10장 비전 트랜스포머

날짜	@2025년 4월 30일 → 2025년 5월 6일
선택	DL세션 과제
주차	9주차
진행 상태	완료

Vit(Vision Transformer)

이미지 인식을 위한 딥러닝 모델

이미지를 자연어 처리 방식처럼 분류해 보려는 시도에서 탄생

- CNN 모델의 합성곱 계층 방법 사용X
  - 이미지 분류를 위해 지역 특징을 추출
- 트랜스포머 모델에서 사용되는 셀프 어텐션을 적용
  - 전체 이미지를 한 번에 처리

    - 왼쪽→오른쪽, 위→아래 방향으로 순차적 입력 ⇒ 2차원 구조의 이미지 특성을 완전히 반영x

  ⇒ 해상도를 계층적으로 학습하는 스윈 트랜스포머(Swin Transformer)와 CvT(Convolutional Vision Transformer)모델이 제안됨

- 스윈 트랜스포머는 로컬 윈도(Local Window)를 활용해 각 계층의 어텐션이 되는 패치의 크기와 개수를 다양하게 구성해 이미지 특성을 학습시킴.
    - 기존 셀프 어텐션을 로컬 윈도 안에 대한 어텐션, 로컬 윈도 간의 어텐션으로 수행하여 이미지 특징을 계층적으로 학습
    - 이 어텐션 함수에는 상대적 위치 편향을 반영하여 어텐션값 자체에 위치적 정보를 포함시킴
- CvT는 기존 합성곱 연산 과정을 ViT에 적용한 모델
    - 저수준 특징과 고수준 특징을 계층적으로 반영할 수 있음
    - 저수준 특징: 눈, 코, 입과 같은 작은 단위
    - 고수준 특징: 눈, 코, 입을 포함한 전체 얼굴
    - 어텐션 연산과정에서 쿼리(Query, Q), 키(Key, K), 값(Value, V) 중 키와 값을 기존 특징 벡터보다 축소해 계산 복잡도를 감소시킴

ViT

트랜스포머 구조 자체를 컴퓨터비전 분야에 적용한 첫 번째 연구

- 이미지가 격자로 작은 단위의 이미지 패치로 나뉘어 순차적으로 입력됨
- ViT 모델에 사용되는 입력 이미지 패치는 왼→오, 위→아래의 시퀀셜 배열을 가정함

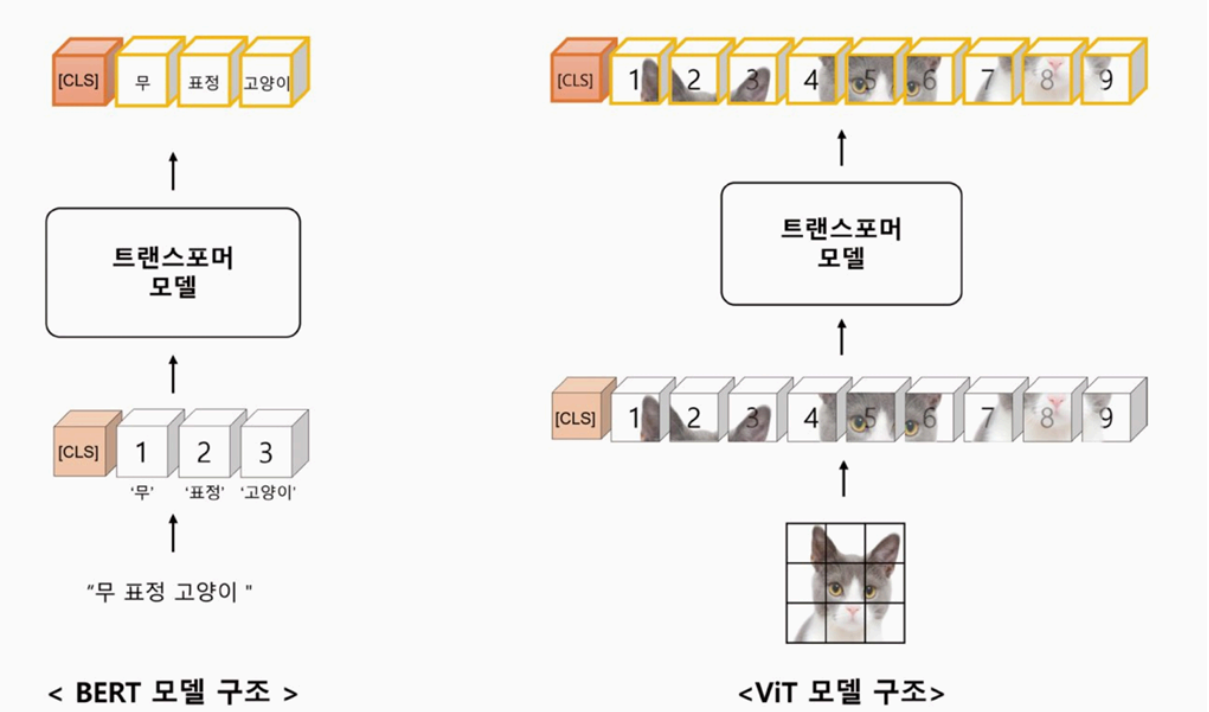

그림 10.2 BERT의 ViT 모델 구조 차이

합성곱 모델과 ViT 모델 비교

✅ (공통점)

이미지 특징을 잘 표현하는 임베딩 만들기

(차이점) : 과정

- 합성곱 신경망: 이미지 패치 중 일부만 선택하여 학습 → 이미지 전체의 특징 추출

  (ex) 왼쪽 눈의 특징 추출 → 이미지에서 해당 부분만 선택학습

- ViT 임베딩은 이미지를 작은 패치들로 나눠 패치 간의 상관관계를 학습
  - 이를 위해 셀프 어텐션 방법을 이용: 모든 패치가 서로에게 주는 영향을 고려해 이미지 전체 특징을 추출 ⇒ 모든 이미지 패치가 학습에 관여하며 높은 수준의 이미지 표현을 제공할 수 있음

📊 따라서 좁은 수용 영역(Receptive Field, RF)을 가진 합성곱 신경망은 전체 이미지 정보를 표현하는데 수많은 계층이 필요하지만, 트랜스포머 모델은 **어텐션 거리(Attention Distance)**를 계산하여 오직

- **한 개의 ViT 레이어로 전체 이미지 정보를 쉽게 표현**

- 패치 단위로 이미지 처리 → 더 작은 모델로도 높은 성능을 얻을 수 O
- 입력 이미지의 크기가 고정 → 이미지 크기를 맞추는 전처리 필요
- 합성곱 신경망: 공간적인 위치 정보 고려 ↔ 패치 간의 상대적인 위치만 고려⇒ 이미지 변환에 취약할 수 O

ViT의 귀납적 편향(Inductive Bias)

: 일반화 성능 향상을 위한 모델의 가정(Assumption)

모델이 가지는 구조와 매개변수들이 데이터에 적합한 가정을 함 → 더 적은 데이터로 높은 일반화 성능을 보일 수 있음 ↔ 강하다면, 다른 유형의 데이터나 관계 표현이 어려움

- 이미지데이터는 공간적 관계를 잘 표현 ⇒ **지역적(Local) 편향**을 가진 합성곱 신경망 모델 사용
- 시계열 데이터: 시간적 관계 표현 ⇒ **시퀀설(Sequential) 편향**을 가진 순환 신경망 모델 사용
    - 이전 시간 상태를 기억 → 현재 입력과 결합 → 다음 상태 예측
    - 순차적인 특징을 강조하는 특성
- BUT, 이러한 특정 관계 편향이 강할수록, 다른 다양한 관계를 표현하기 어려움

  → 귀납적 편향이 약한 모델을 선호하게됨.

➡️ ViT모델은 입력 데이터의 다양한 쿼리, 키, 값의 임베딩 형태로 일반화된 관계를 학습

→ **귀납적 편향이 거의 없음** & 대용량 데이터의 이미지 특징들의 관계를 잘 학습

ViT 모델

⇒ 패치 임베딩과 인코더 계층을 통해 특징을 추출하고, 작업(분류, 회귀 등)에 맞는 출력값으로 변환해 사용

[패치 임베딩(Patch Embedding)] 계층

: 입력 이미지를 트랜스포머 구조에 맞게 일정한 크기의 패치로 나눈다음, 각 패치를 벡터 형태로 변환

[인코더(Encoder)] 계층

: 각 패치와의 관계를 학습

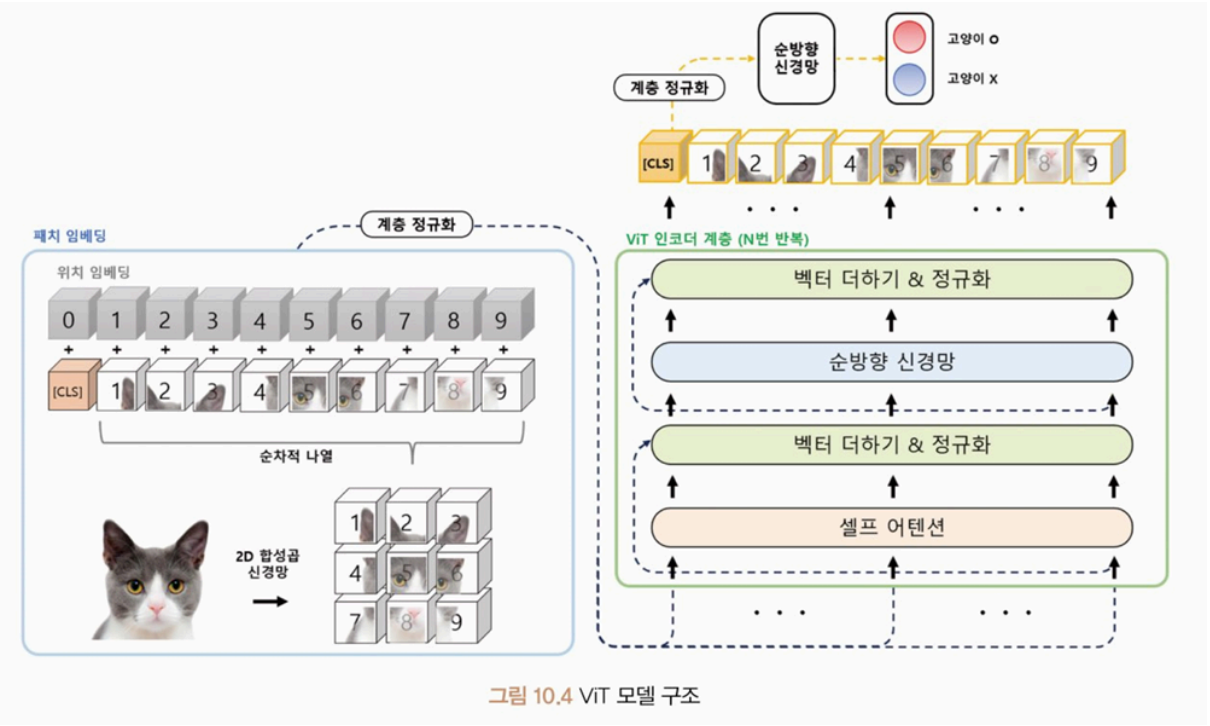

그림 10.4 ViT 모델 구조

패치 임베딩(Patch Embedding)

: 입력 이미지를 작은 패치로 분할하는 과정

→ GPU 메모리 한계를 극복하고, 더 큰 이미지를 처리할 수 있음

1. **이미지 전처리**: 이미지 크기를 일정한 크기로 변경
2. 전체 이미지를 패치 크기로 분할해 **시퀀셜 배열**을 만듦(합성곱 신경망 사용)
    - Kernel Size와 Stride를 설정해야 함: 하이퍼파라미터로 정의
        - 커널: 패치의 크기, 간격: 패치의 이동 폭
        - Patch size = (image size - kernel size) / (stride)
    - 배열은 왼→오, 위→아래 순서로 배열, 배열의 가장 왼쪽에 **분류 토큰**을 추가
    - 분류 토큰(Special Classification Token[CLS]): 전체 이미지를 대표하는 벡터로 특정 문제 예측에 사용
3. **위치 임베딩**(Position Embedding)을 사용하여 인접한 패치 간의 관계 학습
    - 위치 임베딩: 위치 정보를 임베딩 벡터로 변환하고, 기존 이미지 패치 벡터들과 더함
4. 마지막으로 **계층 정규화**를 거치면 패치 임베딩이 만들어짐

인코더(Encoder)

이미지를 일정한 크기의 패치로 나눠 각 패치를 벡터로 변환한 후 **인코더 계층에 입력으로 전달해 다양한 쿼리, 키 , 값 임베딩 관계를 학습함**

- N개의 인코더 레이어를 반복적으로 적용
- → 마지막 레이어에서는 분류 토큰이라고 불리는 특별한 패치의 특징 벡터를 추출
- 분류 토큰 벡터: 이미지 데이터를 잘 표현하는 특징 벡터로, 이후 다양한 이미지 분류 및 검색 문제 해결에 사용

  (Ex) 고양이인지 아닌지 구분하는 모델 학습 → ViT 모델은 Sequential Output을 모두 사용하는 것이 아니라, 분류 토큰 벡터만 사용해서 분류문제를 풀게 됨

  ⇒ 분류 토큰 벡터를 순방향 신경망(또는 완전 연결 계층)에 연결하고 고양이인지 아닌지를 분류하는 방법으로 학습됨.

~ 실습 ~

Swin Transformer

: 2021년에 발표한 대규모 비전 인식 모델

이미지 분류, 객체 감지, 영상 분할과 같은 인식 작업에서 강력한 성능을 내며, 트랜스포머 구조보다 이미지 특성을 더 잘 표현

📌 **기존 트랜스포머를 기반으로 하는 모델의 공통적인 문제점**

1. 고정된 패치 크기 ⇒ 세밀한 예측을 필요로 하는 의미론적 분할(Semantic Segmentation)과 같은 작업에 적합X
2. 트랜스포머의 셀프 어텐션은 입력 이미지 크기에 대한 2차(Quadratic) 계산 복잡도를 가짐 ⇒ 고해상도 이미지 처리가 어려움

⇒ 계층적(hierarchy) 특징 맵을 구성해 1차(Linear) 계산 복잡도를 갖는 스윈 트랜스포머가 등장

- 시프트 윈도(Shifted Window)라는 기술을 이용해 입력 이미지를 일정한 크기의 패치로 분할하고, 이 패치들을 쌓아 윈도우를 구성함
- 구성된 윈도우 영역만 셀프 어텐션을 계산
  - 기존 트랜스포머 모델과 다르게 패치를 겹쳐 더 큰 효율성을 제공
  - 계층마다 어텐션이 수행되는 패치의 크기와 개수를 계층적으로 적용해 처리 → 높은 해상도와 다양한 크기를 가진 객체를 효율적으로 처리할 수 있음

> 이러한 특징을 바탕으로 객체 탐지나 객체 분할 등과 같은 작업에서 백본 모델로 사용

ViT와 스윈 트랜스포머 차이

📌 **ViT 모델의 한계**

- 획일적인 패치 크기&위치에 대한 제약 → 다양한 크기의 종횡비를 가진 데이터 세트에 대해서 적용이 어려움
- 패치 단위로 입력을 받음 → 이미지의 공간 정보가 완전히 보존되지 않을 수 있음.
- 이미지 패치 수 증가 시 학습 데이터의 크기도 커짐 → 모델 학습에 필요한 계산량이 증가(대규모 세트에서 컴퓨터 자원을 많이 사용)

⇒ 이러한 한계점을 보완하기 위해 스윈 트랜스포머는 **Local Window**를 활용해 물체의 크기나 해상도를 계층적으로 학습함.

- 로컬 윈도(Local Window): 입력 이미지를 처리하기 위해 사용되는 고정 크기의 작은 윈도
- 윈도 시프트 기술을 도입 → 입력 이미지를 일정한 크기의 패치로 분할 → 특성 맵을 이동시켜 다음 계층 사용 ⇒ 패치의 위치를 더 유연하게 다룸

⇒ 더 높은 정확도와 더 적은 매개변수 수를 가지면서도 더 넓은 범위의 이미지 크기와 종횡비를 다룰 수 있음.

> 입력 패치의 로컬 윈도를 통해 계층적으로 접근하고, 시프트 윈도로 로컬 윈도를 이동해 가면서 인접 패치 간 정보를 계산

그림 10.6 ViT와 스윈 트랜스포머의 네트워크 구조

- 회색 격자: 패치
- 발간색 격자: 로컨 윈도

ViT의 네크워크는 계층마다 전체 이미지 안에 어텐션이 수행되는 패치와 개수가 동일

↔ 스윈 트랜스포머는 로컬 윈도를 통해 계층마다 어텐션이 되는 패치의 크기와 개수를 다양하게 사용

계층적 접근 방식은 로컬 윈도가 각 패치의 세부 정보에 집중하게 되고, 시프트 윈도를 통해 전역 특징을 확인할 수 있으므로 고해상도 이미지를 효율적으로 처리할 수 있음.

	ViT	스윈 트랜스포머
분류 헤드	[CLS] 토큰 사용	각 토큰의 평균값 사용

로컬 윈도 사용	X	O
상대적 위치 편향 사용	X	O
계층별 패치 크기	동일함	동일함

스윈 트랜스포머 모델 구조

1. 입력 이미지를 일정한 크기의 패치로 분할
2. 각 패치에 대해 선형 임베딩(Linear Embedding)을 수행 ⇒ 각 패치는 고정된 차원의 벡터로 변환됨
3. 분할된 패치들을 기반으로 스윈 트랜스포머 블록을 구성
    a. 어텐션 계층, 계층 정규화, 다층 퍼셉트론 등을 포함
    b. 패치 간 상호 작용을 수행
4. 패치를 병합해 전체 이미지에 대한 분류 수행: 분할된 패치들을 순차적으로 합치면서 이미지의 전체적인 정보를 추출하는 방식 사용

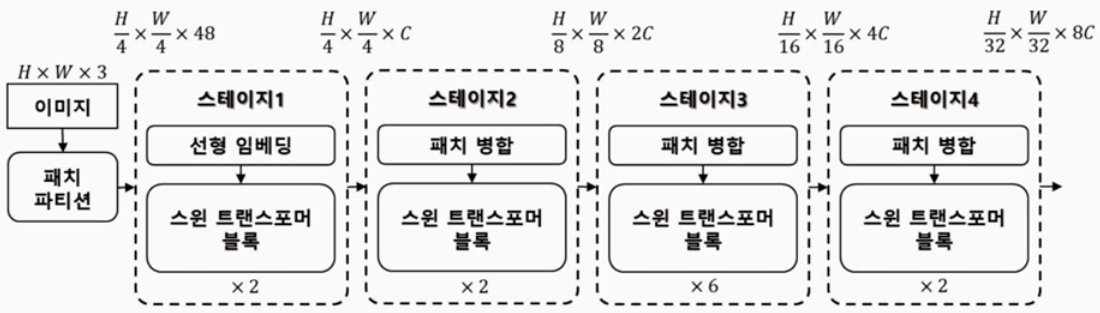

그림 10.7 스윈 트랜스포머 구조

패치 파티션 / 패티 병합

📖 **패치 파티션(Patch Partition)**

입력 이미지를 작은 사각형 패치로 분할해 처리하는 방식. 분할된 이미지는 트랜스포머 계층의 입력으로 사용되며, 입력 이미지의 공간 정보를 보존하고 트랜스포머 계츠엥 대한 계산 효율성을 높임

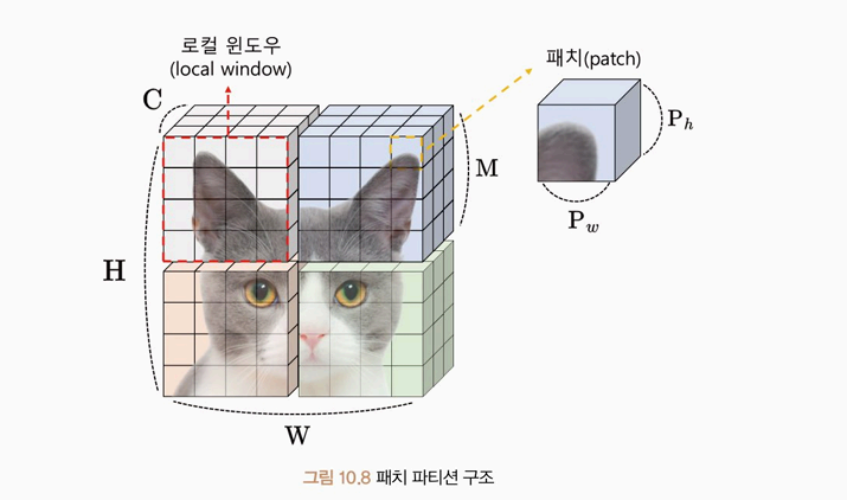

그림 10.8 패치 파티션 구조

- 일반적 이미지 텐서는 [C, H, W]의 구조를 가짐 → 이 구조에서 패치 파티션을 통해 로컬 윈도우를 추가함

> 📖 **패치 병합(Patch Merging)**
>
> 인접한 패치들의 정보를 저차원으로 축소하는 과정
>
> 모델 매개변수의 수를 줄이기 위해 이미지 패치 텐서 [C, H, W]를 [2C, H/2, W/2]로 재정렬한 후, 2C의 차원을 저차원으로 임베딩함

(ex) 채널 3개, 이미지 패치 64개로 구성된 텐서 [3,8,8]

: [6, 4, 4]로 재정렬 → 증가된 6차원을 원래 3차원으로 임베딩해 산출물 [3,4,4] 텐서를 생성함

스윈 트랜스포머 블록

: 패치 파티션 이후 네 개의 스테이지 안에서 반복 적용되며, 선형 임베딩 또는 패치 병합 이후에 수행된다

- 계층 정규화
- W-MSA(Window Multi-head Self-Attention)
- MLP
- SW-MSA(Shifted Window Multi-head Self attention)

등으로 구성된다.

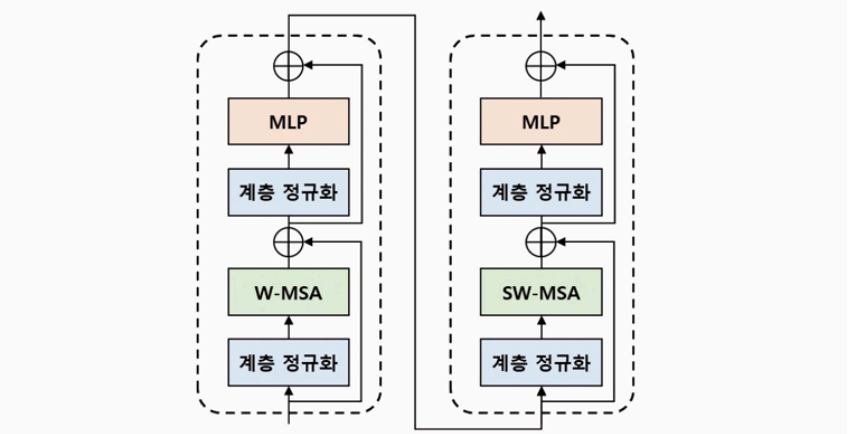

그림 10.9 스윈 트랜스포머 블록 구조

: 스윈 트랜스포머 블록 안에서는 트랜스포머 모델에서 사용되던 MSA가 아닌 W-MSA와 SW-MSA로 구성되며 순차적으로 수행됨

📖 W-MSA

: 로컬 윈도를 사용하는 멀티 헤드 셀프 어텐션

입력 특징 맵을 중첩되지 않게 나눈 다음 각 윈도에서 독립적으로 셀프 어텐션을 수행

- 이미지에서 일정한 크기의 윈도 영역을 설정,
- 해당 영역 내 픽셀 간 어텐션 계산,
- 이를 통해 이미지 내 전체적 어텐션 수행,
- 특정 위치에 대한 정보를 추출,
- 각 윈도 내 지역적 특징 분석,
- 전체 입력에 대한 셀프 어텐션 비용을 2차 계산 복잡도에서 1차 계산 복잡도로 줄임

: W-MSA는 연산량을 대폭 감소시켰지만, 윈도 내부의 이미지 패치 영역에만 셀프 어텐션을 수행하는 단점이 있음

→ 이를 보완: SW-MSA는 윈도 사이의 연결성을 구축해 윈도 간의 관계성 정보를 추출

- 윈도를 가로 또는 세로 방향으로 이동한 후, 인접한 윈도 간의 셀프 어텐션을 계산하는 방식

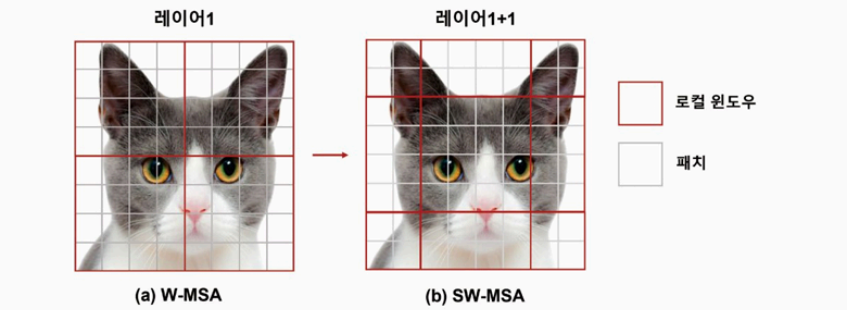

그림 10.10 W-MSA와 SW-MSA 어텐션 영역 비교

W-MSA: 로컬 윈도 안에서 패치 간 셀프 어텐션 수행

→ 로컬윈도우가 이동하면서 각각의 윈도에서 연산을 수행

- 효율적인 배치 계산을 통해 연산량을 감소시킴

SW-MSA: 로켈 윈도 간의 셀프 어텐션을 수행하기 위해 W-MSA에서 사용된 윈도 개수가 더 많아 셀프 어텐션 연산을 요구

→ 순환 시프트(Cyclic Shift) & 어텐션 마스크(Attetion Mask) 사용 → 역순환 시프트 (Reverse Cyclic Shift)

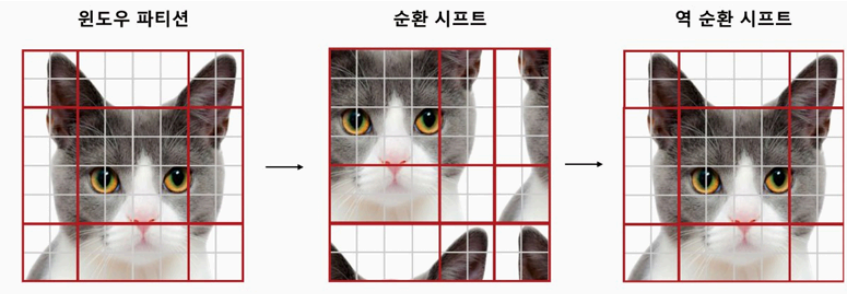

그림 10.11 순환 시프트를 활용한 SW-MSA 어텐션 계산 과정

📖 순환 시프트는 로컬 윈도가 이동할 때 이동한 위치의 정보를 이전 위치에서 가져오는 것

- 윈도 사이즈 M보다 작은 M/2 만큼 로컬 윈도를 이동시킴
- 이동된 위치에서 연산을 수행하지 않도록 어텐션 마스크를 사용해 방시함

어텐션 마스크는 로컬 윈도가 이동한 위치에서 연산을 수행하지 않게 하는 역할

→ 역순환 시프트를 사용해 원래 로컬 구조로 복원

: SW-MSA 방식에서는 순환 시프트와 함께 어텐션 마스크를 사용하여 로컬 윈도 간의 셀프 어텐션 연산을 효율적으로 수행함

- 이때 어텐션 마스크 사용 → 순환 시프트를 통해 이동된 윈도에 대한 정보 보존 + 불필요한 연산을 줄임

> 기존 ViT와 다른 점은 스윈 트랜스포머에서 사용한 셀프 어텐션은 상대적 위치 편향(Relative Position Bias)을 고려한다는 것이다.

📖 **상대적 위치 편향**

: 로컬 윈도 안에 있는 패치 간의 상대적 거리를 임베딩

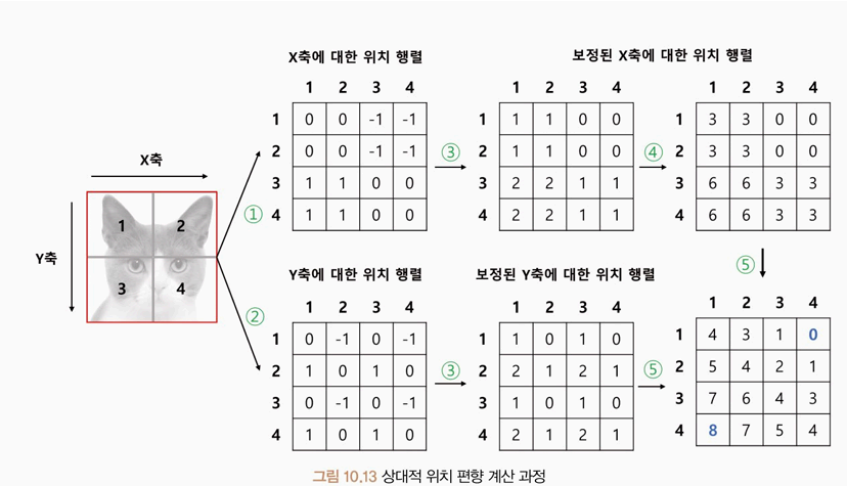

그림 10.13 상대적 위치 편향 계산 과정

: 이미지 패치들이1~4까지 주어진다면 패치 간의 상대적 거리는 X축과 Y축 두가지 방식으로 표기 가능

- X축 방식: 두 패치 간의 가로 방향 거리 - 같은 X축에 있으면 상대적 거리가 0, 오른쪽으로 떨어져 있으면 -1
- Y축 방식: 세로 방향 거리: 같은 y축에 놓여있으면 0, 오른쪽으로 떨어져 있으면 -1, 왼쪽 1

```
## 상대적 위치 편향 계산
import torch

# window_size는 여러 개의 패치를 포함하는 격자의 크기
# 윈도 내외부의 패치를 구분해주는 역할
window_size = 2
coords_h = torch.arange(window_size)
```

```
coords_w = torch.arange(window_size)

# meshgird함수: coords_h 배열 값(i)과 coords_w배열 값(j)으로 사각형 격자 (ij)를 만
# stack: (X,Y) 쌍 배열을 만듦
# flatten: 각 X,Y추겡 대한 위치 인덱스 생성
coords = torch.stack(torch.meshgrid([coords_h, coords_w], indexing="ij"))
coords_flatten = torch.flatten(coords,1)
relative_coords = coords_flatten[:,:,None] - coords_flatten[:,None,:]
print(relative_coords)
print(relative_coords.shape)

## X,Y 축에 대한 위치 행렬
x_coords = relative_coords[0,:,:]
y_coords = relative_coords[1,:,:]

x_coords += window_size -1
y_coords += window_size -1
x_coords *= 2*window_size -1
relatvie_position_index = x_coords + y_coords

## X,Y 축에 대한 상대적 위치 좌표 변환

# MSA를 계산할 때 사용된 헤드 수
num_heads = 1

# 헤드별로 0~8번(9개)의 상대적 위치 편향을 가지는 테이블
relative_position_bias_table = torch.Tensor(
   torch.zeros((2*window_size-1)*(2*window_size-1), num_heads)
)
relative_position_bias = relative_position_bias_table[relative_position_index.vie
relative_position_bias = relative_position_bias.view(
   window_size*window_size, window_size*window_size-1
)
```

수식 10.2 상대적 위치 편향을 고려한 셀프 어텐션

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

- Q 쿼리, K 키, Value 값, d 벡터들의 임베딩 차원

실습

CvT

📖 CvT(Convolutional Vision Transformer)

: 합성곱 신셩망 구조에서 활용하는 계층형 구조(Hierarchical Architecture)를 ViT에 적용한 모델

- ViT의 셀프어텐션
  - 이미지 정보를 바탕으로 각 패치간의 관계를 학습
  - 모든 패치에 대해 동일한 크기와 간격을 사용 → 이미지의 물체 크기나 해상도를 계층적으로 학습하기 어려움
- 스윈 트랜스포머
  - 트랜스포머 블록마다 다른 로컬 윈도 크기를 사용해 이미지의 계층적 특징을 학습
- CvT
  - 합성곱 신경망에서 사용하는 계층적 구조를 ViT에 적용해 이미지의 지역 특징(Local Feature)와 전역 특징(Global Feature)을 모두 활용해 비교적 적은 개수의 매개변수로 높은 성능을 보임
  - 합성곱 계층의 지역 특징을 추출
  - ViT의 셀프 어텐션 계층으로 전역 특징을 결합
  - 스윈트랜스포머보다 더 적은 수의 매개변수로도 비슷한 수준의 성능

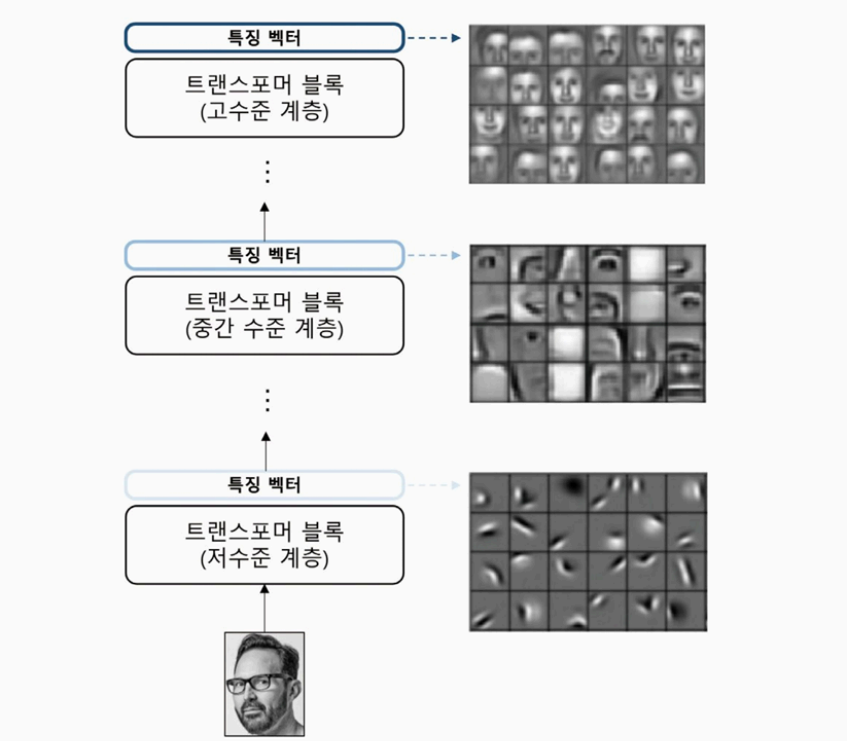

그림 10.14 CvT 모델의 계층적 특징 구조 예시

: 얼굴 이미지가 입력으로 사용된다면

1. 저수준 (Low-level) 계층에서 선 또는 점에 대한 특징을 표현
2. 중간 수준(Mid-level)계층에서 눈이나 귀와 같은 특징 표현
3. 고수준(High-level) 계층에서는 얼굴 구조에 대한 특징 표현

→ CvT의 계층형 구존는 물체의 크기나 해상도를 더 잘 표현

- ViT는 이미지를 패치 단위로 분할, 트랜스포머의 입력으로 사용, 패치를 겹치지 않게 처리해 모델이 전체정보 학습
    - 패치 임베딩과 겹치지 않는 선형 임베딩 사용

↔ CvT는 이미지 처리 전에 합성곱 연산 → 특징맵 추출 → 유의미한 패턴 추출 —> 일정한 크기의 패치로 나눠 트랜스포머 모델 입력으로 사용, 입력되는 패치들을 겹쳐 학습

- 겹치는 합성곱 임베딩, 계층형 구조

합성곱 토큰 임베딩

📖 Convolutional Token Embedding

: 2D로 재구성된 토큰 맵에서 중첩 합성곱 연산을 수행하는 임베딩

ViT에 계층적인 합성곱 신경망의 성질을 추가하기 위해 사용

- **기존 트랜스포머 모델**
  - 이미지의 특징을 추출하기 위해 9개의 이미지 패치를 사용
  - 선형 임베딩 → 쿼리, 키, 값 임베딩으로 변환
  - 임베딩을 사용해 셀프 어텐션을 계산하고 출력 생성 ⇒ 이미지의 근본적인 2D 모양의 구조를 잃게됨

↔ CvT모델

- 이미지의 2D 구조를 보존& 셀프 어텐션 사용
  - 이미지 패치를 2D 합성곱 임베딩을 사용해 쿼리, 키, 값 임베딩으로 변환 → 이 임베딩을 사용해 셀프 어텐션을 계산하고 출력을 생성함

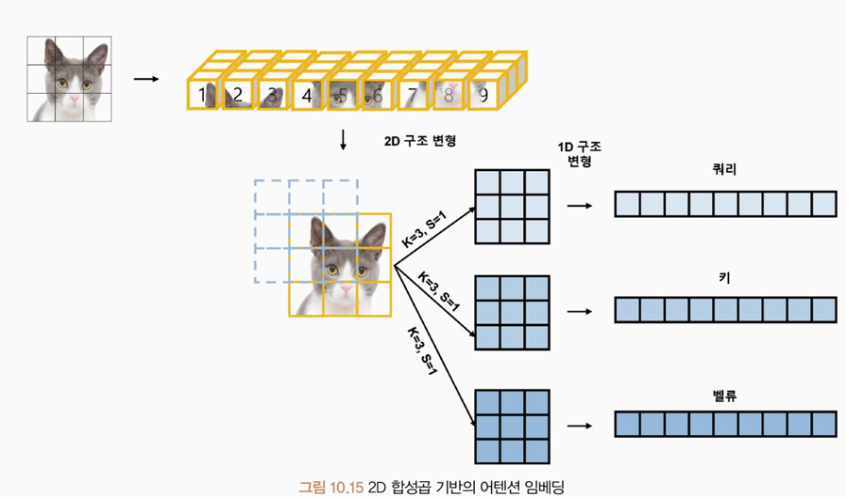

그림 10.15 2D 합성곱 기반의 어텐션 임베딩

: 이미지에 3*3 커널 크기(K), 1 간격(S)의 커널을 적용하면, 패치 크기는 3*3으로 생성됨

→ 각각의 패치를 쿼리, 키, 값에 대해 합성곱 연산을 적용 & 1차원 구조로 재배열해 기존 셀프 어텐션 방법으로 학습

- 지역 특징을 학습 O
- 시퀀스 길이를 줄이는 동시에 토큰 특징의 차원을 증가시킴
- 합성곱 모델에서 수행되는 방식처럼 다운 샘플링과 특징 맵의 수를 늘리게 됨

어텐션에 대한 합성곱 임베딩

📖 Convolutional Projection for Attention

: 2D로 재구성된 토큰맵에서 분리 가능한 합성곱 연산을 사용해 성능 저하 및 계산 복잡도를 낮추는 연산

- 매개 변수를 줄이기 위해 쿼리, 키, 값 임베딩은 차원을 축소시킴

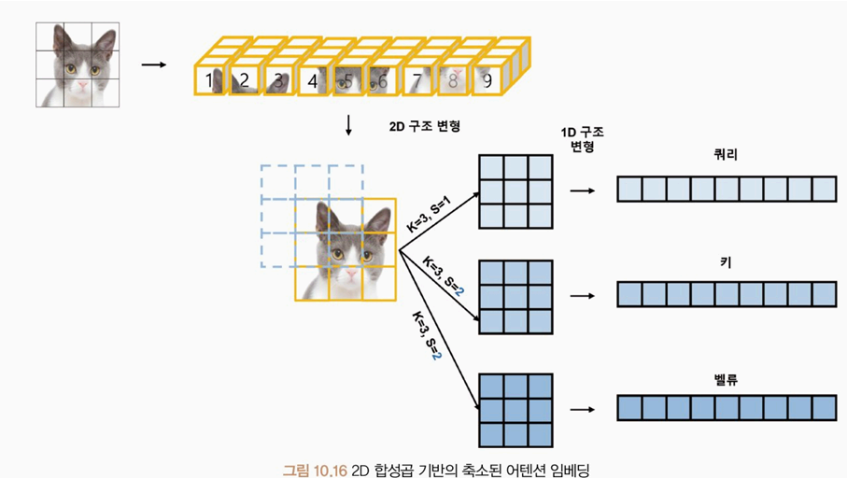

그림 10.16 2D 합성곱 기반의 축소된 어텐션 임베딩

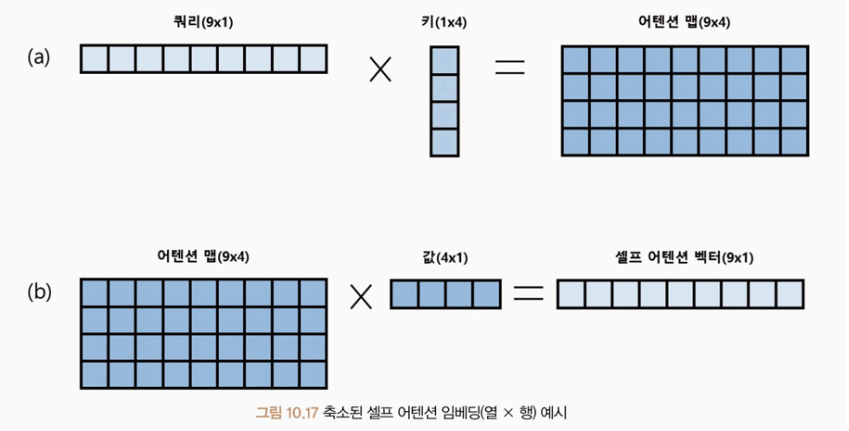

그림 10.17 축소된 셀프 어텐션 임베딩(열 × 행) 예시

(a)에서는 9개의 쿼리 패치 임베딩과 4개의 키 패치 임베딩에 대한 중요도를 의미하는 어텐션 맵을 생성

→(b)와 같이 값 패치 임베딩으로 쿼리 어텐션 맵에 대한 가중치를 반영

⇒ 입력 벡터와 출력 벡터의 패치 길이가 동일함 (연산량 감소)

> CvT 모델은 트랜스포머 계층이 깊어질수록 패치 간의 관계를 학습하는 길이를 축소시키는 대신에, 백터의 차원을 증가시켜 모델의 매개변수 수를 대폭 감소시킨다.

(ex) 첫 번째 트랜스포머 블록의 이미지 패치 크기가 64*64, 임베딩 차원이 128이고 마지막 트랜스포머 블록의 이미지 패치 크기를 4배 감소시켜 8*8로 설정했다면, 기존 임베딩 차원도 4배 증가시켜 512 차원으로 증가시킴

→ 이미지 패치 크기가 줄어든 만큼 임베딩 차원을 늘리는 방법을 선택해 네트워크 안정성을 향상시키고 모델 설계를 단순화