

Lecture 13: Vision Transformers



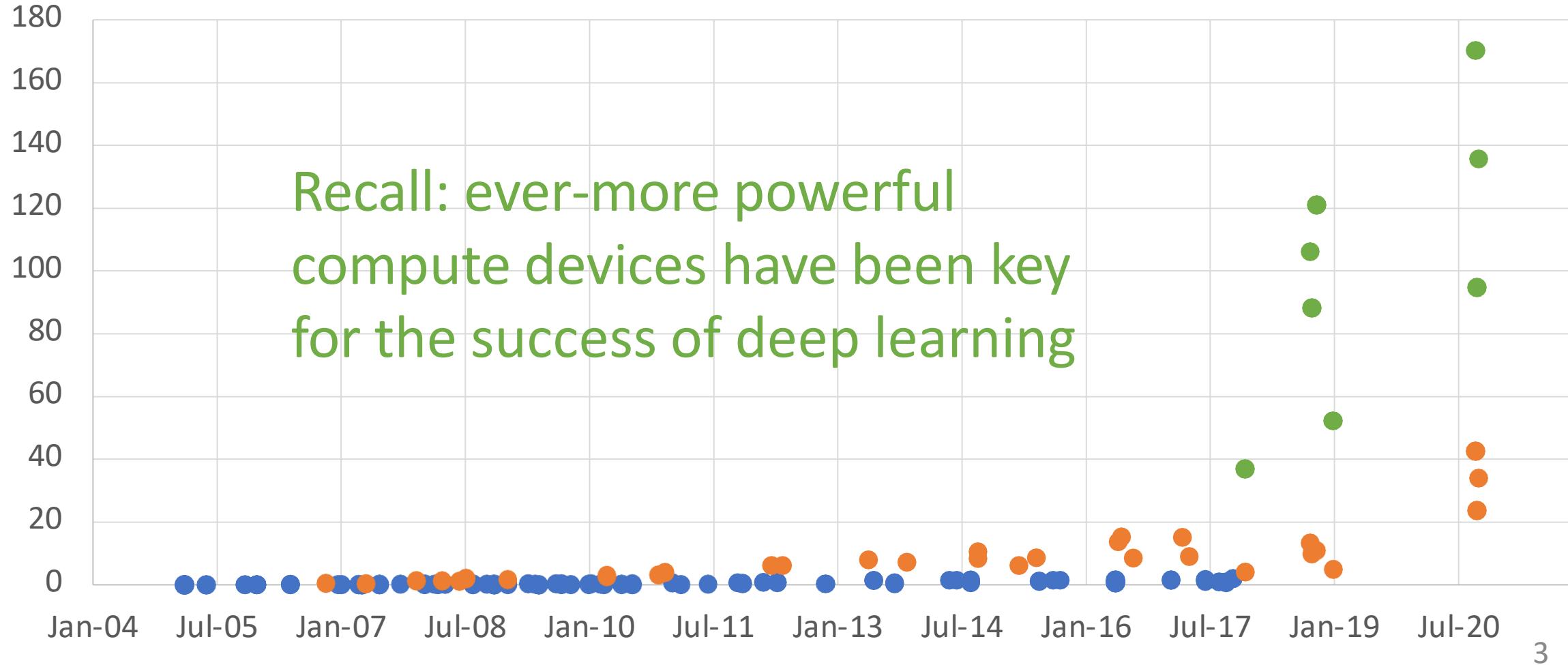
source from University of Michigan EECS498

Today: Vision Transformers

But first...

GFLOP per Dollar

● CPU ● GPU (FP32) ● GPU (Tensor Core)



Recall: ever-more powerful
compute devices have been key
for the success of deep learning

Best GPU money can buy: NVIDIA A100

Memory:

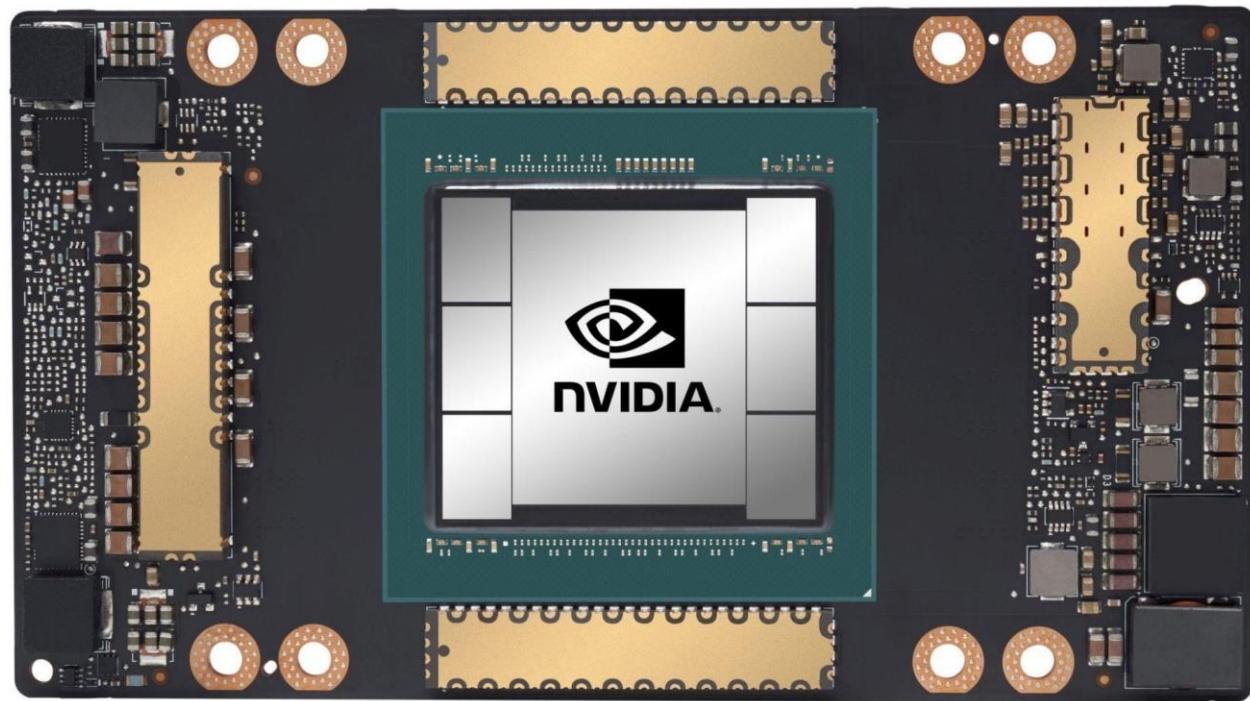
Capacity: 40/80 GB HBM2
Bandwidth: 1.5/2.0 TB/sec

FLOP = “Floating Point Operation”; one addition, multiplication, etc

TFLOP = 1 trillion FLOPs (10^{12})

Compute:

FP64: 9.7 TFLOP/sec
FP32: 19.5 TFLOP/sec
BF16: 39 TFLOP/sec
FP16: 78 TFLOP/sec



Best GPU money can buy: NVIDIA A100

Memory:

Capacity: 40/80 GB HBM2
Bandwidth: 1.5/2.0 TB/sec

FLOP = “Floating Point Operation”; one addition, multiplication, etc

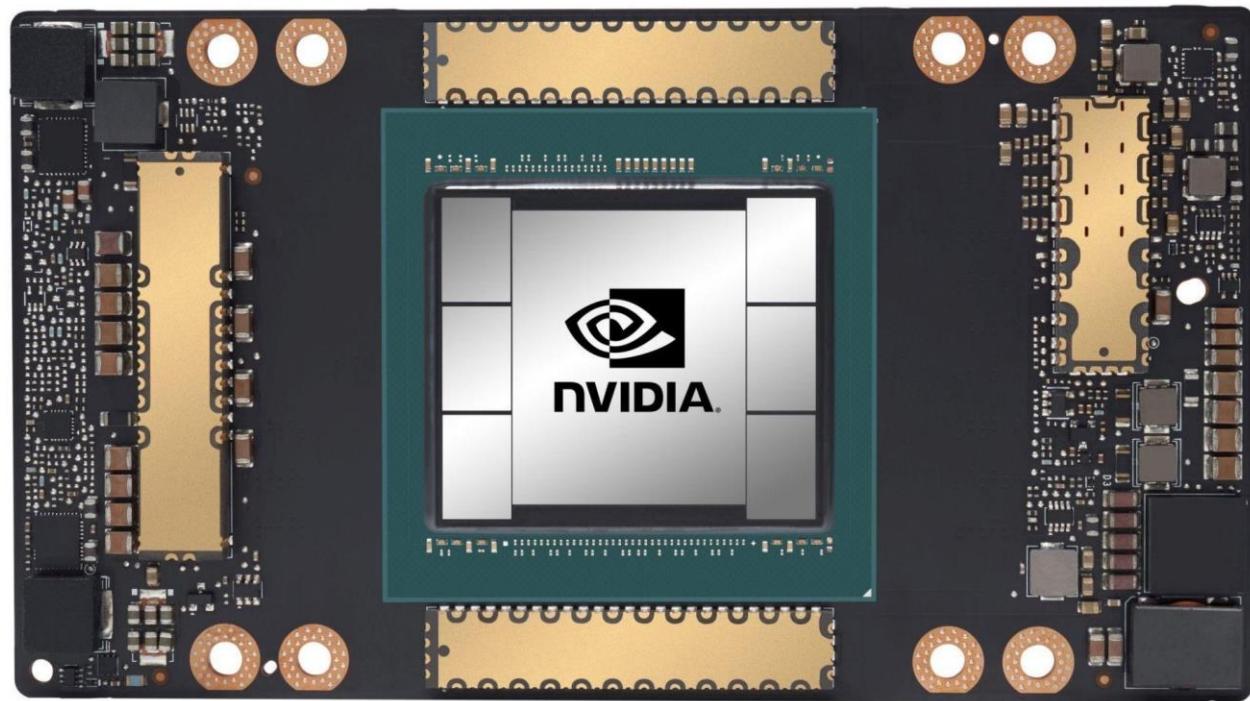
TFLOP = 1 trillion FLOPs (10^{12})

Compute:

FP64: 9.7 TFLOP/sec
FP32: 19.5 TFLOP/sec
BF16: 39 TFLOP/sec
FP16: 78 TFLOP/sec

Tensor Cores:

TF32: 156 TFLOP/sec
FP16/BF16: 312 TFLOP/sec



2022.03.23: New NVIDIA H100 GPU

Memory:

Capacity: 40/80 GB HBM3

Bandwidth: 3.0 TB/sec (**1.5x better**)

FLOP = “Floating Point Operation”; one addition, multiplication, etc

TFLOP = 1 trillion FLOPs (10^{12})

Compute:

FP64: 30 TFLOP/sec (**3x better**)

FP32: 60 TFLOP/sec (**3x better**)

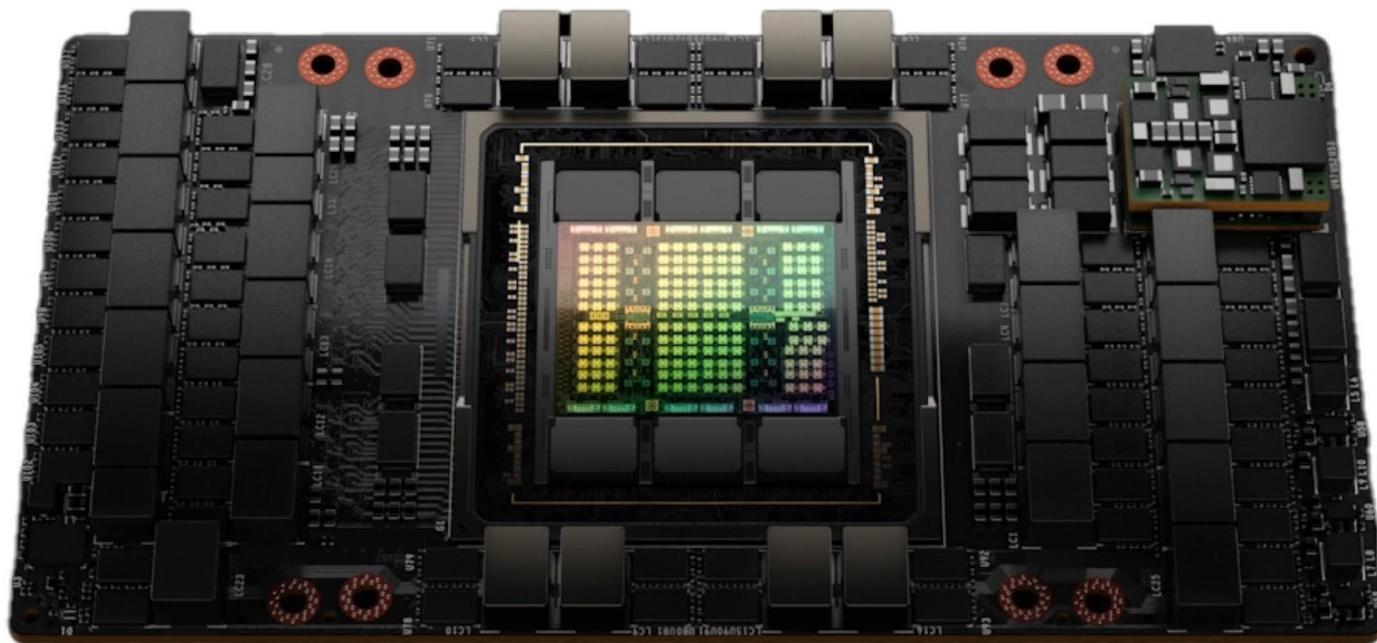
BF16: 120 TFLOP/sec (**3x better**)

FP16: 120 TFLOP/sec (**1.5x better**)

Tensor Cores:

TF32: 500 TFLOP/sec (**3.2x better**)

FP16/BF16: 1000 TFLOP/sec (**3.2x better**)



2022.03.23: New NVIDIA H100 GPU

Memory:

Capacity: 40/80 GB HBM3

Bandwidth: 3.0 TB/sec (**1.5x better**)

FLOP = “Floating Point Operation”; one addition, multiplication, etc
TFLOP = 1 trillion FLOPs (10^{12})

Compute:

What are these?

FP64: 30 TFLOP/sec (**3x better**)

FP32: 60 TFLOP/sec (**3x better**)

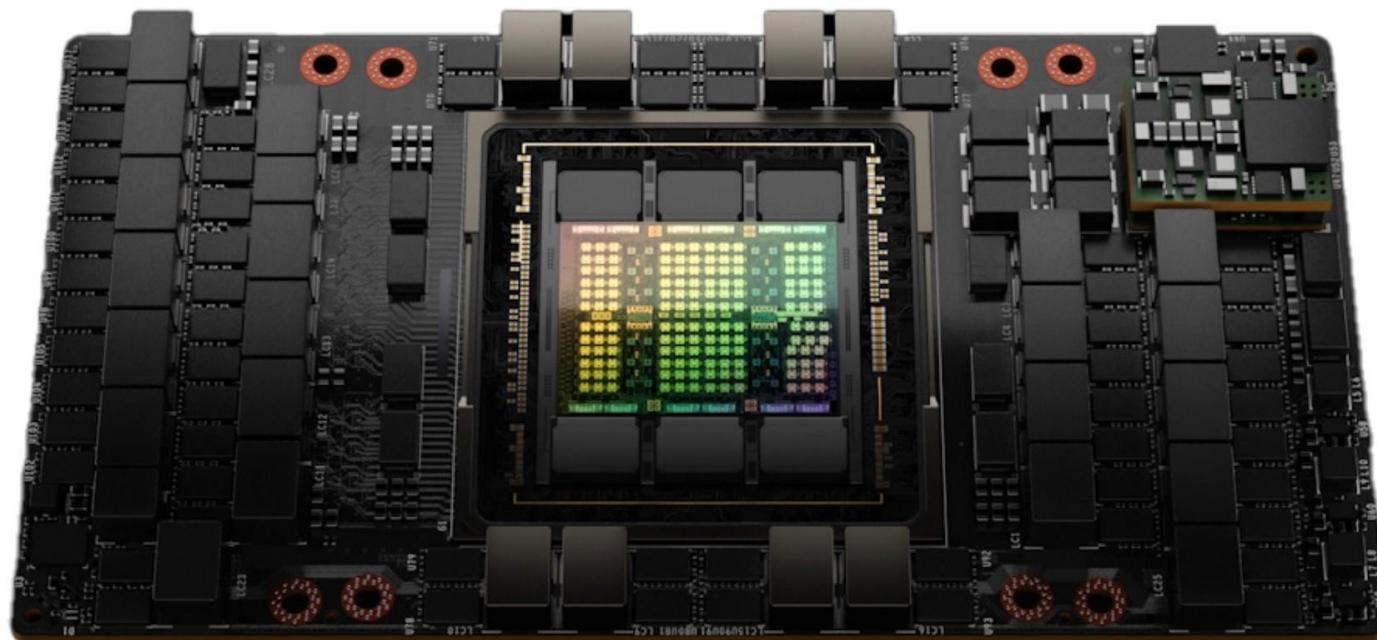
BF16: 120 TFLOP/sec (**3x better**)

FP16: 120 TFLOP/sec (**1.5x better**)

Tensor Cores:

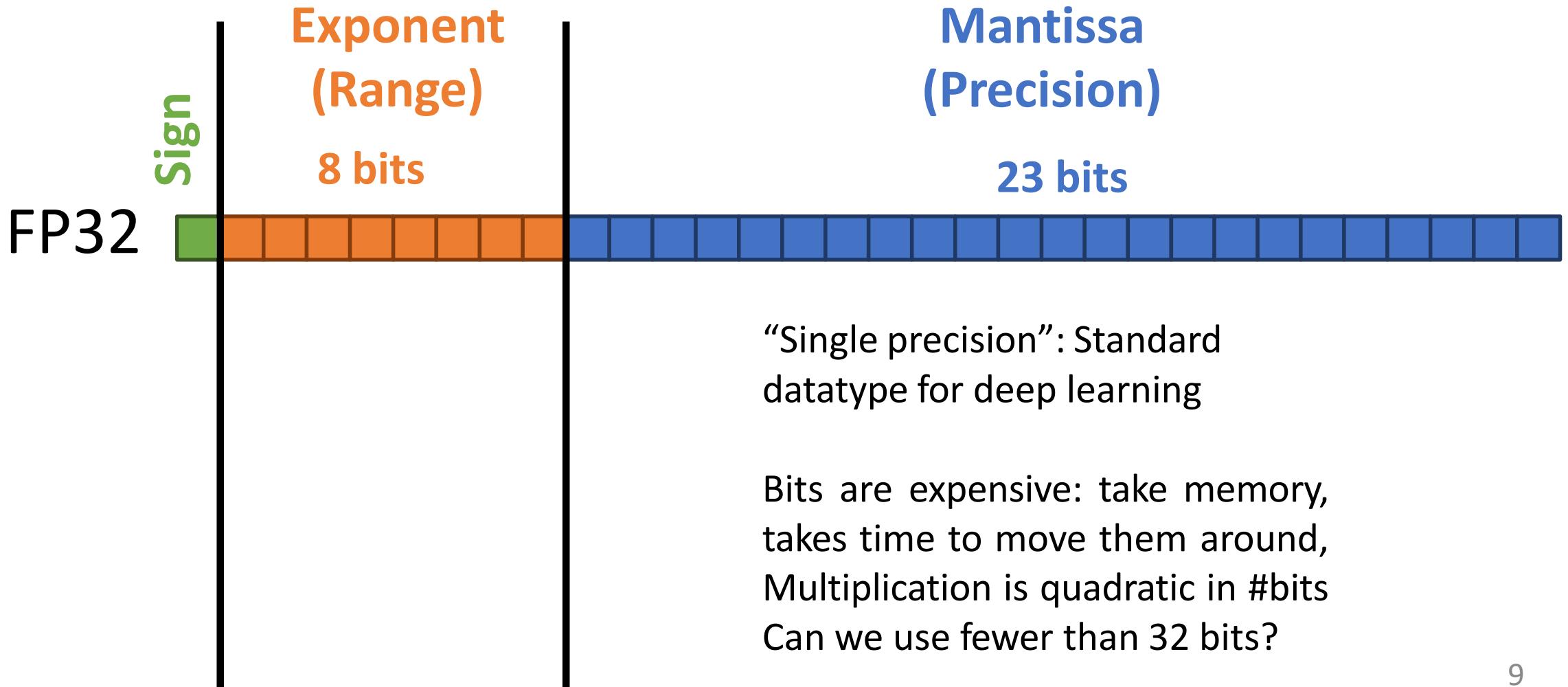
TF32: 500 TFLOP/sec (**3.2x better**)

FP16/BF16: 1000 TFLOP/sec (**3.2x better**)



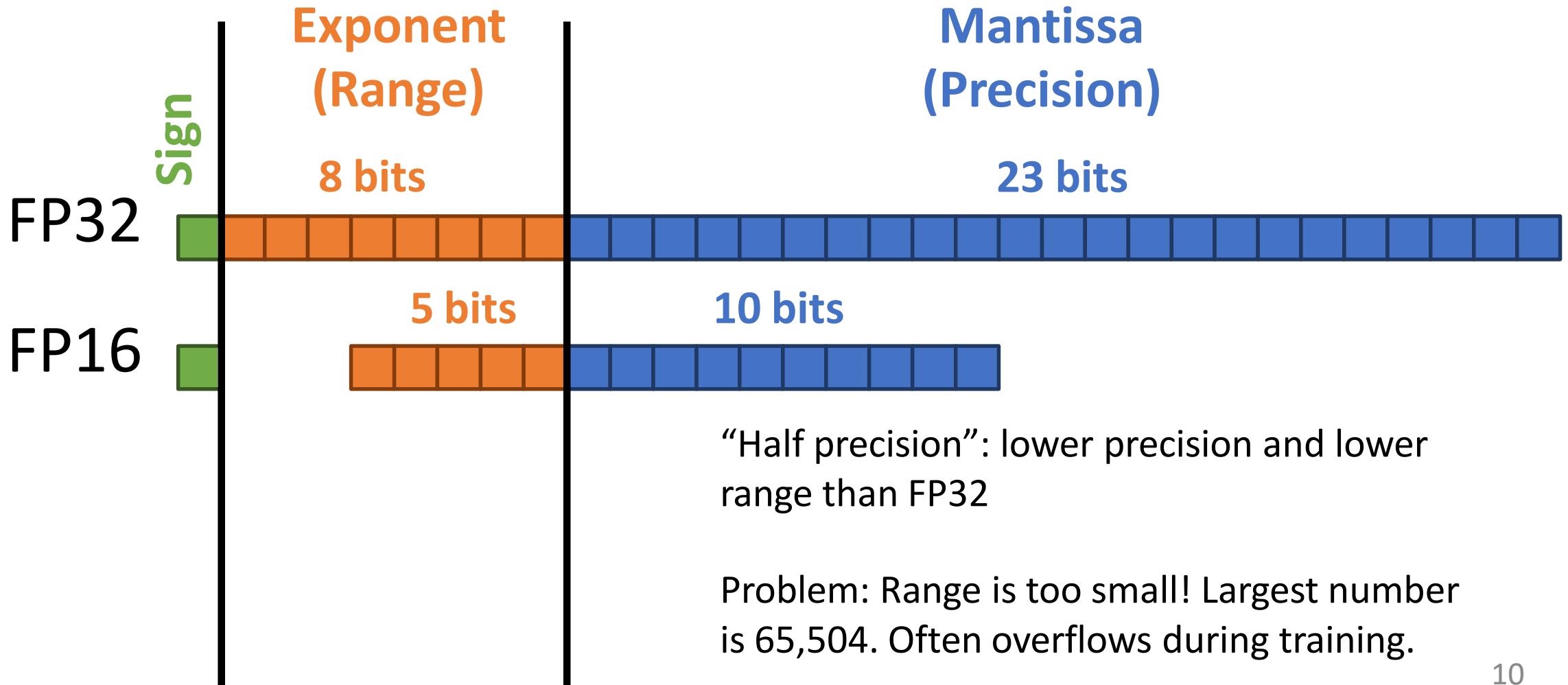
Floating Point Formats $(-1)^s (2^{E+bias}) \left(1 + \frac{M}{2^{|M|}}\right)$

Floating Point Formats $(-1)^S (2^{E+bias}) \left(1 + \frac{M}{2^{|M|}}\right)$

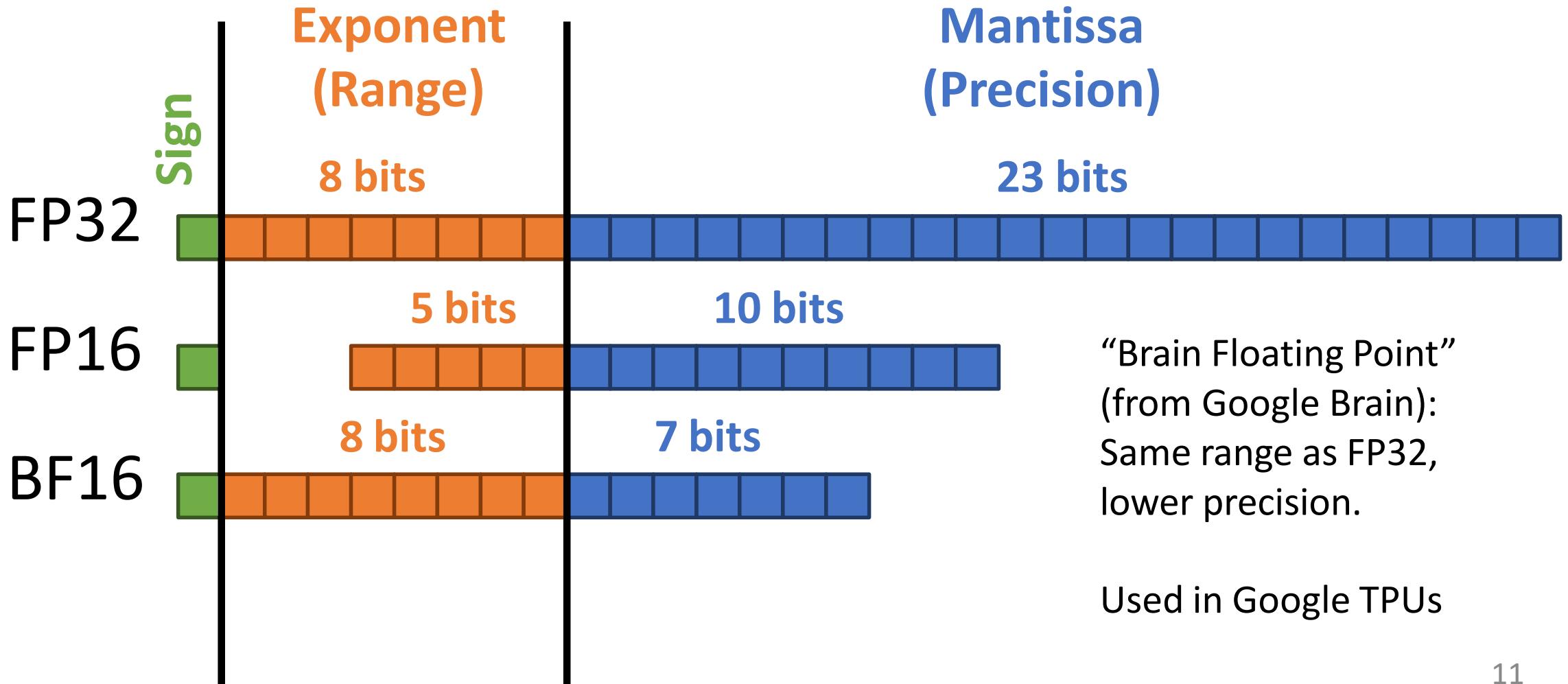


Floating Point Formats

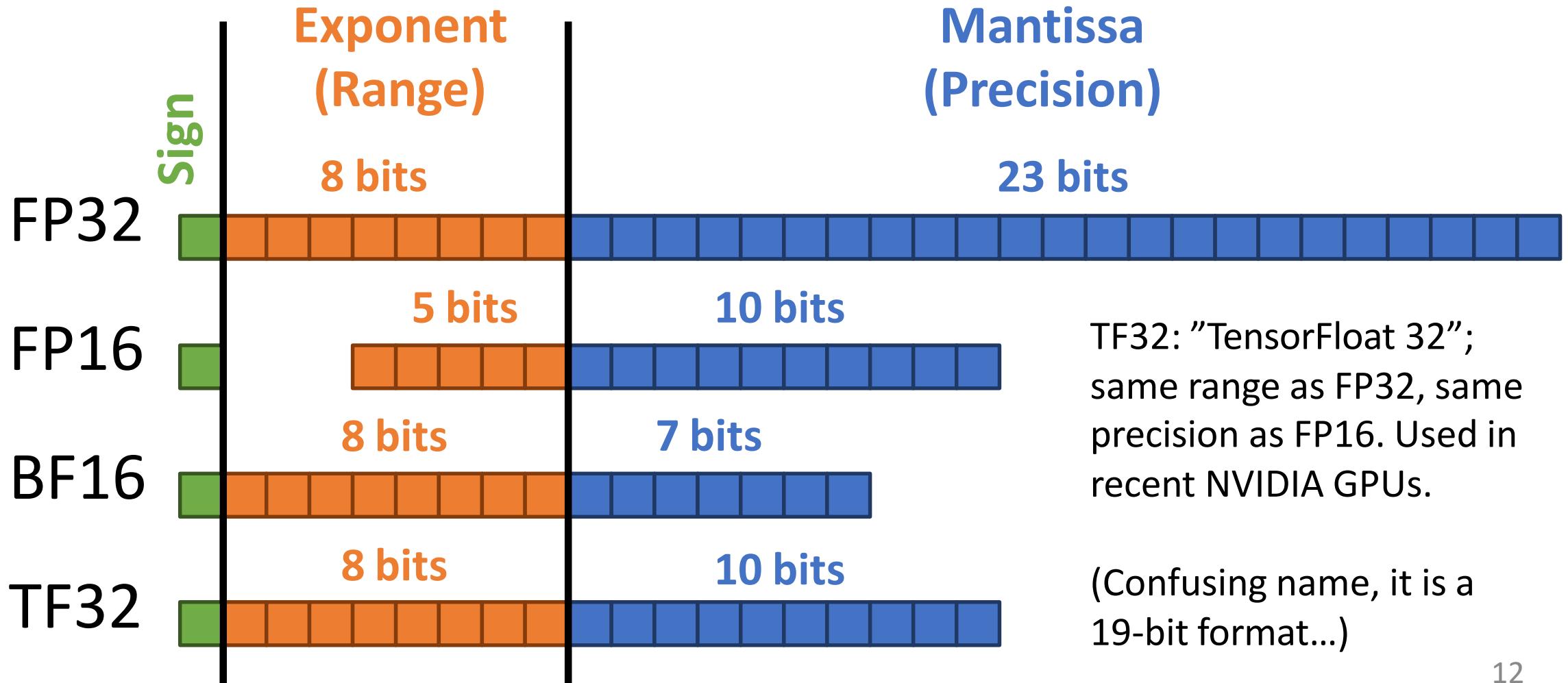
$$(-1)^S (2^{E+bias}) \left(1 + \frac{M}{2^{|M|}}\right)$$



Floating Point Formats $(-1)^S (2^{E+bias}) \left(1 + \frac{M}{2^{|M|}}\right)$



Floating Point Formats $(-1)^S (2^{E+bias}) \left(1 + \frac{M}{2^{|M|}}\right)$



Mixed Precision

We often need to compute dot products (for matrix multiply, convolution, etc):

$$y = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

Mixed Precision

We often need to compute dot products (for matrix multiply, convolution, etc):

$$y = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

Multiplication is more expensive than addition

Idea: Multiply in low precision, add in high precision

Mixed Precision

We often need to compute dot products (for matrix multiply, convolution, etc):

$$y = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

Multiplication is more expensive than addition

Idea: Multiply in low precision, add in high precision

Inputs: x_i, w_i in low precision (FP16, BF16, TF32)

Output: y in high precision (FP32)

$$y = FP32(x_1 w_1) + FP32(x_2 w_2) + \cdots + FP32(x_n w_n)$$

Mixed Precision

We often need to compute dot products (for matrix multiply, convolution, etc):

$$y = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

Multiplication is more expensive than addition

Idea: Multiply in low precision, add in high precision

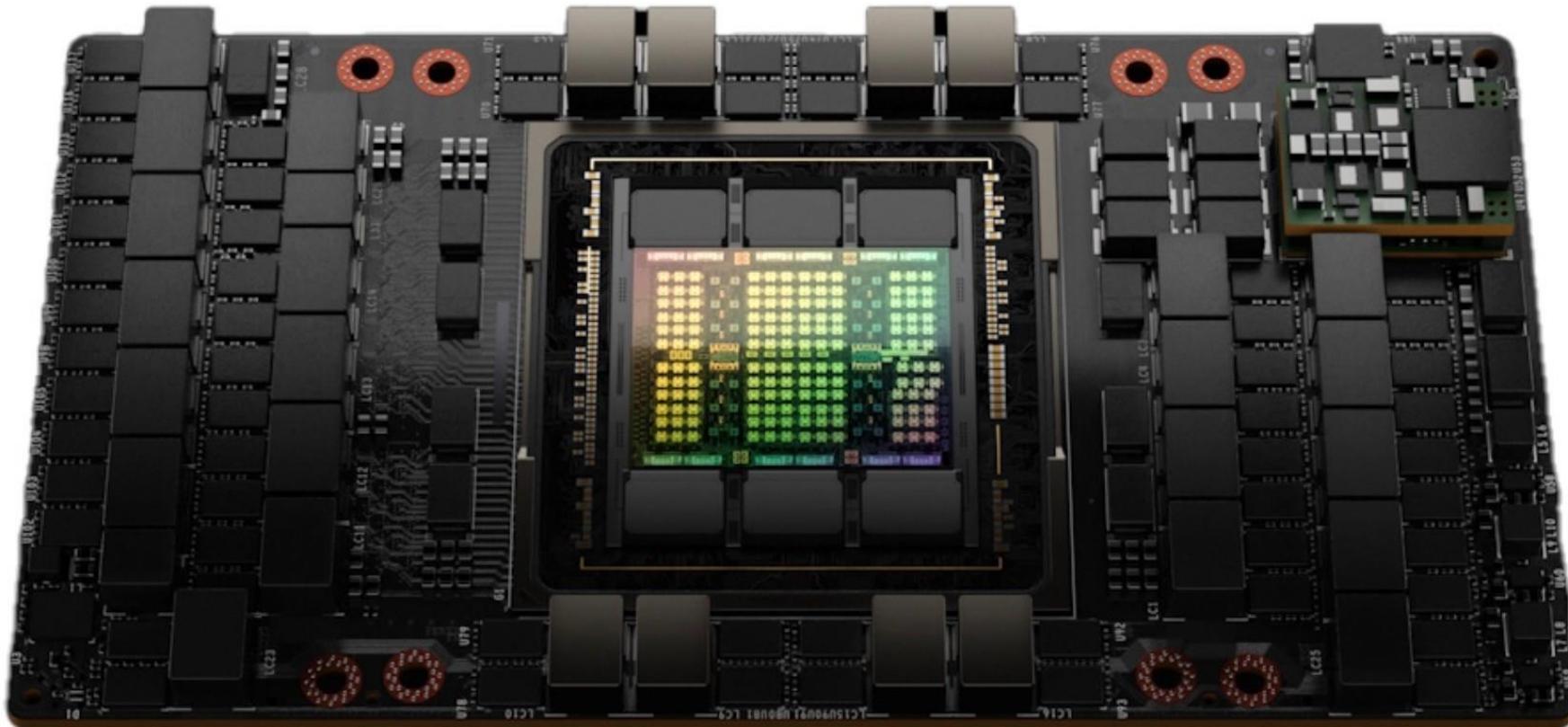
Inputs: x_i, w_i in low precision (FP16, BF16, TF32)

Output: y in high precision (FP32)

$$y = FP32(x_1 w_1) + FP32(x_2 w_2) + \cdots + FP32(x_n w_n)$$

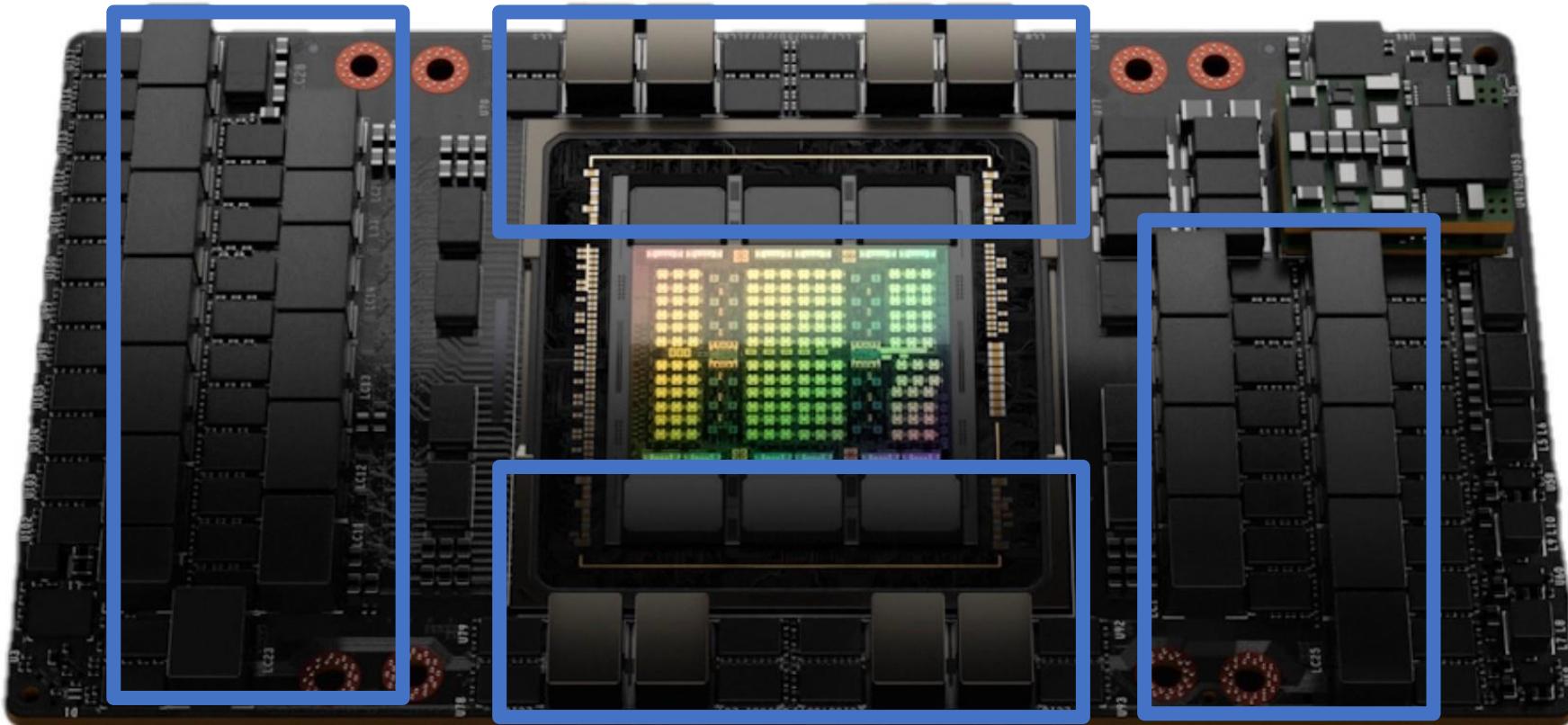
Tensor Cores in NVIDIA GPUs are special hardware for mixed-precision matrix multiplication with different low-precision formats (TF32, BF16 best for neural nets)

2022.03.23 : New NVIDIA H100 GPU



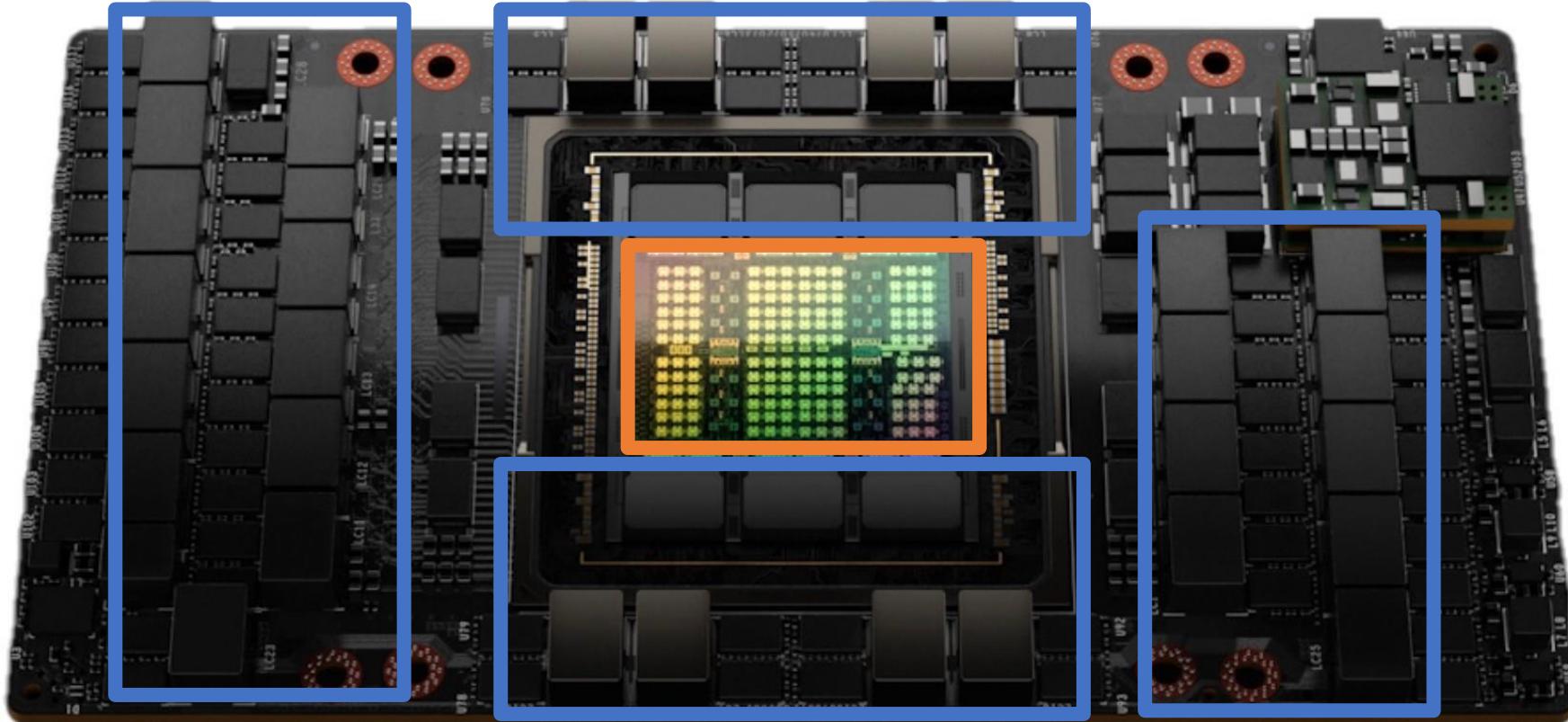
2022.03.23: New NVIDIA H100 GPU

80 GB of HBM3 memory



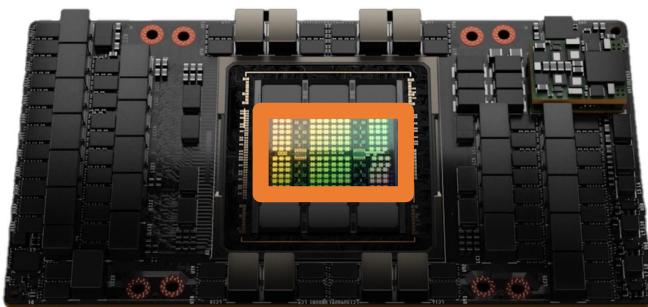
2022.03.23: New NVIDIA H100 GPU

80 GB of HBM3 memory



Processing cores

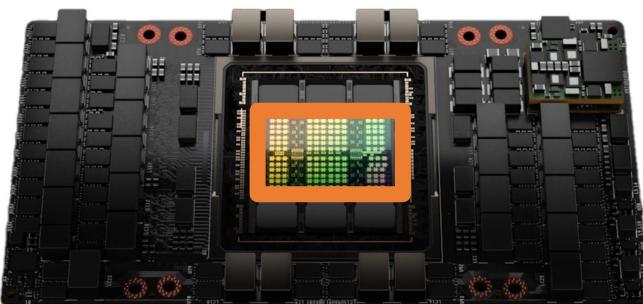
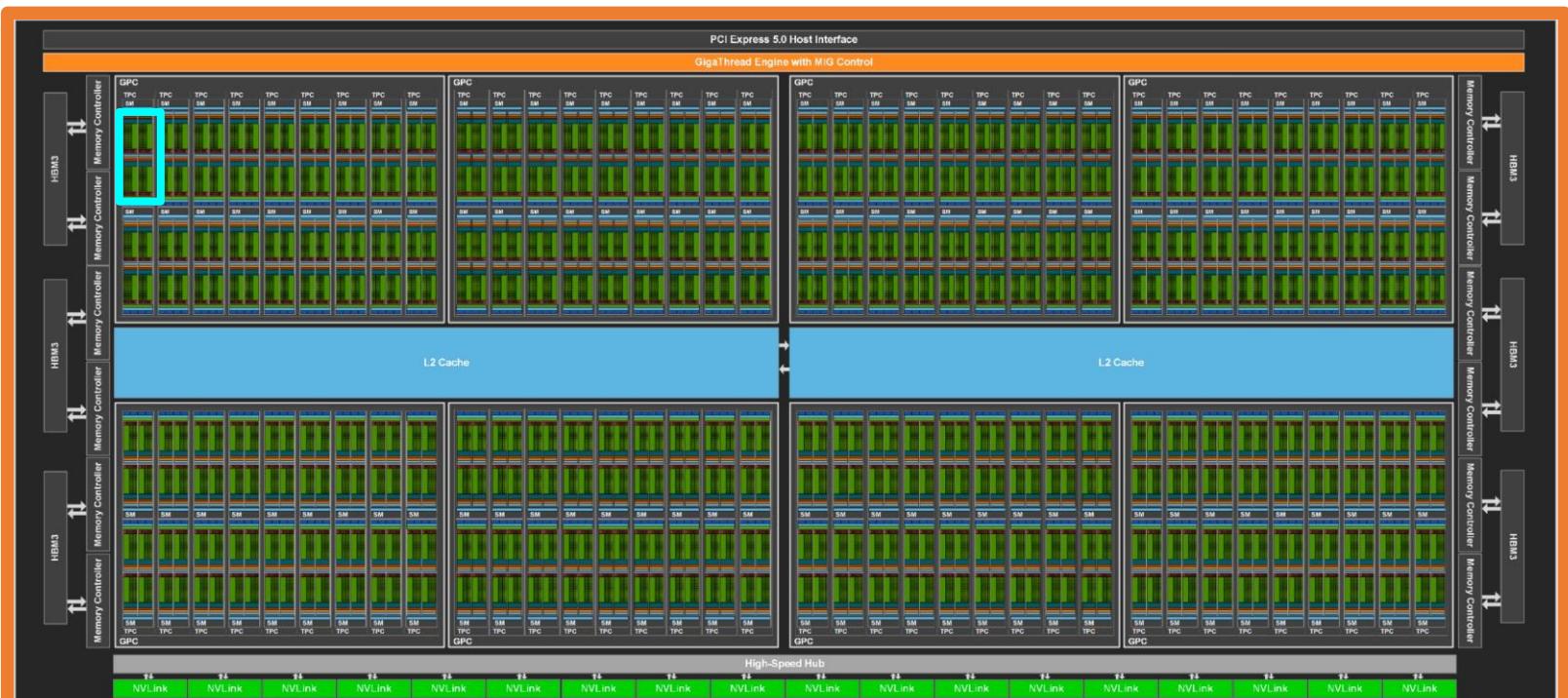
2022.03.23: New NVIDIA H100 GPU



2022.03.23: New NVIDIA H100 GPU

144 “Streaming Multiprocessors”:
Independent multicore
processors

(only 132/144 are enabled due to
issues with yield)



H100 SM

Each SM has 4 subunits that can each simultaneously execute 32 threads (1 warp)

32 **FP32 cores** per subunit; each can compute $y = ax + b$ per clock cycle (1 **multiply-add** = 2 FLOPs)



H100 SM

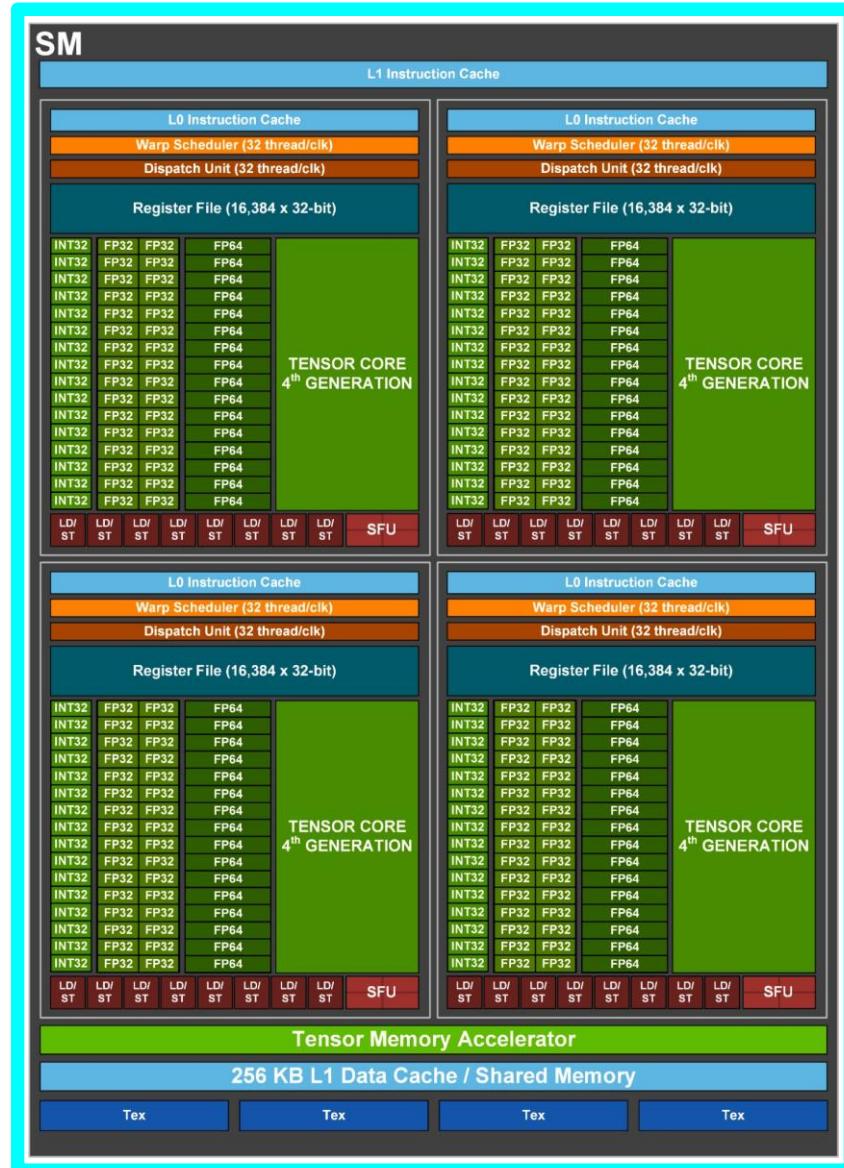
Each SM has 4 subunits that can each simultaneously execute 32 threads (1 warp)

32 **FP32 cores** per subunit; each can compute $y = ax + b$ per clock cycle (1 **multiply-add** = 2 FLOPs)

(132 SMs/GPU) * (128 cores/SM)

* (2 FLOPs/core/cycle) * (1.775 * 10^9 cycles/sec)

= $60 * 10^9$ FLOPs/GPU/sec



H100 SM

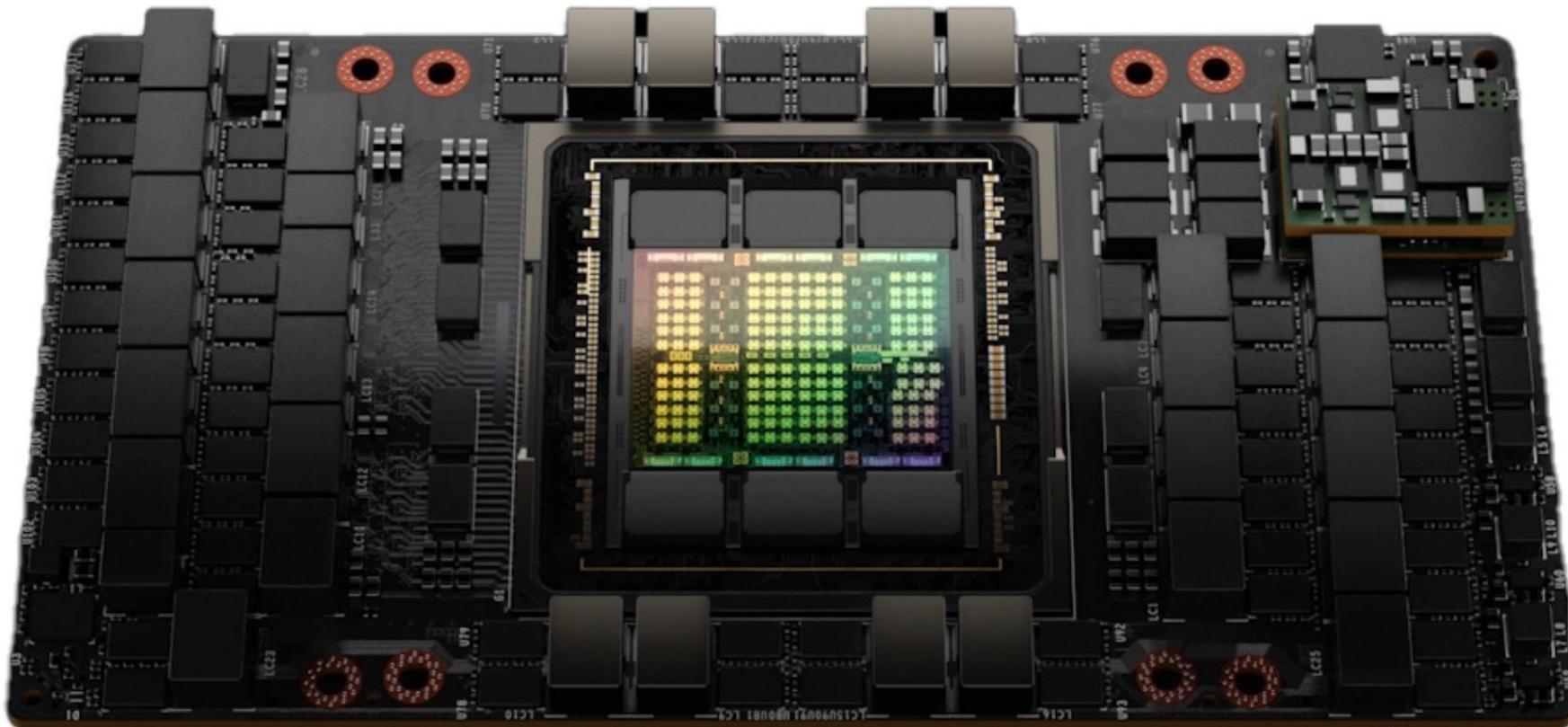
Each SM has 4 subunits that can each simultaneously execute 32 threads (1 warp)

32 **FP32 cores** per subunit; each can compute $y = ax + b$ per clock cycle (1 **multiply-add** = 2 FLOPs)

4 **Tensor cores** per subunit; each can do one tiny matrix multiply per clock: $[4 \times 16] * [16 \times 8] = [4 \times 8]$ (FP16/FP32, $4*8*16*2$ FLOPs = 1024 FLOPs)

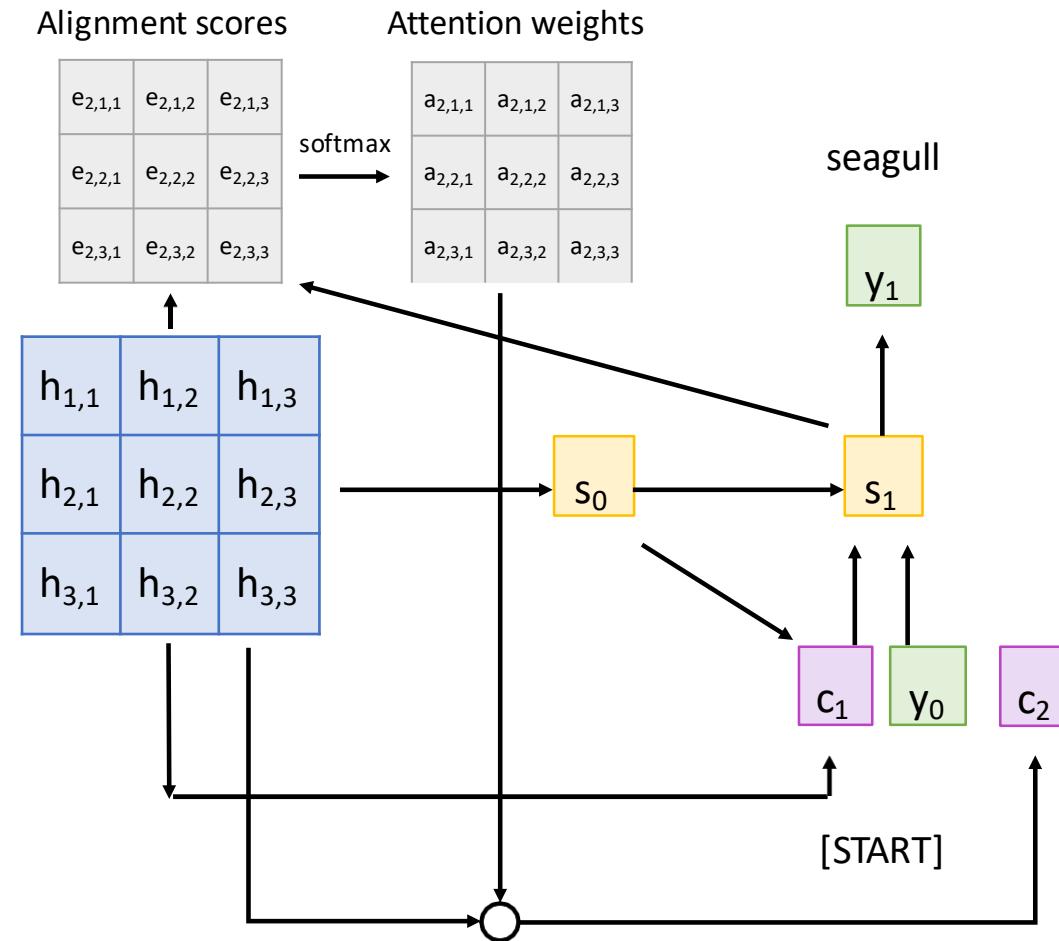
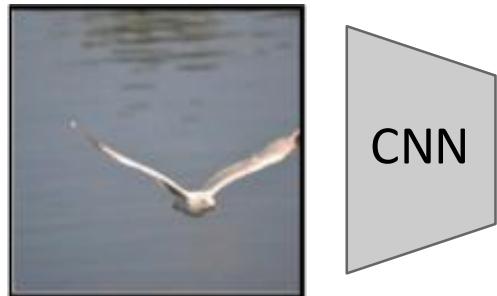


H100 GPU: Expect Bigger Models!



Last Time: Attention

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



Last Time: Self-Attention Layer

Inputs:

Input vectors: \mathbf{X} (Shape: $N_x \times D_x$)

Key matrix: \mathbf{W}_K (Shape: $D_x \times D_Q$)

Value matrix: \mathbf{W}_V (Shape: $D_x \times D_V$)

Query matrix: \mathbf{W}_Q (Shape: $D_x \times D_Q$)

Computation:

Query vectors: $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$

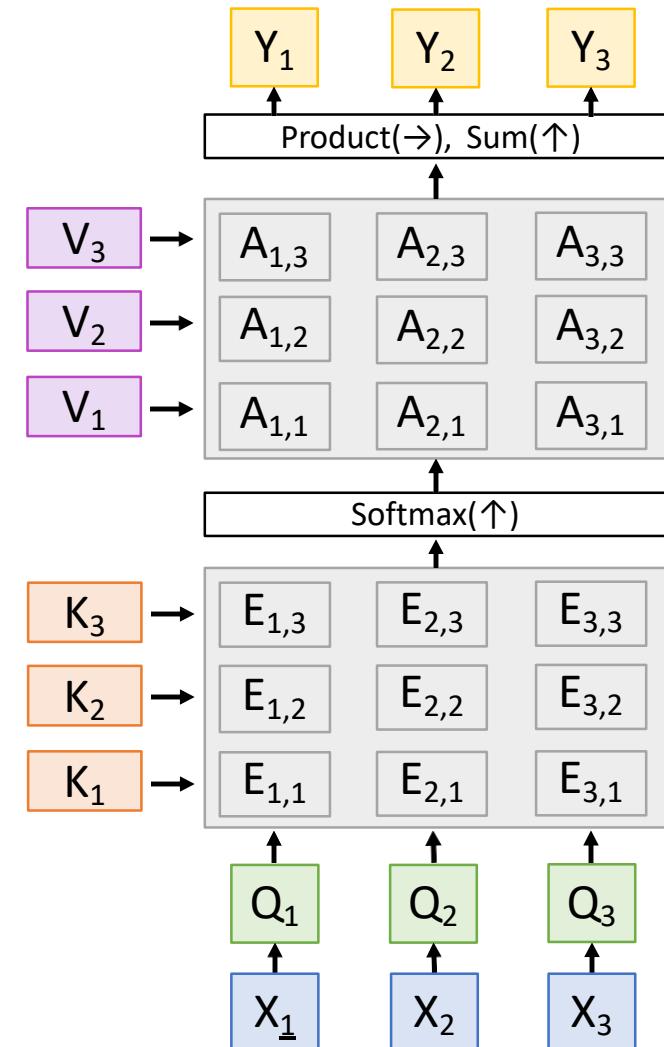
Key vectors: $\mathbf{K} = \mathbf{X}\mathbf{W}_K$ (Shape: $N_x \times D_Q$)

Value Vectors: $\mathbf{V} = \mathbf{X}\mathbf{W}_V$ (Shape: $N_x \times D_V$)

Similarities: $E = \mathbf{Q}\mathbf{K}^T / \sqrt{D_Q}$ (Shape: $N_x \times N_x$) $E_{i,j} = (\mathbf{Q}_i \cdot \mathbf{K}_j) / \sqrt{D_Q}$

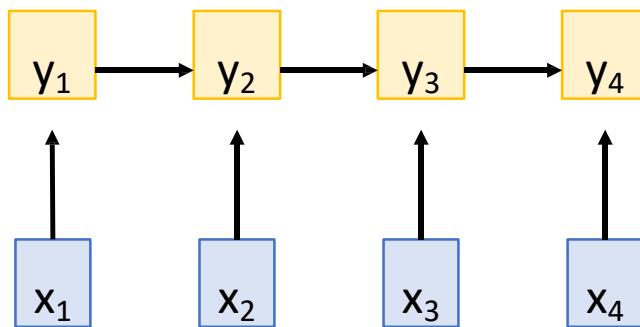
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_x \times N_x$)

Output vectors: $\mathbf{Y} = A\mathbf{V}$ (Shape: $N_x \times D_V$) $Y_i = \sum_j A_{i,j} \mathbf{V}_j$

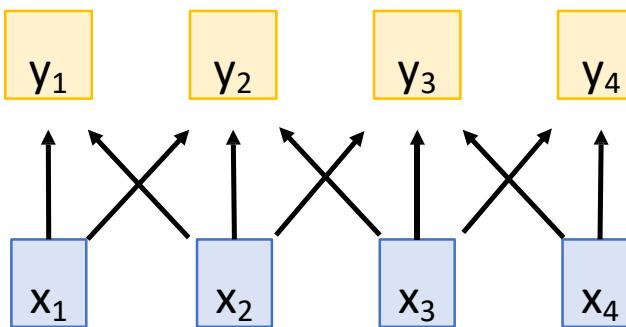


Last Time: Three Ways of Processing Sequences

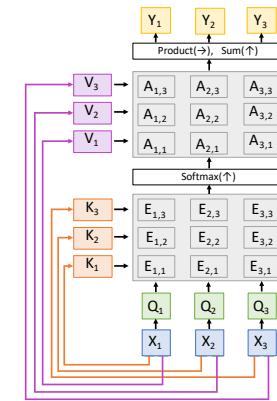
Recurrent Neural Network



1D Convolution



Self-Attention



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!

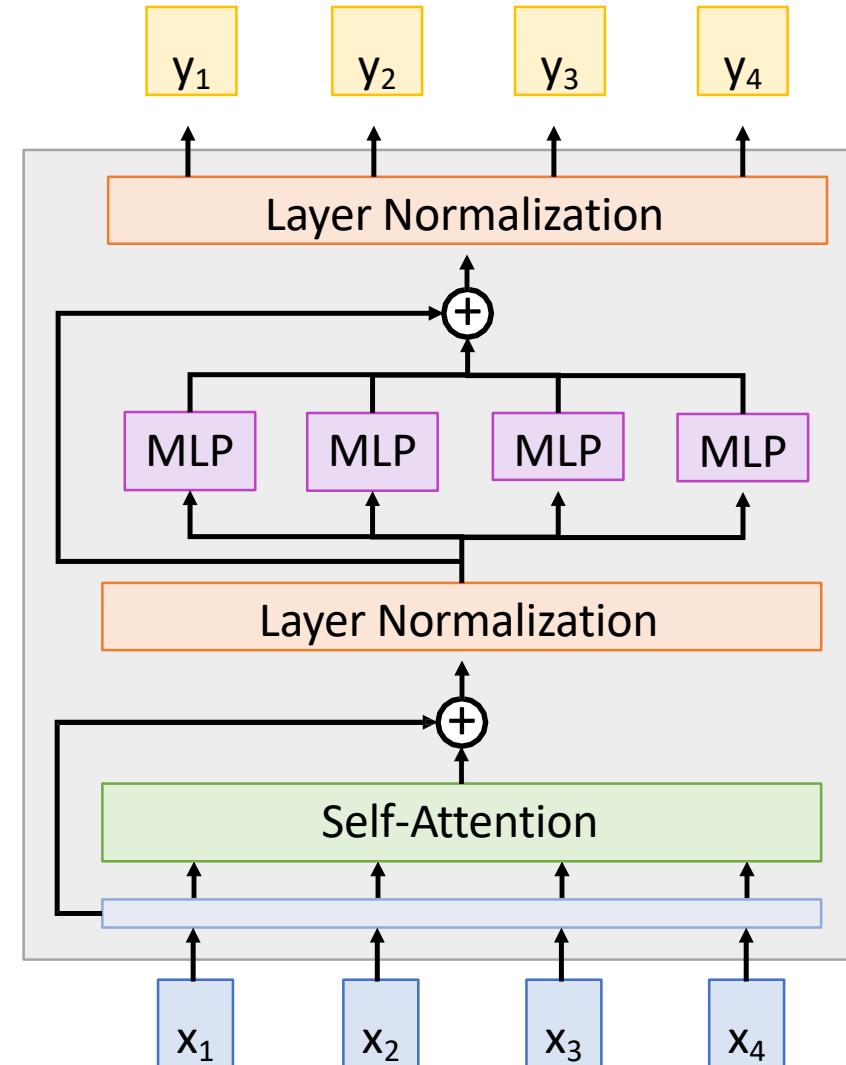
(+) Highly parallel: Each output can be computed in parallel

(-) Very memory intensive

Last Time: Transformer

Transformer block inputs a set of vectors, outputs a set of vectors.

Vectors only communicate via (multiheaded) self-attention



Last Time: Transformer

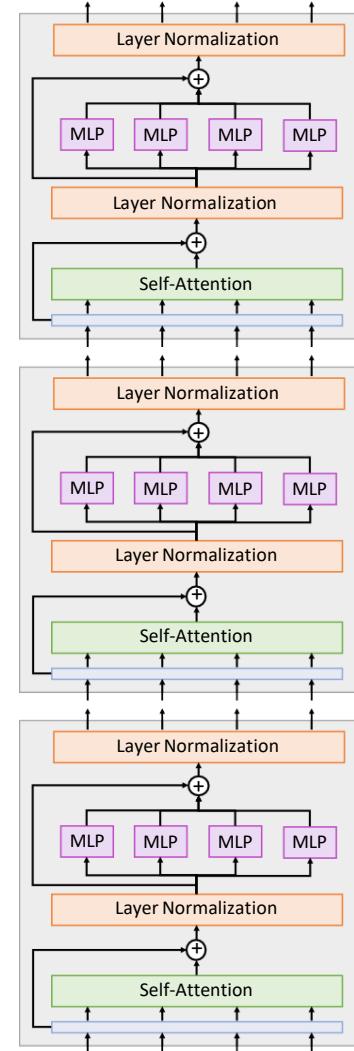
Transformer Block:

Input: Set of vectors x

Output: Set of vectors y

Hyperparameters:

- Number of blocks
- Number of heads per block
- Width (channels per head, FFN width)



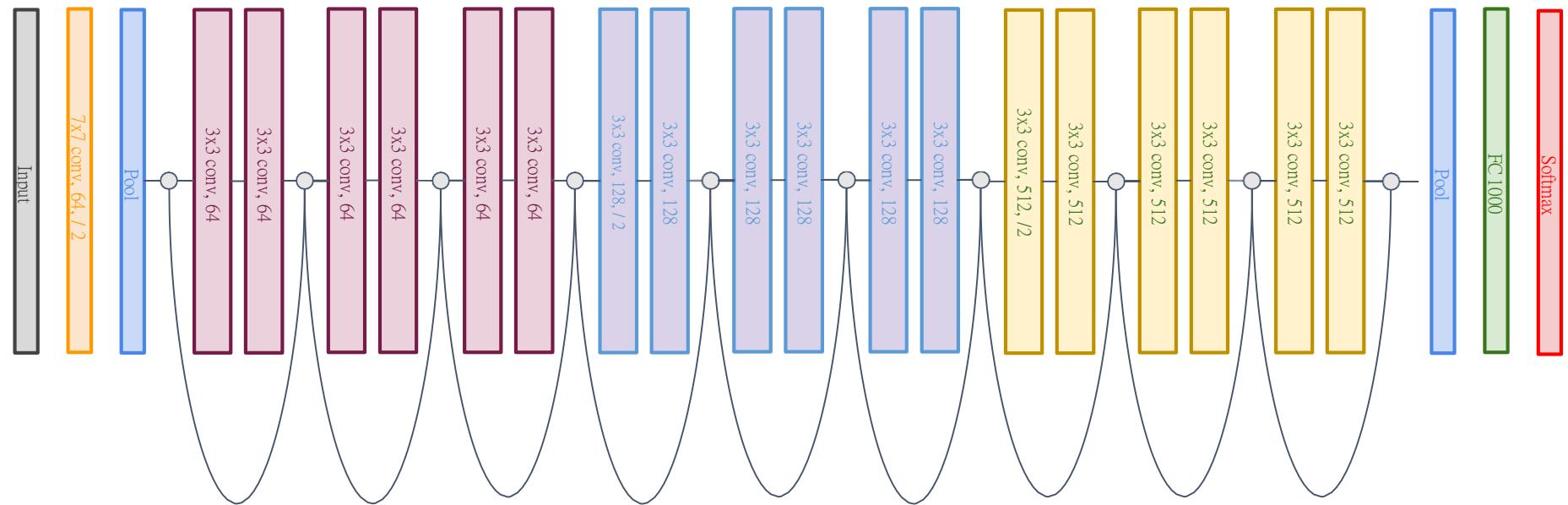
Last Time: Transformers in NLP

Model	Layers	Width	Heads	Params	Data	Training
Transformer-Base	12	512	8	65M		8x P100 (12 hours)
Transformer-Large	12	1024	16	213M		8x P100 (3.5 days)
BERT-Base	12	768	12	110M	13 GB	
BERT-Large	24	1024	16	340M	13 GB	
XLNet-Large	24	1024	16	~340M	126 GB	512x TPU-v3 (2.5 days)
RoBERTa	24	1024	16	355M	160 GB	1024x V100 GPU (1 day)
GPT-2	48	1600	?	1.5B	40 GB	
Megatron-LM	72	3072	32	8.3B	174 GB	512x V100 GPU (9 days)
Turing-NLG	78	4256	28	17B	?	256x V100 GPU
GPT-3	96	12,288	96	175B	694GB	?
Gopher	80	16,384	128	280B	10.55 TB	4096x TPUv3 (38 days)

Today: How to use Attention / Transformers for Vision?

Idea #1: Add attention to existing CNNs

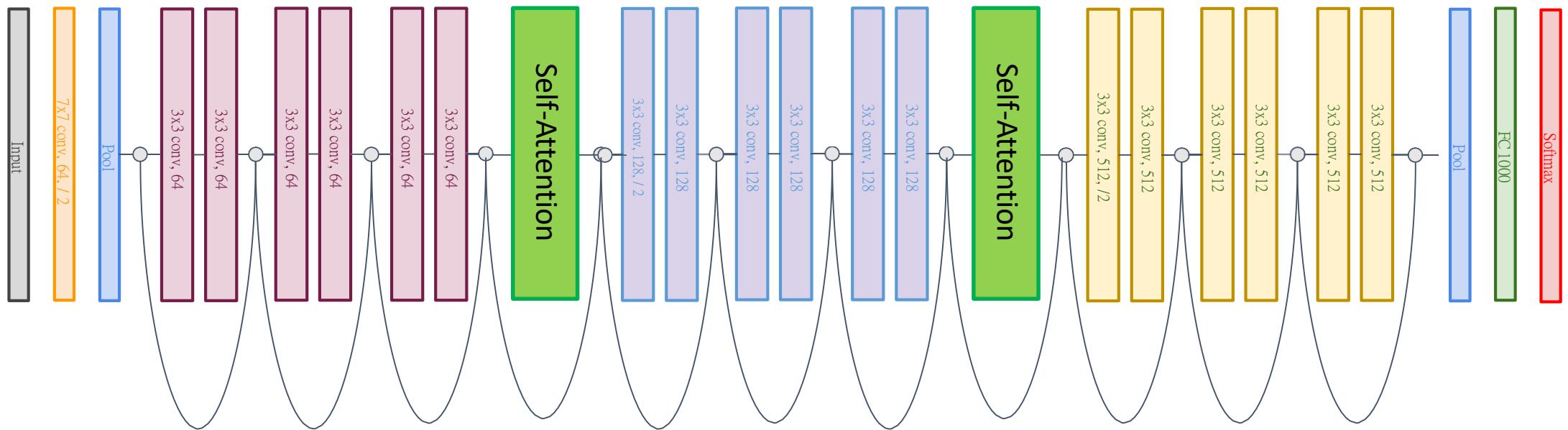
Start from standard CNN architecture (e.g. ResNet)



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

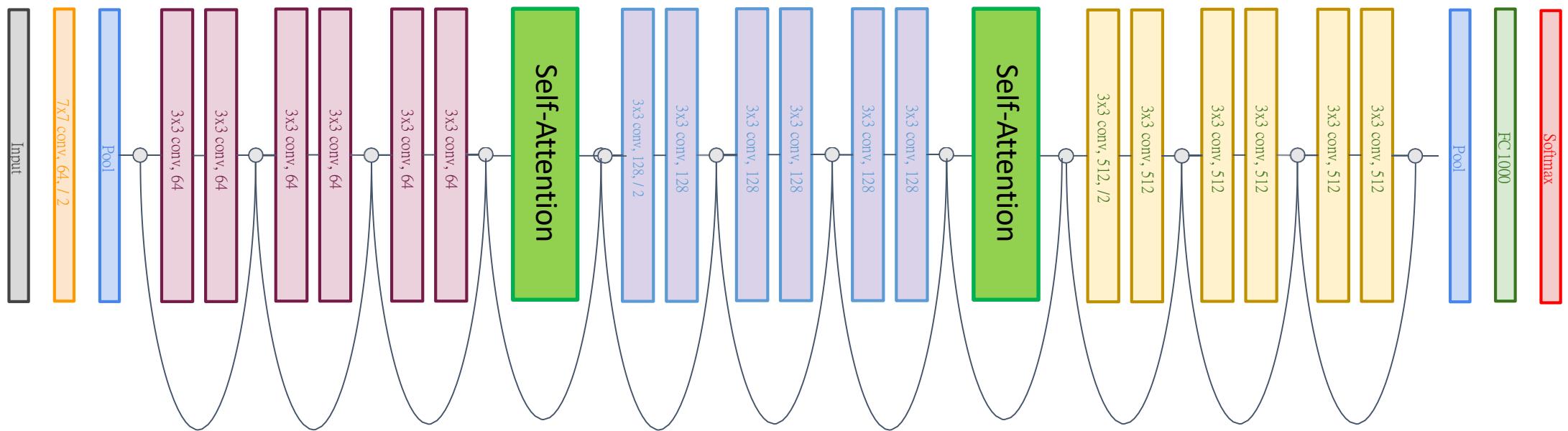
Wang et al, "Non-local Neural Networks", CVPR 2018

Idea #1: Add attention to existing CNNs

Model is still a CNN! Start from standard CNN architecture (e.g. ResNet)

Can we replace

convolution entirely? Add Self-Attention blocks between existing ResNet blocks

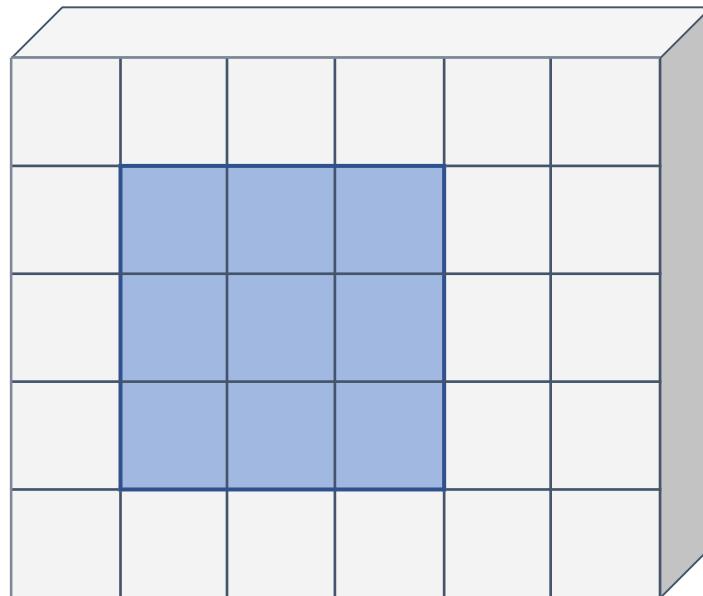


Zhang et al, "Self-Attention Generative Adversarial Networks", ICML 2018

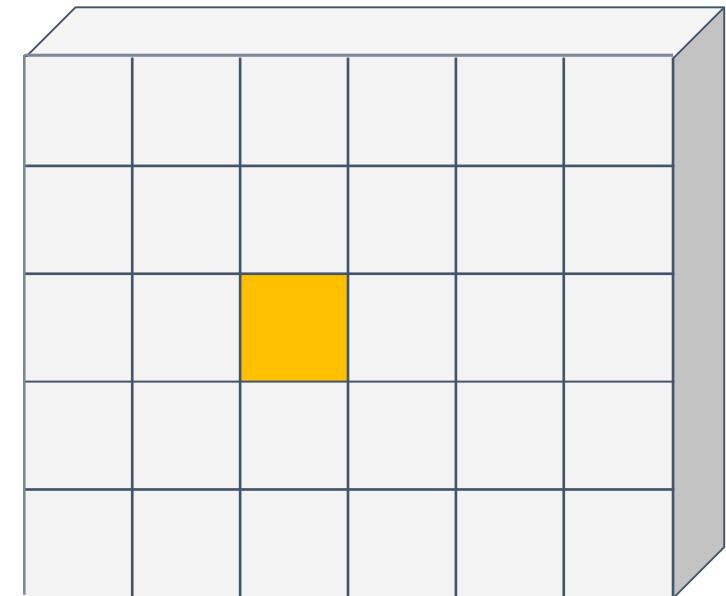
Wang et al, "Non-local Neural Networks", CVPR 2018

Idea #2: Replace Convolution with “Local Attention”

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



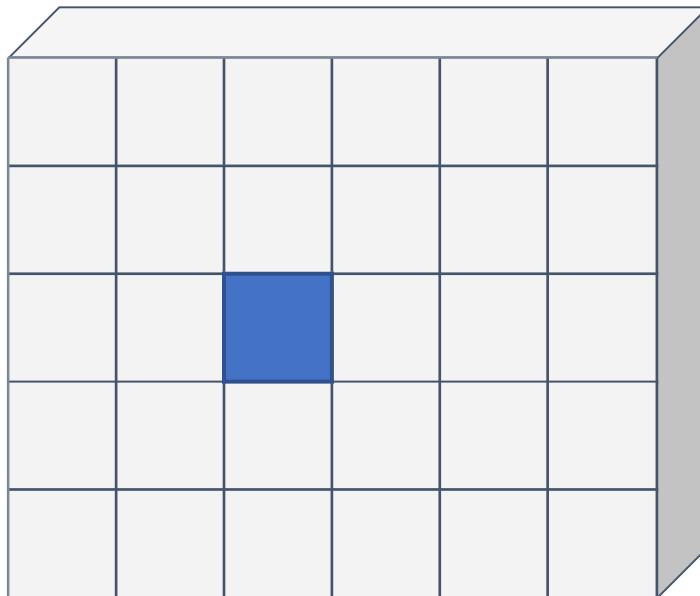
Input: $C \times H \times W$



Output: $C' \times H \times W$

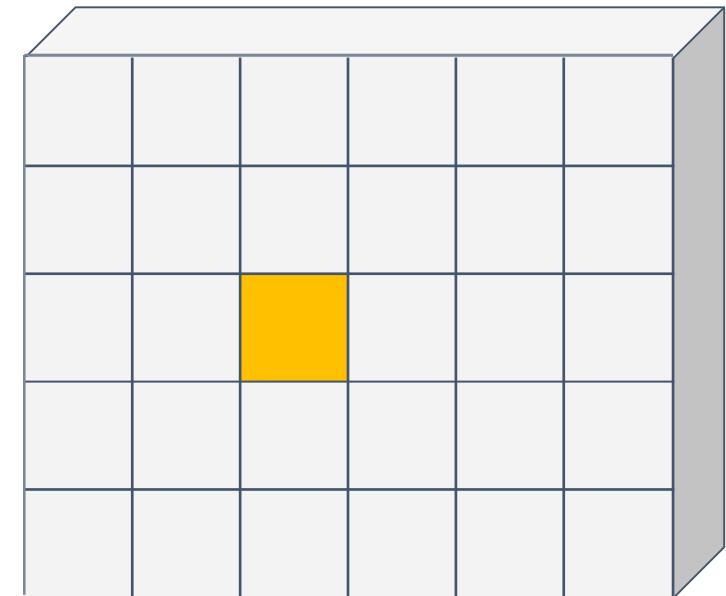
Idea #2: Replace Convolution with “Local Attention”

Map center of receptive field to **query**



Input: $C \times H \times W$

Query: D_Q

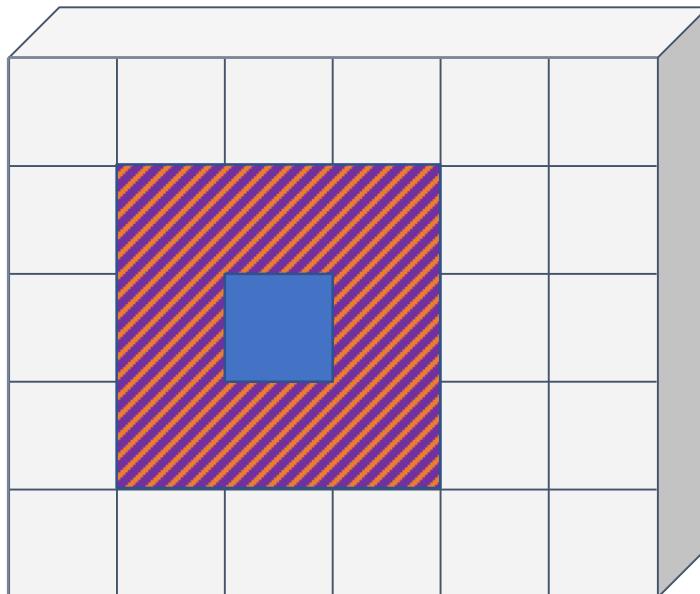


Output: $C' \times H \times W$

Idea #2: Replace Convolution with “Local Attention”

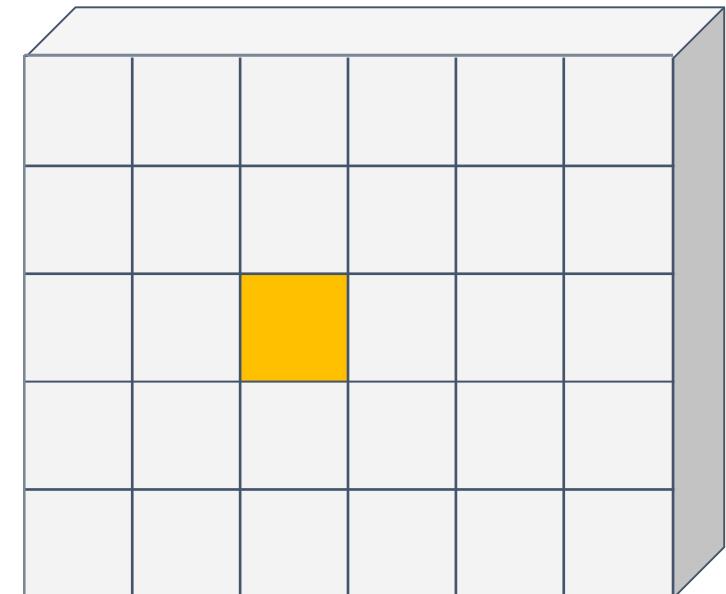
Map center of receptive field to **query**

Map each element in receptive field to **key** and **value**



Input: $C \times H \times W$

Query: D_Q
Keys: $R \times R \times D_Q$
Values: $R \times R \times C'$



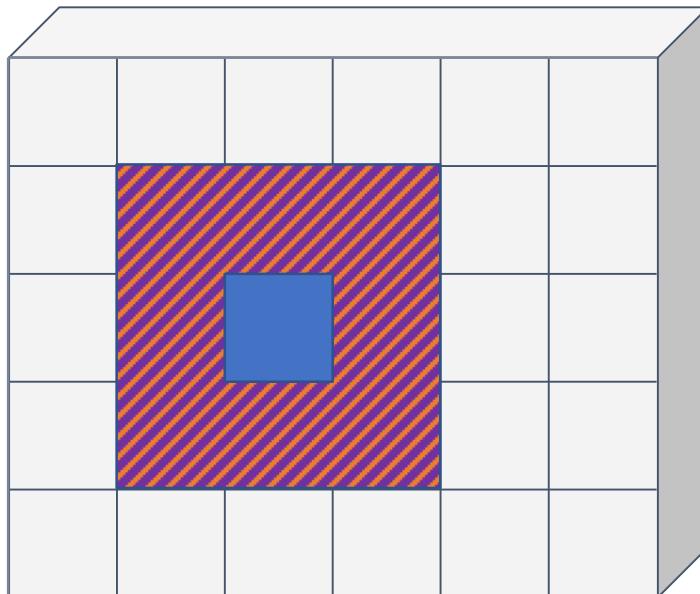
Output: $C' \times H \times W$

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



Input: $C \times H \times W$

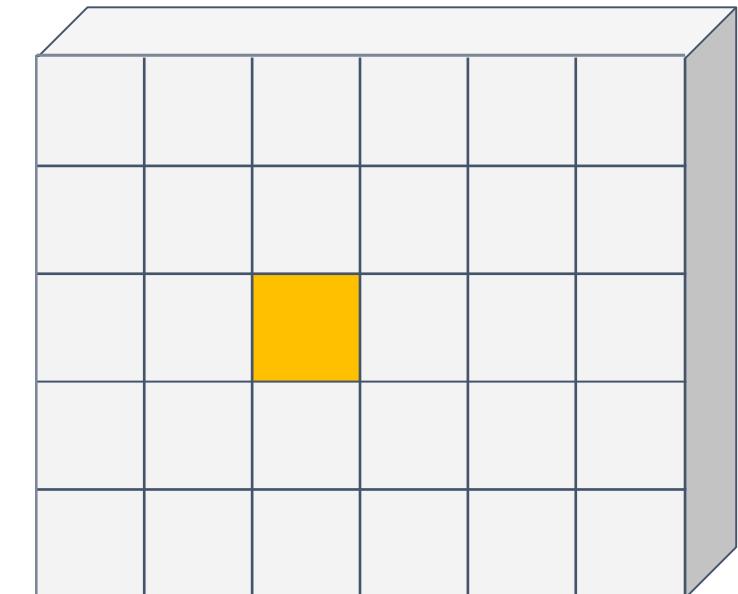
Query: D_Q

Keys: $R \times R \times D_Q$

Values: $R \times R \times C'$

↓
Attention
↑

Output: C



Output: $C' \times H \times W$

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 ($7 \times 7, m=8$)
res1	112×112	7×7 conv, 64, stride 2	$1 \times 1, 64$ 7×7 LR, 64, stride 2
		3×3 max pool, stride 2	3×3 max pool, stride 2
res2	56×56	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3 \text{ conv}, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 100 \\ 7 \times 7 \text{ LR, 100} \\ 1 \times 1, 256 \end{bmatrix} \times 3$
res3	28×28	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3 \text{ conv}, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 200 \\ 7 \times 7 \text{ LR, 200} \\ 1 \times 1, 512 \end{bmatrix} \times 4$
res4	14×14	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3 \text{ conv}, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 400 \\ 7 \times 7 \text{ LR, 400} \\ 1 \times 1, 1024 \end{bmatrix} \times 6$
res5	7×7	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3 \text{ conv}, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 800 \\ 7 \times 7 \text{ LR, 800} \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	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

Idea #2: Replace Convolution with “Local Attention”

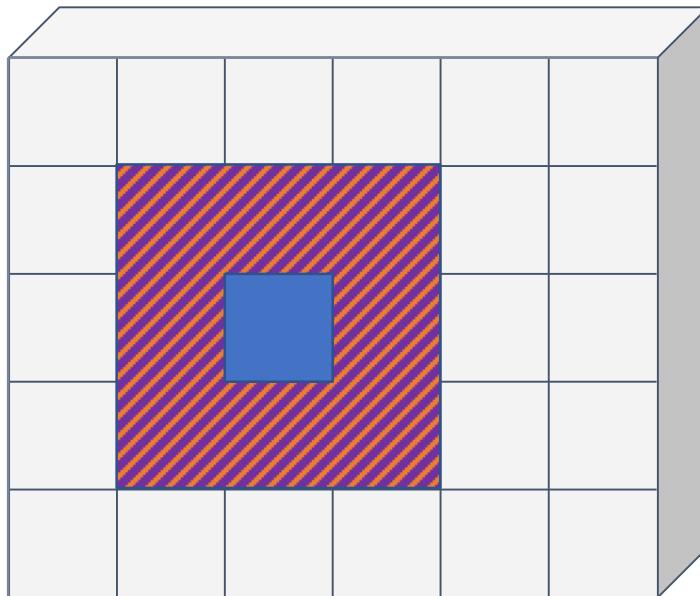
Map center of receptive field to **query**

Map each element in receptive field to **key** and **value**

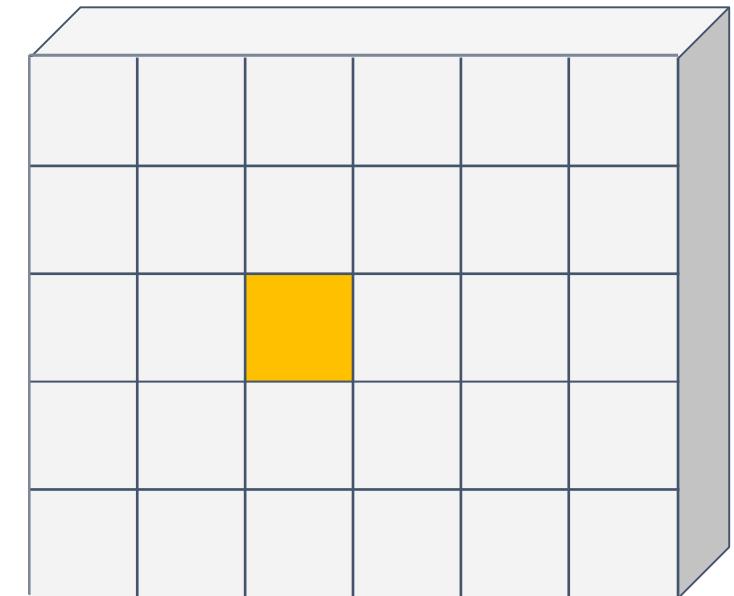
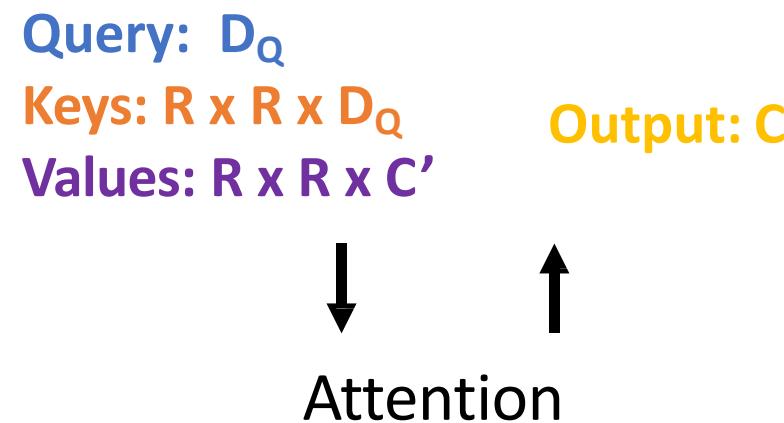
Compute **output** using attention

Replace all conv in ResNet with local attention

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



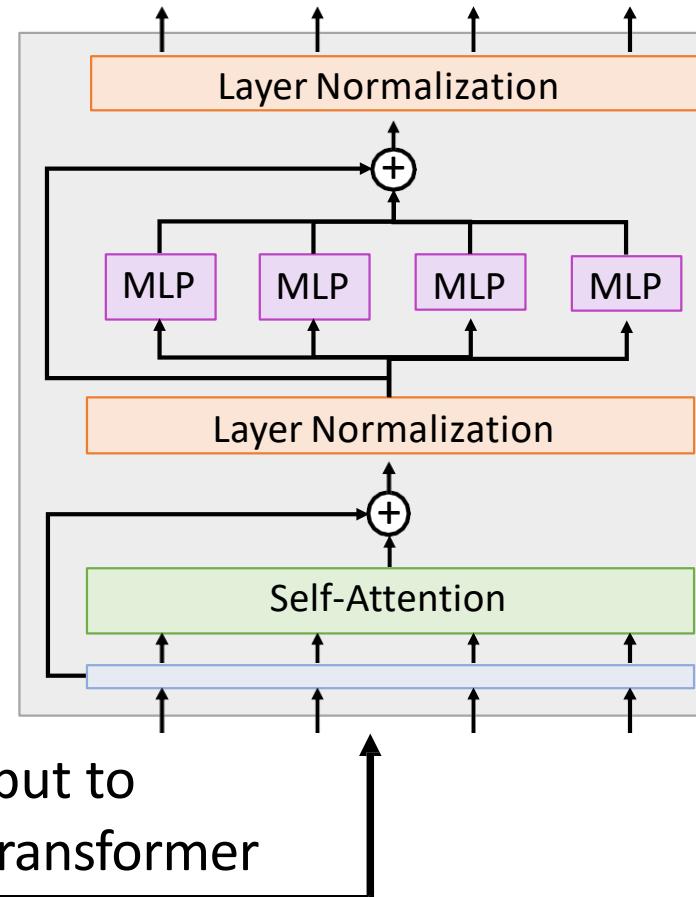
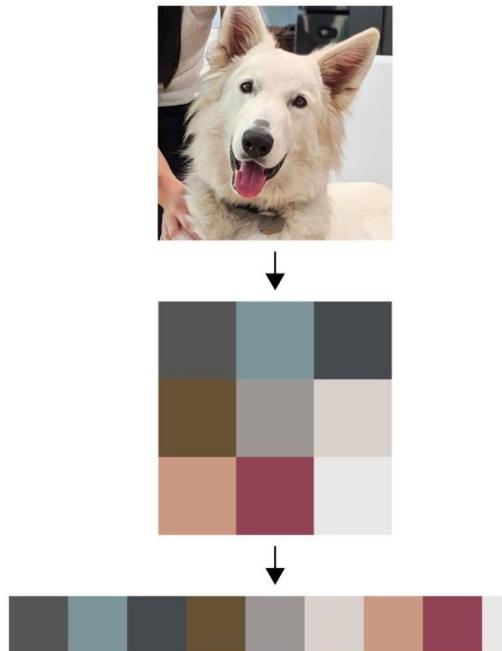
Input: $C \times H \times W$



Output: $C' \times H \times W$

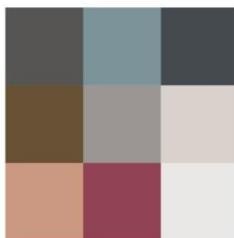
Idea #3: Standard Transformer on Pixels

Treat an image as a set of pixel values

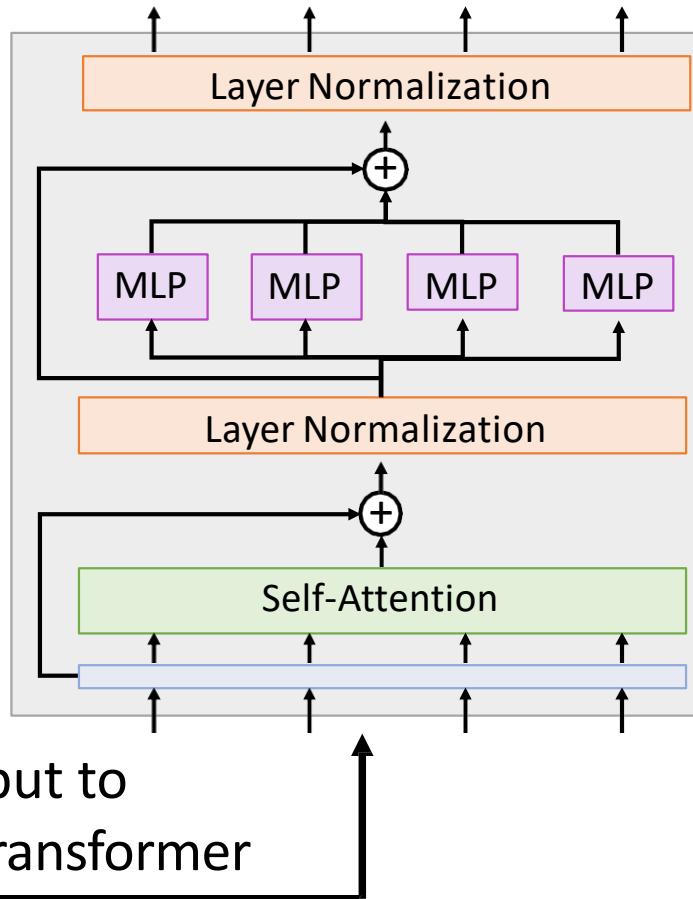


Idea #3: Standard Transformer on Pixels

Treat an image as a set of pixel values



Feed as input to standard Transformer



Problem: Memory use!

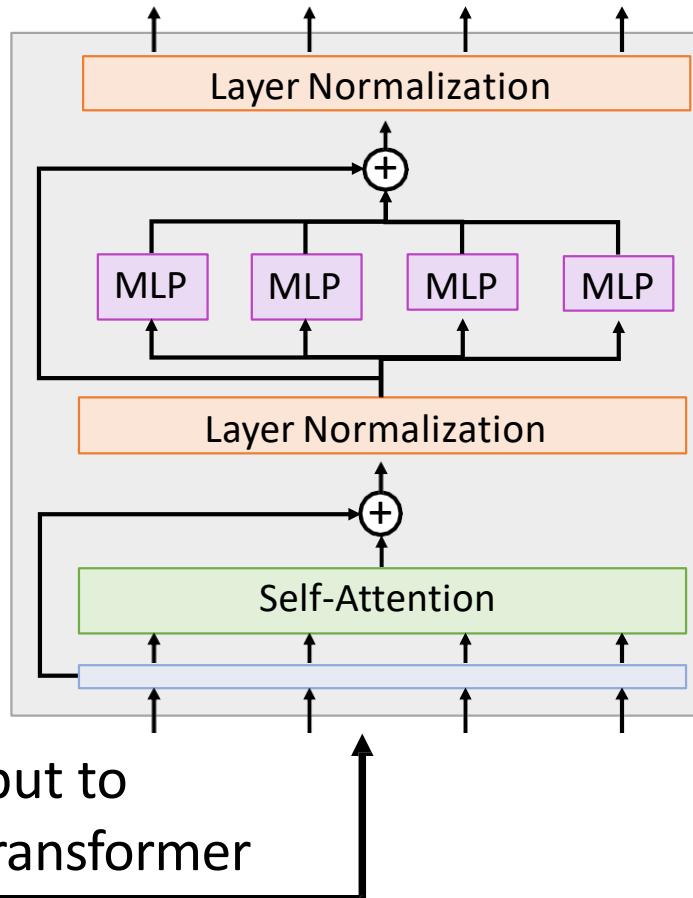
R x R image needs R^4 elements per attention matrix

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...

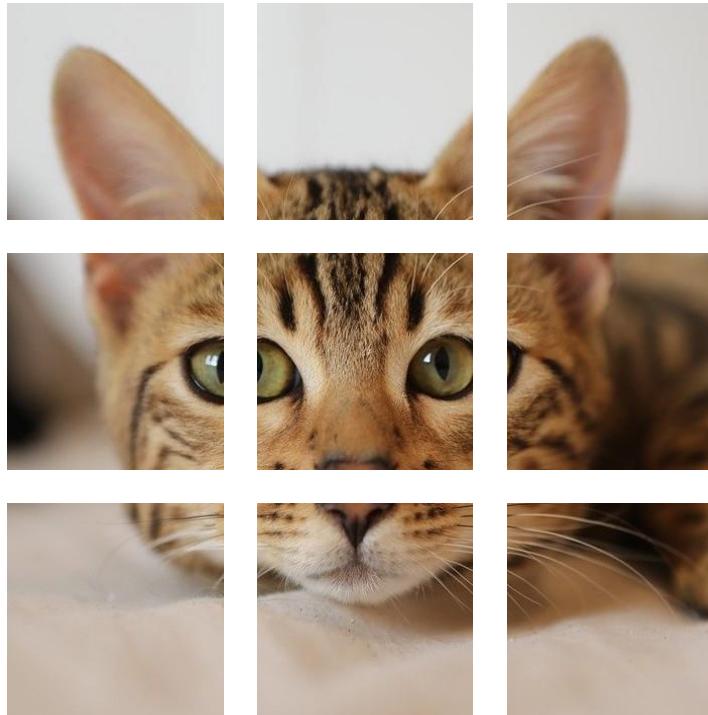
Idea #4: Standard Transformer on Patches



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

[Cat image](#) is free for commercial
use under a [Pixabay license](#)
45

Idea #4: Standard Transformer on Patches



Idea #4: Standard Transformer on Patches

N input patches, each
of shape 3x16x16



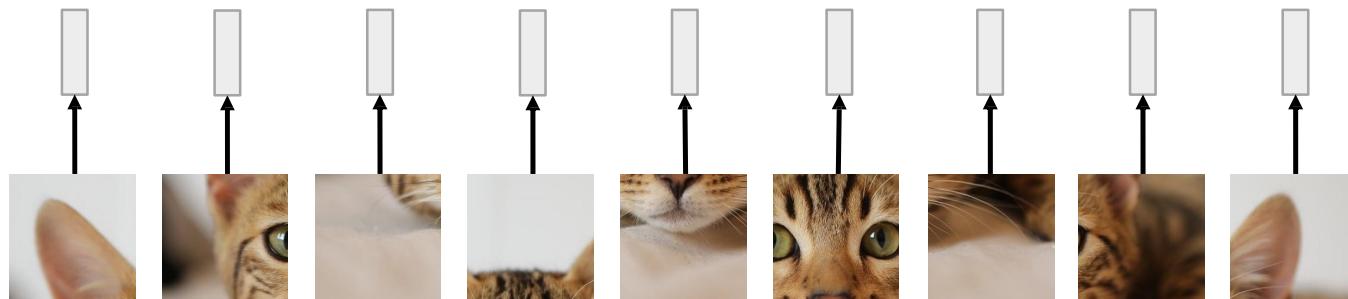
[Cat image](#) is free for commercial
use under a [Pixabay license](#)

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

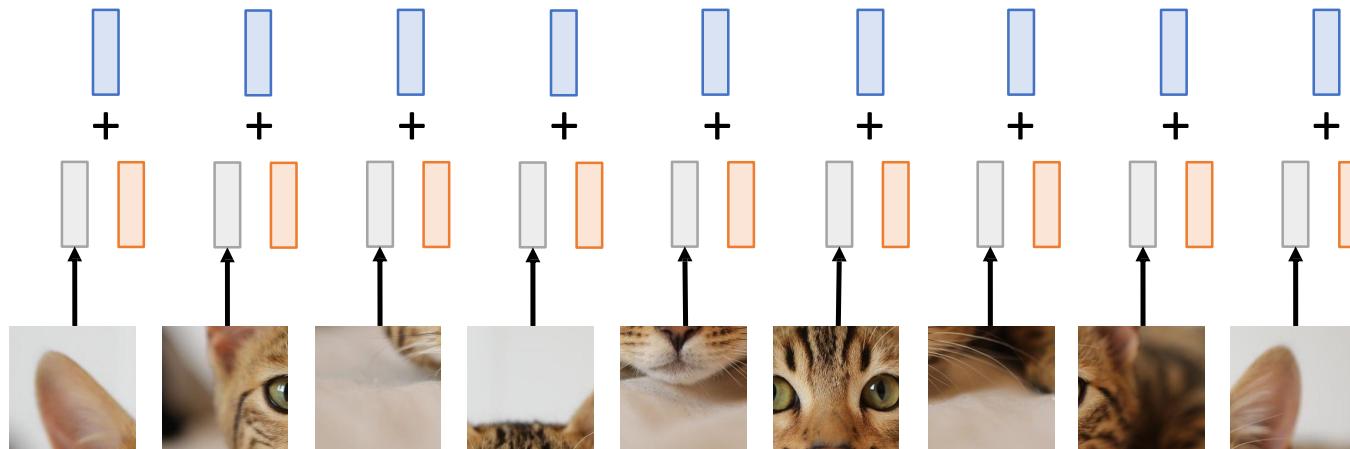
N input patches, each
of shape 3x16x16



[Cat image](#) is free for commercial
use under a [Pixelbay license](#)
48

Idea #4: Standard Transformer on Patches

Add positional embedding: learned D-dim vector per position



[Cat image](#) is free for commercial use under a [Pixabay license](#) 49

Idea #4: Standard Transformer on Patches

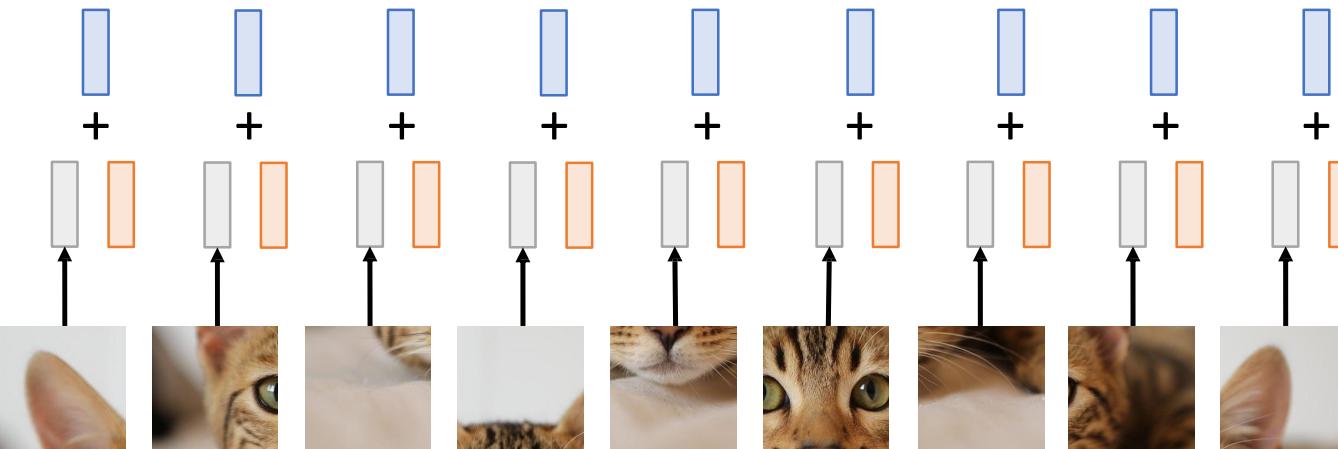
Output vectors



Exact same as
NLP Transformer!

Transformer

Add positional
embedding: learned D-
dim vector per position



Linear projection to
 D -dimensional vector

N input patches, each
of shape $3 \times 16 \times 16$

Idea #4: Standard Transformer on Patches

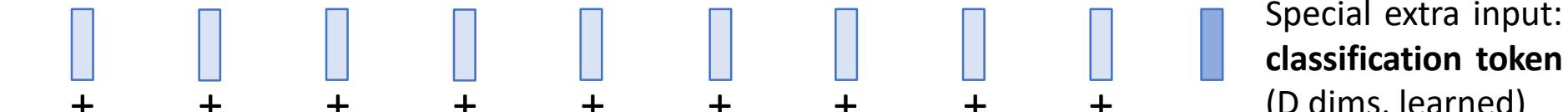
Output vectors



Exact same as
NLP Transformer!

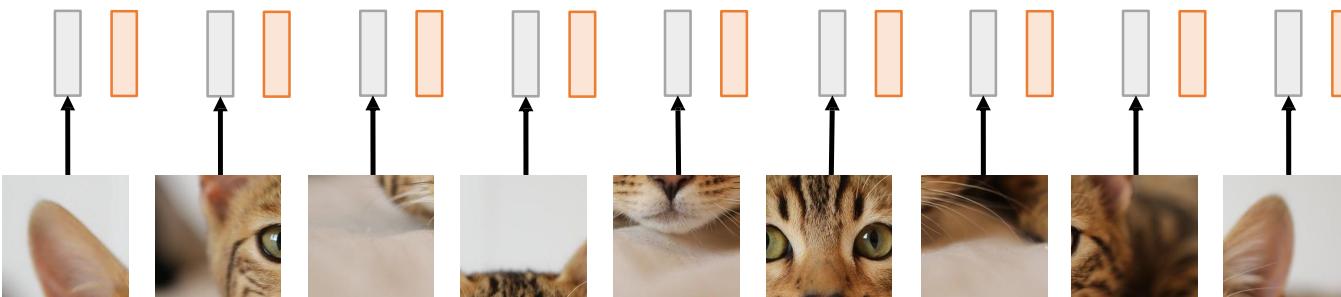
Transformer

Add positional
embedding: learned D-
dim vector per position



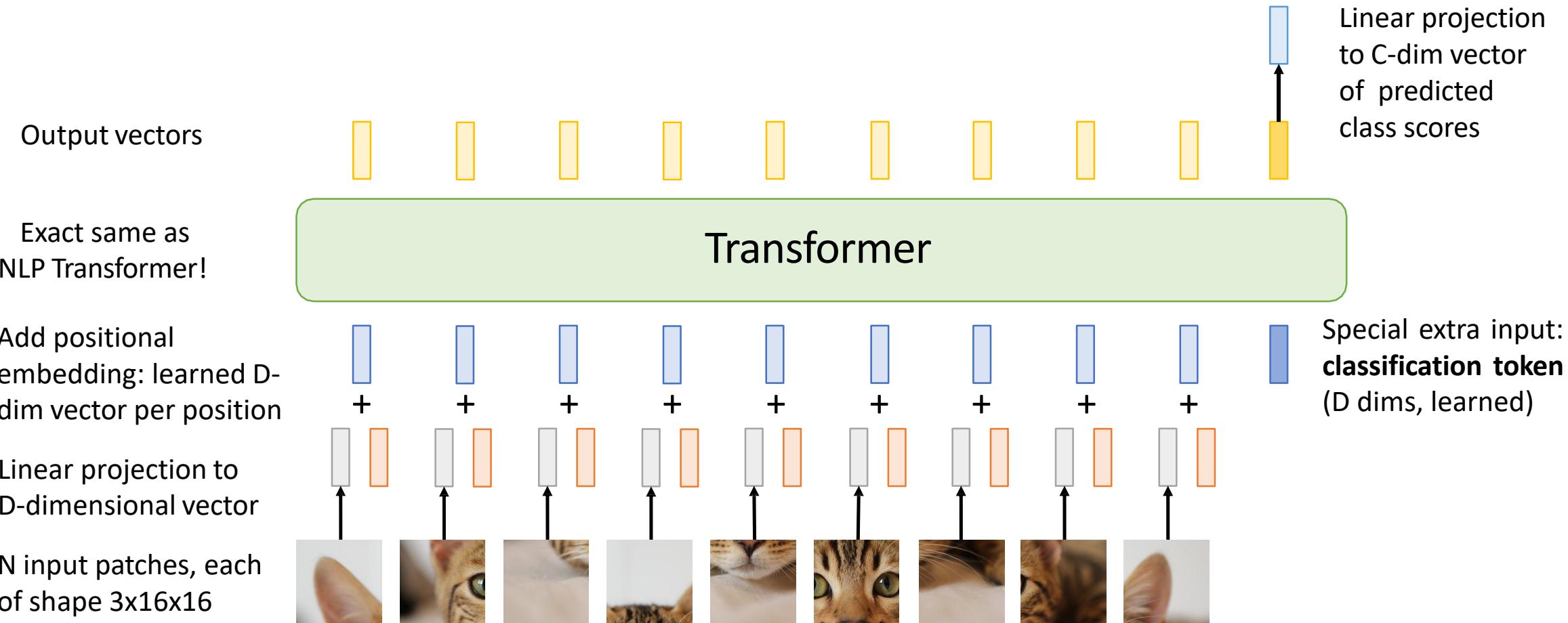
Special extra input:
classification token
(D dims, learned)

Linear projection to
D-dimensional vector



[Cat image](#) is free for commercial
use under a [Pixabay license](#)

Idea #4: Standard Transformer on Patches



Vision Transformer (ViT)

Computer vision model
with no convolutions!

Output vectors



Linear projection
to C-dim vector
of predicted
class scores

Exact same as
NLP Transformer!

Transformer

Add positional
embedding: learned D-
dim vector per position



Special extra input:
classification token
(D dims, learned)

Linear projection to
D-dimensional vector



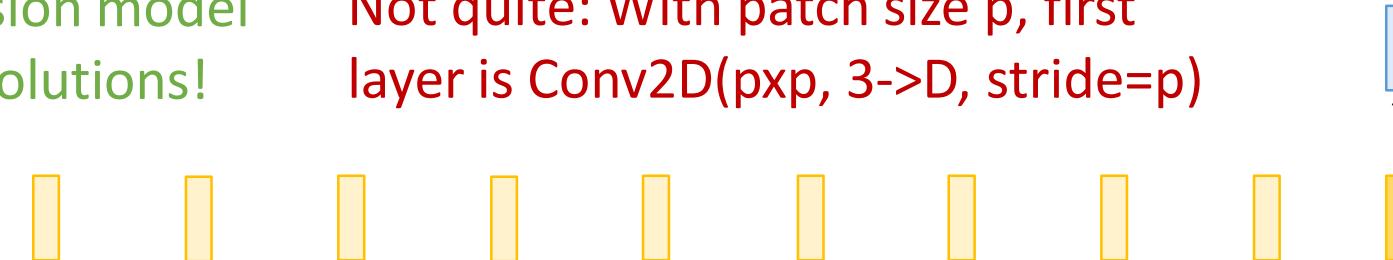
N input patches, each
of shape 3x16x16

Vision Transformer (ViT)

Computer vision model
with no convolutions!

Not quite: With patch size p , first
layer is $\text{Conv2D}(pxp, 3 \rightarrow D, \text{stride}=p)$

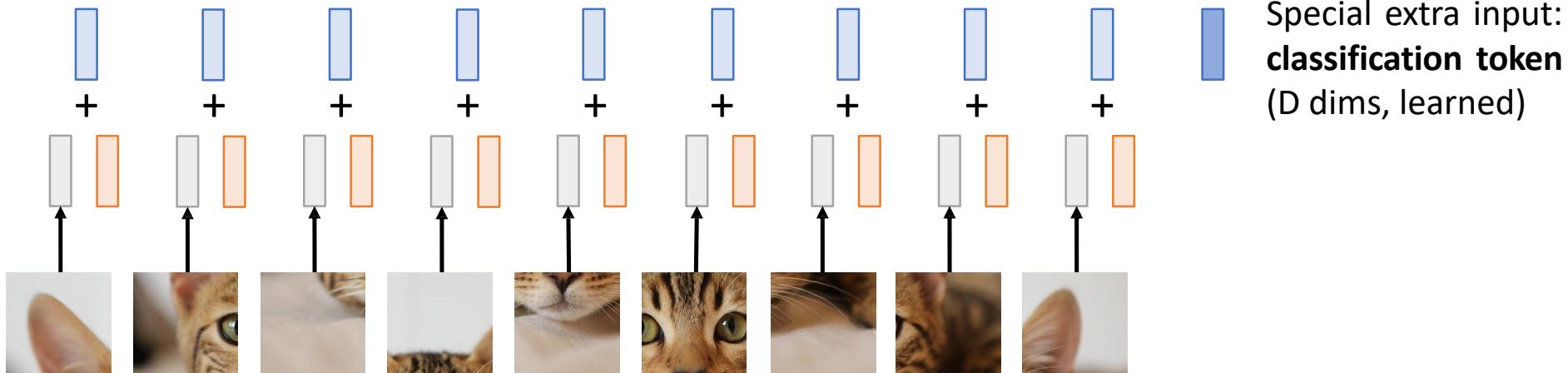
Output vectors



Exact same as
NLP Transformer!

Transformer

Add positional
embedding: learned D -
dim vector per position



Linear projection to
 D -dimensional vector

N input patches, each
of shape $3 \times 16 \times 16$

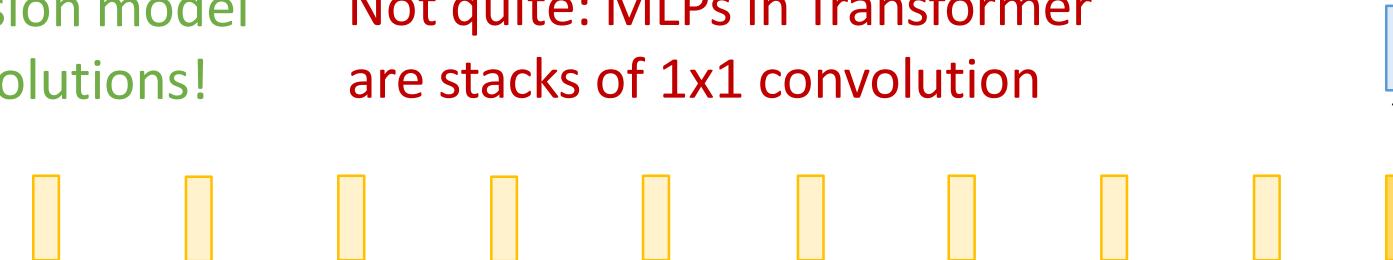
[Cat image](#) is free for commercial
use under a [Pixabay license](#)

Vision Transformer (ViT)

Computer vision model
with no convolutions!

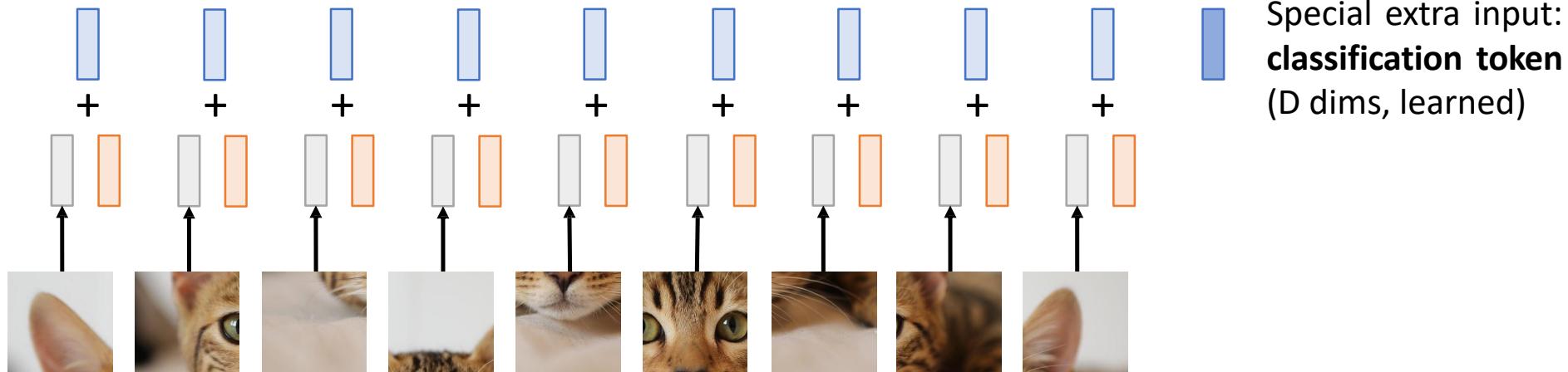
Not quite: MLPs in Transformer
are stacks of 1x1 convolution

Output vectors



Exact same as
NLP Transformer!

Add positional
embedding: learned D-
dim vector per position



Linear projection to
D-dimensional vector

N input patches, each
of shape 3x16x16

Linear projection
to C-dim vector
of predicted
class scores

Transformer

Special extra input:
classification token
(D dims, learned)

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)

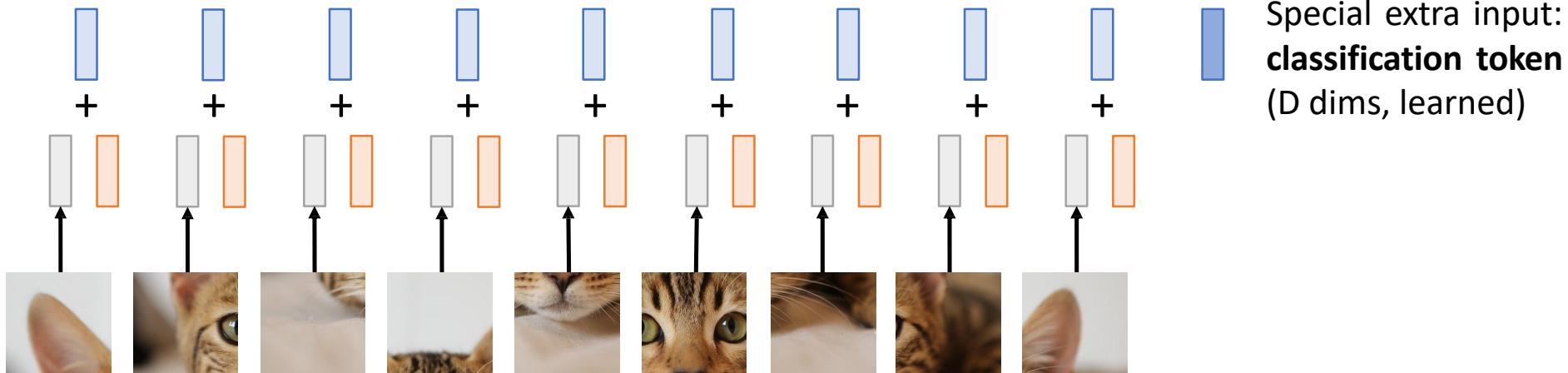
Output vectors



Linear projection to C-dim vector of predicted class scores

Exact same as NLP Transformer!

Add positional embedding: learned D-dim vector per position



Vision Transformer (ViT)

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

Output vectors



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

Linear projection to C-dim vector of predicted class scores

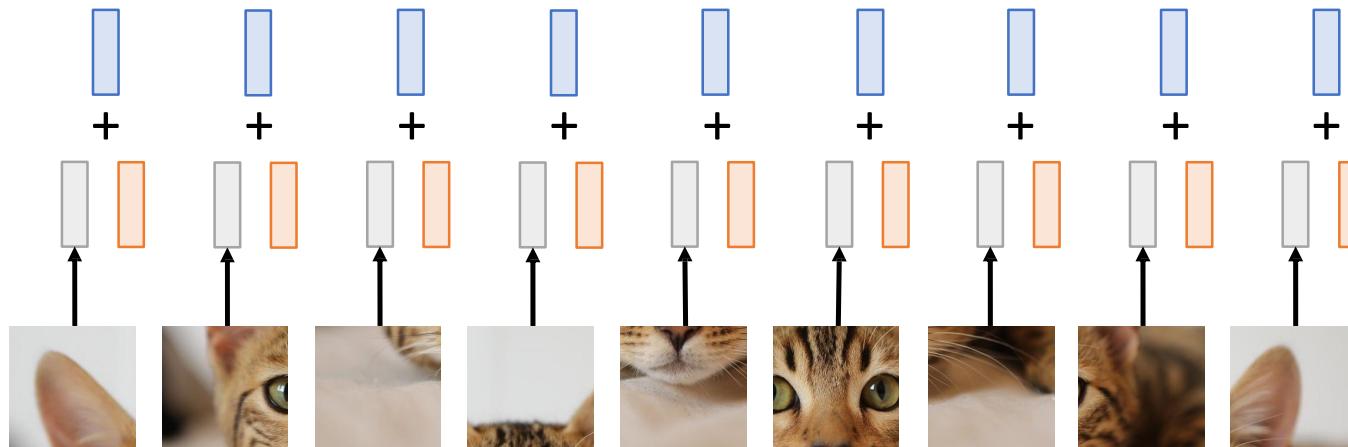
Exact same as NLP Transformer!

Add positional embedding: learned D-dim vector per position

Transformer

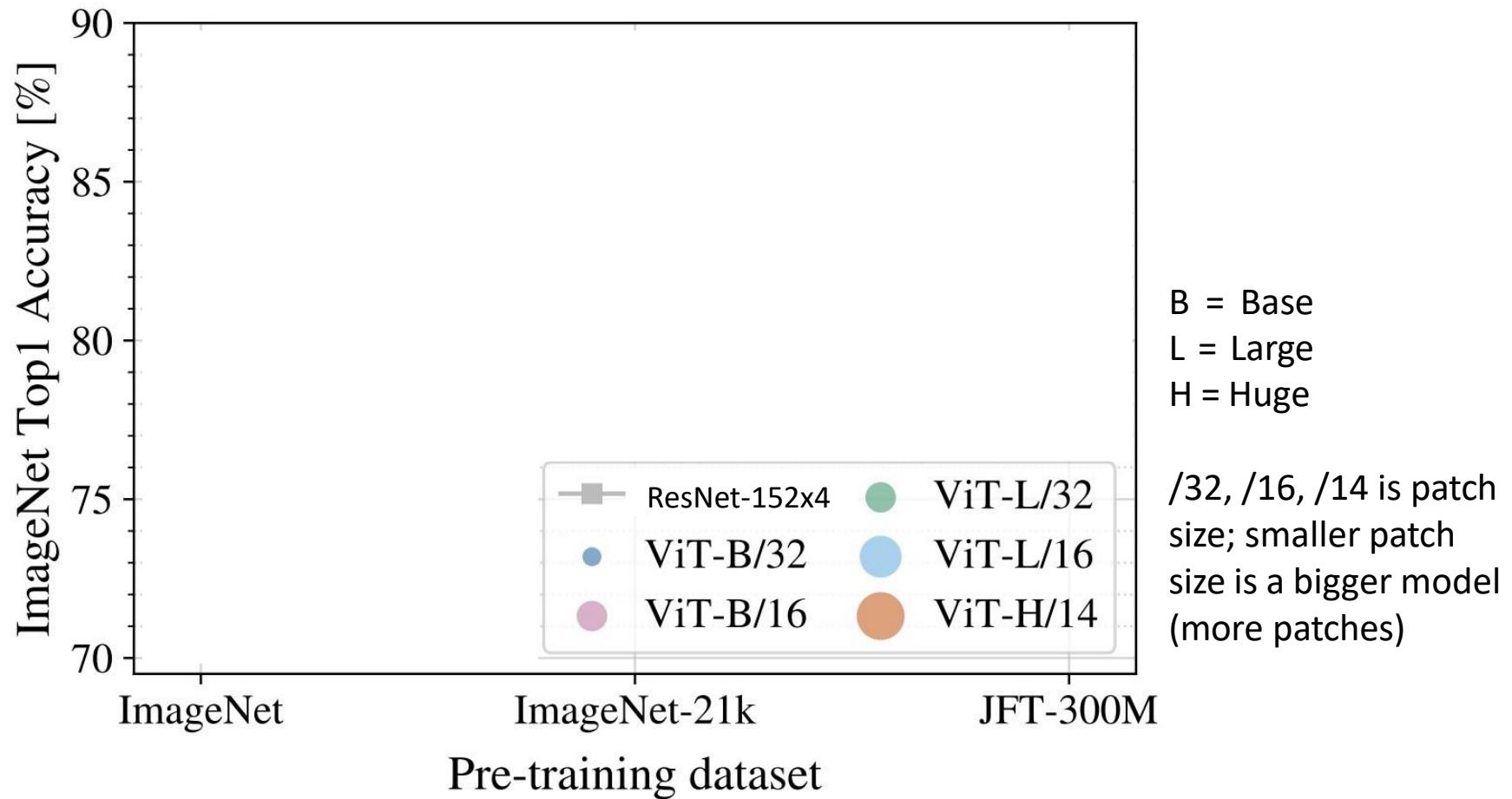
Linear projection to D-dimensional vector

N input patches, each of shape 3x16x16



[Cat image](#) is free for commercial use under a [Pixabay license](#)

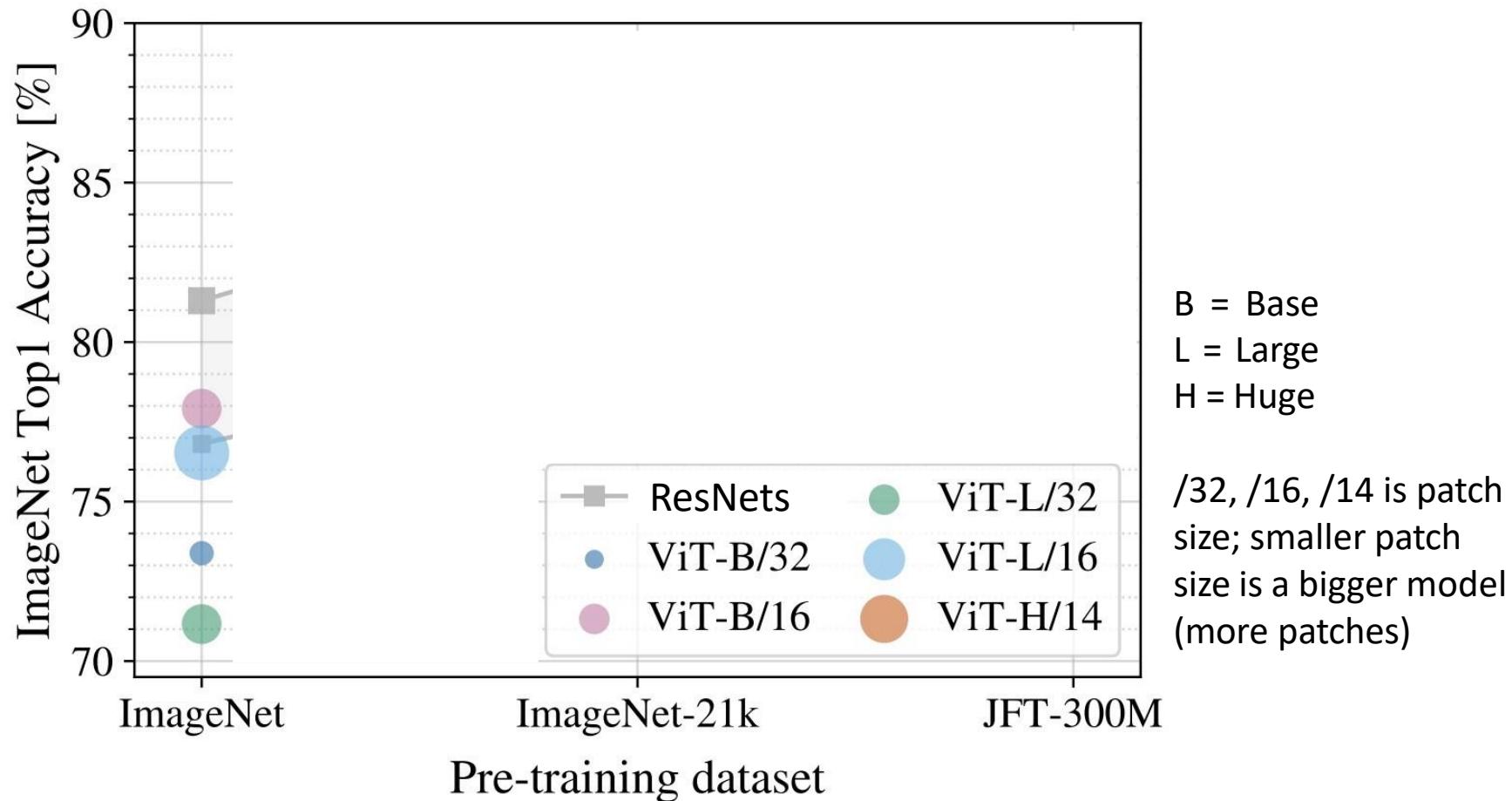
Vision Transformer (ViT) vs ResNets



Vision Transformer (ViT) vs ResNets

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

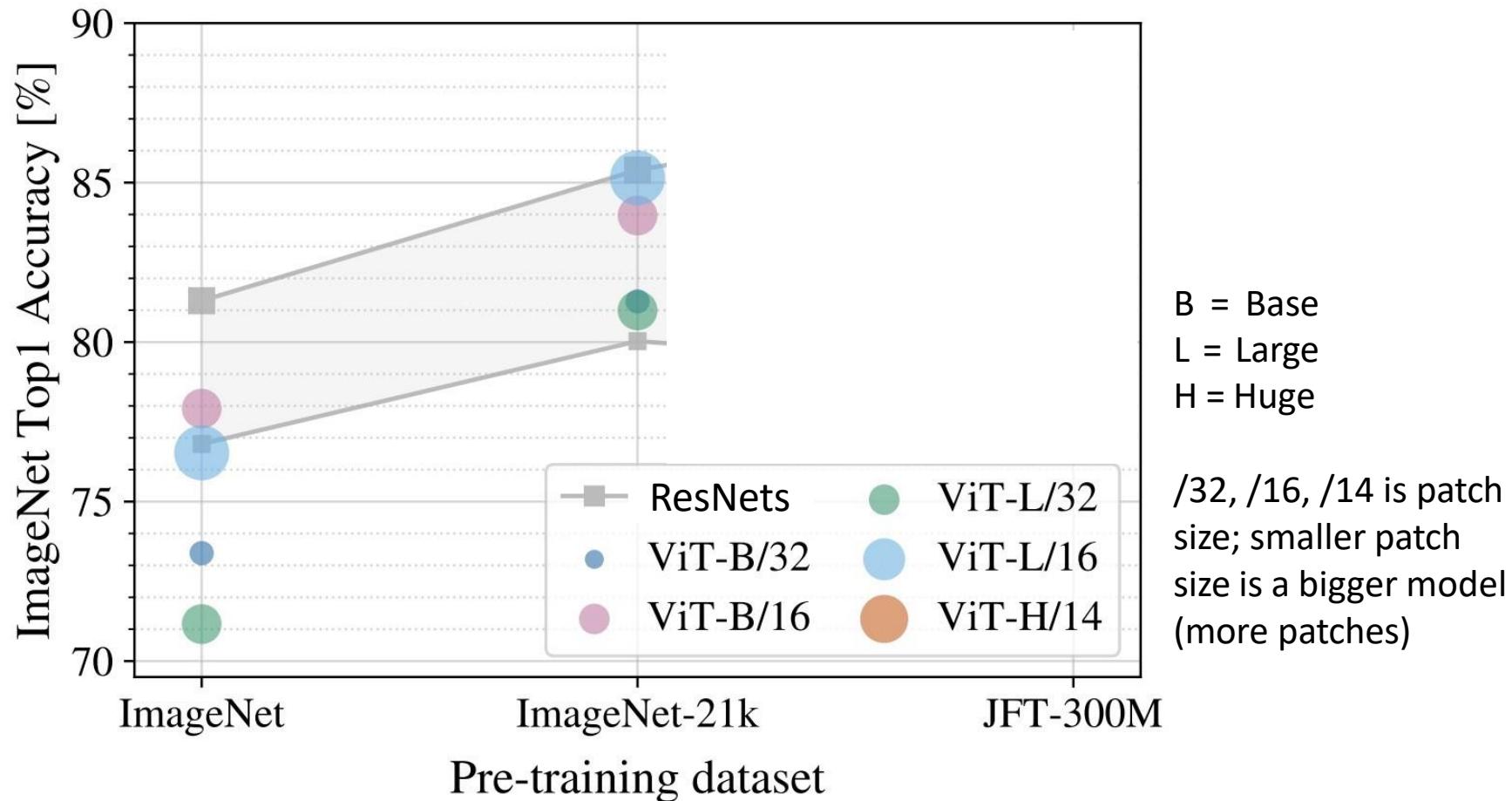
When trained on ImageNet, ViT models perform worse than ResNets



Vision Transformer (ViT) vs ResNets

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

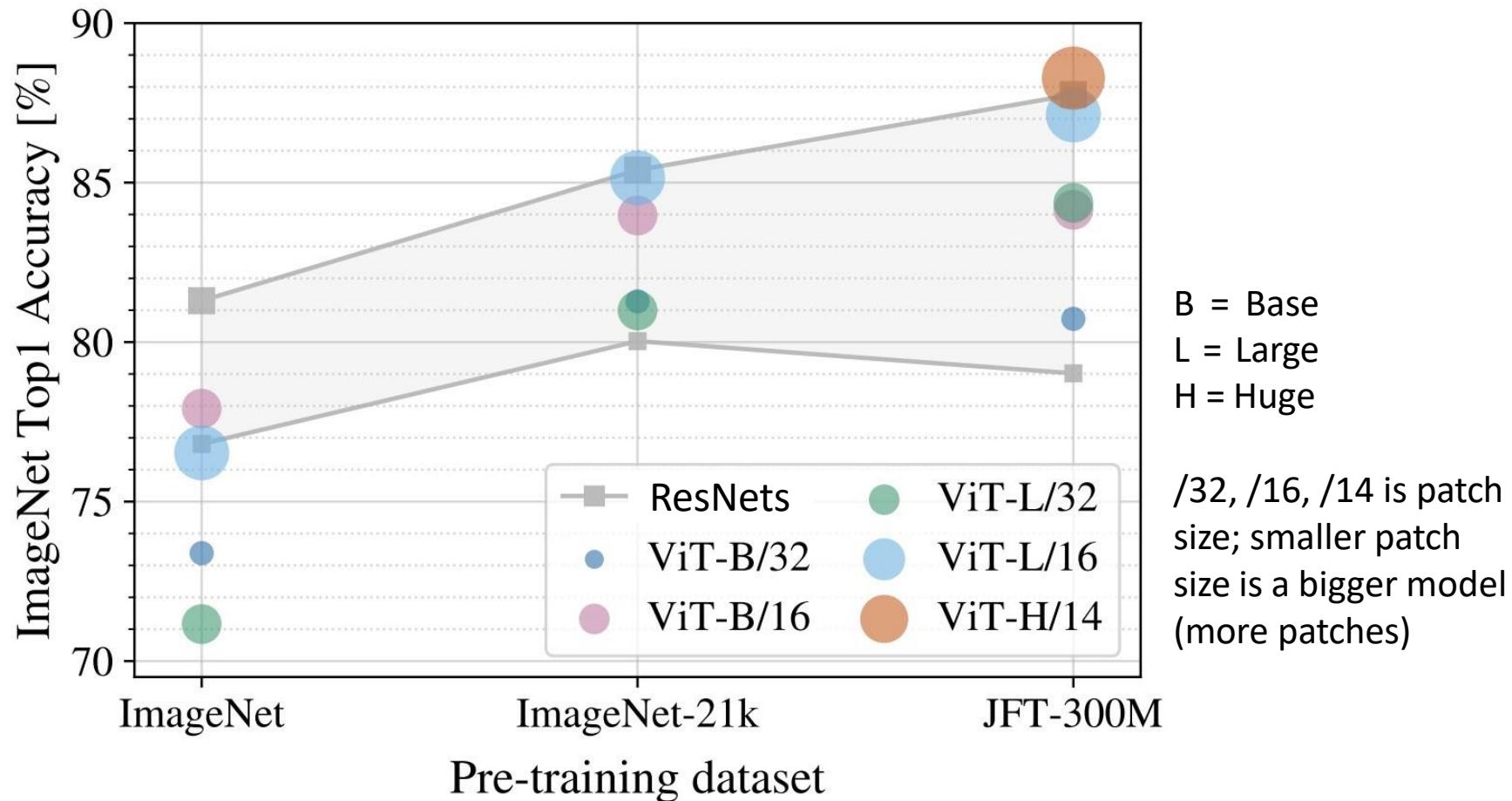
If you pretrain on
ImageNet-21k and
fine-tune on
ImageNet, ViT does
better: big ViTs match
big ResNets



Vision Transformer (ViT) vs ResNets

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

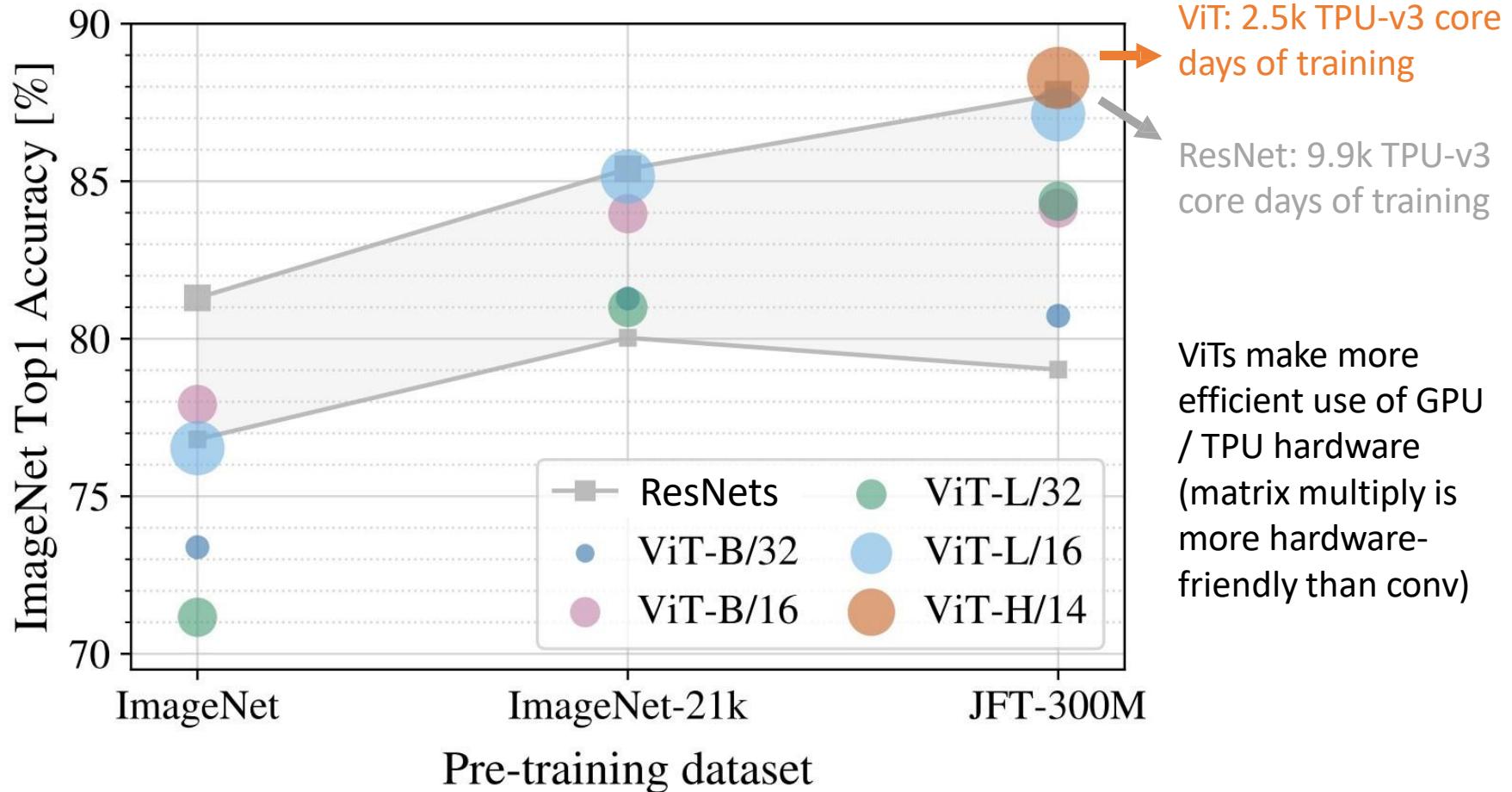
If you pretrain on JFT and finetune on ImageNet, large ViTs outperform large ResNets



Vision Transformer (ViT) vs ResNets

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

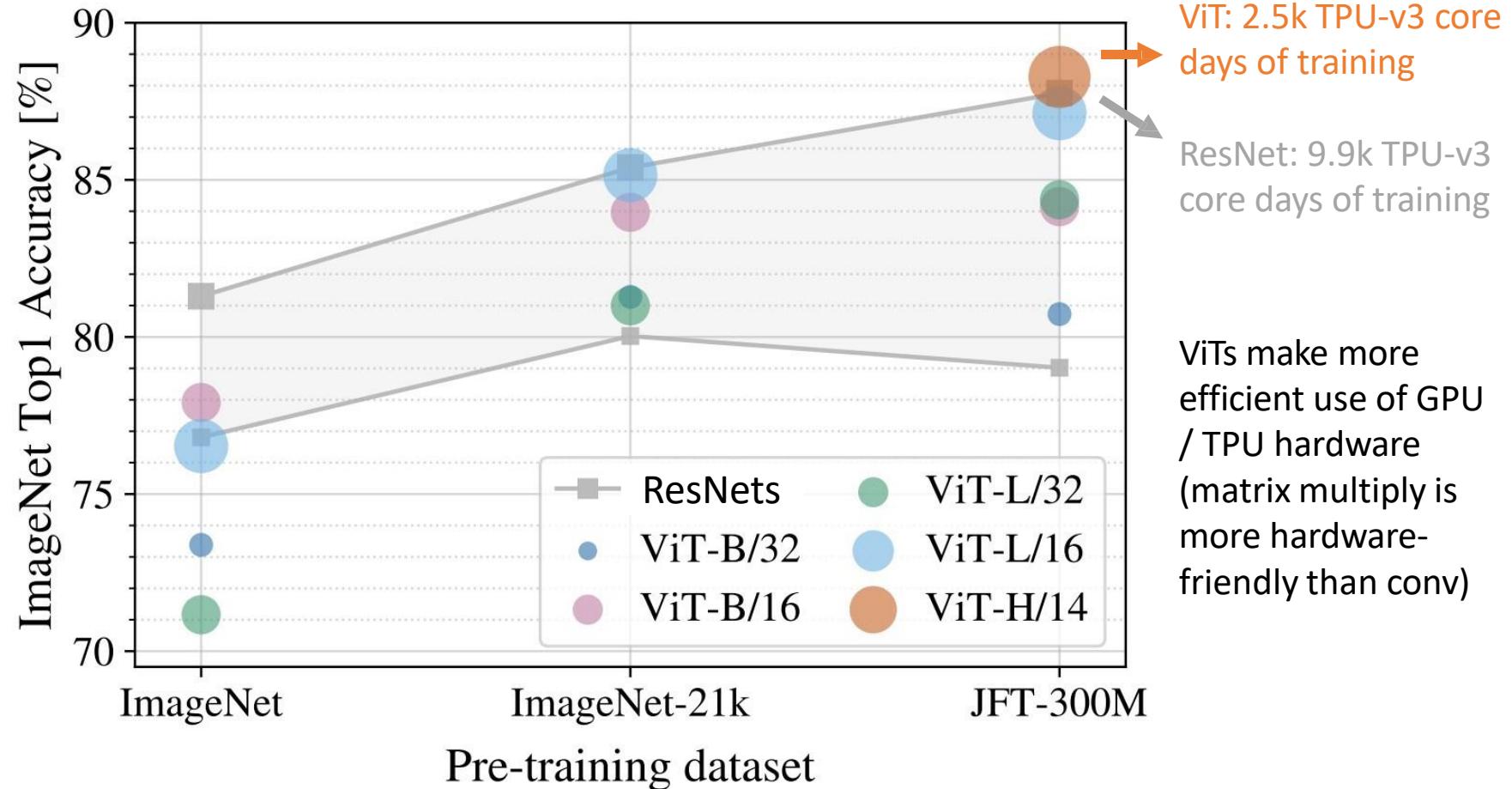
If you pretrain on JFT and finetune on ImageNet, large ViTs outperform large ResNets



Vision Transformer (ViT) vs ResNets

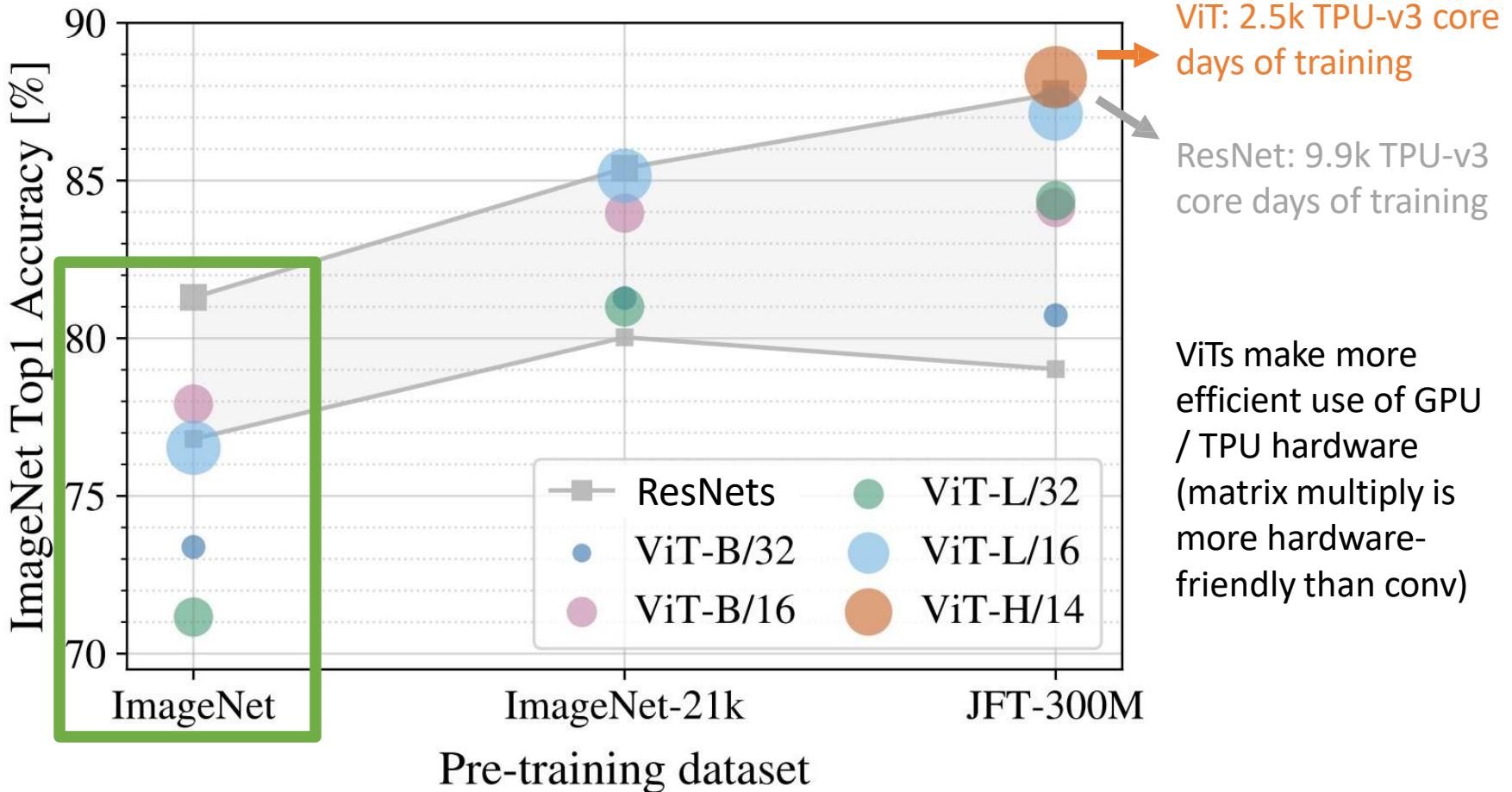
Claim: ViT models have “less inductive bias” than ResNets, so need more pretraining data to learn good features

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



Vision Transformer (ViT) vs ResNets

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



Improving ViT: Augmentation and Regularization

Regularization for ViT models:

- Weight Decay
- Stochastic Depth
- Dropout (in FFN layers of Transformer)

Data Augmentation for ViT

models:

- MixUp
- RandAugment

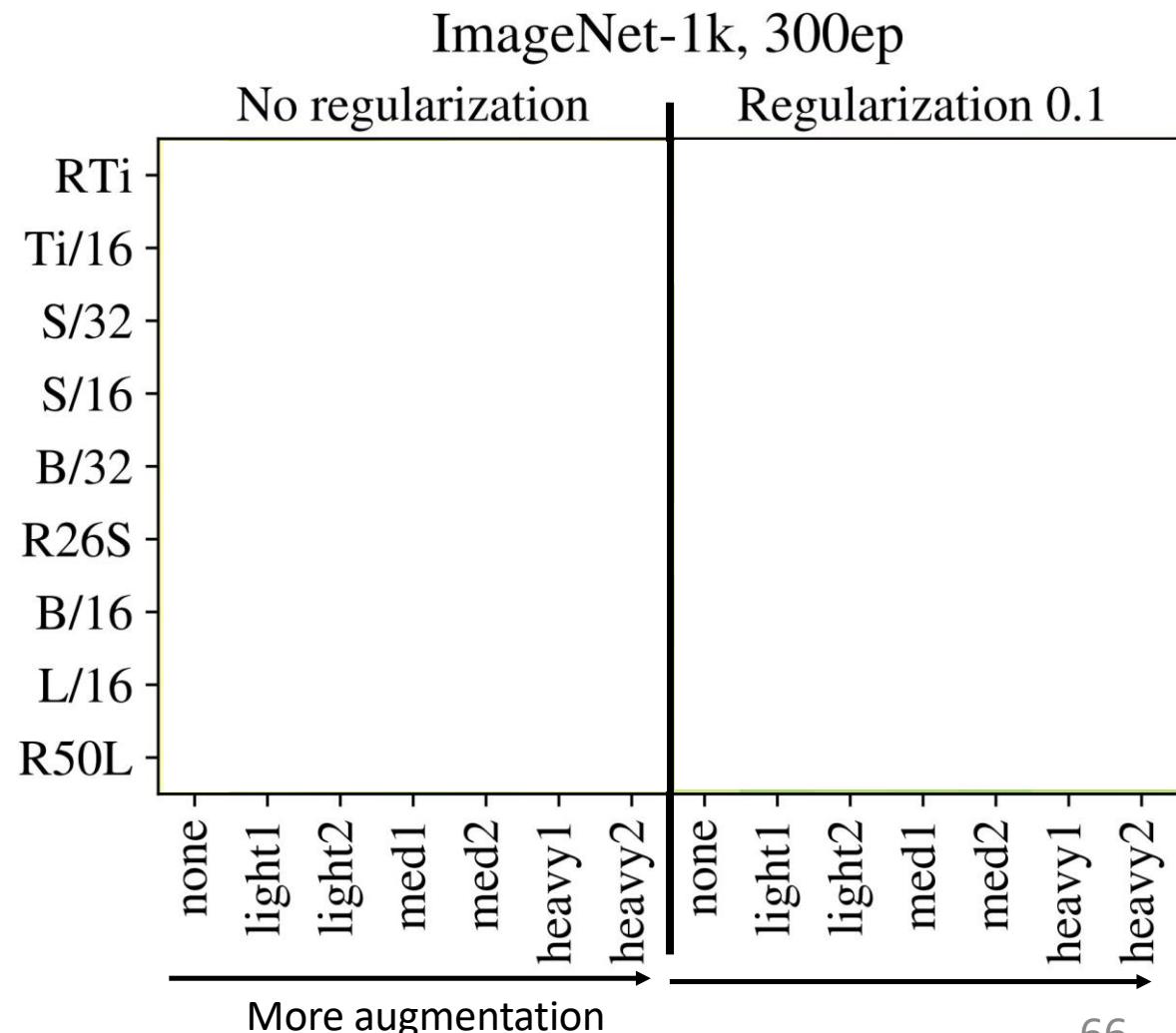
Improving ViT: Augmentation and Regularization

Regularization for ViT models:

- Weight Decay
- Stochastic Depth
- Dropout (in FFN layers of Transformer)

Data Augmentation for ViT models:

- MixUp
- RandAugment



Improving ViT: Augmentation and Regularization

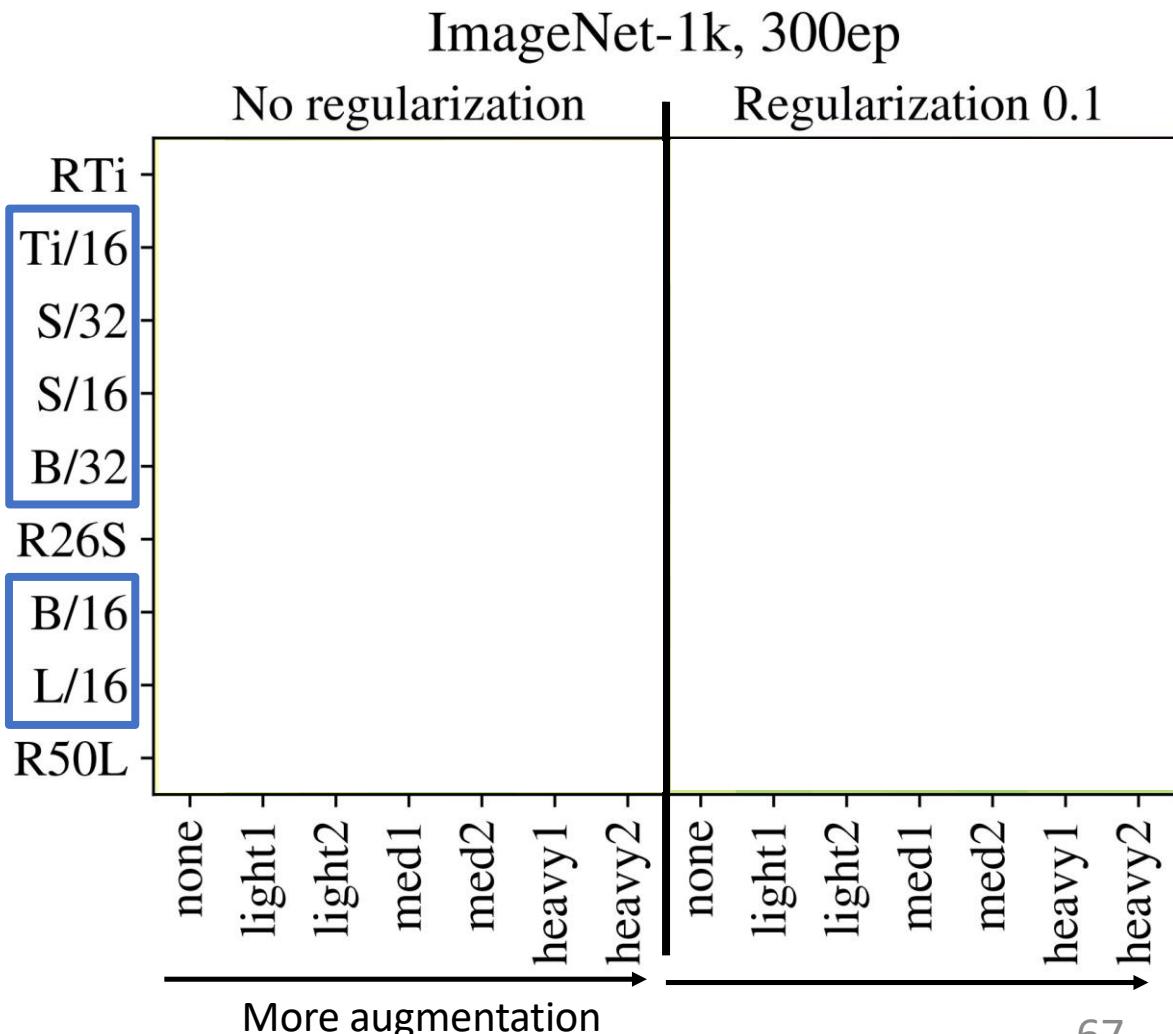
Regularization for ViT models:

- Weight Decay
- Stochastic Depth
- Dropout (in FFN layers of Transformer)

Data Augmentation for ViT models:

- MixUp
- RandAugment

ViT models:
Ti = Tiny
S = Small
B = Base
L = Large



Improving ViT: Augmentation and Regularization

Regularization for ViT models:

- Weight Decay
- Stochastic Depth
- Dropout (in FFN layers of Transformer)

Data Augmentation for ViT models:

- MixUp
- RandAugment

Hybrid models:

ResNet blocks,
then ViT blocks

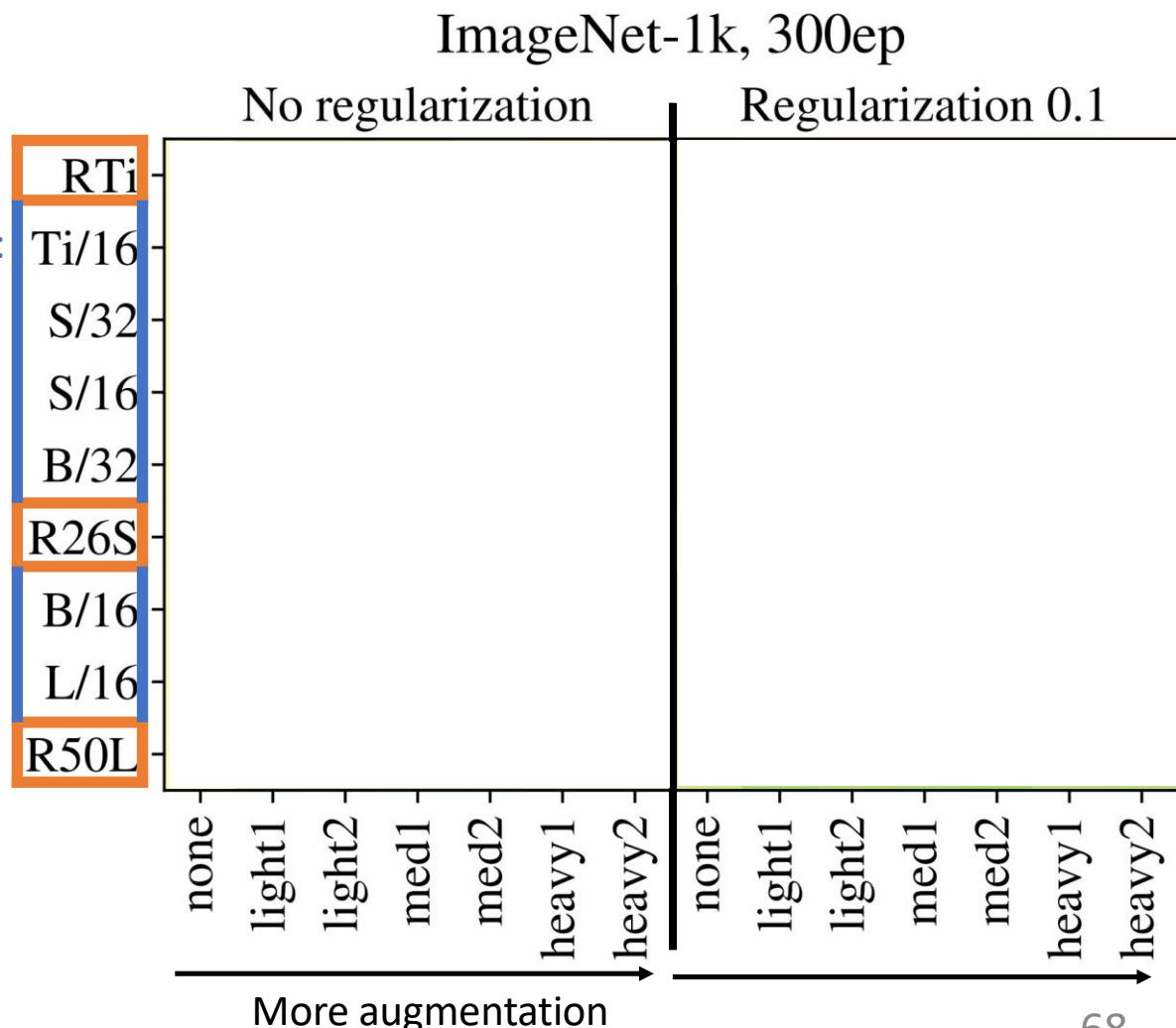
ViT models:

Ti = Tiny

S = Small

B = Base

L = Large



Improving ViT: Augmentation and Regularization

Regularization for ViT models:

- Weight Decay
- Stochastic Depth
- Dropout (in FFN layers of Transformer)

Data Augmentation for ViT models:

- MixUp
- RandAugment

Hybrid models:

ResNet blocks,
then ViT blocks

ViT models:

Ti = Tiny

S = Small

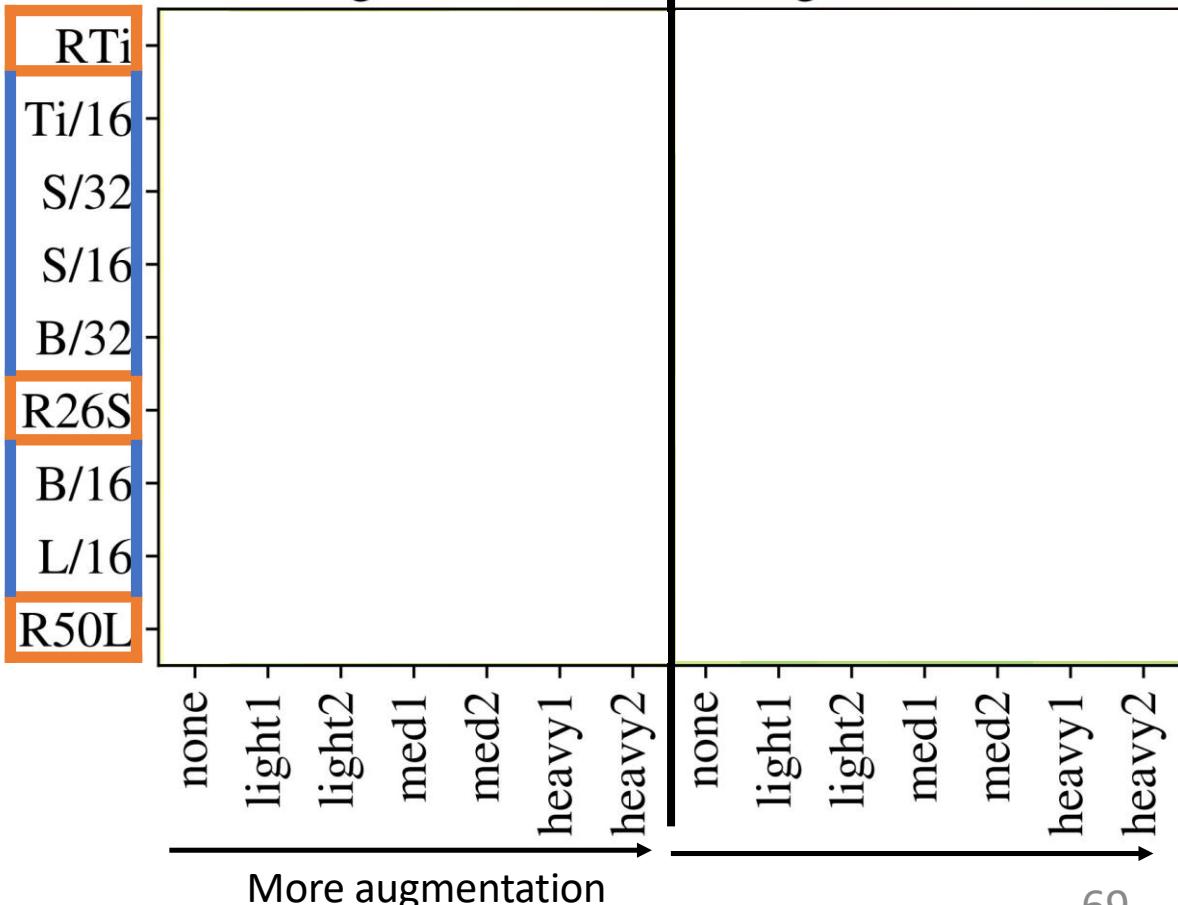
B = Base

L = Large

Original Paper:

77.9

76.53



Improving ViT: Augmentation and Regularization

Regularization for ViT models:

- Weight Decay
- Stochastic Depth
- Dropout (in FFN layers of Transformer)

Data Augmentation for ViT models:

- MixUp
- RandAugment

Adding regularization is
(almost) always helpful

Hybrid models:

ResNet blocks,
then ViT blocks

ViT models:

Ti = Tiny

S = Small

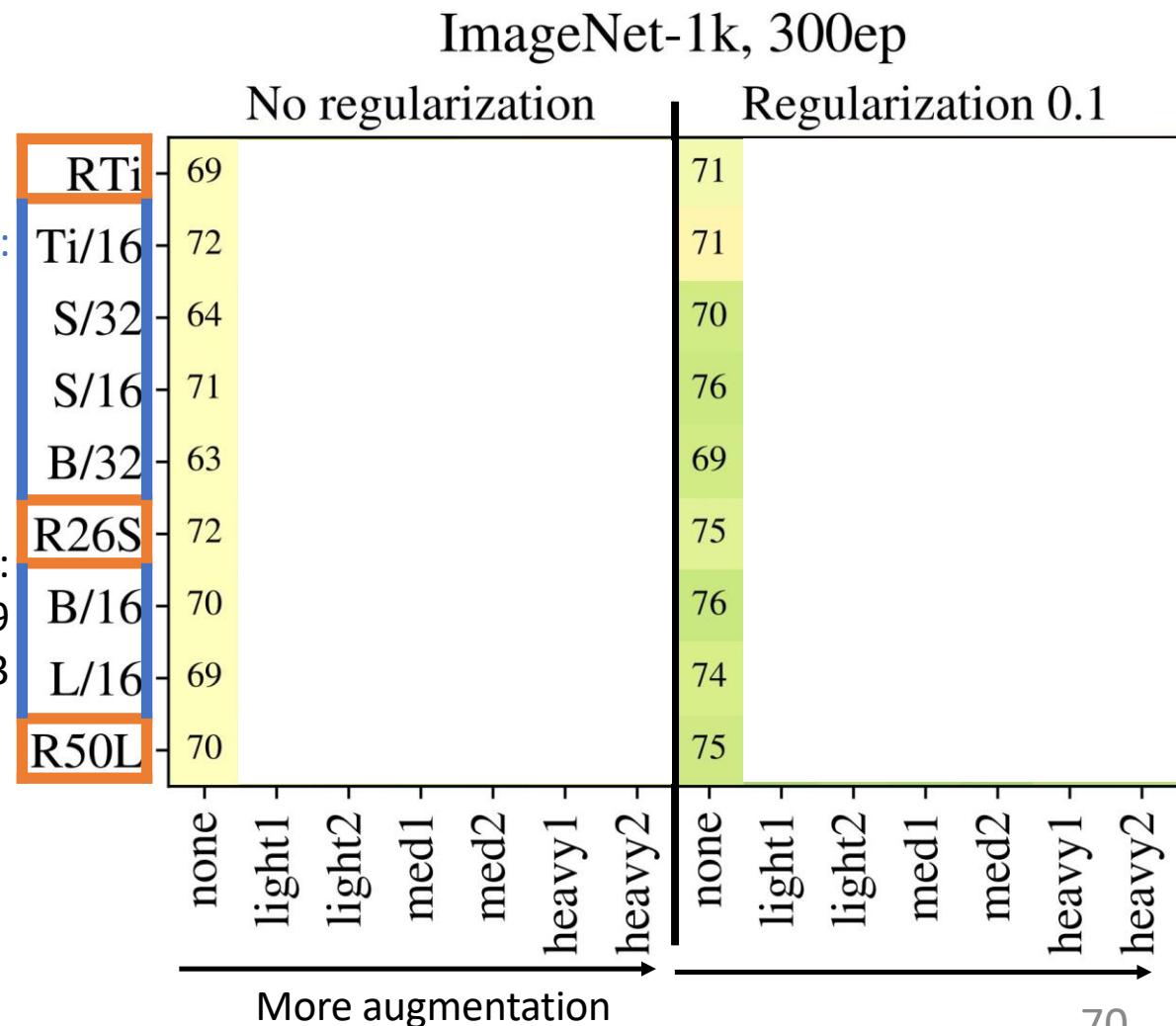
B = Base

L = Large

Original Paper:

77.9

76.53



Improving ViT: Augmentation and Regularization

Regularization for ViT models:

- Weight Decay
- Stochastic Depth
- Dropout (in FFN layers of Transformer)

Data Augmentation for ViT models:

- MixUp
- RandAugment

Regularization +
Augmentation gives
big improvements
over original results

Hybrid models:

ResNet blocks,
then ViT blocks

ViT models:

Ti = Tiny

S = Small

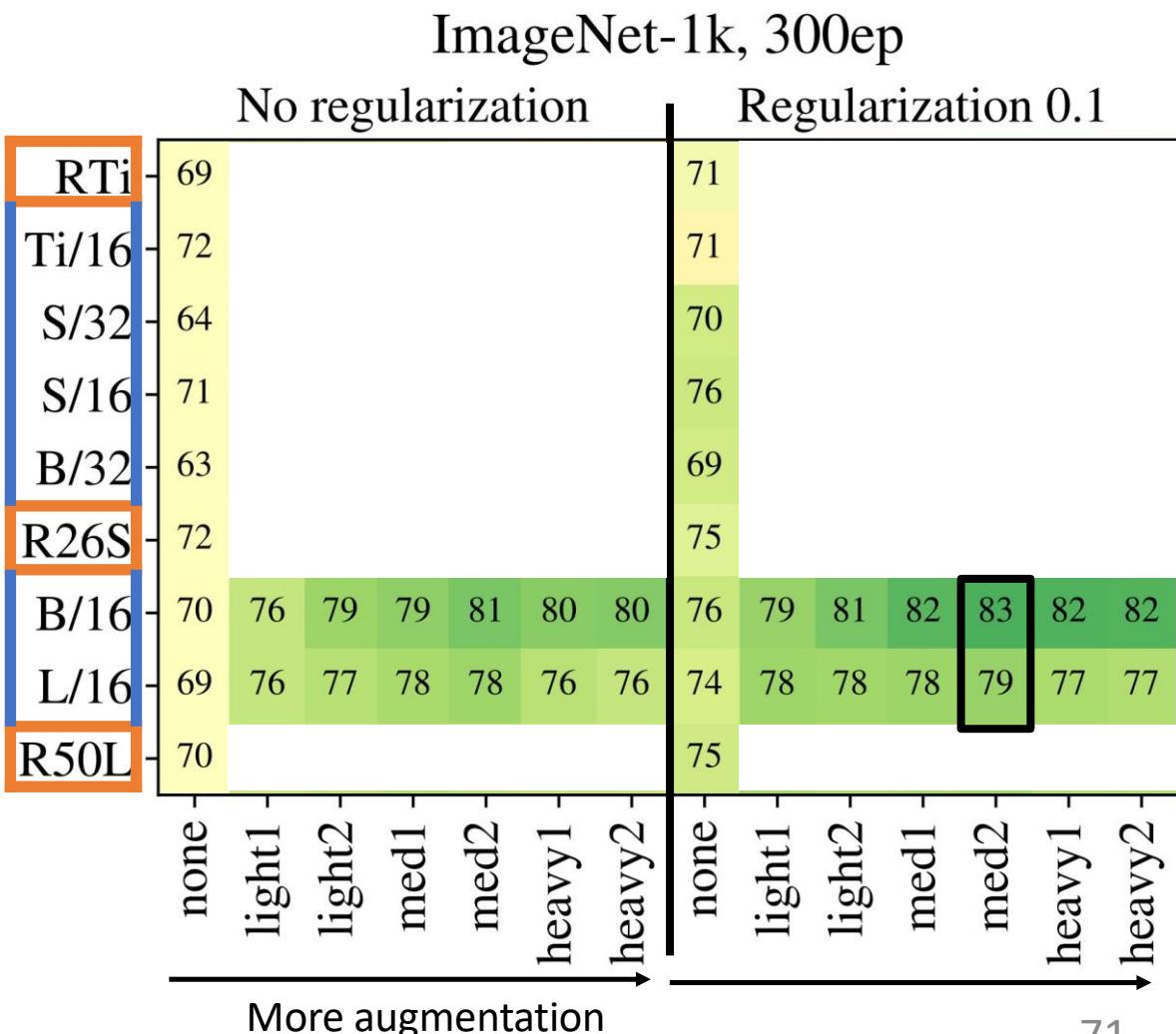
B = Base

L = Large

Original Paper:

77.9

76.53



Improving ViT: Augmentation and Regularization

Regularization for ViT models:

- Weight Decay
- Stochastic Depth
- Dropout (in FFN layers of Transformer)

Data Augmentation for ViT models:

- MixUp
- RandAugment

Hybrid models:

ResNet blocks,
then ViT blocks

ViT models:

Ti = Tiny

S = Small

B = Base

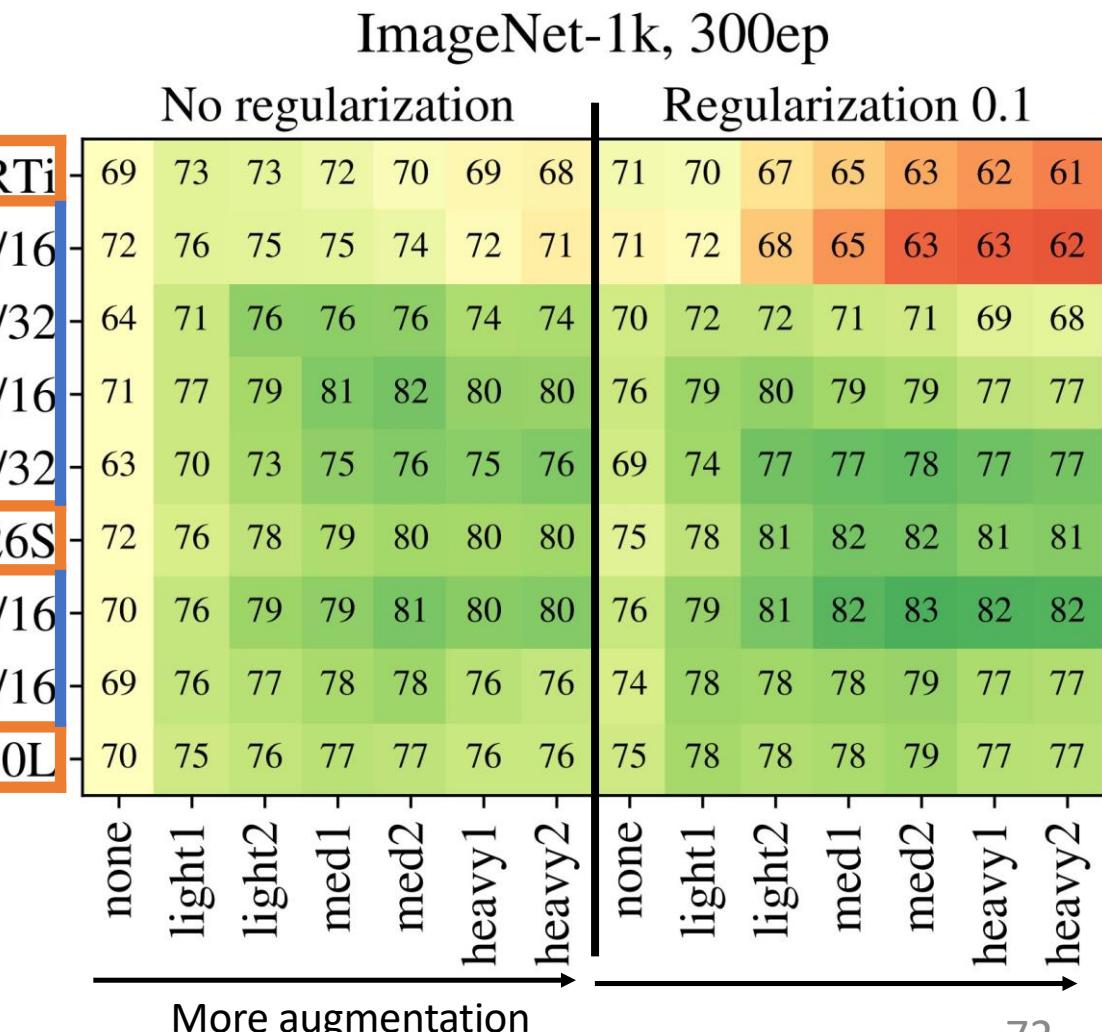
L = Large

Original Paper:

77.9

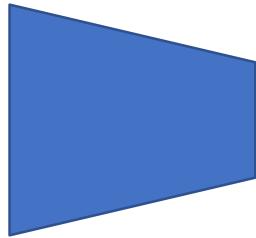
76.53

Lots of other
patterns in
full results



Improving ViT: Distillation

Step 1: Train a **teacher model** on images and ground-truth labels

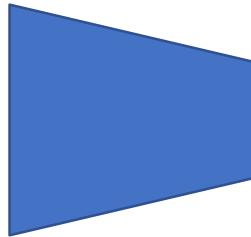
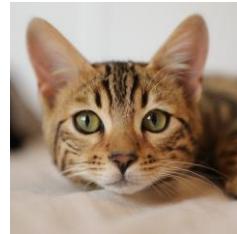


$$\begin{aligned} P(\text{cat}) &= 0.9 \\ P(\text{dog}) &= 0.1 \end{aligned}$$

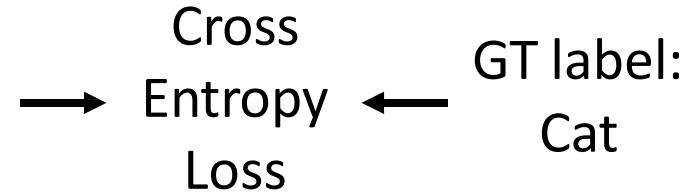
Cross Entropy Loss ← GT label: Cat

Improving ViT: Distillation

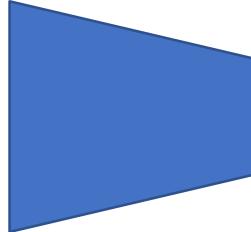
Step 1: Train a **teacher model** on images and ground-truth labels



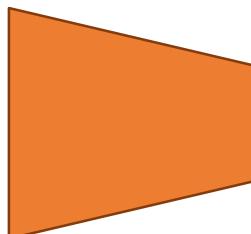
$$\begin{aligned} P(\text{cat}) &= 0.9 \\ P(\text{dog}) &= 0.1 \end{aligned}$$



Step 2: Train a **student model** to match predictions from the **teacher**



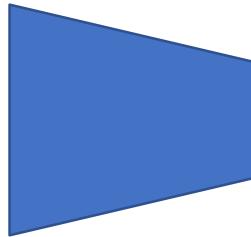
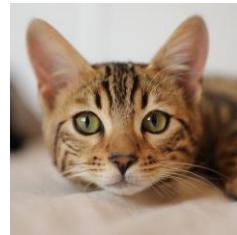
$$\begin{aligned} P(\text{cat}) &= 0.1 \\ P(\text{dog}) &= 0.9 \end{aligned}$$



$$\begin{aligned} P(\text{cat}) &= 0.2 \\ P(\text{dog}) &= 0.8 \end{aligned}$$

Improving ViT: Distillation

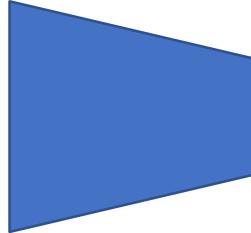
Step 1: Train a **teacher model** on images and ground-truth labels



$$\begin{aligned} P(\text{cat}) &= 0.9 \\ P(\text{dog}) &= 0.1 \end{aligned}$$

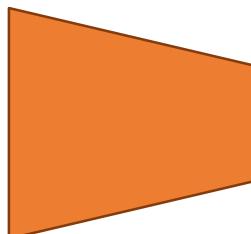
Cross Entropy Loss ← GT label: Cat

Step 2: Train a **student model** to match predictions from the **teacher** (sometimes also to match GT labels)



$$\begin{aligned} P(\text{cat}) &= 0.1 \\ P(\text{dog}) &= 0.9 \end{aligned}$$

KL Divergence Loss

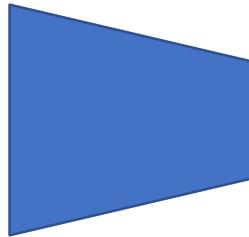
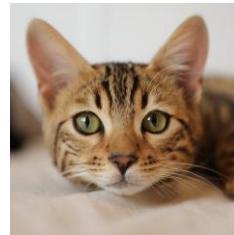


$$\begin{aligned} P(\text{cat}) &= 0.2 \\ P(\text{dog}) &= 0.8 \end{aligned}$$

Cross Entropy Loss ← GT label: Dog

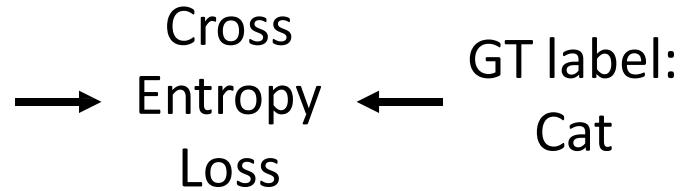
Improving ViT: Distillation

Step 1: Train a **teacher model** on images and ground-truth labels

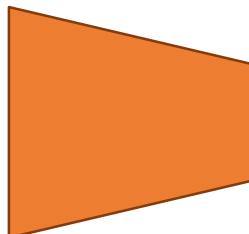
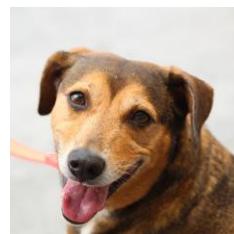
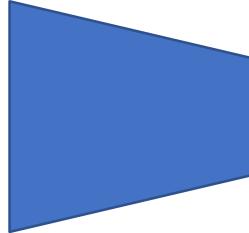


Often works better than training student from scratch (especially if teacher is bigger than student)

$$\begin{aligned} P(\text{cat}) &= 0.9 \\ P(\text{dog}) &= 0.1 \end{aligned}$$



Step 2: Train a **student model** to match predictions from the **teacher** (sometimes also to match GT labels)



$$\begin{aligned} P(\text{cat}) &= 0.1 \\ P(\text{dog}) &= 0.9 \end{aligned}$$

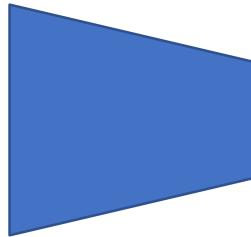


$$\begin{aligned} P(\text{cat}) &= 0.2 \\ P(\text{dog}) &= 0.8 \end{aligned}$$



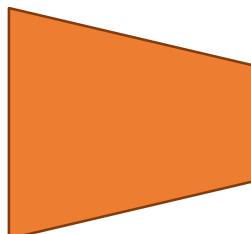
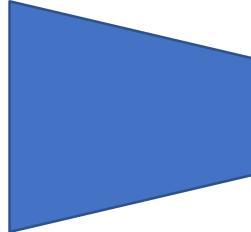
Improving ViT: Distillation

Step 1: Train a **teacher model** on images and ground-truth labels



Can also train student on **unlabeled** data! (Semi-supervised learning)

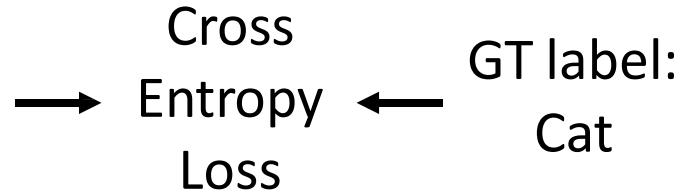
Step 2: Train a **student model** to match predictions from the **teacher** (sometimes also to match GT labels)



$$\begin{aligned} P(\text{cat}) &= 0.9 \\ P(\text{dog}) &= 0.1 \end{aligned}$$

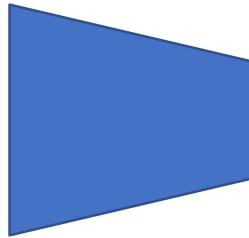
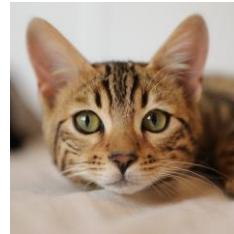
$$\begin{aligned} P(\text{cat}) &= 0.1 \\ P(\text{dog}) &= 0.9 \end{aligned}$$

$$\begin{aligned} P(\text{cat}) &= 0.2 \\ P(\text{dog}) &= 0.8 \end{aligned}$$



Improving ViT: Distillation

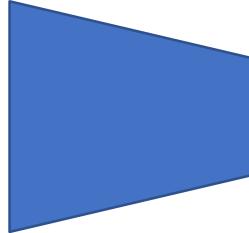
Step 1: Train a **teacher CNN** on ImageNet



$$\begin{aligned} P(\text{cat}) &= 0.9 \\ P(\text{dog}) &= 0.1 \end{aligned}$$

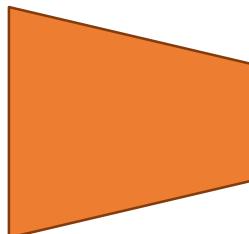
Cross Entropy Loss ← GT label: Cat

Step 2: Train a **student ViT** to match ImageNet predictions from the **teacher CNN** (and match GT labels)



$$\begin{aligned} P(\text{cat}) &= 0.1 \\ P(\text{dog}) &= 0.9 \end{aligned}$$

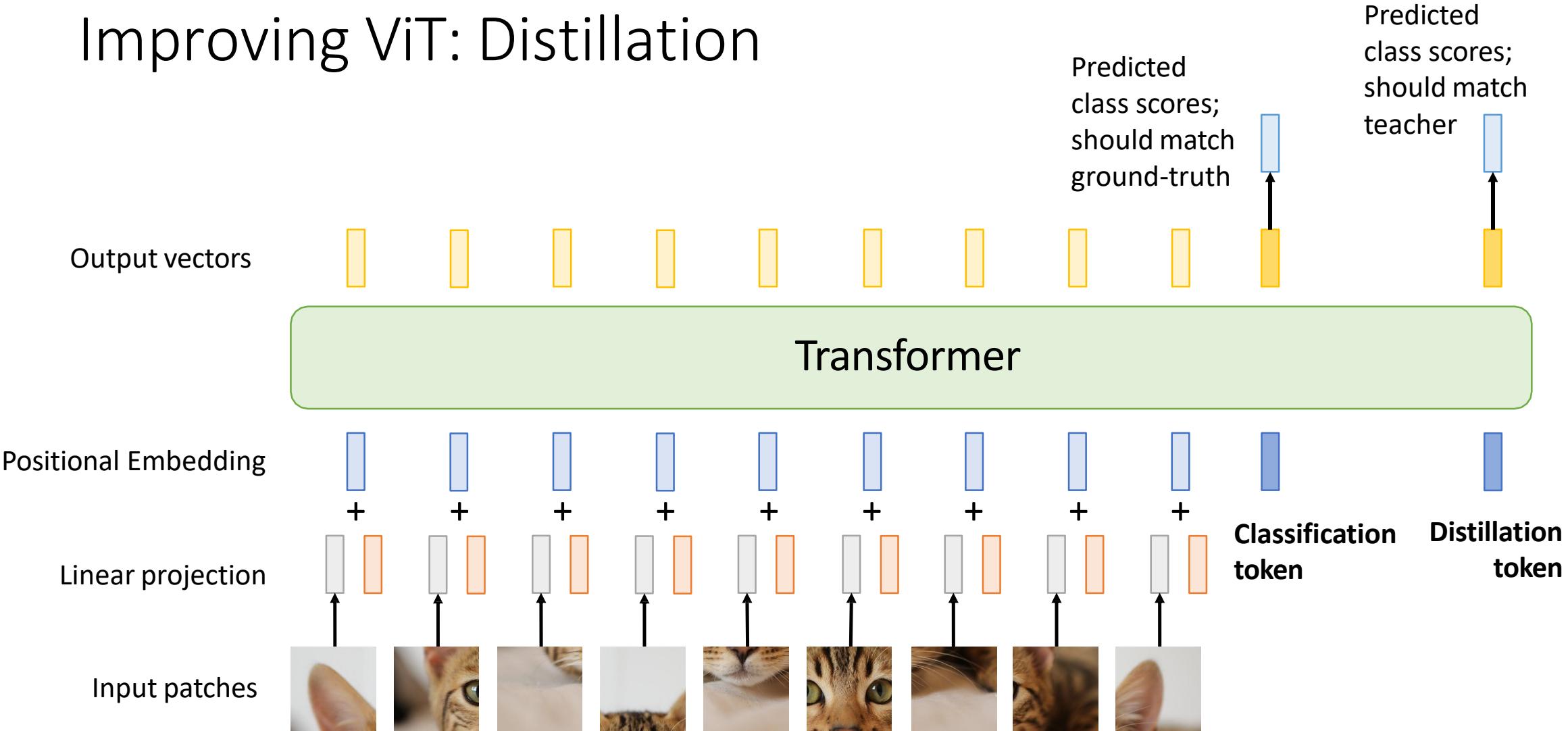
KL Divergence Loss



$$\begin{aligned} P(\text{cat}) &= 0.2 \\ P(\text{dog}) &= 0.8 \end{aligned}$$

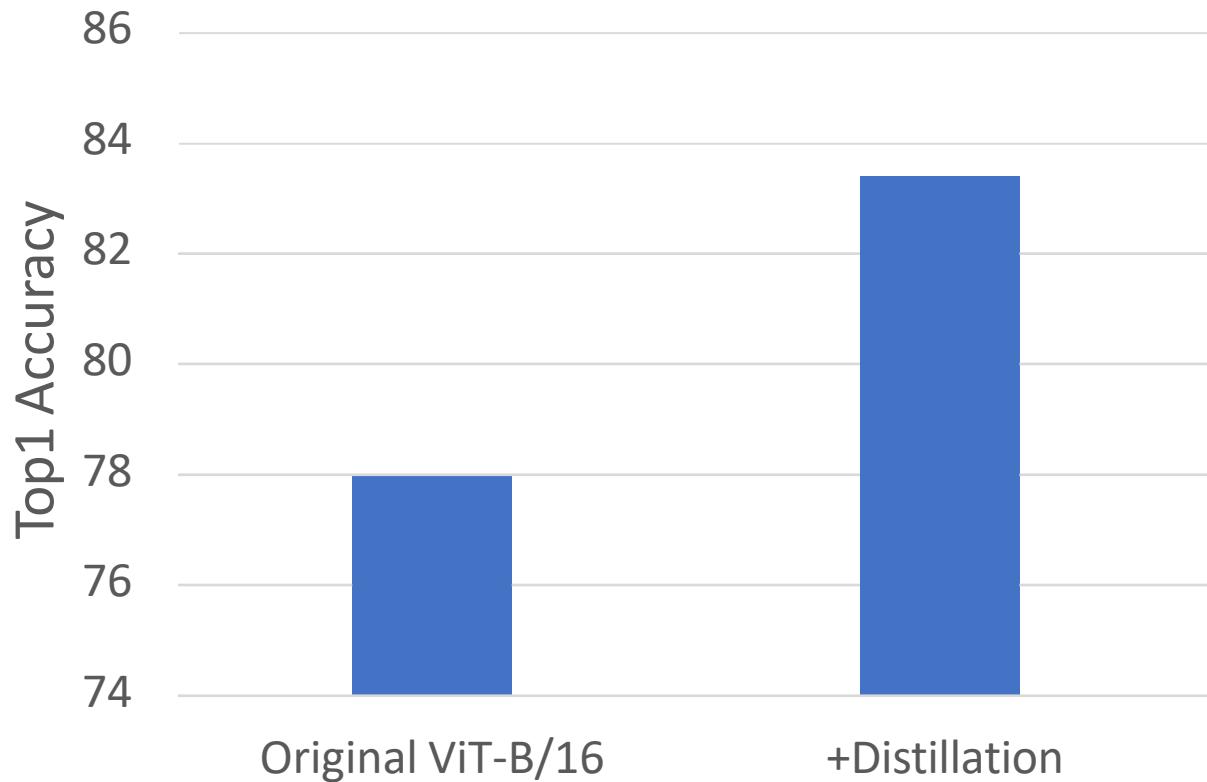
Cross Entropy Loss ← GT label: Dog

Improving ViT: Distillation



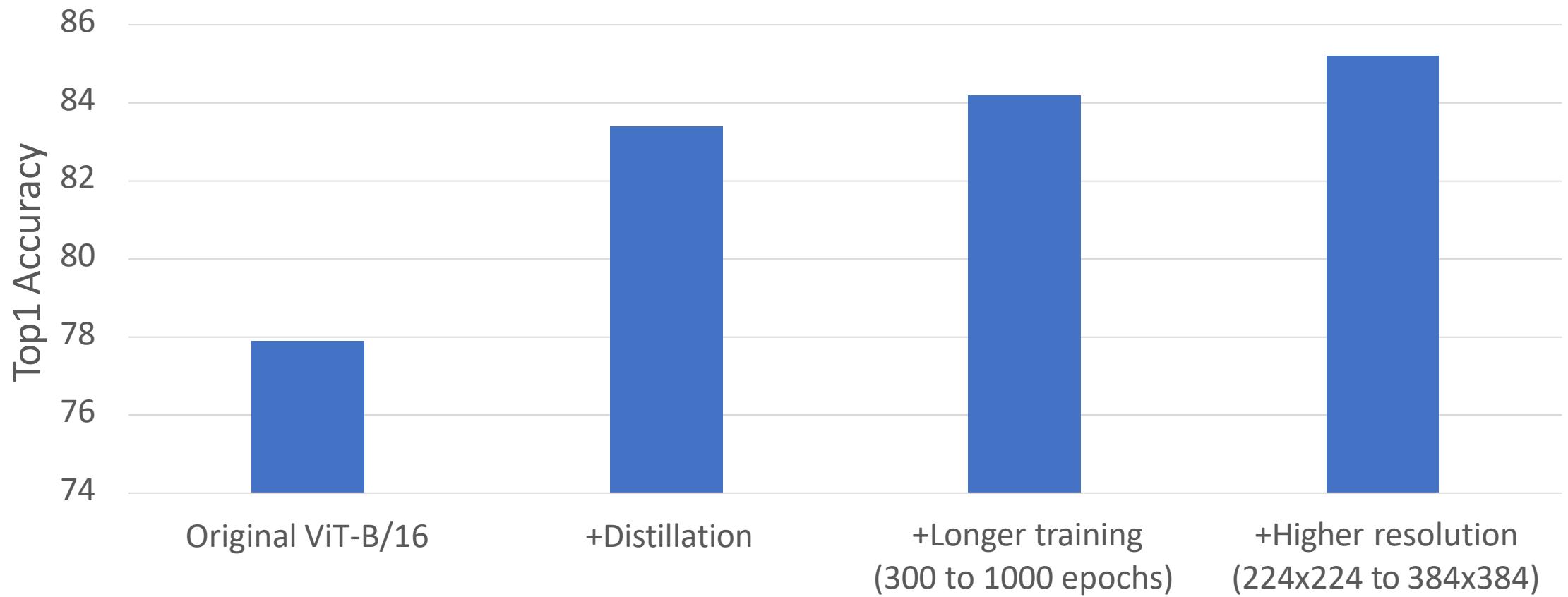
Improving ViT: Distillation

ViT-B/16 on ImageNet

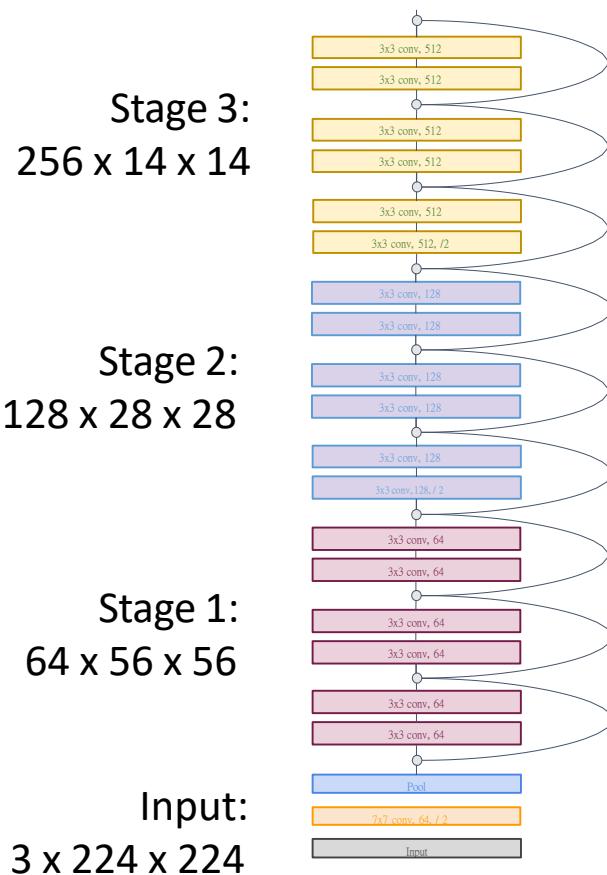


Improving ViT: Distillation

ViT-B/16 on ImageNet



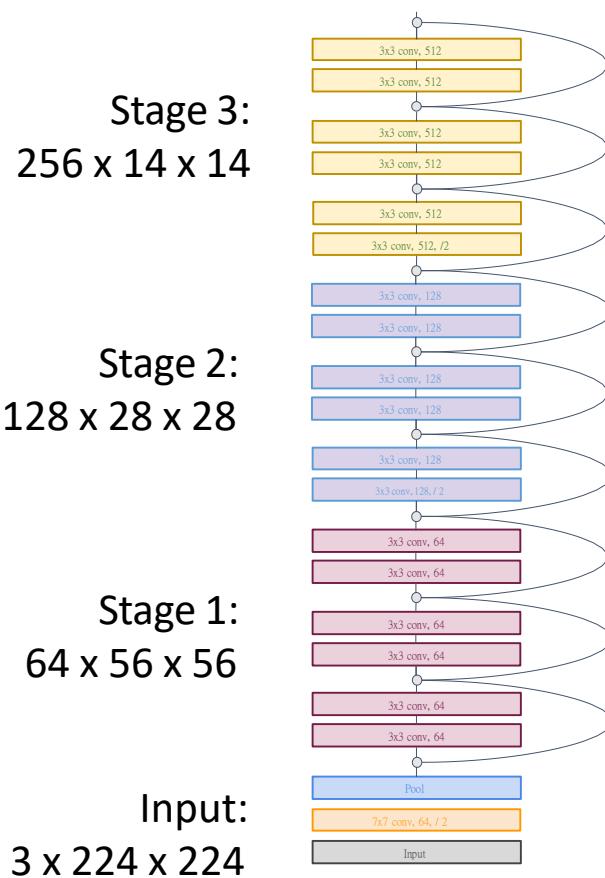
ViT vs CNN



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

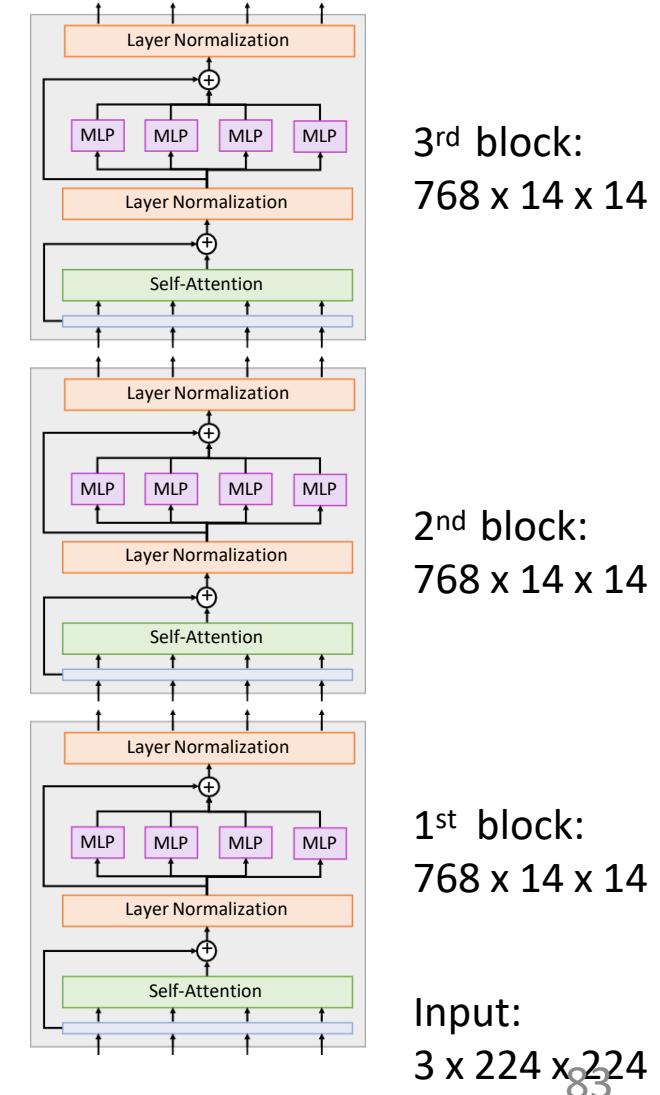
ViT vs CNN



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

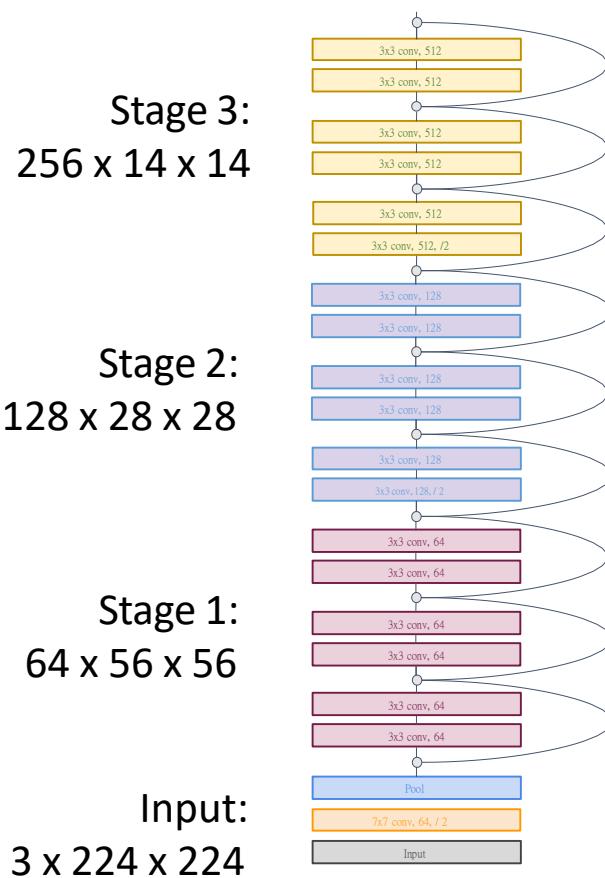
Useful since objects in images can occur at various scales

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



Input:
 $3 \times 224 \times 224$

ViT vs CNN

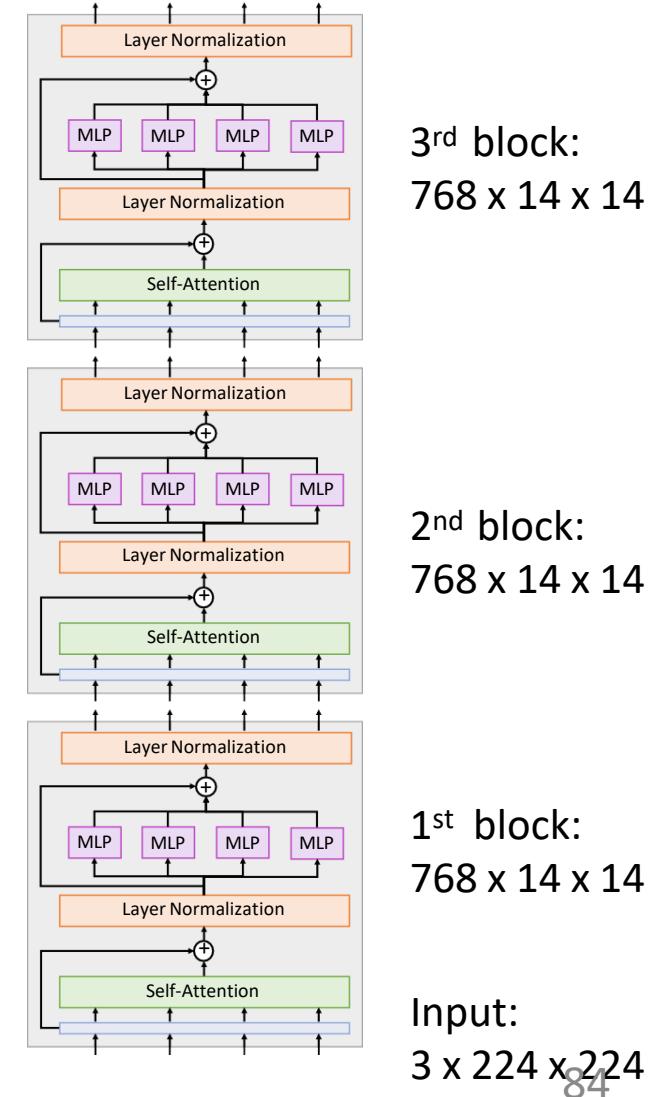


In most CNNs (including ResNets), **decrease** resolution and **increase** channels as you go deeper in the network
(Hierarchical architecture)

Useful since objects in images can occur at various scales

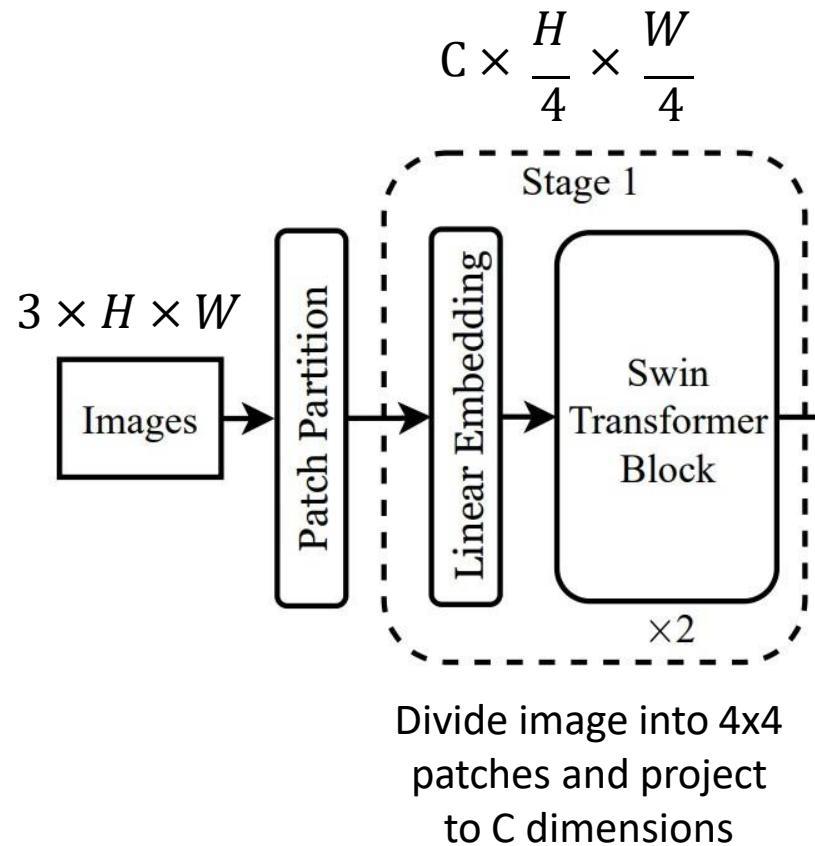
In a ViT, all blocks have same resolution and number of channels
(Isotropic architecture)

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

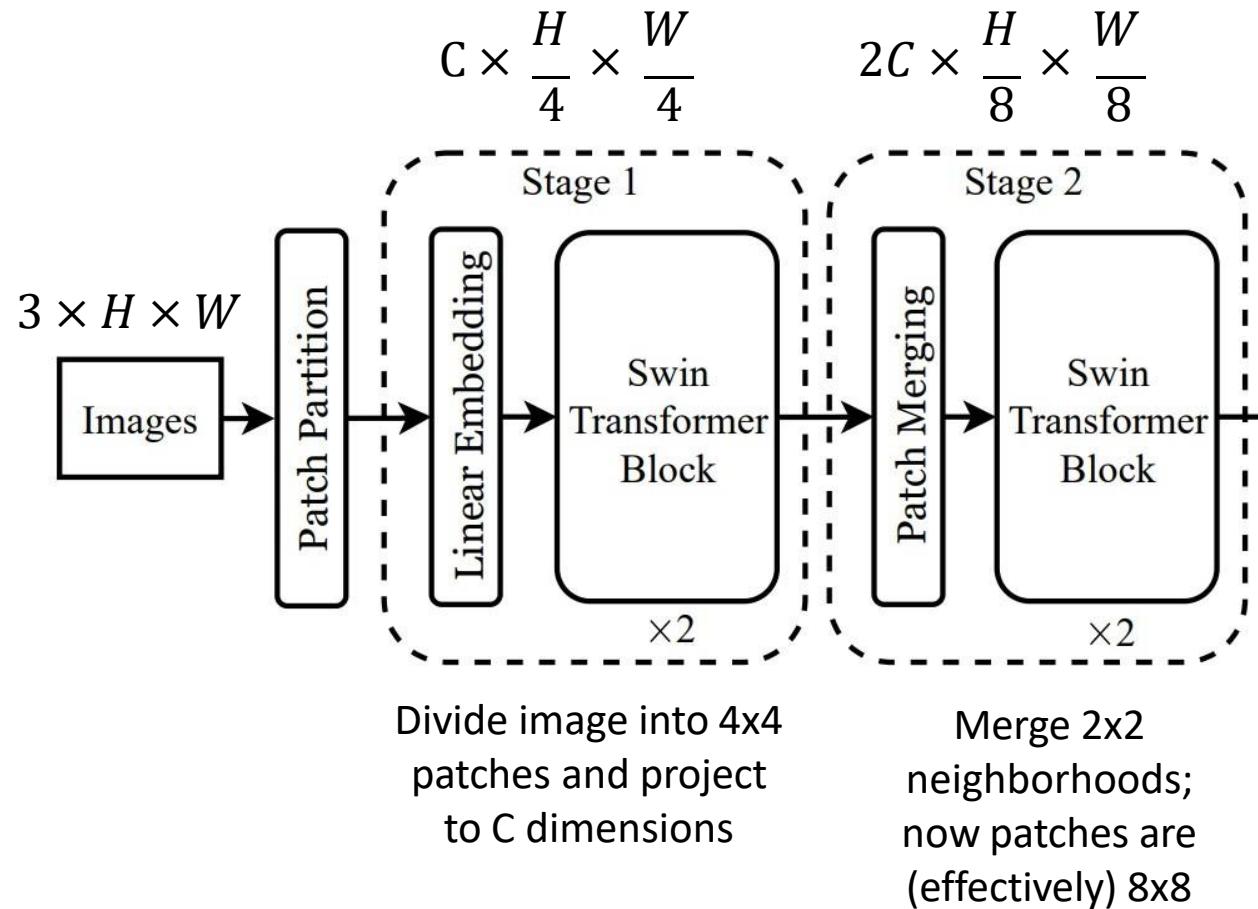


Input:
 $3 \times 224 \times 224$
84

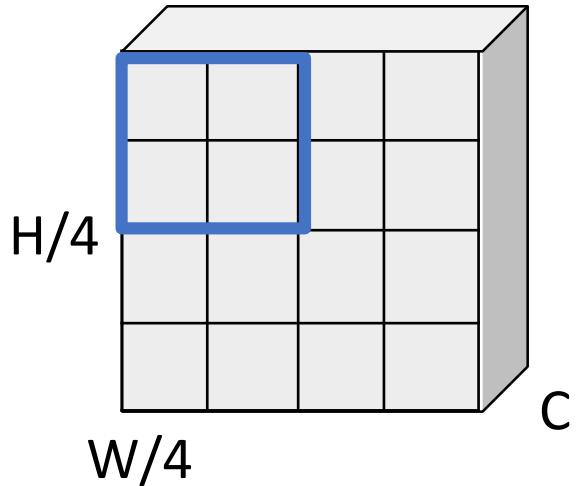
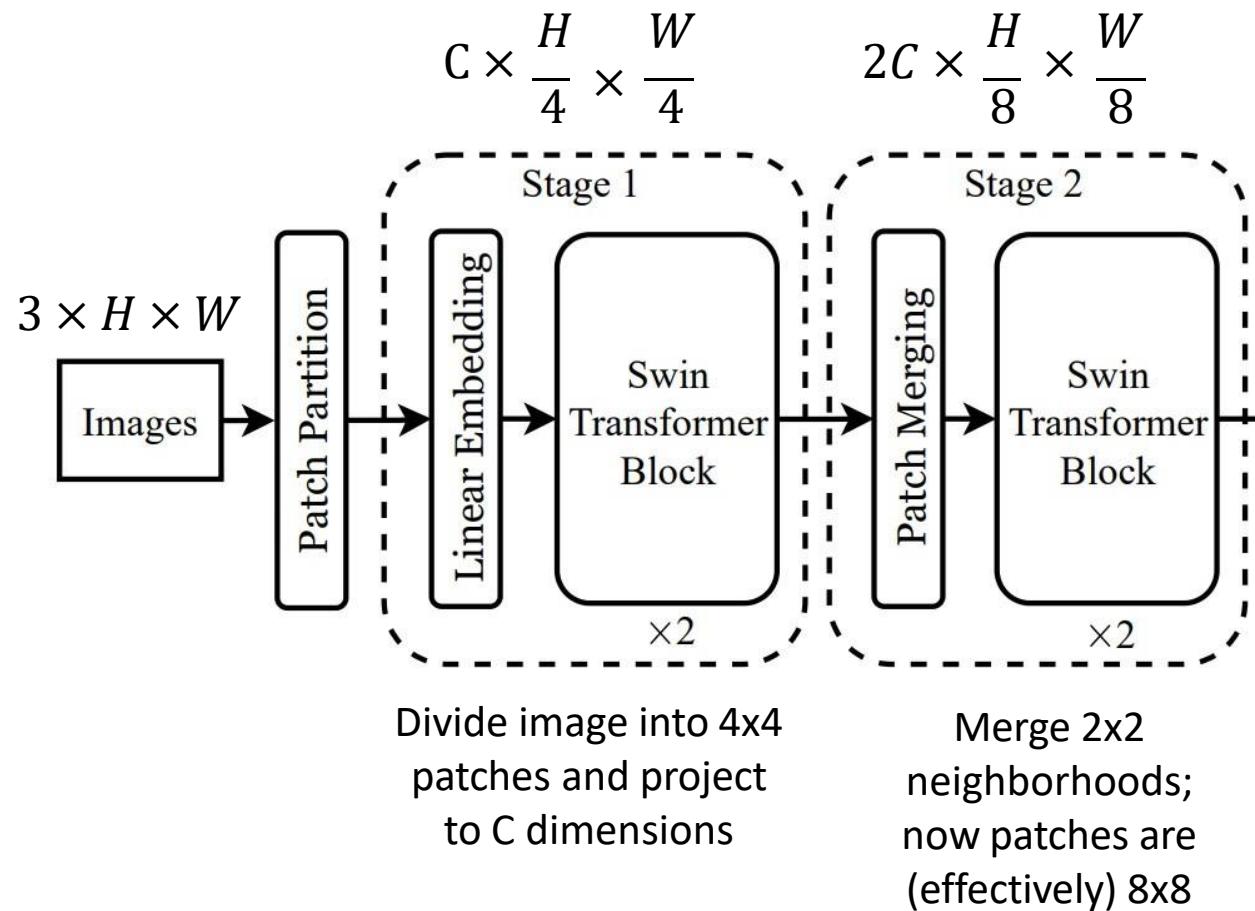
Hierarchical ViT: Swin Transformer



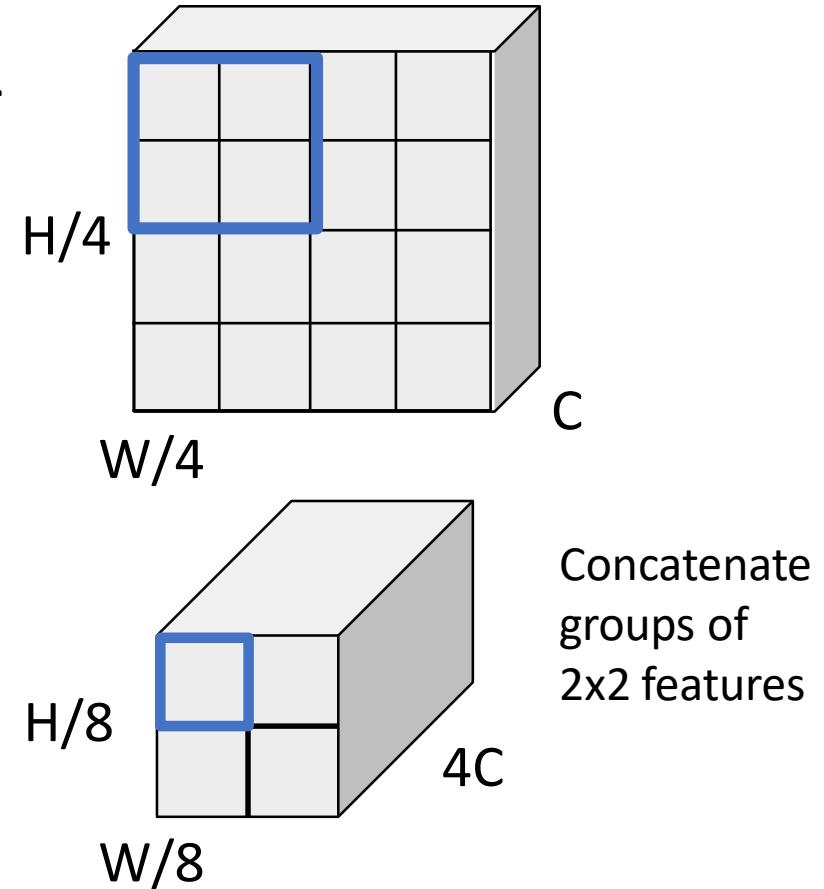
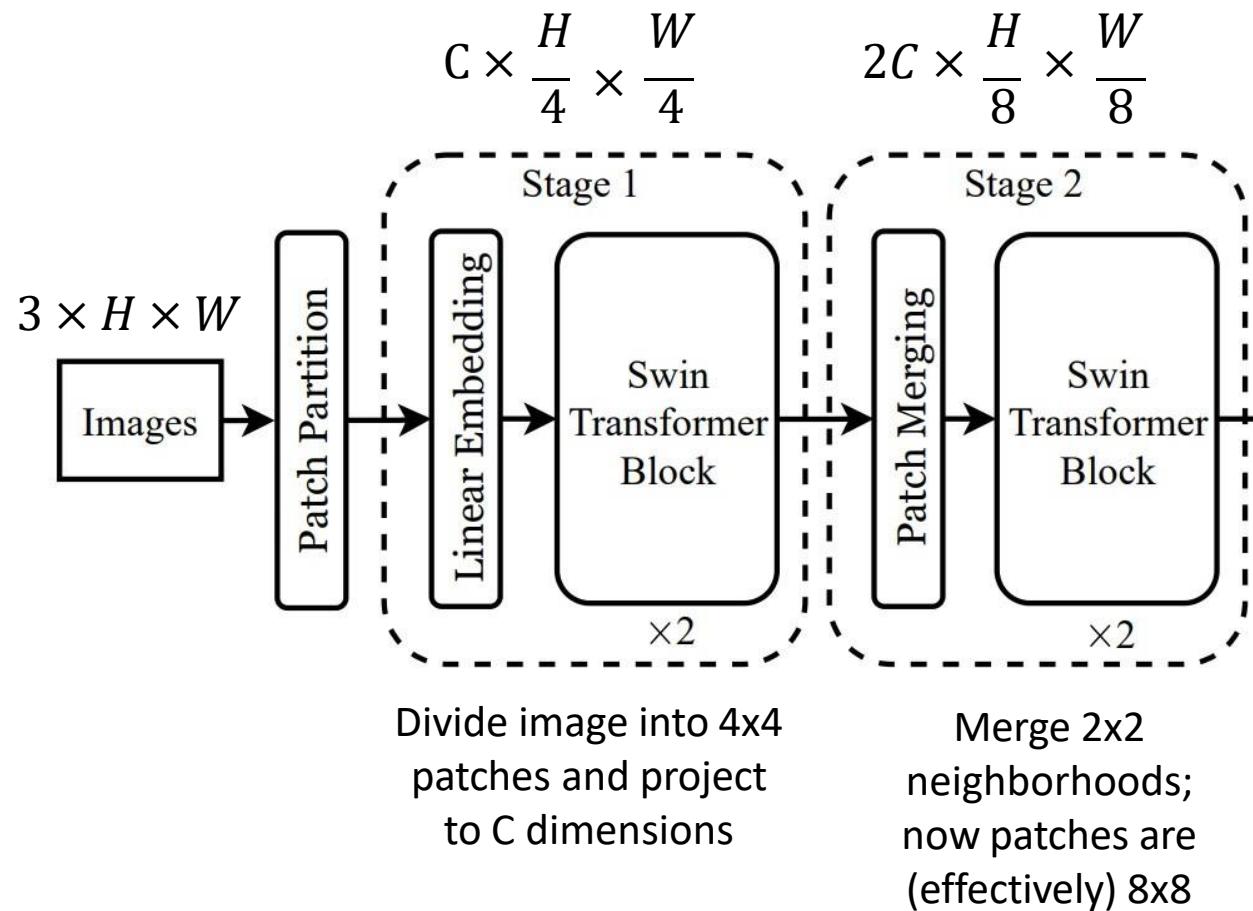
Hierarchical ViT: Swin Transformer



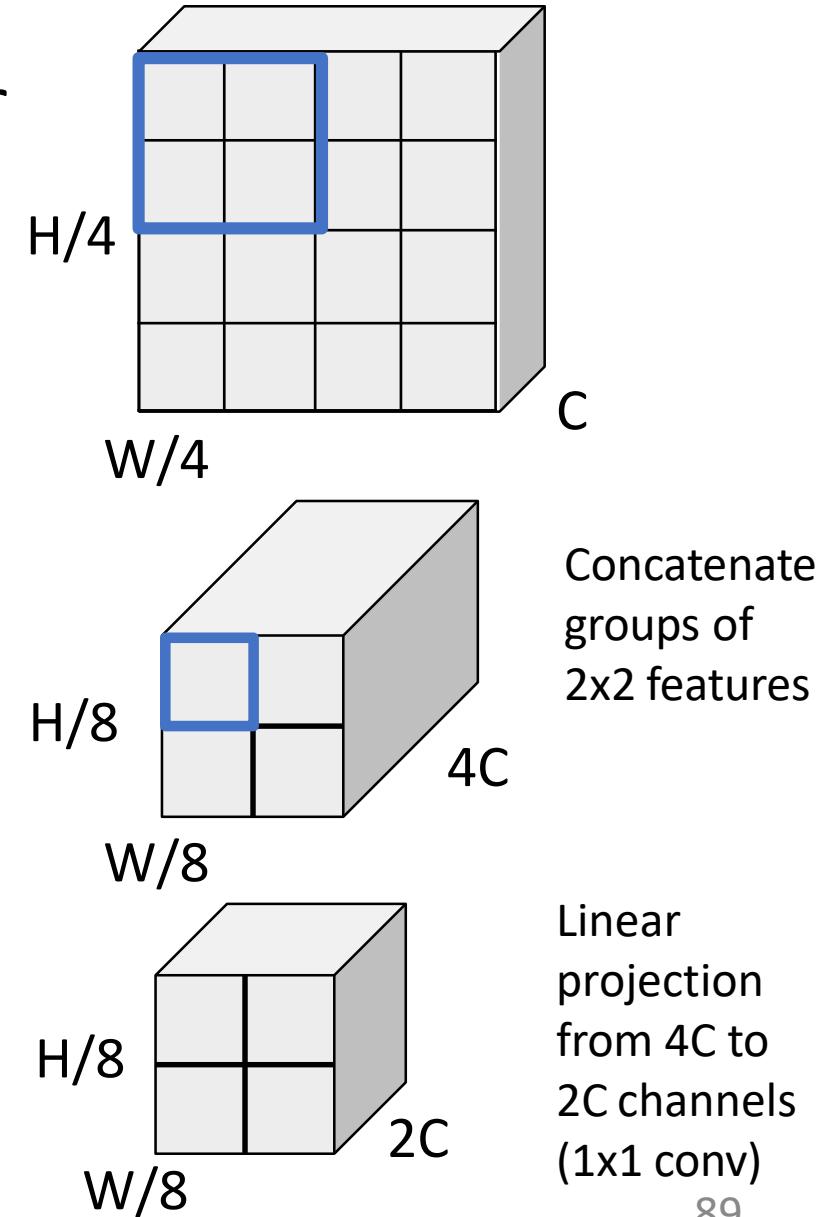
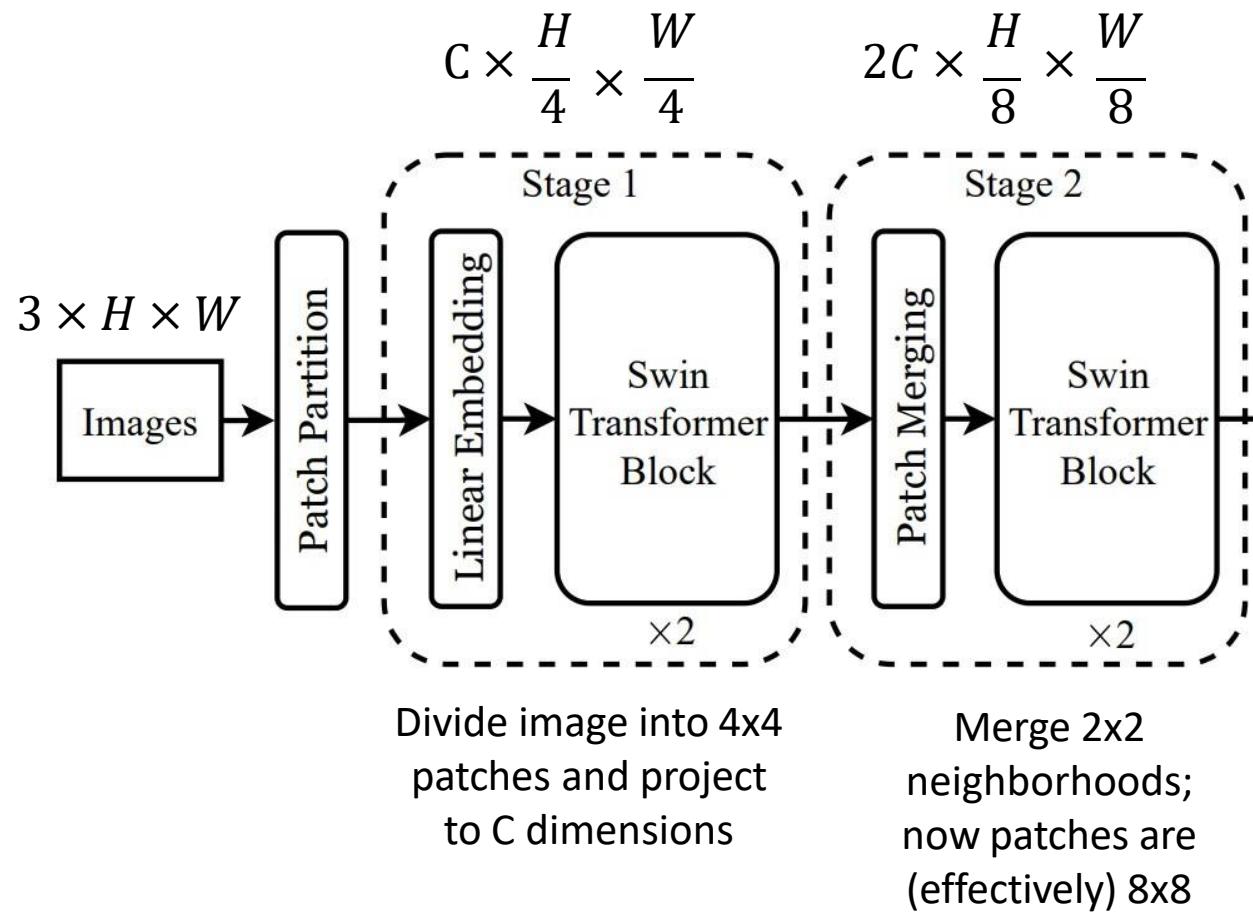
Hierarchical ViT: Swin Transformer



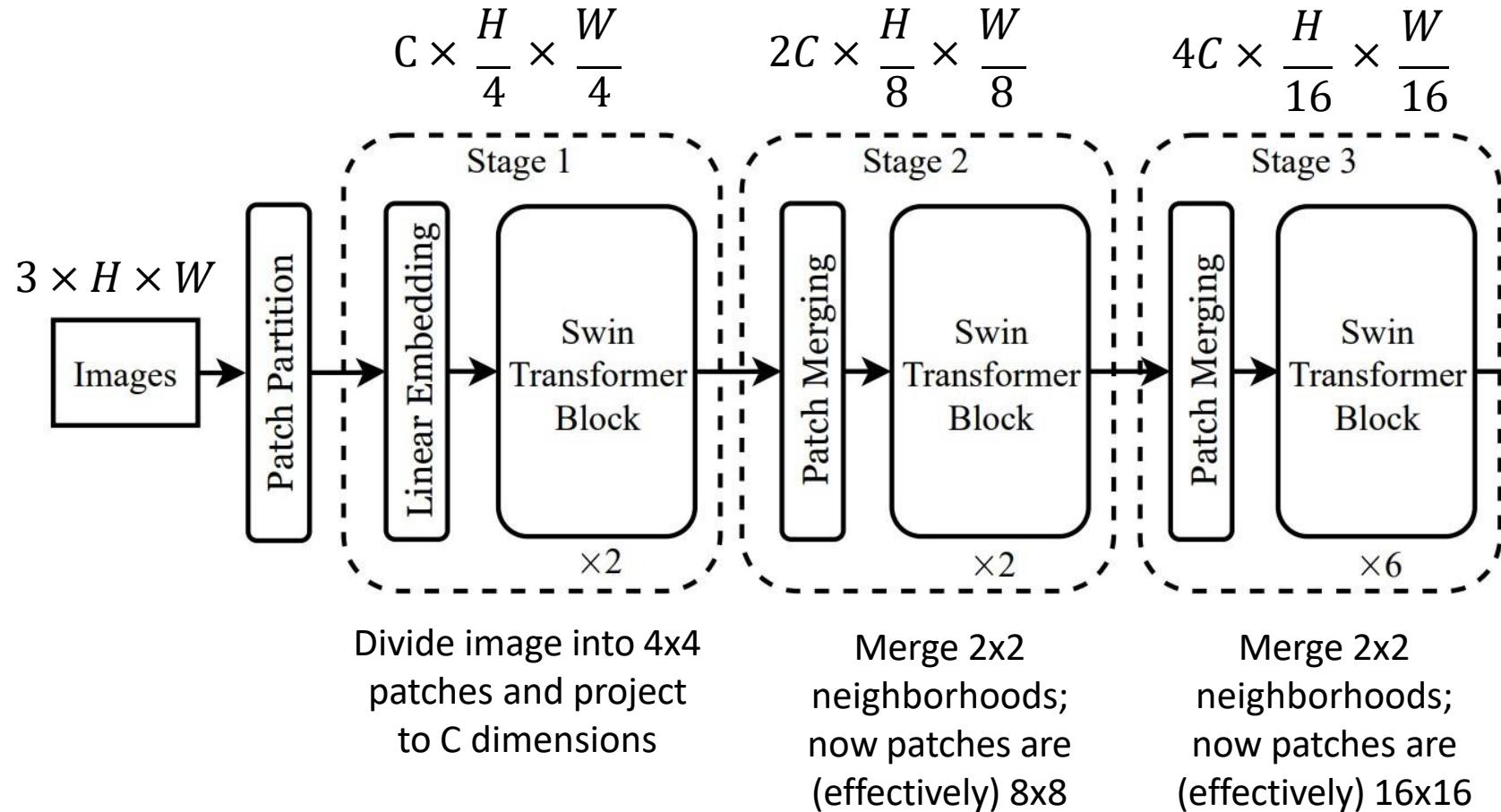
Hierarchical ViT: Swin Transformer



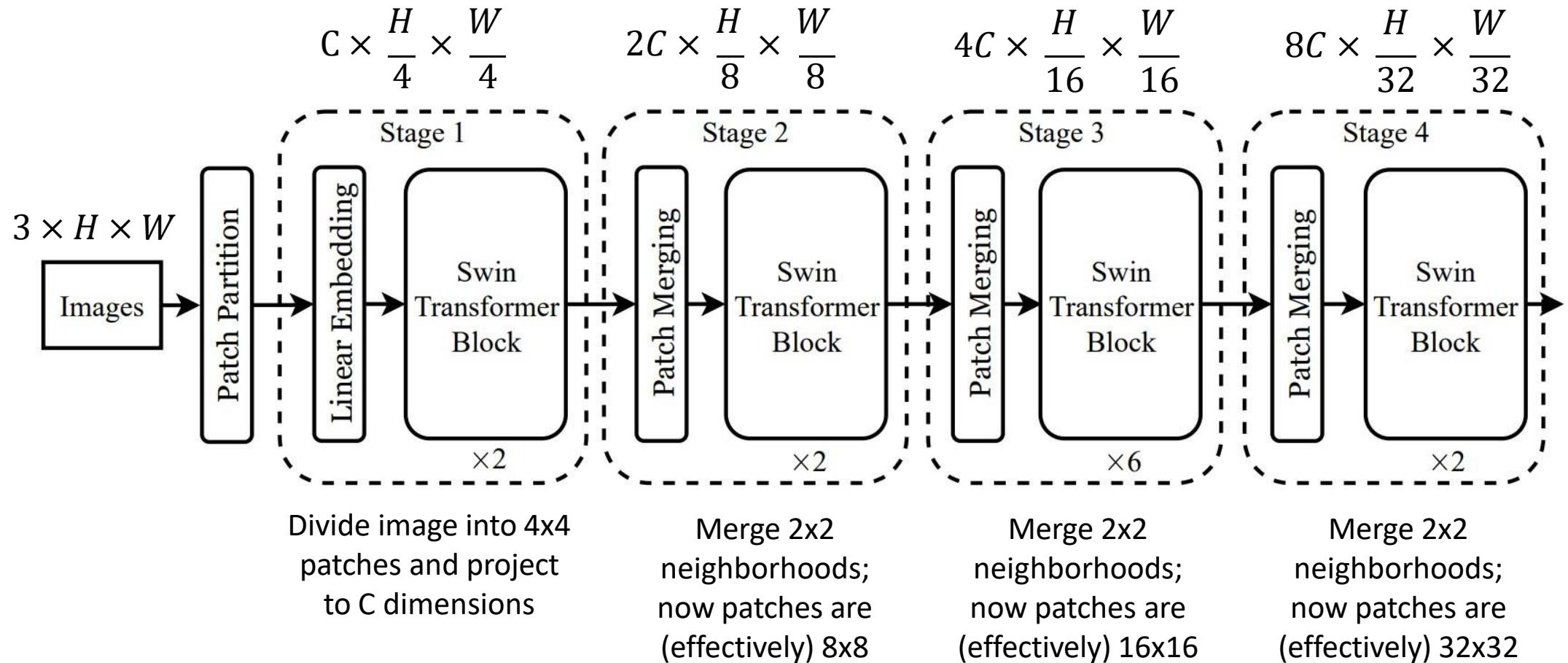
Hierarchical ViT: Swin Transformer



Hierarchical ViT: Swin Transformer



Hierarchical ViT: Swin Transformer



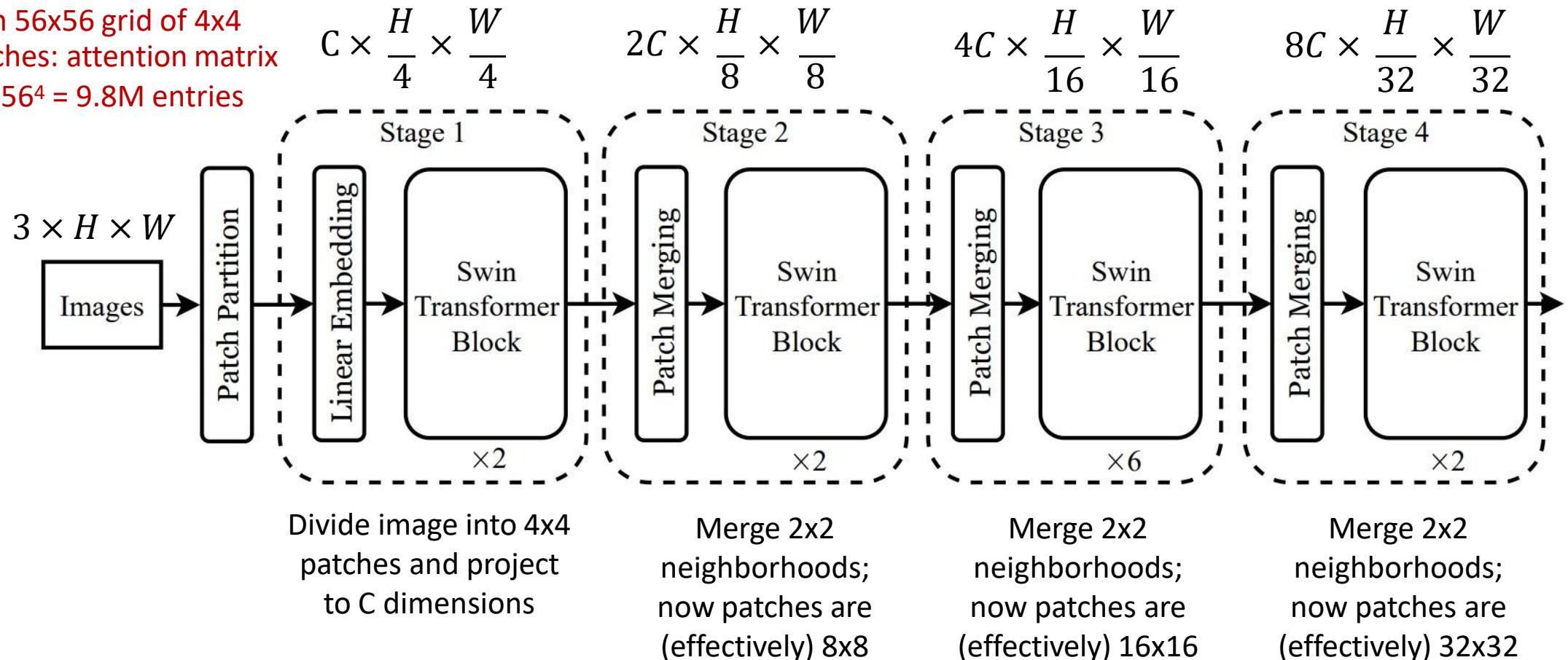
Hierarchical ViT: Swin Transformer

Problem: 224x224 image

with 56x56 grid of 4x4

patches: attention matrix

has $56^4 = 9.8M$ entries



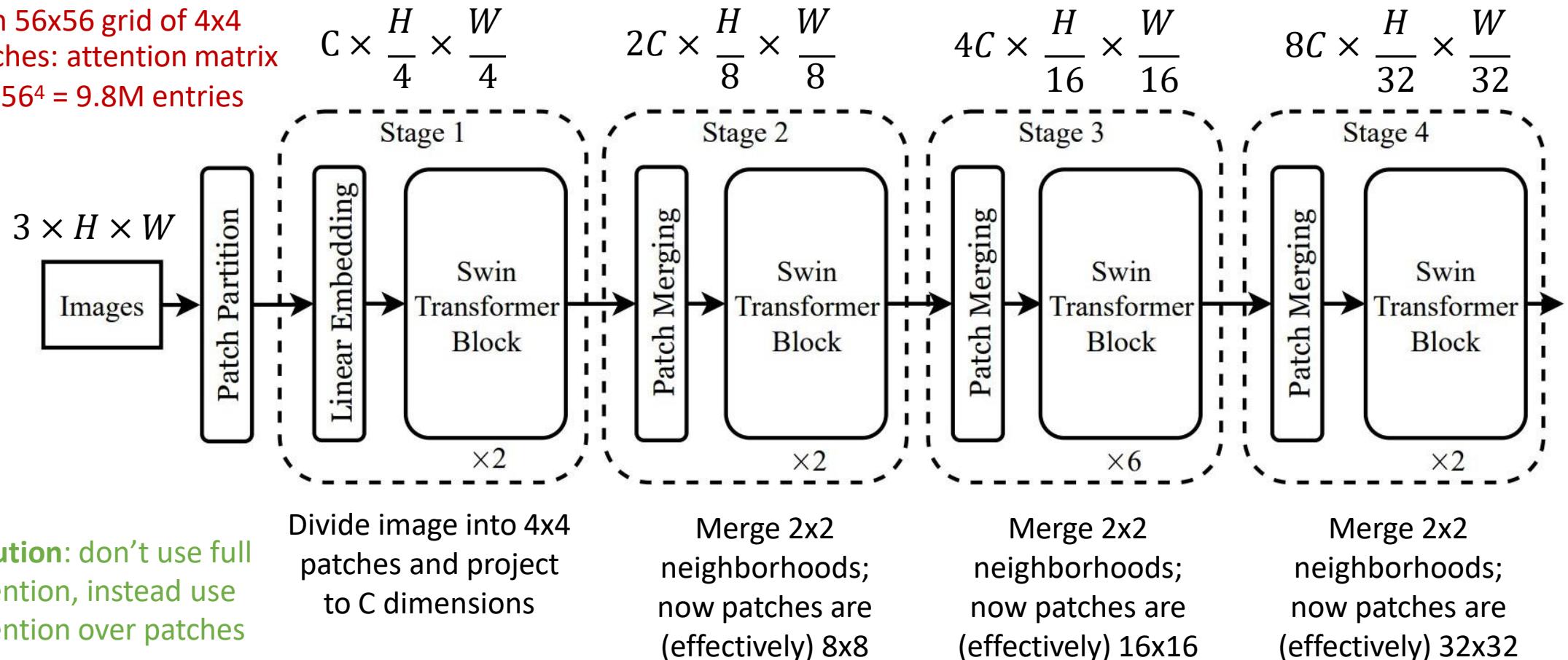
Hierarchical ViT: Swin Transformer

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

Swin Transformer: Window Attention

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

Swin Transformer: Window Attention



With $H \times W$ 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 $M \times M$ tokens (here $M=4$); only compute attention within each window

Swin Transformer: Window Attention



With $H \times W$ 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 $M \times M$ 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



Swin Transformer: Shifted Window Attention

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



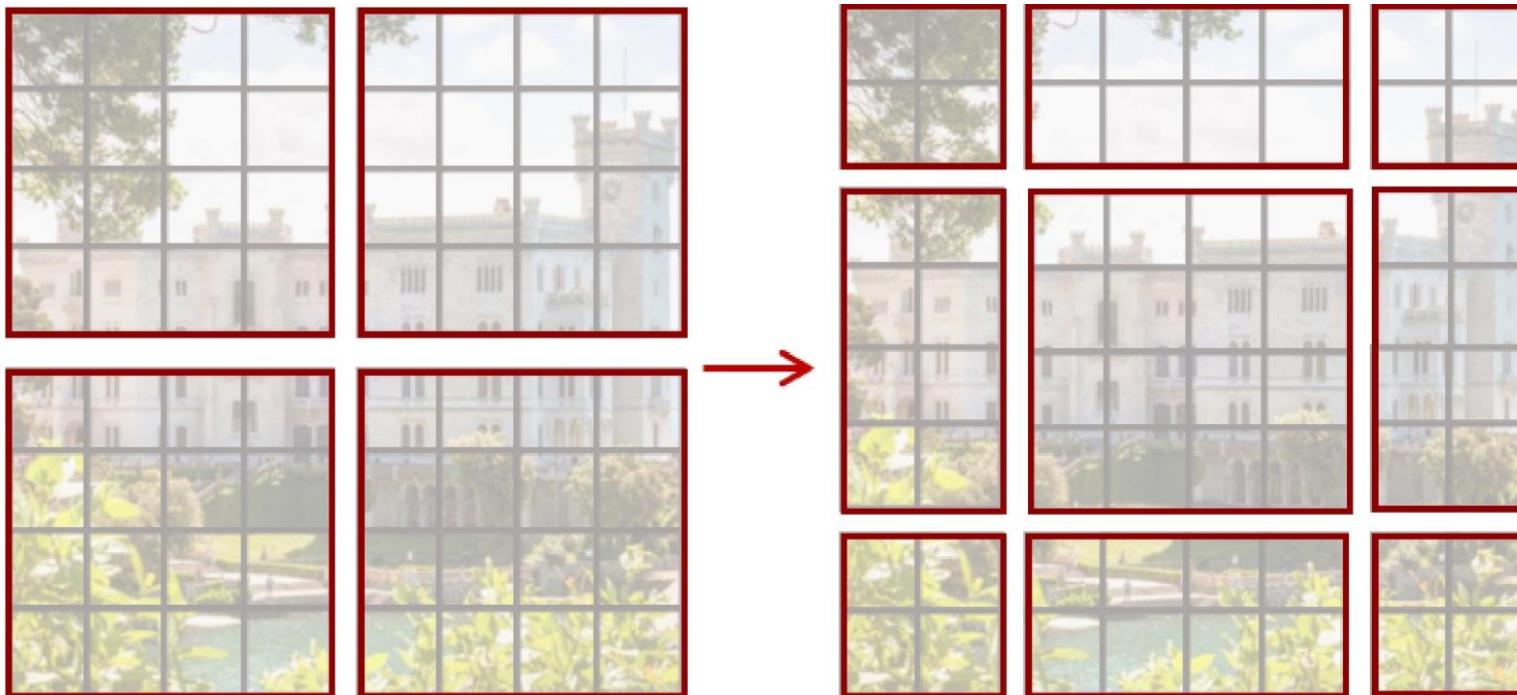
Ugly detail:
Non-square
windows at
edges and
corners

Swin Transformer: Shifted Window Attention

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

Detail: Relative Positional Bias

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



Block L: Normal windows

Block L+1: Shifted Windows

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:

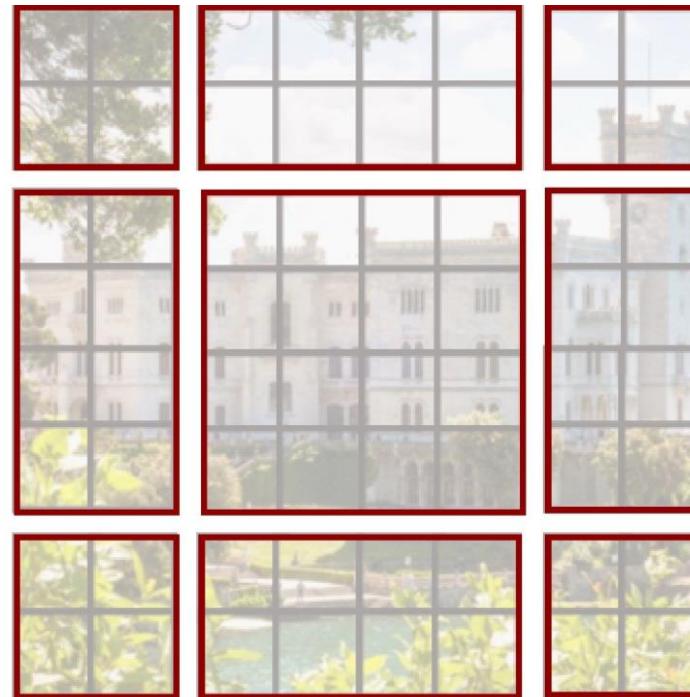
Standard Attention:

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

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

Swin Transformer: Shifted Window Attention

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



Detail: Relative Positional Bias

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

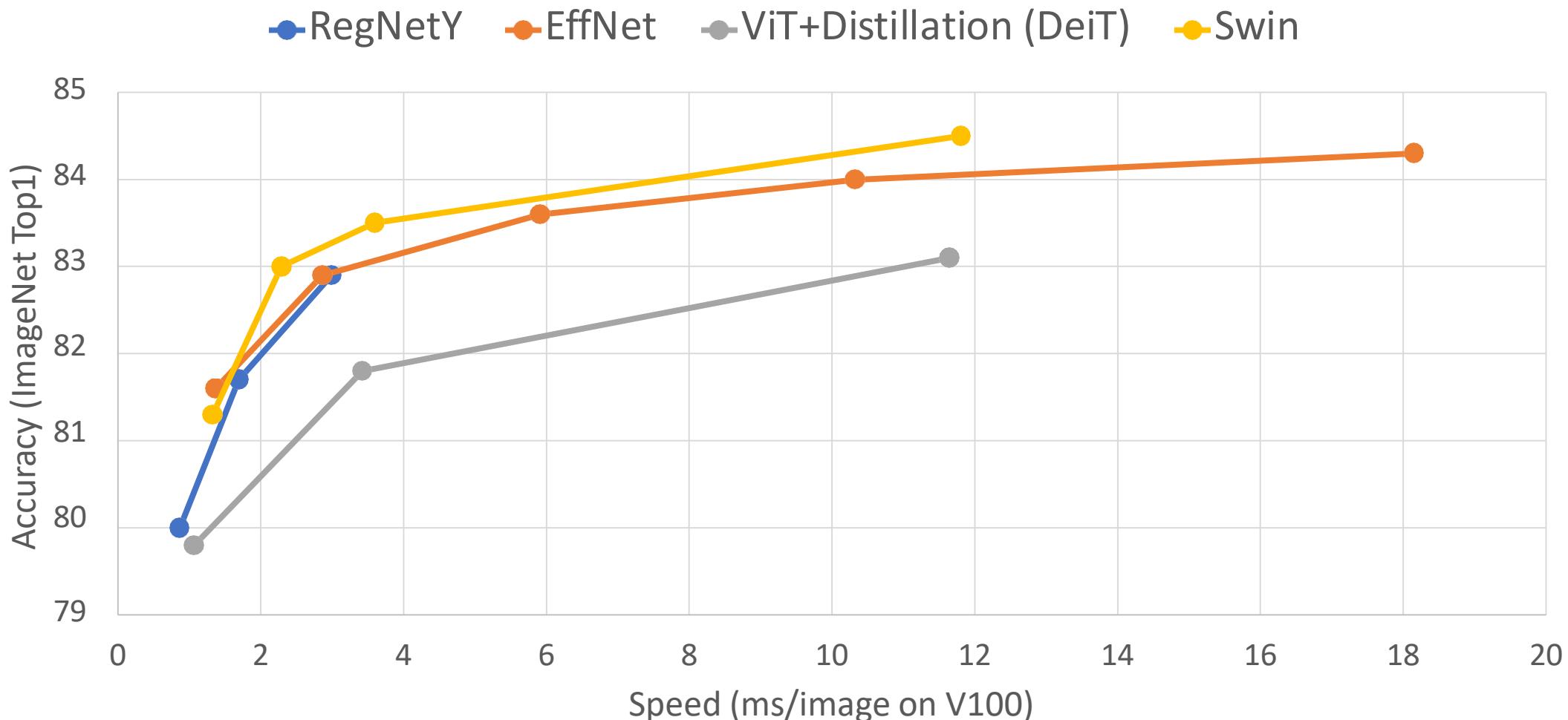
Swin does not use positional embeddings, instead encodes *relative position* between patches when computing attention:

Attention with relative bias:

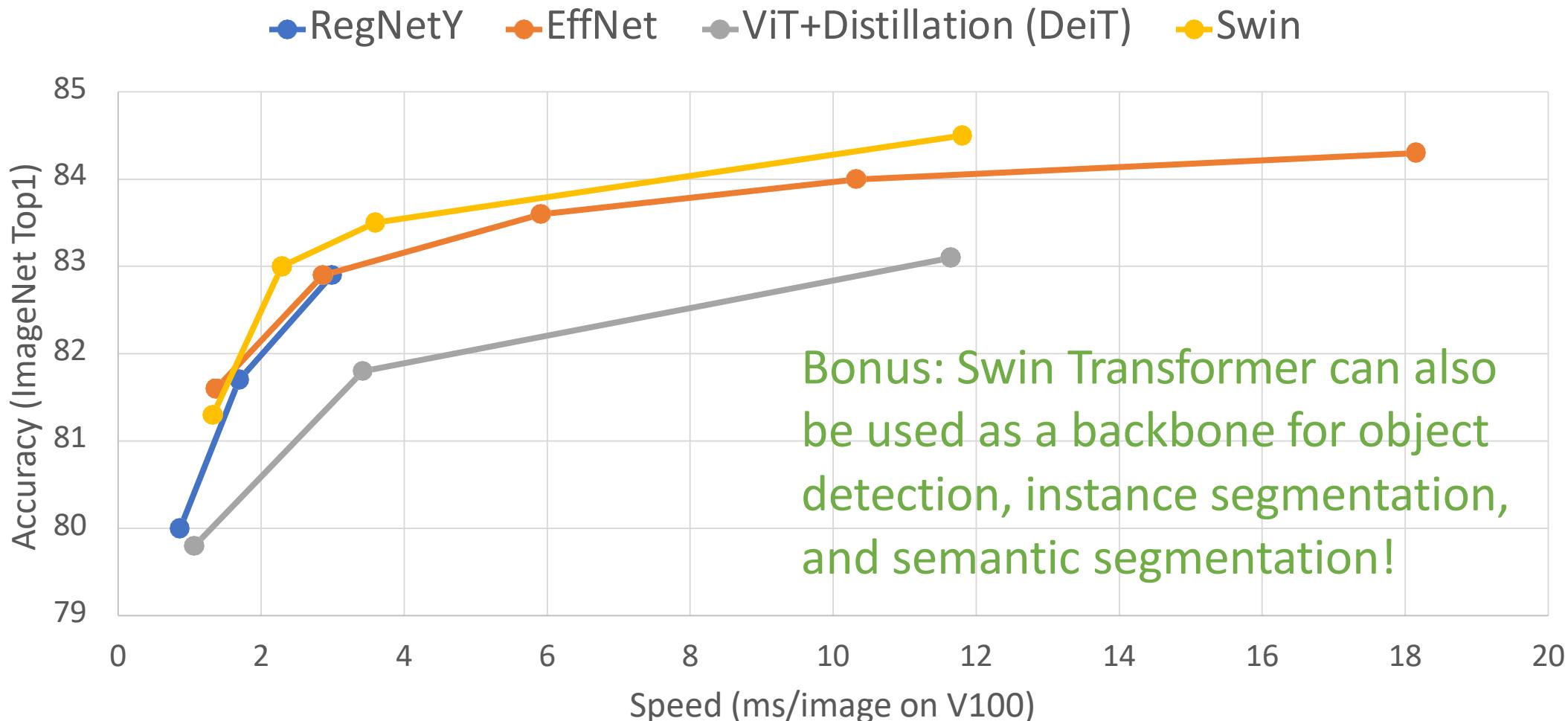
$$A = \text{Softmax} \left(\frac{QK^T}{\sqrt{D}} + B \right) V$$

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

Swin Transformer: Speed vs Accuracy

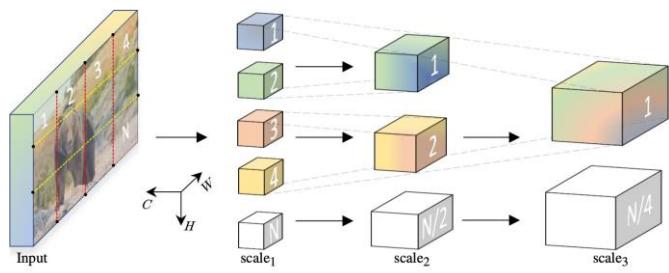


Swin Transformer: Speed vs Accuracy



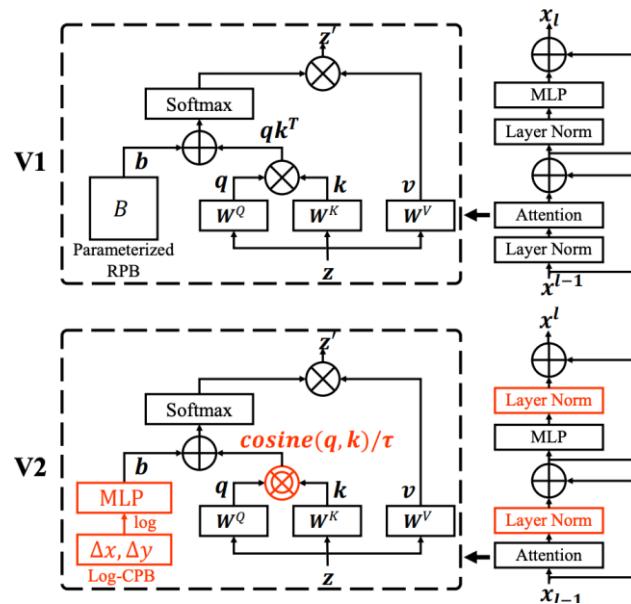
Other Hierarchical Vision Transformers

MViT



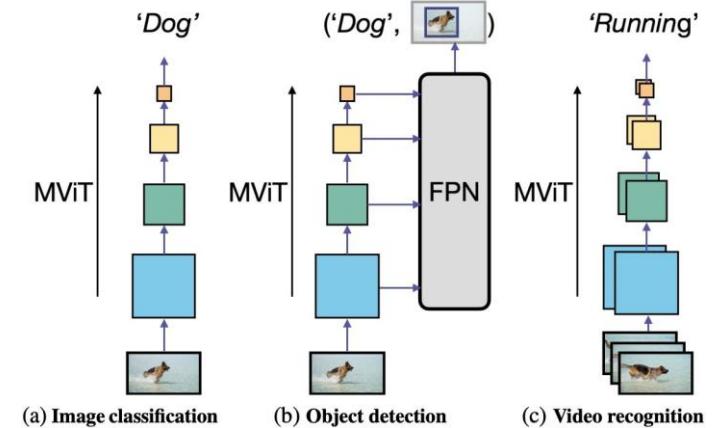
Fan et al, "Multiscale Vision
Transformers", ICCV 2021

Swin-V2



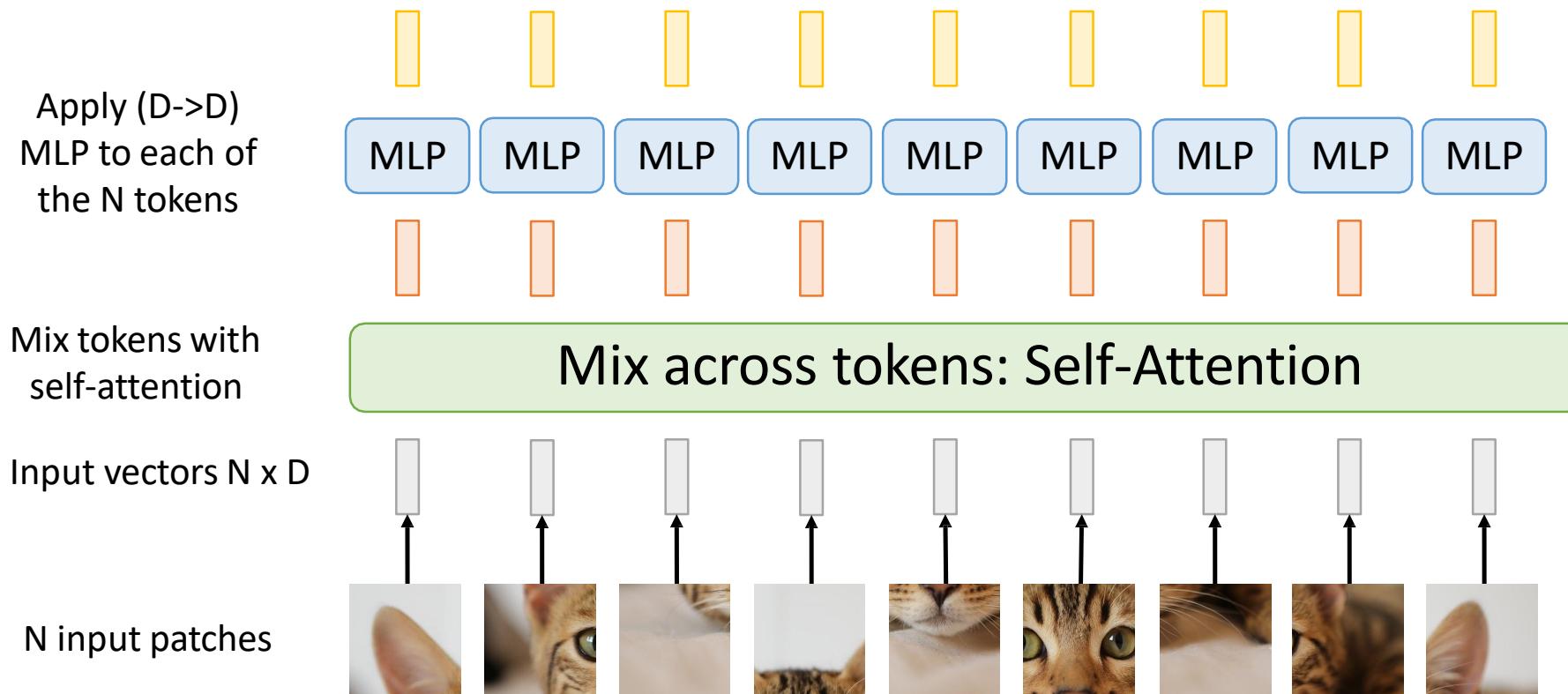
Liu et al, "Swin Transformer V2: Scaling up
Capacity and Resolution" , CVPR 2022

Improved MViT

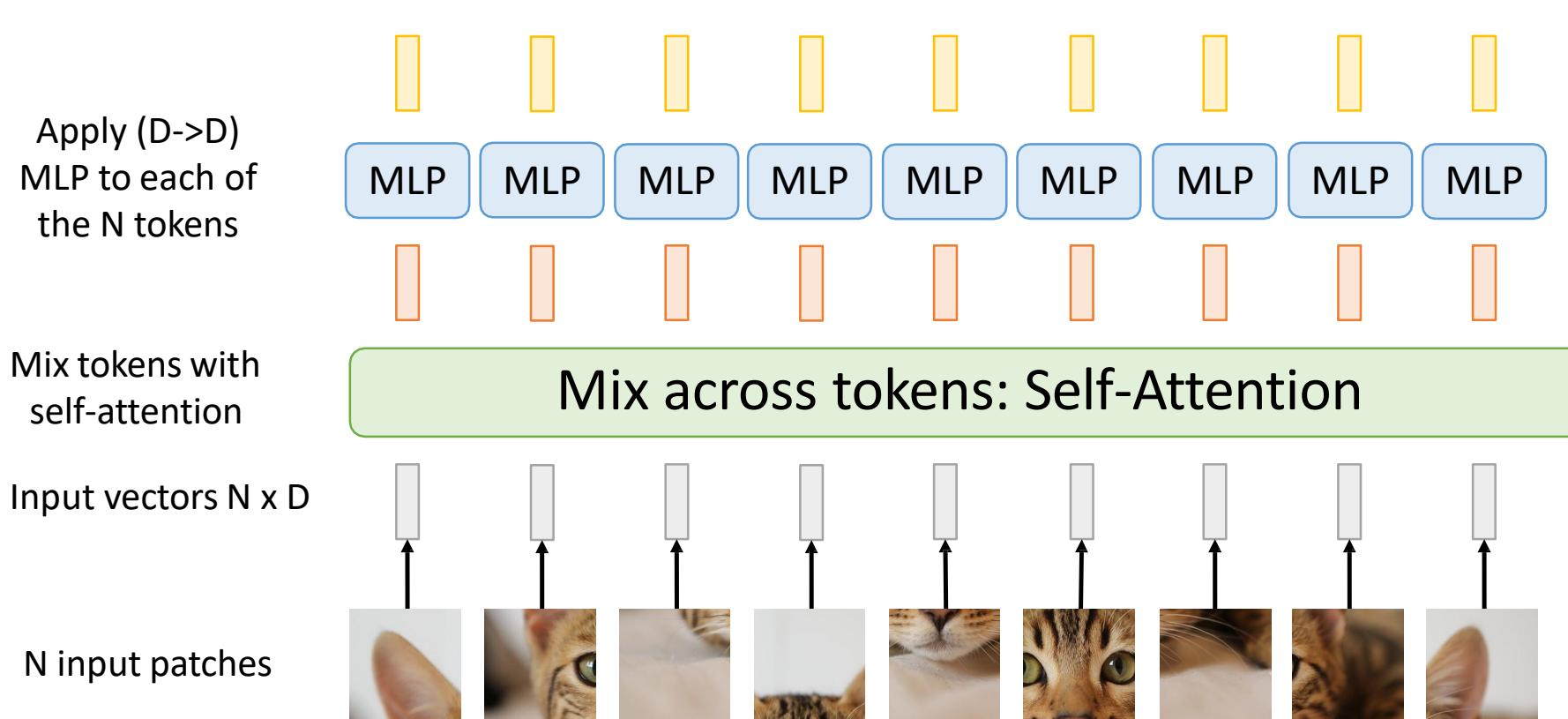


Li et al, "Improved Multiscale Vision Transformers
for Classification and Detection", arXiv 2021

Vision Transformer: Another Look



Vision Transformer: Another Look



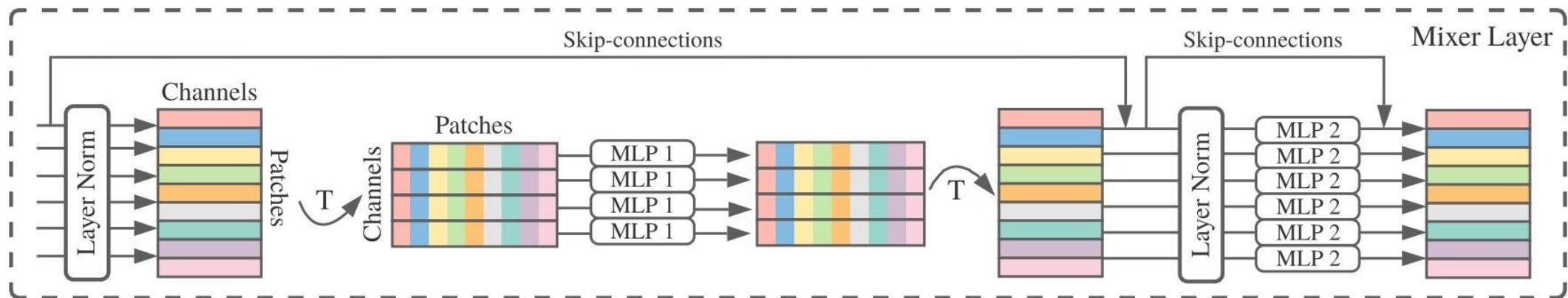
Question: Can we use something simpler than self-attention to mix across tokens?

MLP-Mixer: An All-MLP Architecture



Question: Can we use something simpler than self-attention to mix across tokens?

MLP-Mixer: An All-MLP Architecture



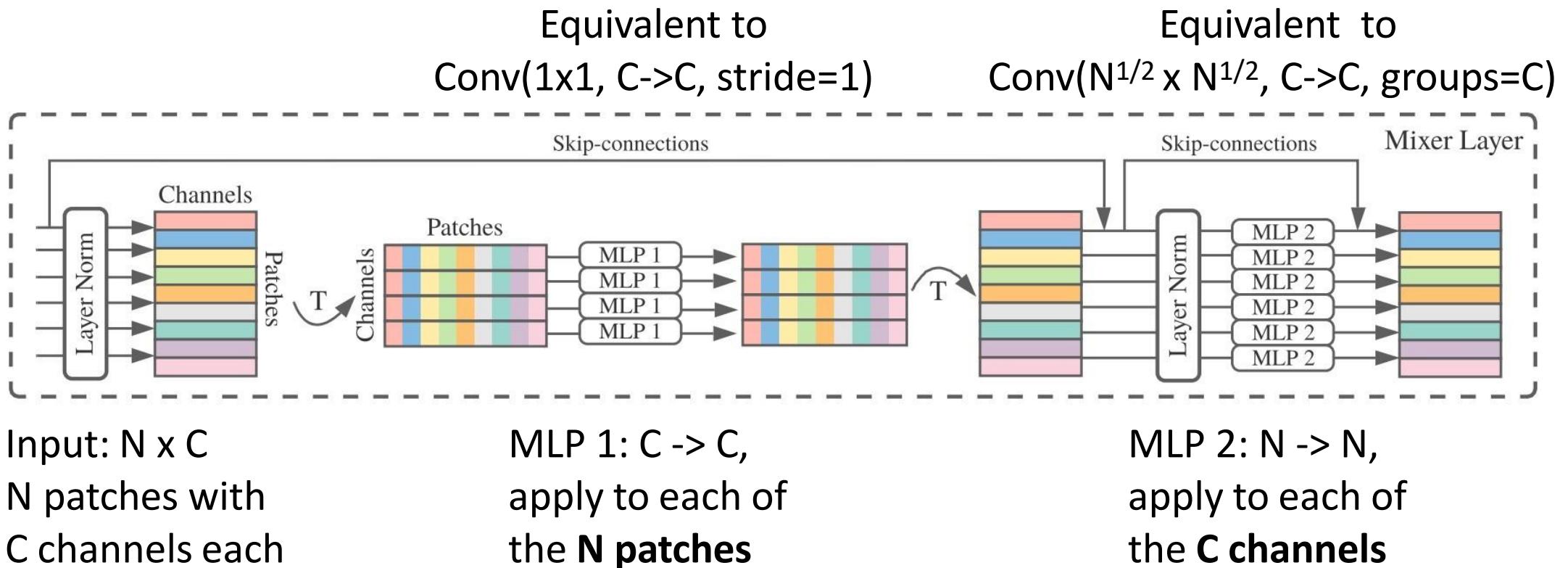
Input: $N \times C$
 N patches with
 C channels each

MLP 1: $C \rightarrow C$,
apply to each of
the **N patches**

MLP 2: $N \rightarrow N$,
apply to each of
the **C channels**

MLP-Mixer: An All-MLP Architecture

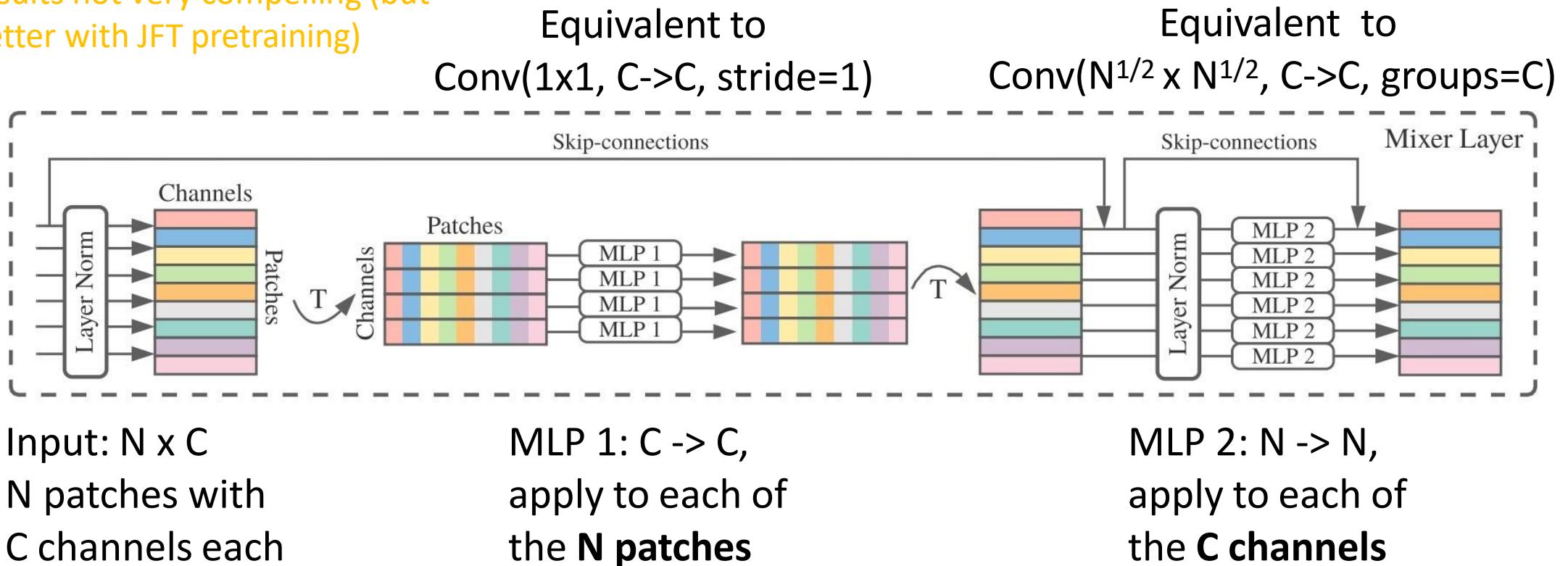
MLP-Mixer is actually just a weird CNN???



MLP-Mixer: An All-MLP Architecture

Cool idea; but initial ImageNet results not very compelling (but better with JFT pretraining)

MLP-Mixer is actually just a weird CNN???



MLP-Mixer: Many concurrent and followups

Touvron et al, “ResMLP: Feedforward Networks for Image Classification with Data-Efficient Training” , arXiv 2021,
<https://arxiv.org/abs/2105.03404>

Tolstikhin et al, “MLP-Mixer: An all-MLP architecture for vision” , NeurIPS 2021, <https://arxiv.org/abs/2105.01601>

Liu et al, “Pay Attention to MLPs” , NeurIPS 2021,
<https://arxiv.org/abs/2105.08050>

Yu et al, “S2-MLP: Spatial-Shift MLP Architecture for Vision” , WACV 2022, <https://arxiv.org/abs/2106.07477>

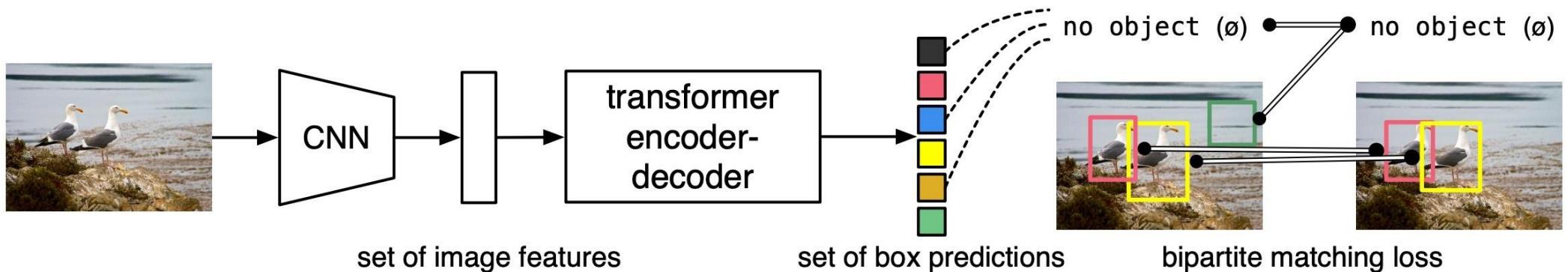
Chen et al, “CycleMLP: A MLP-like Architecture for Dense Prediction” , ICLR 2022, <https://arxiv.org/abs/2107.10224>

Object Detection with Transformers: DETR

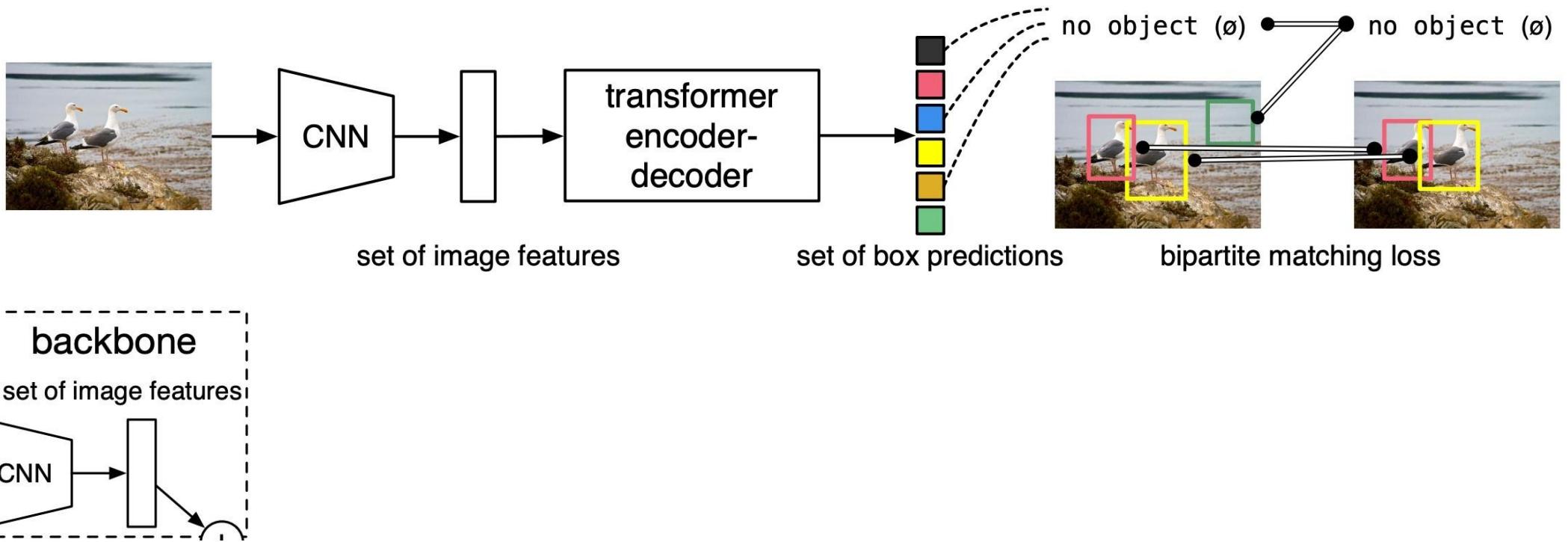
Simple object detection pipeline: directly output a set of boxes from a Transformer

No anchors, no regression of box transforms

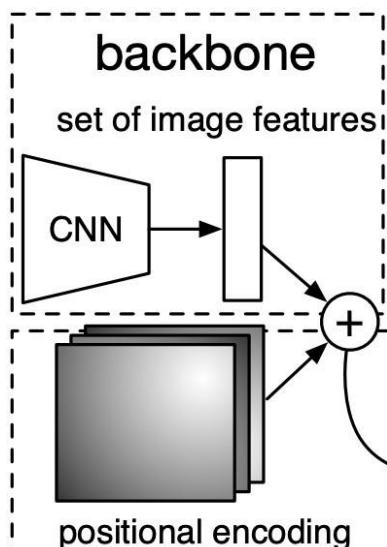
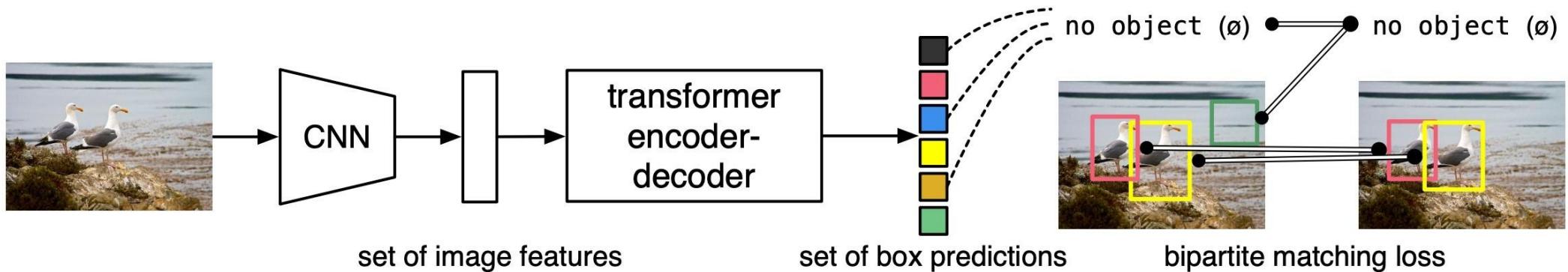
Match predicted boxes to GT boxes with bipartite matching; train to regress box coordinates



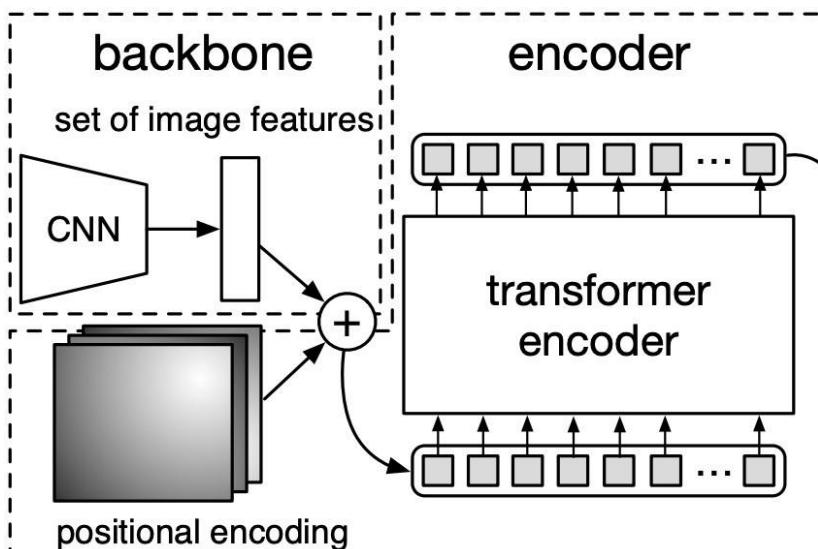
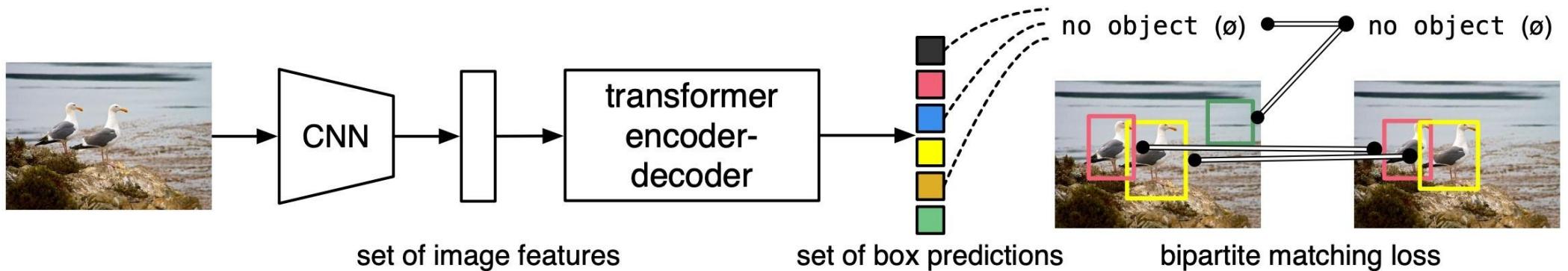
Object Detection with Transformers: DETR



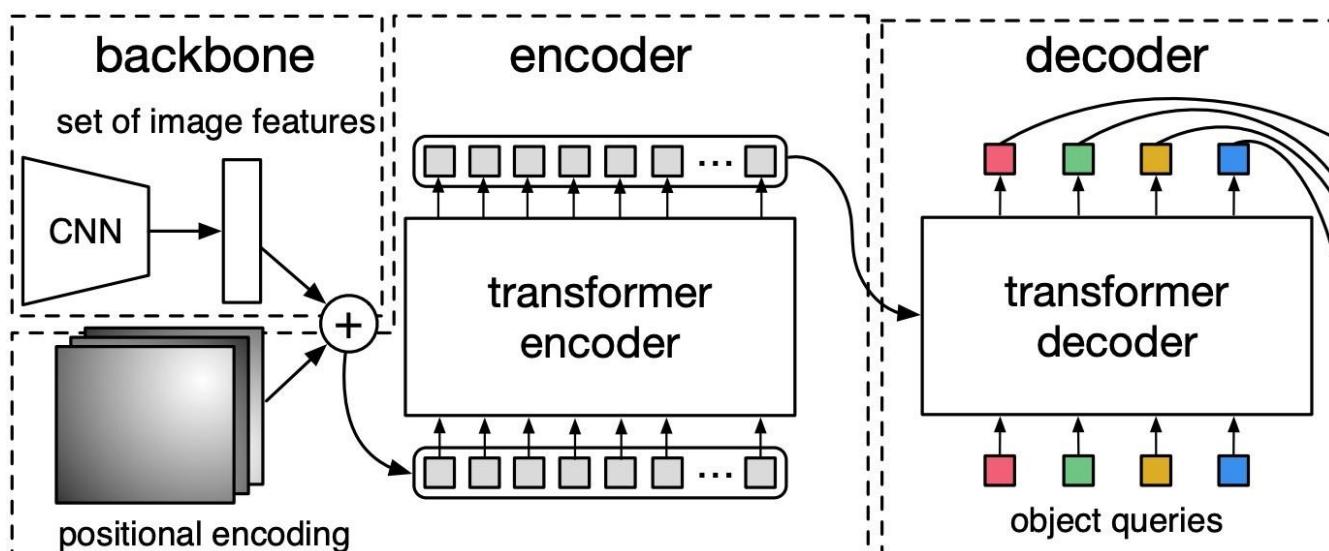
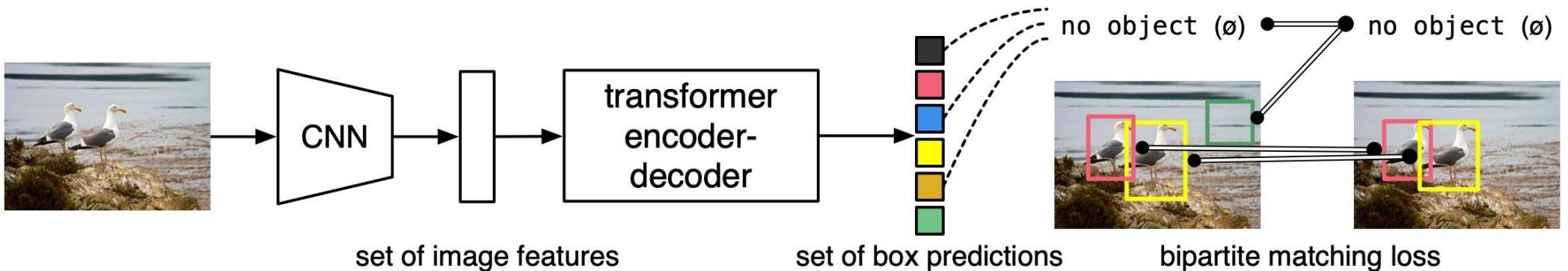
Object Detection with Transformers: DETR



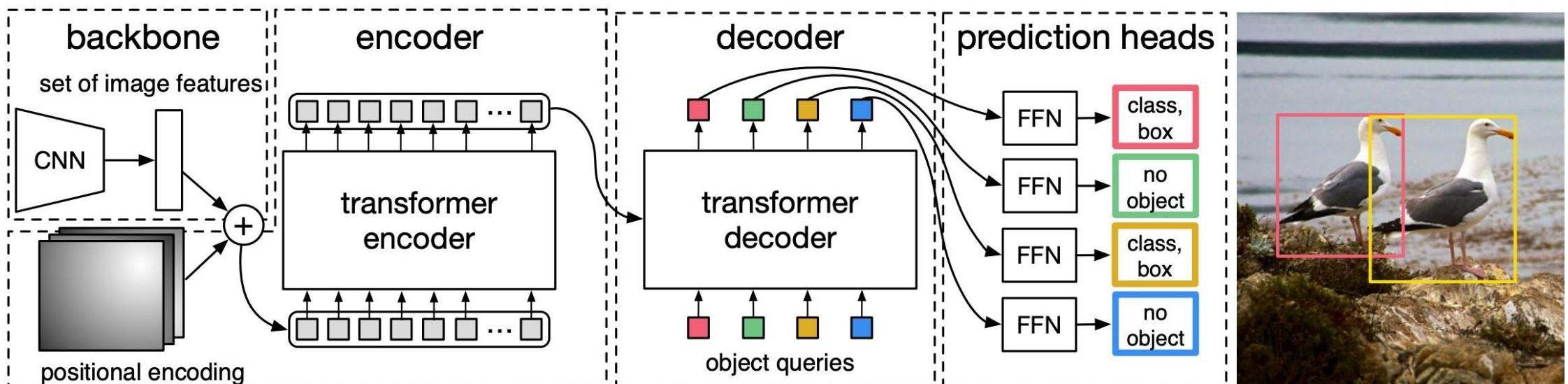
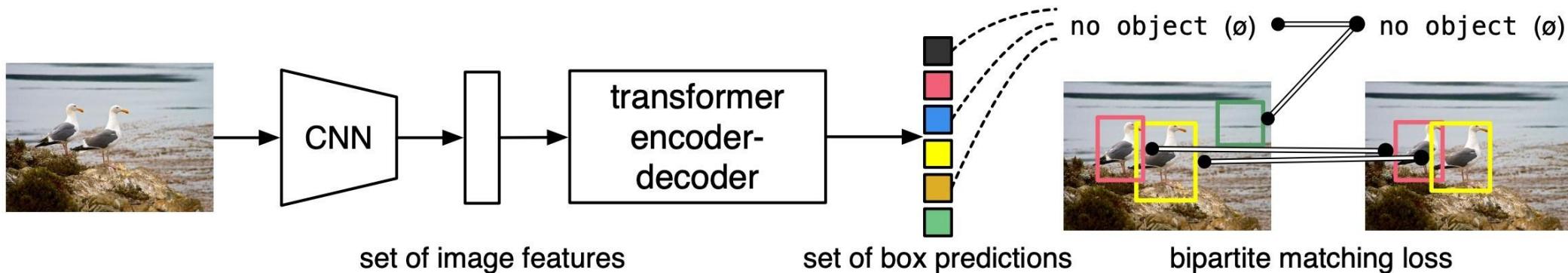
Object Detection with Transformers: DETR



Object Detection with Transformers: DETR



Object Detection with Transformers: DETR



Summary

Vision Transformers have been a super hot topic the past ~1-2 years!

Very different architecture vs traditional CNNs

Applications to all tasks: classification, detection, segmentation, etc

My takeaway: Vision transformers are an evolution, not a revolution.
We can still fundamentally solve the same problems as with CNNs.

Main benefit is probably speed: Matrix multiply is more hardware-friendly than convolution, so ViTs with same FLOPs as CNNs can train and run much faster