ViT

(2020)An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

김승철

Contents

01 Multi Head Attention

02 ViT Concept

03 Results of training with CIFAR10

01 Multi Head Self Attention

컴퓨터는 숫자로 모든 것을 판단하기 때문에, Word Embedding을 통해
문장을 숫자로 바꿔주는 작업이 필요하다.

Word Embedding? 문장 내의 N개의 단어들을 각각 D dimension의 벡터들로 표현하는 것.
그러면 Input 문장은 $\mathbb{R}^{N\times D}$의 matrix가 되고, 이 matrix를 Transformer의 입력으로 넣는다.

하지만 기존의 seq2seq, RNN, GRU와 같은 구조들과 다르게, Self Attention 계산은
Input을 한 번에 dot product로 처리하기 때문에, 위치정보를 보존하기가 힘들다.

이에 위에서 Word Embedding을 거친 문장에 인위적으로 위치 정보를 넣어준다.

이 논문(ViT)에서는 fixed positional encoding을 사용한다.

01 Multi Head Self Attention

Positional Encoding의 수식은 아래와 같다.

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

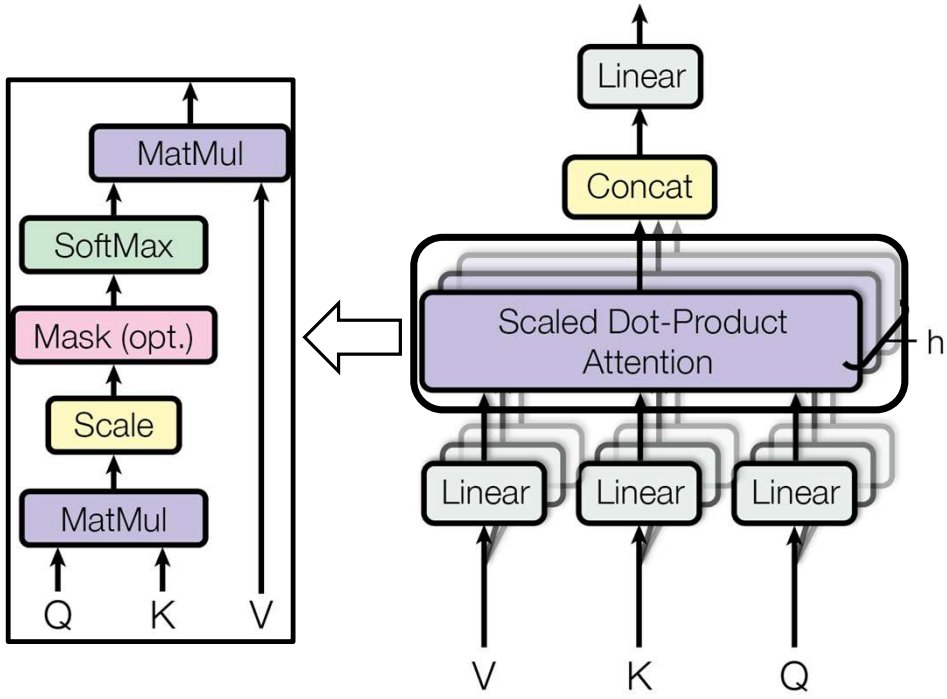
```
def positional_encoding(n, d):
    pe = torch.rand((n, d))
    for i in range(n):
        for j in range(d):
            if j % 2 == 0:
                pe[i][j] = np.sin(i/(10000 ** (j / d)))
            else:
                pe[i][j] = np.cos(i/(10000 ** ((j-1) / d)))


    return pe
```

01 Multi Head Self Attention

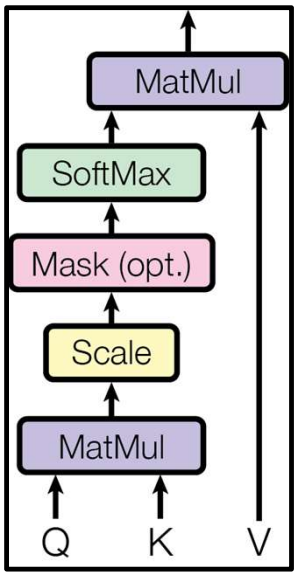
Multi Head Self Attention 를 구성하고 있는
Scaled Dot-Product Attention(Self Attention) 을 보자.

Q. 이름이 왜 "Self" Attention 인가?
A. 자기 자신과의 연산들을 통해, 문장 내의 어떤 단어에
주의를 기울여야 하는지를 계산하기 때문.

(Query, Key, Value라 붙인 이유는 모르겠음)



01 Multi Head Self Attention

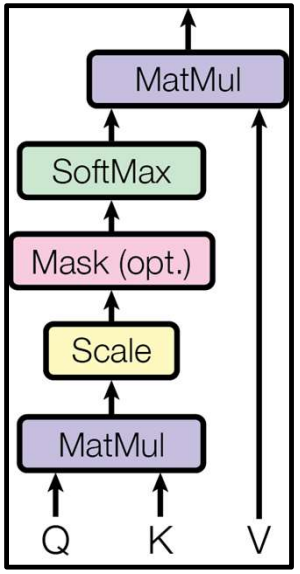


Input으로 $\mathbb{R}^{N\times D}$의 matrix를 받고, W^Q, W^K, $W^V \in \mathbb{R}^{D\times D}$ 라는 weight matrix를 통해서 Input matrix를 각각 Query, Key, Value로 만든다.~~(왜 query, key, value인지는 모름)~~

오른쪽 그림과 같이, Query와 Key를 Matrix Multiplication 해주고, Scale 해준다음, softmax를 거친다. 이 과정의 결과를 Attention score라 한다.

Attention Score와 Value를 마찬가지로 Matrix Multiplication해주면 Self Attention 계산은 끝이다.

01 Multi Head Self Attention



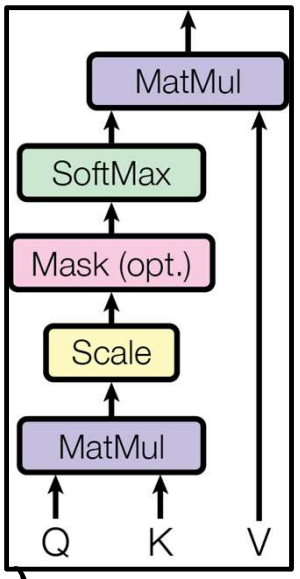
수학적으로 계산하면 아래와 같다.

Input $S \in \mathbb{R}^{N \times D}$, Transformer의 hidden layer의 dim을 D' 이라고 하면,

$S \cdot W^Q = Q \in \mathbb{R}^{N \times D'}$, $S \cdot W^K = K \in \mathbb{R}^{N \times D'}$, $S \cdot W^V = V \in \mathbb{R}^{N \times D'}$ $(W^Q, W^K, W^V \in \mathbb{R}^{D \times D'})$

$$SA(Q,K,V) = AttnScore(Q,K) \cdot V = \frac{Q \cdot K^T}{\sqrt{D'}} \cdot V \in \mathbb{R}^{N \times D'}$$

01 Multi Head Self Attention



어떻게 Attention 연산이 similarity를 표현하는가?

내적(Dot Product) 연산 자체가 similarity를 표현함.(Cosine Similarity)

$a \cdot b = |a||b| \cos\theta$ 이고, $\cos\theta = \frac{a \cdot b}{|a||b|}$ 이다. (여기서 $|a||b|$(euclidean norm)는 Scaling factor)

좀 직관적으로 보면,

$D = (2,2), D = (2,2)$ 의 dot product는 8.

$D = (2,2), C = (-2,-2)$의 dot product는 -8.

$D = (2,2), E = (2,-2)$의 dot product 0.

01 Multi Head Self Attention

어떻게 Attention 연산이 similarity를 표현하는가?

내적(Dot Product) 연산 자체가 similarity를 표현함

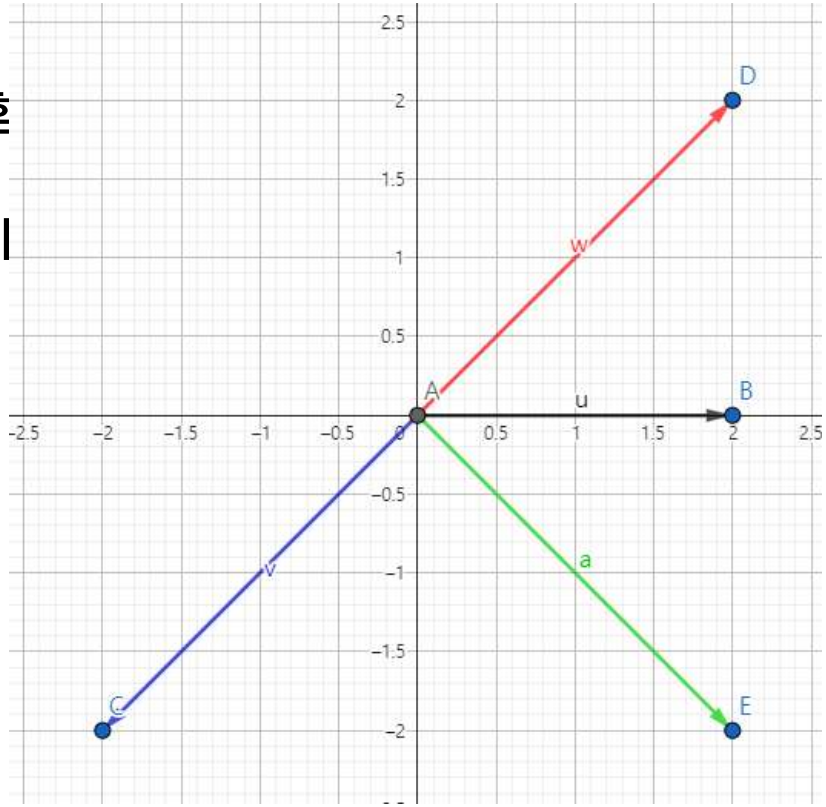
$a \cdot b = |a||b| \cos\theta$ 이고, $\cos\theta = \frac{a \cdot b}{|a||b|}$ 이다. (여기

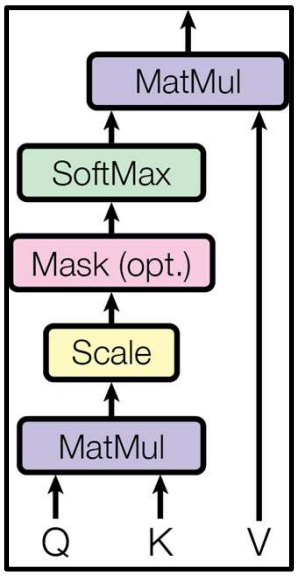
좀 직관적으로 보면,

$D = (2,2), D = (2,2)$ 의 dot product는 8.

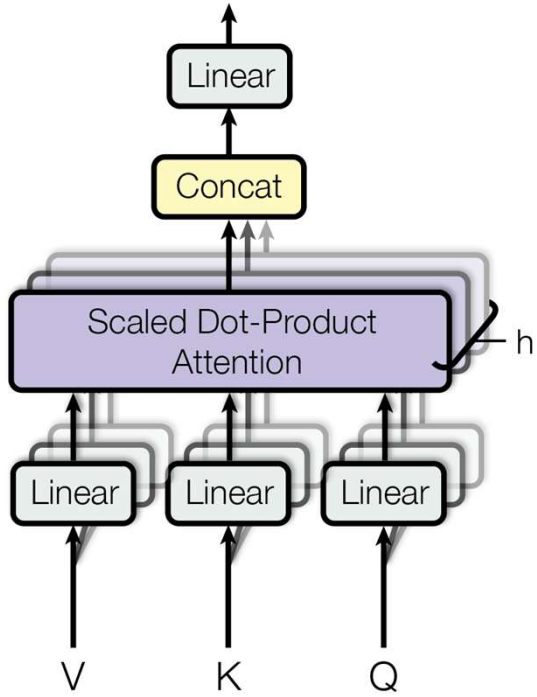
$D = (2,2), C = (-2,-2)$의 dot product는 -8.

$D = (2,2), E = (2,-2)$의 dot product 0.





01 Multi Head Self Attention



Multi Head Self Attention은 아까 Input을 weight $W^{Q,K,V}$ 를 이용해서 projection 시켜 줄 때, weight $W^{Q,K,V} \in \mathbb{R}^{D \times D'}$의 D'을 Head의 개수로 나눠주고, 나중에 $D''(D''/h)$를 기준으로 Concat 하면 된다.

Transformer의 hidden dim을 D'이라 하고, Head의 개수를 h라 하면,

$$MHSA(Q,K,V) = Concat(head_1, \dots, head_h)W^o, \ (W^o \in \mathbb{R}^{h \cdot D'' \times D})$$

$$where head_i = Att\left(Q = S \cdot W_i^Q, K = S \cdot W_i^K, V = S \cdot W_i^V\right) \ \& \ (W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{D \times D''}) \ \& \ D'' = D'/h$$

$$head_i = Att(Q,K,V) = AttScore(Q,K) \cdot V = \frac{Q \cdot K^T}{\sqrt{D'}} \cdot V \in \mathbb{R}^{N \times D''}$$

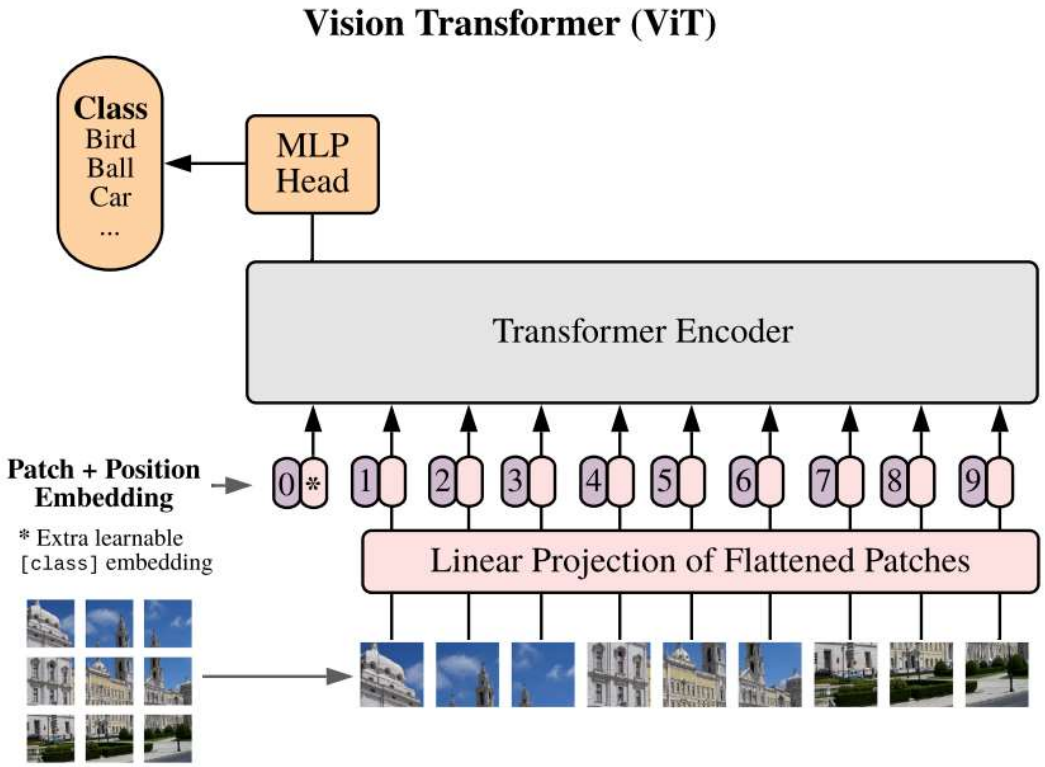
$$\therefore MHSA(Q,K,V) \in \mathbb{R}^{N \times D}$$

02 ViT Concept

ViT의 전체 구조는 오른쪽 그림과 같고,
수식으로는 그림 아래와 같다.

아래 순서대로 ViT를 설명 할 것이다.

① Split Patches
② Linear Projection of Flattened Patches
③ Patch + Positional Embedding
with class embedding
④ Transformer Encoder
⑤ MLP Head for Classification



$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}, \qquad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D},\ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$$

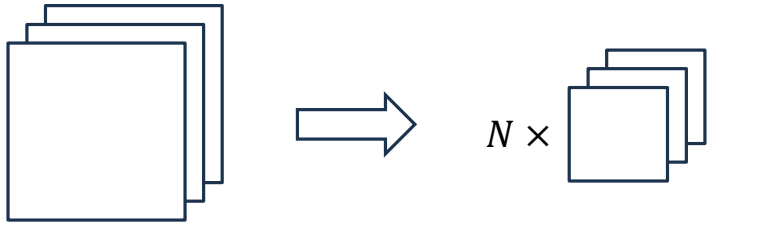
$$\mathbf{z}'_\ell = \text{MSA}(\text{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, \qquad \ell = 1 \ldots L$$

$$\mathbf{z}_\ell = \text{MLP}(\text{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, \qquad \ell = 1 \ldots L$$

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

02 ViT Concept - ① Split Patches

Input Image $\mathbb{R}^{H\times W\times C}$ 를 N개의 patch P ($\mathbb{R}^{p\times p\times c}$) 로 분할한다.
분할 후 하나의 Image에 대한 shape은 $\mathbb{R}^{N\times p\times p\times c}$ 가 된다.



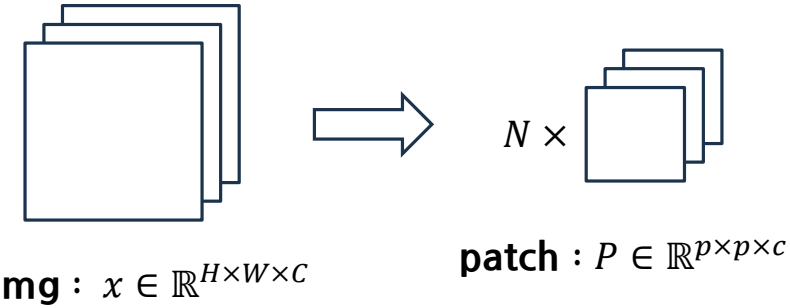
img : $x \in \mathbb{R}^{H\times W\times C}$

patch : $P \in \mathbb{R}^{p\times p\times c}$

```python
class Img2Patch(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.p      = args.patch_size
        self.unfold = nn.Unfold(kernel_size = self.p,
                                stride      = self.p)


    def forward(self, x):
        # x.shape = (b, c, h, w)
        patch       = self.unfold(x) # (b, c*p*p, h//p * w//p)
        bs, c, _, _ = x.shape

        patches = patch.view(bs, c, self.p, self.p, -1).permute(0, 4, 1, 2, 3)

        return patches # patches.shape = (b, n, p, p, c)
```

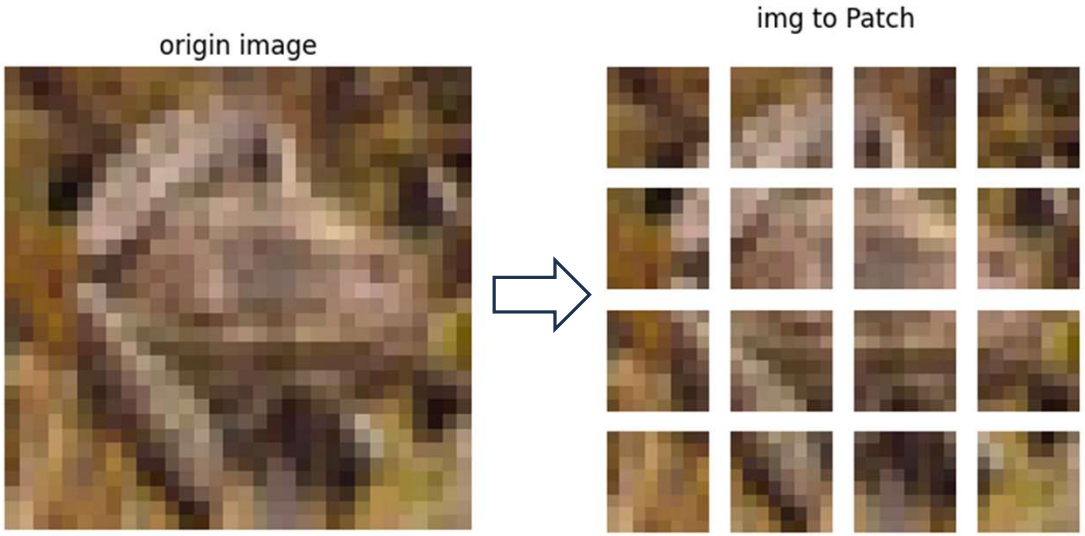
02 ViT Concept - ① Split Patches

Input Image $\mathbb{R}^{H\times W\times C}$ 를 N개의 patch P ($\mathbb{R}^{p\times p\times c}$) 로 분할한다.
분할 후 하나의 Image에 대한 shape은 $\mathbb{R}^{N\times p\times p\times c}$ 가 된다.



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

patch : $P \in \mathbb{R}^{p\times p\times c}$



02 ViT Concept - ② Linear Projection of Flattened Patches

Multi Head Attention은 Natural Language와 같은
Sequence Data에서 작동되도록 설계하였기 때문에
원본 Image 그대로 Multi Head Attention을
적용하는 것은 쉽지 않다.

그래서 Image를 Patch로 분할하고
$N \times (p \times p \times c)$ shape으로 Flatten 한 후
Dimension D로 projection 한다.

Projection 후 shape은 다음과 같다.
$\Longrightarrow proj \in \mathbb{R}^{N \times D}$

```python
class Projection_layer(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.d          = args.hidden_dim
        self.p          = args.patch_size
        self.proj_layer = nn.Linear(self.p * self.p * 3, self.d)

    def forward(self, x):
        b, n, c, p, p = x.shape
        x             = x.reshape(b, n, c*p*p)
        proj          = self.proj_layer(x)

        return proj # proj.shape = (b, n, h_d)
```

02 ViT Concept - ③ Patch + Positional Embedding with Class Embedding

이제 projection 된 patch $\in \mathbb{R}^{N \times D}$ 에
Class Embedding 을 concat해준다.
Class Embedding은 Learnable 해야하므로(논문),
torch.nn.Parameter로 Class Embedding을 만들고
기존의 Patch와 Class Embedding을 concat 한다.
Concat한 Tensor에 Positional Embedding을 더한다.

※ torch.stack(dim=d) 과 torch.cat(dim=d)의 차이점?
torch.stack은 dim=d 에 새로운 Dimension을 추가하여 Tensor들을 결합
torch.cat은 기존 차원 dim=d 을 확장하여 Tensor들을 결합

02 ViT Concept - ③ Patch + Positional Embedding with Class Embedding

이제 projection 된 patch $\in \mathbb{R}^{N \times D}$ 에

Class En

Class En

torch.nn

기존의 Pa

Concat한

```
def positional_encoding(n, d):
    pe = torch.rand((n, d))
    for i in range(n):
        for j in range(d):
            if j % 2 == 0:
                pe[i][j] = np.sin(i/(10000 ** (j / d)))
            else:
                pe[i][j] = np.cos(i/(10000 ** ((j-1) / d)))


    return pe
```

※ torch.

torch.

torch.cat은 기존 차원 dim=d 을 확장하여 Tensor들을 결합

02

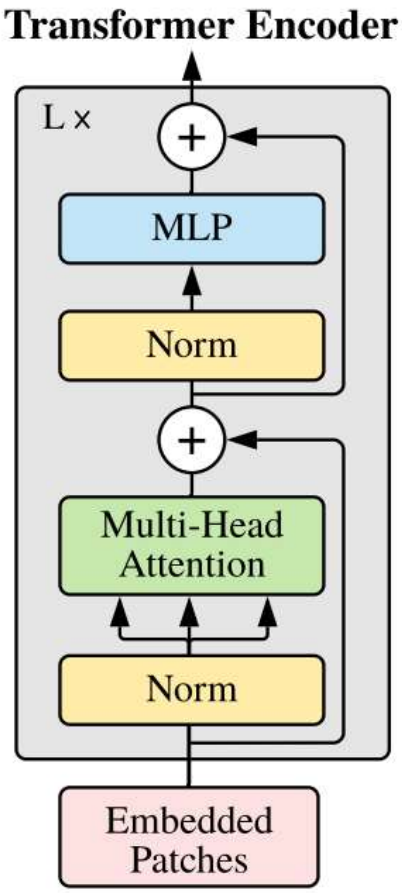
ViT Concept - ③ Patch + Positional Embedding with Class Embedding

```python
class ClassEmbedding(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.d          = args.hidden_dim
        self.device     = args.device
        self.cls_tensor = torch.rand(1, 1, self.d)
        self.cls_emb    = nn.Parameter(self.cls_tensor)

    def forward(self, x):
        # x.shape = (b, n, d)
        # breakpoint()
        b, n, d = x.shape
        cls_emb = self.cls_emb.expand(b, -1, -1)
        concat  = torch.cat([cls_emb, x], dim=1)
        pe      = positional_encoding(n+1, d).unsqueeze(0).expand(b, n+1, d).to(self.device)

        return concat + pe # shape = (b, n+1, d)
```

02 ViT Concept - ④ Transformer Encoder

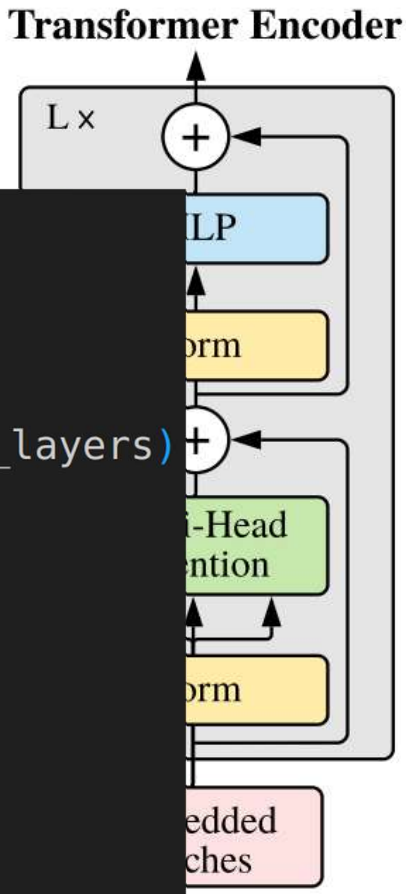
Embedded Patch를 Transformer Encoder Block에 입력.
오른쪽 그림의 화살표대로 Train이 이루어진다.

Encoder 내부에 Encoder Layer가 여러 개 있으므로(Hyperparameter),
torch.nn.ModuleList를 이용하여 Encoder Layer를 List로 쌓은 다음,
for문을 이용하여 model을 만든다.



02 ViT Concept - ④ Transformer Encoder

```python
class TransformerEncoder(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.encoder = nn.ModuleList([
            TransformerEncoderLayer(args) for _ in range(args.num_enc_layers)
            ])

        self.ln = nn.LayerNorm(args.hidden_dim)

    def forward(self, x):
        out = x

        for layer in self.encoder:
            out = layer(out)

        output = self.ln(out)
        return output # output.shape = (b, n+1, d)
```

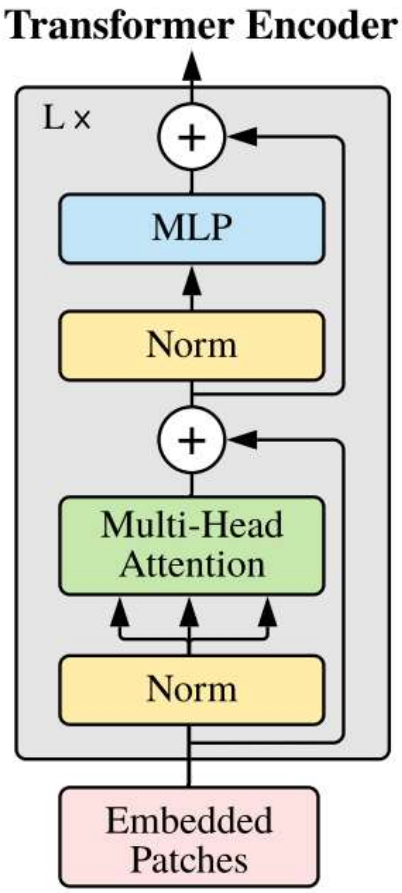


02 ViT Co

Embedded Patch를
오른쪽 그림의 화살표

Encoder 내부에 Enc
torch.nn.ModuleLis
for문을 이용하여 mo

```python
class TransformerEncoderLayer(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.mha  = nn.MultiheadAttention(embed_dim   = args.hidden_dim,
                                          num_heads   = args.num_heads,
                                          dropout     = args.dropout_rate,
                                          batch_first = True)

        self.ln1  = nn.LayerNorm(args.hidden_dim)

        self.ln2  = nn.LayerNorm(args.hidden_dim)

        self.ffn1 = nn.Linear(in_features  = args.hidden_dim,
                              out_features = args.mlp_size)

        self.ffn2 = nn.Linear(in_features  = args.mlp_size,
                              out_features = args.hidden_dim)

        self.act  = nn.GELU()

    def forward(self, x):
        inp    = x # shape = (b, n+1, d)

        ln     = self.ln1(inp)
        mha, _ = self.mha(ln, ln, ln)
        x      = mha + inp

        ln     = self.ln2(x)
        ffn1   = self.ffn1(ln)  # shape = (b, n+1, mlp_size)
        act    = self.act(ffn1)
        ffn2   = self.ffn2(act) # shape = (b, n+1, d)

        out    = x + ffn2

        return out # shape = (b, n+1, d)
```



02

ViT Concept - ⑤ MLP Head for Classification

마지막

MLP H

Tenso

Classi

out = x

out을

$out \in$

마지막

```python
class FeedForwardNet(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.linear  = nn.Linear(in_features  = args.hidden_dim,
                                 out_features = args.num_classes)
        self.act     = nn.GELU()
        self.dropout = nn.Dropout(args.dropout_rate)
        self.ln      = nn.LayerNorm(args.num_classes)

    def forward(self, x):
        inp = x # shape = (b, n+1, d)

        x   = self.linear(inp)
        x   = self.act(x)
        x   = self.dropout(x)
        x   = x[:, 0, :] # shape = (b, num_classes)
        out = self.ln(x)

        return out # shape = (b, num_classes)
```

02 ViT Concept - ⑤ MLP Head for Classification

전체적인 모델 코드는 다음과 같다.

02 ViT

전체적인 모델 코

```python
class ViTModel(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.patch               = Img2Patch(args)

        self.projection          = Projection_layer(args)

        self.class_embedding     = ClassEmbedding(args)

        self.transformer_encoder = TransformerEncoder(args)

        self.feedforwardnet      = FeedForwardNet(args)

    def forward(self, x):
        # x.shape = (b, 3, 32, 32)
        patches = self.patch(x)
        # patches.shape = (b, n, c, p, p)
        proj    = self.projection(patches)
        # proj.shape = (b, n, hidden_dim)
        cls_emb = self.class_embedding(proj)
        # cls_emb.shape = (b, n+1, hidden_dim)
        te      = self.transformer_encoder(cls_emb)
        # te.shape = (n, n+1, hidden_dim)
        ffn     = self.feedforwardnet(te)
        return ffn # ffn.shape = (b, num_classes)
```

02 ViT Concept - ⑤ MLP Head for Classification

모델 summary를 해봤을 때, 다음과 같다.

cation

```
==========================================================================================================================
Layer (type:depth-idx)                                  Input Shape               Output Shape              Param #
==========================================================================================================================
├─Img2Patch: 1-1                                       [-1, 3, 32, 32]           [-1, 16, 3, 8, 8]         --
|    └─Unfold: 2-1                                     [-1, 3, 32, 32]           [-1, 192, 16]             --
├─Projection_layer: 1-2                                [-1, 16, 3, 8, 8]         [-1, 16, 1024]            --
|    └─Linear: 2-2                                     [-1, 16, 192]             [-1, 16, 1024]            197,632
├─ClassEmbedding: 1-3                                  [-1, 16, 1024]            [-1, 17, 1024]            1,024
├─TransformerEncoder: 1-4                              [-1, 17, 1024]            [-1, 17, 1024]            --
|    └─ModuleList: 2                                   []                        []                        --
|    |    └─TransformerEncoderLayer: 3-1               [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-2               [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-3               [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-4               [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-5               [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-6               [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-7               [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-8               [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-9               [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-10              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-11              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-12              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-13              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-14              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-15              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-16              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-17              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-18              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-19              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-20              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-21              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-22              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-23              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    |    └─TransformerEncoderLayer: 3-24              [-1, 17, 1024]            [-1, 17, 1024]            12,596,224
|    └─LayerNorm: 2-3                                  [-1, 17, 1024]            [-1, 17, 1024]            2,048
├─FeedForwardNet: 1-5                                  [-1, 17, 1024]            [-1, 10]                  --
|    └─Linear: 2-4                                     [-1, 17, 1024]            [-1, 17, 10]              10,250
|    └─GELU: 2-5                                       [-1, 17, 10]              [-1, 17, 10]              --
|    └─Dropout: 2-6                                    [-1, 17, 10]              [-1, 17, 10]              --
|    └─LayerNorm: 2-7                                  [-1, 10]                  [-1, 10]                  20
--------------------------------------------------------------------------------------------------------------------------
```

```
Total params: 302,520,350
Trainable params: 302,520,350
Non-trainable params: 0
Total mult-adds (M): 906.53
==========================================
Input size (MB): 0.01
Forward/backward pass size (MB): 22.70
Params size (MB): 1154.02
Estimated Total Size (MB): 1176.74
==========================================
```

02 ViT

하지만 ViT의 단점도 존재한다.

첫 번째로,
Original Image를 Patch로 분할하고 독립적으로 처리하기 때문에, 인접해 있는 Patch간의 local information을 학습하지 못하는 문제가 있다.
물론 CNN 과 비교했을 때, ViT가 Global information을 잘 처리하지만, CNN처럼 local info는 처리하지 못한다.

02 ViT

두 번째로, 이미지의 resolution이 크면 연산량이 증가한다는 단점이 있다.

FLOPs(FLoating point OPerations) 는 초당 부동소수점연산을 의미하는데,
이는 FLOPs가 높을수록 모델의 연산량이 높다는 뜻이다.
(FLOPS=FLoating point Operations Per Second 와 다름)

02 ViT

두 번째

FLOPs
이는 F
(FLOP

Method Type	Network	#Param. (M)	image size	FLOPs (G)	ImageNet top-1 (%)	Real top-1 (%)	V2 top-1 (%)
Convolutional Networks	ResNet-50 [15]	25	224^2	4.1	76.2	82.5	63.3
	ResNet-101 [15]	45	224^2	7.9	77.4	83.7	65.7
	ResNet-152 [15]	60	224^2	11	78.3	84.1	67.0
Transformers	ViT-B/16 [11]	86	384^2	55.5	77.9	83.6	–
	ViT-L/16 [11]	307	384^2	191.1	76.5	82.2	–
	DeiT-S [30][arxiv 2020]	22	224^2	4.6	79.8	85.7	68.5
	DeiT-B [30][arxiv 2020]	86	224^2	17.6	81.8	86.7	71.5
	PVT-Small [34][arxiv 2021]	25	224^2	3.8	79.8	–	–
	PVT-Medium [34][arxiv 2021]	44	224^2	6.7	81.2	–	–
	PVT-Large [34][arxiv 2021]	61	224^2	9.8	81.7	–	–
	T2T-ViT$_t$-14 [41][arxiv 2021]	22	224^2	6.1	80.7	–	–
	T2T-ViT$_t$-19 [41][arxiv 2021]	39	224^2	9.8	81.4	–	–
	T2T-ViT$_t$-24 [41][arxiv 2021]	64	224^2	15.0	82.2	–	–
	TNT-S [14][arxiv 2021]	24	224^2	5.2	81.3	–	–
	TNT-B [14][arxiv 2021]	66	224^2	14.1	82.8	–	–
Convolutional Transformers	**Ours:** CvT-13	20	224^2	4.5	81.6	86.7	70.4
	Ours: CvT-21	32	224^2	7.1	82.5	87.2	71.3
	Ours: CvT-13$_{\uparrow 384}$	20	384^2	16.3	83.0	87.9	71.9
	Ours: CvT-21$_{\uparrow 384}$	32	384^2	24.9	**83.3**	**87.7**	**71.9**
	Ours: CvT-13-NAS	18	224^2	4.1	82.2	87.5	71.3

02 ViT

세 번째로,
충분히 많은 데이터가 없을 때는(예를들어 ImageNet) ViT가 잘 동작하지 않는다는 단점이 있다.
(논문 Abstract에 언급되어 있음.)
이는 Transformer가 Inductive bias가 부족해서 나타나는 결과라고 한다.

Inductive bias? : ML Model이 학습과정에서 새로운 데이터에 대해 잘 일반화 할 수 있도록 돕는 가정
모델이 학습 할 때 우리가 세우는 가정들
CNN : 인접한 픽셀끼리는 서로 관련이 있다는 가정을 함
Linear Regression : 데이터들이 Linear한 관계에 있음을 가정

02 ViT

이 단점들을 해결하기 위해

Swin Transformer, Convolution Vision Transformer(CvT), Pyramid Vision Transformer(PVT), DeiT(Training data-efficient image transformers & distillation through attention)

등이 발표되었다.

03 Results of training with CIFAR10

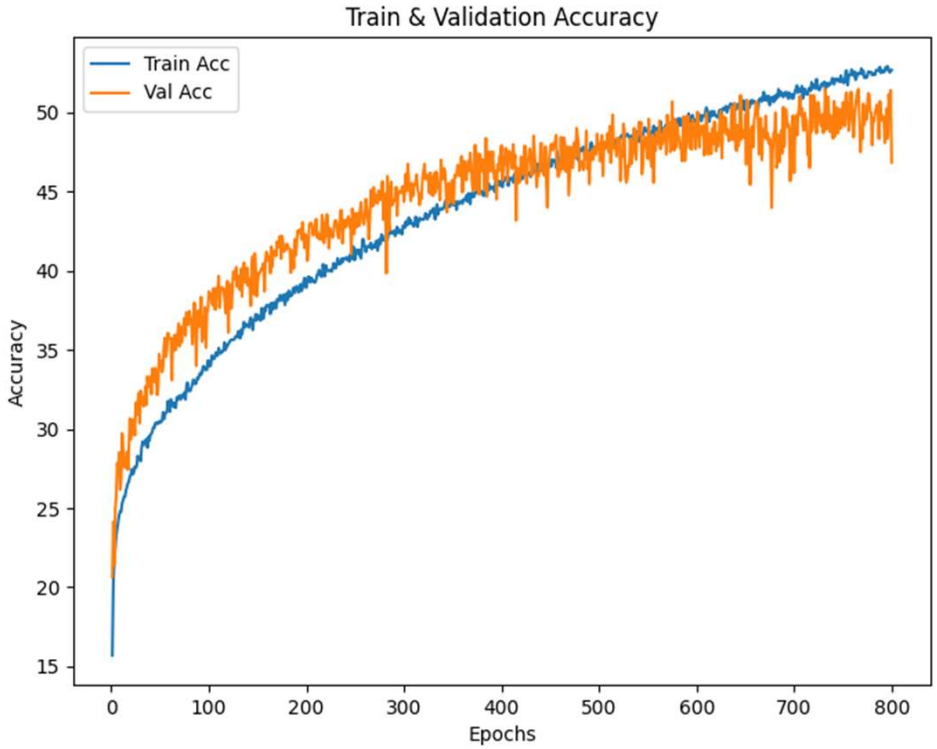
Parameter는 다음과 같이 설계하였다.

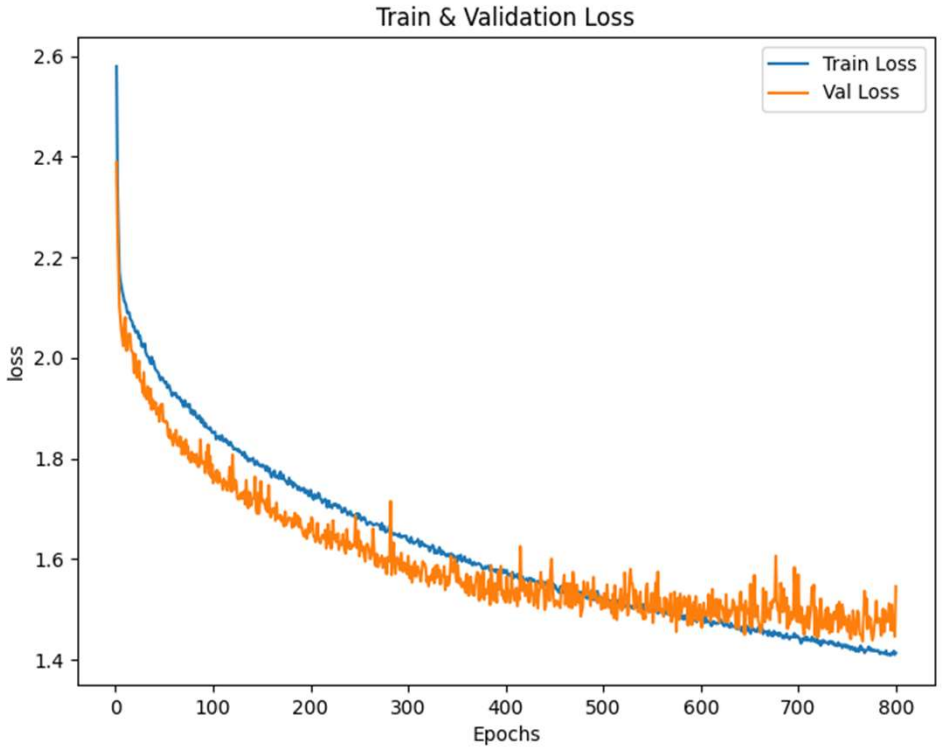
03 Results of training with CIFAR10

Parameter

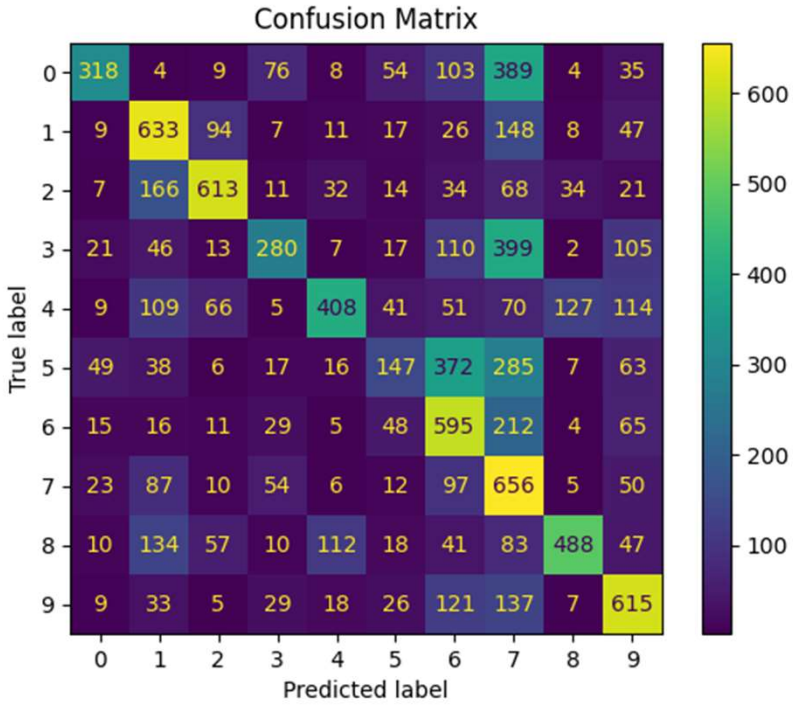
```
training_device: lab
dataset_dir: /Data_RTX4090_server/datasets/cifar10/
epochs: 800
batch_size: 512
learning_rate: 5e-07
optimizer: Adam
act_fn: GeLU
device: cuda
dropout_rate: 0.1
random_seed: 123
model_summary_dir: model_summary.txt
result_dir: ./result_dir/20240826_1747/
classification_report_dir: classification_report.txt
now_epochs: 1
patch_size: 8
num_enc_layers: 24
mlp_size: 4096
hidden_dim: 1024
num_heads: 16
num_classes: 10
```

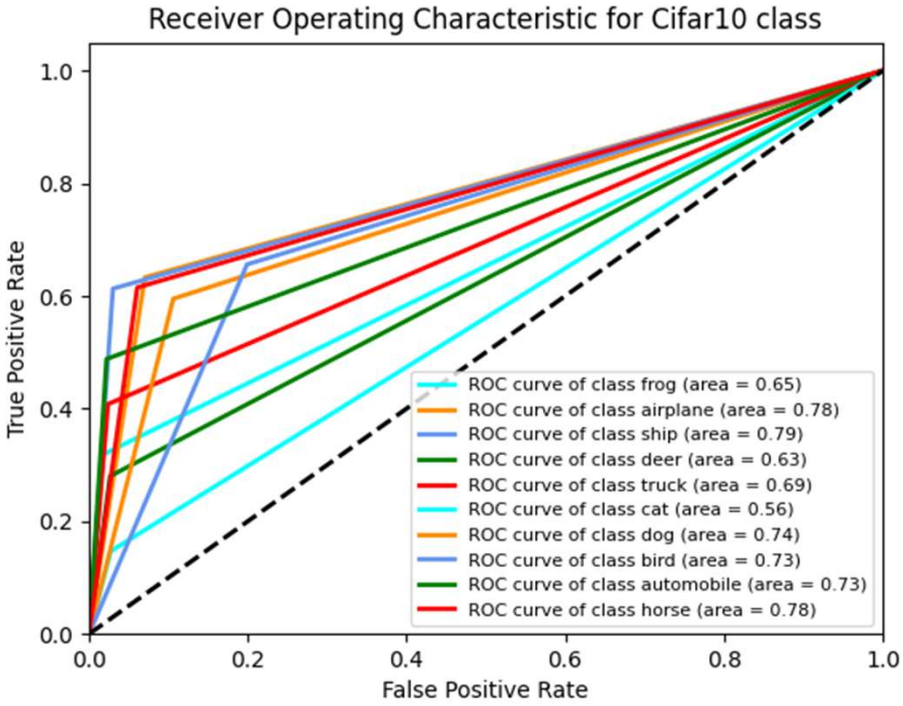
03 Results of training with CIFAR10





03 Results of training with CIFAR10





감사합니다

o

o

o