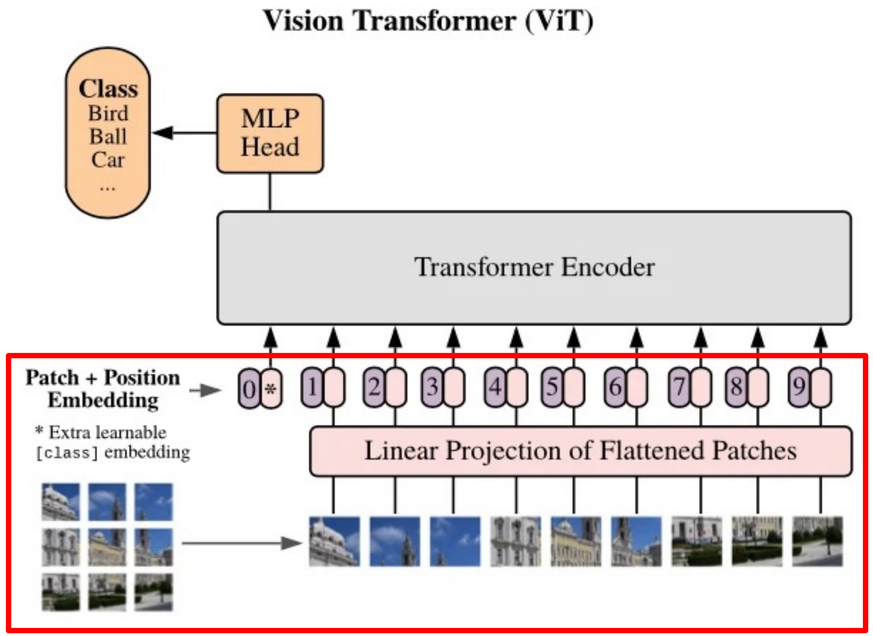

NVIDIA
VIT IMPLEMENTATION PRACTICE

TASK 1 - PREPARE THE TOKENS

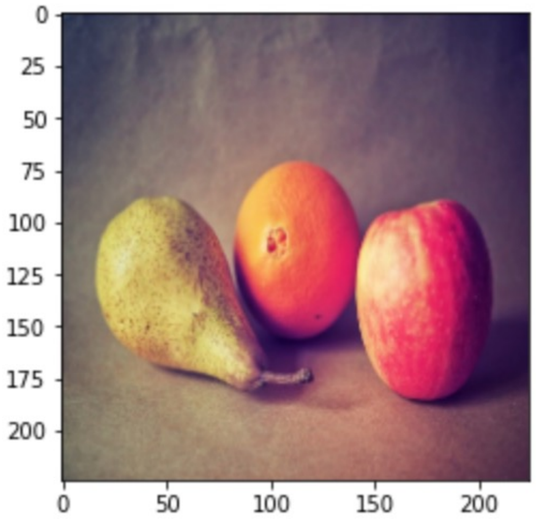



```python
class Image2Tokens(nn.Module):
    def __init__(self, image_size, dim, in_dim=3, patch_size=16, emb_dropout=0.):
        super().__init__()
        image_height, image_width = image_size
        num_patches = (image_height // patch_size) * (image_width // patch_size)
        self.to_patch_embedding = nn.Sequential(
            Rearrange('#! >>> fill the correct einops statement here for prepare patches', p1=patch_size, p2=patch_size),
            #! >>> fill the embedding layer declaration here (hint: transform from `patch_size * patch_size * in_dim` to `dim`
        )
        self.pos_embedding = #! >>> fill the modules declaration here
        self.cls_token = #! >>> fill the modules declaration here
        self.dropout = nn.Dropout(emb_dropout)

    def forward(self, img):
        x = self.to_patch_embedding(img)

        #! >>> fill the neccessary codes here
        # Steps:
        #      1. Split the images into (patch_size x patch_size) non-overlapping patches
        #      2. Apply a linear transformation to make the 2D patches into 1D vectors in given hidden size: `dim`
        #      3. Append a class-token at the very beginning (index 0)
        #      4. Add a learnable embeddings to every tokens (including the added class-token)

        return self.dropout(x)
```

SIDENOTE: PREPARE THE IMAGE PATCHES (1)

```
img = cv2.resize(cv2.imread('fruit.jpg'), (224,224))[..., ::-1].copy()
```

```
plt.imshow(img)
plt.show()
```



Due to the memory alignment, directly reshape might not work

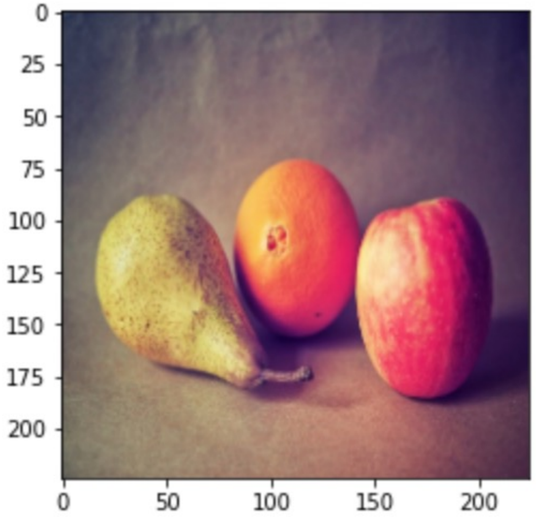
```
patches = img.reshape(16, 16, (224//16)*(224//16), 3)
print(patches.shape)

_, arr = plt.subplots(14, 14, figsize=(10, 10))
for i in range(patches.shape[2]):
    arr[i//14, i%14].imshow(patches[..., i, :])
    arr[i//14, i%14].axis('off')
plt.show()
```

```
(16, 16, 196, 3)
```

SIDENOTE: PREPARE THE IMAGE PATCHES (2)

```
img = cv2.resize(cv2.imread('fruit.jpg'), (224,224))[..., ::-1].copy()
```

```
plt.imshow(img)
plt.show()
```



Same for torch tensor

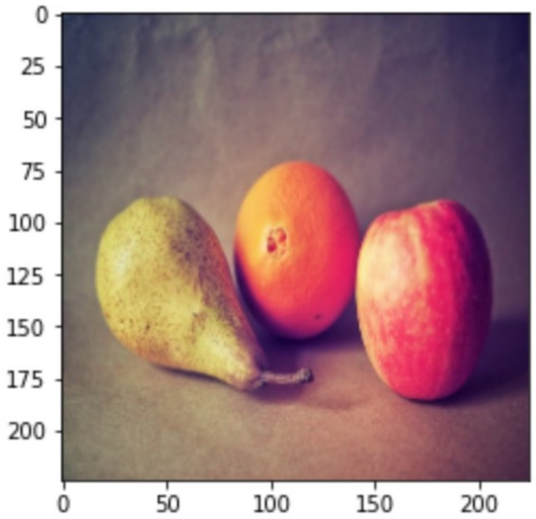
```
torch_img = torch.from_numpy(img)
torch_img = torch_img.permute(2,0,1).unsqueeze(0)
print(torch_img.shape)

patches = torch_img.reshape(1, 3, 16, 16, (224//16)*(224//16))
print(patches.shape)

_, arr = plt.subplots(14, 14, figsize=(10, 10))
for i in range(patches.shape[4]):
    arr[i//14, i%14].imshow(patches[0, ..., i].permute(1,2,0))
    arr[i//14, i%14].axis('off')
plt.show()
```

```
torch.Size([1, 3, 224, 224])
torch.Size([1, 3, 16, 16, 196])
```

SIDENOTE: PREPARE THE IMAGE PATCHES (3)

```
img = cv2.resize(cv2.imread('fruit.jpg'), (224,224))[..., ::-1].copy()
```

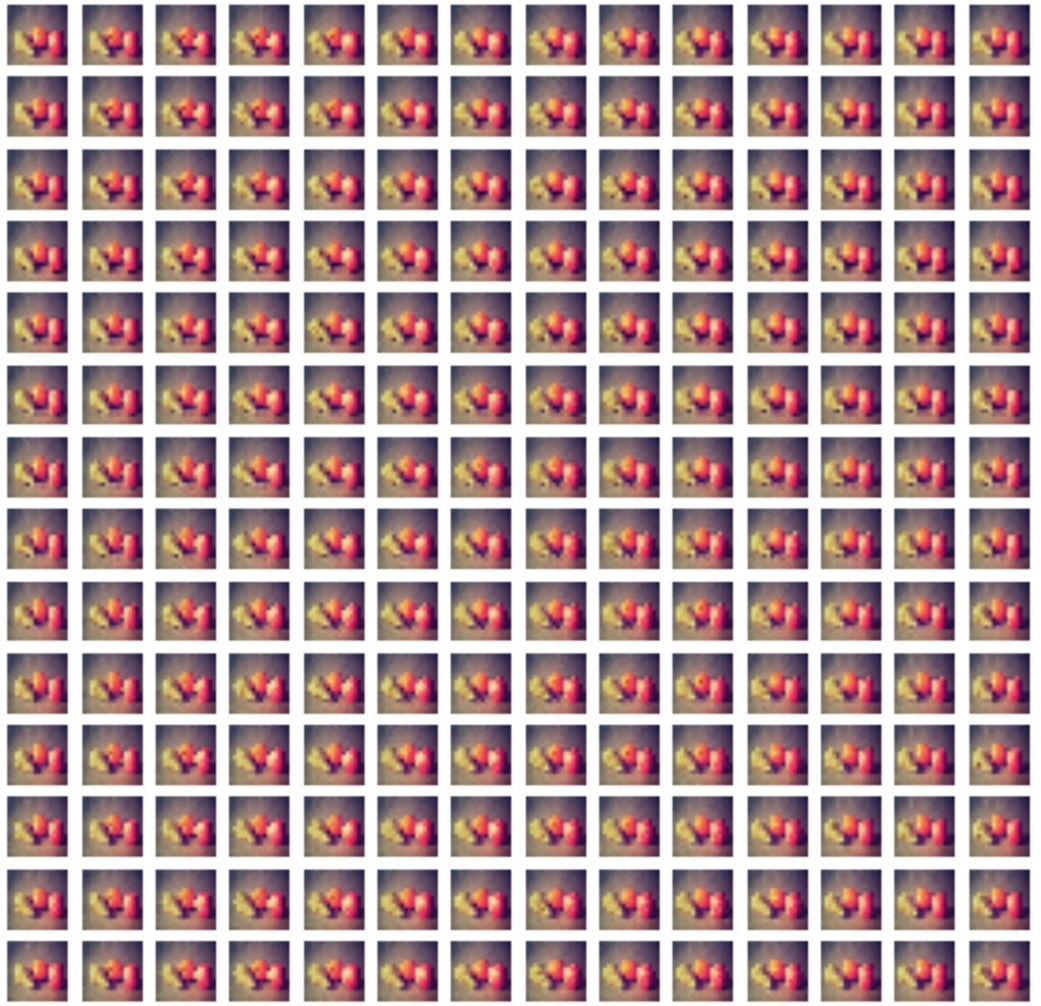
```
plt.imshow(img)
plt.show()
```



Another attemp, no luck ☹

```
patches = img.reshape(16, (224//16), 16, (224//16), 3)
patches = patches.transpose(0,2,1,3,4)
patches = patches.reshape(patches.shape[0], patches.shape[1], -1, 3)
print(patches.shape)

_, arr = plt.subplots(14, 14, figsize=(10, 10))
for i in range(patches.shape[2]):
    arr[i//14, i%14].imshow(patches[..., i, :])
    arr[i//14, i%14].axis('off')
plt.show()
```

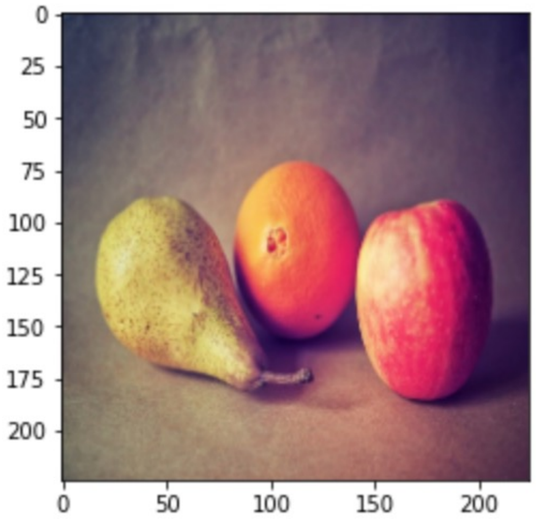
```
(16, 16, 196, 3)
```



SIDENOTE: PREPARE THE IMAGE PATCHES (4)

```
img = cv2.resize(cv2.imread('fruit.jpg'), (224,224))[..., ::-1].copy()
```

```
plt.imshow(img)
plt.show()
```



Finally! ☺

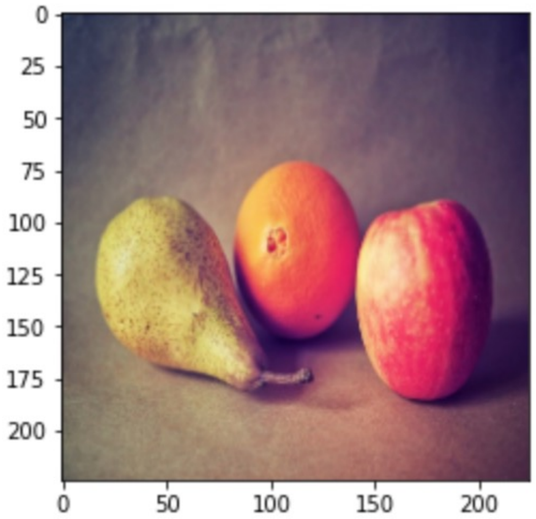
```
patches = img.reshape((224//16), 16, (224//16), 16, 3)
patches = patches.transpose(1,3,0,2,4)
patches = patches.reshape(patches.shape[0], patches.shape[1], -1, 3)
print(patches.shape)

_, arr = plt.subplots(14, 14, figsize=(10, 10))
for i in range(patches.shape[2]):
    arr[i//14, i%14].imshow(patches[..., i, :])
    arr[i//14, i%14].axis('off')
plt.show()
```

```
(16, 16, 196, 3)
```

SIDENOTE: PREPARE THE IMAGE PATCHES (5)

```
img = cv2.resize(cv2.imread('fruit.jpg'), (224,224))[..., ::-1].copy()
```

```
plt.imshow(img)
plt.show()
```



In PyTorch, it is recommended to use high-level einops without headache.

```
from einops.layers.torch import Rearrange

torch_img = torch.from_numpy(img)
torch_img = torch_img.permute(2,0,1).unsqueeze(0)
print(torch_img.shape)

patches = Rearrange('b c (h p1) (w p2) -> b c p1 p2 (h w)', p1=16, p2=16)(torch_img)
print(patches.shape)

_, arr = plt.subplots(14, 14, figsize=(10, 10))
for i in range(patches.shape[4]):
    arr[i//14, i%14].imshow(patches[0, ..., i].permute(1,2,0))
    arr[i//14, i%14].axis('off')
plt.show()
```

```
torch.Size([1, 3, 224, 224])
torch.Size([1, 3, 16, 16, 196])
```

einops

Readable and reliable operations
for deep learning

```
x.transpose(0, 2, 3, 1)
```

becomes

```
rearrange(x, 'b c h w -> b h w c')
```

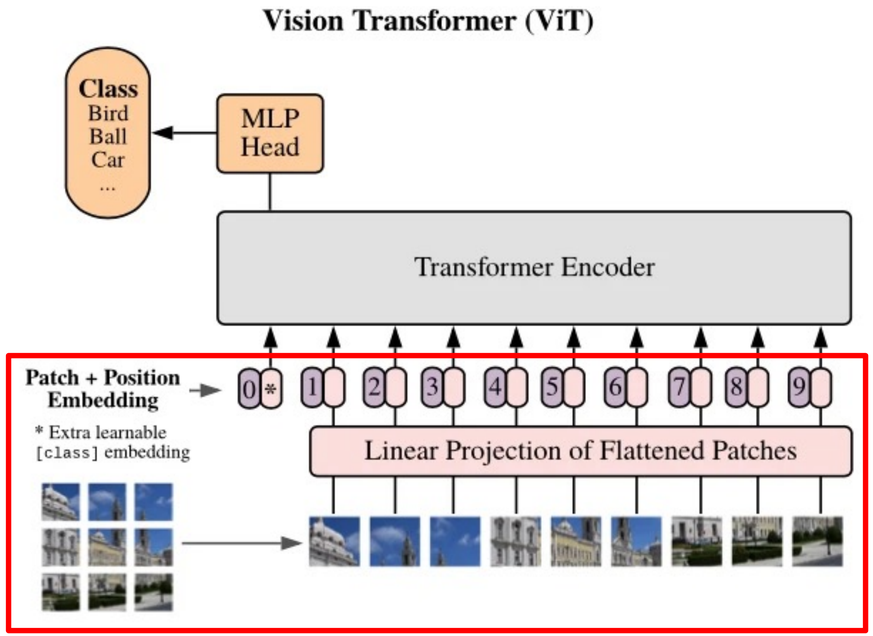
Works with Pytorch, TF, Jax and numpy

```python
def gram_matrix_old(y):
    (b, ch, h, w) = y.size()
    features = y.view(b, ch, w * h)
    features_t = features.transpose(1, 2)
    gram = features.bmm(features_t) / (ch * h * w)
    return gram
```

```python
def gram_matrix_new(y):
    b, ch, h, w = y.shape
    return torch.einsum('bchw,bdhw->bcd', [y, y]) / (h * w)
```

- The basic einops (i.e., einsum) is built-in in PyTorch.
- https://github.com/arogozhnikov/einops implements wider set of einops

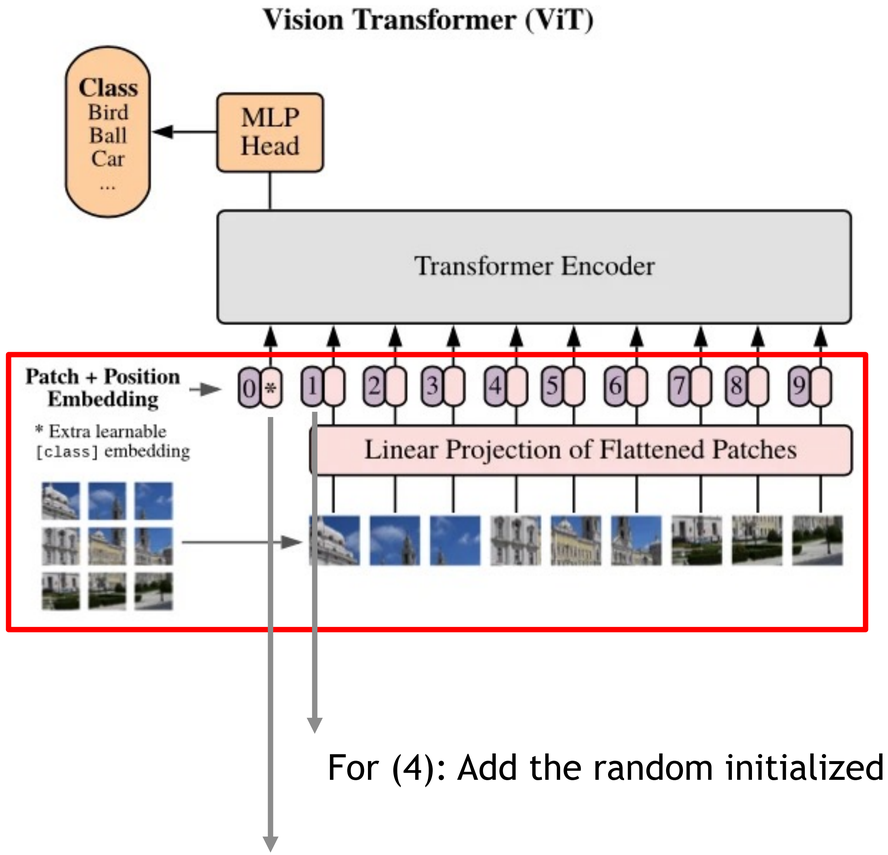
TASK 1 - PREPARE THE TOKENS




```python
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)
        self.to_patch_embedding = nn.Sequential(
            Rearrange('#! >>> fill the correct einops statement here for prepare patches', p1=patch_size, p2=patch_size),
            #! >>> fill the embedding layer declaration here (hint: transform from `patch_size * patch_size * in_dim` to `dim`
        )
        self.pos_embedding = #! >>> fill the modules declaration here
        self.cls_token = #! >>> fill the modules declaration here
        self.dropout = nn.Dropout(emb_dropout)

    def forward(self, img):
        x = self.to_patch_embedding(img)

        #! >>> fill the neccessary codes here
        # Steps:
        #      1. Split the images into (patch_size x patch_size) non-overlapping patches
        #      2. Apply a linear transformation to make the 2D patches into 1D vectors in given hidden size: `dim`
        #      3. Append a class-token at the very beginning (index 0)
        #      4. Add a learnable embeddings to every tokens (including the added class-token)

        return self.dropout(x)
```

Hint: ‘b c (h p1) (w p2) -> ?’

TASK 1 - PREPARE THE TOKENS



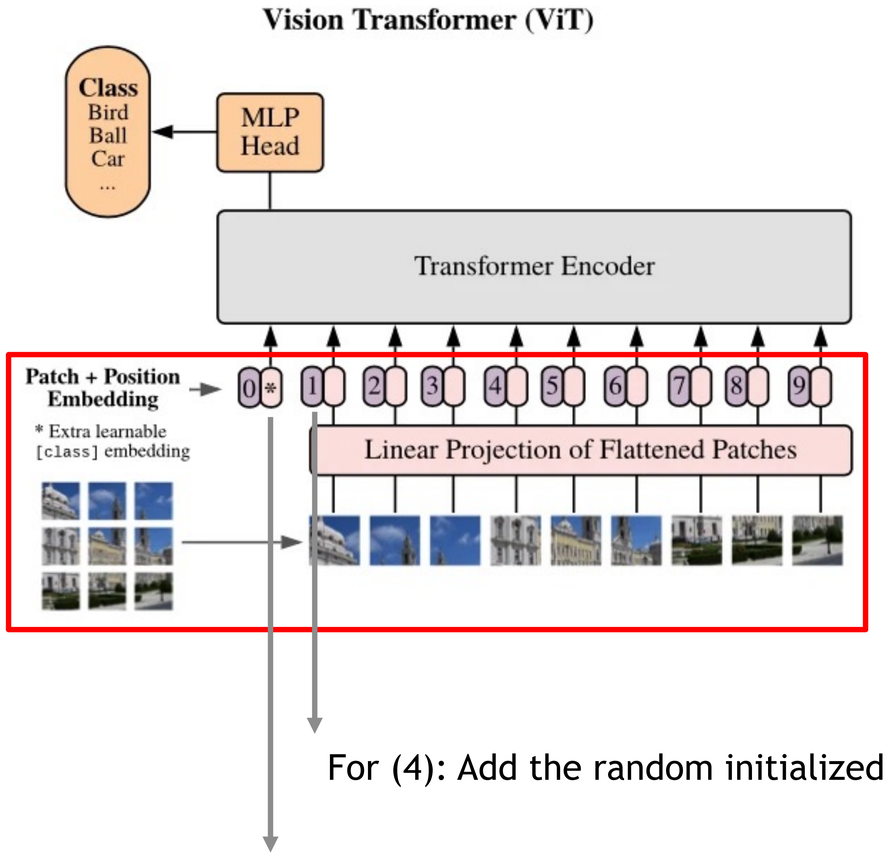

```python
class Image2Tokens(nn.Module):
    def __init__(self, image_size, dim, in_dim=3, patch_size=16, emb_dropout=0.):
        super().__init__()
        image_height, image_width = image_size
        num_patches = (image_height // patch_size) * (image_width // patch_size)
        self.to_patch_embedding = nn.Sequential(
            Rearrange('#! >>> fill the correct einops statement here for prepare patches', p1=patch_size, p2=patch_size),
            #! >>> fill the embedding layer declaration here (hint: transform from `patch_size * patch_size * in_dim` to `dim`
        )
        self.pos_embedding = #! >>> fill the modules declaration here
        self.cls_token = #! >>> fill the modules declaration here
        self.dropout = nn.Dropout(emb_dropout)

    def forward(self, img):
        x = self.to_patch_embedding(img)

        #! >>> fill the neccessary codes here
        # Steps:
        #       1. Split the images into (patch_size x patch_size) non-overlapping patches
        #       2. Apply a linear transformation to make the 2D patches into 1D vectors in given hidden size: `dim`
        #       3. Append a class-token at the very beginning (index 0)
        #       4. Add a learnable embeddings to every tokens (including the added class-token)

        return self.dropout(x)
```

Hint: ‘b c (h p1) (w p2) -> ?’

For (4): Add the random initialized leanable vectors for each individual token

For (3): Prepend a random initialized learnable class-token at beginning of tokens

TASK 1 - PREPARE THE TOKENS




```python
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)
        self.to_patch_embedding = nn.Sequential(
            Rearrange('#! >>> fill the correct einops statement here for prepare patches', p1=patch_size, p2=patch_size),
            #! >>> fill the embedding layer declaration here (hint: transform from `patch_size * patch_size * in_dim` to `dim`
        )
        self.pos_embedding = #! >>> fill the modules declaration here
        self.cls_token = #! >>> fill the modules declaration here
        self.dropout = nn.Dropout(emb_dropout)

    def forward(self, img):
        x = self.to_patch_embedding(img)

        #! >>> fill the neccessary codes here
        # Steps:
        #       1. Split the images into (patch_size x patch_size) non-overlapping patches
        #       2. Apply a linear transformation to make the 2D patches into 1D vectors in given hidden size: `dim`
        #       3. Append a class-token at the very beginning (index 0)
        #       4. Add a learnable embeddings to every tokens (including the added class-token)

        return self.dropout(x)
```

Hint: ‘b c (h p1) (w p2) -> ?’

For (4): Add the random initialized leanable vectors for each individual token

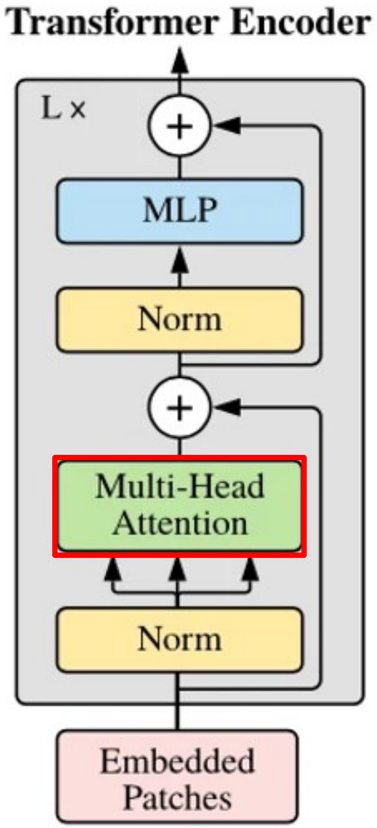
For (3): Prepend a random initialized learnable class-token at beginning of tokens

Note the input may come in batch, repeat the class-token and embedding for each incoming input.

TASK 2 - MULTI-HEAD SELF-ATTENTION

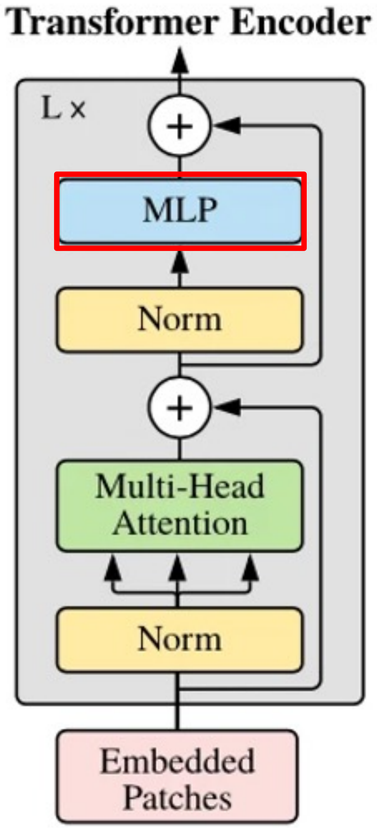
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$




```python
class Attention(nn.Module):
    def __init__(self, dim, heads=8, dropout=0.):
        super().__init__()
        self.heads = heads
        self.scale = #! >>> fill the correct scale here
        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)

        #! >>> fill the implementation to complete the self-attention implementations
        # Hint: the q,k,v are already in multi-headed, with the shape: (batch, heads, num_tokens, dim).

        out = rearrange(out, 'b h n d -> b n (h d)')
        return self.to_out(out)
```

TASK 3 - FEEDFORWARD NETWORK (FFN)



$$FFN = w_2 GELU(w_1 x + b_1) + b_2$$

```python
class FeedForwardNetwork(nn.Module):
    def __init__(self, dim, dropout=0.):
        super().__init__()
        self.net = nn.Sequential(
            #! >>> fill the neccessary modules to complete the FFN module, note the w1 transforms the dim to 4*dim,
            #       and w2 transforms the intermidiate output from 4*dim back to dim.
        )

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

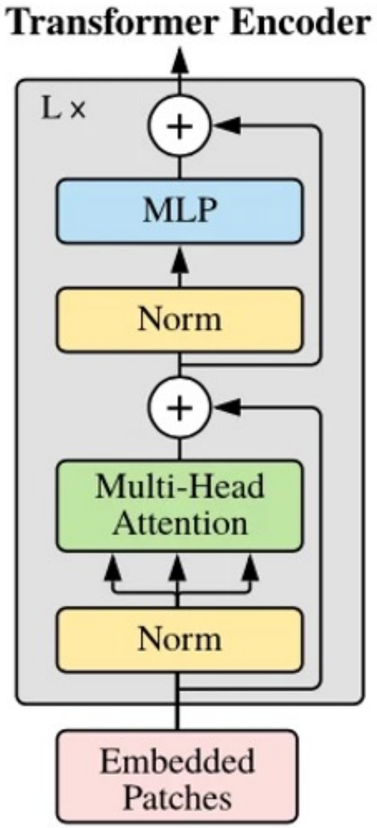
Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Table 1: Details of Vision Transformer model variants.

TASK 4 - THE VISION TRANSFORMER

Putting all together: Encoder

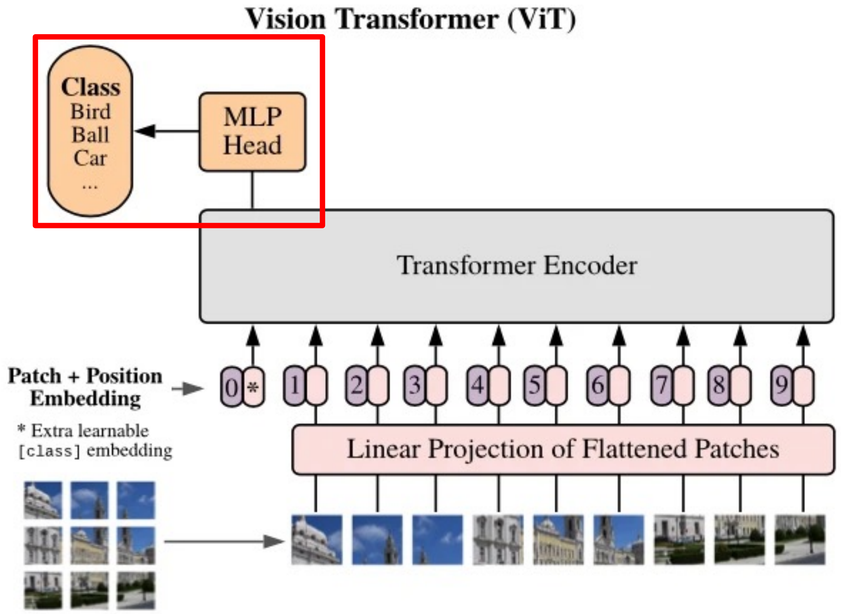
$$
\begin{aligned}
\mathbf{z}'_\ell &= \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1}, && \ell = 1 \dots L \\
\mathbf{z}_\ell &= \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell, && \ell = 1 \dots L
\end{aligned}
$$




```python
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):
        #! >>> Implement the forward flow here, all the neccessary modules are already declared.
        # Remember the return the result :)
```

TASK 5 - 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(#! >>> using the arguments to call the implemented components in previous cells
        self.transformer = Transformer(#! >>> using the arguments to call the implemented components in previous cells
        self.classifier = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, num_classes)
        )

    def forward(self, img):
        #! >>> Implement the forward flow here
        # Note the classifier should accept only class-token, not all the tokens.
        return self.classifier(out)
```


NVIDIA