An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

DEEP LEARNING BASED ANOMALY DETECTION MODELING (GEK6207.01-00)
Coding practice

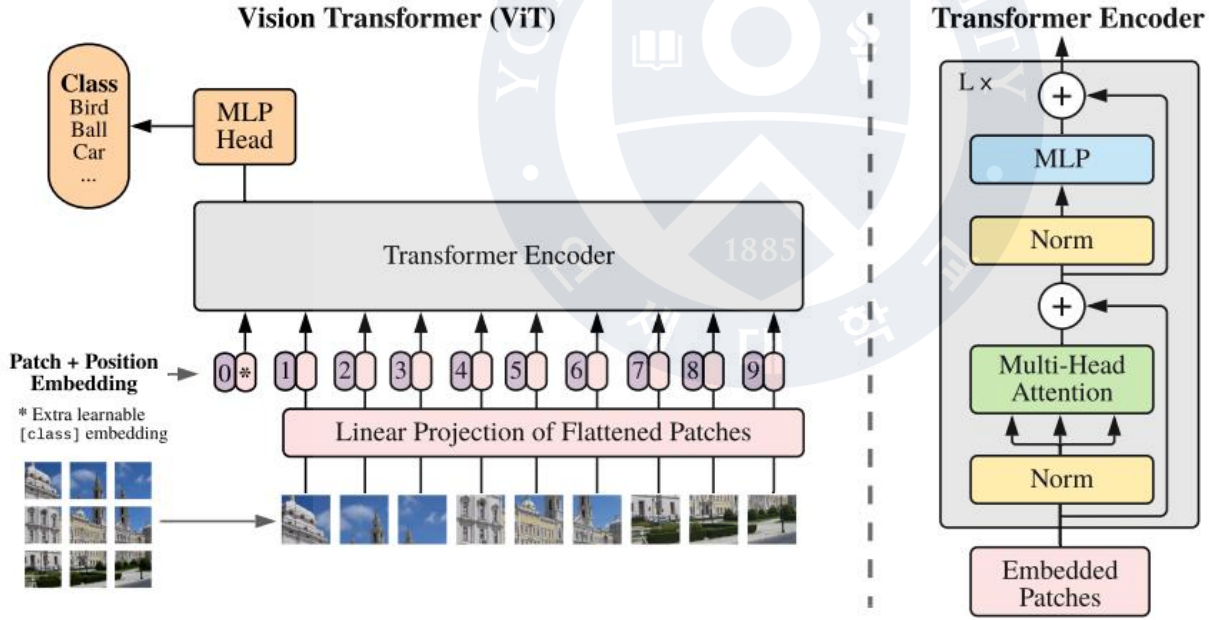
Sunghyun Ahn
skd@yonsei.ac.kr

<2023/11/16>

2 Vision Transformer

What is a ViT?

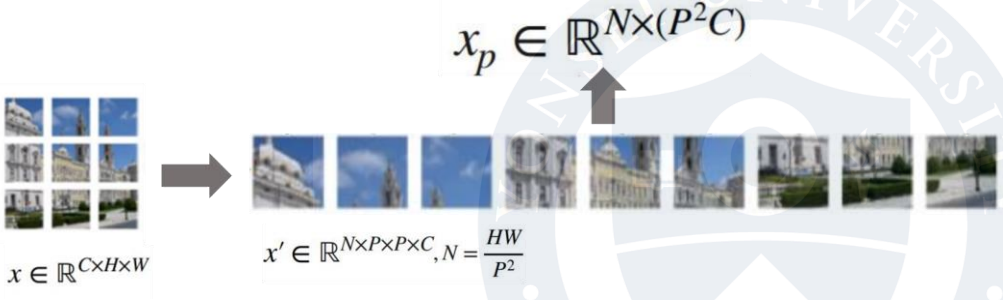
The Vision Transformer (ViT) is a novel deep learning architecture for image classification. Unlike traditional convolutional neural networks (CNNs), ViT adopts a transformer-based model originally designed for natural language processing.
ViT divides input images into fixed-size patches,captures global context through Transformer Encoder.
This approach eliminates the need for conv layers, allowing ViT to efficiently handle long-range dependencies in images.



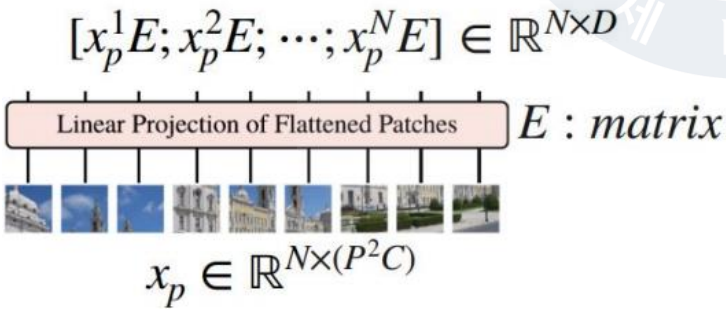
2 Vision Transformer

ViT Process

Step 1. Given an image (x), the image is divided into N patches of size PxP to construct a patch sequence set (x_p).



Step 2. Through linear projection, each patch of x_p is transformed into D dimensions.

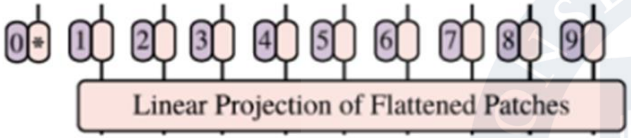


$x_p^i \in \mathbb{R}^{P^2C}, E \in \mathbb{R}^{(P^2C)\times D}$

2 Vision Transformer

ViT Process

Step 3. Add a class token(X_{cls}) to X_pE and perform positional embedding.



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

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

Class token: A trainable vector, when passed through the encoder, serves as the representation vector for the image.

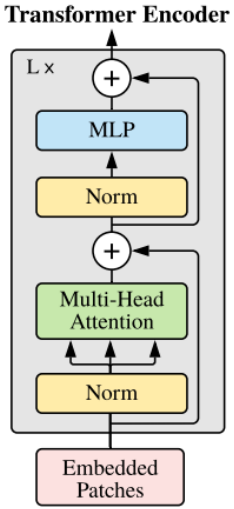
2 Vision Transformer

ViT Process

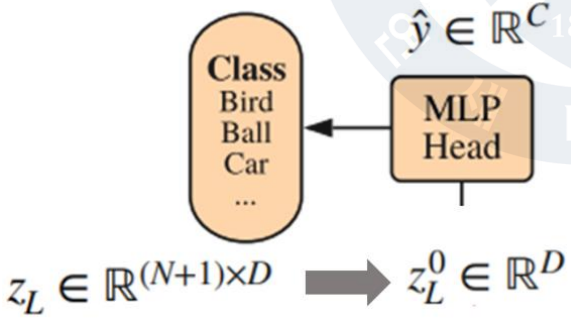
Step 4. Forward z_0 into the Transformer Encoder and perform Multi-Head Self Attention (MSA).

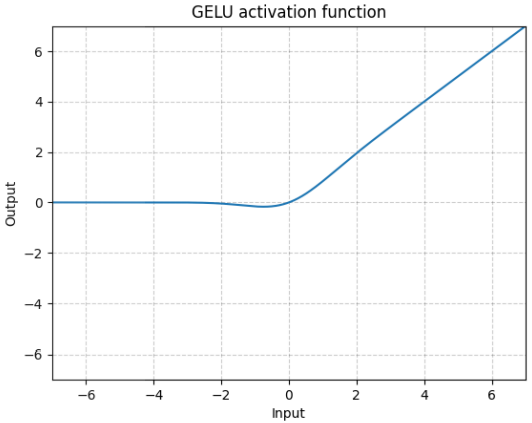
$$z'_l = MSA(LN(z_{l-1})) + z_{l-1}$$

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



Step 5. Classify the class of the image by forwarding the class token, which represents the image, into the MLP.

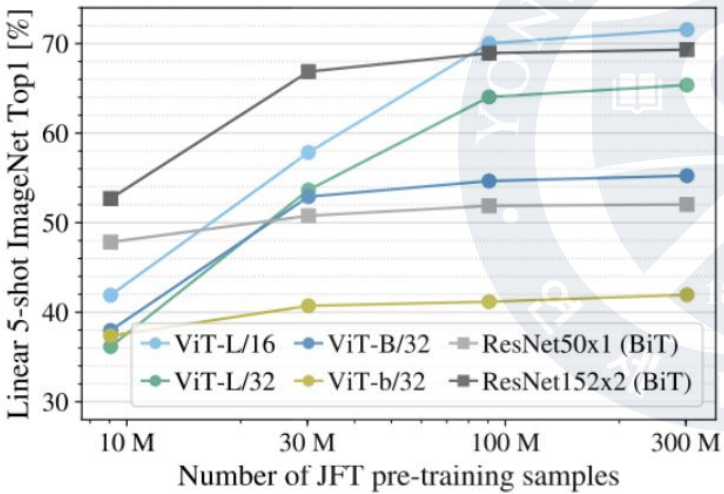




2 Vision Transformer

Experiments

Transformers lack some of the inductive biases inherent to CNNs, such as translation equivariance and locality, therefore do not generalize well when trianed on insufficient amounts of data.



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

ImageNet-21k or JFT (pre-training) -> ImageNet (fine-tuning)

We find that large scale training trumps inductive bias.

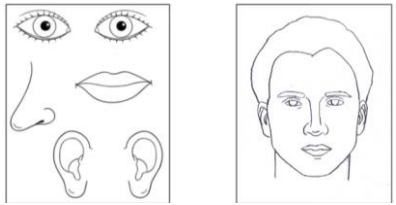
2 Vision Transformer

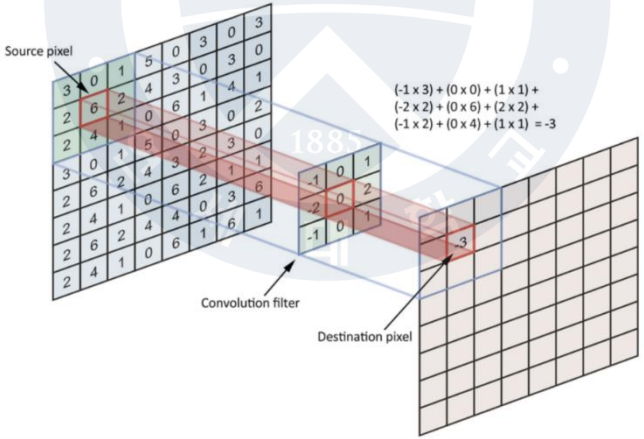
Inductive bias

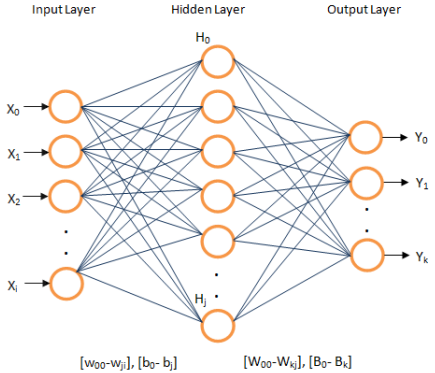
A set of assumptions used to predict outputs accurately for inputs that the model encounters for the first time.

- **Locality:** It is possible to extract features within a specific region of an image simply by checking at it.

- **Translation equivariance:** If the input position changes, the output also transforms to the corresponding position.







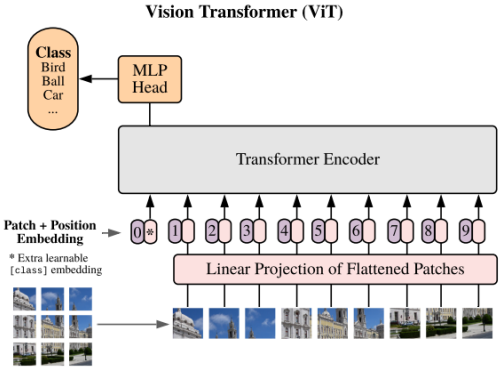
2 Vision Transformer

Dataset

CIFAR-10

- CIFAR-10 is a dataset that consists of 60,000 color images, each with a size of 32x32 pixels.
- These images are divided into 10 distinct classes, and each class contains 6,000 images.
- The dataset is balanced, meaning that there are an equal number of images for each class.
- The dataset is split into two subsets: a training set and a test set.
- The training set contains 50,000 images (5,000 images per class), while the test set consists of 10,000 images (1,000 images per class).

2 Vision Transformer



Patch Embedding

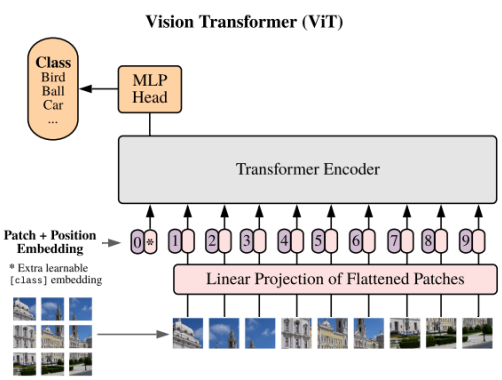
In [81]:

```python
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 8, emb_size: int = 128):
        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) # [b d hw]
        x = x.permute(0,2,1) # [b hw d]
        return x
```

2 Vision Transformer

Positional Embedding

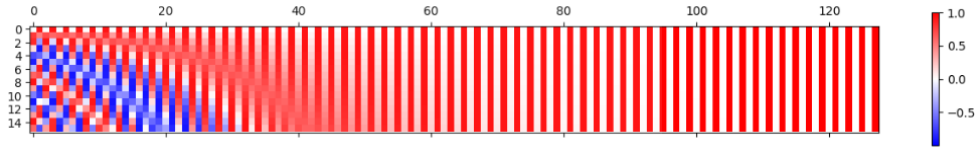
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$



In [83]:

```
pos_num = 16
d_model = 128

# all positions of tokens
posses = np.arange(pos_num) # [0, 1, ..., 14, 15] (size: pos_num = 16)

# exponent of 10000
i = np.arange(d_model)//2 # [0, 0, 1, 1, ...., 63, 63] (size: d_model = 128)
two_i = 2*i # [0, 0, 2, 2, ...., 126, 126]
exponent = two_i/d_model # [0, 0, 0.00x, 0.00x, ...., 0.9xx, 0.9x]

# fraction = positions / 10000^exponent
denominator = np.power(10000, exponent) # [1, 1, ..., 9xxx, 9xxx] (size: d_model = 128)
fraction = posses[:, np.newaxis] / denominator # shape: pos_num x d_model = 16 x 128

# even index of PE vector -> sin(fraction) = sin(pos/10000^exponent)
# odd index of PE vector -> cos(fraction) = cos(pos/10000^exponent)
pos_emb = np.zeros((pos_num, d_model))
pos_emb[:, 0::2] = np.sin(fraction[:, 0::2]) # shape: 16 x 64
pos_emb[:, 1::2] = np.cos(fraction[:, 1::2]) # shape: 16 x 64
print(pos_emb.shape)
```

(16, 128)

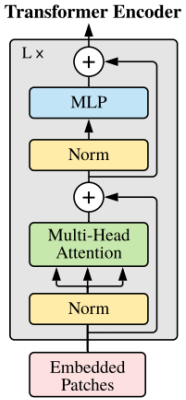


2 Vision Transformer

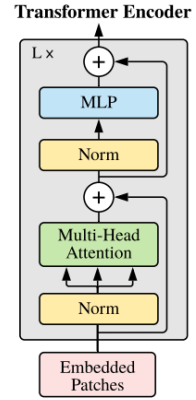
Multi-head Self Attention (MSA)

In [88]:

```python
class MultiheadAttention(nn.Module):
    def __init__(self, emb_size: int = 128, num_heads: int = 4, dropout: float = 0):
        super().__init__()
        self.num_heads = num_heads
        self.embed_dim = emb_size
        self.head_dim = int(emb_size / num_heads)
        self.query = nn.Linear(emb_size, emb_size)
        self.key = nn.Linear(emb_size, emb_size)
        self.value = nn.Linear(emb_size, emb_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        batch_size = x.size(0)
        q = self.query(x) # [b, n, d], n=hw+1
        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) # [b, h, n, d/h]
        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)

        scaling = self.head_dim ** (1/2)
        attention = torch.softmax(q @ k / scaling, dim=-1) # [b, h, n, n]
        x = self.dropout(attention) @ v # [b, h, n, d/h]
        x = x.permute(0,2,1,3).reshape(batch_size, -1, self.embed_dim) # [b, n, d]

        return x, attention
```



2 Vision Transformer



MLP

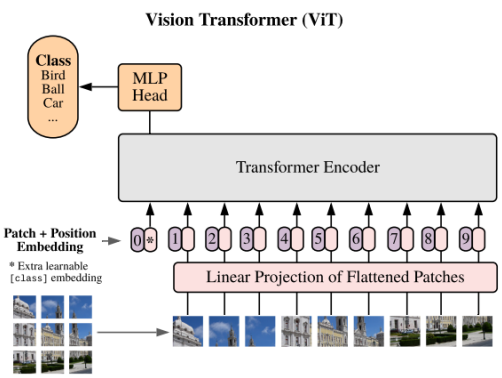
In [91]:
```python
class MLP(nn.Sequential):
    def __init__(self, emb_size: int, expansion: int = 4, drop_p: float = 0.):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Linear(emb_size, expansion * emb_size),
            nn.GELU(),
            nn.Dropout(drop_p),
            nn.Linear(expansion * emb_size, emb_size),
        )

    def forward(self, x):
        return self.mlp(x)
```

Residual Connection

In [89]:
```python
class ResidualAdd1(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

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

In [90]:
```python
class ResidualAdd2(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

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

2 Vision Transformer



Transformer Encoder Block

In [92]:
```
class TransformerEncoderBlock(nn.Sequential):
    def __init__(self,
                 emb_size: int = 128,
                 drop_p = 0.,
                 forward_expansion: int = 4,
                 forward_drop_p: float = 0.,
                 ** kwargs):
        super().__init__()

        self.work1 = ResidualAdd1(nn.Sequential(
                                nn.LayerNorm(emb_size),
                                MultiheadAttention(emb_size, **kwargs)))
        self.work2 = ResidualAdd2(nn.Sequential(
                                nn.LayerNorm(emb_size),
                                MLP(emb_size, expansion=forward_expansion, drop_p=forward_drop_p),
                                nn.Dropout(drop_p)))

    def forward(self, x):
        x, attention = self.work1(x)
        x = self.work2(x)
        return x, attention
```

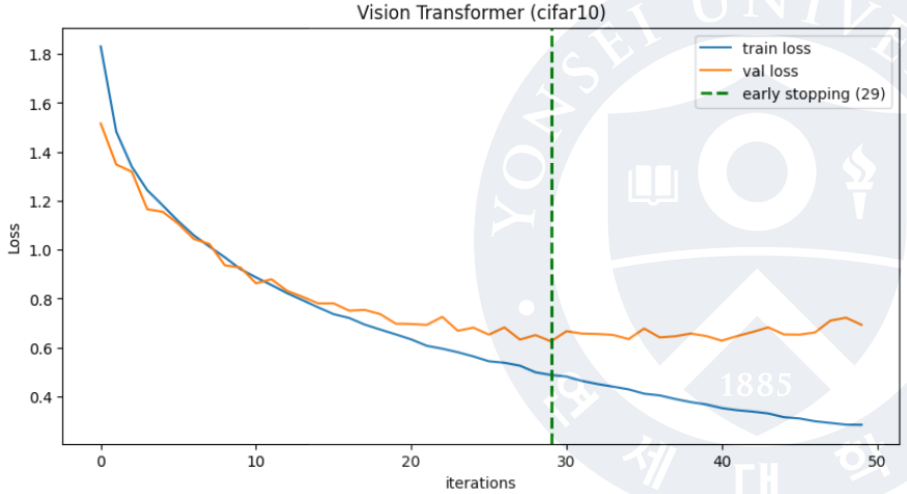
Vision Transformer

In [94]:
```
class VisionTransformer(nn.Sequential):
    def __init__(self, patch_size: int = 8, emb_size: int = 128, num_layers: int = 12, num_heads: int = 4, n_classes: int = 10, cpu=Fal
        super().__init__()
        self.patchembedding = PatchEmbedding(patch_size=patch_size, emb_size=emb_size, cpu=cpu)
        self.transformer = nn.ModuleList([TransformerEncoderBlock(emb_size=emb_size, num_heads=num_heads) for _ in range(num_layers)])
        self.mlp_head = nn.Sequential(nn.LayerNorm(emb_size),
                                      nn.Linear(emb_size, n_classes))

    def forward(self, x):
        x = self.patchembedding(x)
        for layer in self.transformer:
            x, attention = layer(x)
        x = self.mlp_head(x[:,0])
        return x, attention
```

2 Vision Transformer

Training Setting and Loss

- Loss graph



· To extract the best performance among 50 epochs, `early stopping` was employed.
· In this experiment, the validation set and the test set are the same.
· The best model can be confirmed through `weight/cifar_vit_pe_conv.pth`.

- **Model**
  · patch size: 4x4
  · embedding size: 192
  · num layers: 12
  · num classes: 10
  · num heads: 12
- **Training**
  · batch size: 256
  · num epoch: 50
  · optimizer: Adam (lr=0.001, weight_decay=5e-5)
  · data augmentation: RandomCrop(32, padding=4), RandomHorizontalFlip()

$$CE = -\sum_{i}^{C} t_i log(f(s)_i)$$

2 Vision Transformer

Accuracy

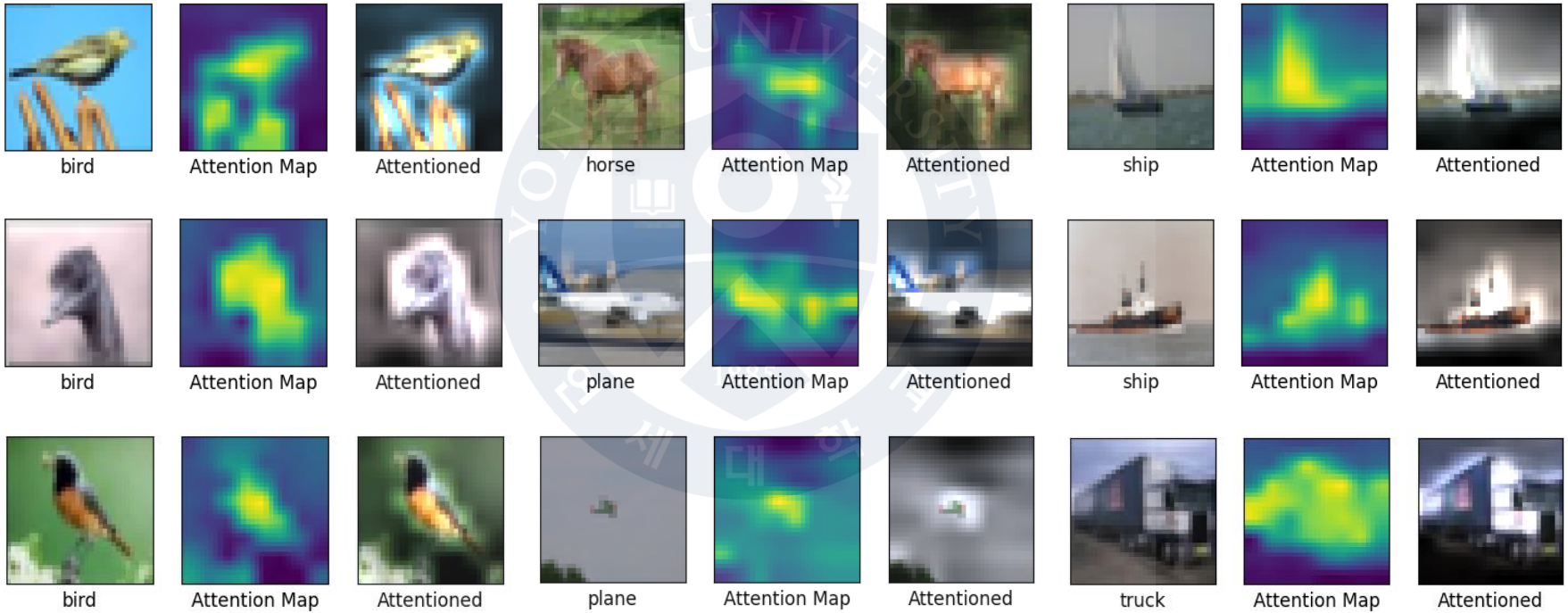
ViT Basic	ViT PE	ViT PE & Conv
75.03 %	76.05 %	**78.55** %

- `ViT Basic` : positional encoding with Training, patch embedding with FC layer.
- `ViT PE` : positional encoding without Training, patch embedding with FC layer.
- `ViT PE & Conv` : position encoding without Training, patch embedding with Conv2d.

1. Creating Positional Encoding vectors can make the training of the model more challenging.
2. Inductive bias is crucial in small datasets.

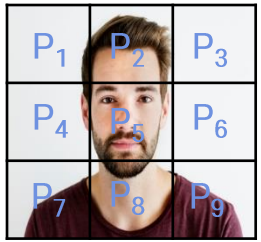
2 Vision Transformer

Visualization



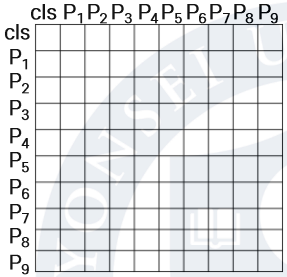
2 Vision Transformer

How to Visualize?



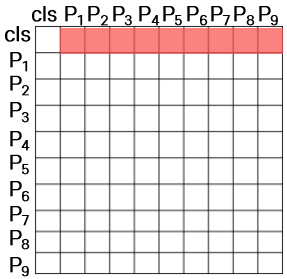
Image(30x30) to 9 patches

ViT

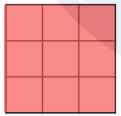


Attention Matrix (10x10)

Class token: A trainable vector, when passed through the encoder, serves as the representation vector for the image.

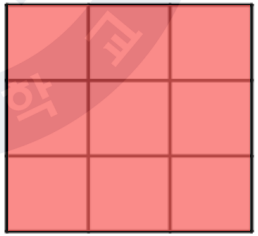


Normalize & Unflatten



3x3

Resizing



30x30

indicating where the model is focusing on in the image through the cls token portion of the attention map.

2 Vision Transformer

Appendix

Patch Embedding *[Positional Encoding with Training]*

In [28]:

```python
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, img_size: int = 32, patch_size: int = 8, emb_size: int = 128):
        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)
        )

        # cls token
        self.cls_token = nn.Parameter(torch.randn(1, 1, emb_size)) # [1, 1, d]
        # pos encoding
        self.positions = nn.Parameter(torch.randn((img_size // patch_size) **2 + 1, emb_size)) # [hw+1, d]

    def forward(self, x: Tensor) -> Tensor:
        b, _, _, _ = x.shape
        x = self.projection(x) # [b, hw, d]
        cls_tokens = repeat(self.cls_token, '1 1 d -> b 1 d', b=b) # [b, 1, d]

        x = torch.concat([cls_tokens, x], dim=1) # [b, hw+1, d]
        x += self.positions
        return x
```

2 Vision Transformer

Appendix

Patch Embedding *[Positional Encoding with Untraining]*

In [86]:

```python
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, img_size: int = 32, patch_size: int = 8, emb_size: int = 128, cpu=False):
        super().__init__()
        self.patch_size = patch_size

        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)
        )

        # cls token
        self.cls_token = nn.Parameter(torch.randn(1, 1, emb_size)) # [1, 1, d]
        # pos encoding
        hw_plus_one = (img_size // patch_size) **2 + 1
        self.positions = PE(pos_num=hw_plus_one, d_model=emb_size, cpu=cpu) # [hw+1, d]

    def forward(self, x: Tensor) -> Tensor:
        b, _, _, _ = x.shape
        x = self.projection(x) # [b, hw, d]
        cls_tokens = repeat(self.cls_token, '1 1 d -> b 1 d', b=b) # [b, 1, d]

        x = torch.concat([cls_tokens, x], dim=1) # [b, hw+1, d]
        x += self.positions
        return x
```

In [85]:

```python
def PE(pos_num, d_model, cpu=False):
    posses = np.arange(pos_num)

    i = np.arange(d_model)//2
    exponent = 2*i/d_model
    pos_emb = posses[:, np.newaxis] / np.power(10000, exponent)

    pos_emb[:, 0::2] = np.sin(pos_emb[:, 0::2])
    pos_emb[:, 1::2] = np.cos(pos_emb[:, 1::2])

    if cpu:
        pos_emb = torch.from_numpy(pos_emb)
    else:
        pos_emb = torch.from_numpy(pos_emb).to("cuda")

    return pos_emb
```

2 Vision Transformer

Appendix

Patch Embedding *[Patch Embedding with Convolution]*

In [170…

```python
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, img_size: int = 32, patch_size: int = 8, emb_size: int = 128, cpu=False):
        super().__init__()
        self.patch_size = patch_size

        self.projection = nn.Sequential(
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size, stride=patch_size),
            Rearrange('b d h w -> b d (h w)')
        )

        # cls token
        self.cls_token = nn.Parameter(torch.randn(1, 1, emb_size)) # [1, 1, d]
        # pos encoding
        hw_plus_one = (img_size // patch_size) **2 + 1
        self.positions = PE(pos_num=hw_plus_one, d_model=emb_size, cpu=cpu) # [hw+1, d]

    def forward(self, x: Tensor) -> Tensor:
        b, _, _, _ = x.shape
        x = self.projection(x) # [b d hw]
        x = x.permute(0,2,1) # [b hw d]
        cls_tokens = repeat(self.cls_token, '1 1 d -> b 1 d', b=b) # [b, 1, d]

        x = torch.concat([cls_tokens, x], dim=1) # [b, hw+1, d]
        x += self.positions
        return x
```

2 Vision Transformer

Appendix

Visualization

```
1. Create a single row vector by averaging 'num_head' row vectors. (using average attention weights)
2. Convert the row vector (1x64) into a 2D matrix (8x8).
3. Resize the matrix to the original image size (32x32).
4. Perform min-max normalization on the matrix. (scaling values to range between 0 and 1)
5. Multiply the matrix with the original image. (incorporating the focused area into the original image)
6. Multiply the resulting matrix by 2 and visualize it. (amplifying the representation of the focused area)
```

```python
# 이미지 시각화
num = 0
img = images[num].to(device)
mean=(0.4914, 0.4822, 0.4465)
std=(0.2023, 0.1994, 0.2010)
show_img = img.clone()
for i in range(3):
    show_img[i] = show_img[i] * std[i] + mean[i]
show_img = show_img.cpu().permute(1,2,0)

fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(1, 5, 1)
ax.axes.get_xaxis().set_visible(False)
ax.axes.get_yaxis().set_visible(False)
ax.imshow(show_img)
ax.text(0.5, -0.15, f'{classes[predicted[num]]}', transform=ax.transAxes, fontsize=12, ha='center')
```

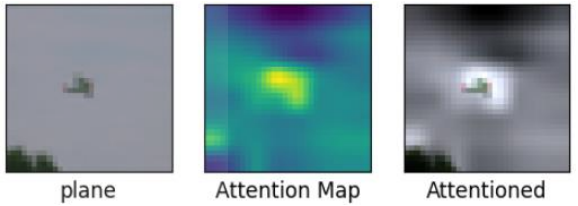
```python
# 어텐션 맵 시각화
output, attention = net(img.unsqueeze(0)) # [1, h, n, n]

sum_heatmap = np.zeros(shape=(32,32), dtype=np.float32)
for i in range(num_heads):
    attn_heatmap = attention[0, i, 0, 1:].reshape((8, 8)).detach().cpu().numpy() # cls_token row of attn map
    attn_heatmap = cv2.resize(attn_heatmap,(32,32))
    sum_heatmap += attn_heatmap
avg_heatmap = sum_heatmap / num_heads
avg_heatmap = (avg_heatmap - avg_heatmap.min()) / (avg_heatmap.max() - avg_heatmap.min())

ax = fig.add_subplot(1, 5, 2)
ax.axes.get_xaxis().set_visible(False)
ax.axes.get_yaxis().set_visible(False)
ax.imshow(avg_heatmap)
ax.text(0.5, -0.15, 'Attention Map', transform=ax.transAxes, fontsize=12, ha='center')
```

(1,2,3)

(4)

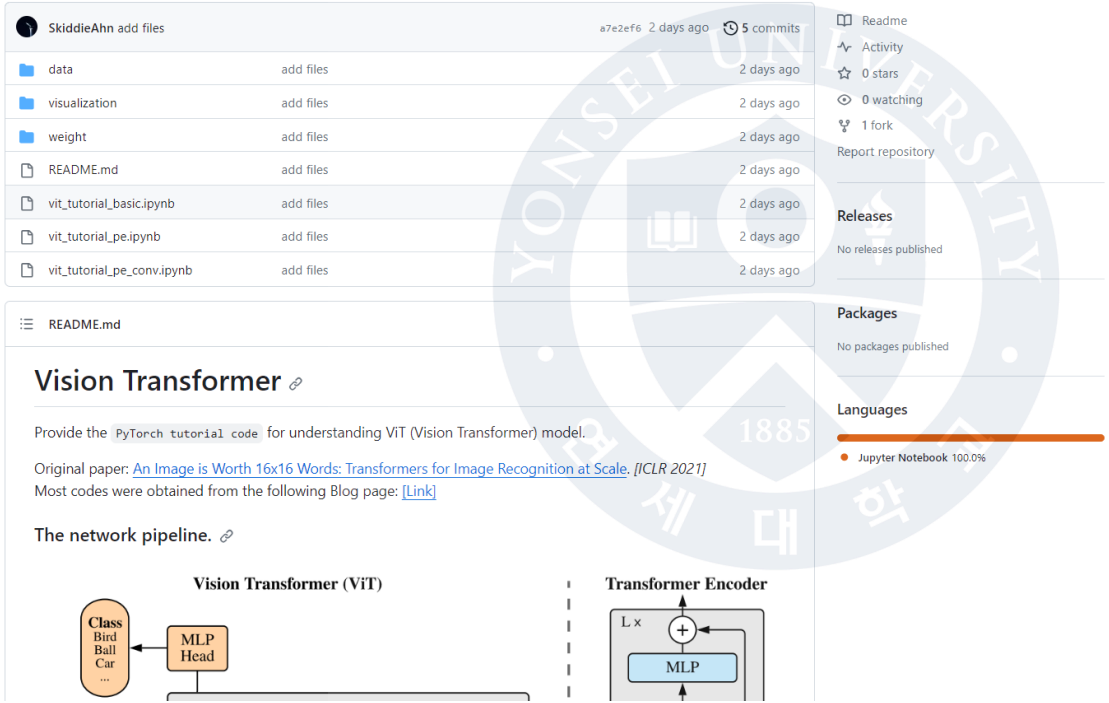
```python
# 어텐션된 이미지 시각화
attn_img = show_img.clone().detach()
attn_img[:,:,0] = (attn_img[:,:,0]*avg_heatmap)*2
attn_img[:,:,1] = (attn_img[:,:,1]*avg_heatmap)*2
attn_img[:,:,2] = (attn_img[:,:,2]*avg_heatmap)*2

ax = fig.add_subplot(1, 5, 3)
ax.axes.get_xaxis().set_visible(False)
ax.axes.get_yaxis().set_visible(False)
ax.imshow(attn_img)
ax.text(0.5, -0.15, 'Attentioned', transform=ax.transAxes, fontsize=12, ha='center')
```

(5,6)



2 Vision Transformer

Github Code



https://github.com/ad-yonsei/Code-Vision-Transformer

2 Vision Transformer

References

- Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." arXiv preprint arXiv:2010.11929 (2020).
- Implementing Vision Transformer (ViT) in PyTorch (https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632)
- Inductive Bias (https://blog.kubwa.co.kr/inductive-bias-4af72375b2d1)

Thank You

Vision Transformer - coding practice

Sunghyun Ahn
skd@yonsei.ac.kr

<2023/11/16>