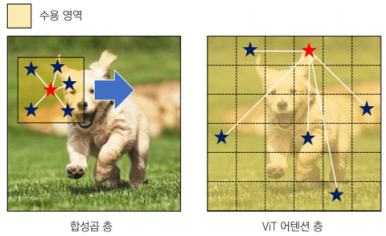
비전 트랜스포머

- **트랜스포머**

  어텐션

  - 어텐션은 딥러닝 모델이 입력 데이터의 다른 부분에 다른 가중치를 부여하도록 하는 기술
  - 어텐션 메커니즘을 통해 입력 데이터의 중요한 부분에 집중함과 동시에 합성곱 신경망처럼 지역으로 한정된 수용 공간(receptive filed)이 아니라 좀 더 전역으로 수용 공간을 가져 학습 과정에서 이미지 전체를 참조

▼ **그림 1 합성곱 층과 어텐션 층의 수용 영역 비교**



1

비전 트랜스포머

- **트랜스포머**
  - 수용 영역은 입력 이미지에서 특정 뉴런의 출력에 영향을 미치는 영역을 말하며, 이는 신경망이 어떤 정보를 집중적으로 처리하는지를 나타냄
  - 합성곱 신경망과 ViT는 이 수용 영역을 다루는 방식에서 중요한 차이점을 가지며, 이 차이는 각각의 모델이 내재한 귀납적 편향(inductive bias)에서 비롯됨
  - 이는 CNN이 이미지의 지역적 패턴과 텍스처를 효율적으로 인식하도록 설계되었음을 의미
  - CNN의 수용 영역은 일반적으로 작고, 이웃하는 뉴런 사이에 겹치며, 이는 모델이 지역적인 정보를 잘 추출할 수 있도록 함
  - 이런 설계는 전체 이미지를 통한 글로벌 패턴 인식에는 제약을 줄 수 있음
  - CNN은 계층을 거치면서 점점 더 넓은 영역의 정보를 통합하지만, 초기 계층에서의 이러한 귀납적 편향 때문에 글로벌 컨텍스트를 학습하는 데 상대적으로 비효율적일 수 있음

2

비전 트랜스포머

- **트랜스포머**

  셀프 어텐션

  - 트랜스포머 모델에서는 여러 어텐션 매커니즘 중 셀프 어텐션(self-attention)을 활용
  - 셀프 어텐션은 주어진 입력에 대해 내부적으로 서로 다른 위치들 간에 어떤 관계가 있는지를 학습하는 방법
  - 셀프 어텐션은 '어텐션' 방식을 사용할 때 문장이 길어질수록 성능이 낮아지는 문제를 해결
  - 데이터를 순환 신경망처럼 순차적으로 처리할 필요가 없으며, 문장 전체를 병렬 구조로 번역해 멀리 있는 단어까지도 연관성을 만들 수 있음
  - 셀프 어텐션은 입력 시퀀스의 모든 요소를 동시에 고려하여 각 요소의 중요도를 계산하는 방법
  - 각 입력 요소는 쿼리(query), 키(key) 및 값(value)으로 표현

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

3

비전 트랜스포머

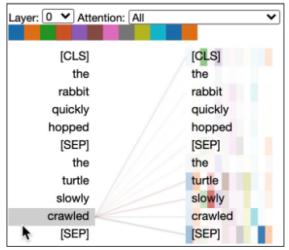
- **트랜스포머**
  - 여기서 Q, K, V는 각각 쿼리, 키, 값에 해당하는 행렬이며, dk는 키의 차원
  - 소프트맥스 함수는 쿼리와 각 키의 유사도를 계산한 뒤, 이를 확률 분포로 변환하여 각 값에 대한 가중치로 사용
    1. **쿼리**: 어텐션 메커니즘의 입력으로 사용되는 쿼리는 주로 새로운 입력 요소에 대해 얻으려는 정보를 나타내는 벡터
       쿼리는 어텐션의 '질문' 역할을 수행하며, 어떤 부분에 주의를 기울여야 하는지를 결정
    2. **키와 값**: 주로 입력 시퀀스의 요소들이 키와 값 쌍으로 구성되며, 쿼리와 비교되고 합산
       키는 주의할 대상을 식별하는 역할을 하고, 값은 해당 키에 대한 정보를 가지고 있음
    3. **점수**: 쿼리와 키 사이의 유사도를 측정하는 점수는 어텐션 메커니즘의 핵심
       일반적으로 내적(dot product), 유클리드 거리 등과 같은 계산 방법을 사용하여 점수를 얻을 수 있음
       결과적으로 어떤 요소들이 서로 더 관련이 깊은지를 나타냄

4

비전 트랜스포머

- **트랜스포머**
  - 다음 그림은 자연어 처리 모델에서 어텐션 가중치가 계산되는 과정을 시각화해주는 라이브러리인 버트비즈(BertViz)의 결과물
  - 왼쪽에 crawled라는 단어에 대해서 서로 다른 색깔의 가중치로 표현된 것을 볼 수 있음
  - 서로 관련이 있는 단어들끼리는 가중치가 표현되는 색깔이 짙은 모습

▼ **그림 2 BertViz 결과물 예시(출처: https://github.com/jessevig/bertviz)**



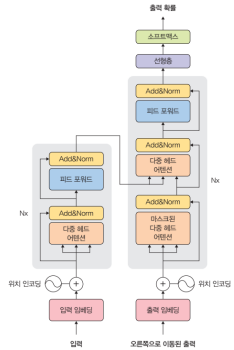
5

비전 트랜스포머

- **트랜스포머**

  트랜스포머 모델 구조

▼ **그림 3 트랜스포머 모델 구조**



6

비전 트랜스포머

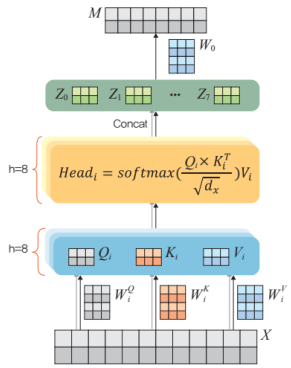
- **트랜스포머**

  다중 헤드 어텐션
  - 다중 헤드 어텐션은 단일 어텐션 메커니즘을 여러 번 병렬로 수행하는 것을 의미
  - 이는 입력 데이터의 서로 다른 특성(subspace)을 동시에 모델링할 수 있게 해줌
  - 구체적으로는 다음과 같은 과정을 거침

7

비전 트랜스포머

▼ **그림 4 다중 헤드 어텐션 과정**



8

비전 트랜스포머

- **트랜스포머**
  1. **선형 변환**: 입력 Q, K, V는 각각 다른 가중치 행렬 W_i^Q, W_i^K, W_i^V 을 사용하여 h번 변환
     이는 데이터를 여러 서브스페이스로 투영하는 효과를 가짐
  2. **병렬 어텐션 계산**: 변환된 Q_i, K_i, V_i에 대해 독립적으로 어텐션 메커니즘을 수행
     각각의 어텐션 결과는 다른 관점에서 입력 데이터를 해석한 것
  3. **결합 및 최종 선형 변환**: 모든 어텐션 헤드의 출력을 결합하고, 추가적인 가중치 행렬 W_o을 사용해 최종 결과를 생성

9

비전 트랜스포머

- **트랜스포머**

  포지셔널 인코딩
  - 포지셔널 인코딩(positional encoding)은 입력 시퀀스의 단어나 토큰이 문장 내에서 차지하는 위치 정보를 모델에 제공하는 중요한 기법
  - 트랜스포머 모델은 기본적으로 순서에 민감하지 않은 구조를 가지고 있기 때문에 문장의 순서 정보를 모델에 알려주기 위해 포지셔널 인코딩이 필요함

  1. **선형 포지셔널 인코딩**

     초기에는 위치 정보를 선형적인 방식으로 인코딩하는 아이디어가 고려되었음
     즉, 첫 번째 위치에는 1, 두 번째 위치에는 2와 같이 각 위치에 대해 선형적으로 증가하는 값을 할당하는 방식
     이 방법은 구현이 간단하고 직관적이라는 장점이 있지만, 몇 가지 문제점을 가지고 있음
     - **규모 민감성**: 시퀀스의 길이가 길어질수록 포지셔널 값이 크게 증가하여, 모델이 위치 정보를 처리할 때 불필요한 편향을 가질 수 있음
     - **일반화 문제**: 모델이 훈련 데이터에서 본 최대 길이를 넘어서는 시퀀스를 처리할 때, 새로운 위치 값에 대한 일반화가 어려움

10

비전 트랜스포머

- **트랜스포머**

  2. **정규화된 선형 포지셔널 인코딩**

     선형 포지셔널 인코딩의 규모 민감성 문제를 해결하기 위해 포지셔널 값에 정규화 과정을 추가하는 방법이 고려되었음
     이는 포지셔널 값의 범위를 제한하여 모델이 위치 정보를 더 일관되게 처리할 수 있도록 함
     이 방법 역시 시퀀스의 길이에 대한 모델의 일반화 능력을 근본적으로 해결하지 못했음

11

비전 트랜스포머

- **트랜스포머**

  3. **사인과 코사인 함수의 도입**

     이러한 문제들을 해결하기 위해 트랜스포머는 사인과 코사인 함수를 사용하여 포지셔널 인코딩을 생성하는 방법을 도입
     - **주기성**: 사인과 코사인 함수는 주기적인 패턴을 가지고 있어 모델이 임의의 길이의 시퀀스를 처리할 때 일관된 방식으로 위치 정보를 인코딩할 수 있음
       이는 모델이 훈련 중에 보지 못한 길이의 시퀀스에 대해서도 일반화하는 능력을 향상시킴
     - **상대적 위치 정보**: 사인과 코사인 함수를 통해 생성된 포지셔널 인코딩은 각 위치 간의 상대적인 차이를 유지할 수 있음
       이는 모델이 단어 간의 상대적인 위치 관계를 더 잘 이해하고 활용할 수 있게 해줌
     - **차원 독립성**: 다양한 주파수의 사인과 코사인 함수를 사용함으로써 모델이 다른 차원에서 위치 정보를 독립적으로 인코딩할 수 있게 함
       이는 모델이 더 복잡한 패턴과 관계를 학습하는 데 도움이 됨

12

비전 트랜스포머

- **트랜스포머**

  포지셔널 인코딩의 원리

  - 포지셔널 인코딩은 각 위치에 대해 고유한 인코딩을 생성하여, 입력 토큰의 임베딩과 합산함으로써 위치 정보를 포함시킴
  - 트랜스포머에서는 주로 사인(sine) 함수와 코사인(cosine) 함수의 조합을 사용하여 이 인코딩을 생성
  - 각 위치 pos와 차원 i에 대한 포지셔널 인코딩은 다음과 같이 정의

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

  - 여기서 pos는 단어의 위치, i는 차원의 인덱스, d_{model}은 모델의 임베딩 차원을 나타냄
  - 이렇게 생성된 포지셔널 인코딩은 각 단어의 임베딩 벡터와 더해져 입력으로 사용

13

비전 트랜스포머

- **트랜스포머**

  포지셔널 인코딩의 특징

  - **순서 정보의 제공**: 포지셔널 인코딩을 통해 트랜스포머 모델은 문장 내에서 단어의 위치 정보를 고려할 수 있게 됨
    이는 문맥 이해에 중요한 역할을 함
  - **학습이 필요 없는 고정된 인코딩**: 포지셔널 인코딩은 모델 학습 과정에서 학습되는 파라미터가 아니라, 미리 정의된 함수에 의해 생성
    이는 추가적인 학습 부담 없이 위치 정보를 모델에 통합할 수 있게 해줌
  - **길이 제한**: 포지셔널 인코딩은 미리 정의된 최대 시퀀스 길이에 의존
    모델이 처리할 수 있는 입력의 최대 길이가 제한
    이를 극복하기 위한 연구도 활발히 이루어지고 있음

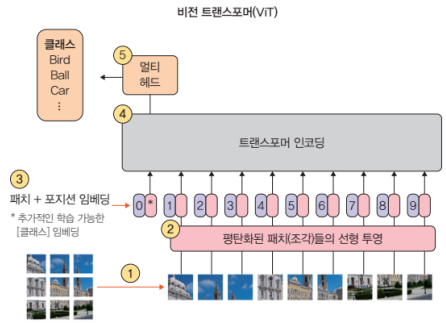
14

비전 트랜스포머

- **비전 트랜스포머**

  ViT

  ▼ 그림 5 ViT의 모델 구조



15

비전 트랜스포머

- **트랜스포머**
  - 기존의 합성곱 신경망 역시 이미지 처리에 매우 효과적인 모델이지만, ViT는 합성곱 층 없이 완전한 트랜스포머 모델을 기반으로 이미지넷에서 최고 성능을 갱신
    1. 자연어로 이루어진 문장이 아닌 이미지를 처리하기 위해 ViT는 문장을 단어 토큰으로 쪼개듯이 입력 이미지를 N×N의 작은 정사각형 조각(patch)으로 쪼갬
    2. 트랜스포머의 인코더는 1차원 벡터를 입력으로 받기 때문에 N×N 이미지 조각을 입력으로 넣어줄 수 없음
       각 조각을 평탄화하여 1차원 벡터로 변환
    3. 기존 셀프 어텐션에서 입력 시퀀스의 단어 순서 정보를 모델에 제공하기 위해 위치 정보인 포지셔널 인코딩(positional encoding)을 제공했던 것과 동일하게 이미지의 위치 정보를 모델에게 제공하기 위해 평탄화된 조각 벡터의 위치 정보를 추가하여 트랜스포머 인코더로 주입
       이때 ViT는 분류 작업을 위해 입력 시퀀스의 맨 앞에 특별한 [CLS] 토큰을 추가
       이 토큰은 모델을 통과하며 전체 이미지에 대한 정보를 집약

16

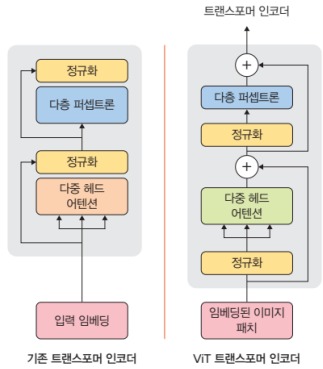
비전 트랜스포머

- **트랜스포머**
  - ❹ 임베딩된 이미지 조각은 ViT의 트랜스포머 인코더 블록의 입력으로 들어옴
    인코더 블록에서 다중 헤드 어텐션을 통하여 서로 다른 이미지 조각에서의 관련성을 학습
    기존 트랜스포머 인코더와 비교했을 때, 정규화의 위치가 바뀐 것을 제외하고는 큰 차이가 없음

17

비전 트랜스포머

▼ **그림 6 ViT의 트랜스포머 인코더와 기존 인코더 비교**



18

비전 트랜스포머

- **트랜스포머**
  - ❺ ViT에서 마지막 MLP(Multi-Layer Perceptron) Head는 모델의 최종 출력을 생성하는 역할
    앞선 단계에서 이미지를 여러 개의 조각으로 나누고, 이 조각들을 트랜스포머 인코더에 입력하여 이미지에 대한 풍부한 특징을 추출
    트랜스포머 인코더를 통과한 [CLS] 토큰의 출력은 이미지 전체를 대표하는 고차원 특징 벡터가 됨
    이 벡터가 MLP의 입력으로 사용
    이 과정에서 학습된 특징들은 모델의 깊은 층을 통해 전달되며, 최종적으로 MLPHead를 통해 이미지 분류에 필요한 출력 형태로 변환

19

비전 트랜스포머

- **트랜스포머**

  ViT 모델 구현 실습
  - 지금부터 ViT 모델을 실제로 구현하는 실습을 진행
  - 다음 코드에서 모델 학습에 필요한 라이브러리를 불러옴

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers
```

20

비전 트랜스포머

- **트랜스포머**
  - cifar100 데이터 세트를 불러옴
  - 데이터 세트를 잘 불러왔는지 간단하게 시각화하여 확인

```
(train_x, train_y), (test_x, test_y) = tf.keras.datasets.cifar100.load_data()
num_classes = 100

plt.figure(figsize=(10, 2))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.imshow(train_x[i])
    plt.title(f"Label: {train_y[i][0]}")
    plt.axis('off')
plt.show()
```

21

비전 트랜스포머

- **트랜스포머**
  - 다음으로 정규화를 진행

```
print(f"스케일링 전 픽셀의 최대 값과 최소 값: {train_x.min()} ~ {train_x.max()}")
train_x = train_x/255.
test_x = test_x/255.
print(f"스케일링 후 픽셀의 최대 값과 최소 값: {train_x.min()} ~ {train_x.max()}")
```

```
스케일링 전 픽셀의 최대 값과 최소 값: 0 ~ 255
스케일링 후 픽셀의 최대 값과 최소 값: 0.0 ~ 1.0
```

22

비전 트랜스포머

- **트랜스포머**
  - 모델의 파라미터를 정의

```
input_shape = (32, 32, 3)
batch_size = 64

image_size = 72
patch_size = 6
num_patches = (image_size // patch_size) ** 2

learning_rate = 1e-3
weight_decay = 1e-4
epochs = 30

transformer_layers = 4
projection_dim = 64
num_heads = 4
transformer_units = [projection_dim * 2, projection_dim]
mlp_head_units = [2048, 1024]
```

23

비전 트랜스포머

- **트랜스포머**
  - 다음은 각 변수에 대한 설명
    - input_shape: 입력되는 이미지 사이즈
    - batch_size: 한 번에 모델이 학습하는 데이터 샘플의 개수인 배치 사이즈
      트랜스포머는 대규모 병렬 처리가 용이하므로 배치 사이즈를 키워보는 것도 좋음
    - image_size: 패치 분할을 위해 입력 이미지를 재조정할 사이즈
      입력 이미지 사이즈와 다름
    - patch_size: 이미지를 나눌 패치의 사이즈
      각 이미지는 patch_size x patch_size 사이즈의 패치로 분할
    - num_patches: 이미지에서 추출될 패치의 총 개수
      이는 이미지 사이즈를 패치 사이즈로 나눈 몫의 제곱으로 계산
    - weight_decay: 가중치 감소는 오버피팅을 방지하기 위해 사용되는 규제 기법 중 하나
      이는 학습 과정에서 가중치의 사이즈를 제한하여, 모델의 복잡도를 줄임
    - transformer_layers: 트랜스포머 내부의 인코더 레이어 수
      더 많은 레이어는 모델의 복잡도와 학습 능력을 증가시키지만, 계산 비용이 더 많이 듦

24

비전 트랜스포머

- **트랜스포머**
  - projection_dim: 패치를 투영할 때의 차원 수
    이는 트랜스포머의 내부 차원과 관련이 있음
  - num_heads: 멀티 헤드 어텐션에서의 헤드 수
    여러 개의 헤드를 사용하면 모델이 다양한 정보를 병렬로 처리할 수 있음
  - transformer_units: 트랜스포머의 피드포워드 네트워크에서 사용되는 유닛의 수
    이 배열은 각 레이어에서의 유닛 수를 나타냄
  - mlp_head_units: 모델의 최종 부분에 위치하는 다층 퍼셉트론(MLP)의 유닛 수
    앞의 코드에 따르면 dense_layer(2048), dense_layer(1024) 두 층이 쌓임(다음에 나오는 다층 퍼셉트론 mlp 함수를 참조)

25

비전 트랜스포머

- **트랜스포머**
  - 다음으로 트랜스포머 인코더의 입력으로 넣어줄 이미지 조각을 만들어주는 클래스를 정의

```
class PatchTokenization(layers.Layer):
    def __init__(self, image_size=image_size, patch_size=patch_size, num_patches=num_
patches, projection_dim=projection_dim, **kwargs):
        super().__init__(**kwargs)
        self.image_size = image_size
        self.patch_size = patch_size
        self.half_patch = patch_size // 2
        self.flatten_patches = layers.Reshape((num_patches, -1))
        self.projection = layers.Dense(units=projection_dim)
        self.layer_norm = layers.LayerNormalization(epsilon=1e-6)
```

26

비전 트랜스포머

- **트랜스포머**
  - 이미지를 패치로 분할하고, 이 패치들을 트랜스포머 모델의 토큰으로 변환하는 PatchTokenization클래스
  - ViT 모델에서 중요한 첫 단계는 입력 이미지를 작은 패치들로 나누고, 각 패치를 벡터로 평탄화한 후, 이를 투영(projection)하여 트랜스포머의 입력 토큰으로 사용하는 것
    - **이미지 사이즈, 패치 사이즈, 패치 수, 투영 차원**: 이 클래스의 생성자(init 메서드)에서는 이미지 사이즈(image_size), 패치 사이즈(patch_size), 패치의 총 수(num_patches), 투영 차원(projection_dim)을 초기화
    - **패치 추출**: call 메서드 내에서 tf.image.extract_patches 함수를 사용해 입력 이미지를 패치로 분할
      이 함수는 이미지에서 지정된 사이즈(patch_size)와 간격(strides)으로 패치를 추출
      padding="VALID"는 이미지 가장자리를 잘라내어 정확히 패치가 맞도록 조정

27

비전 트랜스포머

- **트랜스포머**
  - **패치 평탄화**: 추출된 패치들은 flatten_patches 층(layers.Reshape)을 통해 평탄화
    각 패치는 하나의 긴 벡터로 변환되어, 트랜스포머 모델이 처리할 수 있는 형태로 만들어짐
  - **패치 투영**: 평탄화된 패치들은 projection 층(layers.Dense)을 통해 주어진 투영 차원(projection_dim)으로 투영
    이는 각 패치를 고정된 사이즈의 벡터로 매핑하여, 트랜스포머의 입력으로 사용될 수 있게 함
  - **레이어 정규화**: 선택적으로 layer_norm(layers.LayerNormalization)은 투영된 패치 토큰들을 정규화하여 모델 학습을 안정화하고 가속화할 수 있음
    여기서는 사용되지 않았지만, 일반적인 비전 트랜스포머 아키텍처에서는 입력 토큰에 레이어 정규화를 적용하는 것이 일반적임

28

비전 트랜스포머

- **트랜스포머**
  - 시각화를 통하여 이미지가 패치 단위로 쪼개진 것을 확인

```
image = train_x[np.random.choice(range(train_x.shape[0]))]
resized_image = tf.image.resize(tf.convert_to_tensor([image]), size=(image_size,
image_size))

(token, patch) = PatchTokenization()(resized_image)
(token, patch) = (token[0], patch[0])
n = patch.shape[0]
count = 1
plt.figure(figsize=(4, 4))
for row in range(n):
    for col in range(n):
        plt.subplot(n, n, count)
        count = count + 1
        image = tf.reshape(patch[row][col], (patch_size, patch_size, 3))
        plt.imshow(image)
        plt.axis("off")
plt.show()
```

29

비전 트랜스포머

- **트랜스포머**

30

비전 트랜스포머

- **트랜스포머**
  - 이 코드는 train_x 데이터 세트에서 무작위로 선택한 이미지를 사용하여 PatchTokenization 레이어의 작동 과정을 시각화
  - PatchTokenization 레이어는 이미지를 여러 패치로 분할하고, 각 패치를 트랜스포머 모델에 적합한 형태로 변환하는 역할
  - 이 코드는 분할된 패치들을 추출하고, 각각의 패치를 개별 이미지로서 시각화하는 과정을 보여줌
    1. **이미지 선택과 리사이징**: train_x에서 무작위로 이미지를 하나 선택하고, tf.image.resize 함수를 사용하여 지정된 image_size로 이미지 사이즈를 조정
       이는 PatchTokenization 레이어에 입력하기 전에 필요한 사전 처리 단계
    2. **패치 토큰화**: 조정된 이미지를 PatchTokenization 레이어에 전달하여, 이미지를 패치로 분할하고 각 패치를 투영
       이 과정에서 두 개의 출력을 받음
       투영된 토큰(token)과 원래의 패치(patch)

31

비전 트랜스포머

- **트랜스포머**
    3. **패치 시각화**: 분할된 패치들을 시각화하기 위해, 각 패치를 patch_size에 맞게 재구성하고 plt.imshow를 사용하여 시각화
       이를 통해 PatchTokenization 레이어가 이미지를 어떻게 여러 개의 작은 부분으로 나누는지 확인할 수 있음
    4. **시각화 설정**: plt.figure를 사용하여 시각화의 전체 사이즈를 설정하고, for 반복문을 통해 모든 패치를 순회하며 각각을 서브 플롯에 배치
       plt.axis("off")는 각 서브 플롯 주변의 축을 숨겨 깔끔한 시각화를 도모함

32

비전 트랜스포머

- **트랜스포머**
  - 다음은 패치 인코더

```
class PatchEncoder(layers.Layer):
    def __init__(self, num_patches=num_patches, projection_dim=projection_dim,
**kwargs):
        super().__init__(**kwargs)
        self.num_patches = num_patches
        self.position_embedding = layers.Embedding(input_dim=num_patches, output_
dim=projection_dim)
        self.positions = tf.range(start=0, limit=self.num_patches, delta=1)

    def call(self, encoded_patches):
        encoded_positions = self.position_embedding(self.positions)
        encoded_patches = encoded_patches + encoded_positions
        return encoded_patches
```

33

비전 트랜스포머

- **트랜스포머**
  - PatchEncoder 클래스는 비전 트랜스포머 모델에서 패치의 위치 정보를 인코딩하는 레이어를 정의
  - 이 클래스는 이미 투영된 패치에 대한 위치 정보를 추가하여, 모델이 패치의 상대적 또는 절대적 위치를 고려할 수 있도록 함
  - 위치 정보는 트랜스포머 모델이 이미지의 전체 구조를 이해하는 데 중요한 역할을 함
    1. **num_patches**: 이미지가 분할되는 총 패치의 수
       이 값은 PatchTokenization 레이어에서 이미지를 분할할 때 결정
    2. **projection_dim**: 패치가 투영될 때의 차원 수
       이는 패치를 벡터로 변환할 때의 목표 차원을 의미

34

비전 트랜스포머

- **트랜스포머**
  3. **position_embedding**: 위치 임베딩을 위한 layers.Embedding 레이어
     이 레이어는 각 패치의 위치에 대한 학습 가능한 임베딩을 생성
     input_dim은 임베딩을 생성할 위치의 총 수(즉 num_patches)를, output_dim은 각 위치 임베딩의 차원(즉 projection_dim)을 의미
  4. **positions**: 모든 패치 위치에 대한 인덱스를 생성
     tf.range를 사용하여 0부터 num_patches까지의 정수 시퀀스를 생성
     이 시퀀스는 위치 임베딩 레이어에 입력되어 각 위치에 대한 임베딩을 조회

35

비전 트랜스포머

- **트랜스포머**
  - 다음은 다층 퍼셉트론 함수

```
def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = layers.Dense(units, activation=tf.nn.gelu)(x)
        x = layers.Dropout(dropout_rate)(x)
    return x
```

36

비전 트랜스포머

- **트랜스포머**
  - 이 함수는 다층 퍼셉트론(Multi-Layer Perceptron, MLP)을 정의하는 데 사용
  - 여기서 사용된 GELU(Gaussian Error Linear Unit) 활성화 함수와 드롭아웃은 최근 딥러닝 모델, 특히 트랜스포머 기반 모델에서 매우 흔히 사용되는 기법

```
def create_vit_classifier():
    inputs = layers.Input(shape=input_shape)       # ①
    x = layers.Normalization()(inputs)             # ②
    x = layers.Resizing(image_size, image_size)(x) # ②
    (tokens, _) = PatchTokenization()(x)           # ③
    encoded_patches = PatchEncoder()(tokens)       # ④
    for _ in range(transformer_layers):            # ⑤
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
        attention_output = layers.MultiHeadAttention(num_heads=num_heads, key_
dim=projection_dim, dropout=0.1)(x1, x1)
        x2 = layers.Add()([attention_output, encoded_patches])
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=0.1)
        encoded_patches = layers.Add()([x3, x2])
```

37

비전 트랜스포머

- **트랜스포머**

```
    x = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    x = layers.Flatten()(x)
    x = layers.Dropout(0.5)(x)
    x = mlp(x, hidden_units=mlp_head_units, dropout_rate=0.5) # ⑥
    outputs = layers.Dense(num_classes)(x)                    # ⑦
    model = tf.keras.Model(inputs=inputs, outputs=outputs)    # ⑧
    return model
```

38

비전 트랜스포머

- **트랜스포머**

모델의 구성 단계

1. **입력 정의**: layers.Input을 사용하여 모델의 입력 형태를 정의
   input_shape는 이미지 데이터의 형태를 나타냄
2. **데이터 정규화 및 리사이징**: 입력 이미지는 layers.Normalization을 통해 정규화되고, layersResizing을 사용하여 지정된 image_size로 사이즈 조정
3. **패치 토큰화**: PatchTokenization 레이어를 사용하여 리사이징된 이미지를 패치로 분할하고, 각 패치를 투영
4. **패치 인코딩**: PatchEncoder를 통해 패치의 위치 정보를 인코딩

39

비전 트랜스포머

- **트랜스포머**

5. **트랜스포머 블록**: 지정된 수의 트랜스포머 레이어(transformer_layers)를 순회하면서, 각 레이어에서 멀티 헤드 어텐션(MultiHeadAttention)과 MLP 블록을 포함한 트랜스포머의 인코더 블록을 적용
   이 과정에서 레이어 정규화(LayerNormalization), 어텐션 메커니즘, 그리고 드롭아웃을 사용하여 입력 패치의 특성을 학습
6. **패치 플래트닝 및 최종 MLP**: 트랜스포머 블록의 출력을 평탄화하고, 추가적인 MLP 블록을 적용하여 고차원 특성을 추출
   여기서는 또한 드롭아웃을 사용하여 과적합을 방지
7. **출력 레이어**: layers.Dense를 사용하여 최종 출력 레이어를 정의
   num_classes는 모델이 예측할 클래스의 총 수
8. **모델 구성**: tf.keras.Model을 사용하여 입력부터 출력까지의 모든 레이어를 포함하는 전체 모델을 구성

40

비전 트랜스포머

- **트랜스포머**
  - 다음은 학습률 조정 방법 중 하나인 코사인 감쇠를 구현하기 위한 CosineDecay 클래스

```
class CosineDecay(tf.keras.optimizers.schedules.LearningRateSchedule):
    def __init__(self, learning_rate_base, total_steps, warmup_learning_rate, warmup_
steps):
        super().__init__()
        self.learning_rate_base = learning_rate_base
        self.total_steps = total_steps
        self.warmup_learning_rate = warmup_learning_rate
        self.warmup_steps = warmup_steps
        self.pi = tf.constant(np.pi)

    def __call__(self, step):
        if self.total_steps < self.warmup_steps:
            raise ValueError("total_steps 값이 warmup_steps보다 크거나 같아야 합니다.")
```

41

비전 트랜스포머

- **트랜스포머**

```
        cos_annealed_lr = tf.cos(self.pi * (tf.cast(step, tf.float32) - self.warmup_
steps) / float(self.total_steps - self.warmup_steps))
        learning_rate = 0.5 * self.learning_rate_base * (1 + cos_annealed_lr)

        if self.warmup_steps > 0:
            if self.learning_rate_base < self.warmup_learning_rate:
                raise ValueError(
                    "learning_rate_base 값이 warmup_learning_rate보다 크거나 같아야
합니다.")
            slope = (self.learning_rate_base - self.warmup_learning_rate) / self.
warmup_steps
            warmup_rate = slope * tf.cast(step, tf.float32) + self.warmup_learning_
rate
            learning_rate = tf.where(step < self.warmup_steps, warmup_rate, learning_
rate)
        return tf.where(step > self.total_steps, 0.0, learning_rate, name="learning_
rate")
```

42

비전 트랜스포머

- **트랜스포머**
  - 앞에서 선언한 CosineDecay 스케줄러 인스턴스를 생성

```
total_steps = int((len(train_x) / batch_size) * epochs)
warmup_epoch_percentage = 0.10
warmup_steps = int(total_steps * warmup_epoch_percentage)
scheduled_lrs = CosineDecay(
    learning_rate_base=learning_rate,
    total_steps=total_steps,
    warmup_learning_rate=0.0,
    warmup_steps=warmup_steps,)
```

43

비전 트랜스포머

- **트랜스포머**
  - 각 인수 값을 살펴보자
    - learning_rate_base: 앞서 선언한 하이퍼파라미터를 그대로 사용
    - total_steps: 전체 학습 과정에서의 스텝 수
      이 값은 전체 데이터 세트 사이즈, 배치 사이즈, 그리고 에포크 수를 기반으로 계산
    - warmup_learning_rate: 웜업 기간 동안의 시작 학습률로, 여기서는 0.0으로 설정
      이는 웜업 기간 동안 학습률을 점진적으로 learning_rate_base까지 증가시킴
    - warmup_steps: 웜업을 위해 할당된 스텝 수
      total_steps의 일정 비율(warmup_epoch_percentage)로 설정

44

비전 트랜스포머

- **트랜스포머**
  - 이제 분류기를 생성하고 모델을 컴파일

```
vit = create_vit_classifier()

optimizer = tf.keras.optimizers.AdamW(
    learning_rate=scheduled_lrs,
    weight_decay=weight_decay)

vit.compile(
    optimizer=optimizer,
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy(name="accuracy"),
             tf.keras.metrics.SparseTopKCategoricalAccuracy(5, name="top-
5-accuracy")])
```

45

비전 트랜스포머

- **트랜스포머**
  - create_vit_classifier 함수를 사용하여 비전 트랜스포머(Vision Transformer, ViT) 모델을 생성하고, 이를 컴파일하는 과정을 완료
  - 모델 컴파일 과정에서는 AdamW 최적화 알고리즘과 코사인 감쇠 학습률 스케줄러(scheduled_lrs), 그리고 가중치 감소(weight_decay)를 사용
  - 이는 모델의 학습 과정을 최적화하고, 과적합을 방지하는 데 도움을 줌
    - AdamW: Adam 최적화 알고리즘에 가중치 감소를 추가한 변형
      이는 학습률을 동적으로 조정하면서도 가중치의 사이즈를 제한하여 과적합을 방지하는 효과를 기대할 수 있음
    - SparseCategoricalCrossentropy(from_logits=True)를 사용하여, 모델의 출력이 소프트맥스 활성화 함수를 거치지 않은 로짓(logit) 값이라고 가정하고, 이를 기반으로 교차 엔트로피 손실을 계산
      이는 다중 클래스 분류 문제에 적합한 손실 함수

46

비전 트랜스포머

- **트랜스포머**
  - **평가지표**(metrics): SparseCategoricalAccuracy와 SparseTopKCategoricalAccuracy(5)를 사용하여, 모델의 정확도와 상위 5개 예측 중 정답이 포함된 비율을 각각 측정
    SparseTopKCategoricalAccuracy는 클래스가 많은 경우 성능을 객관적으로 평가하기에 유리

47

비전 트랜스포머

- **트랜스포머**
  - 모델을 학습

```
history = vit.fit(
    x=train_x,
    y=train_y,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.2)
```

```
Epoch 1/40
625/625 [==============================] - 61s 70ms/step - loss: 4.9585 - accura-
cy: 0.0136 - top-5-accuracy: 0.0625 - val_loss: 4.4873 - val_accuracy: 0.0230 - val_
top-5-accuracy: 0.1044
Epoch 2/40
625/625 [==============================] - 40s 64ms/step - loss: 4.4571 - accura-
cy: 0.0252 - top-5-accuracy: 0.1138 - val_loss: 4.2507 - val_accuracy: 0.0456 - val_
top-5-accuracy: 0.1853
…(중략)…
```

48

비전 트랜스포머

- **트랜스포머**

```
Epoch 39/40
625/625 [==============================] - 42s 67ms/step - loss: 0.3964 - accura-
cy: 0.8783 - top-5-accuracy: 0.9889 - val_loss: 2.8727 - val_accuracy: 0.4234 - val_
top-5-accuracy: 0.7096
Epoch 40/40
625/625 [==============================] - 42s 67ms/step - loss: 0.3772 - accuracy:
0.8839 - top-5-accuracy: 0.9904 - val_loss: 2.8761 - val_accuracy: 0.4251 - val_top-5-
accuracy: 0.7112
```

49

비전 트랜스포머

- **트랜스포머**
  - 모델의 결과를 시각화해보자

```
plt.figure(figsize=(14, 5))

plt.subplot(1, 3, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 3, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
```

50

비전 트랜스포머

- **트랜스포머**

```
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim(0, 1)
plt.legend()

plt.subplot(1, 3, 3)
plt.plot(history.history['top-5-accuracy'], label='Training Top-5 Accuracy')
plt.plot(history.history['val_top-5-accuracy'], label='Validation Top-5 Accuracy')
plt.title('Training and Validation Top-5 Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Top-5 Accuracy')
plt.ylim(0, 1)
plt.legend()
plt.show()
```

51

비전 트랜스포머

- **트랜스포머**

52

비전 트랜스포머

- **트랜스포머**
  - 마지막으로 테스트 데이터 세트에서의 성능을 확인

```
vit.evaluate(test_x, test_y)
```

```
313/313 [==============================] - 5s 17ms/step - loss: 2.7925 - accuracy:
0.4320 - top-5-accuracy: 0.7212
[2.7924680709838867, 0.4320000112056732, 0.7211999893188477]
```

53

비전 트랜스포머

- **트랜스포머**

스윈 트랜스포머

- 스윈 트랜스포머(Swin Transformer)는 ViT의 일부 한계를 효과적으로 해결하는 계층적 트랜스포머 아키텍처를 도입하여 비전 트랜스포머 영역에서 상당한 발전을 이루었음
- 특히 다양한 컴퓨터 비전 작업을 좀 더 효율적이고 효과적으로 처리하는 데 있어 ViT에 비해 몇 가지 혁신적인 기능과 개선 사항이 포함

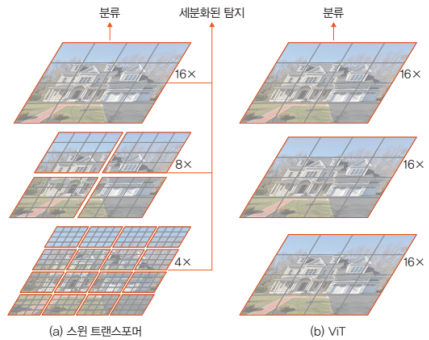
1. **계층적 구조**

스윈 트랜스포머는 여러 스케일로 이미지를 처리하는 계층적 구조를 채택하여 공간 차원은 점차 줄이면서 특징 차원은 늘림
이 접근 방식은 합성곱 신경망(CNN)의 동작을 모방하여 모델이 다양한 해상도에서 특징을 캡처할 수 있도록 하여 다중 스케일 표현이 중요한 물체 감지 및 의미 분할과 같은 작업을 용이하게 함

54

비전 트랜스포머

▼ **그림 7 스윈 트랜스포머와 ViT 비교**



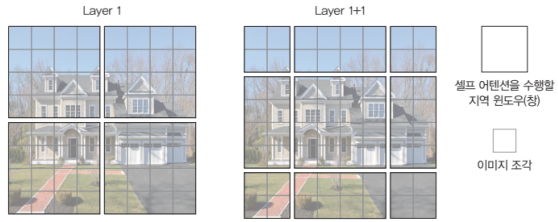
55

비전 트랜스포머

- **트랜스포머**

2. **시프트된 창 파티셔닝**

▼ **그림 8 시프트된 창 파티셔닝**



56

비전 트랜스포머

- **트랜스포머**
  - 스윈 트랜스포머의 핵심 혁신 중 하나는 셀프 어텐션 계산을 위해 시프트 윈도우를 사용한다는 것
  - 연속되는 각 트랜스포머 블록에서 창 분할이 시프트되어 창 간 연결 및 정보 교환이 가능
  - 이 메커니즘은 셀프 어텐션의 계산 복잡성을 이차에서 선형으로 크게 줄여 더 큰 이미지와 데이터 세트에 더 쉽게 확장할 수 있도록 해줌
    - **기본 윈도우 파티셔닝**: 먼저 스윈 트랜스포머는 이미지를 여러 개의 작은 윈도우로 분할
      각 윈도우 내에서만 셀프 어텐션을 계산하여 계산 비용을 크게 줄임
      이는 각 윈도우가 독립적으로 처리되기 때문에 가능
    - **윈도우 이동**: 계층적 구조를 따라 다음 단계에서는 윈도우 분할이 이전 단계와 약간 차이나게 조정
      즉, 윈도우의 분할이 이전 계층과 비교하여 작은 오프셋(offset)을 가지고 '이동(shift)'
      이러한 이동은 인접한 윈도우 간 정보 교환을 가능하게 하며, 전체 이미지에 걸쳐 보다 효과적인 정보 통합을 달성

57

비전 트랜스포머

- **트랜스포머**

  3. 효율성 및 확장성
  - 효율성과 확장성은 컴퓨터 비전 분야에서 널리 채택되는 데 결정적인 역할을 하는 요소
  - 이 두 가지 특성은 스윈 트랜스포머가 다양한 사이즈의 데이터 세트와 다양한 해상도의 이미지를 처리할 수 있게 함
  - 특히 자원이 제한된 환경에서도 뛰어난 성능을 발휘할 수 있도록 함
    - **효율성**(efficiency): 스윈 트랜스포머의 효율성은 주로 시프트된 창 파티셔닝 메커니즘에 의해 달성
      이 방법은 셀프 어텐션 계산의 복잡도를 크게 줄이며, 결과적으로 전체 모델의 계산 비용을 감소시킴
      전통적인 ViT에서와 달리 스윈 트랜스포머는 이미지를 고정 사이즈의 윈도우로 분할하고 각 윈도우 내에서만 셀프 어텐션을 계산하여, 계산량을 줄임
      이러한 접근 방식은 메모리 사용량을 최적화하고, 처리 속도를 향상시키며, 더 큰 이미지와 데이터 세트를 효율적으로 처리할 수 있게 함

58

비전 트랜스포머

- **트랜스포머**
  - **확장성**(scalability): 스윈 트랜스포머의 확장성은 그 계층적 구조에서 비롯됨
    모델은 다양한 해상도의 이미지를 처리할 수 있도록 설계되었으며, 이는 다양한 컴퓨터 비전 작업에 적용될 수 있음
    모델의 사이즈는 필요에 따라 쉽게 조정될 수 있으며, 이는 특히 다양한 자원 제약 조건을 가진 환경에서 모델을 배포할 때 중요
    스윈 트랜스포머는작은 모델에서부터 매우 큰 모델까지, 넓은 범위의 모델 사이즈를 지원하며, 이를 통해 개발자와 연구자들이 특정 작업의 요구 사항과 자원의 가용성에 맞춰 모델을 최적화할 수 있음

59

비전 트랜스포머

- **트랜스포머**

  스윈 트랜스포머 실습
  - 스윈 트랜스포머의 실습에는 오픈 소스로 공개된 tfswin이라는 패키지를 사용

```
!pip install tfswin
```

  - tfswin과 함께 모델 학습과 시각화에 필요한 라이브러리들을 불러옴

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tfswin import SwinTransformerTiny224, preprocess_input
import matplotlib.pyplot as plt
```

  - cifar100 데이터 세트를 불러옴

```
(train_x, train_y), (test_x, test_y) = tf.keras.datasets.cifar100.load_data()
num_classes = 100
```

60

비전 트랜스포머

- **트랜스포머**
  - tfswin 모듈을 사용하여 아래와 같이 간편하게 스윈 트랜스포머 모델을 정의할 수 있음

```
inputs = layers.Input(shape=(32, 32, 3), dtype='uint8')
outputs = layers.Lambda(preprocess_input)(inputs)
outputs = SwinTransformerTiny224(include_top=False)(outputs)
outputs = layers.Dense(num_classes, activation='softmax')(outputs)

swin = models.Model(inputs=inputs, outputs=outputs)
```

  - 다음으로 모델을 컴파일

```
optimizer = tf.keras.optimizers.AdamW()

swin.compile(
    optimizer=optimizer,
    loss='sparse_categorical_crossentropy',
    metrics=[tf.keras.metrics.SparseTopKCategoricalAccuracy(1, name="accuracy"),
        tf.keras.metrics.SparseTopKCategoricalAccuracy(5, name="top-5-accuracy")])
```

61

비전 트랜스포머

- **트랜스포머**
  - 다중 분류를 위하여 sparse_categorical_crossentropy를 손실 함수로 사용하고 Top-1 정확도와 Top-5 정확도를 평가지표로 설정
  - 모델 학습을 진행

```
history = swin.fit(x=train_x, y=train_y, batch_size=128, epochs=10, validation_
split=0.2)
```

```
Epoch 1/10
313/313 [==============================] - 77s 113ms/step - loss: 4.0051 - accura-
cy: 0.0806 - top-5-accuracy: 0.2548 - val_loss: 3.0680 - val_accuracy: 0.2098 - val_
top-5-accuracy: 0.5289
…(중략)…
Epoch 10/10
313/313 [==============================] - 30s 94ms/step - loss: 0.3573 - accuracy:
0.8832 - top-5-accuracy: 0.9924 - val_loss: 2.4540 - val_accuracy: 0.5174 - val_top-5-
accuracy: 0.8009
```

62

비전 트랜스포머

- **트랜스포머**
  - 훈련과 검증 결과를 시각화해보자

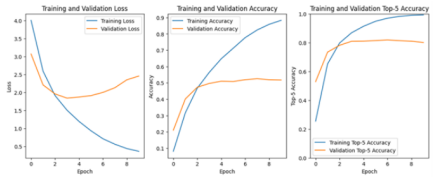
```
plt.figure(figsize=(14, 5))

plt.subplot(1, 3, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 3, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

63

비전 트랜스포머

- **트랜스포머**

```
plt.subplot(1, 3, 3)
plt.plot(history.history['top-5-accuracy'], label='Training Top-5 Accuracy')
plt.plot(history.history['val_top-5-accuracy'], label='Validation Top-5 Accuracy')
plt.title('Training and Validation Top-5 Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Top-5 Accuracy')
plt.ylim(0, 1)
plt.legend()
plt.show()
```



64

비전 트랜스포머

- **트랜스포머**
  - 테스트 데이터로 모델을 평가해보자

```
swin.evaluate(test_x, test_y)
```

```
313/313 [==============================] - 12s 39ms/step - loss: 2.4060 - accuracy:
0.5210 - top-5-accuracy: 0.8115
[2.4060165882110596, 0.5210000276565552, 0.8115000128746033]
```