Swin Transformer
Hierarchical Vision Transformer using Shifted Windows
(ICCV2021 - Best paper)

Ze Liu, Yutong Lin, Yue Cao, Han Hu,
Yixuan Wei, Zheng Zhang, Stephen Lin,
Baining Guo

Microsoft Research Asia

https://theaisummer.com/cnn-architectures/

Artificial Intelligence

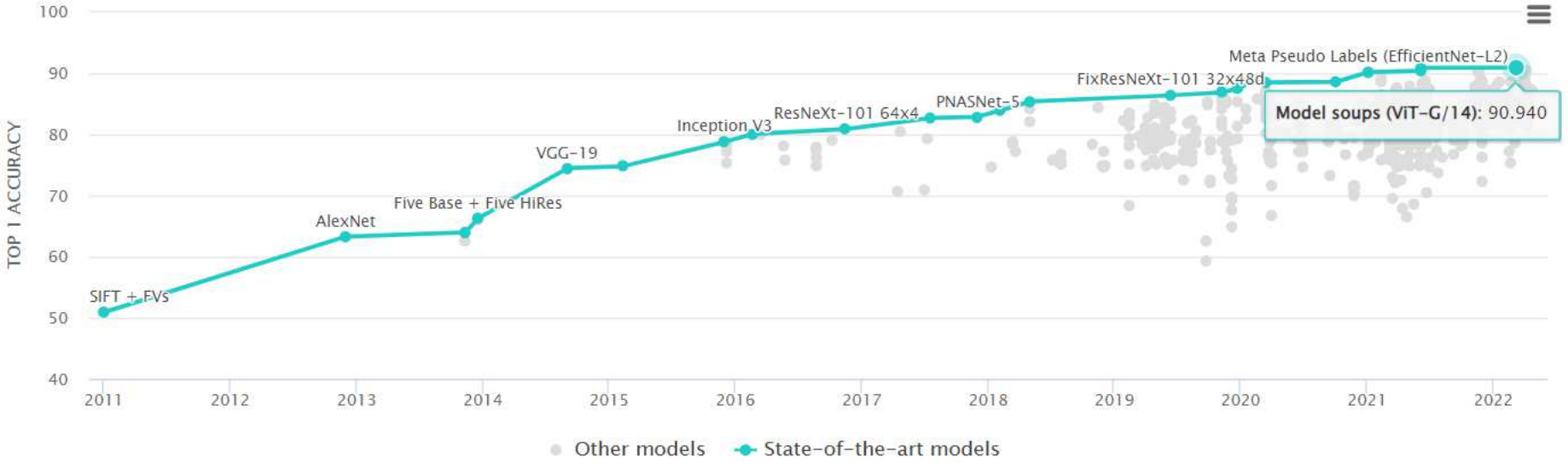
Creating the Future

Dong-A University

Division of Computer Engineering &
Artificial Intelligence

References

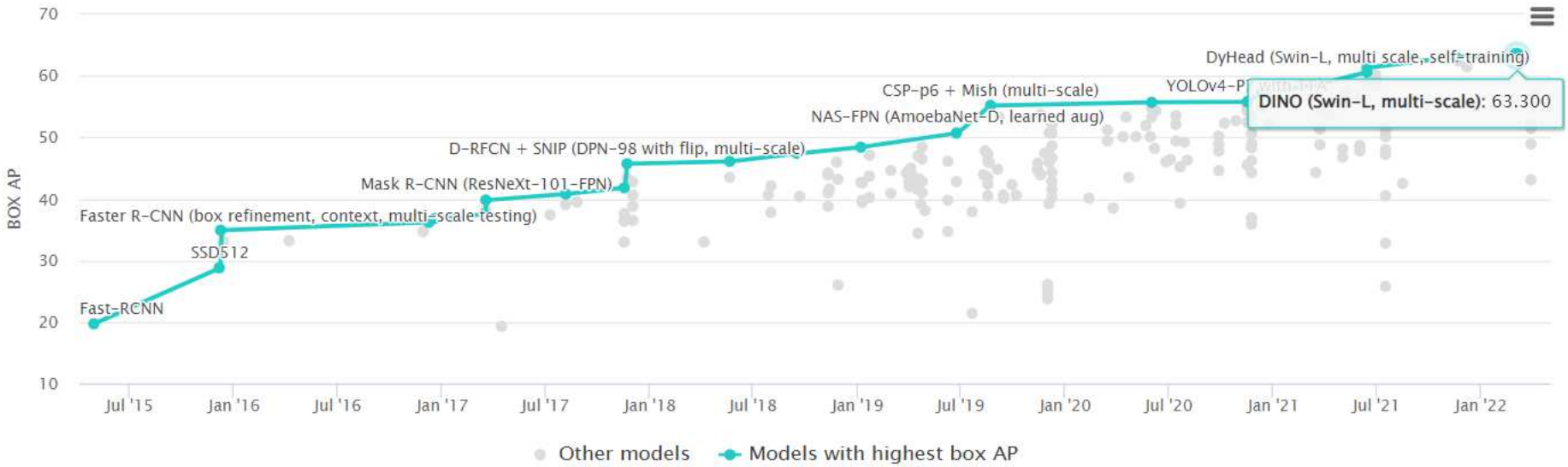
SOTA Classification



Rank	Model	Top 1 Accuracy	Top 5 Accuracy	Number of params	Extra Training Data	Paper	Code	Result	Year	Tags
1	**Model soups** (ViT-G/14)	90.94		1843M	✓	Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time			2022	Transformer, JFT-3B
2	**CoAtNet-7**	90.88%		2440M	×	CoAtNet: Marrying Convolution and Attention for All Data Sizes			2021	Conv+Transformer, JFT-3B
3	**ViT-G/14**	90.45%		1843M	✓	Scaling Vision Transformers			2021	Transformer, JFT-3B
4	**CoAtNet-6**	90.45%		1470M	×	CoAtNet: Marrying Convolution and Attention for All Data Sizes			2021	Conv+Transformer, JFT-3B
5	**DaViT-G**	90.4%		1437M	×	DaViT: Dual Attention Vision Transformers			2022	Transformer
6	**Meta Pseudo Labels** (EfficientNet-L2)	90.2%	98.8%	480M	×	Meta Pseudo Labels			2021	EfficientNet, JFT-300M
7	**DaViT-H**	90.2%		362M	×	DaViT: Dual Attention Vision Transformers			2022	Transformer
8	**SwinV2-G**	90.17%			×	Swin Transformer V2: Scaling Up Capacity and Resolution			2021	Transformer

References

SOTA Object Detection



Rank	Model	box AP	AP50	AP75	APS	APM	APL	AP	Params (M)	Extra Training Data	Paper	Code	Result	Year	Tags
1	**DINO** (Swin-L, multi-scale)	63.3	80.8	69.9	46.7	66.0	76.5			✓	DINO: DETR with Improved DeNoising Anchor Boxes for End-to-End Object Detection			2022	multiscale, Swin-Transformer, End-to-End, Query Denoising
2	**DINO** (Swin-L, single-scale)	63.2								✓	DINO: DETR with Improved DeNoising Anchor Boxes for End-to-End Object Detection			2022	Swin-Transformer, End-to-End, Query Denoising
3	**SwinV2-G** (HTC++)	63.1								✓	Swin Transformer V2: Scaling Up Capacity and Resolution			2021	multiscale, Swin-Transformer
4	**Florence-CoSwin-H**	62.4								✓	Florence: A New Foundation Model for Computer Vision			2021	Swin-Transformer
5	**GLIP** (Swin-L, multi-scale)	61.5	79.5	67.7	45.3	64.9	75.0			✓	Grounded Language-Image Pre-training			2021	multiscale, Vision Language, Dynamic Head, BERT-Base
6	**Soft Teacher + Swin-L** (HTC++, multi-scale)	61.3								✓	End-to-End Semi-Supervised Object Detection with Soft Teacher			2021	multiscale, Swin-Transformer

References

arXiv

- https://arxiv.org/abs/2103.14030

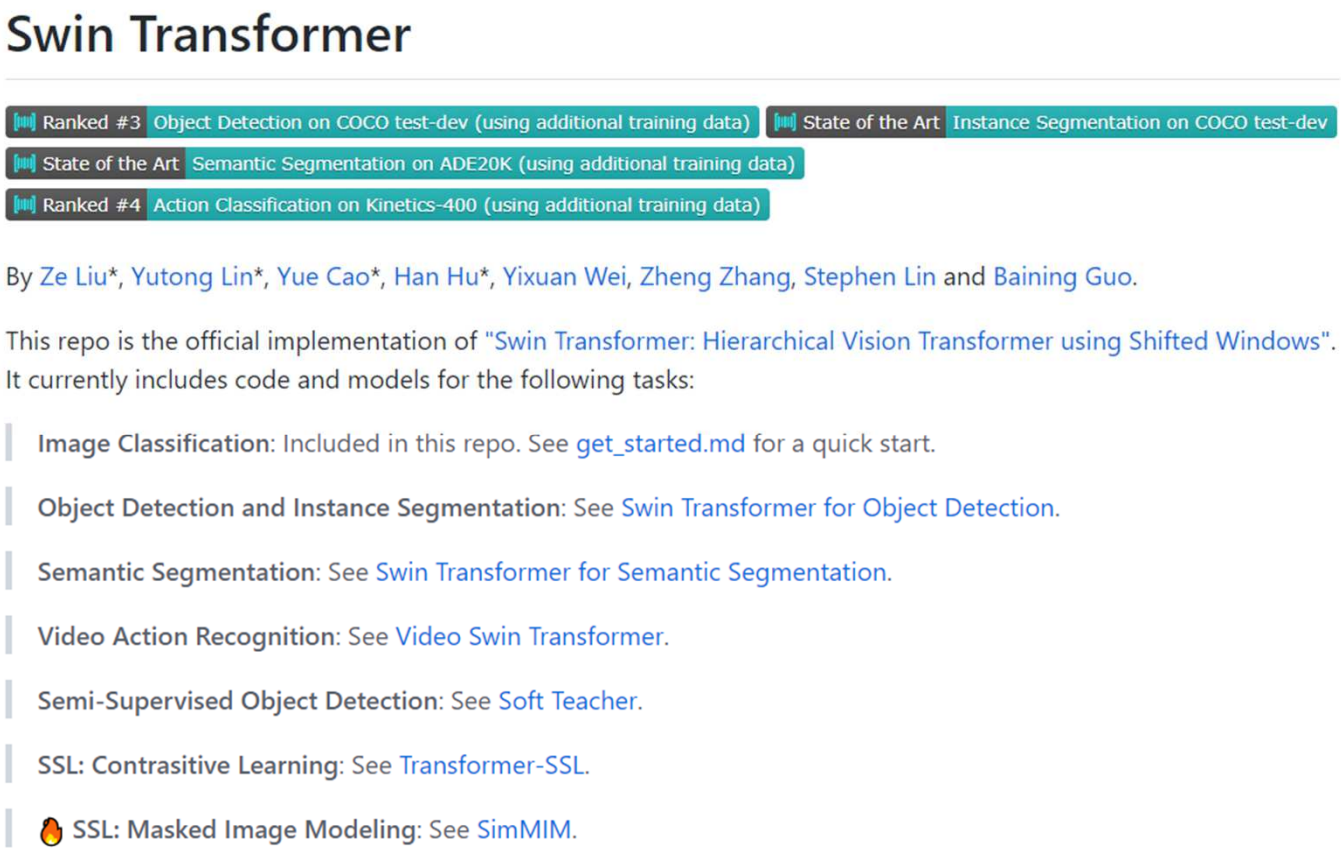
Official github

- https://github.com/microsoft/Swin-Transformer

Blog

- https://visionhong.tistory.com/31
- https://greeksharifa.github.io/computer%20vision/2021/12/14/Swin-Transformer/

- https://www.youtube.com/watch?v=gFEQz8qJ6zY
- https://byeongjo-kim.tistory.com/36



1 Swin Transformer

Abstract

- Hierarchical Transformer whose representation is computed with **Shifted windows**.
- The **shifted windowing** scheme brings greater efficiency by **limiting self-attention computation to non-overlapping local windows** while also **allowing for cross-window connection**.
- This **hierarchical architecture** has **the flexibility to model at various scales** and has **linear computational complexity with respect to image size**.
- Image classification (87.3 top-1 accuracy on ImageNet-1K) and dense prediction tasks such as object detection (58.7 box AP and 51.1 mask AP on COCO testdev) and semantic segmentation (53.5 mIoU on ADE20K val).
- Its performance surpasses the previous SOTA by a large margin of +2.7 box AP and +2.6 mask AP on COCO, and +3.2 mIoU on ADE20K, demonstrating the potential of Transformer-based models as vision backbones.
- The hierarchical design and the shifted window approach also prove beneficial for all-MLP architectures

ViT

- ✓ 이미지의 경우 visual entity의 크기(scale)와 해상도가 매우 다양함
- ✓ ViT에서 모든 patch가 self attention 하여 computation cost 높음

Swin Transformer

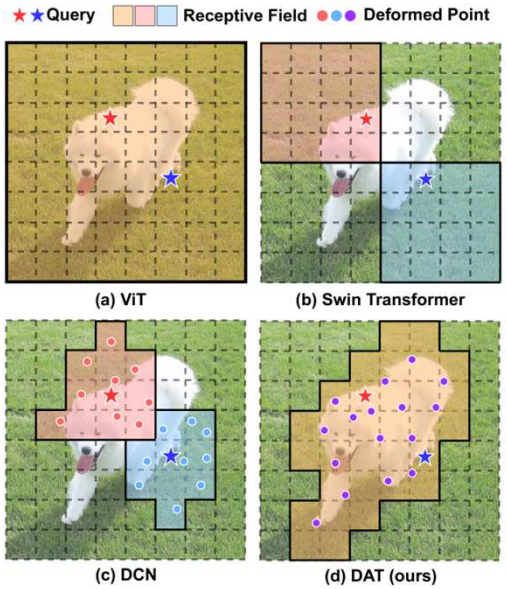
- ✓ 각 patch를 window로 나누어 해당 윈도우 안에서만 self attention을 수행하고 그 윈도우를 한번 shift하고 다시 self attention을 하는 구조를 제시
- ✓ Hierarchical Shifted WINdow 방식은, 기존 self-attention의 제곱에 비례하는 계산량을 선형 비례하게 줄이면서도 다양한 scale을 처리할 수 있는 접근법임.
- ✓ 이로써 image classification, object detection, semantic segmentation 등 다양한 범위의 vision task에서 훌륭한 성과를 보였다.

1 Swin Transformer

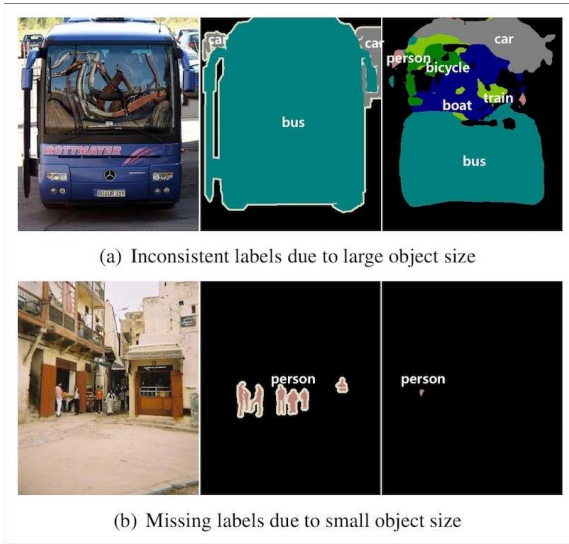
ViT (Vision Transformer) : An Image is Worth 16x16 Words (ICLR2021)

The Drawback 1 : Fixed Scale

- Unlike the word tokens, visual elements can vary in scale
- However, tokens in ViT are all of a fixed scale which is not suitable for vision applications.
- Low accuracy



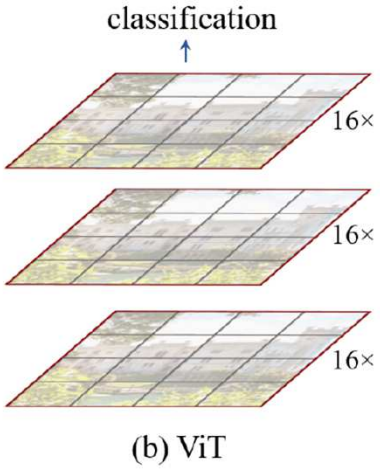
[Ref] Vision Transformer with Deformable Attention (DAT)



[Ref] Learning Deconvolution Network for Semantic Segmentation

The Drawback 2 : Computational Cost

- ViT architecture conduct only global self-attention
- The global computation leads to *quadratic complexity with respect to input image size*.
- Require high FLOPs machine



$$\Omega(\text{MSA}) = 4hwC^2 + 2(hw)^2C$$

1 Swin Transformer

Swin Transformer

h,w : # of patch size in x-y axis
M : size of local window

$$\Omega(\text{W-MSA}) = 4hwC^2 + 2M^2hwC \qquad \Omega(\text{MSA}) = 4hwC^2 + 2(hw)^2C$$

Hierarchical Feature Map (key : Patch Merging)

- Start from small-size patches
- Gradually merging neighboring patches as forwarded
- Various size of receptive fields are adopted indirectly
- Achieving high accuracy

Local Window Self-attention (Key : Shift window based MSA)

- Compute self-attention *locally* only within non-overlapping window
- The complexity becomes *linear* to image size
- Achieving lower FLOPs to train and infer

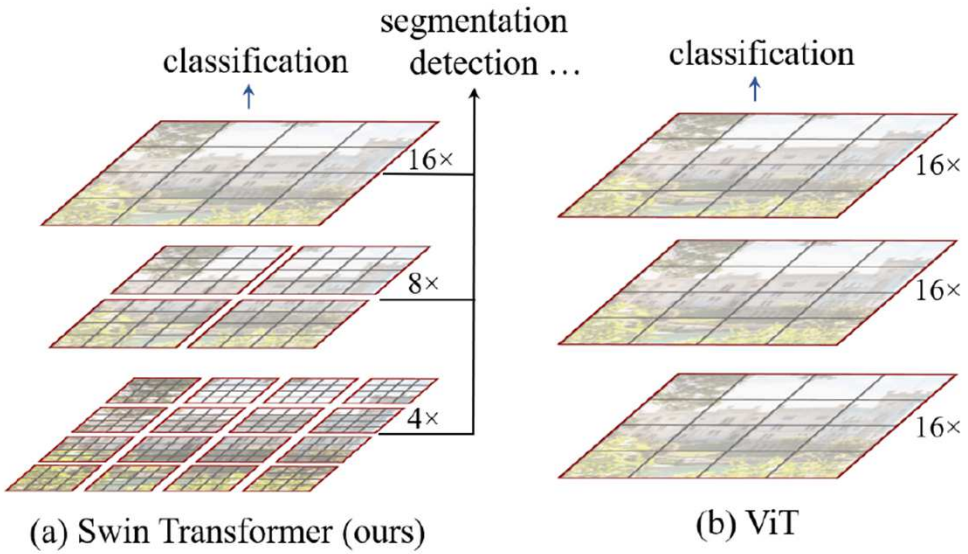
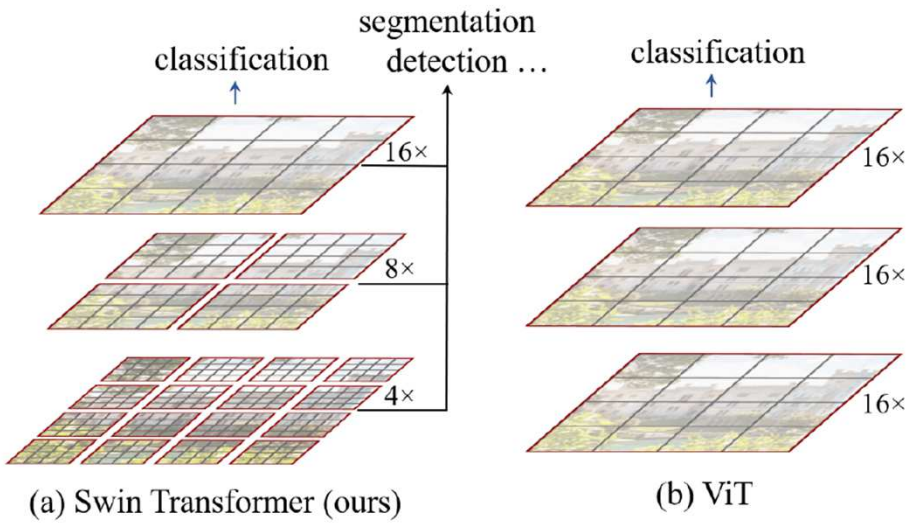


Figure 1. (a) The proposed Swin Transformer builds **hierarchical feature maps** by **merging image patches** (shown in gray) in **deeper layers** and has **linear computation complexity to input image size due to computation of self-attention only within each local window** (shown in red). It can thus serve as a general-purpose backbone for both image classification and dense recognition tasks. (b) In contrast, previous vision Transformers [20] produce feature maps of a single low resolution and have quadratic computation complexity to input image size due to computation of self attention globally.

1 Swin Transformer

Introduction



- 입력 이미지 사이즈가 224x224라고 생각해보자.
- ViT는 각 패치 사이즈를 16x16으로 만들어 총 (224/16)**2=196개 patch를 가진 상태를 유지하고 각 patch와 나머지 전체 patch에 대한 self-attention을 수행한다. (quadratic computational complexity to image size)
- Swin Transformer는 마치 feature pyramid network처럼 작은 patch 4x4에서 시작해서 점점 patch들을 merge 해 나가는 방식을 취한다.
- 그림을 보면 빨간선으로 patch들이 나누어져 있는것을 볼 수 있는데 이것을 각각 window라고 부르고 Swin Transformer는 window내의 patch들끼리만 self-attention을 수행한다. (linear computational complexity to image size)
- 논문에서는 각 window size(M)을 7x7로 한다. 첫번째 레이어에서 4x4 size의 각 patch가 56x56개가 있고 이것을 7x7 size의 window로 나누어 8x8개의 window가 생긴다.
- 즉 첫번째 stage에서 각 patch는 16개의 pixel이 있고 각 윈도우에는 49개의 patch가 있다는 의미 (사실 embedding을 하기 때문에 채널을 곱해줘야 하는데 그림의 이해를 돕기 위해 채널은 곱하지 않았음)

- 입력 이미지 : 224x224, Window size (M) : 7x7
- 1 stage : Patch size 4x4, $\frac{224}{4} \times \frac{224}{4} = 56 \times 56$ patches, $\frac{56}{7} \times \frac{56}{7} = 8 \times 8$ windows

	1stage		2stage		3stage	
Image size	224	224	224	224	224	224
Patch size	4		8	8	16	16
# of patches	56	56	28	28	14	14
Window size	7	7	7	7	7	7
# of windows	8	8	4	4	2	2

1 Swin Transformer

Introduction

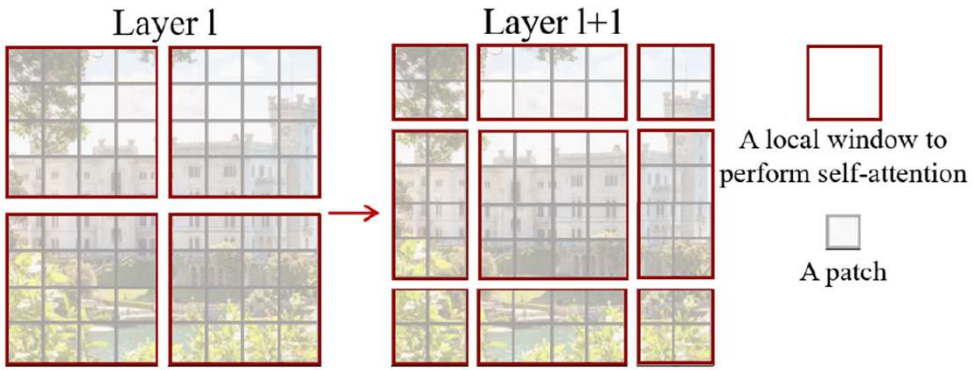


Figure 2. An illustration of the *shifted window* approach for computing self-attention in the proposed Swin Transformer architecture. In layer l (left), a regular window partitioning scheme is adopted, and self-attention is computed within each window. In the next layer $l + 1$ (right), the window partitioning is shifted, resulting in new windows. The self-attention computation in the new windows crosses the boundaries of the previous windows in layer l, providing connections among them.

- Window를 나누어서 계산을 수행하면 각 window의 경계 근처 pixel들은 인접해 있음에도 self-attention 계산이 수행되지 않는데, 이를 위해 window 크기의 절반만큼 shift하여 비슷한 계산을 수행한다

Key Design Element :

Shifted windows (Propose)

- **Shift of the window partition** between consecutive self-attention
- ***The shifted windows*** bridge the windows of the preceding layer, providing connections among them that significantly enhance *modeling power* (see Table 4).
- Efficient in regards to real-world latency: **all *query* patches within a window share the same key set**, which facilitates memory access in hardware.

Sliding windows (Conventional)

- sliding window based self-attention approaches [33, 50] suffer from **low latency** on general hardware due to **different key sets for different query pixels**.

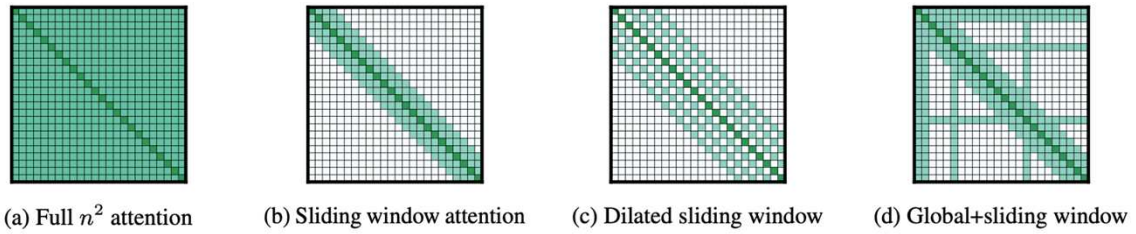


Figure 2: Comparing the full self-attention pattern and the configuration of attention patterns in our Longformer.

1 Swin Transformer

Related Work

CNN and Variants

- Primary backbone architectures for computer vison application
- AlexNet, VGG, GoogleNet, ResNet, DensNet, HRNet, EfficientNet
- Improve individual convolution layers : ***depthwise convolution***, ***deformable convolution***

➢ We **highlight the strong potential of transformer-like architecture for unified modeling between vision and language**.

Self-attention based backbone architectures

- Some works employ *self-attention layers* to replace some or all of the spatial convolution layers in ResNet.
- Han et al. (ICCV2019) compute **self-attention within a local window of each pixel** to expedite optimization [33],
  - ✓ Slightly **better accuracy/FLOPs trade-offs** than ResNet architecture.
  - ✓ However, their **costly memory access** causes their actual latency. Instead of using sliding windows,

➢ We **propose to shift windows between consecutive layers**, which allows for a more efficient implementation in general hardware.

Self-attention/Transformer to complement CNNs

- Some work is to augment a standard CNN architecture with self-attention layers or Transformers.
- The **self-attention layers** can **complement backbones or head networks** by providing the capability to **encode distant dependencies or heterogeneous interactions**.
- More recently, the encoder-decoder design in Transformer has been applied for the **object detection** and **instance segmentation tasks**.

➢ Our work explores **the adaptation of Transformers for basic visual feature extraction** and is complementary to these works

Swin Transformer

Deformable Convolution

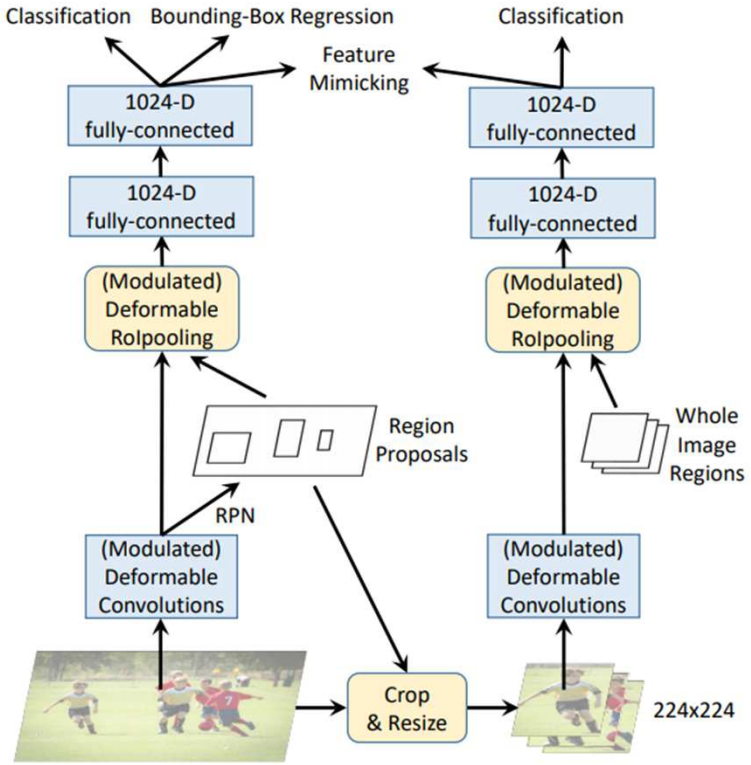
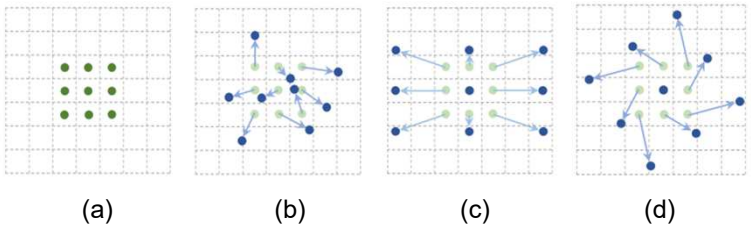


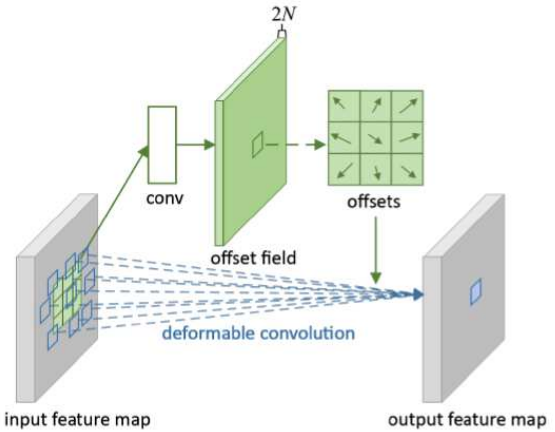
Figure 3. Network training with R-CNN feature mimicking.

X. Zhu et al., “Deformable ConvNets v2: More Deformable, Better Results”, CVPR, 2019



(a) (b) (c) (d)

(a) Conventional Convolution, (b) Deformable Convolution, (c) Special Case of Deformable Convolution with Scaling, (d) Special Case of Deformable Convolution with Rotation



Deformable Convolution

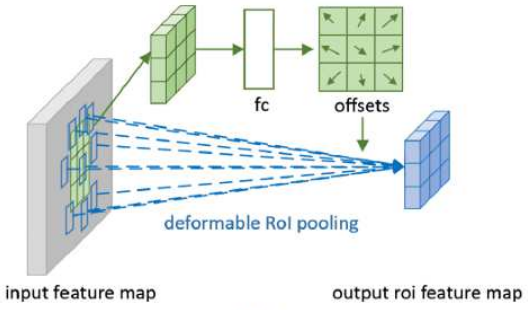
Regular convolution

$$\mathbf{y}(\mathbf{p}_0) = \sum_{\mathbf{p}_n \in \mathcal{R}} \mathbf{w}(\mathbf{p}_n) \cdot \mathbf{x}(\mathbf{p}_0 + \mathbf{p}_n)$$

Deformable convolution

$$\mathbf{y}(\mathbf{p}_0) = \sum_{\mathbf{p}_n \in \mathcal{R}} \mathbf{w}(\mathbf{p}_n) \cdot \mathbf{x}(\mathbf{p}_0 + \mathbf{p}_n + \Delta\mathbf{p}_n)$$

where $\Delta\mathbf{p}_n$ is generated by a sibling branch of regular convolution



deformable RoI Pooling

Regular RoI pooling

$$\mathbf{y}(i,j) = \sum_{\mathbf{p} \in bin(i,j)} \mathbf{x}(\mathbf{p}_0 + \mathbf{p})/n_{ij}$$

Deformable RoI pooling

$$\mathbf{y}(i,j) = \sum_{\mathbf{p} \in bin(i,j)} \mathbf{x}(\mathbf{p}_0 + \mathbf{p} + \Delta\mathbf{p}_{ij})/n_{ij}$$

where $\Delta\mathbf{p}_{ij}$ is generated by a sibling fc branch

$$\Delta\mathbf{p}_{ij} = \gamma \cdot \Delta\widehat{\mathbf{p}}_{ij} \circ (w, h)$$

1 Swin Transformer

Self-Attention based backbone architectures

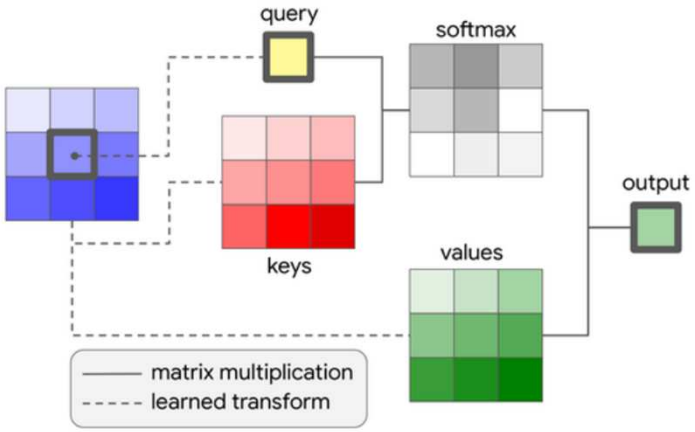


Figure 3: An example of a local attention layer over spatial extent of $k = 3$.

-1, -1	-1, 0	-1, 1	-1, 2
0, -1	0, 0	0, 1	0, 2
1, -1	1, 0	1, 1	1, 2
2, -1	2, 0	2, 1	2, 2

Figure 4: An example of relative distance computation. The relative distances are computed with respect to the position of the highlighted pixel. The format of distances is *row offset*, *column offset*.

P. Ramachandran et al., “Stand-Alone Self-Attention in Vision Models”, NIPS, 2019

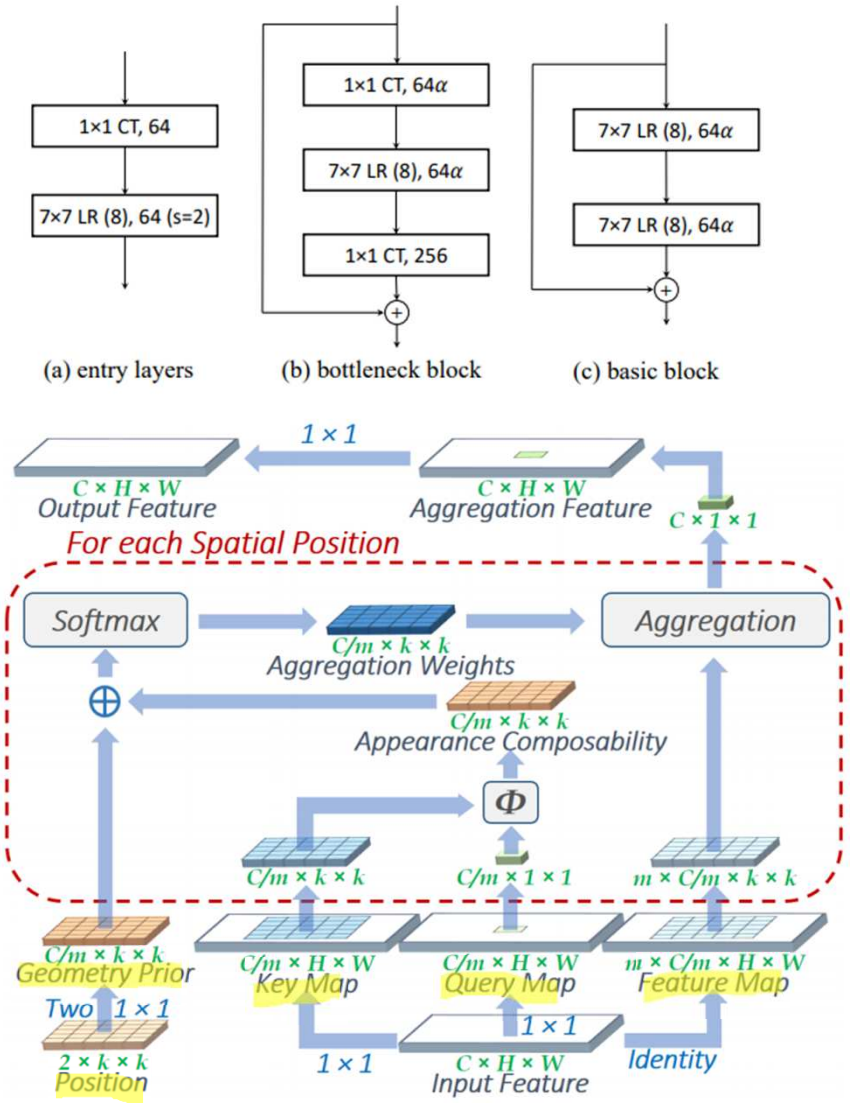


Figure 2. The local relation layer.

H. Hu et al., “Local Relation Networks for Image Recognition”, ICCV, 2019

Swin Transformer

Self-Attention/Transformers to complement CNNs

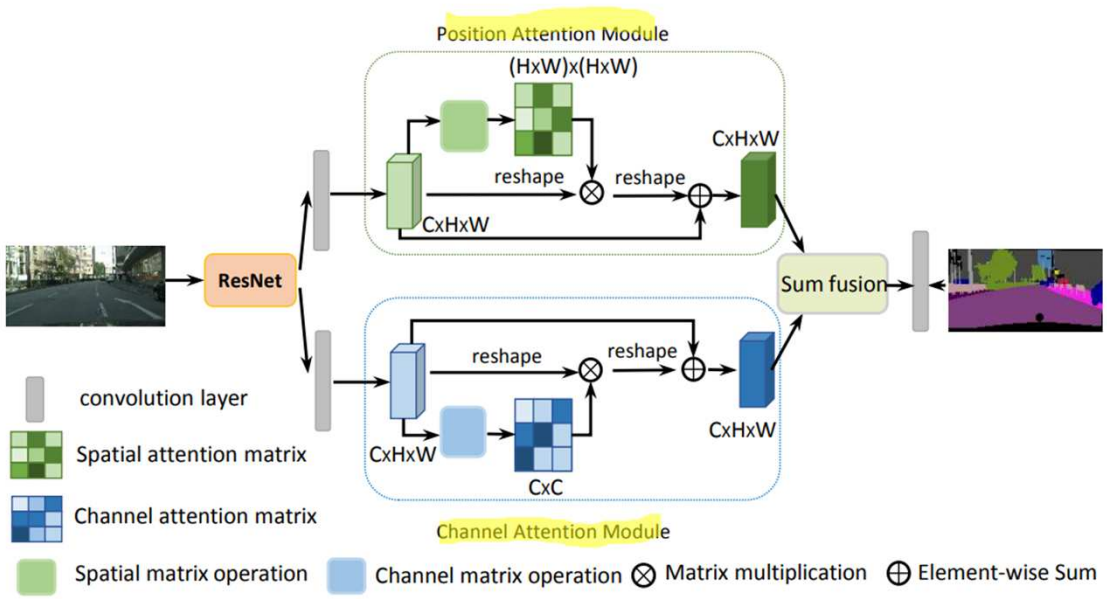


Figure 2: An overview of the Dual Attention Network. (Best viewed in color)

J. Fu et al., “Dual Attention Network for Scene Segmentation”, CVPR, 2019

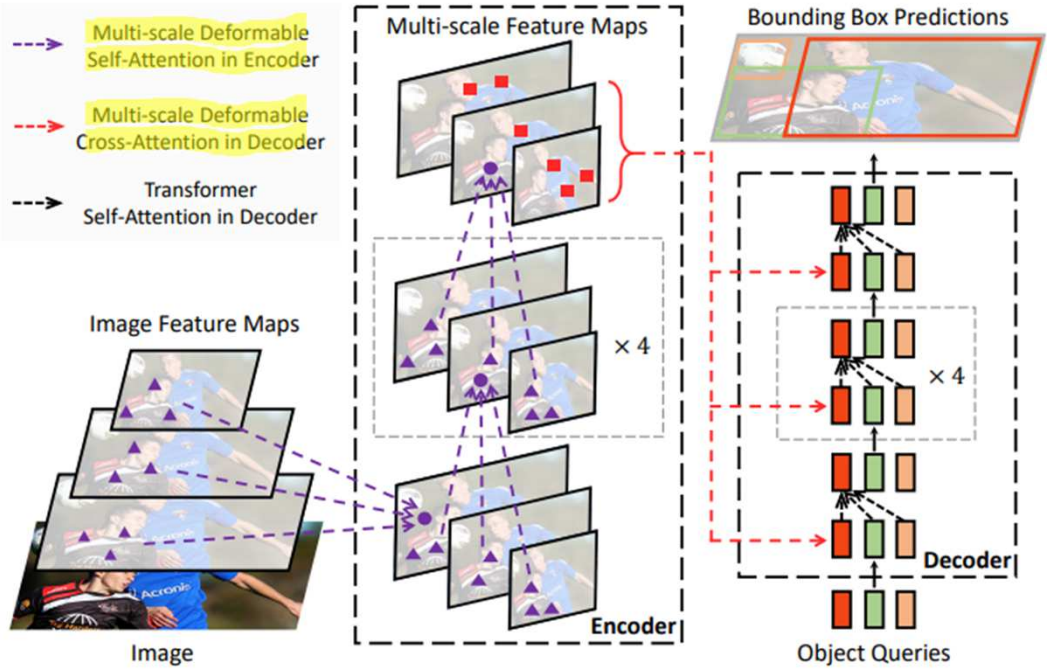


Figure 1: Illustration of the proposed Deformable DETR object detector.

X. Zhu et al., “Deformable DETR: Deformable Transformers for End-to-End Object Detection”, ICLR 2021

1 Swin Transformer

Related Work

Transformer based vision backbones

- **Vision transformer (ViT)**
  - ✓ Directly applies a Transformer architecture on **nonoverlapping medium-sized image patches** for image classification.
  - ✓ Achieves a **speed-accuracy tradeoff** on image classification compared to convolutional networks.
  - ✓ **Requires large-scale training datasets** (i.e., JFT-300M)

❖ **DeiT** [63] introduces several training strategies that allow ViT to also be effective using the **smaller ImageNet-1K dataset.**

- Unsuitable to a general-purpose backbone network on dense vision tasks or high resolution of input image, due to **its low-resolution feature maps and the quadratic increase in complexity with image size.**

➢ Find our Swin Transformer architecture to achieve the **best speed accuracy trade-off**

➢ Build **multi-resolution feature maps** on Transformers but complexity is **linear** (not quadratic to image size) and **operates locally**

1 Swin Transformer

3.1 Overall Architecture

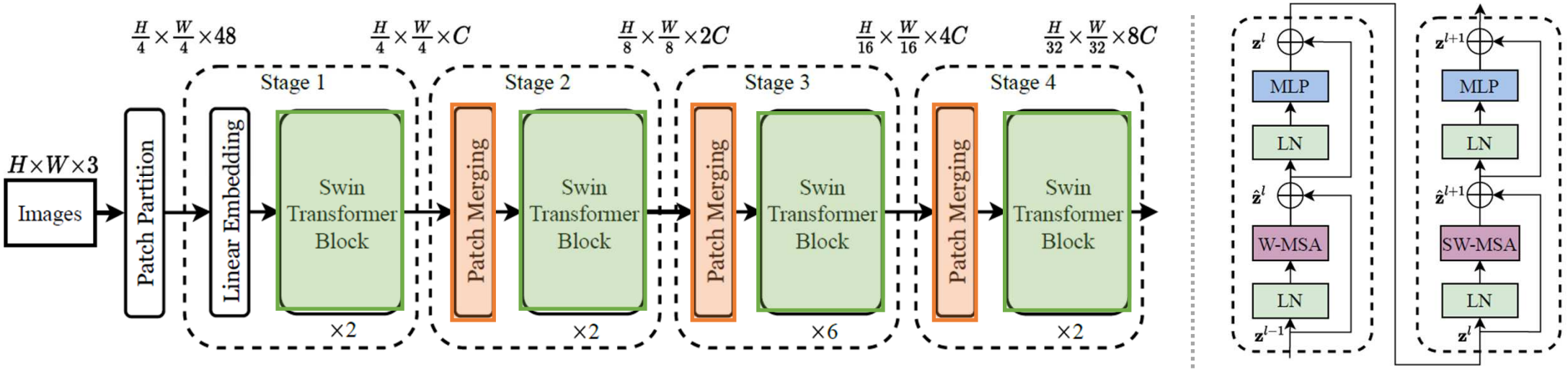


Fig 3. (a) The architecture of a Swin Transformer (Swin-T (Tiny version)

Patch Merging

- For implementing hierarchical feature map
- Various receptive filed

Swin Transformer Block

- For implementing Local window
- Shift window based MSA(Multi-head Self-Attention)

1 Swin Transformer

3.1 Overall Architecture

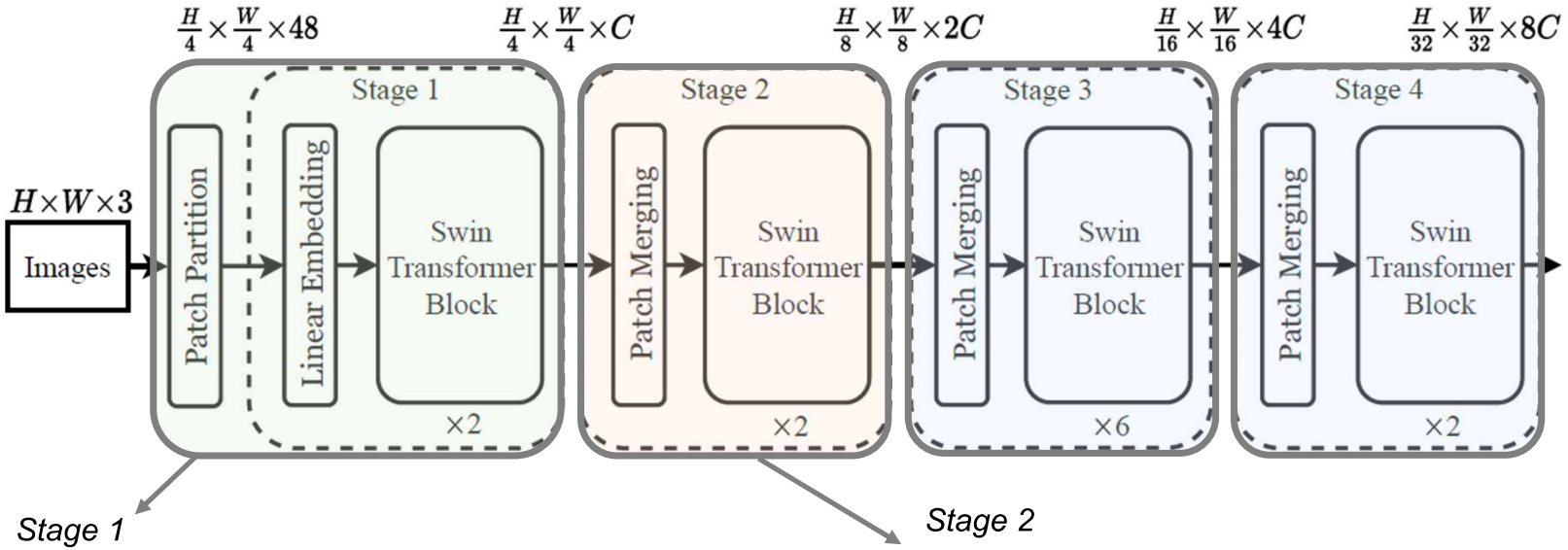


Fig 3. (a) The architecture of a Swin Transformer (Swin-T (Tiny version)

Stage 1

1) Patch Partition : 4x4 non-overlapping patches. (Dimension 4x4x3 = 48)

- Each patch is treated as a "token" and its feature is set as a concatenation of the raw pixel RGB values.

2) **Linear Embedding** : Apply on this raw-valued feature to project it to an arbitrary dimension (denoted as C).

3) **Swin Transformer** on these patch tokens : Maintain the token number $(\frac{H}{4}\times\frac{W}{4})$

Stage 2

1) **Patch Merging** : Reduce the token number
   - Concatenate the features of each group of 2x2 neighboring patches
   - Apply a linear layer on 4C-dim concatenated features.

2) **Swin Transformer** : Maintain token number $(\frac{H}{8}\times\frac{W}{8})$

Stage 3-4 : Repeated twice with output resolution $(\frac{H}{16}\times\frac{W}{16})$, $(\frac{H}{32}\times\frac{W}{32})$

Jointly produce a hierarchical representation, with the same feature map resolution as those of typical CNN (VGG, ResNet)

1 Swin Transformer

3.1 Overall Architecture

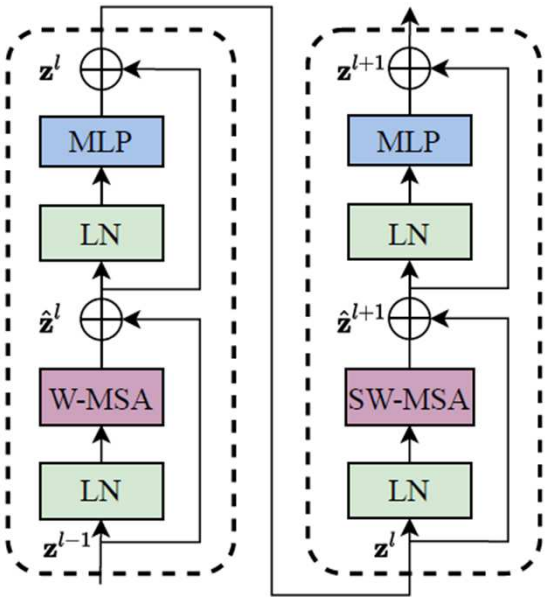


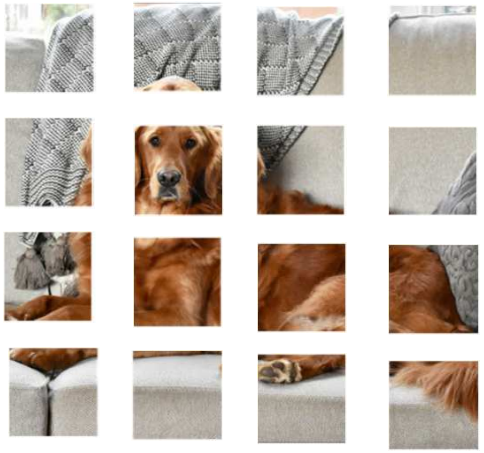
Fig 3. (b) Two successive Swin Transformer Blocks (notation presented with Eq. (3)). W-MSA and SW-MSA are multi-head self attention modules with regular and shifted windowing configurations, respectively

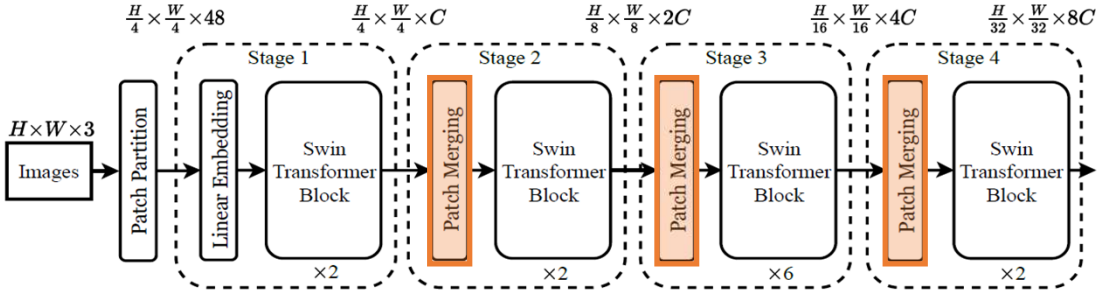
Swin Transformer Block

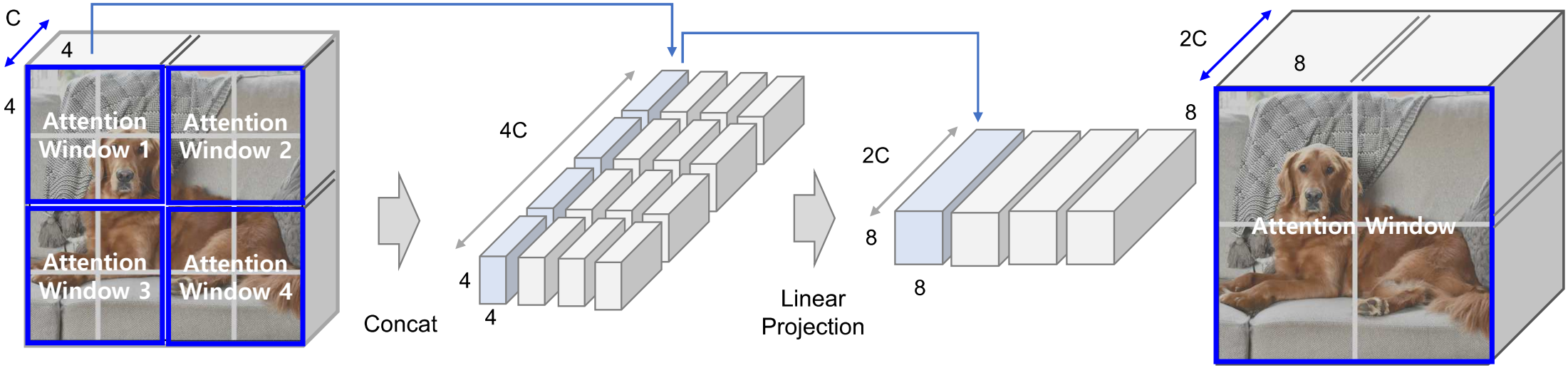
- Replace the standard **multi-head self attention (MSA)** module in a Transformer block by **a module based on shifted windows** (described in Section 3.2), with other layers kept the same.
- Consists of **a shifted window based MSA module**, followed by a **2-layer MLP with GELU nonlinearity** in between.
- A **LayerNorm (LN) layer** is applied before each MSA module and each MLP, and a **residual connection** is applied after each module.

1 Swin Transformer

Patch Merging

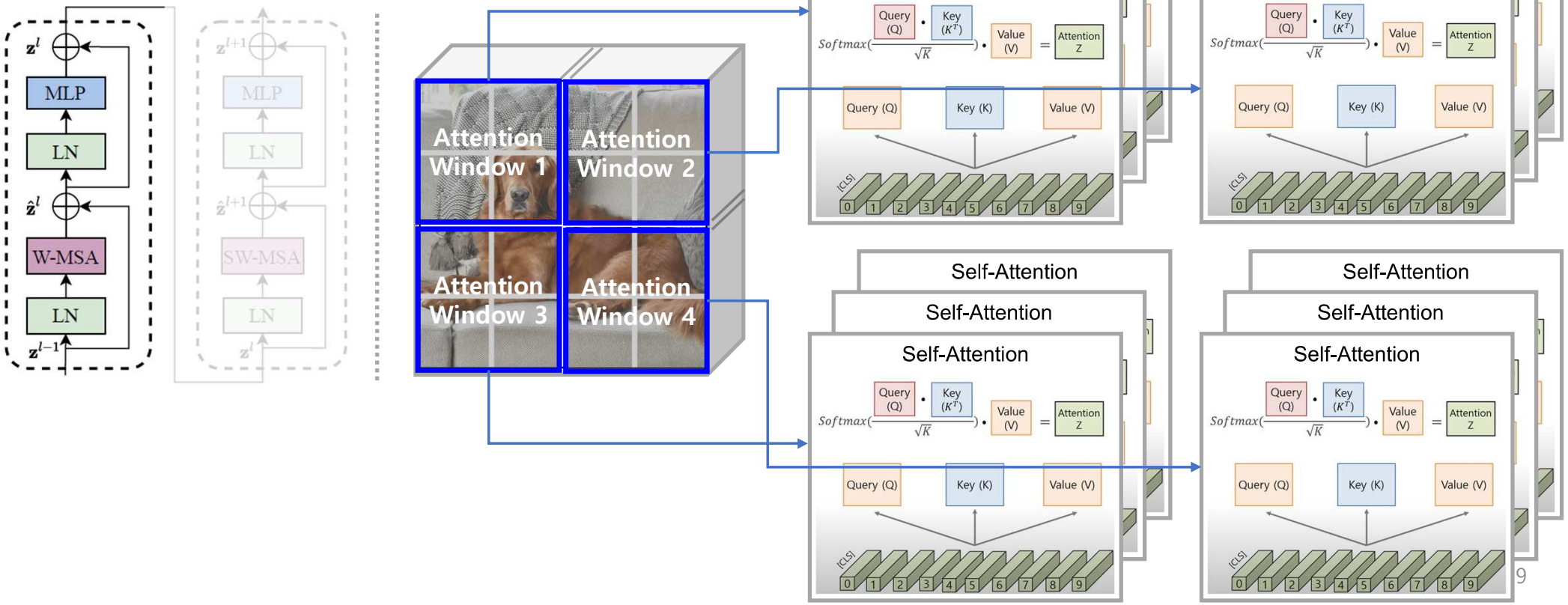






Swin Transformer

Swin Transformer Block

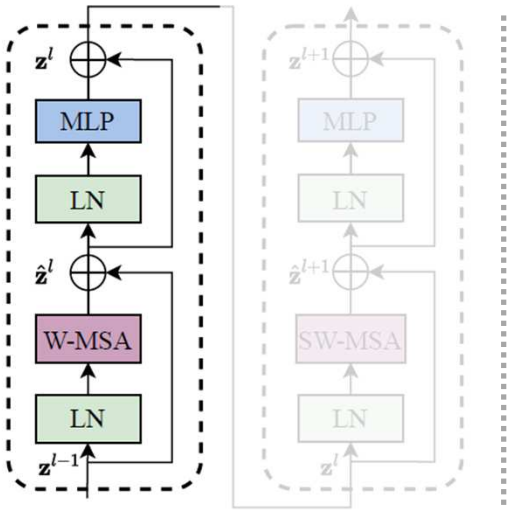


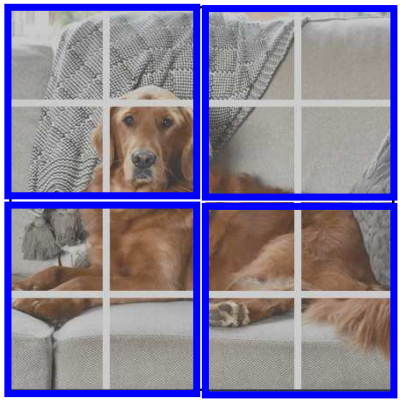
1 Swin Transformer

Swin Transformer Block

W-MSA (Window based Multi Self Attention)

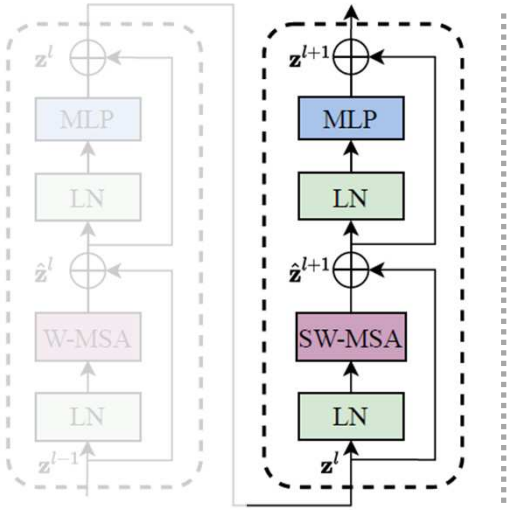
- lacks connections across window





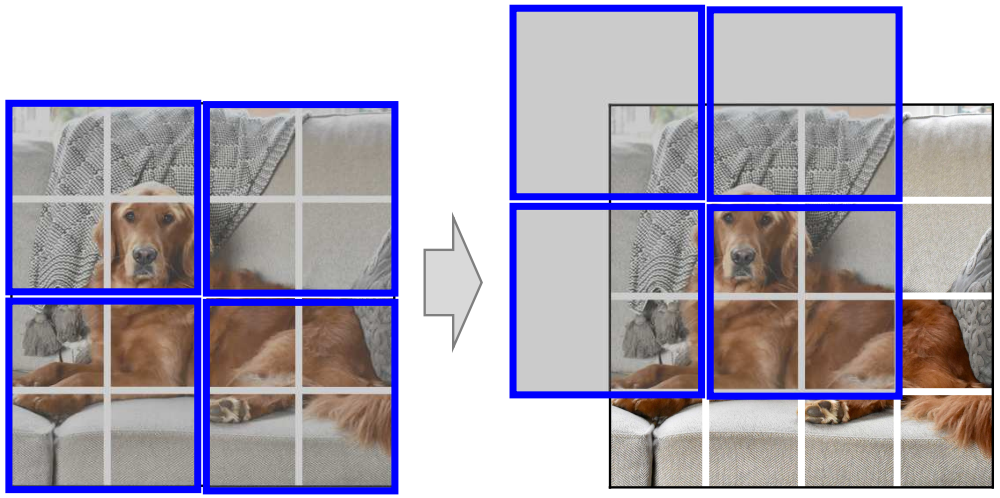
Swin Transformer

Swin Transformer Block



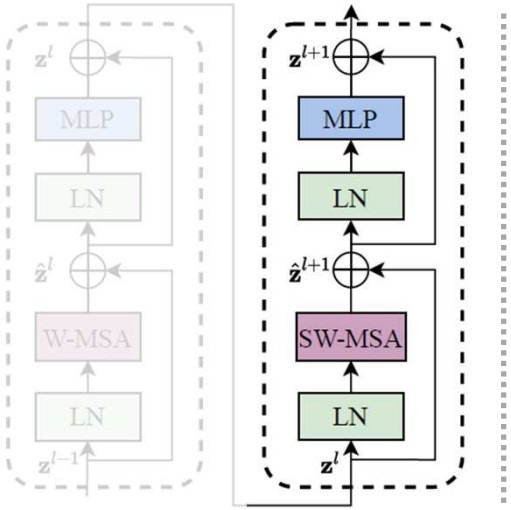
SW-MSA (Shifted Window Multi Self Attention)

- The shifted window partitioning approach introduce connections between neighboring non-overlapping windows
- Efficient batch computation makes the computation efficient



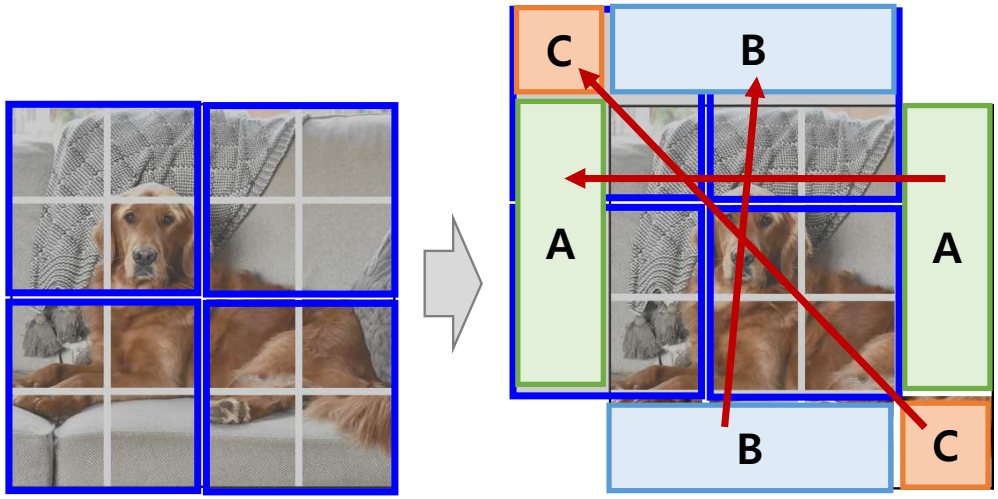
Swin Transformer

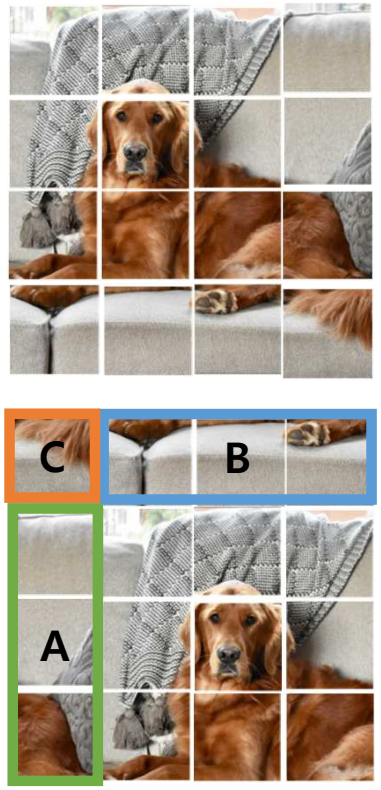
Swin Transformer Block



SW-MSA (Shifted Window Multi Self Attention)

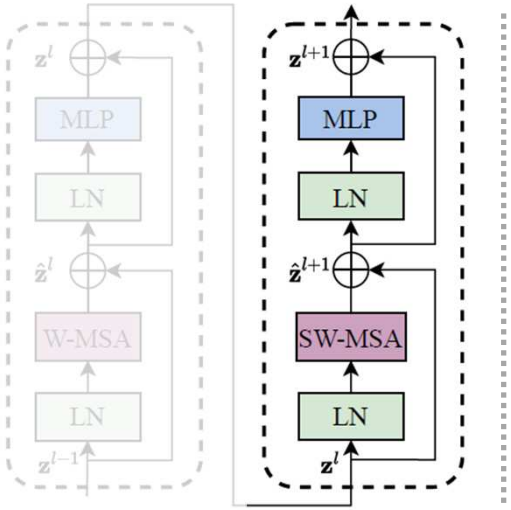
- The shifted window partitioning approach introduce connections between neighboring non-overlapping windows
- Efficient batch computation makes the computation efficient





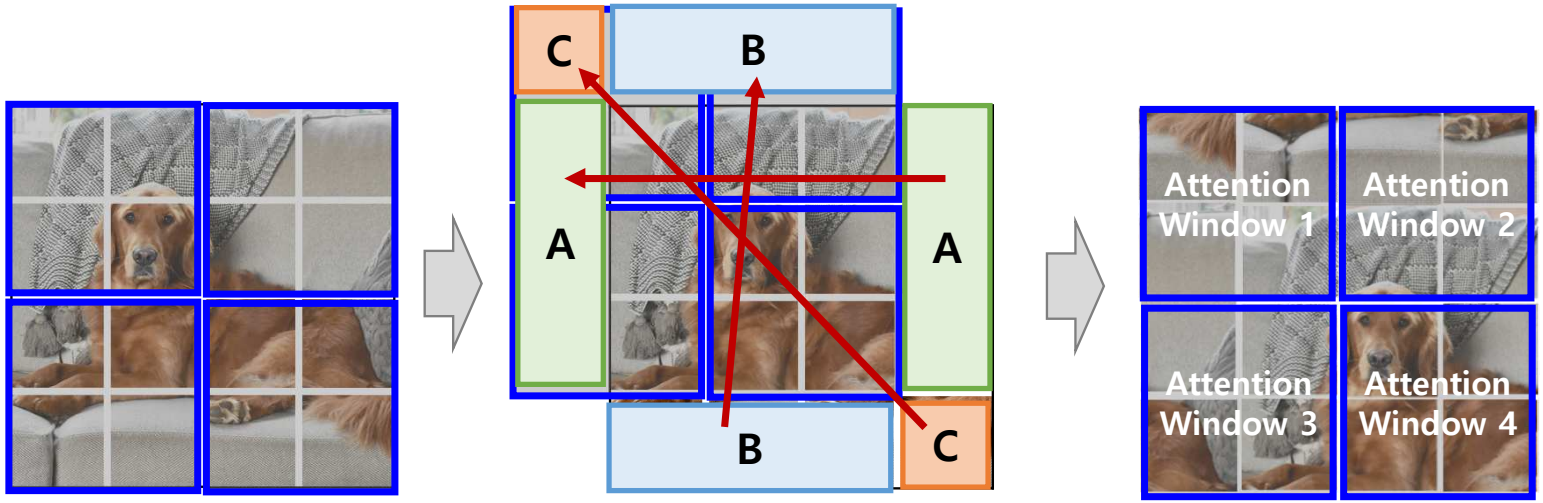
Swin Transformer

Swin Transformer Block



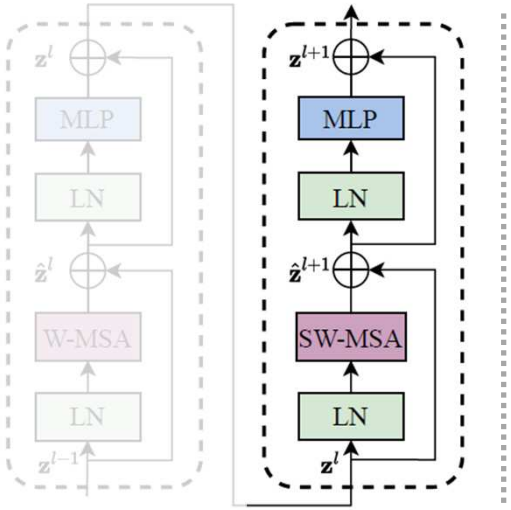
SW-MSA (Shifted Window Multi Self Attention)

- The shifted window partitioning approach introduce connections between neighboring non-overlapping windows
- Efficient batch computation makes the computation efficient



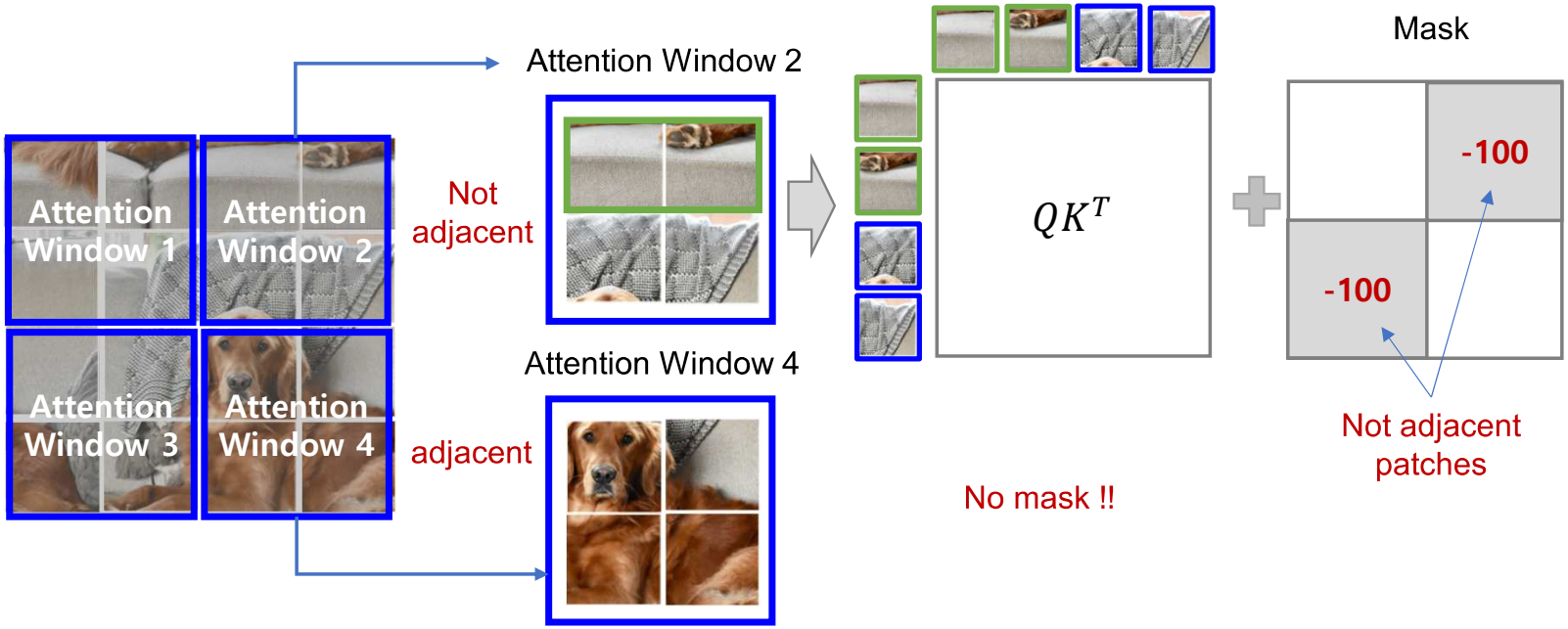
Swin Transformer

Swin Transformer Block



SW-MSA (Shifted Window Multi Self Attention) (con't)

- A window may be composed of several sub-windows that are not adjacent in feature map
- *Masking mechanism* is employed to limit self-attention computation only within adjacent feature map.



1 Swin Transformer

3.2 Shifted Window based Self-Attention

Standard Transformer architecture

- **Global self-attention** : Relationship between a token and all other tokens are computed
- Lead to quadratic complexity on the token number
- Unsuitable to dense prediction or high-resolution image

Self-attention in non-overlapped windows

- Suppose that each window contains $M \times M$ patches
- The computational complexity of a global MSA module and a window based one (W-MSA) on an image of $h \times w$ patches are

Quadratic to patch number *hw*

모든 token과의 attention 계산

$$\Omega(\text{MSA}) = 4hwC^2 + 2(hw)^2C, \quad (1)$$

$$\Omega(\text{W-MSA}) = 4hwC^2 + 2M^2hwC, \quad (2)$$

Linear when M is fixed (Default 7)

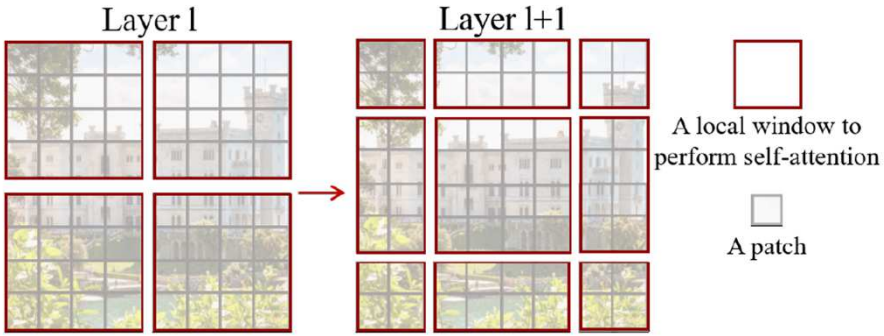
Window 내에서 self-attention

Shifted window partitioning in successive blocks

- Window-based self-attention : Lack connections across windows

→ Cross-window connections

→ **A shifted window partitioning approach** which alternates between two partitioning configurations in consecutive Swin Transformer blocks



For W-MSA

Regular window partition : 8x8 feature map that is partitioned into 2x2 windows of size 4x4 (M=4)

For SW-MSA

Shifted window partition of the preceding layer, by displacing the windows by $(\lfloor \frac{M}{2} \rfloor, \lfloor \frac{M}{2} \rfloor)$ pixels from the regular window partition.

3.2 Shifted Window based Self-Attention

Computational Complexity Comparison

Image Size: 64 x 64
Token Size: 4 x 4

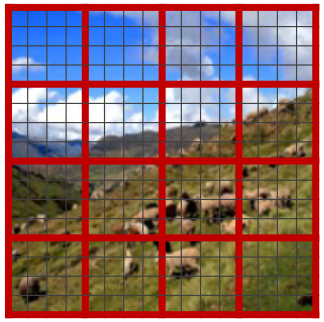
ViT

1 Self-Attention
16 x 16 tokens per SA

Computational Complexity of Self-Attention:
$O(n^2 d)$
$n: Sequence\ length$
$d: Embedding\ dimension$

$$O(n^2 d) \propto (16 \cdot 16)^2 d = 2^{16} d$$

Swin-Transformer



4 x 4 Self-Attention
4 x 4 tokens per SA

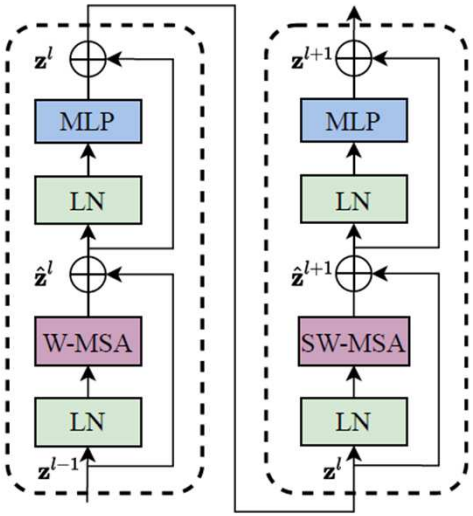
$$O(n^2 d) \cdot 4 \cdot 4 \propto (4 \cdot 4)^2 d \cdot \mathbf{4} \cdot \mathbf{4} = \mathbf{2^{12} d}$$

[참고] http://iislab.skku.edu/iish/seminar/51100

Swin Transformer

3.2 Shifted Window based Self-Attention

Shifted window partitioning in successive blocks



W-MSA, SW-MSA : Window based multi-head self-attention using regular and shifted window partitioning configurations.

- Compute consecutive Swin Transformer blocks

$$\begin{aligned}
\hat{\mathbf{z}}^l &= \text{W-MSA}\left(\text{LN}\left(\mathbf{z}^{l-1}\right)\right) + \mathbf{z}^{l-1}, \\
\mathbf{z}^l &= \text{MLP}\left(\text{LN}\left(\hat{\mathbf{z}}^l\right)\right) + \hat{\mathbf{z}}^l, \\
\hat{\mathbf{z}}^{l+1} &= \text{SW-MSA}\left(\text{LN}\left(\mathbf{z}^l\right)\right) + \mathbf{z}^l, \\
\mathbf{z}^{l+1} &= \text{MLP}\left(\text{LN}\left(\hat{\mathbf{z}}^{l+1}\right)\right) + \hat{\mathbf{z}}^{l+1}, \qquad (3)
\end{aligned}$$

Efficient batch computation for shifted configuration

- Cyclic-shifting toward the top-left direction

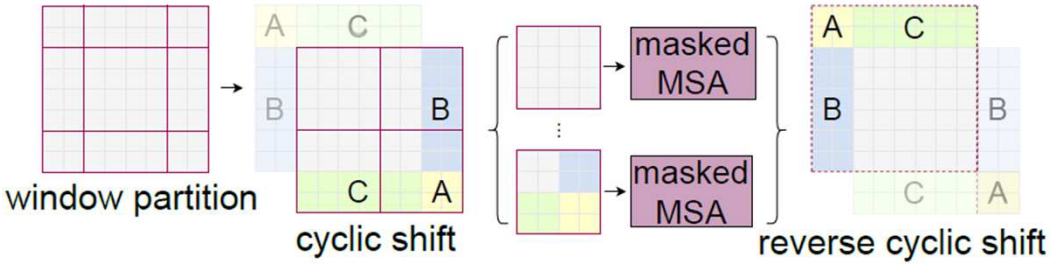


Figure 4. Illustration of an efficient batch computation approach for self-attention in shifted window partitioning.

- After this shift, a batched window may be composed of several sub-windows that are not adjacent in the feature map, so a **masking mechanism** is employed to **limit self-attention computation to within each sub-window**.
- With the cyclic-shift, the number of batched windows remains the same as that of regular window partitioning.

Swin Transformer

3.2 Shifted Window based Self-Attention

Relative position bias

- In computing self-attention, include a **relative position bias** $B \in \mathbb{R}^{M^2 \times M^2}$ to each head in computing similarity

$$\text{Attention}(Q, K, V) = \text{SoftMax}(QK^T/\sqrt{d} + B)V, \quad (4)$$

where $Q, K, V \in \mathbb{R}^{M^2 \times d}$ are the *query*, *key* and *value* matrices; d is the *query/key* dimension, and M^2 is the number of patches in a window. Since the relative position along each axis lies in the range $[-M+1, M-1]$, we parameterize a smaller-sized bias matrix $\hat{B} \in \mathbb{R}^{(2M-1)\times(2M-1)}$, and values in B are taken from $\hat{B}$.

- 기존에 position embedding은 절대좌표를 그냥 더해주었는데 본 논문에서는 상대좌표를 더해주는 것이 더 좋은 방법이라고 제시한다

❖ Ablation Study : Shifted windows, Relative position bias

	ImageNet		COCO		ADE20k
	top-1	top-5	AP^{box}	AP^{mask}	mIoU
w/o shifting	80.2	95.1	47.7	41.5	43.3
shifted windows	**81.3**	**95.6**	**50.5**	**43.7**	**46.1**
no pos.	80.1	94.9	49.2	42.6	43.8
abs. pos.	80.5	95.2	49.0	42.4	43.2
abs.+rel. pos.	81.3	95.6	50.2	43.4	44.0
rel. pos. w/o app.	79.3	94.7	48.2	41.9	44.1
rel. pos.	**81.3**	**95.6**	**50.5**	**43.7**	**46.1**

Table 4. Ablation study on the *shifted windows* approach and different position embedding methods on three benchmarks, using the Swin-T architecture. w/o shifting: all self-attention modules adopt regular window partitioning, without *shifting*; abs. pos.: absolute position embedding term of ViT; rel. pos.: the default settings with an additional relative position bias term (see Eq. (4)); app.: the first scaled dot-product term in Eq. (4).

- Adding absolute position embedding to the input as in [20] drops performance slightly.
- Relative position bias in pre-training can be also used to initialize a model for fine-tuning with a different window size through bi-cubic interpolation

3.2 Shifted Window based Self-Attention

❖ Ablation Study : Different self-attention methods

method	MSA in a stage (ms)				Arch. (FPS)		
	S1	S2	S3	S4	T	S	B
sliding window (naive)	122.5	38.3	12.1	7.6	183	109	77
sliding window (kernel)	7.6	4.7	2.7	1.8	488	283	187
Performer [14]	4.8	2.8	1.8	1.5	638	370	241
window (w/o shifting)	2.8	1.7	1.2	0.9	770	444	280
shifted window (padding)	3.3	2.3	1.9	2.2	670	371	236
shifted window (cyclic)	3.0	1.9	1.3	1.0	755	437	278

Table 5. Real speed of different self-attention computation methods and implementations on a V100 GPU.

	Backbone	ImageNet		COCO		ADE20k
		top-1	top-5	AP^{box}	AP^{mask}	mIoU
sliding window	Swin-T	81.4	95.6	50.2	43.5	45.8
Performer [14]	Swin-T	79.0	94.2	-	-	-
shifted window	Swin-T	81.3	95.6	50.5	43.7	46.1

Table 6. Accuracy of Swin Transformer using different methods for self-attention computation on three benchmarks.

1 Swin Transformer

3.3 Architecture Variants

- **Swin-B** : Base model, model size and computation complexity similar to ViT-B/DeiT-B

• Swin-T: $C = 96$, layer numbers = $\{2, 2, 6, 2\}$	0.25x	ResNet-50 (DeiT-S)
• Swin-S: $C = 96$, layer numbers = $\{2, 2, 18, 2\}$	0.5x	ResNet-101
• Swin-B: $C = 128$, layer numbers = $\{2, 2, 18, 2\}$	1.0x	
• Swin-L: $C = 192$, layer numbers = $\{2, 2, 18, 2\}$	2.0x	

✓ Window size : M=7 by default

✓ Query dimension of each head d=32, Expansion layer of each MLP α=4 for all experiments

✓ C : Channel number of the hidden layers in the first stage

1 Swin Transformer

Detailed Architecture

- Assume 224x224 image size
- “concat 4x4” : concatenation of nxn neighboring features in a patch. This operation results in a downsampling of the feature map by a rate of n.
- “96-d” : A linear layer with an output dimension of 96
- “win. sz. 7x7” : A multi-head self-attention module with window size of 7x7

	downsp. rate (output size)	Swin-T	Swin-S	Swin-B	Swin-L
stage 1	4× (56×56)	concat 4×4, 96-d, LN	concat 4×4, 96-d, LN	concat 4×4, 128-d, LN	concat 4×4, 192-d, LN
		[win. sz. 7×7, dim 96, head 3] × 2	[win. sz. 7×7, dim 96, head 3] × 2	[win. sz. 7×7, dim 128, head 4] × 2	[win. sz. 7×7, dim 192, head 6] × 2
stage 2	8× (28×28)	concat 2×2, 192-d , LN	concat 2×2, 192-d , LN	concat 2×2, 256-d , LN	concat 2×2, 384-d , LN
		[win. sz. 7×7, dim 192, head 6] × 2	[win. sz. 7×7, dim 192, head 6] × 2	[win. sz. 7×7, dim 256, head 8] × 2	[win. sz. 7×7, dim 384, head 12] × 2
stage 3	16× (14×14)	concat 2×2, 384-d , LN	concat 2×2, 384-d , LN	concat 2×2, 512-d , LN	concat 2×2, 768-d , LN
		[win. sz. 7×7, dim 384, head 12] × 6	[win. sz. 7×7, dim 384, head 12] × 18	[win. sz. 7×7, dim 512, head 16] × 18	[win. sz. 7×7, dim 768, head 24] × 18
stage 4	32× (7×7)	concat 2×2, 768-d , LN	concat 2×2, 768-d , LN	concat 2×2, 1024-d , LN	concat 2×2, 1536-d , LN
		[win. sz. 7×7, dim 768, head 24] × 2	[win. sz. 7×7, dim 768, head 24] × 2	[win. sz. 7×7, dim 1024, head 32] × 2	[win. sz. 7×7, dim 1536, head 48] × 2

Table 7. Detailed architecture specifications.

1 Swin Transformer

4. Experiments : image Classification on ImageNet-1K

➢ **Regular ImageNet-1K training**

- 1.28M training images, 50K validation images from 1,000 classes
- AdamW optimizer for 300 epochs using a cosine decay learning rate scheduler and 20 epochs of linear warm-up.
- A batch size of 1024, an initial learning rate of 0.001, and a weight decay of 0.05
- the augmentation and regularization strategies of [63] in training

➢ **Pre-training on ImageNet-22K and fine-tuning on ImageNet-1K**

1) Pre-train on the larger ImageNet-22K dataset

- 14.2 million images and 22K classes, AdamW optimizer for 90 epochs using a linear decay learning rate scheduler with a 5-epoch linear warm-up
- A batch size of 4096, an initial learning rate of 0.001, and a weight decay of 0.01

2) ImageNet-1K fine-tuning

- 30 epochs with a batch size of 1024
- a constant learning rate of 10^{-5}, and a weight decay of 10^{-8}.

(a) Regular ImageNet-1K trained models

method	image size	#param.	FLOPs	throughput (image / s)	ImageNet top-1 acc.
RegNetY-4G [48]	224^2	21M	4.0G	1156.7	80.0
RegNetY-8G [48]	224^2	39M	8.0G	591.6	81.7
RegNetY-16G [48]	224^2	84M	16.0G	334.7	82.9
EffNet-B3 [58]	300^2	12M	1.8G	732.1	81.6
EffNet-B4 [58]	380^2	19M	4.2G	349.4	82.9
EffNet-B5 [58]	456^2	30M	9.9G	169.1	83.6
EffNet-B6 [58]	528^2	43M	19.0G	96.9	84.0
EffNet-B7 [58]	600^2	66M	37.0G	55.1	84.3
ViT-B/16 [20]	384^2	86M	55.4G	85.9	77.9
ViT-L/16 [20]	384^2	307M	190.7G	27.3	76.5
DeiT-S [63]	224^2	22M	4.6G	940.4	79.8
DeiT-B [63]	224^2	86M	17.5G	292.3	81.8
DeiT-B [63]	384^2	86M	55.4G	85.9	83.1
Swin-T	224^2	29M	4.5G	755.2	81.3
Swin-S	224^2	50M	8.7G	436.9	83.0
Swin-B	224^2	88M	15.4G	278.1	83.5
Swin-B	384^2	88M	47.0G	84.7	84.5

(b) ImageNet-22K pre-trained models

method	image size	#param.	FLOPs	throughput (image / s)	ImageNet top-1 acc.
R-101x3 [38]	384^2	388M	204.6G	-	84.4
R-152x4 [38]	480^2	937M	840.5G	-	85.4
ViT-B/16 [20]	384^2	86M	55.4G	85.9	84.0
ViT-L/16 [20]	384^2	307M	190.7G	27.3	85.2
Swin-B	224^2	88M	15.4G	278.1	85.2
Swin-B	384^2	88M	47.0G	84.7	86.4
Swin-L	384^2	197M	103.9G	42.1	87.3

Table 1. Comparison of different backbones on ImageNet-1K classification. Throughput is measured using the GitHub repository of [68] and a V100 GPU, following [63].

Swin Transformer

4. Experiments : Object Detection on COCO

- COCO2017; 118K training, 5K validation and 20K test-dev images
- Typical object detection frameworks : Cascade Mask R-CNN [29, 6], ATSS [79], RepPoints v2 [12], and Sparse RCNN [56]

- Same settings: **multi-scale training** (resizing the input such that the shorter side is between 480 and 800 while the longer side is at most 1333), **AdamW** optimizer (initial learning rate of 0.0001, weight decay of 0.05, and batch size of 16), and 3x schedule (36 epochs).

(a) Various frameworks

Method	Backbone	AP^{box}	AP^{box}_{50}	AP^{box}_{75}	#param.	FLOPs	FPS
Cascade Mask R-CNN	R-50	46.3	64.3	50.5	82M	739G	18.0
	Swin-T	**50.5**	**69.3**	**54.9**	86M	745G	15.3
ATSS	R-50	43.5	61.9	47.0	32M	205G	28.3
	Swin-T	**47.2**	**66.5**	**51.3**	36M	215G	22.3
RepPointsV2	R-50	46.5	64.6	50.3	42M	274G	13.6
	Swin-T	**50.0**	**68.5**	**54.2**	45M	283G	12.0
Sparse R-CNN	R-50	44.5	63.4	48.2	106M	166G	21.0
	Swin-T	**47.9**	**67.3**	**52.3**	110M	172G	18.4

Swin Transformer vs RestNet-50 on 4 object detection frameworks

(b) Various backbones w. Cascade Mask R-CNN

	AP^{box}	AP^{box}_{50}	AP^{box}_{75}	AP^{mask}	AP^{mask}_{50}	AP^{mask}_{75}	param	FLOPs	FPS
DeiT-S†	48.0	67.2	51.7	41.4	64.2	44.3	80M	889G	10.4
R50	46.3	64.3	50.5	40.1	61.7	43.4	82M	739G	18.0
Swin-T	**50.5**	**69.3**	**54.9**	**43.7**	**66.6**	**47.1**	86M	745G	15.3
X101-32	48.1	66.5	52.4	41.6	63.9	45.2	101M	819G	12.8
Swin-S	**51.8**	**70.4**	**56.3**	**44.7**	**67.9**	**48.5**	107M	838G	12.0
X101-64	48.3	66.4	52.3	41.7	64.0	45.1	140M	972G	10.4
Swin-B	**51.9**	**70.9**	**56.5**	**45.0**	**68.4**	**48.7**	145M	982G	11.6

Swin Transformer vs RestNe(X)t, DeiT on Cascade Mask R-CNN

(c) System-level Comparison

Method	mini-val AP^{box}	mini-val AP^{mask}	test-dev AP^{box}	test-dev AP^{mask}	#param.	FLOPs
RepPointsV2* [12]	-	-	52.1	-	-	-
GCNet* [7]	51.8	44.7	52.3	45.4	-	1041G
RelationNet++* [13]	-	-	52.7	-	-	-
SpineNet-190 [21]	52.6	-	52.8	-	164M	1885G
ResNeSt-200* [78]	52.5	-	53.3	47.1	-	-
EfficientDet-D7 [59]	54.4	-	55.1	-	77M	410G
DetectoRS* [46]	-	-	55.7	48.5	-	-
YOLOv4 P7* [4]	-	-	55.8	-	-	-
Copy-paste [26]	55.9	47.2	56.0	47.4	185M	1440G
X101-64 (HTC++)	52.3	46.0	-	-	155M	1033G
Swin-B (HTC++)	56.4	49.1	-	-	160M	1043G
Swin-L (HTC++)	57.1	49.5	57.7	50.2	284M	1470G
Swin-L (HTC++)*	**58.0**	**50.4**	**58.7**	**51.1**	284M	-

Table 2. Results on COCO object detection and instance segmentation. †denotes that additional decovolution layers are used to produce hierarchical feature maps. * indicates multi-scale testing.

Adopt **improved HTC** [9] (denoted as HTC++) with **instaboost** [22], **stronger multi-scale training** [7], **6x schedule** (72 epochs), **soft-NMS** [5], and **ImageNet-22K pre-trained model as initialization.**

Swin Transformer

4. Experiments : Semantic Segmentation on ADE20K

- 150 semantic categories, 25K images in total (20K for training, 2K for validation, 3K for testing)
- Utilize UperNet in mmseg as our base framework

ADE20K Method	Backbone	val mIoU	test score	#param.	FLOPs	FPS
DANet [23]	ResNet-101	45.2	-	69M	1119G	15.2
DLab.v3+ [11]	ResNet-101	44.1	-	63M	1021G	16.0
ACNet [24]	ResNet-101	45.9	38.5	-		
DNL [71]	ResNet-101	46.0	56.2	69M	1249G	14.8
OCRNet [73]	ResNet-101	45.3	56.0	56M	923G	19.3
UperNet [69]	ResNet-101	44.9	-	86M	1029G	20.1
OCRNet [73]	HRNet-w48	45.7	-	71M	664G	12.5
DLab.v3+ [11]	ResNeSt-101	46.9	55.1	66M	1051G	11.9
DLab.v3+ [11]	ResNeSt-200	48.4	-	88M	1381G	8.1
SETR [81]	T-Large‡	50.3	61.7	308M	-	-
UperNet	DeiT-S†	44.0	-	52M	1099G	16.2
UperNet	Swin-T	46.1	-	60M	945G	18.5
UperNet	Swin-S	49.3	-	81M	1038G	15.2
UperNet	Swin-B‡	51.6	-	121M	1841G	8.7
UperNet	Swin-L‡	**53.5**	**62.8**	234M	3230G	6.2

Table 3. Results of semantic segmentation on the ADE20K val and test set. † indicates additional deconvolution layers are used to produce hierarchical feature maps. ‡ indicates that the model is pre-trained on ImageNet-22K.

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

Swin Transformer : Model Implementation Code

- PatchEmbed module -> BasicLayer modules(nn.ModuleList) -> norm&avgpool -> nn.Linear

```
class SwinTransformer(nn.Module):
    def __init__(self, img_size=224, patch_size=4, in_chans=3, num_classes=1000,
                 embed_dim=96, depths=[2, 2, 6, 2], num_heads=[3, 6, 12, 24],
                 window_size=7, mlp_ratio=4., qkv_bias=True, qk_scale=None,
                 drop_rate=0., attn_drop_rate=0., drop_path_rate=0.1,
                 norm_layer=nn.LayerNorm, ape=False, patch_norm=True,
                 use_checkpoint=False, **kwargs):
        super().__init__()

        # ...

        self.patch_embed = PatchEmbed( img_size=img_size,
                    patch_size=patch_size, in_chans=in_chans,
                    embed_dim=embed_dim, norm_layer=norm_layer if self.patch_norm
else None)

        # ...
```

```
 self.layers = nn.ModuleList()
        for i_layer in range(self.num_layers):
            layer = BasicLayer(dim=int(embed_dim * 2 ** i_layer),
                               input_resolution=(patches_resolution[0] // (2 ** i_layer),
                                                 patches_resolution[1] // (2 ** i_layer)),
                               depth=depths[i_layer],
                               num_heads=num_heads[i_layer],
                               window_size=window_size,
                               mlp_ratio=self.mlp_ratio,
                               qkv_bias=qkv_bias, qk_scale=qk_scale,
                               drop=drop_rate, attn_drop=attn_drop_rate,
                               drop_path=dpr[sum(depths[:i_layer]):sum(depths[:i_layer + 1])],
                               norm_layer=norm_layer,
                               downsample=PatchMerging if (i_layer < self.num_layers - 1)
else None, use_checkpoint=use_checkpoint)
            self.layers.append(layer)

        self.norm = norm_layer(self.num_features)
        self.avgpool = nn.AdaptiveAvgPool1d(1)
        self.head = nn.Linear(self.num_features, num_classes) if num_classes > 0
else nn.Identity()

        # ...
# ...
```

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

Swin Transformer : Model Implementation Code

- PatchEmbed module -> BasicLayer modules(nn.ModuleList) -> norm&avgpool -> nn.Linear

```
def forward_features(self, x):
        x = self.patch_embed(x)
        if self.ape:
            x = x + self.absolute_pos_embed
        x = self.pos_drop(x)

        for layer in self.layers:
            x = layer(x)

        x = self.norm(x)  # B L C
        x = self.avgpool(x.transpose(1, 2))  # B C 1
        x = torch.flatten(x, 1)
        return x

    def forward(self, x):
        x = self.forward_features(x)
        x = self.head(x)
        return x

    # ...
```

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

PatchEmbed (Patch Partition + Linear Embedding)

- 이미지를 패치로 나누고 각 패치를 Embedding하는 module
- Convolution layer(코드에서 self.proj = nn.Conv2d(..) 부분)에 kernel size와 stride를 모두 patch size 만큼 주기 때문에, patch partition과 embedidng을 동시에 진행할 수 있다.
- (B, 3, 224, 224) -> self.proj(x).flatten(2).transpose(1, 2) -> (B, 56x56, 96)

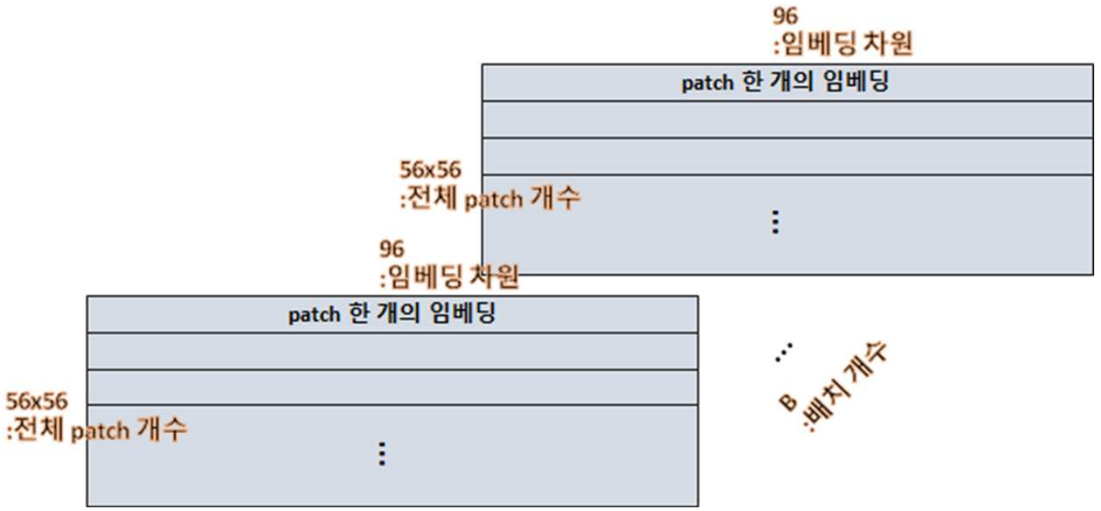
```
class PatchEmbed(nn.Module):
    r""" Image to Patch Embedding
    Args:
        img_size (int): Image size.  Default: 224.
        patch_size (int): Patch token size. Default: 4.
        in_chans (int): Number of input image channels. Default: 3.
        embed_dim (int): Number of linear projection output channels. Default: 96.
        norm_layer (nn.Module, optional): Normalization layer. Default: None
    """

    def __init__(self, img_size=224, patch_size=4, in_chans=3, embed_dim=96,
norm_layer=None):
        super().__init__()
        img_size = to_2tuple(img_size)
        patch_size = to_2tuple(patch_size)
        patches_resolution = [img_size[0] // patch_size[0], img_size[1] //
patch_size[1]]
        self.img_size = img_size
        self.patch_size = patch_size
        self.patches_resolution = patches_resolution
        self.num_patches = patches_resolution[0] * patches_resolution[1]

        self.in_chans = in_chans
        self.embed_dim = embed_dim

        self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size,
stride=patch_size)
        if norm_layer is not None:
            self.norm = norm_layer(embed_dim)
        else:
            self.norm = None

    def forward(self, x):
        B, C, H, W = x.shape
        # FIXME look at relaxing size constraints
        assert H == self.img_size[0] and W == self.img_size[1], \
            f"Input image size ({H}*{W}) doesn't match model
({self.img_size[0]}*{self.img_size[1]})."
        x = self.proj(x).flatten(2).transpose(1, 2)  # B Ph*Pw C
        if self.norm is not None:
            x = self.norm(x)
        return x

    # ...
```

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

PatchEmbed (Patch Partition + Linear Embedding)

- (B, 3, 224, 224) -> self.proj(x).flatten(2).transpose(1, 2) -> (B, 56x56, 96)
- 처음 x는 (B, 3, 224, 224) 였지만 self.proj(x)를 통해 (B, 96, 56, 56), .flatten(2)를 통해 (B, 96, 56x56), .transpose(1, 2)를 통해 (B, 56x56, 96)이 되었다.



1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

Swin Transformer Block – Basic Layer

- BasicLayer Module의 코드 부분
- depth개의 SwinTransformerBlock과 downsample로 구성되어 있다. 홀수번째 SwinTransforemrBlock은 shift_size를 window_size//2로 설정되며, downsample은 뒤에 설명하는 Patch Merging 과정이다.

```
class BasicLayer(nn.Module):
    """ A basic Swin Transformer layer for one stage.
    Args:
        dim (int): Number of input channels.
        input_resolution (tuple[int]): Input resolution.
        depth (int): Number of blocks.
        num_heads (int): Number of attention heads.
        # ...
    """

    def __init__(self, dim, input_resolution, depth, num_heads, window_size,
                 mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0.,
                 drop_path=0., norm_layer=nn.LayerNorm, downsample=None,
use_checkpoint=False):

        super().__init__()
        # ...
```

```
        # build blocks
        self.blocks = nn.ModuleList([
            SwinTransformerBlock(dim=dim, input_resolution=input_resolution,
                                 num_heads=num_heads, window_size=window_size,
                                 shift_size=0 if (i % 2 == 0) else window_size // 2,
                                 mlp_ratio=mlp_ratio,
                                 qkv_bias=qkv_bias, qk_scale=qk_scale,
                                 drop=drop, attn_drop=attn_drop,
                                 drop_path=drop_path[i] if isinstance(drop_path, list)
                                       else drop_path, norm_layer=norm_layer)
            for i in range(depth)])

        # patch merging layer
        if downsample is not None:
            self.downsample = downsample(input_resolution, dim=dim,
norm_layer=norm_layer)
        else:
            self.downsample = None

    def forward(self, x):
        for blk in self.blocks:
            if self.use_checkpoint:
                x = checkpoint.checkpoint(blk, x)
            else:
                x = blk(x)
        if self.downsample is not None:
            x = self.downsample(x)
        return x

    # ...
```

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

Swin Transformer Block

```
class SwinTransformerBlock(nn.Module):
    r""" Swin Transformer Block.
    Args:
        dim (int): Number of input channels.
        input_resolution (tuple[int]): Input resulotion.
        num_heads (int): Number of attention heads.
        window_size (int): Window size.
        shift_size (int): Shift size for SW-MSA.
        # ...
    """
    def __init__(self, dim, input_resolution, num_heads, window_size=7,
        shift_size=0, mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0.,
        attn_drop=0., drop_path=0., act_layer=nn.GELU, norm_layer=nn.LayerNorm):
        super().__init__()
        self.dim = dim
        self.input_resolution = input_resolution
        self.num_heads = num_heads
        self.window_size = window_size
        self.shift_size = shift_size
        self.mlp_ratio = mlp_ratio
    # ...
        if min(self.input_resolution) <= self.window_size:
            # if window size is larger than input resolution, we don't partition windows
            self.shift_size = 0
            self.window_size = min(self.input_resolution)
        assert 0 <= self.shift_size < self.window_size, "shift_size must in 0-
window_size"
```

```
        self.norm1 = norm_layer(dim)
        self.attn = WindowAttention(
            dim, window_size=to_2tuple(self.window_size), num_heads=num_heads,
            qkv_bias=qkv_bias, qk_scale=qk_scale, attn_drop=attn_drop,
proj_drop=drop)

        self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.Identity()
        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim,
act_layer=act_layer, drop=drop)

        if self.shift_size > 0:
            # calculate attention mask for SW-MSA
            H, W = self.input_resolution
            img_mask = torch.zeros((1, H, W, 1))  # 1 H W 1
            h_slices = (slice(0, -self.window_size),
                        slice(-self.window_size, -self.shift_size),
                        slice(-self.shift_size, None))
            w_slices = (slice(0, -self.window_size),
                        slice(-self.window_size, -self.shift_size),
                        slice(-self.shift_size, None))
            cnt = 0
            for h in h_slices:
                for w in w_slices:
                    img_mask[:, h, w, :] = cnt
                    cnt += 1

            mask_windows = window_partition(img_mask, self.window_size)  # nW,
window_size, window_size, 1
            mask_windows = mask_windows.view(-1, self.window_size *
self.window_size)
```

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

Swin Transformer Block

```
            attn_mask = mask_windows.unsqueeze(1) - mask_windows.unsqueeze(2)
            attn_mask = attn_mask.masked_fill(attn_mask != 0, float(-
100.0)).masked_fill(attn_mask == 0, float(0.0))
        else:
            attn_mask = None
            self.register_buffer("attn_mask", attn_mask)

    def forward(self, x):
        H, W = self.input_resolution
        B, L, C = x.shape
        assert L == H * W, "input feature has wrong size"

        shortcut = x
        x = self.norm1(x)
        x = x.view(B, H, W, C)

        # cyclic shift
        if self.shift_size > 0:
            shifted_x = torch.roll(x, shifts=(-self.shift_size, -self.shift_size), dims=(1, 2))
        else:
            shifted_x = x

        # partition windows
        x_windows = window_partition(shifted_x, self.window_size)  # nW*B,
window_size, window_size, C
        x_windows = x_windows.view(-1, self.window_size * self.window_size, C)  #
nW*B, window_size*window_size, C
```

- forward method의 input x는 현재(B, 56x56, 96) size를 갖고 있다. 이는 .view를 통해 (B, 56, 56, 96)으로 바뀐다. 이 x는 각 윈도우에 self-attention을 적용 하기위해 windows_partition 함수를 수행하게 된다.
- (B, 56x56, 96) -> (B, 56, 56, 96) -> windows_partition

```
        # merge windows
        attn_windows = attn_windows.view(-1, self.window_size, self.window_size, C)
        shifted_x = window_reverse(attn_windows, self.window_size, H, W)  # B H'
W' C

        # reverse cyclic shift
        if self.shift_size > 0:
            x = torch.roll(shifted_x, shifts=(self.shift_size, self.shift_size), dims=(1, 2))
        else:
            x = shifted_x
        x = x.view(B, H * W, C)

        # FFN
        x = shortcut + self.drop_path(x)
        x = x + self.drop_path(self.mlp(self.norm2(x)))

        return x

    # ...
```

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

Swin Transformer Block

- BasicLayer Module의 코드 부분

```
        # merge windows
        attn_windows = attn_windows.view(-1, self.window_size, self.window_size, C)
        shifted_x = window_reverse(attn_windows, self.window_size, H, W)  # B H'
W' C

        # reverse cyclic shift
        if self.shift_size > 0:
            x = torch.roll(shifted_x, shifts=(self.shift_size, self.shift_size), dims=(1, 2))
        else:
            x = shifted_x
        x = x.view(B, H * W, C)

        # FFN
        x = shortcut + self.drop_path(x)
        x = x + self.drop_path(self.mlp(self.norm2(x)))

        return x

    # ...
```

```
        else:
            attn_mask = None
            self.register_buffer("attn_mask", attn_mask)

    def forward(self, x):
        H, W = self.input_resolution
        B, L, C = x.shape
        assert L == H * W, "input feature has wrong size"

k=self.attn_mask)  # nW*B, window_size*window_size, C
```

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

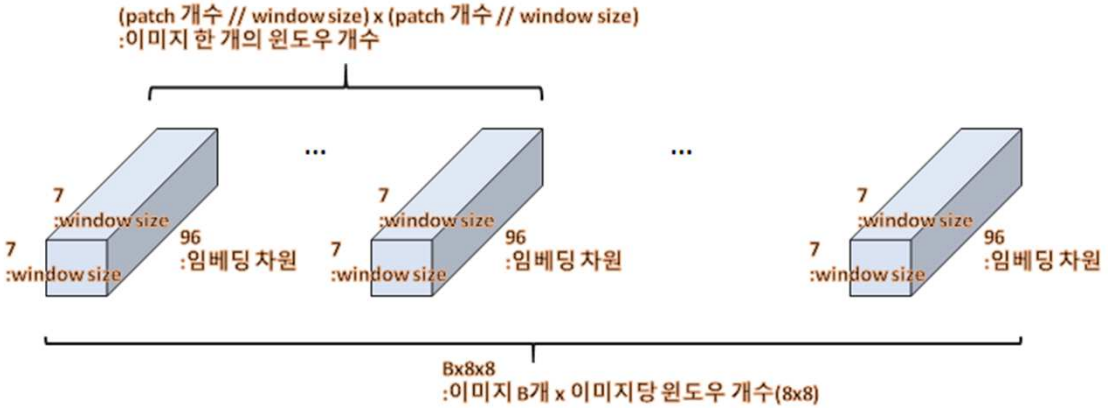
Window partition

- (B, 56x56, 96) -> (B, 56, 56, 96) -> windows_partition

```
def window_partition(x, window_size):
    """
    Args:
        x: (B, H, W, C)
        window_size (int): window size
    Returns:
        windows: (num_windows*B, window_size, window_size, C)
    """
    B, H, W, C = x.shape
    x = x.view(B, H // window_size, window_size, W // window_size, window_size, C)
    windows = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(-1, window_size,
window_size, C)
    return windows
```

- window_partition은 정해진 window_size(논문/코드에서는 7로 고정)로 패치들을 나눈 후, tensor을 윈도우 기준으로 생성 해준다.
- 우선 x = x.view(B, H // window_size, window_size, W // window_size, window_size, C) 부분을 통해 x를 window_size로 나누어 새로운 차원을 생성해준다. 현재 x의 shape은 (B, 8, 7, 8, 7, 96) 이다. 그 후 permute을 통해 (B, 8, 8, 7, 7, 96), .view을 통해 (Bx8x8, 7, 7, 96) shape이 된다.

- (B, 56, 56, 96) -> (B, 8, 7, 8, 7, 96) -> (B, 8, 8, 7, 7, 96) -> (Bx8x8, 7, 7, 96)



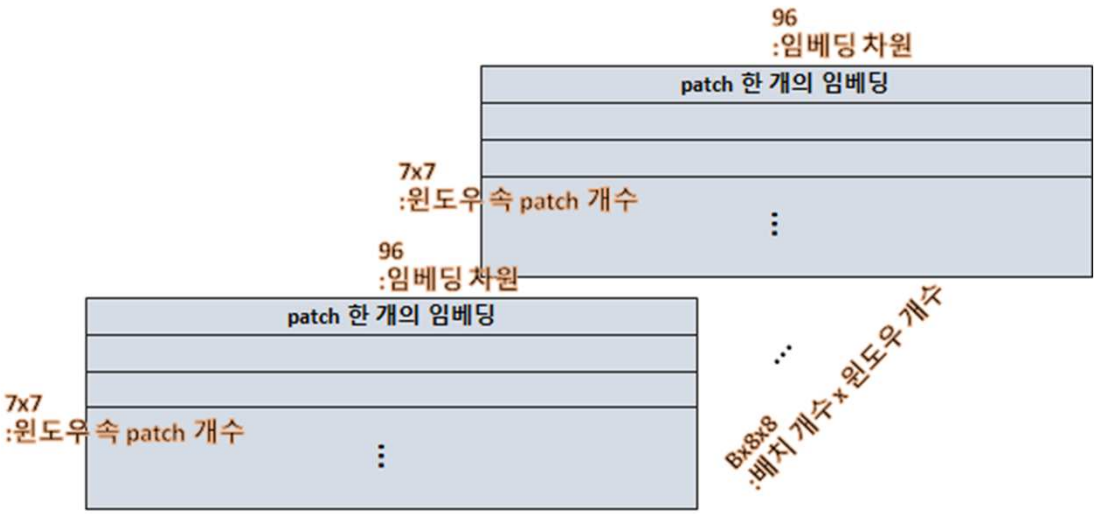
- 다시 위 SwinTransformerBlock으로 돌아가서 (Bx8x8, 7, 7, 96) tensor 속 각 윈도우는 self-attention 처리를 위해 이차원 tensor가 되어야한다. 즉 (Bx8x8, 7x7, 96)이 되어 window 내에 있는 패치끼리 self-attention이 이루어진다.
- window partition 함수에서 다른 이미지의 window지만 Bx8x8로 묶은 이유는 계산 편의성에 있다. window 내부 패치들끼리만 이루어지는 계산이기 때문에 다른 이미지 window와 독립적이다.

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

Window partition

- self-attention을 통해 나온 output은 self-attention 특성상 input과 동일한 shape을 갖게 된다. 그리고나서 window_reverse 함수를 통해 기존 shape (B, 56x56, 96)으로 복구가 된다.



- (Bx8x8, 7, 7, 96) -> (Bx8x8, 7x7, 96) -> self-attention -> (Bx8x8, 7x7, 96) -> (B, 56x56, 96)

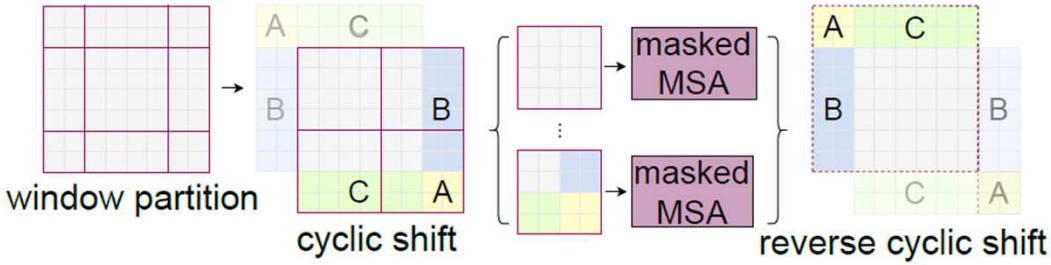


Figure 4. Illustration of an efficient batch computation approach for self-attention in shifted window partitioning.

- shift된 window를 사용할 때에도 고려 해야한다. 위 figure 4를 보면 알 수 있듯이, 논문에서는 window를 그대로 shift 한 후, 겹치지 않은 부분(A, B, C)를 빈 공간에 넣어주는 방법이 고안되었다.

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

Window partition

- 논문에서는 window를 그대로 shift 한 후, 겹치지 않은 부분(A, B, C)를 빈 공간에 넣어주는 방법이 고안되었다.
- 이는 torch.roll 함수를 통해 간단히 해결된다. torch.roll은 딱 저 기능을 처리해준다.

```
#torch.roll examples
>>> x = torch.tensor(
    [
        [1, 2, 3, 4],
        [5, 6, 7, 8]
    ]
)

>>> x = torch.roll(x, 1)
>>> x
tensor(
    [8, 1, 2, 3],
    [4, 5, 6, 7]
)
```

- torch.roll을 통해 patch sfhit 후, window_partion -> self-attention -> window_reverse 진행한다.
- 그리고 다시 반대 방향으로 torch.roll을 통해 patch를 원상복귀하면 된다(위 코드에서 두개의 torch.roll 부분).

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

Patch merging

- Hierarchical 한 특성을 위해 한 인풋에 존재하는 Patch의 개수를 점점 줄이며 학습 한다. 이를 통해 이미지의 작은 물체부터 큰 물체까지 모든 정보가 학습에 사용하게된다.

```
class PatchMerging(nn.Module):
    r""" Patch Merging Layer.
    Args:
        input_resolution (tuple[int]): Resolution of input feature.
        dim (int): Number of input channels.
        norm_layer (nn.Module, optional): Normalization layer.  Default:
nn.LayerNorm
    """

    def __init__(self, input_resolution, dim, norm_layer=nn.LayerNorm):
        super().__init__()
        # ...

    def forward(self, x):
        """
        x: B, H*W, C
        """
        H, W = self.input_resolution
        B, L, C = x.shape
        assert L == H * W, "input feature has wrong size"
        assert H % 2 == 0 and W % 2 == 0, f"x size ({H}*{W}) are not even."

        x = x.view(B, H, W, C)
```

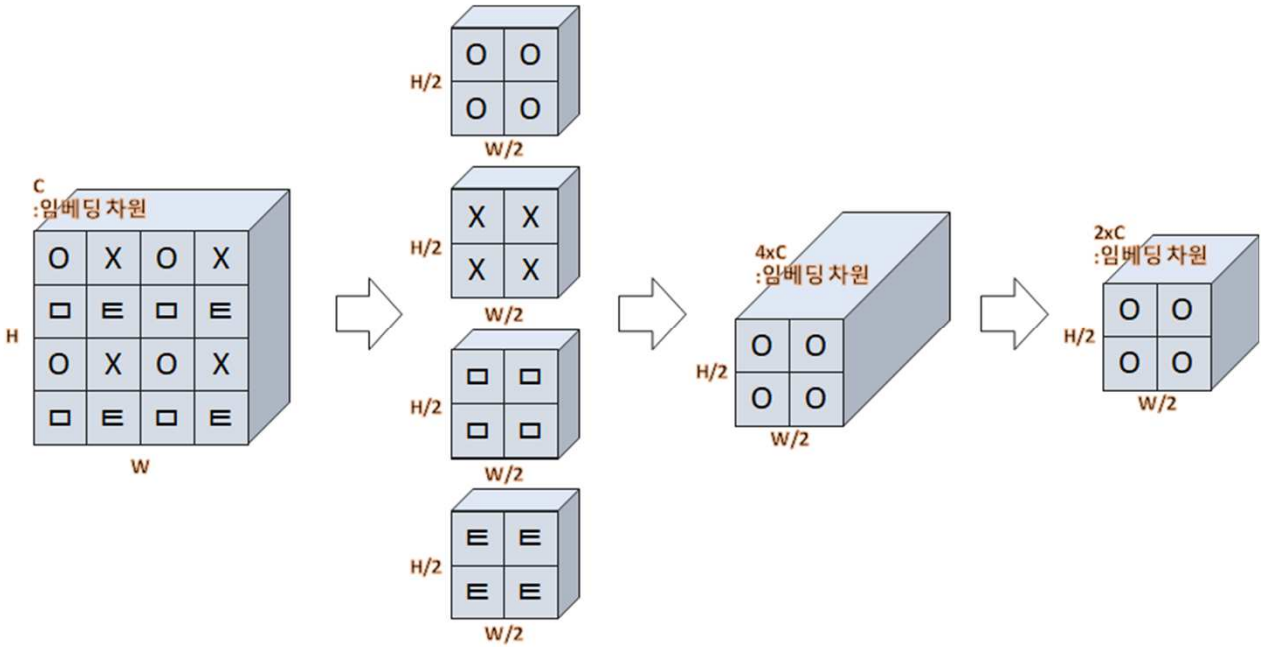
```
        x0 = x[:, 0::2, 0::2, :]  # B H/2 W/2 C
        x1 = x[:, 1::2, 0::2, :]  # B H/2 W/2 C
        x2 = x[:, 0::2, 1::2, :]  # B H/2 W/2 C
        x3 = x[:, 1::2, 1::2, :]  # B H/2 W/2 C
        x = torch.cat([x0, x1, x2, x3], -1)  # B H/2 W/2 4*C
        x = x.view(B, -1, 4 * C)  # B H/2*W/2 4*C

        x = self.norm(x)
        x = self.reduction(x)

        return x

    # ...
```

1 Swin Transformer - Pytorch

[출처] https://byeongjo-kim.tistory.com/36
[Official] https://github.com/microsoft/Swin-Transformer

Patch merging

- 인풋 x에 대해 width, height를 각각 한칸 씩 띄어서 새로운 이미지(x0, x1, x2, x3)를 만들고, 임베딩 차원을 기준으로 concat한다. 그리고 self.reduction을 통해 임베딩 차원을 조정해준다.



- (B, 56x56, 96) -> (B, 56, 56, 96) -> (B, 56/2, 56/2, 96x4) -> (B, 56/2, 56/2, 96x2)

- 이렇게 patch merging(downsample)이 완료된 tensor은 다음 BasicLayer의 input으로 사용된다. 모든 BasicLayer를 거친 tensor은 norm/avgpool/flatten/nn.Linear을 통해 class 개수에 맞춘 tensor가 되어 loss가 계산된다.