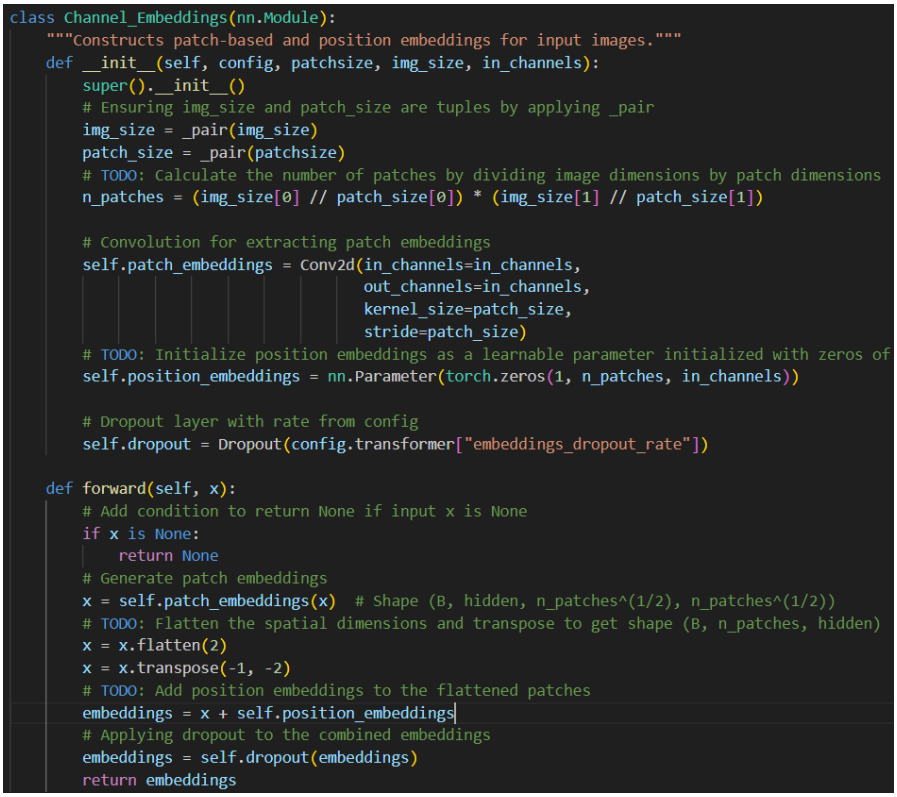
تمرین چهارم شناسایی الگو

برای این پروژه ما باید فایل ChannelTransformer.py را تکمیل می کردیم که یک مدل channel-wise است. در ادامه بخش های کد و عملکرد کلی آن را بیان می کنیم.



```python
class Channel_Embeddings(nn.Module):
    """Constructs patch-based and position embeddings for input images."""
    def __init__(self, config, patchsize, img_size, in_channels):
        super().__init__()
        # Ensuring img_size and patch_size are tuples by applying _pair
        img_size = _pair(img_size)
        patch_size = _pair(patchsize)
        # TODO: Calculate the number of patches by dividing image dimensions by patch dimensions
        n_patches = (img_size[0] // patch_size[0]) * (img_size[1] // patch_size[1])

        # Convolution for extracting patch embeddings
        self.patch_embeddings = Conv2d(in_channels=in_channels,
                                       out_channels=in_channels,
                                       kernel_size=patch_size,
                                       stride=patch_size)
        # TODO: Initialize position embeddings as a learnable parameter initialized with zeros of
        self.position_embeddings = nn.Parameter(torch.zeros(1, n_patches, in_channels))

        # Dropout layer with rate from config
        self.dropout = Dropout(config.transformer["embeddings_dropout_rate"])

    def forward(self, x):
        # Add condition to return None if input x is None
        if x is None:
            return None
        # Generate patch embeddings
        x = self.patch_embeddings(x)  # Shape (B, hidden, n_patches^(1/2), n_patches^(1/2))
        # TODO: Flatten the spatial dimensions and transpose to get shape (B, n_patches, hidden)
        x = x.flatten(2)
        x = x.transpose(-1, -2)
        # TODO: Add position embeddings to the flattened patches
        embeddings = x + self.position_embeddings
        # Applying dropout to the combined embeddings
        embeddings = self.dropout(embeddings)
        return embeddings
```

در این کلاس برای Embedding تصاویر ورودی ایجاد میشود. که آن ترکیبی از اطلاعات مربوط به پچ های کوچک تصویر و موقعیت مکانی آن هاست که برای استفاده در مدل ما مناسب است.ابتدا ابعاد تصویر به بخش های کوچکتر (پچ ها) تقسیم می شود و با استفاده از یک کانولوشن دو بعدی (Conv2d) هر پچ به یک Embedding تبدیل می شود.

سپس به هر پچ یک بردار Position Embedding اضافه می شود تا مدل بتواند اطلاعات مکانی پچ ها را نیز درک کند. در ادامه برای جلوگیری از Overfitting dropout به embedding ها اعمال می شود.

```python
class Reconstruct(nn.Module):
    """Reconstructs feature map to original resolution using upsampling and convolution."""
    def __init__(self, in_channels, out_channels, kernel_size, scale_factor):
        super(Reconstruct, self).__init__()
        # Setting padding based on kernel size (1 for kernel size 3, else 0)
        padding = 1 if kernel_size == 3 else 0
        # Convolution to reconstruct feature maps
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, padding=padding)
        # Batch normalization and activation
        self.norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.ReLU(inplace=True)
        self.scale_factor = scale_factor

    def forward(self, x):
        # Adding condition to return None if x is None
        if x is None:
            return None
        # Reshape and upsample the input tensor
        B, n_patch, hidden = x.size()  # Reshape from (B, n_patch, hidden) to (B, h, w, hidden)
        # TODO: Calculate the dimensions of height and width. they will be the sqrt of number of patches
        h, w = int(np.sqrt(n_patch)), int(np.sqrt(n_patch))
        # Permute and reshape x to match 2D convolution input format
        x = x.permute(0, 2, 1).contiguous().view(B, hidden, h, w)
        # Upsample x to scale factor
        x = nn.Upsample(scale_factor=self.scale_factor)(x)
        # Apply convolution, normalization, and activation
        out = self.conv(x)
        out = self.norm(out)
        out = self.activation(out)
        return out
```

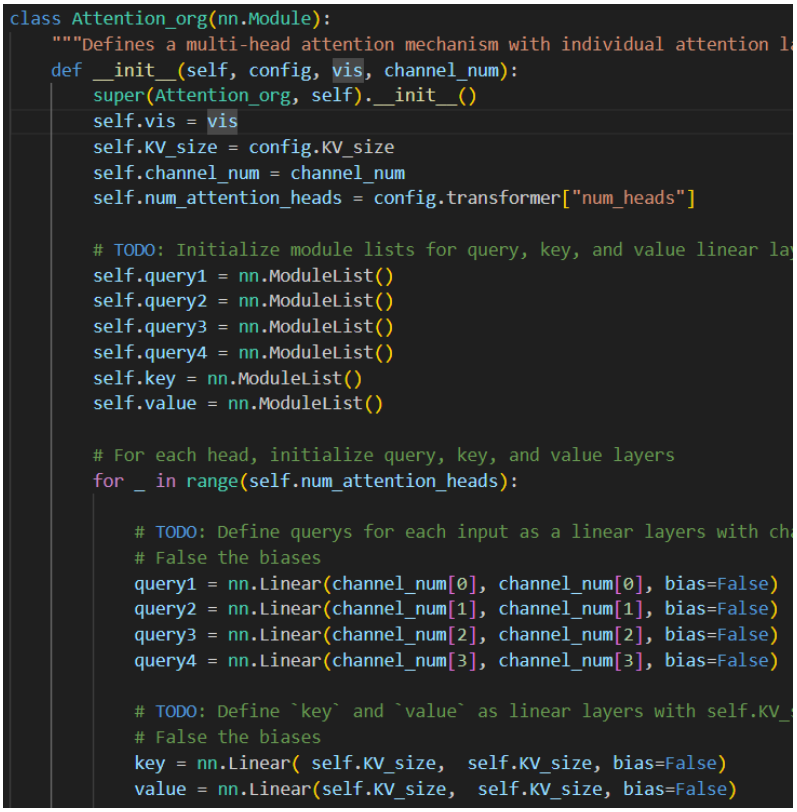
در کلاس Reconstruct یک لایه کانولوشنی تنظیم می شود که مسئول بازسازی feature map ها خواهد بود. اندازه کرنل و پدینگ بر اساس پارامترهای مشخص شده تعریف می شوند تا اطمینان حاصل شود که ابعاد خروجی به طور مناسب مدیریت می شوند.

علاوه بر این، نرمال سازی دسته ای و تابع فعال سازی ReLU برای بهبود نمایش ویژگی ها و تثبیت آموزش قرار داده شده است. پارامتر scale_factor تعیین می کند که feature map ورودی چقدر باید افزایش مقیاس یابد که این امکان را برای تنظیم رزولوشن خروجی فراهم می کند.
در تابع فوروارد کلاس Reconstruct یک تنسور ورودی که نمایانگر ویژگی های پردازش شده است را می گیرد.در ادامه ابتدا این تنسور را تغییر شکل می دهد تا برای convolution دو بعدی آماده شود.

ارتفاع و عرض تنسور از جذر تعداد پچ ها به دست می آید که نحوه تقسیم اولیه ورودی را نشان می دهد. پس از تغییر شکل، تنسور طبق ضریب مقیاس مشخص شده افزایش مقیاس می یابد و سپس از لایه کانولوشن عبور کرده و به دنبال آن نرمال سازی دسته ای و فعال سازی انجام می شود.

خروجی نهایی یک feature map بازسازی شده است که برای کارهایی مانند segmentation یا classification تصویر مناسب می سازد.

عکس بخشی از این کلاس:



```python
class Attention_org(nn.Module):
    """Defines a multi-head attention mechanism with individual attention l
    def __init__(self, config, vis, channel_num):
        super(Attention_org, self).__init__()
        self.vis = vis
        self.KV_size = config.KV_size
        self.channel_num = channel_num
        self.num_attention_heads = config.transformer["num_heads"]

        # TODO: Initialize module lists for query, key, and value linear la
        self.query1 = nn.ModuleList()
        self.query2 = nn.ModuleList()
        self.query3 = nn.ModuleList()
        self.query4 = nn.ModuleList()
        self.key = nn.ModuleList()
        self.value = nn.ModuleList()

        # For each head, initialize query, key, and value layers
        for _ in range(self.num_attention_heads):

            # TODO: Define querys for each input as a linear layers with ch
            # False the biases
            query1 = nn.Linear(channel_num[0], channel_num[0], bias=False)
            query2 = nn.Linear(channel_num[1], channel_num[1], bias=False)
            query3 = nn.Linear(channel_num[2], channel_num[2], bias=False)
            query4 = nn.Linear(channel_num[3], channel_num[3], bias=False)

            # TODO: Define `key` and `value` as linear layers with self.KV_
            # False the biases
            key = nn.Linear( self.KV_size,  self.KV_size, bias=False)
            value = nn.Linear(self.KV_size,  self.KV_size, bias=False)
```

کلاس Attention_org یک بخش مهم و حیاتی از معماری شبکه عصبی است که یک مکانیزم توجه چندسر(multi-head) را پیاده سازی می کند که به طور خاص برای پردازش چندین کانال داده ورودی طراحی شده است. عملکرد اصلی آن محاسبه attention scores است.

در مرحله ی اول کلاس Attention_org چندین لایه ی خطی برای query ها، key ها و value ها برای هر attention head تنظیم می کند. هر لایه خطی مسئول تبدیل ورودی های embedding به فرمت مناسب برای محاسبه attention scores است.

در این کلاس همچنین لایه های نرمال سازی، لایه های دراپ اوت و تبدیل های خطی خروجی را برای اطمینان از عملکرد موثر و کارآمد مکانیزم توجه اولیه سازی می کند. پارامترهای پیکربندی جنبه هایی مانند تعداد attention head ها و اندازه جفت های key-value را تعیین می کنند که برای کنترل پیچیدگی و عملکرد مدل حیاتی هستند.
در ادامه کلاس چندین embedding را به عنوان ورودی می گیرد و query ها، key ها و value ها را برای multi-head تولید می کند.

attention score ها را با انجام ضرب ماتریسی بین query ها و key ها محاسبه می کند، که سپس با مقیاس بندی برای تثبیت گرادیان ها در طول آموزش همراه است. سپس attention score ها از

طریق یک تابع سافت مکس عبور داده می شوند تا احتمال های attention به دست بیایند، که نشان می دهند چه مقدار تمرکز باید بر روی هر بخش از ورودی گذاشته شود. درصدی از این احتمالات به منظور جلوگیری از overfitting اعمال می شود تا اطمینان حاصل شود که مدل به خوبی به داده های دیده نشده تعمیم می یابد.

در نهایت کلاس Attention_org لایه های context را با ضرب کردن احتمال های attention با تنسورهای value مربوطه محاسبه می کند. این کار منجر به لایه های context می شود که اطلاعات مهم از تمامی کانال های ورودی را در خود جای می دهند و در عین حال روابط ابعادی را حفظ می کنند. به طور کلی این کلاس نقش حیاتی در بهبود توانایی مدل برای شناسایی الگوها و وابستگی های پیچیده در داده های چندکاناله از طریق مکانیزم توجه پیشرفته خود ایفا می کند.

```python
class Mlp(nn.Module):
    """Defines a Multi-Layer Perceptron (MLP) with two fully connected layers
    def __init__(self, config, in_channel, mlp_channel):
        super(Mlp, self).__init__()
        # TODO: Define the first fully connected layer (fc1) with input size i
        self.fc1 = nn.Linear(in_channel, mlp_channel)
        # TODO: Define the second fully connected layer (fc2) with input size
        self.fc2 = nn.Linear(mlp_channel, in_channel)
        # Activation function
        self.act_fn = nn.GELU()
        # Dropout layer with rate from config
        self.dropout = Dropout(config.transformer["dropout_rate"])
        # Initialize weights
        self._init_weights()

    def _init_weights(self):
        """Initializes weights for fully connected layers."""
        # TODO: Use Xavier uniform initialization for fc1 weights
        nn.init.xavier_uniform_(self.fc1.weight)
        # TODO: Use Xavier uniform initialization for fc2 weights
        nn.init.xavier_uniform_(self.fc2.weight)
        # TODO: Initialize biases for fc1 and fc2 with normal distribution (me
        nn.init.normal_(self.fc1.bias, std=1e-6)
        nn.init.normal_(self.fc2.bias, std=1e-6)

    def forward(self, x):
        # TODO: Apply first fully connected layer (fc1) to the input x
        x = self.fc1(x)
        # TODO: Apply activation function (GELU)
        x = self.act_fn(x)
        # TODO: Apply dropout after the activation function
        x = self.dropout(x)
        # TODO: Apply the second fully connected layer (fc2) to the transforme
        x = self.fc2(x)
        # TODO: Apply dropout again after the second layer
        x = self.dropout(x)
        # Return the output
        return x
```

کلاس Mlp یک ساختار پرسپترون چند لایه (MLP) را در یک شبکه عصبی تعریف می کند که شامل دو لایه کاملا متصل با تابع فعال سازی GELU است. هدف اصلی آن تبدیل نمایش های ویژگی ورودی

از طریق نگاشت های غیرخطی است که توانایی مدل را در یادگیری الگوهای پیچیده در داده ها افزایش می دهد.

MLP به عنوان یک شبکه feed-forward عمل می کند که می تواند در معماری های مختلف، از جمله Vision Transformer ها و سایر مدل های یادگیری عمیق ادغام شود.
ابتدا کلاس Mlp دو لایه خطی به نام های fc1 و fc2 راه اندازی می کند. لایه اول (fc1) اندازه ورودی را که توسط in_channel تعریف شده است می گیرد و اندازه ای را که توسط mlp_channel مشخص شده است خروجی می دهد.

لایه دوم (fc2) این تبدیل را معکوس می کند، خروجی لایه اول را گرفته و آن را به اندازه ورودی اصلی باز می گرداند. این کلاس همچنین شامل یک تابع فعال سازی GELU است که غیرخطی بودن را به مدل معرفی می کند و یک لایه دراپ اوت برای کاهش اورفیتینگ در حین آموزش قرار میدهد.

مقداردهی اولیه وزن ها با استفاده از توزیع یکنواخت Xavier برای هر دو لایه انجام می شود. در تابع فوروارد تنسور ورودی به صورت متوالی از طریق دو لایه کاملا متصل پردازش می شود. اولین مرحله، عبور از fc1 است که به دنبال آن فعال سازی GELU و دراپ اوت قرار دارد. سپس، به fc2 وارد می شود، جایی که قبل از تولید خروجی نهایی، دوباره یک دراپ اوت انجام می شود.

بخشی از کلاس Block_ViT:

```python
class Block_ViT(nn.Module):
    """Defines a Vision Transformer (ViT) block with attention normalization, channel attention, and fe
    def __init__(self, config, vis, channel_num):
        super(Block_ViT, self).__init__()
        # Expansion ratio for the hidden size in the feed-forward network
        expand_ratio = config.expand_ratio
        # Define LayerNorm layers for attention normalization for each input channel, with eps=1e-6
        self.attn_norm1 = LayerNorm(channel_num[0], eps=1e-6)
        self.attn_norm2 = LayerNorm(channel_num[1], eps=1e-6)
        self.attn_norm3 = LayerNorm(channel_num[2], eps=1e-6)
        self.attn_norm4 = LayerNorm(channel_num[3], eps=1e-6)
        # Define LayerNorm for the concatenated channel embeddings using config.KV_size as dimension
        self.attn_norm = LayerNorm(config.KV_size, eps=1e-6)
        # TODO: Channel-wise attention mechanism using Attention_org class with proper arguments
        self.channel_attn = Attention_org(config, vis, channel_num)
        # Define LayerNorm layers for feed-forward normalization for each input channel
        self.ffn_norm1 = LayerNorm(channel_num[0], eps=1e-6)
        self.ffn_norm2 = LayerNorm(channel_num[1], eps=1e-6)
        self.ffn_norm3 = LayerNorm(channel_num[2], eps=1e-6)
        self.ffn_norm4 = LayerNorm(channel_num[3], eps=1e-6)
        # Define feed-forward network (MLP) layers for each channel with the specified expansion ratio
        self.ffn1 = Mlp(config, channel_num[0], channel_num[0] * expand_ratio)
        # TODO: do the same for ffn2, ffn3, ffn4
        self.ffn2 = Mlp(config,channel_num[1],channel_num[1]*expand_ratio)
        self.ffn3 = Mlp(config,channel_num[2],channel_num[2]*expand_ratio)
        self.ffn4 = Mlp(config,channel_num[3],channel_num[3]*expand_ratio)
```

کلاس Block_ViT یک بلاک مهم در معماری ViT است که مکانیزم های attention و شبکه های feed-forward را برای پردازش موثر ورودی های embedding شده ادغام می کند که هدف اصلی آن بهبود استخراج ویژگی ها از چندین کانال ورودی از طریق channel-wise attention و تبدیلات غیرخطی است.

در کلاس Block_ViT چندین مولفه را تنظیم می کند. از جمله لایه های LayerNorm برای نرمال سازی ورودی ها قبل و بعد از عملیات attention و فیدفوروارد. همچنین کلاس Attention_org را ایجاد می کند که یک مکانیزم multi-head attention را برای پردازش چندین کانال پیاده سازی می کند.

علاوه بر این کلاس MLP را برای هر کانال ورودی تعریف می کند که امکان تبدیل های غیرخطی را فراهم می کند و نمایش های ویژگی را گسترش می کند. استفاده از نسبت گسترش به کنترل پیچیدگی این شبکه ها کمک می کند و اطمینان حاصل می کند که آن ها می توانند نمایش های غنی را یاد بگیرند بدون اینکه بیش از حد محاسباتی شوند.

در ادامه کلاس Block_ViT ورودی های تعبیه شده را ابتدا با لایه های LayerNorm مربوطه نرمال سازی می کند. سپس تمام embedding های موجود را به هم متصل می کند تا یک نمای جامع از داده های ورودی ایجاد کند. سپس هر embedding نرمال شده از طریق شبکه feed-forward مربوطه خود پردازش می شود که تبدیلات غیرخطی را برای بهبود ویژگی های یادگرفته شده اعمال می کند.

```python
class Encoder(nn.Module):
    """Defines an encoder with multiple ViT blocks and layer normalization for each input channel."""
    def __init__(self, config, vis, channel_num):
        super(Encoder, self).__init__()
        self.vis = vis
        # Initialize a ModuleList to hold the sequence of Block_ViT layers
        self.layer = nn.ModuleList()
        # Define LayerNorm for each input channel, with eps=1e-6 for numerical stability
        self.encoder_norm1 = LayerNorm(channel_num[0], eps=1e-6)
        self.encoder_norm2 = LayerNorm(channel_num[1], eps=1e-6)
        self.encoder_norm3 = LayerNorm(channel_num[2], eps=1e-6)
        self.encoder_norm4 = LayerNorm(channel_num[3], eps=1e-6)
        # Add a specified number of Block_ViT layers to the encoder
        for _ in range(config.transformer["num_layers"]):
            # TODO: Initialize a Block_ViT layer with the given config, vis, and channel_num
            layer = Block_ViT(config, vis, channel_num)
            # TODO: Append a deep copy of the initialized Block_ViT layer to self.layer
            self.layer.append(copy.deepcopy(layer))

    def forward(self, emb1, emb2, emb3, emb4):
        # Initialize a list to hold attention weights if visualization is enabled
        attn_weights = []
        # TODO: Pass each embedding (emb1-emb4) through the sequence of Block_ViT layers
        for layer_block in self.layer:
            # The Block_ViT layer processes all four embeddings and returns updated embeddings and a
            emb1, emb2, emb3, emb4, weights = layer_block(emb1,emb2,emb3,emb4)
            # Append attention weights if visualization (vis) is enabled
            if self.vis:
                attn_weights.append(weights)
        # TODO: Apply LayerNorm to the final outputs of each embedding channel
        emb1 = self.encoder_norm1(emb1) if emb1 is not None else None
        emb2 = self.encoder_norm2(emb2) if emb2 is not None else None
        emb3 = self.encoder_norm3(emb3) if emb3 is not None else None
        emb4 = self.encoder_norm4(emb4) if emb4 is not None else None
        # Return the normalized embeddings and collected attention weights
        return emb1, emb2, emb3, emb4, attn_weights
```

عملکرد کلاس Encoder پردازش ورودی های embedding شده از طریق یک سری بلوک های ترنسفورمر است که به مدل اجازه می دهد تا نمایش های پیچیده ای از داده ها را یاد بگیرد. انکودر معمولا از چندین لایه تشکیل شده است که هر کدام شامل مکانیزم های توجه و شبکه های feed-forward هستند که به طور مشترک برای بهبود استخراج ویژگی ها و درک زمینه ای کار می کنند.
کلاس Encoder چندین بلوک ترنسفورمر را راه اندازی می کند که ممکن است شامل نمونه هایی از کلاس هایی مانند Block_ViT باشد. هر بلوک مسئول اعمال multi-head attention و تبدیلات غیرخطی به ورودی های embedding شده است.

تعداد بلوک ها می تواند در config مشخص شود که این امکان را برای انعطاف پذیری در عمق و پیچیدگی مدل فراهم می کند. Encoder همچنین لایه های نرمال سازی را برای تثبیت آموزش و بهبود همگرایی ادغام می کند و اطمینان حاصل می کند که ورودی های embedding شده قبل از پردازش به درستی مقیاس بندی شده اند.
در ادامه کلاس Encoder ورودی های embedding شده را گرفته و به صورت متوالی از هر بلوک ترنسفورمر عبور می دهد. همان طور که داده ها از این بلوک ها عبور می کنند مکانیزم های توجه به مدل این امکان را می دهند که بر روی ویژگی های مرتبط تمرکز کند و اطلاعات کم اهمیت تر را نادیده بگیرد.

خروجی هر بلوک از طریق شبکه های feed-forward ایجاد می شود که تبدیل های غیرخطی را اعمال می کنند تا نمایه ویژگی ها را بیشتر تقویت کنند. خروجی نهایی انکودر شامل embedding های غنی شده ای است که اطلاعات زمینه ای محلی و جهانی را به طور همزمان در بر می گیرد.

```python
class ChannelTransformer(nn.Module):
    """Combines patch embeddings, an encoder, and reconstruction layers for a complete channel transformer model."""
    def __init__(self, config, vis, img_size, channel_num=[64, 128, 256, 512], patchSize=[32, 16, 8, 4]):
        super().__init__()

        # Define patch sizes for each embedding layer using elements of patchSize
        self.patchSize_1 = patchSize[0]
        self.patchSize_2 = patchSize[1]
        self.patchSize_3 = patchSize[2]
        self.patchSize_4 = patchSize[3]

        # TODO: Define Channel_Embeddings for each channel with specified patch size and image size
        self.embeddings_1 = Channel_Embeddings(config, self.patchSize_1, img_size=img_size, in_channels=channel_num[0])
        self.embeddings_2 = Channel_Embeddings(config,self.patchSize_2, img_size=img_size//2, in_channels=channel_num[1])
        self.embeddings_3 = Channel_Embeddings(config,self.patchSize_3, img_size=img_size//4, in_channels=channel_num[2])
        self.embeddings_4 = Channel_Embeddings(config,self.patchSize_4, img_size=img_size//8, in_channels=channel_num[3])

        # TODO: Initialize the Encoder with config, vis flag, and channel_num list
        self.encoder = Encoder(config, vis, channel_num)

        # TODO: Define reconstruction layers to upsample back to the original feature map dimensions
        self.reconstruct_1 = Reconstruct(channel_num[0], channel_num[0], kernel_size=1, scale_factor=(self.patchSize_1, self.patchSize_1))
        self.reconstruct_2 = Reconstruct(channel_num[1], channel_num[1], kernel_size=1,scale_factor=(self.patchSize_2,self.patchSize_2))
        self.reconstruct_3 = Reconstruct(channel_num[2], channel_num[2], kernel_size=1,scale_factor=(self.patchSize_3,self.patchSize_3))
        self.reconstruct_4 = Reconstruct(channel_num[3], channel_num[3], kernel_size=1,scale_factor=(self.patchSize_4,self.patchSize_4))
```

کلاس ChannelTransformer برای پیاده سازی یک معماری ترنسفورمر طراحی شده است که داده های ورودی چندکاناله(تصویر) را پردازش می کند.

این معماری شامل چندین مولفه از جمله embedding های کانال، مکانیزم های توجه و شبکه های feed-forward است تا به طور موثری نمایش های پیچیده ای از داده های ورودی یاد بگیرد.

در ابتدا کلاس ChannelTransformer ماژول های مختلفی را که برای عملکرد آن ضروری هستند، راه اندازی می کند. این شامل نمونه هایی از کلاس Channel_Embeddings برای تولید Embedding های مبتنی بر پچ و موقعیتی از تصاویر ورودی می شود. این کلاس همچنین شامل چندین بلوک ترنسفورمر است که از کلاس Block_ViT استفاده می کند، که مکانیزم های توجه و شبکه های feed-forward را ادغام می کند.

این بلوک ها به صورت پشته ای قرار داده شده اند تا یک معماری عمیق ایجاد کنند که قادر به استخراج ویژگی های محلی و جهانی از داده های ورودی باشد. علاوه بر این، لایه های نرمال سازی برای تثبیت آموزش و بهبود همگرایی قرار داده شده اند.
در ادامه ChannelTransformer ورودی های Embed شده را به صورت متوالی از طریق اجزای مختلف خود پردازش می کند. در ابتدا کانال امبدینگ ها را تولید می کند و سپس این امبدینگ ها را از طریق چندین بلاک ترنسفورمر عبور می دهد. هر بلوک از multi-head attention برای تمرکز بر

ویژگی های مرتبط استفاده می کند در حالی که از شبکه های feed-forward برای بهبود یادگیری نمایش استفاده می کند.

خروجی بلوک نهایی ترنسفورمر مجموعه ای از embedding های غنی شده است که اطلاعات مهمی را در تمام کانال ها در بر می گیرد.

برای اجرای این پروژه در کولب فایل نوت بوک main.ipynb را ایجاد کردیم که ابتدا gdown را نصب کردیم.

```
!pip install gdown
```

سپس پروژه تکمیل شده را که در گوگل درایو آپلود کردیم دانلود میکنیم.

```
!gdown https://drive.google.com/uc?id=1tGrRuYGIbYSexA8PBDfVRWvq0bkaMhu9
```

سپس فایل را از حالت فشرده خارج کردیم.

```
!unzip final.zip
```

در نهایت پس از نصب کتابخانه های لازم با دستور زیر فایل train_model.py را اجرا کردیم.

```
!python /content/modelChannelTransformer/train_model.py
```

نتیجه نهایی:

```
========= Epoch [175/2001] =========
Test_session_12.29_15h05
Training with batch size : 4
   [Train] Epoch: [175][1/6]  Loss:0.200 (Avg 0.2004) Dice:0.8203 (Avg 0.8203) LR 6.89e-04   (AvgTime 1.4)
   [Train] Epoch: [175][2/6]  Loss:0.197 (Avg 0.1986) Dice:0.8490 (Avg 0.8347) LR 6.89e-04   (AvgTime 0.9)
   [Train] Epoch: [175][3/6]  Loss:0.229 (Avg 0.2088) Dice:0.8095 (Avg 0.8263) LR 6.89e-04   (AvgTime 0.8)
   [Train] Epoch: [175][4/6]  Loss:0.190 (Avg 0.2042) Dice:0.8180 (Avg 0.8242) LR 6.89e-04   (AvgTime 0.7)
   [Train] Epoch: [175][5/6]  Loss:0.182 (Avg 0.1997) Dice:0.8278 (Avg 0.8249) LR 6.89e-04   (AvgTime 0.7)
   [Train] Epoch: [175][6/6]  Loss:0.179 (Avg 0.1962) Dice:0.8306 (Avg 0.8259) LR 6.89e-04   (AvgTime 0.6)
Validation
   [Val] Epoch: [175][1/2]  Loss:0.241 (Avg 0.2415) Dice:0.7837 (Avg 0.7837) (AvgTime 0.6)
   [Val] Epoch: [175][2/2]  Loss:0.376 (Avg 0.2865) Dice:0.7304 (Avg 0.7659) (AvgTime 0.4)
         Mean dice:0.7659 does not increase, the best is still: 0.7840 in epoch 124
         early_stopping_count: 51/50
         early_stopping!
```

همانطور که مشاهده می شود در ایپاک 175 متوقف شده و یک early stop رخ داده چرا که early stop با مقدار Patience 50 = تنظیم شده است و در ایپاک 124 ما به نتیجه قابل قبول که با توجه به معیار dice هست رسیدیم.
early stop از مصرف اضافی منابع محاسباتی جلوگیری می کند و مانع از یادگیری غیرمفید شده است.

محمد حقیقت – 403722042

برای رفع برخی ایرادات و ابهامات از Chatgpt استفاده شده است.