Image Classification Competition

202182183 신상훈
202182184 이은주
202182185 조근수

Vision Transformer

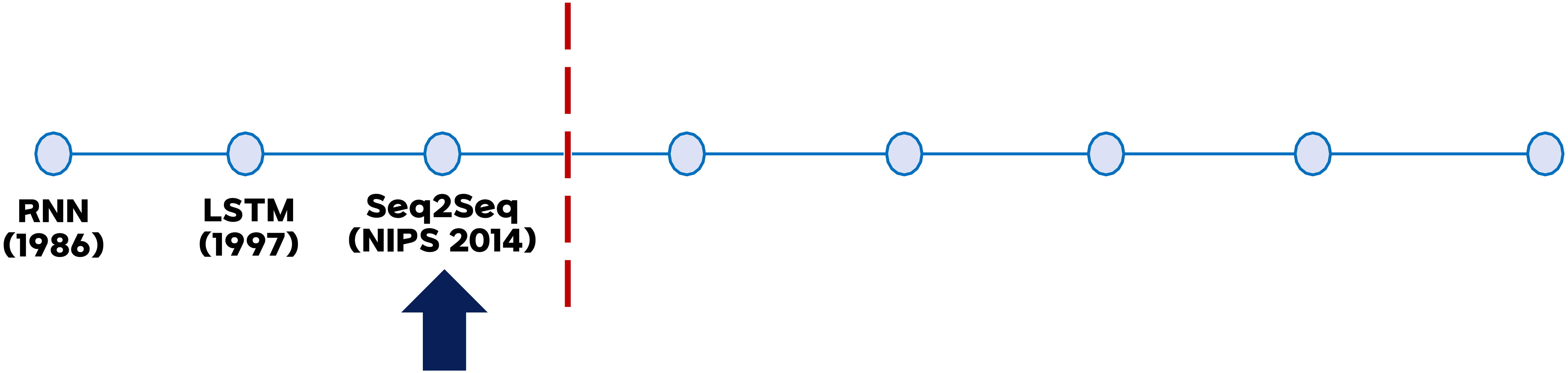
- ✓ 기존의 Convolution을 사용하지 않음
- ✓ 이미지 패치를 단어와 같이 다룸
- ✓ 사전학습을 통해 뛰어난 성능을 보임

CONTENTS

01

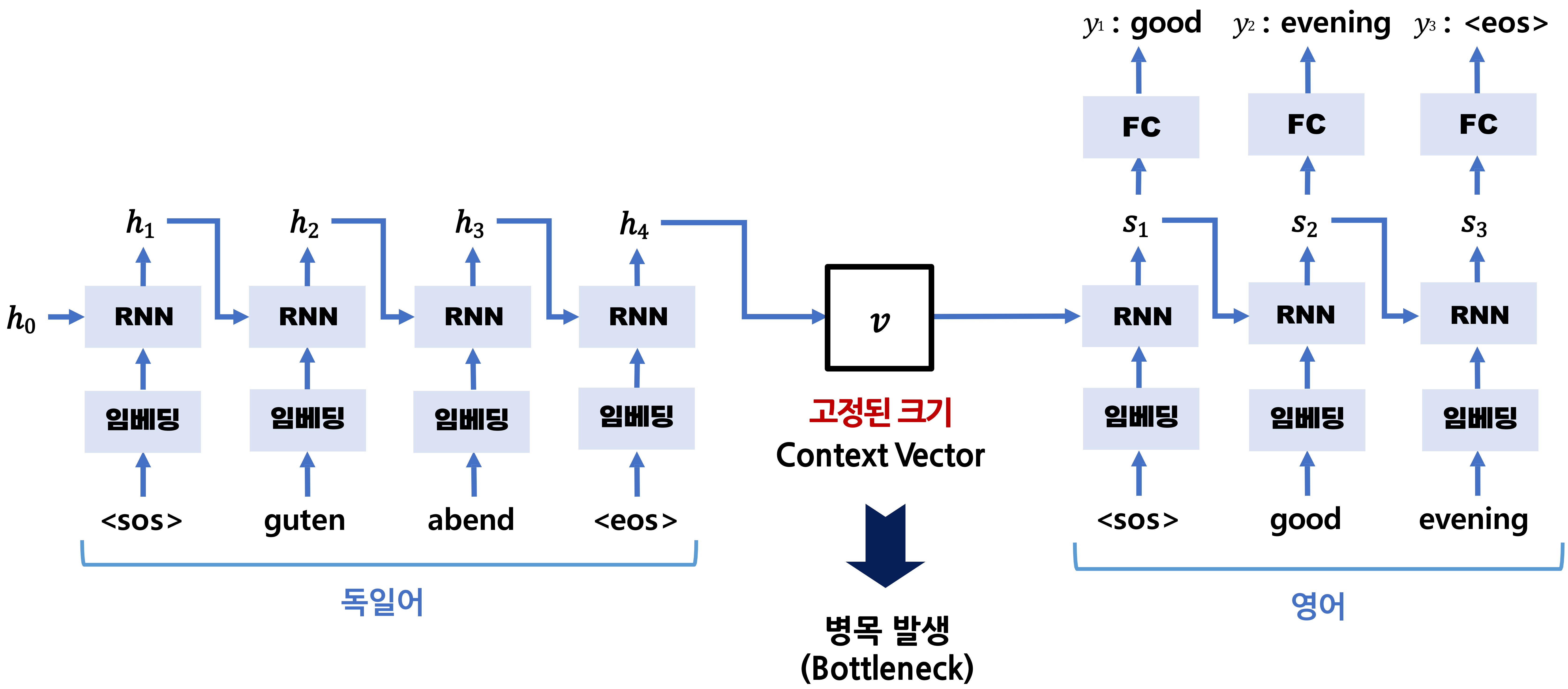
Introduction

1. Limit of Seq2Seq
2. Background of VIT



고정된 크기의
Context Vector 사용

Seq2Seq


y1 : good
y2 : evening
y3 : <eos>
FC
FC
FC
h1
h2
h3
h4
s1
s2
s3
h0
RNN
RNN
RNN
RNN
v
RNN
RNN
RNN
임베딩
임베딩
임베딩
임베딩
고정된 크기
Context Vector
임베딩
임베딩
임베딩
<sos>
guten
abend
<eos>
<sos>
good
evening
독일어
병목 발생
(Bottleneck)
영어

$\boxed{v}$

고정된 크기
Context Vector

: 하나의 벡터가 문장의 모든 정보를 포함

v

고정된 크기
Context Vector

: 하나의 벡터가 [illegible]의 모든 정보를 포함

성능 저하

v

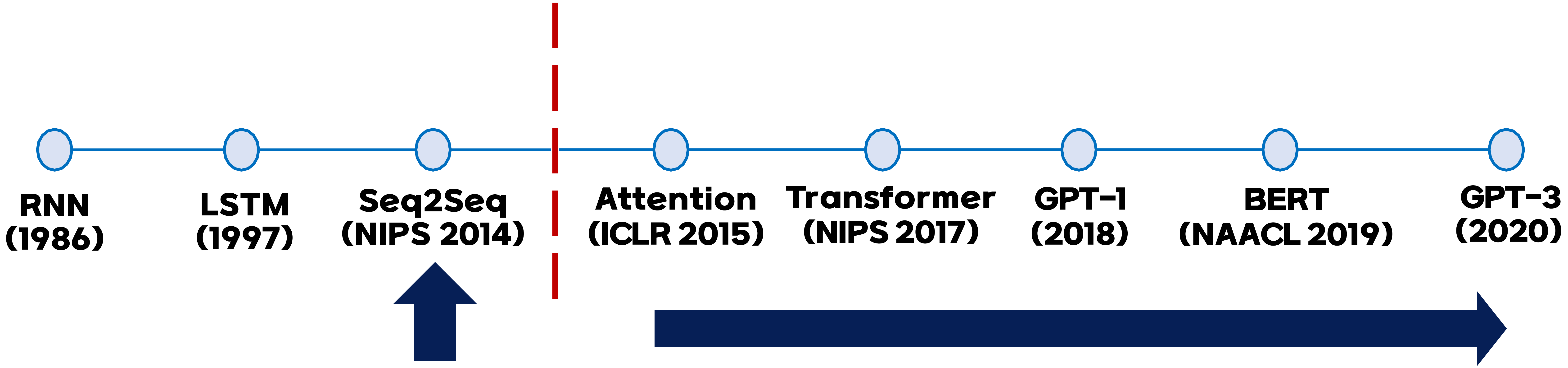
고정된 크기
Context Vector

: 하나의 벡터가 문장의 모든 정보를 포함

성능 저하

→ 매번 소스 문장에서의 출력 전부를 입력으로 받는 방식

이후 발전과정



고정된 크기의
Context Vector 사용

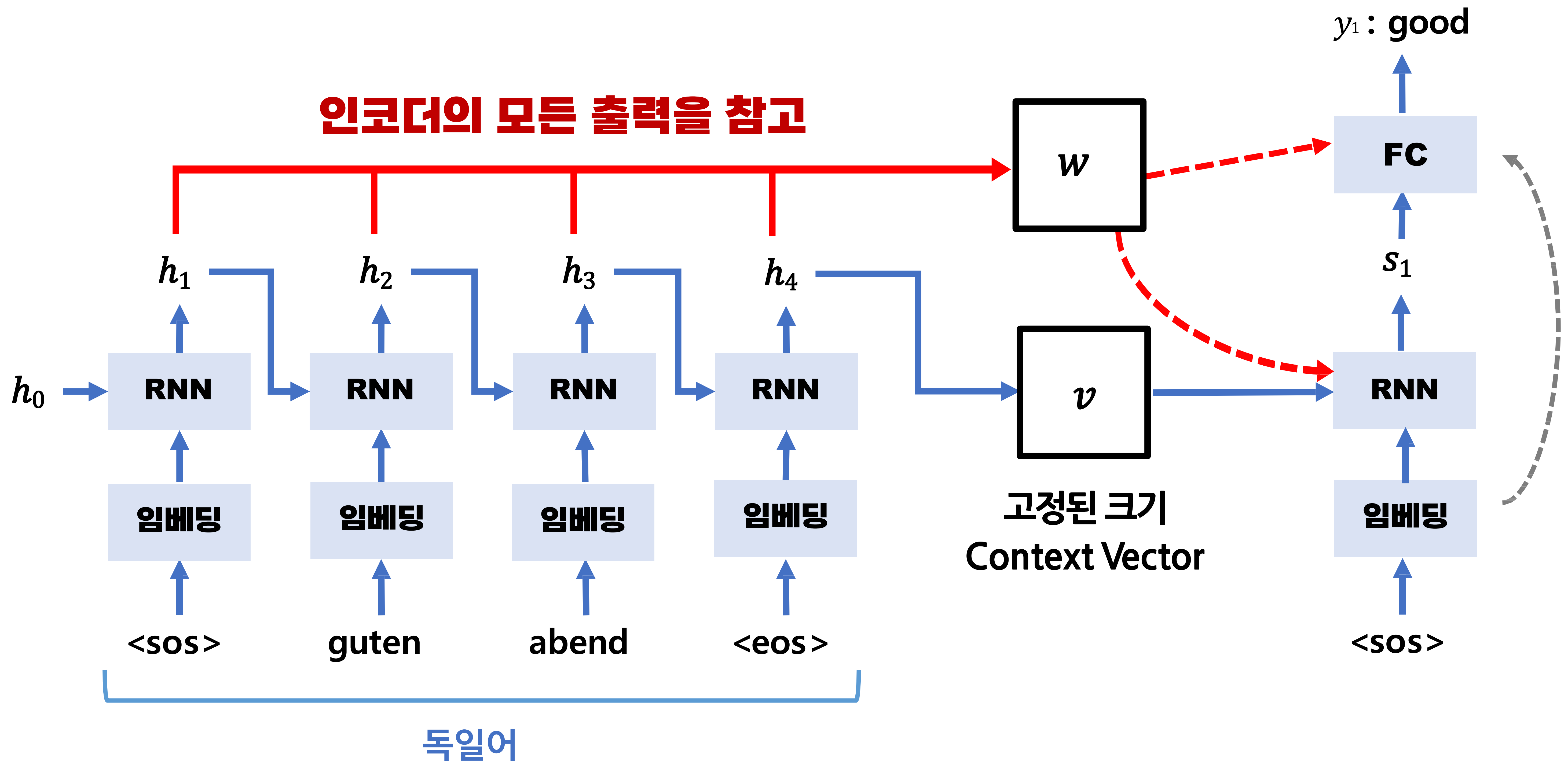
입력 시퀀스 전체에서 정보를 추출하는 방향으로 발전

02

Method

1. Attention
2. Transformer
3. Vision Transformer

Attention – Seq2Seq



Attention – Query, Key, Value

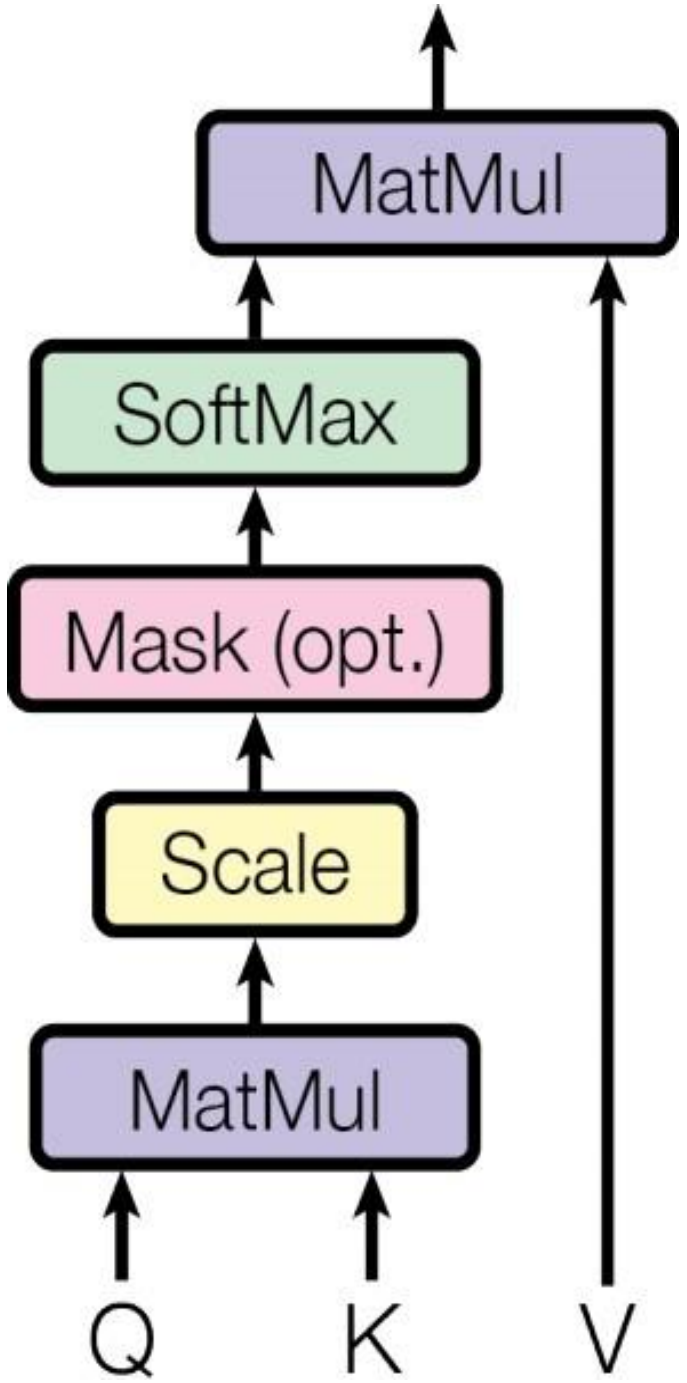
Key : 물어보는 대상
'I am a teacher'

Query : { 'I':10 , 'am':7 , 'a':2, 'teacher':1 }

물어보는 주체
' I '

Value : 한 단어가 다른 단어들과
얼마나 강한 연관성이 있는가

Attention - Architecture



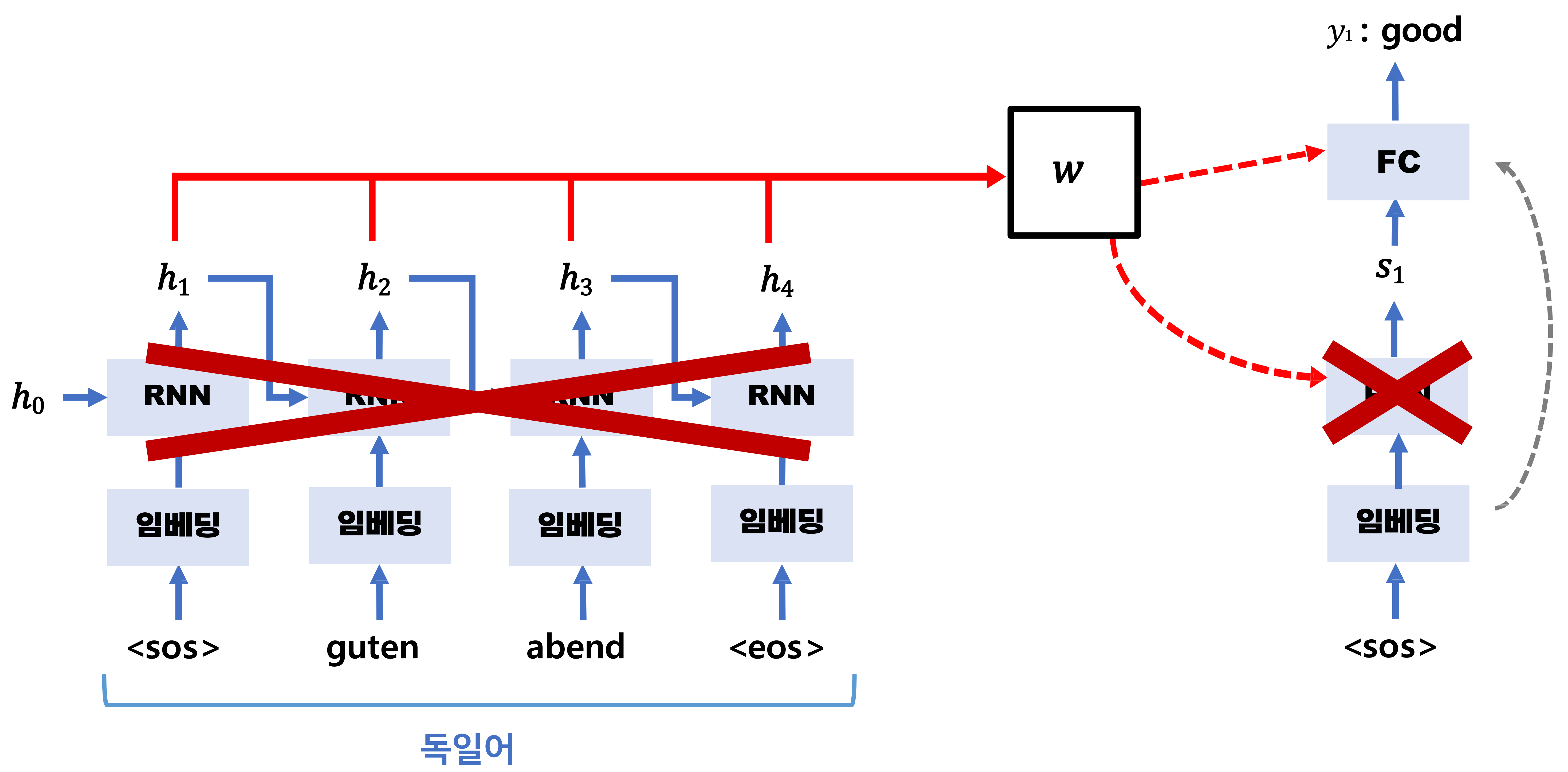
$\mathrm{Softmax}(\mathbf{QK^T}/\sqrt{\mathbf{d_k}})\cdot \mathrm{V}$ = Attention

$\mathrm{Softmax}(\mathbf{QK^T}/\sqrt{\mathbf{d_k}})$

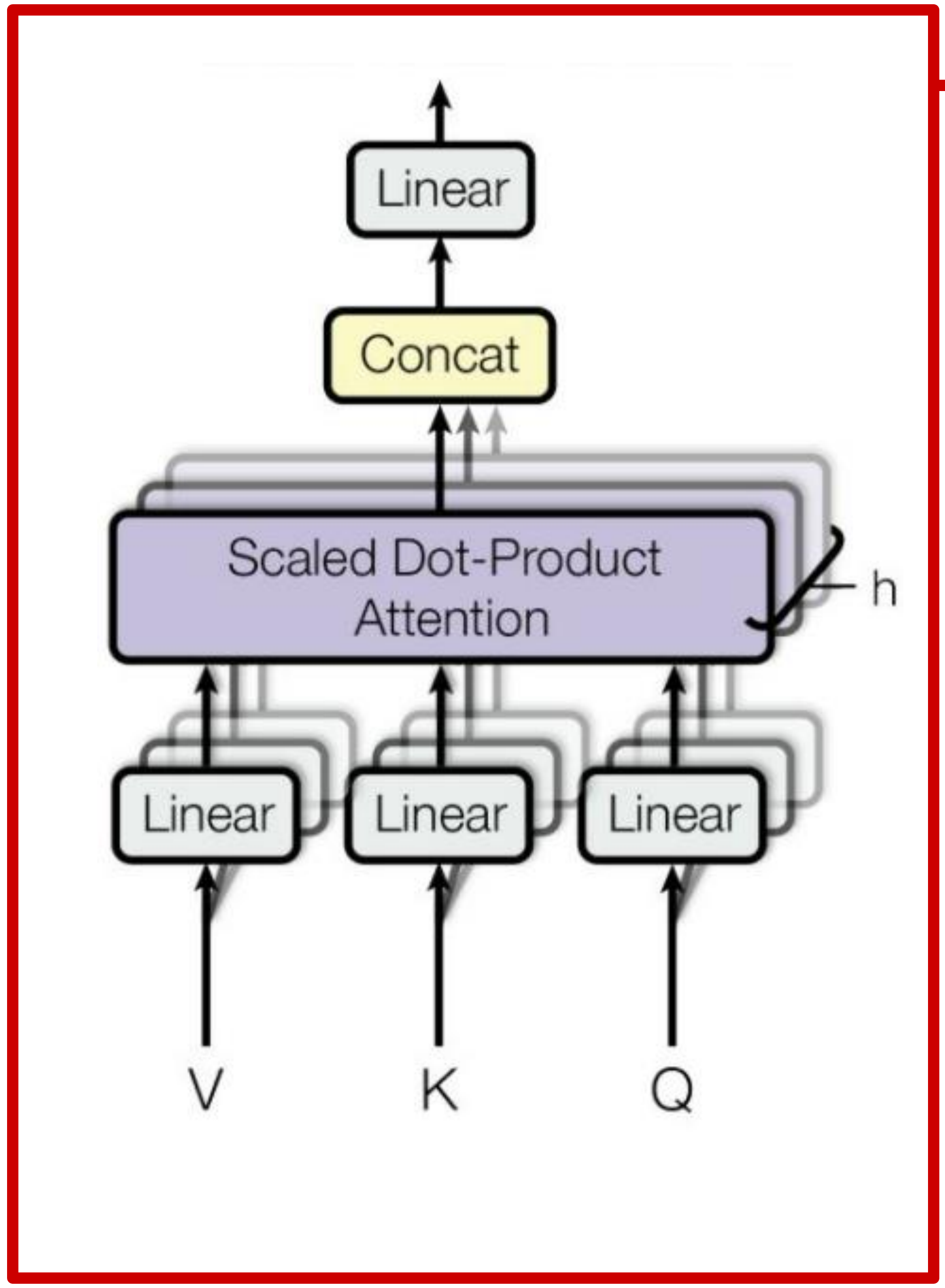
$\mathbf{QK^T}/\sqrt{\mathbf{d_k}}$ ※ Gradient Vanishing 문제↓

QK^T

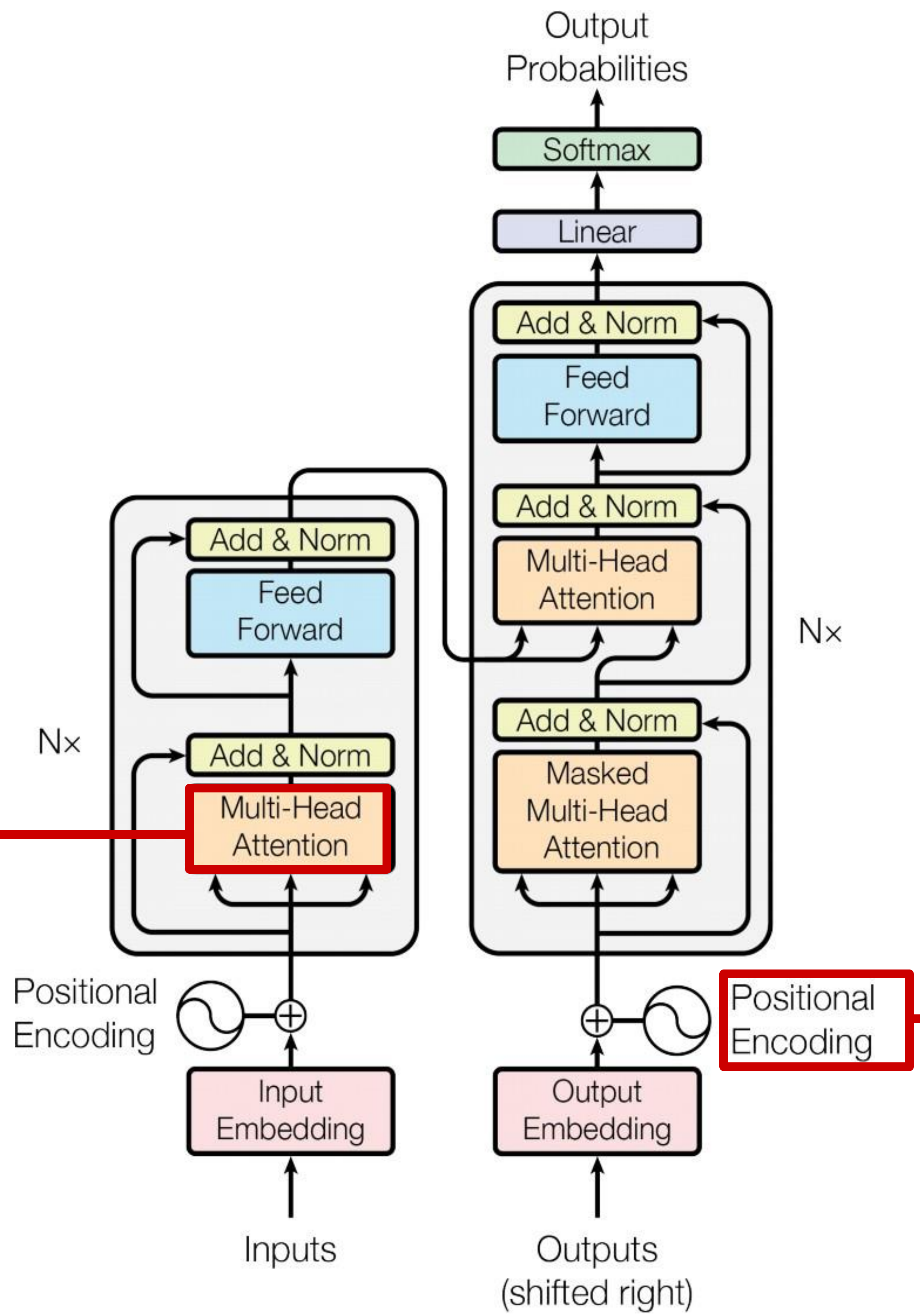
Transformer

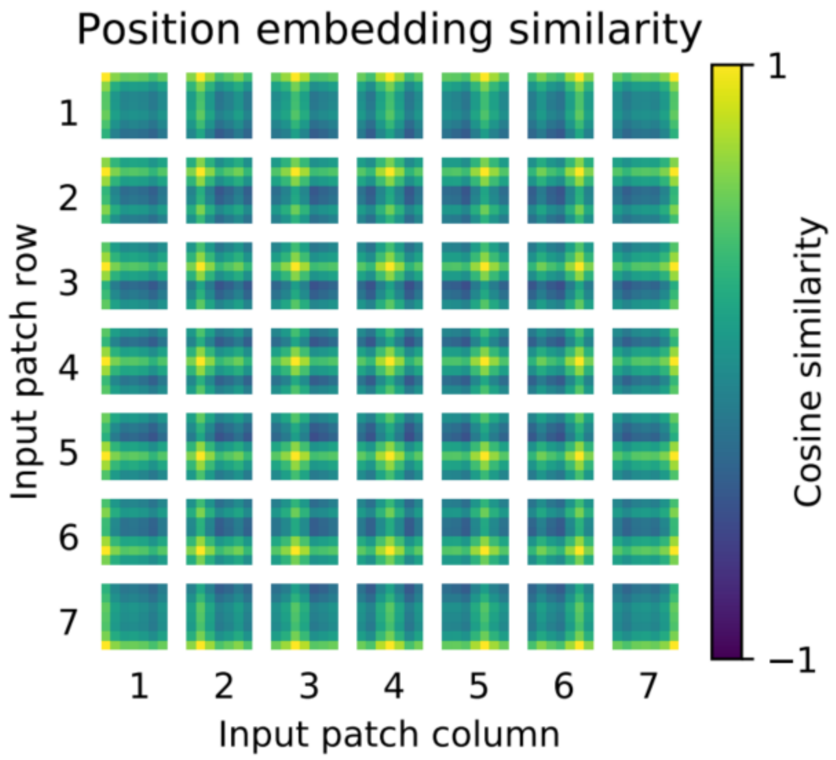


Transformer



여러 관점에서 정보 수집





위치 및 순서 정보

Transformer

[기존]

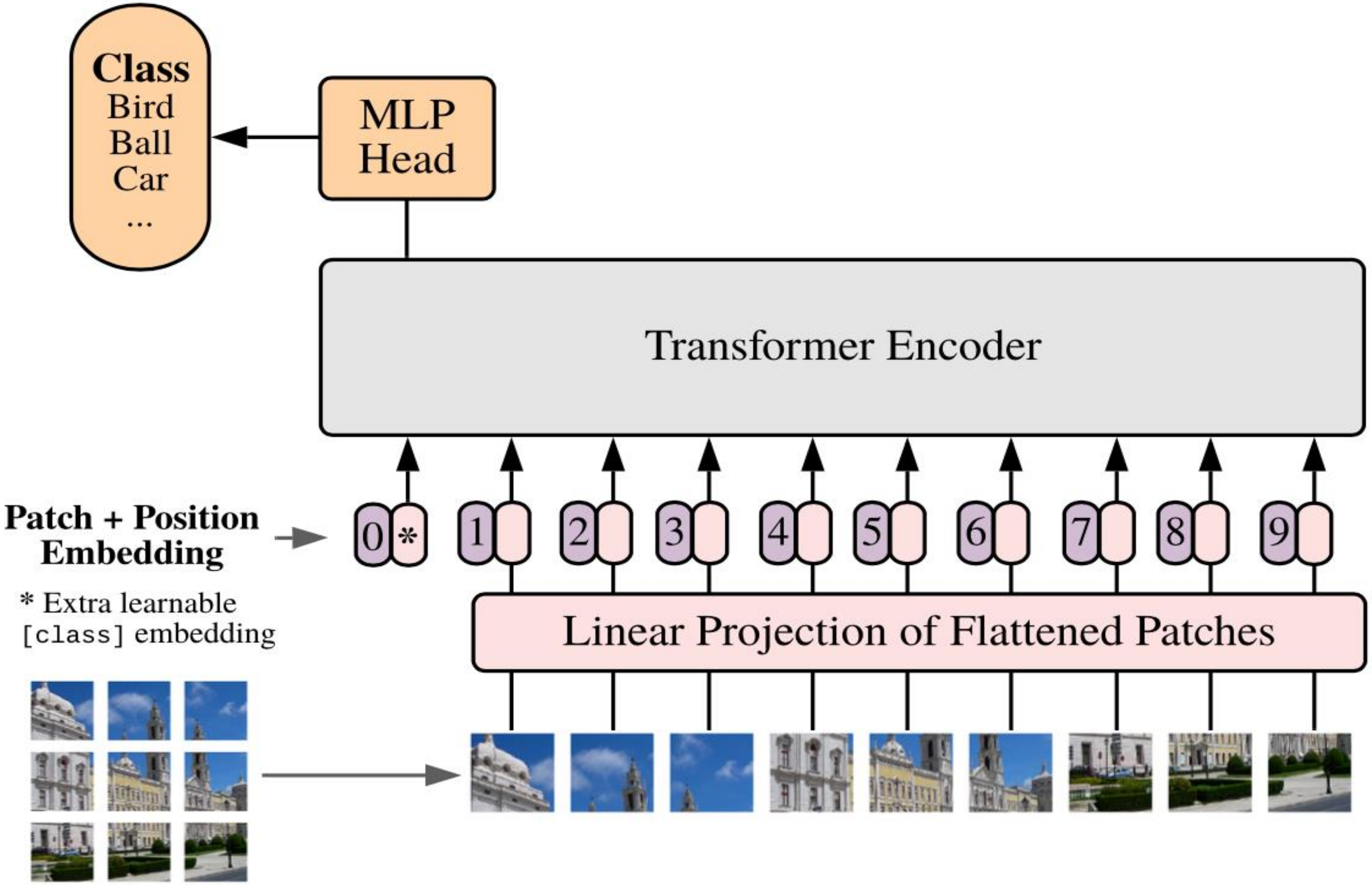
Encoder – Decoder 포함하여
순환신경망 사용

[Transformer]

Attention 메커니즘만을 사용한
Encoder – Decoder 구조

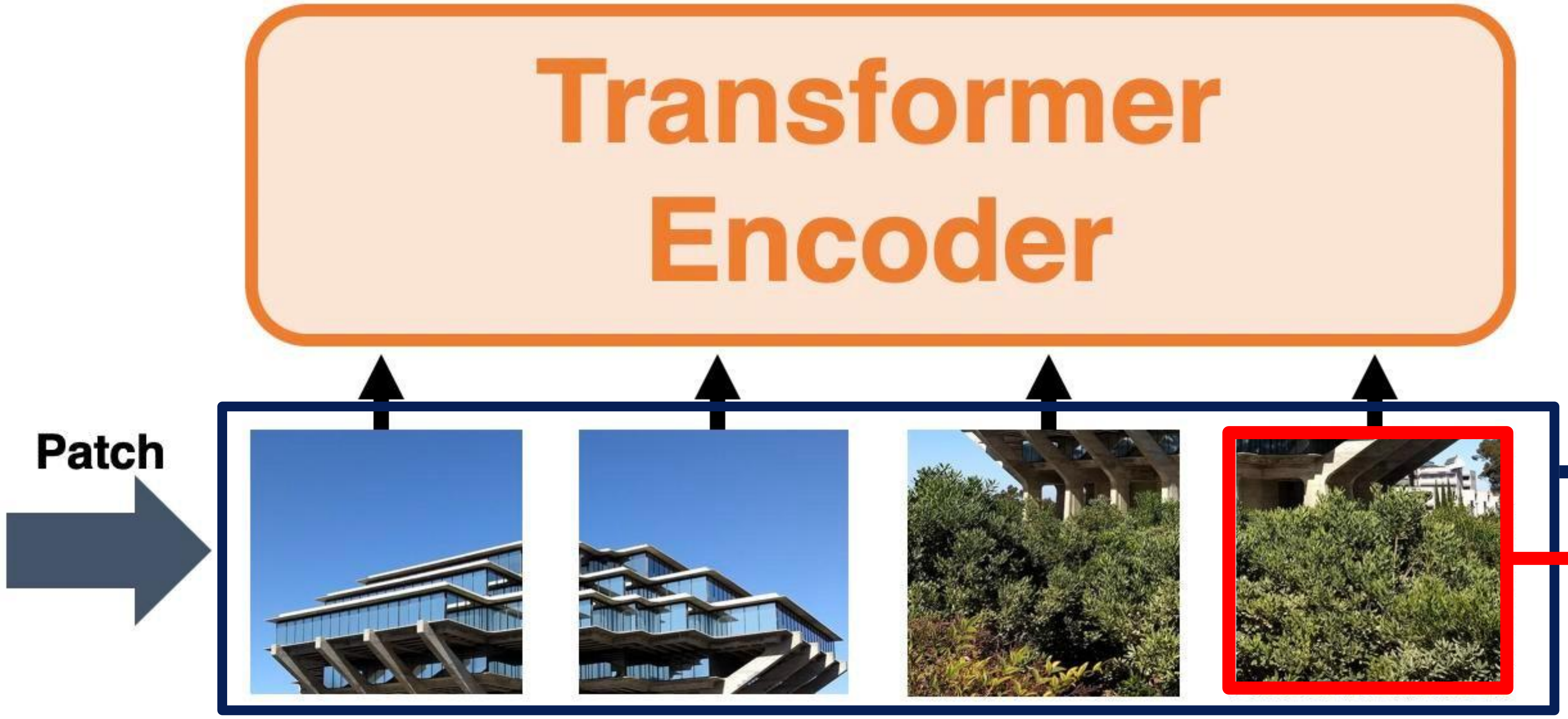
1. 기계번역 Task에서 매우 좋은 성능
2. 학습 시 우수한 병렬화에 따라 훨씬 적은 시간 소요
3. 구문 분석 분야에서도 우수한 성능, 즉 일반화 또한 뛰어남

Vision Transformer (ViT)



ViT – (1) Input Image

$\boldsymbol{X} \in \mathbb{R}^{\mathrm{H} \times \mathrm{W} \times \mathrm{C}}$

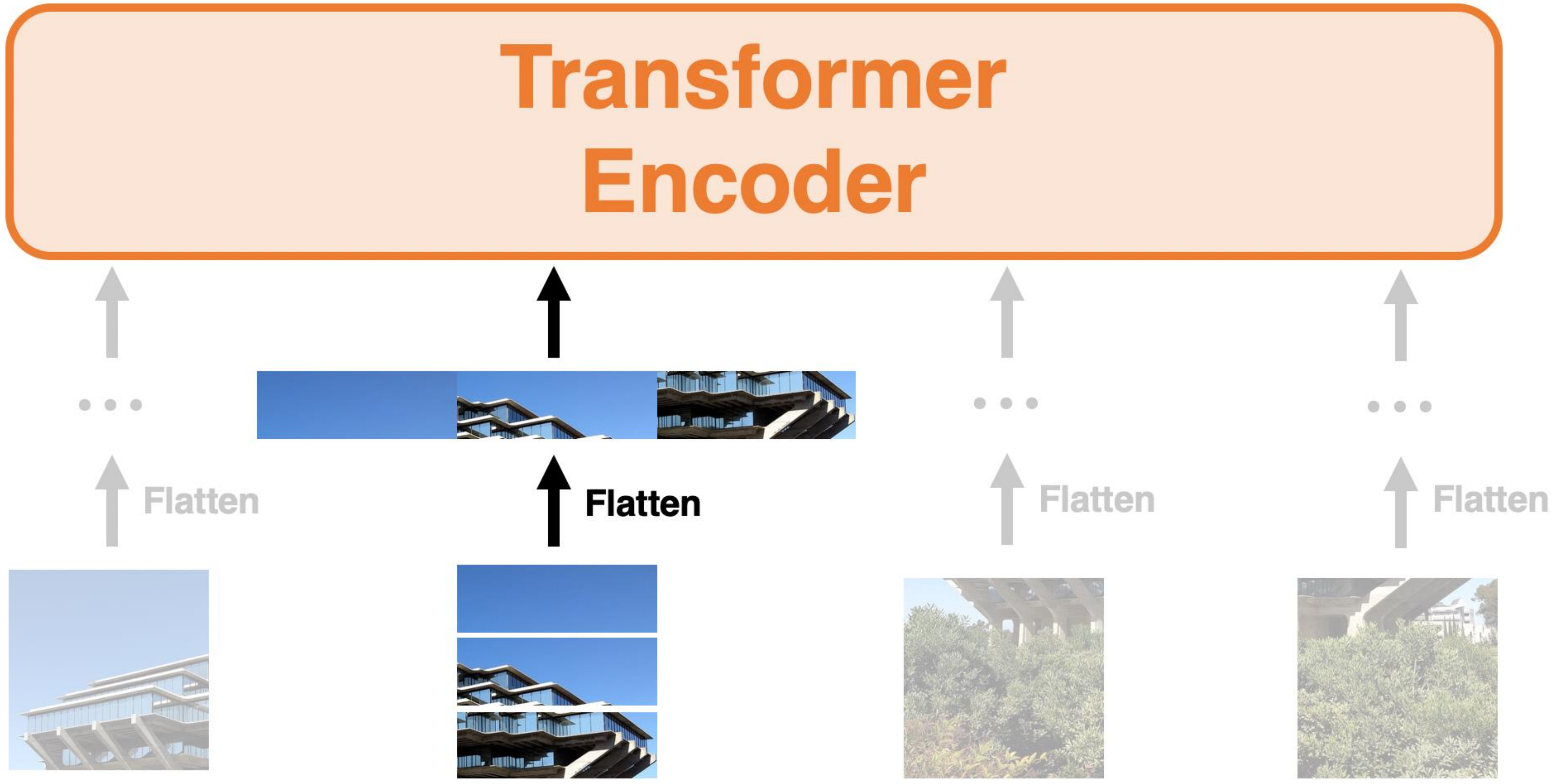


개수 : $\mathrm{N} = \mathrm{HW}/\mathrm{P}^2$

크기 : P^2

이미지 패치를 나누어 각 패치를 단어처럼 다룸

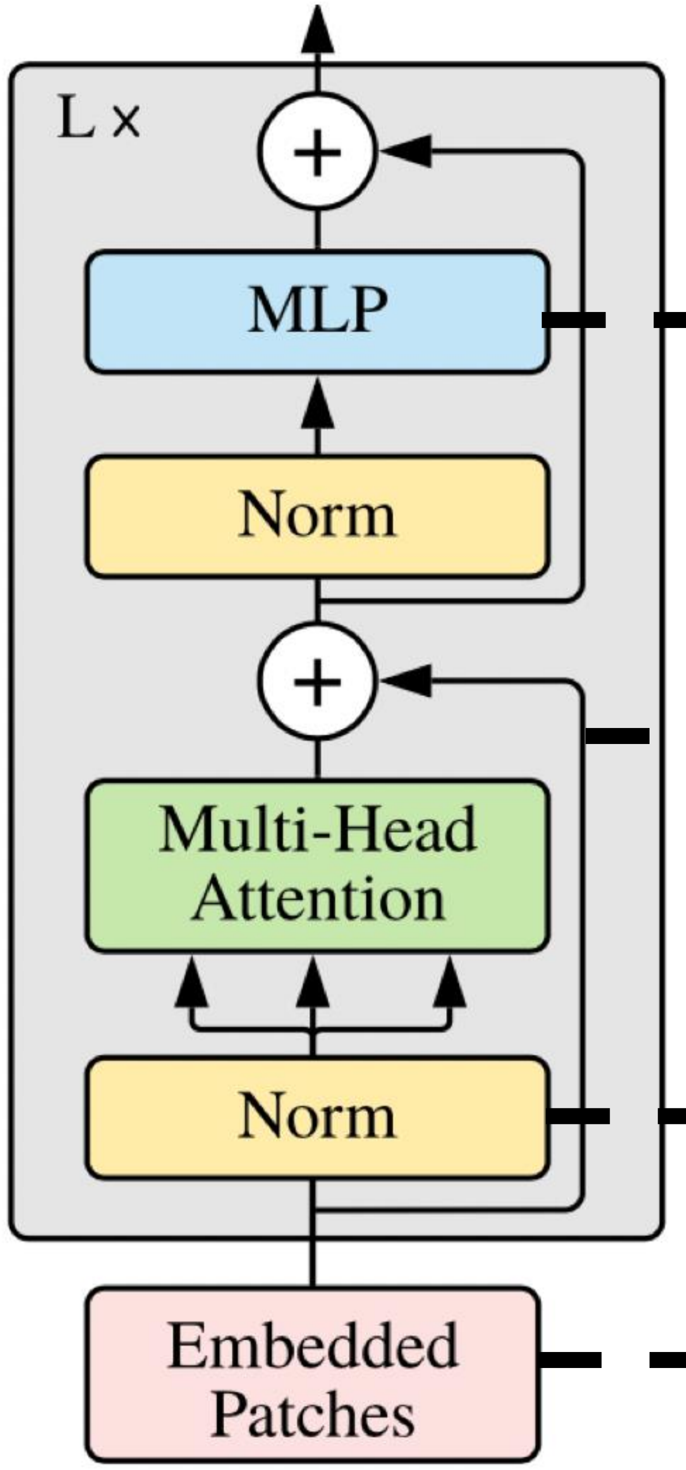
ViT – (1) Input Image



단어처럼 처리

$$\boldsymbol{X}_p \in \mathbb{R}^{N \times (P^2 C)}$$

ViT – (2) Transformer Encoder



Activation Function : GELU

※ $GELU(x) = x \cdot P(X \leq x)$

Residual Block

Pre-Norm

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

ViT – (3) Pretrained model & Fine tuning

Pretrained model

Fine tuning

새로운 목적에 맞게 변형

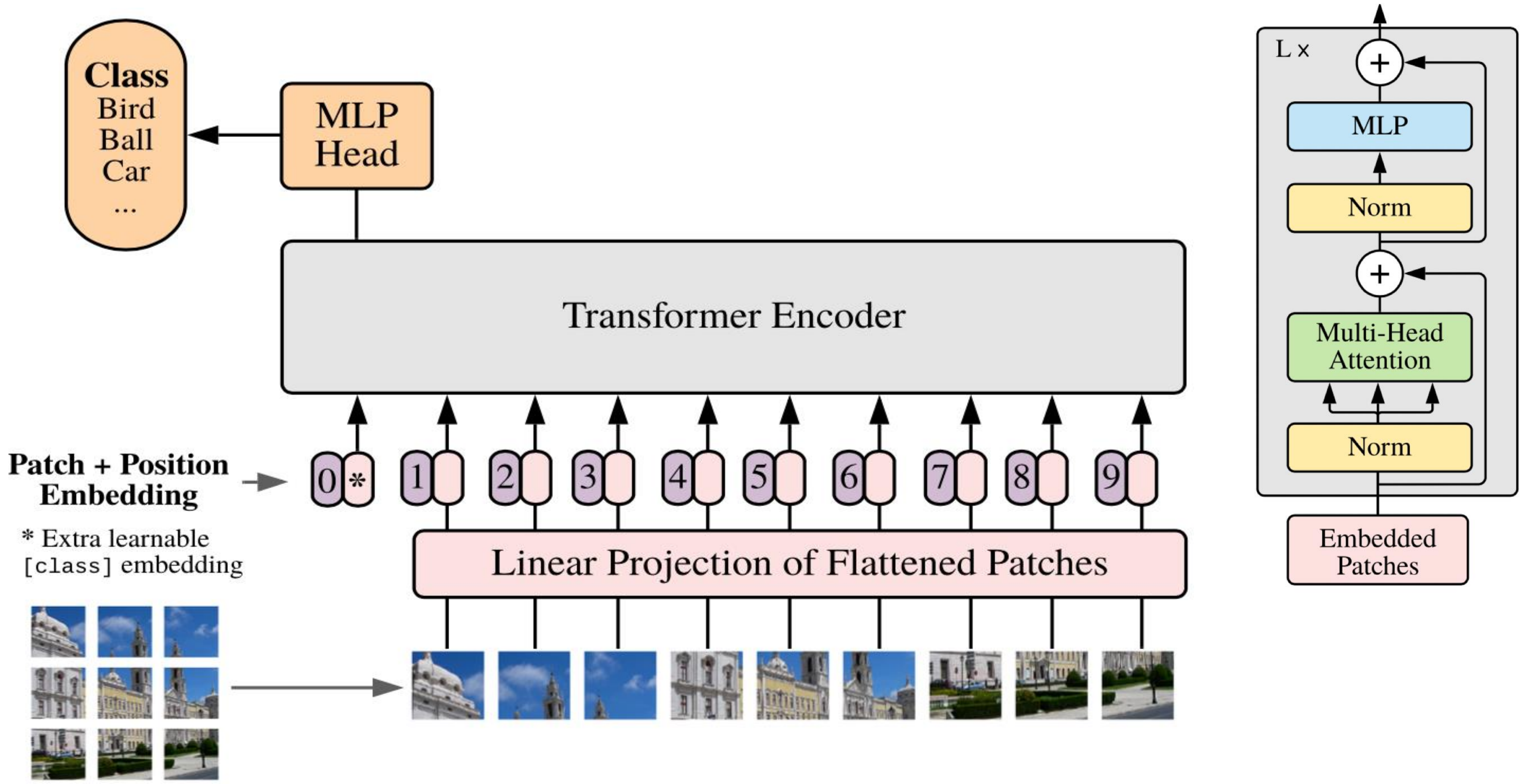
Model to use

사이즈가 큰 데이터로 미리 학습

Transfer Learning (전이학습)

Vision Transformer (ViT)

Pretrained model



$$z_l = MLP(LN(z_l')) + z_l'$$

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

$$z_0 = [x_{class};\ x_p^1 E;\ x_p^2 E;\ \cdots;\ x_p^N E] + E_{pos}$$

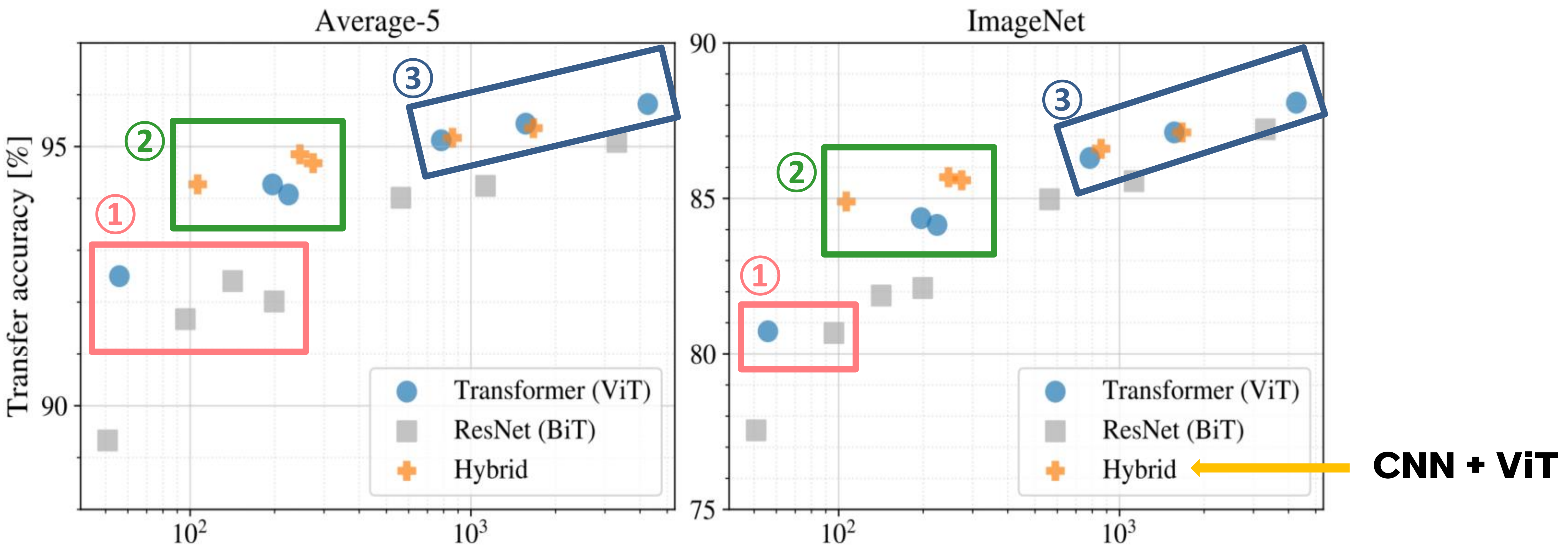
Fine tuning

ViT - Performance

	ViT-H/14 (JFT)	ViT-L/16 (JFT)	ViT-L/16 (I21k)	ResNet 152×4
ImageNet	**88.36**	87.61	85.3	87.54
CIFAR-10	**99.50**	99.42	99.15	99.37
CIFAR-100	**94.55**	93.90	93.25	93.51
VTAB (19 tasks)	**77.16**	75.91	72.72	76.29

Low data & Diverse task

ViT - Performance



① ViT는 ResNet보다 비용 대비 성능이 좋음

② 비용이 한정된 경우, Hybrid가 제일 효과적 => 높은 비용일 경우 차이 거의 없음

③ ViT에 대한 더 업그레이드된 성능향상을 기대할 수 있음

03

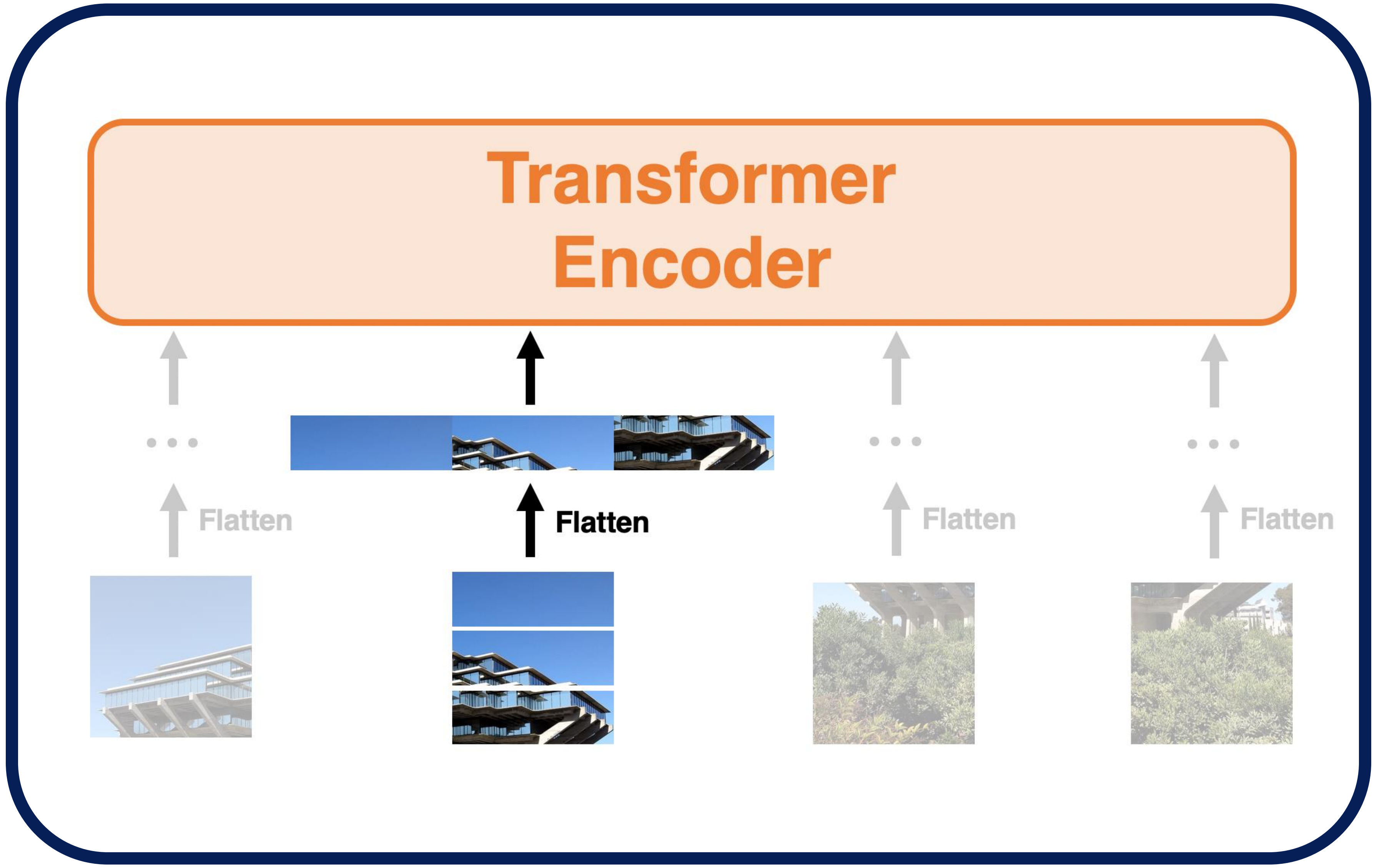
Analysis

1. Preprocessing
2. Vision Transformer
3. Result

Preprocessing

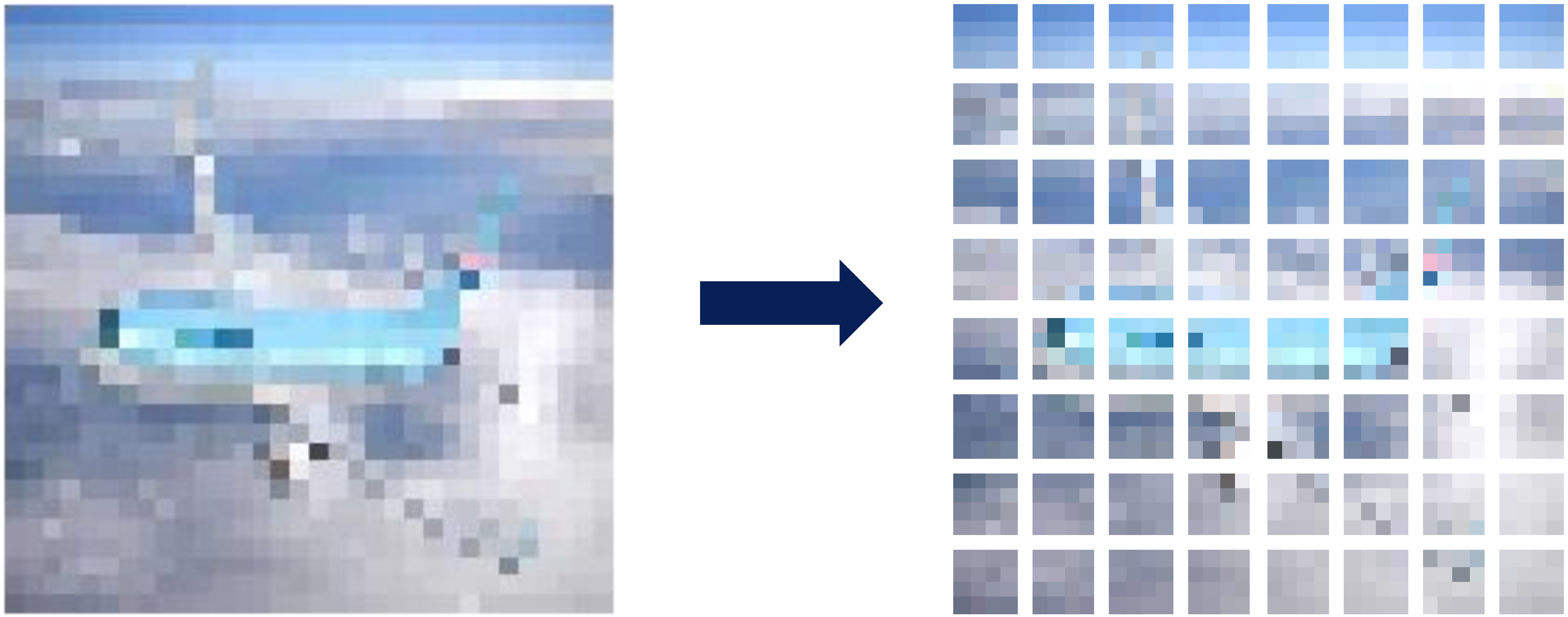
patchdata.Flattened2Dpatches(dataname, img_size, patch_size, batch_size)

32 **4** **512**




```
class Flattened2Dpatches:
  def patchdata(self):
    mean = (0.4914, 0.4822, 0.4465)
    std = (0.2023, 0.1994, 02010)
    transform = transforms.Compose([transforms.Resize(self.img_size),
                    transforms.RandomCrop(self.img_size, padding=2),
                    transforms.RandomHorizontalFlip(), transforms.ToTensor(),
                    transforms.Normalize(mean, std),
                    PatchGenerator(self.patch_size)])
```

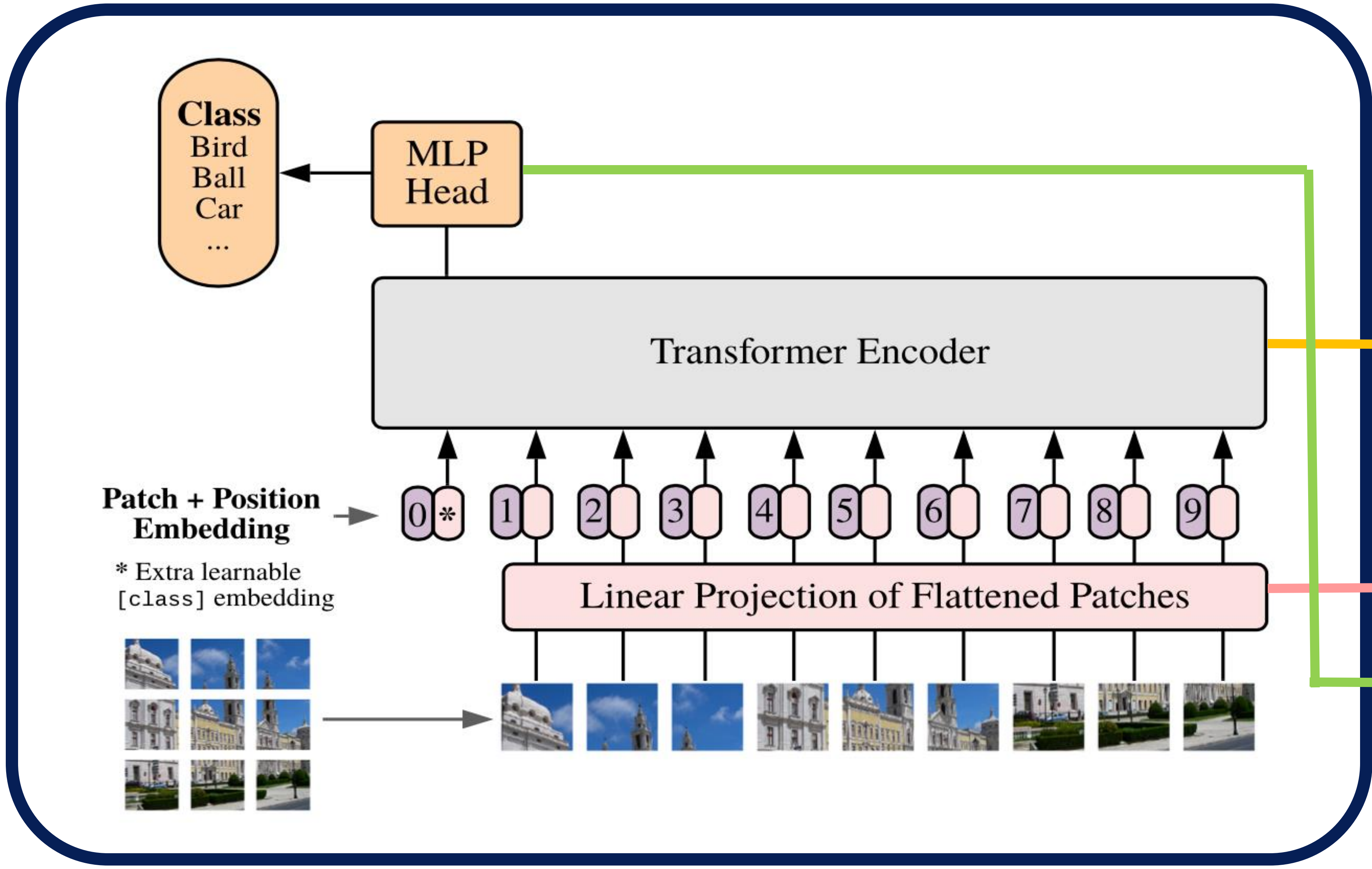
Preprocessing



Vision Transformer

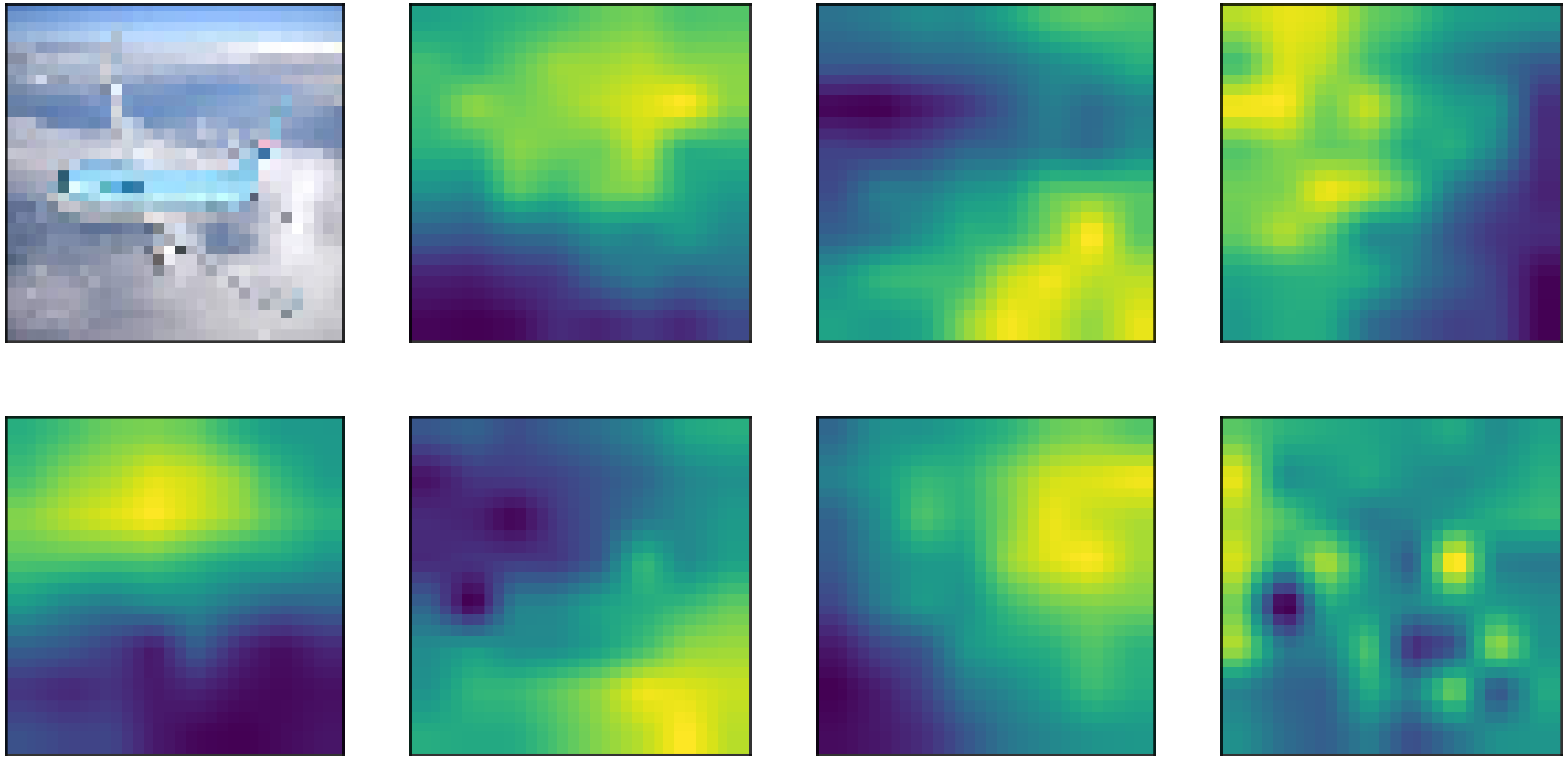
model.VisionTransformer(patch_vec_size, num_patches, latent_vec_dim, num_heads, mlp_hidden_dim, drop_rate, num_layers, num_classes)

latent_vec_dim: **128**, num_heads: **8**, drop_rate: **0.01**, num_layers: **12**, num_classes: **10**



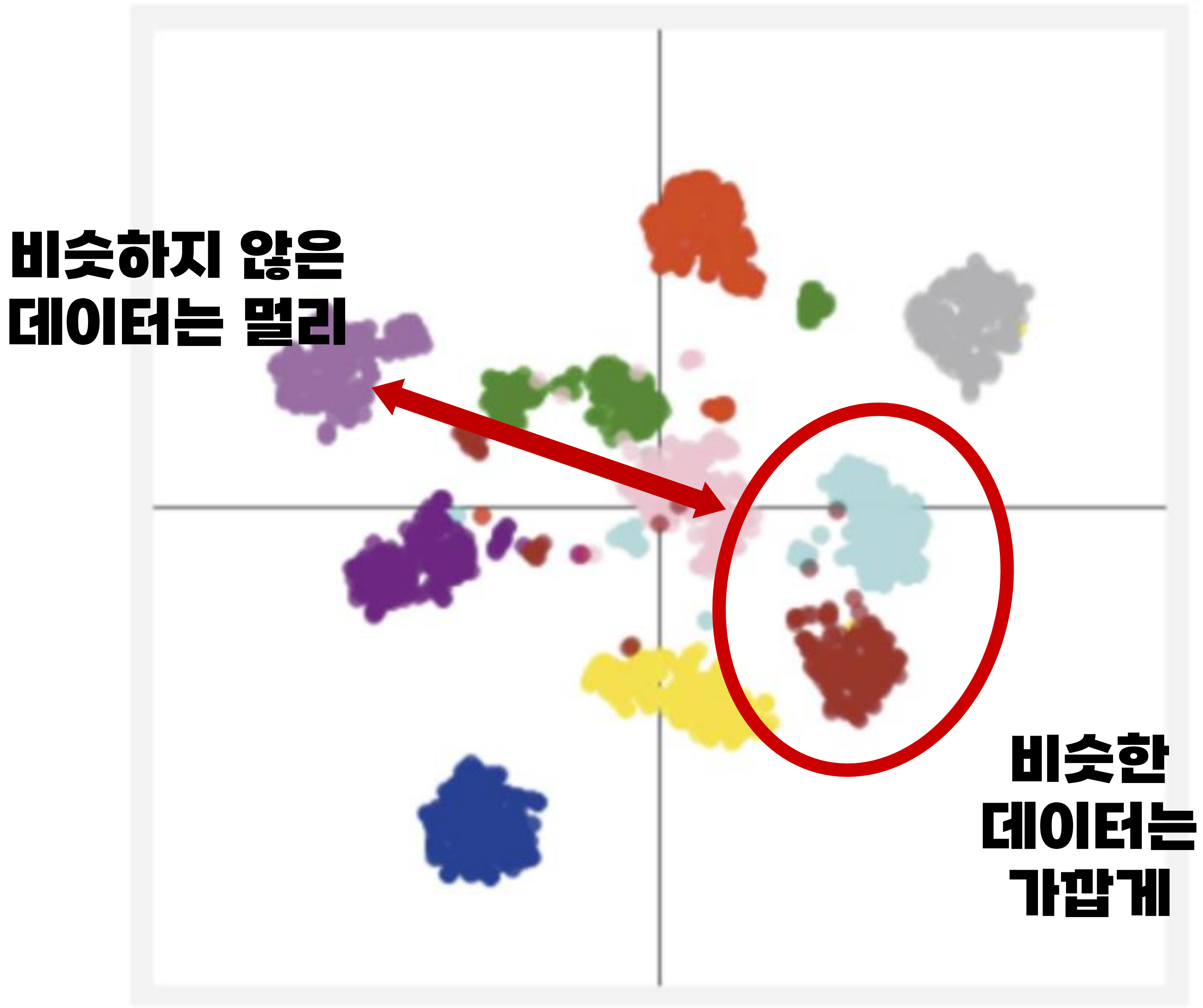

```
class VisionTransformer(nn.Module):
  def __init__(self, patch_vec_size, num_patches, latent_vev_dim, num_heads,
                    mlp_hidden_dim, drop_rate, num_layers, num_classes) :

    self.patchembedding = LinearProjection(patch_vec_size, num_patches,
                                           latent_vec_dim, drop_rate)
    self.transformer = nn.ModuleList([TFencoderLayer(latent_vec_dim,
                                     num_heads, mlp_hidden_dim, drop_rate)
                                     for _ in range(num_layers)])
    self.mlp_head = nn.Sequential(nn.LayerNorm(latent_vec_dim),
                                  nn.Linear(latent_vec_dim, num_classes))
```

Vision Transformer

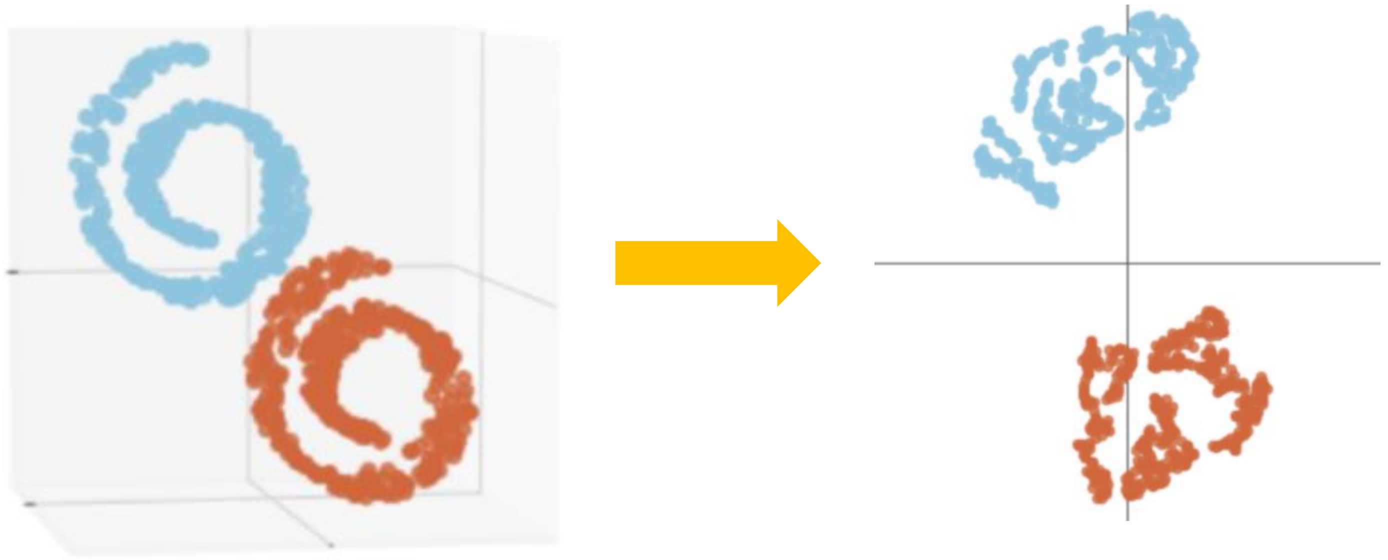


Result : T - SNE

※ T – SNE 란?

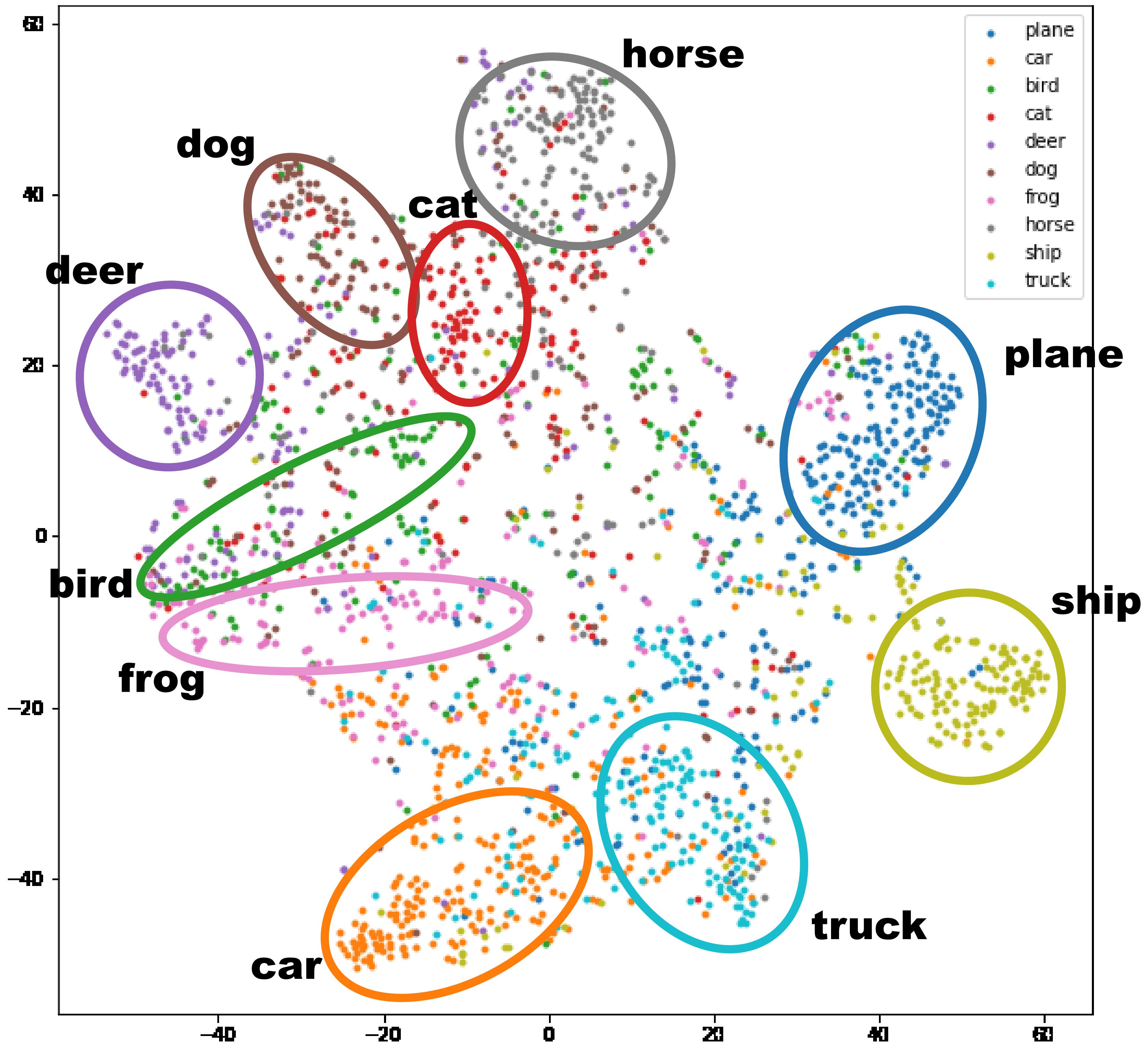


:



높은 차원의 복잡한 데이터를 2차원으로 축소
시각화를 통해 데이터 구조 이해

Result : T - SNE


plane
car
bird
cat
deer
dog
frog
horse
ship
truck
horse
dog
cat
deer
plane
bird
ship
frog
car
truck
60
40
20
0
−20
−40
−40
−20
0
20
40
60

Result : Confusion Matrix

True label \ Predicted label	Plane	Car	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck
Plane	260 (87%)	10	8	2	5	0	3	2	7	3
Car	22	223 (74%)	3	2	0	0	5	1	14	30
Bird	4	2	102 (51%)	44	7	13	23	5	0	0
Cat	1	0	8	131 (66%)	15	21	8	15	1	0
Deer	1	0	8	12	167 (84%)	4	3	5	0	0
Dog	7	0	13	32	13	111 (56%)	4	17	2	1
Frog	19	2	27	13	7	4	118 (59%)	2	2	6
Horse	2	0	11	13	32	3	1	136 (68%)	2	1
Ship	17	1	5	2	2	0	1	0	172 (86%)	0
Truck	28	32	3	5	0	0	4	1	4	123 (62%)

Result : Loss & Accuracy

	Train	Validation	Test
Loss	0.171	0.546	1.151
Accuracy	93.90%	84.29%	70.10%

Thank you

Reference

[1] [논문] Attention Is All You Need

– Google Research, Brain Team

[2] Transformer: Attention Is All You Need (꼼꼼한 딥러닝 논문 리뷰와 코드 실습)

: https://www.youtube.com/watch?v=AA621UofTUA

[3] [논문] An Image Is Worth 16×16 Words: Transformers For Image Recognition At Scale

– Google Research, Brain Team

[4] 최근 AI의 이미지 인식에서 화제인 "Vision Transformer"에 대한 해설

: https://engineer-mole.tistory.com/133

[5] t-SNE 개념과 사용법

: https://gaussian37.github.io/ml-concept-t_sne/

CODE

```python
class PatchGenerator:

    def __init__(self, patch_size):
        self.patch_size = patch_size

    def __call__(self, img):
        num_channels = img.size(0)
        patches = img.unfold(1, self.patch_size, self.patch_size).unfold(2, self.patch_size, self.patch_size).reshape(num_channels, -1,
self.patch_size, self.patch_size)
        patches = patches.permute(1,0,2,3)
        num_patch = patches.size(0)

        return patches.reshape(num_patch,-1)

class Flattened2Dpatches:

    def __init__(self, patch_size=16, dataname='imagenet', img_size=256, batch_size=64):
        self.patch_size = patch_size
        self.dataname = dataname
        self.img_size = img_size
        self.batch_size = batch_size

    def make_weights(self, labels, nclasses):
        labels = np.array(labels)
        weight_list = []
        for cls in range(nclasses):
            idx = np.where(labels == cls)[0]
            count = len(idx)
            weight = 1 / count
            weights = [weight] * count
            weight_list += weights
        return weight_list
```

```python
def patchdata(self):
    mean = (0.4914, 0.4822, 0.4465)
    std = (0.2023, 0.1994, 0.2010)
    train_transform = transforms.Compose([transforms.Resize(self.img_size), transforms.RandomCrop(self.img_size, padding=2),
                                          transforms.RandomHorizontalFlip(), transforms.ToTensor(), transforms.Normalize(mean, std),
                                          PatchGenerator(self.patch_size)])
    test_transform = transforms.Compose([transforms.Resize(self.img_size), transforms.ToTensor(),
                                         transforms.Normalize(mean, std), PatchGenerator(self.patch_size)])

    if self.dataname == 'cifar10':
        trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=train_transform)
        valset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=test_transform)

        testset = torchvision.datasets.ImageFolder(root='./class', transform=test_transform)

    elif self.dataname == 'imagenet':
        pass

    weights = self.make_weights(trainset.targets, len(trainset.classes))  # 가중치 계산
    weights = torch.DoubleTensor(weights)
    sampler = torch.utils.data.sampler.WeightedRandomSampler(weights, len(weights))
    trainloader = DataLoader(trainset, batch_size=self.batch_size, sampler=sampler)
    valloader = DataLoader(valset, batch_size=self.batch_size, shuffle=False)
    testloader = DataLoader(testset, batch_size=self.batch_size, shuffle=False)

    return trainloader, valloader, testloader
```

CODE

```python
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
```

```python
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
```

CODE

```python
latent_vec_dim = args.latent_vec_dim
mlp_hidden_dim = int(latent_vec_dim/2)
num_patches = int((args.img_size * args.img_size) / (args.patch_size * args.patch_size))
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Image Patches
d = patchdata.Flattened2Dpatches(dataname=args.dataname, img_size=args.img_size, patch_size=args.patch_size,
                                 batch_size=args.batch_size)
trainloader, valloader, testloader = d.patchdata()
image_patches, _ = iter(trainloader).next()

# Model
vit = model.VisionTransformer(patch_vec_size=image_patches.size(2), num_patches=image_patches.size(1),
                              latent_vec_dim=latent_vec_dim, num_heads=args.num_heads, mlp_hidden_dim=mlp_hidden_dim,
                              drop_rate=args.drop_rate, num_layers=args.num_layers, num_classes=args.num_classes).to(device)

if args.pretrained == 1:
    vit.load_state_dict(torch.load('./model.pth'))
if args.pretrained == 2:
    vit.load_state_dict(torch.load('./model1.pth'))
if args.mode == 'train':
    # Loss and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(vit.parameters(), lr=args.lr, weight_decay=args.weight_decay)

    #optimizer = torch.optim.SGD(vit.parameters(), lr=args.lr, momentum=0.9)
    #scheduler = optim.lr_scheduler.OneCycleLR(optimizer, max_lr=args.lr, steps_per_epoch=len(trainloader), epochs=args.epochs)

    # Train
    n = len(trainloader)
    best_acc = args.save_acc
    for epoch in range(args.epochs):
        running_loss = 0
        for img, labels in trainloader:
            optimizer.zero_grad()
            outputs, _ = vit(img.to(device))
            loss = criterion(outputs, labels.to(device))
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            #scheduler.step()

        train_loss = running_loss / n
        val_acc, val_loss = test.accuracy(valloader, vit)
        # if epoch % 5 == 0:
        print('[%d] train loss: %.3f, validation loss: %.3f, validation acc %.2f %%' % (epoch, train_loss, val_loss, val_acc))

        if val_acc > best_acc:
            best_acc = val_acc
            # print('[%d] train loss: %.3f, validation acc %.2f - Save the best model' % (epoch, train_loss, val_acc))
            torch.save(vit.state_dict(), './model.pth')

else:
    test_acc, test_loss = test.accuracy(testloader, vit)
    print('test loss: %.3f, test acc %.2f %%' % (test_loss, test_acc))
```