Week 4

From Sequence Modeling to Transformer

[ECEA0649/ECE40049] Deep Learning for Image Processing | Spring 2026

Kihyun Na (Research Professor)

BK21 AI Project Group & Institute for Information and Communication Technology,
Handong Global University

HANDONG GLOBAL UNIVERSITY

Curriculum

** Adjusted based on survey results*

Week	Topic
Wk 1	OT + Introduction
Wk 2	DL Fundamentals Review
Wk 3	CNN
Wk 4	From Sequence Modeling to Transformer (Today)
Wk 5	Transformer in Vision
Wk 6	Detection + Segmentation
Wk 7	Self-Supervised Learning
Wk 8	Review Literacy + Role Explanation
Wk 9	Foundation Models (CLIP, SAM) + Paper #1
Wk 10	Diffusion Models + Paper #2
Wk 11	Conditional Generation + Paper #3
Wk 12	Vision-Language Models + Paper #4
Wk 13	VLM Applications + Paper #5
Wk 14	Video Understanding + Paper #6
Wk 15	Embodied AI & Robot Vision + Paper #7
Wk 16	Miniconference (Final Project)

Lecture　　Lecture + Paper　　Miniconference

Today's Agenda

01

Why Sequences Matter in Vision

Image captioning, RNN basics, vanishing gradients, LSTM/GRU/SSM

15 min

02

Attention Mechanism

Seq2Seq bottleneck → Bahdanau → Image Captioning with Attention

20 min

03

Self-Attention & Multi-Head

Q/K/V, scaled dot-product, multi-head, RNN vs CNN vs SA, attention variants

30 min

04

The Transformer

Three components: Tokenization, Attention as mixer, Positional Embedding

25 min

Part 1

Why Sequences Matter in Vision

CNNs classify. But what if we need to generate, describe, or reason?

When Vision Needs Sequence

Image Captioning

one to many

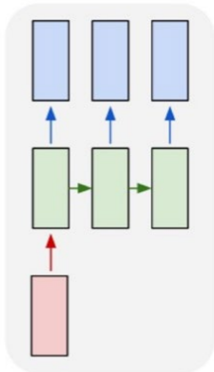
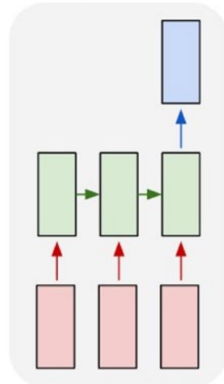


Image → Sequence of words

Action Prediction

many to one



Frame sequence
→ Action class

Video Captioning

many to many

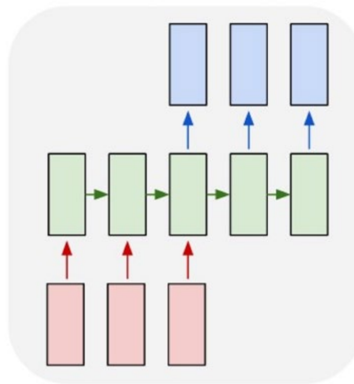
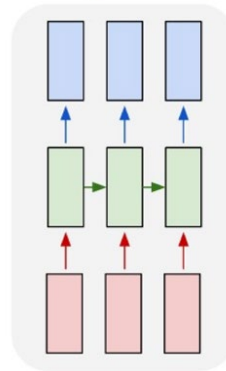


Image → Text
sequence

Frame-level Video Classification

many to many

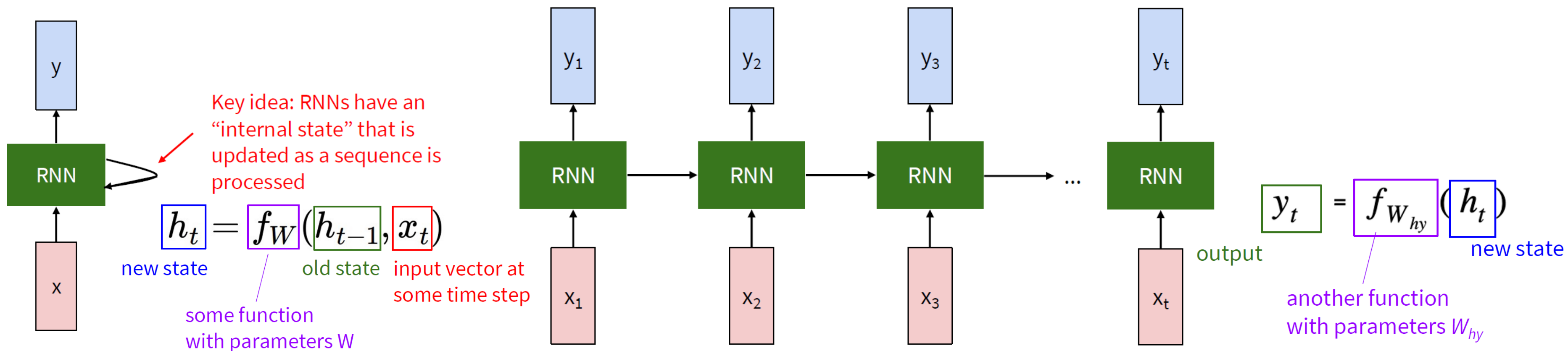


Frame sequence
→ Per-frame class

All require processing or generating ordered sequences. CNNs alone can't do this.

(Vanilla) Recurrent Neural Networks

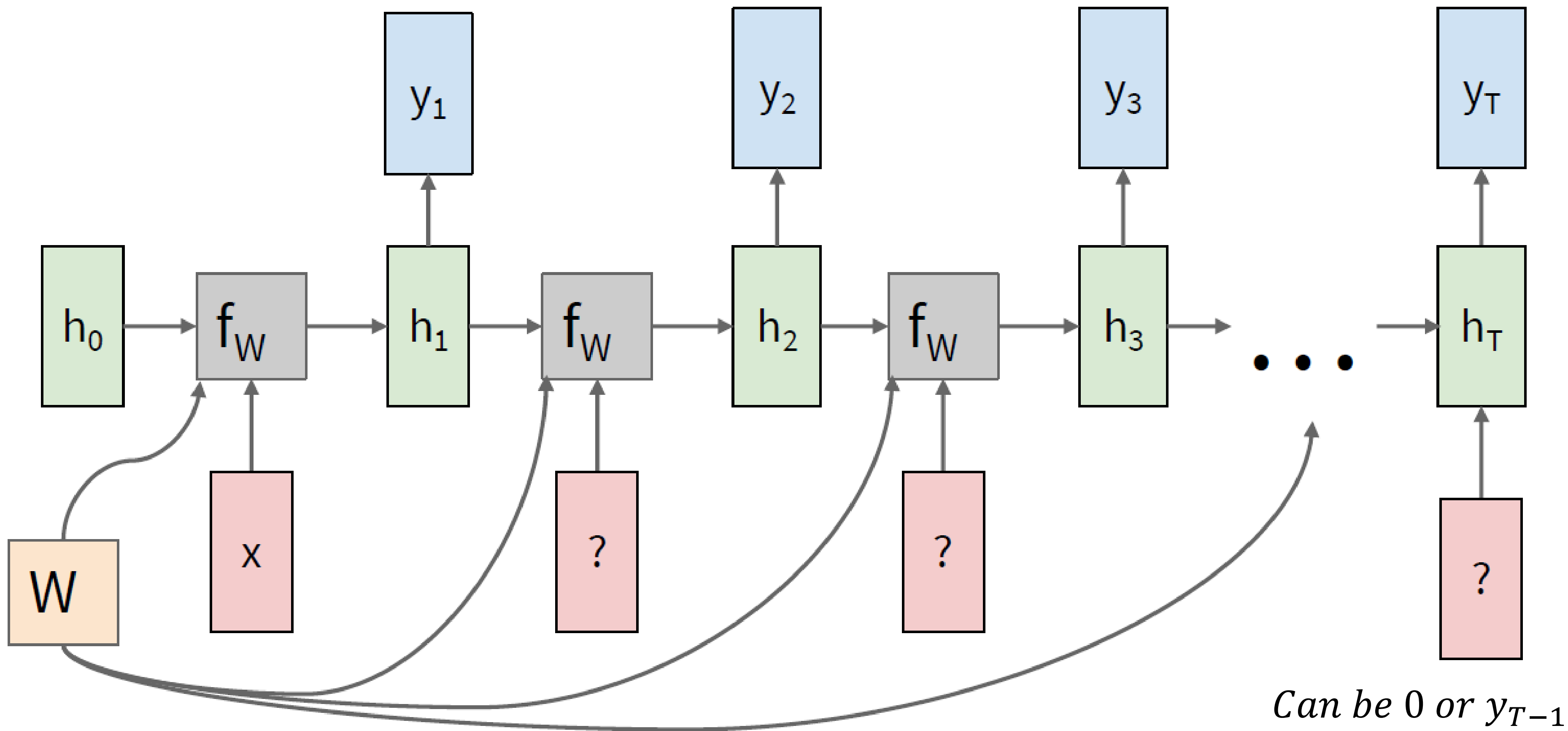
Process sequences one step at a time, maintaining a hidden state as "memory".



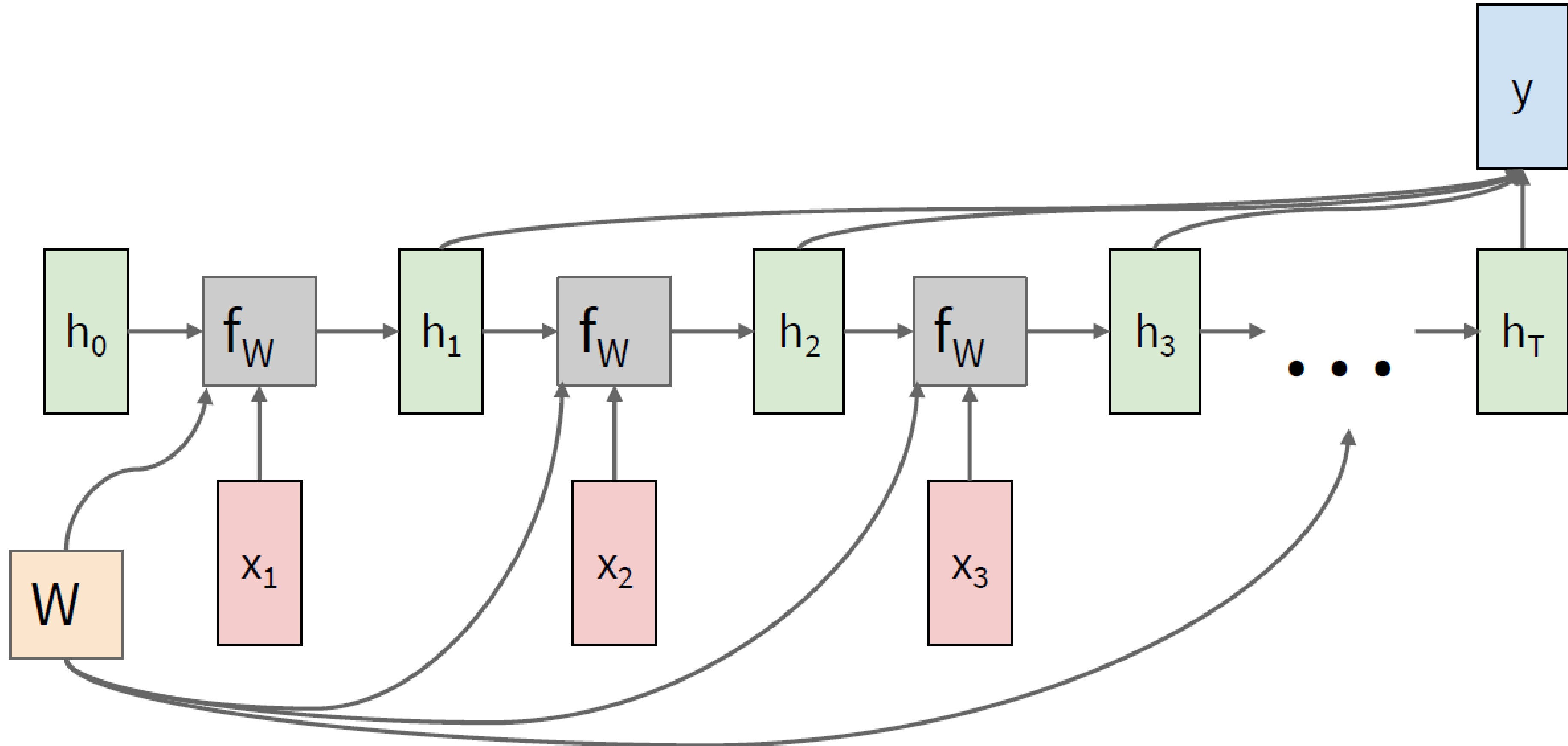
$$h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t + b) \qquad y_t = W_{hy} \cdot h_t$$

Same weights at every step = parameter sharing. But sequential processing = can't parallelize.

(Abstraction level) RNN: One to Many



(Abstraction level) RNN : Many to One



(Abstraction level) RNN : Many to Many

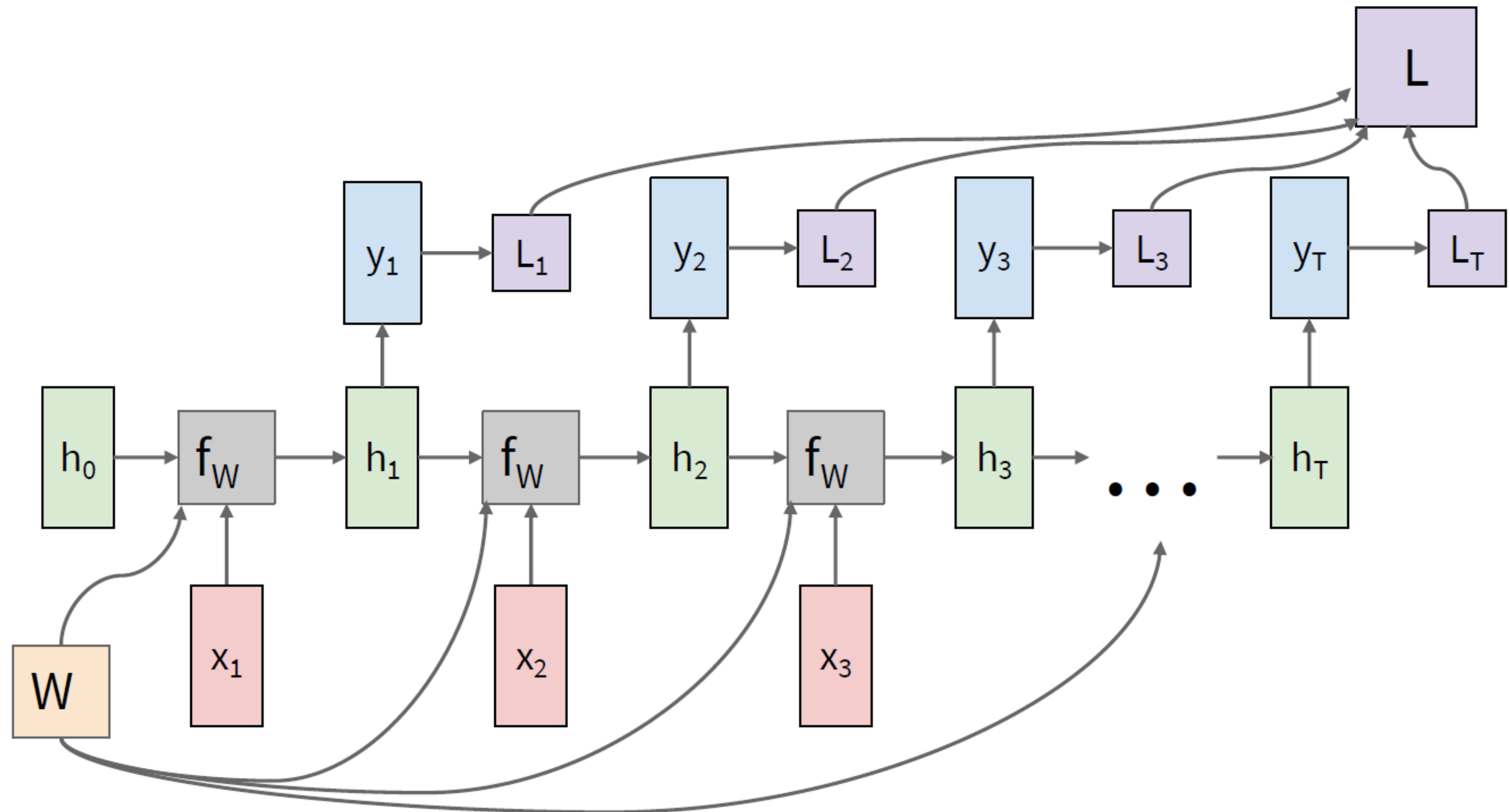
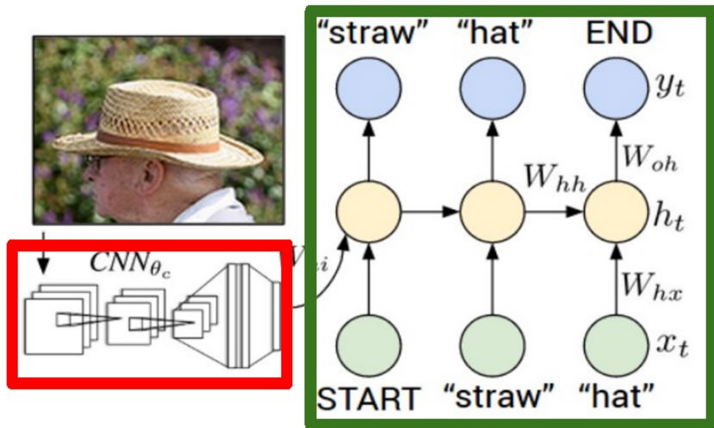
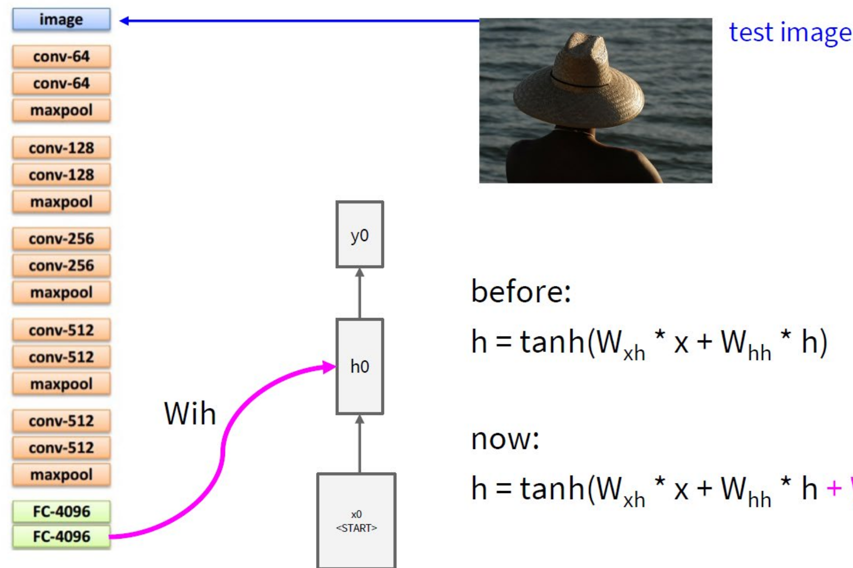


Image Captioning Example





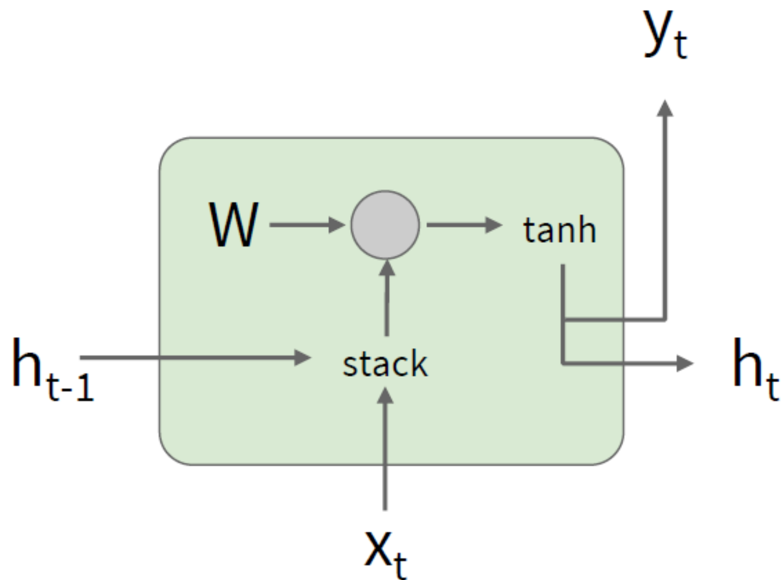
A tennis player in action on the court

A cat sitting on a suitcase on the floor

Captions generated using neuraltalk2
All images are CC0 Public domain: cat suitcase, cat tree, dog, bear, surfers, tennis, giraffe, motorcycle

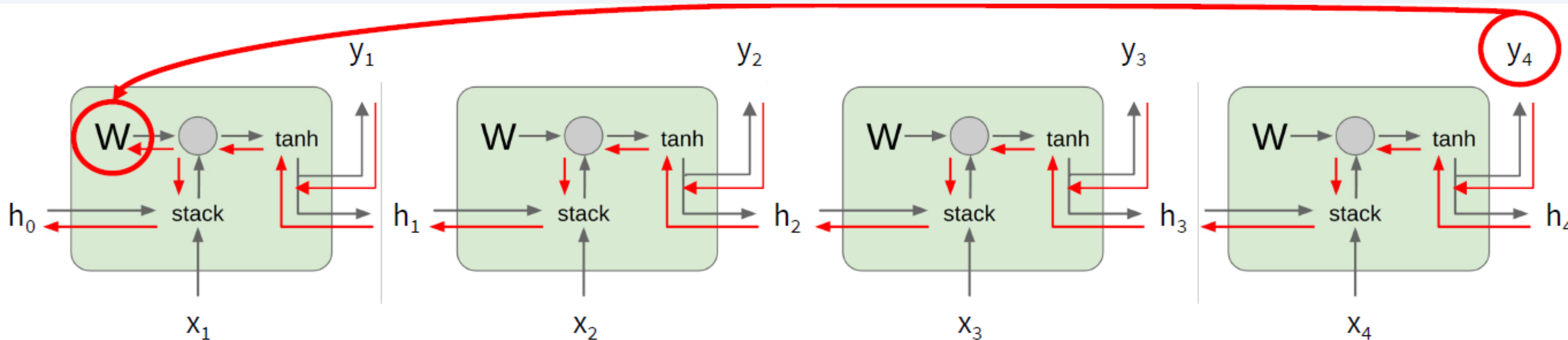
RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



$$
\begin{aligned}
h_t &= \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
&= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \\
&= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)
\end{aligned}
$$

RNN Gradient Flow



$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

$$\frac{\partial h_t}{\partial h_{t-1}} = tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}}\right)\frac{\partial h_1}{\partial W}$$

RNN Gradient Flow

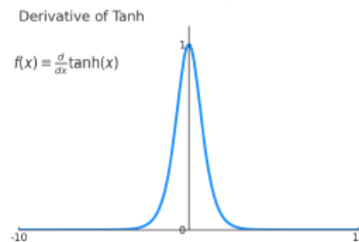
y1 y2 y3 y4

W tanh stack h0 h1 h2 h3 h4 x1 x2 x3 x4

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

Almost always < 1

Vanishing gradients



$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} tanh'(W_{hh}h_{t-1} + W_{xh}x_t)\right)W_{hh}^{T-1}\frac{\partial h_1}{\partial W}$$

RNN Variants: Long Short Term Memory

Vanilla RNN

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

LSTM

Four gates

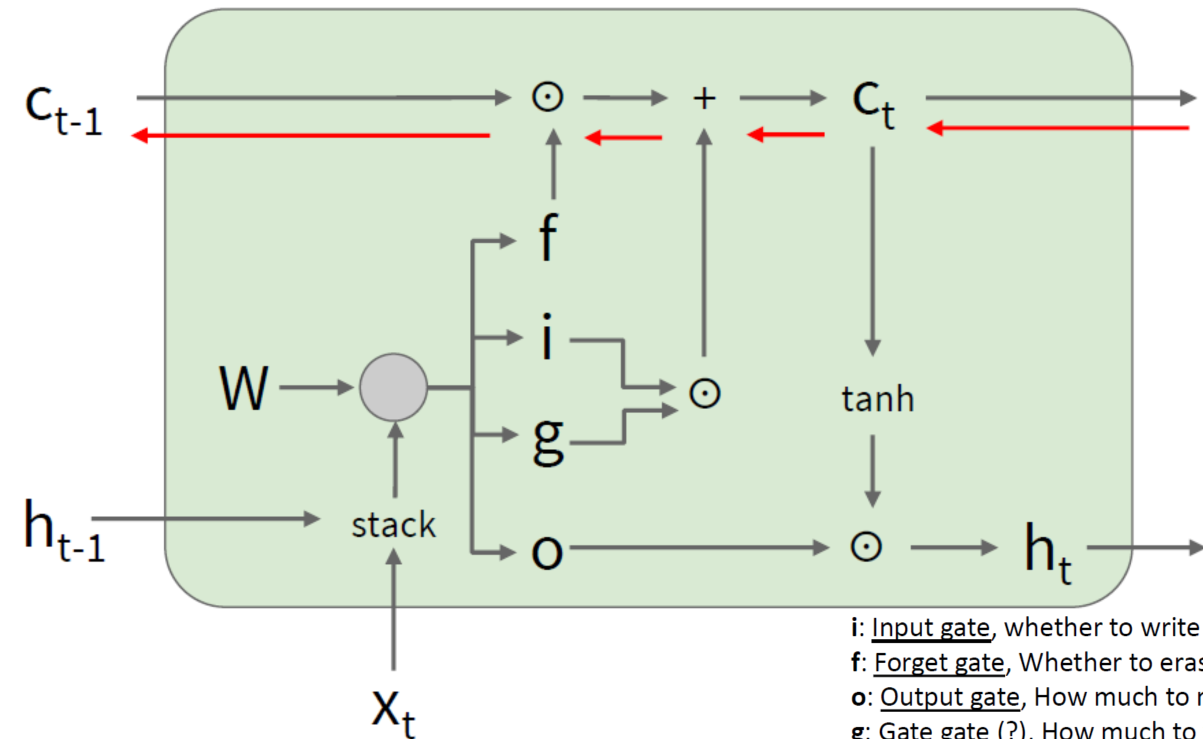
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

Cell state → $c_t = f \odot c_{t-1} + i \odot g$

Hidden state → $h_t = o \odot \tanh(c_t)$

Hochreiter and Schmidhuber, “Long Short Term Memory”, Neural Computation 1997

How LSTM mitigates vanishing gradients



Backpropagation from c_t to c_{t-1} only elementwise multiplication by f, no matrix multiply by W

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

i: Input gate, whether to write to cell

f: Forget gate, Whether to erase cell

o: Output gate, How much to reveal cell

g: Gate gate (?), How much to write to cell

The Sequential Bottleneck (w/RNNs)

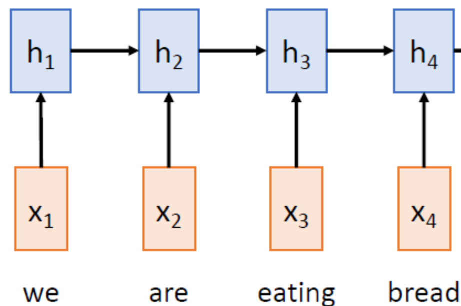
Input: Sequence $x_1, \dots x_T$

Output: Sequence $y_1, \dots, y_{T'}$

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

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

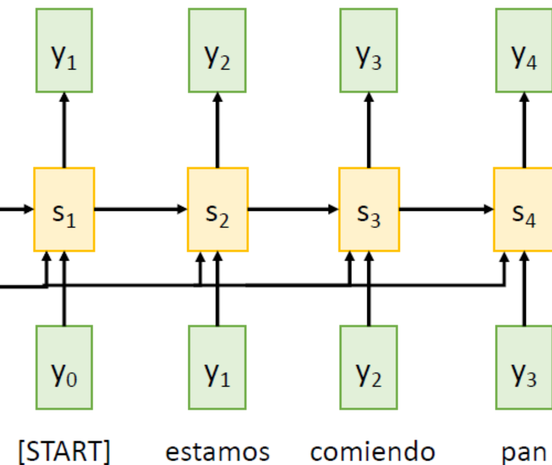
From final hidden state predict:
Initial decoder state s_0
Context vector c (often $c=h_T$)



s_0

c

Problem: Input sequence bottlenecked through fixed-sized vector. What if T=1000?



Idea: use new context vector at each step of decoder!

Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Problem
Entire input compressed to one vector.
Information loss inevitable for long sequences.

Solution
At each decoder step, look back at ALL encoder hidden states. This is Attention.

RNN Limitations

Vanishing Gradients

Gradients shrink exponentially when backpropagating through many time steps.

Early inputs are effectively forgotten in long sequences.

LSTM/GRU: gating mechanisms partially mitigate this.

Sequential Bottleneck

h_t depends on h_{t-1}.
Must process tokens one by one.

Cannot leverage GPU parallelism.
Slow training on long sequences.

No architectural fix within RNN.
This is fundamental.

Fixed Context Vector

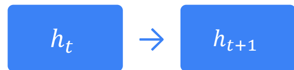
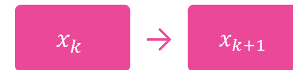
In Seq2Seq, entire input is compressed to a single vector c.

Information loss for long inputs.
"Trying to memorize a book in one sentence."

→ This is exactly what Attention will solve (Part 2).

Attempts to Fix RNNs

LSTM (1997) / GRU (2014) Added gating mechanisms to control what to remember/forget. Solved vanishing gradient, but still sequential → slow training. Standard in 2014–2017.

	RNN / LSTM / GRU	Transformer	SSM / Mamba
Training	One at a time. Must wait.	All at once. N×N matrix.	All at once. Linear operation.
Inference	 O(1) per step. Fast.	KV cache grows. O(N) per step.	 O(1) per step. Fast.

→ Wk 14: Video Understanding

	RNN/LSTM	Transformer	SSM/Mamba
Training	Slow (sequential)	Fast (parallel)	Fast (parallel)
Inference (per step)	Fast (O(1))	Expensive (O(N))	Fast (O(1))
Long-range	Hard (vanishing grad)	Easy (direct access)	Moderate
Expressiveness	High (non-linear)	Highest (attention)	Growing (selective)
Maturity	Legacy (2014–2017)	Dominant (2018→)	Emerging (2023→)

Attempts to Fix RNNs (Suppl.)

Each solves some problems but not all. The fundamental breakthrough comes from Attention (Part 2).

LSTM (1997)

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(\dots)$$

3 gates control information flow:
Forget: what to erase
Input: what to write
Output: what to reveal

Cell state c_t = gradient highway

✓ Vanishing gradient
✗ Still sequential (no parallel)
✗ Seq2Seq bottleneck remains

GRU (2014): simplified to 2 gates, similar performance.

SSM / Mamba (2023) — A Different Paradigm

Core idea: linear state space model from control theory

Continuous:
$$x'(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t)$$

discretize →

Discrete:
$$x_k = \bar{A}\, x_{k-1} + \bar{B} u_k$$
$$y_k = C\, x_k$$

The key insight: same math, two implementations

Recurrence View

$$x_k = \bar{A}\, x_{k-1} + \bar{B} u_k$$

Like RNN: step by step
✓ O(1) memory per step
✓ Fast inference
✗ Sequential (slow training)

=

Convolution View

$$y = u * \bar{K} \quad (\bar{K} \text{ from } C, \bar{A}, \bar{B})$$

Like CNN: one operation
✓ Fully parallel
✓ Fast training on GPU
✗ O(N) memory

Mamba adds: B, C, Δ become input-dependent (selective) → content-aware, like soft gating.

→ Wk 14: Video Understanding

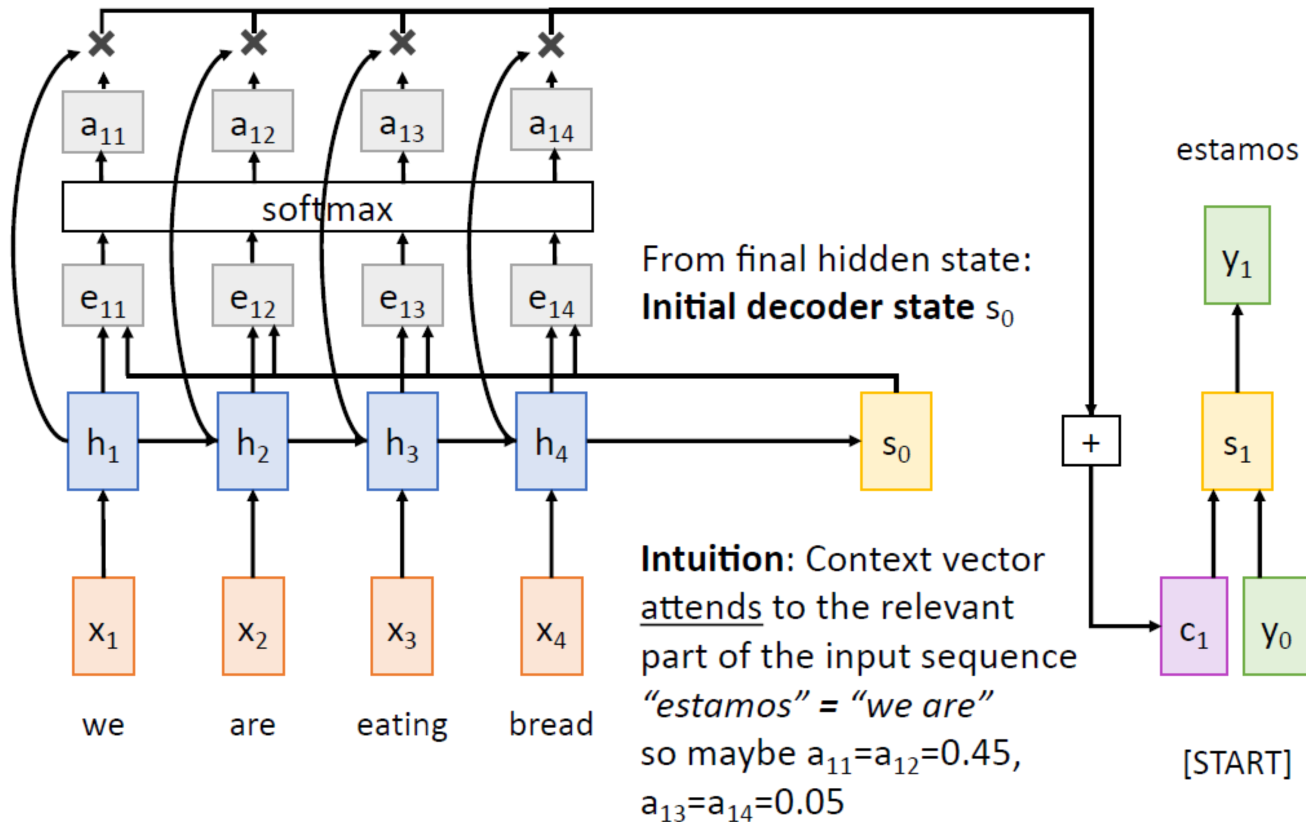
None of these solve the Seq2Seq bottleneck. **→ Attention (next) addresses this directly.**

Part 2

Attention Mechanism

"Don't compress everything. Look back at the whole input."

Seq2Seq with RNNs and Attention



Compute (scalar) **alignment scores**

$e_{t,i} = f_{att}(s_{t-1}, h_i)$ (f_{att} is an MLP)

Normalize alignment scores to get **attention weights**

$0 < a_{t,i} < 1$ $\sum_i a_{t,i} = 1$

Compute context vector as linear combination of hidden states

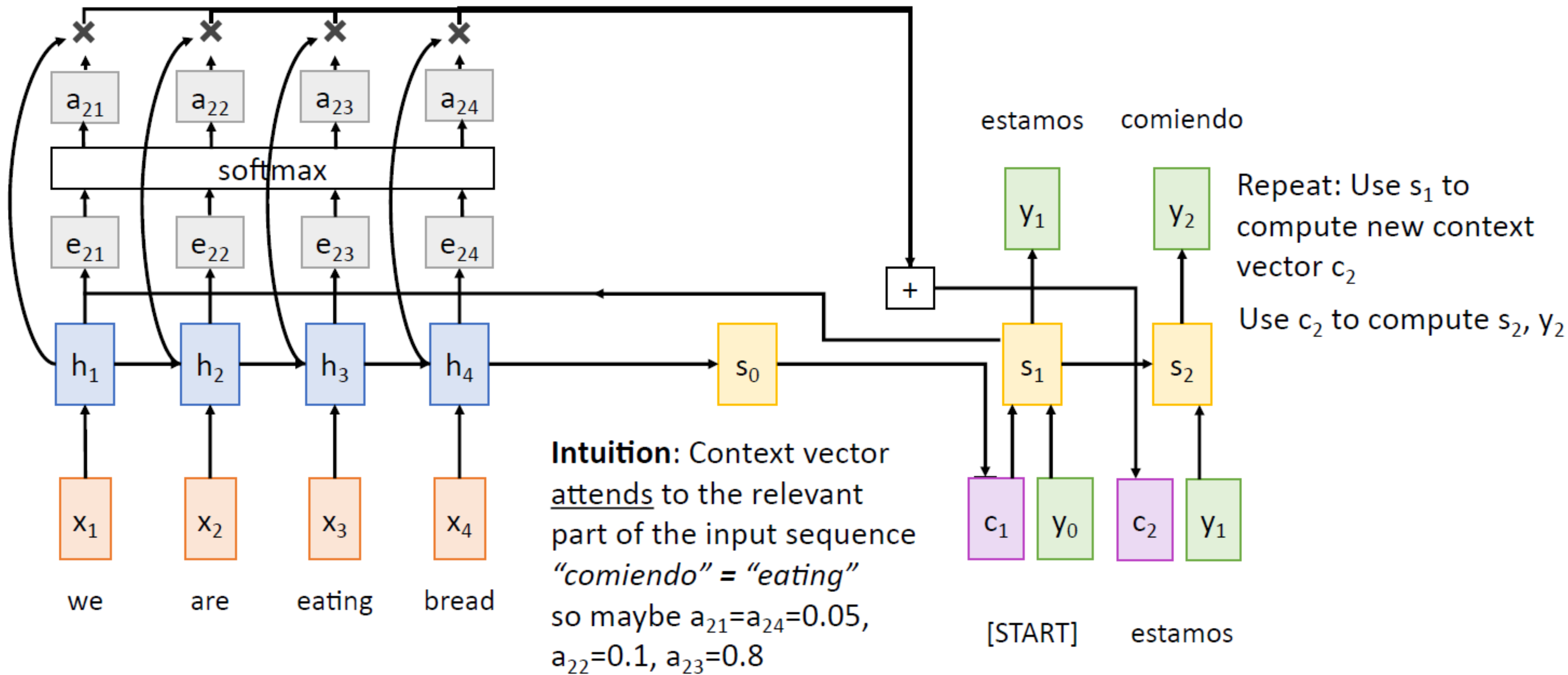
$c_t = \sum_i a_{t,i} h_i$

Use context vector in decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c_t)$

This is all differentiable! Do not supervise attention weights – backprop through everything

Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Seq2Seq with RNNs and Attention



Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Seq2Seq 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."

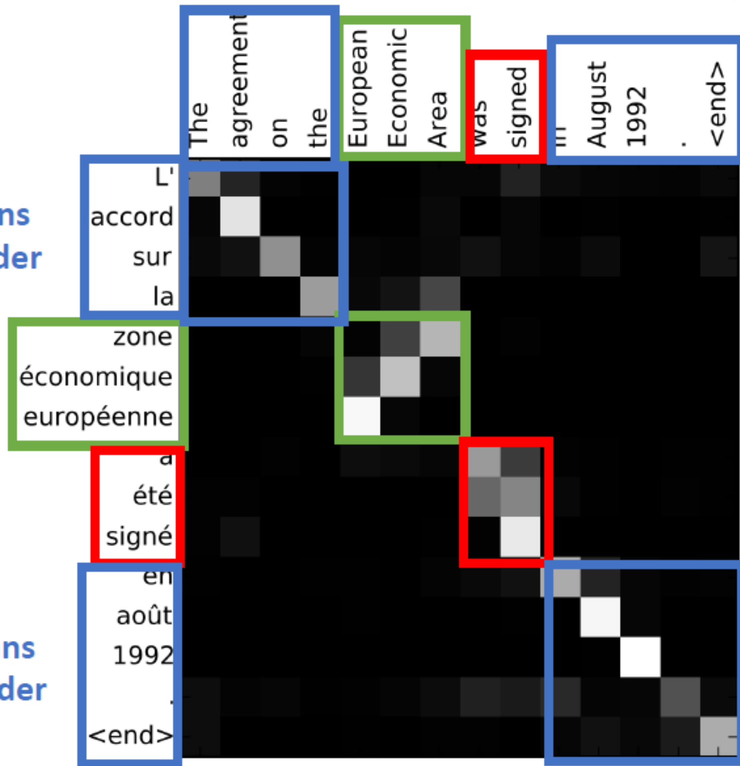
Diagonal attention means words correspond in order

Attention figures out different word orders

Verb conjugation

Diagonal attention means words correspond in order

Visualize attention weights $a_{t,i}$



Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Attention in Vision: Image Captioning

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

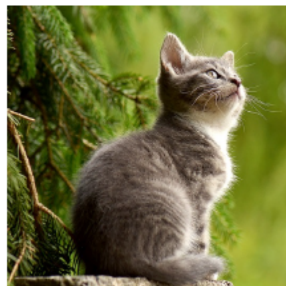
Alignment scores

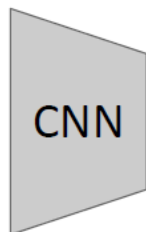
$e_{2,1,1}$	$e_{2,1,2}$	$e_{2,1,3}$
$e_{2,2,1}$	$e_{2,2,2}$	$e_{2,2,3}$
$e_{2,3,1}$	$e_{2,3,2}$	$e_{2,3,3}$

softmax

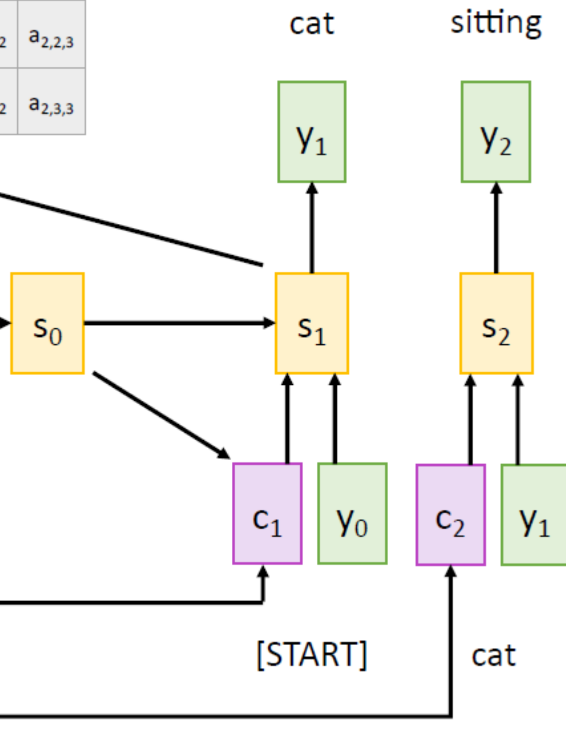
Attention weights

$a_{2,1,1}$	$a_{2,1,2}$	$a_{2,1,3}$
$a_{2,2,1}$	$a_{2,2,2}$	$a_{2,2,3}$
$a_{2,3,1}$	$a_{2,3,2}$	$a_{2,3,3}$





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



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

Xu et al, "Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Attention in Vision: 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

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

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

cat sitting outside [STOP]

y_1 y_2 y_3 y_4

s_0 s_1 s_2 s_3 s_4

c_1 y_0 c_2 y_1 c_3 y_2 c_4 y_3

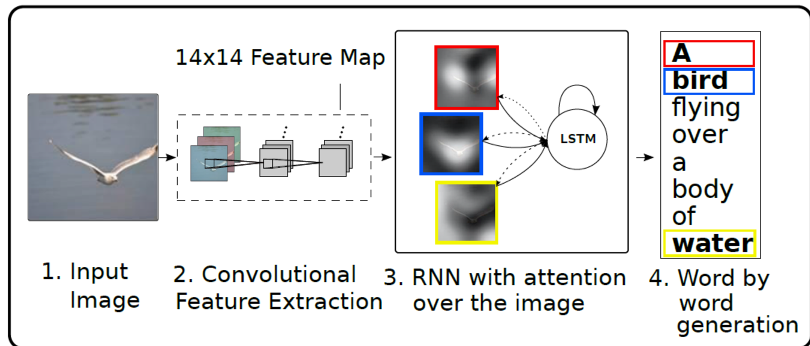
[START] cat sitting outside

Xu et al, “Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015

Attention in Vision: Image Captioning

The first time vision + sequence + attention come together. The ancestor of modern VLMs.

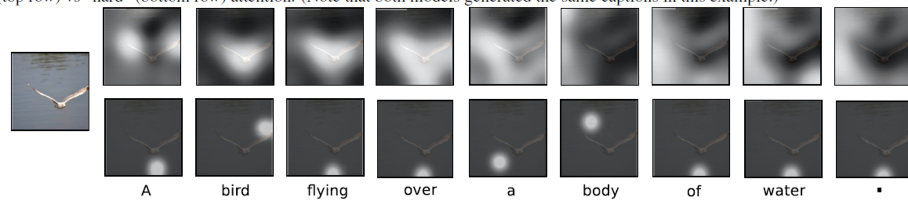
Show, Attend and Tell (Xu et al., 2015)



Key: At each word generation step, attention weights tell the decoder WHERE in the image to look.

Generating "bird" → attention on the bird region
Generating "water" → attention shifts to the water

Figure 2. Attention over time. As the model generates each word, its attention changes to reflect the relevant parts of the image. "soft" (top row) vs "hard" (bottom row) attention. (Note that both models generated the same captions in this example.)



From Captioning to VLMs: An Evolution

Attention: A General Framework

Abstracting from Bahdanau: attention as a query-based lookup.

$$Attention(query, keys, values) = \Sigma\ similarity(query, key_i) \times value_i$$

Query (Q)

"What am I looking for?"

In Bahdanau: decoder state s_{t-1}
In self-attention: each input itself

Key (K)

"What do I contain?"

In Bahdanau: encoder states h_i
In self-attention: each input itself

Value (V)

"What do I provide?"

In Bahdanau: same as keys (h_i)
In self-attention: each input itself

This Q/K/V framework is the universal language of all attention variants.

Part 3

Self-Attention & Multi-Head Attention

"Every token attends to every other token. No recurrence needed."

Attention

<u>**Inputs**</u>:

Query vector: $\mathbf{q}$ (Shape: D_Q)

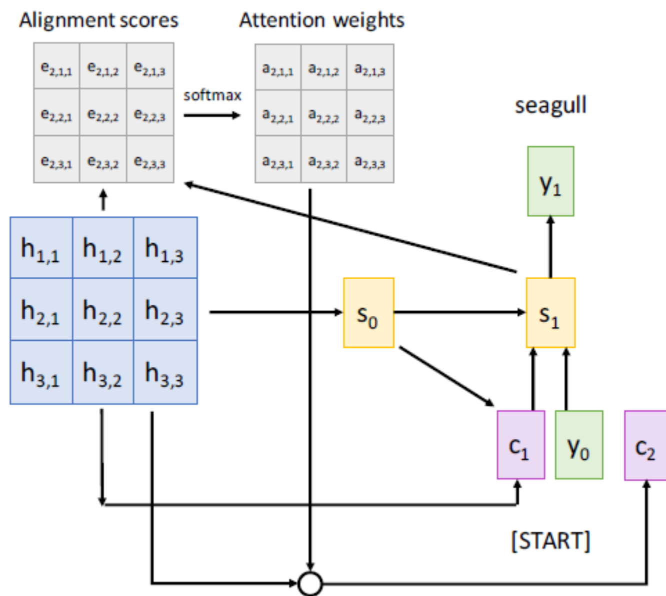
Input vectors: $\mathbf{X}$ (Shape: N_X x D_X)

Similarity function: f_{att}

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

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

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



<u>**Computation**</u>:

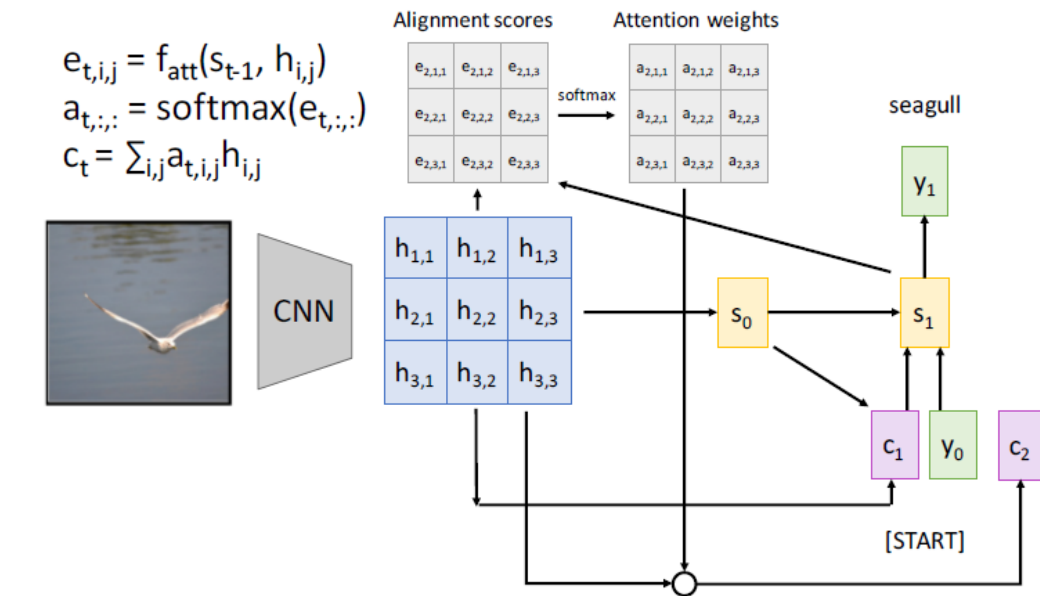
Similarities: e (Shape: N_X) $e_i = f_{att}(\mathbf{q}, \mathbf{X}_i)$

Attention weights: $a = \text{softmax}(e)$ (Shape: N_X)

Output vector: $y = \sum_i a_i \mathbf{X}_i$ (Shape: D_X)

Attention

Inputs:
Query vector: **q** (Shape: D_Q)
Input vectors: **X** (Shape: N_X x D_Q)
Similarity function: dot product



Computation:
Similarities: e (Shape: N_X) $e_i = \mathbf{q} \cdot \mathbf{X}_i$
Attention weights: a = softmax(e) (Shape: N_X)
Output vector: $y = \sum_i a_i \mathbf{X}_i$ (Shape: D_X)

Changes:
- Use dot product for similarity

Attention

Inputs:

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

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

Similarity function: scaled dot product

Large similarities will cause softmax to saturate and give vanishing gradients

Recall a · b = |a||b| cos(angle)

Suppose that a and b are constant vectors of dimension D

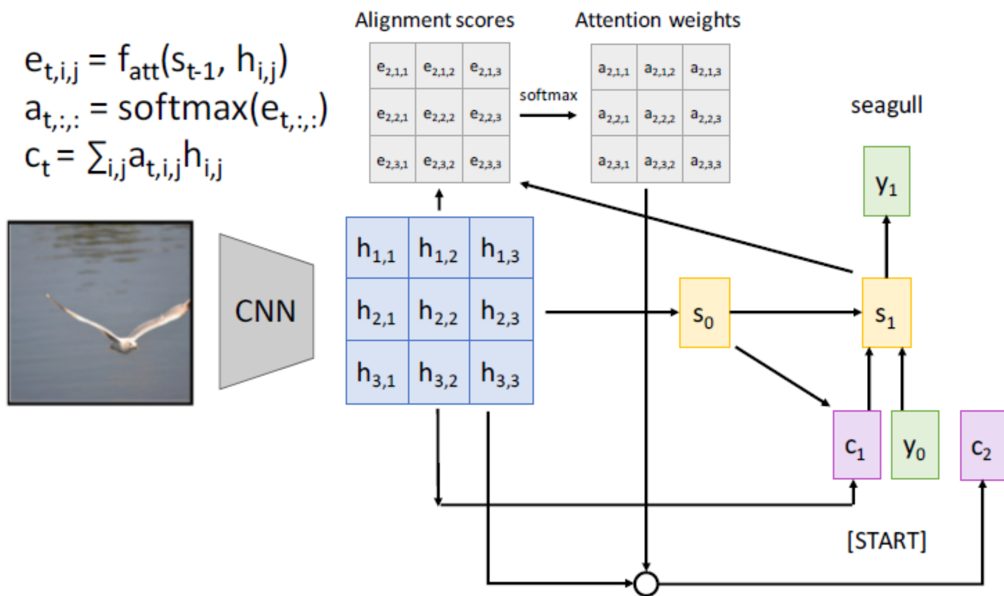
Then $|a| = (\sum_i a^2)^{1/2} = a\sqrt{D}$

Computation:

Similarities: e (Shape: N_X) $e_i = \mathbf{q} \cdot \mathbf{X}_i / \sqrt{D_Q}$

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

Output vector: $y = \sum_i a_i \mathbf{X}_i$ (Shape: D_X)



Changes:

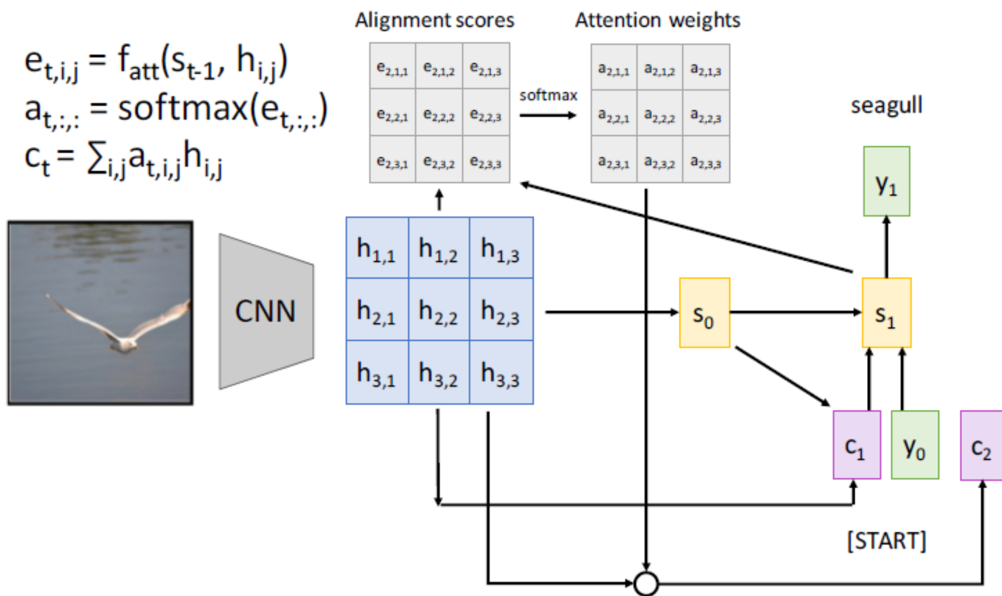
- Use **scaled** dot product for similarity

Attention

<u>Inputs</u>:
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)

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

Changes:
- Use scaled dot product for similarity
- Multiple query vectors
- Separate key and value



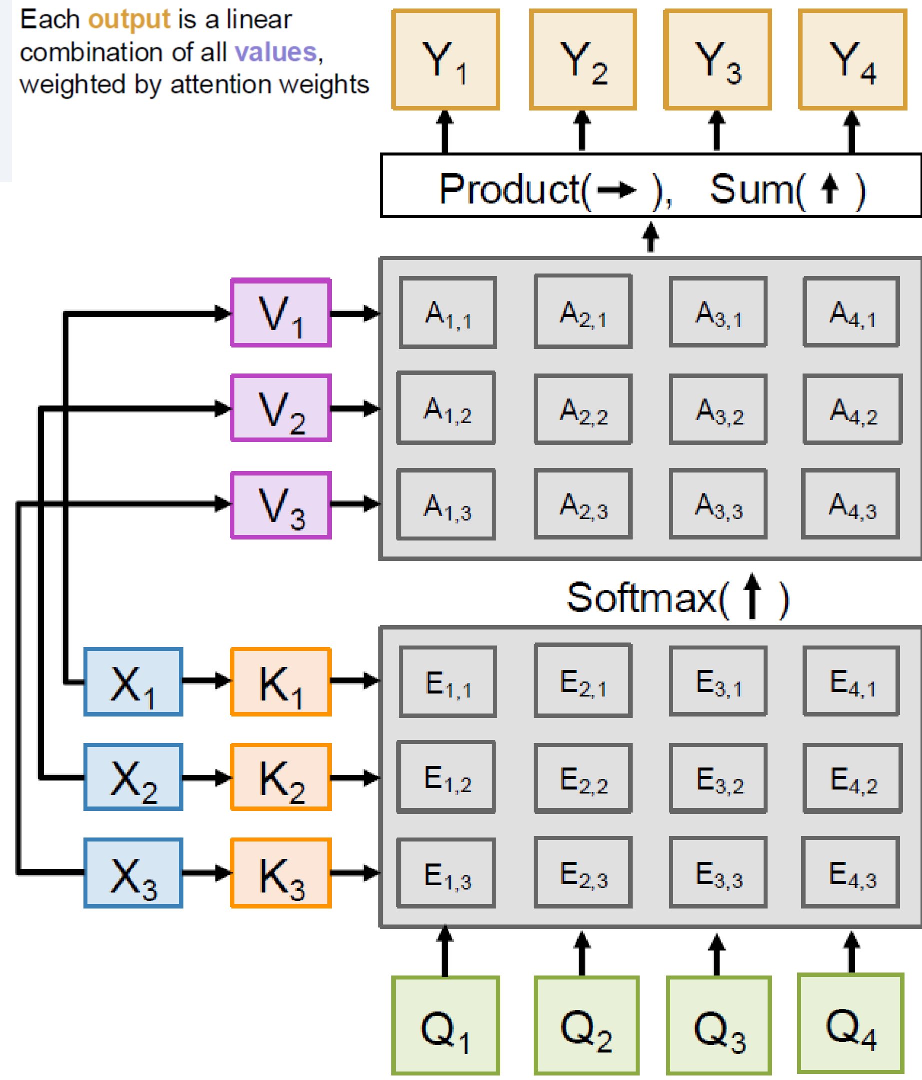
Attention

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: $E = QK^T / \sqrt{D_Q}$ (Shape: N_Q x N_X) $E_{i,j} = (Q_i \cdot K_j) / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: $Y = AV$ (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} V_j$



Self-Attention

Inputs:
Input vectors: X [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$

Consider permuting **inputs**:

Queries, **keys**, and **values** will be the same but permuted

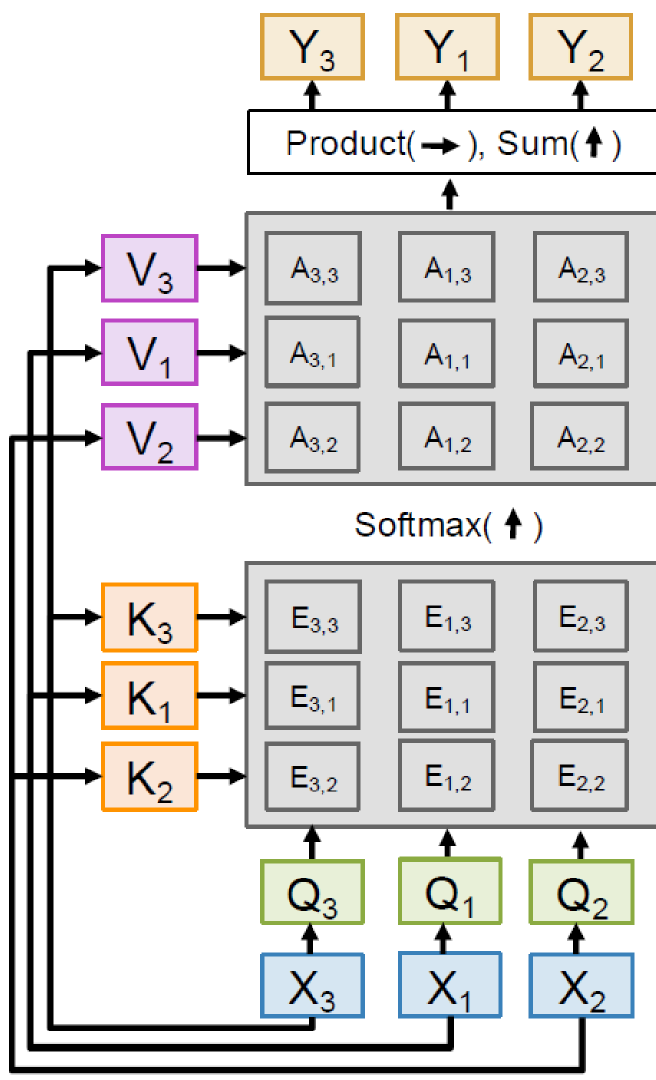
Similarities are the same but permuted

Attention weights are the same but permuted

Outputs are the same but permuted

Self-Attention is permutation equivariant:
$F(\sigma(X)) = \sigma(F(X))$

This means that Self-Attention works on **sets of vectors**



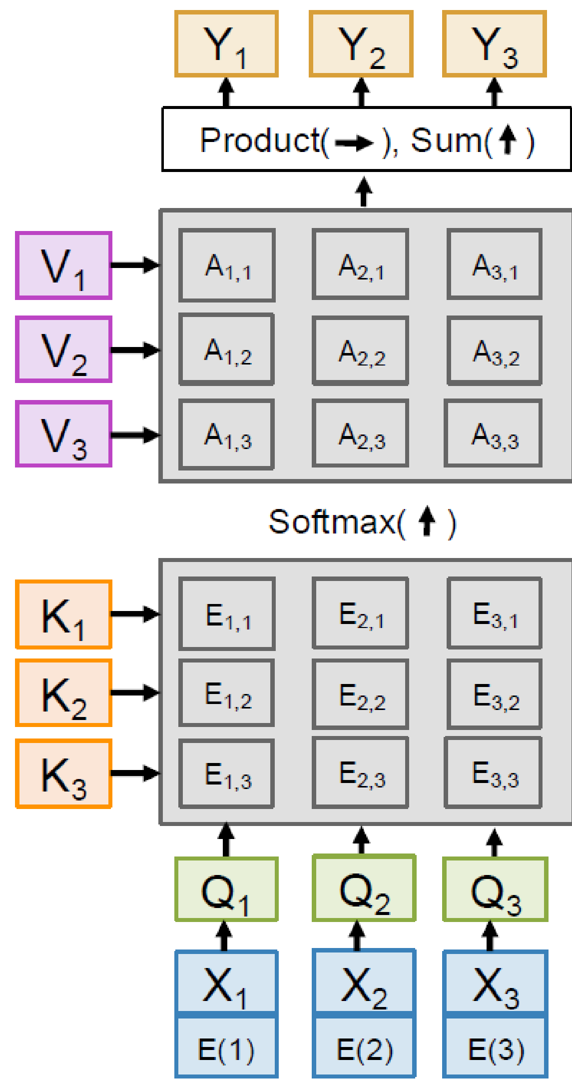
Self-Attention

Inputs:
Input vectors: **X** [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$

Problem: Self-Attention does not know the order of the sequence

Solution: Add positional encoding to each input; this is a vector that is a fixed function of the index

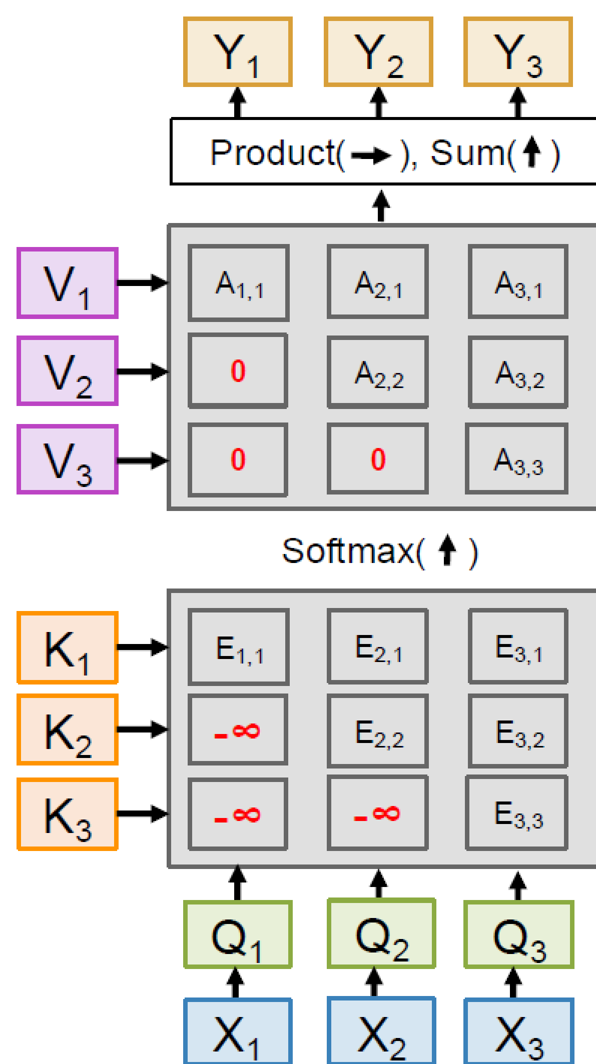


Masked Self-Attention

Inputs:
Input vectors: X [$N \times D_{in}$]
Key matrix: W_K [$D_{in} \times D_{out}$]
Value matrix: W_V [$D_{in} \times D_{out}$]
Query matrix: W_Q [$D_{in} \times D_{out}$]

Override similarities with -inf; this controls which inputs each vector is allowed to look at.

Computation:
Queries: $Q = XW_Q$ [$N \times D_{out}$]
Keys: $K = XW_K$ [$N \times D_{out}$]
Values: $V = XW_V$ [$N \times D_{out}$]
Similarities: $E = QK^T / \sqrt{D_Q}$ [$N \times N$]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N \times N$]
Output vector: $Y = AV$ [$N \times D_{out}$]
$Y_i = \sum_j A_{ij} V_j$



Multi-Head Attention

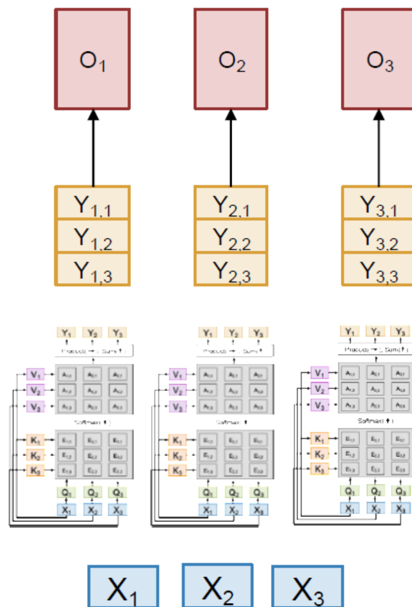
Run attention multiple times in parallel, each with different learned projections.

Inputs:
Input vectors: X [N x D]
Key matrix: W_K [D x HD_H]
Value matrix: W_V [D x HD_H]
Query matrix: W_Q [D x HD_H]
Output matrix: W_O [HD_H x D]

Computation:
Queries: $Q = XW_Q$ [H x N x D_H]
Keys: $K = XW_K$ [H x N x D_H]
Values: $V = XW_V$ [H x N x D_H]
Similarities: $E = QK^T / \sqrt{D_Q}$ [H x N x N]
Attention weights: A = softmax(E, dim=2) [H x N x N]
Head outputs: $Y = AV$ [H x N x D_H] => [N x HD_H]
Outputs: $O = YW_O$ [N x D]

Each of the H parallel layers use a qkv dim of D_H = "head dim"

Usually $D_H = D / H$, so inputs and outputs have the same dimension



Why Multiple Heads?

Different heads learn to attend to different types of relationships.
One head: syntactic. Another: semantic.
In vision: one local, another global.

Practical Details

Typical: h=8 or h=12 heads.

$$d_{model} = 512 \rightarrow d_k = d_v = 512/8 = 64.$$

Same total computation as single-head with full d_{model}. Just reorganized.

Computation

Inputs:
Input vectors: X [N x D]
Key matrix: W_K [D x HD_H]
Value matrix: W_V [D x HD_H]
Query matrix: W_Q [D x HD_H]
Output matrix: W_O [HD_H x D]

Computation:
Queries: $Q = XW_Q$ [H x N x D_H]
Keys: $K = XW_K$ [H x N x D_H]
Values: $V = XW_V$ [H x N x D_H]
Similarities: $E = QK^T / \sqrt{D_Q}$ [H x N x N]
Attention weights: A = softmax(E, dim=2) [H x N x N]
Head outputs: $Y = AV$ [H x N x D_H] => [N x HD_H]
Outputs: $O = YW_O$ [N x D]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x D_H x N] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]
4. Output Projection
   [N x HD_H] [HD_H x D] => [N x D]

Q: How much compute does this take as the number of vectors N increases?
A: $O(N^2)$

Computation

Inputs:
Input vectors: **X** [N x D]
Key matrix: W_K [D x HD_H]
Value matrix: W_V [D x HD_H]
Query matrix: W_Q [D x HD_H]
Output matrix: W_O [HD_H x D]

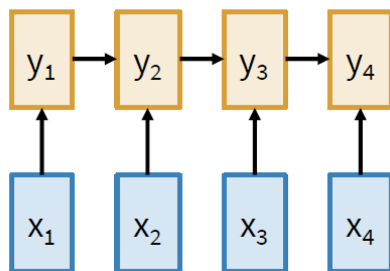
Computation:
Queries: $Q = XW_Q$ [H x N x D_H]
Keys: $K = XW_K$ [H x N x D_H]
Values: $V = XW_V$ [H x N x D_H]
Similarities: $E = QK^T / \sqrt{D_Q}$ [H x N x N]
Attention weights: A = softmax(E, dim=2) [H x N x N]
Head outputs: Y = AV [H x N x D_H] => [N x HD_H]
Outputs: $O = YW_O$ [N x D]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x D_H x N] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]
4. Output Projection
   [N x HD_H] [HD_H x D] => [N x D]

Q: How much memory does this take as the number of vectors N increases?
A: $O(N^2)$

RNN vs CNN vs Self-Attention

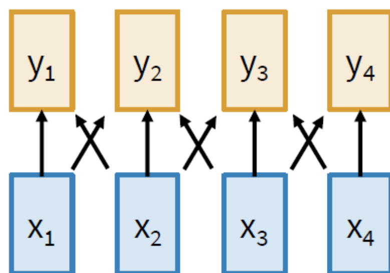
Recurrent Neural Network



Works on 1D ordered sequences

(+) Theoretically good at long sequences: O(N) compute and memory for a sequence of length N
(-) Not parallelizable. Need to compute hidden states sequentially

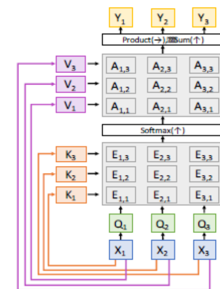
Convolution



Works on N-dimensional grids

(-) Bad for long sequences: need to stack many layers to build up large receptive fields
(+) Parallelizable, outputs can be computed in parallel

Self-Attention

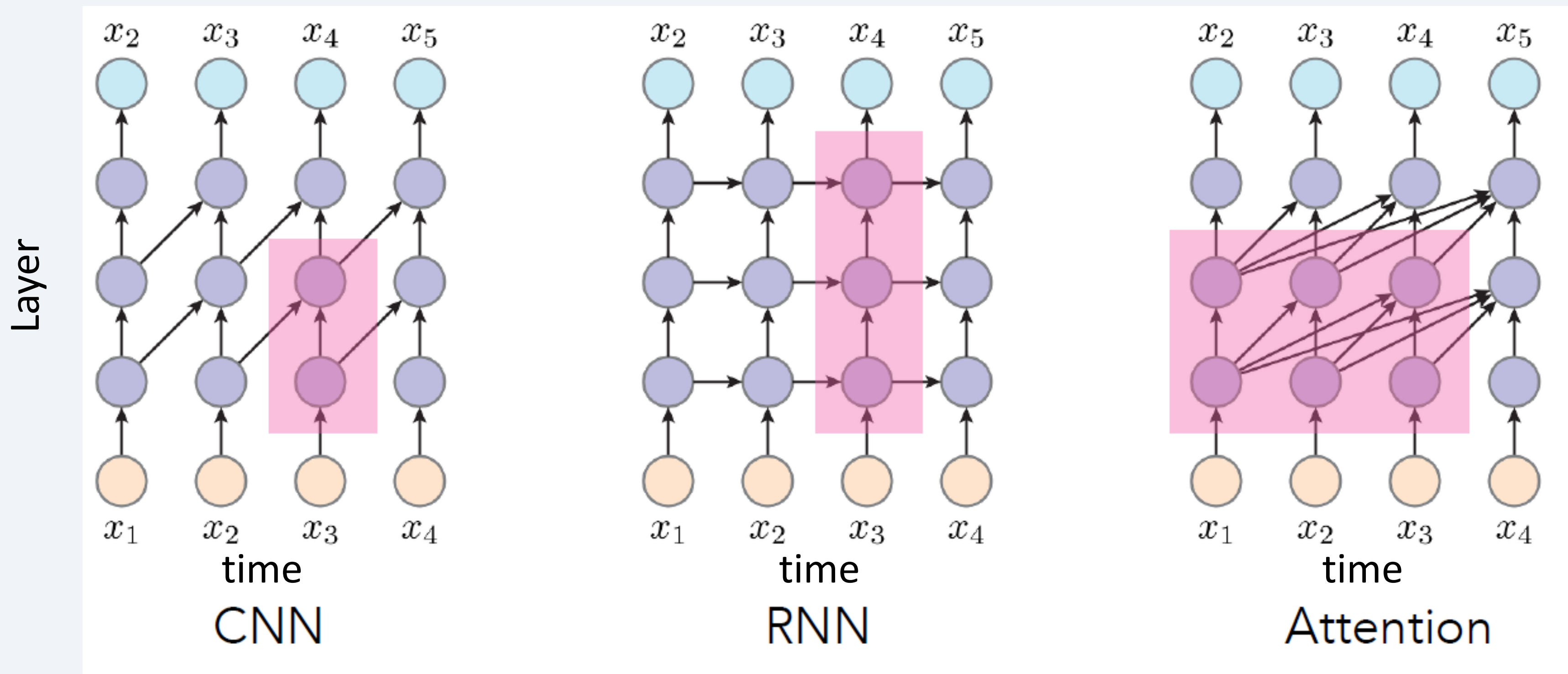


Works on sets of vectors

(+) Great for long sequences; each output depends directly on all inputs
(+) Highly parallel, it's just 4 matmuls
(-) Expensive: $O(N^2)$ compute, O(N) memory for sequence of length N

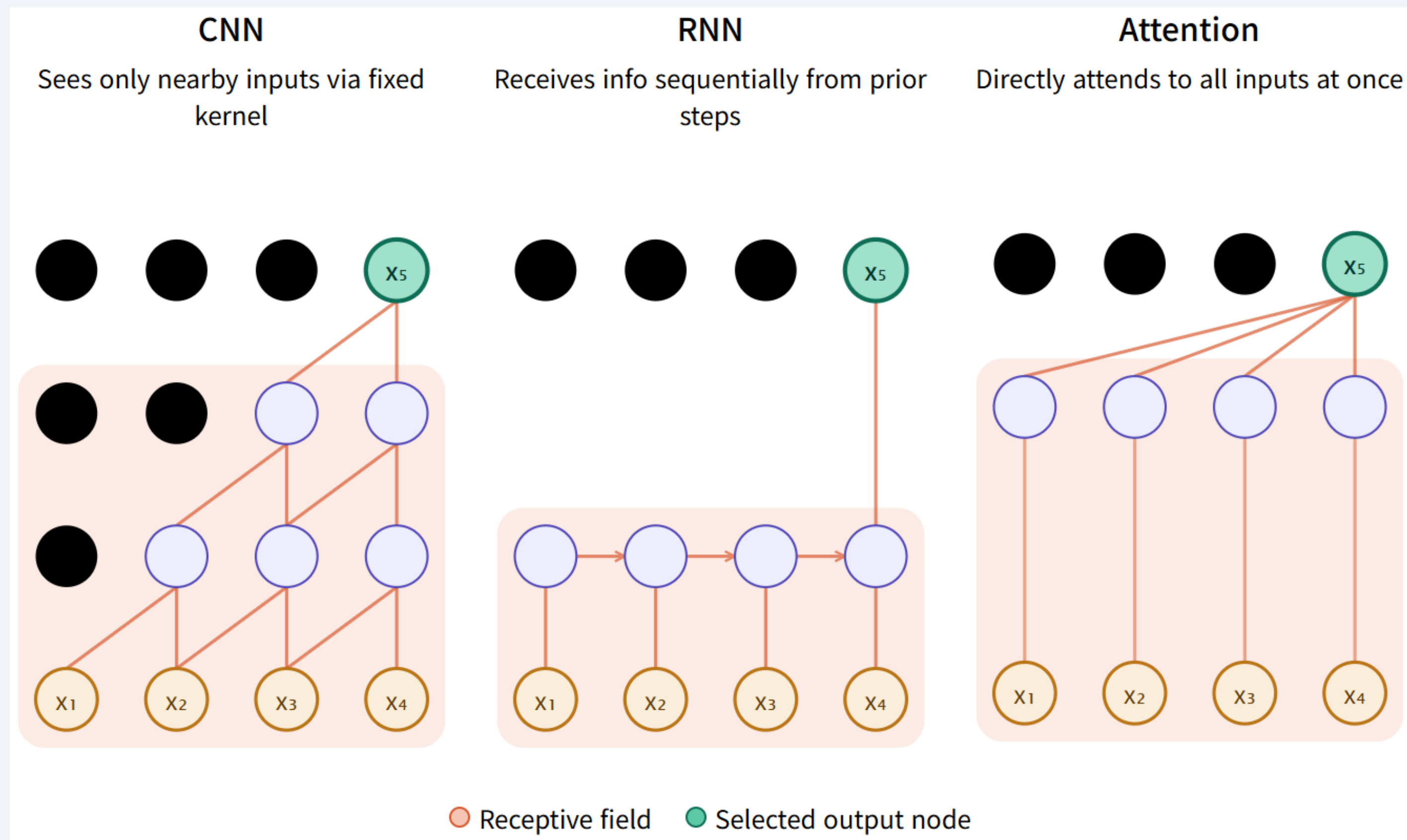
	RNN	CNN	Self-Attention
Parallelizable?	No	Yes	Yes
Long-range deps	Hard (vanishing)	Needs many layers	Direct (1 layer)
Compute per layer	O(N)	O(N)	$O(N^2)$
Dominant use	Legacy	Images (CNN era)	Everything (2020→)

What act as the "memory"? (Suppl.)



Consider Conv1D

Receptive Field Point of View (Suppl.)



Beyond Vanilla Attention

Self-Attention is $O(N^2)$. Various approaches to reduce cost or extend capability.

Cross-Attention	Flash Attention	Grouped Query (GQA)	Linear Attention	Windowed / Sparse	SSM / Mamba
Q from one modality, K/V from another.	Exact attention, IO-aware for GPU memory hierarchy.	Share K/V heads across Q heads. Faster inference.	Kernel trick replaces softmax. $O(N^2) \to O(N)$. Less expressive.	Local windows or sparse patterns. Swin Transformer.	State space models. Recurrence + parallel training. $O(N)$.
→ Wk 11–13	→ Wk 5, 9	→ Wk 12–13	→ Wk 14	→ Wk 5, 14	→ Wk 14

Today We mastered vanilla Self-Attention and Multi-Head Attention — the foundation everything else builds on.
These variants will appear naturally in later weeks when we hit the problems they solve.

Part 4

The Transformer

"Attention Is All You Need" — Vaswani et al., 2017

The Transformer with three components

1. Tokenization
2. Attention (+FFN, Norm)
3. Positional Encoding

Transformer Block:

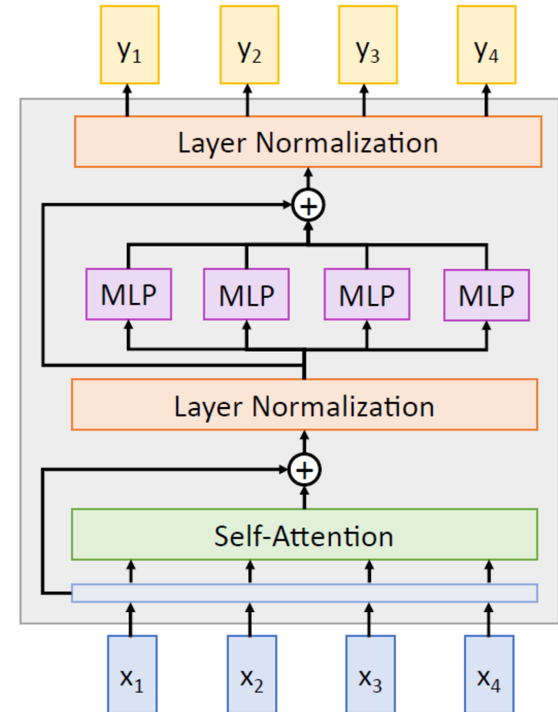
Input: Set of vectors x

Output: Set of vectors y

Self-attention is the only interaction between vectors!

Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable



Component 1: Tokenization

Transformer operates on sequences of vectors. How do we convert raw input into tokens?

Text (NLP)

Word / Subword

"attention" → ["at", "ten", "tion"]

Each subword mapped to a learned embedding vector.

Vocabulary: 30K–100K tokens. BPE, WordPiece, etc.

BERT, GPT

Image (Vision)

Patch

224×224 image → 14×14 grid of 16×16 patches.

Each patch flattened and linearly projected to D dims.

196 tokens per image.

ViT (Week 5!)

Audio / Video

Frame / Segment

Audio: spectrogram slices.
Video: per-frame patches.

Token count grows fast → efficiency matters (Wk 14).

Whisper, ViViT

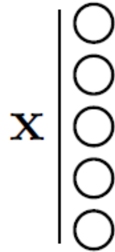
The Transformer itself doesn't care what the tokens are. Only the tokenization changes between domains.

A New Data Type: Tokens

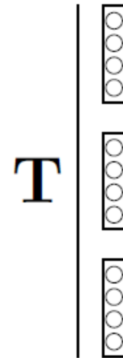
- A **token** is just a vector of neurons. (note: GNNs also operate over tokens, but over there we called them “node attributes” or node “feature descriptors”)

- But the connotation is that a token is an encapsulated bundle of information; with transformers we will operate over tokens rather than over neurons.

array of **neurons**



array of **tokens**

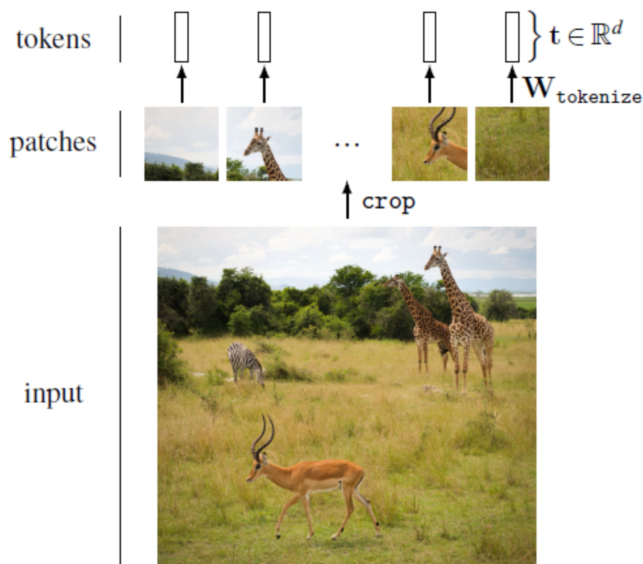


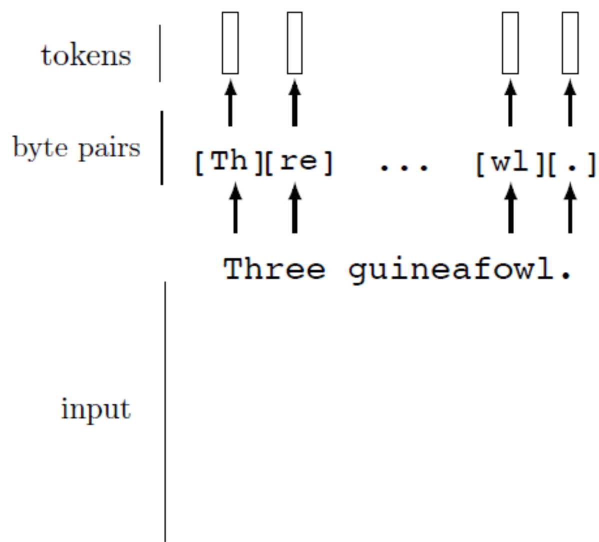
Note: sometimes the word “token” is instead used to refer to the atomic units of the data sequence we will model. In this usage tokens are the representation of the data only at the input and output layers. We use a more general definition where tokens are the representation of the data at *any* layer.

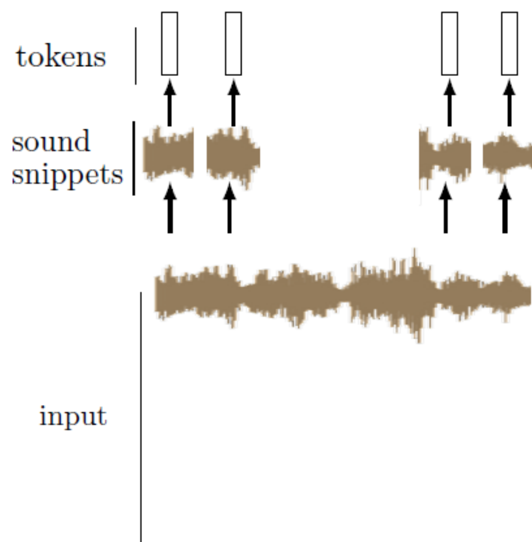
Tokenizing the input data

You can tokenize anything.

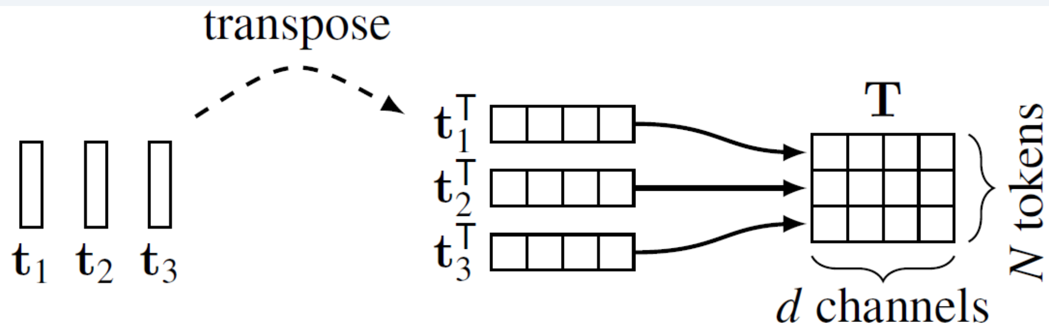
General strategy: chop the input up into chunks, project each chunk to a vector.



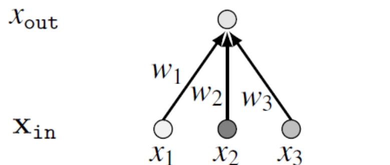




Linear combination of tokens



Linear combination of neurons

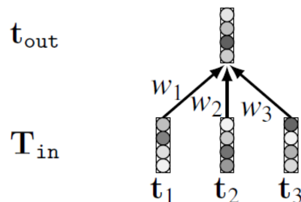


$$x_{\text{out}} = w_1 x_1 + w_2 x_2 + w_3 x_3$$

$$x_{\text{out}}[i] = \sum_{j=1}^{N} w_{ij} x_{\text{in}}[j]$$

$$\mathbf{x}_{\text{out}} = \mathbf{W}\mathbf{x}_{\text{in}}$$

Linear combination of tokens



$$\mathbf{t}_{\text{out}} = w_1 \mathbf{t}_1 + w_2 \mathbf{t}_2 + w_3 \mathbf{t}_3$$

$$\mathbf{T}_{\text{out}}[i,:] = \sum_{j=1}^{N} w_{ij} \mathbf{T}_{\text{in}}[j,:]$$

$$\mathbf{T}_{\text{out}} = \mathbf{W}\mathbf{T}_{\text{in}}$$

$$\mathbf{x}_{\text{out}} = \begin{bmatrix} \texttt{relu}(x_{\text{in}}[0]) \\ \vdots \\ \texttt{relu}(x_{\text{in}}[N-1]) \end{bmatrix}$$

$$\mathbf{T}_{\text{out}} = \begin{bmatrix} F_\theta(\mathbf{T}_{\text{in}}[0,:]) \\ \vdots \\ F_\theta(\mathbf{T}_{\text{in}}[N-1,:]) \end{bmatrix}$$

F is typically an MLP

Equivalent to a CNN with 1x1 kernels run over token sequence

Component 2: Attention/FFN as Mixer

Transformer block = Token Mixer (attention) + Channel Mixer (FFN) + Residual + LayerNorm.

Token Mixer: Attention

Multi-Head Self-Attention
mixes information BETWEEN tokens.

"Each token looks at all other tokens
and updates itself accordingly."

This is the sequence processing part.

In CNN terms: like a conv layer,
but with global receptive field from layer 1.

Channel Mixer: FFN(FC)

Feed-Forward Network (2-layer MLP)
mixes information WITHIN each token.

$\text{FFN}(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2$
Expand 4× then shrink back. (*Note: original transformer uses ReLU)

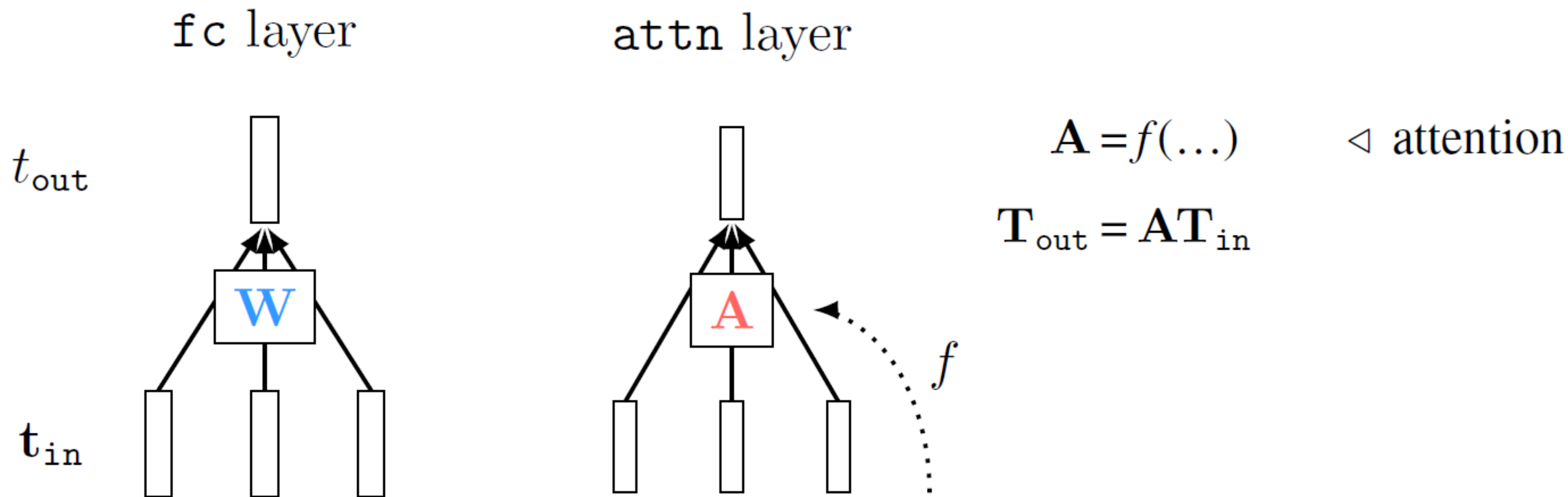
Applied to each token independently.
Same weights for all positions.

In CNN terms: like a 1×1 convolution.

One Transformer Block

```
x → [Multi-Head Attention] → Add & LayerNorm → [FFN] → Add & LayerNorm → output
     token mixer                residual + LN     channel mixer    residual + LN
```

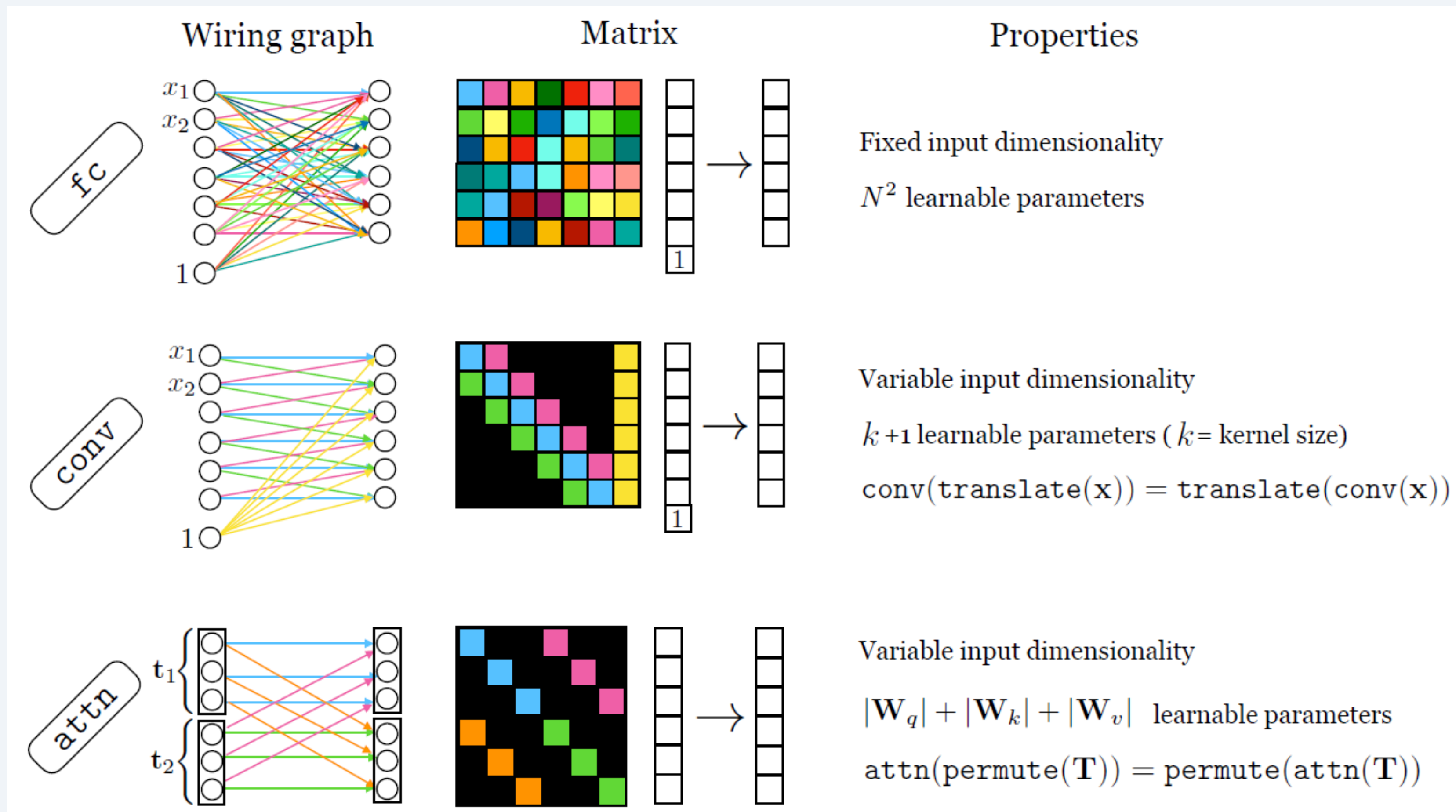
Component 2: Attention/FFN as Mixer



W is free *parameters*.

A is a function of some input *data*. The data tells us which tokens to attend to (assign high weight in weighted sum)

A family of linear layers (Suppl.)



Component 3: Positional Embedding

Self-attention is permutation-invariant. We must inject position information explicitly.

The Problem

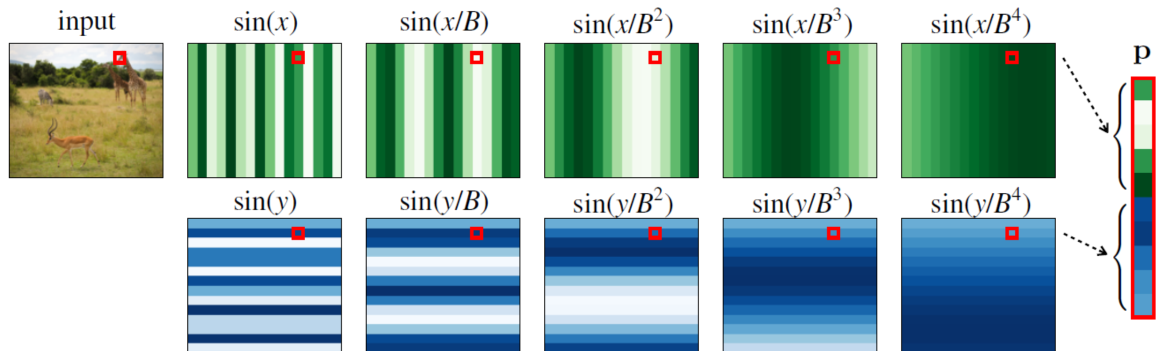
"The cat sat on the mat" and
"The mat sat on the cat" produce
identical attention outputs.

Self-attention treats input as a SET,
not a SEQUENCE. Position is lost.

Two Approaches

Sinusoidal (original Transformer):
Fixed, no learning. Generalizes to unseen lengths.

Learned (ViT, BERT):
Learn a D-dim embedding per position.
More flexible, standard in modern models.



Fourier Positional Encoding

1D formula (per axis):

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d})$$

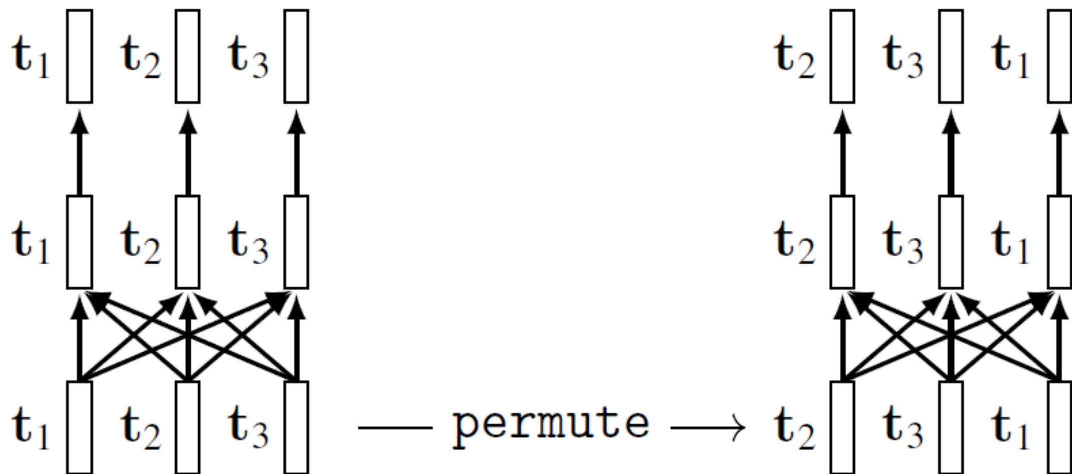
pos=position, i=dim index, d=d_{model}

2D image (figure ↓):

Apply 1D PE twice, then concat:
Row 1: pos=**x**(col) → sin(x), sin(x/B)...
Row 2: pos=**y**(row) → sin(y), sin(y/B)...
Each col = different *i* (freq); cos pairs omitted

Inductive bias trade-off CNN knows position through structure (Wk3). Transformer must be told explicitly.
Less bias = more flexible, but needs more data.

Permutation equivariance



$$F_\theta(\mathtt{permute}(\mathbf{T}_{\mathtt{in}})) = \mathtt{permute}(F_\theta(\mathbf{T}_{\mathtt{in}}))$$

$$\mathtt{attn}(\mathtt{permute}(\mathbf{T}_{\mathtt{in}})) = \mathtt{permute}(\mathtt{attn}(\mathbf{T}_{\mathtt{in}}))$$

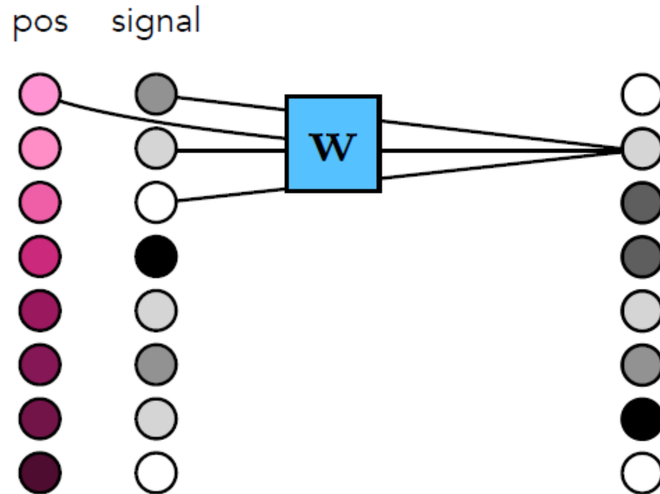
Set2Set

$$\mathtt{transformer}(\mathtt{permute}(\mathbf{T}_{\mathtt{in}})) = \mathtt{permute}(\mathtt{transformer}(\mathbf{T}_{\mathtt{in}}))$$

What if you don't want to be shift invariant?

1. Use an architecture that is not shift invariant (e.g., MLP)

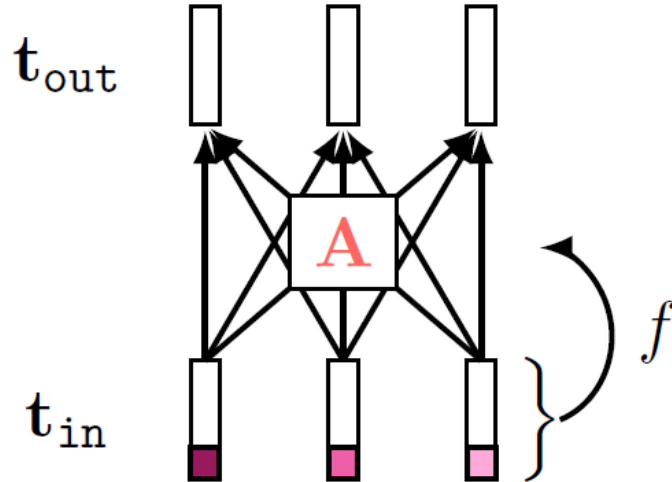
2. Add location information to the *input* to the convolutional filters — this is called **positional encoding**



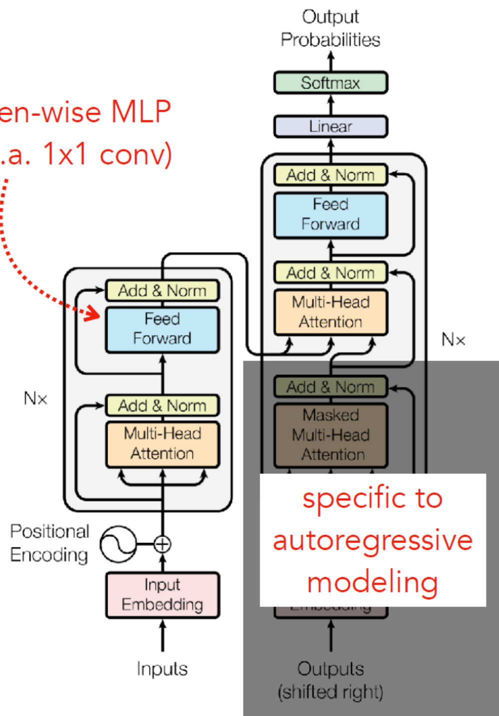
What if you don't want to be permutation invariant?

1. Use an architecture that is not permutation invariant (e.g., MLP)

2. Add location information to the token code vectors — this is called **positional encoding**



Transformer: Full Architecture (Recap)



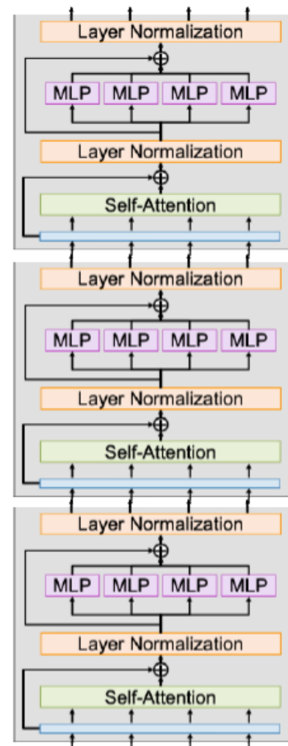
A **Transformer** is just a stack of identical Transformer blocks!

They have not changed much since 2017… but have gotten a lot bigger

Original: [Vaswani et al, 2017]
12 blocks, D=1024, H=16, N=512
213M params

GPT-2: [Radford et al, 2019]
48 blocks, D=1600, H=25, N=1024
1.5B params

GPT-3: [Brown et al, 2020]
96 blocks, D=12288, H=96, N=2048
175B params



What the Transformer Changed

Encoder Only

BERT (2018)

Classification, embeddings, understanding

Decoder Only

GPT (2018)

Text generation, language models, reasoning

Encoder-Decoder

Original Transformer (2017) T5 (2019)

Translation, summarization, Seq2Seq tasks

Vision Encoder

ViT (2020)

Image classification, feature extraction, foundation models

One Architecture, Every Domain

The same Transformer powers ChatGPT (decoder), BERT (encoder), and next week's ViT (encoder for images).
The key question: how do you turn an image into a sequence of tokens?

Summary & Resources

What We Covered

Sequence tasks in vision (captioning, VQA)
RNN basics + 3 key limitations
Bahdanau attention → Image Captioning
Q/K/V framework → Self-Attention
Scaled dot-product + Multi-Head
RNN vs CNN vs Self-Attention comparison
Attention variant roadmap
Transformer: tokenizing, attention, positional encoding

Further Reading & Self-Study

Michigan EECS 498 (YouTube):
L15-16 RNN | L17 Attention

3Blue1Brown:
"Attention in Transformers" (~25 min)

Key Papers:
Xu et al. (2015) — Show, Attend & Tell
Bahdanau et al. (2015) — Attention
Vaswani et al. (2017) — Transformer

CS231n 2025 L8 slides

MIT 6.7960 L4, L5

Next Week Week 5: Transformer in Vision — ViT, CNN vs ViT, Swin, and vision encoders in the wild

Acknowledgments

Some slides and figures in this course are adapted from or inspired by the following open resources.

Stanford CS231n: Deep Learning for Computer Vision (Spring 2025)

Fei-Fei Li, Ehsan Adeli, et al. | cs231n.stanford.edu

UMich EECS 498/598: Deep Learning for Computer Vision (Winter 2022)

Justin Johnson | web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/

MIT 6.8300: Advances in Computer Vision (Spring 2025)

Vincent Sitzmann | scenerepresentations.org/courses/2025/spring/advances-in-cv/

MIT 6.7960: Deep Learning (Fall 2024)

Phillip Isola, Sara Beery, Jeremy Bernstein | ocw.mit.edu/courses/6-7960-deep-learning-fall-2024/

CMU 16-824: Visual Learning and Recognition (Fall 2025)

Jun-Yan Zhu | visual-learning.cs.cmu.edu