Vision Transformer

PT

1. Introduce
2. Architecture
3. Training Method
4. Experiments

DELAB 2

PT는 위와 같이 네 파트로 준비하였다. Introduce에서는 ViT의 개념, ViT의 장,단점, ViT가 단점이 있는데도 가치있는 이유 등을 소개하겠다. Architecture는 어떻게 이미지를 트랜스포머로 처리하는지 ViT의 아키텍쳐를 스텝별로 준비하였다. Training Method는 ViT의 훈련 방법에 대해 소개하였고 Experiments는 실험 결과와 함께 모델을 분석해보았다.

Introduce

- An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale
  - 2021년 ICLR에 Accept된 논문
  - Google Research Team에서 ViT라고 불리는 모델을 제안한 논문
  - **ViT:** Vision Transformer

Alexey Dosovitskiy*,†, Lucas Beyer*, Alexander Kolesnikov*, Dirk Weissenborn*, Xiaohua Zhai*, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, Neil Houlsby*,†
*equal technical contribution, †equal advising
Google Research, Brain Team
{adosovitskiy, neilhoulsby}@google.com

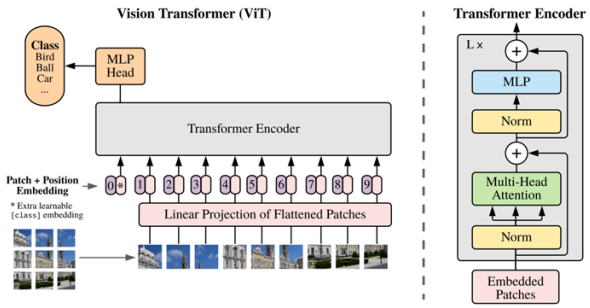
DELAB 3

일단 필자가 소개하려는 논문은 An Image is worth 16x16 words: Transformer for Image Recognition at scale이라는 논문으로 2021년 ICLR에 Accept된 논문이다. 구글 리서치 팀에서 ViT

라고 불리는 모델을 제안한 논문으로 5000회 이상 인용되었다. 중요도가 높은 논문으므로 모델 구조를 상세하게 설명하겠다.

Introduce

- ViT
  - 이미지 분야에서 Attention 기법을 사용할 경우 대부분 CNN과 함께 사용되거나 전체 CNN 구조를 유지하면서 CNN의 특정 구성 요소를 대체하는데 사용됨
  - CNN에 의존하지 않고 이미지 패치의 시퀀스를 입력 값으로 사용하는 Transformer 모델



사실 이미지 분야에서 Attention 기법을 사용할 경우 대부분 CNN과 함께 사용되거나 혹은 전체 CNN 구조를 유지하면서 CNN의 특정 구성 요소를 대체하는데 사용되었다.

ViT는 CNN에 의존하지 않고 이미지 패치의 시퀀스를 입력 값으로 사용하는 Transformer 모델이다. 본 논문 저자들은 Transformer의 Encoder 부분을 가져와 총 세 가지 장점을 얻었다고 한다.

Introduce

- ViT의 장점
  - Transformer 구조를 거의 그대로 사용하므로 확장성이 좋다
  - Large Scale 학습에서 매우 우수한 성능을 보인다 (CNN 성능을 능가함)
  - Transfer Learning 시 CNN보다 적은 계산 리소스를 사용한다

- ViT의 단점
  - Inductive bias의 부족으로 인해 CNN보다 데이터가 많이 요구된다
  - Attention(이미지 전체에서 어느 부분을 주의 깊게 봐야 되는지를 나타내는 기술)
    -> locality 따위의 가정이 없음 -> 전체적인 부분에서 특징을 찾아야 됨
  - 중간 사이즈인 ImageNet을 학습에 사용할 경우 ResNet보다 성능이 낮게 나왔음

1. Transformer 구조를 거의 그대로 사용하기 때문에 확장성이 좋다. (Transformer는 Multi Head

Attention을 사용해서 다양한 패턴을 처리할 수 있으므로 확장성이 좋다. Self attention만 사용했을 때는 특정 task에만 성능이 좋은 경향이 있었다고 한다.)

2. Large Scale 학습에서 매우 우수한 성능을 보인다.

3. Transfer Learning 시 CNN보다 훈련에 더 적은 계산 리소스를 사용한다.

반면 단점은 Inductive bias의 부족으로 인해 CNN보다 데이터가 많이 요구된다.

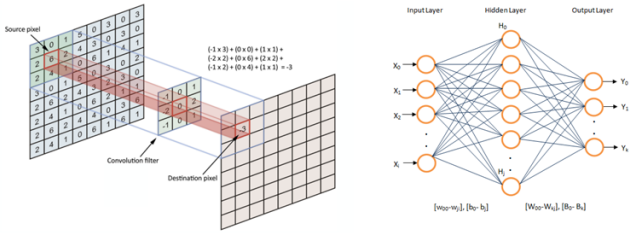
Inductive bias란 '학습자가 처음보는 입력에 대한 출력을 예측하기 위해 사용하는 일련의 가정'이다. 즉 올바른 예측(출력)을 위해서는 학습할 때 어떤 가정이 필요하다고 미리 정해놓은 것이다.

CNN은 대표적으로 translation equivariance랑 locality를 가정한다. 먼저 locality는 convolution 연산을 할 때 이미지의 특정 영역만 보고도 그 안에서 특징을 추출할 수 있다고 가정한 것이다. 예를 들어 사람 얼굴 이미지가 입력으로 들어왔을 때 '코'의 특징을 찾고 싶으면 전체 영역이 아닌 코 주변 영역만으로(코 주변 영역과 합성곱 연산을 함으로써) 특징을 찾을 수 있다는 것이다.

Introduce

- Inductive bias
  - 학습자가 처음보는 입력에 대한 출력을 예측하기 위해 사용하는 일련의 가정
  - CNN은 locality와 translation equivariance를 가정함

- translation equivariance
  - 입력 위치가 변하면 출력도 동일한 위치로 변환되어 나온다
  - **ex)** '동물의 귀'가 좌측 상단 -> Feature 위치도 좌측 상단, '동물의 귀'가 우측 하단 -> Feature 위치도 우측 하단



DELAB 7

두 번째는 translation equivariance이다. 이건 입력 위치가 변하면 출력도 동일한 위치로 변환되어 나온 다는 것이다. 예를 들어 이미지의 입력이 동물이라고 가정하고, 동물의 귀가 좌측 상단에 위치하 면 Feature Map의 좌측 상단 부분에 동물의 귀에 대한 특징이 저장된다. 그런데 만약 이미지에서 동물의 귀가 우측 하단에 위치하면 Feature Map의 우측 하단 부분에 동물의 귀에 대한 특징이 저장된다.

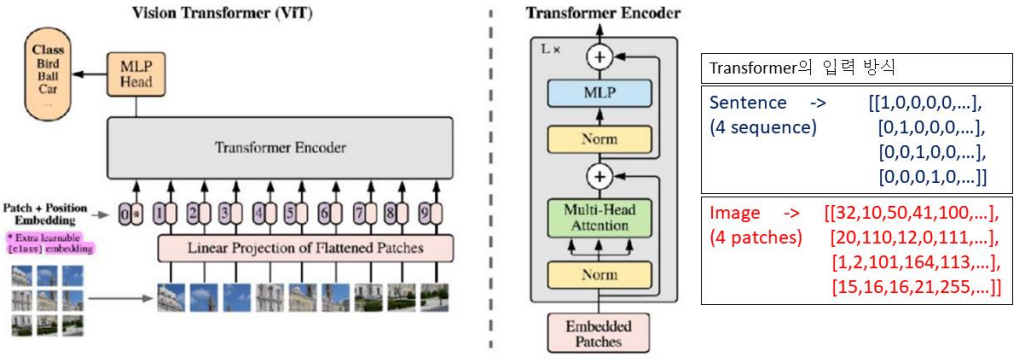
이러한 가정들이 존재하기 때문에 MLP보다 CNN이 영상 처리 성능이 좋은 것이다. 똑같은 예시를 MLP에 대입하자면, 객체의 위치가 바뀌었을 때 입력층의 벡터 값이 달라지고 은닉층과 연결된 가중치들이 달라지기 때문에 은닉층의 벡터 값이 완전히 다르게 나오게 된다.

그러나 다시 돌아가서(5page) Transfomer에서 사용하는 Attention은 전체를 보고 어디가 어떤 지를 말하는 모델(어디가 중요하고 어디가 덜 중요한지 말하는 모델)이지만 어디를 어떻게 보라는 가정이 존재하지 않다. 따라서 전체적인 부분에서(전체를 보고) 패턴을 찾는 것이고 CNN보다 많은 데이터가 필요하다. (비유하자면 CNN은 한 이미지에서 여러 특징을 찾아내지만 Attention은 한 이미지에서 한 가지 특징을 찾아내므로 당연히 많은 데이터가 필요함)

실제로 중간 사이즈인 이미지 넷을 학습에 사용할 경우 ResNet보다 성능이 낮게 나왔다고 한다. 하지만 중요한 점은 Large Scale로 학습을 했더니 강한 Inductive bias를 가진 CNN을 능가했다는 점이 포인트이다. 이건 비전 분야에서 전례가 없었던 사례였기 때문에 ViT가 많은 관심을 받았고 현재는 Backbone을 대체할 정도로 중요도가 높다.

Architecture

- Vision Transformer (ViT)
  - Transforemr의 Encoder를 가져와서 사용 -> 이미지 패치들을 시퀀스 형태로 나열한 뒤 입력으로 줌
  - 클래스를 예측하는 '클래스 토큰' 이 추가됨



DELAB 8

지금부터는 본격적으로 ViT의 구조에 대해 소개하겠다.

ViT는 기본적으로 트랜스포머의 인코더를 가져와서 사용한다. 따라서 트랜스포머에 맞는 입력 값을 만들어서 넣어줘야 한다. 기존 인코더는 시퀀스 모음에 해당하는 행렬을 입력으로 받는다.
예를 들어 NLP분야에서 시퀀스가 4개인 문장이 있을 때 각각의 시퀀스를 원핫벡터로 나타내고 이러한 시퀀스 모음인 행렬을 입력으로 주었다.
본 논문에서 이미지를 처리할 때는, 이미지를 쪼개서 시퀀스 형태로 나열한 뒤 이 모음을 통째로 입력으로 넣어주는 방식을 채 택했다. 예를 들어 이미지를 쪼갰더니 4개의 패치가 나왔다고 하면 각각의 패치를 1차원 벡터로 바꾸고 이러한 벡터 모음인 행렬을 입력으로 주게 된다.

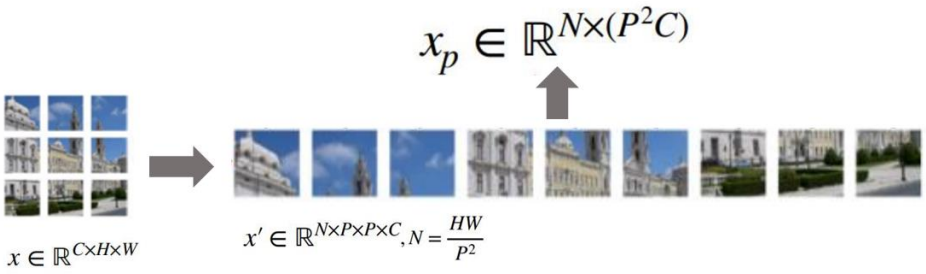
이후 트랜스포머처럼 임베딩 및 Positional Encoding 작업을 하고 인코더 레이어(블록)에 넣어주게 된다.

잘라진 이미지 조각은 패치라고 하는 데 그림에 보이는 것처럼 패치는 정사각형 형태이다. 그래서 Flatten하게 펼치고 Lienar 연산을 해서 Embedding과정을 거치게 된다. 그럼 패치별로 임베딩된 벡터들이 나오는 데 여기다가 클래스를 예측하는 '클래스 토큰' 벡터를 추가한다. 이후 Positional encoding을 하고 인코더 레이어에 들어가는 것이다.

인코더 레이어의 입력과 출력 크기는 같기 때문에 레이어는 L번 반복해서 사용한다. 마지막으로 계산된 부분의 클래스 토큰만 떼서 MLP Head에 넣는다. 결국 클래스 토큰 벡터를 입력으로 받는 MLP를 통해 Classification을 할 수 있다.

Architecture

- Vision Transformer (ViT)
  - **Step 1.** 이미지(x)가 있을 때, 이미지를 PxP 크기의 패치 N개로 분할하여 패치 시퀀스 모음(x_p)을 구축한다



- 이미지 x: C (채널 수) x H (높이) x W (너비)
- 패치 모음 x': N (패치 수) x P (패치 크기) x P (패치 크기) x C (채널 수)
- 패치 시퀀스 모음 x_p: N (패치 수) x P^2C (flattened patch)

DELAB 9

지금부터 각 과정을 순서대로 살펴 보겠다.
먼저 이미지가 입력으로 왔을 때 이미지를 PxP 크기의 패치 N개로 분할하여 패치 시퀀스 모음을 구축한다.

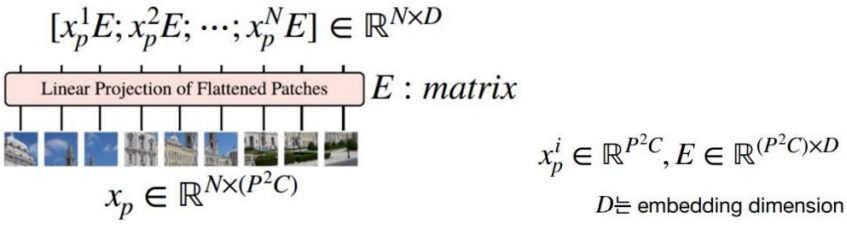
이미지 한 장(x)은 Channel(채널 수) x Height(높이) x Width(너비)라는 크기를 가지게 된다. 예를 들어 컬러 이미지의 높이, 너비가 모두 64라면 3x64x64 크기를 가지게 된다.

이 이미지는 패치들로 쪼개지는데, 패치들(x')은 N(패치의 개수) x P(패치의 크기) x P(패치의 크기) x C(채널 수)로 표현된다. 예를 들어 패치의 크기가 4면 패치 하나는 이미지(64x64)를 배경으로 가로로 16개, 세로로 16개가 생기니 N은 256이 된다. 따라서 패치들은 256x4x4x3으로 표현된다. 이때 N을 구하는 공식은 HW/p2이 성립한다.

패치(PxPxC)를 각각 일자로 피면 P2C 크기인 벡터를 만들 수 있다. 이 벡터들이 N개가 존재하므로 Nx(P2C)가 되고 이 행렬(이미지를 쪼개고 변환된 벡터들 모임, x_p)이 트랜스포머의 입력으로 사용이 된다. 앞의 예시를 그대로 따르면 입력은 256x48이 될 것이다.

Architecture

- Vision Transformer (ViT)
  - **Step 2.** Linear Projection을 통해 x_p의 각 패치를 D차원으로 변환한다 (Embedding)



- 임베딩 필터 E: P^2C x D (사용자가 지정한 차원)
- 패치 시퀀스 모음 x_p (Nx(P^2C)) -- Embedding --> 패치 임베딩 모음 x_pE (NxD)

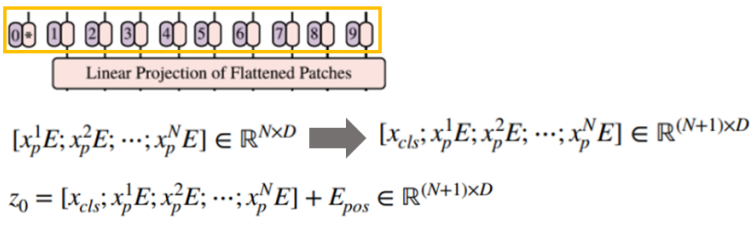
두 번째는 Linear Projection을 통해 X_p의 각 패치를 D차원으로 변환한다.

X_p에 속하는 벡터들 각각을 x_p^1에서 x_p^n으로 나타낸다면 임베딩한 결과는 x_p^1E,..., x_p^nE가 된다는 것을 식으로 나타냈다. E는 임베딩에 사용되는 행렬(임베딩 필터)을 의미하는 데, (P2C)xD라는 크기를 가진다. x_p^i의 크기가 (1xP^2C)이므로 행렬 곱(Linear Projection)을 하면 1xD라는 임베딩 결과가 나오 는 것이다.

이러한 크기가 D인 벡터들이 N개 존재하므로 x_p에 대한 임베딩 결과는 NxD로 표현된다.

Architecture

- Vision Transformer (ViT)
  - **Step 3.** x_pE에 class token을 추가하고 Positional Embedding 을 진행한다
  - class token: 학습 가능한 벡터로써 Encoder를 거쳐서 나왔을 때, 이미지에 대한 Representation Vector 역할을 수행한다.



- 클래스 토큰 x_{cls}: 1 x D
- 위치 임베딩 필터 E_{pos}: (N+1) x D
- 패치 임베딩 모음 x_pE ((N+1) x D) -- Positional Embedding --> 위치 정보가 포함된 패치 임베딩 모음 z_0 ((N+1) x D)

세 번째는 임베딩 결과인 x_p^E에 class token을 추가하고 positional embedding을 진행한다.

클래스 토큰은 학습 가능한 벡터로써 이미지에 대한 인코더를 거쳐서 나왔을 때, 이미지에 대한 1차원 representation vector로써 역할을 한다. (이미지 전체에 대한 특징을 담고 있는 벡터)

패치 임베딩(x_p^iE)이 1xD 크기이므로 클래스 토큰도 1xD가 된다. 이 토큰이 행렬의 맨 앞에 추가가 되므로 결국 (N+1)xD라는 행렬이 만들어진다.

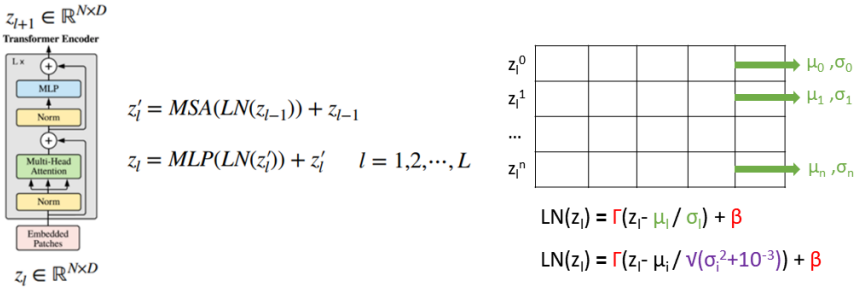
이 행렬의 각각의 행은 이미지 패치라고 볼 수 있는데, 패치들에 대한 위치 정보는 모르는 상태이다. 따라서 크기가 같은 E_{pos}라는 행렬을 더하는 방식으로 positional embedding을 진행한다. Epos는 트랜스포머 설명 때와 마찬가지로 학습 가능한 파라미터이고, 학습이 잘 됐다면 위치 정보 를 잘 나타내게 된다.

이렇게 임베딩으로 차원을 조절하고 positional embedding을 통해 위치 정보까지 추가되면 트랜스 포머 인코더 레이어로 들어갈 조건을 마친다.

앞으로 인코더 레이어의 input을 z0라고 하겠다.

Architecture

- Vision Transformer (ViT)
  - **Step 4.** z_0를 Encoder Layer의 input으로 넣어 Attention을 수행한다 (L:12)



- MSA: Multi-Head Self Attention
- LN: Layer Norm

DELAB 12

네 번째 스텝은 인코더 레이어의 내부 연산 과정에 해당된다.
인코더 레이어는 여러 번 반복돼서 계산이 되므로 zl이 입력으로 들어가면 zl+1이 출력으로 나온다 고 표현한다. 그리고 수식에 보이는 것처럼 zl-1이 입력으로 쓰이고 zl이 나온다고 생각해도 된다.

참고로 z의 크기가 NxD인 것은 그냥 일반화시킨 것이다. 원래는 (N+1)xD가 맞다.

인코더 레이어 구조를 보면 기존 인코더와는 살짝 차이가 있는 데 LayerNorm을 먼저하고 Attention을 계산하는 것이다. 이후 다시 LayerNorm을 하고 MLP를 하게 된다. 그 외 Residual Connection은 그대로 사용된다.

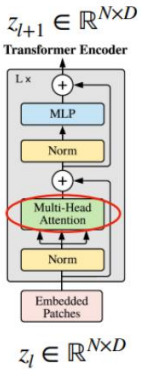
따라서 MSA(LN(zl-1))+z-1을 통해 Attention 결과(zl')를 구하고 MLP(LN(zl'))+zl'를 통해 출력을 만드 는 것이다. 이 때 LayerNorm은 각 feature에 대해서 정규화를 진행하는 데, 다시 말해 z_l의 각 벡터(1xD)에 대해서 진행이 된다.

LN은 트랜스포머와 똑같이 스케일링하고 학습 가능한 두 파라미터인 gamma, beta는 매번 업데이트된다.

다만 표준화 식에서 분모가 0이 되면 안 되므로 분모인 sigma를 sqrt(sigma2+10-3)으로 대체하는 방법을 사용했다고 한다.

Architecture

- Vision Transformer (ViT)
  - Multi-Head Self Attention (MSA)



$$[q,k,v] = zU_{qkv} \qquad U_{qkv} \in \mathbb{R}^{D\times 3D_h}$$

$$q,k,v \in \mathbb{R}^{N\times D_h}$$

$$A = softmax(qk^T/\sqrt{D_h}) \in \mathbb{R}^{N\times N}$$

$$SA(z) = Av \in \mathbb{R}^{N\times D_h}$$

$$MSA(z) = [SA_1(z); SA_2(z); \cdots; SA_k(z)]U_{msa}$$

$$U_{msa} \in \mathbb{R}^{k\cdot D_h\times D}$$

k는 헤드 수

$D_h = D/k$ 로 설정

DELAB 13

멀티 헤드 어텐션(MSA)을 좀 더 자세히 나타내면 위와 같다.

트랜스포머의 Query, Key, Value를 구할 때는 z에 각각 Linear연산을 하는 방식으로 진행이 됐다.

[q=zwq, k=zwk, v=zwv]

w의 사이즈는 DxDh로 표현이 된다. 그런데 만약 크기가 Dx3Dh인 행렬 U와 Linear연산을 하면 q,k,v를 한 번에 구할 수 있게 된다. 계산을 해서 Nx3Dh가 나오면 NxDh 3개로 나누면 될 일이다.

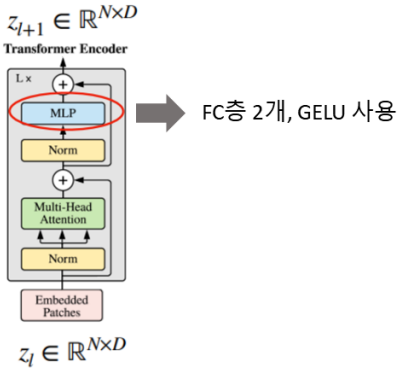
이후 q와 k의 유사도를 구하고 sqrt(Dh)로 나눠서 vanishing 문제를 막아준다. 이렇게 Attention Map을 구하면 NxN 크기를 가지게 되는 데 여기에 value(NxDh)를 행렬 곱함으로써 value에서 어느 부분이 중요한지를 알아낸다.

이렇게 나온 Self Attention Map(SA, NxDh)을 k개 구한 뒤 concat한 다음 Linear연산을 하면서 MSA를 마치게 된다. 마지막에 Linear 연산을 할 때 행렬 U를 사용하는 데, 이 때 U는 인코더의 입력과 출력을 맞추기 위해서 (kDh)xD가 사용이 된다.

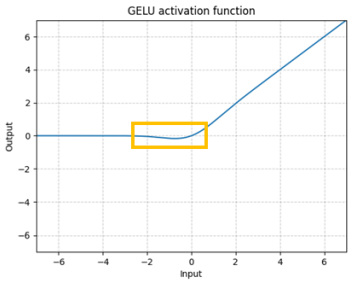
참고로 k는 헤드 수이고 Dh는 D/k로 결정된다고 한다. 예를 들어 D가 64이고 k가 8, n이 48이면 Dh는 8이기 때문에, U(64x24)와 Linear연산한 뒤 쪼개진 q,k,v는 각각 48x8이 되고 q,k의 내적으로 구한 Attention Map은 48x48, value를 곱한 Self Attention Map은 48x8, concat한 SA들은 48x64, U(64x64)와 Linear연산한 최종 output은 48x64가 된다.

Architecture

- Vision Transformer (ViT)
  - Multi Layer Perceptron (MLP)



FC층 2개, GELU 사용



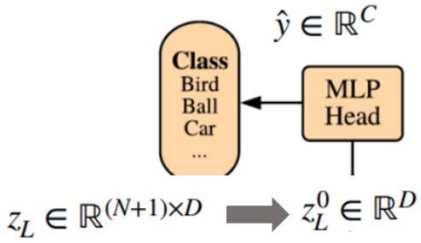
- **GELU:** Gaussian Error Linear Unit

DELAB 14

MLP는 FC층 2개와 GELU라는 Activation Function이 사용된다. 여기서 GELU는 Gaussian Error Linear Unit이라는 함수이고 x*CDF(누적 분포 함수) 라는 식인 데 위 그림과 같은 형태를 지니고 있 다. RELU의 음수 부분 일부가 0이 아닌 형태이다. RELU는 입력이 음수가 되는 순간 기울기가 0이 므로 그 노드에 연결된 파라미터들이 업데이트 되지가 않는데 이러한 현상을 극복하기 위해 GELU가 사용이 된다.

Architecture

- Vision Transformer (ViT)
  - **Step 5.** MLP에 Image Representation인 class token을 넣어 이미지의 class를 분류함



- 인코더 출력 Z_L: (N+1) x D
- 클래스 토큰 Z_L^0: 1xD
- 분류 예측치 $y^$: 1xC

DELAB 15

마지막 스텝은 MLP에 학습된 class token을 넣어서 이미지의 class를 분류한다.

다시 말하지만 클래스 토큰은 이미지에 대한 1차원 representation vector로써 역할을 한다. 이 토큰을 Linear연산을 사용하는 MLP Head에 넣어서 데이터의 클래스 수에 맞는 노드를 출력하 는 것이다.

이 때 CrossEntopy Loss가 사용되고 LayerNorm이 마찬가지로 적용된다.

여기까지가 ViT의 구조에 대한 설명이다.

Training Method

- Training
  - Large 데이터셋으로 사전 학습 -> 더 작은 데이터 셋에 대해 Transfer Learning을 적용
  - ImageNet-21k or JFT (pre-trained) -> CIFAR10, ImageNet-Real (fine-tuning)

- ImageNet, 1K classes, 1.3M images
- ImageNet-21k, 21K classes, 14M images
- JFT, 18K classes, 303M images

Pre-train	
optimizer	Adam
Scheduling	Linear Learning rate decay
Weight decay	0.1
Batch Size	4096
Etc	Label Smoothing, Early Stopping

Fine-tuning	
optimizer	SGD Momentum
Scheduling	Cosine Learning rate decay
Batch Size	512
Image Resize	384 (base)

DELAB 16

이제는 ViT의 학습 방법 및 실험 결과에 대해 소개하겠다.

ViT는 large 데이터 셋으로 사전 학습 후 더 작은 데이터 셋에 대해 Transfer Learning하는 방식을 사용한다. 이 때 학습을 위해 Large 데이터 셋인 ImageNet, ImageNet21k, JFT를 사용했다고 한다.

이 때 ImageNet21k는 클래스 수가 21000개 있고 1k보다 약 10배 정도 데이터가 많다. JFT같은 경우 구글에서 비공개하는 데이터 셋으로 무려 3억 3천만 개의 데이터가 수집되었다.

이러한 데이터 셋으로 사전 학습한 뒤에는 CIFAR10이나 ImageNet-Real 같은 소규모 데이터셋으로 파인튜닝하였다.

사전 학습할 때 Optimizer는 Adam, 스케줄링은 Linear Learning rate decay(epoch마다 learning rate를 다르게 사용), weight decay는 0,1(L2 Regularization의 패널티 값), 배치 사이즈는 4096, Label smoothing(실제 값인 원 핫 벡터를 조절함, 0이면 살짝 올리고 1을 살짝 늘려서 정규화에 도움을 줌), early stopping을 사용하였다. (validation accuracy를 계산하고 더 높아야 모델 저장)

파인 튜닝할 때 Optimizer는 SGD 모멘텀, 스케줄링은 cosine learining rate decay, 배치 사이즈는 512, resize는 384로 했다고 한다. (스케일링하면 resolution도 크게 설정함)

Training Method

- Training
  - Model Scailing을 적용해 총 세 가지 모델을 제작
  - ViT-H/16의 resolution: 512, ViT-H/14의 resolution: 518

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Table 1: Details of Vision Transformer model variants.

그리고 논문에서는 모델 스케일링을 해서 총 세 가지 모델이 있는데 Base는 Encoder의 Layer 수가 12, Large는 24, Huge는 32라고 생각하면 된다. 그리고 ViT-Huge같은 경우 resolution을 키웠을 때 더 성능이 잘 나왔다고 한다.

Experiments

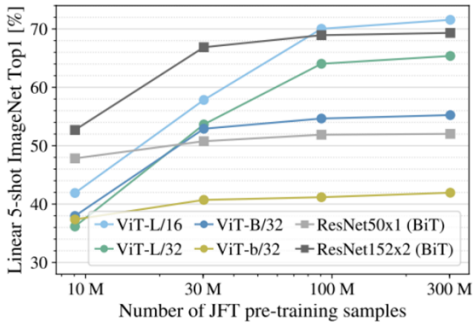
- Results
  - JFT Dataset으로 pre-train한 결과 CNN 모델보다 성능이 좋음 (large scale training trumps inductive bias)
  - Training Time이 CNN 계열보다 훨씬 적음

	Ours-JFT (ViT-H/14)	Ours-JFT (ViT-L/16)	Ours-I21k (ViT-L/16)	BiT-L (ResNet152x4)	Noisy Student (EfficientNet-L2)
ImageNet	**88.55** ± 0.04	87.76 ± 0.03	85.30 ± 0.02	87.54 ± 0.02	88.4/88.5*
ImageNet ReaL	**90.72** ± 0.05	90.54 ± 0.03	88.62 ± 0.05	90.54	90.55
CIFAR-10	**99.50** ± 0.06	99.42 ± 0.03	99.15 ± 0.03	99.37 ± 0.06	–
CIFAR-100	**94.55** ± 0.04	93.90 ± 0.05	93.25 ± 0.05	93.51 ± 0.08	–
Oxford-IIIT Pets	**97.56** ± 0.03	97.32 ± 0.11	94.67 ± 0.15	96.62 ± 0.23	–
Oxford Flowers-102	99.68 ± 0.02	**99.74** ± 0.00	99.61 ± 0.02	99.63 ± 0.03	–
VTAB (19 tasks)	**77.63** ± 0.23	76.28 ± 0.46	72.72 ± 0.21	76.29 ± 1.70	–
TPUv3-core-days	2.5k	0.68k	0.23k	9.9k	12.3k

실험 결과에서 모델명 뒤에 적힌 숫자(ex.14)는 패치의 크기이다. JFT Dataset을 사용한 게 가장 성능이 좋았고 21k도 준수했다고 한다. 여기서 ViT가 CNN보다 성능이 좋다는 것을 볼 수 있다. 또한 Training한 일수도 CNN 계열보다 훨씬 적다는 것을 알 수 있다.

Experiments

- Analysis
  - CNN과는 달리 ViT는 데이터 셋의 크기에 영향을 많이 받는다.



다음은 데이터 셋의 크기와 성능 간의 관계이다. CNN과 달리 ViT는 데이터 셋의 크기에 영향을 많이 받는 것을 볼 수 있다. 데이터 셋이 적을 때와 많을 때의 성능 차이가 확실하다.

Experiments

- Analysis
  - 학습된 모델을 분석한 결과 (Embedding Filter E, Position Embedding Filter E_{pos}, depth-attention distance)

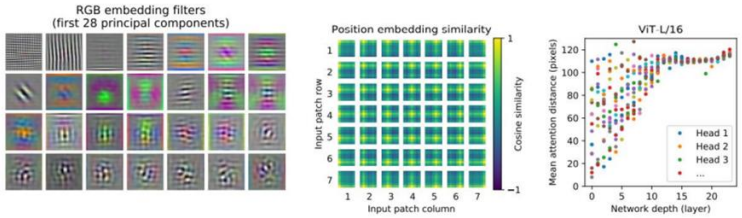


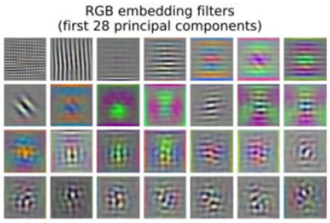
Figure 7: **Left:** Filters of the initial linear embedding of RGB values of ViT-L/32. **Center:** Similarity of position embeddings of ViT-L/32. Tiles show the cosine similarity between the position embedding of the patch with the indicated row and column and the position embeddings of all other patches. **Right:** Size of attended area by head and network depth. Each dot shows the mean attention distance across images for one of 16 heads at one layer. See Appendix D.7 for details.

다음은 학습된 모델을 분석한 사진이다. 첫 번째는 RGB 임베딩 필터들이고 두 번째는 Positional Embedding에 사용된 포지션 임베딩 필터(E_{pos})의 similarity이고 세 번째는 Network depth와 Attention distance와의 관계를 시각화한 것이다

이 결과들이 다 유의미하므로 각각에 대해서 살펴보겠다.

Experiments

- Analysis
  - Embedding Filter E



- 패치 시퀀스 모음 -- Embedding --> 패치 임베딩 모음

$$E \in \mathbb{R}^{(P^2C)\times D}$$

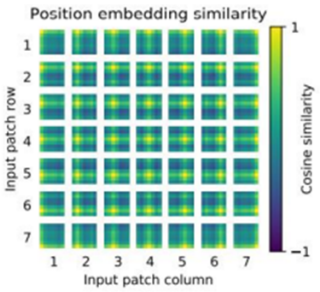
컬러 영상(CxPxP) D개 중 28개를 시각화

- ViT의 첫 번째 Layer를 시각화 -> CNN의 low level과 비슷한 그림이 나타남 -> 학습이 잘 됨

첫 번째는 RGB 임베딩 필터들이다. 임베딩 필터(E)의 크기가 P2CxD인데 여기서 P2C를 컬러 영상(CxPxP)으로 취급하고 28개만 가져온 것이다. 임베딩 필터는 ViT의 첫 번째 Layer라고 말할 수 있는데 CNN의 low level에서도 이런 그림이 나타난다고 한다.

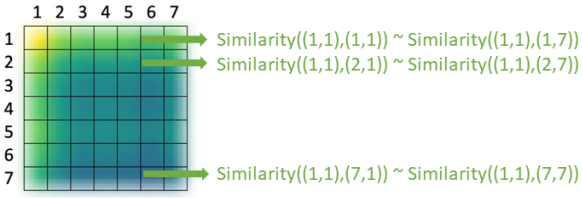
Experiments

- Analysis
  - Position Embedding Filter E_{pos} Similarity



- 패치 임베딩 모음 -- Positional Embedding --> 위치 정보가 포함된 패치 임베딩 모음

$$E_{pos} \in \mathbb{R}^{(N+1)\times D}$$ (N:49)



- 자신과 가까운 패치는 유사도가 높고 먼 패치는 유사도가 낮음 -> E_{pos}가 잘 학습됨

두 번째는 Positional Embedding에 사용된 포지션 인코딩 필터(Epos)의 similarity를 나타낸 것이다. 이 행렬은 (N+1)xD 크기인데 첫 번째 행은 Class Token이므로 제외하고 나머지 행들에 대한 similarity를 구한 것이다.

이 때 Epos의 각 행들은 패치에 대한 위치를 각각 나타내므로 패치들간의 similarity를 구했다고

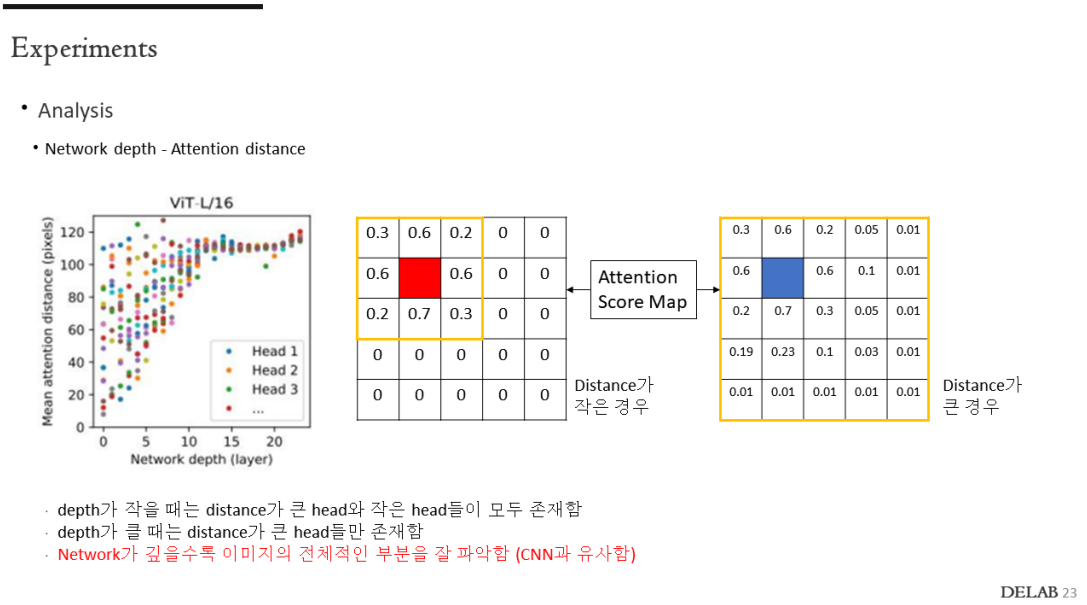
볼 수 있다.

결국 위 그림은 각 패치 별로 모든 패치 간의 similarity를 시각화한 것이다. 여기서 패치의 개수는 49개가 되는데 먼저 좌측 상단 맨 처음 그림부터 살펴보자.

첫 번째 패치는 1행 1열에 위치한 패치, 마지막 패치(49번째 패치)는 7행 7열에 위치한 패치가 되므로 1행 1열 패치를 기준으로 모든 행과 열에 위치한 패치들과 similarity를 구하면 좌측 상단으로 갈수록 유사도가 높겠고 우측 하단으로 갈수록 유사도가 낮을 것이다. 물론 자기 자신과 가장 유사도가 높으므로 1행 1열 위치가 노란 색이 된다.

반대로 마지막 우측 하단에 위치한 그림은 첫 번째 그림과 정 반대 형태가 될 것이다. 우측 하단으로 갈수록 유사도가 높겠고 좌측 상단으로 갈수록 유사도가 낮을 것이다.

이렇게 패치별로 관계들이 잘 시각화된 것을 보면 위치 정보를 담고 있는 Epos가 잘 학습됐다고 볼 수 있다. (자신과 가까운 패치일수록 유사도가 높고 멀수록 유사도가 낮으니 자신의 위치를 정확하게 안 다는 것)



세 번째는 Network depth와 Attention distance와의 관계를 시각화한 것이다.

여기서 Newtork depth는 Encoder Layer가 얼마나 연속해서 사용되는가이고, Attention distance는 보통 softmax(qkT)된 Attention Map에서 0이 많으면 작다고 표현하고 0이 적으면 크다고 표현한다.

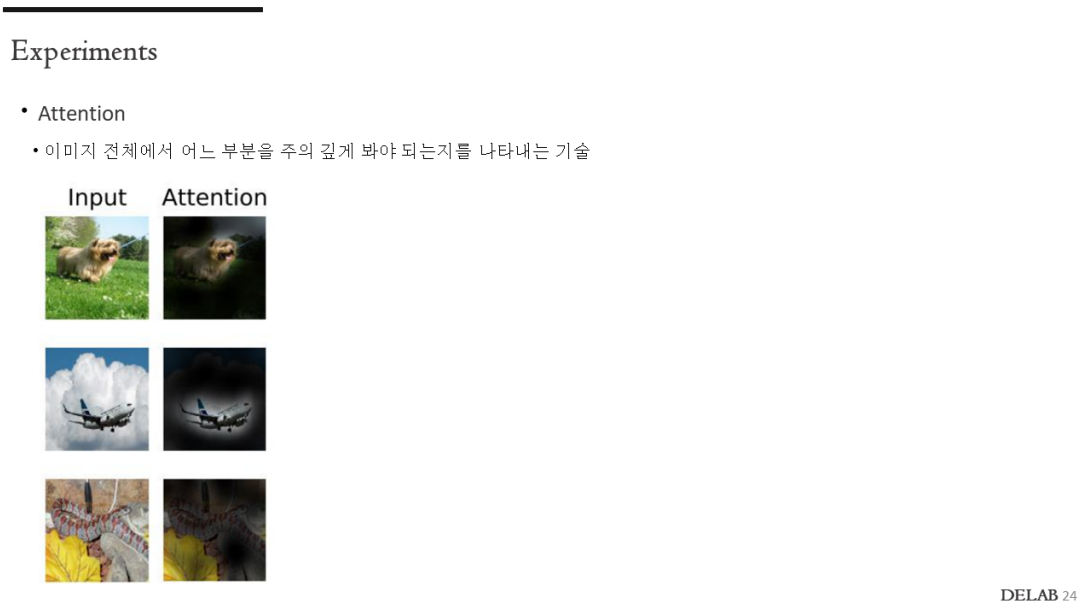
예를 들어 query가 한 픽셀로 주어졌다고 하면 해당 픽셀과 모든 픽셀 간의 유사도를 계산하게 된다. 이 때 주어진 한 픽셀 근처 값이 (0.3, 0.6, 0.2)따위이고 나머지 값들은 모두 0이라고 가정하면 거리는 이 노란 구간 안에서만 계산하므로 distance 는 작은 것이다. 반대로 주어진 픽셀과 멀

더라도 유사도 값이 0.01 따위로 주어진다면 distance가 큰 것이다.

이 때 distance가 작다면 이미지의 국소적인 부분만을 파악했다고 볼 수 있고 distance가 크다면 이미지 의 전체적인 모습을 파악했다고 볼 수 있다. 즉 CNN의 Receptive Field와 비슷한 내용이다.

결론적으로 distance가 클수록 이미지의 어느 부분이 중요하고 어느 부분이 덜 중요한지를 정확히 표현할 수 있다는 것이니 Attention이 잘 학습됐다고 볼 수 있다.

결국 layer를 깊게 쌓을수록 전체적인 부분을 더 잘 파악할 수 있다는 것을 나타내는 그림으로써 초반 layer에서는 국소적인 부분을 보고 후반 layer에서는 전체적인 부분을 보는 CNN과 비슷한 맥락이라고 이해할 수 있다.



마지막으로 Attention을 시각화했을 때 위와 같이 이미지에서 중요한 부분을 잘 나타낼 수 있었다고 한다.