Visual Transformer in CV

Artificial Intelligence

Creating the Future

Dong-A University

Division of Computer Engineering &
Artificial Intelligence

References

Main

- https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
- https://jalammar.github.io/illustrated-transformer/

blog Sub

- https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632

Newly tutorials

- https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html

Main

- https://github.com/lucidrains/vit-pytorch
- https://github.com/FrancescoSaverioZuppichini/ViT
- https://pypi.org/project/vision-transformer-pytorch/

1 ViT (Vision Transformer)

[Google Research, Brain Team]
A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

We show that

- **The reliance on CNNs is not necessary**
- **Pure transformer applied directly to sequences of image patches** can perform **very well on image classification tasks**.

- When pre-trained on large amounts of data and transferred to multiple mid-sized or small image recognition benchmarks (ImageNet, CIFAR-10, VTAB, etc), **ViT** attains **excellent results compared to SOTA conventional networks** while **requiring substantially fewer computational resources to train**.

google-research/vision_transformer ★ 2,952
official
Quickstart in Colab

huggingface/transformers ★ 47,764 PyTorch

rwightman/pytorch-image-models ★ 11,204 PyTorch
Quickstart in Colab

lucidrains/vit-pytorch ★ 4,799 PyTorch

facebookresearch/vissl ★ 1,742 PyTorch
Quickstart in Colab

See all 50 implementations

	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

Table 2: Comparison with state of the art on popular image classification benchmarks. We report mean and standard deviation of the accuracies, averaged over three fine-tuning runs. Vision Transformer models pre-trained on the JFT-300M dataset outperform ResNet-based baselines on all datasets, while taking substantially less computational resources to pre-train. ViT pre-trained on the smaller public ImageNet-21k dataset performs well too. *Slightly improved 88.5% result reported in Touvron et al. (2020).

1 ViT (Vision Transformer)

[Google Research, Brain Team]
A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

- Fine-tuning code and pre-trained models are available on https://github.com/google-research/vision_transformer

Vision Transformer and MLP-Mixer Architectures

Update (20.6.2021): Added the "How to train your ViT? ..." paper, and a new Colab to explore the >50k pre-trained and fine-tuned checkpoints mentioned in the paper.

Update (18.6.2021): This repository was rewritten to use Flax Linen API and `ml_collections.ConfigDict` for configuration.

In this repository we release models from the papers

- An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale
- MLP-Mixer: An all-MLP Architecture for Vision
- How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers

The models were pre-trained on the ImageNet and ImageNet-21k datasets. We provide the code for fine-tuning the released models in JAX/Flax.

- Fine-tuning code and pre-trained models are available on https://github.com/jeonsworld/ViT-pytorch

1. Download Pre-trained model (Google's Official Checkpoint)

- Available models: ViT-B_16(**85.8M**), R50+ViT-B_16(**97.96M**), ViT-B_32(**87.5M**), ViT-L_16(**303.4M**), ViT-L_32(**305.5M**), ViT-H_14(**630.8M**)
  - imagenet21k pre-train models
    - ViT-B_16, ViT-B_32, ViT-L_16, ViT-L_32, ViT-H_14
  - imagenet21k pre-train + imagenet2012 fine-tuned models
    - ViT-B_16-224, ViT-B_16, ViT-B_32, ViT-L_16-224, ViT-L_16, ViT-L_32
  - Hybrid Model(Resnet50 + Transformer)
    - R50-ViT-B_16

```
# imagenet21k pre-train
wget https://storage.googleapis.com/vit_models/imagenet21k/{MODEL_NAME}.npz

# imagenet21k pre-train + imagenet2012 fine-tuning
wget https://storage.googleapis.com/vit_models/imagenet21k+imagenet2012/{MODEL_NAME}.npz
```

JAX is Autograd and XLA(Accelerated Linear Algebra), brought together for high-performance numerical computing and machine learning research. It provides composable transformations of Python+NumPy programs: *differentiate*, *vectorize*, *parallelize*, *Just-In-Time compile to GPU/TPU*, and more.

Flax is a neural network library and ecosystem for JAX that is designed for flexibility. Flax is in use by a growing community of researchers and engineers at Google who happily use Flax for their daily research.

1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

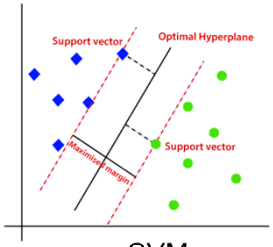
[출처] Open DMQA Seminar, Transformer in Computer Vision, Korea Univ.

- Apply a standard Transformer directly to images, with the fewest possible modifications.

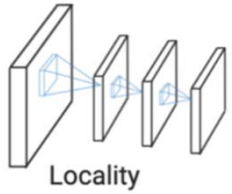
1) **Split an image into patches** and provide **the sequence of linear embeddings of these patches as an input** to a Transformer
2) **Treat image patches the same way as tokens (words)** in an NLP tasks. Train the model on image classification.

- When **trained on mid-sized datasets** (ImageNet) without strong regularization, ViT yield modest accuracies of a few percentage points below ResNets of comparable size.
- **Transformers *lack some of the inductive biases*** inherent to CNNs (such as *translation equivalence* and *locality*), and therefore do **not generalize well when trained on insufficient amounts of data.**
- **Large scale training trumps inductive bias. ViT attains excellent results when pre-trained at sufficient scale and transferred to tasks with fewer datapoints.** (Pretrained on **ImageNet-21K** or **JFT-300M** datasets)

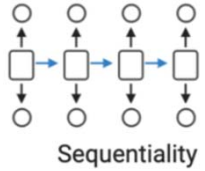
- ***Inductive Bias*** (or learning bias) : 학습 모델이 지금까지 만나보지 못했던 상황에서 정확한 예측을 하기 위해 사용하는 추가적인 가정을 의미
- SVM – Margin 최대화, CNN – 지역적 정보, RNN – 순차적 정보



SVM

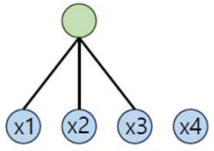


CNN

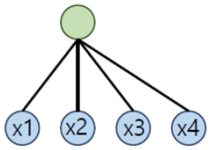


RNN

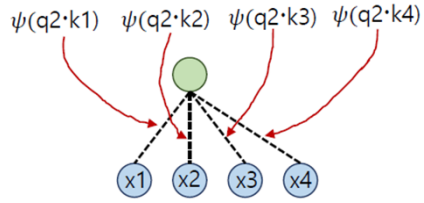
- Transformer : Inductive bias ↓, 모델의 자유도 ↑
  - ✓ 1차원 벡터로 만든 후 self attention (Global, 2차원의 지역적 정보 x)
  - ✓ Weight이 Input에 따라 유동적으로 변함
- CNN : 2차원의 지역적 특성 유지, 학습 후 Weight 고정



Convolution layer
- Locally connected
- Same weights for all inputs



Dense layer
- Fully connected
- Same weights for all inputs



Attention layer
- Fully connected
- Different weights for all inputs

1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

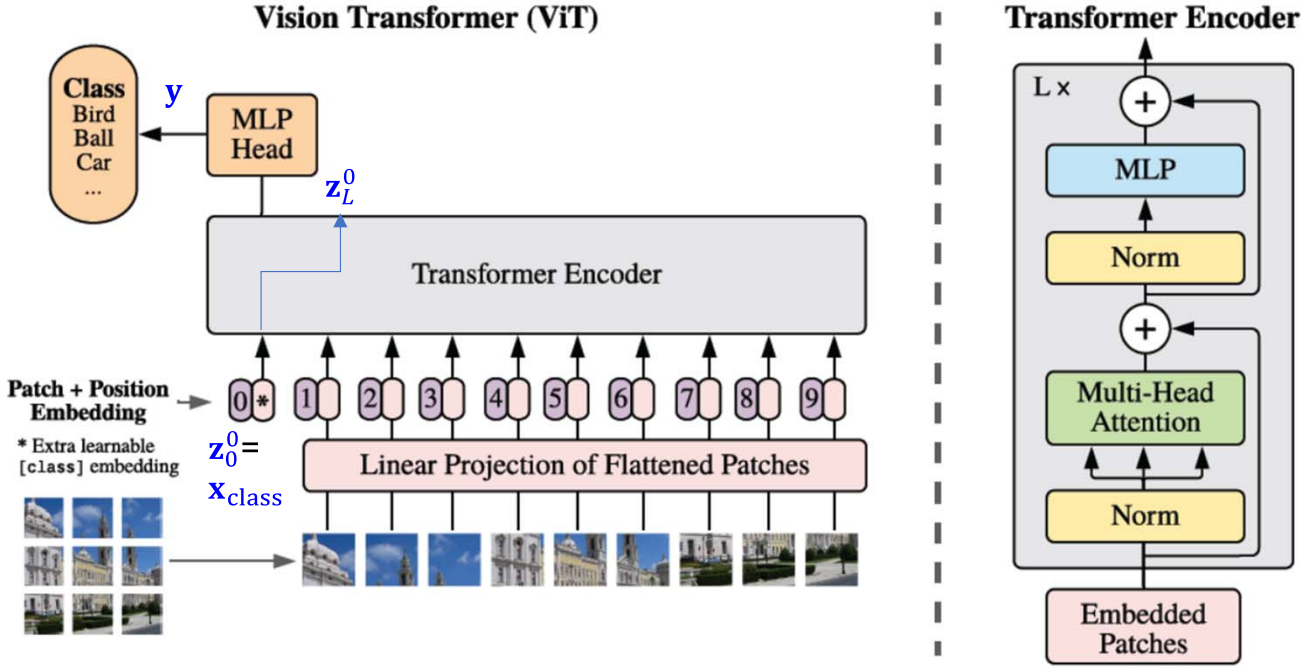


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable “*classification token*” to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

- Reshape an image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$

  (P, P) : Resolution of each patch

  $N = HW/P^2$: Number of patches (serve as the effective input sequence length for Transformer)

Embed size

- Transformer uses constant latent vector size D through all of its layers. → Flatten the patches and map to D dimensions with a trainable linear projection (*Patch embeddings*)

$$\mathbf{z}_0 = \left[\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}\right] + \mathbf{E}_{\text{pos}} \quad (1)$$

$$\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$$

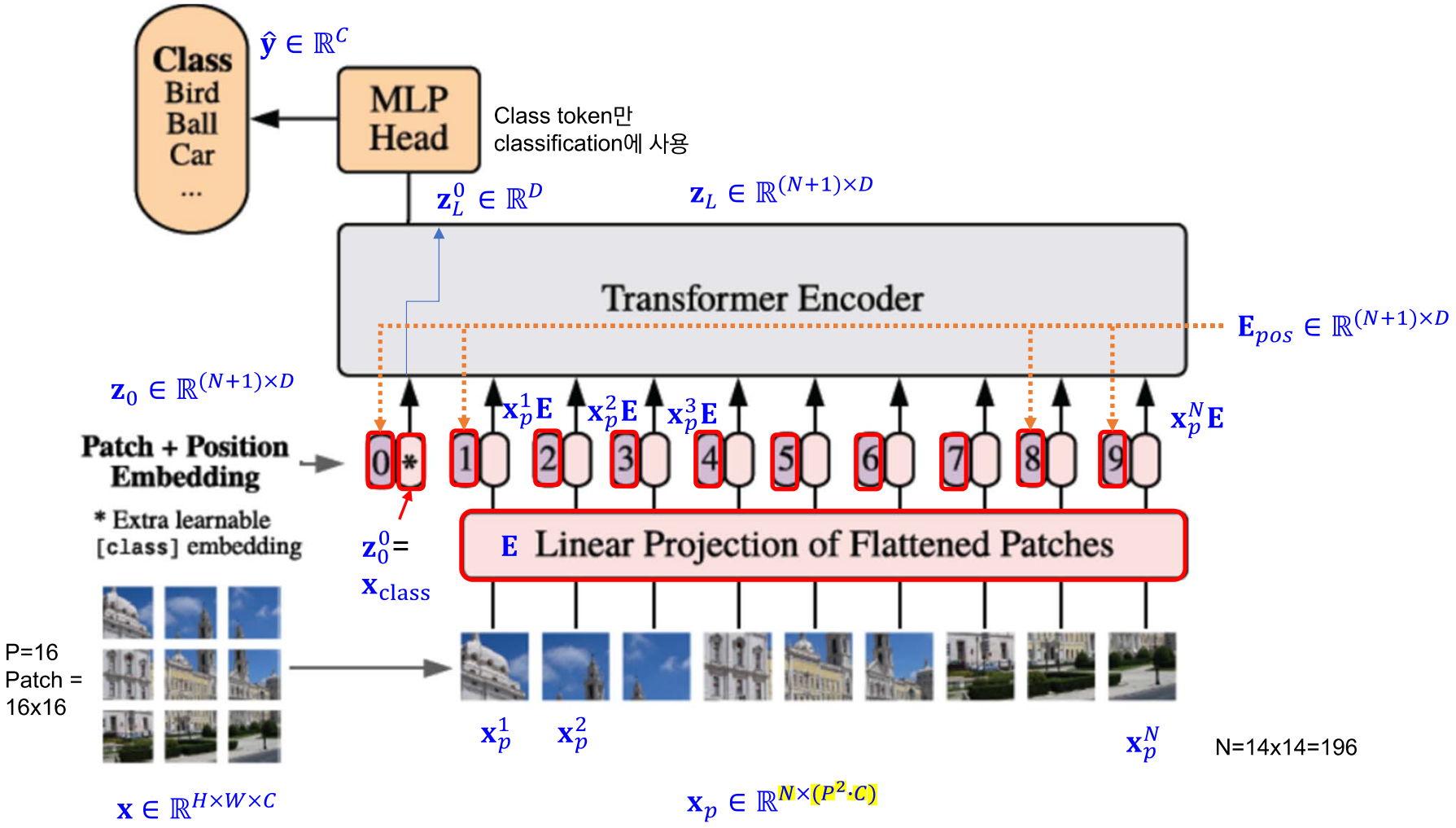
where $\mathbf{E}$ is the patch embedding projection

- Prepend a *learnable embedding* (Similar *class token*) to the sequence of embedded patches ($\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$), whose state at the output of Transformer Encoder ($\mathbf{z}_L^0$), serve as the image representation $\mathbf{y}$

- Both during pre-training and fine-tuning, a classification head is attached to $\mathbf{z}_L^0$. The classification head is implemented by a MLP with one hidden layer at pre-training time and by a single linear layer at fine-tuning time.

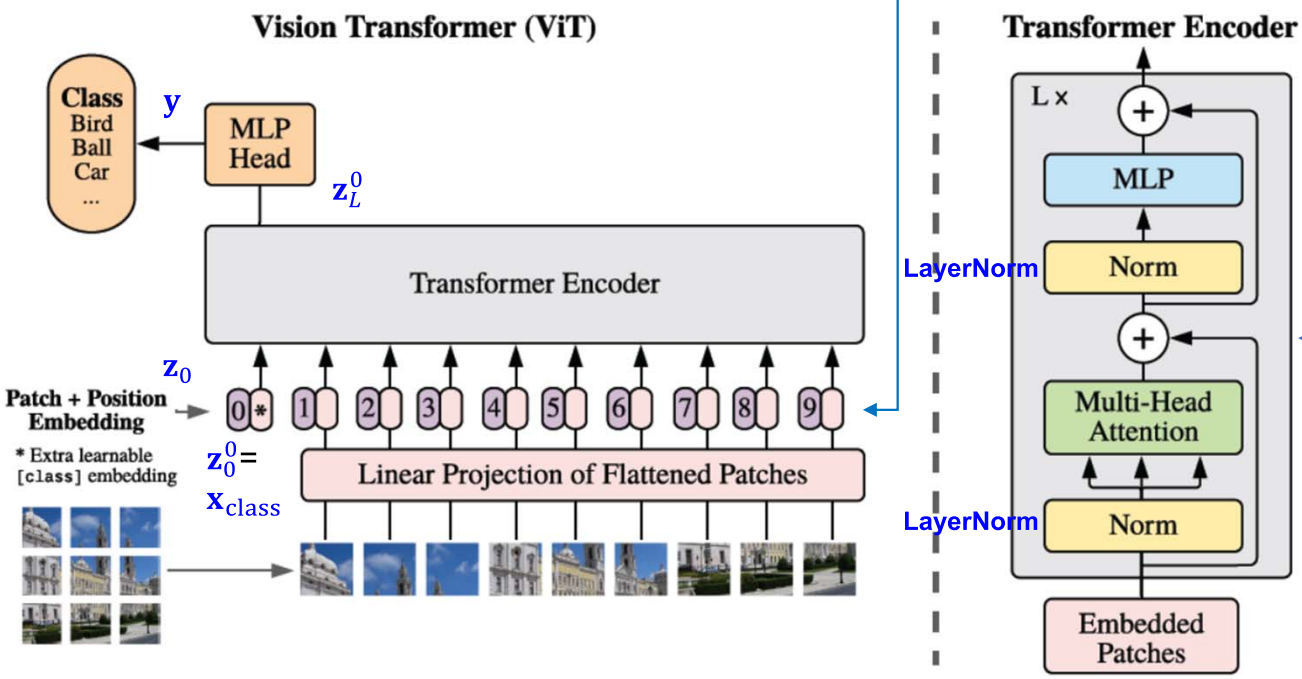
1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021



1 ViT (Vision Transformer)

A. Dosovitskiy, et al. "An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale," ICLR2021



$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{\text{pos}} \quad (1)$$

$$\mathbf{z}'_l = MSA(LN(\mathbf{z}_{l-1})) + \mathbf{z}_{l-1} \quad l = 1, \ldots, L \quad (2)$$

$$\mathbf{z}_l = MLP(LN(\mathbf{z}'_l)) + \mathbf{z}'_l \quad (3)$$

$$\mathbf{y} = LN(\mathbf{z}_L^0) \quad (4)$$

- MLP contains two layers with a *GELU non-linearity*.
- Both during pre-training and fine-tuning, a classification head is attached to $\mathbf{z}_L^0$.
- The classification head is implemented by a MLP with one hidden layer at pre-training time and by a single linear layer at fine-tuning time.

- Add Position Embedding to the patch embeddings; use standard learnable 1D position embeddings. → The resulting sequence of embedding vectors serves as input to Transformer Encoder.

- LayerNorm (LN) is applied before every block, and residual connections after every block

➢ GELU (Gaussian Error Linear Units)

$$\text{GELU}(x) = xP(X \le x) = x\Phi(x)$$
$$\approx 0.5x\left(1 + \tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right)$$

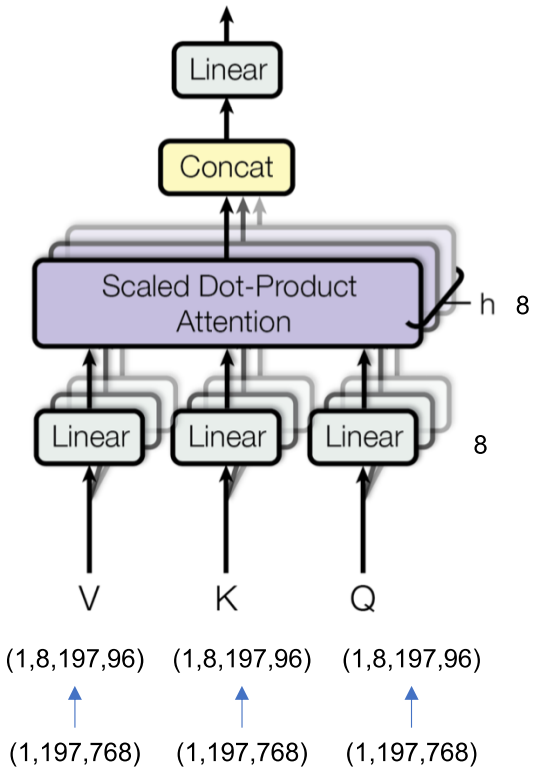
- LayerNorm : 각 Feature에 normalization (D dimension)

$$z_i = [z_i^1, z_i^2, z_i^3, \cdots, z_i^N, z_i^{N+1}] \qquad \text{LN}(z_i^j) = \gamma\frac{z_i^j - \mu_i}{\sigma_i} + \beta \quad = \gamma\frac{z_i^j - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta$$

1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

MHA (Multi Head Attention)



MSA (Multi Head Self-Attention)

Reshape an image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$

(P, P) : Resolution of each patch, $N = HW/P^2$: Number of patches

Appendix

Standard $\mathbf{qkv}$ self-attention (SA, Vaswani et al. (2017)) is a popular building block for neural architectures. For each element in an input sequence $\mathbf{z} \in \mathbb{R}^{N \times D}$, we compute a weighted sum over all values $\mathbf{v}$ in the sequence. The attention weights A_{ij} are based on the pairwise similarity between two elements of the sequence and their respective query $\mathbf{q}^i$ and key $\mathbf{k}^j$ representations.

$$[\mathbf{q}, \mathbf{k}, \mathbf{v}] = \mathbf{z}\mathbf{U}_{qkv} \qquad \mathbf{U}_{qkv} \in \mathbb{R}^{D \times 3D_h}, \tag{5}$$

$$A = \text{softmax}\left(\mathbf{q}\mathbf{k}^\top / \sqrt{D_h}\right) \qquad A \in \mathbb{R}^{N \times N}, \tag{6}$$

$$\text{SA}(\mathbf{z}) = A\mathbf{v}\,. \tag{7}$$

Multihead self-attention (MSA) is an extension of SA in which we run k self-attention operations, called “heads”, in parallel, and project their concatenated outputs. To keep compute and number of parameters constant when changing k, D_h (Eq. 5) is typically set to D/k.

$$\text{MSA}(\mathbf{z}) = [\text{SA}_1(z); \text{SA}_2(z); \cdots; \text{SA}_k(z)]\,\mathbf{U}_{msa} \qquad \mathbf{U}_{msa} \in \mathbb{R}^{k \cdot D_h \times D} \tag{8}$$

MSA 구할 때, 각 head의 q,k,v 에 대한 연산을 따로 하지 않고 한번에 처리

$$\text{head 1} : q_1 = z \cdot w_q^1, k_1 = z \cdot w_k^1, v_1 = z \cdot w_v^1$$

$$\text{head 2} : q_2 = z \cdot w_q^2, k_2 = z \cdot w_k^2, v_2 = z \cdot w_v^2$$

head 에서 weight만 달라지게 되므로

$$\text{Single Head} : q, k, v \in \mathbb{R}^{N \times D_h} \rightarrow \text{Multi Head} : q, k, v \in \mathbb{R}^{N \times k \times D_h}$$

1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

Model Variants

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.

- ViT-L/16 means the “Large” variant with 16x16 input patch size.
- Note that the Transformer’s sequence length is inversely proportional to the square of the patch size, thus models with smaller patch size are computationally more expensive.

Results

		ViT-B/16	ViT-B/32	ViT-L/16	ViT-L/32	ViT-H/14
ImageNet	CIFAR-10	98.13	97.77	97.86	97.94	-
	CIFAR-100	87.13	86.31	86.35	87.07	-
	ImageNet	77.91	73.38	76.53	71.16	-
	ImageNet ReaL	83.57	79.56	82.19	77.83	-
	Oxford Flowers-102	89.49	85.43	89.66	86.36	-
	Oxford-IIIT-Pets	93.81	92.04	93.64	91.35	-
ImageNet-21k	CIFAR-10	98.95	98.79	99.16	99.13	99.27
	CIFAR-100	91.67	91.97	93.44	93.04	93.82
	ImageNet	83.97	81.28	85.15	80.99	85.13
	ImageNet ReaL	88.35	86.63	88.40	85.65	88.70
	Oxford Flowers-102	99.38	99.11	99.61	99.19	99.51
	Oxford-IIIT-Pets	94.43	93.02	94.73	93.09	94.82
JFT-300M	CIFAR-10	99.00	98.61	99.38	99.19	99.50
	CIFAR-100	91.87	90.49	94.04	92.52	94.55
	ImageNet	84.15	80.73	87.12	84.37	88.04
	ImageNet ReaL	88.85	86.27	89.99	88.28	90.33
	Oxford Flowers-102	99.56	99.27	99.56	99.45	99.68
	Oxford-IIIT-Pets	95.80	93.40	97.11	95.83	97.56

Table 5: Top1 accuracy (in %) of Vision Transformer on various datasets when pre-trained on ImageNet, ImageNet-21k or JFT300M. These values correspond to Figure 3 in the main text. Models are fine-tuned at 384 resolution. Note that the ImageNet results are computed without additional techniques (Polyak averaging and 512 resolution images) used to achieve results in Table 2.

1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

Results

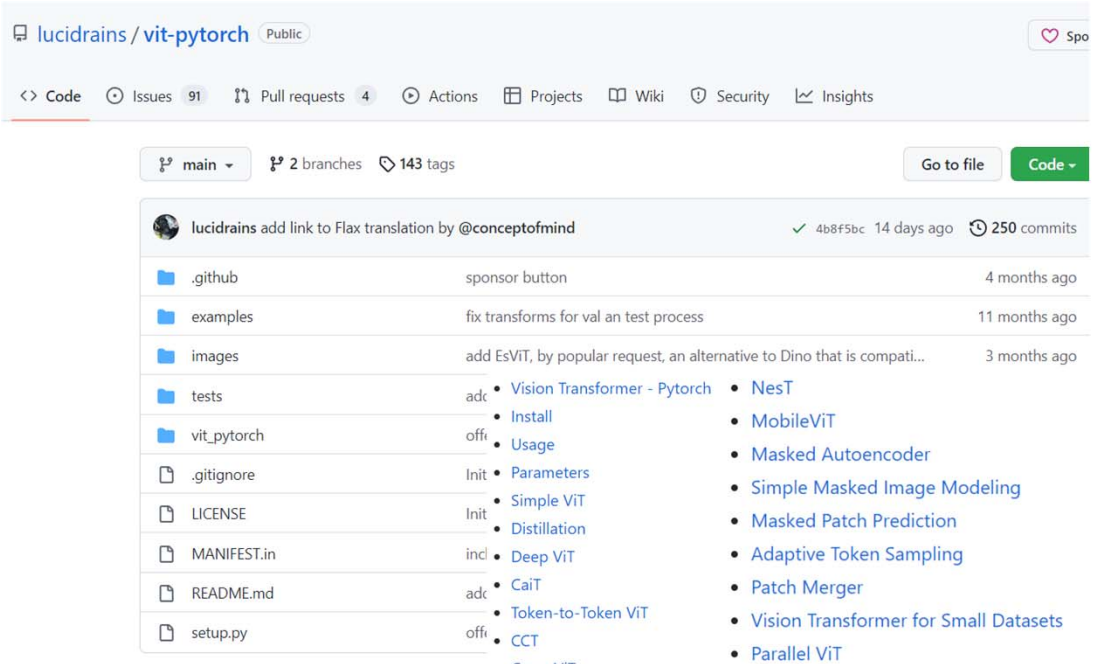
name	Epochs	ImageNet	ImageNet ReaL	CIFAR-10	CIFAR-100	Pets	Flowers	exaFLOPs
ViT-B/32	7	80.73	86.27	98.61	90.49	93.40	99.27	55
ViT-B/16	7	84.15	88.85	99.00	91.87	95.80	99.56	224
ViT-L/32	7	84.37	88.28	99.19	92.52	95.83	99.45	196
ViT-L/16	7	86.30	89.43	99.38	93.46	96.81	99.66	783
ViT-L/16	14	87.12	89.99	99.38	94.04	97.11	99.56	1567
ViT-H/14	14	88.08	90.36	99.50	94.71	97.11	99.71	4262
ResNet50x1	7	77.54	84.56	97.67	86.07	91.11	94.26	50
ResNet50x2	7	82.12	87.94	98.29	89.20	93.43	97.02	199
ResNet101x1	7	80.67	87.07	98.48	89.17	94.08	95.95	96
ResNet152x1	7	81.88	87.96	98.82	90.22	94.17	96.94	141
ResNet152x2	7	84.97	89.69	99.06	92.05	95.37	98.62	563
ResNet152x2	14	85.56	89.89	99.24	91.92	95.75	98.75	1126
ResNet200x3	14	87.22	90.15	99.34	93.53	96.32	99.04	3306
R50x1+ViT-B/32	7	84.90	89.15	99.01	92.24	95.75	99.46	106
R50x1+ViT-B/16	7	85.58	89.65	99.14	92.63	96.65	99.40	274
R50x1+ViT-L/32	7	85.68	89.04	99.24	92.93	96.97	99.43	246
R50x1+ViT-L/16	7	86.60	89.72	99.18	93.64	97.03	99.40	859
R50x1+ViT-L/16	14	87.12	89.76	99.31	93.89	97.36	99.11	1668

Table 6: Detailed results of model scaling experiments. These correspond to Figure 5 in the main paper. We show transfer accuracy on several datasets, as well as the pre-training compute (in exaFLOPs).

1 ViT (Visual Transformer) in PyTorch

lucidrains / vit-pytorch

https://github.com/lucidrains/vit-pytorch



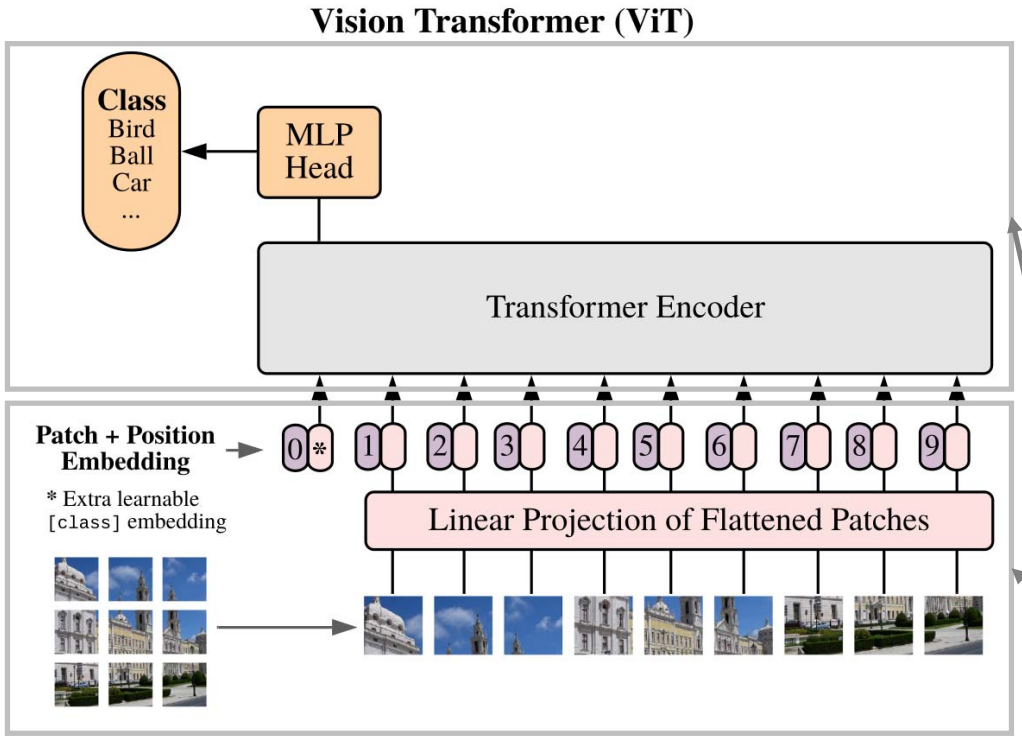
1 ViT (Visual Transformer) in PyTorch

[Google Research, Brain Team]

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

- ViT uses a normal transformer (the one proposed in Attention is All You Need) that works on images.



Organization Sections

- Data
- Patches Embeddings
  - CLS Token
  - Position Embedding
- Transformer
  - Attention
  - Residuals
  - MLP
  - TransformerEncoder
- Head
- ViT

4) The resulting tensor is passed first into a standard Transformer and then to a classification head.

1) Decompose the input image into 16x16 flatten patches (the image is not in scale).
2) Then embed them using a normal fully connected layer,
3) Add a special cls token in front of them and sum the positional encoding.

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

Prerequisite

- Have a look at the amazing The Illustrated Transformer website
- Watch Yannic Kilcher video about ViT
- Read Einops doc

```
pip install eniops
```

Packages

```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt

from torch import nn
from torch import Tensor
from PIL import Image
from torchvision.transforms import Compose, Resize, ToTensor
from einops import rearrange, reduce, repeat
from einops.layers.torch import Rearrange, Reduce
from torchsummary import summary
```

Sample Data

- ViT uses a normal transformer (the one proposed in Attention is All You Need) that works on images.

```
img = Image.open('./cat.jpg')
fig = plt.figure()
plt.imshow(img)

# resize to imagenet size
transform = Compose([Resize((224, 224)), ToTensor()])
x = transform(img)
x = x.unsqueeze(0) # add batch dim
```

```
x.shape : torch.Size([1, 3, 224, 224])
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

1. Patch Embeddings

1) Patches

- The first step is to break-down the image in multiple patches and flatten them.

b c (h s1) (w s2) = 1, 3, (3x16), (3x16) b c (h w) (s1 s2 c) = 1, (3x3), (16x16x3)

- Reshape

$$\mathbf{x} \in \mathbb{R}^{H \times W \times C} \longrightarrow \mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$$

2D image A sequence of flattened 2D patches

$P \times P$: resolution of each patch (16, 16)

$N = HW/P^2$ Number of patches

- This can be easily done using einops.

```
patch_size = 16 # 16 pixels
patches = rearrange(x, 'b c (h s1) (w s2) -> b (h w) (s1 s2 c)', s1=patch_size, s2=patch_size)
```

```
x.shape : torch.Size([1, 3, 224, 224])
```

```
patches.shape : torch.Size([1, 196, 768])
```

b c (h s1) (w s2) = 1, 3, (14x16), (14x16)
b (h w) (s1 s2 c) = 1, (14x14), (16x16x3)

An overview of the model is depicted in Figure 1. The standard Transformer receives as input a 1D sequence of token embeddings. To handle 2D images, we reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$, where (H, W) is the resolution of the original image, C is the number of channels, (P, P) is the resolution of each image patch, and $N = HW/P^2$ is the resulting number of patches, which also serves as the effective input sequence length for the Transformer. The Transformer uses constant latent vector size D through all of its layers, so we

einops

[주 출처] https://github.com/arogozhnikov/einops/
[참고] https://yhkim4504.tistory.com/5
[중요] https://rockt.github.io/2018/04/30/einsum

einops

[중요 참고] EINSUM IS ALL YOU NEED - EINSTEIN SUMMATION IN DEEP LEARNING
https://rockt.github.io/2018/04/30/einsum

Tutorials

- Part 1: einops fundamentals
- Part 2: einops for deep learning
- Part 3: real code fragments improved with einops (so far only for pytorch)

```
from einops import rearrange, reduce, repeat
# rearrange elements according to the pattern
output_tensor = rearrange(input_tensor, 't b c -> b c t')
# combine rearrangement and reduction
output_tensor = reduce(input_tensor, 'b c (h h2) (w w2) -> b h w c', 'mean', h2=2, w2=2)
# copy along a new axis
output_tensor = repeat(input_tensor, 'h w -> h w c', c=3)
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

1) Patches

- Create a PatchEmbedding class

Ver1

```
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16, emb_size: int = 768):
        self.patch_size = patch_size
        super().__init__()
        self.projection = nn.Sequential(
            # break-down the image in s1 x s2 patches and flat them
            Rearrange('b c (h s1) (w s2) -> b (h w) (s1 s2 c)', s1=patch_size,
s2=patch_size),
            nn.Linear(patch_size * patch_size * in_channels, emb_size)
        )

    def forward(self, x: Tensor) -> Tensor:
        x = self.projection(x)
        return x

PatchEmbedding()(x).shape
```

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,
padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')
```

Ver2

```
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16, emb_size: int = 768):
        self.patch_size = patch_size
        super().__init__()
        self.projection = nn.Sequential(
            # using a conv layer instead of a linear one -> performance gains
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size,
stride=patch_size),
            Rearrange('b e (h) (w) -> b (h w) e'),
        )

    def forward(self, x: Tensor) -> Tensor:
        x = self.projection(x)
        return x

PatchEmbedding()(x).shape
```

- [Note] After checking out the original implementation, I found out that the authors are using a Conv2d layer instead of a Linear one for performance gain. This is obtained by using a kernel_size and stride equal to the `patch_size`. Intuitively, the convolution operation is applied to each patch individually. So, we have to first apply the conv layer and then flat the resulting images.

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

1) Patches

- Create a PatchEmbedding class

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')
```

```
patches = rearrange(x, 'b c (h s1) (w s2) -> b (h w) (s1 s2 c)', s1=patch_size, s2=patch_size)
patches.shape
```

```
torch.Size([1, 196, 768])
```

```
linear = nn.Linear(patch_size * patch_size * 3, 768)
linear(patches).shape
```

```
torch.Size([1, 196, 768])
```

b c (h s1) (w s2) = [1, 3, (14x16), (14x16)] = [1, 3, 224, 224]

b c (h w) (s1 s2 c) = 1, (14x14), (16x16x3) = [1, 196, 768]

```
conv1 = nn.Conv2d(3, 768, 16, 16)
conv1(x).shape
```

```
torch.Size([1, 768, 14, 14])
```

```
rearrang = Rearrange('b e (h) (w) -> b (h w) e')
rearrang(conv1(x)).shape
```

```
torch.Size([1, 196, 768])
```

conv1 = [1, 768, 14, 14]

b e (h) (w) = [1, 768, 14, 14]

b (h w) e = [1, 14x14, 768] = [1, 196, 768]

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

2) CLS Token

- Next, add the cls token and the position embedding.
- The cls token is just a number placed in from of **each** sequence (of projected patches)

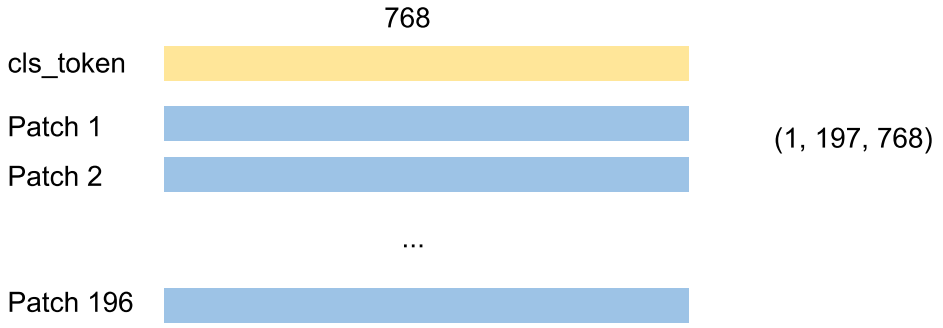
Ver3

```
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16, emb_size: int = 768):
        self.patch_size = patch_size
        super().__init__()
        self.proj = nn.Sequential(
            # using a conv layer instead of a linear one -> performance gains
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size,
stride=patch_size),
            Rearrange('b e (h) (w) -> b (h w) e'),
        )

        self.cls_token = nn.Parameter(torch.randn(1,1, emb_size))

    def forward(self, x: Tensor) -> Tensor:
        b, _, _, _ = x.shape
        x = self.proj(x)
        cls_tokens = repeat(self.cls_token, '() n e -> b n e', b=b)
        # prepend the cls token to the input
        x = torch.cat([cls_tokens, x], dim=1)
        return x

PatchEmbedding()(x).shape
```

```
torch.Size([1, 197, 768])
```

```
emb_size = 768
b=1
cls_token = nn.Parameter(torch.randn(1,1, emb_size))
cls_token.shape
cls_tokens = repeat(cls_token, '() n e -> b n e', b=b)
cls_tokens.shape
```

```
torch.Size([1, 1, 768])

torch.Size([1, 1, 768])
```



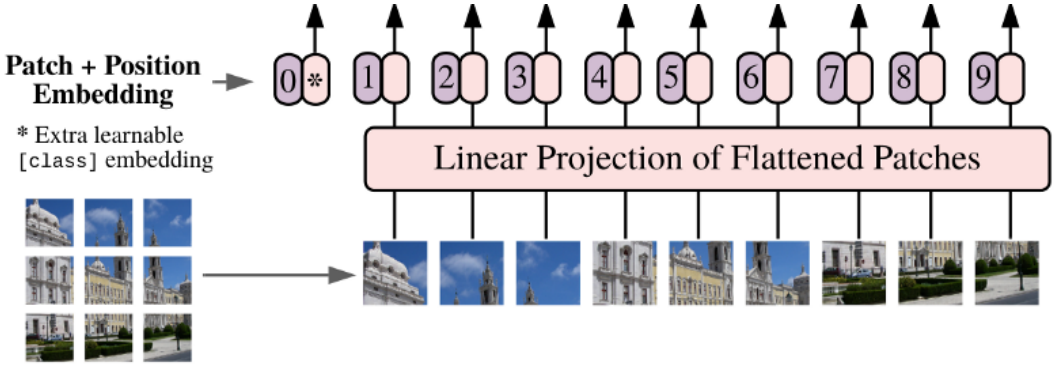
- cls_token is a torch Parameter randomly initialized, in the forward the method it is copied b (batch) times and prepended before the projected patches using torch.cat

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

3) Position Embedding

- So far, the model has no idea about the original position of the patches. We need to pass this spatial information. This can be done in different ways, in ViT we let the model learn it. The position embedding is just a tensor of shape (N_PATCHES + 1 (token), EMBED_SIZE) that is added to the projected patches.



- We added the position embedding in the .positions field and sum it to the patches in the .forward function

```
emb_size = 768
img_size = 224
patch_size = 16
b=1
```

```
positions = nn.Parameter(torch.randn((img_size // patch_size) **2 + 1, emb_size))
positions.shape
```

```
torch.Size([197, 768])
```

Ver4

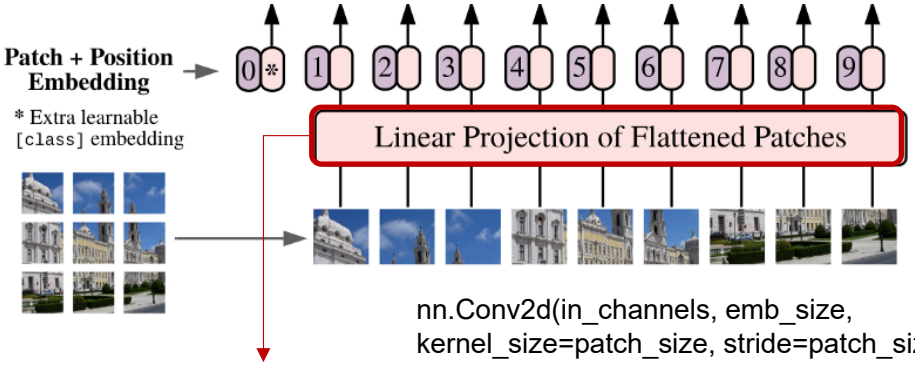
```
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16, emb_size: int = 768,
img_size: int = 224):
        self.patch_size = patch_size
        super().__init__()
        self.projection = nn.Sequential(
            # using a conv layer instead of a linear one -> performance gains
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size,
stride=patch_size),
            Rearrange('b e (h) (w) -> b (h w) e'),
        )
        self.cls_token = nn.Parameter(torch.randn(1,1, emb_size))
        self.positions = nn.Parameter(torch.randn((img_size // patch_size) **2 + 1,
emb_size))
                                                    (224//16)^2+1=14^2+1=197
            768

    def forward(self, x: Tensor) -> Tensor:
        b, _, _, _ = x.shape
        x = self.projection(x)
        cls_tokens = repeat(self.cls_token, '() n e -> b n e', b=b)
        # prepend the cls token to the input
        x = torch.cat([cls_tokens, x], dim=1)
        # add position embedding
        x += self.positions
        return x

PatchEmbedding()(x).shape
```

```
torch.Size([1, 197, 768])
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

Inspecting Vision Transformer



nn.Conv2d(in_channels, emb_size, kernel_size=patch_size, stride=patch_size),

self.positions = nn.Parameter(torch.randn((img_size // patch_size) **2 + 1, emb_size))

RGB embedding filters
(first 28 principal components)

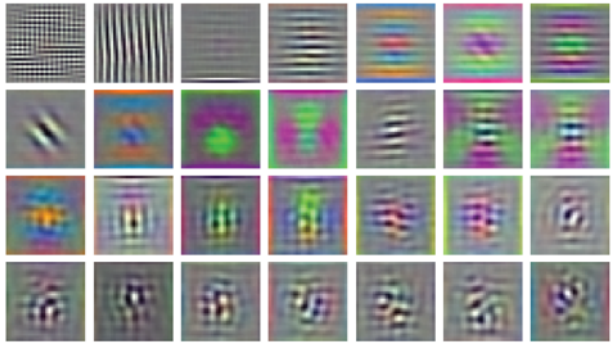


Fig. 7. Filters of the initial linear embedding of RGB values of ViT-L/32.

The first layer of ViT linearly projects the flattened patches into a lower-dimensional space.

Top principal components of the learned embedding filters. – Resemble plausible basis function for a low-dimensional representation of the fine structure within each patch.

$$\mathbf{z}_0 = \left[\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}\right] + \mathbf{E}_{\text{pos}}$$

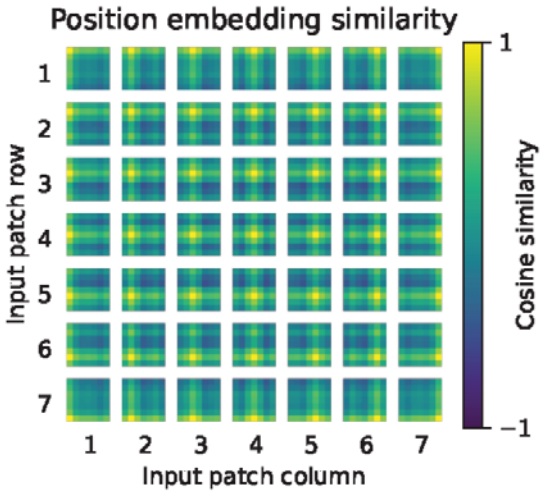


Fig. 7. Similarity of position embeddings of ViT-L/32.

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.

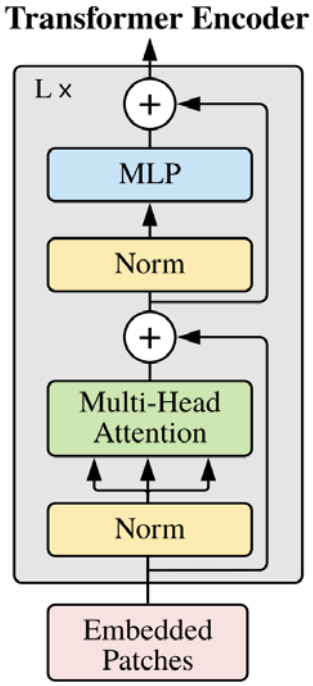
The model learns to encode distance between the image in the similarity of position embeddings. Closer patches tend to have more similar position embeddings.

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[출처] https://yhkim4504.tistory.com/6
[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

2. Transformer Encoder

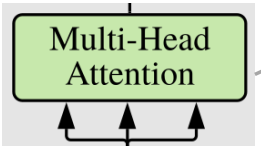
- Now we need the implement Transformer. In ViT only the Encoder is used, the architecture is visualized in the following picture.



(1, 197, 768)

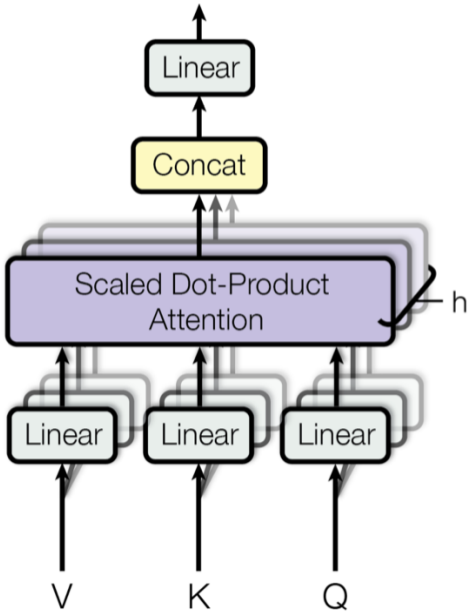
1) Multi Head Attention

- The attention takes three inputs, the famous *queries*, *keys*, and *values*, and computes the attention matrix using queries and values and use it to "attend" to the values. In this case, we are using multi-head attention meaning that the computation is split across n heads with smaller input size.



- 4 fully connected layers, one for queries, keys, values, and a final one dropout.

MHA (Multi Head Attention)



Linear 연산 (Matrix Mult)를 이용해 Q, K, V의 차원을 감소
Q와 K의 차원이 다른 경우 이를 이용해 동일하게 맞춤
h개의 Attention Layer를 병렬적으로 사용 – 더 넓은 계층
출력 직전 Linear 연산을 이용해 Attention Value의 차원을
필요에 따라 변경
이 메커니즘을 통해 병렬 계산에 유리한 구조를 가지게 됨

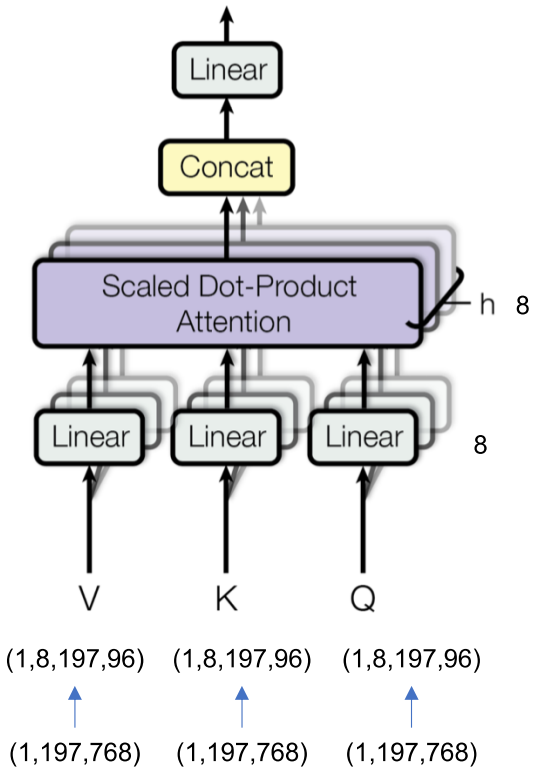
$$\text{Linear}_i(V) = VW_{V,i} \quad W_{V,i} \in \mathbb{R}^{d_v \times d_{model}}$$
$$\text{Linear}_i(K) = KW_{K,i} \quad W_{K,i} \in \mathbb{R}^{d_k \times d_{model}}$$
$$\text{Linear}_i(Q) = QW_{Q,i} \quad W_{Q,i} \in \mathbb{R}^{d_q \times d_{model}}$$

ViT에서의 MHA는 QKV가 같은 텐서로 입력됨. 입력텐서는 3개의 Linear Projection을 통해 임베딩된 후 여러 개의 Head로 나눠진 후 각각 Scaled Dot-Product Attention을 진행함

1 ViT (Visual Transformer) in PyTorch

[출처] https://yhkim4504.tistory.com/6

MHA (Multi Head Attention)



(1) Linear Projection

```
emb_size = 768
num_heads = 8

keys = nn.Linear(emb_size, emb_size)
queries = nn.Linear(emb_size, emb_size)
values = nn.Linear(emb_size, emb_size)
print(keys, queries, values)
```

```
Linear(in_features=768, out_features=768, bias=True)
Linear(in_features=768, out_features=768, bias=True)
Linear(in_features=768, out_features=768, bias=True)
```

먼저 임베딩된 입력텐서를 받아서 다시 임베딩 사이즈로 Linear Projection을 하는 레이어를 3개 만듬. 입력 텐서를 QKV로 만드는 각 레이어는 모델 훈련과정에서 학습됨.

$$q = z \cdot w_q \quad (w_q \in \mathbb{R}^{D \times D_h})$$

$$k = z \cdot w_k \quad (w_k \in \mathbb{R}^{D \times D_h})$$

$$v = z \cdot w_v \quad (w_v \in \mathbb{R}^{D \times D_h})$$

$$[q, k, v] = z \cdot U_{qkv} \quad (U_{qkv} \in \mathbb{R}^{D \times 3D_h})$$

(2) Multi Head

```
z=PatchEmbedding()(x)

queries = rearrange(queries(z), "b n (h d) -> b h n d", h=num_heads)
keys = rearrange(keys(z), "b n (h d) -> b h n d", h=num_heads)
values  = rearrange(values(z), "b n (h d) -> b h n d", h=num_heads)

print('shape :', queries.shape, keys.shape, values.shape)
```

```
shape : torch.Size([1, 8, 197, 96]) torch.Size([1, 8, 197, 96])
        torch.Size([1, 8, 197, 96])
```

```
z=PatchEmbedding()(x)
queries(z).shape
```

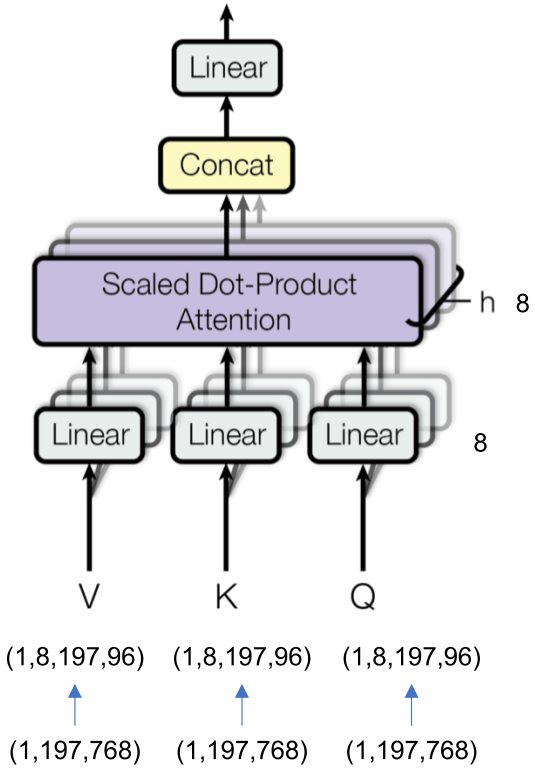
```
torch.Size([1, 197, 768])
```

이후 각 Linear Projection을 거친 QKV를 rearrange를 통해 8개의 Multi-Head로 나눠주게 됩니다.

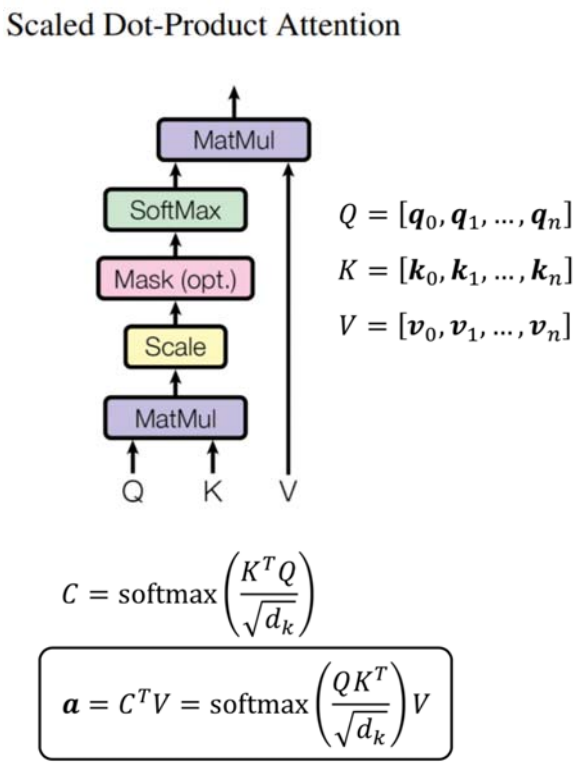
1 ViT (Visual Transformer) in PyTorch

[출처] https://yhkim4504.tistory.com/6
[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

MHA (Multi Head Attention)



(3) Scaled Dot Product Attention



위 그림과 같이 Q와 K를 곱합니다. einops를 이용해 자동으로 transpose 후 내적이 진행됩니다. 그 다음 scaling 해준 후 얻어진 Attention Score와 V를 내적하고 다시 emb_size로 rearrange 하면 MHA의 output이 나오게 됩니다.

```
# Queries * Keys
energy = torch.einsum('bhqd, bhkd -> bhqk', queries, keys)
print('energy :', energy.shape)

# Get Attention Score
scaling = emb_size ** (1/2)
att = F.softmax(energy, dim=-1) / scaling
print('att :', att.shape)

# Attention Score * values
out = torch.einsum('bhal, bhlv -> bhav ', att, values)
print('out :', out.shape)

# Rearrage to emb_size
out = rearrange(out, "b h n d -> b n (h d)")
print('out2 : ', out.shape)
```

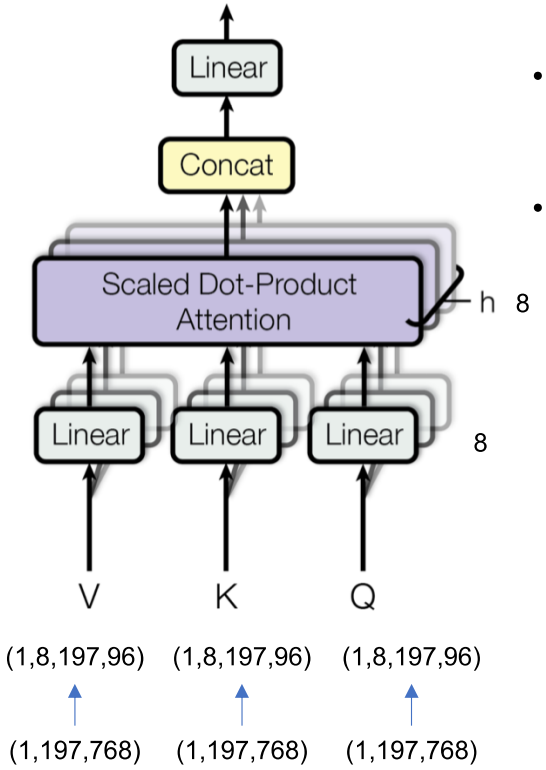
```
energy : torch.Size([1, 8, 197, 197])
att : torch.Size([1, 8, 197, 197])
out : torch.Size([1, 8, 197, 96])
out2 :  torch.Size([1, 197, 768])
```

torch.einsum
https://rockt.github.io/2018/04/30/einsum
https://baekyeongmin.github.io/dev/einsum/

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

2. Transformer Encoder



- 4 fully connected layers, one for queries, keys, values, and a final one dropout.
- The *forward* method takes as input the queries, keys, and values from the previous layer and projects them using the three linear layers. Since we implementing multi heads attention, we have to *rearrange* the result in multiple heads.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_size: int = 768, num_heads: int = 8, dropout: float = 0):
        super().__init__()
        self.emb_size = emb_size
        self.num_heads = num_heads
        self.keys = nn.Linear(emb_size, emb_size)
        self.queries = nn.Linear(emb_size, emb_size)
        self.values = nn.Linear(emb_size, emb_size)
        self.att_drop = nn.Dropout(dropout)
        self.projection = nn.Linear(emb_size, emb_size)

    def forward(self, x : Tensor, mask: Tensor = None) -> Tensor:
        # split keys, queries and values in num_heads
        queries = rearrange(self.queries(x), "b n (h d) -> b h n d", h=self.num_heads)
        keys = rearrange(self.keys(x), "b n (h d) -> b h n d", h=self.num_heads)
        values  = rearrange(self.values(x), "b n (h d) -> b h n d", h=self.num_heads)

        # sum up over the last axis
        # b : batch, h : num_heads, q : query_len, k : key_len
        energy = torch.einsum('bhqd, bhkd -> bhqk', queries, keys)
        if mask is not None:
            fill_value = torch.finfo(torch.float32).min  # -3.4028234663852886e+38
            energy.mask_fill(~mask, fill_value)

        scaling = self.emb_size ** (1/2)
        att = F.softmax(energy, dim=-1) / scaling
        att = self.att_drop(att)
        # sum up over the third axis
        out = torch.einsum('bhal, bhlv -> bhav ', att, values)
        out = rearrange(out, "b h n d -> b n (h d)")
        out = self.projection(out)
        return out
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

2. Transformer Encoder

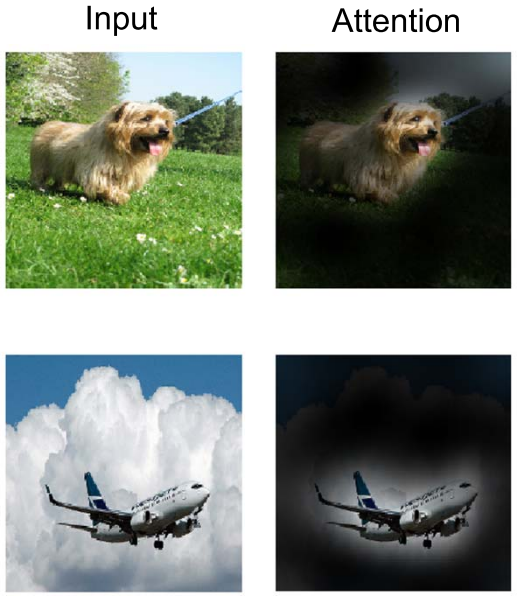


Figure 6: Representative examples of attention from the output token to the input space. See Appendix D.7 for details.

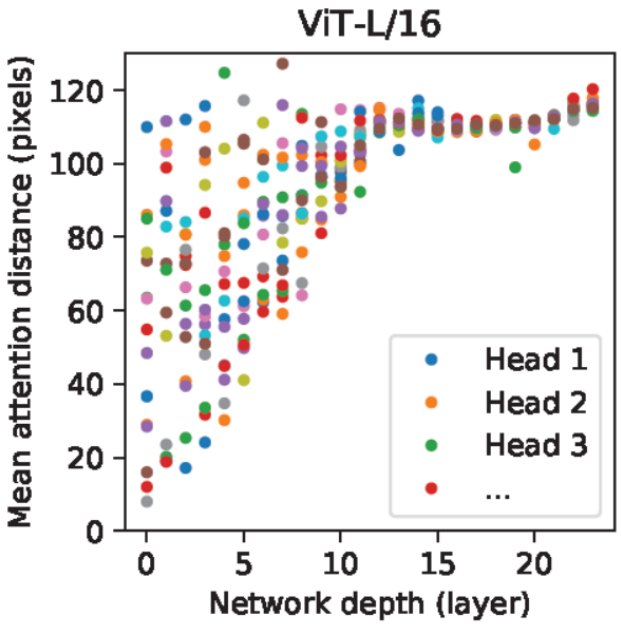


Fig. 7. Size of attended area by head and network depth. Each dot shows the mean attention distance across images for one of 16 heads at one layer.

Self-attention allows ViT to integrate information across the entire image even in the lowest layers.

We investigate to what degree the network makes use of this capability. Specifically, we compute the **average distance** in image space across which information is integrated, based on the attention weights (Figure 7, right). This "**attention distance**" is analogous to receptive field size in CNNs.

We find that some heads attend to most of the image already in the lowest layers, showing that the ability to integrate information globally is indeed used by the model.

Other attention heads have consistently small attention distances in the low layers. This highly localized attention is less pronounced in hybrid models that apply a ResNet before the Transformer (Figure 7, right), suggesting that it may serve a similar function as early convolutional layers in CNNs.

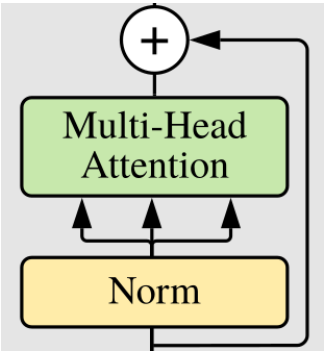
Further, the attention distance increases with network depth. Globally, we find that the model attends to image regions that are semantically relevant for classification (Figure 6).

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

2. Transformer Encoder

2) Residual



- The transformer block has residuals connection
- We can create a nice wrapper to perform the residual addition, it will be handy later on.

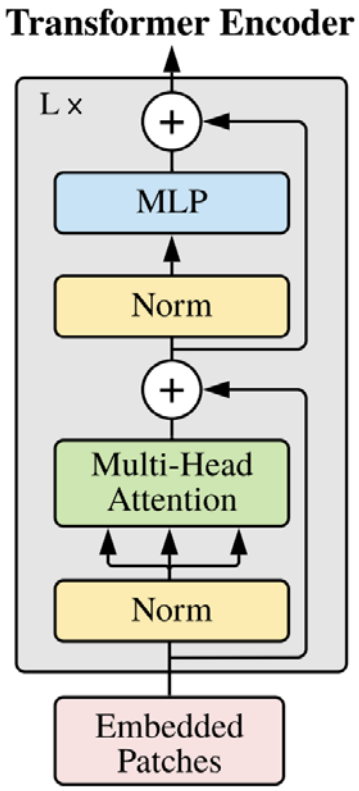
```
class ResidualAdd(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

    def forward(self, x, **kwargs):
        res = x
        x = self.fn(x, **kwargs)
        x += res
        return x
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

2. Transformer Encoder

3) MLP



- The attention's output is passed to a fully connected layer composed of two layers that upsample by a factor of *expansion* the input

```
class FeedForwardBlock(nn.Sequential):
    def __init__(self, emb_size: int, expansion: int = 4, drop_p: float = 0.):
        super().__init__(
            nn.Linear(emb_size, expansion * emb_size),
            nn.GELU(),
            nn.Dropout(drop_p),
            nn.Linear(expansion * emb_size, emb_size),
        )
```

- *Just a quick side note*. I don't know why but I've never seen people subclassing nn.Sequential to avoid writing the forward method. Start doing it, this is how object programming works!

```
class TransformerEncoderBlock(nn.Sequential):
    def __init__(self,
                 emb_size: int = 768,
                 drop_p: float = 0.,
                 forward_expansion: int = 4,
                 forward_drop_p: float = 0.,
                 ** kwargs):
        super().__init__(
            ResidualAdd(nn.Sequential(
                nn.LayerNorm(emb_size),
                MultiHeadAttention(emb_size, **kwargs),
                nn.Dropout(drop_p)
            )),
            ResidualAdd(nn.Sequential(
                nn.LayerNorm(emb_size),
                FeedForwardBlock(
                    emb_size, expansion=forward_expansion, drop_p=forward_drop_p),
                nn.Dropout(drop_p)
            )
            ))
```

- Finally, we can create the Transformer Encoder Block
- *ResidualAdd* allows us to define this block in an elegant way

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

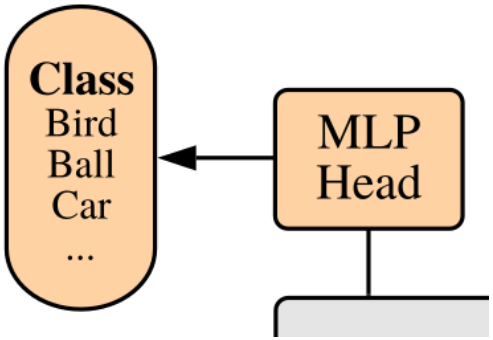
2. Transformer Encoder

- Test it.

```
patches_embedded = PatchEmbedding()(x)
TransformerEncoderBlock()(patches_embedded).shape
```

```
torch.Size([1, 197, 768])
```

- you can also *PyTorch build-in multi-head attention* but it will expect 3 inputs: queries, keys, and values. You can subclass it and pass the same input



Transformer

- In ViT only the Encoder part of the original transformer is used. Easily, the encoder is L blocks of TransformerBlock.

```
class TransformerEncoder(nn.Sequential):
    def __init__(self, depth: int = 12, **kwargs):
        super().__init__(*[TransformerEncoderBlock(**kwargs) for _ in range(depth)])
```

Head

- The last layer is a normal fully connect that gives the class probability. It first performs a basic mean over the whole sequence.

```
class ClassificationHead(nn.Sequential):
    def __init__(self, emb_size: int = 768, n_classes: int = 1000):
        super().__init__(
            Reduce('b n e -> b e', reduction='mean'),
            nn.LayerNorm(emb_size),
            nn.Linear(emb_size, n_classes))
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

3. Vi(sual) T(rasnformer)

- We can compose PatchEmbedding, TransformerEncoder and ClassificationHead to create the final ViT architecture.

```
class ViT(nn.Sequential):
    def __init__(self,
                 in_channels: int = 3,
                 patch_size: int = 16,
                 emb_size: int = 768,
                 img_size: int = 224,
                 depth: int = 12,
                 n_classes: int = 1000,
                 **kwargs):
        super().__init__(
            PatchEmbedding(in_channels, patch_size, emb_size, img_size),
            TransformerEncoder(depth, emb_size=emb_size, **kwargs),
            ClassificationHead(emb_size, n_classes)
        )
```

- We can use torchsummary to check the number of parameters

```
summary(ViT(), (3, 224, 224), device='cpu')
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

3. Vi(sual) T(rasnformer)

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 768, 14, 14]	590,592
Rearrange-2	[-1, 196, 768]	0
PatchEmbedding-3	[-1, 197, 768]	0
LayerNorm-4	[-1, 197, 768]	1,536
Linear-5	[-1, 197, 768]	590,592
Linear-6	[-1, 197, 768]	590,592
Linear-7	[-1, 197, 768]	590,592
Dropout-8	[-1, 8, 197, 197]	0
Linear-9	[-1, 197, 768]	590,592
MultiHeadAttention-10	[-1, 197, 768]	0
Dropout-11	[-1, 197, 768]	0
ResidualAdd-12	[-1, 197, 768]	0
LayerNorm-13	[-1, 197, 768]	1,536
Linear-14	[-1, 197, 3072]	2,362,368
GELU-15	[-1, 197, 3072]	0
Dropout-16	[-1, 197, 3072]	0
Linear-17	[-1, 197, 768]	2,360,064
Dropout-18	[-1, 197, 768]	0
ResidualAdd-19	[-1, 197, 768]	0
LayerNorm-20	[-1, 197, 768]	1,536
Linear-21	[-1, 197, 768]	590,592
Linear-22	[-1, 197, 768]	590,592
Linear-23	[-1, 197, 768]	590,592
Dropout-24	[-1, 8, 197, 197]	0
Linear-25	[-1, 197, 768]	590,592
MultiHeadAttention-26	[-1, 197, 768]	0
Dropout-27	[-1, 197, 768]	0
ResidualAdd-28	[-1, 197, 768]	0
LayerNorm-29	[-1, 197, 768]	1,536
Linear-30	[-1, 197, 3072]	2,362,368
GELU-31	[-1, 197, 3072]	0
Dropout-32	[-1, 197, 3072]	0
Linear-33	[-1, 197, 768]	2,360,064
Dropout-34	[-1, 197, 768]	0
ResidualAdd-35	[-1, 197, 768]	0
LayerNorm-36	[-1, 197, 768]	1,536
Linear-37	[-1, 197, 768]	590,592
Linear-38	[-1, 197, 768]	590,592
Linear-39	[-1, 197, 768]	590,592
Dropout-40	[-1, 8, 197, 197]	0
Linear-41	[-1, 197, 768]	590,592
MultiHeadAttention-42	[-1, 197, 768]	0
Dropout-43	[-1, 197, 768]	0
ResidualAdd-44	[-1, 197, 768]	0
LayerNorm-45	[-1, 197, 768]	1,536
Linear-46	[-1, 197, 3072]	2,362,368
GELU-47	[-1, 197, 3072]	0
Dropout-48	[-1, 197, 3072]	0
Linear-49	[-1, 197, 768]	2,360,064
Dropout-50	[-1, 197, 768]	0
ResidualAdd-51	[-1, 197, 768]	0
LayerNorm-52	[-1, 197, 768]	1,536
Linear-53	[-1, 197, 768]	590,592
Linear-54	[-1, 197, 768]	590,592
Linear-55	[-1, 197, 768]	590,592
Dropout-56	[-1, 8, 197, 197]	0
Linear-57	[-1, 197, 768]	590,592
MultiHeadAttention-58	[-1, 197, 768]	0
Dropout-59	[-1, 197, 768]	0
ResidualAdd-60	[-1, 197, 768]	0
LayerNorm-61	[-1, 197, 768]	1,536
Linear-62	[-1, 197, 3072]	2,362,368
GELU-63	[-1, 197, 3072]	0
Dropout-64	[-1, 197, 3072]	0
Linear-65	[-1, 197, 768]	2,360,064
Dropout-66	[-1, 197, 768]	0
ResidualAdd-67	[-1, 197, 768]	0
LayerNorm-68	[-1, 197, 768]	1,536
Linear-69	[-1, 197, 768]	590,592
Linear-70	[-1, 197, 768]	590,592
Linear-71	[-1, 197, 768]	590,592
Dropout-72	[-1, 8, 197, 197]	0
Linear-73	[-1, 197, 768]	590,592
MultiHeadAttention-74	[-1, 197, 768]	0
Dropout-75	[-1, 197, 768]	0
ResidualAdd-76	[-1, 197, 768]	0
LayerNorm-77	[-1, 197, 768]	1,536
Linear-78	[-1, 197, 3072]	2,362,368
GELU-79	[-1, 197, 3072]	0
Dropout-80	[-1, 197, 3072]	0
Linear-81	[-1, 197, 768]	2,360,064
Dropout-82	[-1, 197, 768]	0
ResidualAdd-83	[-1, 197, 768]	0
LayerNorm-84	[-1, 197, 768]	1,536
Linear-85	[-1, 197, 768]	590,592
Linear-86	[-1, 197, 768]	590,592
Linear-87	[-1, 197, 768]	590,592
Dropout-88	[-1, 8, 197, 197]	0
Linear-89	[-1, 197, 768]	590,592

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

3. Vi(sual) T(rasnformer)

MultiHeadAttention-90	[-1, 197, 768]	0	MultiHeadAttention-122	[-1, 197, 768]	0	MultiHeadAttention-154	[-1, 197, 768]	0
Dropout-91	[-1, 197, 768]	0	Dropout-123	[-1, 197, 768]	0	Dropout-155	[-1, 197, 768]	0
ResidualAdd-92	[-1, 197, 768]	0	ResidualAdd-124	[-1, 197, 768]	0	ResidualAdd-156	[-1, 197, 768]	0
LayerNorm-93	[-1, 197, 768]	1,536	LayerNorm-125	[-1, 197, 768]	1,536	LayerNorm-157	[-1, 197, 768]	1,536
Linear-94	[-1, 197, 3072]	2,362,368	Linear-126	[-1, 197, 3072]	2,362,368	Linear-158	[-1, 197, 3072]	2,362,368
GELU-95	[-1, 197, 3072]	0	GELU-127	[-1, 197, 3072]	0	GELU-159	[-1, 197, 3072]	0
Dropout-96	[-1, 197, 3072]	0	Dropout-128	[-1, 197, 3072]	0	Dropout-160	[-1, 197, 3072]	0
Linear-97	[-1, 197, 768]	2,360,064	Linear-129	[-1, 197, 768]	2,360,064	Linear-161	[-1, 197, 768]	2,360,064
Dropout-98	[-1, 197, 768]	0	Dropout-130	[-1, 197, 768]	0	Dropout-162	[-1, 197, 768]	0
ResidualAdd-99	[-1, 197, 768]	0	ResidualAdd-131	[-1, 197, 768]	0	ResidualAdd-163	[-1, 197, 768]	0
LayerNorm-100	[-1, 197, 768]	1,536	LayerNorm-132	[-1, 197, 768]	1,536	LayerNorm-164	[-1, 197, 768]	1,536
Linear-101	[-1, 197, 768]	590,592	Linear-133	[-1, 197, 768]	590,592	Linear-165	[-1, 197, 768]	590,592
Linear-102	[-1, 197, 768]	590,592	Linear-134	[-1, 197, 768]	590,592	Linear-166	[-1, 197, 768]	590,592
Linear-103	[-1, 197, 768]	590,592	Linear-135	[-1, 197, 768]	590,592	Linear-167	[-1, 197, 768]	590,592
Dropout-104	[-1, 8, 197, 197]	0	Dropout-136	[-1, 8, 197, 197]	0	Dropout-168	[-1, 8, 197, 197]	0
Linear-105	[-1, 197, 768]	590,592	Linear-137	[-1, 197, 768]	590,592	Linear-169	[-1, 197, 768]	590,592
MultiHeadAttention-106	[-1, 197, 768]	0	MultiHeadAttention-138	[-1, 197, 768]	0	MultiHeadAttention-170	[-1, 197, 768]	0
Dropout-107	[-1, 197, 768]	0	Dropout-139	[-1, 197, 768]	0	Dropout-171	[-1, 197, 768]	0
ResidualAdd-108	[-1, 197, 768]	0	ResidualAdd-140	[-1, 197, 768]	0	ResidualAdd-172	[-1, 197, 768]	0
LayerNorm-109	[-1, 197, 768]	1,536	LayerNorm-141	[-1, 197, 768]	1,536	LayerNorm-173	[-1, 197, 768]	1,536
Linear-110	[-1, 197, 3072]	2,362,368	Linear-142	[-1, 197, 3072]	2,362,368	Linear-174	[-1, 197, 3072]	2,362,368
GELU-111	[-1, 197, 3072]	0	GELU-143	[-1, 197, 3072]	0	GELU-175	[-1, 197, 3072]	0
Dropout-112	[-1, 197, 3072]	0	Dropout-144	[-1, 197, 3072]	0	Dropout-176	[-1, 197, 3072]	0
Linear-113	[-1, 197, 768]	2,360,064	Linear-145	[-1, 197, 768]	2,360,064	Linear-177	[-1, 197, 768]	2,360,064
Dropout-114	[-1, 197, 768]	0	Dropout-146	[-1, 197, 768]	0	Dropout-178	[-1, 197, 768]	0
ResidualAdd-115	[-1, 197, 768]	0	ResidualAdd-147	[-1, 197, 768]	0	ResidualAdd-179	[-1, 197, 768]	0
LayerNorm-116	[-1, 197, 768]	1,536	LayerNorm-148	[-1, 197, 768]	1,536	LayerNorm-180	[-1, 197, 768]	1,536
Linear-117	[-1, 197, 768]	590,592	Linear-149	[-1, 197, 768]	590,592	Linear-181	[-1, 197, 768]	590,592
Linear-118	[-1, 197, 768]	590,592	Linear-150	[-1, 197, 768]	590,592	Linear-182	[-1, 197, 768]	590,592
Linear-119	[-1, 197, 768]	590,592	Linear-151	[-1, 197, 768]	590,592	Linear-183	[-1, 197, 768]	590,592
Dropout-120	[-1, 8, 197, 197]	0	Dropout-152	[-1, 8, 197, 197]	0	Dropout-184	[-1, 8, 197, 197]	0
Linear-121	[-1, 197, 768]	590,592	Linear-153	[-1, 197, 768]	590,592	Linear-185	[-1, 197, 768]	590,592

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

3. Vi(sual) T(rasnformer)

```
MultiHeadAttention-186            [-1, 197, 768]               0
         Dropout-187            [-1, 197, 768]               0
     ResidualAdd-188            [-1, 197, 768]               0
       LayerNorm-189            [-1, 197, 768]           1,536
          Linear-190           [-1, 197, 3072]       2,362,368
            GELU-191           [-1, 197, 3072]               0
         Dropout-192           [-1, 197, 3072]               0
          Linear-193            [-1, 197, 768]       2,360,064
         Dropout-194            [-1, 197, 768]               0
     ResidualAdd-195            [-1, 197, 768]               0
          Reduce-196                 [-1, 768]               0
       LayerNorm-197                 [-1, 768]           1,536
          Linear-198                [-1, 1000]         769,000
================================================================
Total params: 86,415,592
Trainable params: 86,415,592
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 364.33
Params size (MB): 329.65
Estimated Total Size (MB): 694.56
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://gaussian37.github.io/dl-concept-vit/

- Another code

```
import torch
import torch.nn as nn

class LinearProjection(nn.Module):

    def __init__(self, patch_vec_size, num_patches, latent_vec_dim, drop_rate):
        super().__init__()
        self.linear_proj = nn.Linear(patch_vec_size, latent_vec_dim)
        self.cls_token = nn.Parameter(torch.randn(1, latent_vec_dim))
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches+1,
latent_vec_dim))
        self.dropout = nn.Dropout(drop_rate)

    def forward(self, x):
        batch_size = x.size(0)
        x = torch.cat([self.cls_token.repeat(batch_size, 1, 1), self.linear_proj(x)],
dim=1)
        x += self.pos_embedding
        x = self.dropout(x)
        return x
```

```
class MultiheadedSelfAttention(nn.Module):
    def __init__(self, latent_vec_dim, num_heads, drop_rate):
        super().__init__()
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        self.num_heads = num_heads
        self.latent_vec_dim = latent_vec_dim
        self.head_dim = int(latent_vec_dim / num_heads)
        self.query = nn.Linear(latent_vec_dim, latent_vec_dim)
        self.key = nn.Linear(latent_vec_dim, latent_vec_dim)
        self.value = nn.Linear(latent_vec_dim, latent_vec_dim)
        self.scale = torch.sqrt(latent_vec_dim*torch.ones(1)).to(device)
        self.dropout = nn.Dropout(drop_rate)

    def forward(self, x):
        batch_size = x.size(0)
        q = self.query(x)
        k = self.key(x)
        v = self.value(x)
        q = q.view(batch_size, -1, self.num_heads, self.head_dim).permute(0,2,1,3)
        k = k.view(batch_size, -1, self.num_heads, self.head_dim).permute(0,2,3,1) # k.t
        v = v.view(batch_size, -1, self.num_heads, self.head_dim).permute(0,2,1,3)
        attention = torch.softmax(q @ k / self.scale, dim=-1)
        x = self.dropout(attention) @ v
        x = x.permute(0,2,1,3).reshape(batch_size, -1, self.latent_vec_dim)

        return x, attention
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://gaussian37.github.io/dl-concept-vit/

```
class TFencoderLayer(nn.Module):
    def __init__(self, latent_vec_dim, num_heads, mlp_hidden_dim, drop_rate):
        super().__init__()
        self.ln1 = nn.LayerNorm(latent_vec_dim)
        self.ln2 = nn.LayerNorm(latent_vec_dim)
        self.msa = MultiheadedSelfAttention(latent_vec_dim=latent_vec_dim,
num_heads=num_heads, drop_rate=drop_rate)
        self.dropout = nn.Dropout(drop_rate)
        self.mlp = nn.Sequential(nn.Linear(latent_vec_dim, mlp_hidden_dim),
                                 nn.GELU(), nn.Dropout(drop_rate),
                                 nn.Linear(mlp_hidden_dim, latent_vec_dim),
                                 nn.Dropout(drop_rate))

    def forward(self, x):
        z = self.ln1(x)
        z, att = self.msa(z)
        z = self.dropout(z)
        x = x + z
        z = self.ln2(x)
        z = self.mlp(z)
        x = x + z

        return x, att
```

```
class VisionTransformer(nn.Module):
    def __init__(self, patch_vec_size, num_patches, latent_vec_dim, num_heads,
mlp_hidden_dim, drop_rate, num_layers, num_classes):
        super().__init__()
        self.patchembedding = LinearProjection(patch_vec_size=patch_vec_size,
num_patches=num_patches,
                                               latent_vec_dim=latent_vec_dim,
drop_rate=drop_rate)
        self.transformer =
nn.ModuleList([TFencoderLayer(latent_vec_dim=latent_vec_dim,
num_heads=num_heads, mlp_hidden_dim=mlp_hidden_dim, drop_rate=drop_rate)
                                       for _ in range(num_layers)])

        self.mlp_head = nn.Sequential(nn.LayerNorm(latent_vec_dim),
nn.Linear(latent_vec_dim, num_classes))

    def forward(self, x):
        att_list = []
        x = self.patchembedding(x)
        for layer in self.transformer:
            x, att = layer(x)
            att_list.append(att)
        x = self.mlp_head(x[:,0])

        return x, att_list
```