COMP541
DEEP LEARNING

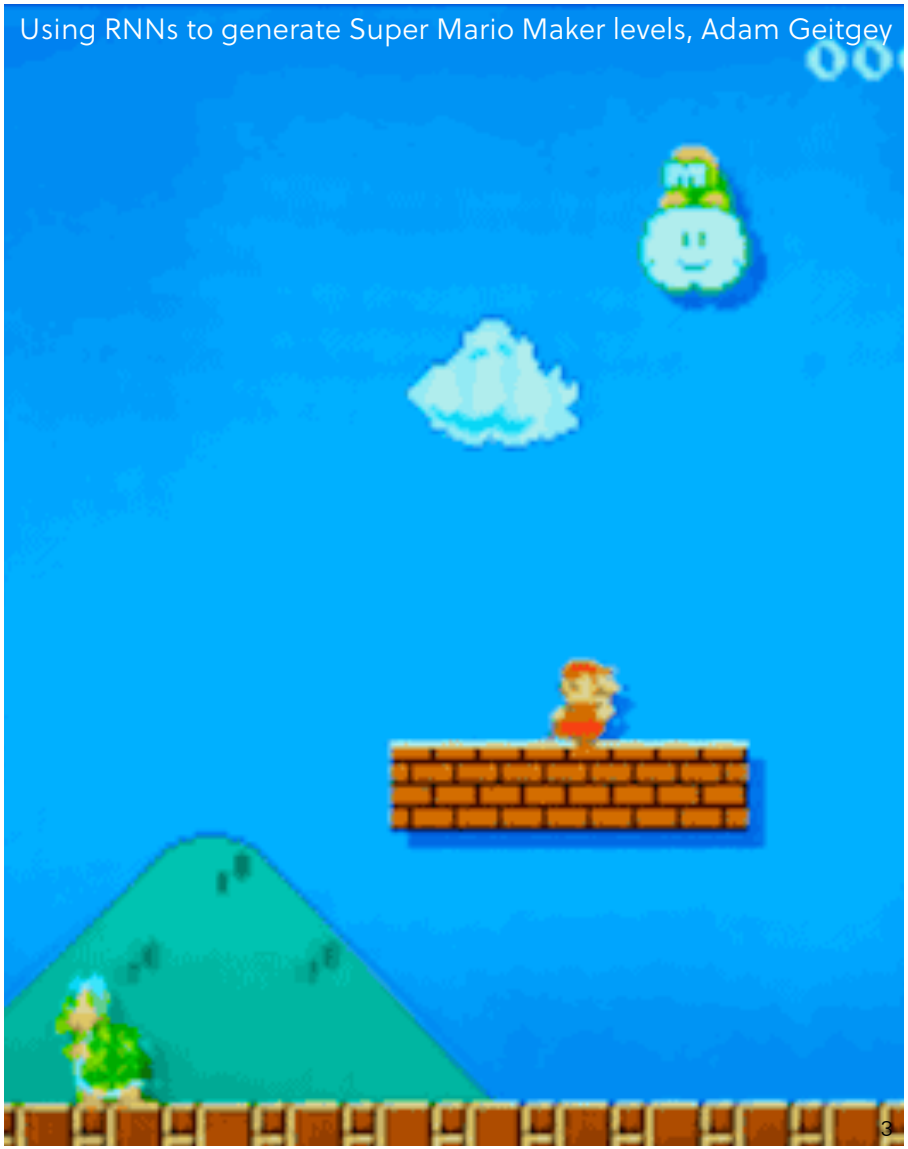
Lecture #08 – Attention and Transformers

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2025

Previously on COMP541

- sequence modeling
- recurrent neural networks (RNNs)
- the vanilla RNN unit
- how to train RNNs
- the long short-term memory (LSTM) unit and its variants
- gated recurrent unit (GRU)



Lecture overview

- what is attention?
- attention pre-transformers
- self-attention and transformer networks
- vision transformers
- pretraining during transformers

Disclaimer: Much of the material and slides for this lecture were borrowed from
— Kyunghyun Cho's slides on neural sequence modeling
— Wenhu Chen's UWaterloo CS886 class
— Justin Johnson's EECS 498/598 class
— Philip Isola and Stefanie Jegelka's MIT 6.S898 Deep Learning class

What is Attention?

- The notion of **exploiting context** is not new
  - **CNN –** context from **spatial locality** (useful for images)
  - **RNN –** context from **temporal locality** (useful for sequences/time-series data)
  - Embedding priors into models forces them to pay "attention" to relevant features for a given problem

- What we now call "attention" in DL
  - The idea of paying "attention" to the most relevant or important parts of the input at a given step
  - Very useful in **sequence-to-sequence** modelling
  - Ideally, we'd like to **learn** this!

What is a Learned Attention Mechanism?

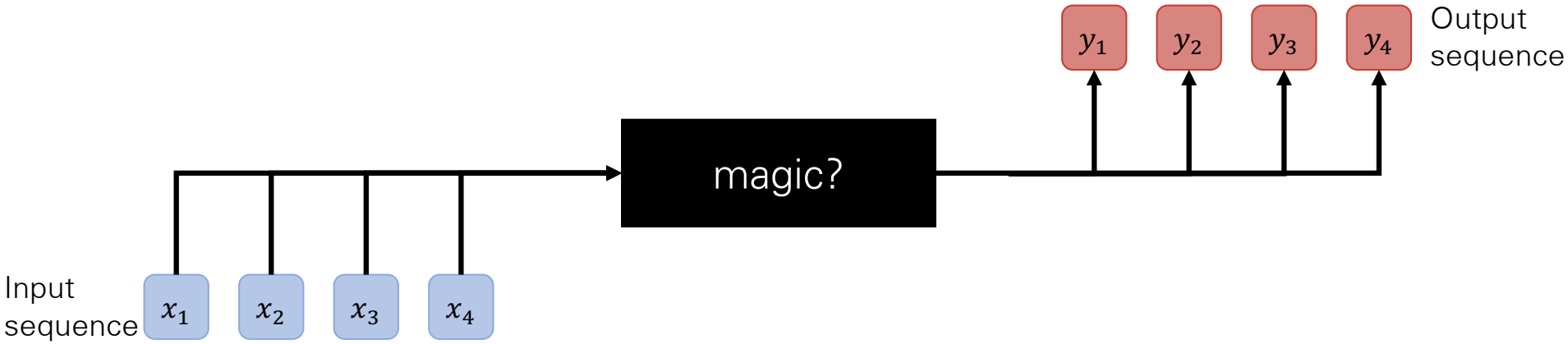
- An attention mechanism typically refers to **function** that allows a model to attend to different content

- There are many forms of attention mechanisms
  - Additive
  - Dot-product

- We have names to distinguish attention based on what is attended to
  - Self-attention (intra attention)
  - Cross-attention (encoder-decoder attention/inter attention)

Sequence to Sequence

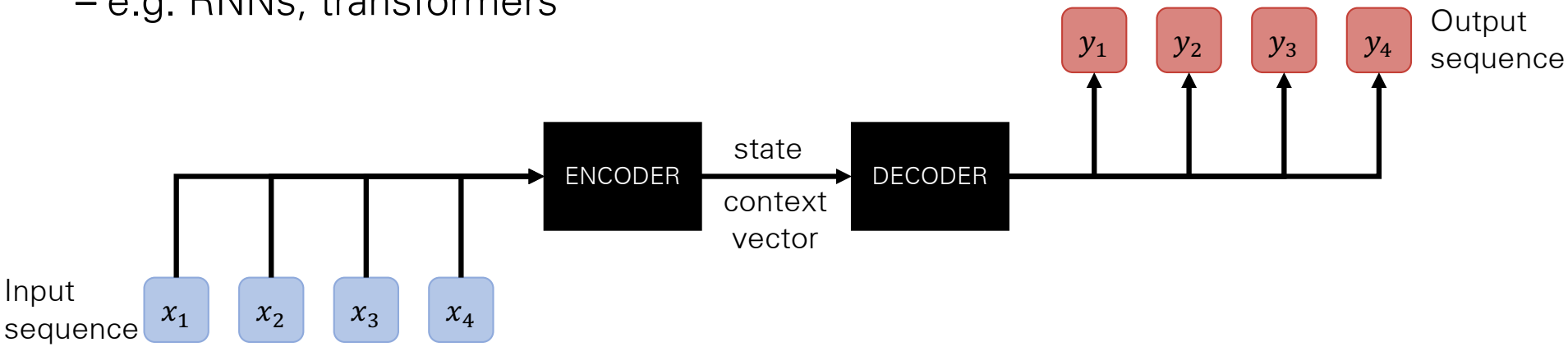
- Example Scenarios
  - Text → Text (e.g. Q/A, translation, text summarization)
  - Image → Text (e.g. image captioning)



Sequence to Sequence

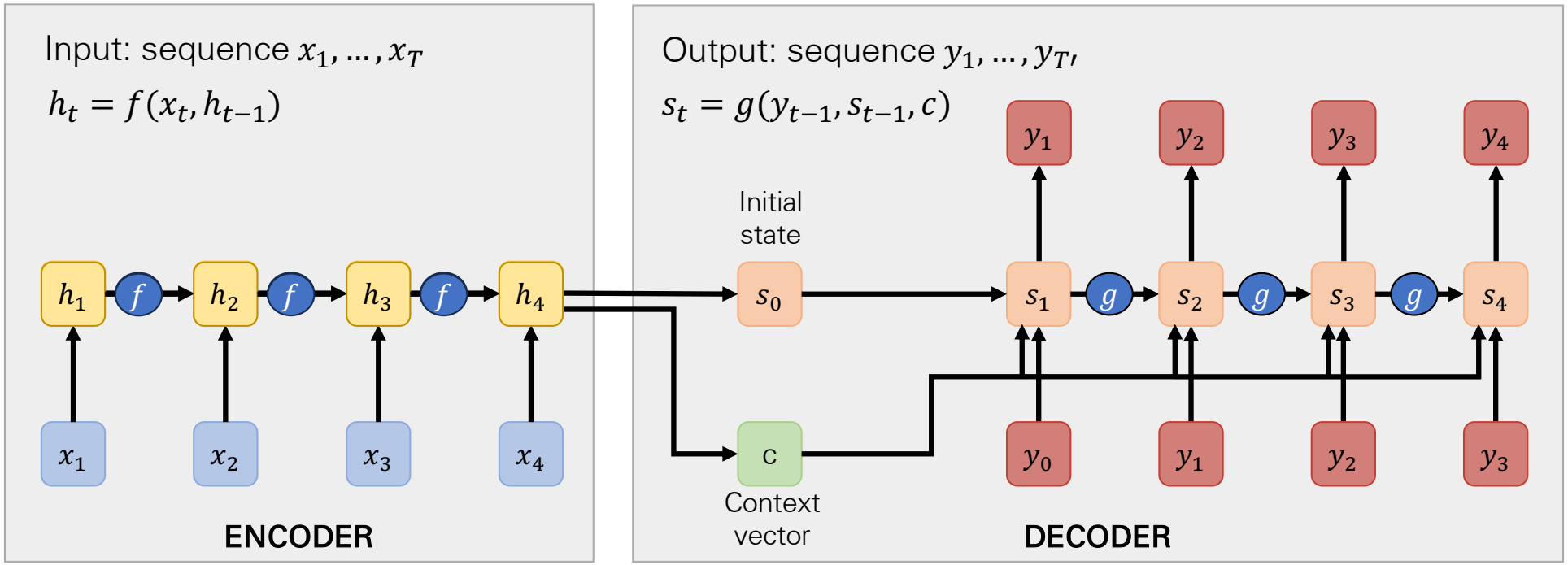
- **Example Scenarios**
  - Text → Text (e.g. Q/A, translation, text summarization)
  - Image → Text (e.g. image captioning)

- **How?** Usually Encoder-Decoder models
  - e.g. RNNs, transformers



Sequence to Sequence with RNNs

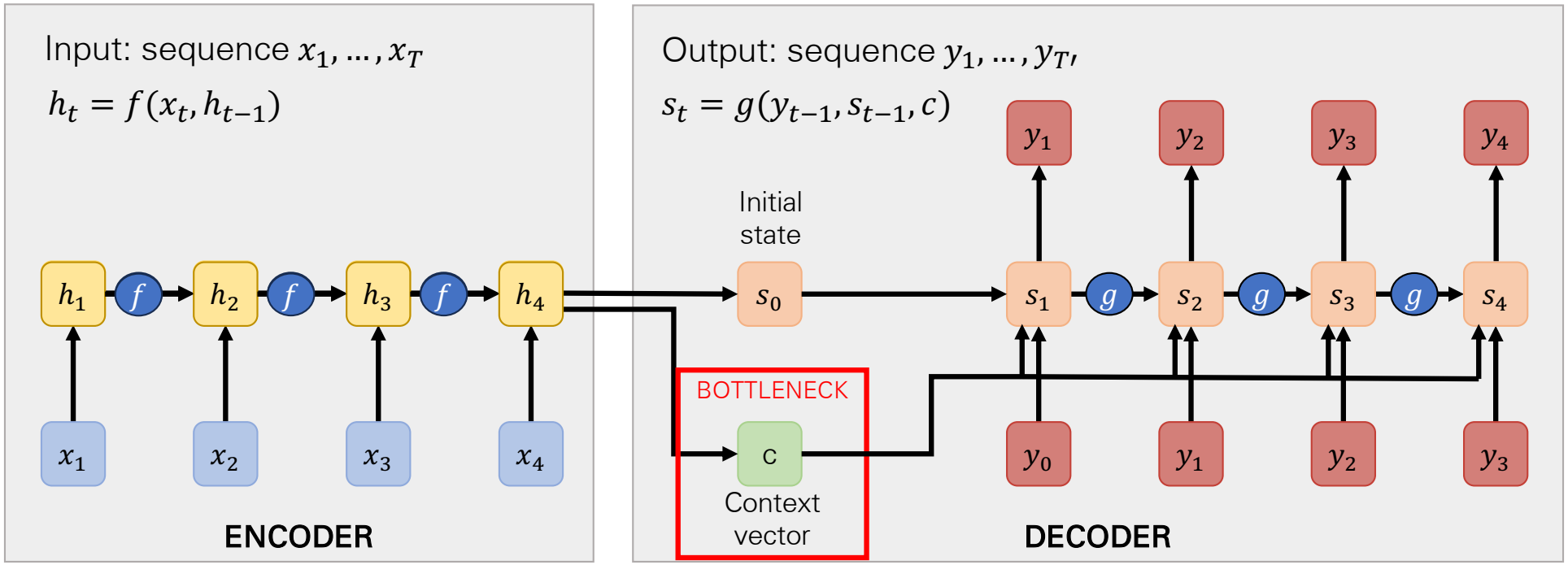
- Encoder (LSTM) and decoder (LSTM)
- Fixed-length context vector



I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS), 2014, pp. 3104–3112.

Sequence to Sequence with RNNs

- Encoder (LSTM) and decoder (LSTM)
- Fixed-length context vector (bottleneck)



I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS), 2014, pp. 3104–3112.

Sequence to Sequence with RNNs + **Attention**

- **Idea!** Use a different context vector for each timestep in the decoder

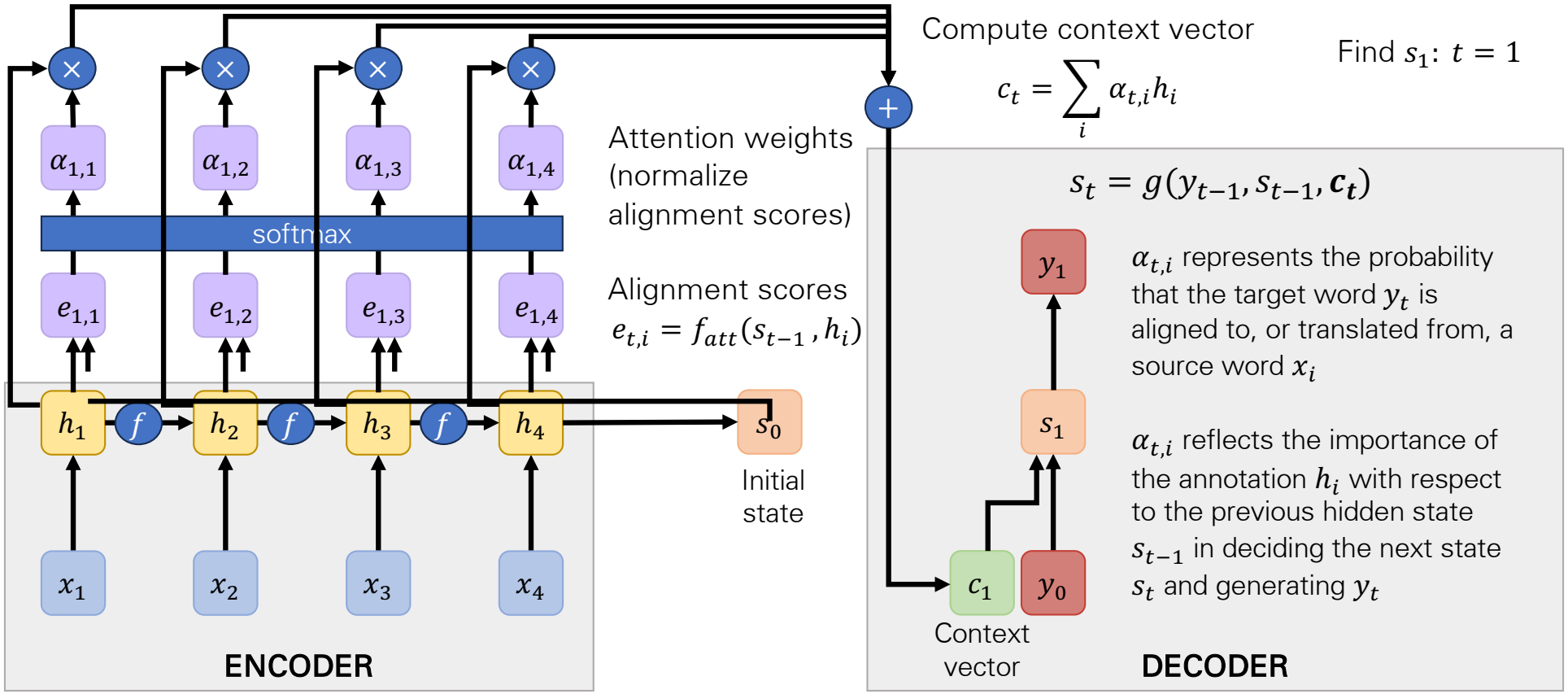
$$s_t = g(y_{t-1}, s_{t-1}, \boldsymbol{c_t})$$

- No more bottleneck through a single vector

- Craft the context vector so that it "looks at" different parts of the input sequence for each decoder timestep

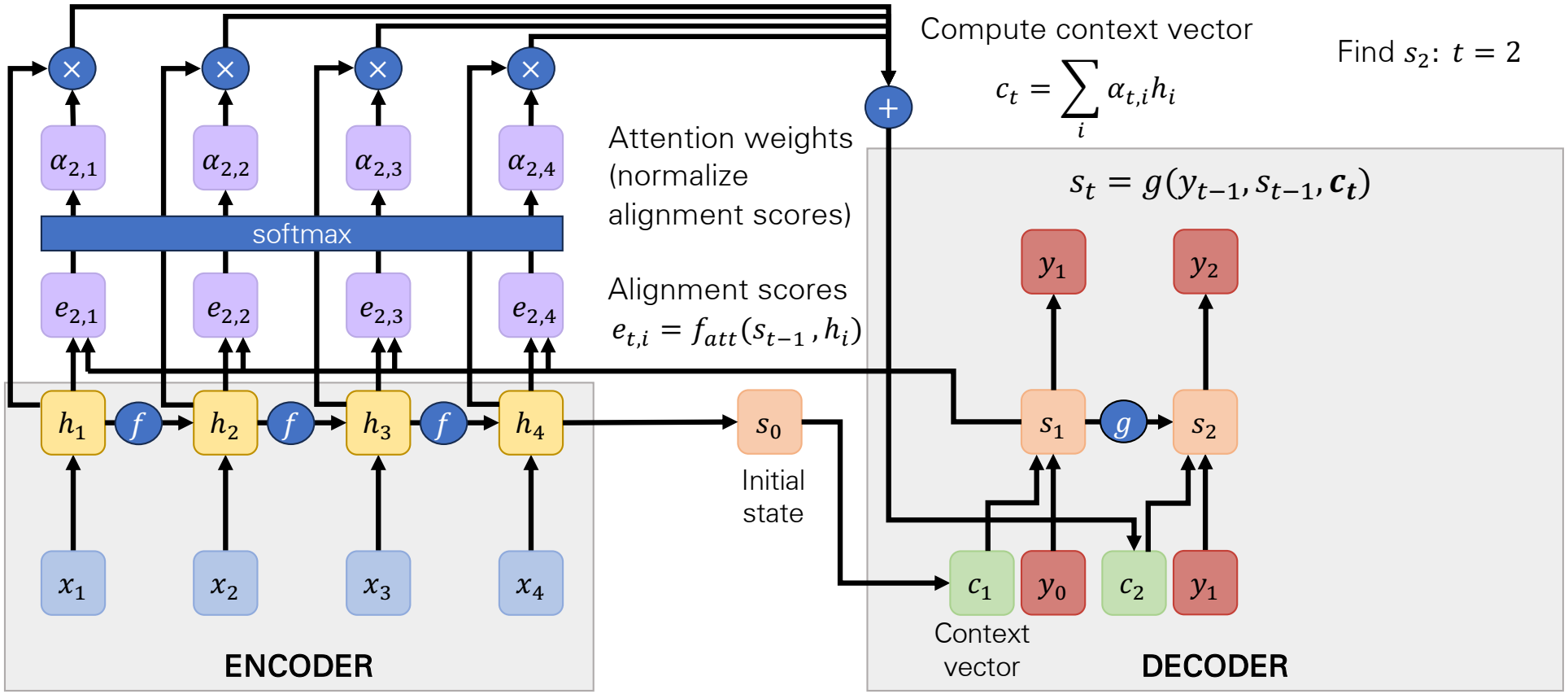
D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in 3rd International Conference on Learning Representations (ICLR), 2015.

Sequence to Sequence with RNNs + **Attention**



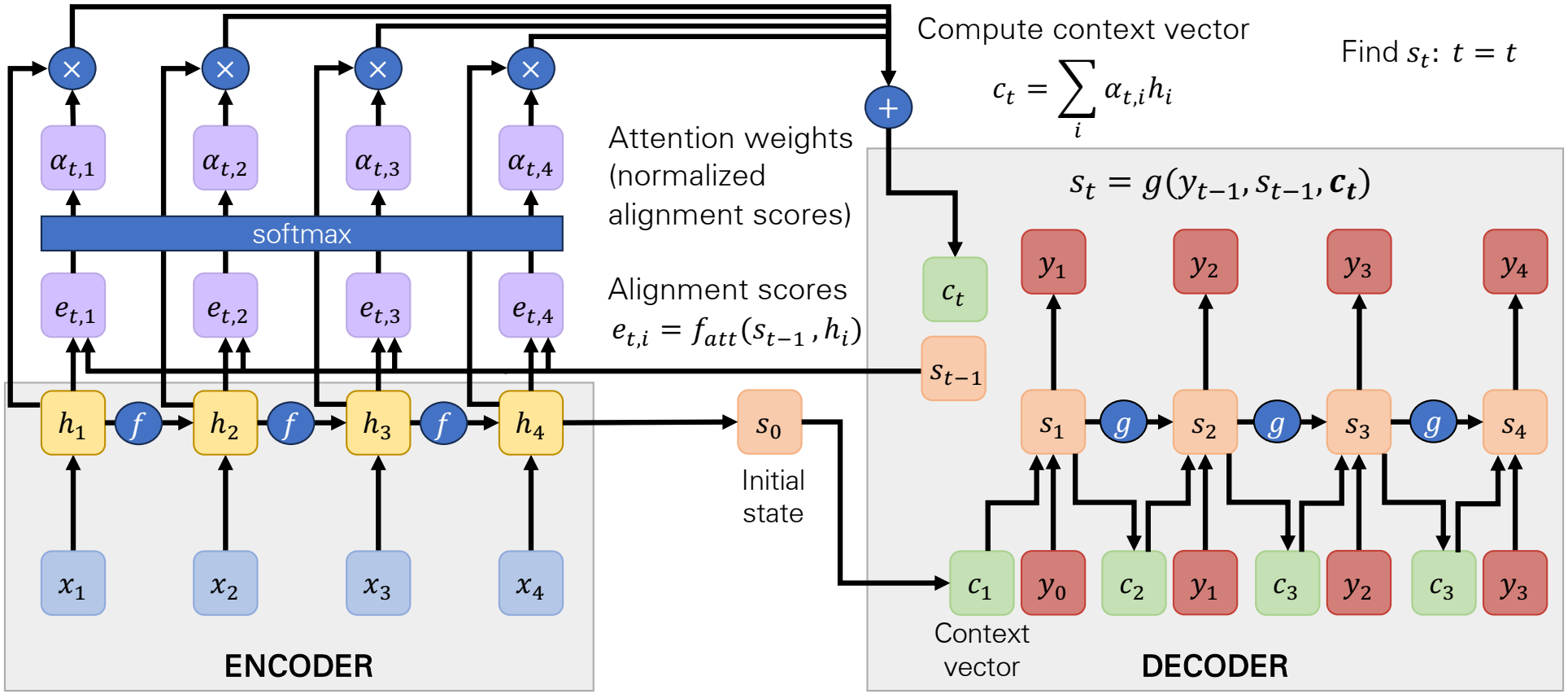
D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in 3rd International Conference on Learning Representations (ICLR), 2015.

Sequence to Sequence with RNNs + **Attention**



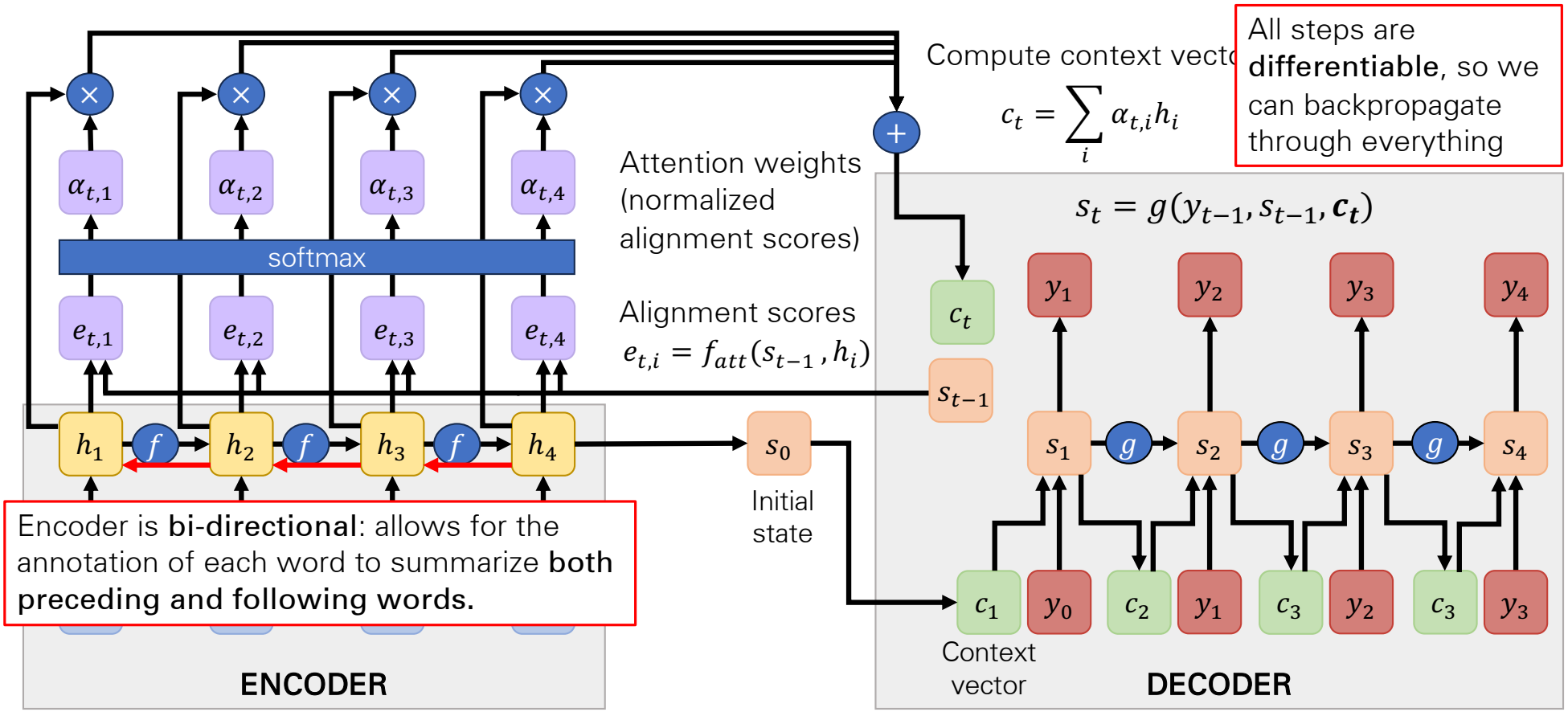
D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in 3rd International Conference on Learning Representations (ICLR), 2015.

Sequence to Sequence with RNNs + **Attention**



D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in 3rd International Conference on Learning Representations (ICLR), 2015.

Sequence to Sequence with RNNs + **Attention**

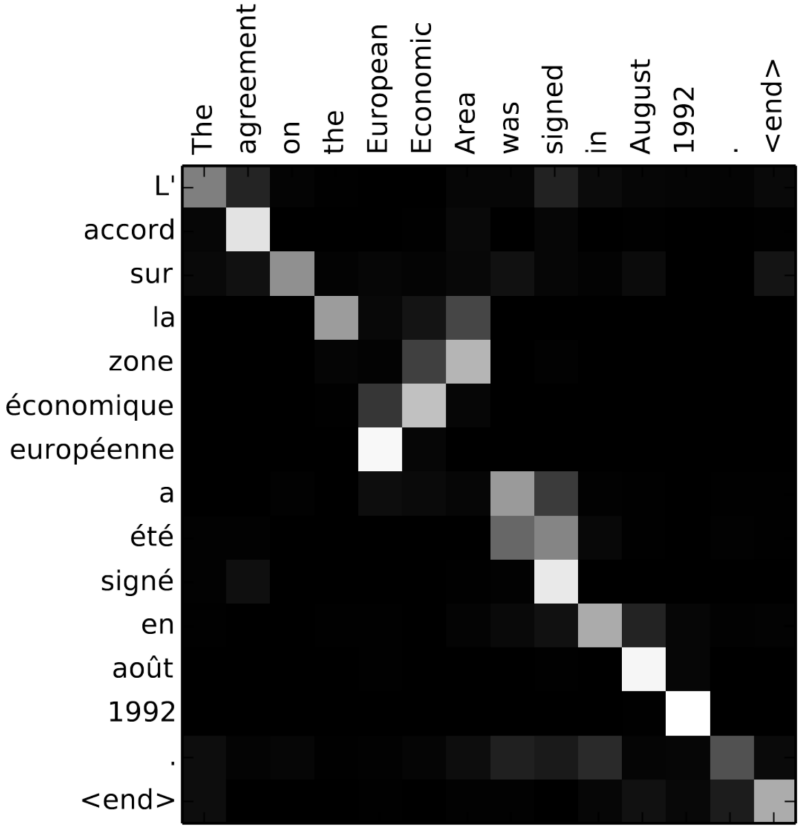


D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in 3rd International Conference on Learning Representations (ICLR), 2015.

Sequence to Sequence with RNNs + **Attention**

Application: translation

Each pixel shows the weight $\alpha_{t,i}$ of the annotation of the i-th source word for the t-th target word.



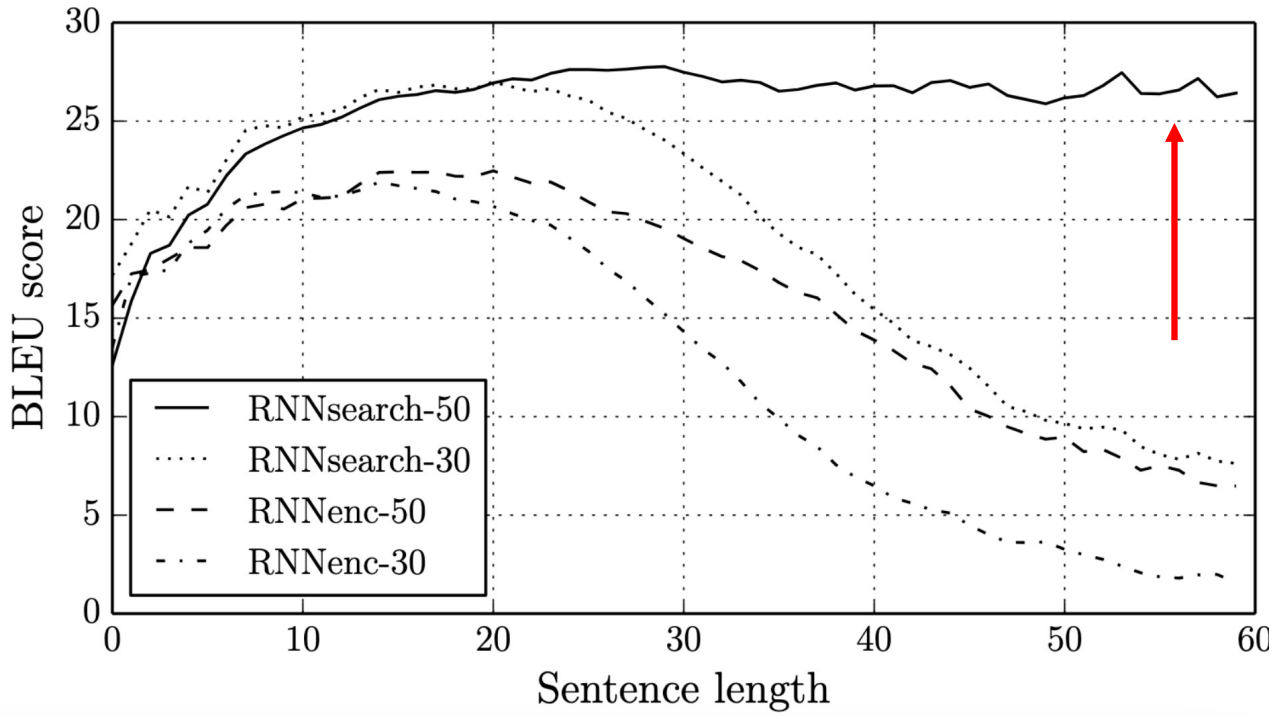
D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in 3rd International Conference on Learning Representations (ICLR), 2015.

Sequence to Sequence with RNNs + **Attention**

Application:
text translation

RNN:
RNNenc

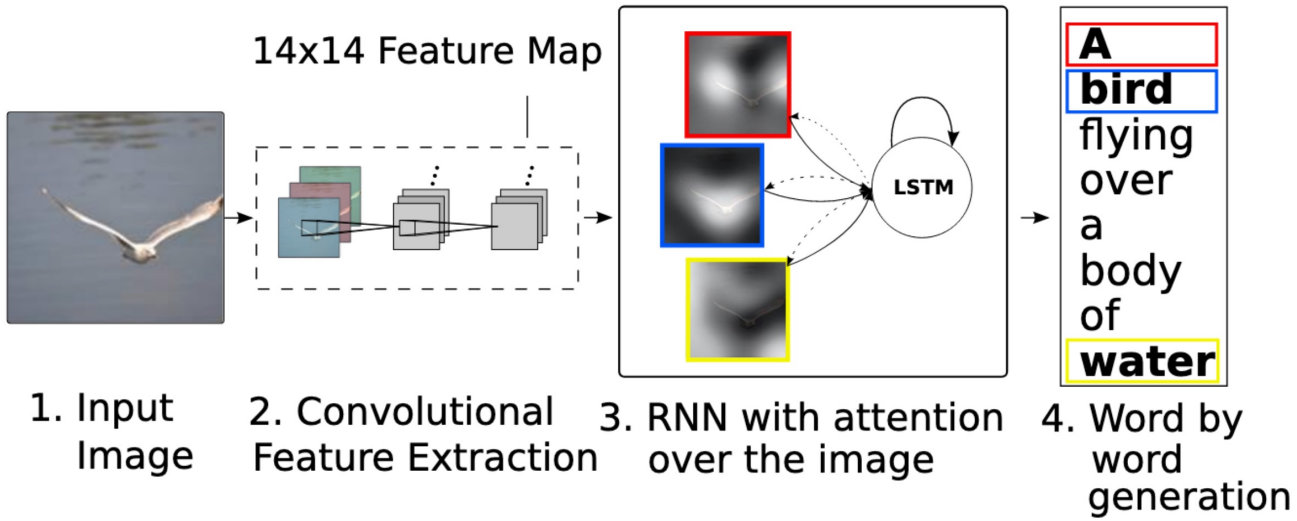
RNN + attention:
RNNsearch



D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in 3rd International Conference on Learning Representations (ICLR), 2015.

Image Captioning with Visual Attention

- We can similarly use attention for image captioning (image → text)
- Builds directly on previous work



K. Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," in PMLR, 2015, pp. 2048–2057.

Image Captioning with Visual Attention

h_i corresponds to a part of the image

$f_{att}(\cdot)$ is an MLP

$$e_{ti} = f_{\text{att}}(h_i, s_{t-1})$$

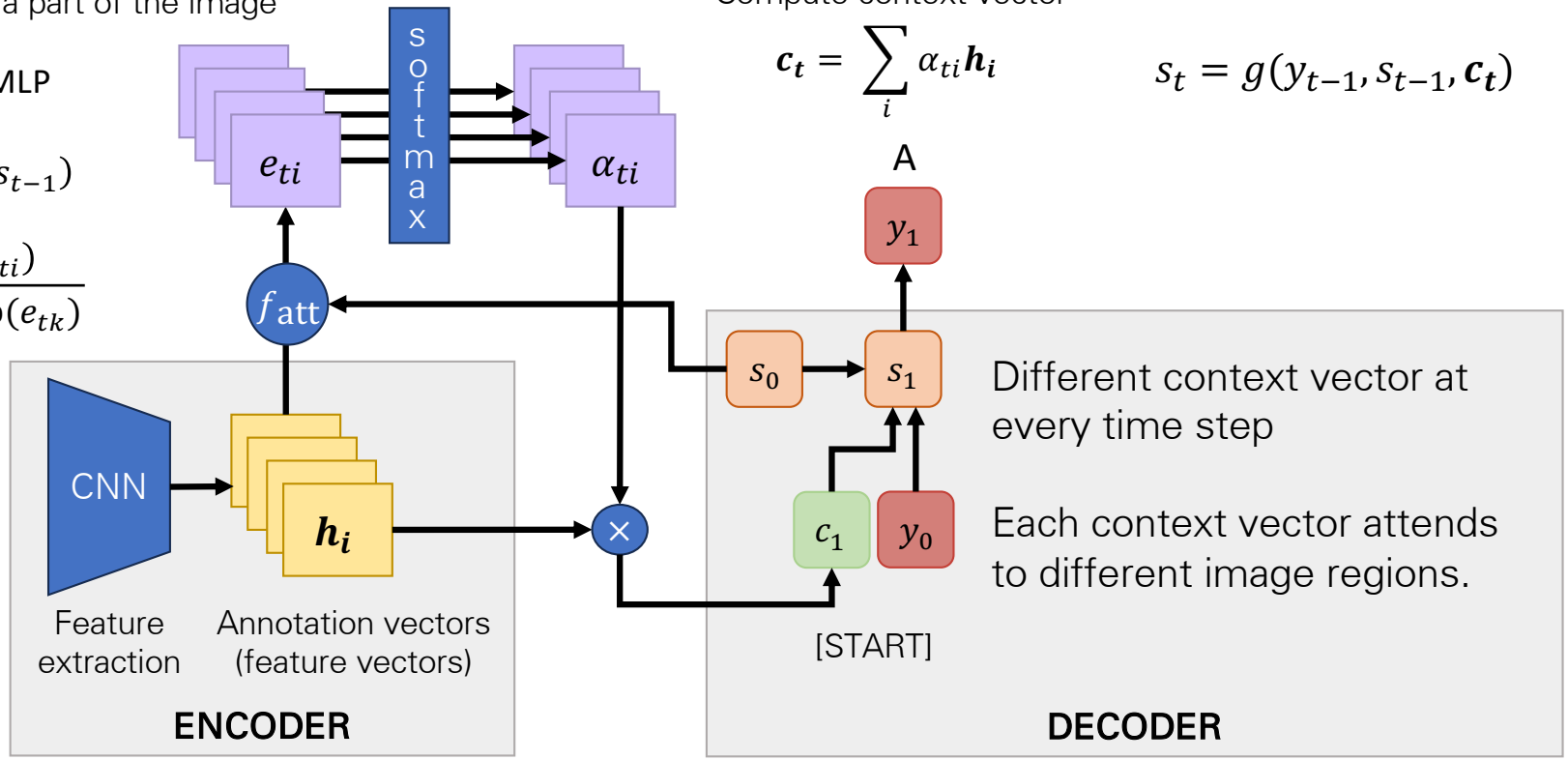
$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{k=1}^{L} \exp(e_{tk})}$$

Compute context vector

$$\boldsymbol{c_t} = \sum_i \alpha_{ti} \boldsymbol{h_i}$$

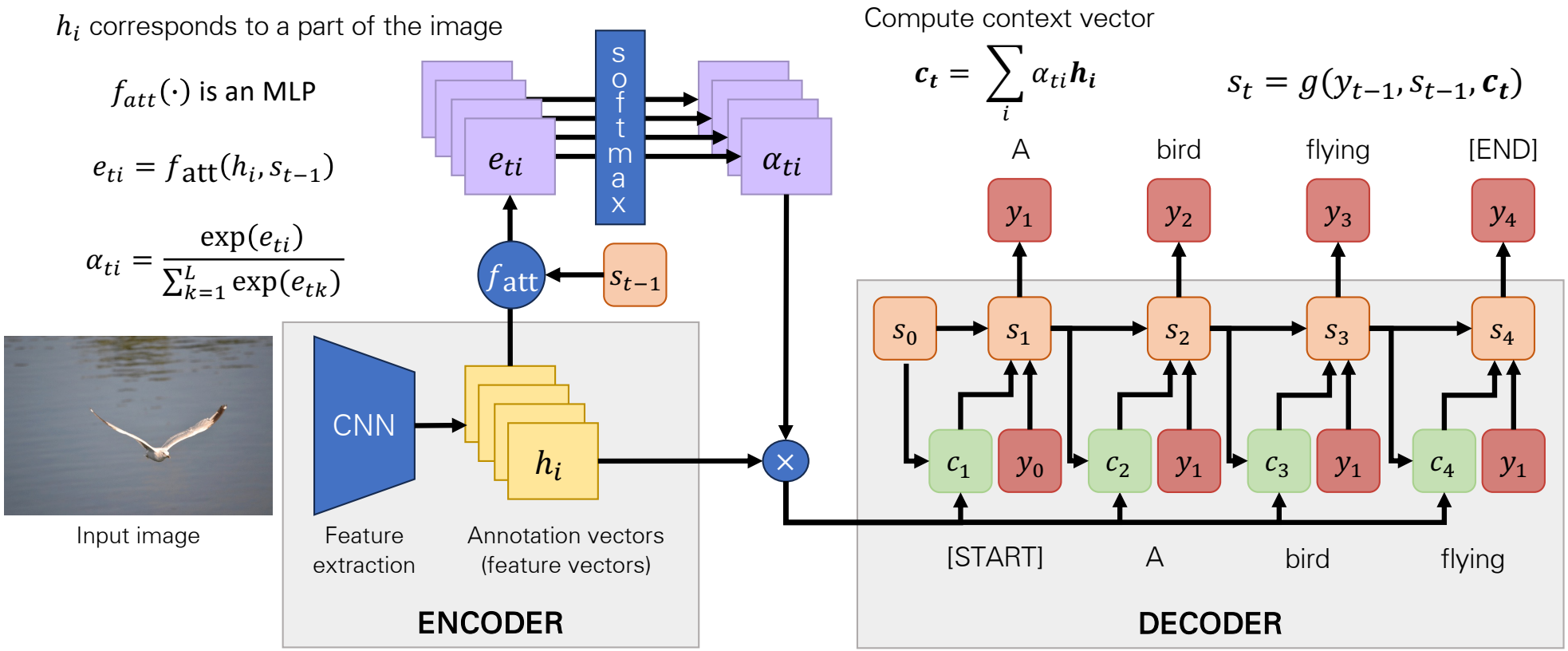
$$s_t = g(y_{t-1}, s_{t-1}, \boldsymbol{c_t})$$

Input image



K. Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," in PMLR, 2015, pp. 2048–2057.

Image Captioning with Visual Attention



K. Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," in PMLR, 2015, pp. 2048–2057.

Image Captioning with Visual Attention

All steps are **differentiable**, so we can backpropagate through everything.

Each context vector attends to different image regions.

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{k=1}^{L} \exp(e_{tk})}$$

Input image

CNN

Feature extraction

h_i

Annotation vectors (feature vectors)

ENCODER

f_{att}

s_{t-1}

α_{ti}

Compute context vector

$$\boldsymbol{c_t} = \sum_i \alpha_{ti} \boldsymbol{h_i}$$

$$s_t = g(y_{t-1}, s_{t-1}, \boldsymbol{c_t})$$

A bird flying [END]

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_1 c_4 y_1

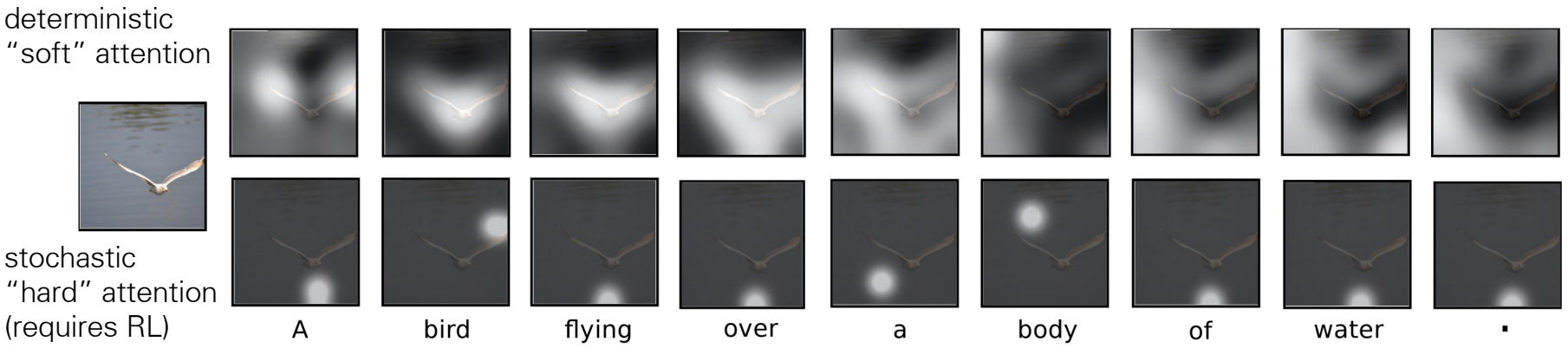
[START] A bird flying

DECODER

K. Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," in PMLR, 2015, pp. 2048–2057.

Image Captioning with Visual Attention

- Visualization of the attention for each generated word
  - Gives insight to "where" and "what" the attention focused on when generating each word



K. Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," in PMLR, 2015, pp. 2048–2057.

Image Captioning with Visual Attention

A woman is throwing a frisbee in a park.

A dog is standing on a hardwood floor.

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

A little girl sitting on a bed with a teddy bear.

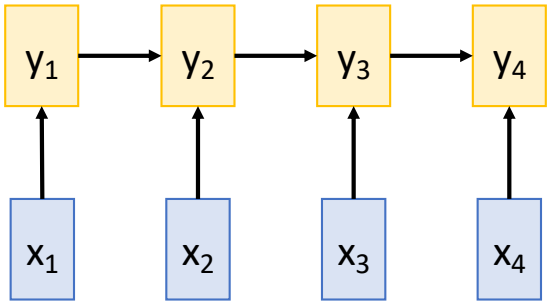
A group of people sitting on a boat in the water.

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

K. Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," in PMLR, 2015, pp. 2048–2057.

Three Ways of Processing Sequences

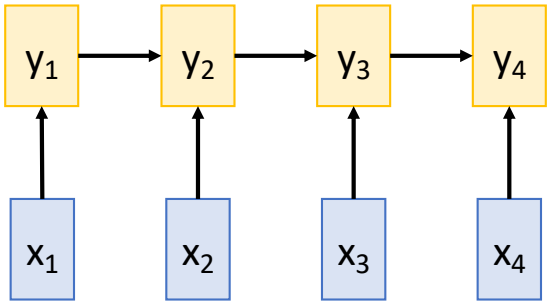
Recurrent Neural Network



works on **ordered sequences**
(+) good at long sequences: after one RNN layer, h_T "sees" the whole sequence
(-) not parallelizable: need to compute hidden states sequentially

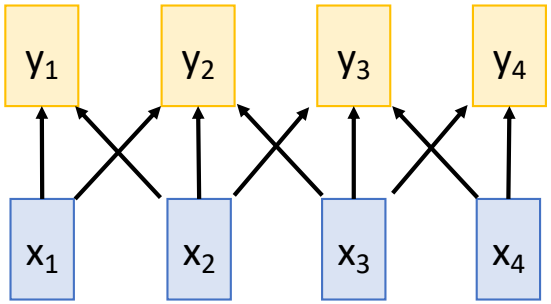
Three Ways of Processing Sequences

Recurrent Neural Network



works on **ordered sequences**
(+) good at long sequences: after one RNN layer, h_T "sees" the whole sequence
(-) not parallelizable: need to compute hidden states sequentially

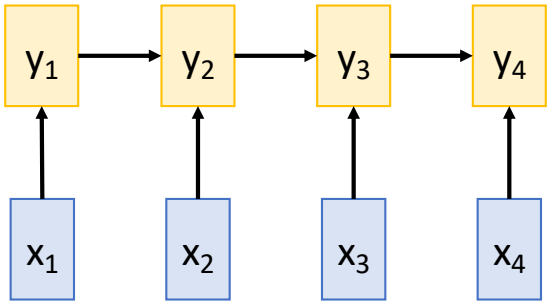
1D Convolution



works on **multidimensional grids**
(-) bad at long sequences: need to stack many conv layers for outputs to "see" the whole sequence
(+) highly parallel: each output can be computed in parallel

Three Ways of Processing Sequences

Recurrent Neural Network

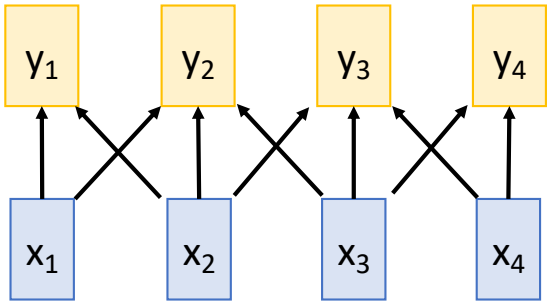


works on **ordered sequences**

(+) good at long sequences: after one RNN layer, h_T "sees" the whole sequence

(-) not parallelizable: need to compute hidden states sequentially

1D Convolution

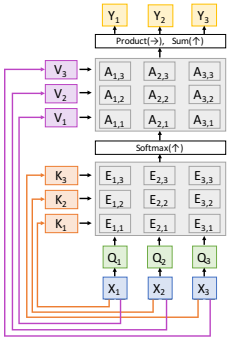


works on **multidimensional grids**

(-) bad at long sequences: need to stack many conv layers for outputs to "see" the whole sequence

(+) highly parallel: each output can be computed in parallel

Self-Attention



works on **sets of Vectors**

(-) good at long sequences: after one self-attention layer, each output "sees" all inputs!

(+) highly parallel: each output can be computed in parallel

(-) very memory intensive

Three Ways of Processing Sequences

Recurrent Neural Network | 1D Convolution | Self-Attention

Attention is all you need

Vaswani et al, NeurIPS 2017

works on **ordered sequences**
(+) good at long sequences: after one RNN layer, h_T "sees" the whole sequence
(-) not parallelizable: need to compute hidden states sequentially

works on **multidimensional grids**
(-) bad at long sequences: need to stack many conv layers for outputs to "see" the whole sequence
(+) highly parallel: each output can be computed in parallel

works on **sets of Vectors**
(-) good at long sequences: after one self-attention layer, each output "sees" all inputs!
(+) highly parallel: each output can be computed in parallel
(-) very memory intensive

Attention is All you Need (2017)

- **Key Idea:**
  - Decouple attention from RNNs
  - Use self-attention to make this efficient

- **Contributions:**
  - Multi-head attention
  - Transformer architecture

- Highly impactful (as we'll touch on later)

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

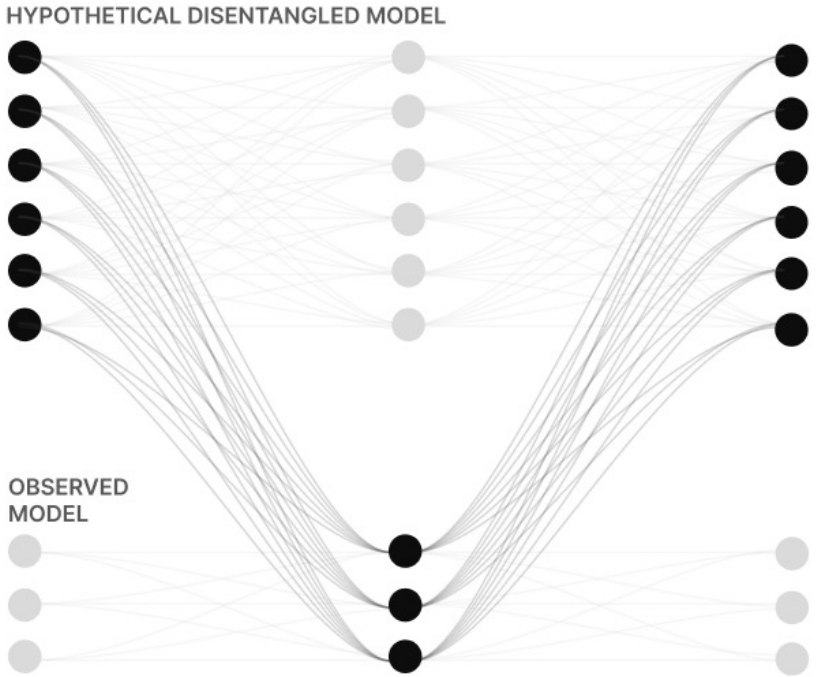
Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

A. Vaswani et al., "Attention is All you Need," in Advances in Neural Information Processing Systems (NeurIPS), 2017.

Feature Superposition (Polysemanticity)

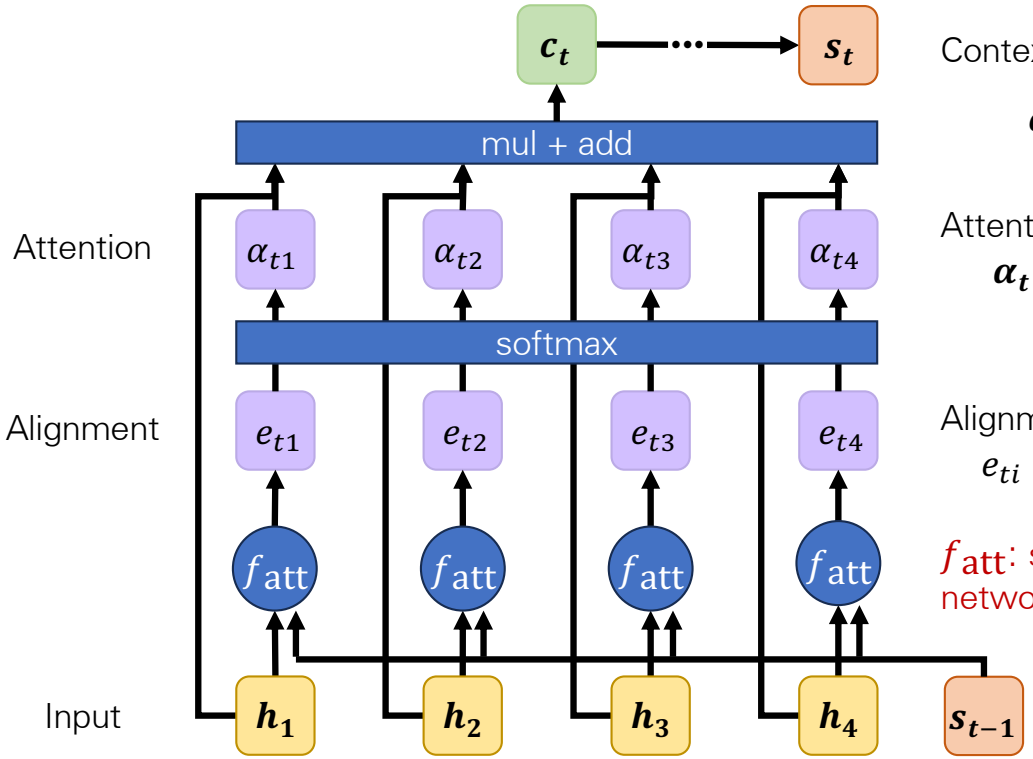
- A NN neural activation often does not represent a single thing
- "Neural networks want to represent more features than they have neurons for"[1]
- **Superposition of features:** "often pack many unrelated concepts into a single neuron" [1]
- Results in decreased explainability
  - A paper from Anthropic seeks to add explainability in LLMs [2]



[1] N. Elhage et al., "Toy Models of Superposition." arXiv, 2022. doi: 10.48550/arXiv.2209.10652.
[2] T. Bricken et al., "Towards Monosemanticity: Decomposing Language Models With Dictionary Learning." 2023. [Online]. Available: https://transformer-circuits.pub/2023/monosemantic-features/index.html

Attention we've seen so far

Now known as **"additive" recurrent attention** (type of encoder-decoder attention)



Context vector

$$\boldsymbol{c_t} = \sum_i \alpha_{ti} \boldsymbol{h_i}$$

Attention weights

$$\boldsymbol{\alpha_t} = \text{softmax}(\boldsymbol{e_t})$$

Alignment scores

$$e_{ti} = f_{\text{att}}(\boldsymbol{s_{t-1}}, \boldsymbol{h_i})$$

f_{att}: simple feedforward network (e.g. MLP)

Issues with Recurrent Attention

- Scalability issues
  - Performance degrades as the distance between words increases
- Parallelization limitations
  - Recurrent processes lacks ability to be parallelized
- Memory constraints
  - RNNs have limited memory and struggle with long-range dependencies
  - Diluted impact of earlier elements on output as sequence progresses

- **Potential solution:** decouple attention from RNNs
  - How? Separate the attention mechanism into smaller, self-contained components

Decoupling from RNNs

- **Recall:** attention determines the **importance of elements** to be passed forward in the model.
  - These weights lets the model pay **attention** to the **most significant parts**

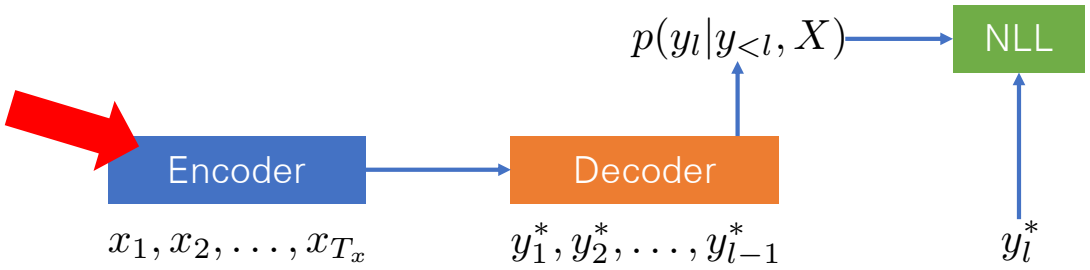
- **Objective**: a more general attention mechanism not confined to RNNs
  - We need a modified procedure to:
    1. Determine weights based on context that indicate the elements to attend to
    2. Apply these weights to enhance attended features

Parametrization – Recurrent Neural Nets

- Following Bahdanau et al. [2015]
- The encoder turns a sequence of tokens into a sequence of contextualized vectors.

$$h_t = [\overrightarrow{h}_t; \overleftarrow{h}_t], \text{ where } \overrightarrow{h}_t = \text{RNN}(x_t, \overrightarrow{h}_{t-1}), \overleftarrow{h}_t = \text{RNN}(x_t, \overleftarrow{h}_{t+1})$$

- The underlying principle behind recently successful contextualized embeddings
  - ELMo [Peters et al., 2018], BERT [Devlin et al., 2019] and all the other muppets



Parametrization – Recurrent Neural Nets

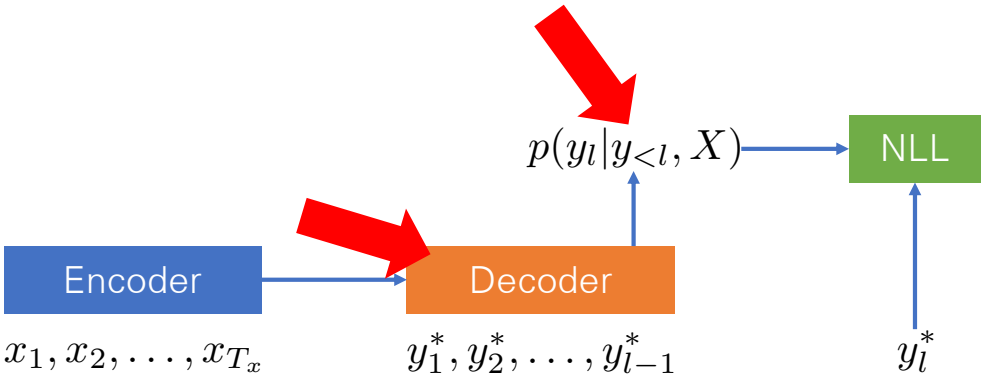
- Following Bahdanau et al. [2015]
- The decoder consists of three stages
  1. Attention: attend to a small subset of source vectors
  2. Update: update its internal state
  3. Predict: predict the next token

$$\alpha_{t'} \propto \exp(\mathrm{ATT}(h_{t'}, z_{t-1}, y_{t-1}))$$

$$c_t = \sum_{t'=1}^{T_x} \alpha_{t'} h_{t'}$$

$$z_t = \mathrm{RNN}([y_{t-1}; c_t], z_{t-1})$$

$$p(y_t = v | y_{<t}, X) \propto \exp(\mathrm{OUT}(z_t, v))$$



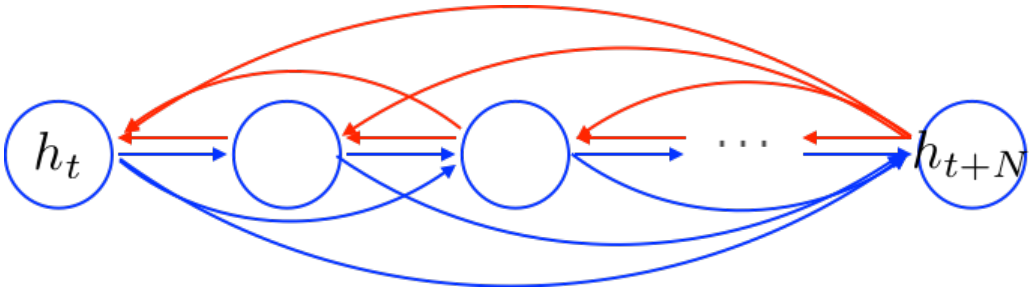
- Attention has become the core component in many recent advances
  - Transformers [Vaswani et al., 2017], …

Side-note: gated recurrent units to attention

- A key idea behind LSTM and GRU is the additive update

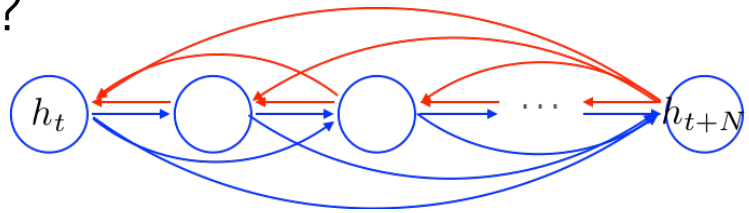
$$h_t = u_t \odot h_{t-1} + (1 - u_t) \odot \tilde{h}_t, \text{ where } \tilde{h}_t = f(x_t, h_{t-1})$$

- This additive update creates linear short-cut connections



Side-note: gated recurrent units to attention

- What are these shortcuts?



- If we unroll it, we see it's a weighted combination of all previous hidden vectors:

$$
\begin{aligned}
h_t =& u_t \odot h_{t-1} + (1-u_t) \odot \tilde{h}_t, \\
=& u_t \odot (u_{t-1} \odot h_{t-2} + (1-u_{t-1}) \odot \tilde{h}_{t-1}) + (1-u_t) \odot \tilde{h}_t, \\
=& u_t \odot (u_{t-1} \odot (u_{t-2} \odot h_{t-3} + (1-u_{t-2}) \odot \tilde{h}_{t-2}) + (1-u_{t-1}) \odot \tilde{h}_{t-1}) + (1-u_t) \odot \tilde{h}_t, \\
& \vdots \\
=& \sum_{i=1}^{t} \left(\prod_{j=i}^{t-i+1} u_j \right) \left(\prod_{k=1}^{i-1} (1-u_k) \right) \tilde{h}_i
\end{aligned}
$$

Side-note: gated recurrent units to attention

1. Can we "free" these dependent weights?
2. Can we "free" candidate vectors?
3. Can we separate keys and values?
4. Can we have multiple attention heads?

$$h_t = \sum_{i=1}^{t} \left(\prod_{j=i}^{t-i+1} u_j \right) \left(\prod_{k=1}^{i-1} (1 - u_k) \right) \tilde{h}_i \quad \textbf{0}$$

$$h_t = \sum_{i=1}^{t} \alpha_i \tilde{h}_i, \text{ where } \alpha_i \propto \exp(\text{ATT}(\tilde{h}_i, x_t)) \quad \textbf{1}$$

$$h_t = \sum_{i=1}^{t} \alpha_i f(x_i), \text{ where } \alpha_i \propto \exp(\text{ATT}(f(x_i), x_t)) \quad \textbf{2}$$

$$h_t = \sum_{i=1}^{t} \alpha_i V(f(x_i)), \text{ where } \alpha_i \propto \exp(\text{ATT}(K(f(x_i)), Q(x_t))) \quad \textbf{3}$$

$$h_t = [h_t^1; \cdots; h_t^K], \text{ where } h_t^k = \sum_{i=1}^{t} \alpha_i^k V^k(f(x_i)), \text{ where } \alpha_i^k \propto \exp(\text{ATT}(K^k(f(x_i)), Q^k(x_t))) \quad \textbf{4}$$

→ Transformers

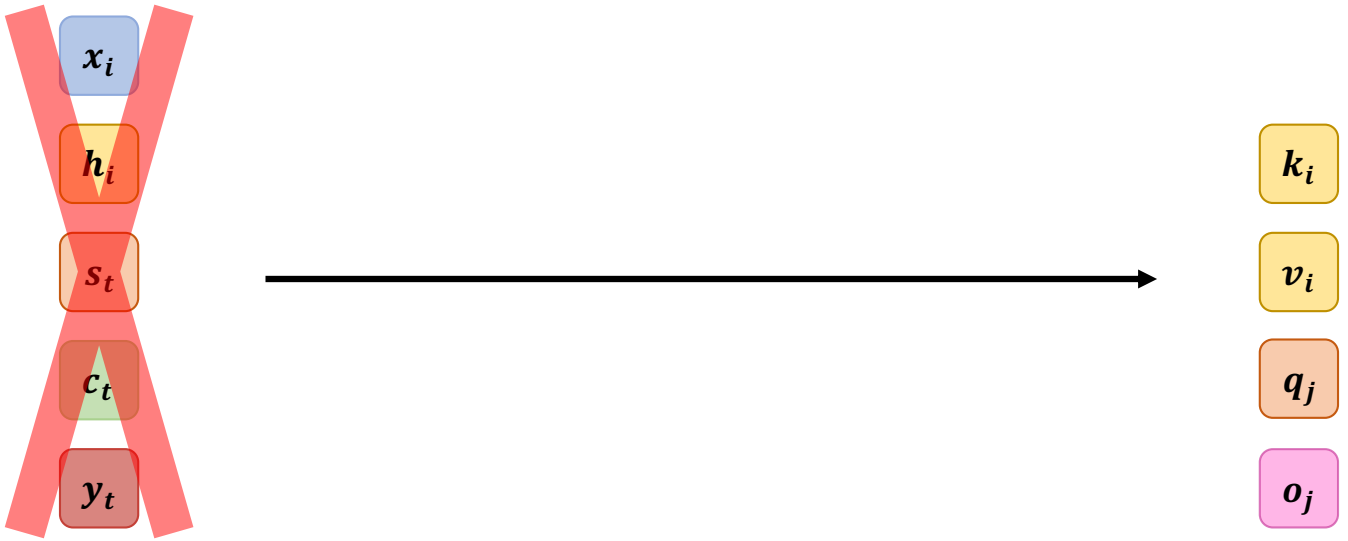
Decoupling from RNNs

- RNN Notation

x_i	Input for position i in source sequence
h_i	Hidden states for position i in source sequence
s_t	Hidden states for position t in target sequence
c_t	Context vector for position t in target sequence
y_t	Output for position t in target sequence

Decoupling from RNNs

- New notation



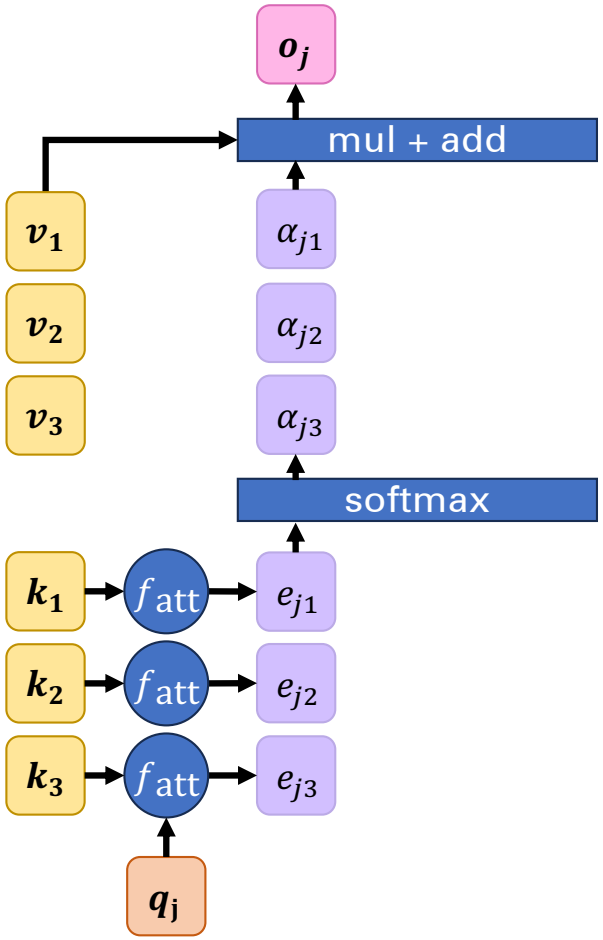
Decoupling from RNNs

- New notation

$\boldsymbol{k_i}$	**Key** vector for position $\boldsymbol{i}$ in an arbitrary sequence
$\boldsymbol{v_i}$	**Value** vector for position $\boldsymbol{i}$ in an arbitrary sequence
$\boldsymbol{q_j}$	**Query** vector for position $\boldsymbol{j}$ in a (same/different) arbitrary sequence
$\boldsymbol{o_j}$	**Output** vector corresponding to position $\boldsymbol{j}$

A more general attention

$Attention(\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v})$



Output vectors

$$\boldsymbol{o_j} = \sum_i \alpha_{ji} \boldsymbol{v_i} \quad \in \mathbb{R}^{d_k}$$

Attention weights

$$\boldsymbol{\alpha_j} = \text{softmax}(\boldsymbol{e_j})$$

Alignment scores

$$e_{ij} = f_{\text{att}}(\boldsymbol{q_j}, \boldsymbol{k_i})$$

Goal: find the "alignment" or "compatibility" of keys with a query to scale values

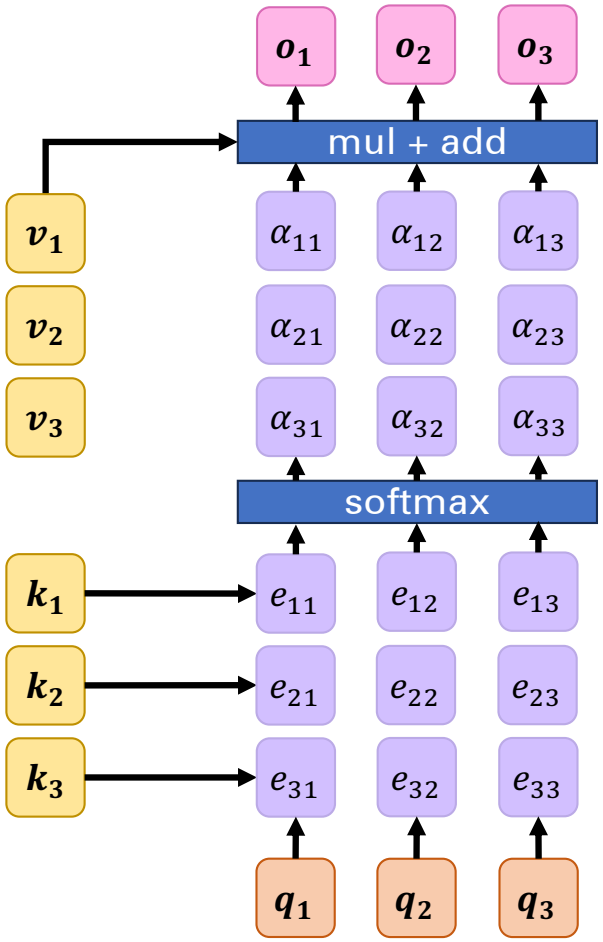
$f_{\text{att}}(\cdot)$

Keys: $\boldsymbol{k_i} \in \mathbb{R}^{d_k}$
Values: $\boldsymbol{v_i} \in \mathbb{R}^{d_k}$
Query: $\boldsymbol{q_j} \in \mathbb{R}^{d_q}$

$i \in \{1, \ldots, N\}$
j $\in \{1, \ldots, T\}$

A more general attention

$Attention(\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v})$



Output vectors

$\boldsymbol{o_j} \in \mathbb{R}^{d_k}$,

for $j \; in \; \{1, \ldots, T\}$

Attention

Attention weights

$\boldsymbol{\alpha_j} = \text{softmax}(\boldsymbol{e_j})$

Alignment

Alignment scores

$e_{ij} = f_{\text{att}}(\boldsymbol{q_j}, \boldsymbol{k_i})$

$f_{\text{att}}(\cdot)$

Keys: $\boldsymbol{k_i} \in \mathbb{R}^{d_k}$

Values: $\boldsymbol{v_i} \in \mathbb{R}^{d_k}$

Query: $\boldsymbol{q_j} \in \mathbb{R}^{d_q}$

$i \in \{1, \ldots, N\}$

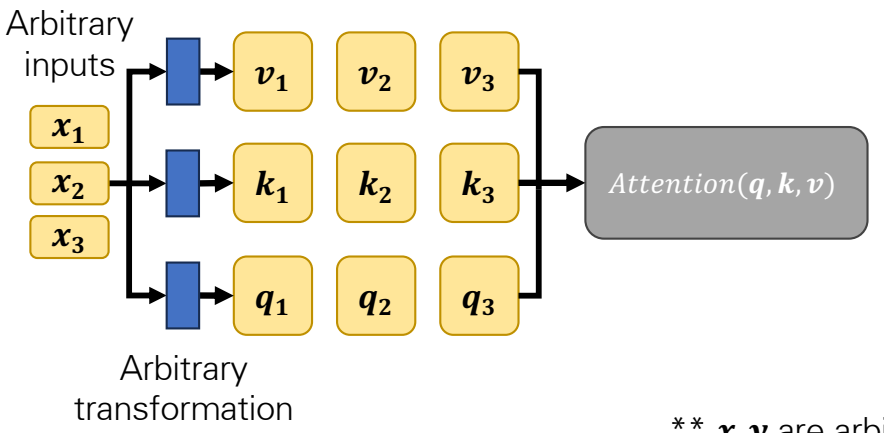
j $\in \{1, \ldots, T\}$

Goal: find the "alignment" or "compatibility" between keys and queries to scale values

Applying the Attention Mechanism

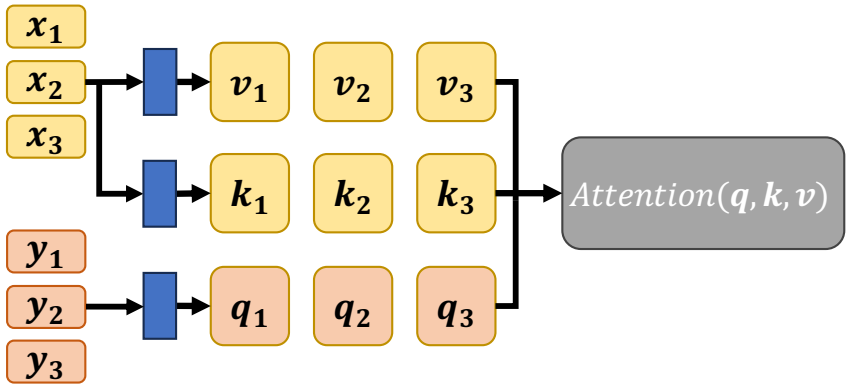
Self-Attention

- Keys, values, and queries are all derived from the same source



Cross-Attention

- Keys-values and queries are derived from separate sources



** x, y are arbitrary sequences

Attention Mechanism in Attention is All You Need

To use a **decoupled attention mechanism**, it is implemented with properties:

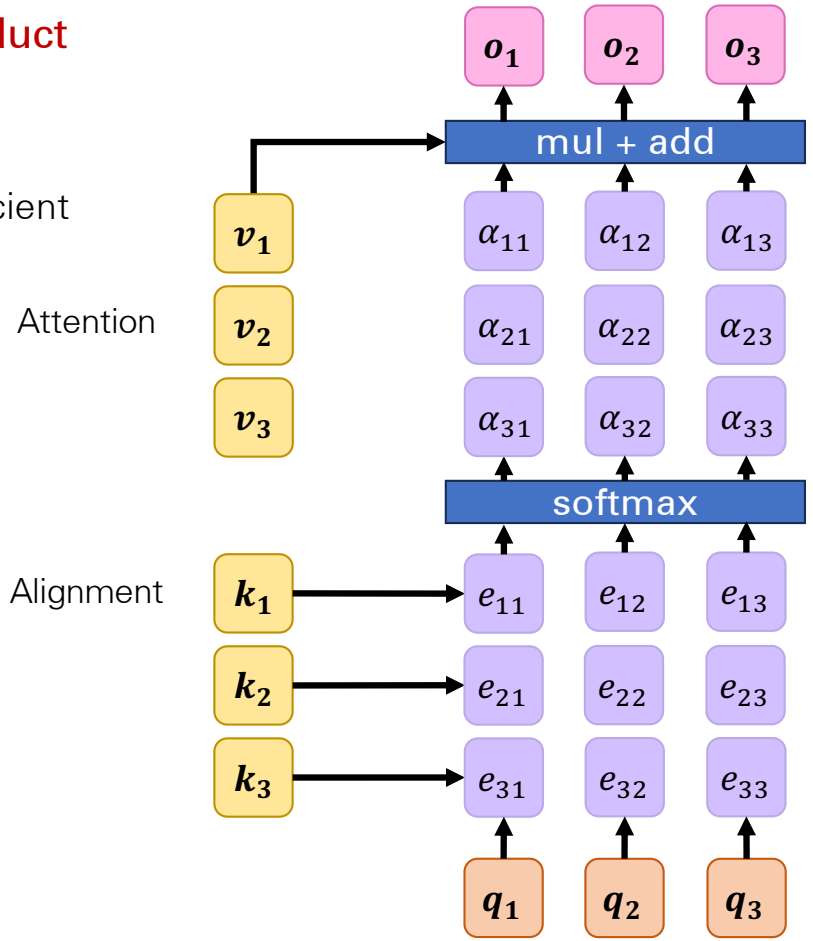
1. $f_{\text{att}}(\cdot)$ = **scaled dot-product attention**
   - Good representation of compatibility
   - Fast and interpretable computation
   - Parallelizable evaluation across all queries (can leverage GPUs)
   - Scaled dot-products for stable softmax gradients in high dimensions (prevents large magnitudes)
2. Imposed a **common dimension** for keys, values, and queries
   - Requirement for dot-product
   - Simplifies architecture with predictable attention output shape
   - Provides consistent hidden state dimensions for easier model analysis

A. Vaswani et al., "Attention is All you Need," in Advances in Neural Information Processing Systems (NeurIPS), 2017.

Attention in Attention is All you Need

Scaled Dot-Product Attention

- Faster
- More space-efficient



Attention

Alignment

Output vectors

$$o_j \in \mathbb{R}^{d_k}$$

Attention weights

$$\alpha_j = \text{softmax}(e_j)$$

Alignment scores

$$e_{ij} = f_{\text{att}}(q_j, k_i) = \frac{q_j \cdot k_i}{\sqrt{d_k}}$$

$f_{\text{att}}(\cdot)$ = scaled dot-product

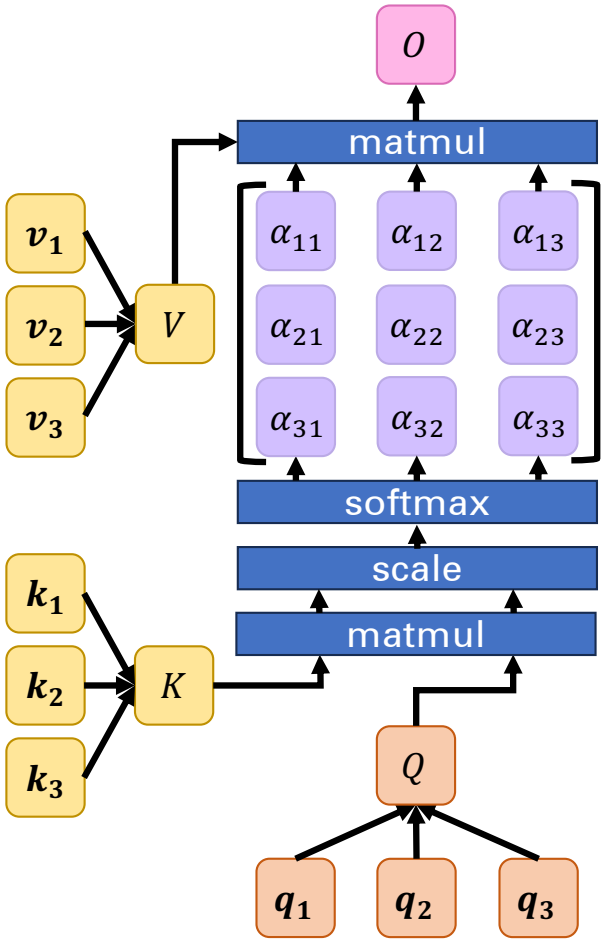
Keys: $k_i \in \mathbb{R}^{d_k}$
Values: $v_i \in \mathbb{R}^{d_k}$
Query: $q_j \in \mathbb{R}^{d_k}$

} Share the same dimension

Attention in Attention is All you Need

Calculate dot-products in **parallel** with **matrix multiplication**

- High concurrency on modern hardware (GPUs)
- Independently calculates each query



Matrix of outputs O ∈ $\mathbb{R}^{T \times d_k}$

Matrix containing attention weights

$$A[j,i] = \text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right)$$

Alignment scores

$e_{ji} = f_{\text{att}}(\boldsymbol{q_j}, \boldsymbol{k_i}) = \frac{Q \cdot K^T}{\sqrt{d_k}}$ [j, i]

$f_{\text{att}}(\cdot)$ = scaled dot-product

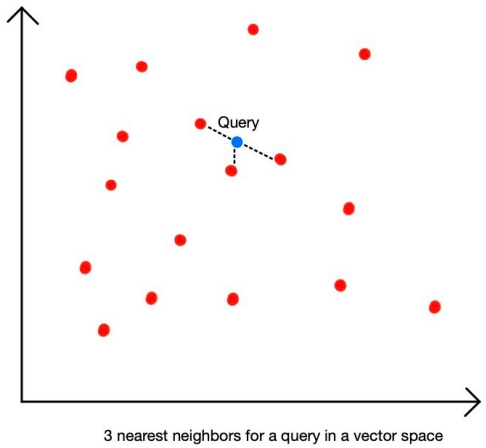
Keys: $K \in \mathbb{R}^{N \times d_k}$

Values: $V \in \mathbb{R}^{N \times d_k}$

Query: $Q \in \mathbb{R}^{T \times d_k}$

Misconceptions about Transformers (1)

- What?
  - Attention in transformers performs a vector similarity search
- Why?
  - Over-simplification in terminology
  - The key-query value explanation is convenient, and many don't know to look past it

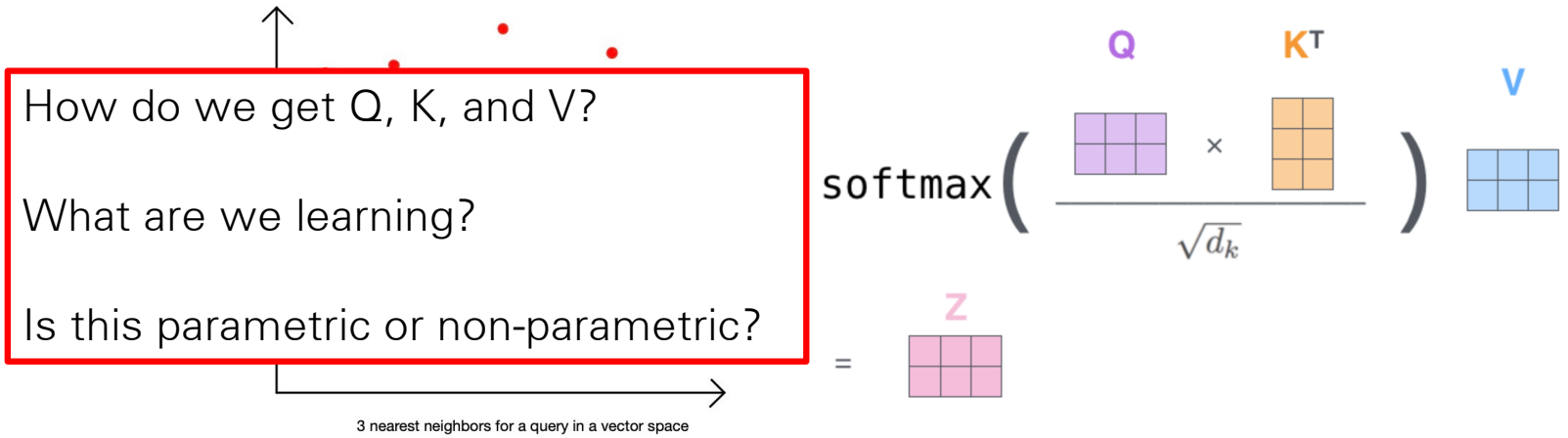


3 nearest neighbors for a query in a vector space

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V = Z$$

Misconceptions about Transformers (1)

- What?
  - Attention in transformers performs a vector similarity search
- Why?
  - Over-simplification in terminology
  - The key-query value explanation is convenient, and many don't know to look past it



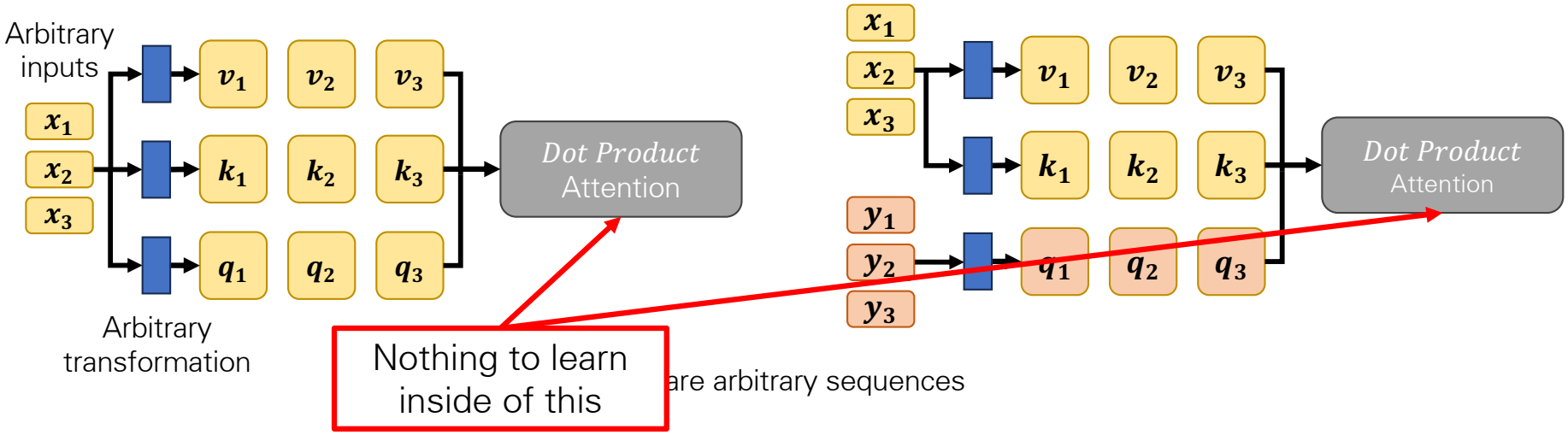
Learning Transformer Attention

Self-Attention

- Keys, values, and queries are all derived from the same source

Cross-Attention

- Keys-values and queries are derived from separate sources



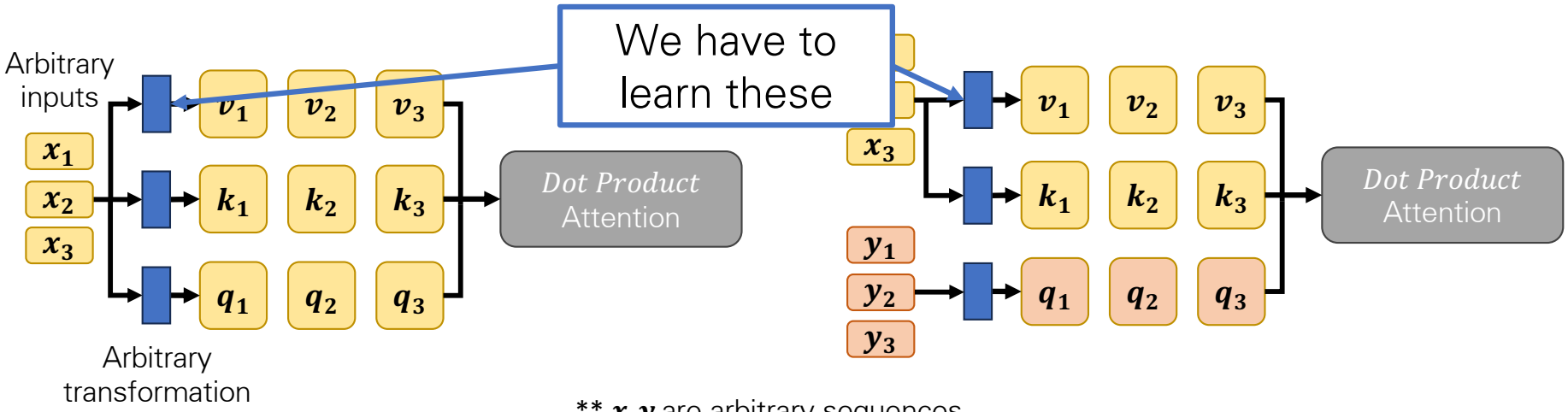
Learning Transformer Attention

Self-Attention

- Keys, values, and queries are all derived from the same source

Cross-Attention

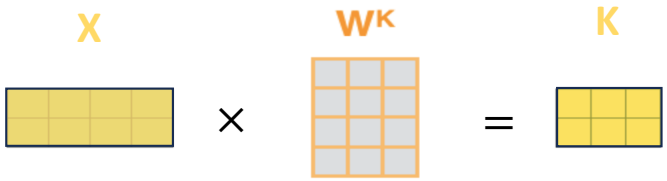
- Keys-values and queries are derived from separate sources



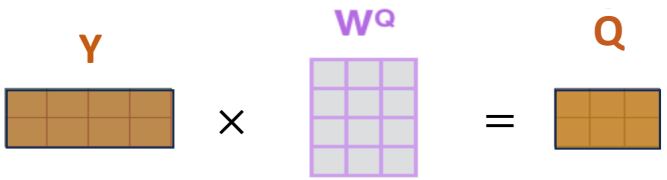
** x, y are arbitrary sequences

Learning Transformer Attention

Self-Attention



Cross-Attention



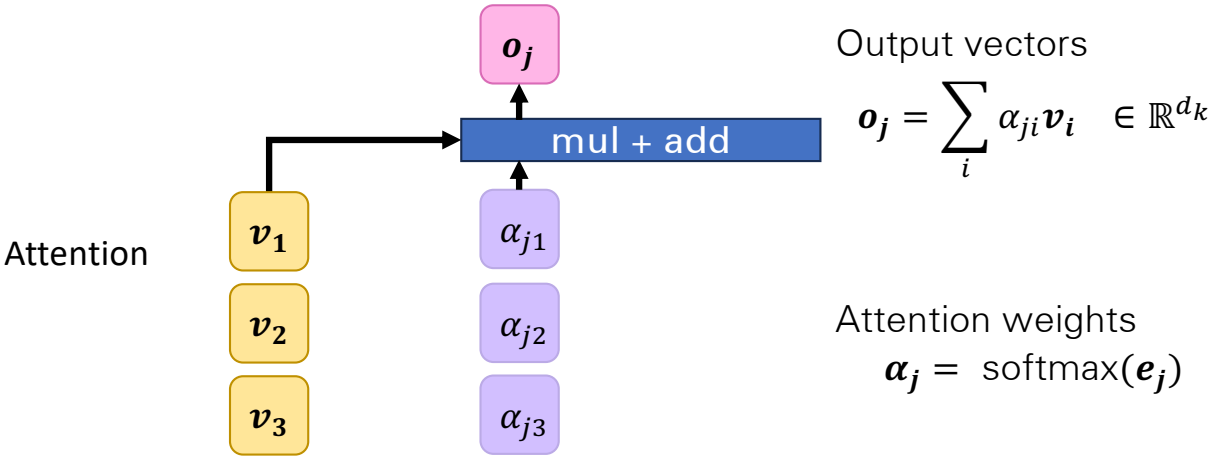
** X, Y are matrices of arbitrary sequences

Multi-Head Attention

- Builds on Scaled Dot-Product Attention
- Extension of generalized attention mentioned outlined previously
- Leverages multiple heads to attend to different things

Learning Multi-Head Attention

Why do we need multiple heads?



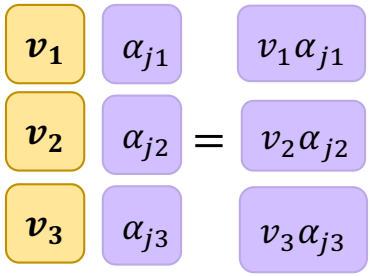
Learning Multi-Head Attention

Why do we need multiple heads?

Output vectors

$$\boldsymbol{o_j} = \sum_i \alpha_{ji} \boldsymbol{v_i} \quad \in \mathbb{R}^{d_k}$$

Attention

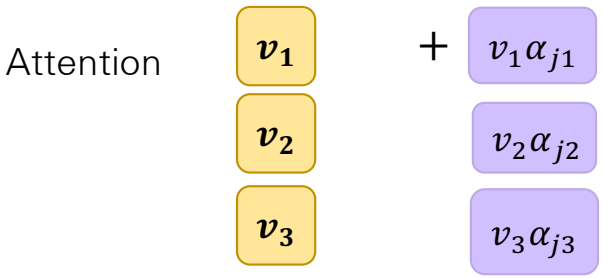


Attention weights

$$\boldsymbol{\alpha_j} = \text{softmax}(\boldsymbol{e_j})$$

Learning Multi-Head Attention

Why do we need multiple heads?



Output vectors

$$\boldsymbol{o_j} = \sum_i \alpha_{ji}\boldsymbol{v_i} \quad \in \mathbb{R}^{d_k}$$

Attention weights

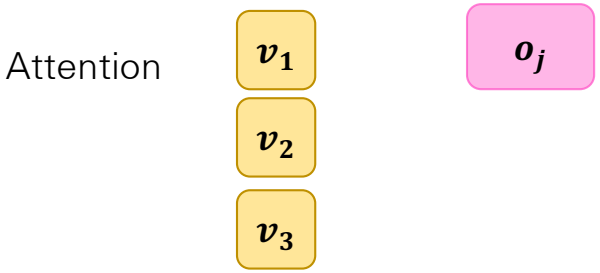
$$\boldsymbol{\alpha_j} = \text{softmax}(\boldsymbol{e_j})$$

Learning Multi-Head Attention

Why do we need multiple heads?

Output vectors

$$\boldsymbol{o_j} = \sum_i \alpha_{ji} \boldsymbol{v_i} \quad \in \mathbb{R}^{d_k}$$

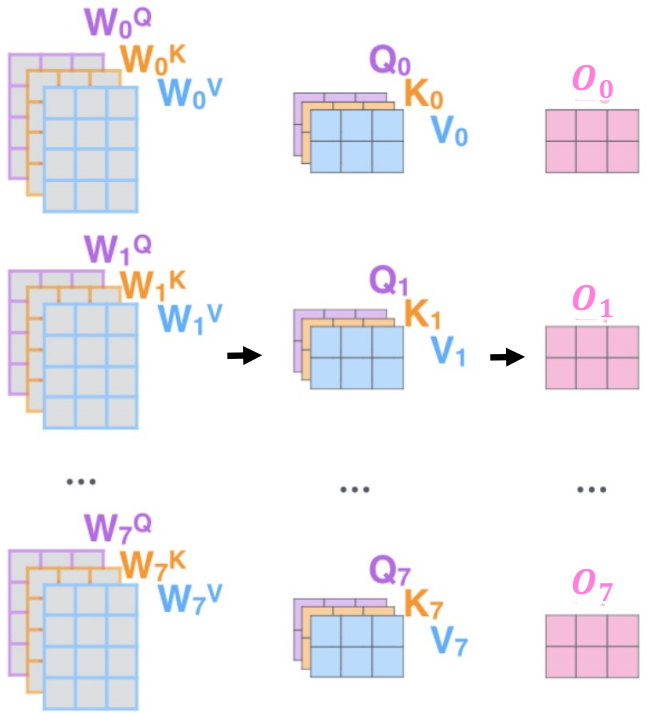


Since we summed through the n positions, we lose resolution in our representation

Learning Multi-Head Attention

- Main idea:
  - Learn multiple sets of weights matrices to attend to different things
  - Preserve resolution since more heads increases chance that the information is maintained

- Allows model to jointly attend to information from different representation subspaces (like ensembling)



Learning Multi-Head Attention

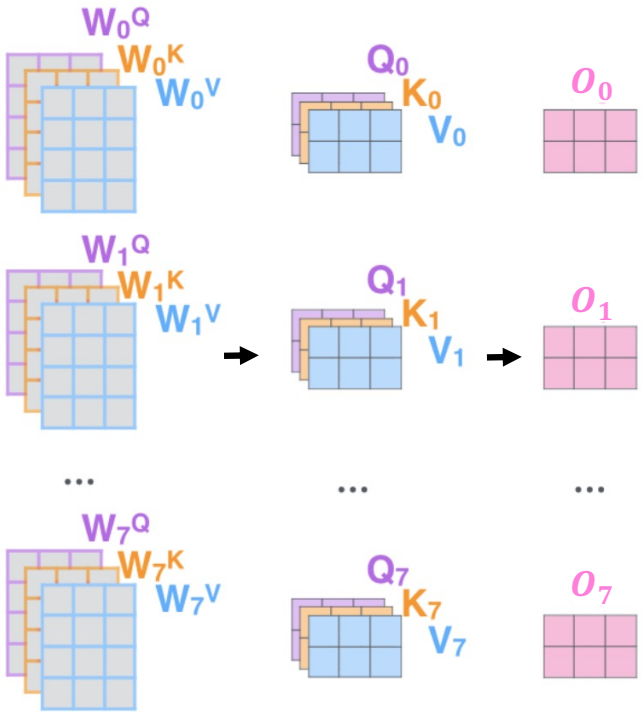
- To make computation efficient, weight matrices project to subspaces

$$W_{h_i}^{Q} \in \mathbb{R}^{d_{model} \times d_k} \rightarrow Q = XW_{h_i}^{Q} \in \mathbb{R}^{t \times d_k},$$
$$W_{h_i}^{K} \in \mathbb{R}^{d_{model} \times d_k} \rightarrow K = XW_{h_i}^{K} \in \mathbb{R}^{n \times d_k},$$
$$W_{h_i}^{V} \in \mathbb{R}^{d_{model} \times d_k} \rightarrow V = XW_{h_i}^{V} \in \mathbb{R}^{n \times d_k},$$

  where $d_k = d_{model}/h$ (512/8 = 64 in paper)

- Together all heads take roughly the same computational time as one fully dimensioned attention head

Learning Multi-Head Attention

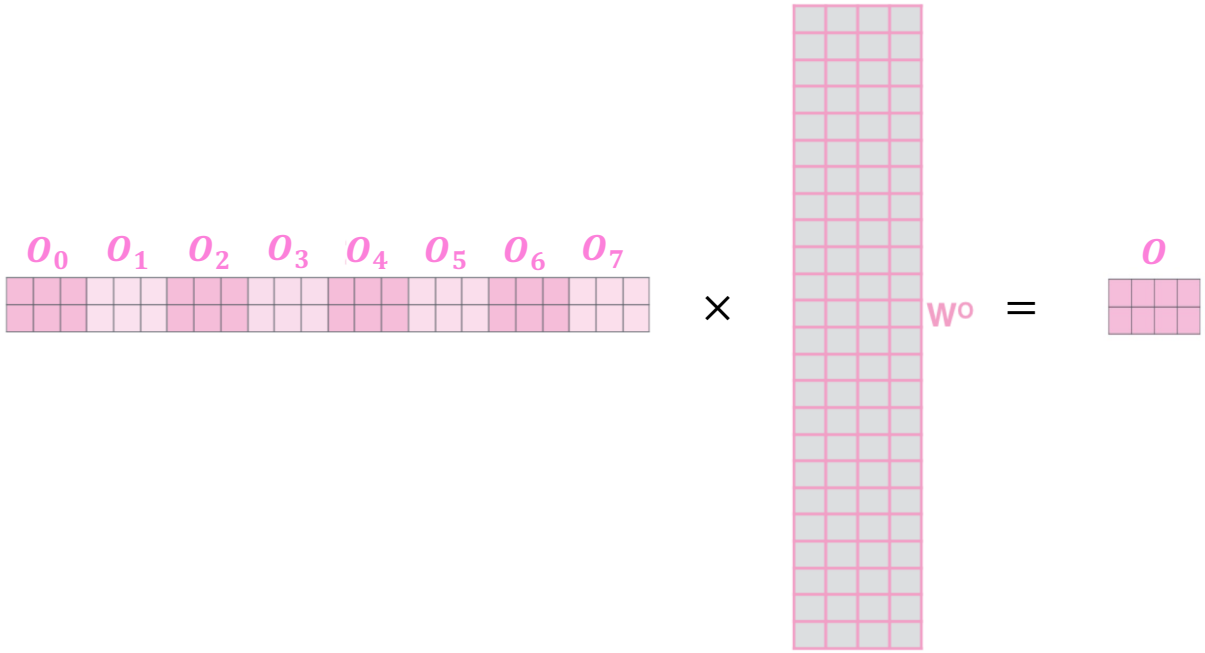


- Each $O_{h_i} \in \mathbb{R}^{t \times d_k}$
- $h_i \in \{0, \ldots, 7\}$, one output for each head
- Recall, model expects vectors of dimension d_{model}
  → Indicates we need to reduce to a single $O \in \mathbb{R}^{t \times d_{model}}$ matrix

Learning Multi-Head Attention

$$MultiHead(Q,K,V) = Concat\left(head_{0,}\dots,head_{h-1}\right)W^{O}$$

where $head_{h_i} = Attention(\mathrm{QW}_{\mathrm{h_i}}^{\mathrm{Q}}, \mathrm{KW}_{\mathrm{h_i}}^{\mathrm{K}}, \mathrm{VW}_{\mathrm{h_i}}^{\mathrm{V}})$



Transformer Architecture

Ways attention is used in the transformer:

- **Self-attention in the encoder**
  - Allows the model to attend to all positions in the previous encoder layer
  - Embeds context about how elements in the sequence relate to one another
- **Masked self-attention in the decoder**
  - Allows the model to attend to all positions in the previous decoder layer up to and including the current position (during auto-regressive process)
  - Prevents forward looking bias by stopping leftward information flow during training
  - Also embeds context about how elements in the sequence relate to one another
- **Encoder-decoder cross-attention**
  - Allows decoder layers to attend all parts of the latent representation produced by the encoder
  - Pulls context from the encoder sequence over to the decoder"

A. Vaswani et al., "Attention is All you Need," in Advances in Neural Information Processing Systems (NeurIPS), 2017.

Transformer Architecture

Why Self-Attention?

- Lower computational complexity
- Greater amount of the computation that can be parallelized
- Each representation encodes the positional information of the sequence

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Transformer Architecture

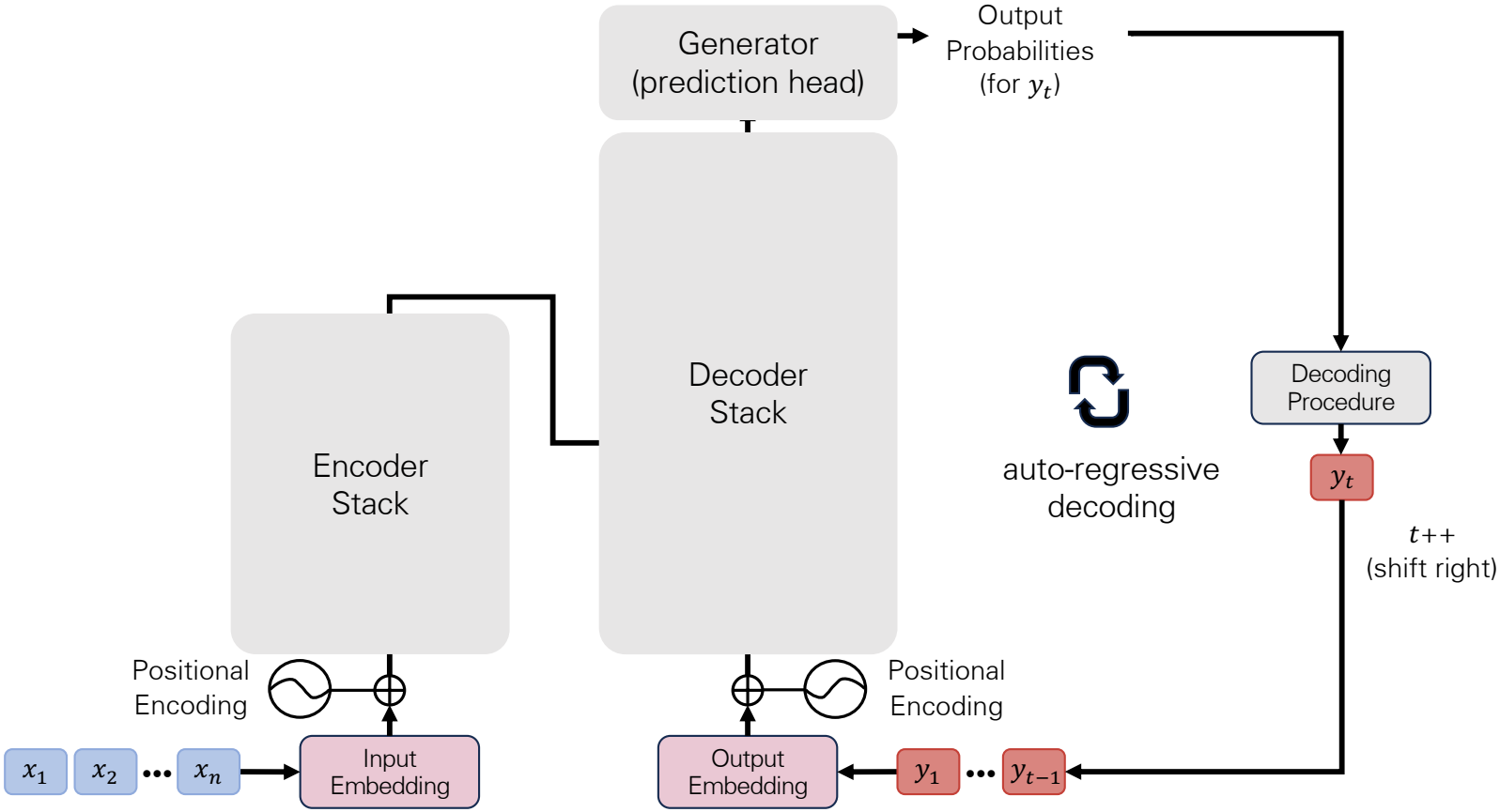
Why Self-Attention?

- Cheaper (more power, less parameters)
- Faster to train

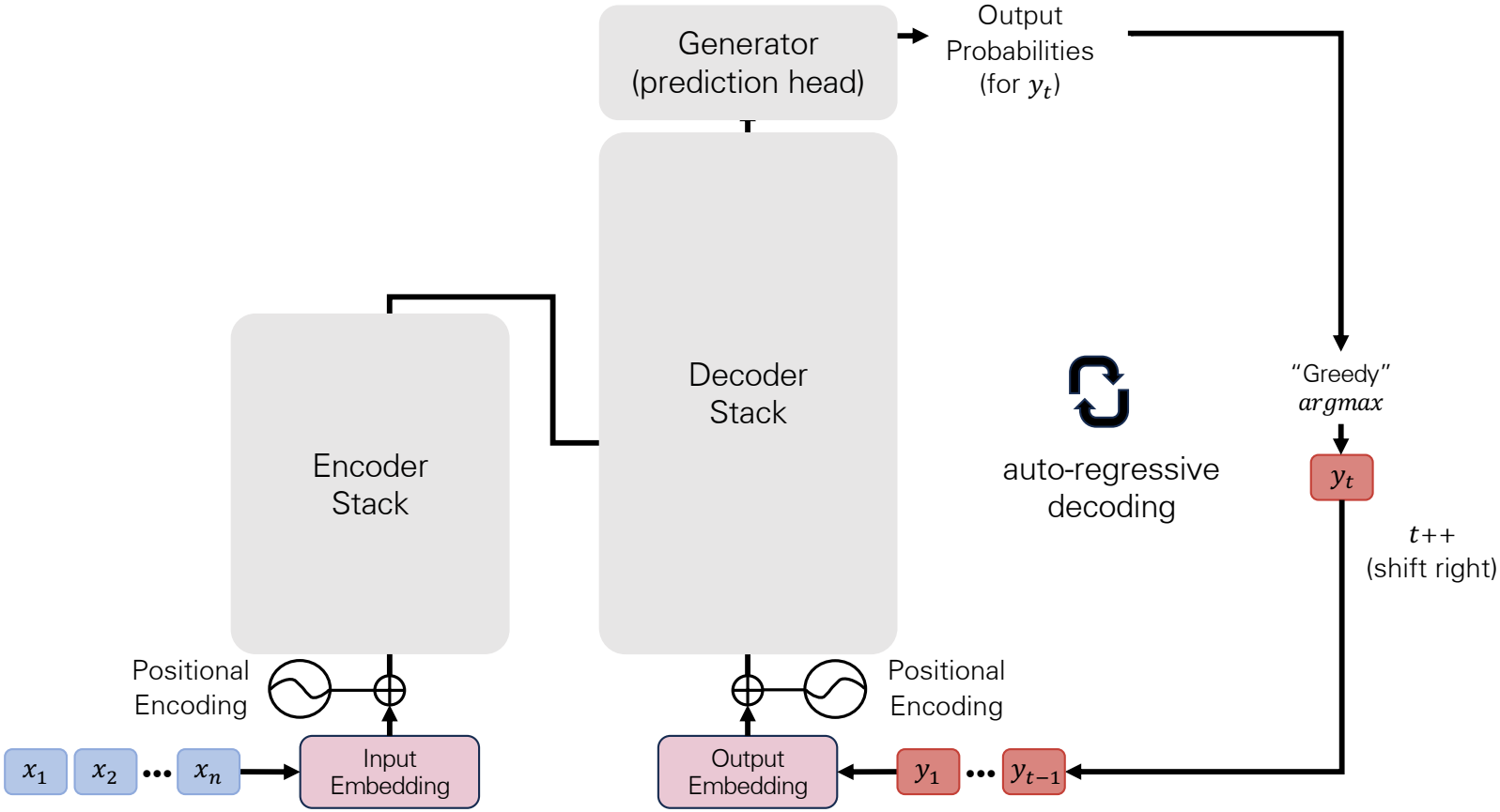
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$		
Convolutional	$O(k \cdot n \cdot d^2)$		
Self-Attention (restricted)	$O(r \cdot n \cdot d)$		

$n < d$ for sequence representations

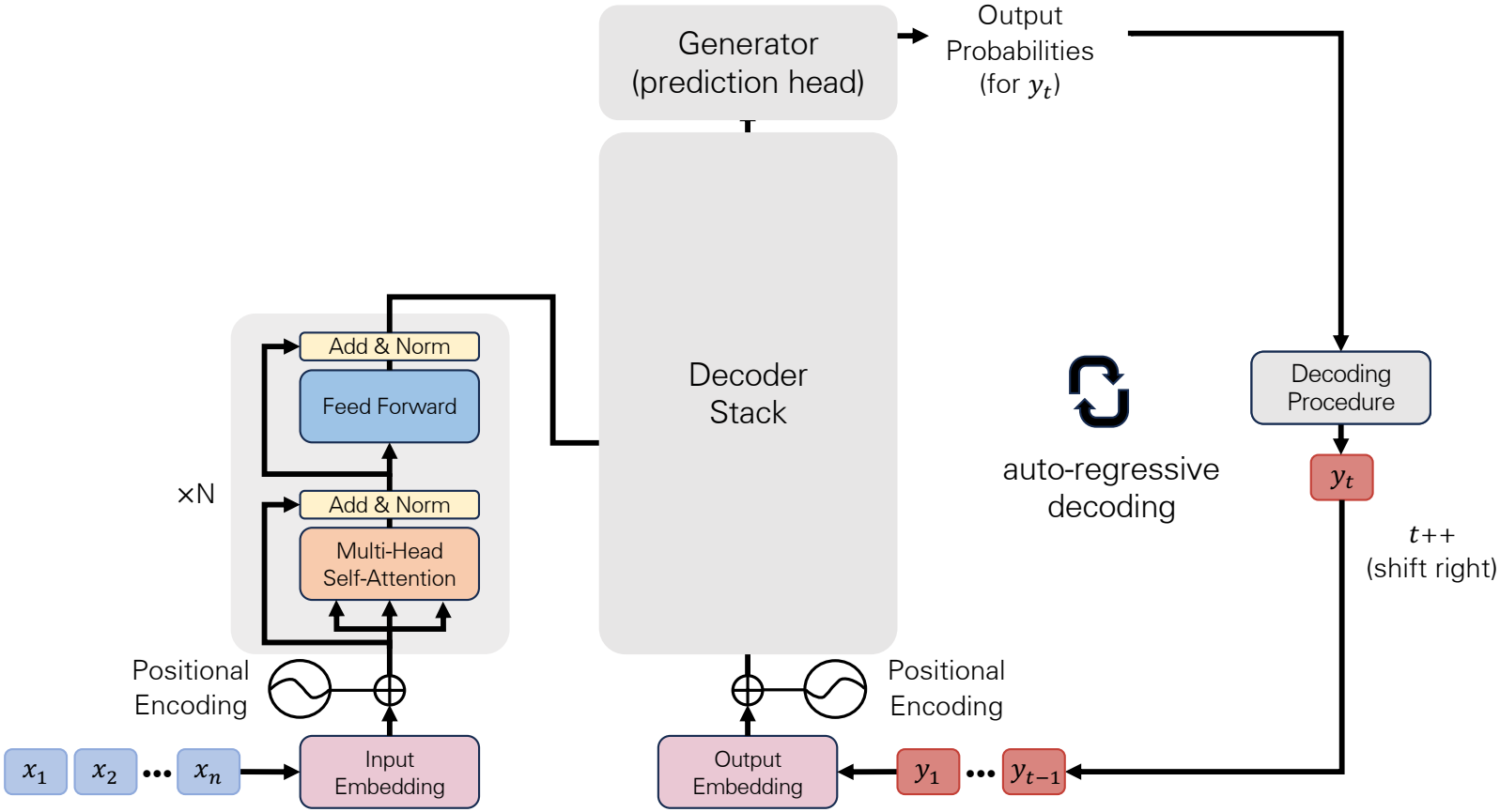
Transformer Architecture



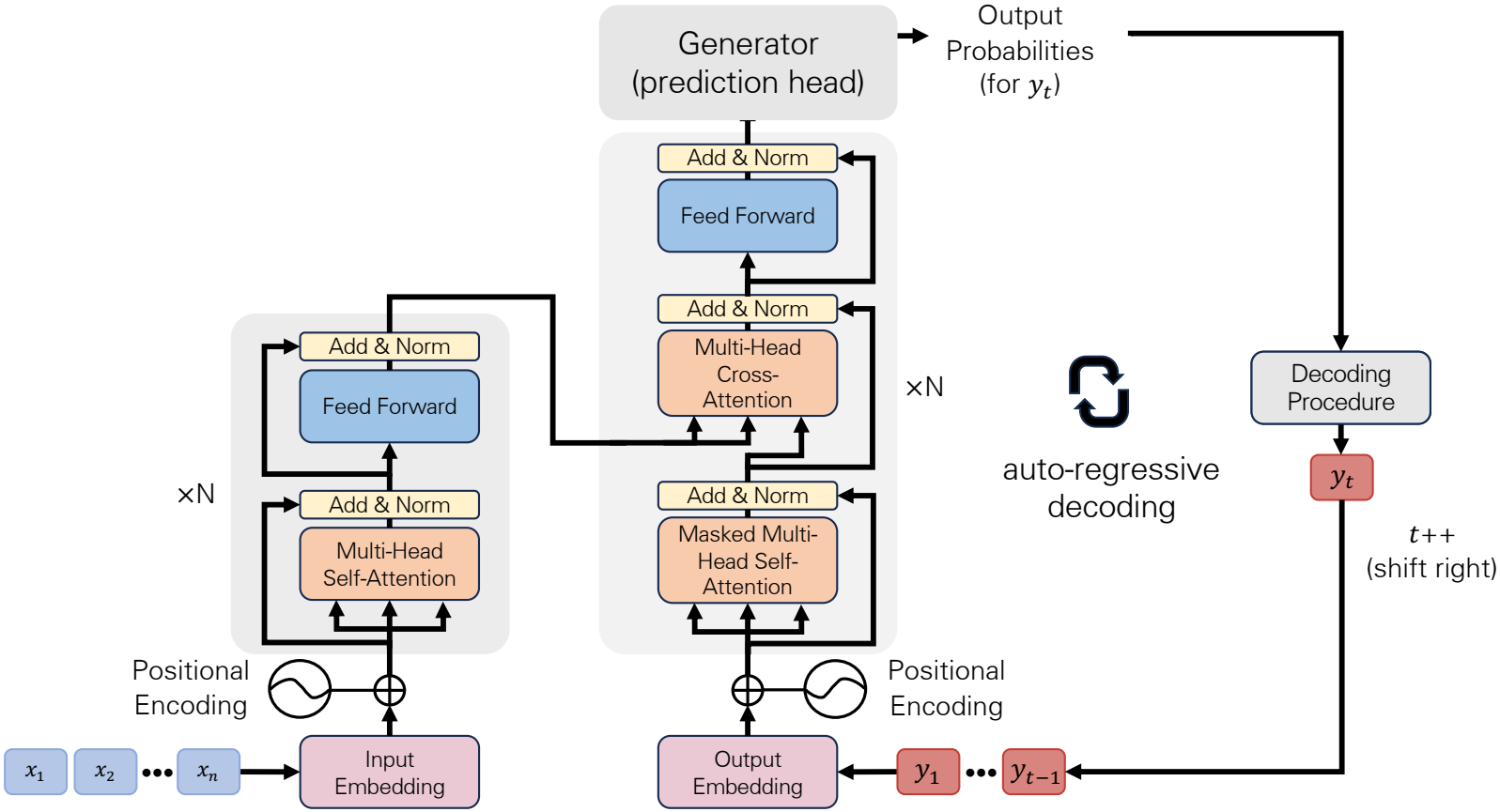
Transformer Architecture



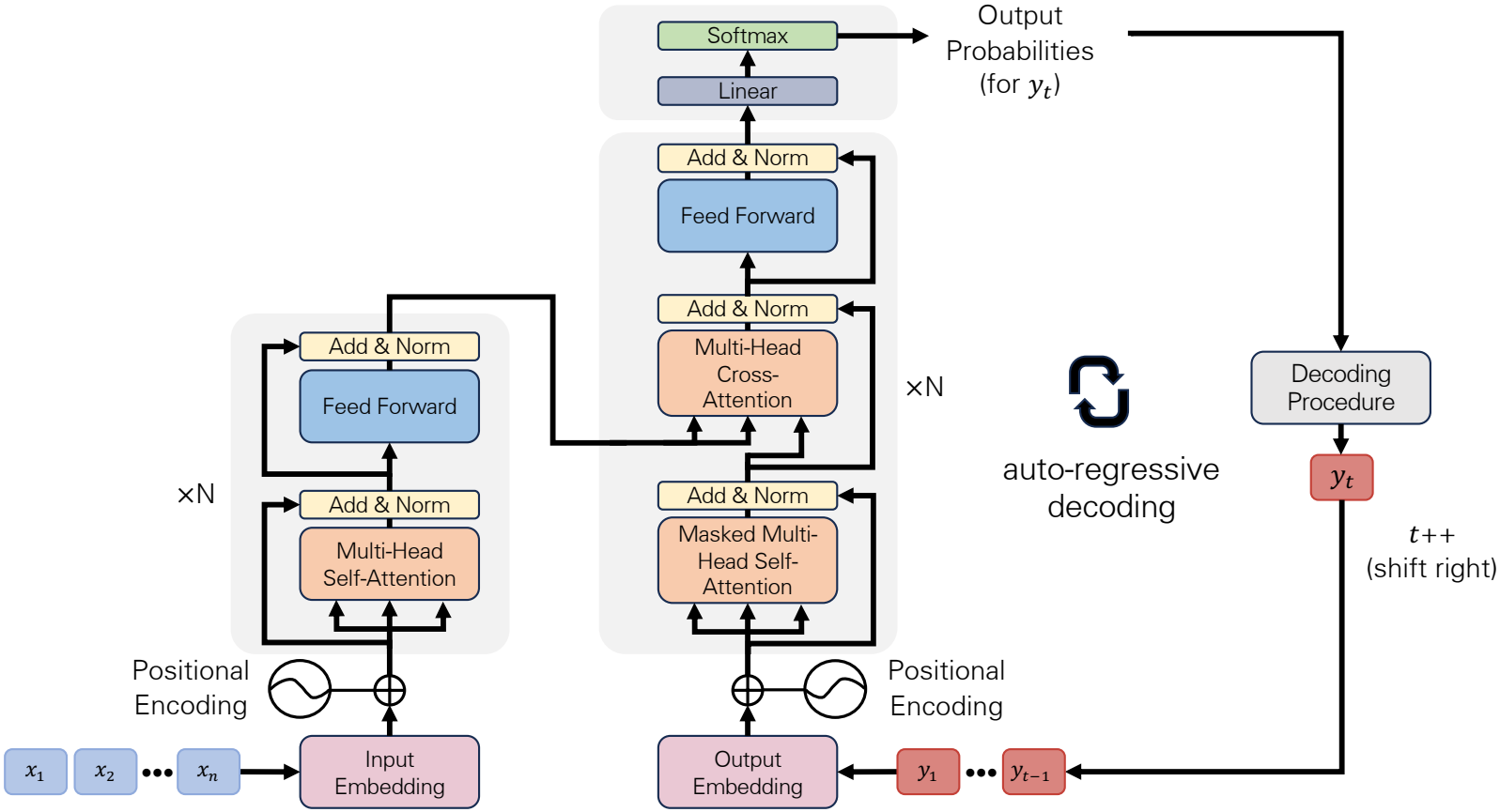
Transformer Architecture



Transformer Architecture

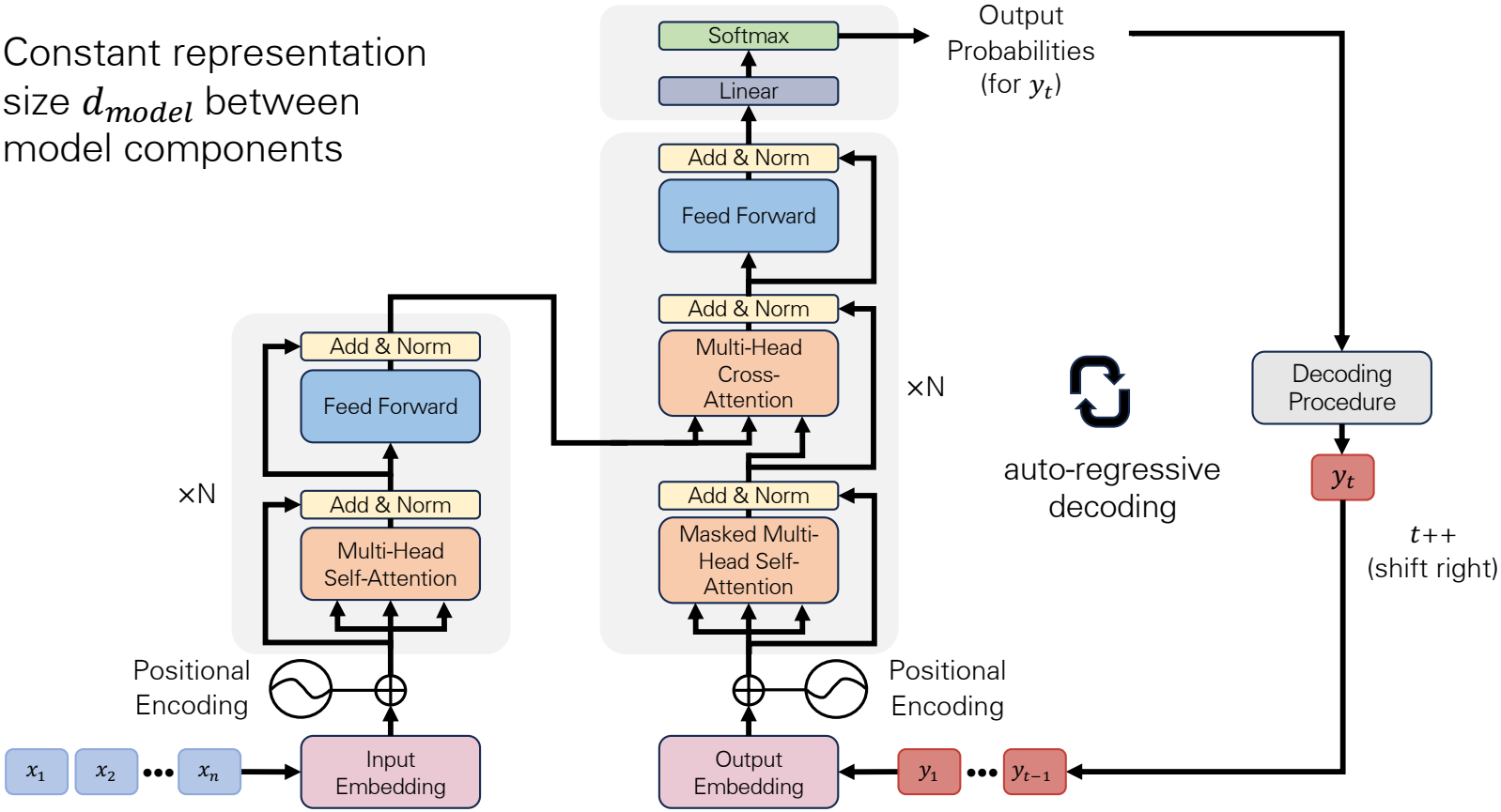


Transformer Architecture



Transformer Architecture

Constant representation size d_{model} between model components



Transformers From The Ground Up

Harvard, “The Annotated Transformer.” [Online]. Available: https://nlp.seas.harvard.edu/annotated-transformer/

Model Creation Helper

Transformers From The Ground Up

Clones Helper function

- What?
  - Create N copies of `pytorch nn.Module`
- Why?
  - The Transformer's structure contains a lot of design repetition (like VGG)

```
def clones(module, N):
    "Produce N identical layers."
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
```

Remember these clones shouldn't share parameters (for the most part)

Make sure to initialize all model parameters to keep clones independent

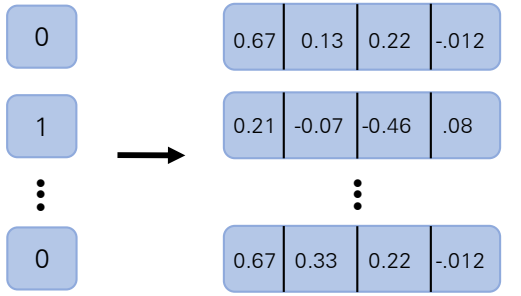
Getting Data into the Transformer (1)

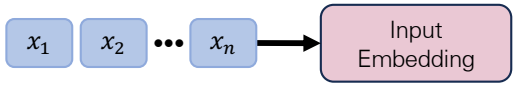
Transformers From The Ground Up

Creating Embeddings

- What?
  - Create vector representation of sequence vocabulary
- Why?
  - Can be computed on by neural architecture
  - Dimensionality usually reduced
    - ~37,000 words → 512 in paper

    → More efficient computation
- How?
  - Learned mapping (linear projection)





Getting Data into the Transformer (1)

Transformers From The Ground Up

Implementing Embeddings

```
class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super(Embeddings, self).__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model

    def forward(self, x):
        return self.lut(x) * math.sqrt(self.d_model)
```

`nn.Embedding` creates a lookup table to map sequence vocabulary to unique vectors

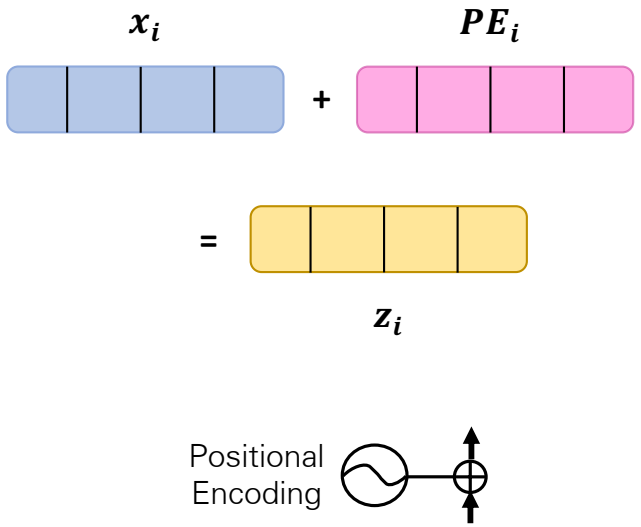
Uses learned weights to handle this mapping (essentially a `nn.Linear`)

Getting Data into the Transformer (2)

Transformers From The Ground Up

Positional encoding

- What?
  - Add information about an element's position in a sequence to its representation
- Why?
  - Removes need for recurrence or convolution
- How?
  - Element wise addition of sinusoidal encoding

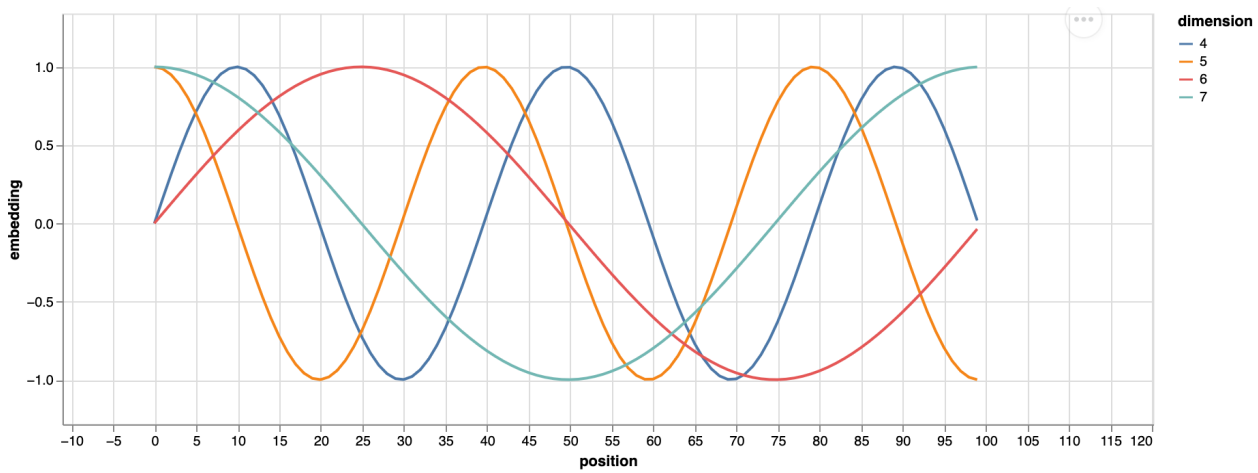


Getting Data into the Transformer (2)

Transformers From The Ground Up

Sinusoidal positional encoding

"May allow the model to easily learn to attend by relative positions"



$$PE(i, 2_l) = sin\left(\frac{i}{10000^{\frac{2_l}{d_{model}}}}\right)$$

$$PE(i, 2_{l+1}) = cos\left(\frac{i}{10000^{\frac{2_l}{d_{model}}}}\right)$$

$$i \in \{1, \dots, N\}, l \in \{1, \dots, d_{model}\}$$

Getting Data into the Transformer (2)

Transformers From The Ground Up

Implementing sinusoidal positional encoding

```
class PositionalEncoding(nn.Module):
    "Implement the PE function."

    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(
            torch.arange(0, d_model, 2) * -(math.log(10000.0) / d_model)
        )
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer("pe", pe)

    def forward(self, x):
        x = x + self.pe[:, : x.size(1)].requires_grad_(False)
        return self.dropout(x)
```

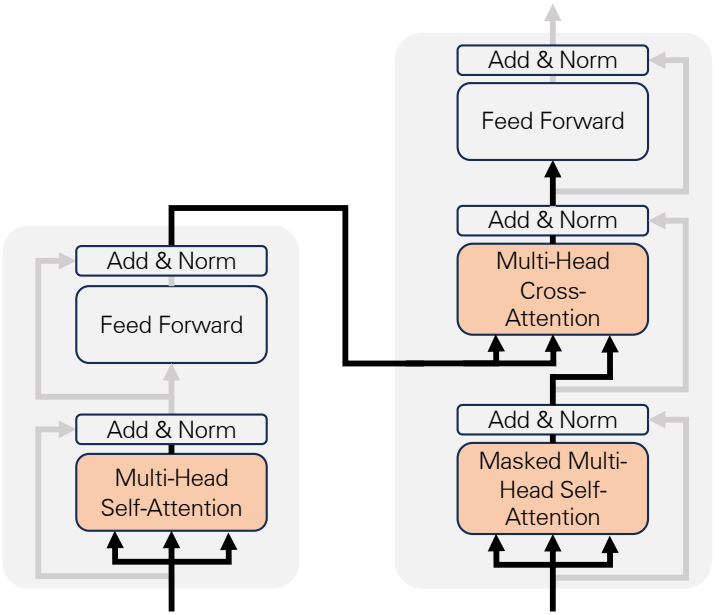
- Know d_{model} at model creation time, so precompute positional encoding
- Dim is consistent with x, so we use in-place addition to add positional context to x

Encoder-Decoder Sublayers (1)

Transformers From The Ground Up

Multi-Head Attention Sublayers

- What?
  - Carries out multi-head attention and learns weights for creating keys, values, and queries
- Why?
  - To extract relevant context from input sequence
  - Multiple heads provide greater resolution
    - Attend to different sub-representations
- How?
  - Implemented as previously discussed



Encoder-Decoder Sublayers (1)

Transformers From The Ground Up

Implementing Multi-Head Attention

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

Encoder-Decoder Sublayers (1)

Transformers From The Ground Up

Implementing Multi-Head Attention

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)
```

```
def forward(self, query, key, value, mask=None):
    "Implements Figure 2"
    if mask is not None:
        # Same mask applied to all h heads.
        mask = mask.unsqueeze(1)
    nbatches = query.size(0)

    # 1) Do all the linear projections in batch from d_model => h x d_k
    query, key, value = [
        lin(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
        for lin, x in zip(self.linears, (query, key, value))
    ]

    # 2) Apply attention on all the projected vectors in batch.
    x, self.attn = attention(
        query, key, value, mask=mask, dropout=self.dropout
    )

    # 3) "Concat" using a view and apply a final linear.
    x = (
        x.transpose(1, 2)
        .contiguous()
        .view(nbatches, -1, self.h * self.d_k)
    )
    del query
    del key
    del value
    return self.linears[-1](x)
```

Encoder-Decoder Sublayers (2)

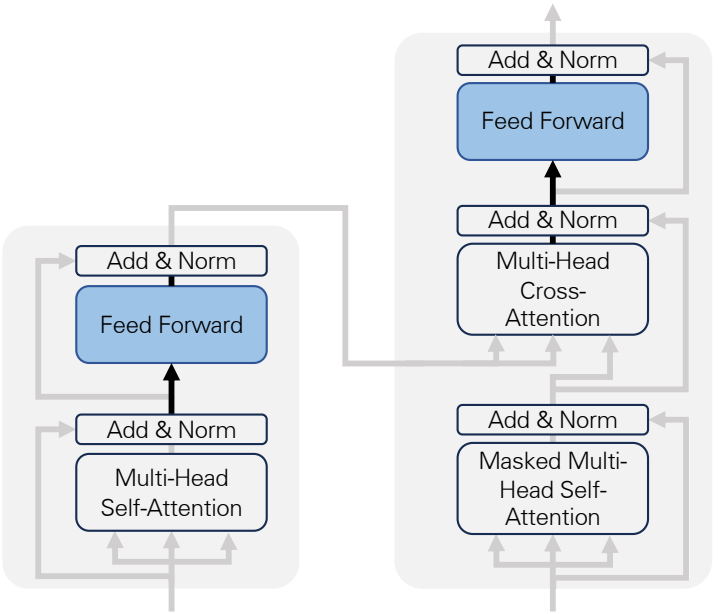
Transformers From The Ground Up

Position-wise Feed Forward Network

- What?
  - Applies learned transformations to each position in input representation
    - Applied separately and identically
- Why?
  - Exploits context added by previous sublayers
  - Adds depth to network so it can approximate greater complexity
  - Increases resolution to pull out different parts of the superposition
- How?
  - Linear MLP (FC) layers with ReLU activation in between
  - Hidden space with higher dimension

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

$$W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}, W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$$



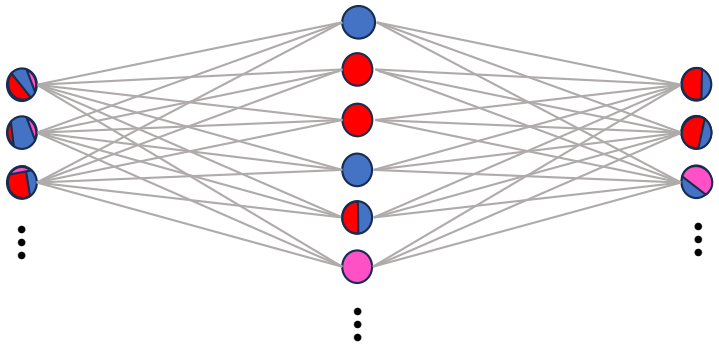
Encoder-Decoder Sublayers (2)

Transformers From The Ground Up

Implementing position-wise Feed Forward Network

```
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."

    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(self.w_1(x).relu()))
```

- $d_{ff} = 2048 = 4\, d_{model}$



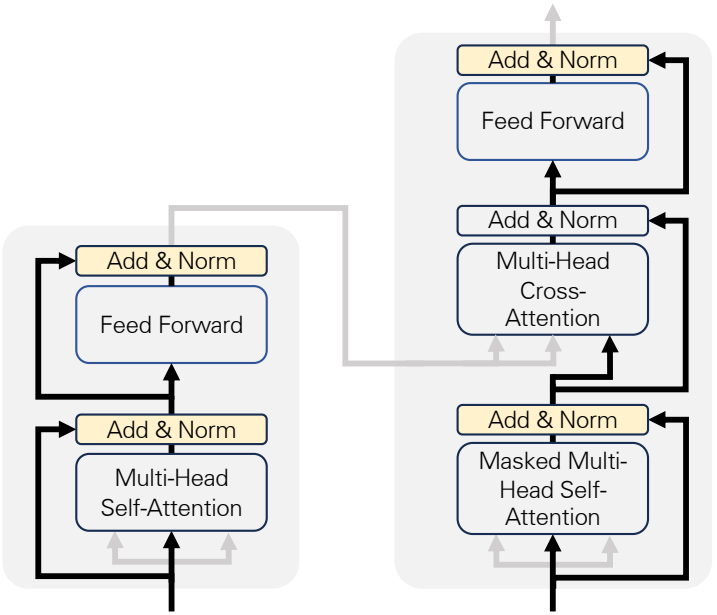
Encoder-Decoder Sublayers (3)

Transformers From The Ground Up

Sublayer connections

- Residual connection (recall resnet)
  - Can be less expensive to learn residuals
  - Elevates vanishing gradient
  - Preserves more of the input signal through skip connection
- Dropout (recall resnet)
  - Regularizes model (combats overfitting)
  - Encourages diversity of attention heads
- LayerNorm
  - Combats vanishing gradient
  - Combats exploding gradient

$$LayerNorm(x + Dropout(Sublayer(x)))$$



Encoder-Decoder Sublayers (3)

Transformers From The Ground Up

Implementing sublayer connections

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."

    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

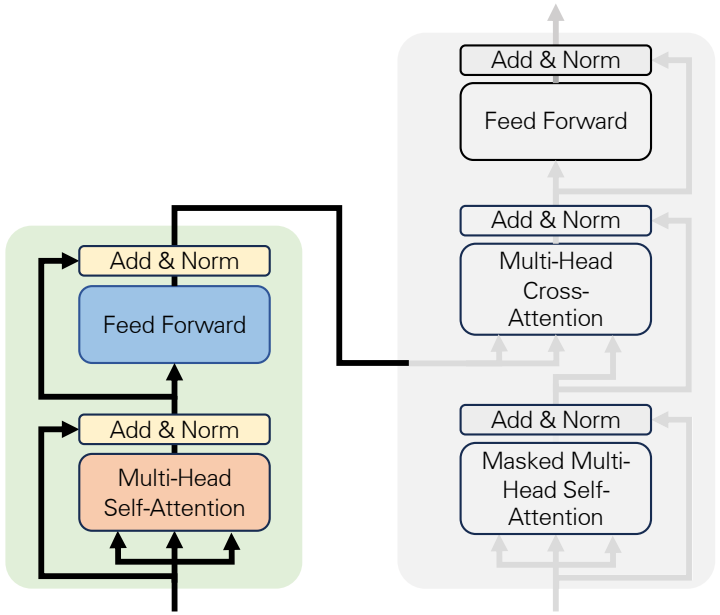
```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """

    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

Encoder-Decoder Layers (1)

Transformers From The Ground Up

Encoder Layer

- What?
  - Composable blocks for the task of encoding an input sequence representation with attention
- Why?
  - Easy construction of model
  - Allows encoder layers to be stacked to achieve depth
  - Repeating Multi-head attention
    → Model more complex position interactions
- How?
  - Multi-head self-attention (8 heads used) sublayer
  - Position-wise feed forward network sublayer
  - All sublayers are surrounded by sublayer connections

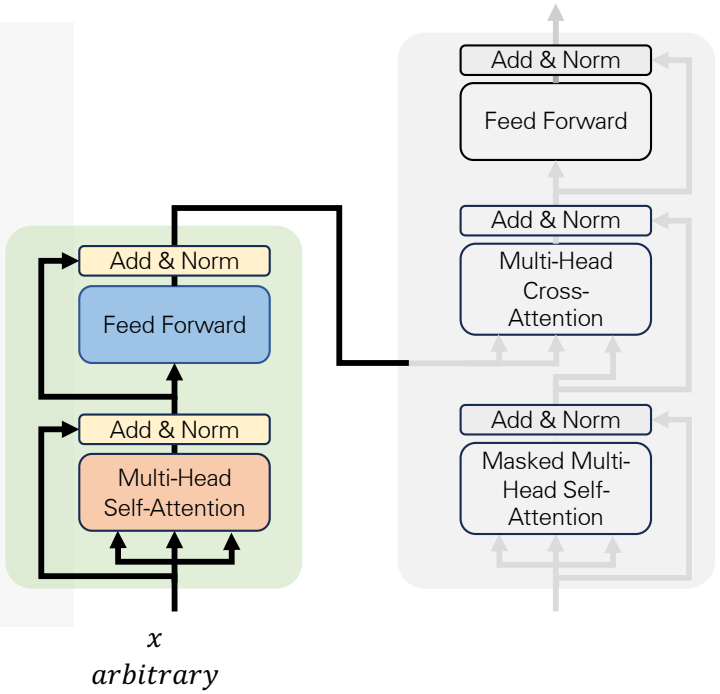


Encoder-Decoder Layers (1)

Transformers From The Ground Up

Implementing the encoder layer

```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"

    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

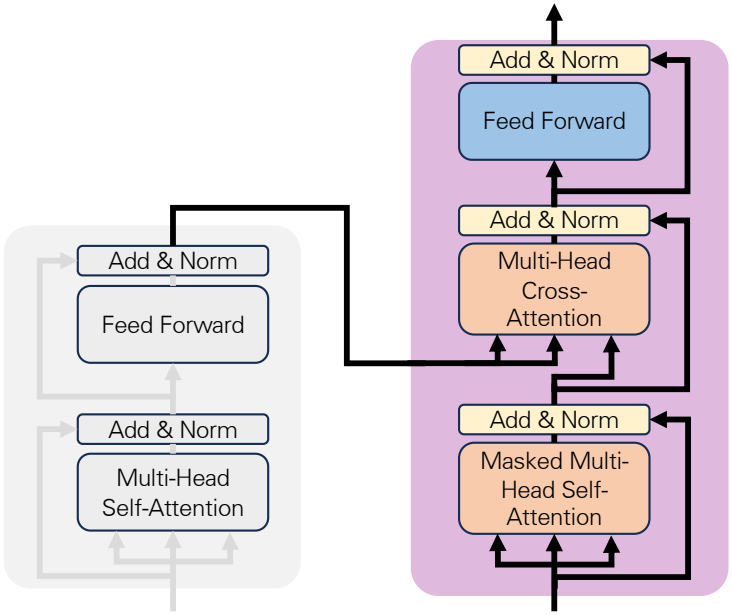


Encoder-Decoder Layers (2)

Transformers From The Ground Up

Decoder Layer

- What?
  - Composable blocks for the task of decoding a target sequence auto-regressively
- Same as encoder layers other than:
  1. the additional multi-head attention block to preform cross-attention with the output representation from the encoder
  2. the addition of masking in self-attention
     →This prevents cheating(forward looking bias)
     → Model purely attends to past info

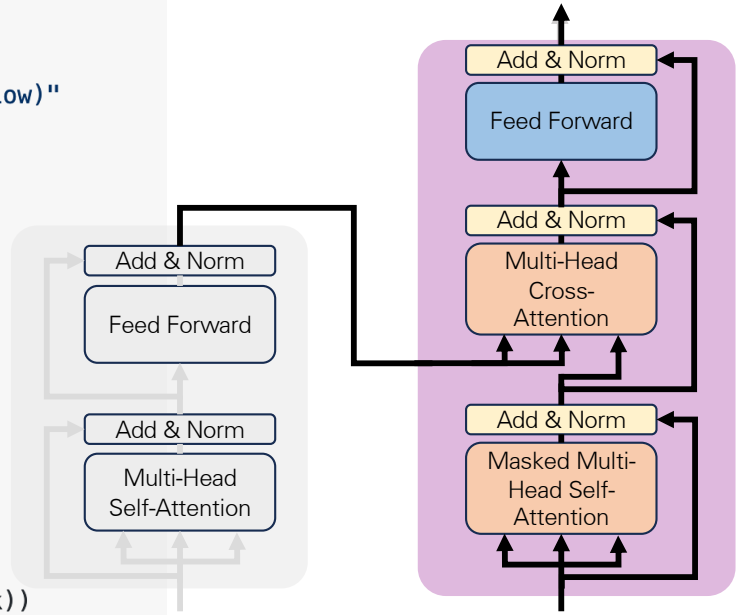


Encoder-Decoder Layers (2)

Transformers From The Ground Up

Implementing the decoder layer

```
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"

    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.sublayer[2](x, self.feed_forward)
```



The Prediction Head

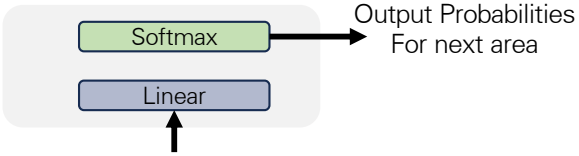
Transformers From The Ground Up

Generator

- Sometimes referred to as the predictor

- A final linear mapping
  - Internal Representation -> logits that capture maximum likelihood of next element in sequence
  - In seq2seq language translation this maps back to vocab corpora

- Apply softmax to convert logits to probabilities



The Prediction Head

Transformers From The Ground Up

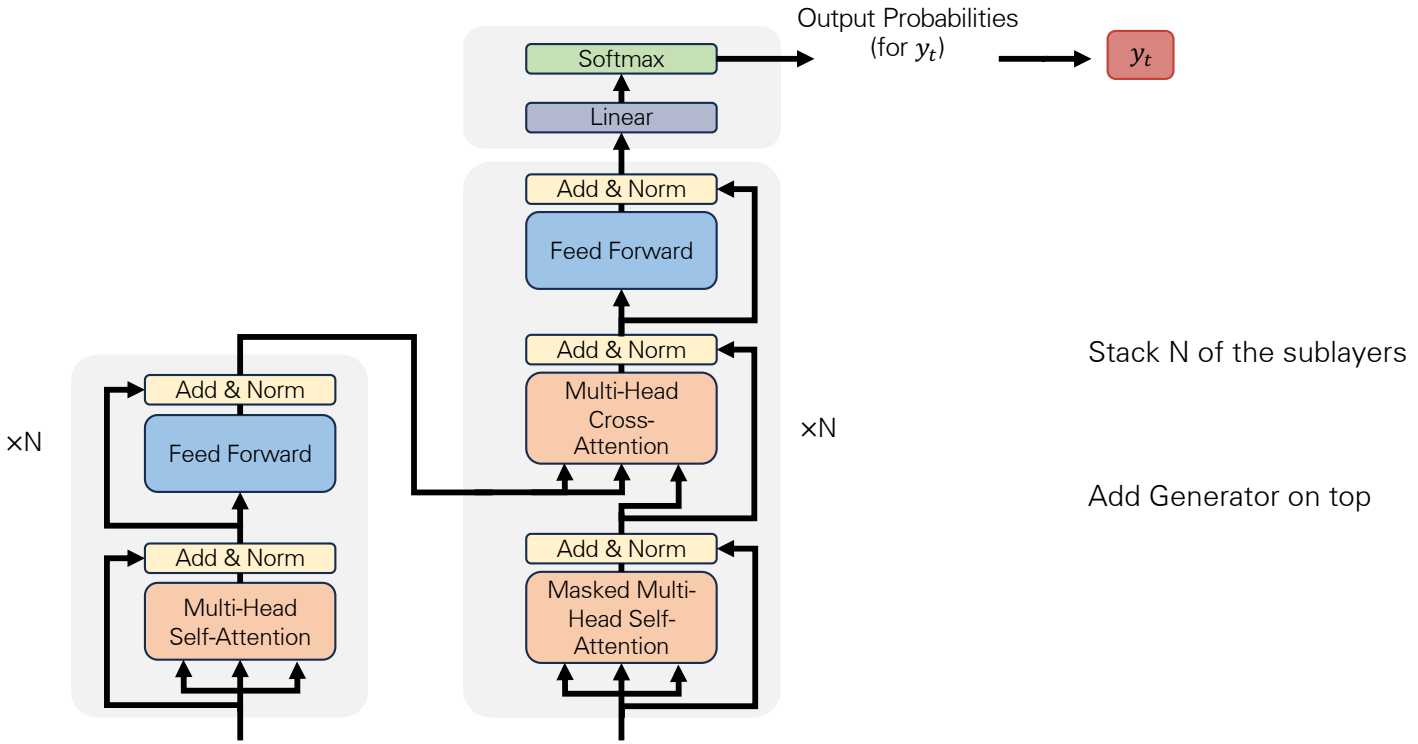
Implementing a generator

```
class Generator(nn.Module):
    "Define standard linear + softmax generation step."

    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        return log_softmax(self.proj(x), dim=-1)
```

Assembling the Encoder-Decoder

Transformers From The Ground Up

- Encoder-Decoder



Assembling the Encoder-Decoder

Transformers From The Ground Up

- Encoder-Decoder implementation

```
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"

    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

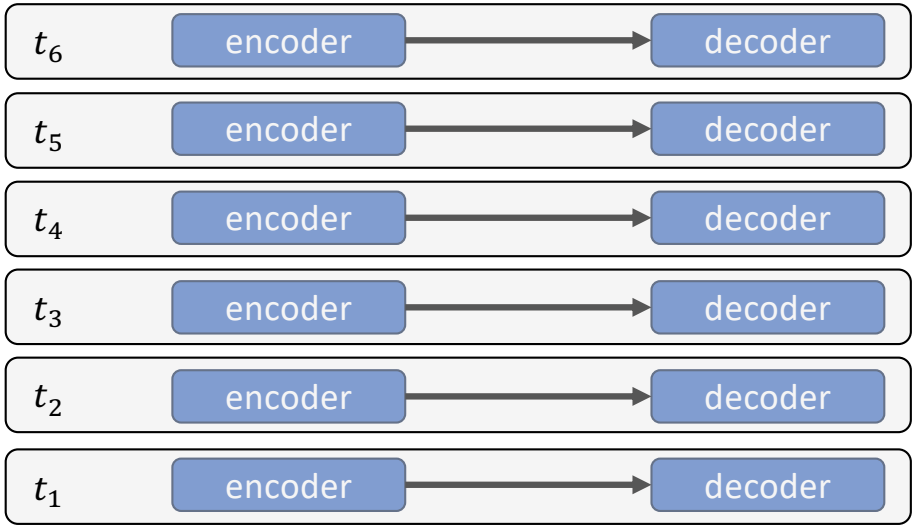
```
class Decoder(nn.Module):
    "Generic N layer decoder with masking."

    def __init__(self, layer, N):
        super(Decoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, memory, src_mask, tgt_mask):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x)
```

Misconceptions about Transformers (2)

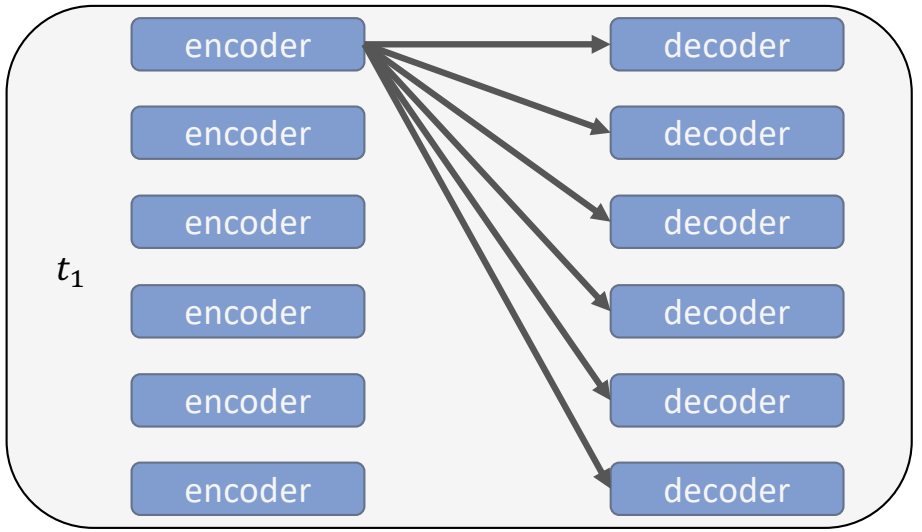
- What?
  - Notion of a whole "transformer block" that is stackable in the vanilla transformer architecture
  - Incorrect belief that encoder-decoder attention connection is layer wise
- Why?
  - Incorrect understanding of stacking layers
  - Pervasive amount of bad figures

Misconceptions about Transformers (2)

Incorrect



Correct



Assembling the Encoder-Decoder

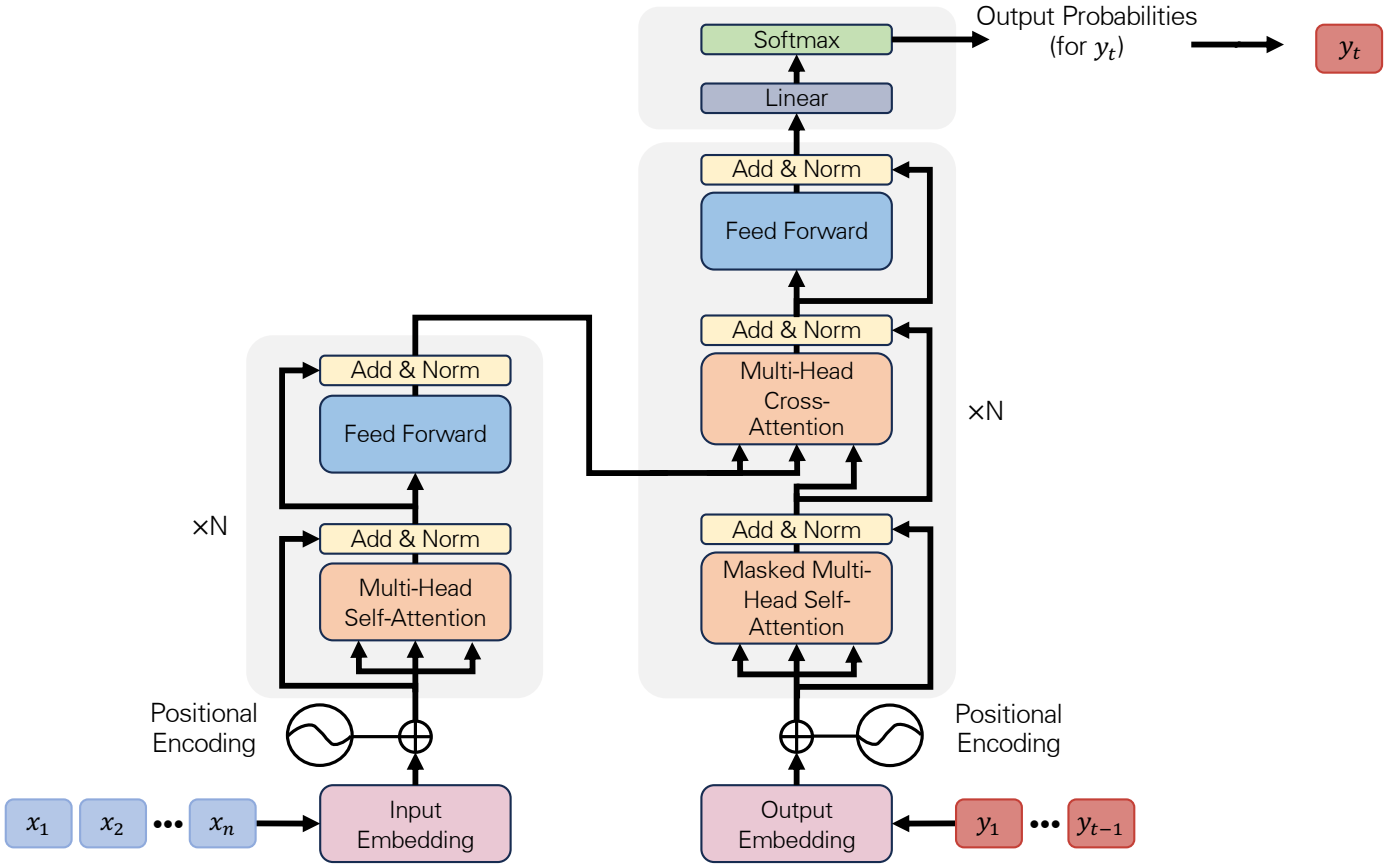
Transformers From The Ground Up

- Encoder-Decoder implementation

```
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture. Base for this and many
    other models.
    """

    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src, src_mask), src_mask, tgt, tgt_mask)

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```

Putting it all together

Transformers From The Ground Up



Putting it all together

Transformers From The Ground Up

```
def make_model(
    src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, dropout=0.1
):
    "Helper: Construct a model from hyperparameters."
    c = copy.deepcopy
    attn = MultiHeadedAttention(h, d_model)
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)
    position = PositionalEncoding(d_model, dropout)
    model = EncoderDecoder(
        Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
        Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff), dropout), N),
        nn.Sequential(Embeddings(d_model, src_vocab), c(position)),
        nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
        Generator(d_model, tgt_vocab),
    )

    # This was important from their code.
    # Initialize parameters with Glorot / fan_avg.
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
    return model
```

Training Transformers

- "Architecture alone does not make a model"

Architecture + Training = Model

- A model expresses different properties depending on how it is trained
- Like nature vs. nurture, both impact what the model does
- Training is what influences parameters

Training Transformers

- Models fit to training data
- If shown examples that encourage bidirectional attention, it will learn that
- If shown only examples that require right attention, it may express more unidirectional behavior (won't generalize as well)
- BERT uses large scale pre-training to do this

Training Transformers

- Masked training

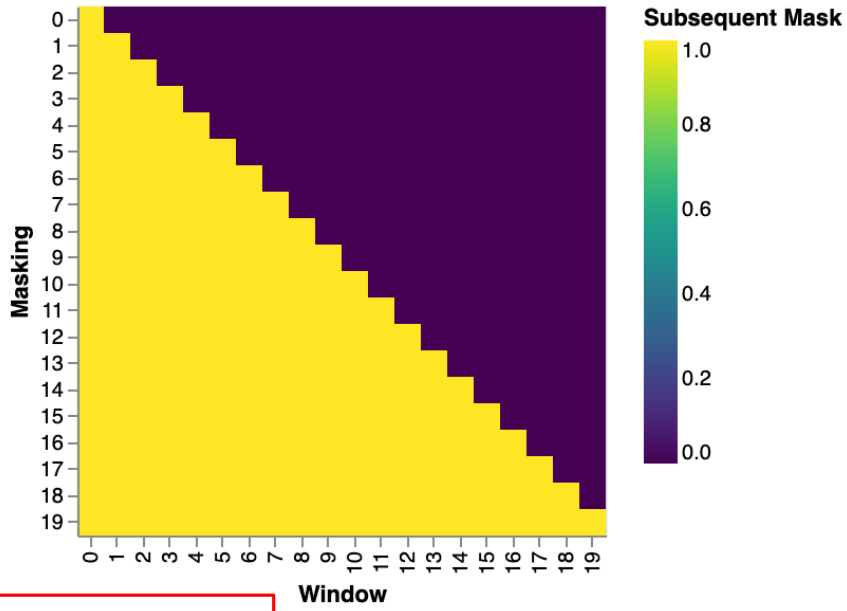
- Attention mechanism can build a masking support directly
- Motivation:
  - Want to prevent the model from learning from future information in the output sequence
- Main idea:
  - Since each decode layer starts with a self-attention block, we can add custom logic to mask out positions in target sequence which it shouldn't see yet
- Implemented as rolling window

Training Transformers

- Masked training

```
def subsequent_mask(size):
    "Mask out subsequent positions."
    attn_shape = (1, size, size)
    subsequent_mask = torch.triu(torch.ones(attn_shape), diagonal=1).type(
        torch.uint8
    )
    return subsequent_mask == 0
```

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = scores.softmax(dim=-1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```



$-1e9$ is very negative, softmax(-1e9) → 0

Results and Impact

Performance

- Experimentation on text translation: (1) EN-DE and (2) EN-FR

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

A. Vaswani et al., "Attention is All you Need," in Advances in Neural Information Processing Systems (NeurIPS), 2017.

Paper Impact

- Highly influential
- Paper has 113,405 citations
- Transformer architecture has been used as the basis for many state-of-the-art models
- Transformer is a fundamental building block of all LLMs (e.g. GPT-4, LLaMA 2, Gemini, etc.)

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

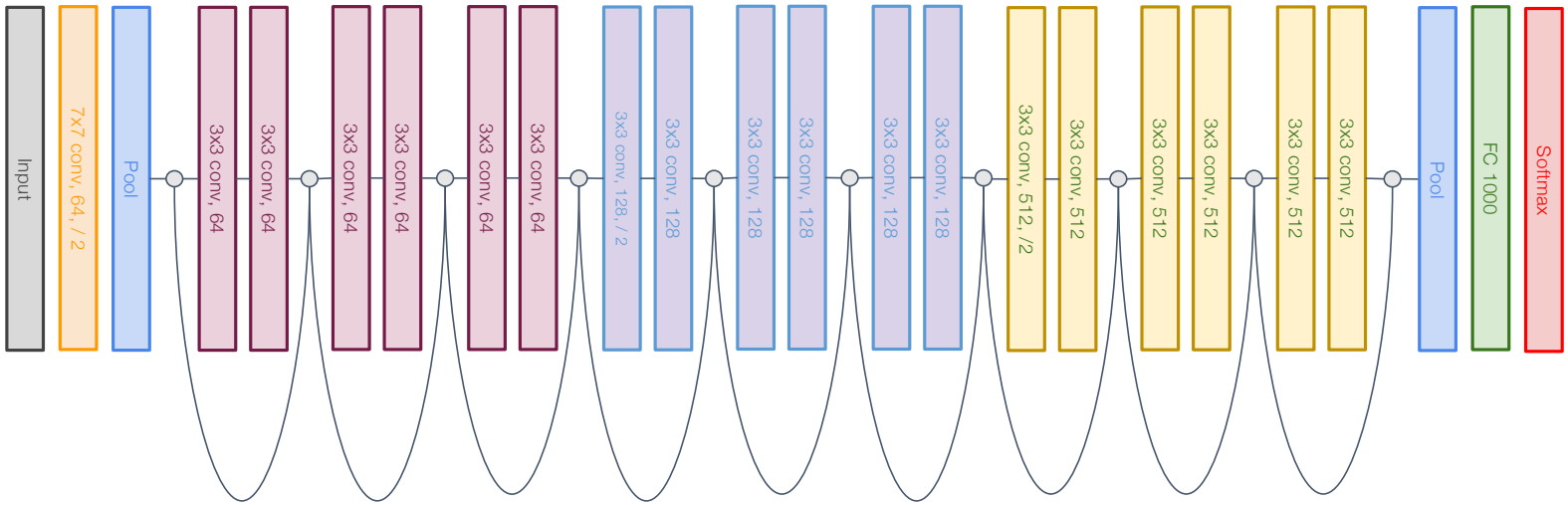
Łukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

How to use Attention / Transformers for Vision?

Idea #1: Add attention to existing CNNs

Start from standard CNN architecture (e.g. ResNet)

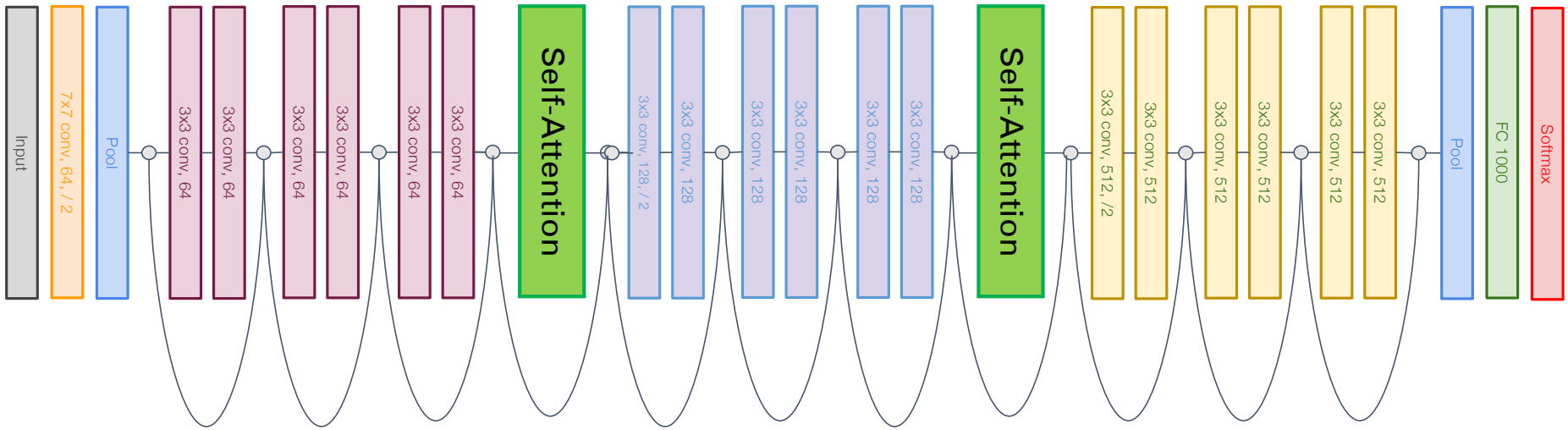


Zhang et al., "Self-Attention Generative Adversarial Networks", ICML 2018
Wang et al., "Non-local Neural Networks", CVPR 2018

Idea #1: Add attention to existing CNNs

Start from standard CNN architecture (e.g. ResNet)

Add Self-Attention blocks between existing ResNet blocks



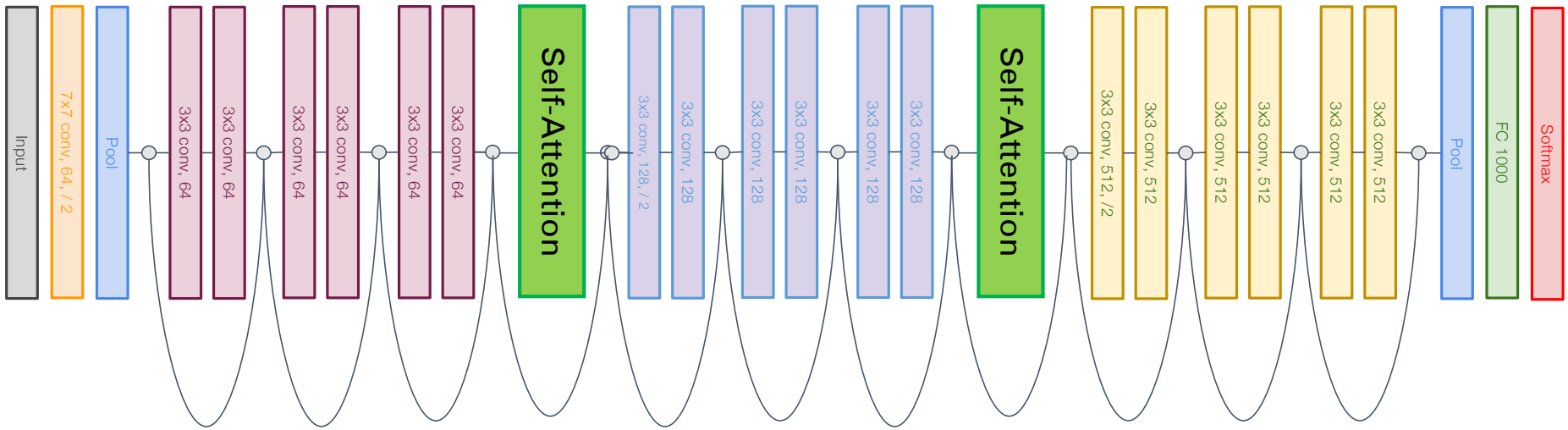
Zhang et al., "Self-Attention Generative Adversarial Networks", ICML 2018
Wang et al., "Non-local Neural Networks", CVPR 2018

Idea #1: Add attention to existing CNNs

Model is still a CNN! Can we replace convolution entirely?

Start from standard CNN architecture (e.g. ResNet)

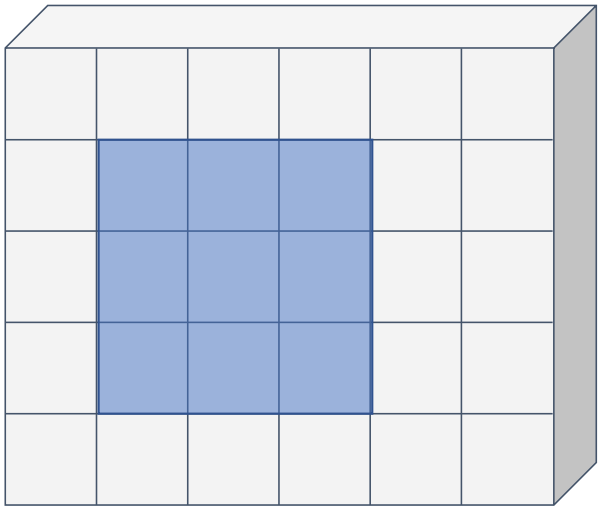
Add Self-Attention blocks between existing ResNet blocks



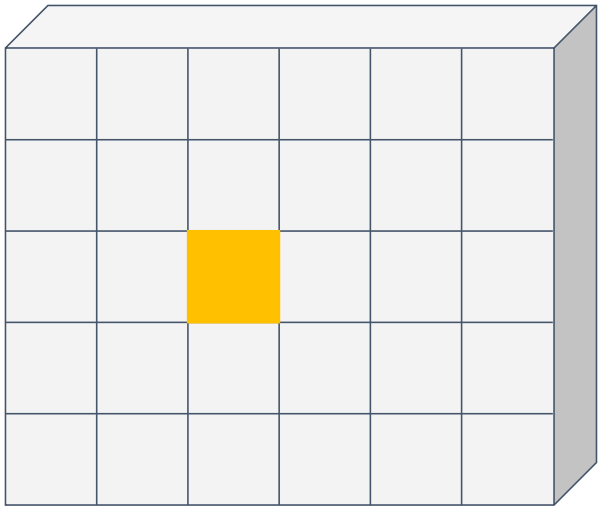
Zhang et al., "Self-Attention Generative Adversarial Networks", ICML 2018
Wang et al., "Non-local Neural Networks", CVPR 2018

Idea #2: Replace Convolution with "Local Attention"

Convolution: Output at each position is inner product of conv kernel with receptive field in input



Input: C x H x W

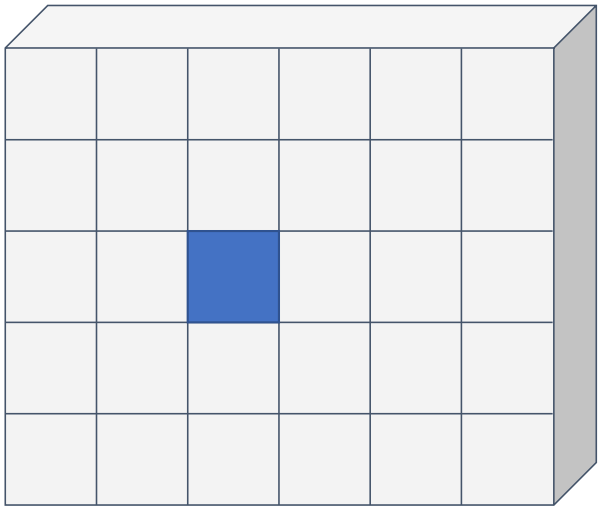


Output: C' x H x W

Hu et al., "Local Relation Networks for Image Recognition", ICCV 2019;
Ramachandran et al., "Stand-Alone Self-Attention in Vision Models", NeurIPS 2019

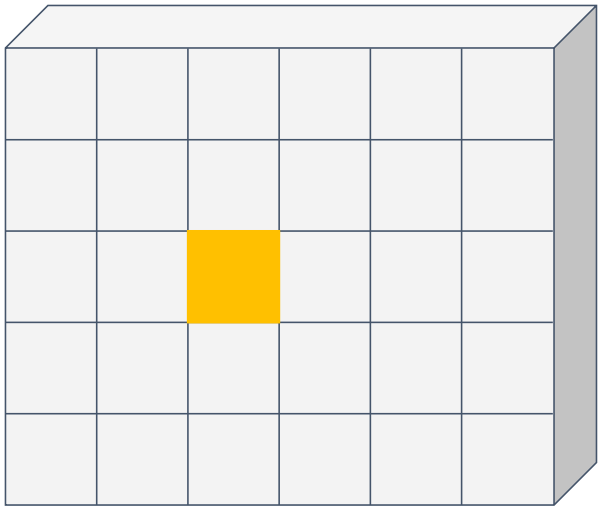
Idea #2: Replace Convolution with "Local Attention"

Map center of receptive field to query



Query: D_Q

Input: C x H x W

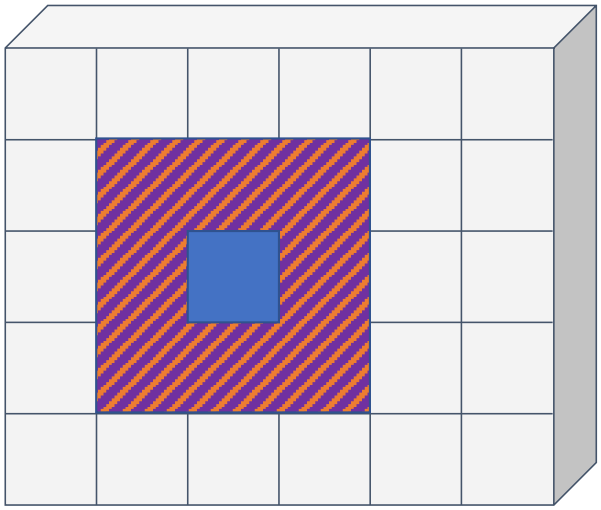


Output: C' x H x W

Hu et al., "Local Relation Networks for Image Recognition", ICCV 2019;
Ramachandran et al., "Stand-Alone Self-Attention in Vision Models", NeurIPS 2019

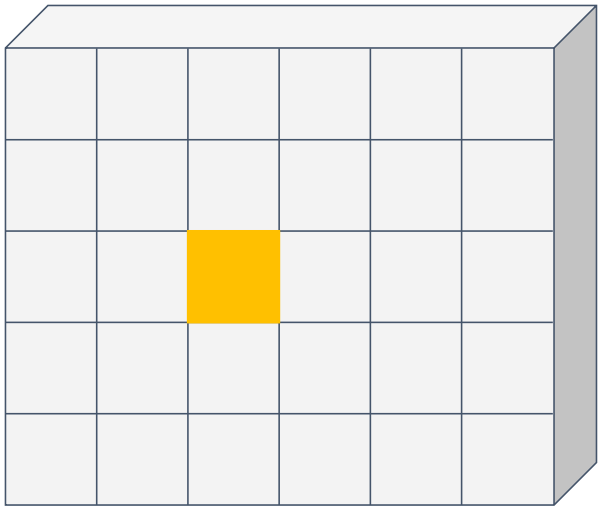
Idea #2: Replace Convolution with "Local Attention"

Map center of receptive field to query
Map each element in receptive field to key and value



Input: C x H x W

Query: D_Q
Keys: R x R x D_Q
Values: R x R x C'

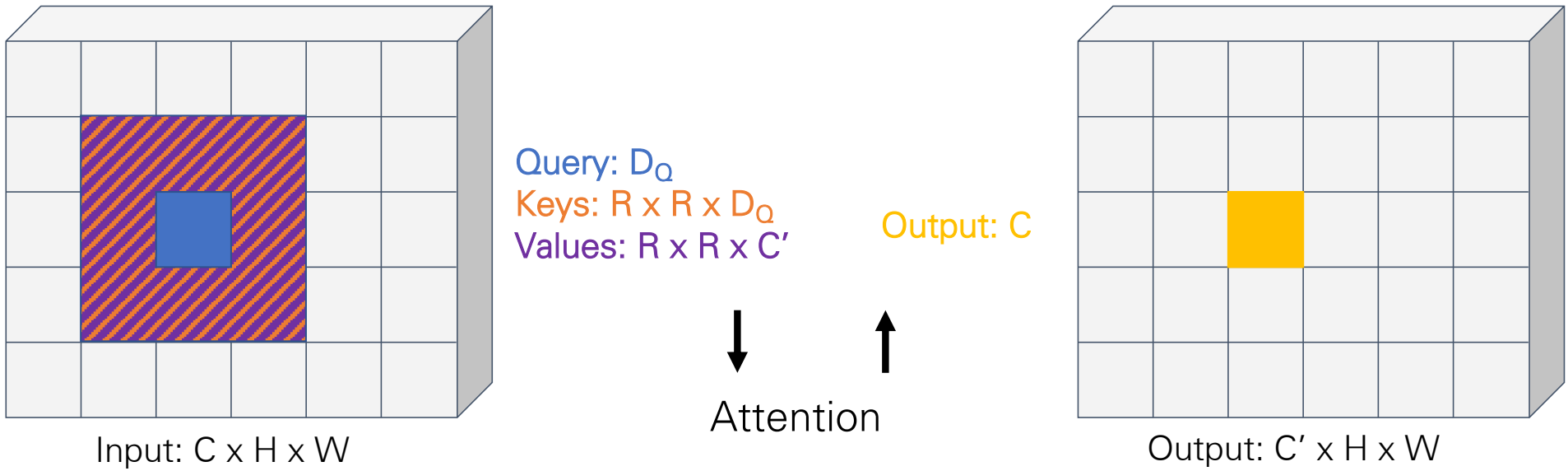


Output: C' x H x W

Hu et al., "Local Relation Networks for Image Recognition", ICCV 2019;
Ramachandran et al., "Stand-Alone Self-Attention in Vision Models", NeurIPS 2019

Idea #2: Replace Convolution with "Local Attention"

Map center of receptive field to query
Map each element in receptive field to key and value
Compute output using attention



Hu et al., "Local Relation Networks for Image Recognition", ICCV 2019;
Ramachandran et al., "Stand-Alone Self-Attention in Vision Models", NeurIPS 2019

Idea #2: Replace Convolution with "Local Attention"

Map center of receptive field to query
Map each element in receptive field to key and value
Compute output using attention
Replace all conv in ResNet with local attention

LR = "Local Relation"

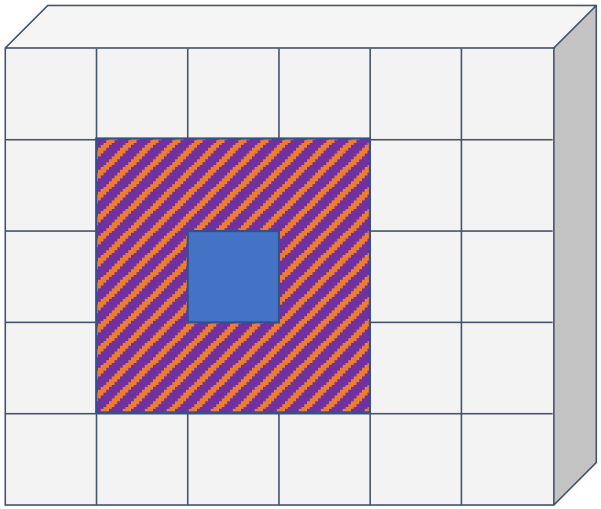
stage	output	ResNet-50	**LR-Net-50 (7×7, m=8)**
res1	112×112	7×7 conv, 64, stride 2	**1×1, 64** **7×7 LR, 64, stride 2**
res2	56×56	3×3 max pool, stride 2	3×3 max pool, stride 2
		[1×1, 64 3×3 conv, 64 1×1, 256] ×3	[1×1, 100 **7×7 LR, 100** 1×1, 256] ×3
res3	28×28	[1×1, 128 3×3 conv, 128 1×1, 512] ×4	[1×1, 200 **7×7 LR, 200** 1×1, 512] ×4
res4	14×14	[1×1, 256 3×3 conv, 256 1×1, 1024] ×6	[1×1, 400 **7×7 LR, 400** 1×1, 1024] ×6
res5	7×7	[1×1, 512 3×3 conv, 512 1×1, 2048] ×3	[1×1, 800 **7×7 LR, 800** 1×1, 2048] ×3
	1×1	global average pool 1000-d fc, softmax	global average pool 1000-d fc, softmax
# params		**25.5**×10^6	**23.3**×10^6
FLOPs		**4.3**×10^9	**4.3**×10^9

Hu et al., "Local Relation Networks for Image Recognition", ICCV 2019;
Ramachandran et al., "Stand-Alone Self-Attention in Vision Models", NeurIPS 2019

Idea #2: Replace Convolution with "Local Attention"

Map center of receptive field to query
Map each element in receptive field to key and value
Compute output using attention
Replace all conv in ResNet with local attention

Lots of tricky details, hard to implement, only marginally better than ResNets

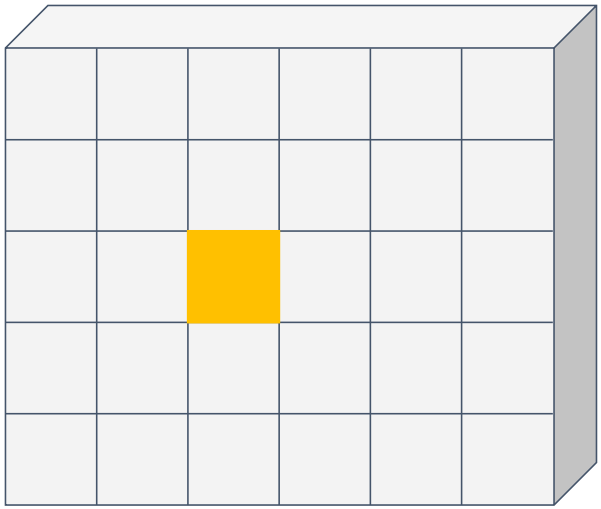


Input: C x H x W

Query: D_Q
Keys: R x R x D_Q
Values: R x R x C'

Attention

Output: C

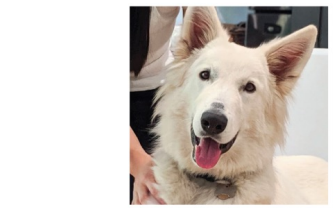


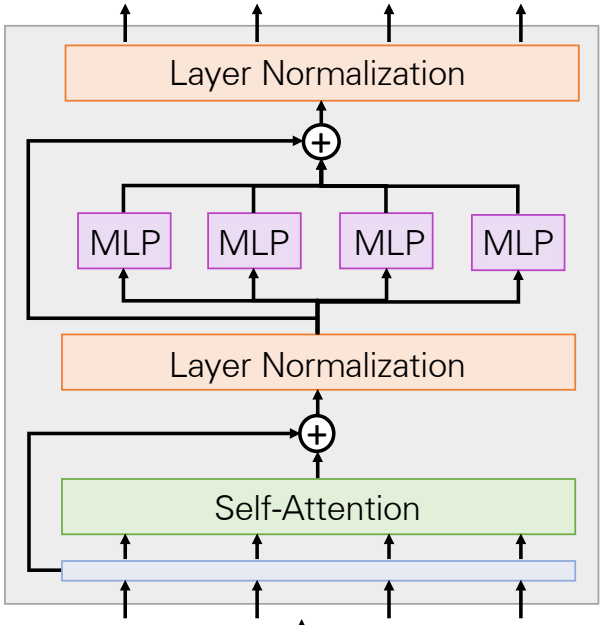
Output: C' x H x W

Hu et al., "Local Relation Networks for Image Recognition", ICCV 2019;
Ramachandran et al., "Stand-Alone Self-Attention in Vision Models", NeurIPS 2019

Idea #3: Standard Transformer on Pixels

Treat an image as a set of pixel values



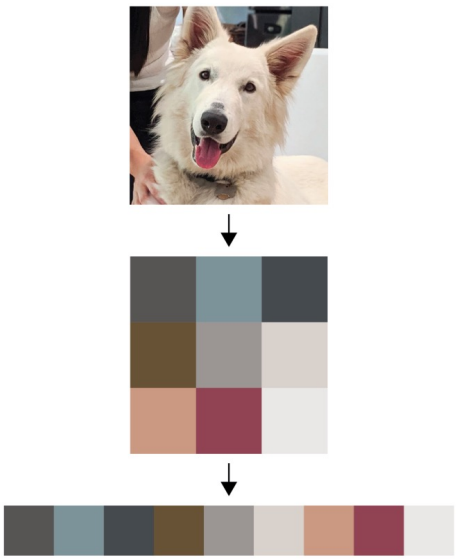


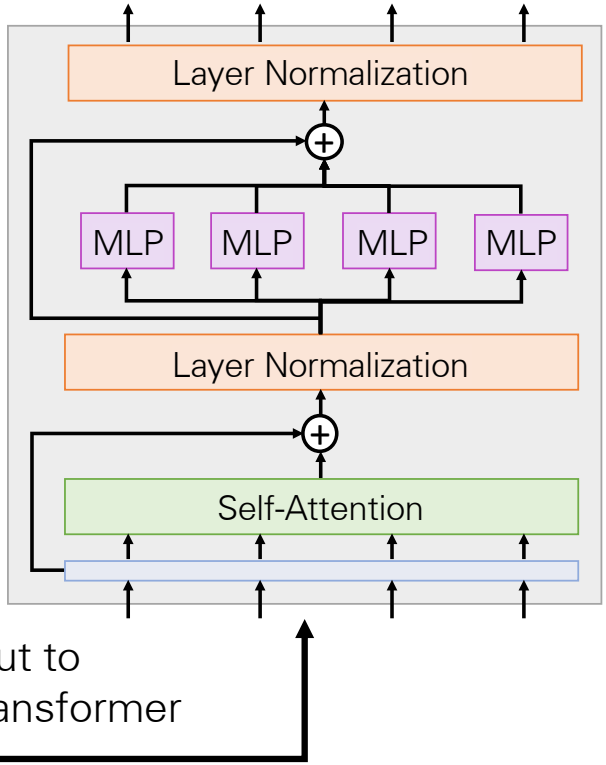
Feed as input to standard Transformer

Chen et al., "Generative Pretraining from Pixels", ICML 2020

Idea #3: Standard Transformer on Pixels

Treat an image as a set of pixel values





Feed as input to standard Transformer

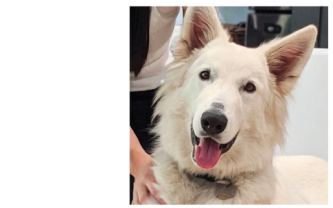
Problem: Memory use!

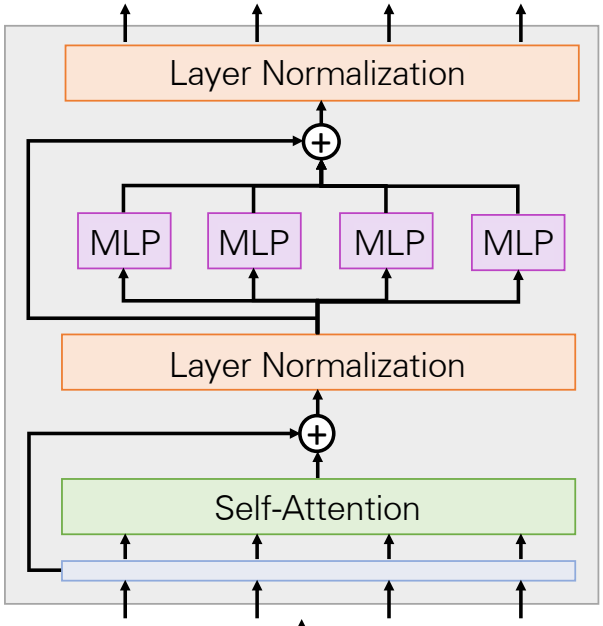
R x R image needs R^4 elements per attention matrix

Chen et al., "Generative Pretraining from Pixels", ICML 2020

Idea #3: Standard Transformer on Pixels

Treat an image as a set of pixel values





Feed as input to standard Transformer

Problem: Memory use!

R x R image needs R^4 elements per attention matrix

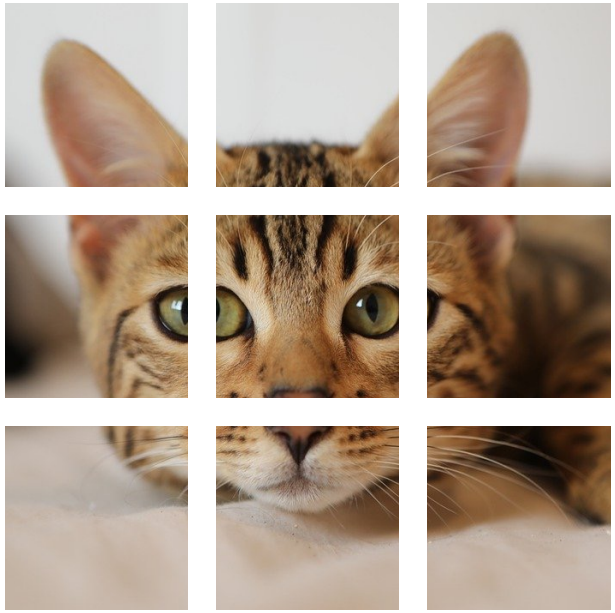
R=128, 48 layers, 16 heads per layer takes 768GB of memory for attention matrices for a single example…

Chen et al., "Generative Pretraining from Pixels", ICML 2020

Idea #4: Standard Transformer on Patches

Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

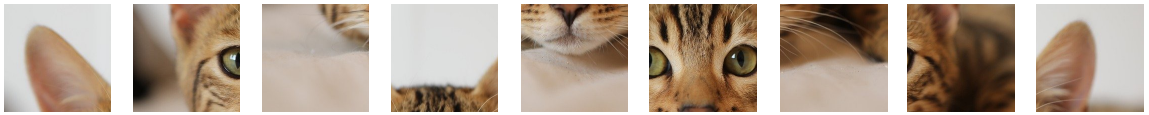
Idea #4: Standard Transformer on Patches



Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Idea #4: Standard Transformer on Patches

N input patches, each of shape 3x16x16

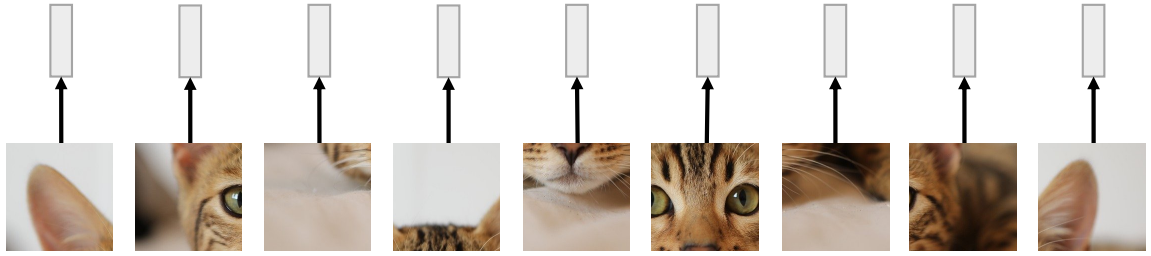


Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Idea #4: Standard Transformer on Patches

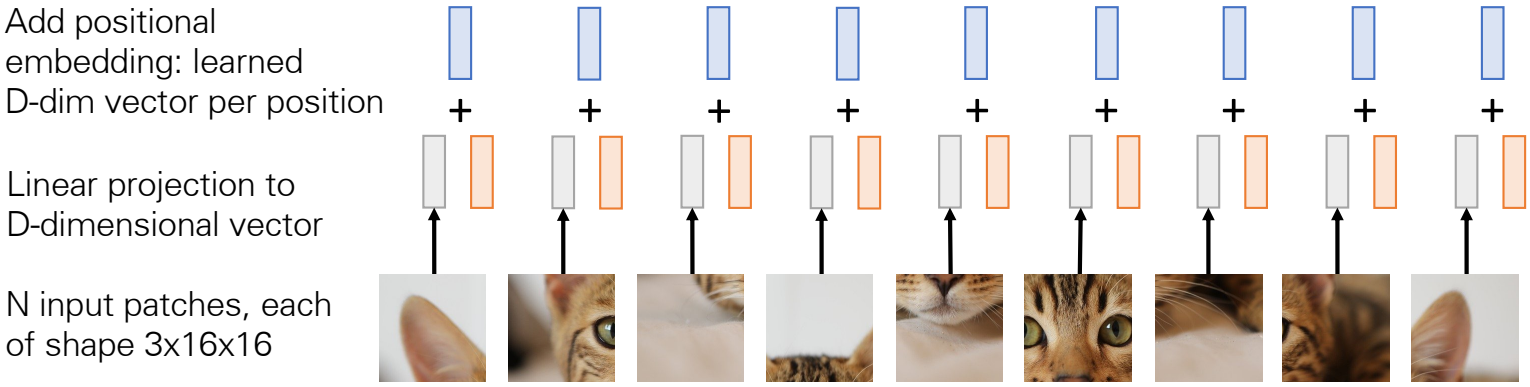
Linear projection to D-dimensional vector

N input patches, each of shape 3x16x16



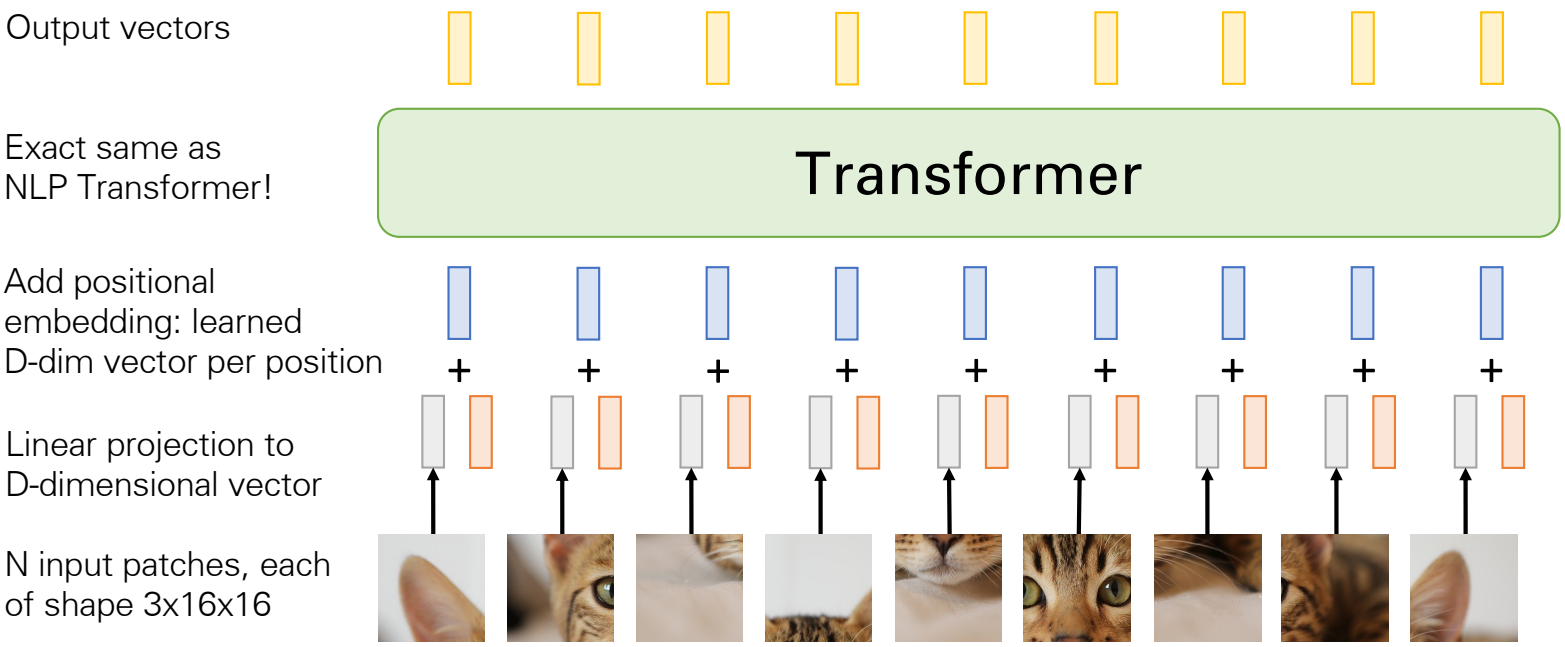
Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Idea #4: Standard Transformer on Patches



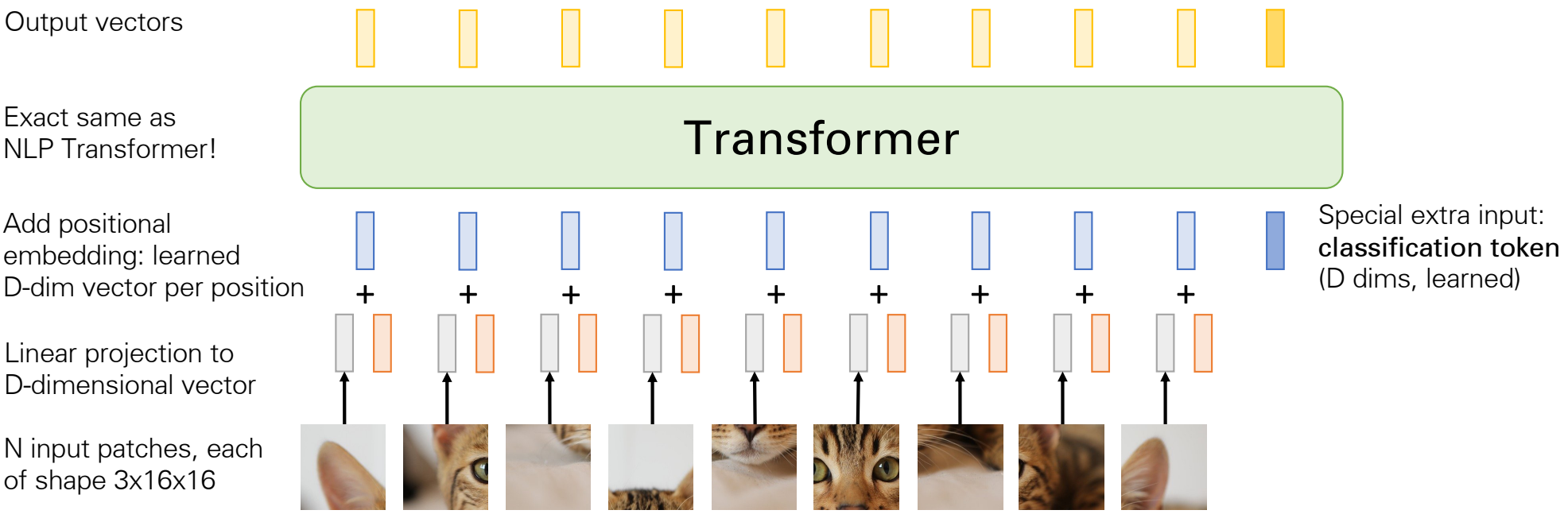
Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Idea #4: Standard Transformer on Patches



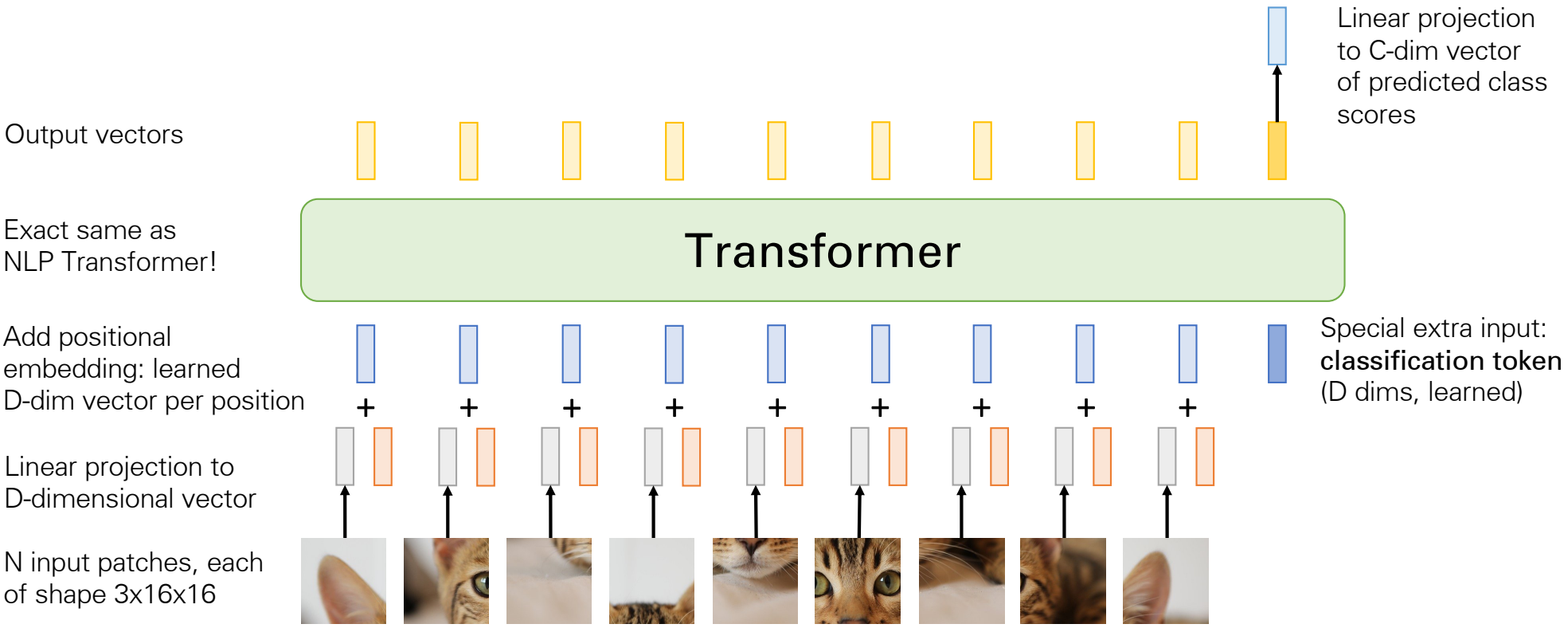
Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Idea #4: Standard Transformer on Patches



Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Idea #4: Standard Transformer on Patches

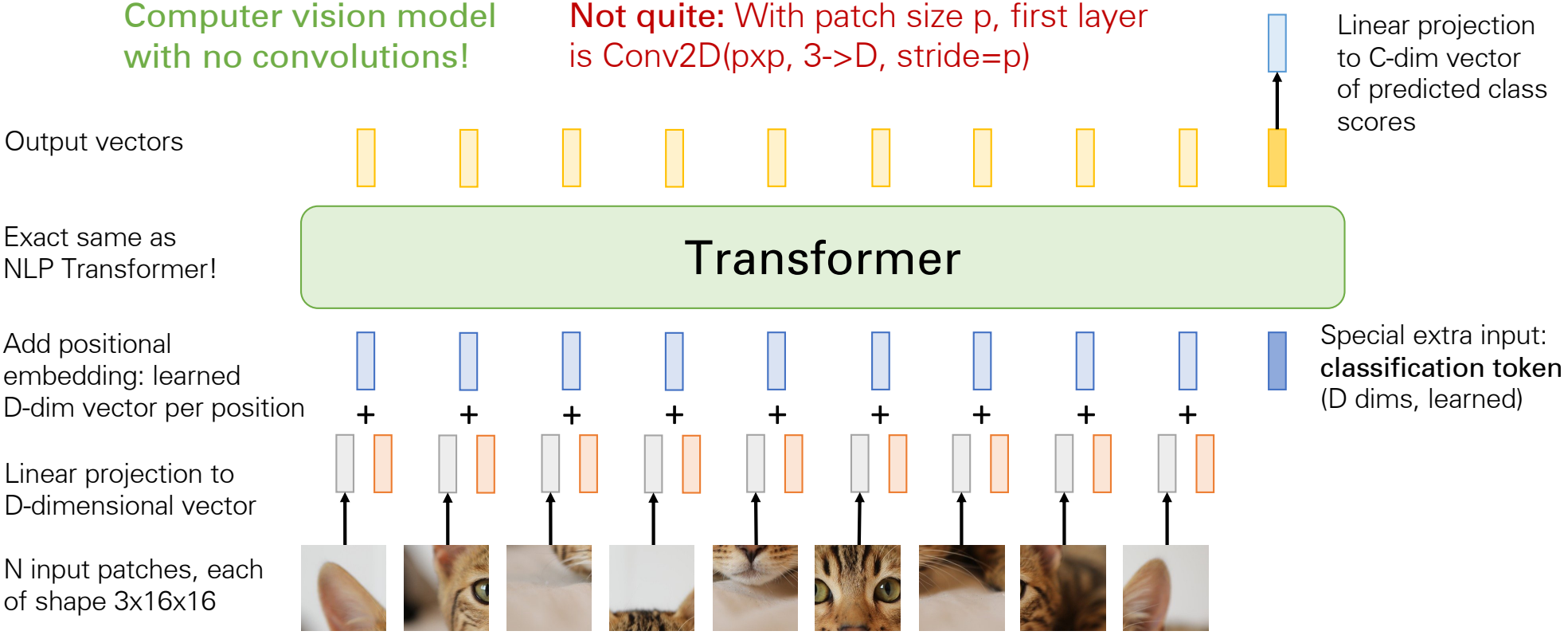


Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer (ViT)

Computer vision model with no convolutions!

Not quite: With patch size p, first layer is Conv2D(pxp, 3->D, stride=p)

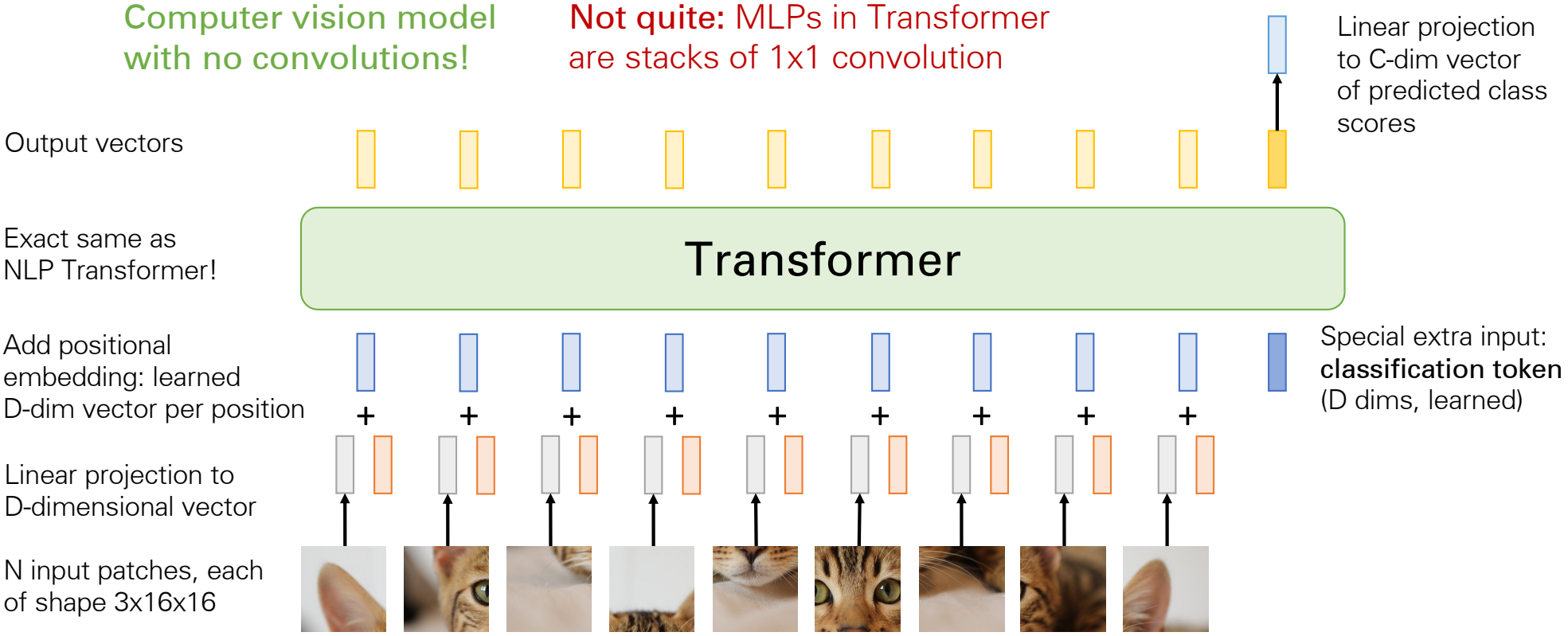


Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer (ViT)

Computer vision model with no convolutions!

Not quite: MLPs in Transformer are stacks of 1x1 convolution

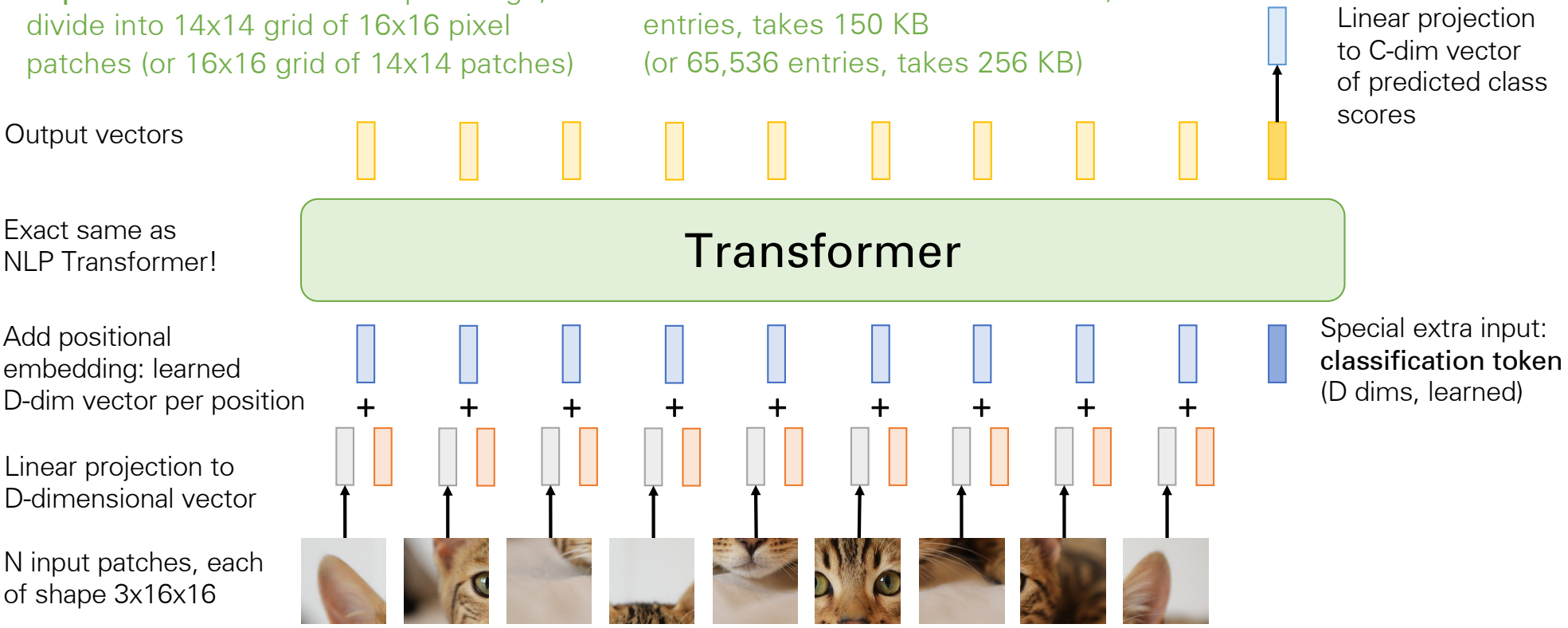


Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer (ViT)

In practice: take 224x224 input image, divide into 14x14 grid of 16x16 pixel patches (or 16x16 grid of 14x14 patches)

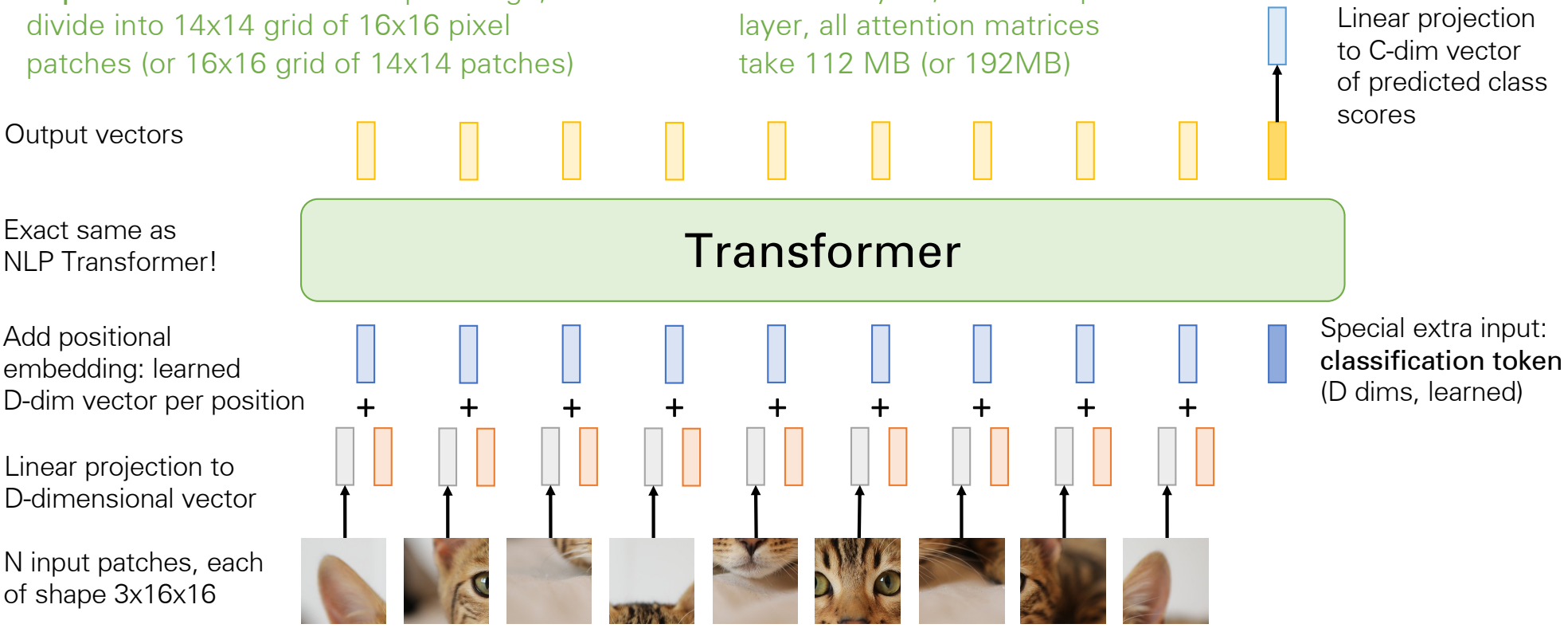
Each attention matrix has $14^4 = 38{,}416$ entries, takes 150 KB
(or 65,536 entries, takes 256 KB)



Vision Transformer (ViT)

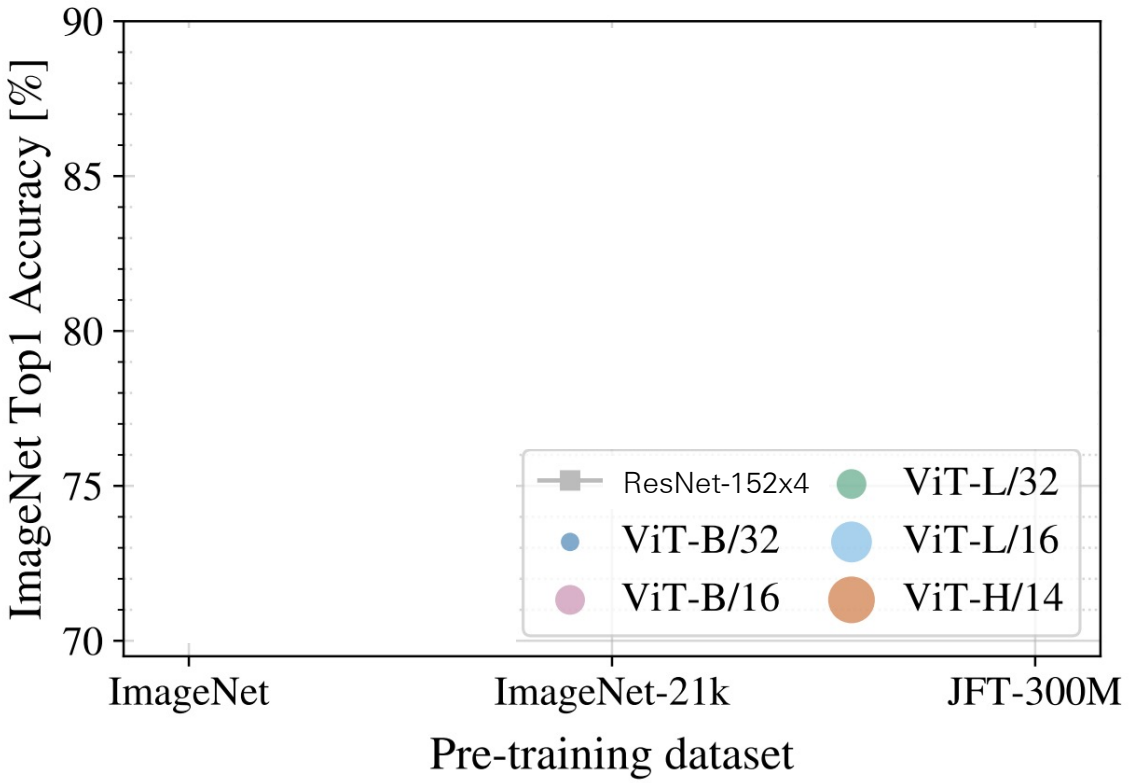
In practice: take 224x224 input image, divide into 14x14 grid of 16x16 pixel patches (or 16x16 grid of 14x14 patches)

With 48 layers, 16 heads per layer, all attention matrices take 112 MB (or 192MB)



Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer (ViT) vs ResNets



B = Base
L = Large
H = Huge

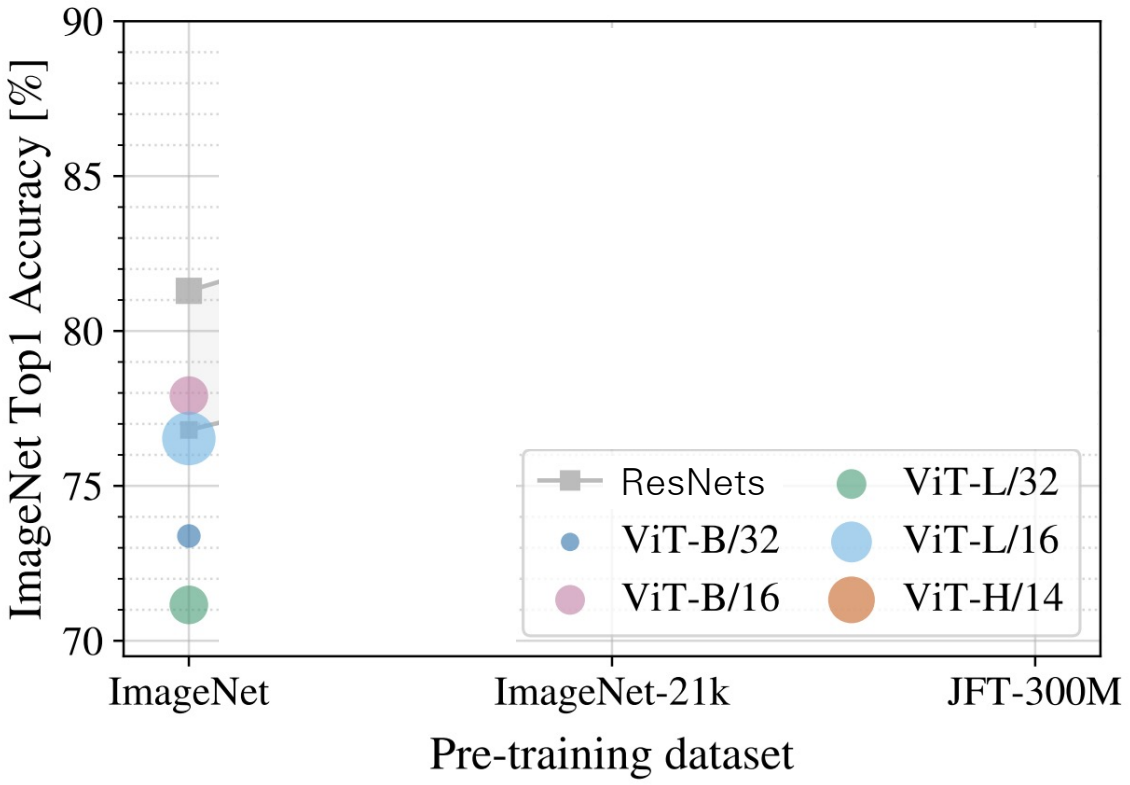
/32, /16, /14 is patch size; smaller patch size is a bigger model (more patches)

Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer (ViT) vs ResNets

Recall: ImageNet dataset has 1k categories, 1.2M images

When trained on ImageNet, ViT models perform worse than ResNets



B = Base
L = Large
H = Huge

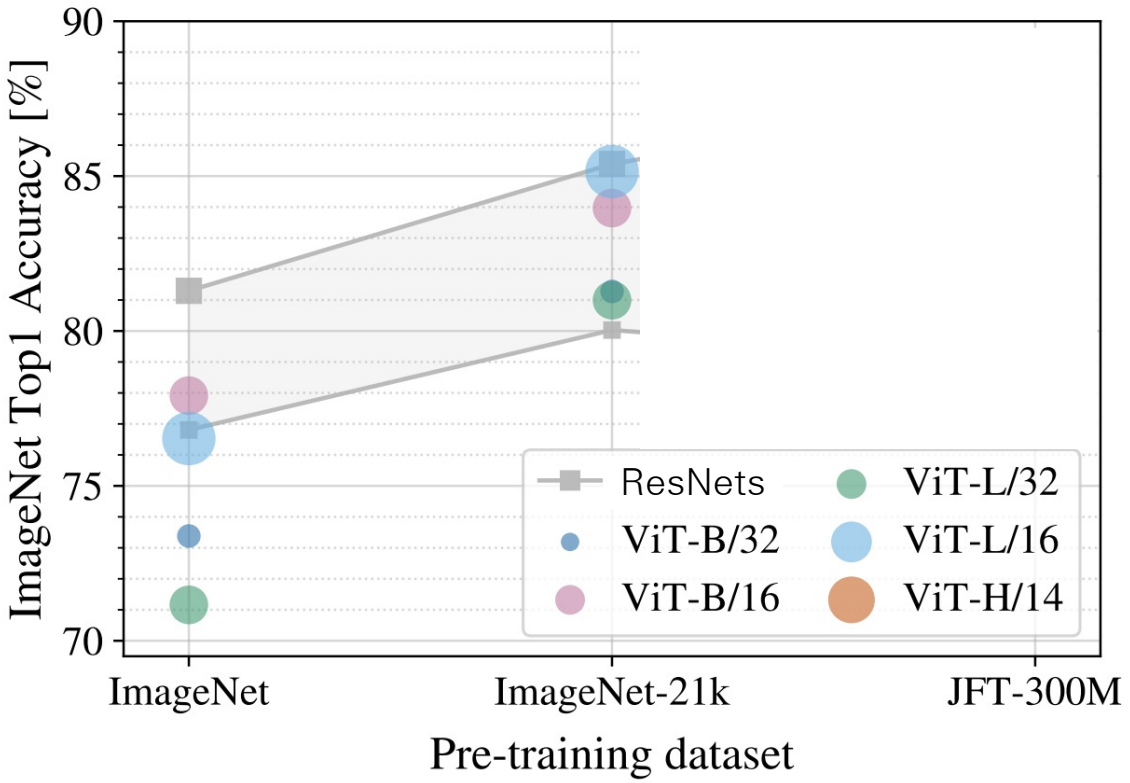
/32, /16, /14 is patch size; smaller patch size is a bigger model (more patches)

Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer (ViT) vs ResNets

ImageNet-21k has 14M images with 21k categories

If you pretrain on ImageNet-21k and fine-tune on ImageNet, ViT does better: big ViTs match big ResNets



B = Base
L = Large
H = Huge

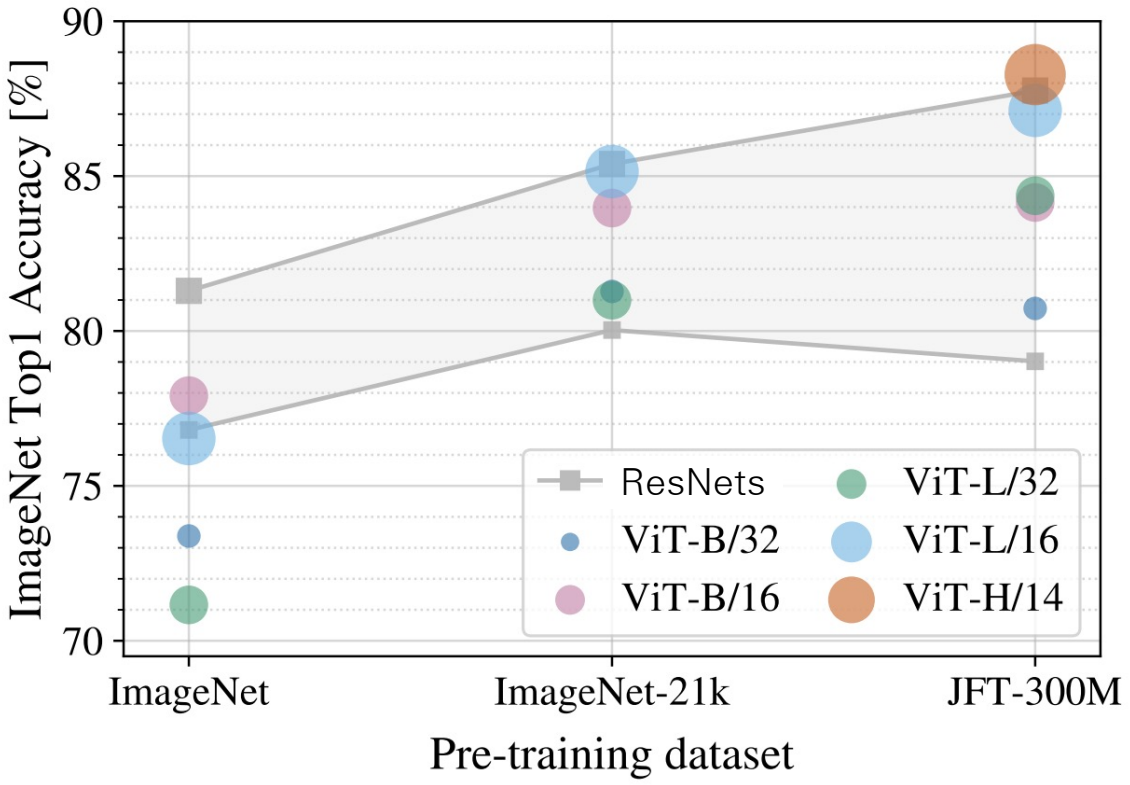
/32, /16, /14 is patch size; smaller patch size is a bigger model (more patches)

Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer (ViT) vs ResNets

JFT-300M is an internal Google dataset with 300M labeled images

If you pretrain on JFT and finetune on ImageNet, large ViTs outperform large ResNets



B = Base
L = Large
H = Huge

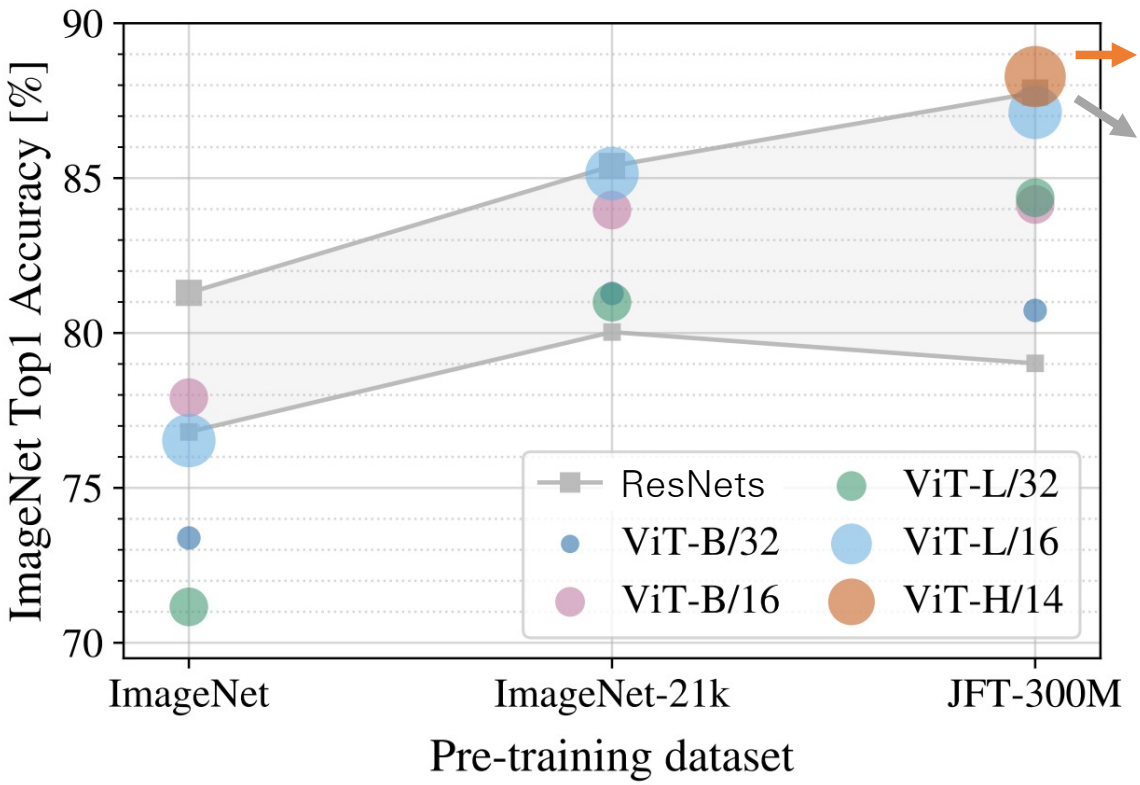
/32, /16, /14 is patch size; smaller patch size is a bigger model (more patches)

Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer (ViT) vs ResNets

JFT-300M is an internal Google dataset with 300M labeled images

If you pretrain on JFT and finetune on ImageNet, large ViTs outperform large ResNets



ViT: 2.5k TPU-v3 core days of training

ResNet: 9.9k TPU-v3 core days of training

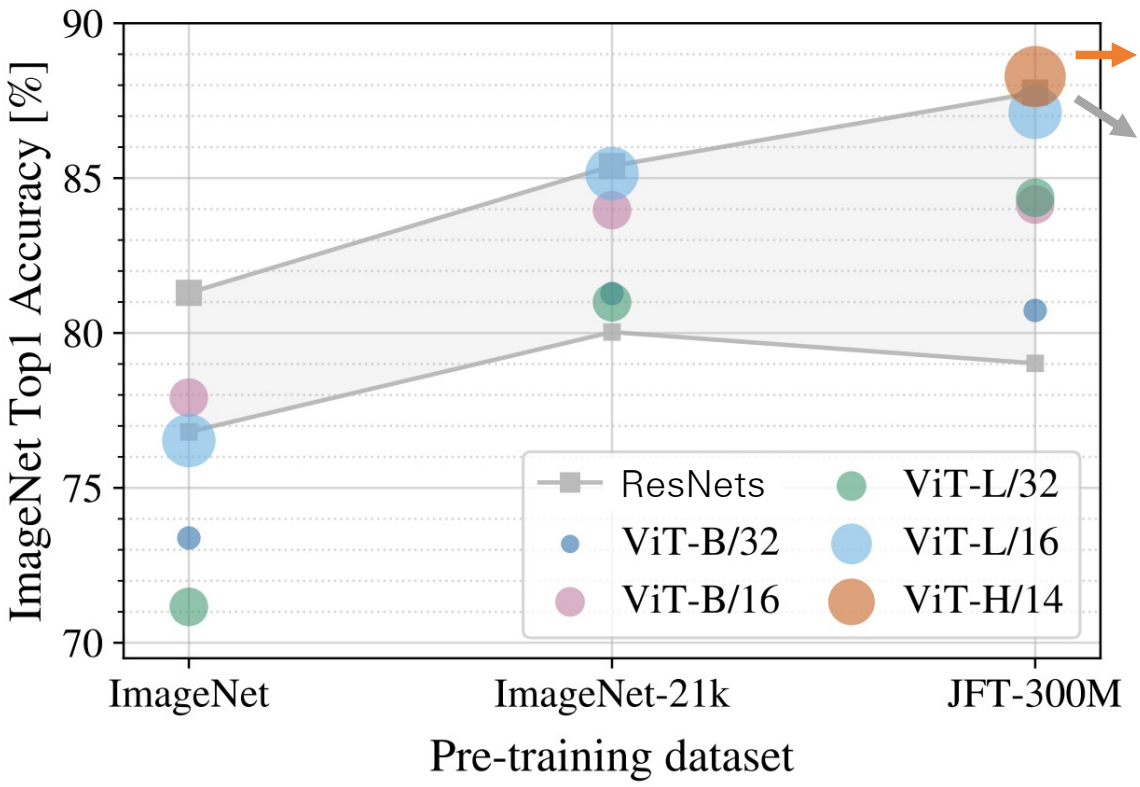
ViTs make more efficient use of GPU / TPU hardware (matrix multiply is more hardware-friendly than conv)

Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer (ViT) vs ResNets

Claim: ViT models have "less inductive bias" than ResNets, so need more pretraining data to learn good features

(Not sure I buy this explanation: "inductive bias" is not a well-defined concept we can measure!)



ViT: 2.5k TPU-v3 core days of training

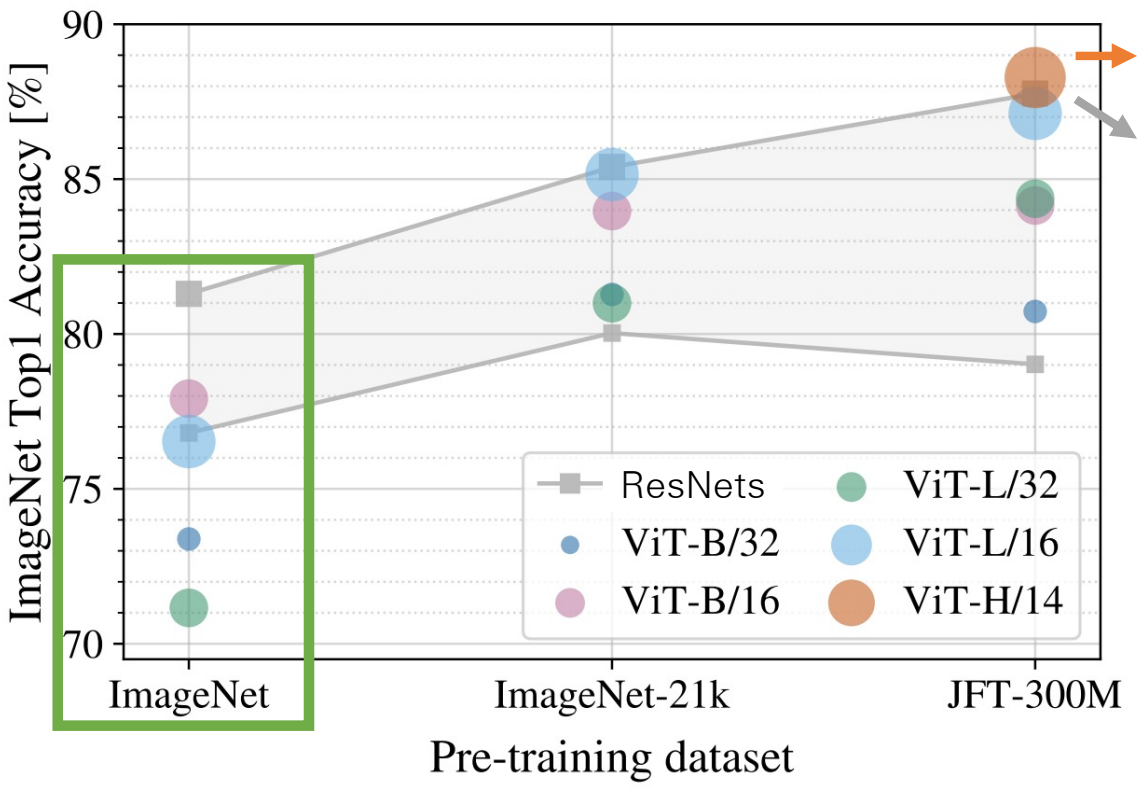
ResNet: 9.9k TPU-v3 core days of training

ViTs make more efficient use of GPU / TPU hardware (matrix multiply is more hardware-friendly than conv)

Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer (ViT) vs ResNets

How can we improve the performance of ViT models on ImageNet?



ViT: 2.5k TPU-v3 core days of training

ResNet: 9.9k TPU-v3 core days of training

ViTs make more efficient use of GPU / TPU hardware (matrix multiply is more hardware-friendly than conv)

Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

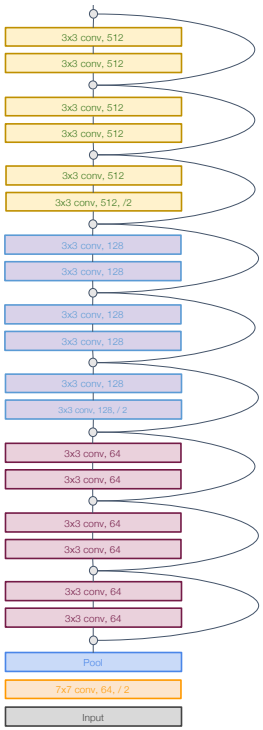
ViT vs CNN

Stage 3:
256 x 14 x 14

Stage 2:
128 x 28 x 28

Stage 1:
64 x 56 x 56

Input:
3 x 224 x 224



In most CNNs (including ResNets), **decrease** resolution and **increase** channels as you go deeper in the network (Hierarchical architecture)

Useful since objects in images can occur at various scales

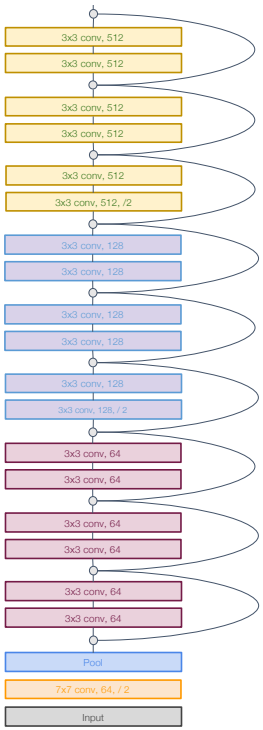
ViT vs CNN

Stage 3:
256 x 14 x 14

Stage 2:
128 x 28 x 28

Stage 1:
64 x 56 x 56

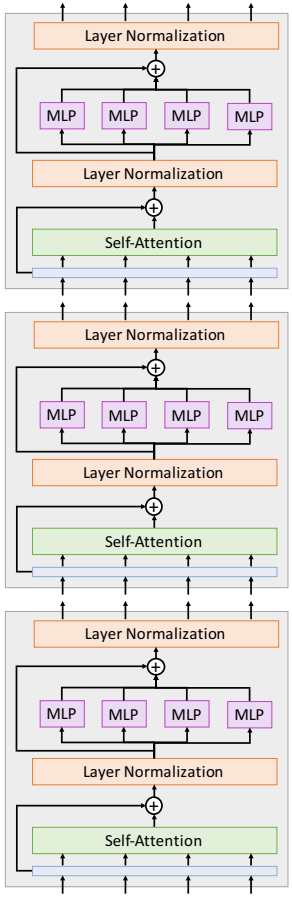
Input:
3 x 224 x 224



In most CNNs (including ResNets), **decrease** resolution and **increase** channels as you go deeper in the network (Hierarchical architecture)

Useful since objects in images can occur at various scales

In a ViT, all blocks have same resolution and number of channels (Isotropic architecture)



3rd block:
768 x 14 x 14

2nd block:
768 x 14 x 14

1st block:
768 x 14 x 14

Input:
3 x 224 x 224

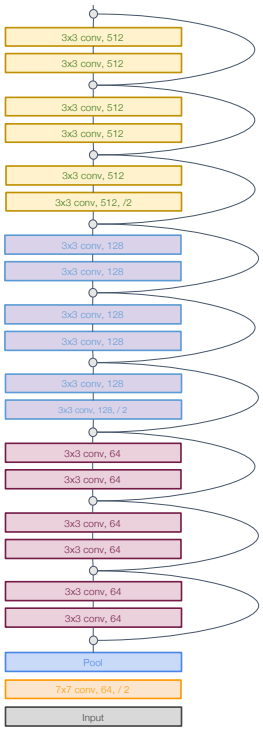
ViT vs CNN

Stage 3:
256 x 14 x 14

Stage 2:
128 x 28 x 28

Stage 1:
64 x 56 x 56

Input:
3 x 224 x 224

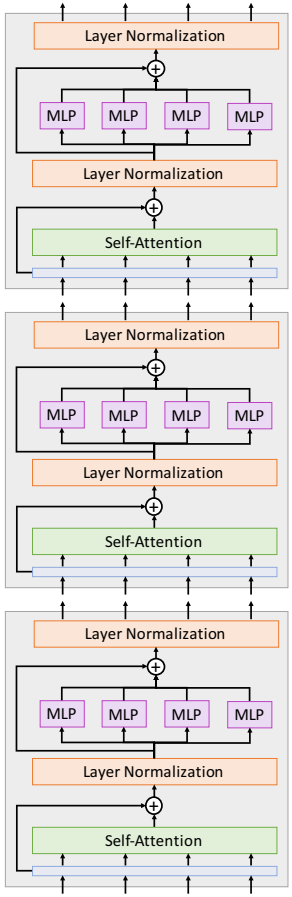


In most CNNs (including ResNets), **decrease** resolution and **increase** channels as you go deeper in the network (Hierarchical architecture)

Useful since objects in images can occur at various scales

In a ViT, all blocks have same resolution and number of channels (Isotropic architecture)

Can we build a **hierarchical** ViT model?



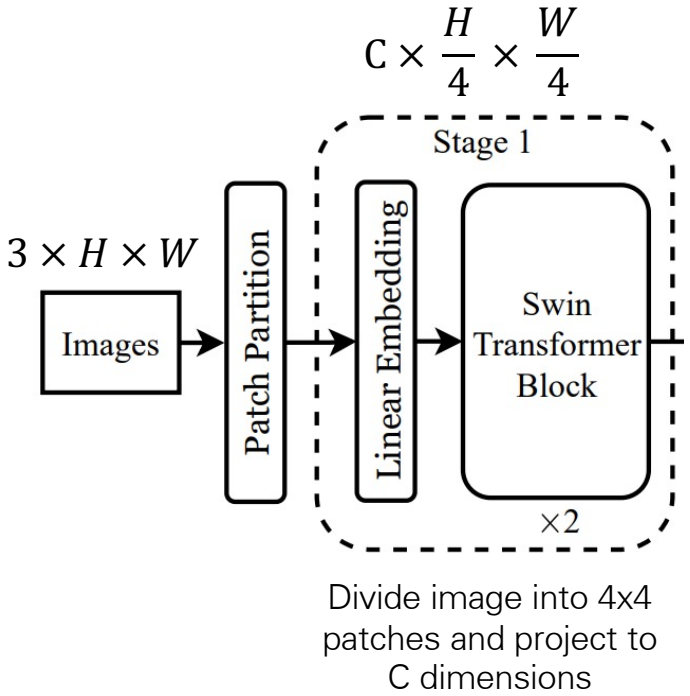
3rd block:
768 x 14 x 14

2nd block:
768 x 14 x 14

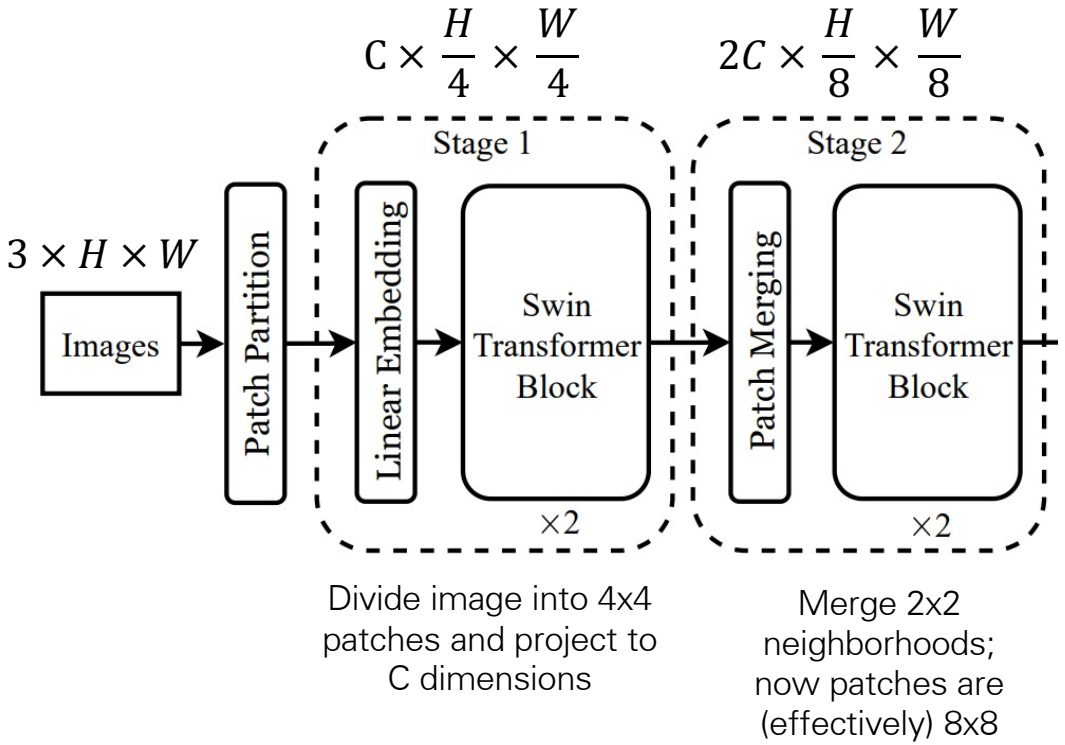
1st block:
768 x 14 x 14

Input:
3 x 224 x 224

Hierarchical ViT: Swin Transformer

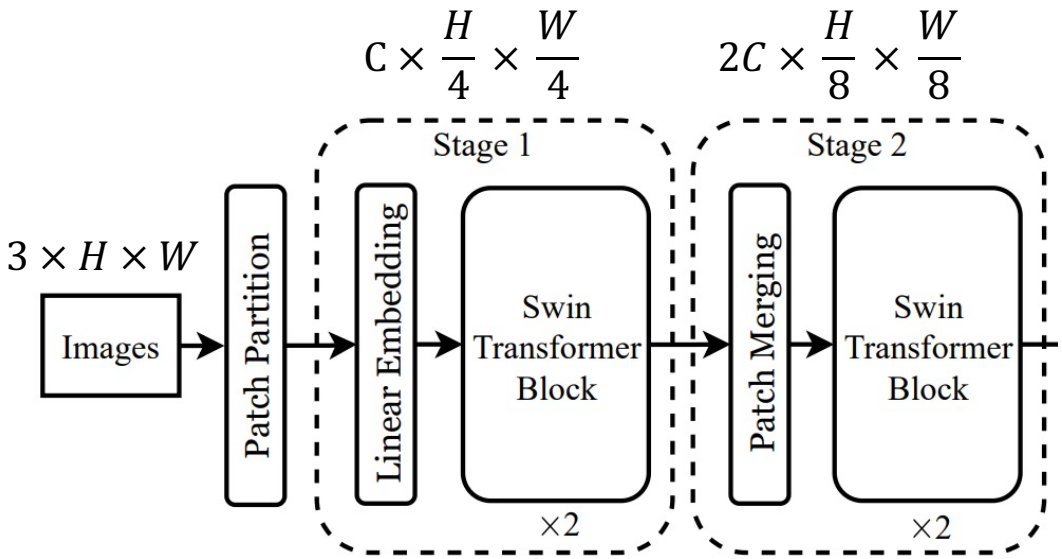


Hierarchical ViT: Swin Transformer



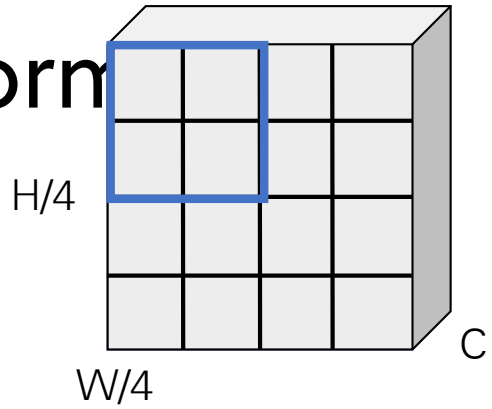
Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

Hierarchical ViT: Swin Transform

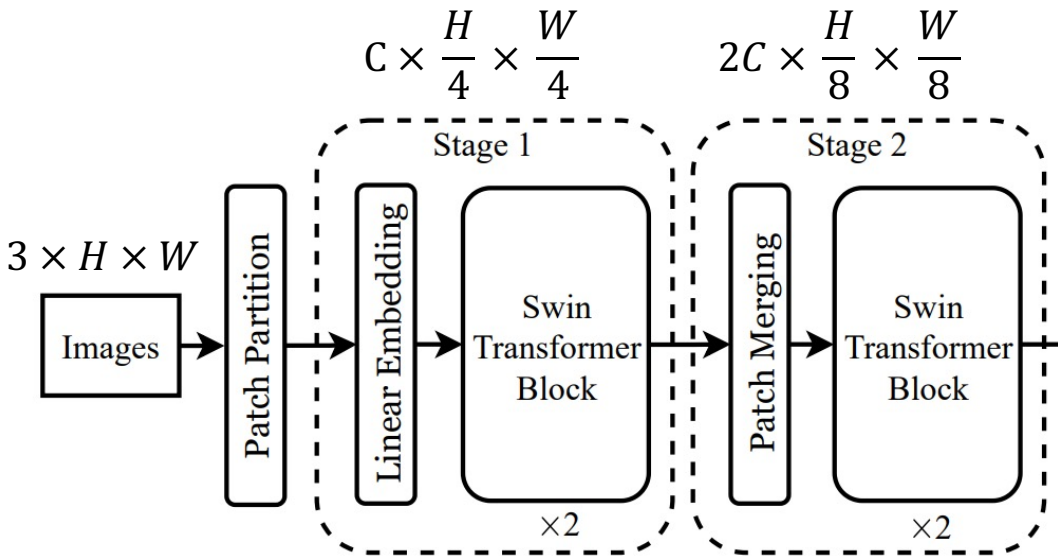


Divide image into 4x4 patches and project to C dimensions

Merge 2x2 neighborhoods; now patches are (effectively) 8x8

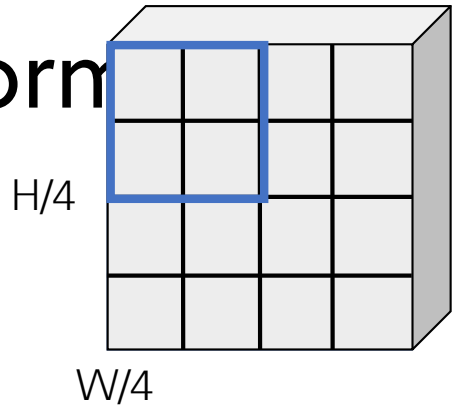


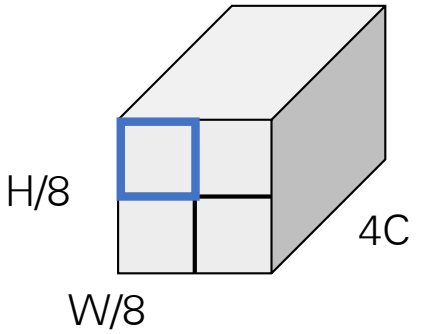
Hierarchical ViT: Swin Transform



Divide image into 4x4 patches and project to C dimensions

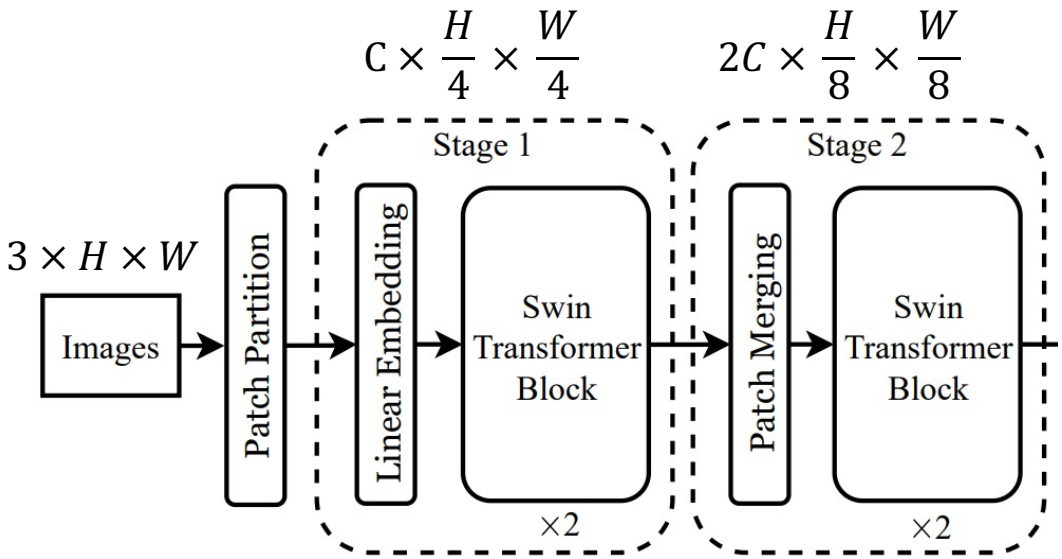
Merge 2x2 neighborhoods; now patches are (effectively) 8x8

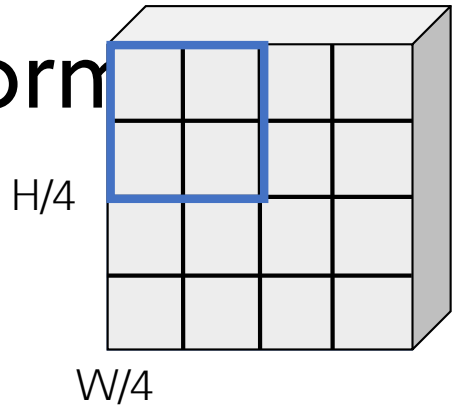


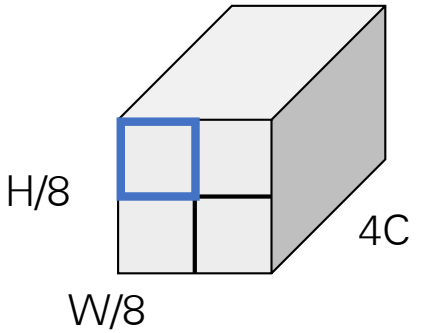


Concatenate groups of 2x2 features

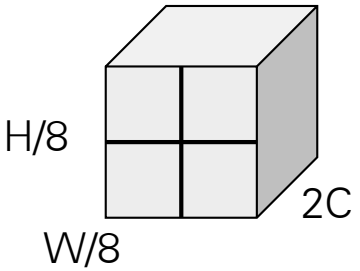
Hierarchical ViT: Swin Transform





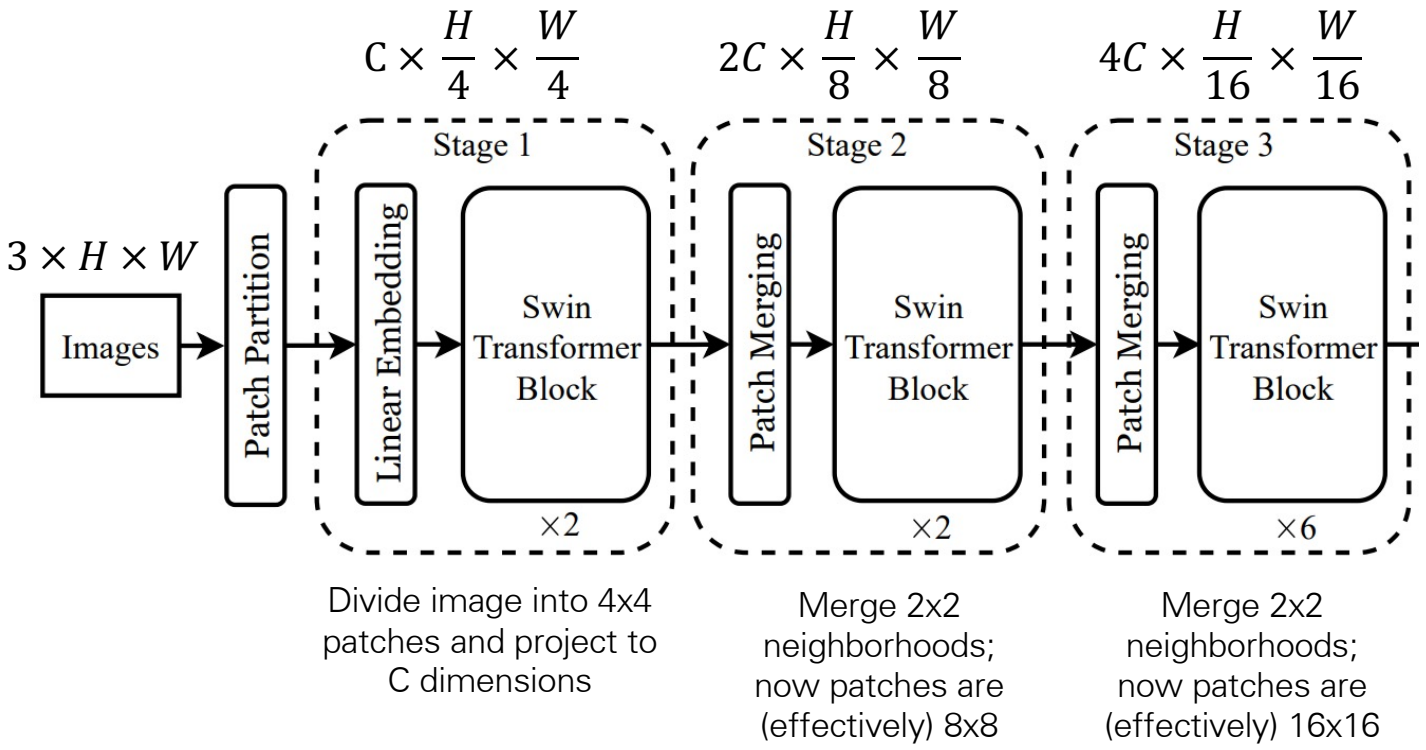


Concatenate groups of 2x2 features



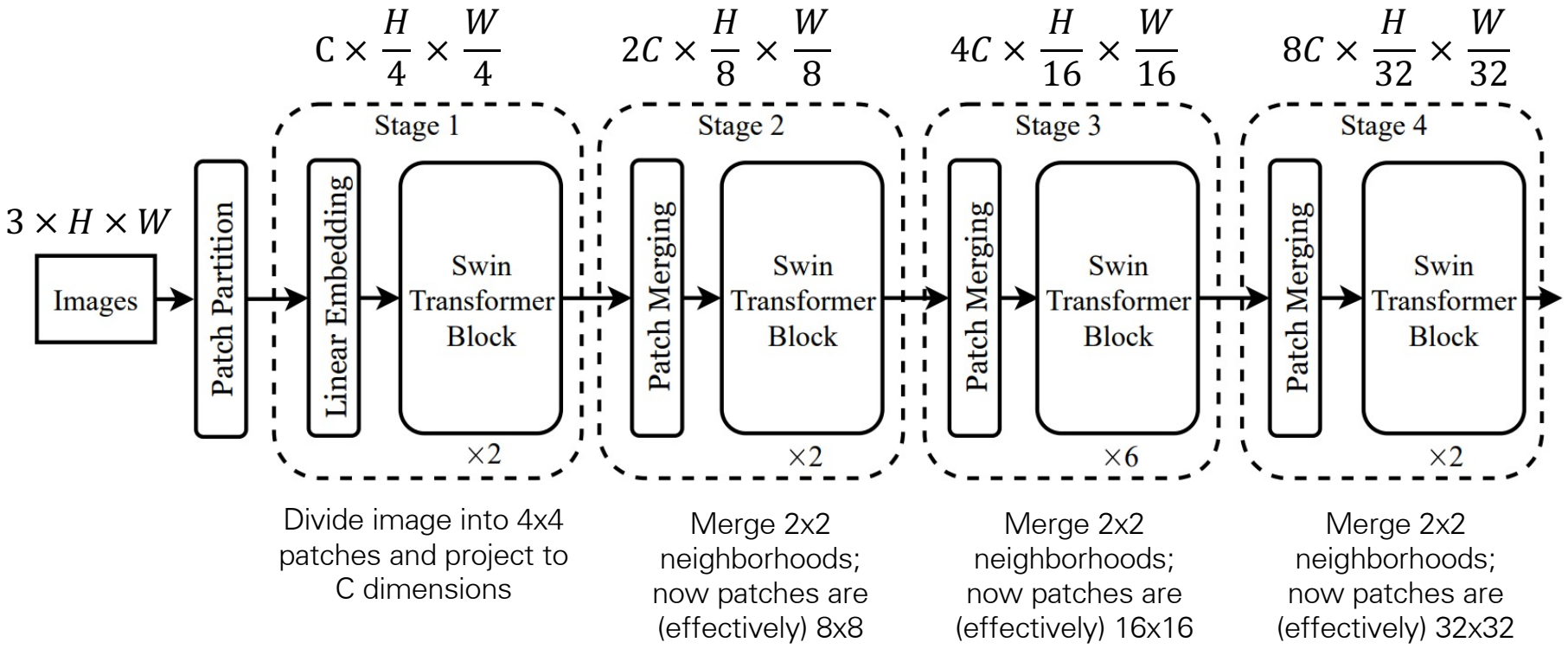
Linear projection from 4C to 2C channels (1x1 conv)

Hierarchical ViT: Swin Transformer



Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

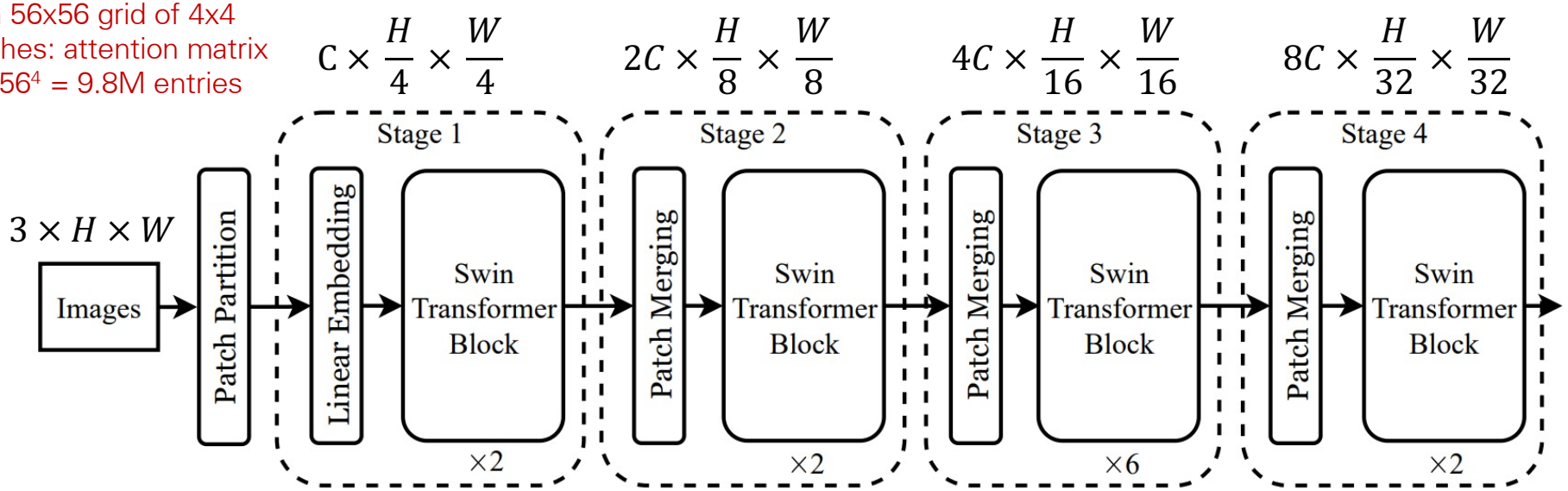
Hierarchical ViT: Swin Transformer



Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

Hierarchical ViT: Swin Transformer

Problem: 224x224 image with 56x56 grid of 4x4 patches: attention matrix has 56^4 = 9.8M entries



Divide image into 4x4 patches and project to C dimensions

Merge 2x2 neighborhoods; now patches are (effectively) 8x8

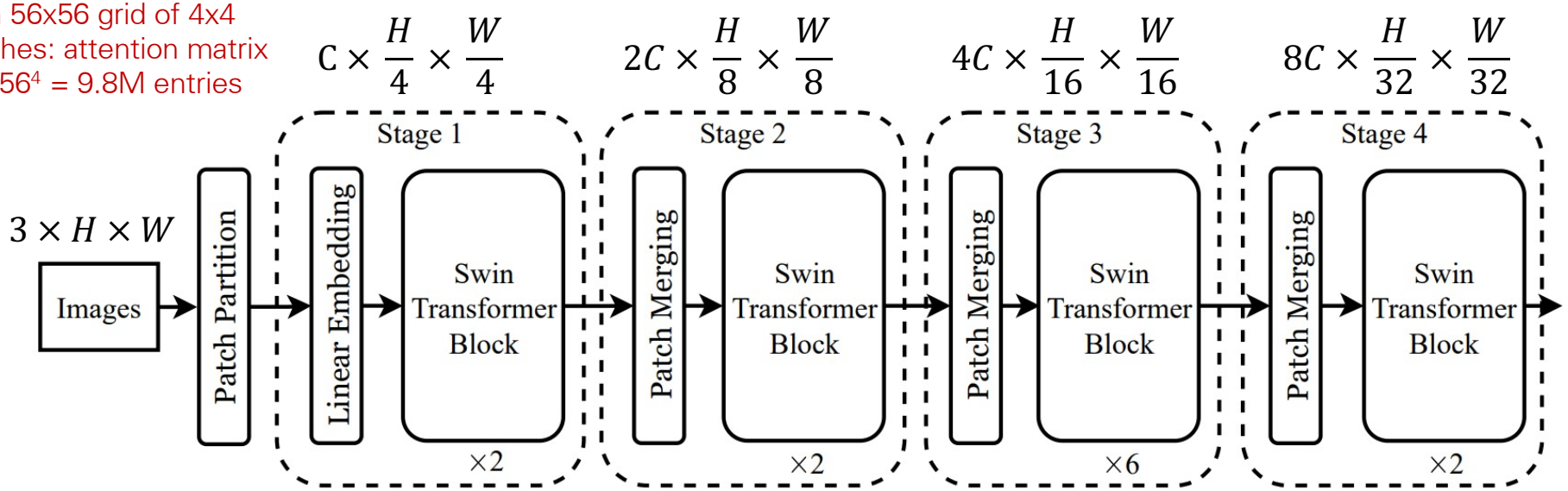
Merge 2x2 neighborhoods; now patches are (effectively) 16x16

Merge 2x2 neighborhoods; now patches are (effectively) 32x32

Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

Hierarchical ViT: Swin Transformer

Problem: 224x224 image with 56x56 grid of 4x4 patches: attention matrix has 56^4 = 9.8M entries



Solution: don't use full attention, instead use attention over patches

Divide image into 4x4 patches and project to C dimensions

Merge 2x2 neighborhoods; now patches are (effectively) 8x8

Merge 2x2 neighborhoods; now patches are (effectively) 16x16

Merge 2x2 neighborhoods; now patches are (effectively) 32x32

Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

Swin Transformer: Window Attention

With H x W grid of tokens, each attention matrix is H^2W^2 – **quadratic** in image size

Swin Transformer: Window Attention

With H x W grid of tokens, each attention matrix is H^2W^2 – **quadratic** in image size

Rather than allowing each token to attend to all other tokens, instead divide into windows of M x M tokens (here M=4); only compute attention within each window

Swin Transformer: Window Attention

With H x W grid of tokens, each attention matrix is H^2W^2 – **quadratic** in image size

Rather than allowing each token to attend to all other tokens, instead divide into windows of M x M tokens (here M=4); only compute attention within each window

Total size of all attention matrices is now:
$M^4(H/M)(W/M) = M^2HW$

Linear in image size for fixed M!
Swin uses M=7 throughout the network

Swin Transformer: Window Attention

Problem: tokens only interact with other tokens within the same window; no communication across windows

Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

Swin Transformer: Shifted Window Attention

Solution: Alternate between normal windows and shifted windows in successive Transformer blocks

Block L: Normal windows

Block L+1: Shifted Windows

Ugly detail: Non-square windows at edges and corners

Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

Swin Transformer: Shifted Window Attention

Solution: Alternate between normal windows and shifted windows in successive Transformer blocks

Block L: Normal windows

Block L+1: Shifted Windows

Detail: Relative Positional Bias

ViT adds positional embedding to input tokens, encodes absolute position of each token in the image

Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

Swin Transformer: Shifted Window Attention

Solution: Alternate between normal windows and shifted windows in successive Transformer blocks

Block L: Normal windows

Block L+1: Shifted Windows

Detail: Relative Positional Bias

ViT adds positional embedding to input tokens, encodes absolute position of each token in the image

Swin does not use positional embeddings, instead encodes relative position between patches when computing attention:

Standard Attention:

$$A = Softmax\left(\frac{QK^T}{\sqrt{D}}\right)V$$

$Q, K, V: M^2 \times D$ (Query, Key, Value)

Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

Swin Transformer: Shifted Window Attention

Solution: Alternate between normal windows and shifted windows in successive Transformer blocks

Block L: Normal windows

Block L+1: Shifted Windows

Detail: Relative Positional Bias

ViT adds positional embedding to input tokens, encodes absolute position of each token in the image

Swin does not use positional embeddings, instead encodes relative position between patches when computing attention:

Attention with relative bias:

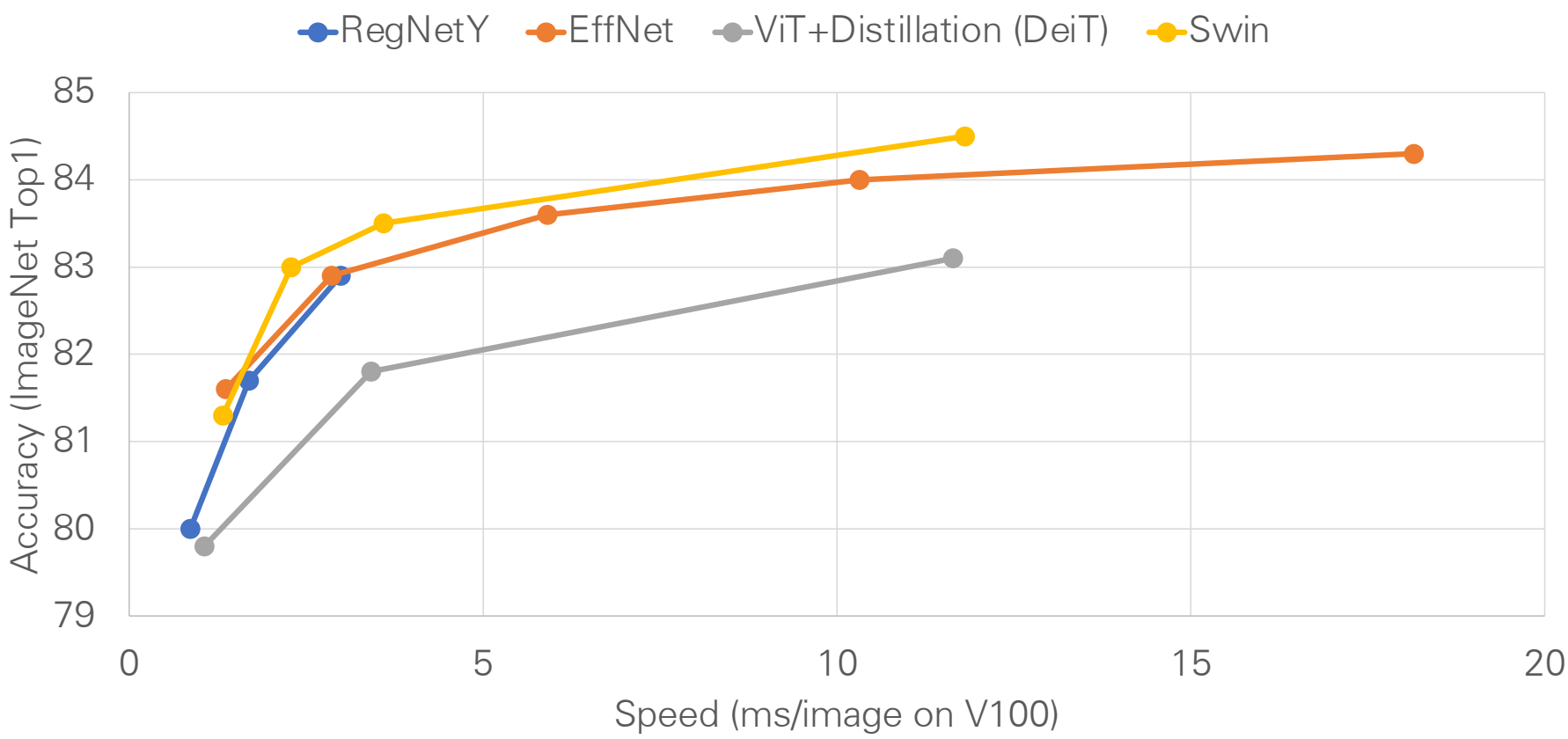
$$A = Softmax\left(\frac{QK^T}{\sqrt{D}} + B\right)V$$

$Q, K, V: M^2 \times D$ (Query, Key, Value)

$B: M^2 \times M^2$ (learned biases)

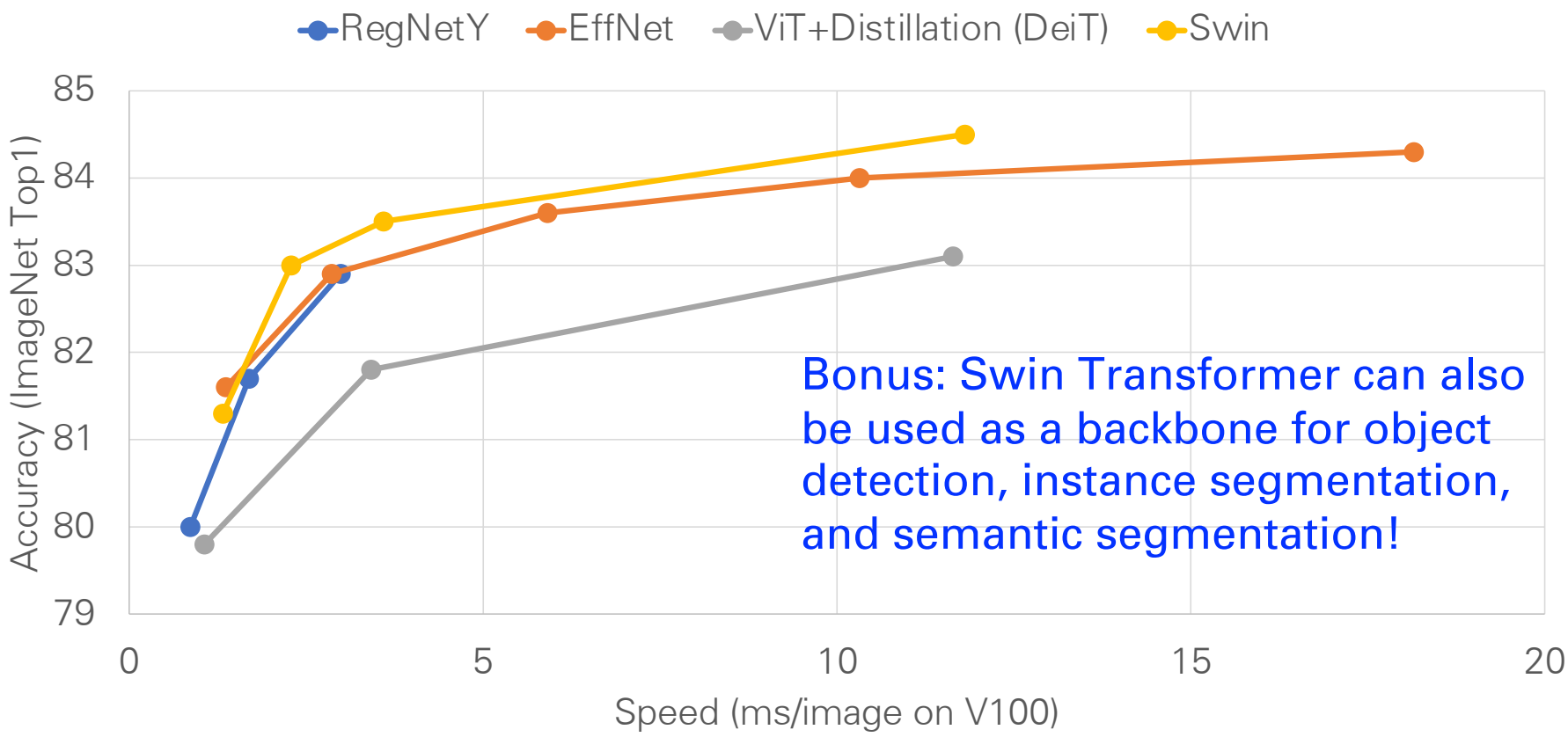
Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

Swin Transformer: Speed vs Accuracy



Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

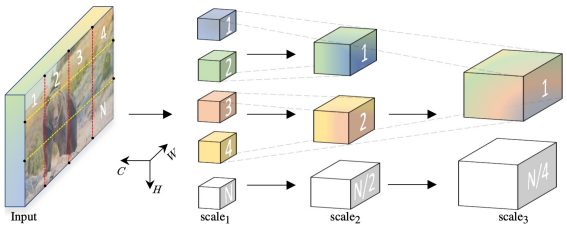
Swin Transformer: Speed vs Accuracy



Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", CVPR 2021

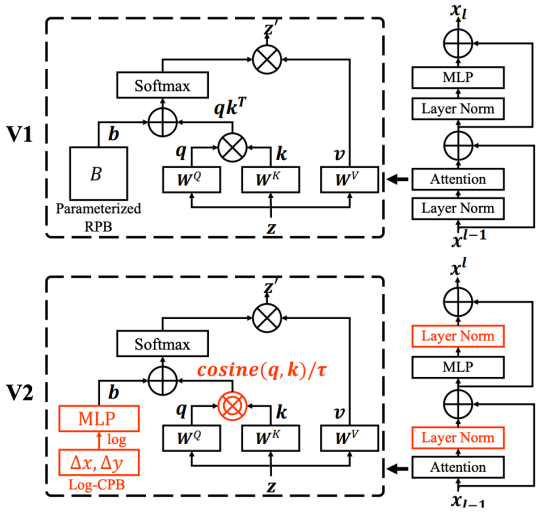
Other Hierarchical Vision Transformers

MViT



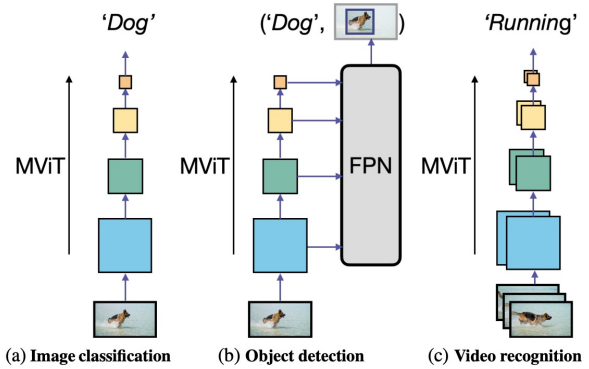
Fan et al., "Multiscale Vision Transformers", ICCV 2021

Swin-V2



Liu et al, "Swin Transformer V2: Scaling up Capacity and Resolution", CVPR 2022

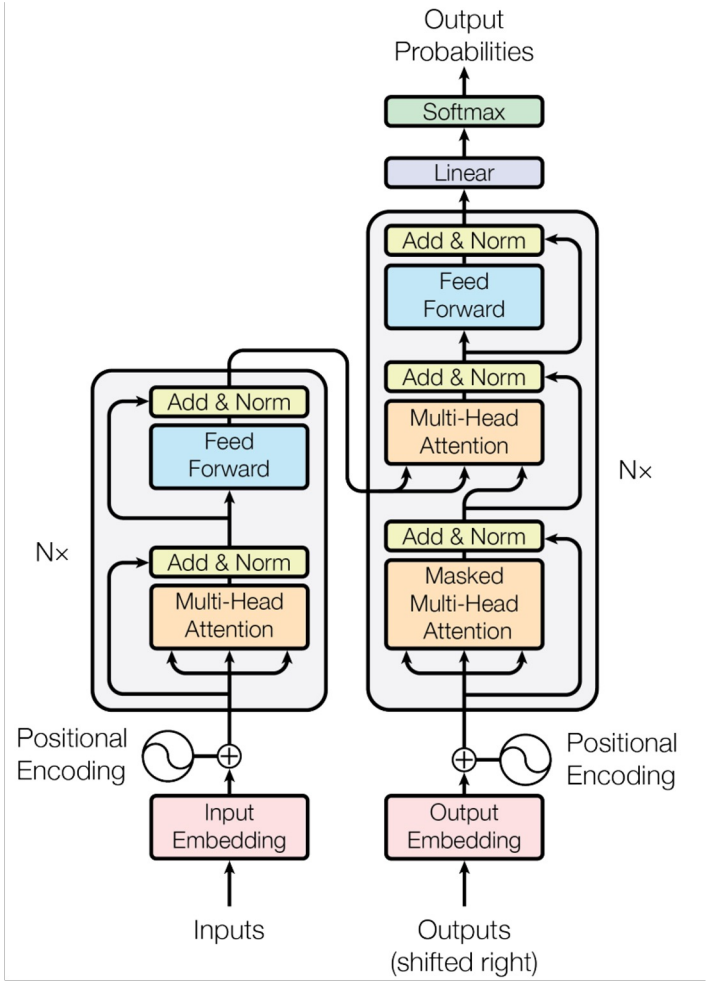
Improved MViT



Li et al, "Improved Multiscale Vision Transformers for Classification and Detection", arXiv 2021

Recap of Transformers

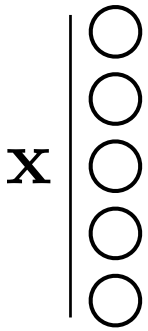
- Three key ideas
  - Tokens
  - Attention
  - Positional encoding



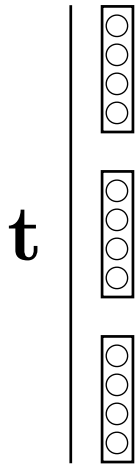
Tokens: A new data structure

- A **token** is just transformer lingo for 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**

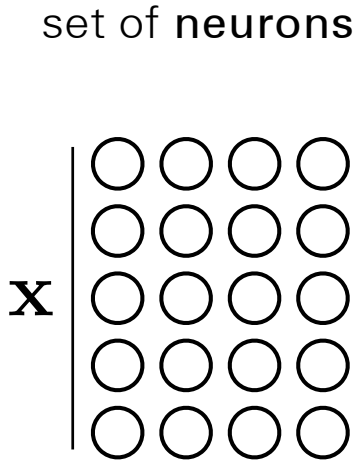


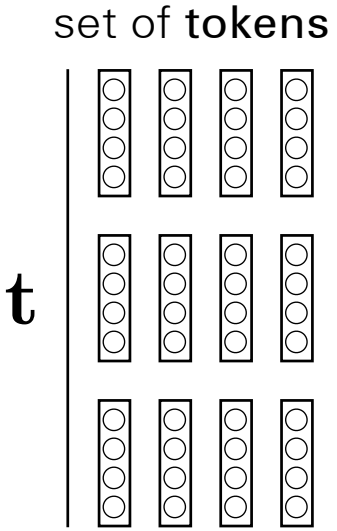
array of **tokens**



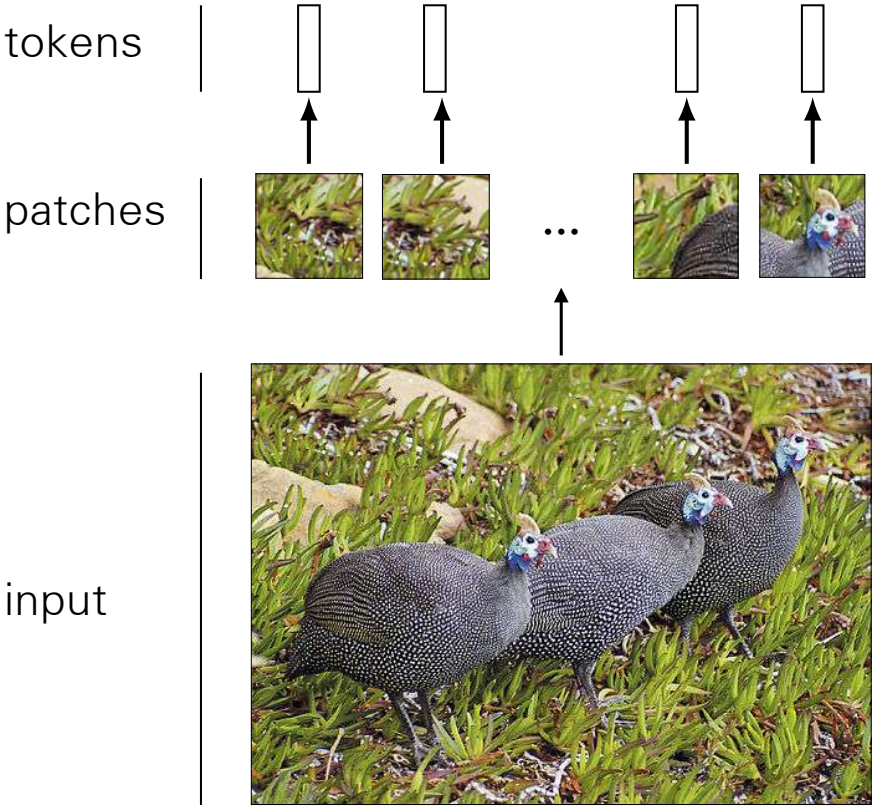
Tokens: A new data structure

- A **token** is just transformer lingo for 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





Tokenizing the input data

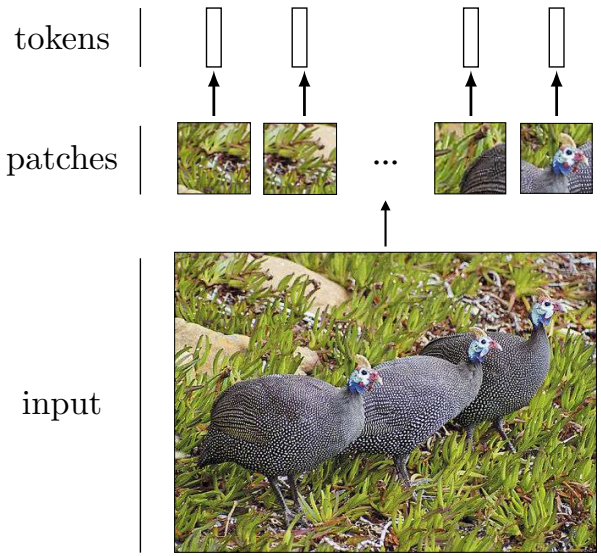


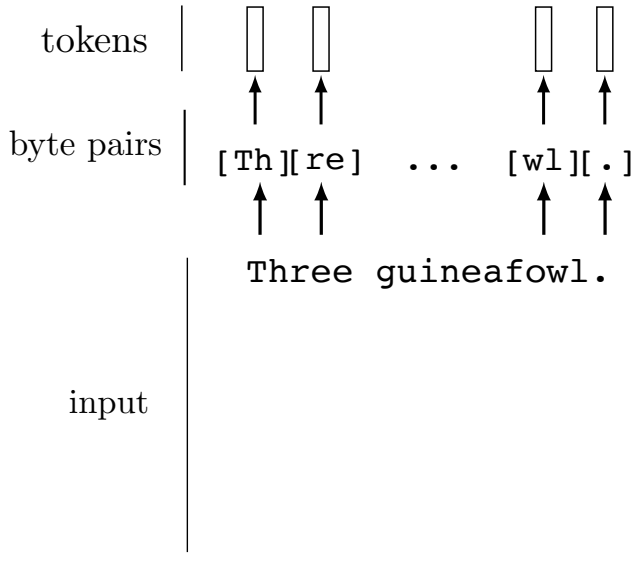
e.g., linear projection

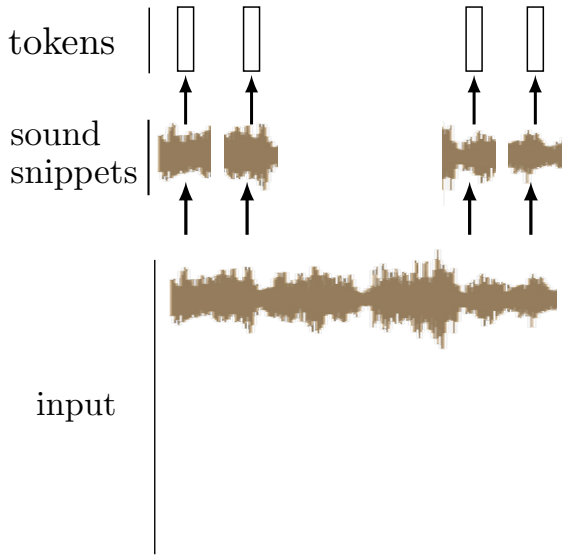
- When operating over neurons, we represent the input as an array of scalar-valued measurements (e.g., pixels)
- When operating over tokens, we represent the input as an array of vector-valued measurements

Tokenizing the input data

- You can tokenize anything.
- General strategy: chop the input up into chunks, project each chunk to a vector.

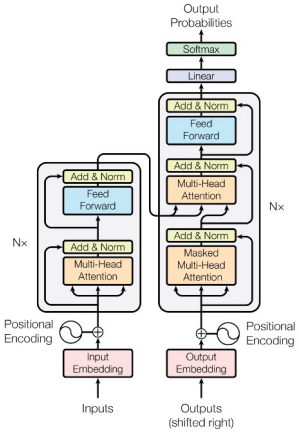




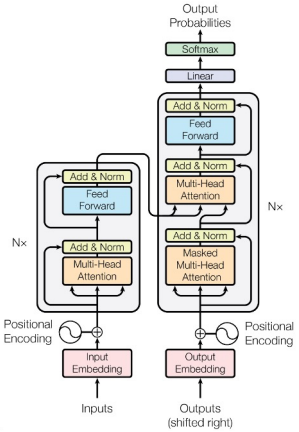


Transformers

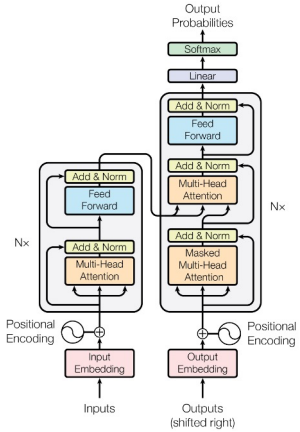
- Transformers takeover the communities since their introduction.



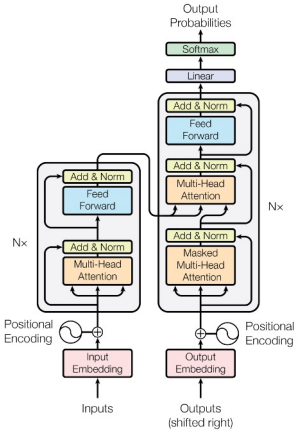
Computer Vision



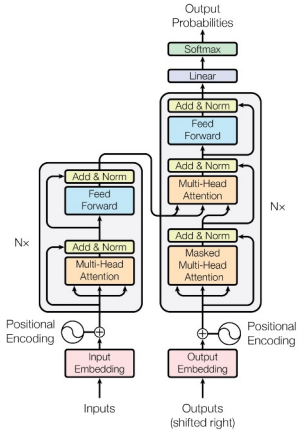
Natural Lang. Proc.



Speech

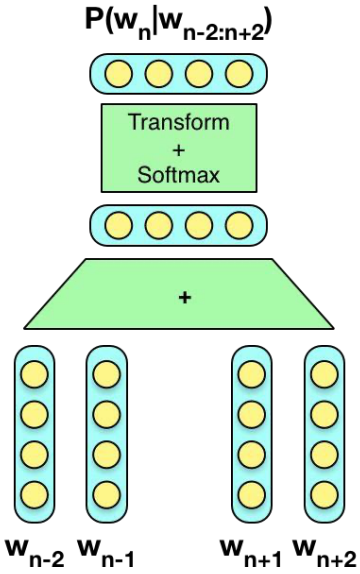


Reinf. Learning

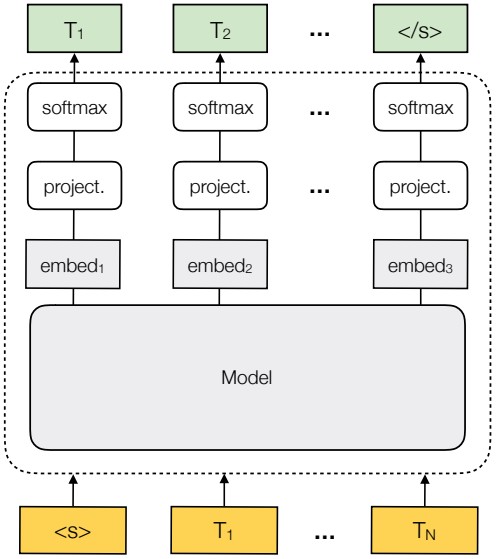


Graphs / Science

Pre-training in NLP (before Transformers)



word embeddings
word2vec
[Mikolov et al., 2013]

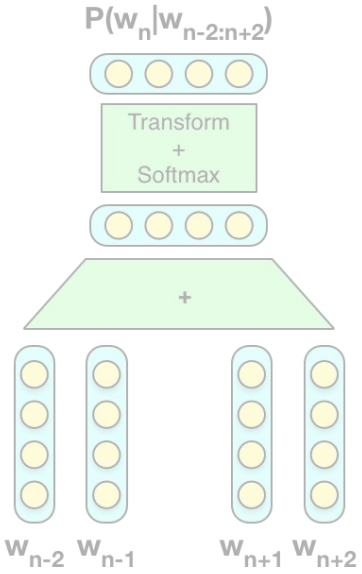


contextualized
word embeddings via LM
ELMo
[Peters et al., 2018]

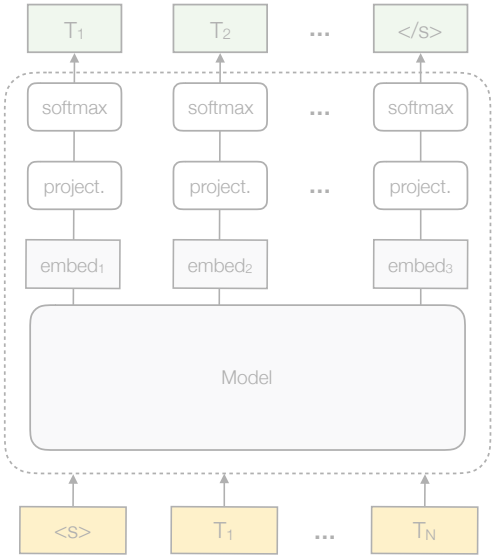
- Word embeddings ⇒ Contextualized word embeddings

Image credit: Noe Casas

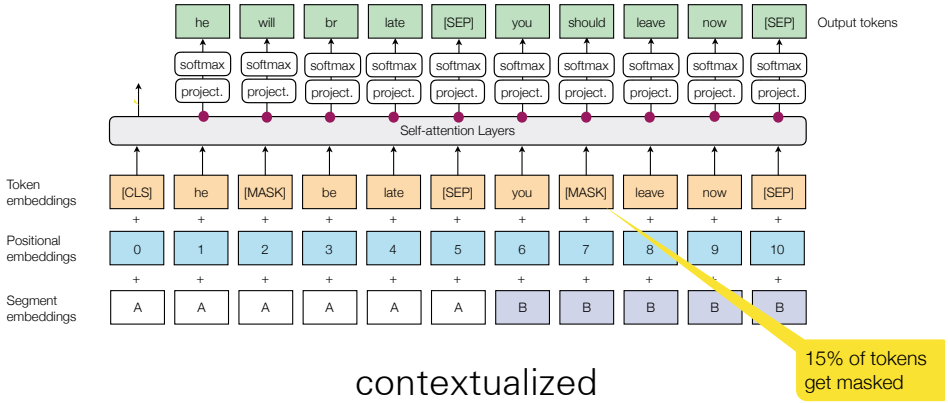
Pre-training in NLP (during Transformers)



word embeddings
word2vec
[Mikolov et al., 2013]



contextualized
word embeddings via LM
ELMo
[Peters et al., 2018]



contextualized
word embeddings via
masked LM +
next sentence prediction
BERT
[Devlin et al., 2019]

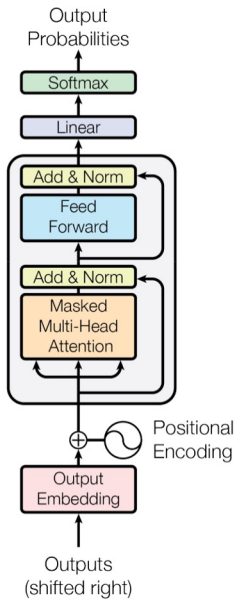
- Word embeddings ⇒ Contextualized word embeddings ⇒ Transformers
- Transformer-based models take over the language modelling / NLP domain

Image credit: Noe Casas

Pre-training in NLP (during Transformers)

Decoder-only
GPT

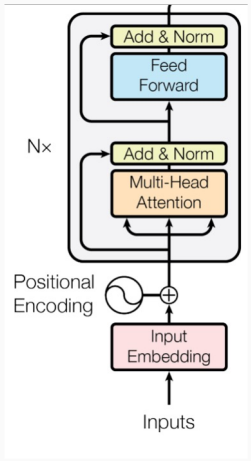
[sat_]



[START] [The_] [cat_]

Encoder-only
BERT

[*] [*] **[sat_]** [*] **[the_]** [*]



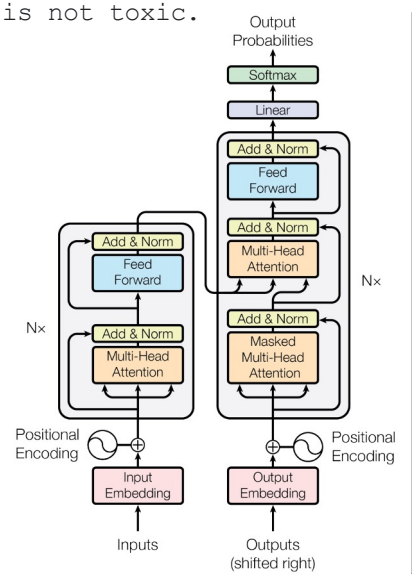
[The_] [cat_] **[MASK]** [on_] **[MASK]** [mat_]

Enc-Dec
T5

Das ist gut.

A storm in Attala caused 6 victims.

This is not toxic.



Translate EN-DE: This is good.

Summarize: state authorities dispatched…

Is this toxic: You look beautiful today!

Pre-training in Vision (during Transformers)

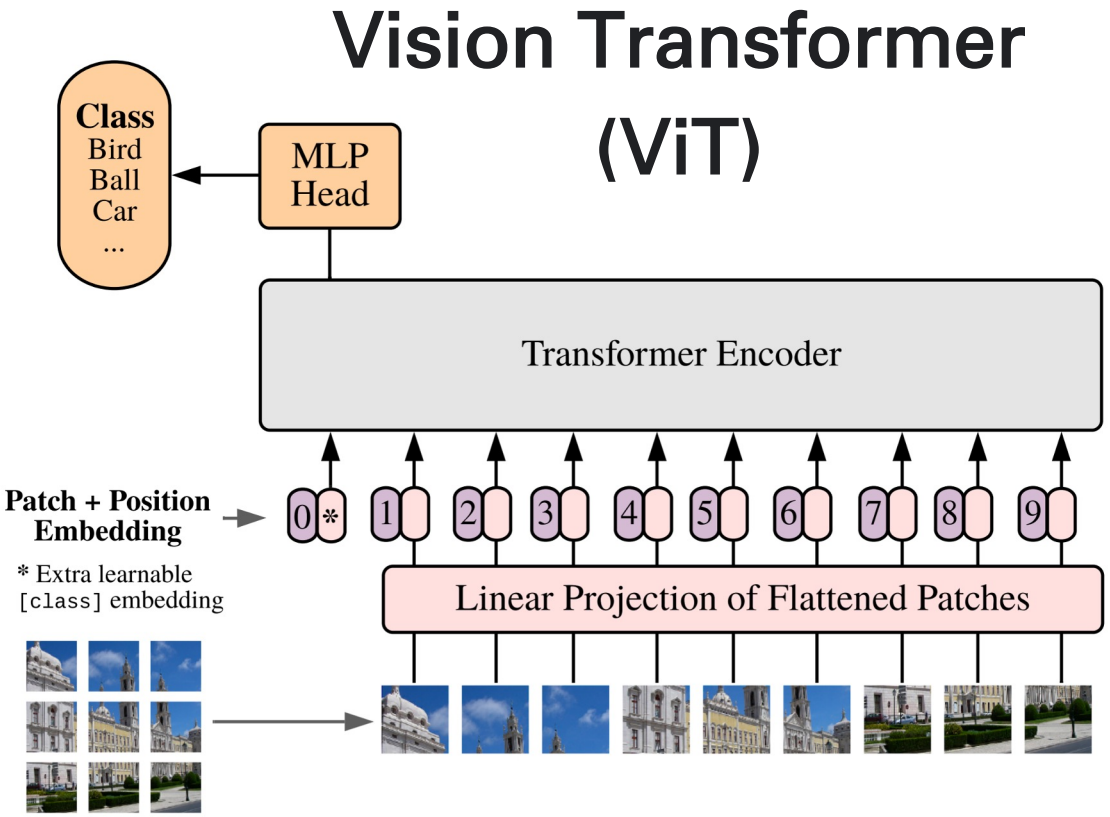
Many prior works attempted to introduce self-attention at the pixel level.

For 224px², that's 50k sequence length, too much!

Thus, most works restrict attention to local pixel neighborhoods, or as high-level mechanism on top of detections.

The **key breakthrough** in using the full Transformer architecture, standalone, was to **"tokenize" the image** by **cutting it into patches** of 16px², and treating each patch as a token, e.g. embedding it into input space.

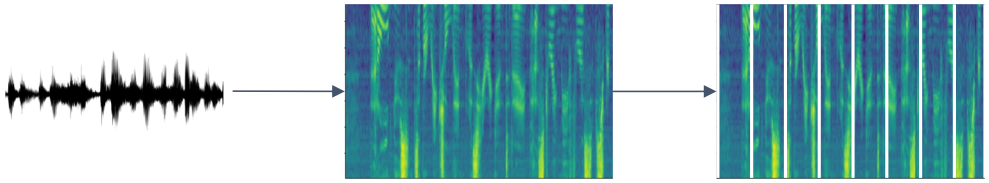
Transformer-based models take over the vision domain!



Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Pre-training in Speech (during Transformers)

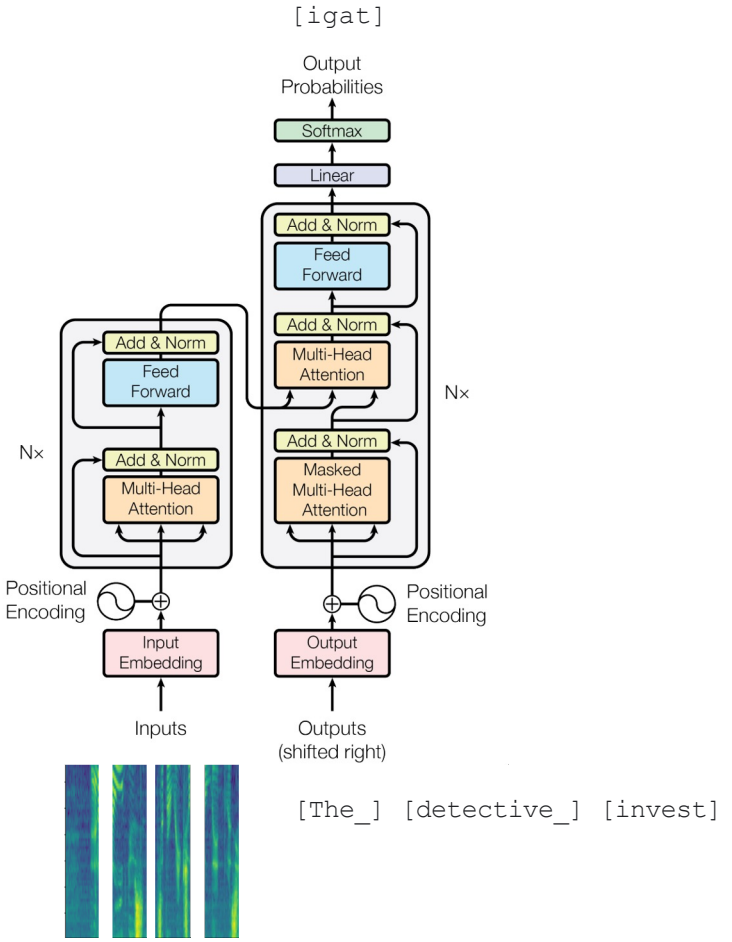
Largely the same story as in computer vision.
But with spectrograms instead of images.



Add a third type of block using convolutions, and slightly reorder blocks, but overall very transformer-like.

Exists as encoder-decoder variant, or as encoder-only variant with CTC loss.

Transformer-based models take over the speech domain!



Gulati et al. Conformer: Convolution-augmented Transformer for Speech Recognition. In INTERSPEECH 2020

Summary

- Attention is used to focus on parts of inputs/outputs
- It can be content/location-based and hard/soft
- Its three main distinct uses are
  - connecting encoder and decoder in sequence-to-sequence task
  - achieving scale-invariance and focus on image processing
  - self-attention can be a basic building block for neural nets, often replacing RNNs and CNNs [recent research, take it with a grain of salt]
- ViTs are an evolution, not a revolution. We can still fundamentally solve the same problems as with CNNs.
- Matrix multiply is more hardware-friendly than convolution, so ViTs with same FLOPs as CNNs can train and run much faster

Next lecture:
Graph Neural Networks