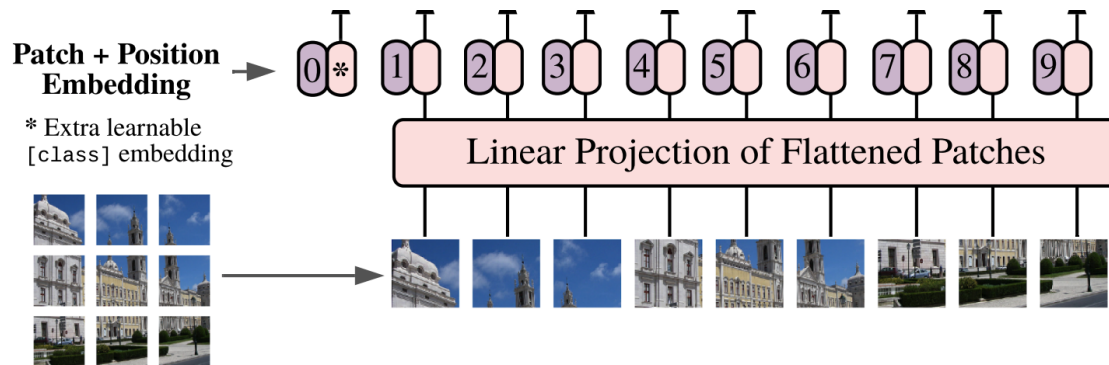


2-3 InputLayer



step1 patch分割



224*224の画像を16*16個もしくは14*14個のパッチに分割します。（画像は3*3個）

※Transformerに倣うと一枚の画像を16*16個の単語として扱います。

入力次元

$x \in \mathbb{R}^{H \times W \times C}$: 入力画像

$224 \times 224 \times 3$

↓

↓

$x_p \in \mathbb{R}^{N_p \times (P^2 \cdot C)}$: パッチ

$16 \times 16 \times (14^2 \cdot 3)$

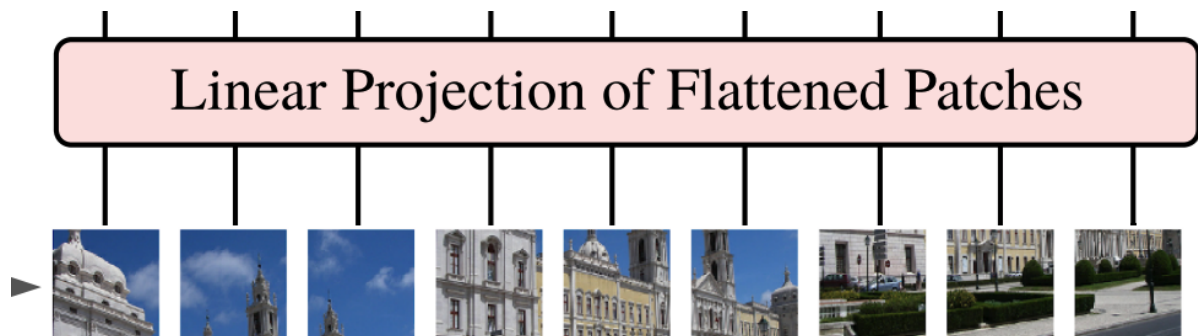
N_p : パッチ数 P : ピクセル数

C : チャンネル

$(P^2 \cdot C)$: パッチのベクトルサイズ

一般的に入力画像は前処理で H と W は224,224になり、正方形の画像として扱います。

step2 flatten (平坦化)



3次元テンソルのpatchを1次元のベクトルに変換します。

入力次元(1つのパッチに対して)

$$14^2 \cdot 3$$

↓

$$1 \times 588$$

step3 Embedding (埋め込み処理)

Patchのベクトルよりも**より良いベクトル**を得るための埋め込みを実施します。

1層の線形層で埋め込み（重みの乗算）を行い、各パッチを連結します。

埋め込みの数式表現は以下になります。

$$[x_p^1 E; x_p^2 E; \cdots x_p^{N_p} E] \in \mathbb{R}^{N_p \times D}$$

$[\cdot]$ パッチの結合

$$x_p^i \in \mathbb{R}^{(P^2 \cdot C)} \quad i = (1 \sim N_p) : i\text{番目のパッチのベクトル}$$

$$E \in \mathbb{R}^{(P^2 \cdot C) \times D} : \text{埋め込み(Embedding)}$$

$(P^2 \cdot C)$: 埋め込み前のパッチのベクトルサイズ

D : 埋め込み後のベクトルの長さ

step4 class token (クラストークン)

$$x_{class} \in \mathbb{R}^D : \text{クラストークン}$$

画像全体の情報を凝縮したベクトル。

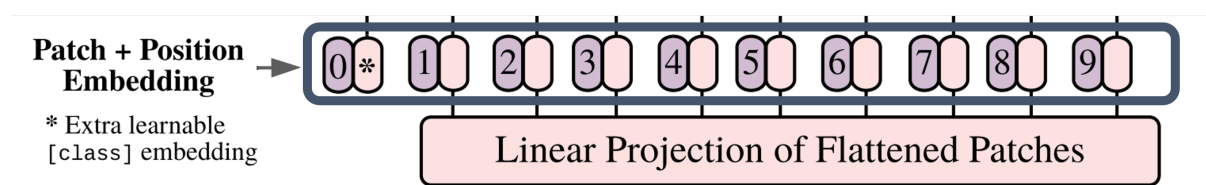
学習パラメータであり、バッチの埋め込みと同じ長さです。

標準正規分布に従った乱数をクラストークンの初期値とします。

全てのパッチの埋め込みの先頭にクラストークンを新たに結合してEncoderに入力します。
数式表現は以下になります。

$$[x_{class}; x_p^1 E; x_p^2 E; \dots; x_p^{N_p} E] \in \mathbb{R}^{(N_p+1) \times D}$$

step5 Positional Embedding (位置埋め込み)



矩形内ピンク：埋め込み処理後とクラストークン結合

矩形内紫：位置埋め込み 1つの数字はEposの行にあたる

位置埋め込みをクラストークンと各パッチの結合ベクトルに加算します。

数式表現は以下になります。

$$Z = [x_{class}; x_p^1 E; x_p^2 E; \dots; x_p^{N_p} E] + E_{pos} \in \mathbb{R}^{(N_p+1) \times D}$$

$E_{pos} \in \mathbb{R}^{(N_p+1) \times D}$: 位置埋め込み

位置埋め込みは学習可能なパラメータであり、初期値は標準正規分布による乱数になります。


損失の最小化によって更新されます。

この処理によってパッチの位置情報をベクトルに取り入れます。

パッチの位置情報とはパッチが画像内のどこに位置するかを示す情報です。

実装

Google Colaboratory

 <https://colab.research.google.com/drive/1nffFJxd7zgM-Qbm5gS79ISa4JONicurw?usp=sharing>



```

# -----
# 2-3 Input Layer
# -----
print("=====2-3 Input Layer=====")

class ViTInputLayer(nn.Module):
    def __init__(self, in_channels:int=3, emb_dim:int=384, num_patch_row:int=2, image_size:int=32):
        """
        引数:
            in_channels: 入力画像のチャンネル数
            emb_dim: 埋め込み後のベクトルの長さ
            num_patch_row: 高さ方向のパッチの数。例は2x2であるため、2をデフォルト値とした
            image_size: 入力画像の1辺の大きさ。入力画像の高さと幅は同じであると仮定
        """
        super(ViTInputLayer, self).__init__()
        self.in_channels=in_channels
        self.emb_dim = emb_dim
        self.num_patch_row = num_patch_row
        self.image_size = image_size

        # パッチの数
        ## 例: 入力画像を2x2のパッチに分ける場合、num_patchは4
        self.num_patch = self.num_patch_row**2

        # パッチの大きさ
        ## 例: 入力画像の1辺の大きさが32の場合、patch_sizeは16
        self.patch_size = int(self.image_size // self.num_patch_row)

        # 入力画像のパッチへの分割 & パッチの埋め込みを一気に行う層
        self.patch_emb_layer = nn.Conv2d(
            in_channels=self.in_channels,
            out_channels=self.emb_dim,
            kernel_size=self.patch_size,
            stride=self.patch_size
        )

        # クラストークン
        self.cls_token = nn.Parameter(
            torch.randn(1, 1, emb_dim)
        )

        # 位置埋め込み
        ## クラストークンが先頭に結合されているため、
        ## 長さemb_dimの位置埋め込みベクトルを(パッチ数+1)個用意
        self.pos_emb = nn.Parameter(
            torch.randn(1, self.num_patch+1, emb_dim)
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        引数:
            x: 入力画像。形状は、(B, C, H, W)。[式(1)]
            B: パッチサイズ、C: チャンネル数、H: 高さ、W: 幅
        返り値:
            z_0: ViTへの入力。形状は、(B, N, D)。
            B: パッチサイズ、N: トークン数、D: 埋め込みベクトルの長さ
        """
        # パッチの埋め込み & flatten [式(3)]
        ## パッチの埋め込み (B, C, H, W) -> (B, D, H/P, W/P)
        ## ここで、Pはパッチ1辺の大きさ
        z_0 = self.patch_emb_layer(x)

        ## パッチのflatten (B, D, H/P, W/P) -> (B, D, Np)

```

```

## ここで、Npはパッチの数(=H*W/P^2)
z_0 = z_0.flatten(2)

## 軸の入れ替え (B, D, Np) -> (B, Np, D)
z_0 = z_0.transpose(1, 2)

# パッチの埋め込みの先頭にクラストークンを結合 [式(4)]
## (B, Np, D) -> (B, N, D)
## N = (Np + 1)であることに留意
## また、cls_tokenの形状は(1,1,D)であるため、
## repeatメソッドによって(B,1,D)に変換してからパッチの埋め込みとの結合を行う
z_0 = torch.cat(
    [self.cls_token.repeat(repeats=(x.size(0),1,1)), z_0], dim=1)

# 位置埋め込みの加算 [式(5)]
## (B, N, D) -> (B, N, D)
z_0 = z_0 + self.pos_emb
return z_0

```