Spring, 2022
AI6315

Introduction to self-driving cars

6. Vision Transformer(practice)

Prof. Yong-Gu Lee

Course Instructor

Prof. Yong-Gu Lee

Office. Mecha. Bldg. Rm. 210

Tel. 062-715-2396

Email. lygu@gist.ac.kr

Teaching Assistant

Sungjae Lee 이성재

Office. Mecha. Bldg. Rm. 225

Tel. 062-715-3267

Email. leesungjae@gm.gist.ac.kr

Classroom & Class Hours

Online & AI Grad School Rm. TBD 16:00-17:15 every Mon**day and Wednesday**

1. Introduction

- An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, Alexey Dosovitskiy, ICLR 2021

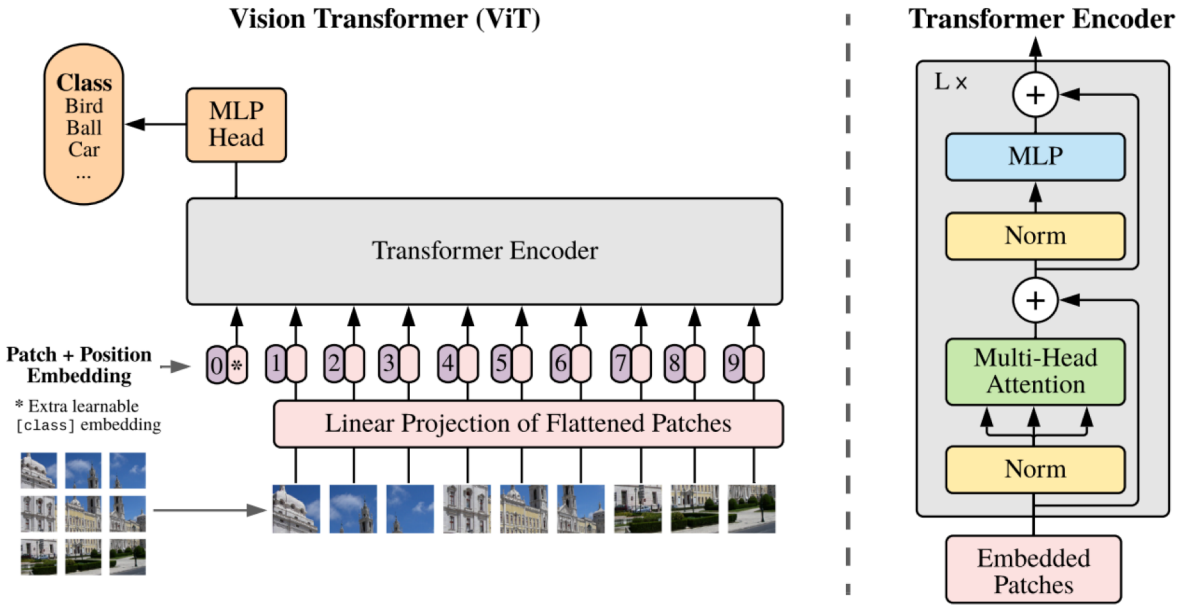


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification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

1. Introduction

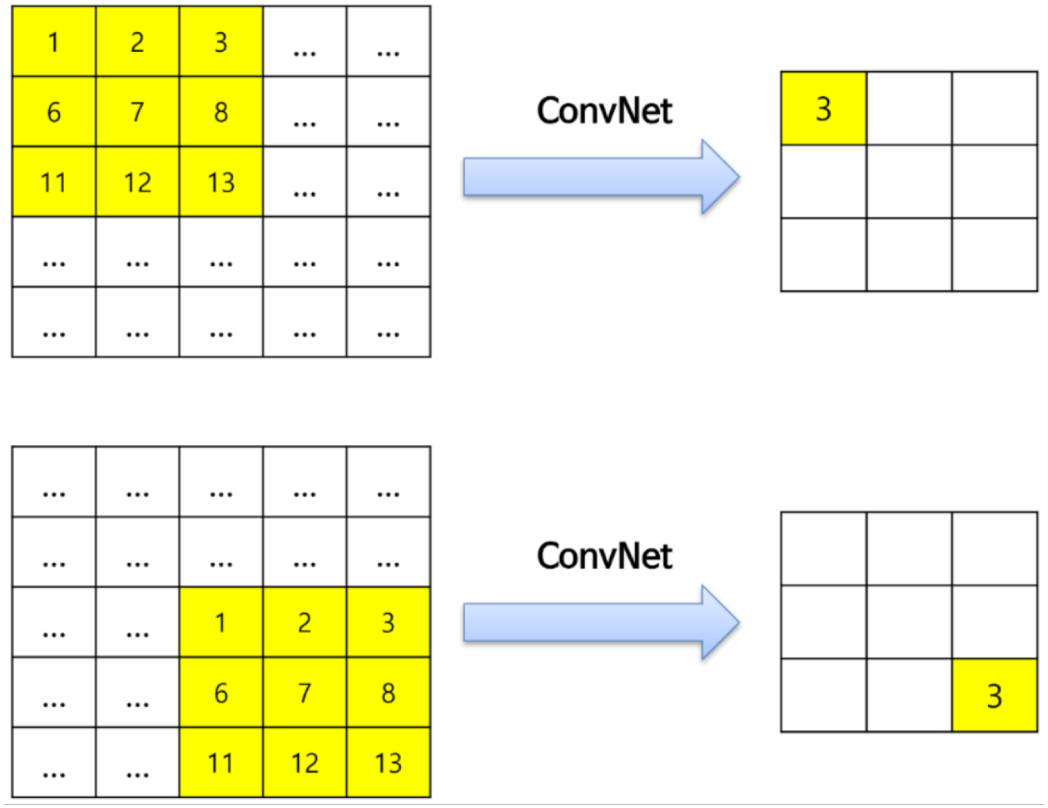
- The transformer used in the NLP field is applied to image classification.
- Does not use CNN.
- SOTA performance in ImageNet, ImageNet-Real and CIFAR-100.
- Advantage
  - Extensibility is good because it uses the transformer structure.
  - Excellent performance for training large scale datasets.
  - Less resource usage than CNN.
- Disadvantage
  - Amount of data required for training.

  => Since there are no inductive bias inherent in CNN such as translation equivariance and locality, more data is needed.

  => Therefore, pre-trained weights learned from JFT-300M are used.

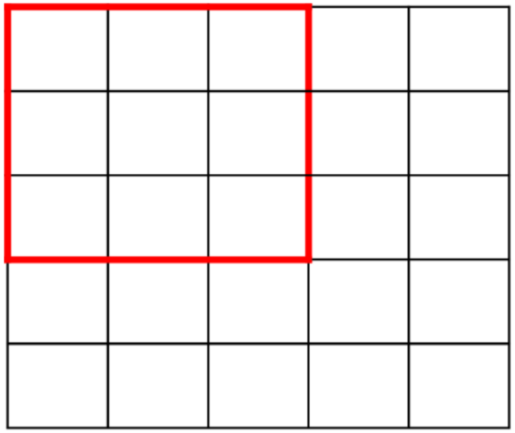
1. Introduction

- Translation equivariance
  : In CNN, the position of the output values changes according to the input value.



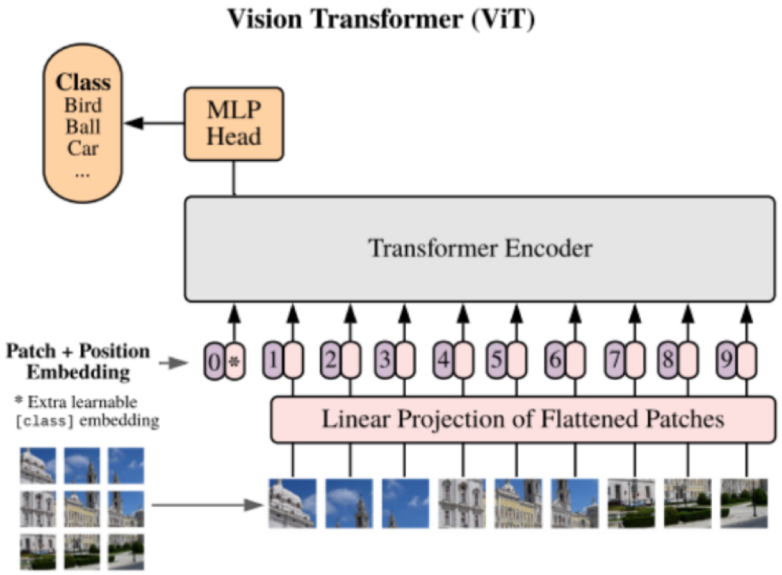
1. Introduction

- Locality
  : In the convolution operation, it is assumed that the filter sees a part of the image from the whole image, and the filter can extract features by looking at this specific region.
  : The transformer model only uses attention.
  => Mechanism for looking at the data as a whole and positioning attention.
  => Requires more data than CNN.



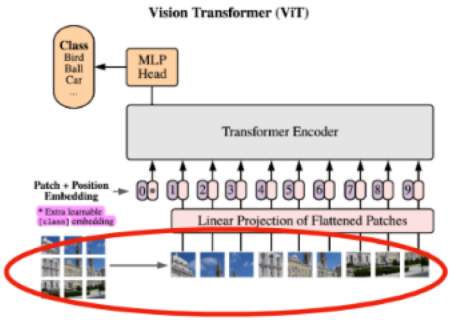
2. Model Architecture

1. After cutting the image into multiple patches(16*16), make a one-dimensional embedding dimension(16*16*3) for each patch.
2. Concatenate the class token and add position embedding to each patch.
3. Execute the transformer encoder n times.
4. Classification is performed through linear operation.



3. Detailed process: How the input image is divided into patches

- (C, H, W) = channel, height, width;
- P= patch length(horizontal/vertical)
- Shape of each patch= (C, P, P);
- N = Number of patches divided.
-



p

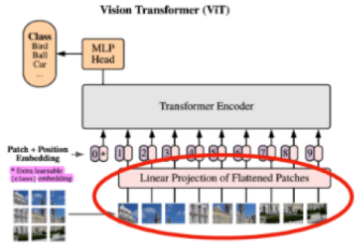
$$x_p \in \mathbb{R}^{N\times(P^2C)}$$

$$x' \in \mathbb{R}^{N\times P\times P\times C}, N = \frac{HW}{P^2}$$

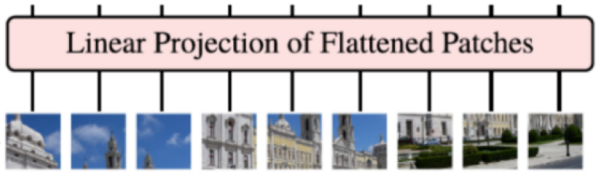
$$x \in \mathbb{R}^{C\times H\times W}$$

3. Detail Process

- Operate x_p with matrix E to embed x_p
- Shape of matrix E = (P^2C, D)
- D means to transform a vector of "embedding dimension(P^2C) shape" into D.
- Shape of x_p = (N, P^2C), Shape of E = (P^2C, D) => (N, D) matrix
- If considering the batch size => (B, N, D), B=batch size.



$$[x_p^1 E; x_p^2 E; \cdots; x_p^N E] \in \mathbb{R}^{N \times D}$$



$E : matrix$

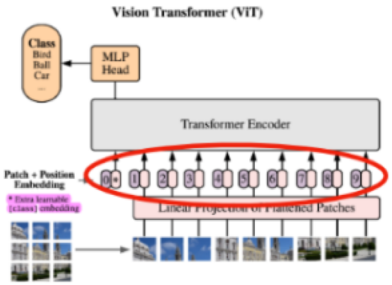
$$x_p \in \mathbb{R}^{N \times (P^2 C)}$$

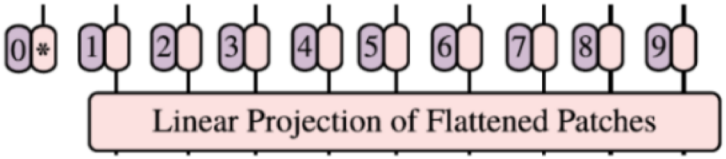
$$x_p^i \in \mathbb{R}^{P^2 C}, E \in \mathbb{R}^{(P^2 C) \times D}$$

D는 embedding dimension

3. Detail Process

- Add class token to embedding result.
- (N,D) => (N+1, D)
- Class token is learnable parameter.
- Finally, by adding a positional embedding, the input value z_0 is obtained.



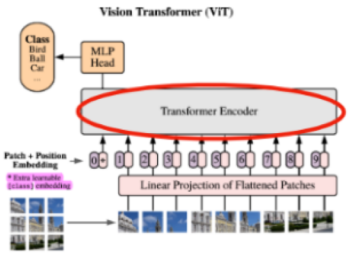


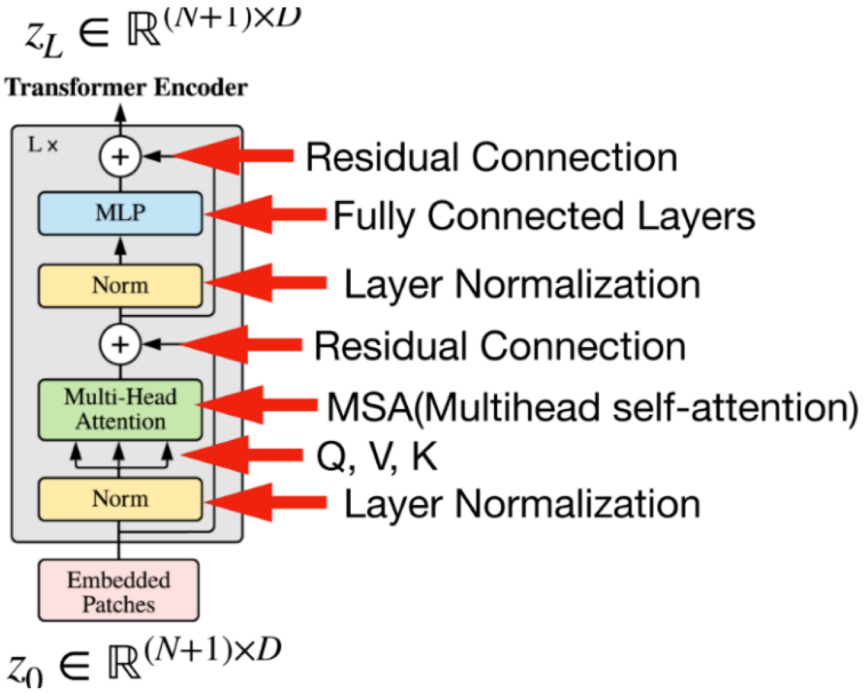
$$[x_p^1 E; x_p^2 E; \cdots; x_p^N E] \in \mathbb{R}^{N \times D} \Rightarrow [x_{cls}; x_p^1 E; x_p^2 E; \cdots; x_p^N E] \in \mathbb{R}^{(N+1) \times D}$$

$$\Rightarrow z_0 = [x_{cls}; x_p^1 E; x_p^2 E; \cdots; x_p^N E] + E_{pos} \in \mathbb{R}^{(N+1) \times D}$$

3. Detail Process

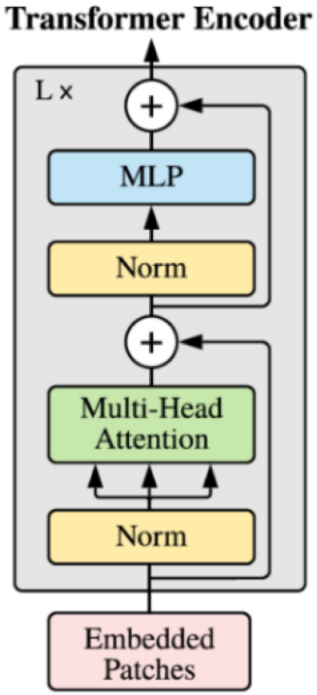
- Keep the shape(size) of the input and output the same to repeat the encoder of the vision transformer L times.
- Input : z_0, Output : z_L
- Multihead attention = self attention => MSA(Multihead self attention)





3. Detail Process

- Transformer implementation by combining LM(Layer Norm), MSA, and MLP operations.



$$z'_l = MSA(LN(z_{l-1})) + z_{l-1}$$

$$z_l = MLP(LN(z'_l)) + z'_l \qquad l = 1,2,\cdots,L$$

3. Detail Process

- Layer Normalization
  : Normalization proceeds for each feature in the D dimension.
  : When the transformer encoder repeats L times, the input at the i-th is called z_i

$$z_i = [z_i^1, z_i^2, z_i^{3,}, \cdots, z_i^N, z_i^{N+1}] \tag{1}$$

$$\mathrm{LN}(z_i^j) = \gamma \frac{z_i^j - \mu_i}{\sigma_i} + \beta \tag{2}$$

$$= \gamma \frac{z_i^j - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta \tag{3}$$

3. Detail Process

- Multi-head attention
  : It has q, k and v, and q, k, and v are configured according to the self-attention structure.
  : SA = self-attention.
  : MSA = Multi head self attention

$$q = z \cdot w_q \quad (w_q \in \mathbb{R}^{D \times D_h}) \tag{4}$$

$$k = z \cdot w_k \quad (w_k \in \mathbb{R}^{D \times D_h}) \tag{5}$$

$$v = z \cdot w_v \quad (w_v \in \mathbb{R}^{D \times D_h}) \tag{6}$$

$$[q, k, v] = z \cdot U_{qkv} \quad (U_{qkv} \in \mathbb{R}^{D \times 3D_h}) \tag{7}$$

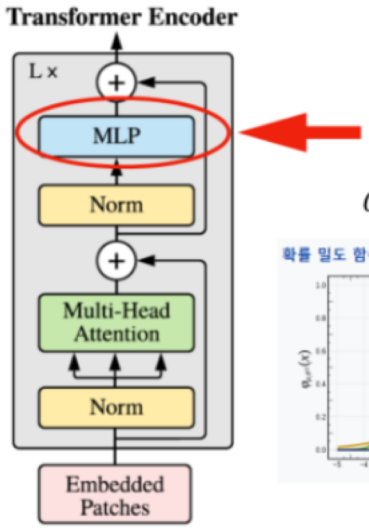
$$A = \text{softmax}(\frac{q \cdot k^T}{\sqrt{D_h}}) \in R^{N \times N} \tag{8}$$

$$\text{SA}(z) = A \cdot v \in R^{N \times D_h} \tag{9}$$

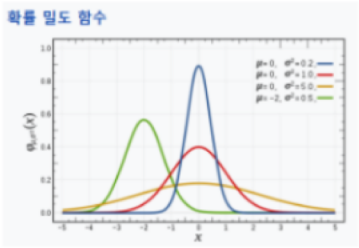
$$\text{MSA}(z) = [\text{SA}_1(z); \text{SA}_2(z); \cdots; \text{SA}_k(z)] U_{msa} \tag{10}$$

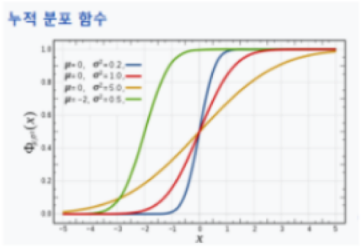
3. Detail Process

- GELU activation



$$GELU(x) = x\,P(X \le x) = x\,\Phi(x)$$





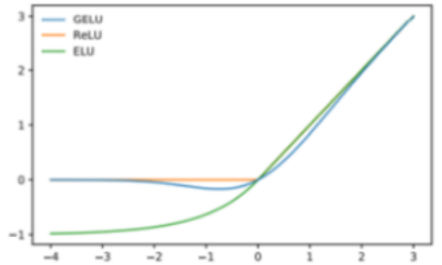
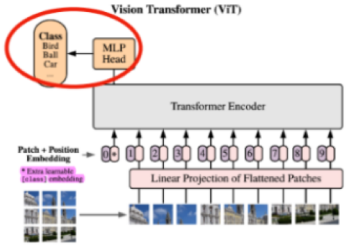


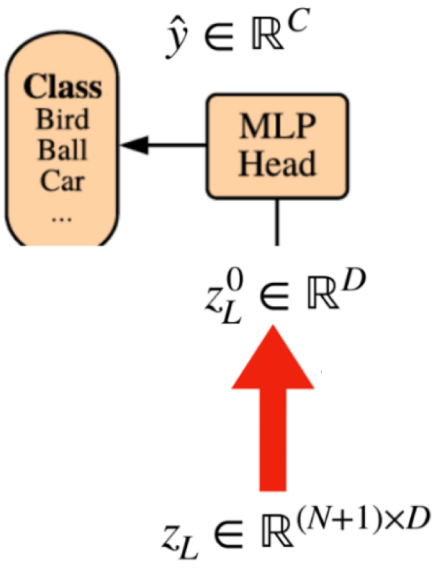
Figure 1: The GELU ($\mu = 0, \sigma = 1$), ReLU, and ELU ($\alpha = 1$).

https://paperswithcode.com/method/gelu

3. Detail Process

- Only the class token part from the last output of the transformer encoder iterated L times is used for the classification problem.
- Finally, class is classified using additional MLP. :)





4. Practice

- Preparation

```
import os
import matplotlib.pyplot as plt
import numpy as np
import PIL

import torch
import torch.nn.functional as F
import torchvision
import torchvision.transforms as T

from timm import create_model
```

4. Practice

- Prepare Model and Data

```
model_name = "vit_base_patch16_224"
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print("device = ", device)
# create a ViT model : https://github.com/rwightman/pytorch-image-models/blob/master/timm/models/vision_transformer.py
model = create_model(model_name, pretrained=True).to(device)
```

```
device =  cpu
Downloading: "https://github.com/rwightman/pytorch-image-models/releases/download/v0.1-vitjx/jx_vit_base_p16_224-80ecf9dd.
```

4. Practice

- Prepare Model and Data

```
# Define transforms for test
IMG_SIZE = (224, 224)
NORMALIZE_MEAN = (0.5, 0.5, 0.5)
NORMALIZE_STD = (0.5, 0.5, 0.5)
transforms = [
              T.Resize(IMG_SIZE),
              T.ToTensor(),
              T.Normalize(NORMALIZE_MEAN, NORMALIZE_STD),
              ]

transforms = T.Compose(transforms)
```

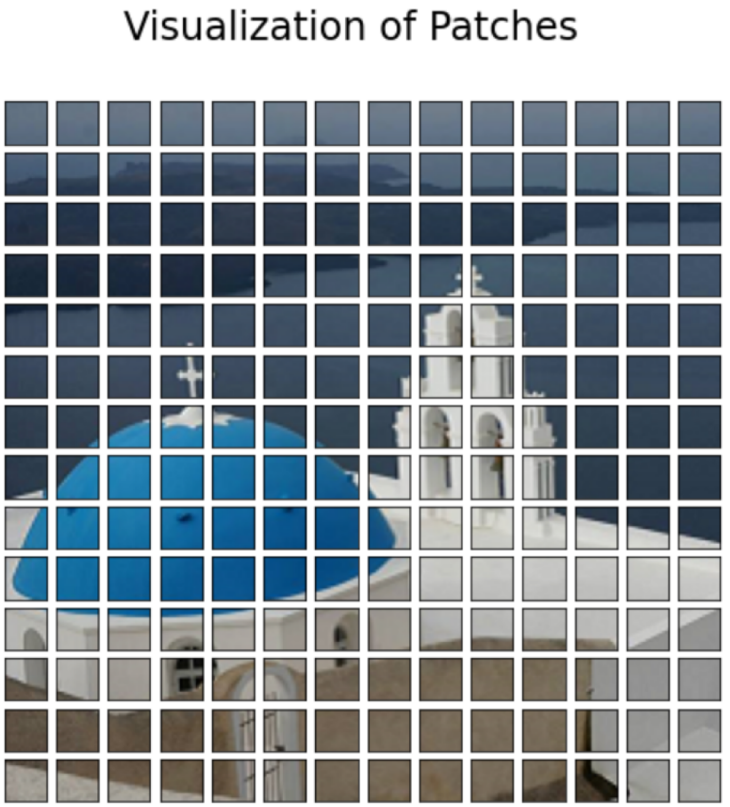
```
%%capture
# ImageNet Labels
!wget https://storage.googleapis.com/bit_models/ilsvrc2012_wordnet_lemmas.txt
imagenet_labels = dict(enumerate(open('ilsvrc2012_wordnet_lemmas.txt')))

# Demo Image
!wget https://github.com/hirotomusiker/schwert_colab_data_storage/blob/master/images/vit_demo/santorini.png?raw=true -O santorini.png
img = PIL.Image.open('santorini.png')
img_tensor = transforms(img).unsqueeze(0).to(device)
```

4. Practice

- Split Image into patches

```
patches = model.patch_embed(img_tensor)  # patch embedding convolution
print("Image tensor: ", img_tensor.shape)
print("Patch embeddings: ", patches.shape)
# This is NOT a part of the pipeline.
# Actually the image is divided into patch embeddings by Conv2d
# with stride=(16, 16) shown above.
fig = plt.figure(figsize=(8, 8))
fig.suptitle("Visualization of Patches", fontsize=24)
fig.add_axes()
img = np.asarray(img)
for i in range(0, 196):
    x = i % 14
    y = i // 14
    patch = img[y*16:(y+1)*16, x*16:(x+1)*16]
    ax = fig.add_subplot(14, 14, i+1)
    ax.axes.get_xaxis().set_visible(False)
    ax.axes.get_yaxis().set_visible(False)
    ax.imshow(patch)
```



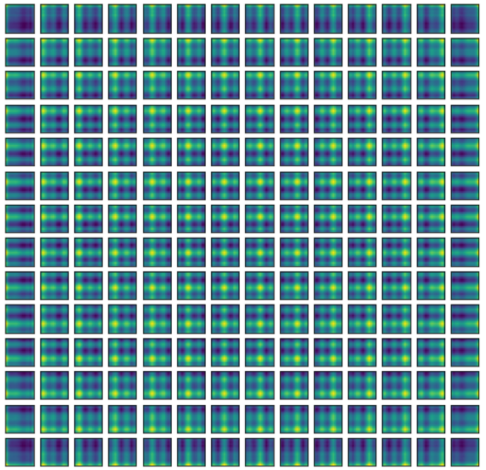
4. Practice

- Add Position Embeddings

```
pos_embed = model.pos_embed
print(pos_embed.shape)
```

```
# Visualize position embedding similarities.
# One cell shows cos similarity between an embedding and all the other embeddings.
cos = torch.nn.CosineSimilarity(dim=1, eps=1e-6)
fig = plt.figure(figsize=(8, 8))
fig.suptitle("Visualization of position embedding similarities", fontsize=24)
for i in range(1, pos_embed.shape[1]):
    sim = F.cosine_similarity(pos_embed[0, i:i+1], pos_embed[0, 1:], dim=1)
    sim = sim.reshape((14, 14)).detach().cpu().numpy()
    ax = fig.add_subplot(14, 14, i)
    ax.axes.get_xaxis().set_visible(False)
    ax.axes.get_yaxis().set_visible(False)
    ax.imshow(sim)
transformer_input = torch.cat((model.cls_token, patches), dim=1) + pos_embed
print("Transformer input: ", transformer_input.shape)
```

Visualization of position embedding similarities



4. Practice

- Transformer

```
print("Input tensor to Transformer (z0): ", transformer_input.shape)
x = transformer_input.clone()
for i, blk in enumerate(model.blocks):
    print("Entering the Transformer Encoder {}".format(i))
    x = blk(x)
x = model.norm(x)
transformer_output = x[:, 0]
print("Output vector from Transformer (z12-0):", transformer_output.shape)
```

```
Input tensor to Transformer (z0):  torch.Size([1, 197, 768])
Entering the Transformer Encoder 0
Entering the Transformer Encoder 1
Entering the Transformer Encoder 2
Entering the Transformer Encoder 3
Entering the Transformer Encoder 4
Entering the Transformer Encoder 5
Entering the Transformer Encoder 6
Entering the Transformer Encoder 7
Entering the Transformer Encoder 8
Entering the Transformer Encoder 9
Entering the Transformer Encoder 10
Entering the Transformer Encoder 11
Output vector from Transformer (z12-0): torch.Size([1, 768])
```

4. Practice

- How to Attention works

```
print("Transformer Multi-head Attention block:")
attention = model.blocks[0].attn
print(attention)
print("input of the transformer encoder:", transformer_input.shape)
```

```
Transformer Multi-head Attention block:
Attention(
  (qkv): Linear(in_features=768, out_features=2304, bias=True)
  (attn_drop): Dropout(p=0.0, inplace=False)
  (proj): Linear(in_features=768, out_features=768, bias=True)
  (proj_drop): Dropout(p=0.0, inplace=False)
)
input of the transformer encoder: torch.Size([1, 197, 768])
```

```
# fc layer to expand the dimension
transformer_input_expanded = attention.qkv(transformer_input)[0]
print("expanded to: ", transformer_input_expanded.shape)
```

```
expanded to:  torch.Size([197, 2304])
```

```
# Split qkv into mulitple q, k, and v vectors for multi-head attantion
qkv = transformer_input_expanded.reshape(197, 3, 12, 64)  # (N=197, (qkv), H=12, D/H=64)
print("split qkv : ", qkv.shape)
q = qkv[:, 0].permute(1, 0, 2)  # (H=12, N=197, D/H=64)
k = qkv[:, 1].permute(1, 0, 2)  # (H=12, N=197, D/H=64)
kT = k.permute(0, 2, 1)  # (H=12, D/H=64, N=197)
print("transposed ks: ", kT.shape)
```

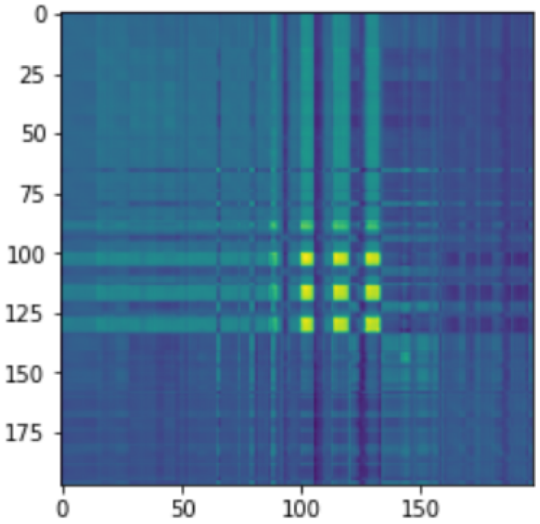
```
split qkv :  torch.Size([197, 3, 12, 64])
transposed ks:  torch.Size([12, 64, 197])
```

4. Practice

- How to Attention works

```
# Attention Matrix
attention_matrix = q @ kT
print("attention matrix: ", attention_matrix.shape)
plt.imshow(attention_matrix[3].detach().cpu().numpy())
```

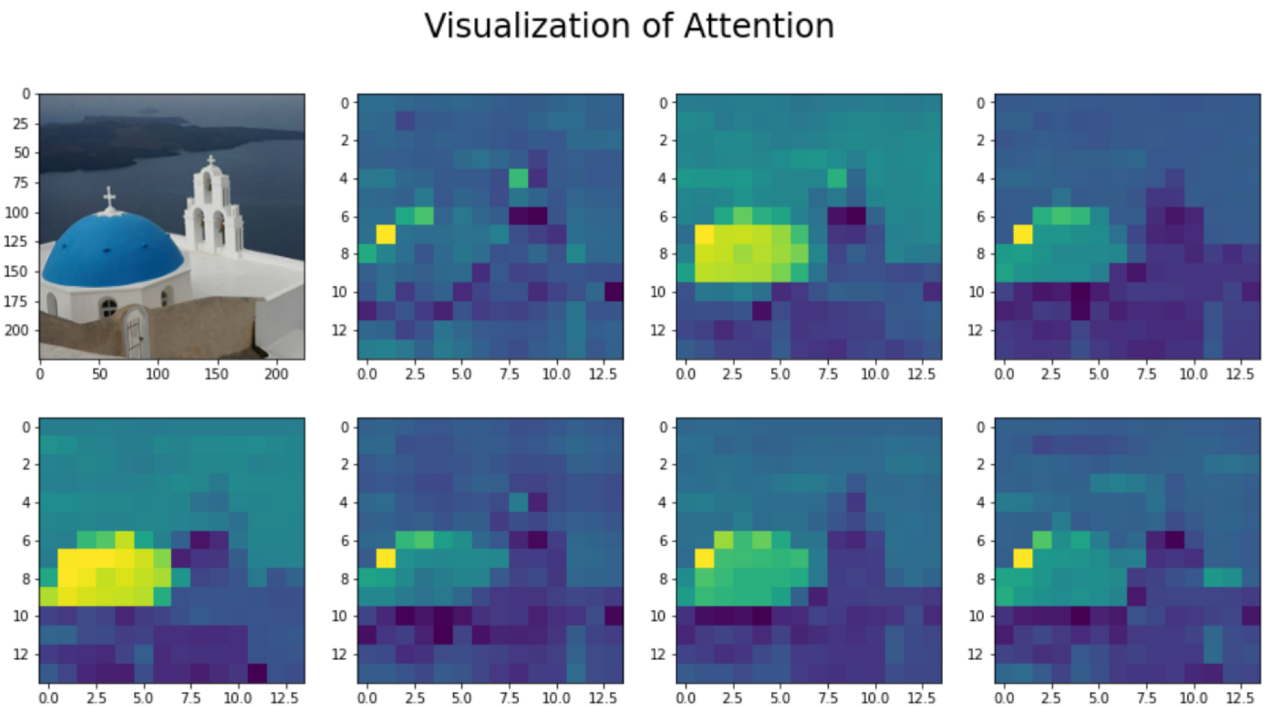
```
attention matrix:  torch.Size([12, 197, 197])
<matplotlib.image.AxesImage at 0x7fb0aa9c0d68>
```



4. Practice

- Visualize :)

```
# Visualize attention matrix
fig = plt.figure(figsize=(16, 8))
fig.suptitle("Visualization of Attention", fontsize=24)
fig.add_axes()
img = np.asarray(img)
ax = fig.add_subplot(2, 4, 1)
ax.imshow(img)
for i in range(7):  # visualize the 100th rows of attention matrices in the 0-7th heads
    attn_heatmap = attention_matrix[i, 100, 1:].reshape((14, 14)).detach().cpu().numpy()
    ax = fig.add_subplot(2, 4, i+2)
    ax.imshow(attn_heatmap)
```

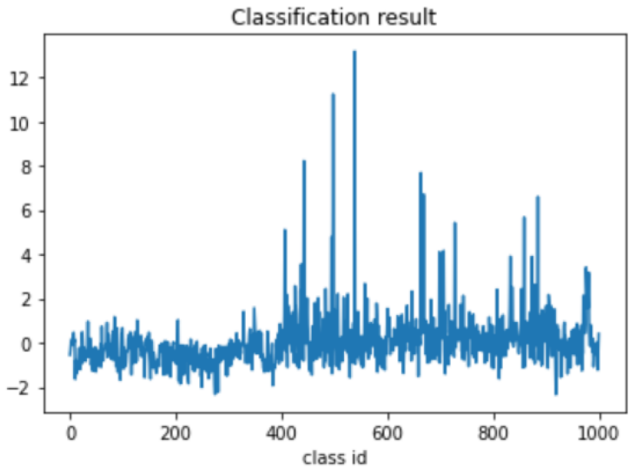


4. Practice

- MLP(Classification) Head

```
print("Classification head: ", model.head)
result = model.head(transformer_output)
result_label_id = int(torch.argmax(result))
plt.plot(result.detach().cpu().numpy()[0])
plt.title("Classification result")
plt.xlabel("class id")
print("Inference result : id = {}, label name = {}".format(
    result_label_id, imagenet_labels[result_label_id]))
```

```
Classification head:  Linear(in_features=768, out_features=1000, bias=True)
Inference result : id = 538, label name = dome
```



HW5

- Ipynb format
- Filename: HW5_student number_name.ipynb
  (ex HW5_20220308_sungjae lee.ipynb)
- Due : June. 1 (Mon.) 10 P.M.
- Send to 'leesungjae@gm.gist.ac.kr'

Elements

We learned ViT today.
You need to write the code which we learned today in Jupyter file.
Thus, you need to follow the step to check ViT process
In this time, you will understand the process by writing the code yourself.

** You explain the results for each cell in ipynb files(use markdown cell).
** accuracy is not important for score. Just try this process following today's class.