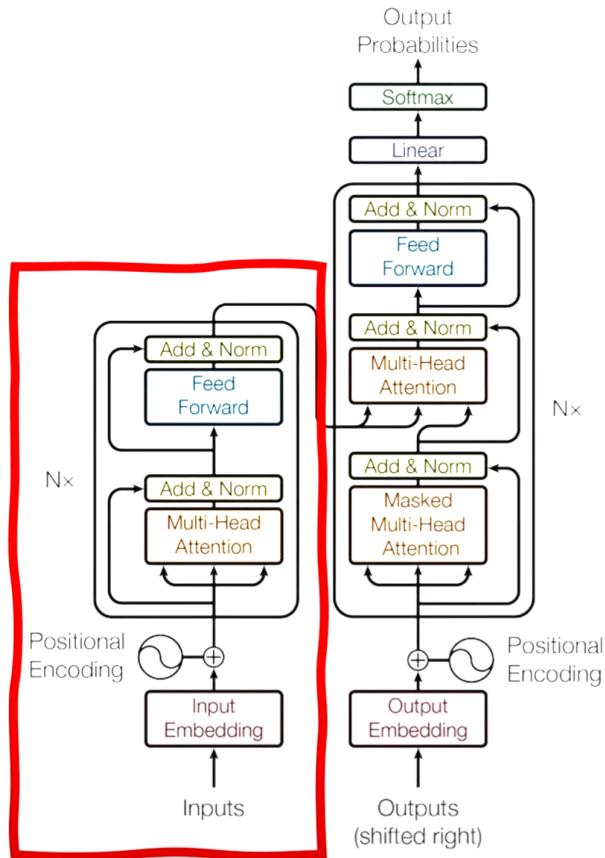


VIT

- מודל הרחבת עבור Transformers רק עבר מידע ויזואלי (תמונות).
המודל מורכב מ-Encoder ו Decoder כלהלן:



בצורת Encoder עובד?

תחילה הקלט מומר לוקטורים נומריים או ל-Embeddings (התאמת חח"ע בין מידע לוקטורים) שנעשה ע"י Image Patches בה התמונה מתפרקת לסדרה על ידי הרכיב הראשון במודל. כל צבע בתמונה עובר דרך נוירונים באמצעות שכבות לנאריות והפלט הוא Embedding.

איך ניתן לבצע את ה-Image Patching בצורה יעילה?

כל תמונה מורכבת מ-3 ערוצי צבע (אדום, ירוק וכחול), מרוחב ומוארך. בשביב לפרק את התמונה לרוב משתמש בחיבור בשם Einops שמאפשרת בסדר מערכים רב ממדים. ניתן לבצע את הסידור באמצעות הקוד הבא:

```
Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=patch_size, p2=patch_size)
```

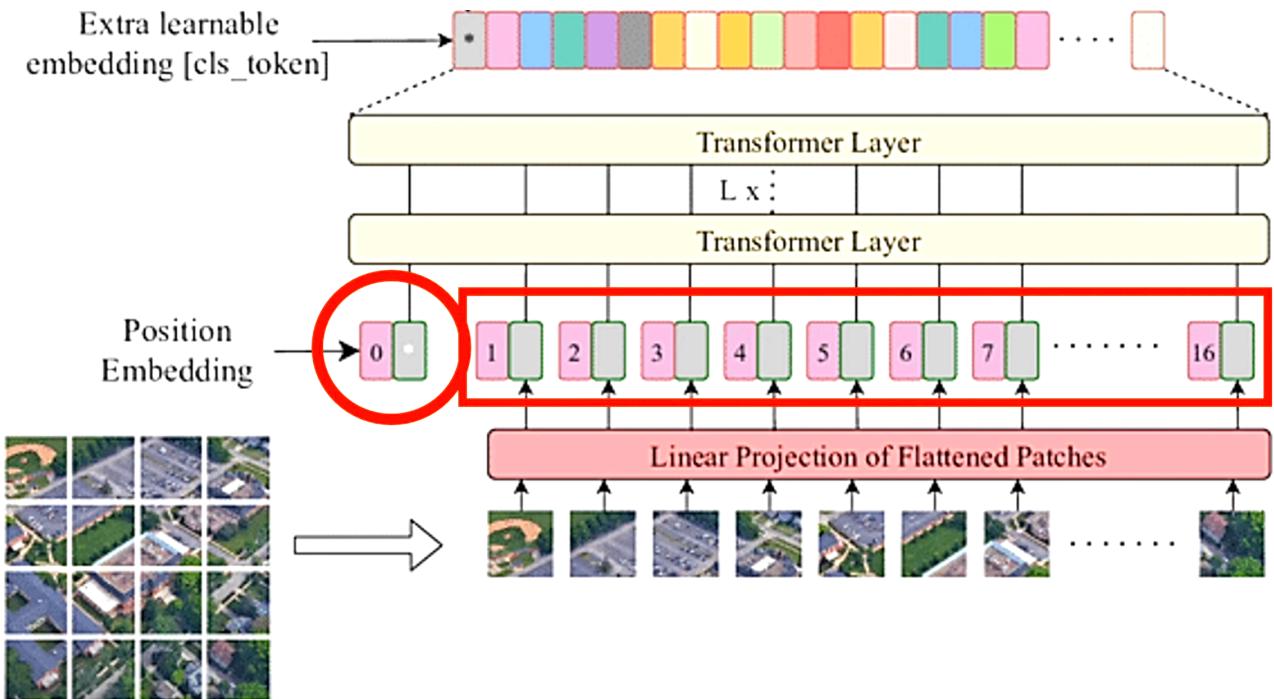
כאשר b זה גודל ה- batch , c זה 3 (כמויות ערוצי הצבע), h זה הגובה וה- w זה הרוחב ו- $p1$ ו- $p2$ זה לפי כמות ה-patches שרצים לפרק את התמונה למשול 16×16 או $p1, p2 = 16$

להלן הקוד עבור ה-Patch Embedding

```
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels = 3, patch_size = 8, emb_size = 128):
        self.patch_size = patch_size
        super().__init__()
        self.projection = nn.Sequential(
            # break-down the image in s1 x s2 patches and flat them
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=patch_size, p2=patch_size),
            nn.Linear(patch_size * patch_size * in_channels, emb_size)
        )

    def forward(self, x: Tensor) -> Tensor:
        x = self.projection(x)
        return x
```

הופך את כל המידע לייצוג גלובלי אחד בלבד בשביל הסיווג.
בוואו נזכר כי כל Patch הופך לוקטור כולל עוד וקטור למידה בהתחלה הנקרא **Position Embedding** המציג את כל התמונה ומתקבל מילדי מכל הקלטים. מהו הטוקן המשמש כ-**Feature Extractor**?



- נזכר כי מודל Transformers הוא תלוי מיקום ולכן תורם למודל להבין איפא כל Patch נמצא בתמונה המקורי.
לא מדובר במספר סידורי אלא בוקטור (מעumi נוחות וחיסכון) שמורכב משילוב פונקציות \sin ו- \cos .
נבחין כי ב-VIT ה-Positional Embedding נעשה לאחר Input Embedding ולא במקביל.
כפי שניתן לראות להלן, זהו פרמטר וקטור שניינו למידה עם מימדים:

```
# Learnable params
num_patches = (img_size // patch_size) ** 2
self.pos_embedding = nn.Parameter(
    torch.randn(1, num_patches + 1, emb_dim))
self.cls_token = nn.Parameter(torch.rand(1, 1, emb_dim))
```

- בлок זה מתבצע N פעמים. נחلك אותו לרכיביו:
• **ה-Encoder**: מאפשר לשתף מידע בין הקלטים השונים.
• **ה-Multi-Head Attention**: יש לו 3 קלטים שהם query, key, value והמיושם הוא כדלהלן (יש כבר מימוש מספרית `(torch)`):

```
class Attention(nn.Module):
    def __init__(self, dim, n_heads, dropout):
        super().__init__()
        self.n_heads = n_heads
        self.att = torch.nn.MultiheadAttention(embed_dim=dim,
                                              num_heads=n_heads,
                                              dropout=dropout)
        self.q = torch.nn.Linear(dim, dim)
        self.k = torch.nn.Linear(dim, dim)
        self.v = torch.nn.Linear(dim, dim)

    def forward(self, x):
        q = self.q(x)
        k = self.k(x)
        v = self.v(x)
        attn_output, attn_output_weights = self.att(x, x, x)
        return attn_output
```

- **ה-Norm-Add:** שכבה הנורמליזציה. מנורמלת כל קלט לשכבה. הסיבה מדוע לא משתמשים ב-Batch Norm המקביל שכן Transformers משתמש בסדרות.attention השימוש השתרמשנו במיומש PreNorm בה הוא משתמש בפונקציה למשל

```
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)

norm = PreNorm(128, Attention(dim=128, n_heads=4, dropout=0.))
norm(torch.ones((1, 5, 128))).shape
```

- **ה-Feed Forward:** שכבה לינארית שמטרתה לקחת את ה-Attention ומעבירה אותו לשכבה הבאה או מוציאה אותו כפלט. בקוד אנו משתמשים ב-2 שכבות לינאריות ובפונקציית ה-GELU בשם Activision ובנוסף אנו משתמשים ב-Dropout כדי להימנע מ-Overfitting

```
class FeedForward(nn.Sequential):
    def __init__(self, dim, hidden_dim, dropout = 0.):
        super().__init__(
            nn.Linear(dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout)
        )
ff = FeedForward(dim=128, hidden_dim=256)
ff(torch.ones((1, 5, 128))).shape
```

- **ה-Residual:** מטרתו לסייע בזרימה של הרשת למפלס כדי למנוע את בעיית Vanishing Gradients

```
class ResidualAdd(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

    def forward(self, x, **kwargs):
        res = x
        x = self.fn(x, **kwargs)
        x += res
        return x

residual_att = ResidualAdd(Attention(dim=128, n_heads=4, dropout=0.))
residual_att(torch.ones((1, 5, 128))).shape
```

כל המודל:

בקוד הבא חסרים כמה נורמות ו-Dropouts והוא מורכב מ-4 שכבות בלבד ומועד לתמונה קטנה יותר.

```
class ViT(nn.Module):
    def __init__(self, ch=3, img_size=144, patch_size=4, emb_dim=32,
                 n_layers=6, out_dim=37, dropout=0.1, heads=2):
        super(ViT, self).__init__()

        # Attributes
        self.channels = ch
        self.height = img_size
        self.width = img_size
        self.patch_size = patch_size
        self.n_layers = n_layers

        # Patching
        self.patch_embedding = PatchEmbedding(in_channels=ch, patch_size=patch_size, emb_size=emb_dim)
        # Learnable params
        num_patches = (img_size // patch_size) ** 2
        self.pos_embedding = nn.Parameter(
            torch.randn(1, num_patches + 1, emb_dim))
        self.cls_token = nn.Parameter(torch.rand(1, 1, emb_dim))

        # Transformer Encoder
        self.layers = nn.ModuleList([])
        for _ in range(n_layers):
            transformer_block = nn.Sequential(
                ResidualAdd(PreNorm(emb_dim, Attention(emb_dim, n_heads = heads, dropout = dropout))),
                ResidualAdd(PreNorm(emb_dim, FeedForward(emb_dim, emb_dim, dropout = dropout))))
            self.layers.append(transformer_block)

        # Classification head
        self.head = nn.Sequential(nn.LayerNorm(emb_dim), nn.Linear(emb_dim, out_dim))

    def forward(self, img):
        # Get patch embedding vectors
        x = self.patch_embedding(img)
        b, n, _ = x.shape

        # Add cls token to inputs
        cls_tokens = repeat(self.cls_token, '1 1 d -> b 1 d', b = b)
        x = torch.cat([cls_tokens, x], dim=1)
        x += self.pos_embedding[:, :(n + 1)]

        # Transformer layers
        for i in range(self.n_layers):
            x = self.layers[i](x)

        # Output based on classification token
        return self.head(x[:, 0, :])
```

הבדלים בין CNN ל-ViT ומתי להשתמש באיזה מודל:

