

=> $E_{i,j} = Q_i \cdot K_j$ / sqrt (D_Q)

Attention weights: A = softmax(E, dim=1)

(Shape: N_Q x N_X)

~~**Output** vector: Y =AX, (Shape: N_Q x D_X)~~

~~$\Rightarrow Y_i = j \sum A_{i,j} X_j$~~

Output vector: Y =AV, (Shape: N_X x D_V)

=> $Y_i = \sum_j A_{i,j} V_j$

Attention Layer

Inputs:

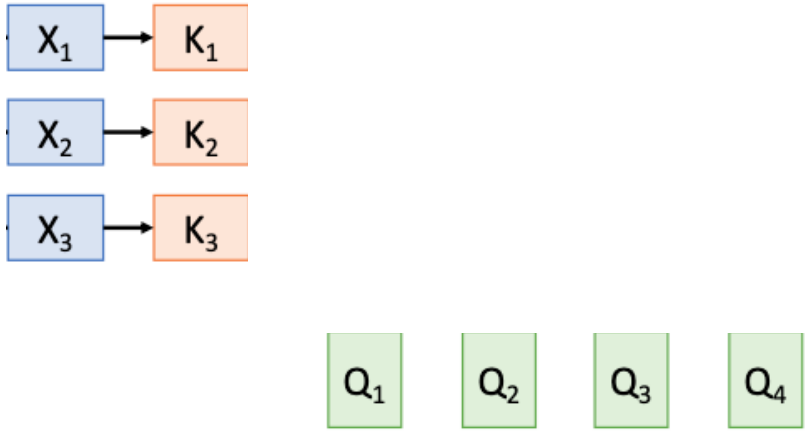
Query vectors: Q (Shape: N_Q x D_Q)
Input vectors X (Shape: N_X x D_X)
Key Matrix: W_K (Shape: D_X x D_Q)
Value Matrix: W_V (Shape: D_X x D_V)

Computation:

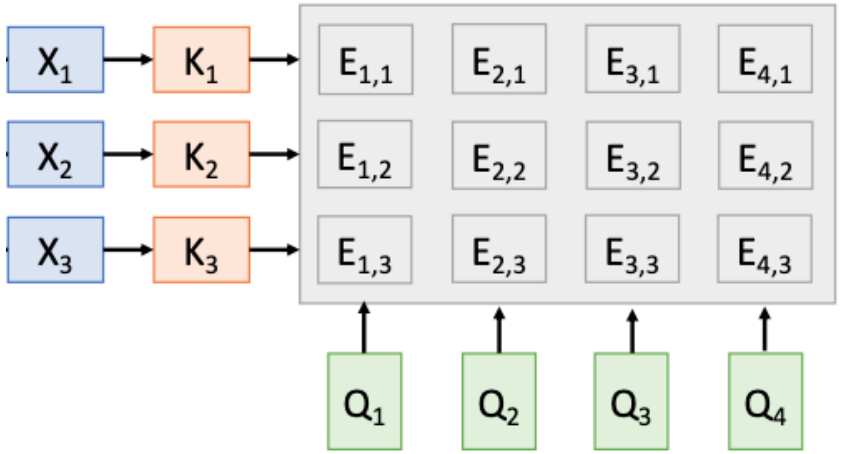
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: (Shape:N_X x N_X) **$E = Q \cdot K^T$**
=> $E_{i,j} = Q_i \cdot K_j$ / sqrt (D_Q)
Attention weights: A = softmax(E, dim=1)
(Shape: N_Q x N_X)
Output vector: Y =AV, (Shape: N_X x D_Q)
=> $Y_i = \sum_j A_{i,j} V_j$

+ more flexibility

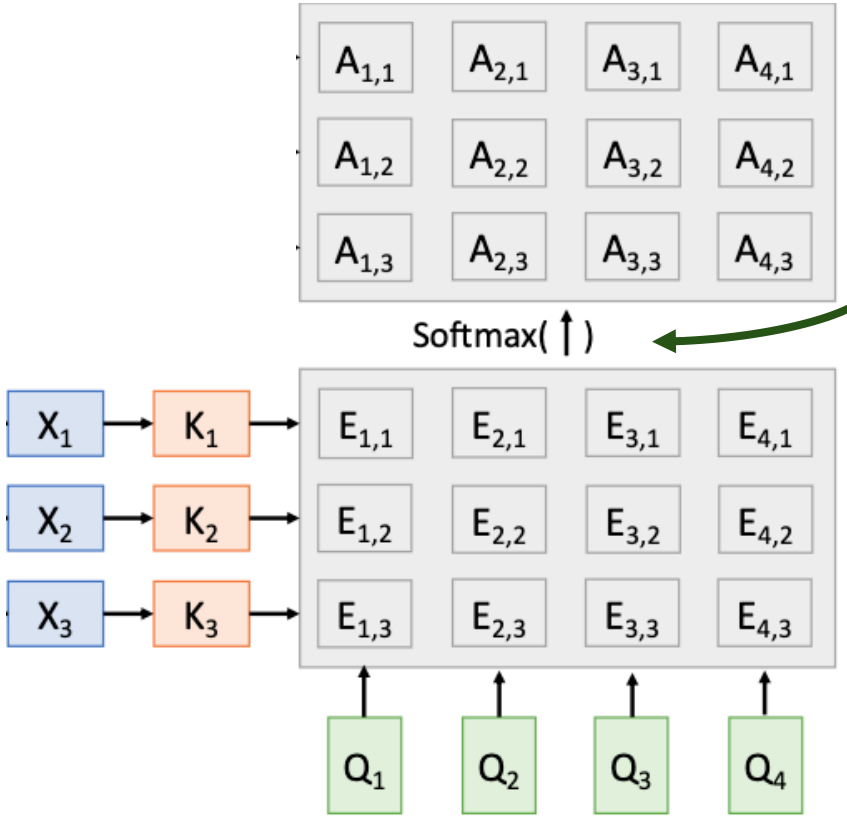
Attention Layer



Attention Layer



Attention Layer

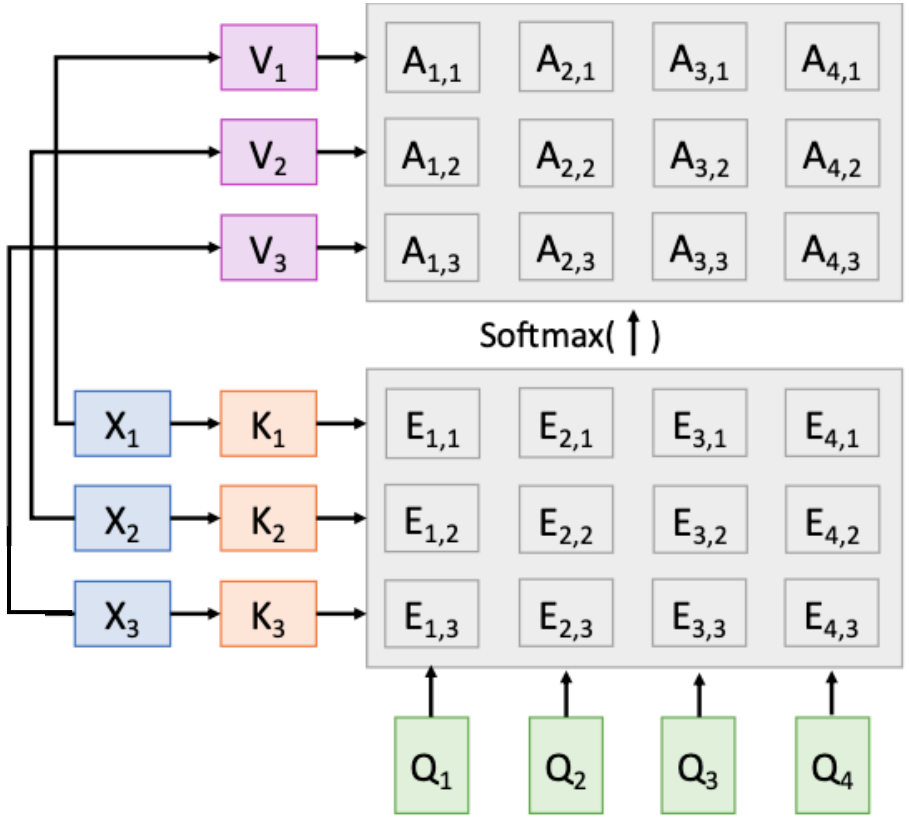


Softmax over **vertical dim** of E

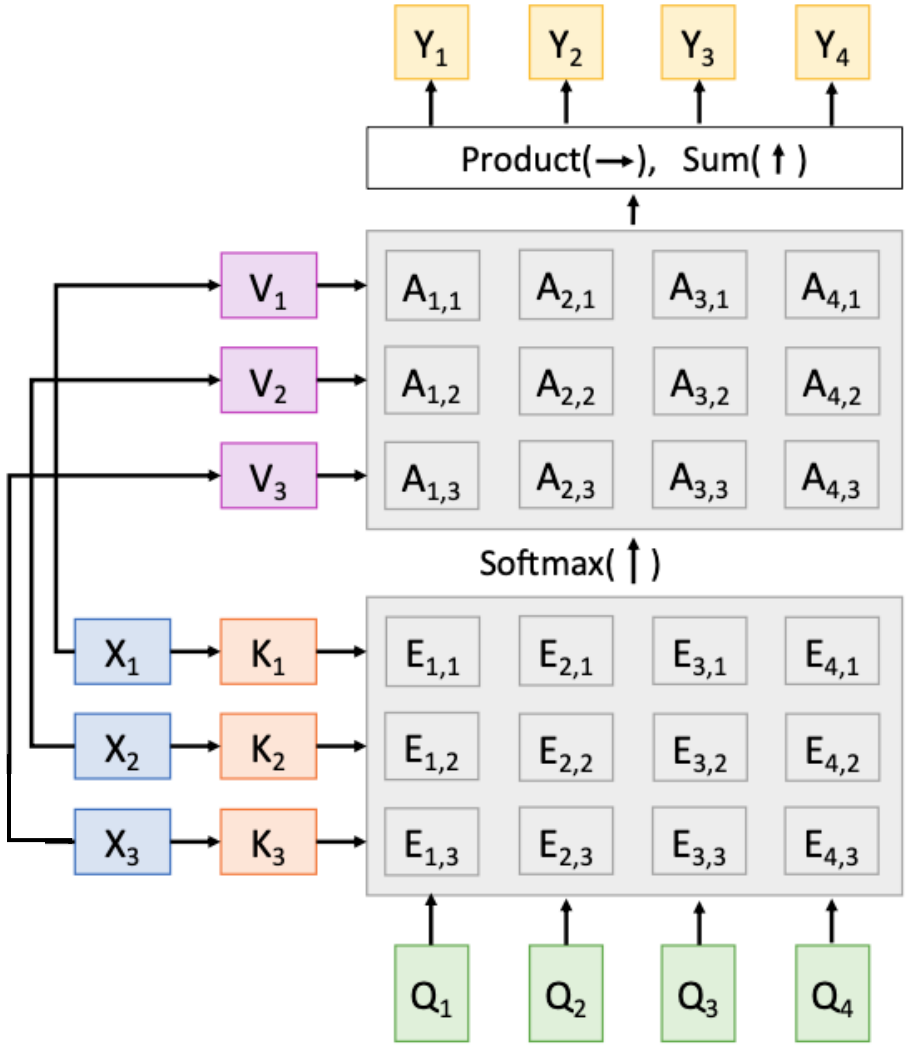
→

each column of A: probability distribution over all inputs X1, X2, X3

Attention Layer



Attention Layer



Outline: Part II

- **Attention and Self-attention**
  - Attention Layer
  - **Self-attention layer**
  - Properties of self-attention
  - Self-attention variations
  - Examples of CNNs with self-attention

Self-Attention Layer

One query per input vector

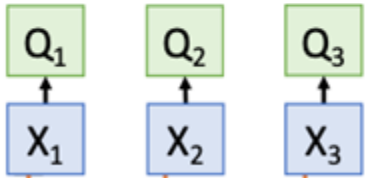
Inputs:

Query vectors: Q (Shape: N_Q x D_Q)
Input vectors X (Shape: N_X x D_X)
Key Matrix: W_K (Shape: D_X x D_Q)
Value Matrix: W_V (Shape: D_X x D_V)
Query Matrix: W_Q (Shape: D_X x D_V)

Computation:

Query vectors: $Q=XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: (Shape:N_X x N_X) **$E = Q \cdot K^T$**
=> $E_{i,j} = Q_i \cdot K_j$ / sqrt (D_Q)
Attention weights: A = softmax(E, dim=1)
(Shape: N_Q x N_X)
Output vectors: Y =AV, (Shape: N_X x D_V)
=> $Y_i = \sum_j A_{i,j} V_j$

Self-Attention Layer



One query per input vector

Inputs:

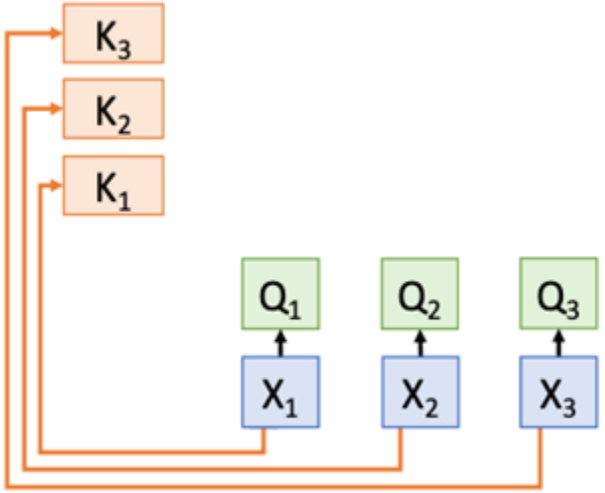
Query vectors: Q (Shape: N_Q x D_Q)
Input vectors X (Shape: N_X x D_X)
Key Matrix: W_K (Shape: D_X x D_Q)
Value Matrix: W_V (Shape: D_X x D_V)
Query Matrix: W_Q (Shape: D_X x D_V)

Computation:

Query vectors: $Q=XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: (Shape:N_X x N_X) **$E = Q \cdot K^T$**
=> $E_{i,j} = Q_i \cdot K_j$ / sqrt (D_Q)
Attention weights: A = softmax(E, dim=1)
(Shape: N_Q x N_X)
Output vectors: Y =AV, (Shape: N_X x D_V)
=> $Y_i = \sum_j A_{i,j} V_j$

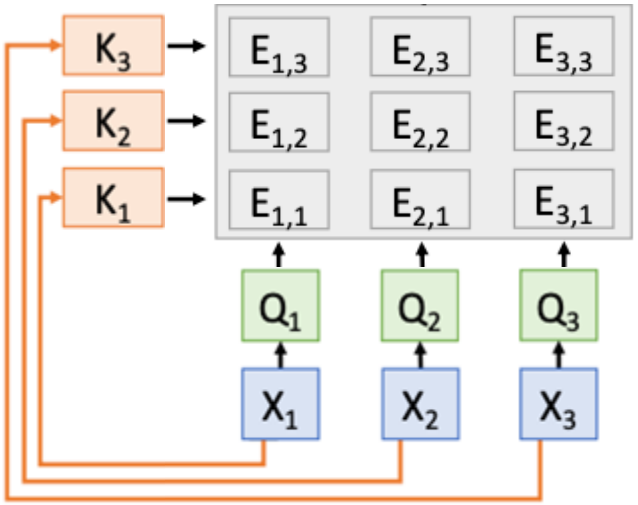
Self-Attention Layer

One query per input vector



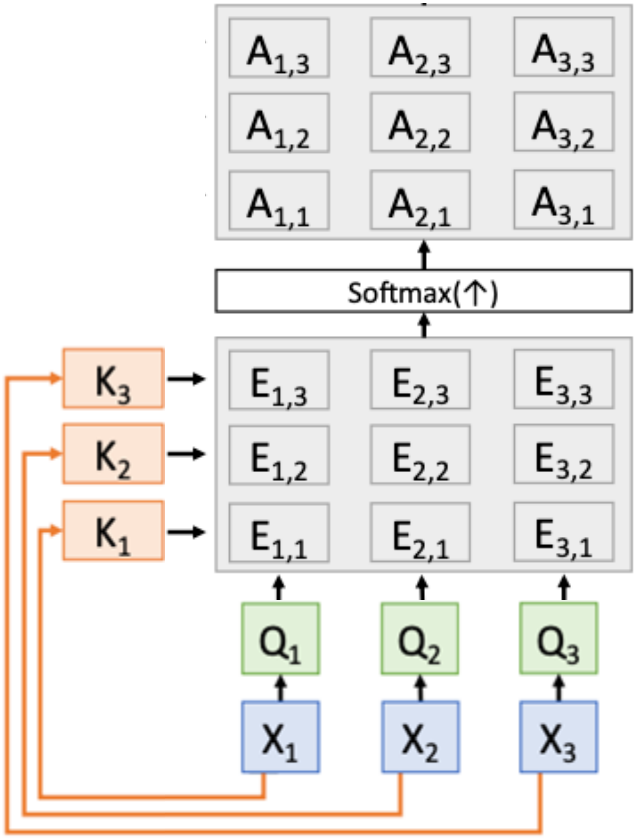
Self-Attention Layer

One query per input vector



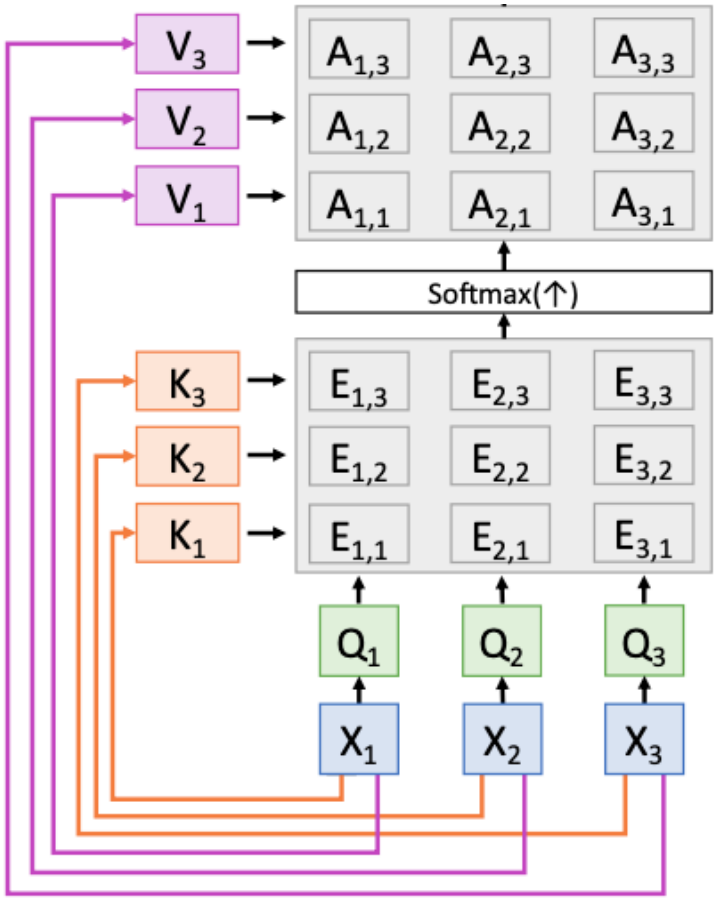
Self-Attention Layer

One query per input vector

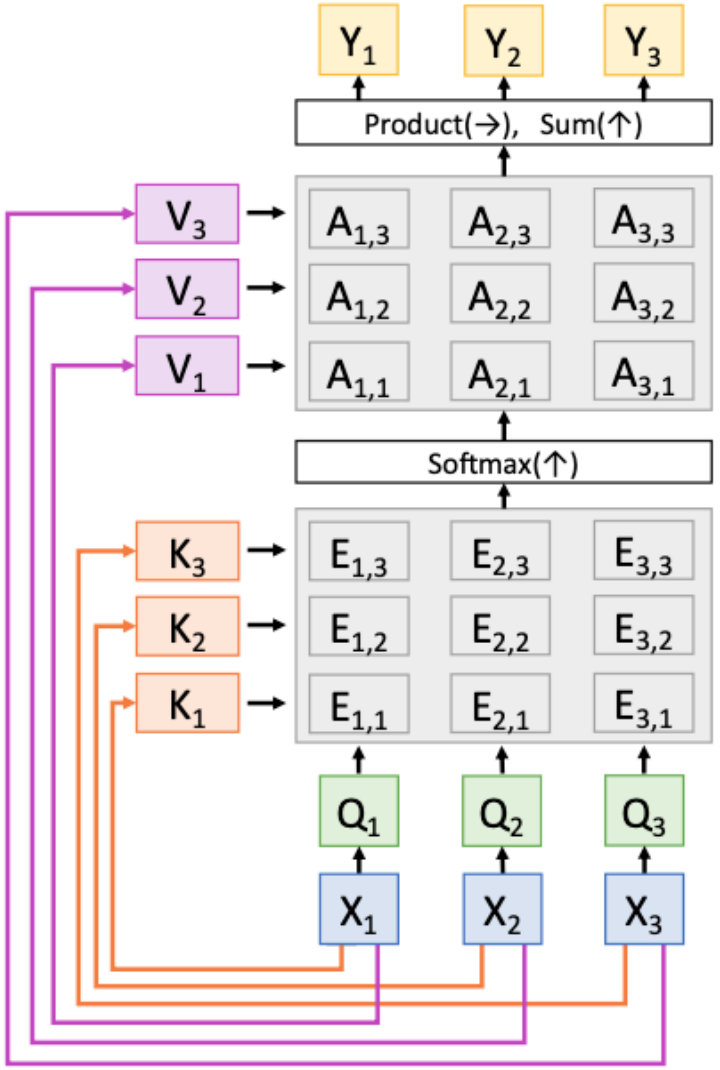


Self-Attention Layer

One query per input vector



Self-Attention Layer



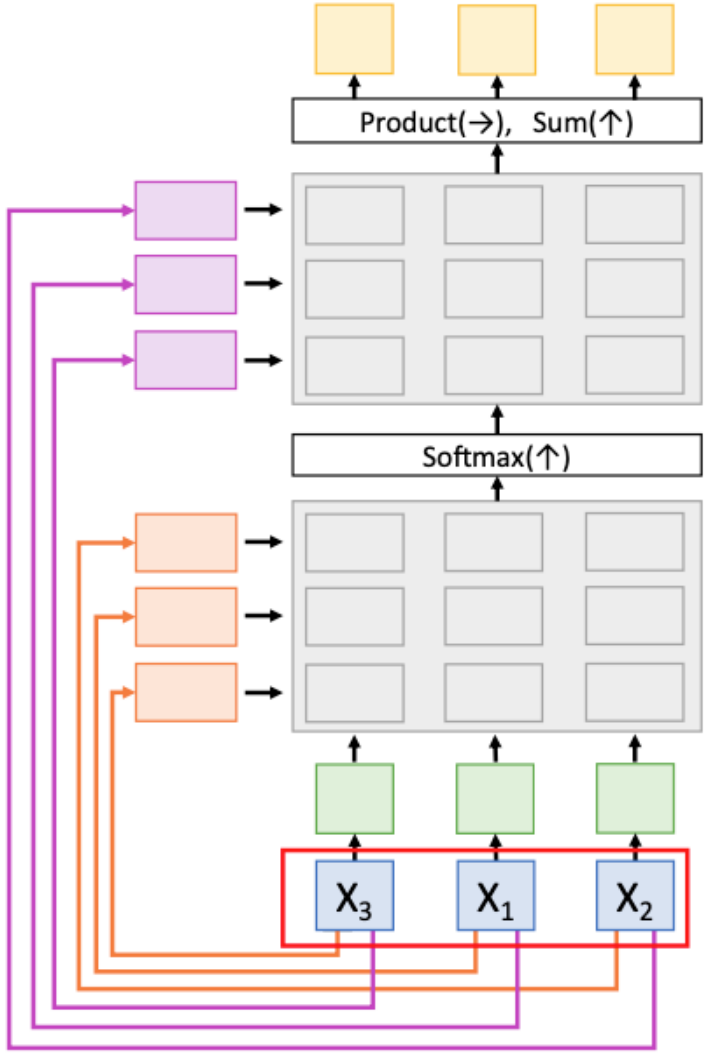
One query per input vector

New layer
→
comparing each input vector with each other in a non-linear way,
as decided by the network

Outline: Part II

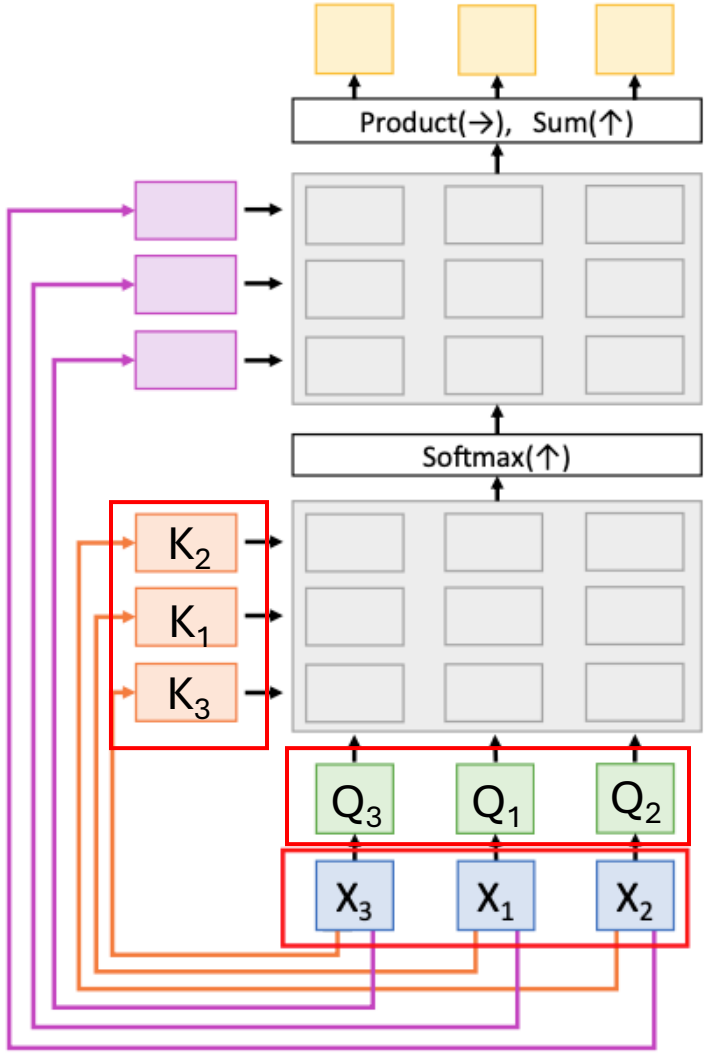
- **Attention and Self-attention**
  - Attention Layer
  - Self-attention layer
  - **Properties of self-attention**
  - Self-attention variations
  - Examples of CNNs with self-attention

Attention Layer - Properties



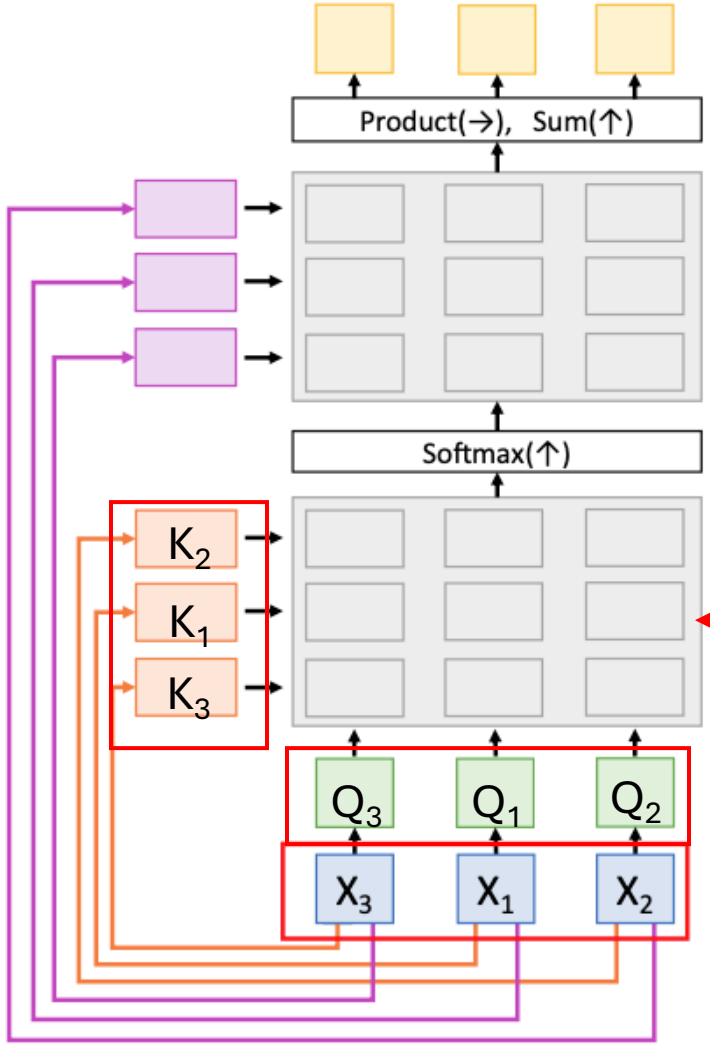
Permute input vectors X

Attention Layer - Properties



Permute input vectors X

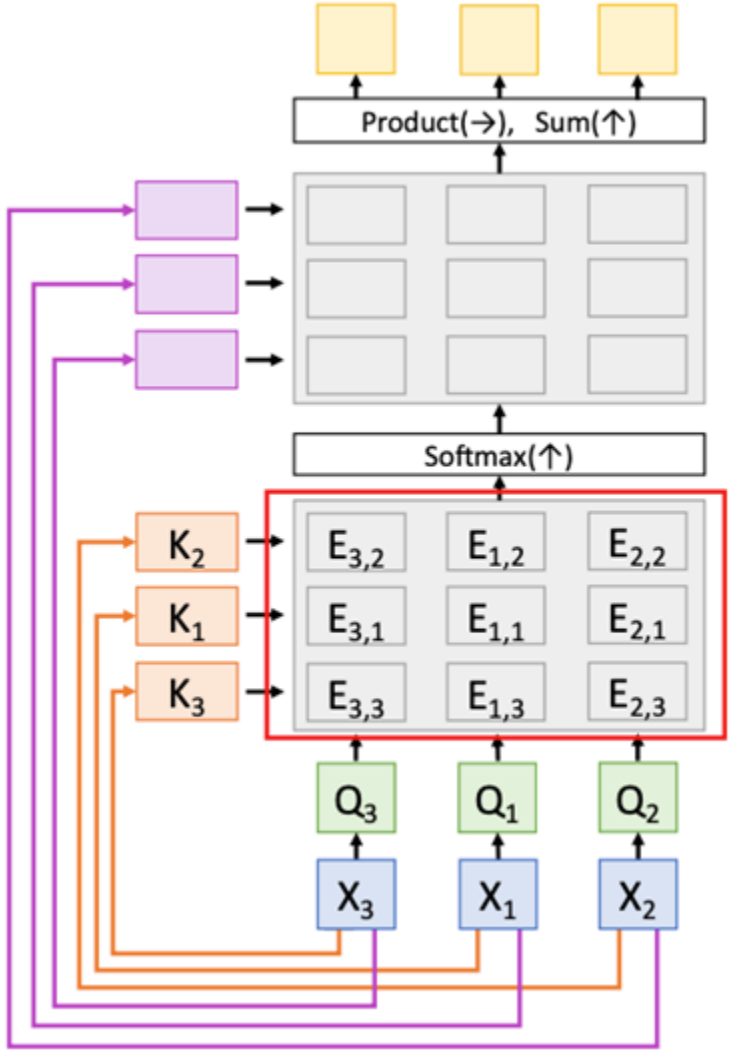
Pop Quiz



Permute input vectors X

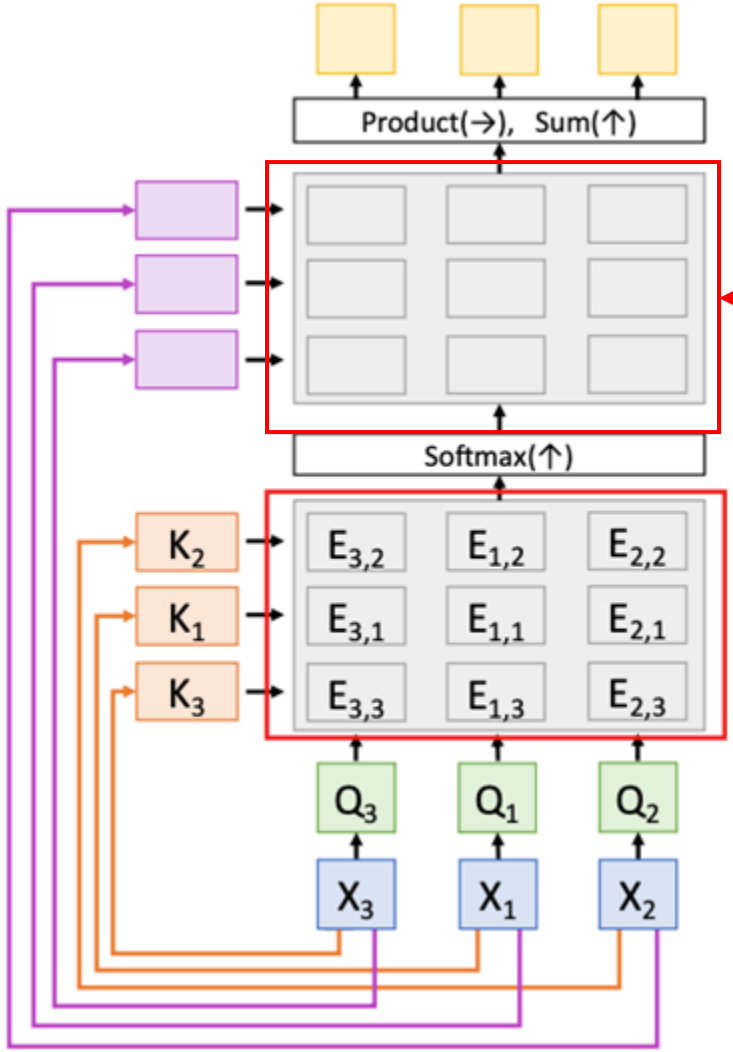
What is going to happen with the similarity scores?

Attention Layer - Properties



Permute input vectors X

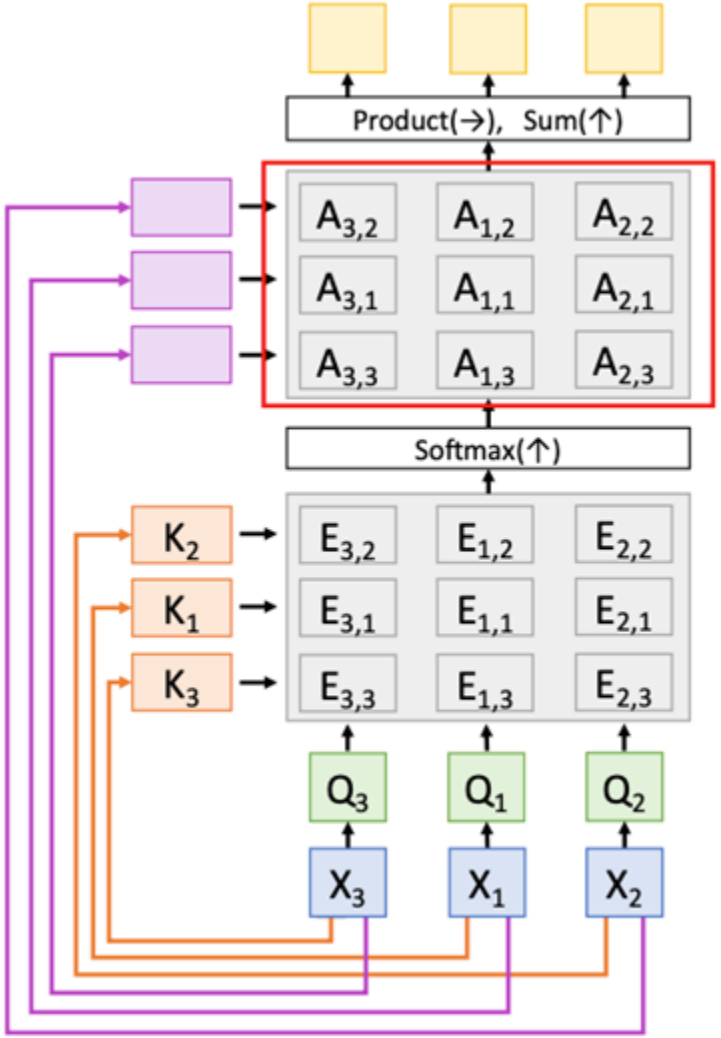
Pop Quiz



Permute input vectors X

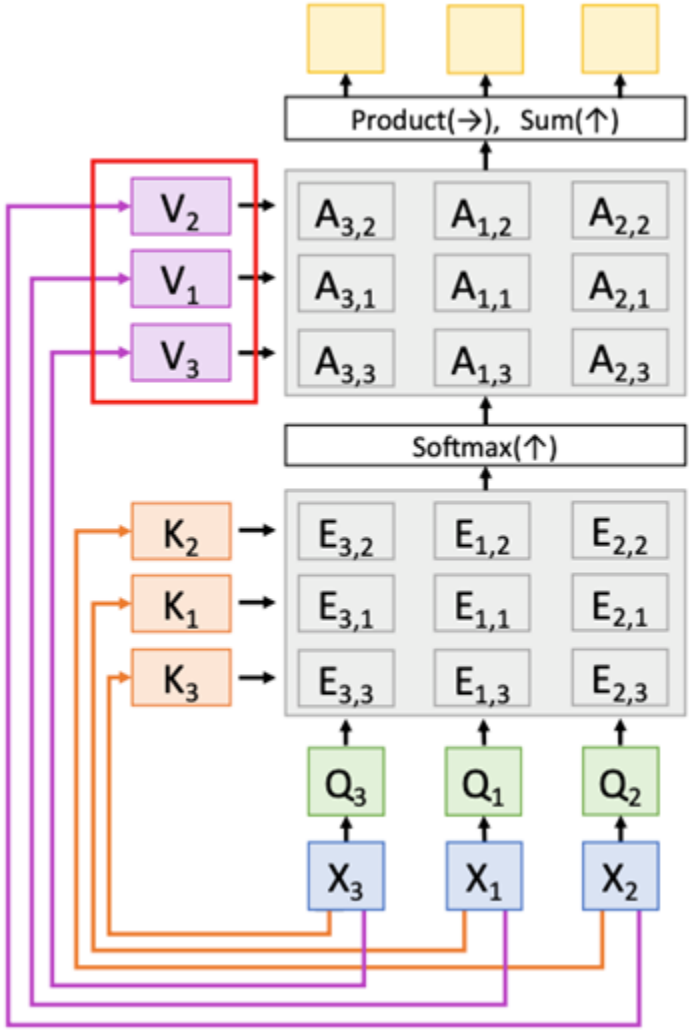
What is going to happen with the attention weights?

Attention Layer - Properties



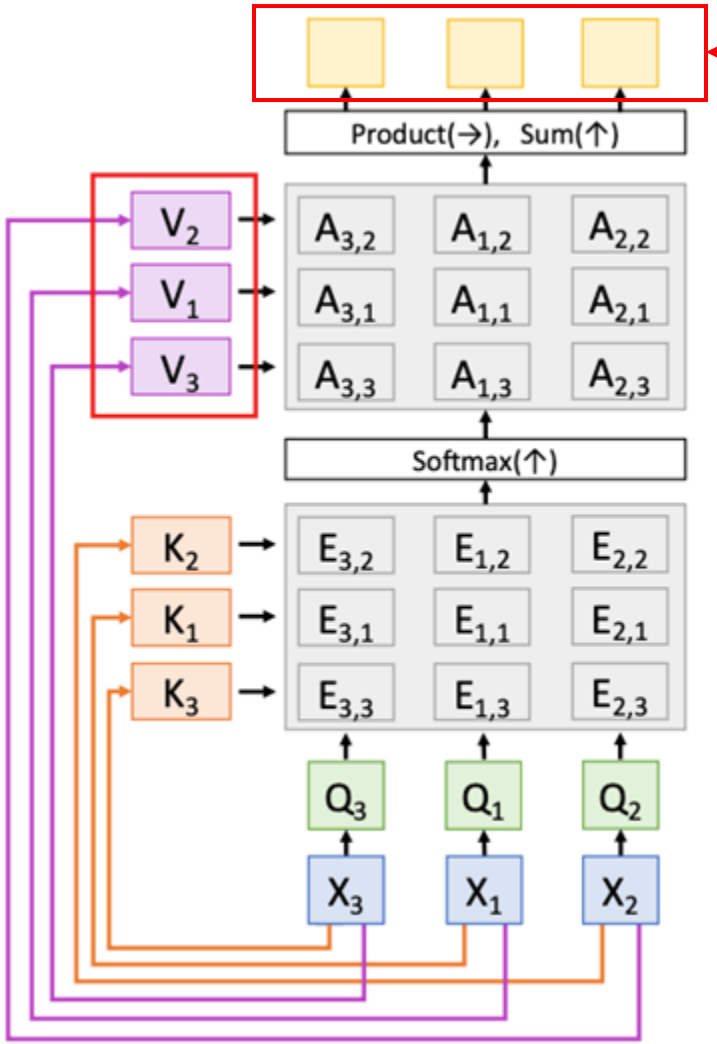
Permute input vectors X

Attention Layer - Properties



Permute input vectors X

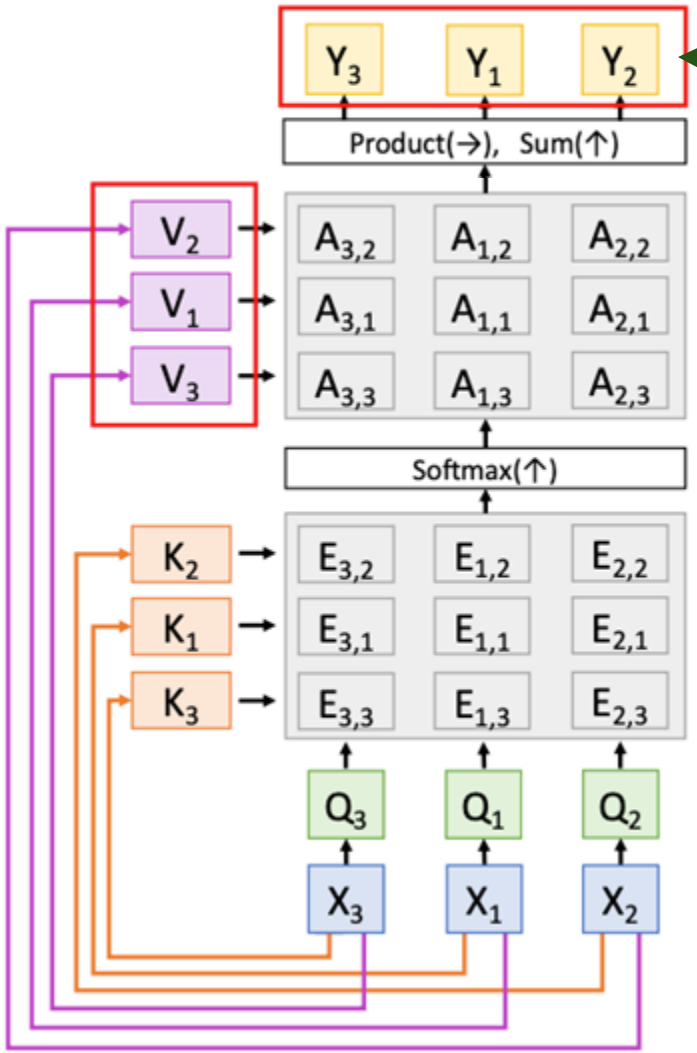
Attention Layer - Properties



Permute input vectors X

What is going to happen with the output vectors Y?

Attention Layer - Properties



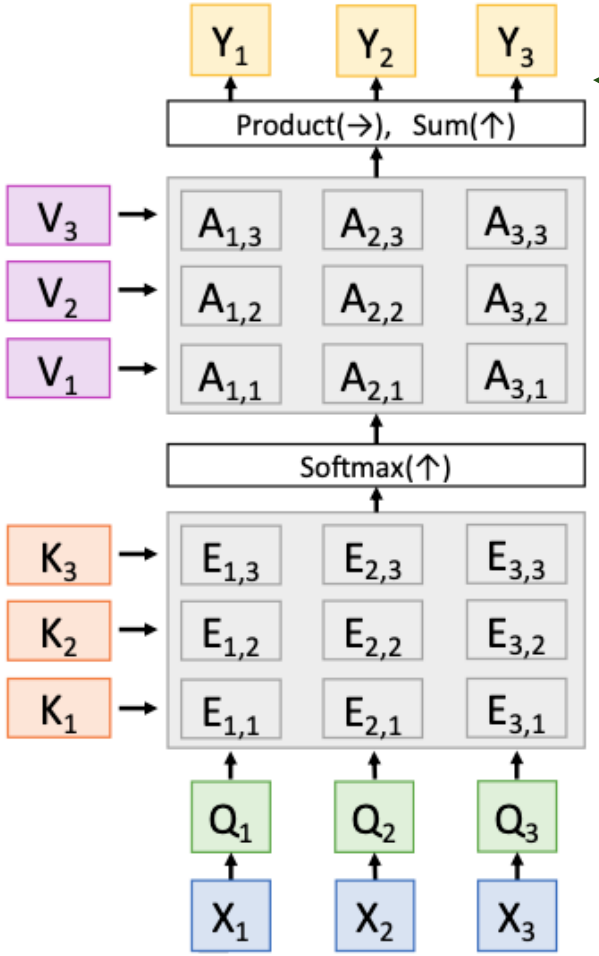
Permute input vectors X

Permuted inputs → Permuted outputs

→

Self-attention layer → Permutation equivalent

Attention Layer - Properties



Permuted inputs → Permuted outputs

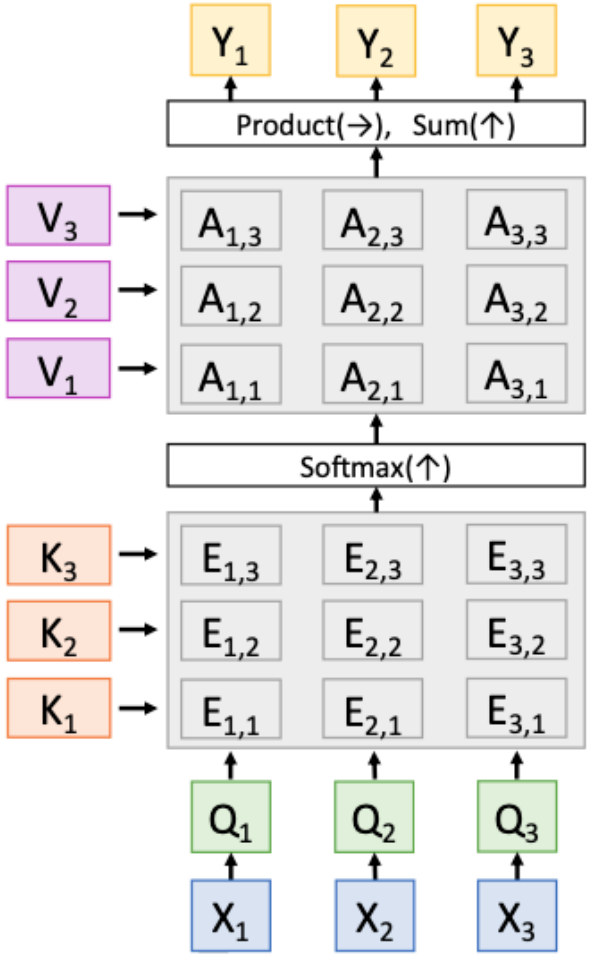
→

Self-attention layer → Permutation equivalent

→ Not care about order of inputs

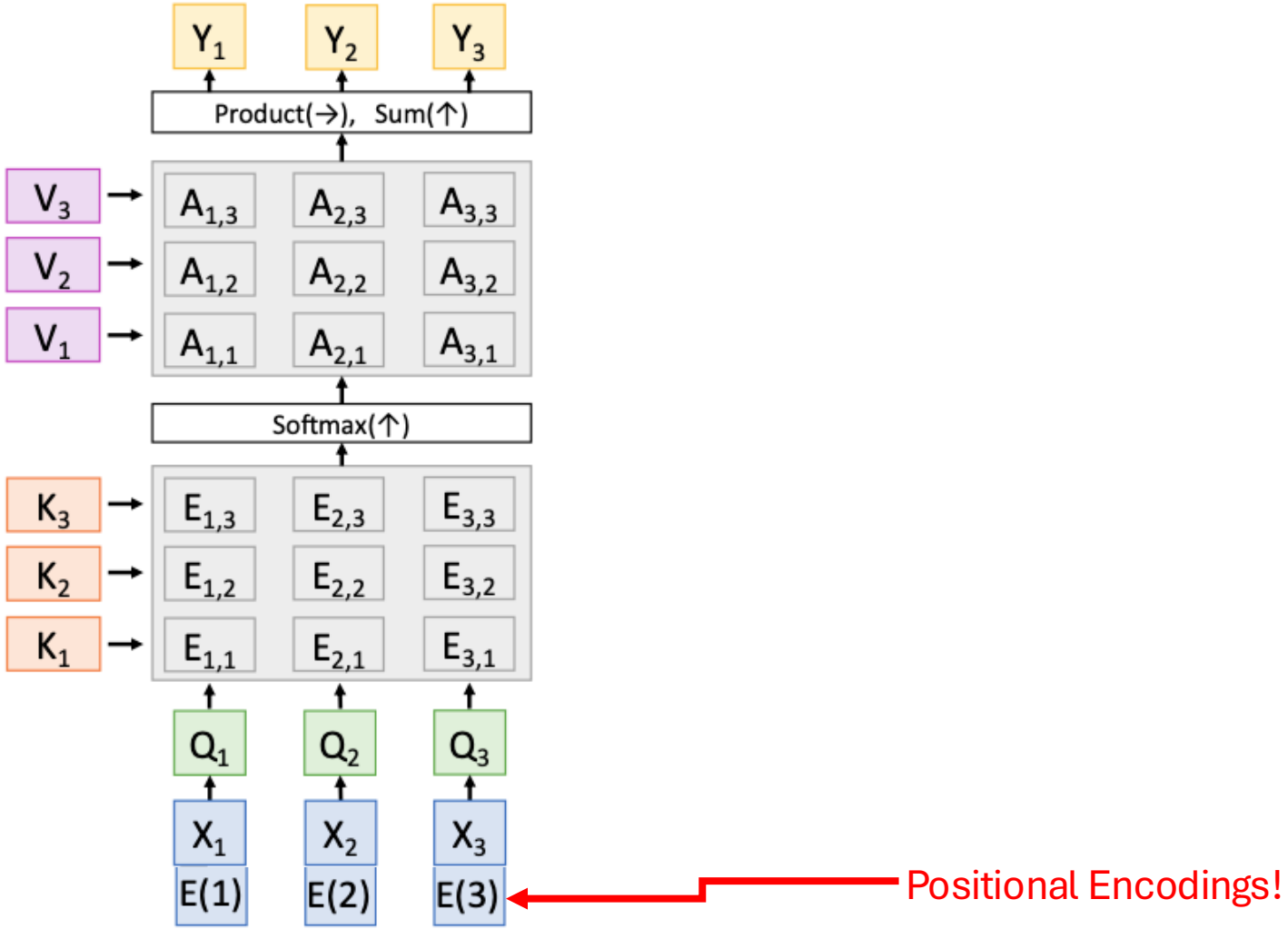
→ No knowledge about order of input vectors

Pop Quiz



How to let the model know the order of the input vectors?

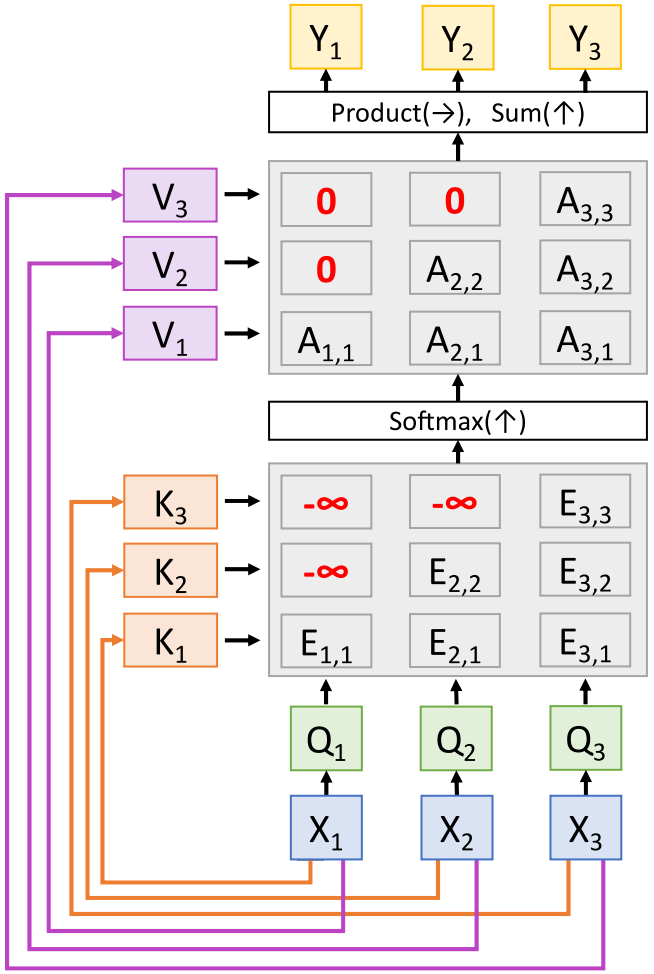
Attention Layer - Properties



Outline: Part II

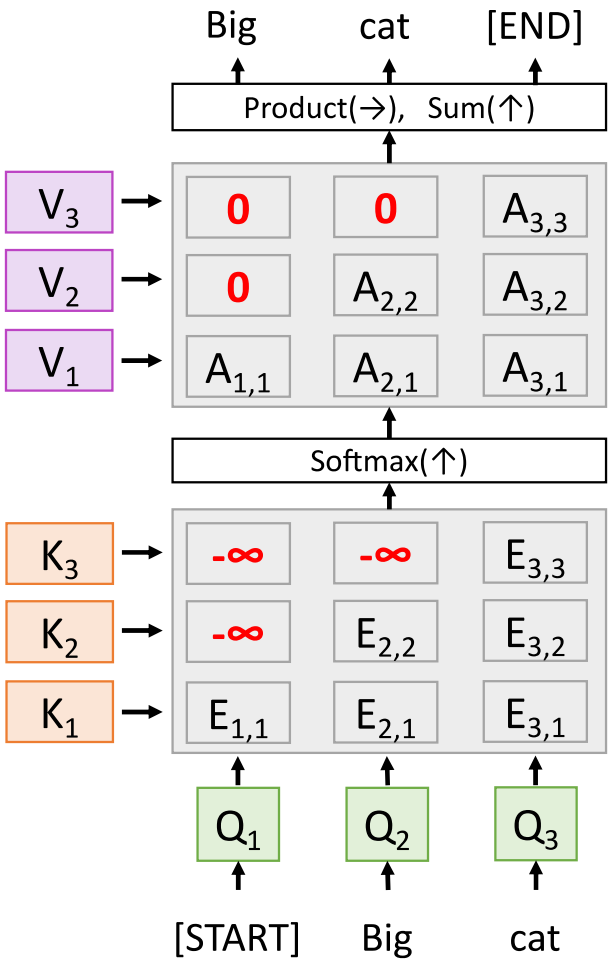
- **Attention and Self-attention**
  - Attention Layer
  - Self-attention layer
  - Properties of self-attention
  - **Self-attention variations**
  - Examples of CNNs with self-attention

Masked Self-attention



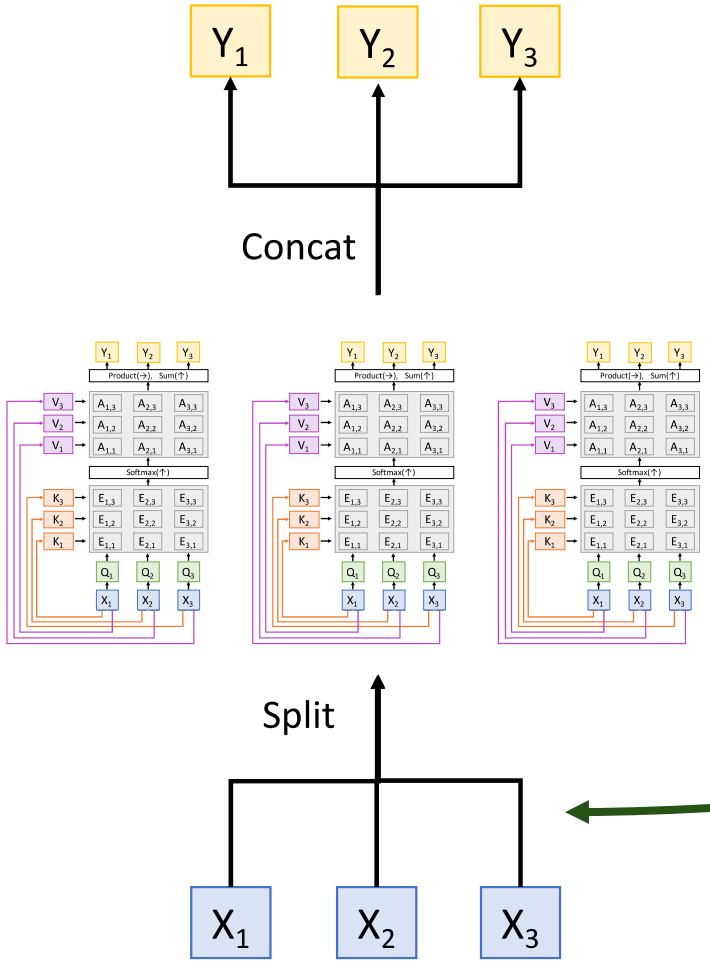
Intuition →
Force the network to only use information from the PAST!

Masked Self-attention



Put – inf in every position where we want to force the model NOT to pay attention

Multi-head self-attention



Choose number of heads h to run self-attention independently, in parallel

→

Input X, split into h chucks

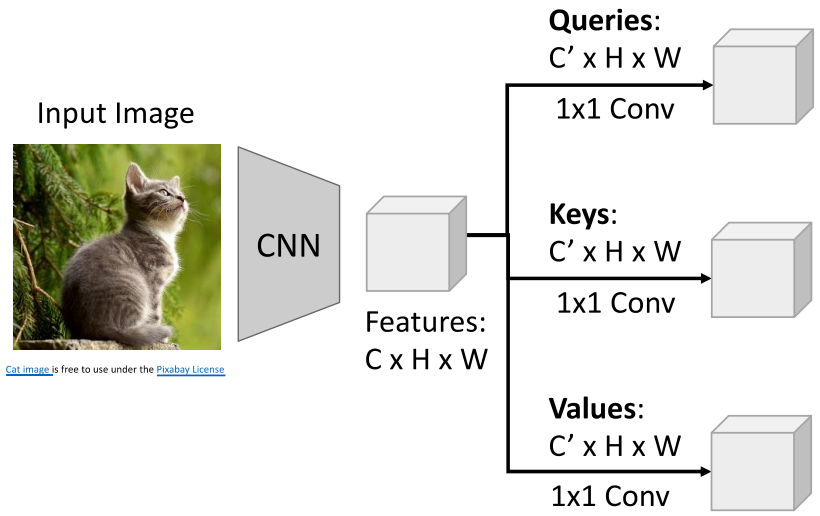
Parameters:

1. query dimension Dq
2. number of heads (= width or size of each head)

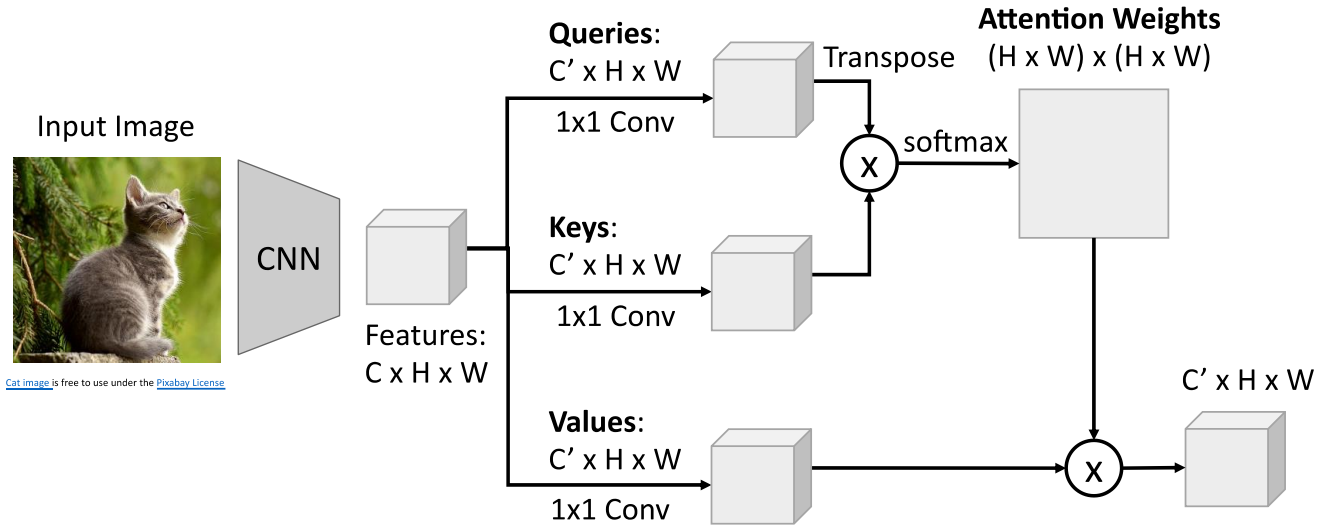
Outline: Part II

- **Attention and Self-attention**
  - Attention Layer
  - Self-attention layer
  - Properties of self-attention
  - Self-attention variations
  - **Examples of CNNs with self-attention**

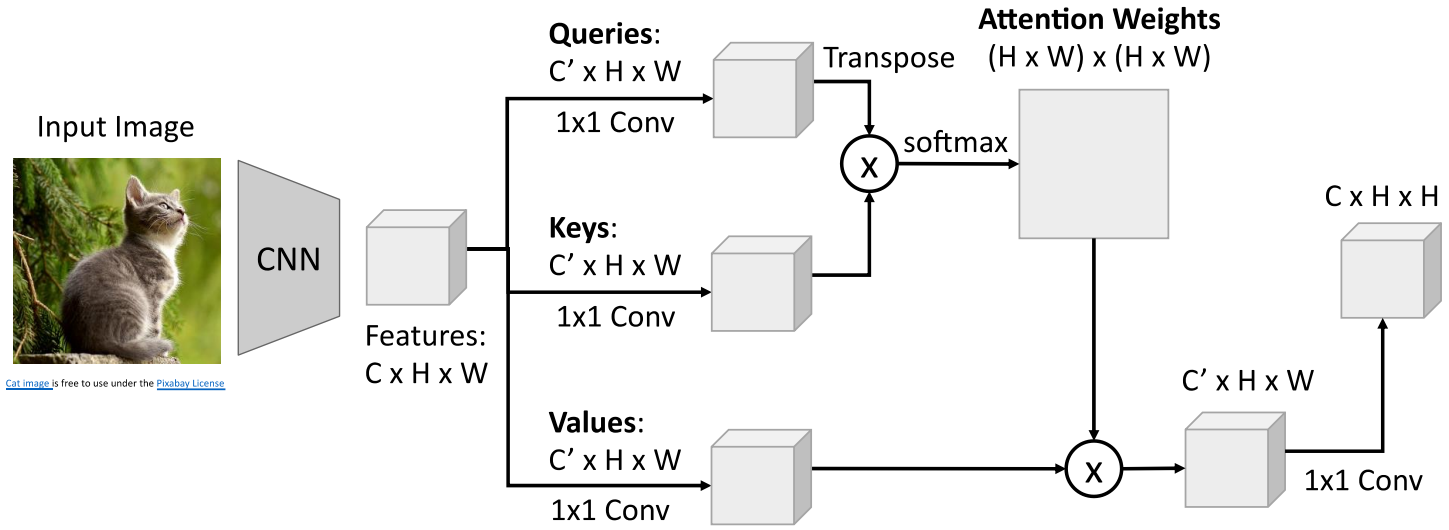
CNN with self-attention



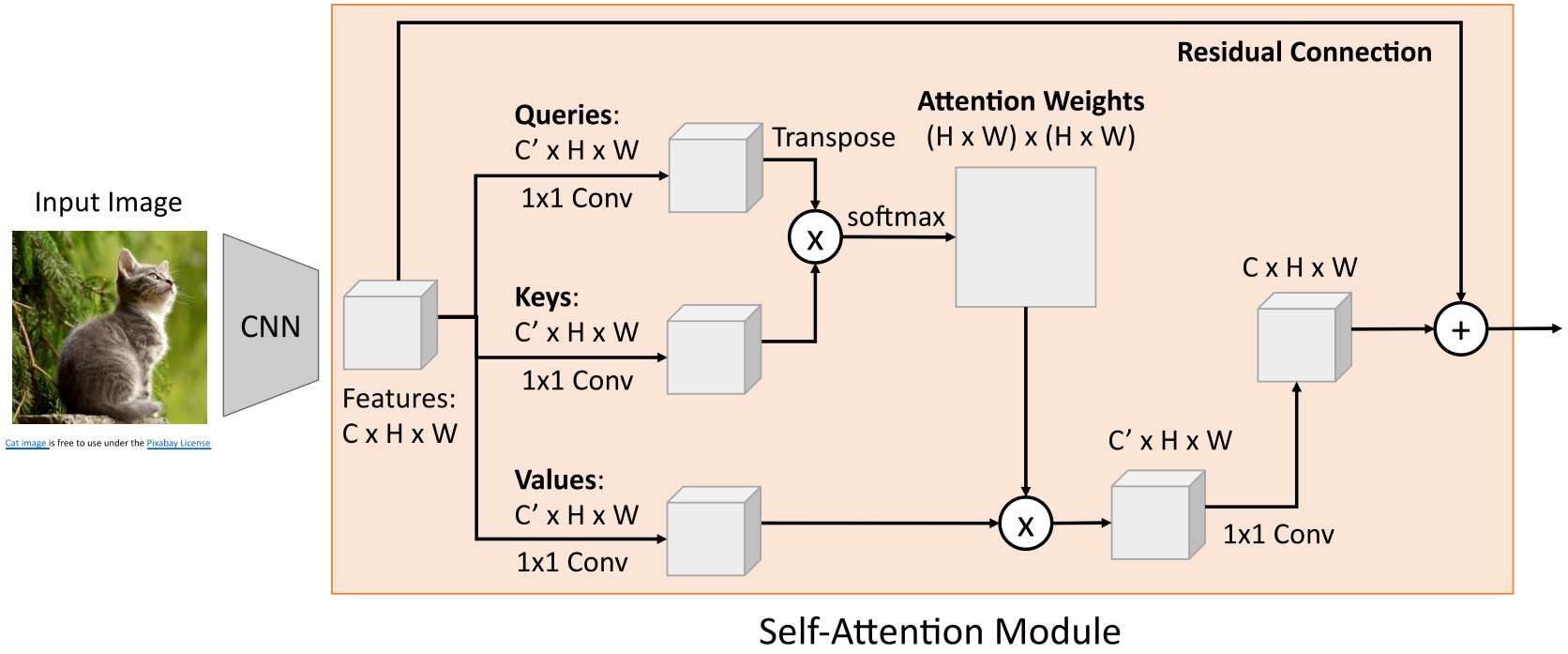
CNN with self-attention



CNN with self-attention

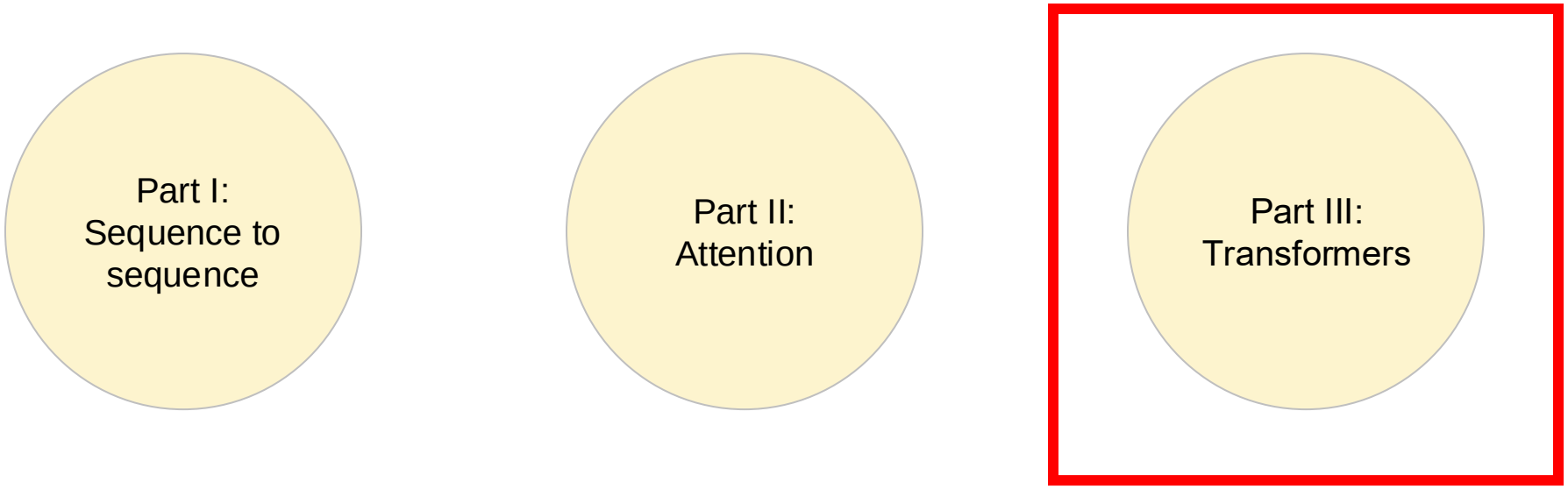


CNN with self-attention



Self-Attention Module

Today's lecture



Some slides adapted from various resources: [Intro to Large Language Models, Andrej Karpathy, Fei-Fei Li, Xi Wang, VGG Oxford, Executive Education Polytechnique, Udemy, Deeplearning.ai, Stanford University CS231n, Financial Times, New York Times, Justin Johnson]

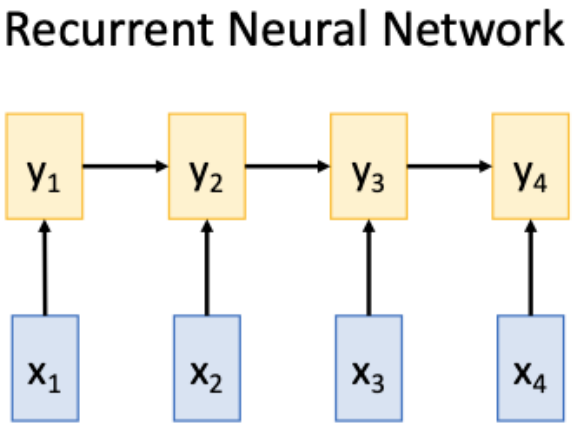
Outline: Part III

- **Transformers**
  - Processing Sequences
  - Transformers: High-level
  - Transformer block
  - Types of Transformer architectures
  - Scaling-up Transformers
  - Vision Transformer

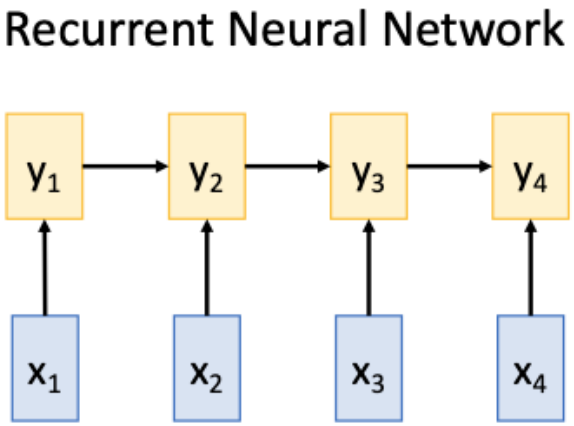
Outline: Part III

- **Transformers**
  - **Processing Sequences**
  - Transformers: High-level
  - Transformer block
  - Types of Transformer architectures
  - Scaling-up Transformers
  - Vision Transformer

Processing Sequences

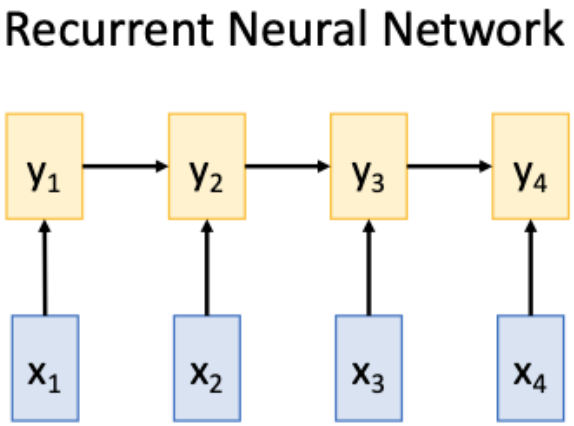


Pop Quiz



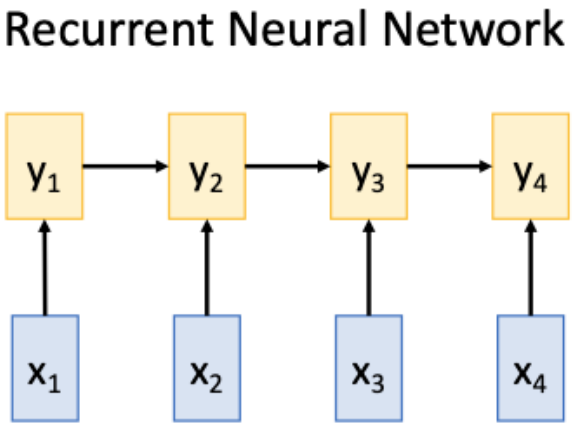
Advantages of RNNs?

Processing Sequences



(+) handle long sequences
(+) summarize entire input sequence

Pop Quiz

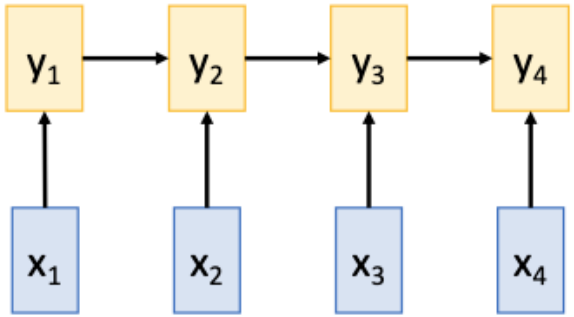


Disadvantages of RNNs?

(+) handle long sequences
(+) summarize entire input sequence

Processing Sequences

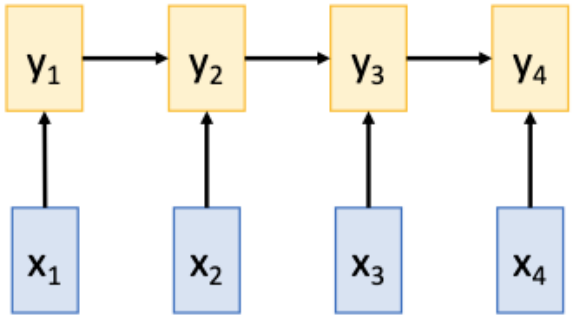
Recurrent Neural Network



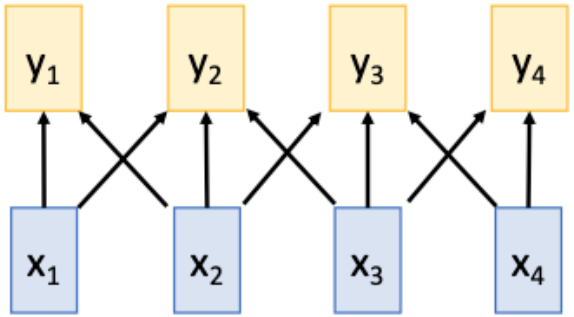
(+) handle long sequences
(+) summarize entire input sequence
(-) not parallelizable: sequential dependency on the data

Processing Sequences

Recurrent Neural Network



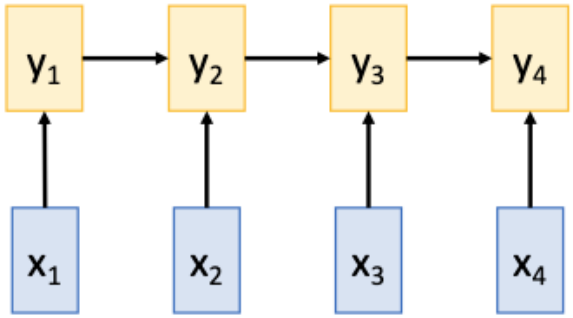
1D Convolution



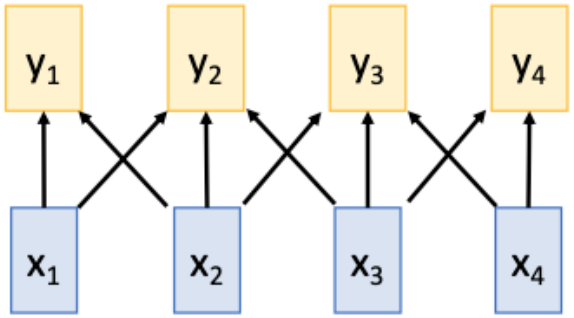
(+) handle long sequences
(+) summarize entire input sequence
(-) not parallelizable: sequential dependency on the data

Pop Quiz

Recurrent Neural Network



1D Convolution

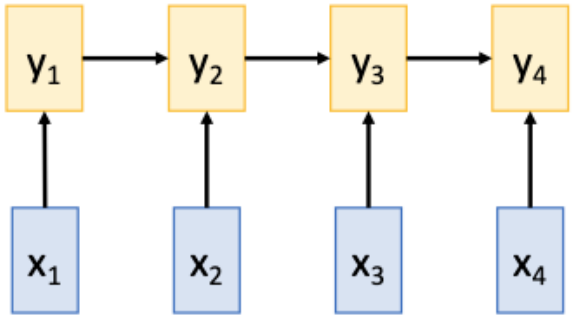


Advantages of 1D convolutions?

(+) handle long sequences
(+) summarize entire input sequence
(-) not parallelizable: sequential dependency on the data

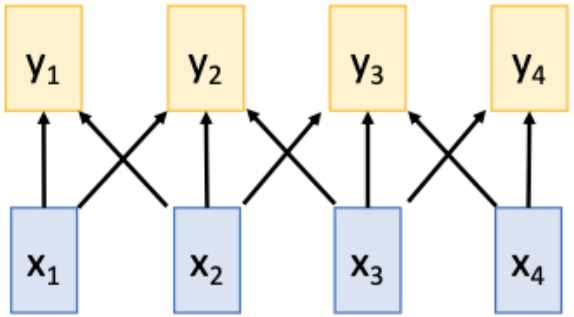
Processing Sequences

Recurrent Neural Network



(+) handle long sequences
(+) summarize entire input sequence
(-) not parallelizable: sequential dependency on the data

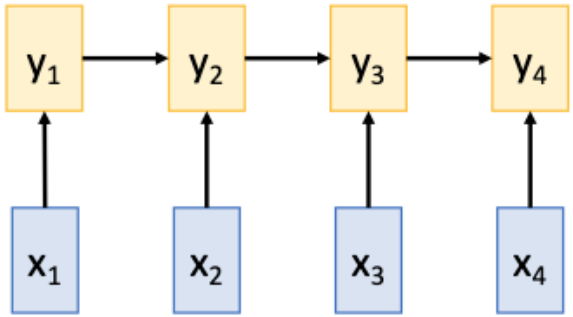
1D Convolution



(+) highly parallelizable: each output can be computed in parallel
(+) works on multi-dimensional grids

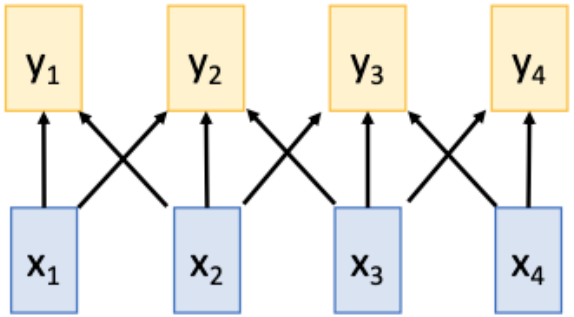
Pop Quiz

Recurrent Neural Network



(+) handle long sequences
(+) summarize entire input sequence
(-) not parallelizable: sequential dependency on the data

1D Convolution

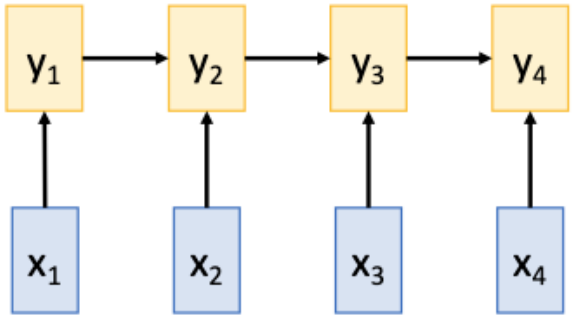


(+) highly parallelizable: each output can be computed in parallel
(+) works on multi-dimensional grids

Disadvantages of 1D convolutions?

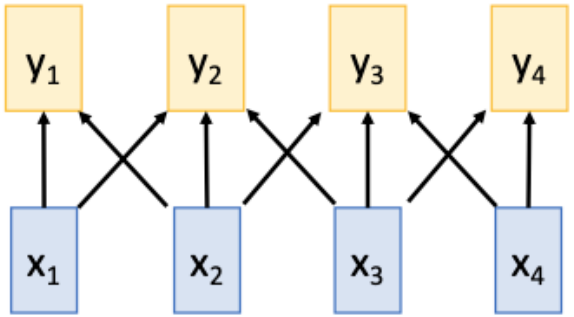
Processing Sequences

Recurrent Neural Network



(+) handle long sequences
(+) summarize entire input sequence
(-) not parallelizable: sequential dependency on the data

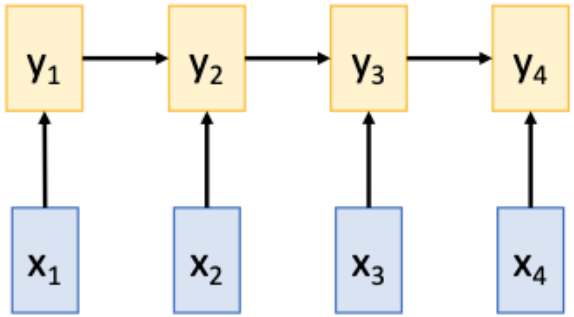
1D Convolution



(+) highly parallelizable: each output can be computed in parallel
(+) works on multi-dimensional grids
(-) bad at long sequences: need to stack many conv layers to see the whole sequence

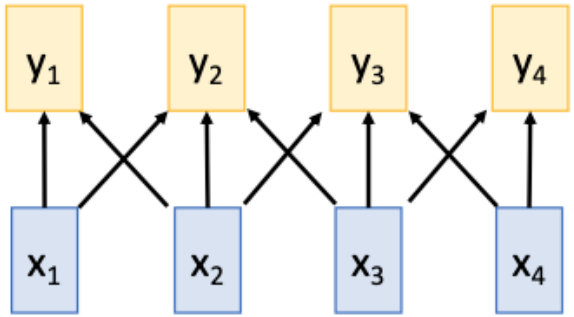
Processing Sequences

Recurrent Neural Network



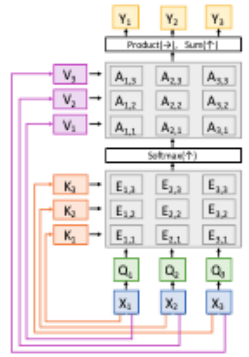
(+) handle long sequences
(+) summarize entire input sequence
(-) not parallelizable: sequential dependency on the data

1D Convolution



(+) highly parallelizable: each output can be computed in parallel
(+) works on multi-dimensional grids
(-) bad at long sequences: need to stack many conv layers to see the whole sequence

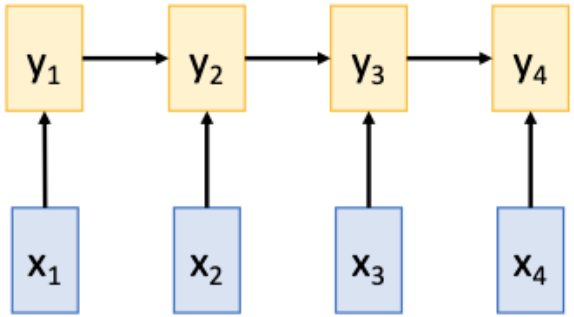
Self-Attention



(+) good at long sequences: after one self-attention layer, each output sees all inputs
(+) highly parallelizable: each output can be computed in parallel

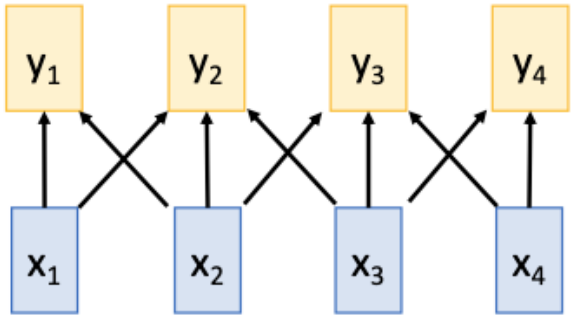
Processing Sequences

Recurrent Neural Network



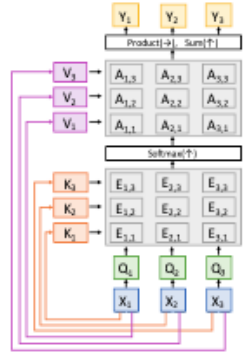
(+) handle long sequences
(+) summarize entire input sequence
(-) not parallelizable: sequential dependency on the data

1D Convolution



(+) highly parallelizable: each output can be computed in parallel
(+) works on multi-dimensional grids
(-) bad at long sequences: need to stack many conv layers to see the whole sequence

Self-Attention

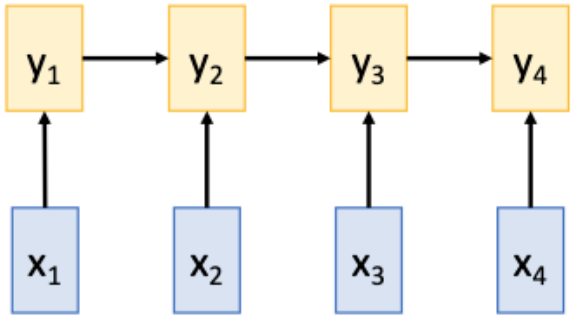


(+) good at long sequences: after one self-attention layer, each output sees all inputs
(+) highly parallelizable: each output can be computed in parallel
(-) memory intensive

Pop Quiz

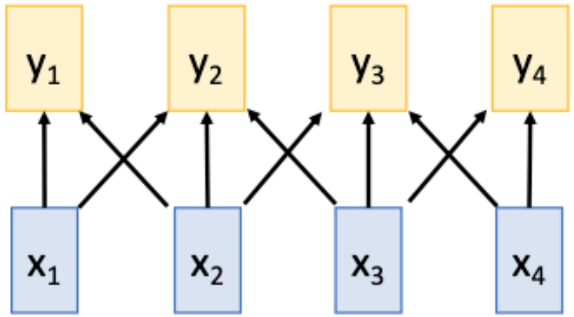
Which one to use? Can I combine?

Recurrent Neural Network



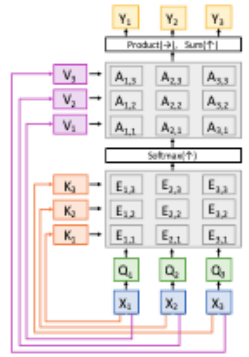
(+) handle long sequences
(+) summarize entire input sequence
(-) not parallelizable: sequential dependency on the data

1D Convolution



(+) highly parallelizable: each output can be computed in parallel
(+) works on multi-dimensional grids
(-) bad at long sequences: need to stack many conv layers to see the whole sequence

Self-Attention



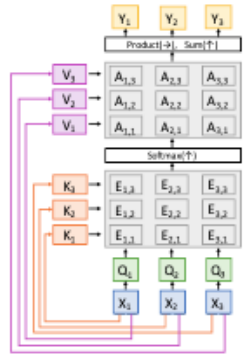
(+) good at long sequences: after one self-attention layer, each output sees all inputs
(+) highly parallelizable: each output can be computed in parallel
(-) memory intensive

Processing Sequences

Attention is all you need ☺

[Vaswani et. al., NeurIPS 2017]

Self-Attention



(+) good at long sequences: after one self-attention layer, each output sees all inputs
(+) highly parallelizable: each output can be computed in parallel
(-) memory intensive

Generative AI exists because of the **Transformer**

Transformers

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

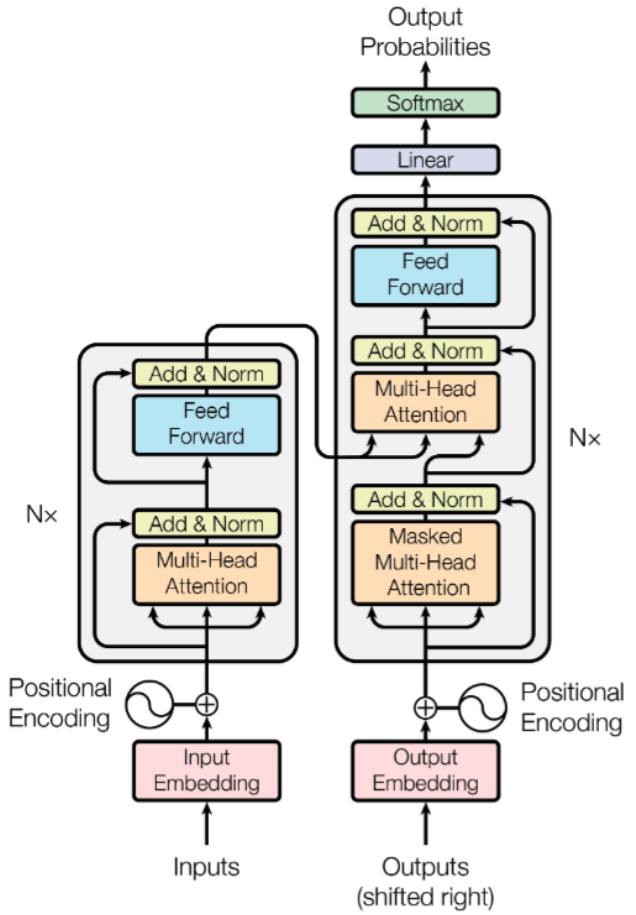
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to



Transformers

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

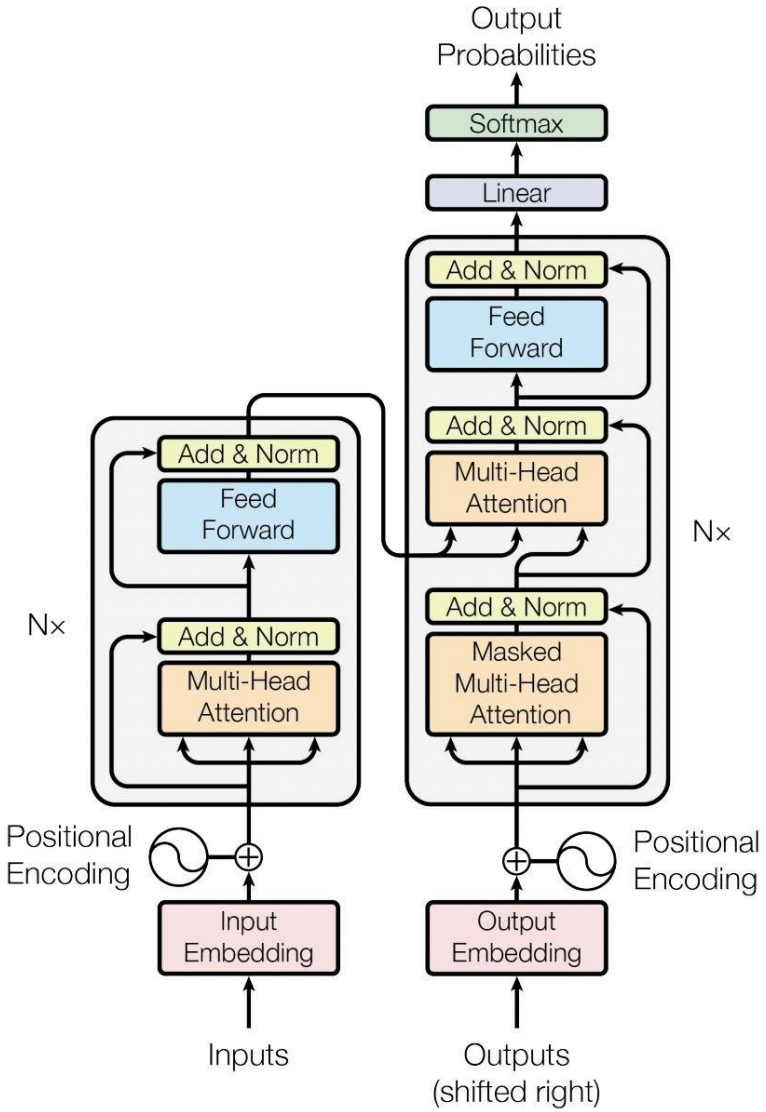
The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to

- Scale efficiently
- Parallel process
- Attention to input meaning

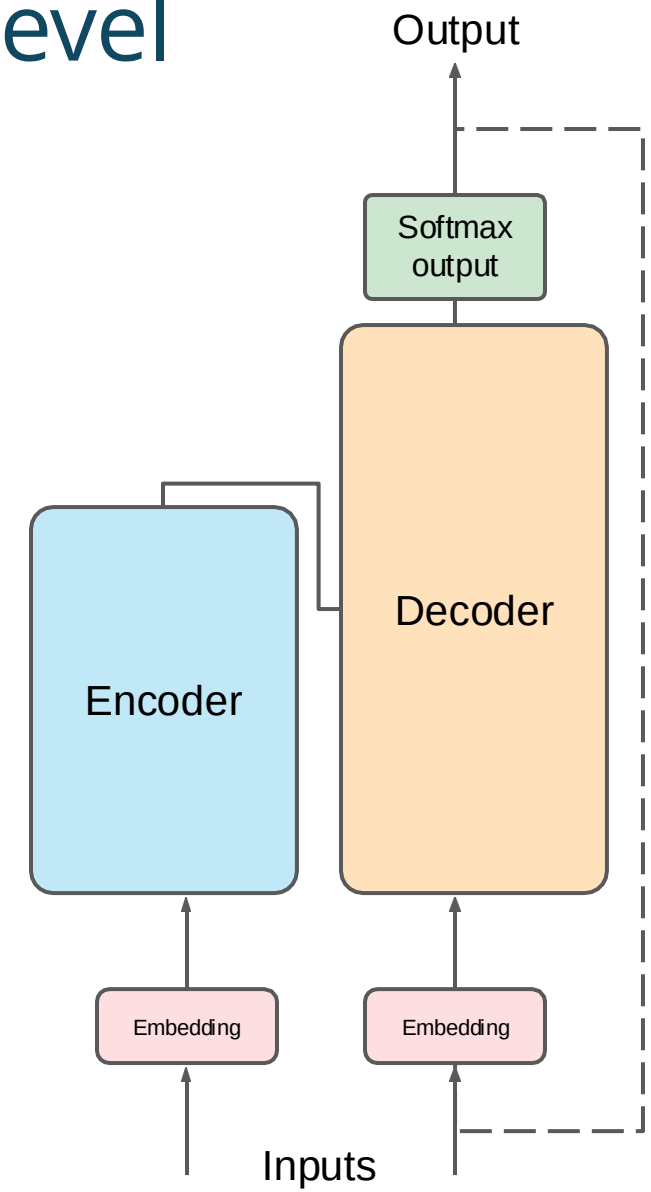
Outline: Part III

- **Transformers**
  - Processing Sequences
  - **Transformers: High-level**
  - Transformer block
  - Types of Transformer architectures
  - Scaling-up Transformers
  - Vision Transformer

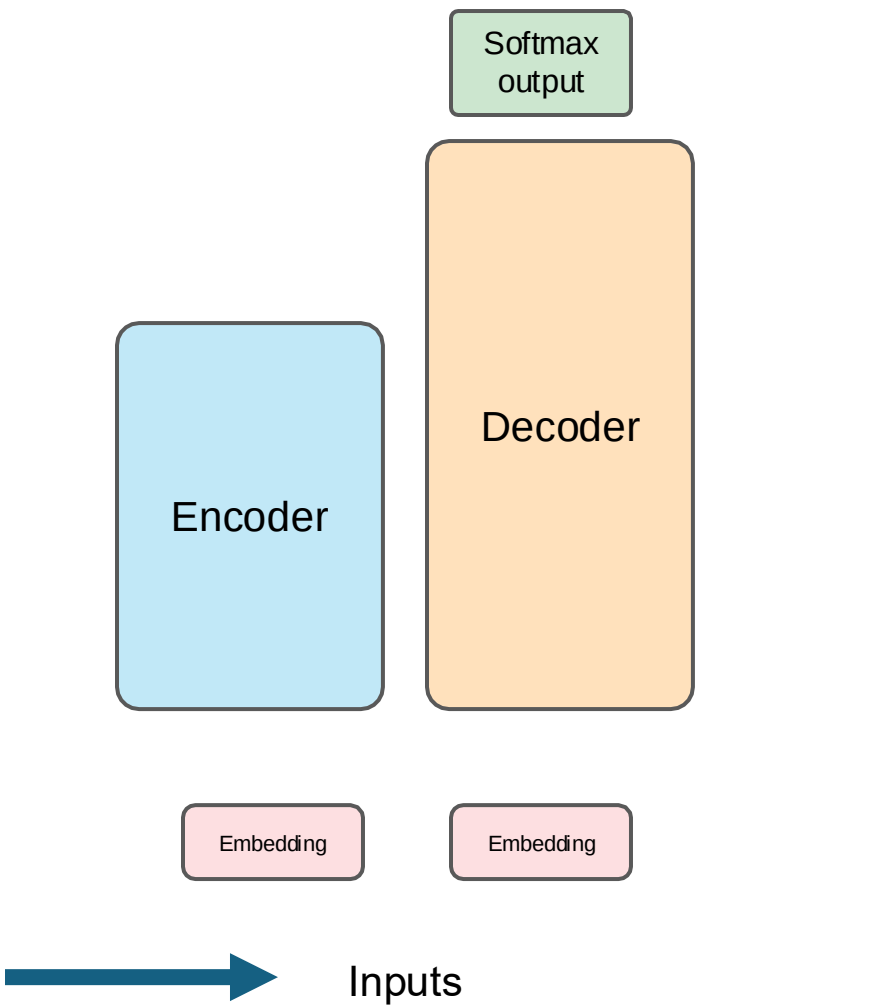
Transformers: High-level



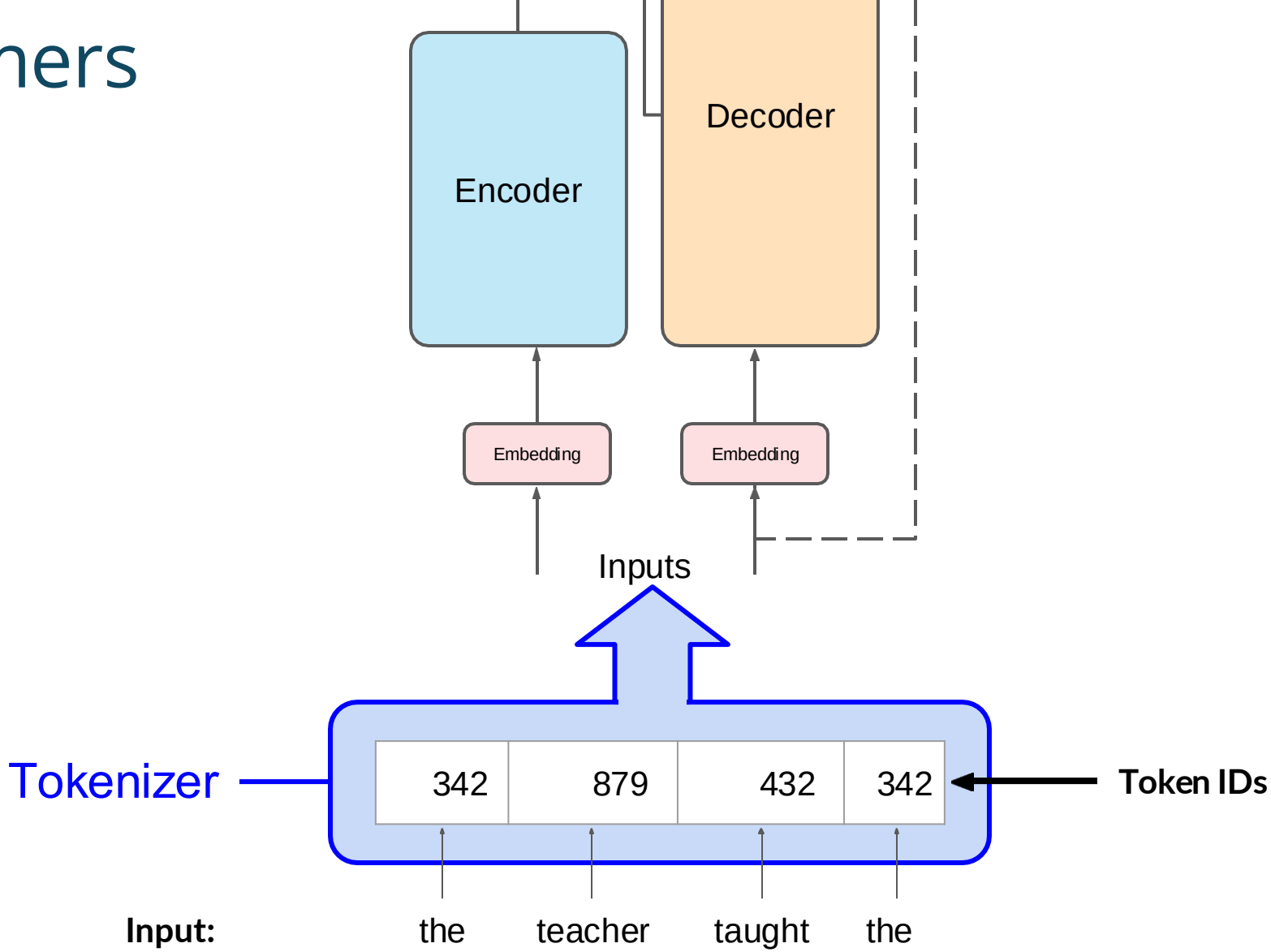
Transformers: High-level



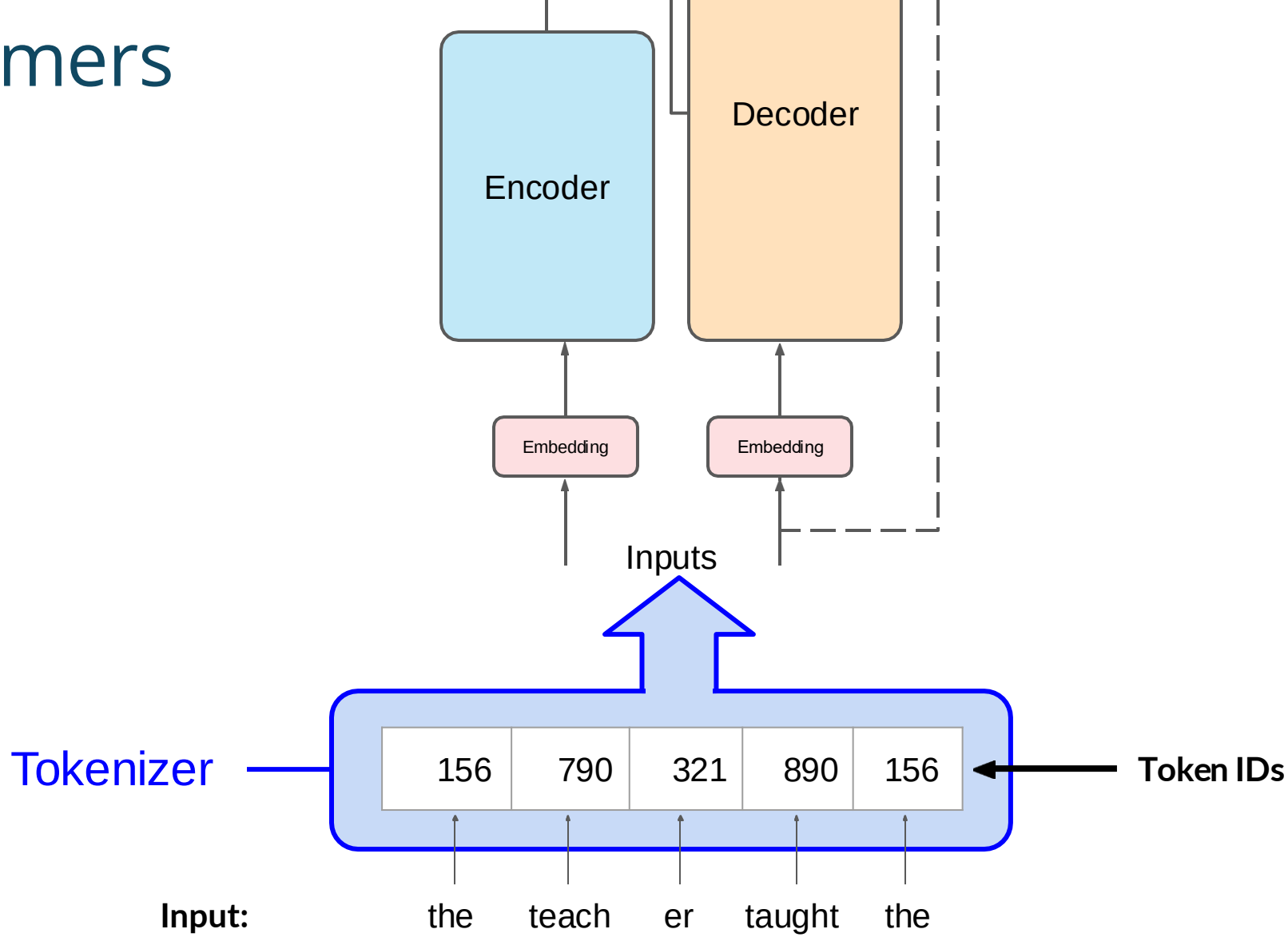
Transformers: High-level



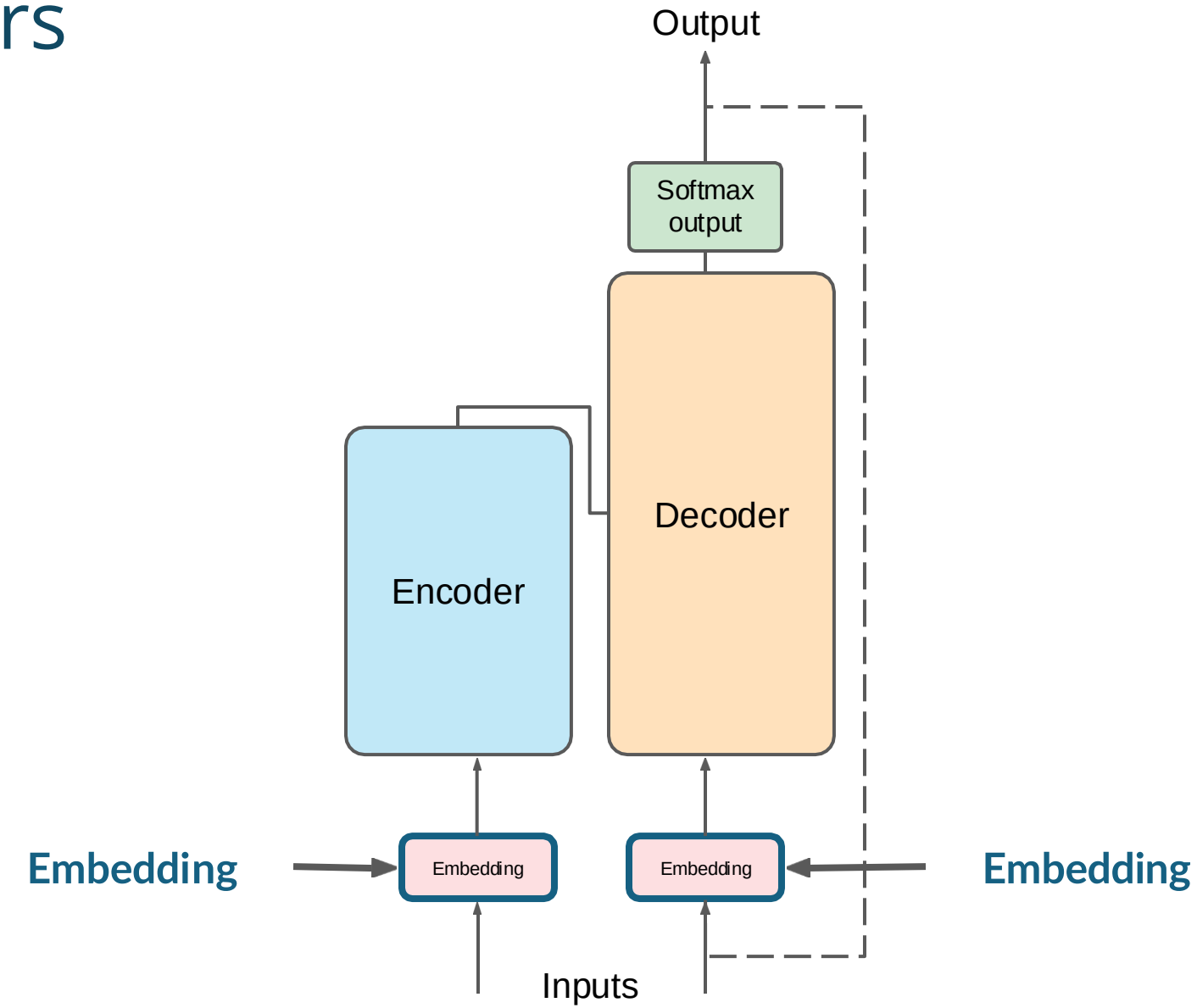
Transformers



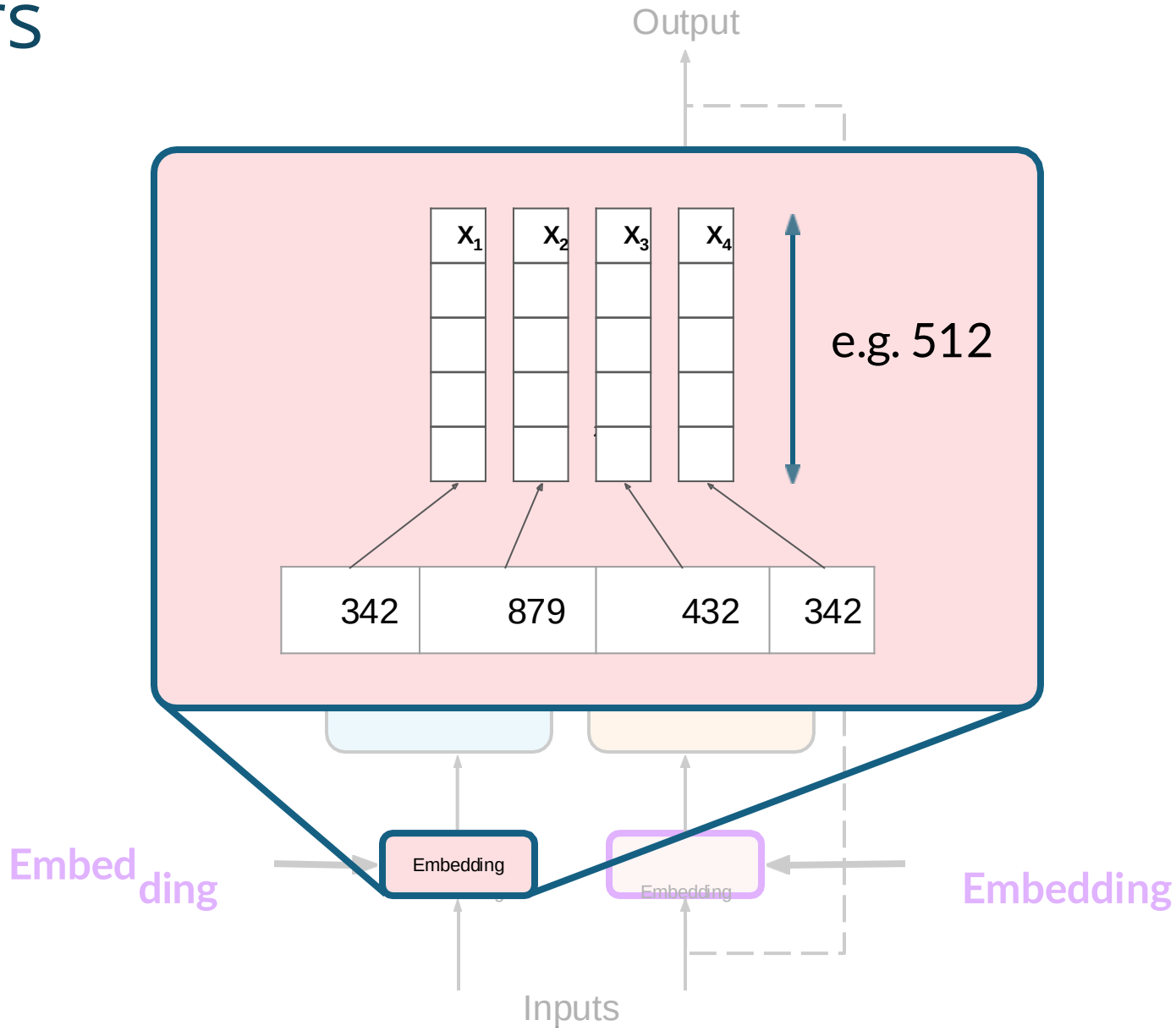
Transformers



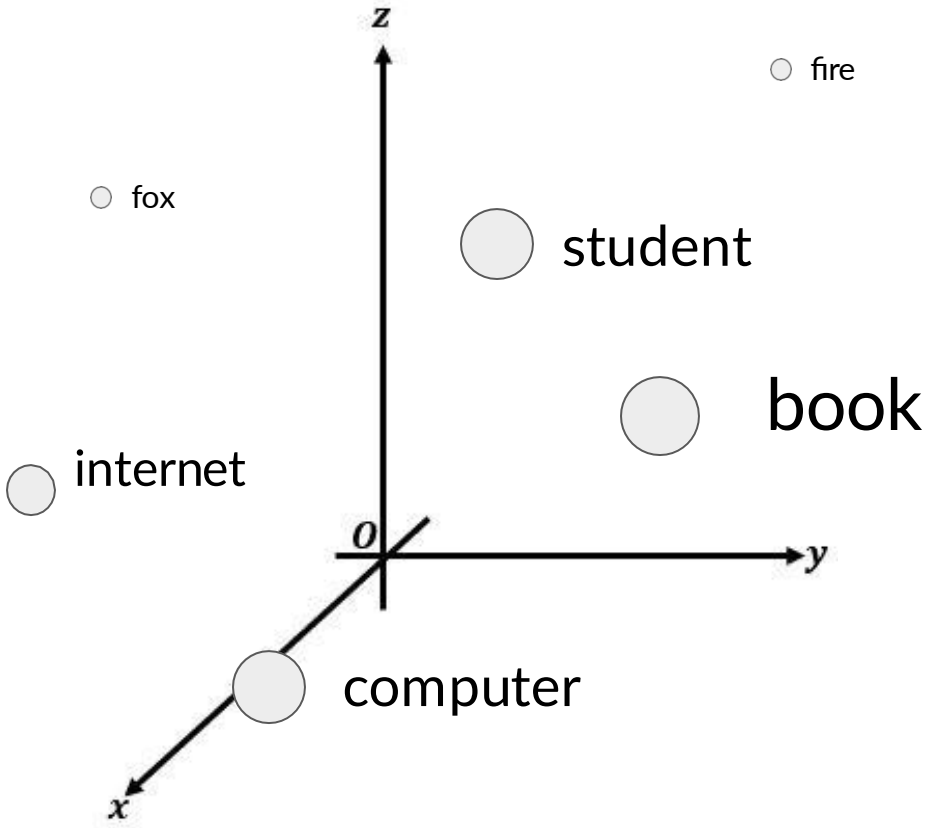
Transformers



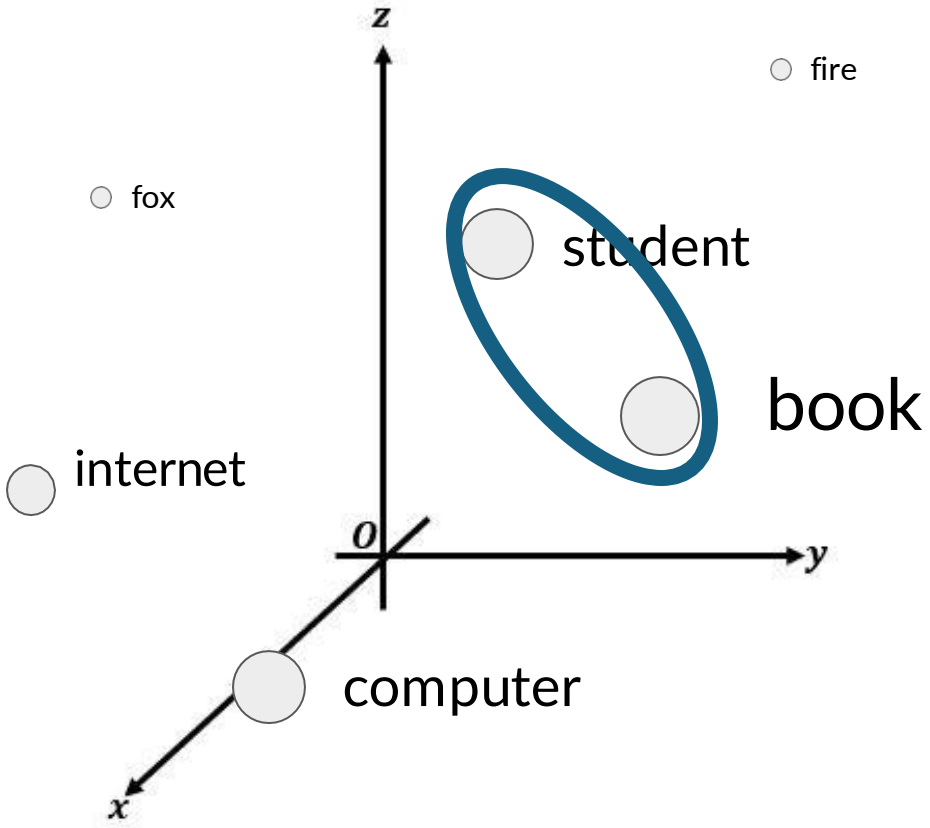
Transformers



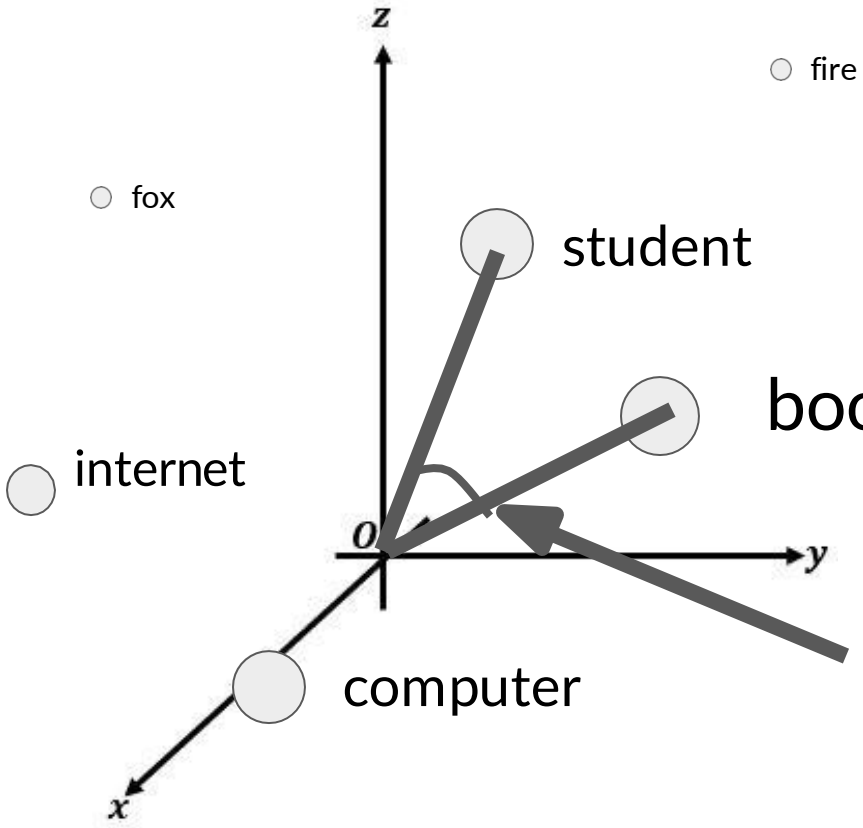
Transformers: High-level



Transformers: High-level

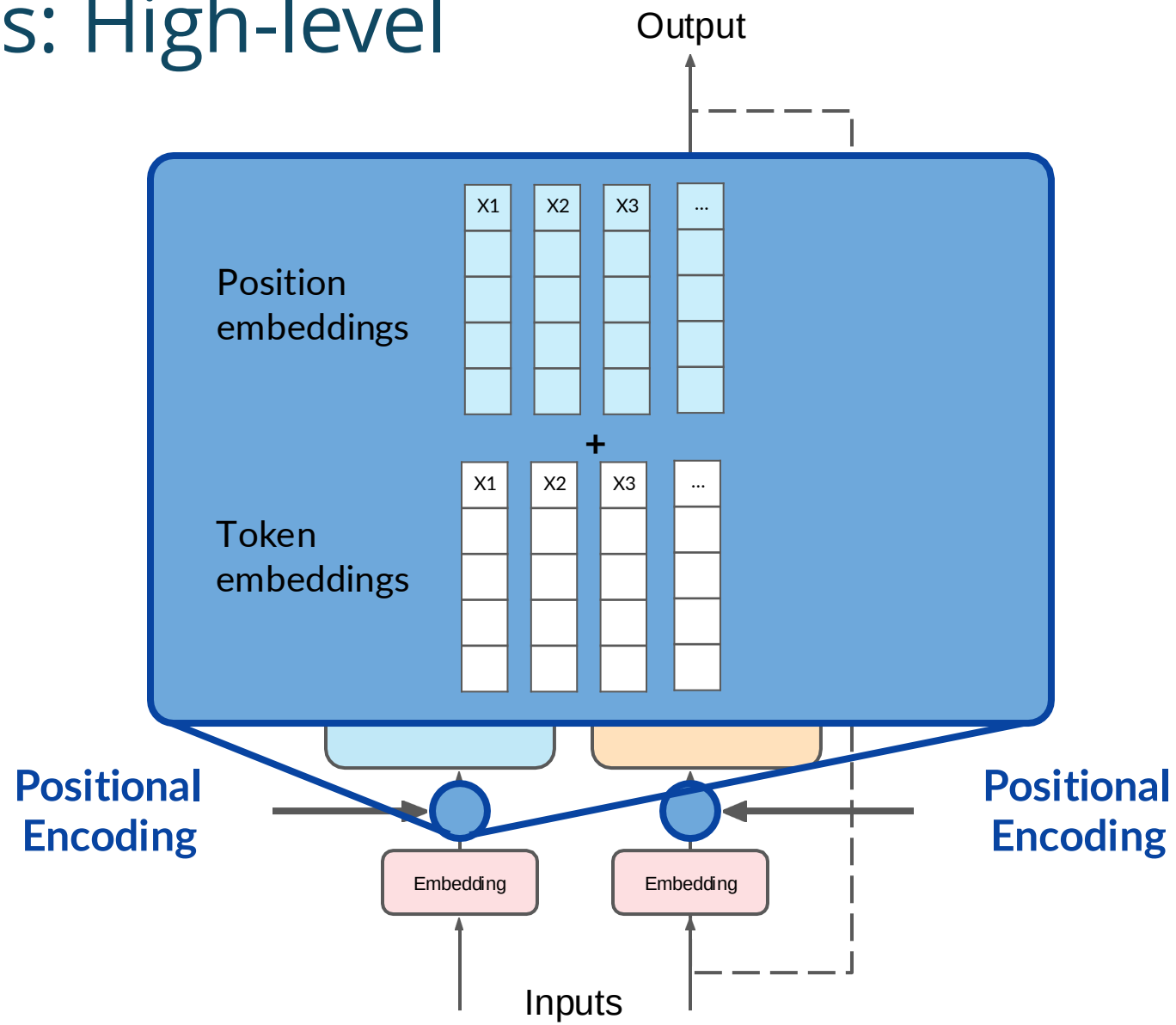


Transformers: High-level

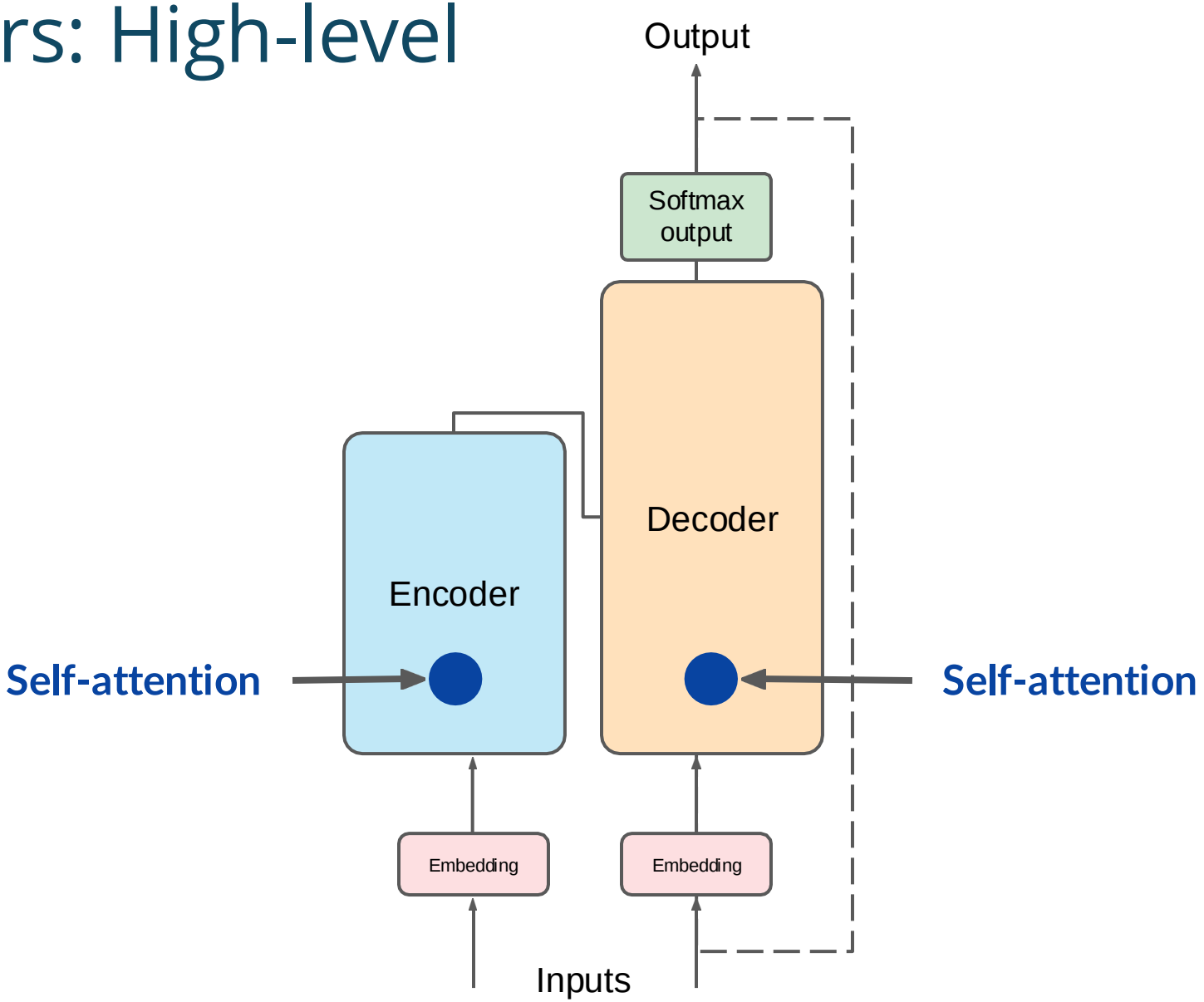


Angle measures distance between words

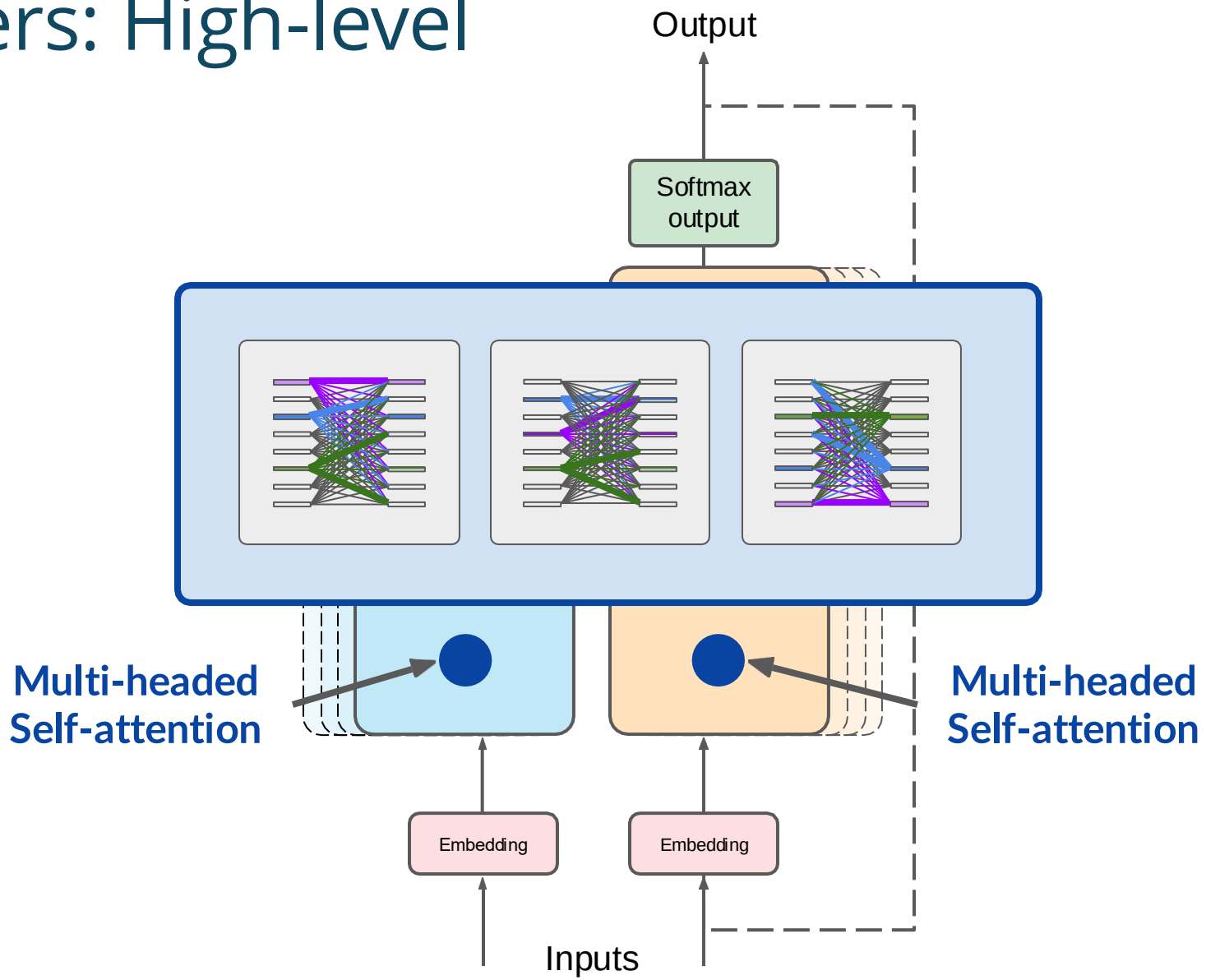
Transformers: High-level



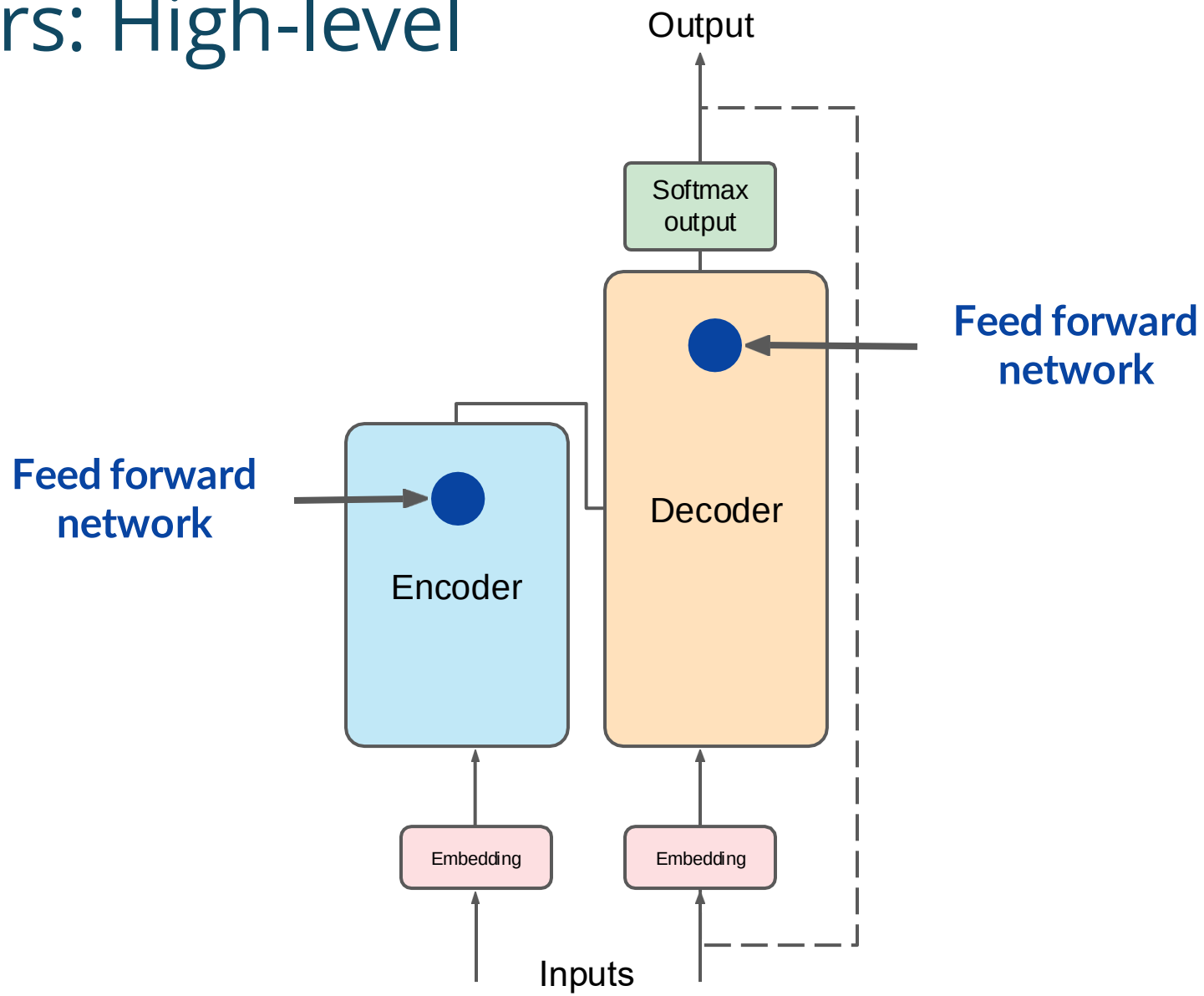
Transformers: High-level



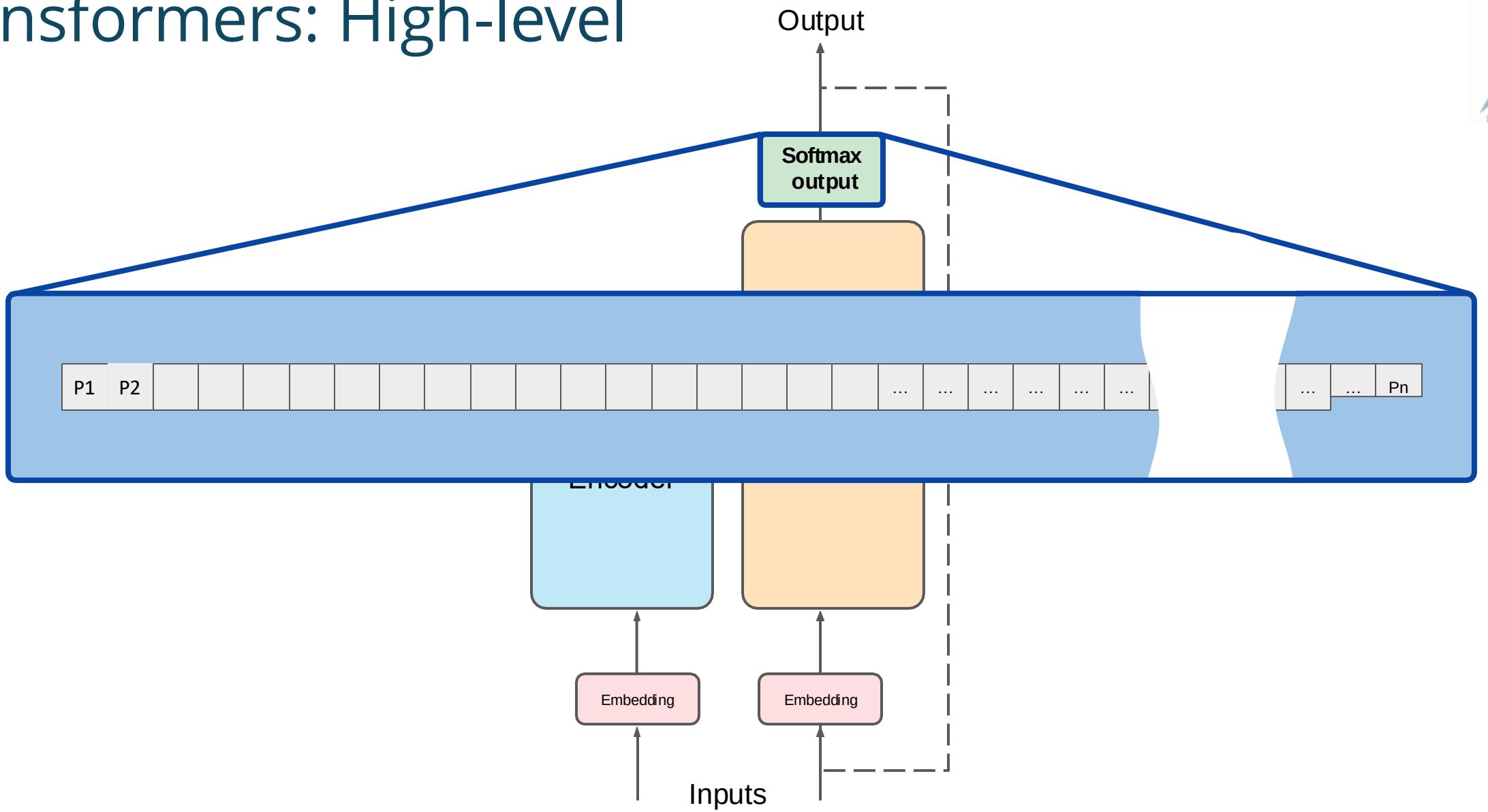
Transformers: High-level



Transformers: High-level



Transformers: High-level

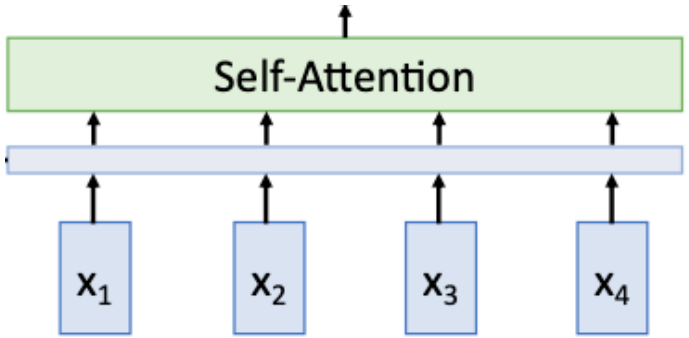


Outline: Part III

- **Transformers**
  - Processing Sequences
  - Transformers: High-level
  - **Transformer block**
  - Types of Transformer architectures
  - Scaling-up Transformers
  - Vision Transformer

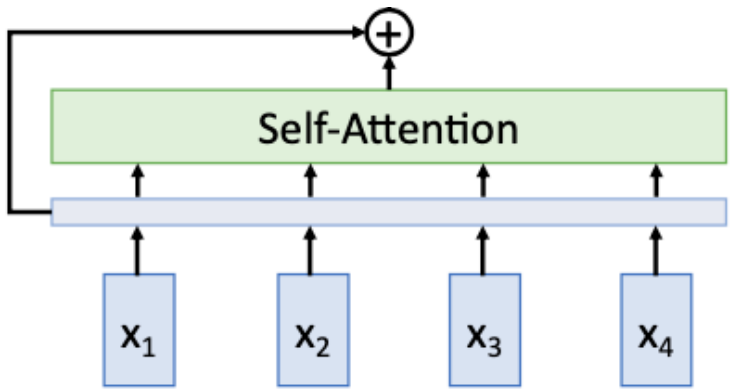
New block: Transformer Block

Transformers



All vectors interact with each other!

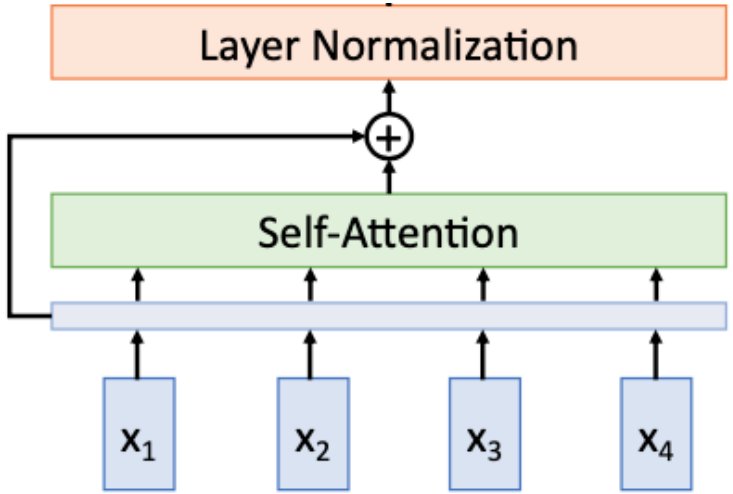
Transformers



Residual connection

All vectors interact with each other!

Transformers

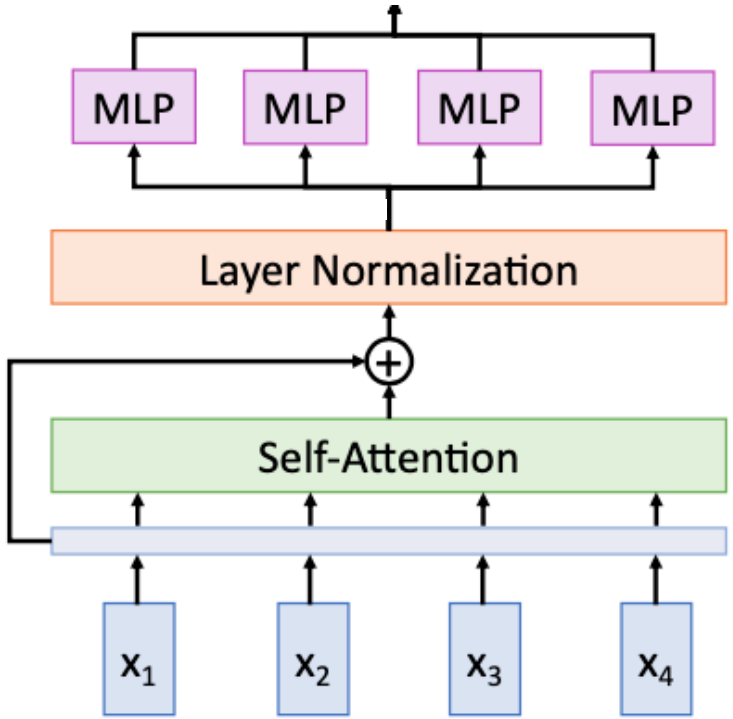


Aids optimization (no vector communication)

Residual connection

All vectors interact
with each other!

Transformers



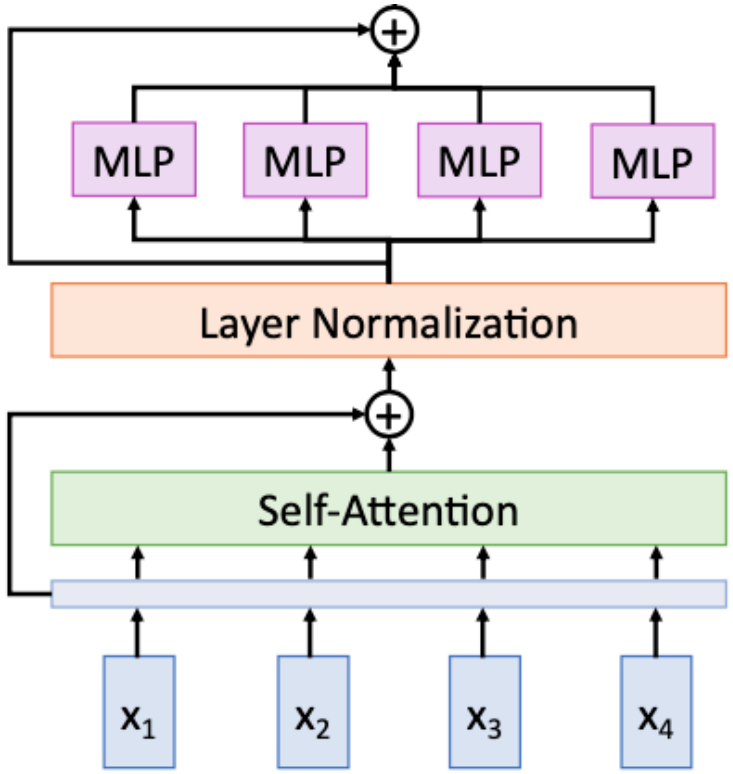
MLP independently on each vector

Aids optimization (no vector communication)

Residual connection

All vectors interact with each other!

Transformers



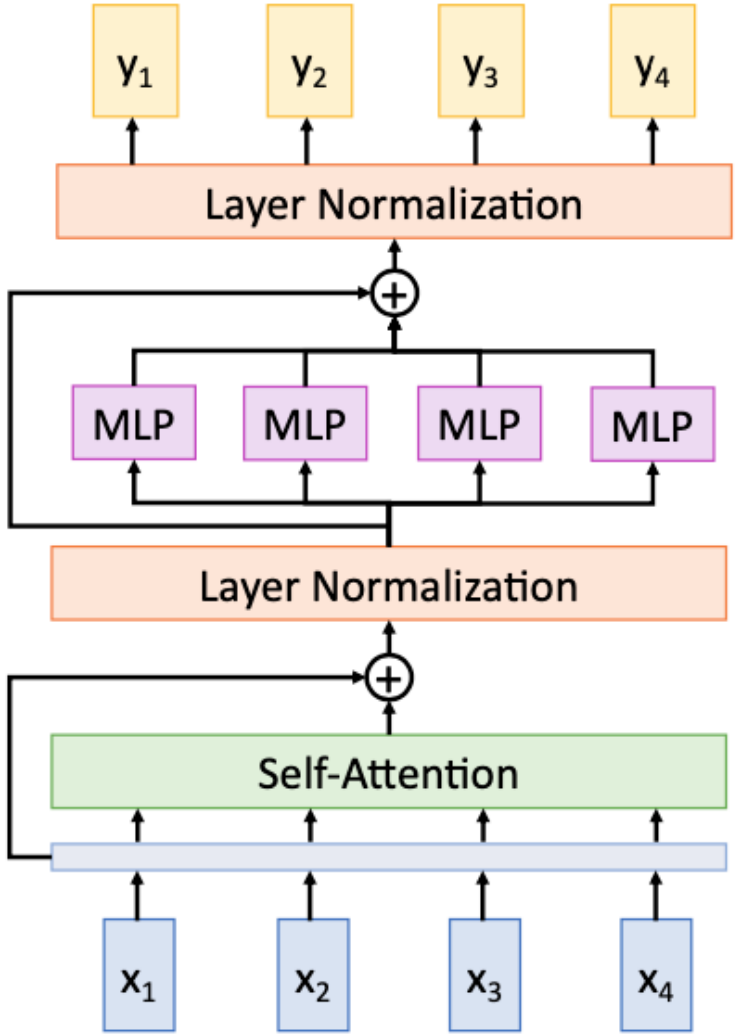
MLP independently on each vector

Aids optimization (no vector communication)

Residual connection

All vectors interact with each other!

Transformers



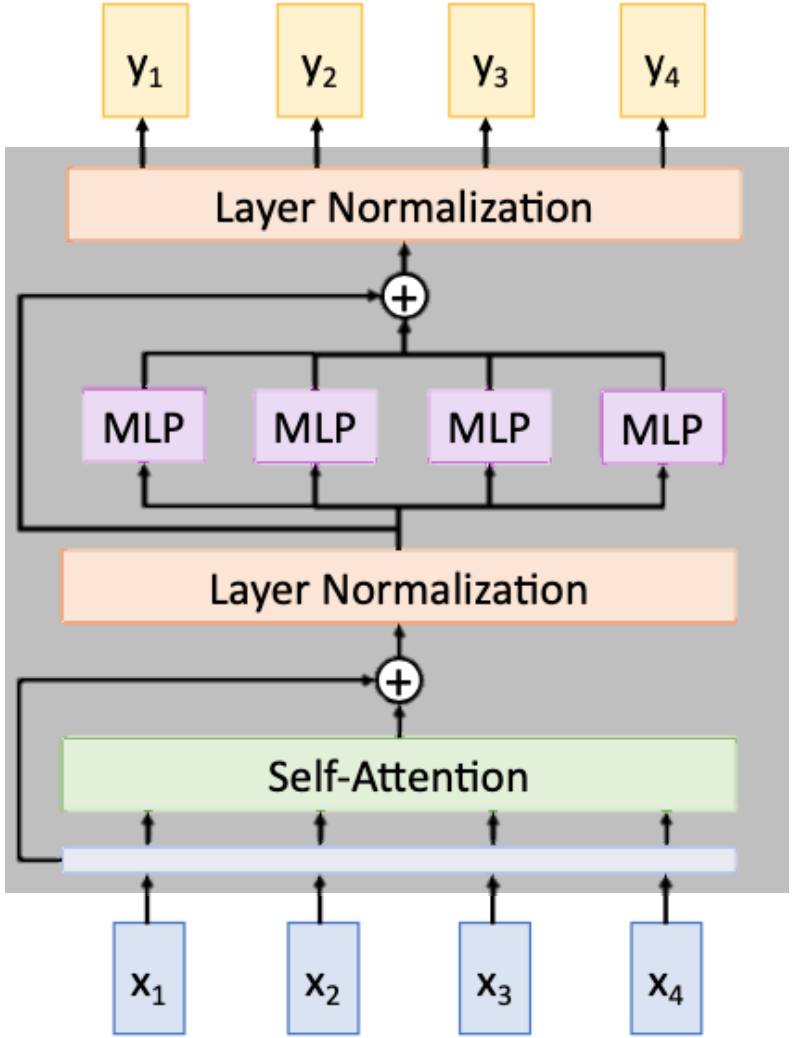
MLP independently on each vector

Aids optimization (no vector communication)

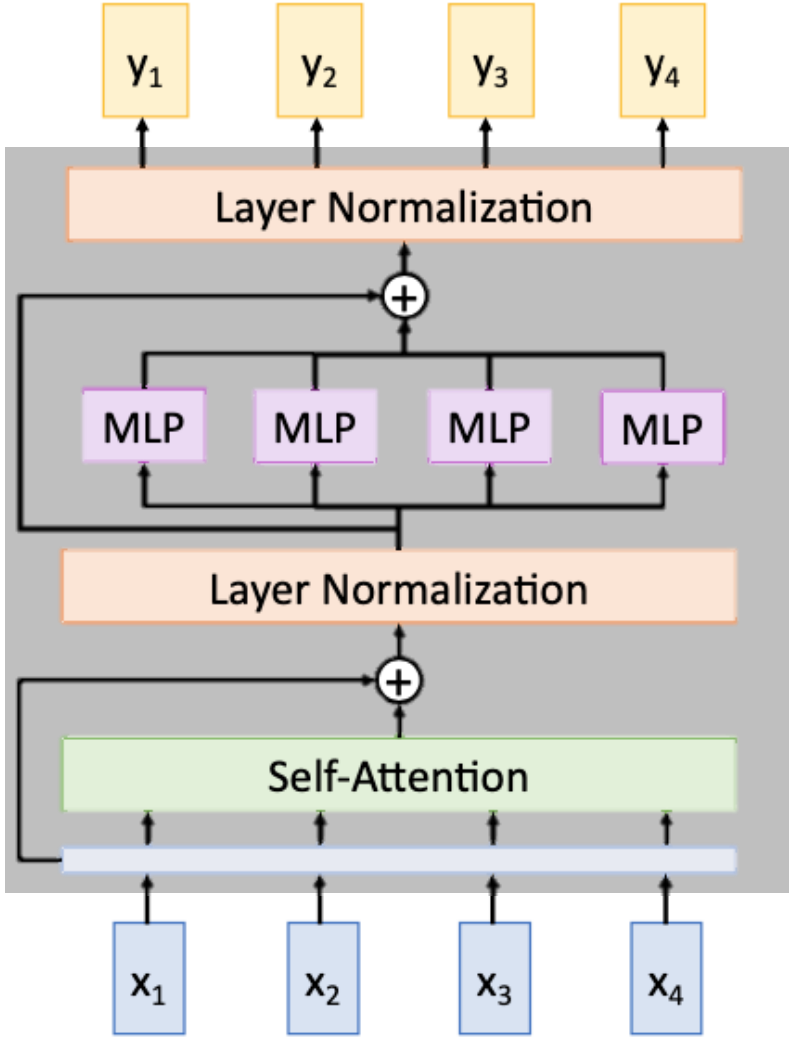
Residual connection

All vectors interact with each other!

Transformers



Transformers



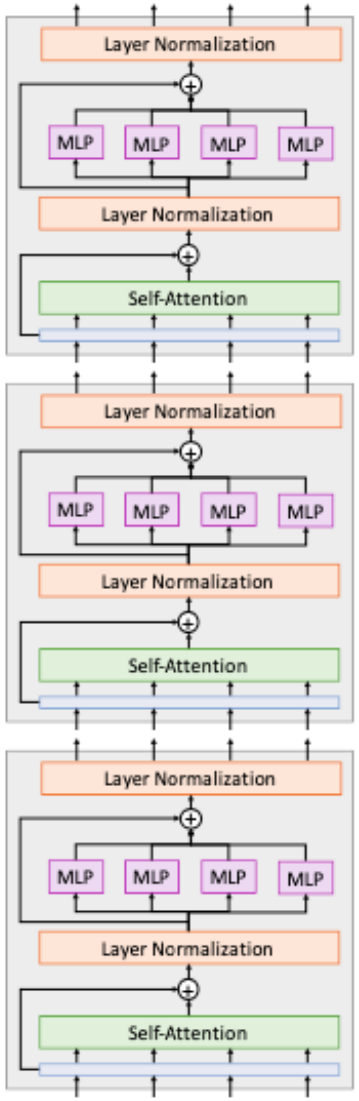
Input: set of vectors x
Output: set of vectors y

Self-attention is the only interaction between vectors

Layer normalization and MLP operate independently per vector

(+) highly scalable
(+) highly parallelizable

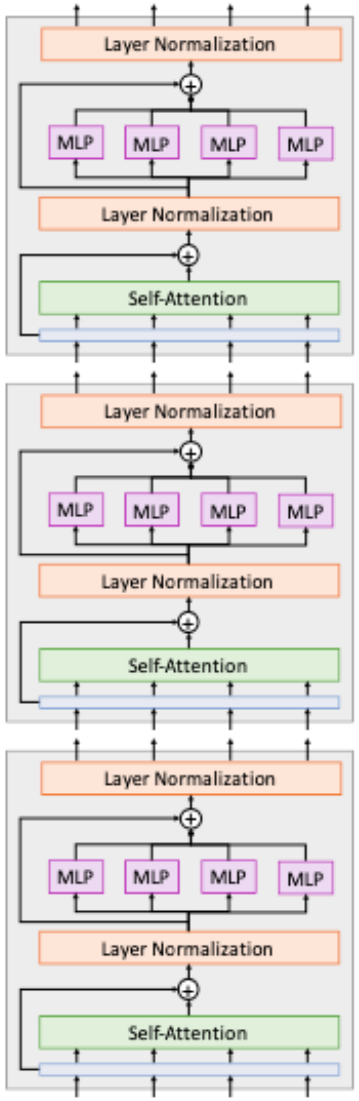
Transformers



Transformer: sequence of transformer blocks

Hyper-parameters: Attention is all you need
12 blocks, $D_Q = 512$, 6 heads

Transformers



Transformer: sequence of transformer blocks

"ImageNet Moment for NLP"

Pre-training:

- Download text from internet
- Train a giant Transformer for language modeling

Fine-tuning:

- Fine-tune transformer on own NLP task

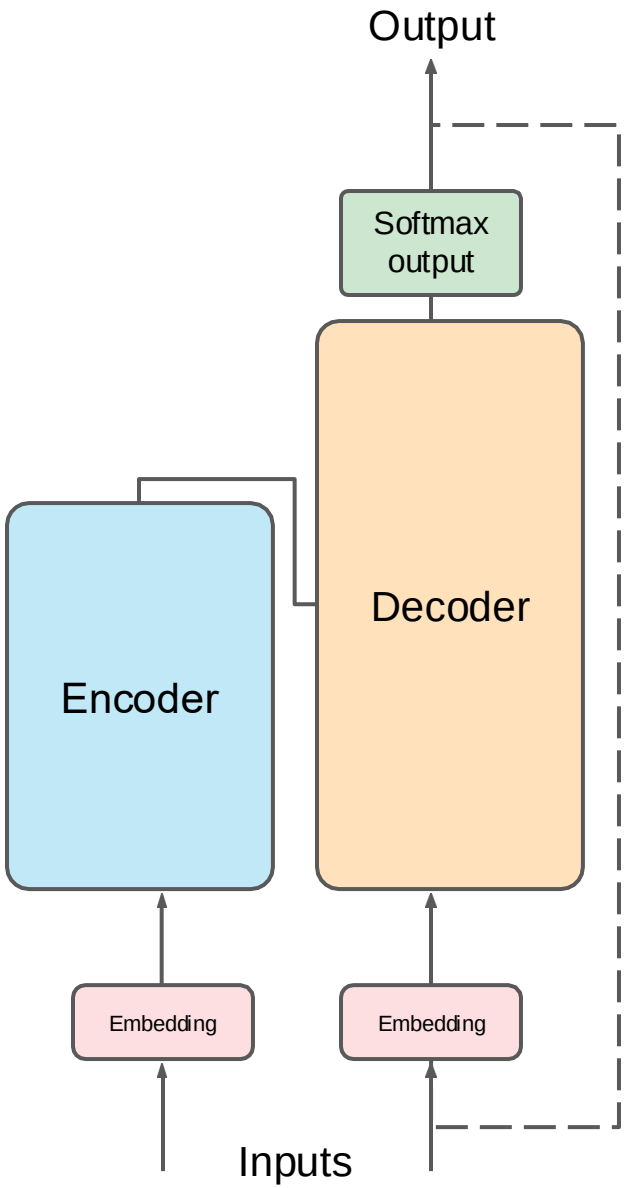
Outline: Part III

- **Transformers**
  - Processing Sequences
  - Transformers: High-level
  - Transformer block
  - **Types of Transformer architectures**
  - Scaling-up Transformers
  - Vision Transformer

Transformers

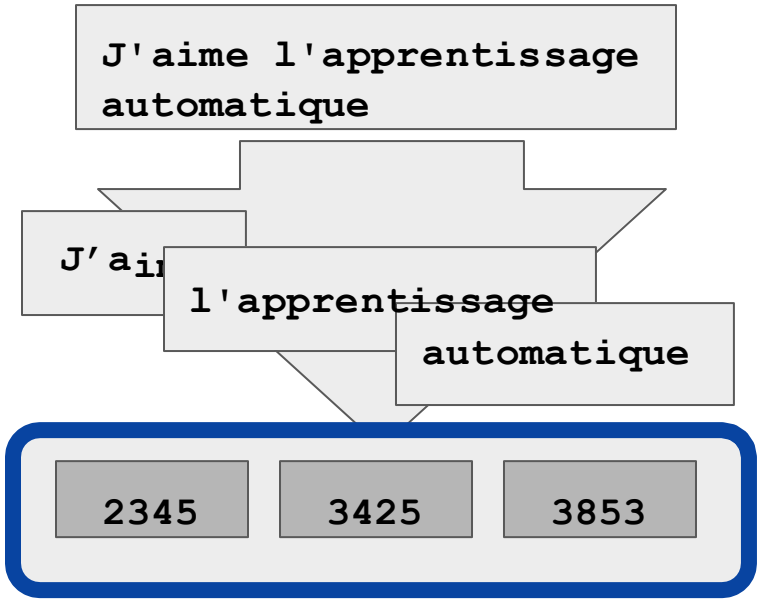
Translation:
sequence-to-sequence task

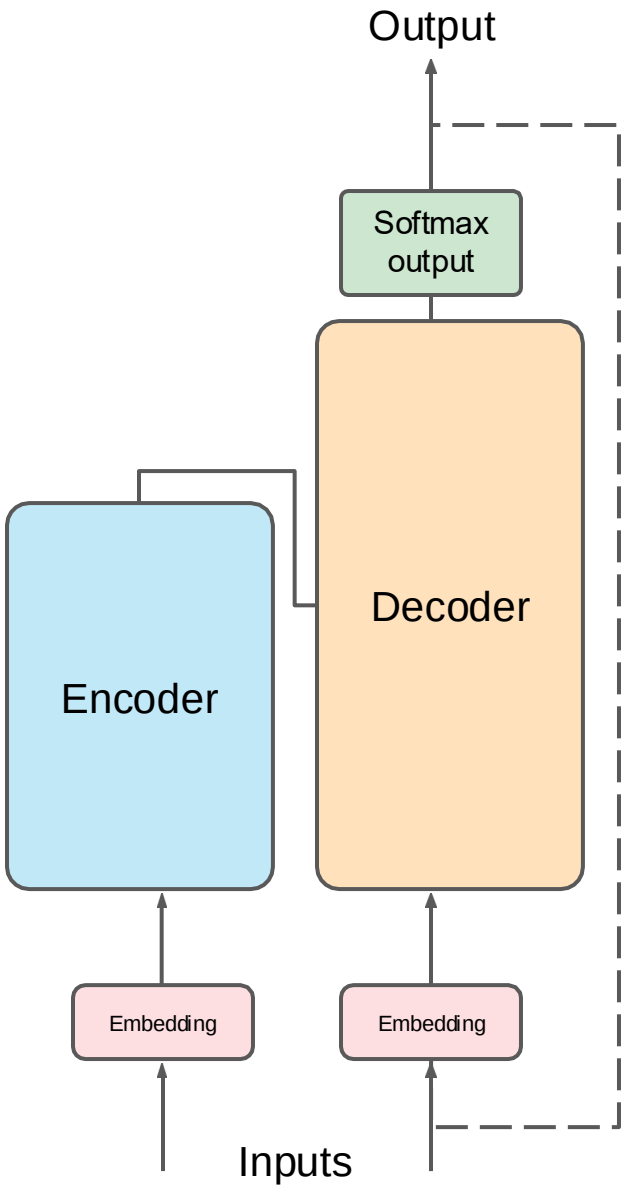
```
J'aime l'apprentissage automatique
```



Transformers

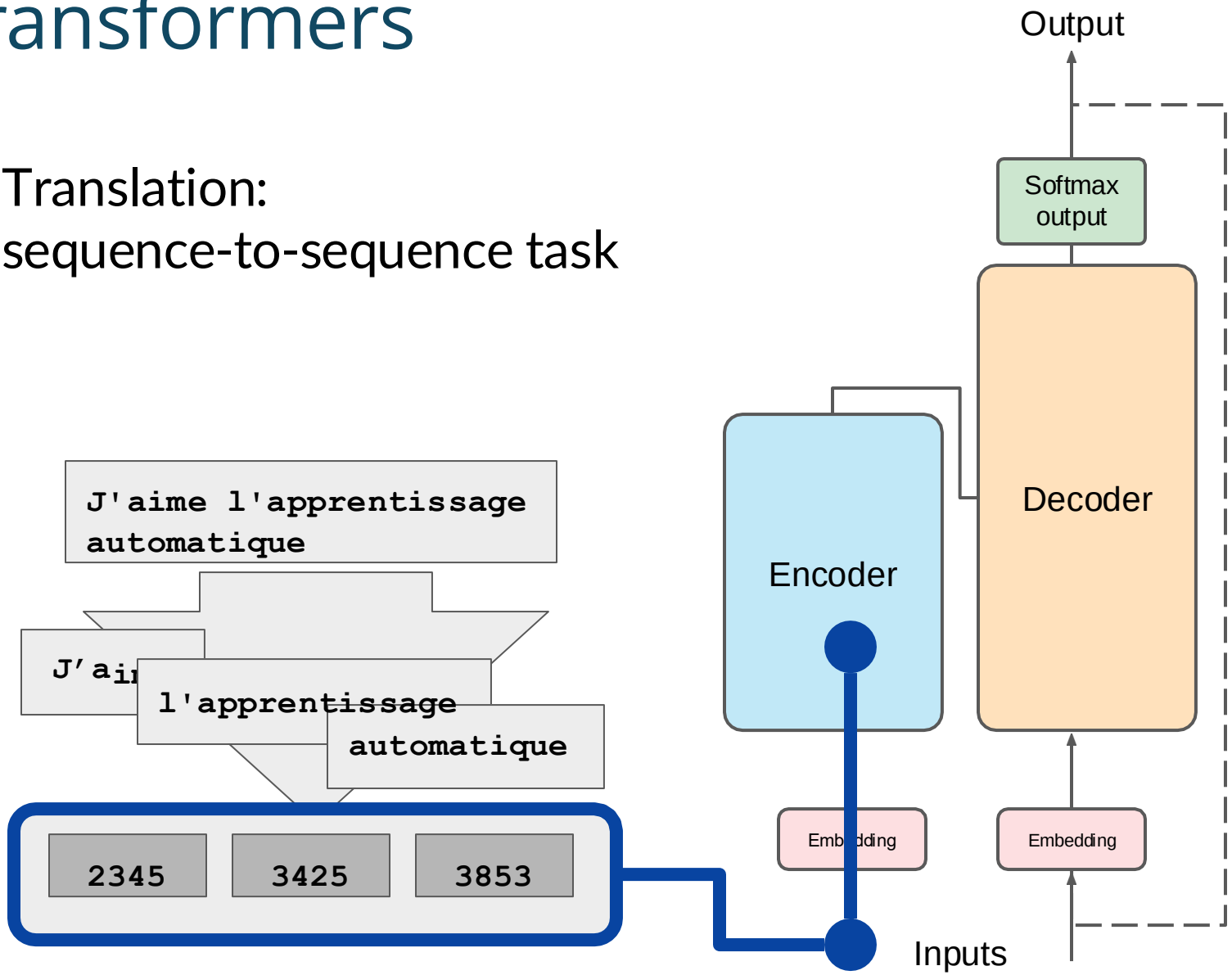
Translation:
sequence-to-sequence task





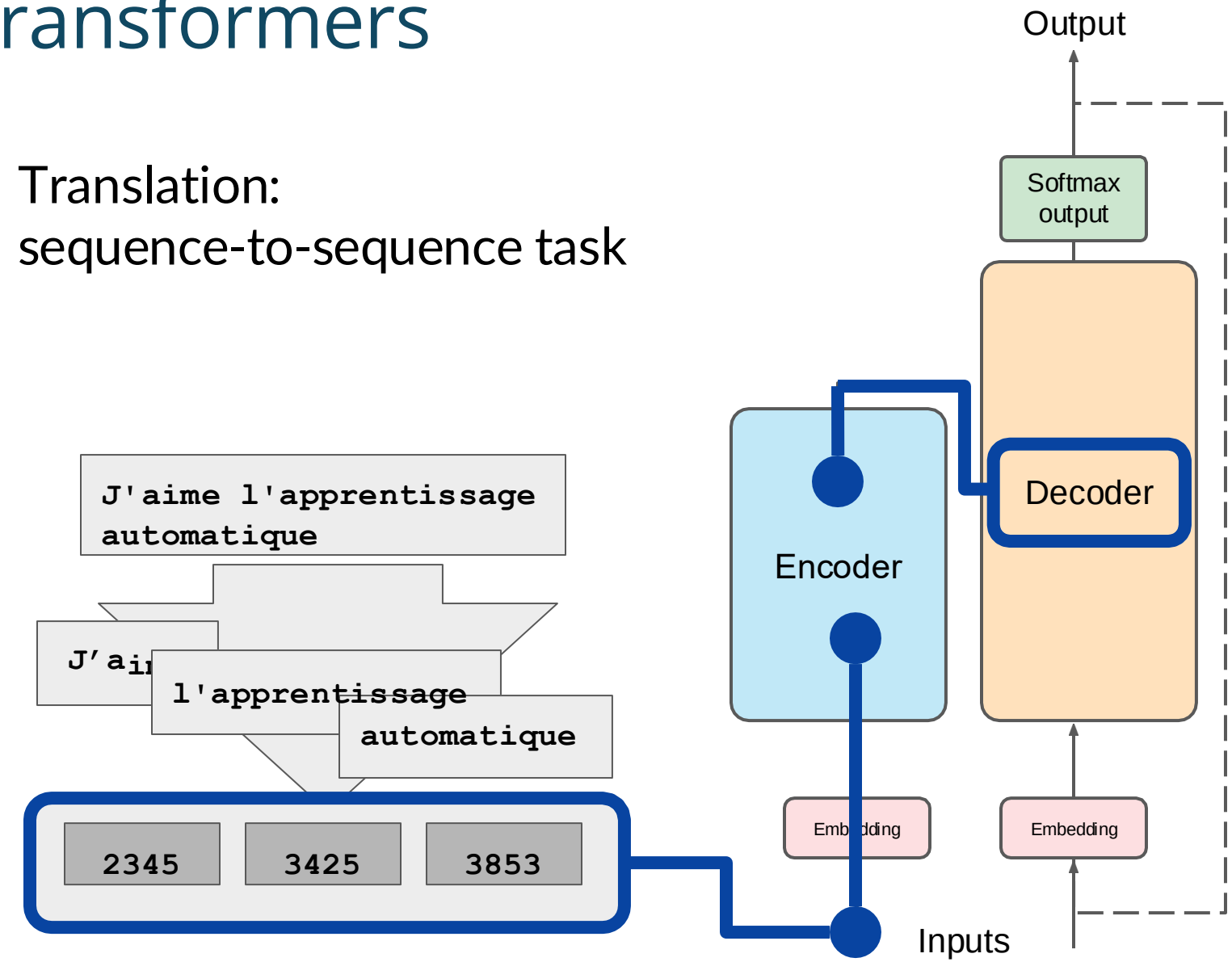
Transformers

Translation:
sequence-to-sequence task



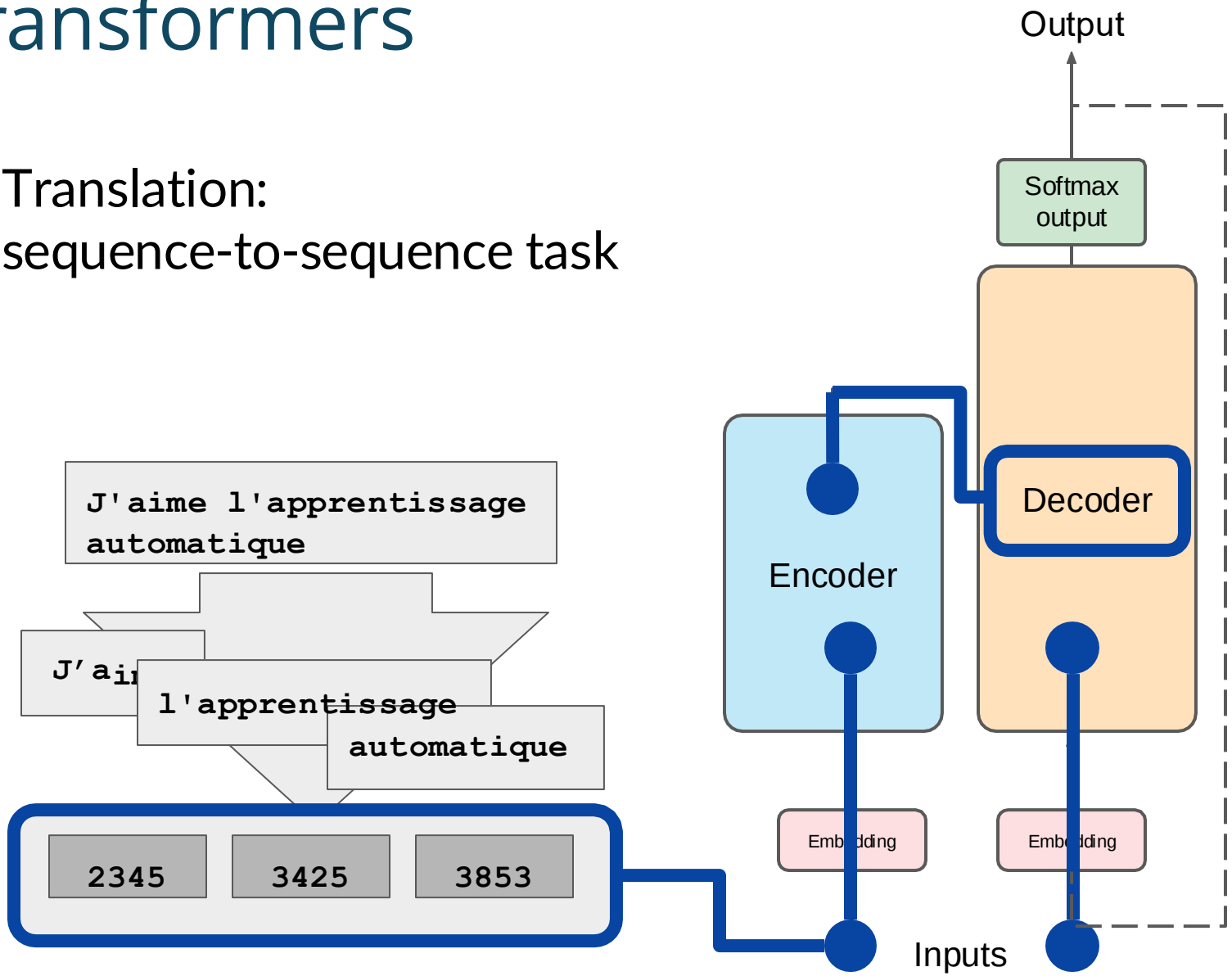
Transformers

Translation:
sequence-to-sequence task



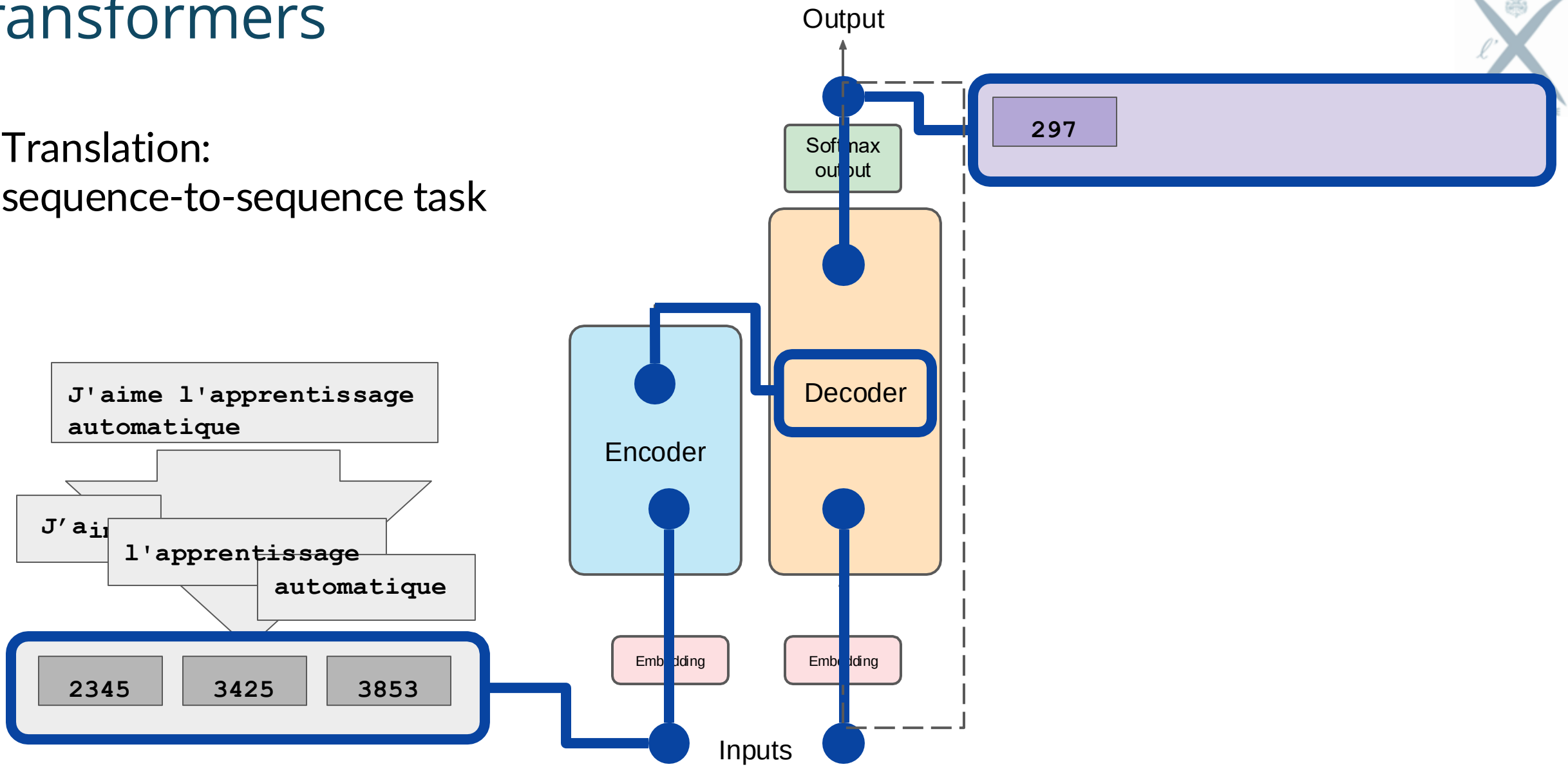
Transformers

Translation:
sequence-to-sequence task



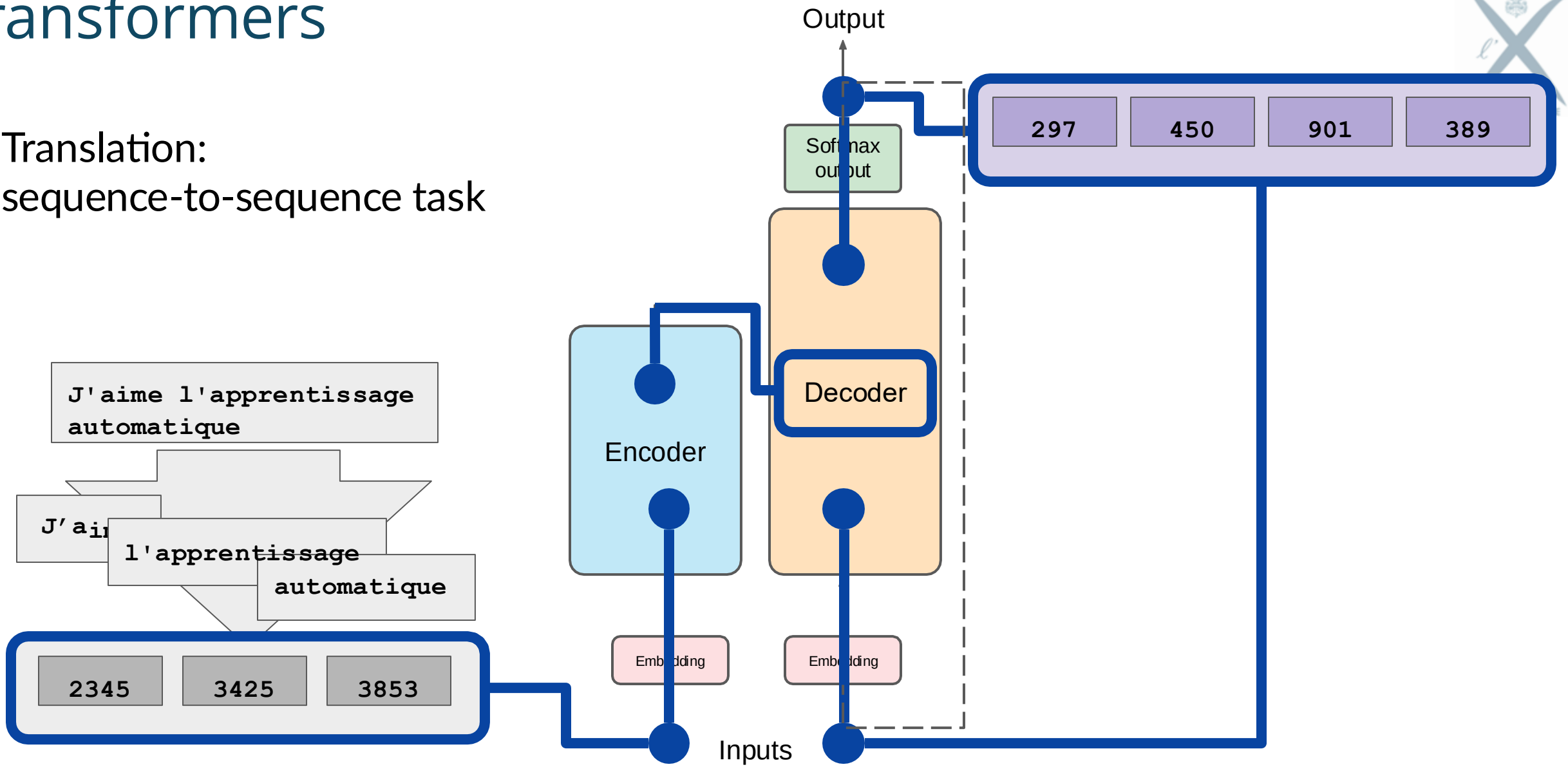
Transformers

Translation:
sequence-to-sequence task



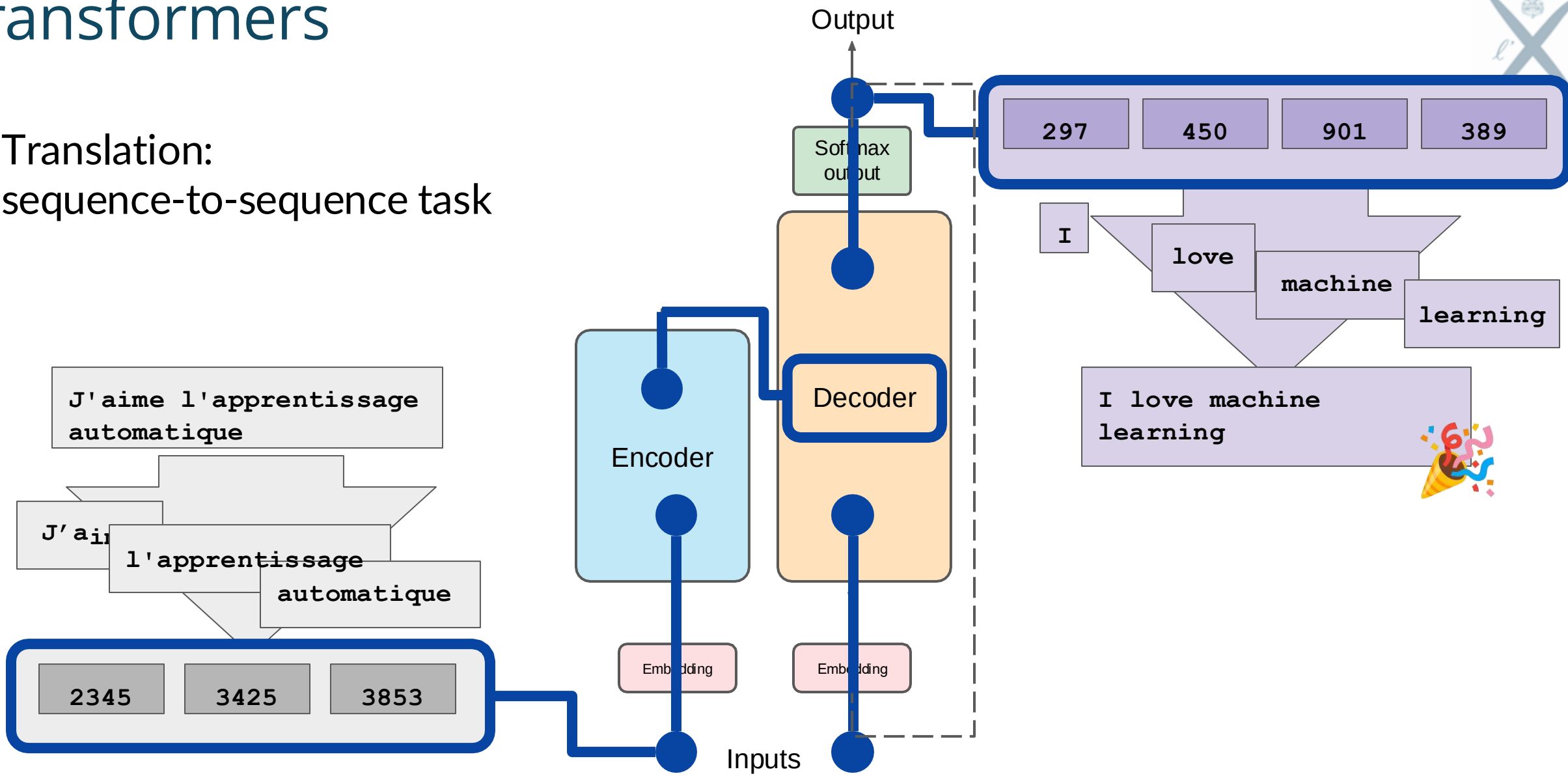
Transformers

Translation:
sequence-to-sequence task

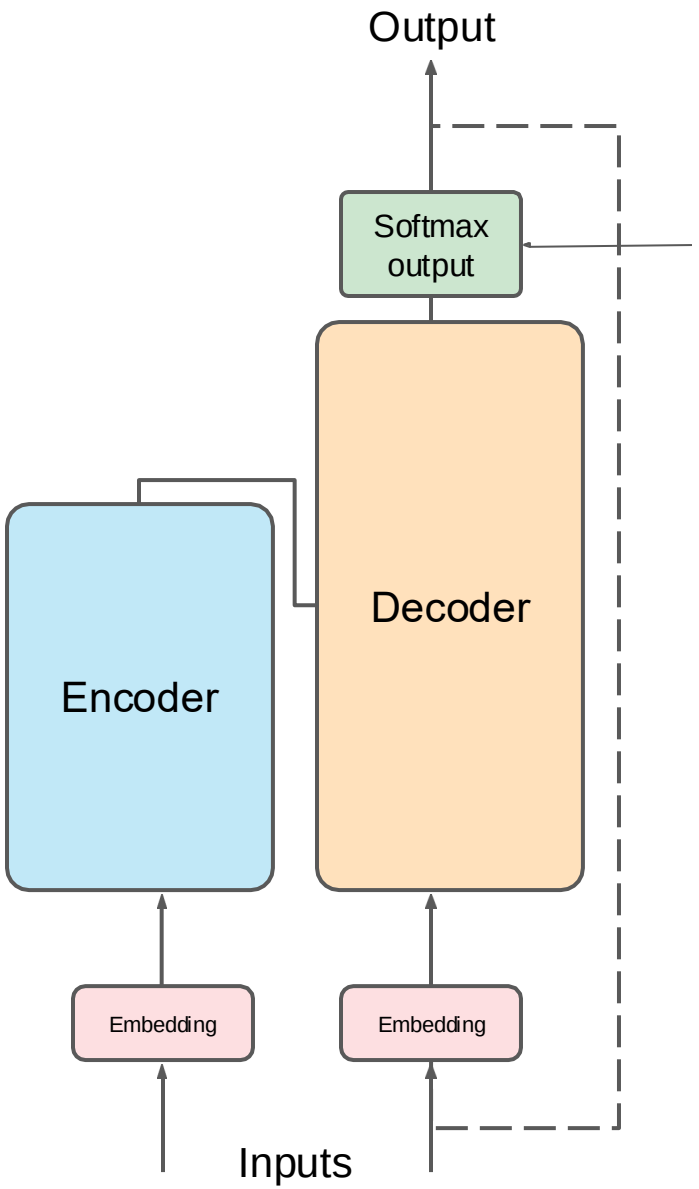


Transformers

Translation:
sequence-to-sequence task



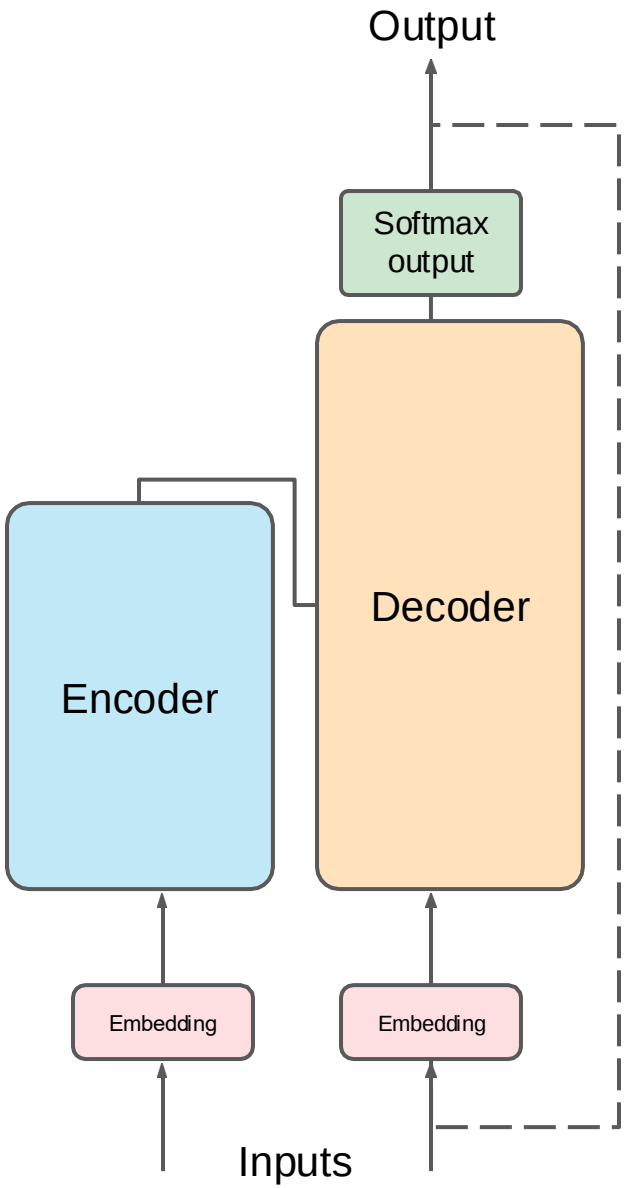
Transformers



Softmax output

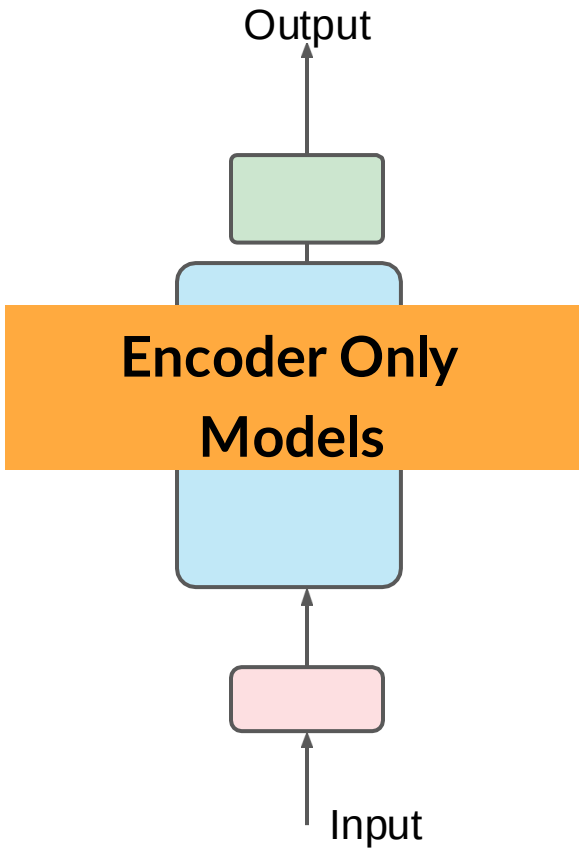
Transformers

Encoder
Encodes inputs (“prompts”) with contextual understanding and produces one vector per input token.

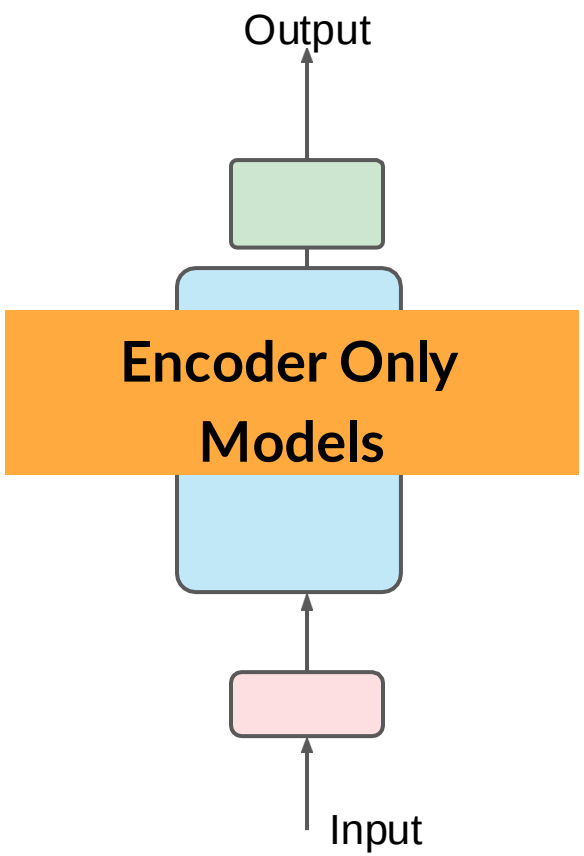


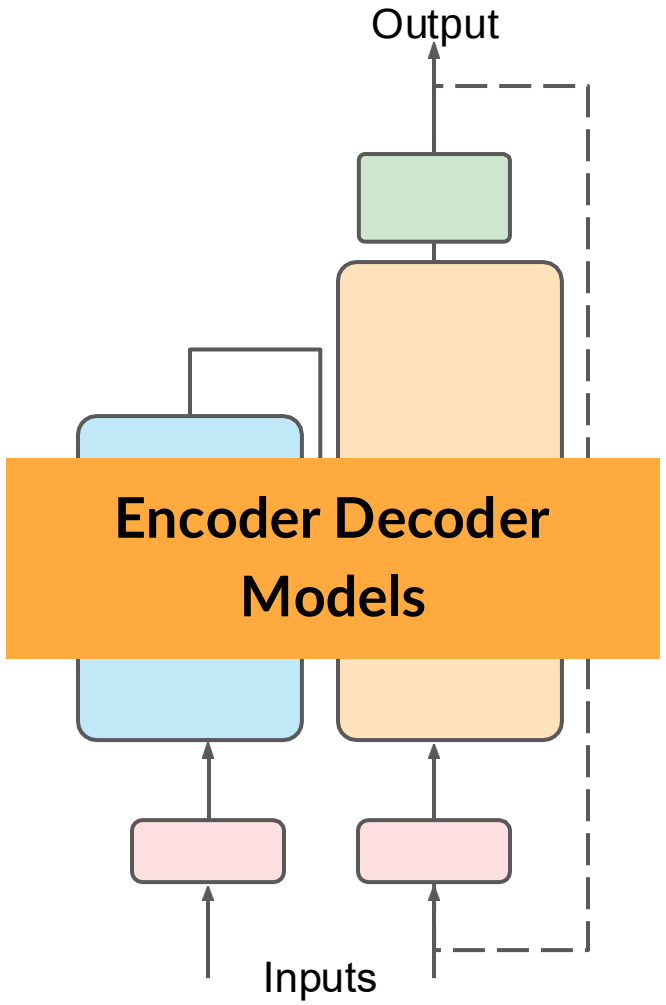
Decoder
Accepts input tokens and generates new tokens.

Transformers

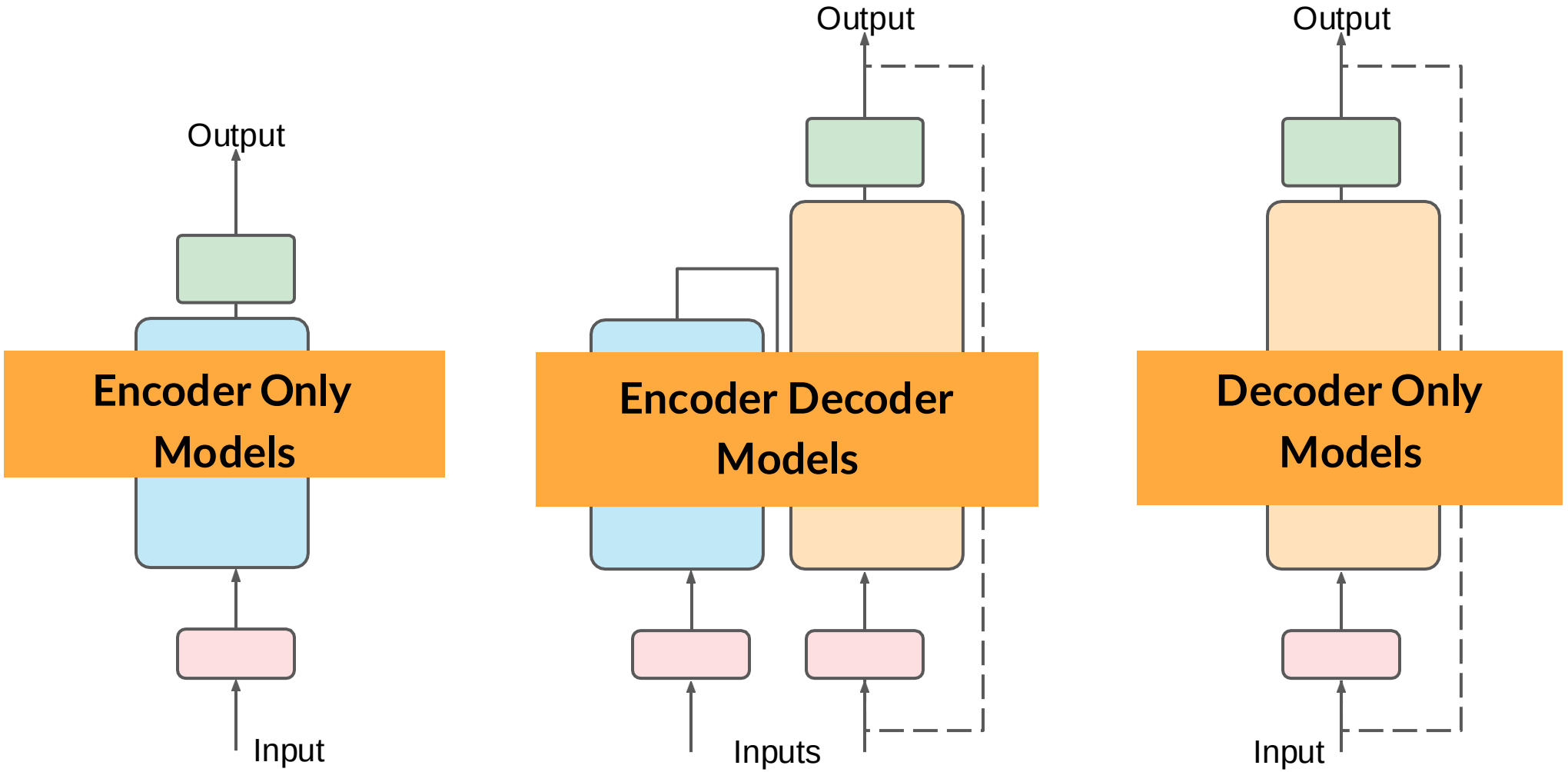


Transformers





Transformers



Outline: Part III

- **Transformers**
  - Processing Sequences
  - Transformers: High-level
  - Transformer block
  - Types of Transformer architectures
  - **Scaling-up Transformers**
  - Vision Transformer

Scaling up Transformers

Model	Layers	Width	Heads	Params	Data	Training
Transformer Base	12	512	8	65M		8xP100 (12h)
Transformer Large	12	1024	16	213M		8xP100 (12h)

Scaling up Transformers

Model	Layers	Width	Heads	Params	Data	Training
Transformer Base	12	512	8	65M		8xP100 (12h)
Transformer Large	12	1024	16	213M		8xP100 (12h)
Bert Base	12	768	12	110M	13GB	
Bert Large	24	1024	16	340M	13GB	

Scaling up Transformers

Model	Layers	Width	Heads	Params	Data	Training
Transformer Base	12	512	8	65M		8xP100 (12h)
Transformer Large	12	1024	16	213M		8xP100 (12h)
Bert Base	12	768	12	110M	13GB	
Bert Large	24	1024	16	340M	13GB	
XLNet Large	24	1024	16	~340M	126GB	512xTPUv3 (2.5 days)
RoBERTa	24	1024	16	355M	160GB	1024xV100 GPU (1 day)

Scaling up Transformers

Model	Layers	Width	Heads	Params	Data	Training
Transformer Base	12	512	8	65M		8xP100 (12h)
Transformer Large	12	1024	16	213M		8xP100 (12h)
Bert Base	12	768	12	110M	13GB	
Bert Large	24	1024	16	340M	13GB	
XLNet Large	24	1024	16	~340M	126GB	512xTPUv3 (2.5 days)
RoBERTa	24	1024	16	355M	160GB	1024xV100 GPU (1 day)
GPT-2	48	1600	?	1.5B	40GB	

Scaling up Transformers

Model	Layers	Width	Heads	Params	Data	Training
Transformer Base	12	512	8	65M		8xP100 (12h)
Transformer Large	12	1024	16	213M		8xP100 (12h)
Bert Base	12	768	12	110M	13GB	
Bert Large	24	1024	16	340M	13GB	
XLNet Large	24	1024	16	~340M	126GB	512xTPUv3 (2.5 days)
RoBERTa	24	1024	16	355M	160GB	1024xV100 GPU (1 day)
GPT-2	48	1600	?	1.5B	40GB	
Megatron-LM	72	2072	32	8.3B	174GB	512xV100 GPU (9 days)
Turing NLG	78	4256	28	17B	?	256xV100 GPU

Pop Quiz

Model	Layers	Width	Heads	Params	Data	Training
Transformer Base	12	512	8	65M		8xP100 (12h)
Transformer Large	12	1024	16	213M		8xP100 (12h)
Bert Base	12	768	12	110M	13GB	
Bert Large	24	1024	16	340M	13GB	
XLNet Large	24	1024	16	~340M	126GB	512xTPUv3 (2.5 days)
RoBERTa	24	1024	16	355M	160GB	1024xV100 GPU (1 day)
GPT-2	48	1600	?	1.5B	40GB	
Megatron-LM	72	2072	32	8.3B	174GB	512xV100 GPU (9 days)
Turing NLG	78	4256	28	17B	?	256xV100 GPU

Scaling up Transformers

Model	Layers	Width	Heads	Params	Data	Training
Transformer Base	12	512	8	65M		8xP100 (12h)
Transformer Large	12	1024	16	213M		8xP100 (12h)
Bert Base	12	768	12	110M	13GB	
Bert Large	24	1024	16	340M	13GB	
XLNet Large	24	1024	16	~340M	126GB	512xTPUv3 (2.5 days)
RoBERTa	24	1024	16	355M	160GB	1024xV100 GPU (1 day)
GPT-2	48	1600	?	1.5B	40GB	
Megatron-LM	72	2072	32	8.3B	174GB	512xV100 GPU (9 days)
Turing NLG			28	17B	?	256xV100 GPU

~350k euros!

Scaling up Transformers

Model	Layers	Width	Heads	Params	Data	Training
Transformer Base	12	512	8	65M		8xP100 (12h)
Transformer Large	12	1024	16	213M		8xP100 (12h)
Bert Base	12	768	12	110M	13GB	
Bert Large	24	1024	16	340M	13GB	
XLNet Large	24	1024	16	~340M	126GB	512xTPUv3 (2.5 days)
RoBERTa	24	1024	16	355M	160GB	1024xV100 GPU (1 day)
GPT-2	48	1600	?	1.5B	40GB	
Megatron-LM	72	2072	32	8.3B	174GB	512xV100 GPU (9 days)
Turing NLG	78	4256	28	17B	?	256xV100 GPU
GPT-3	96	12288	96	175B	694GB	

Examples

http://www.youtube.com/watch?v=_x9AwxfjxvE&t=1m45s

Outline: Part III

- **Transformers**
  - Processing Sequences
  - Transformers: High-level
  - Transformer block
  - Types of Transformer architectures
  - Scaling-up Transformers
  - **Vision Transformer**

Vision Transformers

Vision Transformers

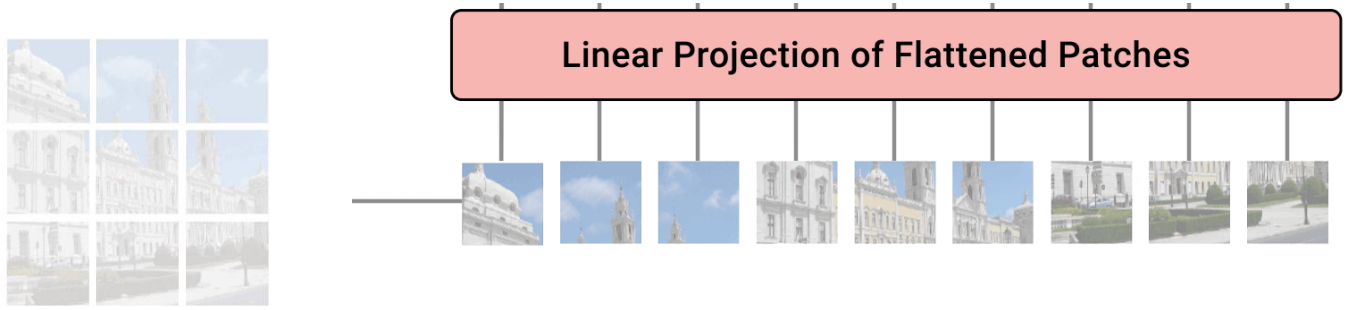
1. Split an image into patches

Vision Transformers

1. Split an image into patches
2. Flatten the patches

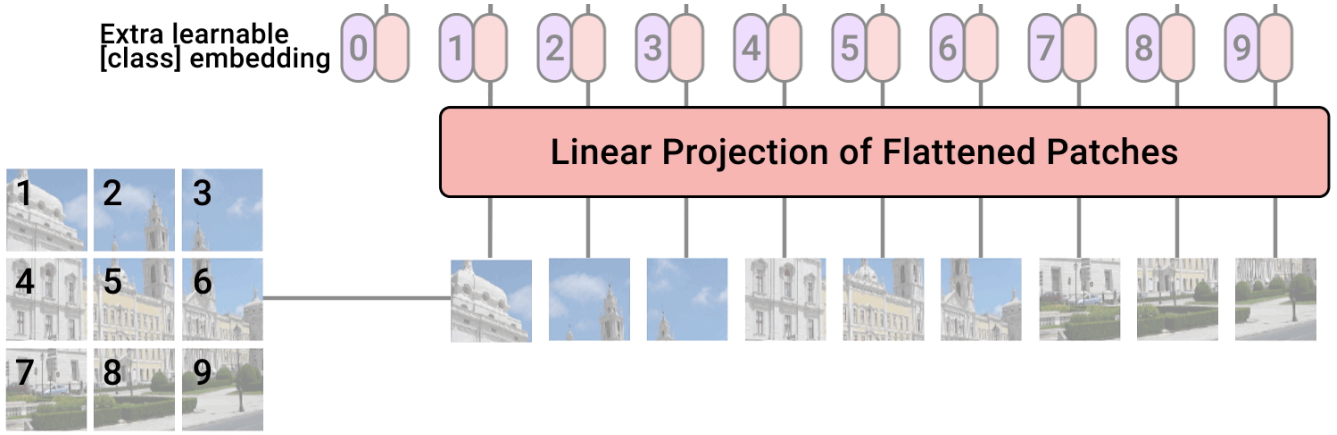
Vision Transformers

1. Split an image into patches
2. Flatten the patches
3. Produce lower-dimensional linear embeddings from the flattened patches



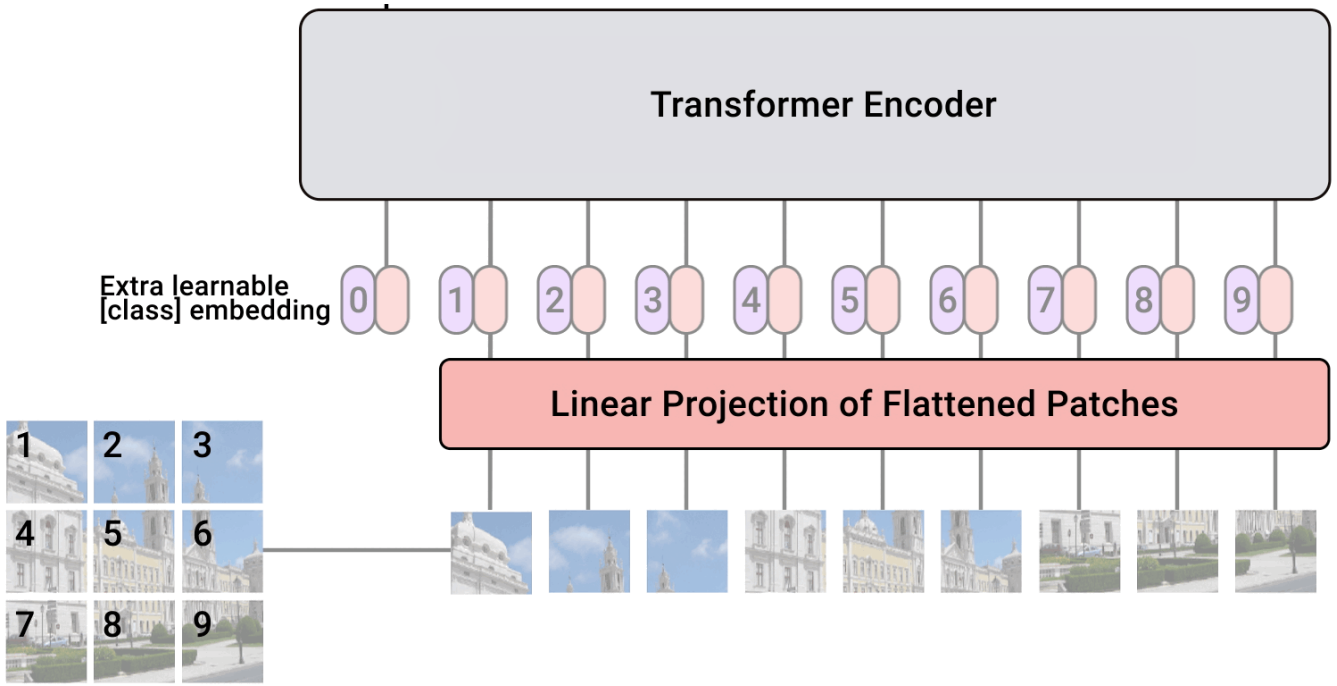
Vision Transformers

1. Split an image into patches
2. Flatten the patches
3. Produce lower-dimensional linear embeddings from the flattened patches
4. Add positional embeddings

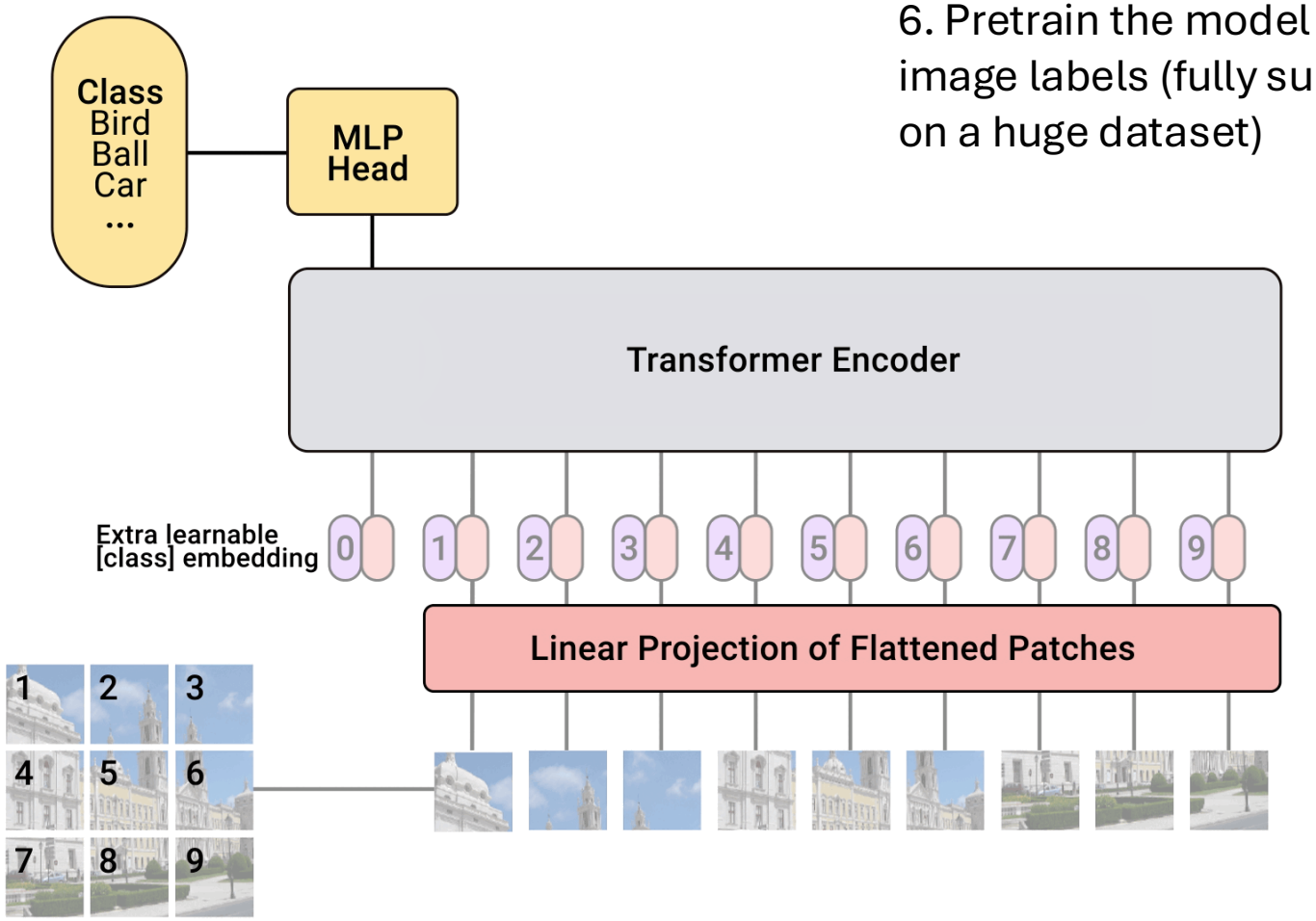


Vision Transformers

5. Feed the sequence as an input to a standard transformer encoder



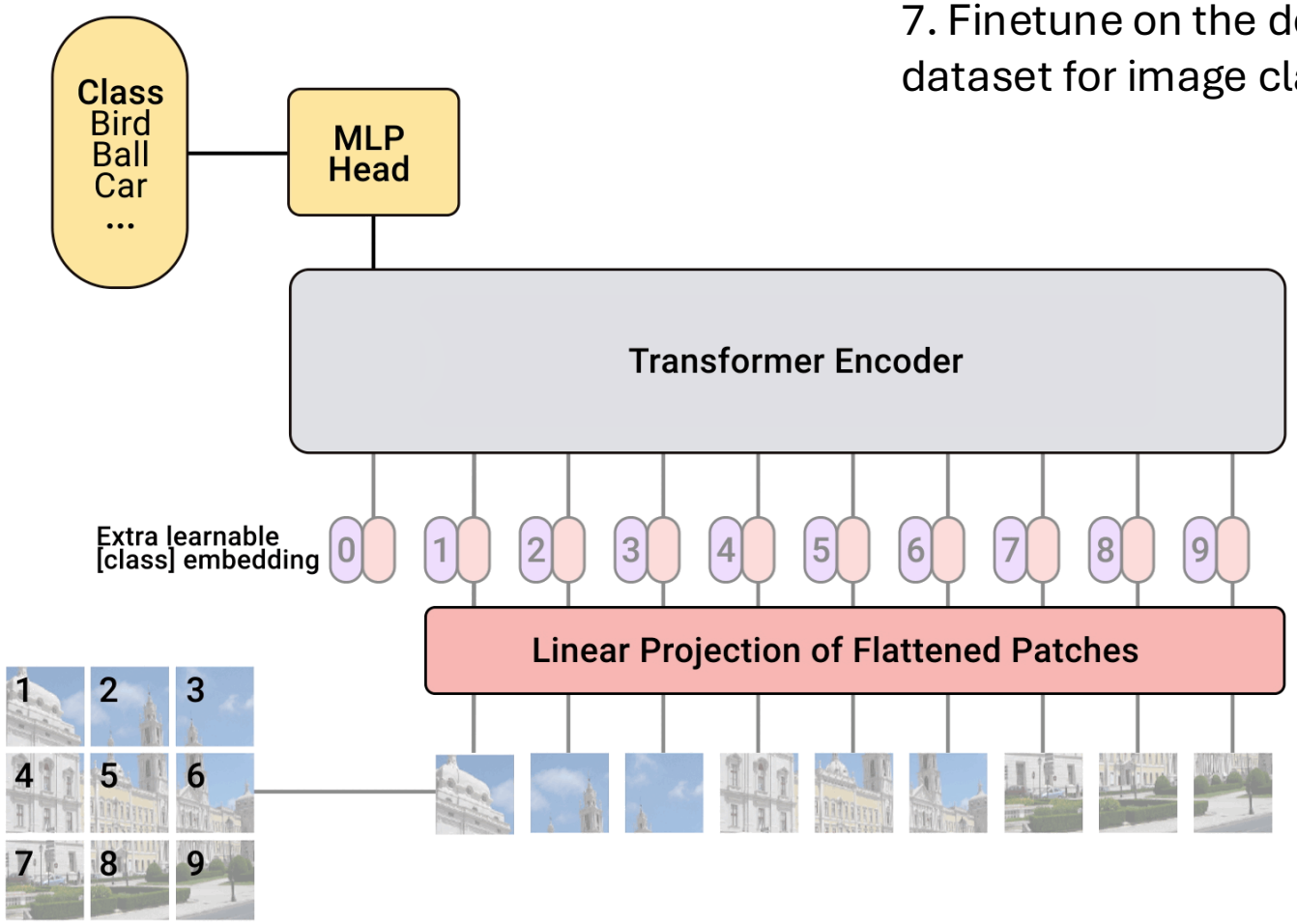
Vision Transformers



6. Pretrain the model with image labels (fully supervised on a huge dataset)

Vision Transformers

7. Finetune on the downstream dataset for image classification



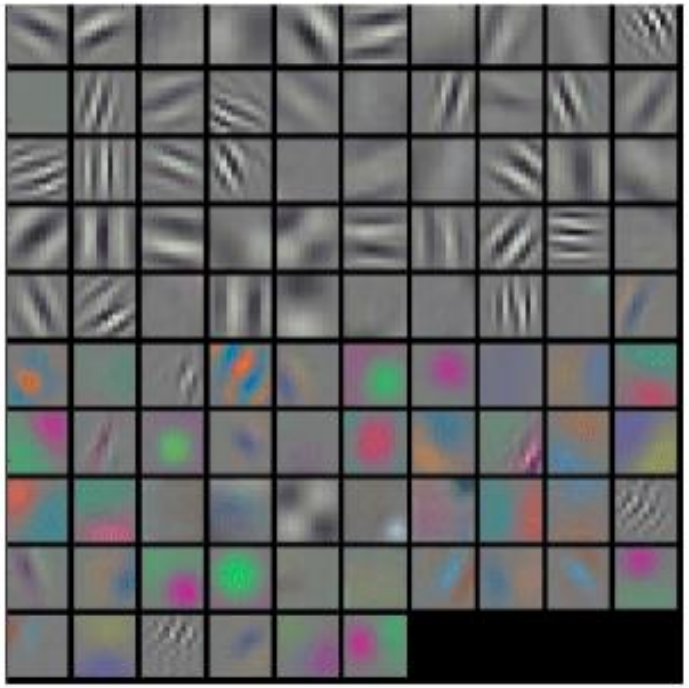
Vision Transformers

Vision Transformers

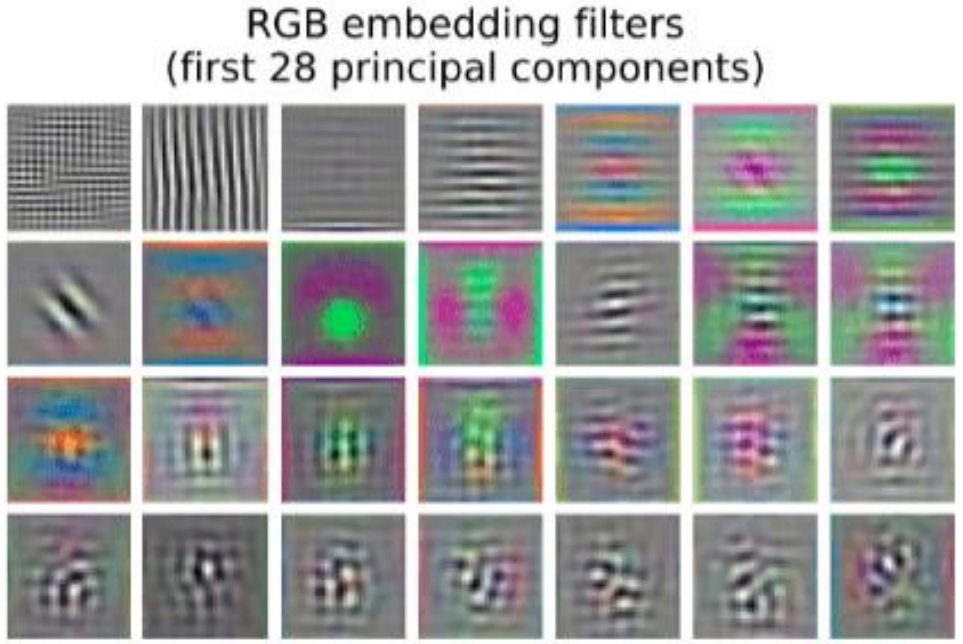
Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Vision Transformers

Alexnet 1st conv filters



ViT 1st linear embedding filters



Thank you