Swin Transformer

PT

1. Introduction
2. Architecture
3. Compare
4. Experiments

Swin transformer에 대한 발표를 시작하겠습니다.
pt 구성은 introduction에서 swin의 배경에 대해 말씀드리고, Architecture에서 모델의 구성과 방법에 대해 말씀드리겠습니다. 이후 Compare에서 ViT와 Swin의 연산량을 비교해볼 것이고 마지막으로 실험 결과에 대해 말씀드리겠습니다.

Introduction

- Swin Transformer: Hierarchical Vision Transformer using Shited Windows
  - 2021년 ICCV에 Accept된 논문
  - Microsoft에서 Swin이라고 불리는 모델을 제안한 논문
  - **Swin:** Swin Transformer

Swin Transformer: Hierarchical Vision Transformer using Shifted Windows

Ze Liu[†] Yutong Lin[†*] Yue Cao[*] Han Hu[*‡] Yixuan Wei[†]
Zheng Zhang Stephen Lin Baining Guo
Microsoft Research Asia
{v-zeliu1,v-yutlin,yuecao,hanhu,v-yixwe,zhez,stevelin,bainguo}@microsoft.com

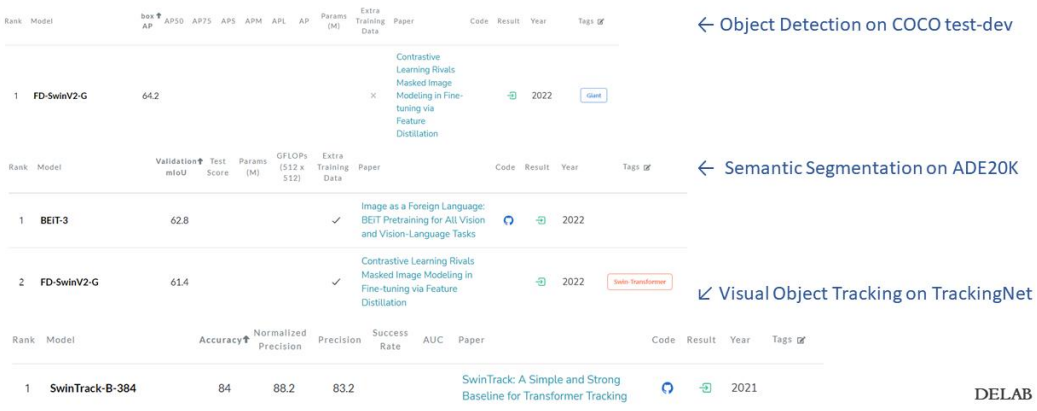
2021 ICCV OCTOBER 11-17 VIRTUAL

먼저 이 논문은 2021년 ICCV에 Accept된 논문이고 Microsoft에서 Swin이라고 불리는 모델을 제안한 논문입니다. Swin은 Swin Transformer에 해당합니다. 그리고 논문의 제목에서도 보다 싶이

ViT를 계층적으로 쌓았다는 것을 짐작해볼 수 있습니다.

Introduction

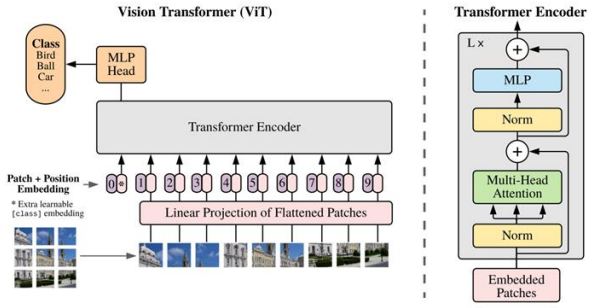
- PapersWithCode
  - 컴퓨터 비전의 다양한 분야에서 Swin을 활용한 모델이 SOTA를 기록함
  - Scene Understanding: 객체 탐지, 영상 분할, 객체 추적 등



제가 이 논문을 선정하게 된 배경은, PapersWithCode로 제가 관심이 있는 Scene Understanding 분야를 찾아봤는데, 거의 SOTA아니면 2등을 찍고 있길래 어떤 기법을 사용했는지 궁금해서 찾아보게 되었습니다. Scene Understading이라는 것은 컴퓨터가 영상을 보고 어떤 상황인지 이해를 하는 작업으로, 객체가 어디에 있는지 찾는 Object Detection이나, 객체 영역과 배경 영역을 분할하는 segmentation(배경과 객체의 영역을 찾는 작업), 이전에 선택한 객체가 현재 어느 위치에 있는지 찾는 tracking 등을 예시로 들 수 있습니다.

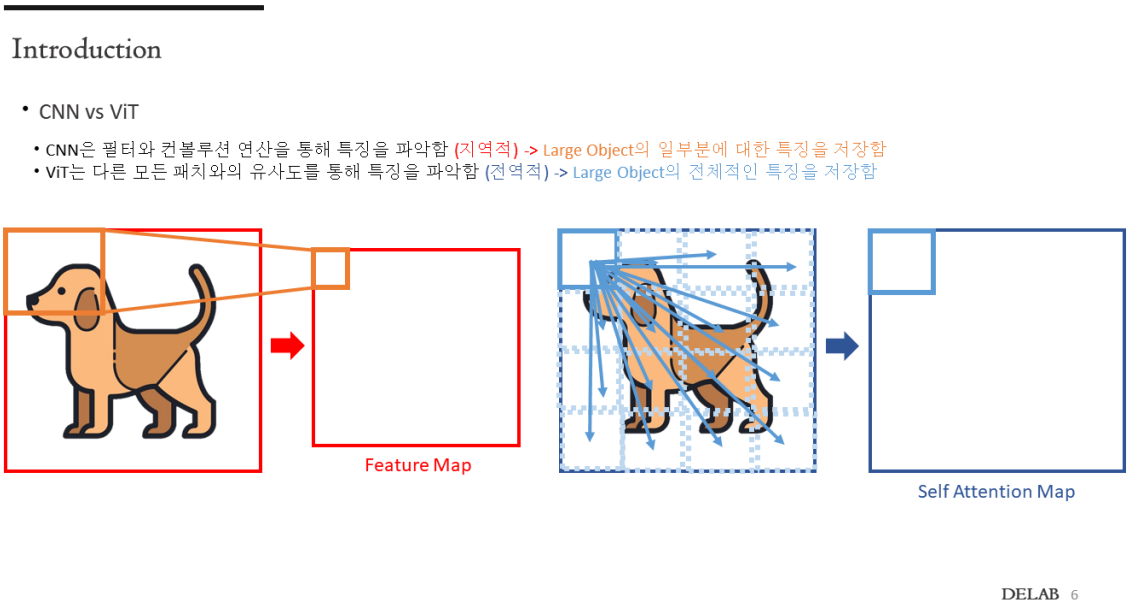
Introduction

- Vision Transformer (ViT)
  - 컴퓨터 비전 분야에 Transformer 기술을 도입한 논문
  - 이미지를 같은 크기의 패치들로 분할 후 시퀀스 형태로 나열한 뒤 Transformer Encoder의 입력으로 사용
  - Large Scale 학습에서 매우 우수한 성능을 보인다 (CNN 성능을 능가함)



이 논문은 처음부터 ViT랑 비교를 많이 합니다. 그래서 ViT를 리마인드해보겠습니다.
ViT는 컴퓨터비전 분야에 Transformer 기술을 도입한 논문이고, 이미지를 같은 크기의 패치들로 분할 후 시퀀스 형태로 나열한 뒤 Transformer의 Encoder의 입력으로 사용하는 구조를 지니고 있습니다. 여기서 시퀀스라는 말은 원래 NLP분야에서 문장을 나누는 단어 각각을 의미합니다. 근데 시퀀스는 원핫벡터로 이루어지기 때문에, 이미지 패치들도 Flatten한 뒤 벡터로 표현해서 나열했다고 받아들이면 될 것 같습니다.

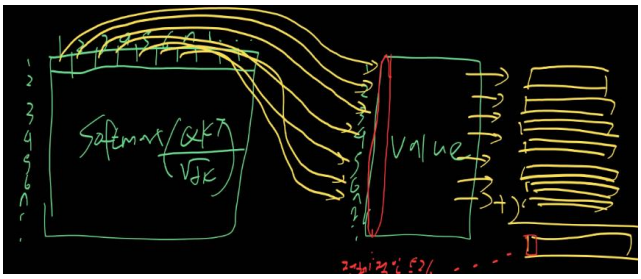
그리고 ViT의 장점중에 하나는 Large Scale 학습에서 매우 우수한 성능을 보였다는 것입니다. 깊게 쌓은 ResNet152보다 더 높은 성능을 보였습니다.



그 이유는 CNN과 ViT가 특징을 어떻게 찾는지를 비교하면 대략적으로 알 수 있습니다.
먼저 CNN은 Feature Map에 특징이 저장되고, ViT는 Self Attention Map에 특징이 저장됩니다. 그래서 각 Map의 한 원소를 기준으로 어떻게 만들어진건지 비교를 해보겠습니다.
그리고 이미지를 꽉 채우는 Large Scale 강아지를 입력 이미지로 사용하겠습니다.
먼저 CNN은 필터와 필터의 크기에 대응되는 이미지 공간의 Convolution 연산을 진행합니다. 따라서 이미지의 일부분만 보고 특징을 파악을 하므로 지역적인 정보가 담기게 됩니다.
반면 ViT는 한 패치와 다른 모든 패치간의 유사도를 통해 특징을 파악합니다. 이렇게 특징을 전역적으로 찾기 때문에 Self Attention Map의 원소는 모두 전역적인 특징이 저장이 됩니다.
따라서 위 강아지 예시에 따르면, '강아지의 부분적인 특징을 많이 저장하는 것보다, 전체적인 특징을 많이 저장한 것이 강아지를 인식하기에 더 좋더라' 라는 의미가 됩니다.

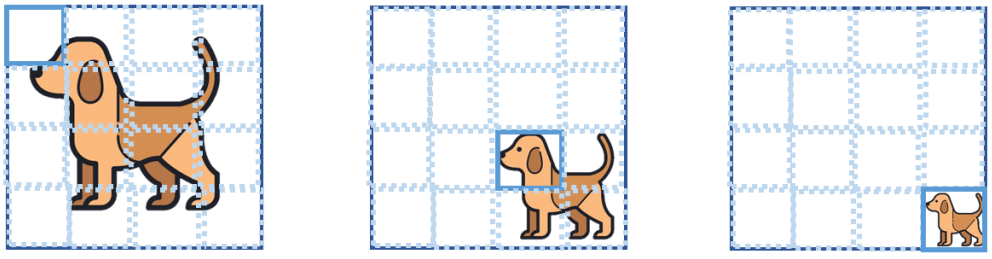
TMI) ViT를 설명할 때 그냥 유사도를 이용한다고만 설명을 했는데, 좀 더 자세히 설명하면 이미지를 Q,K,V로 나누고 Q와 K간의 내적으로 유사도를 구한 다음, Value행렬의 각 행에 대응되는 유사

도를 곱한 뒤 모든 행을 더하는 방식으로 특징을 파악합니다. 따라서 그 결과 행렬(self attention map)은 각 행마다(혹은 각 원소마다) 전체적인 정보를 담고 있습니다.
ex) 첫 행은 첫 번째 패치와 유사한 부분이 강조된 전체적인 이미지 특징, 2행은 두 번째 패치와 유사한 부분이 강조된 전체적인 이미지 특징 등등



Introduction

- ViT의 문제점
  - 이미지의 제곱에 해당하는 연산량를 가진다 (이미지의 모든 패치로 어텐션을 계산하기 때문)
  - 다양한 스케일에 대한 학습 성능이 낮다 (동일한 패치 사이즈를 갖기 때문)



Large Scale Object
강아지의 특징을 구하기 위해 영향을 끼치는 패치가 **16개**

Middle Scale Object
강아지의 특징을 구하기 위해 영향을 끼치는 패치가 **4개**

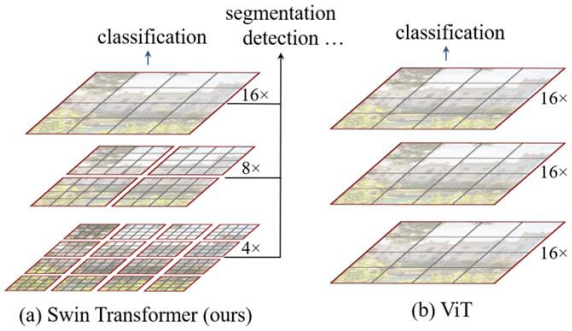
Small Scale Object
강아지의 특징을 구하기 위해 영향을 끼치는 패치가 **1개**

DELAB 7

그리고 ViT의 문제점으로는 크게 두 가지가 있는데, 첫 번째는 이미지의 제곱에 해당하는 연산량을 가진다는 것입니다. 이것은 이미지의 모든 패치로 어텐션을 계산하기 때문입니다. 그리고 다양한 스케일에 대한 학습 성능이 낮습니다. 특히 object가 작을수록 성능이 좀 떨어지는 경향이 있습니다. 아래 예시는 좀 극단적인 강아지 사진 세 개를 준비해봤습니다. 먼저 large scale object가은 경우는 강아지의 특징을 구하기 위해 영향을 끼치는 패치가 모든 패치가 되기 때문에 정보량이 많아 큰 객체에 대한 인식률이 좋지만, 이미지가 작아질수록 특징을 구하기 위해 영향을 끼치는 패치가 점점 줄어들어 정보량이 적어서 작은 객체 대한 인식률은 안 좋다라고 설명할 수 있습니다. (결국 지역적인 정보를 찾기 힘들다는 말) 물론 작은 객체가 다른 여러 패치에 존재하면 성능이 높을 수 있지만 일반적으로 큰 객체에 대한 성능보다는 떨어진다는 것이 많이 알려진 사실입니다.
이러한 문제는 이미지의 스케일에 상관없이 동일한 패치 사이즈를 갖기 때문이라고 논문에서 언급합니다.

Introduction

- Swin Transformer
  - **윈도우:** 패치들의 모음 (ViT는 패치들의 모음 -> 이미지, Swin은 윈도우의 모음 -> 이미지)
  - 이미지의 제곱에 해당하는 연산량을 가짐 -> 윈도우 내부의 어텐션을 계산함 (윈도우 개수만큼)
  - 다양한 스케일에 대한 학습 성능이 낮음 -> 패치 사이즈가 다른 다양한 윈도우에서 Attention을 계산함



DELAB 8

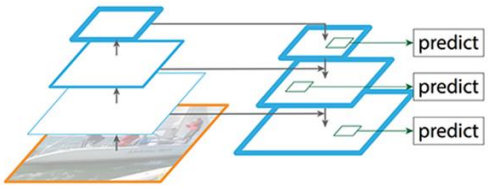
따라서 Swin은 윈도우라는 개념을 도입했습니다. 원래 ViT에서는 이미지를 패치들로 쪼갰기 때문에 패치들의 모음이 이미지가 됐습니다. 근데 Swin은 이 패치들을 묶어서 윈도우라고 명칭하고, 윈도우들이 모여서 이미지가 된다고 설명합니다. 그리고 스테이지마다 윈도우에 있는 패치의 사이즈를 다르게 설정했습니다. 예를 들어 첫 번째 스테이지는 패치의 크기가 4(패치 하나가 픽셀 4x4)였다면, 두 번째 스테이지는 8, 세 번째 스테이지는 16 이런 식으로 다르게 설정을 한 것입니다. 이렇게 설정한 이유는, 기존 ViT의 문제점을 해결하기 위해서라고 말씀드릴 수 있습니다.

먼저 ViT는 이미지의 제곱에 해당하는 연산량을 가졌습니다. 그런데 Swin은 윈도우 내부에서 어텐션을 계산하는 과정을 윈도우 개수만큼 진행을 합니다. 따라서 연산량을 더 낮출 수 있었습니다. 그리고 다양한 스케일에 대한 학습 성능이 낮은 부분은 각 스테이지별로 패치 사이즈를 다르게 함으로써 해결을 한 것입니다. 예를 들어, 첫 번째 스테이지는 local window에서의 attention이 진행되므로, 지역적인 정보에 해당하는 작은 객체에 대한 인식 성능이 높은 것입니다.

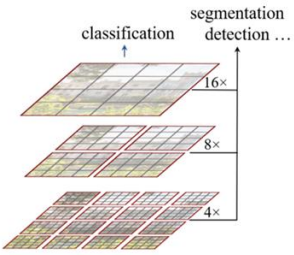
그리고 스테이지가 올라갈수록 global window에서의 attention이 진행되므로, 전역적인 정보에 해당하는 큰 객체에 대한 인식 성능이 높은 것입니다.

Architecture

- Swin Transformer
  - Swin은 FPN과 같이 계층적인 구조(Hierarchical Architecture)를 지닌다
  - 계층적인 구조는 다양한 스케일을 학습하는데 유리하다
  - 두 모델은 하위 Stage로 갈수록 작은 객체를 잘 파악하고, 상위 Stage로 갈수록 큰 객체를 잘 파악한다.



Feature Pyramid Network



Swin Transformer

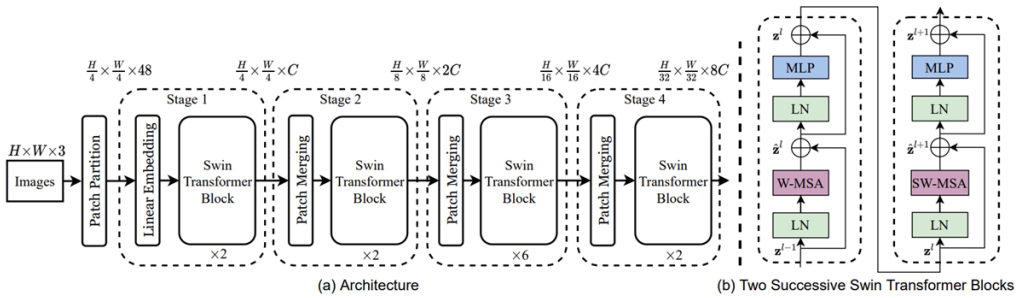
DELAB 9

이러한 아키텍쳐는 FPN 구조를 연상케 합니다. FPN도 계층적인 구조를 가지고 있으므로 다양한 스케일을 학습하는데 유리합니다. FPN을 간단하게 설명하면, ResNet의 각 스테이지를 이용해서 Bottom up layer를 구성하였고, nearest neighbor라는 upsampling 기법을 이용해서 top down layer를 구성한 모델입니다. 따라서 특징이 점점 작아졌다가, 다시 커지는 구조이며, 하위 layer에서는 지역적인 정보를 담고 있고, 상위 layer에서는 전역적인 정보를 가지고 있습니다. 그 이유는 CNN 자체가 상위 layer로 갈수록 receptive field가 커지면서 간접적으로 전역적인 정보를 확인할 수 있기 때문이라고 설명할 수도 있고, conv filter 사이즈는 똑같은데 이미지의 크기가 달라지면 이미지가 작은 상태보다 큰 상태에서 지역적인 정보를 파악하기가 당연히 더 좋기 때문입니다.
그리고 Swin도 특징을 파악하는 방법은 다르지만, FPN처럼 계층적인 구조를 사용하면서 다양한 스케일을 학습하는데 더 유리해졌다고 말씀드릴 수 있습니다.

TMI) FPN을 이용해서 object detection을 할 때는 각 스테이지별 output으로 predict를 진행합니다. 따라서 스케일별로 다양한 후보 박스들이 나오기 때문에 NMS 알고리즘을 거쳤을 때 기존보다 더 정확한 특징을 찾을 수 있게 됩니다. Swin을 이용해서 객체 탐지를 할 때도 각 스테이지별 output으로 predict를 할 수 있습니다.

Architecture

- Swin Transformer
  - Swin은 Patch Merging을 이용하는 다양한 스테이지로 구성돼있다
  - Swin의 Attention은 두 개의 연속적인 Block으로 계산한다
  - Swin Architecture는 VGGNet과 유사하다 (3차원 Feature Map 형태로 변환하면, Resolution은 점점 작아지고 Channel은 점점 커짐)



이제 본격적으로 Swin의 구조에 대해 말씀드리겠습니다.

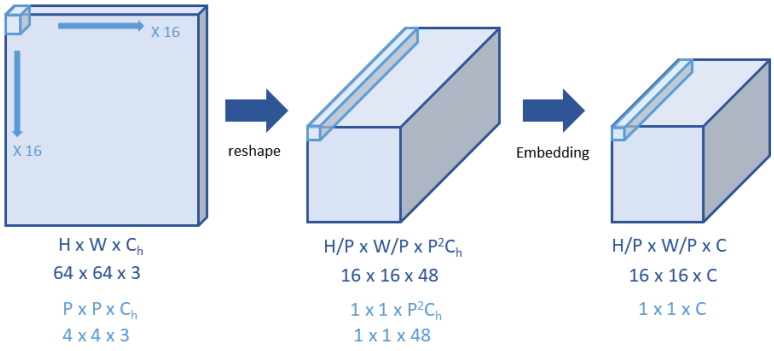
- 먼저 제일 눈에 띄는 부분은 Swin Blocks에 해당이 되는데, 기존 ViT는 MSA를 이용한 Transformer Encoder만을 사용했다면, Swin은 W-MSA와 SW-MSA를 연속적으로 수행하는 모듈을 통해 어텐션을 계산합니다. 그리고 Stage1에서는 Blocks가 1개 사용이 되고(Block이 2개 사용), Stage3에서는 Blocks가 3개 사용이 되는(Block이 6개 사용) 구조를 지닙니다.

- 구조를 보면 각 스테이지별 입출력을 3차원으로 표현을 했는데, 실제로 Attention을 계산할 때는 기존 Transformer처럼 2차원으로 처리를 한다는 것을 미리 말씀드립니다. (stage1에서 SwinBlock의 입력은 H/4*W/4 X 48이 됨)

- Swin은 Patch Merging이라는 기술을 도입해서 다양한 스테이지를 구성했습니다. Patch Merging은 말 그대로 패치들을 병합하는 것을 의미하는데, 그렇기 때문에 stage1에서는 패치 사이즈가 작아 패치의 개수가 H/4*W/4라면, stage2에서는 패치 사이즈가 커져 패치 개수가 H/8*W/8가 되게 됩니다. 그래서 Feature Map처럼 생각하면 Resolution은 점점 작아지고 Channel은 점점 커지는 구조이므로 VGGNet을 연상케하는 구조입니다.

Architecture

- Patch Partition & Linear Embedding
  - Patch Partition: 이미지를 크기가 P인 패치들로 분할 후 변환하는 과정
  - Linear Embedding: 채널축을 C로 재구성하는 과정 (벡터의 차원이 C가 되는 과정)



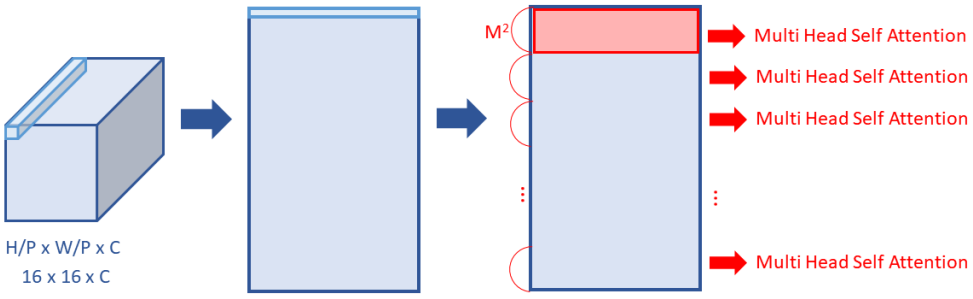
DELAB 11

그래서 모델의 가장 첫 모듈에 해당하는 Patch Partition과 Linear Embedding부터 말씀드리겠습니다. Patch Partition은 이미지를 크기가 P인 패치들로 분할 후 변환하는 과정을 의미합니다. 그래서 논문에서는 이해하기 쉽게 Patch의 크기를 4로 구성했는데, 초기 이미지가 64x64x3이라면, 4x4x3 크기의 패치가 가로로 16개, 세로로 16개가 되도록 분할할 수 있다는 것입니다. 근데 앞으로 Swin에서 3차원 텐서가 보이면, 텐서의 한 요소는 픽셀이 아니라 패치로 취급을 할 것이기 때문에, 4x4x3크기를 1x1x48로 변환을 하게 됩니다. 따라서 원본 이미지를 기준으로 Width와 Height에 Patch의 크기인 P만큼 나누게 되고, 채널수는 P^2을 곱한만큼 커지게 됩니다. 이것이 Patch Partition의 과정입니다.

그리고 Transformer에서 벡터의 차원을 변경하는 Embedding 과정이 있기 때문에, Swin도 Patch Partition 결과에 대해서 채널축을 C로 변환하게 됩니다.

Architecture

- W-MSA
  - Window-Multihead Self Attention
  - 윈도우 개수만큼, 윈도우 내부에서 멀티 헤드 셀프 어텐션을 진행한다.
  - 윈도우 사이즈: M (M x M 패치가 한 윈도우)

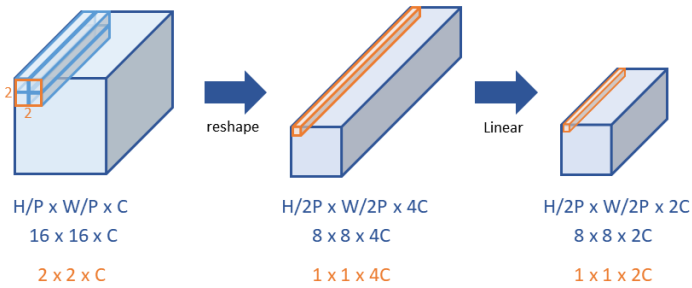


DELAB 12

다음은 Swin Block의 W-MSA에 대해 소개하겠습니다. W-MSA는 윈도우 개수만큼, 윈도우 내부에서 멀티 헤드 셀프 어텐션을 진행하겠다는 의미가 됩니다. 따라서 SwinBlock의 입력으로 사용되기 위해 2차원이 변환이 된 후에, 전체 행렬로 MSA를 진행하게 되지만, 여기서는 윈도우의 사이즈가 M이라고 할 때, 즉 MxM 패치가 한 윈도우일 때, M^2개의 행으로 MSA를 하겠다는 것입니다. 이러한 연산 과정을 N/M^2번 하게 됩니다.

Architecture

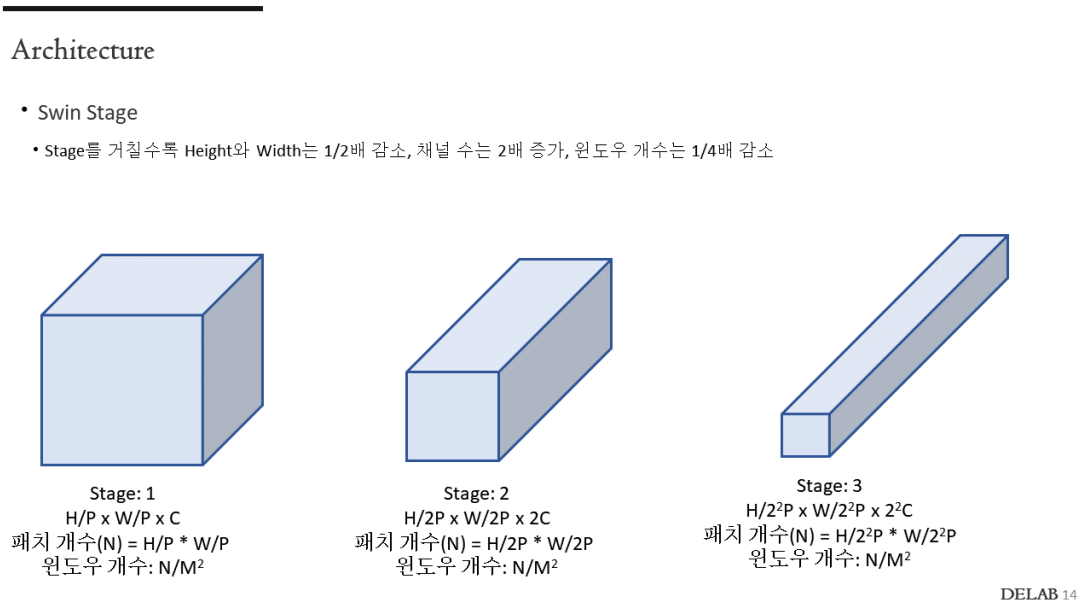
- Patch Merging
  - 이웃한 2x2패치들을 하나의 패치로 재구성하는 과정
  - Linear 연산을 통해 차원 수를 ½배로 변경
  - Resolution의 width와 height는 2배로 줄고, channel은 2배만큼 증가되는 구조



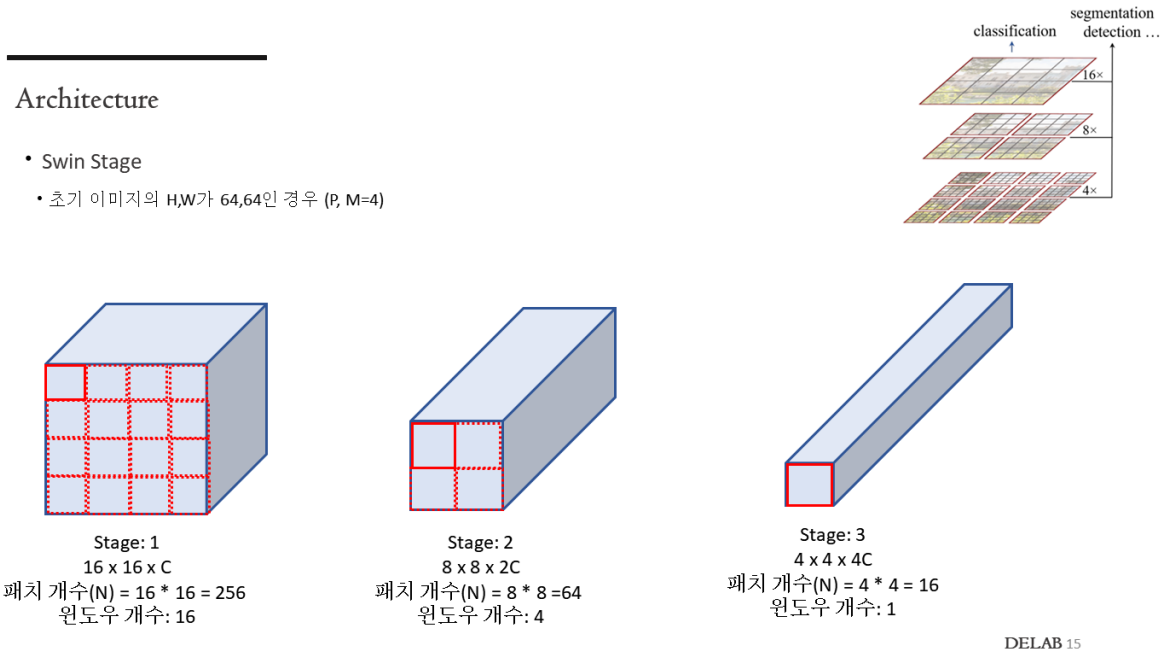
DELAB 13

SW-MSA를 설명하기 전에, 그 다음 단계인 Patch Merging부터 설명하겠습니다. (바로 설명하기 개념이 어려움) Patch Merging은 이웃한 2x2 패치들을 하나의 패치로 재구성하는 과정을 의미합니다. 그래서 패치 4개가 2x2xC 형태로 존재한다면, 이를 1x1x4C형태로 변환하겠다는 것을 의미합

니다. 이 과정을 전체에서 따져보면, Width와 Height는 2배씩 줄어들고 채널은 4배가 증가되는 구조입니다. 근데 이후 Linear 연산을 추가로 거쳐서 채널을 2배 다시 감소하는 과정을 겪는다고 합니다.



그래서 Swin모델의 Stage 출력을 비교해보면, Stage를 거칠수록 Height와 Width는 1/2배로 감소하고 채널수와 패치의 크기는 2배씩 증가하며, 윈도우의 개수와 패치의 개수는 1/4배로 감소가 됩니다.

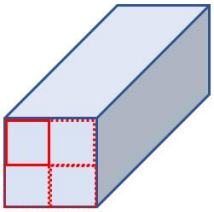


예를 들어, 초기 이미지의 H와 W가 64고, 처음 패치의 크기가 4, 윈도우의 크기가 4라고 하면 처

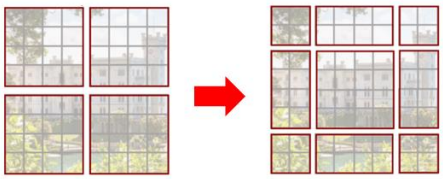
음에는 Patch Partition을 통해 16x16XC가 되지만, 두 번째 스테이지부터는 Patch Merging을 통해 8x8x2C, 4x4x4C가 되는 구조입니다. 패치와 윈도우의 개수도 처음에는 256개, 16개에서 1/4배로 점점 줄어들게 됩니다.

Architecture

- W-MSA의 한계
  - 윈도우의 위치가 고정되어 있다
    -> 윈도우간 인접한 패치들은 어텐션이 계산되지 않는다
    -> 이미지의 중간 영역에 대한 특징이 부족하다



Stage: 2
8 x 8 x 2C
패치 개수(N) = 8 * 8 =64
윈도우 개수: 4

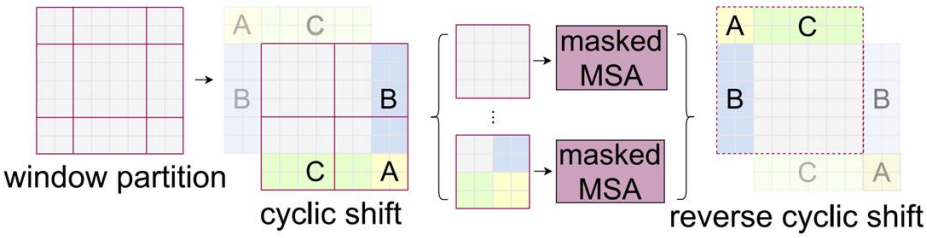


윈도우 개수를 기존 개수보다 가로, 세로 1개씩 늘리고
각각 어텐션을 계산할 필요성이 있음!

다음은 W-MSA의 한계입니다.
윈도우 내부에서만 어텐션을 수행하고, 이 윈도우가 고정돼 있기 때문에, 윈도우 간 인접한 패치들, 즉 이미지의 중간 부분은 어텐션이 계산되지 않는다는 것입니다. 따라서 윈도우를 다르게 그림처럼 가로, 세로 1개씩 늘리는 구조로 새롭게 배치한 후 각 윈도우마다 어텐션을 계산할 필요성이 있습니다.

Architecture

- SW-MSA
  - Shifted Window-Multihead Self Attention
  - 윈도우 위치를 이동시켜서 W-MSA와 같은 크기,같은 개수의 윈도우를 형성
  - 각 윈도우에 인접하지 않은 영역이 생기므로 masked MSA를 진행

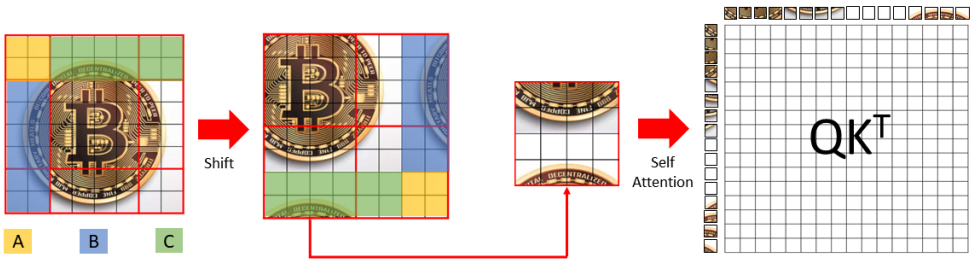


근데 이 논문에서는 한 가지 아이디어를 냅니다. 윈도우를 재구성한 뒤 바로 Attention을 계산하는 것이 아니라, 윈도우들의 위치를 그림과 같이 이동시켜서 W-MSA을 할 때와 같은 크기, 같은 개수의 윈도우를 형성한다는 것입니다. 이후 어텐션을 계산합니다. 하지만 이렇게 하면 각 윈도우마다 인접하지 않은 영역들이 생깁니다. 예를 들어 세 번째 윈도우는 초록색 부분이 인접하지 않습니다. 따라서 maskedMSA를 계산합니다. 이 계산이 끝나면 윈도우들의 위치를 원상복귀시킵니다.

TMI) 왜 굳이 shifted를 했냐고 묻는다면 shifted를 하지 않고 9개의 윈도우로 어텐션을 할 때보다 연산량이 줄어들기 때문이다. 9개의 윈도우로 Attention을 하기 전에 zero padding을 해서 정사각형 형태로 변환해서 4x4 윈도우 9개로 어텐션을 하는 것보다, shifted된 4x4 윈도우 4개로 어텐션을 하는 것이 연산량면에서 더 효율적이다.

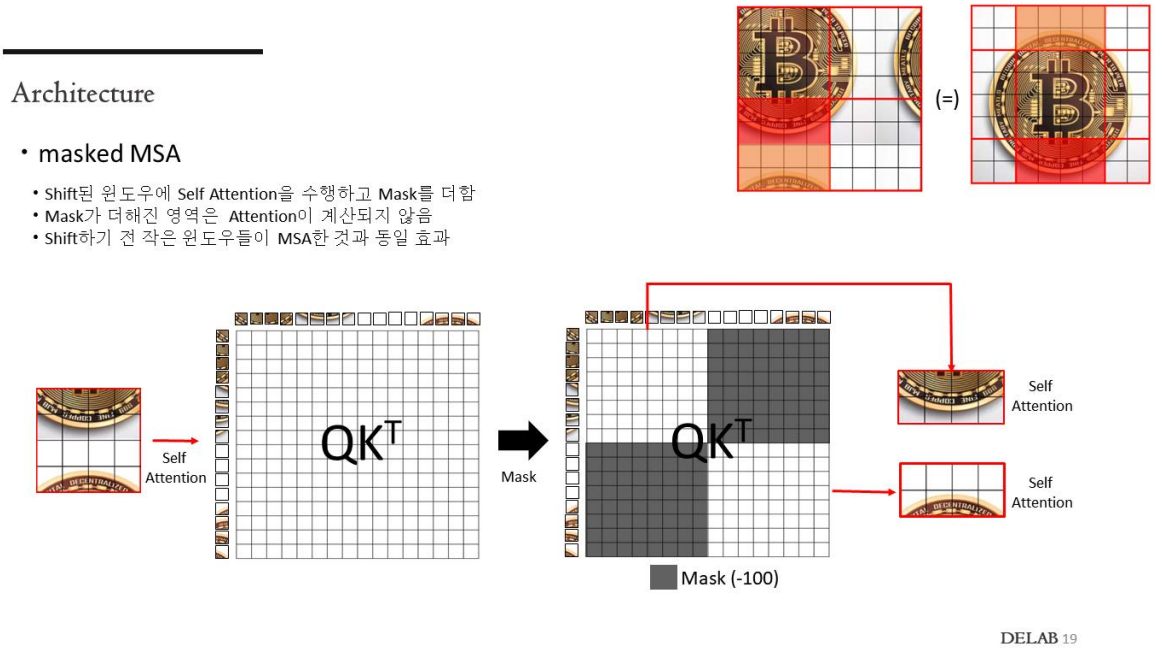
Architecture

- Shifted Window
  - Window의 개수 : {ceil(h/M) + 1} * {ceil(w/M) +1} <h는 패치의 행개수, w는 패치의 열개수>
  - Shift된 Window의 개수 : {ceil(h/M)} * {ceil(w/M)}
  - 새롭게 변환된 Window마다 Masked Multihead Self Attention을 수행함



DELAB 18

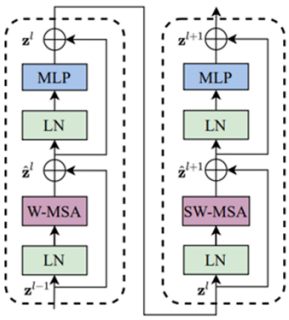
그래서 원래는 패치의 행개수(h)에 윈도우의 크기(M)만큼 나눈 값에 천장함수를 씌우고 +1한 것이 윈도우의 새로운 행개수가 되고, 패치의 열개수(w)에 윈도우의 크기(M)만큼 나눈 값에 천장함수를 씌우고 +1한 것이 윈도우의 새로운 열개수가 되는 구조인데, shift연산을 통해 이전 윈도우와 윈도우의 개수가 같게 됩니다. 그리고 새롭게 변환된 윈도우마다 Masked Self Attention을 수행하게 됩니다.



MaskedMSA는 selfAttention을 수행하고 Mask를 더하는 SelfAttention을 head번 진행하겠다는 의미가 됩니다. 이 때 마스크는 윈도우마다 다르게 구성되는데, 위 그림은 세 번째 윈도우에 대한 masked Self Attention 과정을 나타냅니다. 여기서는 마스크를 그림과 같이 구성하게 되는데, 이렇게 하면 실질적으로는 세 번째 윈도우에서의 윗 부분과 아랫부분에 대한 self Attention을 한 것과 동일 효과를 보입니다. 그리고 shift하기 전 그림과 비교하면 이미지에서 중간 부분에 대한 어텐션을 수행한 것을 확인할 수 있습니다. 나머지 부분도 이와 같은 방법으로 모두 어텐션이 계산되게 됩니다.

Architecture

- SwinTransformer Blocks
  - ViT에서 사용하는 Transformer Encoder와 유사함
  - 원본 Transformer와 달리 LayerNorm을 먼저 수행함
  - 윈도우 내 어텐션(W-MSA)과 윈도우 간 어텐션(SW-MSA)을 순차적으로 수행하는 구조



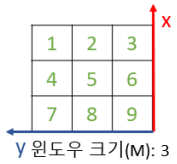
$$\hat{\mathbf{z}}^{l} = \text{W-MSA}\left(\text{LN}\left(\mathbf{z}^{l-1}\right)\right) + \mathbf{z}^{l-1},$$
$$\mathbf{z}^{l} = \text{MLP}\left(\text{LN}\left(\hat{\mathbf{z}}^{l}\right)\right) + \hat{\mathbf{z}}^{l},$$
$$\hat{\mathbf{z}}^{l+1} = \text{SW-MSA}\left(\text{LN}\left(\mathbf{z}^{l}\right)\right)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

그래서 SwinBlocks를 다시 살펴보면 위와 같은 과정을 통해 연산하게 됩니다.
윈도우 내 어텐션이라고 불리는 W-MSA와 윈도우 간 어텐션이라고 불리는 SW-MSA를 순차적으로 수행하는 구조이며, 원본 Transformer와는 달리 LayerNorm을 먼저 수행하는 구조입니다.

Architecture

- Relative Position bias
  - Swin은 ViT와 달리 Positional Encoding을 하지 않음
  - 대신 Attention 과정에서 Relative Position bias를 더함
  - **Relative Position bias:** 패치 간 상대적인 이동거리 (M^2xM^2)

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



	1	2	3	4	5	6	7	8	9
1	0	-1	-2	0	-1	-2	0	-1	-2
2	1	0	-1	1	0	-1	1	0	-1
3	2	1	0	2	1	0	2	1	0
4	0	-1	-2	0	-1	-2	0	-1	-2
5	1	0	-1	1	0	-1	1	0	-1
6	2	1	0	2	1	0	2	1	0
7	0	-1	-2	0	-1	-2	0	-1	-2
8	1	0	-1	1	0	-1	1	0	-1
9	2	1	0	2	1	0	2	1	0

y축 기준 상대 거리 [-M+1, M-1]

	1	2	3	4	5	6	7	8	9
1	0	0	0	-1	-1	-1	-2	-2	-2
2	0	0	0	-1	-1	-1	-2	-2	-2
3	0	0	0	-1	-1	-1	-2	-2	-2
4	1	1	1	0	0	0	-1	-1	-1
5	1	1	1	0	0	0	-1	-1	-1
6	1	1	1	0	0	0	-1	-1	-1
7	2	2	2	1	1	1	0	0	0
8	2	2	2	1	1	1	0	0	0
9	2	2	2	1	1	1	0	0	0

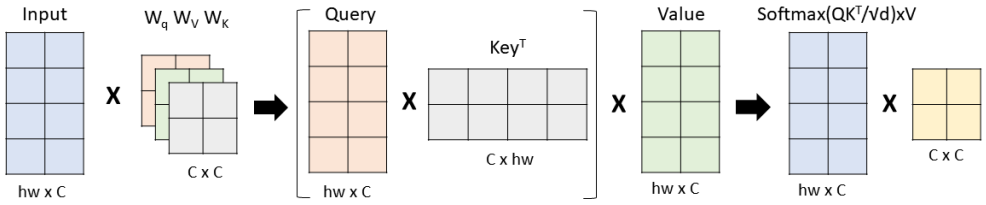
x축 기준 상대 거리 [-M+1, M-1]

마지막으로, Relative Position bias에 대해 말씀드리겠습니다. 일단 Swin은 ViT와 달리 Positional Encoding을 진행하지 않았습니다. 즉 Swin의 입력으로 들어갈 때, sin과 cos함수를 이용한 절대좌표를 더하지 않았다는 것인데, 대신 Attention 과정에서 상대 좌표에 해당하는 Relative Posision bias를 더하게 됩니다. 예를 들어 윈도우의 크기가 3이여서 3x3패치들 간 어텐션을 계산한다고 해보겠습니다. 그럼 Bias를 더한다는 것은, 첫 번째 패치와 나머지 패치간의 상대적인 이동거리,

두 번째 패치와 나머지 패치간의 상대적인 이동거리 등을 전부 고려하겠다는 것입니다.
따라서 B는 y축 기준 상대거리와 x축 기준 상대거리를 이용해서 만든 M^2xM^2 크기를 가지게 됩니다. (B가 실제로 어떻게 구성되는지는 논문에 자세히 안 나와 있었음)

Compare

- MSA의 연산량
  - 계산 편의를 위해 Head 수는 1로 지정함



1. Query, Key, Value를 구하는 과정: 3 * hwC^2
2. Scale dot product Attention: QK^T = C * $(hw)^2$, Value와 Matrix Multiplication = C * $(hw)^2$
3. Input 행렬과 차원을 맞춰주는 과정: hwC^2

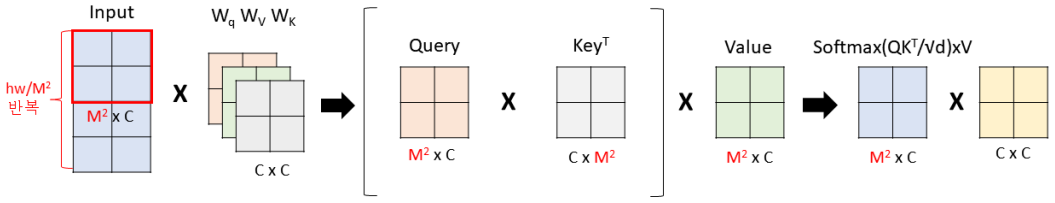
$$\Omega(\text{MSA}) = 4hwC^2 + 2(hw)^2C,$$

DELAB 22

다음은 ViT와 Swin에서 MSA의 연산량과 W-MSA의 연산량을 비교하겠습니다.
논문에서는 계산 편의를 위해 MSA에서 head의 수는 1로 지정하였습니다.
먼저 Query를 구한다고 했을 때, Input의 한 행과 W_q의 한 열이 내적을 수행하게 됩니다. 여기서 C번 연산을 하게 되고, 이러한 내적을 C번 반복합니다. 그리고 이 과정을 Input의 모든 행에 대해서 수행해야 되므로 hw*C^2번 연산을 합니다. 이 작업을 Key와 Value를 구할 때도 똑같이 진행하므로 3*hwC^2이 됩니다.
그리고 Scale dot product attention이나 마지막에 차원수를 맞춰주는 과정도 다 행렬 연산을 통해 진행이 되므로 계산 과정은 생략하겠습니다.
그래서 결론적으로 $4hwC^2$ + $2(hw)^2C$번 연산을 하므로 패치의 개수에 제곱한만큼 연산을 한다는 것입니다. 다른 말로 하면 이미지의 크기의 제곱만큼 연산을 하게 됩니다.

Compare

- W-MSA의 연산량
  - 계산 편의를 위해 Head 수는 1로 지정함



1. Query, Key, Value를 구하는 과정: $(3 * M^2C^2) * hw/M^2 = 3C^2hw$
2. Scale dot product Attention: $QK^T = (C * M^4) * hw/M^2 = M^2Chw$, Value와 Matrix Multiplication = $(C * M^4) * hw/M^2 = M^2Chw$
3. Input 행렬과 차원을 맞춰주는 과정: $(C^2M^2) * hw/M^2 = C^2hw$ ➡ $\Omega(\text{W-MSA}) = 4hwC^2 + 2M^2hwC,$

DELAB 23

다음은 W-MSA의 연산량을 계산하겠습니다.

먼저 Query를 제작할 때, Input전체가 쓰이는 것이 아니라, 윈도우만큼, 즉 M^2개의 행만큼 사용이 됩니다. 따라서 윈도우의 한 행과 W_q의 한 열이 내적을 수행하게 됩니다. 여기서 C번 연산을 하게 되고, 이러한 내적을 C번 반복합니다. 그리고 이 과정을 윈도우의 모든 행에 대해서 수행해야 되므로 M^2*C^2번 연산을 합니다. 그런데 이러한 윈도우 개수가 hw/M^2개만큼 있고, Key, Value까지 고려해야 하므로 M^2C^2에 3을 곱하고 hw/M^2을 곱한만큼 계산하면 Input에 대해서 Query, Key, Value를 구하는 연산량이 나옵니다.

비슷한 과정으로 scale dot attention과 차원 축소까지 거치면, $4hwC^2+2M^2hwC$번 연산을 하게 됩니다. 여기서는 이미지의 크기와 관련하여 선형 계산 복잡성을 가진다.

따라서 Swin이 ViT보다 더 효율적인 모델이라고 볼 수 있다.

Experiments

- Swin Models
  - SwinTransformer Block 개수와 변경할 차원 수(C)와 Head 수를 다르게 설정하며 (Swin-T ~ Swin-L)을 제작함

- Swin-T: $C = 96$, layer numbers = $\{2, 2, 6, 2\}$
- Swin-S: $C = 96$, layer numbers =$\{2, 2, 18, 2\}$
- Swin-B: $C = 128$, layer numbers =$\{2, 2, 18, 2\}$
- Swin-L: $C = 192$, layer numbers =$\{2, 2, 18, 2\}$

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

다음은 실험입니다. 논문 저자는 Swin Model을 제작할 때 SwinBlock수나 변경할 차원 수, head 수 따위를 다르게 설정하며 여러 버전의 Swin모델을 만들었습니다. 가장 작은 모델은 SwinT로 Linear Embedding 과정에서 채널수는 96이 되고, W-MSA와 SW-MSA를 계산 시 head수는 stage 별로 3,6,12,24로 설정하였고, 세 번째 Stage에서 SwinBlock수는 6으로 선정하였습니다.

Experiments

- Results
  - 이미지의 input size를 다르게 설정하여 세 모델으로 실험 (classification)
  - 이미지가 클수록 accuracy가 높지만 throughput이 낮아지는 결과
  - 큰 모델일수록 accuracy가 높아지지만 throughtput이 낮아지는 결과

	Swin-T		Swin-S		Swin-B	
input size	top-1 acc	throughput (image / s)	top-1 acc	throughput (image / s)	top-1 acc	throughput (image / s)
224^2	81.3	755.2	83.0	436.9	83.3	278.1
256^2	81.6	580.9	83.4	336.7	83.7	208.1
320^2	82.1	342.0	83.7	198.2	84.0	132.0
384^2	82.2	219.5	83.9	127.6	84.5	84.7

Table 8. Swin Transformers with different input image size on ImageNet-1K classification.

실험 결과는 예상한 대로, 모델이 클수록 accuracy가 높아졌고 throughput(처리량)이 낮아지는 결과를 보였습니다. 그리고 해상도의 크기에 따라서도 accuracy는 비례하고, throughtput은 반비례한 결과를 보였습니다.

Experiments

- Object Detection

(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

COCO Dataset

- Semantic Segmentation

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.

다음은 Swin을 객체 탐지와 영상 분할 모델의 backbone network로 사용했을 때의 성능 비교입니다. 객체 탐지에서는 ResNet50과 비교해서 성능이 전체적으로 향상되었고, 영상 분할에서는 ResNet 뿐 만이 아니라 DeiT이라고 하는 ViT이후에 나온 트랜스포머 모델보다도 성능이 높은 것을 확인하였습니다.

Experiments

- Shifted Windows & Relative Position bias
  - Shifted Windows를 사용하니 (이미지 분류, 객체 탐지, 영상 분할) 모든 Task에서 성능 향상이 됨
  - 절대 좌표를 더하는 것(positional encoding)보다 상대 좌표를 더하는 것(relative position bias)이 더 좋은 성능을 보임

	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).

그리고 shifted windows와 relative position bias에 대한 실험도 진행하였습니다.

- shifting을 진행하지 않고 각 윈도우마다 어텐션을 수행했을 때보다 shifting을 한 것이 더 성능이 좋았다고 합니다.
- position 정보는 상대좌표만 추가했을 때만 가장 높은 성능을 보였습니다.