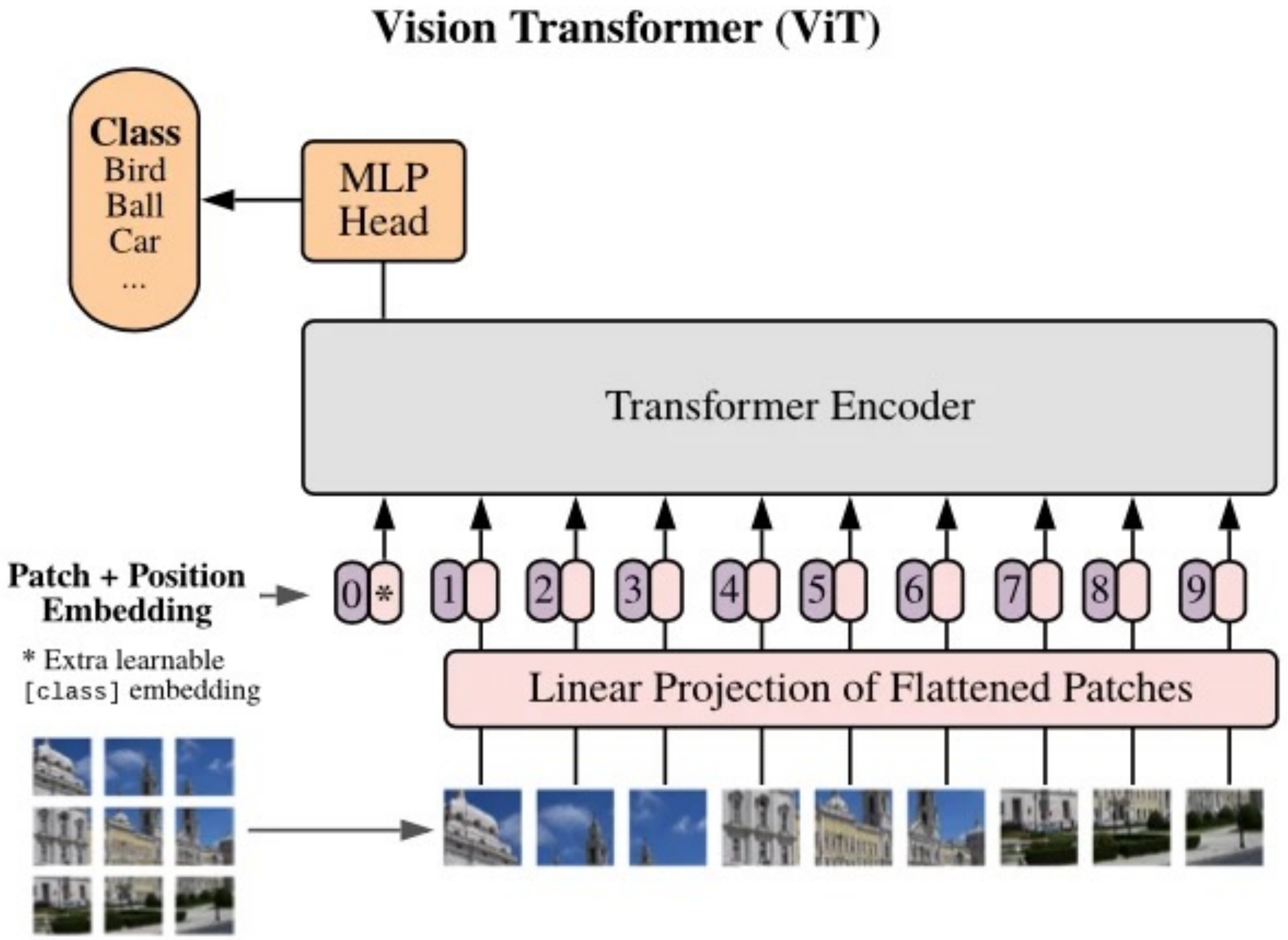
REFERENCED SOLUTION - VIT IMPLEMENTATION PRACTICE

PREPARE THE TOKENS

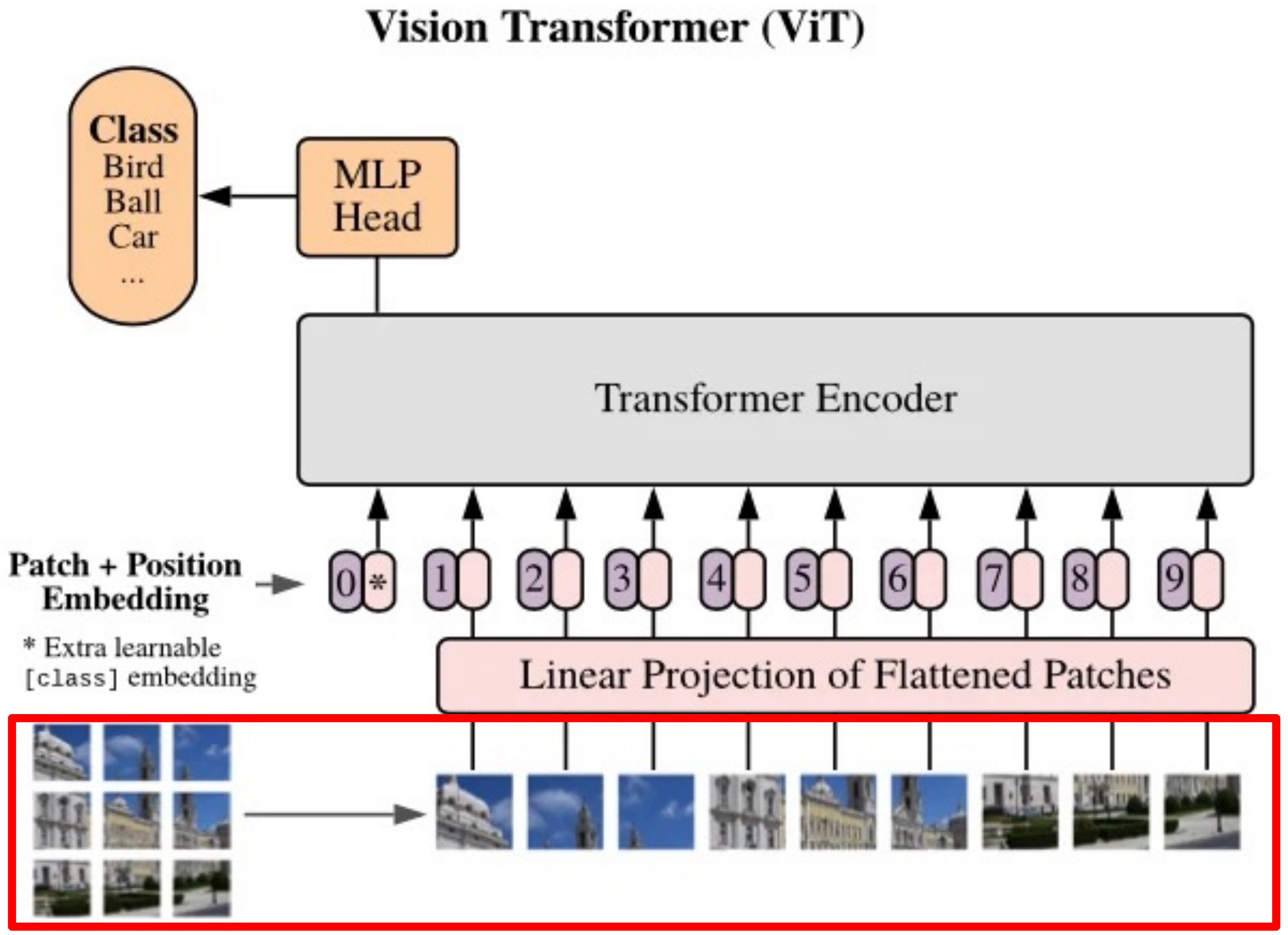



```python
import torch
from torch import nn

from einops import rearrange, repeat
from einops.layers.torch import Rearrange


class Image2Tokens(nn.Module):
    def __init__(self, image_size, dim, in_dim=3, patch_size=16, emb_dropout=0.):
        super().__init__()
        image_height, image_width = image_size
        num_patches = (image_height // patch_size) * (image_width // patch_size)
        patch_dim = in_dim * patch_size * patch_size
        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=patch_size, p2=patch_size),
            nn.Linear(patch_dim, dim),
        )
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)

    def forward(self, img):
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape

        cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]
        return self.dropout(x)
```

```python
tokenizer = Image2Tokens(image_size=(224,224), dim=768)
tokenizer(torch.randn(1,3,224,224)).shape
```

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

PREPARE THE TOKENS

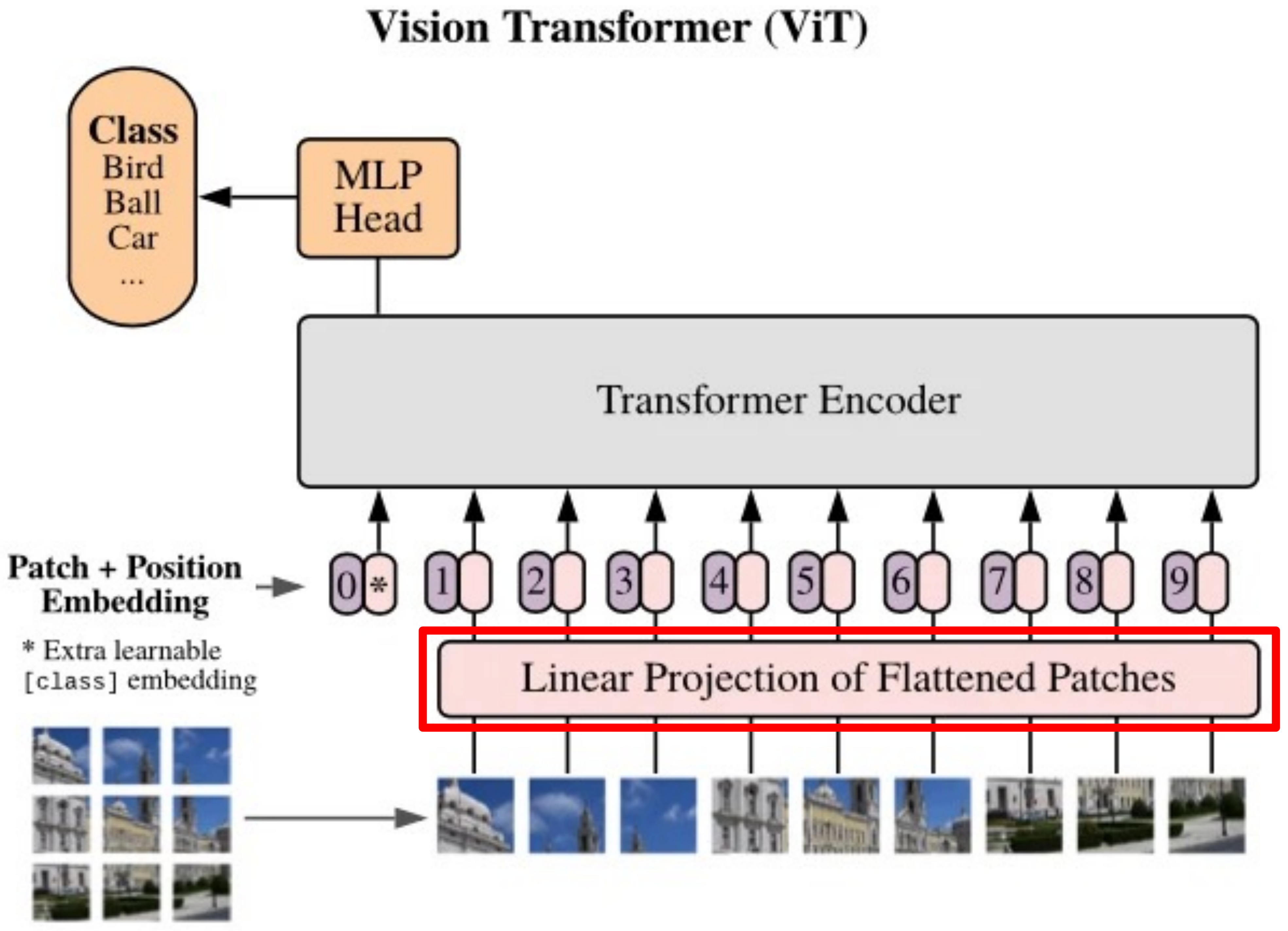



```python
import torch
from torch import nn

from einops import rearrange, repeat
from einops.layers.torch import Rearrange



class Image2Tokens(nn.Module):
    def __init__(self, image_size, dim, in_dim=3, patch_size=16, emb_dropout=0.):
        super().__init__()
        image_height, image_width = image_size
        num_patches = (image_height // patch_size) * (image_width // patch_size)
        patch_dim = in_dim * patch_size * patch_size
        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=patch_size, p2=patch_size),
            nn.Linear(patch_dim, dim),
        )
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)

    def forward(self, img):
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape

        cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]
        return self.dropout(x)
```

```python
tokenizer = Image2Tokens(image_size=(224,224), dim=768)
tokenizer(torch.randn(1,3,224,224)).shape
```

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

PREPARE THE TOKENS



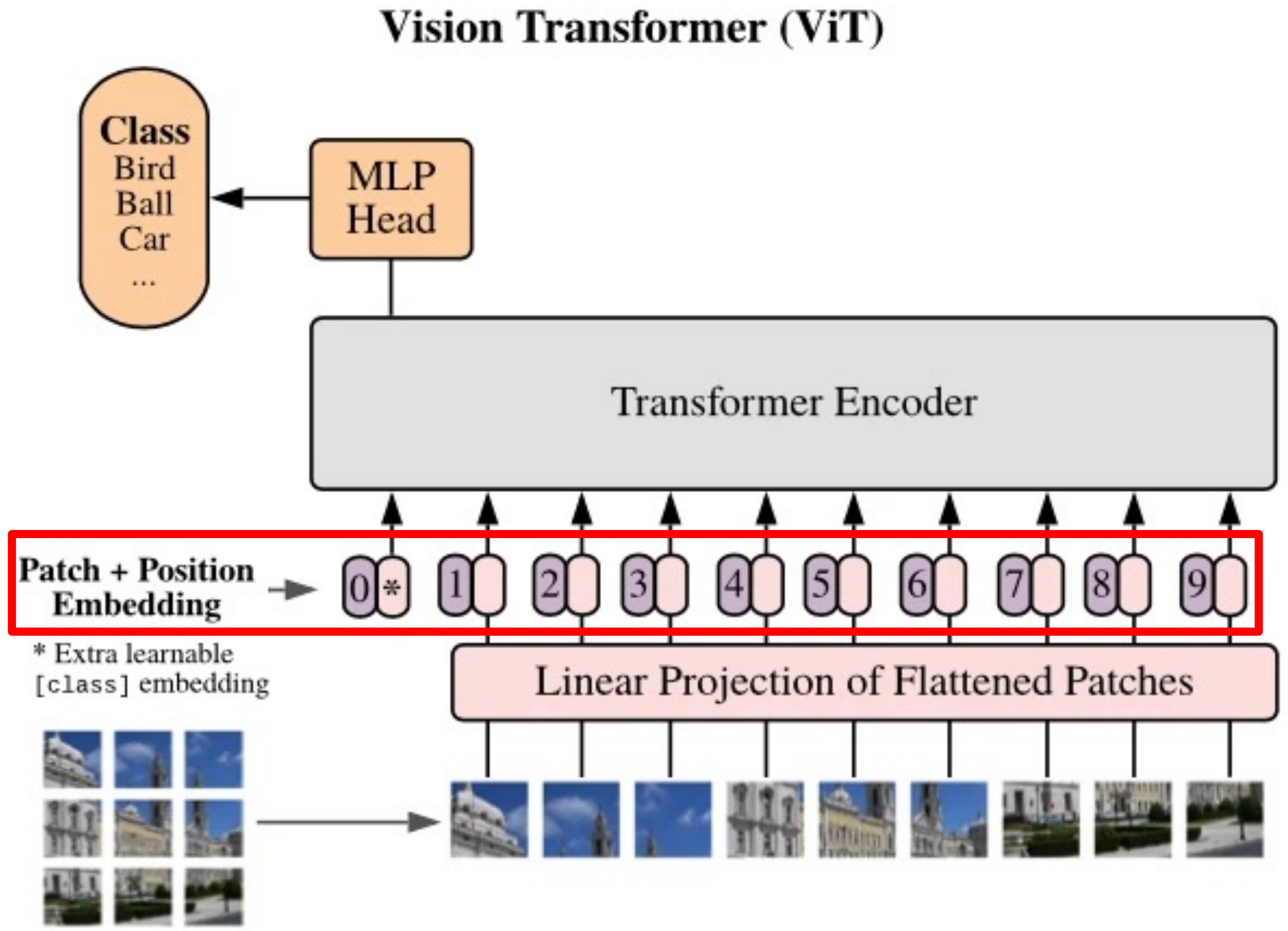
Now the tensor becomes 1D with shape (batch, patches, dim)

```python
import torch
from torch import nn

from einops import rearrange, repeat
from einops.layers.torch import Rearrange


class Image2Tokens(nn.Module):
    def __init__(self, image_size, dim, in_dim=3, patch_size=16, emb_dropout=0.):
        super().__init__()
        image_height, image_width = image_size
        num_patches = (image_height // patch_size) * (image_width // patch_size)
        patch_dim = in_dim * patch_size * patch_size
        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=patch_size, p2=patch_size),
            nn.Linear(patch_dim, dim),
        )
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)

    def forward(self, img):
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape

        cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]
        return self.dropout(x)
```

```python
tokenizer = Image2Tokens(image_size=(224,224), dim=768)
tokenizer(torch.randn(1,3,224,224)).shape
```

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

PREPARE THE TOKENS



Optional Dropout could be applied after embeddings ☺

```python
import torch
from torch import nn

from einops import rearrange, repeat
from einops.layers.torch import Rearrange



class Image2Tokens(nn.Module):
    def __init__(self, image_size, dim, in_dim=3, patch_size=16, emb_dropout=0.):
        super().__init__()
        image_height, image_width = image_size
        num_patches = (image_height // patch_size) * (image_width // patch_size)
        patch_dim = in_dim * patch_size * patch_size
        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=patch_size, p2=patch_size),
            nn.Linear(patch_dim, dim),
        )
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)

    def forward(self, img):
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape

        cls_tokens = repeat(self.cls_token, '() n d -> b n d', b=b)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]
        return self.dropout(x)
```

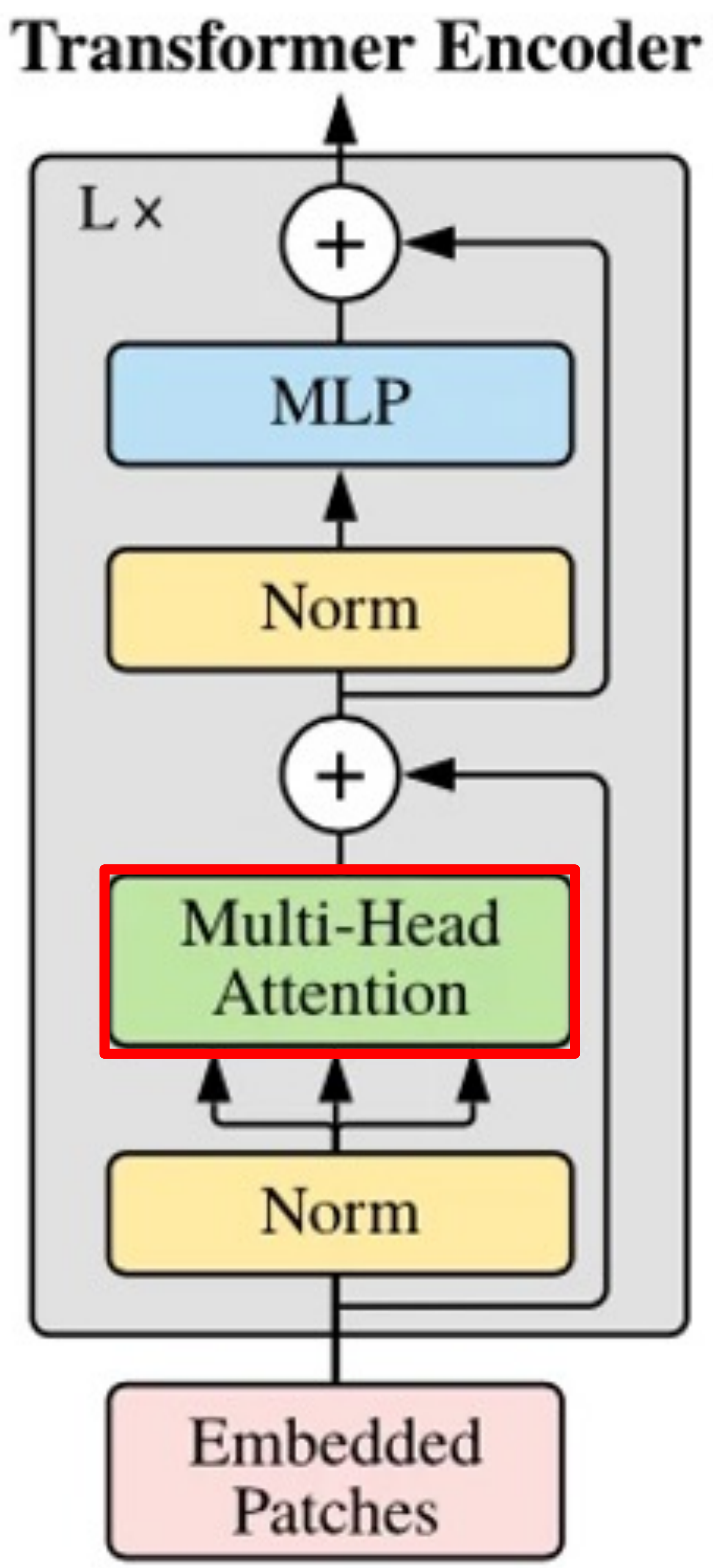
```python
tokenizer = Image2Tokens(image_size=(224,224), dim=768)
tokenizer(torch.randn(1,3,224,224)).shape
```

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

MULTI-HEAD SELF-ATTENTION

$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

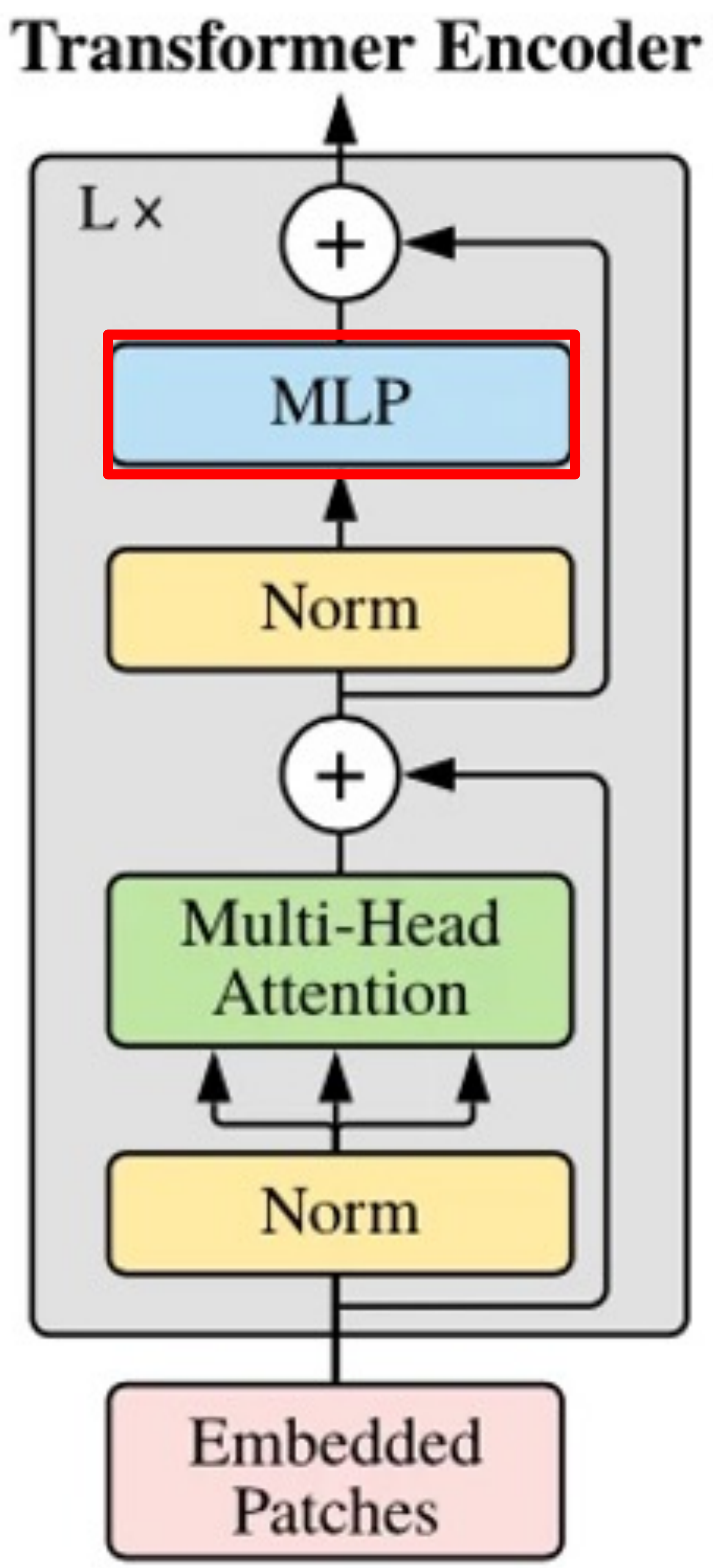



```python
class Attention(nn.Module):
    def __init__(self, dim, heads=8, dropout=0.):
        super().__init__()
        self.heads = heads
        self.scale = (dim // heads) ** -0.5
        self.attend = nn.Softmax(dim=-1)
        self.to_qkv = nn.Linear(dim, dim*3)
        self.to_out = nn.Sequential(
            nn.Linear(dim, dim),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        qkv = self.to_qkv(x).chunk(3, dim = -1)
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h=self.heads), qkv)
        dots = (q @ k.transpose(-1, -2)) * self.scale
        attn = self.attend(dots)
        out = attn @ v
        out = rearrange(out, 'b h n d -> b n (h d)')
        return self.to_out(out)
```

```python
attn_op = Attention(dim=768)
attn_op(torch.randn(1,197,768)).shape
```

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

FEEDFORWARD NETWORK (FFN)



$$\boxed{FFN = w_2 GELU(w_1 x + b_1) + b_2}$$

```
class FeedForwardNetwork(nn.Module):
    def __init__(self, dim, dropout=0.):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, dim*4),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(dim*4, dim),
            nn.Dropout(dropout)
        )

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

Note this is the only nonlinear operator in transformer ☺!

```
ffn = FeedForwardNetwork(768)
ffn(torch.randn(1,197,768)).shape
```

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

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

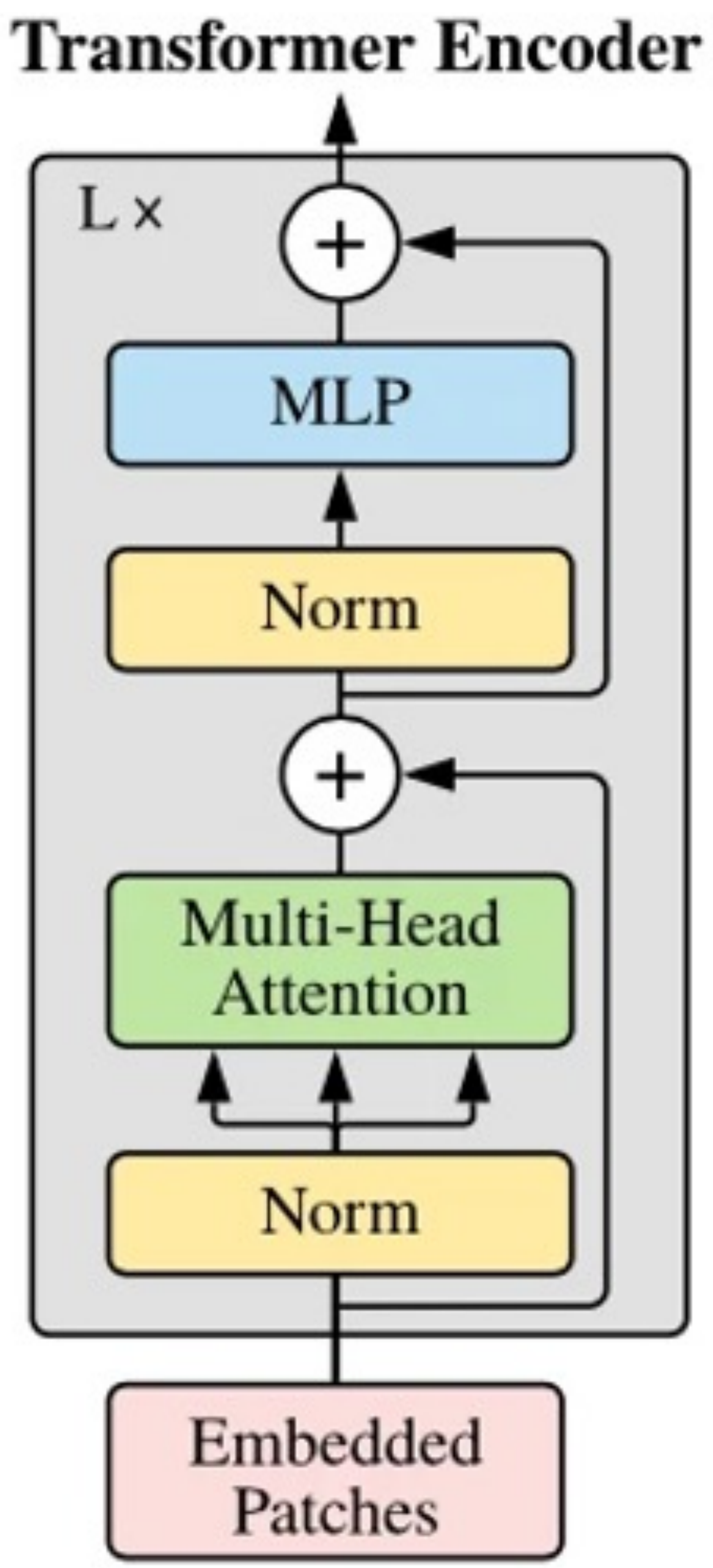
Table 1: Details of Vision Transformer model variants.

THE VISION TRANSFORMER

Putting all together: Encoder

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

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



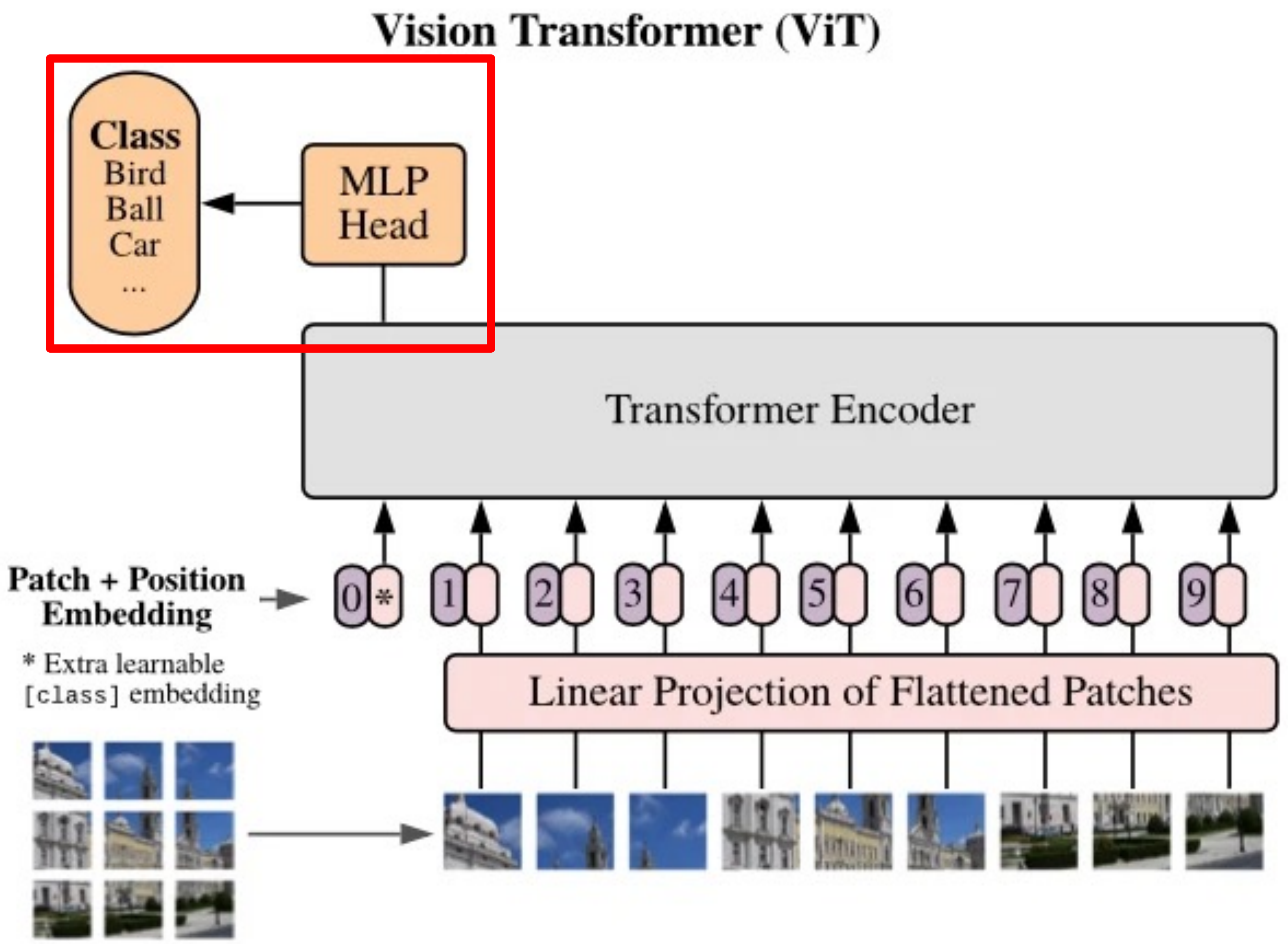

```
class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)


class Transformer(nn.Module):
    def __init__(self, layers, dim, heads=8, dropout=0.):
        super().__init__()
        self.layers = nn.ModuleList([])
        for _ in range(layers):
            self.layers.append(nn.ModuleList([
                PreNorm(dim, Attention(dim, heads=heads, dropout=dropout)),
                PreNorm(dim, FeedForwardNetwork(dim, dropout=dropout))
            ]))
    def forward(self, x):
        for attn, ff in self.layers:
            x = attn(x) + x
            x = ff(x) + x
        return x
```

```
model = Transformer(layers=12, dim=768)
model(torch.randn(1,197,768)).shape
```

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

THE VISION TRANSFORMER

Putting all together: Complete Modeling




```python
class ViT(nn.Module):
    def __init__(self, layers, dim, heads, image_size, num_classes, patch_size=16, in_dim=3, dropout=0., emb_dropout=0.):
        super().__init__()
        self.tokenizer = Image2Tokens(image_size=image_size, dim=dim, in_dim=in_dim, patch_size=patch_size, emb_dropout=emb_dropout)
        self.transformer = Transformer(layers=layers, dim=dim, heads=heads, dropout=dropout)
        self.classifier = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, num_classes)
        )

    def forward(self, img):
        out = self.tokenizer(img)
        out = self.transformer(out)
        out = out[:, 0]
        return self.classifier(out)
```

Tip: place the cls-token at the index 0 to get it easily for classifier

```python
model = ViT(layers=12, dim=768, heads=12, image_size=(224,224), num_classes=1000)
model(torch.randn(1,3,224,224)).shape
```

```
torch.Size([1, 1000])
```

```python
def num_of_parameters(model):
    params = 0
    for i in model.parameters():
        params += i.numel()
    return params
```

```python
vit_base = ViT(layers=12, dim=768, heads=12, image_size=(256,256), num_classes=1000)
vit_large = ViT(layers=24, dim=1024, heads=16, image_size=(256,256), num_classes=1000)
vit_huge = ViT(layers=32, dim=1280, heads=16, image_size=(256,256), num_classes=1000)

print(num_of_parameters(vit_base))
print(num_of_parameters(vit_large))
print(num_of_parameters(vit_huge))
```

```
86613736
304388072
632276200
```

Sanity check on model parameters

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-Large(L) is actually with 304M params

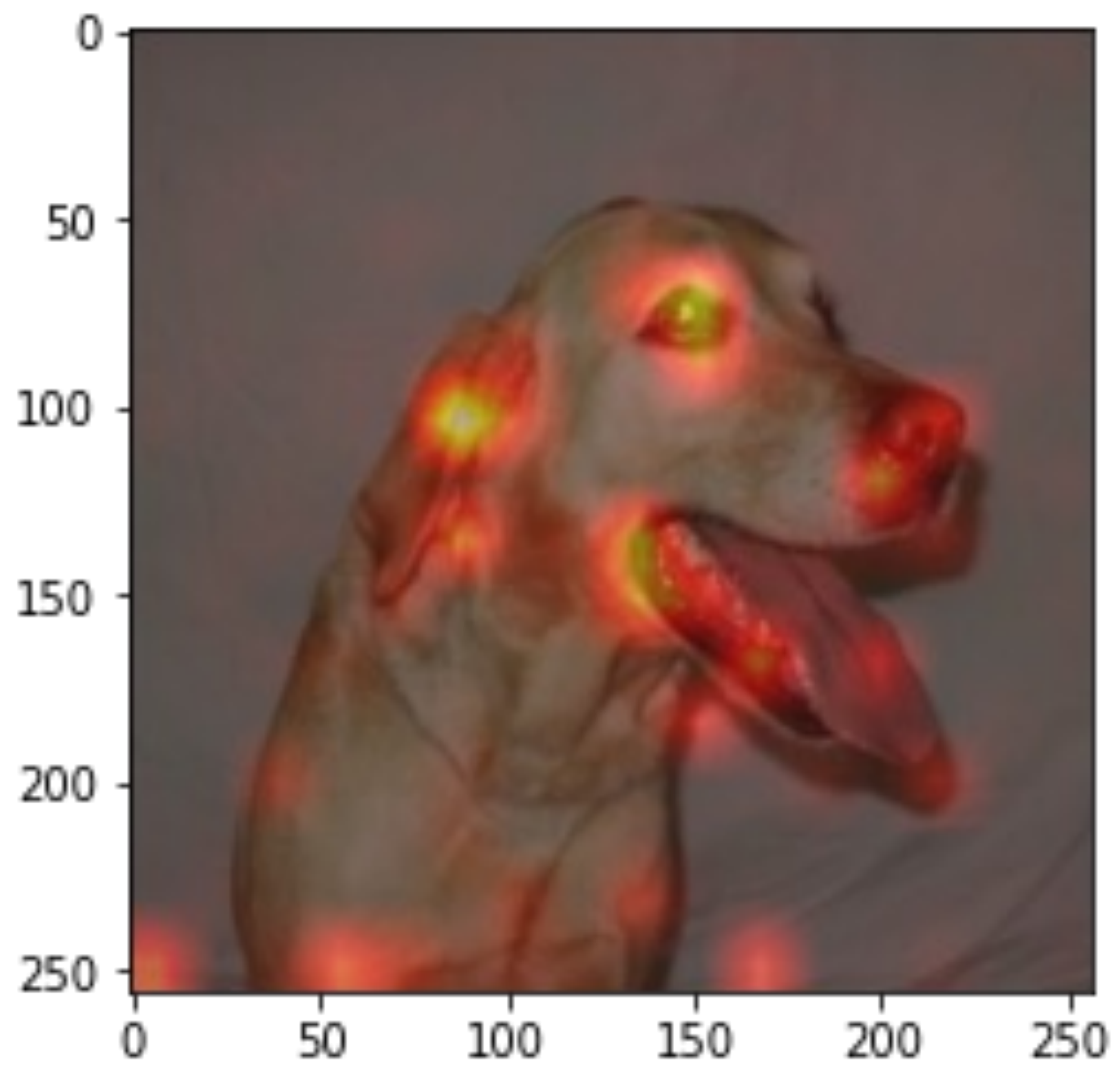
Chen, Xinlei, Saining Xie, and Kaiming He. "An empirical study of training self-supervised vision transformers." *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021.

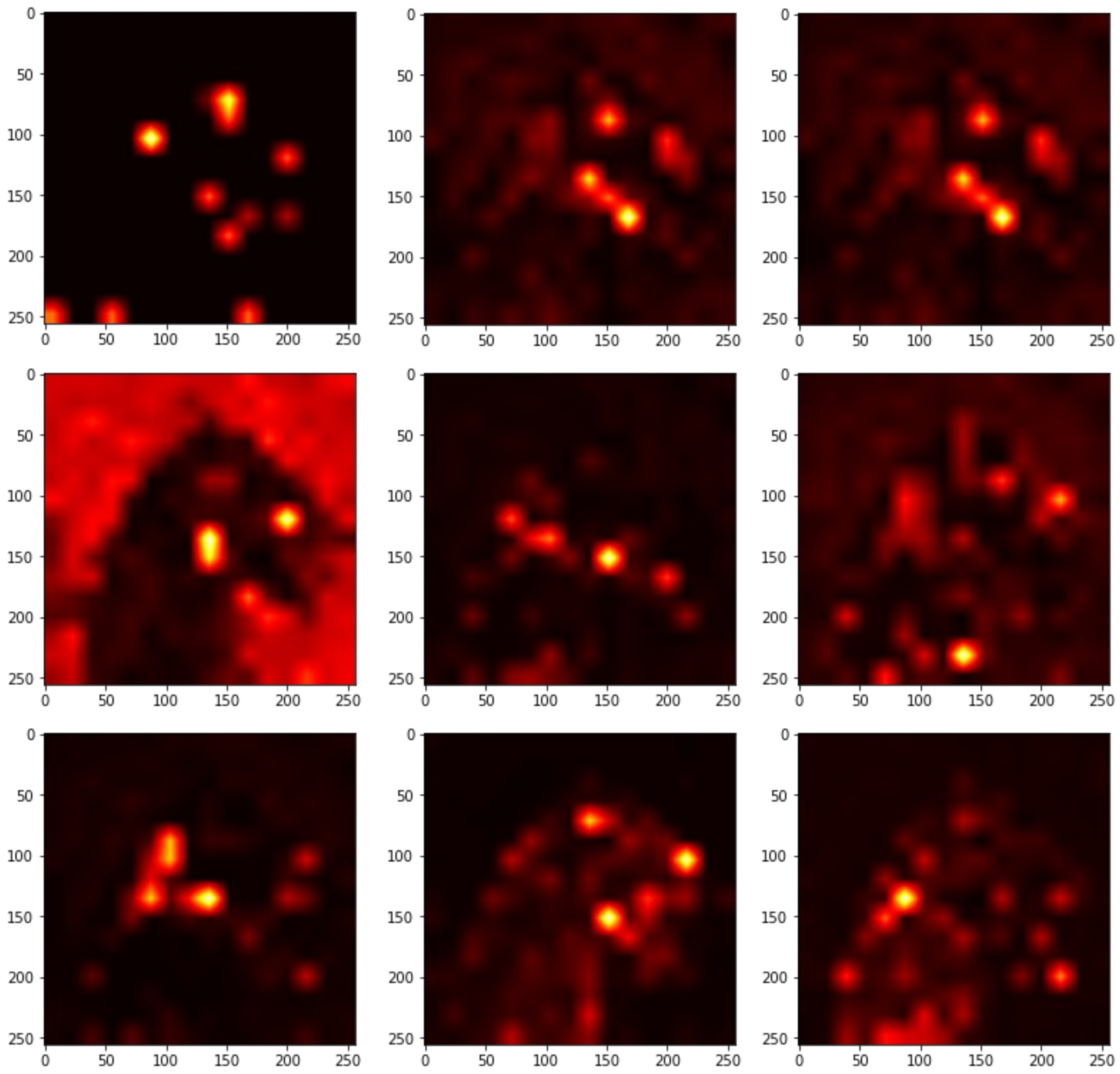
framework	model	params
linear probing:		
iGPT [9]	iGPT-L	1362M
iGPT [9]	iGPT-XL	6801M
MoCo v3	ViT-B	86M
MoCo v3	ViT-L	304M
MoCo v3	ViT-H	632M

NVIDIA

CATS AND DOGS CLASSIFICATION

- Download the training set at https://www.kaggle.com/competitions/dogs-vs-cats-redux-kernels-edition/data?select=train.zip
- Consider this dataset is small, the ViT used in this case is configured to 9 layers and 192 dimensions with 12 heads.
- Training from Scratch: **77.51%**
- Pretraining from ImageNet: **98.61%**





- A simple way for visualization is to retrieve the attention weightings computed from the dot-product attention between query and keys. The information flows from each token to the class token can indicate the spatial locations the model focus on.


NVIDIA