UNIVERSITY
OF OSLO

# IN4310/3310 Vision Transformers

dhananjt@ifi.uio.no

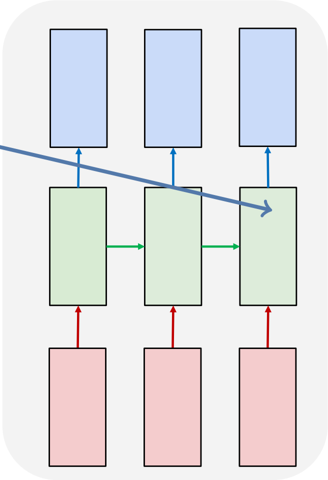
# Outline

# Outline

# RNNs suffer from a bottleneck problem

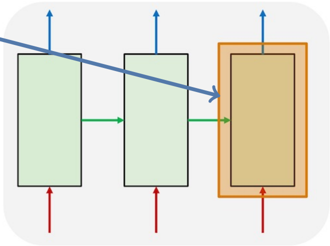
This vector must encode all the previous hidden states.

This keeps getting more difficult with longer sequences.



# Idea: Use all previous hidden states

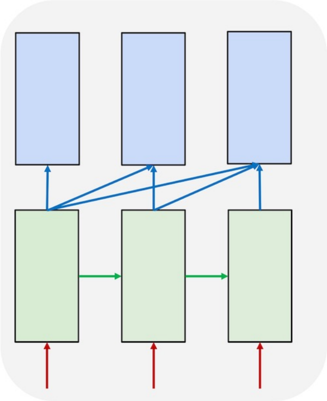
Instead of using only this hidden state as input, use all the previous hidden states as input.



# Idea: Use all previous hidden states

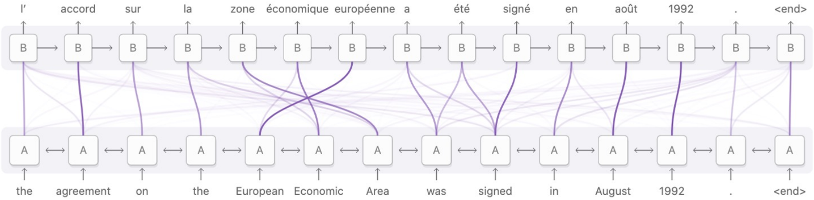
**Naïve method:** Connect the output layer to all previous hidden states.
It has at least three problems:

- **Additional parameters:** Adding connections to k hidden states instead of just one will increase the parameter count k-times for the layer.
- **Fixed length:** Since the parameters of a network are fixed, we can process hidden states only up to a certain length.
- **Training long sequences:** If the sequence length in the training data varies, some parameters might receive an insufficient amount of gradient updates to learn something useful.



# Attention mechanism

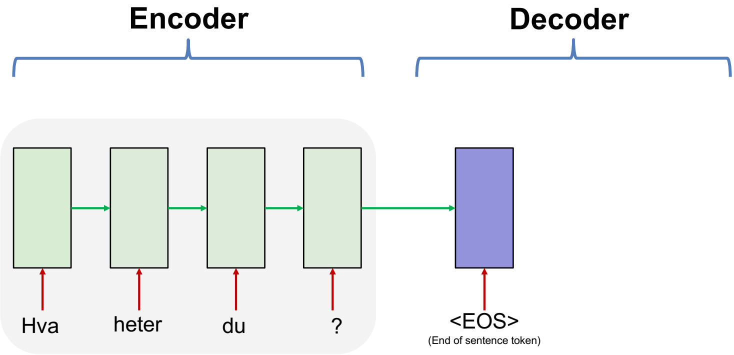
- When people say "attention mechanism", they usually refer to either the attention mechanism proposed by Bahdanau et al. for Neural Machine Translation (NMT) or one of its derived forms.
- Basic intuition: use all the hidden states of the input sentence but focus more on the relevant ones.




**Source:** Olah & Carter, "Attention and Augmented Recurrent Neural Networks", Distill, 2016.
http://doi.org/10.23915/distill.00001
Diagram derived from Fig. 3 of Bahdanau et al. 2014

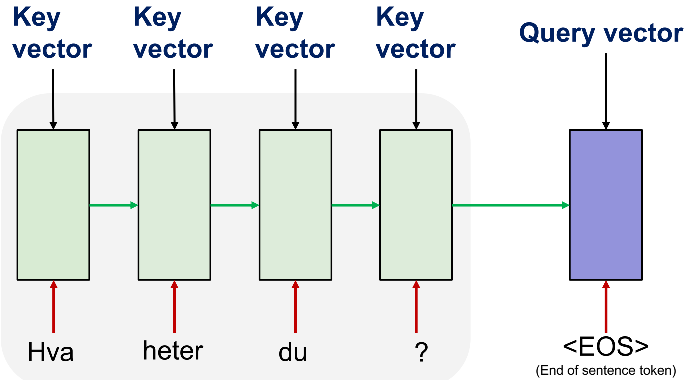
# Attention mechanism

- We will look at attention mechanism via Neural Machine Translation (NMT), i.e., translating text from a source language to a target language using neural networks.
- NMT typically uses an encoder-decoder architecture.
- The decoder makes use of the attention mechanism.

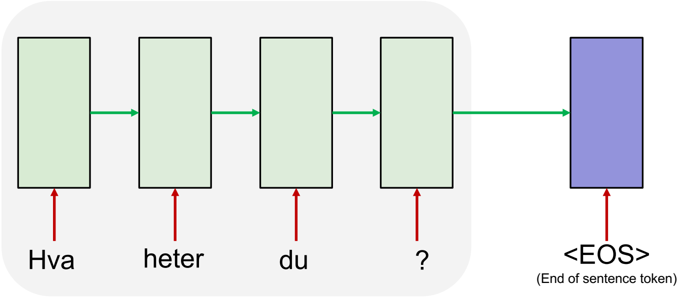


# Attention mechanism

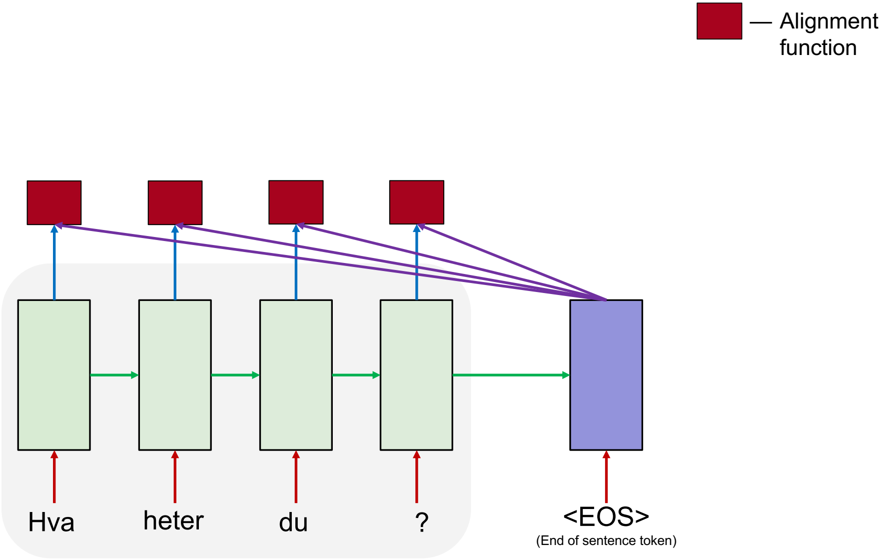
- The input to the attention mechanism is a single query vector and multiple key vectors. Additionally, we can also have value vectors.
- Each query-key pair gets a score that determines how much attention each key vector (or the value vector if present) will get.
- We will use the encoder's hidden states as both key and value vectors.



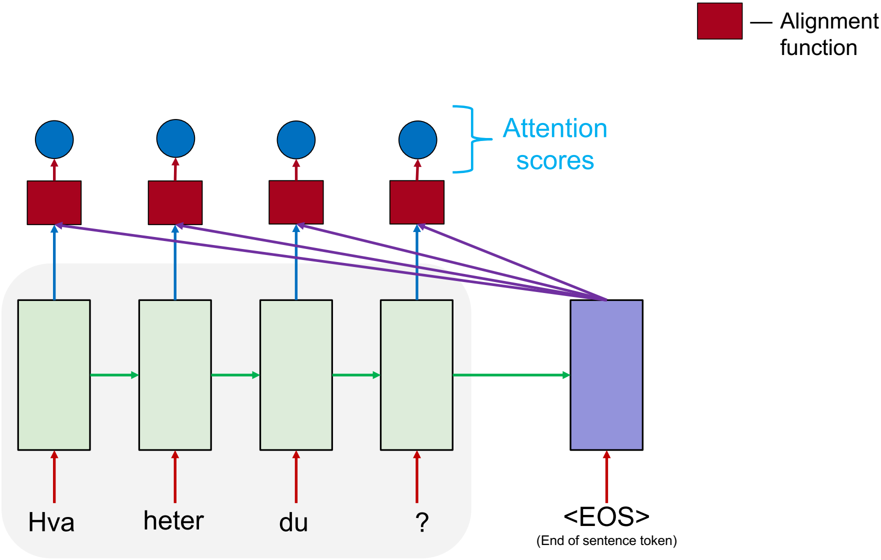
# Attention mechanism



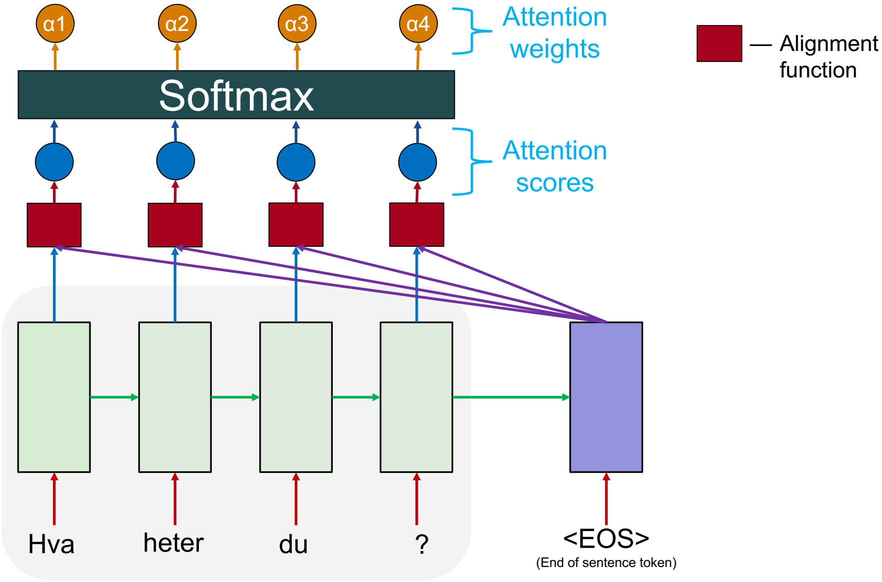
# Attention mechanism



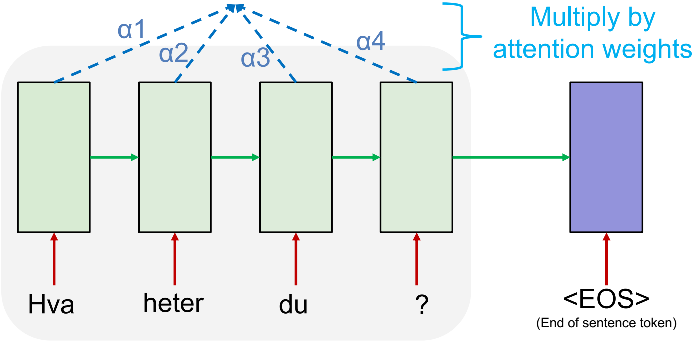
# Attention mechanism



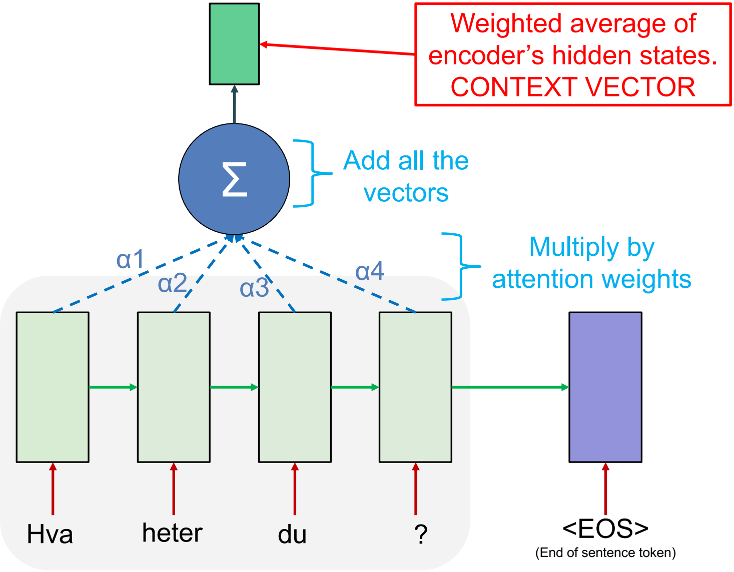
# Attention mechanism



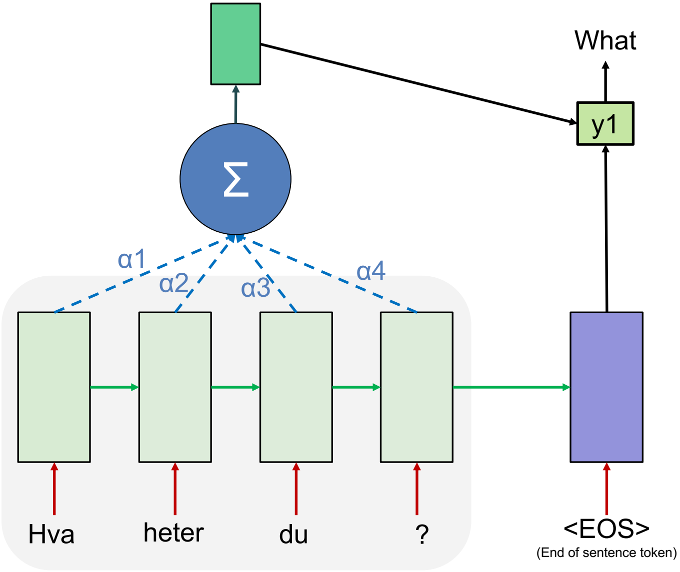
# Attention mechanism



# Attention mechanism

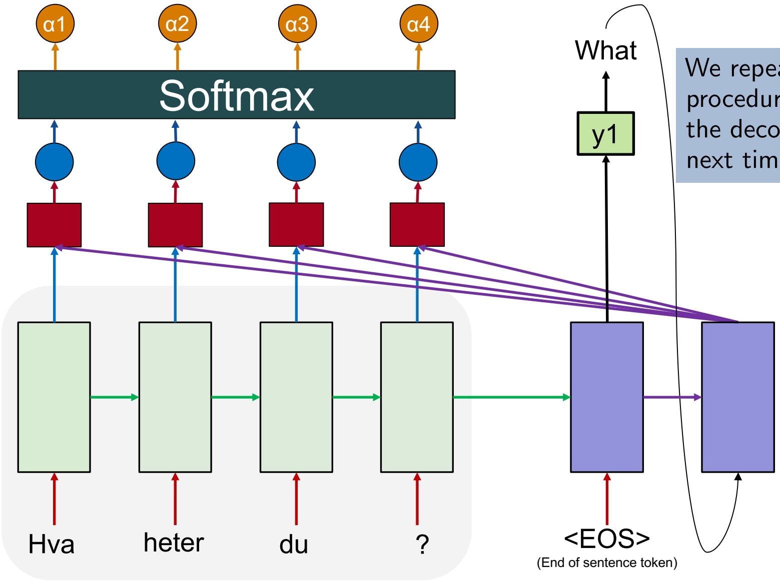


# Attention mechanism



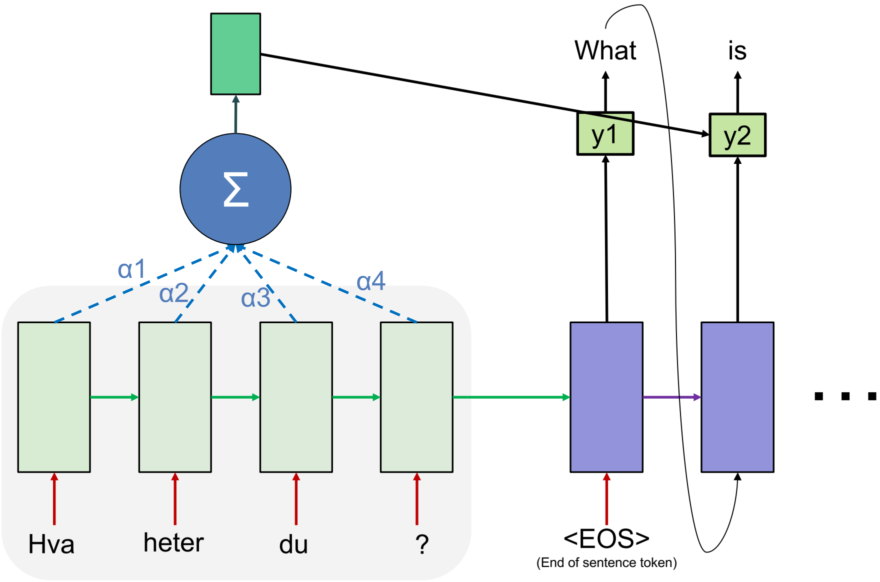

The context vector and the hidden state of the decoder are used to generate output y1.

# Attention mechanism




We repeat the procedure for the decoder's next time step.

# Attention mechanism



## Attention mechanism: alignment function

- Alignment function takes a query (q) and a key (k) vector as input and outputs a score for the query-key pair.
- Popular alignment functions:
  - Additive/concat attention: $\mathrm{a(q,k)} = \mathrm{v^T \tanh(W_a\,[q;k])}$ Bahdanau et al. 2014
  - General attention: $\mathrm{a(q,k)} = \mathrm{q^T W_a k}$ Luong et al. 2015
  - Dot-product attention: $\mathrm{a(q,k)} = \mathrm{q^T k}$ Luong et al. 2015
  - Scaled dot-product attention: $\mathrm{a(q,k)} = \frac{\mathrm{q^T k}}{\sqrt{\mathrm{d_k}}}$ where $\mathrm{d_k}$ = length of k Vaswani et al. 2017
- Transformers use scaled dot-product attention in self-attention.

# Attention mechanism

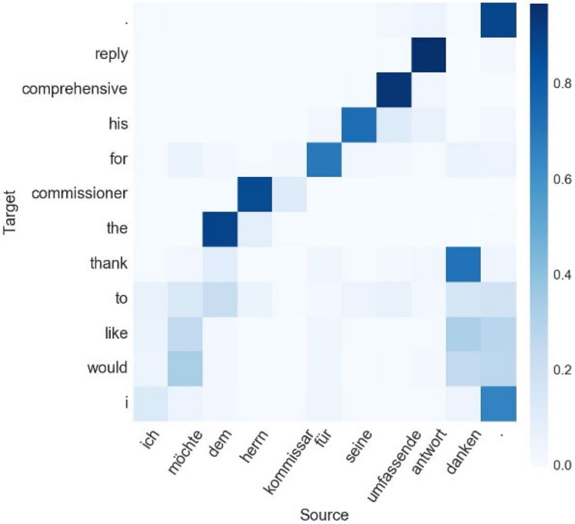
- The attention mechanism solves the bottleneck problem.
  - The encoder doesn't need to store all the information in the last hidden state, as the decoder now looks directly at all the hidden states.
- The attention mechanism helps with the vanishing gradient problem.
  - Gradients can now be transferred directly from the decoder to all the time steps in the encoder.
- The attention mechanism provides some interpretability.
  - By looking at the attention weights, we can determine what the decoder focused on.

# Attention mechanism

- The attention mechanism solves the bottleneck problem.
  - The encoder doesn't need to store all the information in the last hidden state, as the decoder now looks directly at all the hidden states.
- The attention mechanism helps with the vanishing gradient problem.
  - Gradients can now be transferred directly from the decoder to all the time steps in the encoder.
- The attention mechanism provides some interpretability.
  - By looking at the attention weights, we can determine what the decoder focused on.

- The attention mechanism solves the bottleneck problem.
    - The encoder doesn't need to store all the information in the last hidden state, as the decoder now looks directly at all the hidden states.
- The attention mechanism helps with the vanishing gradient problem.
    - Gradients can now be transferred directly from the decoder to all the time steps in the encoder.
- The attention mechanism provides some interpretability.
    - By looking at the attention weights, we can determine what the decoder focused on.

# Attention mechanism: interpretability example




**Source:** Ghader, Hamidreza, and Christof Monz. "What does Attention in Neural Machine Translation Pay Attention to?" Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers). 2017.

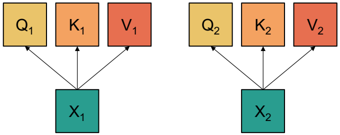
# Attention mechanism: interpretability example


**Source:** Xu, Kelvin, et al. "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention." arXiv preprint arXiv:1502.03044 (2015).

# Outline

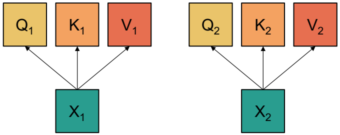
# Self-Attention: Step 1: Compute Query, Key, Value vectors

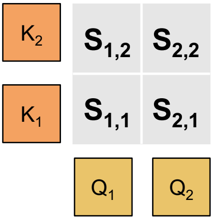
- In step 1, each input vector Xi is multiplied with 3 different matrices to get query, key, and value vectors.
- You can imagine this as having 3 `torch.nn.Linear` layers:
  - $\mathtt{L_Q}$ `= torch.nn.Linear(size_of_x_vector, size_of_query_vector)`
  - $\mathtt{L_K}$ `= torch.nn.Linear(size_of_x_vector, size_of_key_vector)`
  - $\mathtt{L_V}$ `= torch.nn.Linear(size_of_x_vector, size_of_value_vector)`
  - $\mathtt{Q = L_Q(X)} \quad \mathtt{K = L_Q(X)} \quad \mathtt{V = L_Q(X)}$
- Query, key, and value vectors typically have the same dimensions.



# Self-Attention: Step 2a: Compute score

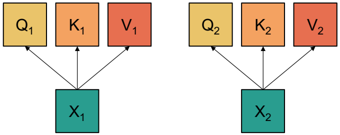
- Apply *scaled dot product attention* using query and key vectors to get scores for each pair.
- Each Ki, Qi pair generates a score.
  - $\mathtt{S}_{1,1} = \mathtt{dot\_product}(\mathtt{Q}_1, \mathtt{K}_1)$ $\mathtt{S}_{1,2} = \mathtt{dot\_product}(\mathtt{Q}_1, \mathtt{K}_2)$
  - $\mathtt{S}_{2,1} = \mathtt{dot\_product}(\mathtt{Q}_2, \mathtt{K}_1)$ $\mathtt{S}_{2,2} = \mathtt{dot\_product}(\mathtt{Q}_2, \mathtt{K}_2)$
- Each score is divided by the square root of the dimension of the key vector.
  - $\mathsf{S}_{\mathrm{i,j}} = \mathsf{S}_{\mathrm{i,j}} \ / \ \sqrt{\text{dimension of } \mathrm{K_i} \text{ vectors}}$
  - This is done to stabilise gradients during backpropagation.

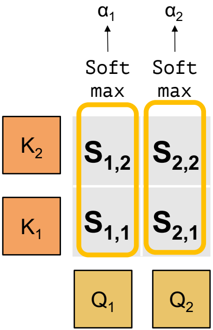


# Self-Attention: Step 2b: Compute softmax score

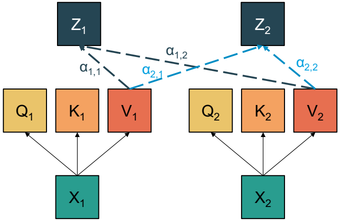
- Next, scores corresponding to each query vector are passed through a softmax layer to get softmax scores.
  - $\alpha_1 = \texttt{Softmax}(\texttt{S}_{1,1}, \texttt{S}_{1,2})$
  - $\alpha_2 = \texttt{Softmax}(\texttt{S}_{2,1}, \texttt{S}_{2,2})$

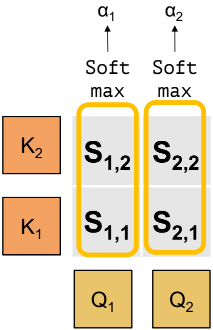


## Self-Attention: Step 2b: Compute softmax score

- For every Xi, compute the weighted average of the value vectors using the softmax scores (corresponding to Xi) as weights.
  - $\mathtt{Z}_1 = (\alpha_{1,1} * \mathtt{V}_1) + (\alpha_{1,2} * \mathtt{V}_2)$
  - $\mathtt{Z}_2 = (\alpha_{2,1} * \mathtt{V}_2) + (\alpha_{2,2} * \mathtt{V}_2)$

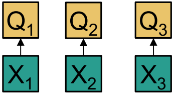


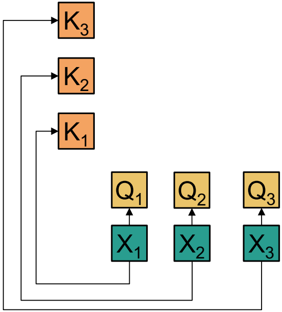
# Self-Attention: Overview

# Self-Attention: Overview

Query vectors: $\mathtt{Q} = \mathtt{X}\mathtt{W}_{\mathtt{Q}}$

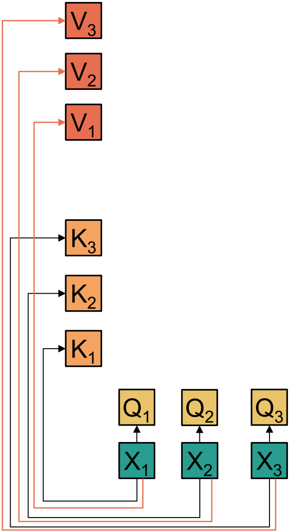


# Self-Attention: Overview

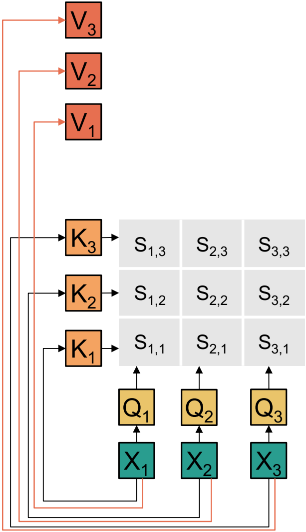



Query vectors: $\mathtt{Q} = \mathtt{X}\mathtt{W}_\mathtt{Q}$
Key vectors: $\mathtt{K} = \mathtt{X}\mathtt{W}_\mathtt{K}$

# Self-Attention: Overview

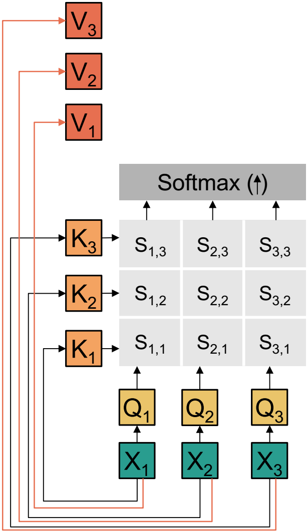



Query vectors: $\mathtt{Q} = \mathtt{X}\mathtt{W}_\mathtt{Q}$
Key vectors: $\mathtt{K} = \mathtt{X}\mathtt{W}_\mathtt{K}$
Value vectors: $\mathtt{V} = \mathtt{X}\mathtt{W}_\mathtt{V}$



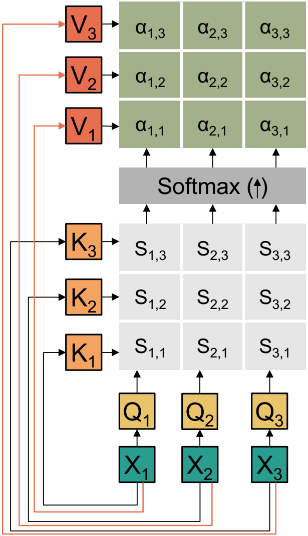

Query vectors: $\mathtt{Q} = \mathtt{XW_Q}$
Key vectors: $\mathtt{K} = \mathtt{XW_K}$
Value vectors: $\mathtt{V} = \mathtt{XW_V}$
Scores: $\mathtt{S_{i,j}} = \mathtt{Q_i} \cdot \mathtt{K_j} / \sqrt{\mathtt{D}}$

# Self-Attention: Overview



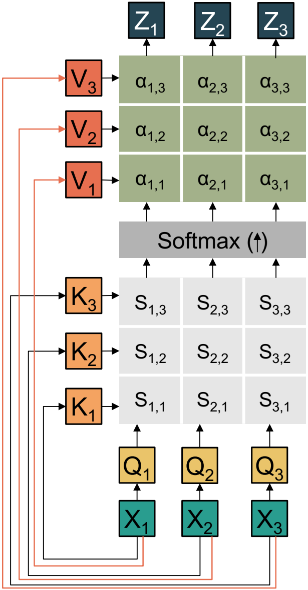

Query vectors: $\mathtt{Q} = \mathtt{XW_Q}$
Key vectors: $\mathtt{K} = \mathtt{XW_K}$
Value vectors: $\mathtt{V} = \mathtt{XW_V}$
Scores: $\mathtt{S_{i,j}} = \mathtt{Q_i} \cdot \mathtt{K_j} / \sqrt{\mathtt{D}}$

# Self-Attention: Overview




Query vectors: $\mathtt{Q} = \mathtt{X}\mathtt{W}_\mathtt{Q}$
Key vectors: $\mathtt{K} = \mathtt{X}\mathtt{W}_\mathtt{K}$
Value vectors: $\mathtt{V} = \mathtt{X}\mathtt{W}_\mathtt{V}$
Scores: $\mathtt{S}_{\mathtt{i},\mathtt{j}} = \mathtt{Q}_\mathtt{i} \cdot \mathtt{K}_\mathtt{j} / \sqrt{\mathtt{D}}$

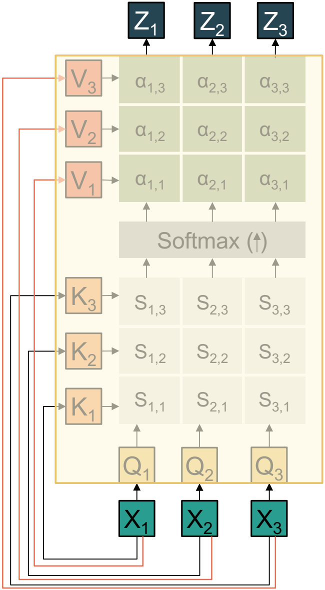
# Self-Attention: Overview




$$\mathtt{Z}_1 = (\alpha_{1,1} * \mathtt{V}_1) + (\alpha_{1,2} * \mathtt{V}_2) + (\alpha_{1,3} * \mathtt{V}_3))$$
$$\mathtt{Z}_2 = (\alpha_{2,1} * \mathtt{V}_1) + (\alpha_{2,2} * \mathtt{V}_2) + (\alpha_{2,3} * \mathtt{V}_3))$$
$$\mathtt{Z}_3 = (\alpha_{3,1} * \mathtt{V}_1) + (\alpha_{3,2} * \mathtt{V}_2) + (\alpha_{3,3} * \mathtt{V}_3))$$

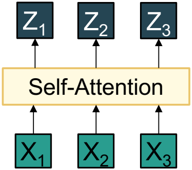
Query vectors: $\mathtt{Q} = \mathtt{X}\mathtt{W}_\mathtt{Q}$
Key vectors: $\mathtt{K} = \mathtt{X}\mathtt{W}_\mathtt{K}$
Value vectors: $\mathtt{V} = \mathtt{X}\mathtt{W}_\mathtt{V}$
Scores: $\mathtt{S}_{\mathtt{i},\mathtt{j}} = \mathtt{Q}_\mathtt{i} \cdot \mathtt{K}_\mathtt{j} / \sqrt{\mathtt{D}}$

# Self-Attention: Overview

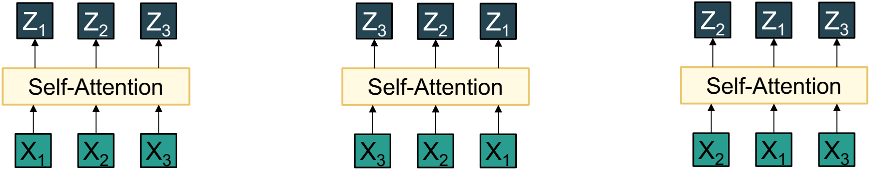



$$Z_1 = (\alpha_{1,1} * V_1) + (\alpha_{1,2} * V_2) + (\alpha_{1,3} * V_3))$$
$$Z_2 = (\alpha_{2,1} * V_1) + (\alpha_{2,2} * V_2) + (\alpha_{2,3} * V_3))$$
$$Z_3 = (\alpha_{3,1} * V_1) + (\alpha_{3,2} * V_2) + (\alpha_{3,3} * V_3))$$

Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$
Value vectors: $V = XW_V$
Scores: $S_{i,j} = Q_i \cdot K_j / \sqrt{D}$



# Self-Attention: Permutation equivariance

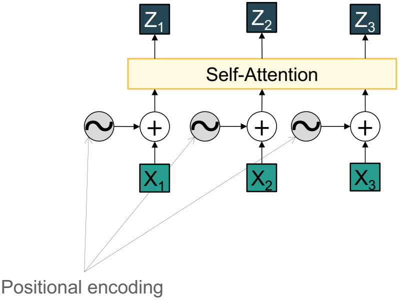



- Self-attention layer is permutation equivariant.
- Self-attention layer doesn't care about the order of the input sequence.

**Problem:** How to encode the order of input sequence when it matters, like in language or images?

# Positional encoding

- There are multiple ways of encoding position:
  - Absolute/fixed positional encoding.
  - Relative positional encoding.
  - A hybrid of relative and absolute positional encoding.
  - Learned positional encoding.
  - etc.
- Positional encodings are vectors with the same dimension as the input vectors.
- Positional encodings are usually added to the input vectors.

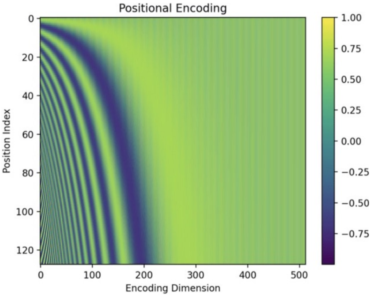


# Positional encoding: Absolute positional encoding

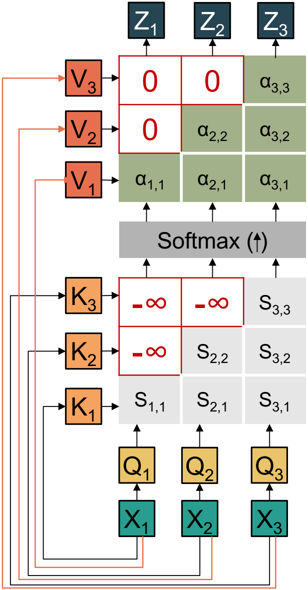
- Absolute positional encoding at position **pos** for dimension i = 0...(d/2 - 1):
  - $PE(pos, 2i) = \sin\left(pos/10000^{2i/d}\right)$
  - $PE(pos, 2i+1) = \cos\left(pos/10000^{2i/d}\right)$
- A 512-dimensional encoding for i in the sequence:

| | | | | |
|---|---|---|---|---|
| $\sin\left(\frac{i}{10000^{0/512}}\right)$ | $\cos\left(\frac{i}{10000^{0/512}}\right)$ | $\sin\left(\frac{i}{10000^{2/512}}\right)$ | $\cos\left(\frac{i}{10000^{2/512}}\right)$ | $\ldots$ |

| | | |
|---|---|---|
| $\ldots$ | $\sin\left(\frac{i}{10000^{510/512}}\right)$ | $\cos\left(\frac{i}{10000^{510/512}}\right)$ |



# Self-Attention: Overview




$$Z_1 = (\alpha_{1,1} * V_1)$$
$$Z_2 = (\alpha_{2,1} * V_1) + (\alpha_{2,2} * V_2)$$
$$Z_3 = (\alpha_{3,1} * V_1) + (\alpha_{3,2} * V_2) + (\alpha_{3,3} * V_3))$$

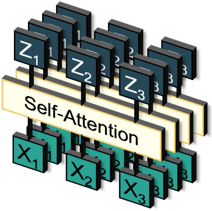
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$
Value vectors: $V = XW_V$
Scores: $S_{i,j} = Q_i \cdot K_j / \sqrt{D}$

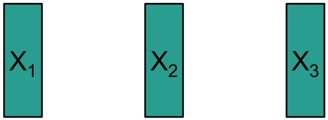
- Don't let vectors look ahead in the sequence at future vectors.
- Manually set scores to -infinity.
- Especially relevant for Language Modelling.

# Outline

# Multi-head Self-Attention

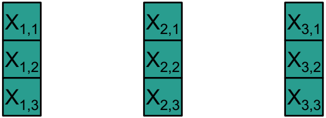
- Use H independent "attention heads" in parallel.
    - Attention heads do NOT share parameters.
- Each Input embedding is split into H parts.
- Each part is then processed by an attention head.



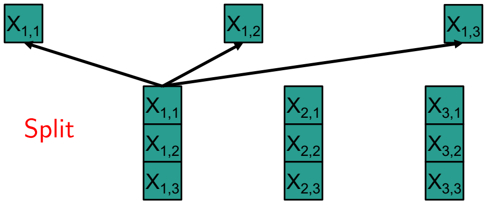



# Multi-head Self-Attention

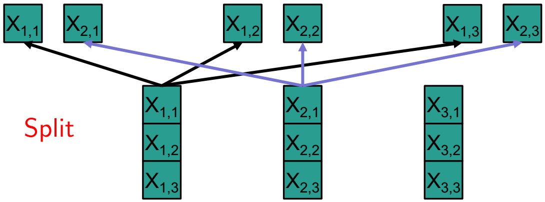
Split



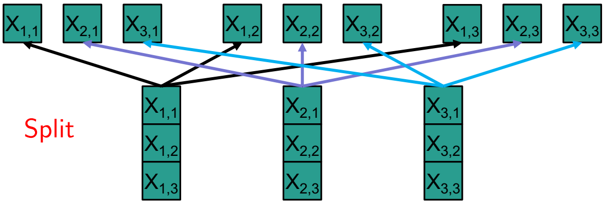
# Multi-head Self-Attention



# Multi-head Self-Attention

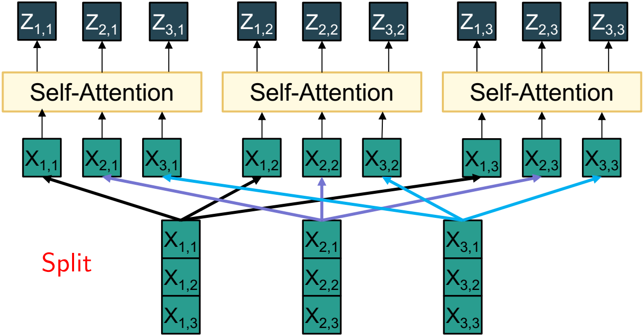


# Multi-head Self-Attention

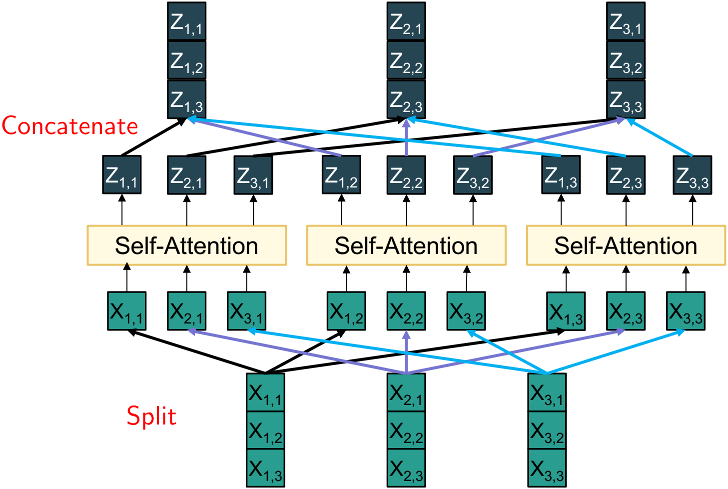


# Multi-head Self-Attention

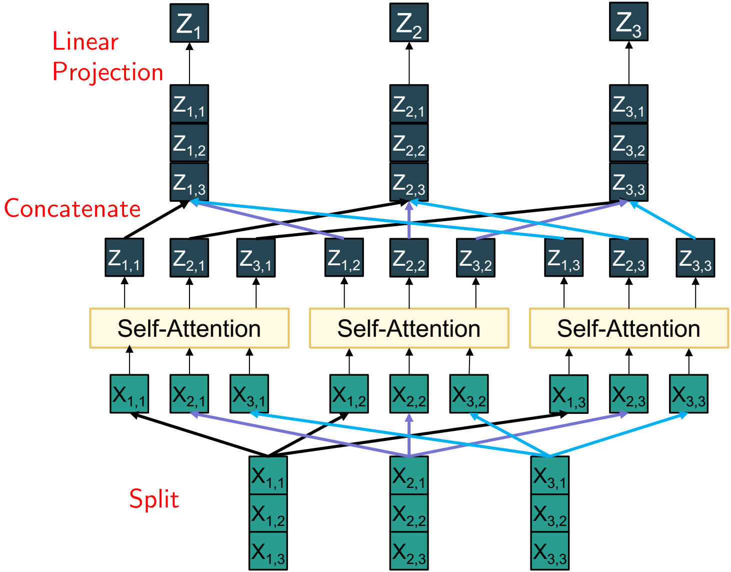
Run self-attention in parallel on each set of input vectors using a different set of parameters for each self-attention head.



# Multi-head Self-Attention

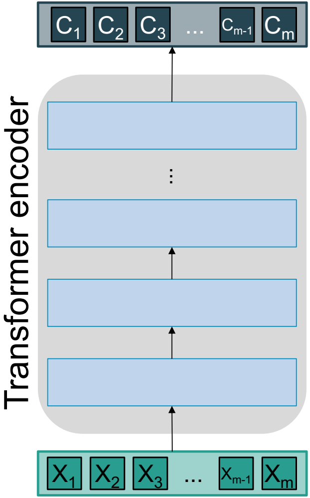


## Multi-head Self-Attention



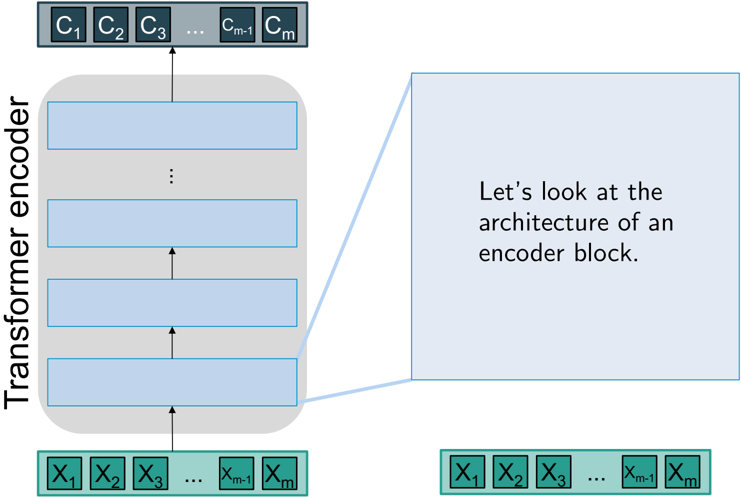
# Outline

# Transformer encoder block

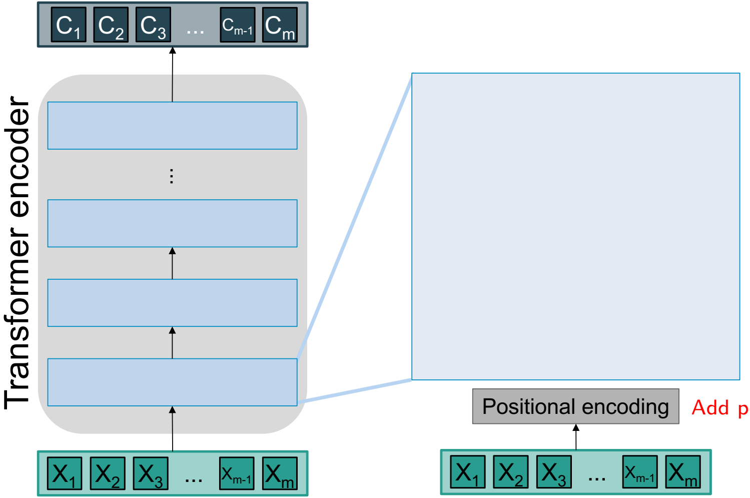



Transformer encoder is made up of a series of N encoder blocks. In the original model (Vaswani et al.) N=6

# Transformer encoder block

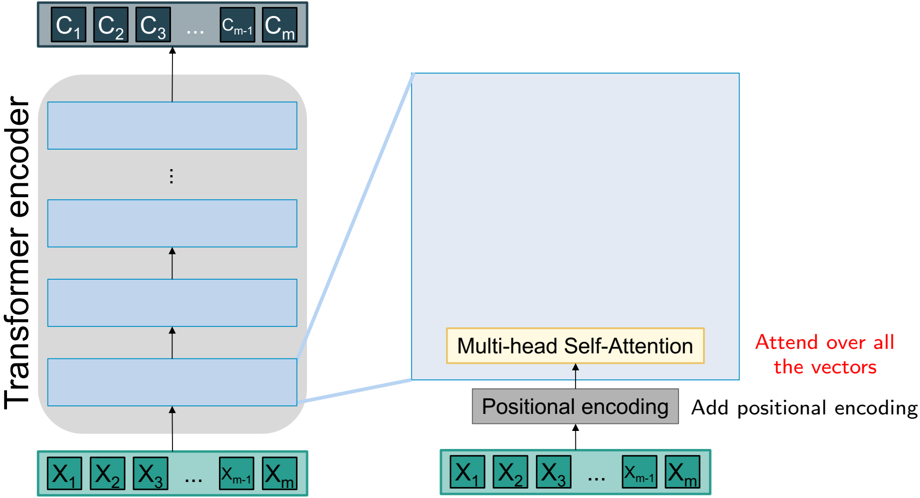


# Transformer encoder block

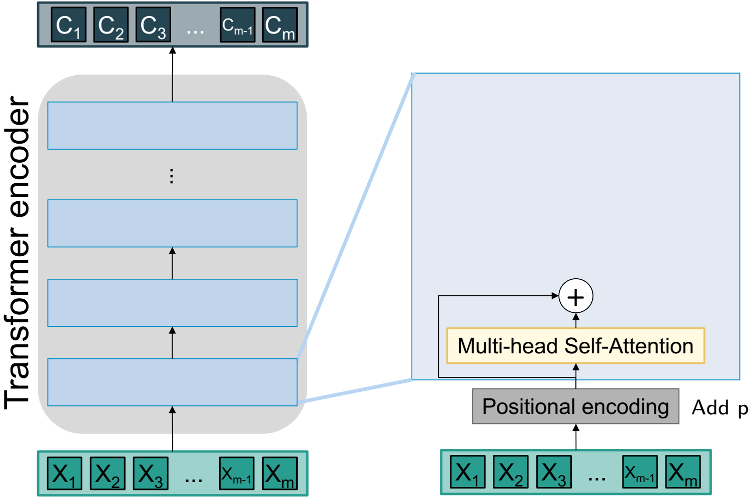



Add positional encoding

# Transformer encoder block



# Transformer encoder block

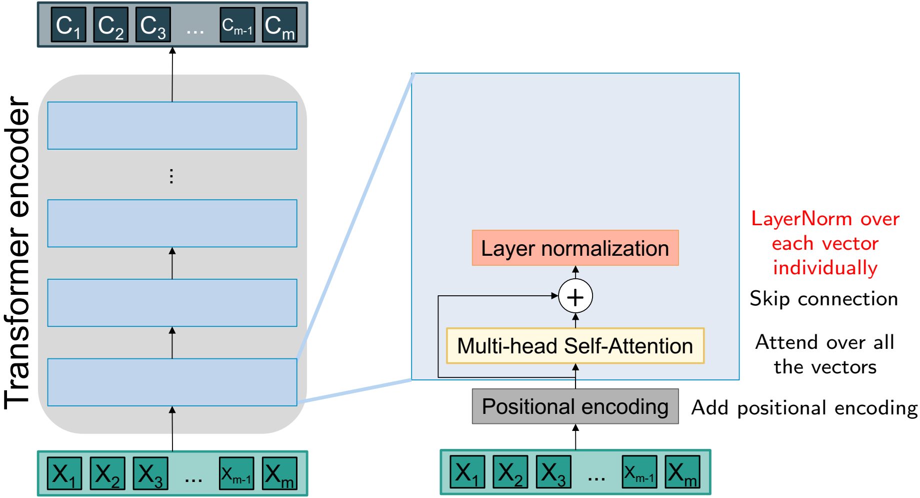



Skip connection

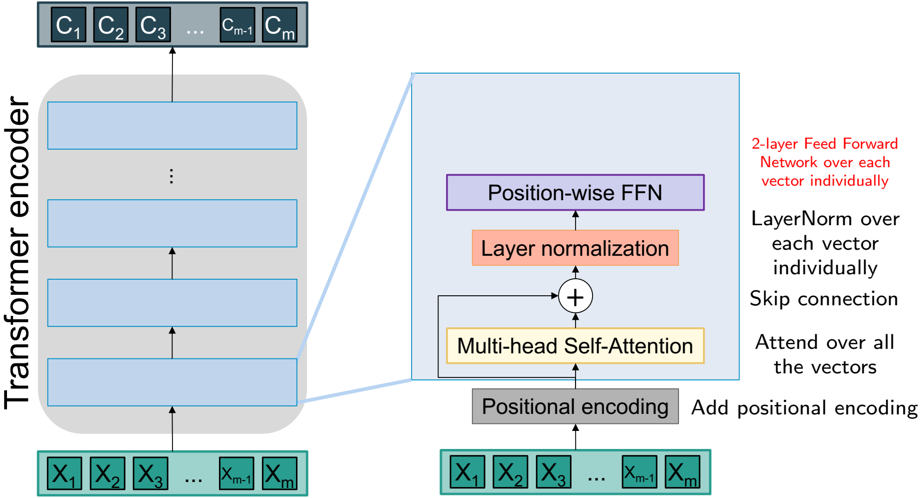
Attend over all the vectors

Add positional encoding

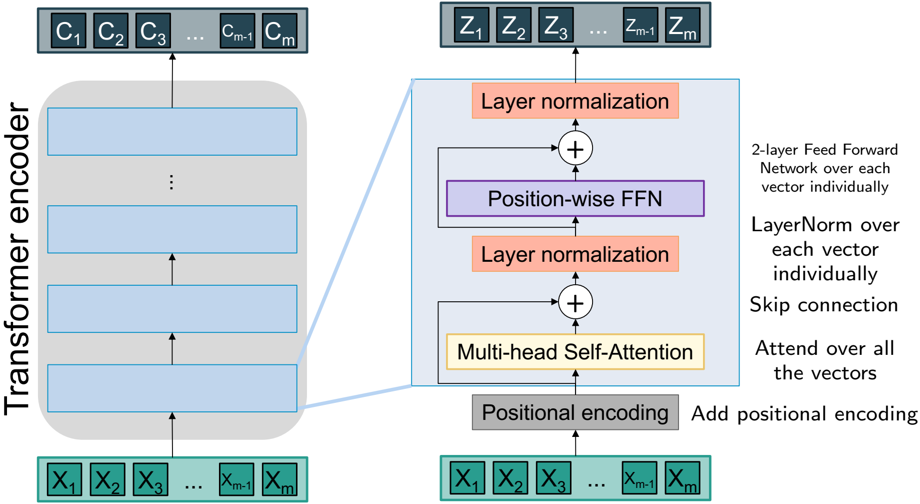
# Transformer encoder block



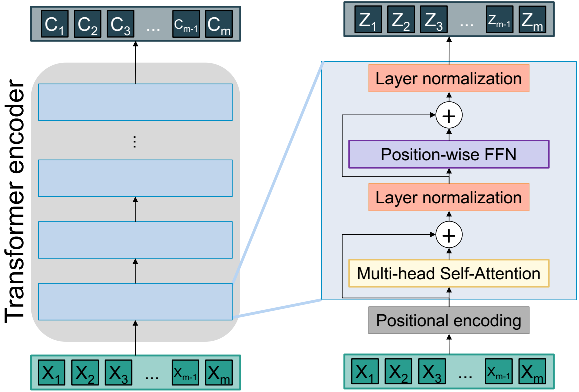
# Transformer encoder block



# Transformer encoder block



# Transformer encoder block




**Transformer Encoder Block:**

**Input:** Set of vectors X
**Output:** Set of vectors Z

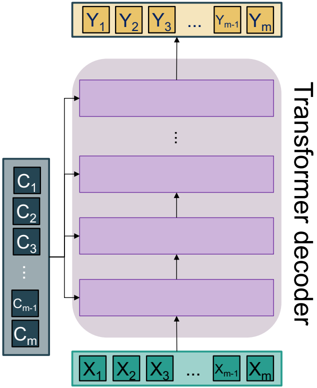
Self-attention is the only interaction between vectors.

LayerNorm and FFN operate independently per vector.

Highly parallelisable but has high memory usage.

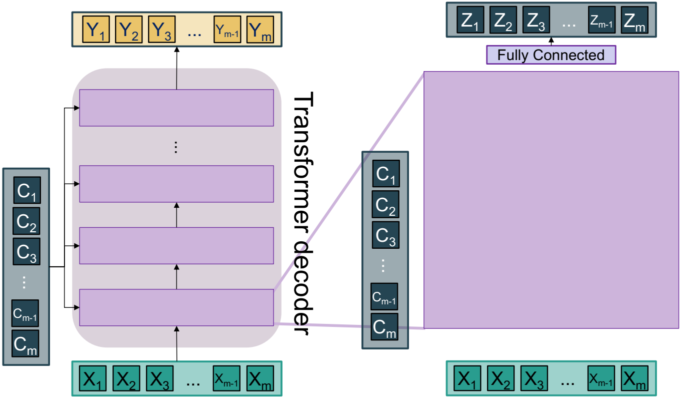
# Outline

# Transformer decoder block



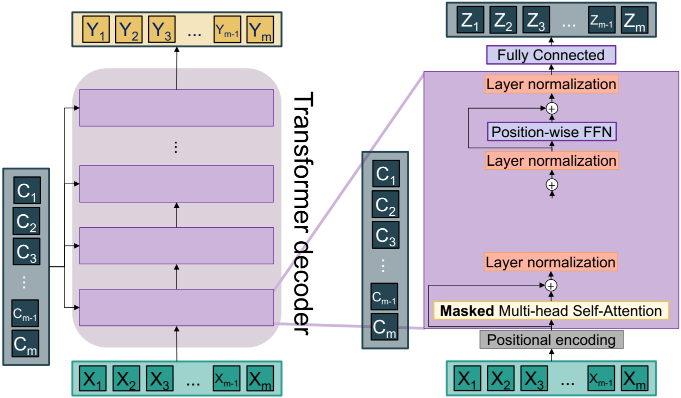

Transformer decoder is made up of a series of N decoder blocks. In the original model (Vaswani et al.) N=6

# Transformer decoder block



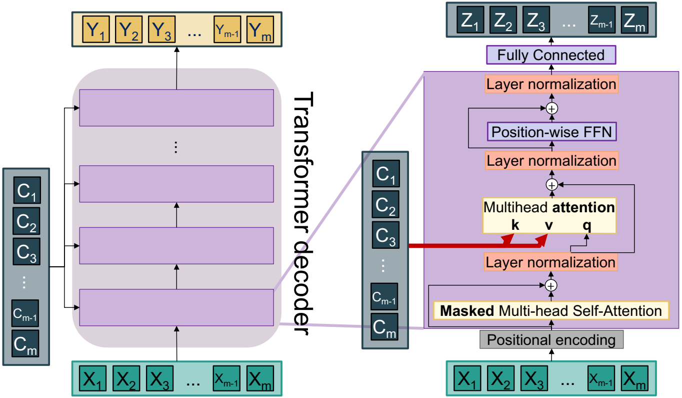

Let's look at the architecture of a decoder block.

# Transformer decoder block



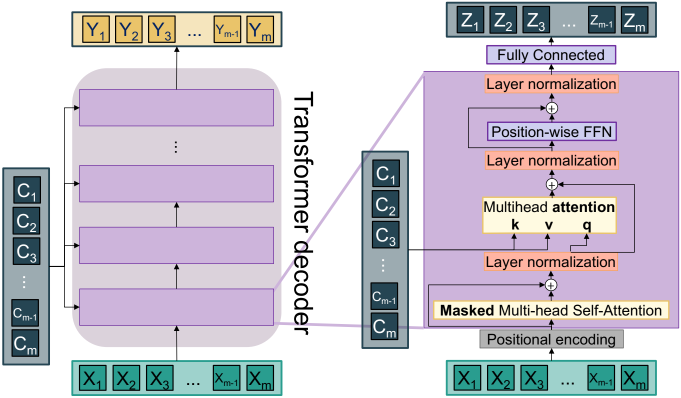

Most of the block is the same as the encoder block.

# Transformer decoder block




Multi-head attention block attends over encoder's outputs.

# Transformer decoder block




**Transformer Decoder Block:**

**Input:** Set of vectors X and set of context vectors C

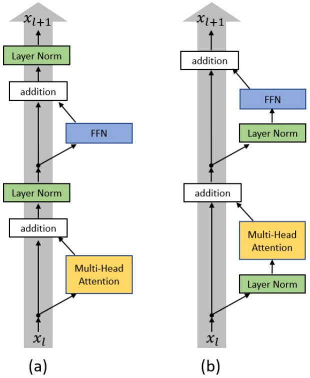
**Output:** Set of vectors Z

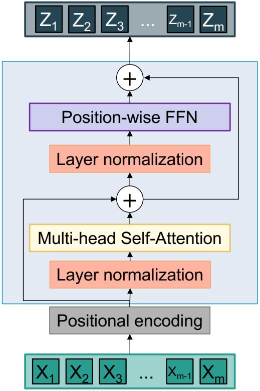
Masked Self-attention interacts only with past inputs.

Multi-head attention is NOT self-attention. It attends over the encoder's outputs.

Highly parallelisable but has high memory usage.

# Transformer block: Pre-Norm Transformer




**Pre-Norm Transformer:**

**Layer normalization** is inside residual (skip) connections.

Gives more stable training.

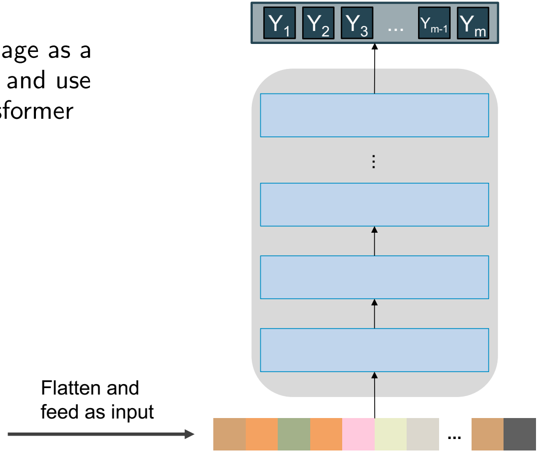
(a) Post-norm transformer (b) Pre-norm transformer.
**Image Source:** https://arxiv.org/pdf/2002.04745.pdf

# Outline

# How to use transformers for images?

**Idea:** Treat an image as a sequence of pixels and use the standard transformer

Photo by Josh Hild on Unsplash



# How to use transformers for images?

**Problem:** Memory usage!

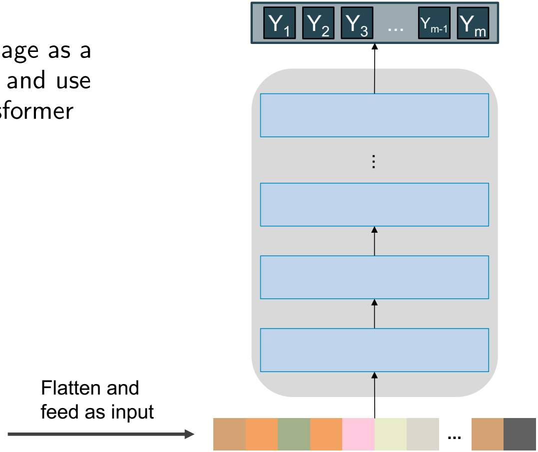
N x N image has $N^2$ pixels (sequence length).

Self-attention is an $O(N^2)$ operation.

Therefore, self-attention would need $N^4$ elements!

**Idea:** Treat an image as a sequence of pixels and use the standard transformer

Photo by Josh Hild on Unsplash



# Idea: Use standard transformer on patches

Photo by Josh Hild on Unsplash

# Idea: Use standard transformer on patches

Photo by Josh Hild on Unsplash

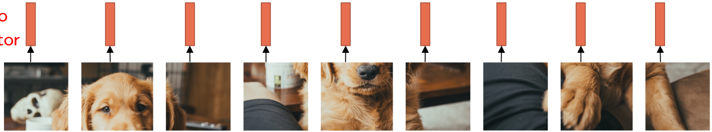
# Vision Transformer (ViT)

N input patches, each of shape 3x16x16

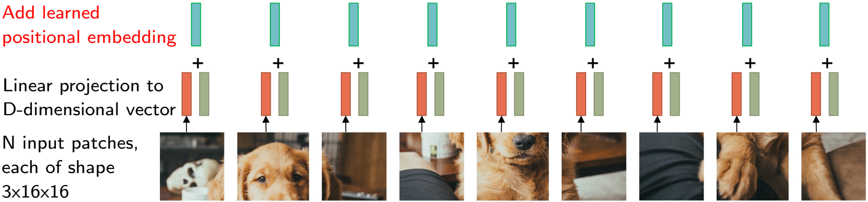
# Vision Transformer (ViT)

Linear projection to
D-dimensional vector

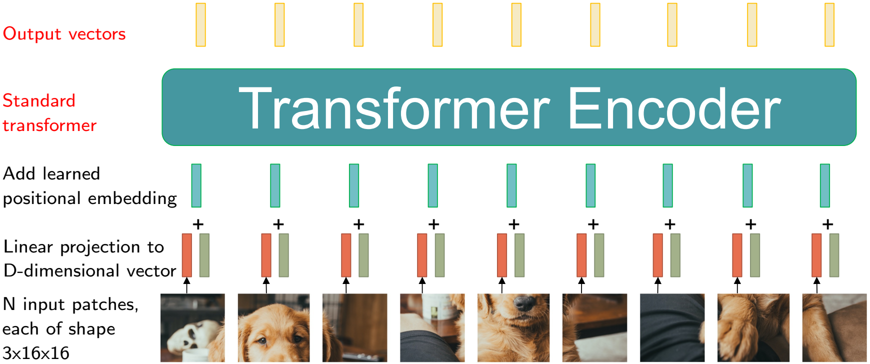
N input patches,
each of shape
3x16x16



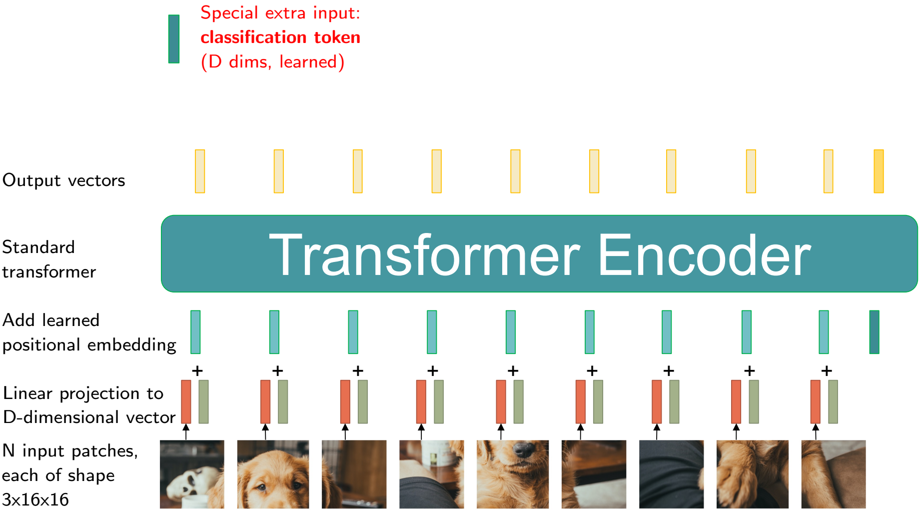
# Vision Transformer (ViT)



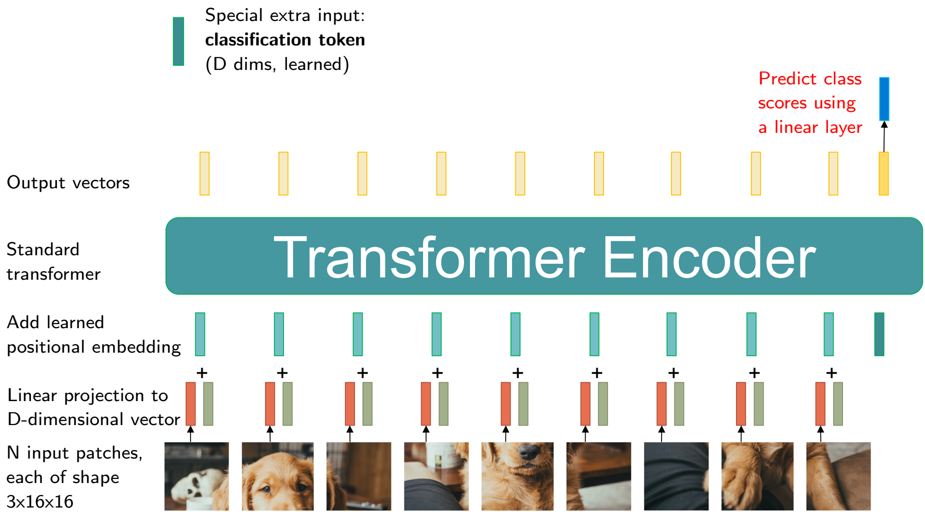
# Vision Transformer (ViT)



# Vision Transformer (ViT)



# Vision Transformer (ViT)



# Outline

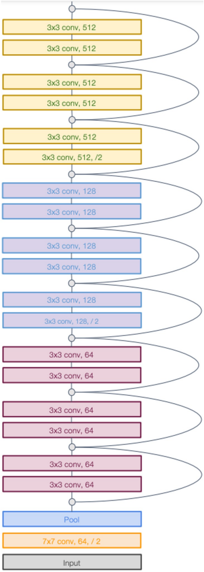
# ViT vs ConvNet

**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 ConvNets, as we go deeper, resolution decreases and #channels increase. (Hierarchical architecture)

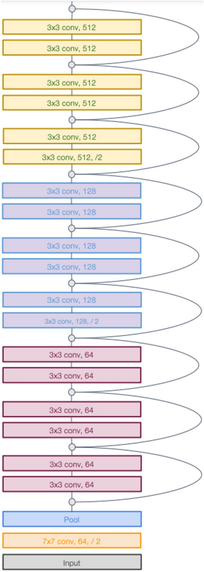
# ViT vs ConvNet

**Stage 3:**
256 x 14 x 14

**Stage 2:**
128 x 28 x 28

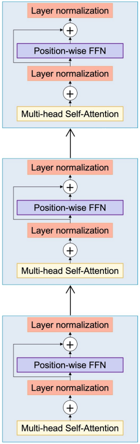
**Stage 1:**
64 x 56 x 56

**Input:**
3 x 224 x 224




In most ConvNets, as we go deeper, resolution decreases and #channels increase. (Hierarchical architecture)

In ViT, all blocks have the same resolution and number of channels




**3rd Block:**
768 x 14 x 14

**2nd Block:**
768 x 14 x 14

**1st Block:**
768 x 14 x 14

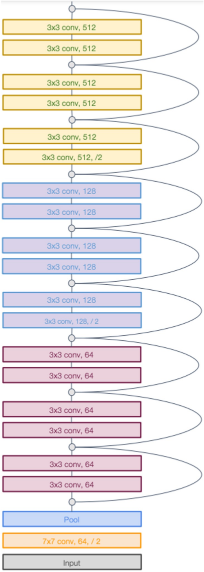
# ViT vs ConvNet

**Stage 3:**
256 x 14 x 14

**Stage 2:**
128 x 28 x 28

**Stage 1:**
64 x 56 x 56

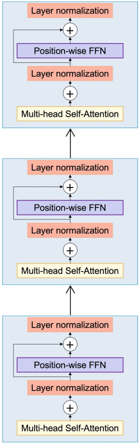
**Input:**
3 x 224 x 224




In most ConvNets, as we go deeper, resolution decreases and #channels increase. (Hierarchical architecture)

In ViT, all blocks have the same resolution and number of channels

Can we build a hierarchical ViT model?

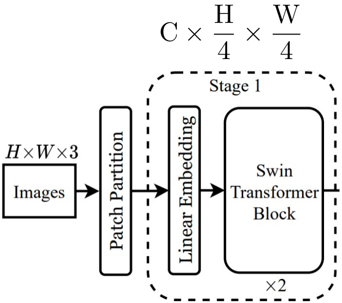



**3rd Block:**
768 x 14 x 14

**2nd Block:**
768 x 14 x 14

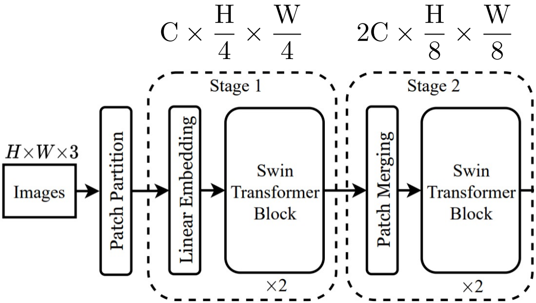
**1st Block:**
768 x 14 x 14

# Hierarchical ViT: Swin Transformer




Divide image into 4x4 patches and project to C dimensions

# Hierarchical ViT: Swin Transformer




Divide image into 4x4 patches and project to C dimensions

Merge 2x2 neighbourhoods. Now patches are (effectively) 8x8

**Patch Merging:**

1. For every group of 2x2 patches, concatenate the corresponding 4 vectors.
2. Pass the concatenated vectors (of size 4C) through a linear layer to get vectors of size 2C.

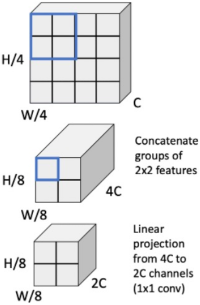
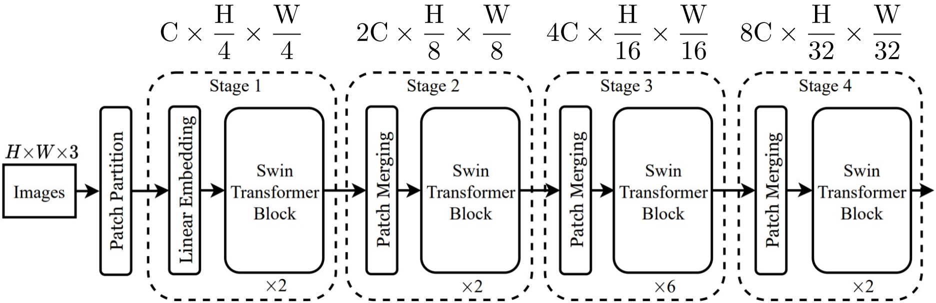



Image Source: EECS 498-007 / 598-005

# Hierarchical ViT: Swin Transformer




Divide image into 4x4 patches and project to C dimensions

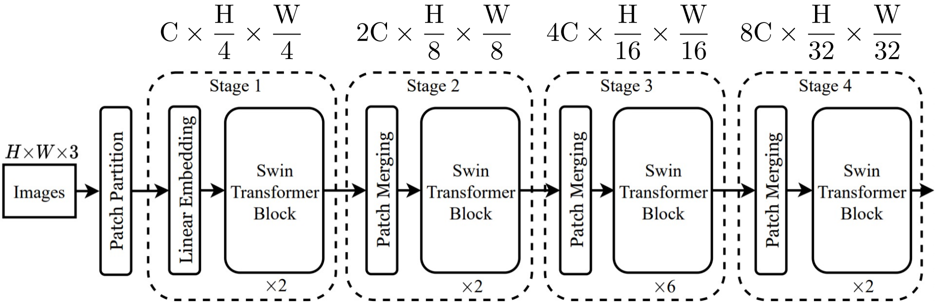
Merge 2x2 neighbourhoods. Now patches are (effectively) 8x8

Merge 2x2 neighbourhoods. Now patches are (effectively) 16x16

Merge 2x2 neighbourhoods. Now patches are (effectively) 32x32

# Hierarchical ViT: Swin Transformer

**Problem:** 224x224 image with 56x56 grid of 4x4 patches $\Rightarrow$ attention matrix has $56^4 = 9.8$M elements




Divide image into 4x4 patches and project to C dimensions

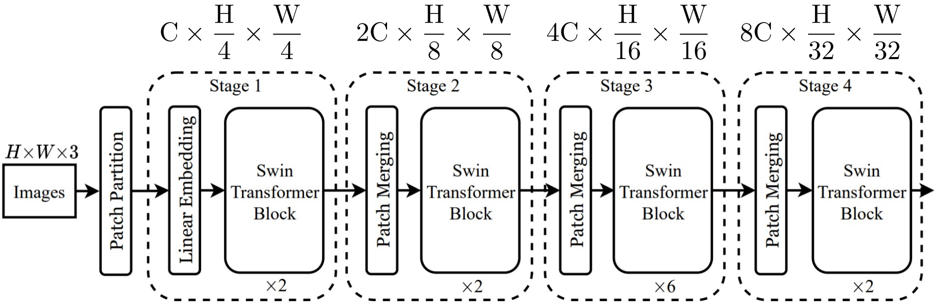
Merge 2x2 neighbourhoods. Now patches are (effectively) 8x8

Merge 2x2 neighbourhoods. Now patches are (effectively) 16x16

Merge 2x2 neighbourhoods. Now patches are (effectively) 32x32

# Hierarchical ViT: Swin Transformer

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




Divide image into 4x4 patches and project to C dimensions

Merge 2x2 neighbourhoods. Now patches are (effectively) 8x8

Merge 2x2 neighbourhoods. Now patches are (effectively) 16x16

Merge 2x2 neighbourhoods. Now patches are (effectively) 32x32

**Solution:** Limit the attention of a patch to its "window".

# Swin Transformer: Window Attention

Instead of allowing each token to attend to all other tokens, divide the image into M x M windows (here M=4).

Compute attention only within each window.

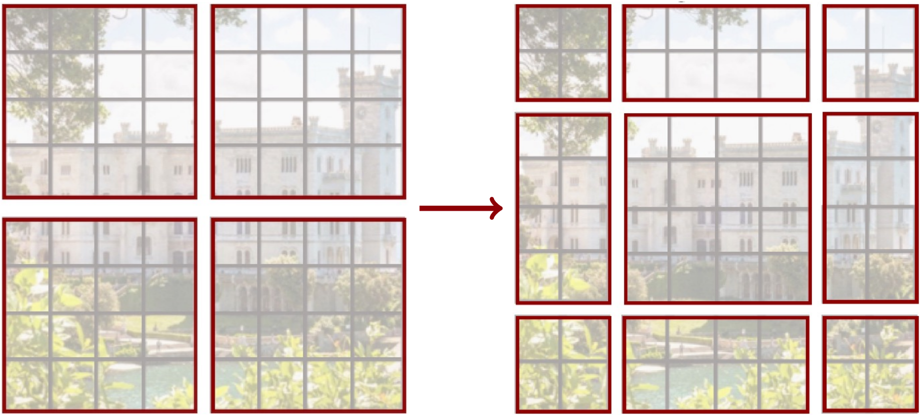
A local window to perform self-attention

A patch

**Problem:** No communication across windows. Global context?

# Swin Transformer: Window Attention

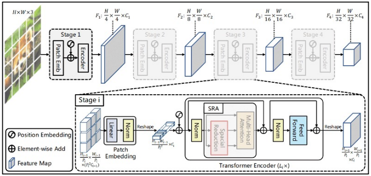
**Solution:** Alternate between normal windows and shifted windows in successive Transformer blocks.



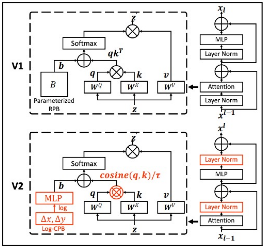
Block L: Normal windows

Block L+1: Shifted windows

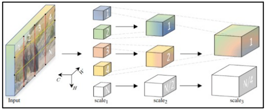
# Other hierarchical vision transformers



Wang, Wenhai, et al. "Pyramid vision transformer: A versatile backbone for dense prediction without convolutions.", ICCV 2021



Liu, Ze, et al. "Swin transformer v2: Scaling up capacity and resolution.", arXiv 2021, CVPR 2022



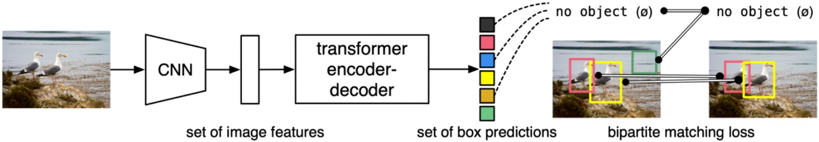
Fan ,Haoqi, et al. "Multiscale vision transformers.", ICCV 2021

# Outline

1. Attention mechanism
2. Self-attention
3. Multi-head self-attention
4. Transformer encoder block
5. Transformer decoder block
6. Vision transformer (ViT)
7. Swin transformer
8. DEtection TRansformer (DETR) for object detection
9. Suggested resources

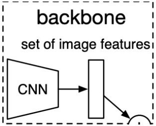
# Object detection with transformers: DETR

- Simple object detection pipeline: directly output a set of boxes from a Transformer.
- Anchor-free approach, i.e., no pre-defined anchors are used.
- No Non-Maximum Suppression (NMS) used. It is learned by the transformer automatically.
- Match predicted boxes to ground truth boxes with bipartite matching.




Carion et al, "End-to-End Object Detection with Transformers", ECCV 2020

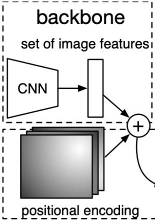
# Object detection with transformers: DETR




A backbone ConvNet (like ResNet50) is used to extract features from the input.

1x1 conv layer is used to reduce the number of channels.

# Object detection with transformers: DETR

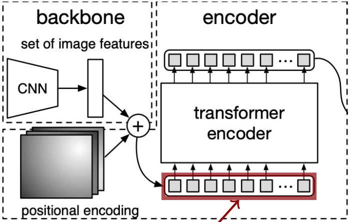



A backbone ConvNet (like ResNet50) is used to extract features from the input.

1x1 conv layer is used to reduce the number of channels.

Fixed positional encodings are added to every encoder block's input

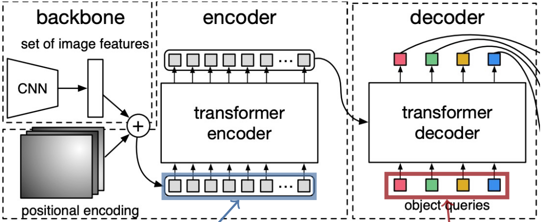
# Object detection with transformers: DETR




Standard transformer encoder is used to process the flattened image features.

HW x C
Flattened image features

# Object detection with transformers: DETR



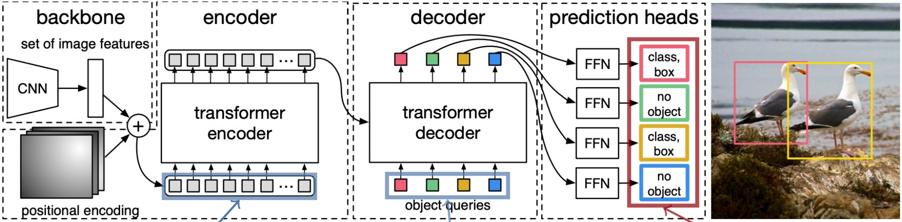

HW x C
Flattened image features

N x C
learnable object queries
(randomly initialised)

A standard transformer decoder is used to generate outputs.

**Input:** N object queries, encoder's output

# Object detection with transformers: DETR




HW x C
Flattened image features

N x C
learnable object queries
(randomly initialised)

Each object query
predicts one
bounding box

# Is self-attention magical and absolutely necessary?

- A lot of works have explored replacing self-attention with X, and some of them achieve competitive results.
  - Wu et al. replace self-attention with convolution.
  - Tay et al. replace alignment scores in self-attention with random projection.
  - Tolstikhin et al. replace self-attention with an MLP-mixer, making the architecture exclusively based on MLPs.
  - Lee-Thorp et al. replace self-attention with Fourier Transforms.
  - Tatsunami and Taki replace self-attention with LSTM.
- But, self-attention continues to be the standard choice.

# Outline

## Suggested resources

- The Illustrated Transformer: A famous blog post covering the basics of Transformers.
- The Annotated Transformer: A blog post that implements Transformer line by line with explanation.