OMNIVISION

APRIL 2022

Shape Changes in Vision Transformers

w.r.t. Swin Transformer and MobileVit

Xiao Shijie
OMNIVISION Singapore Algorithm Computer Vision Team

Outline

- Swin Transformer
- MobileVit

Outline

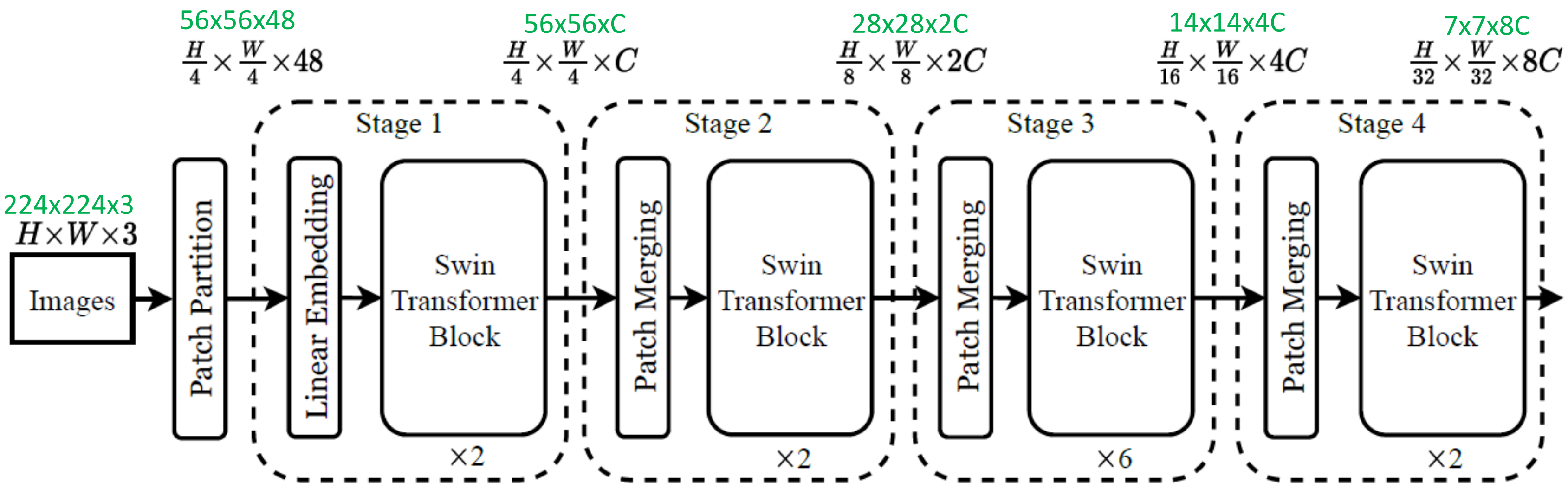
- **Swin Transformer**
- MobileVit

configs

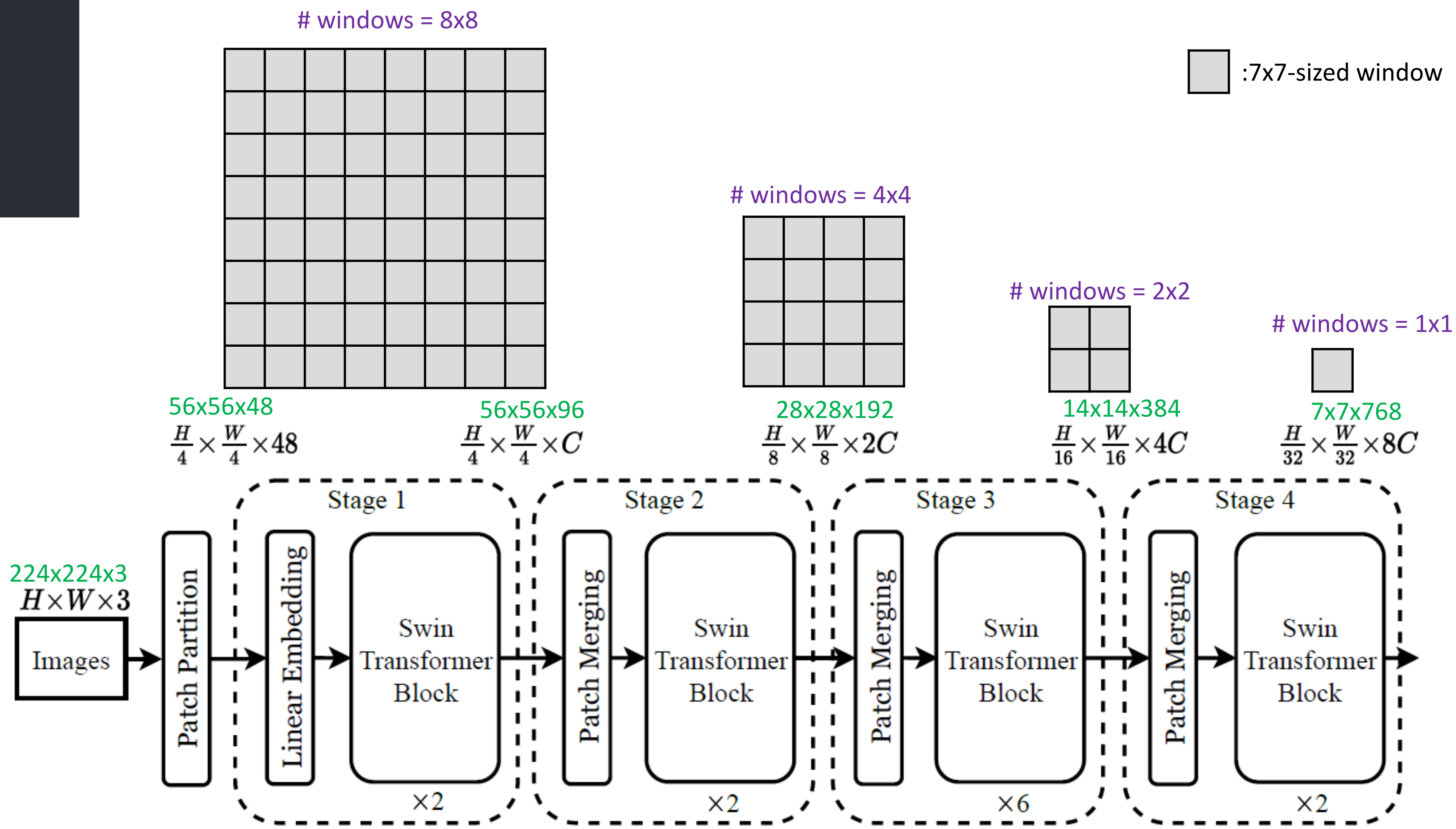
```yaml
MODEL:
  TYPE: swin
  NAME: swin_tiny_patch4_window7_224
  DROP_PATH_RATE: 0.2
  SWIN:
    EMBED_DIM: 96
    DEPTHS: [ 2, 2, 6, 2 ]
    NUM_HEADS: [ 3, 6, 12, 24 ]
    WINDOW_SIZE: 7
```

```yaml
MODEL:
  TYPE: swin
  NAME: swin_small_patch4_window7_224
  DROP_PATH_RATE: 0.3
  SWIN:
    EMBED_DIM: 96
    DEPTHS: [ 2, 2, 18, 2 ]
    NUM_HEADS: [ 3, 6, 12, 24 ]
    WINDOW_SIZE: 7
```

```yaml
MODEL:
  TYPE: swin
  NAME: swin_base_patch4_window7_224
  DROP_PATH_RATE: 0.5
  SWIN:
    EMBED_DIM: 128
    DEPTHS: [ 2, 2, 18, 2 ]
    NUM_HEADS: [ 4, 8, 16, 32 ]
    WINDOW_SIZE: 7
```



```yaml
MODEL:
  TYPE: swin
  NAME: swin_tiny_patch4_window7_224
  DROP_PATH_RATE: 0.2
  SWIN:
    EMBED_DIM: 96
    DEPTHS: [ 2, 2, 6, 2 ]
    NUM_HEADS: [ 3, 6, 12, 24 ]
    WINDOW_SIZE: 7
```



```
class SwinTransformerBlock(nn.Module):

def forward(self, x):
    H, W = self.input_resolution 56, 56
    B, L, C = x.shape  (B, 3136, 96)
    assert L == H * W, "input feature has wrong size"

    shortcut = x (B, 3136, 96)
    x = self.norm1(x)
    x = x.view(B, H, W, C) (B, 56, 56, 96)

    # cyclic shift
    if self.shift_size > 0:
        shifted_x = torch.roll(x, shifts=(-self.shift_size, -self.shift_size), dims=(1, 2))
    else:
        shifted_x = x  (B, 56, 56, 96)

    # partition windows
    x_windows = window_partition(shifted_x, self.window_size)  # nW*B, window_size, window_size, C  (64B, 7, 7, 96)
    x_windows = x_windows.view(-1, self.window_size * self.window_size, C)  # nW*B, window_size*window_size, C (64B, 49, 96)

    # W-MSA/SW-MSA
    attn_windows = self.attn(x_windows, mask=self.attn_mask)  # nW*B, window_size*window_size, C (64B, 49, 96)

    # merge windows
    attn_windows = attn_windows.view(-1, self.window_size, self.window_size, C)  (64B, 7, 7, 96)
    shifted_x = window_reverse(attn_windows, self.window_size, H, W)  # B H' W' C (B, 56, 56, 96)

    # reverse cyclic shift
    if self.shift_size > 0:
        x = torch.roll(shifted_x, shifts=(self.shift_size, self.shift_size), dims=(1, 2))
    else:
        x = shifted_x (B, 56, 56, 96)
    x = x.view(B, H * W, C)  (B, 56x56, 96) = (B, 3136, 96)

    # FFN
    x = shortcut + self.drop_path(x)
    x = x + self.drop_path(self.mlp(self.norm2(x)))

    return x (B, 3136, 96)
```

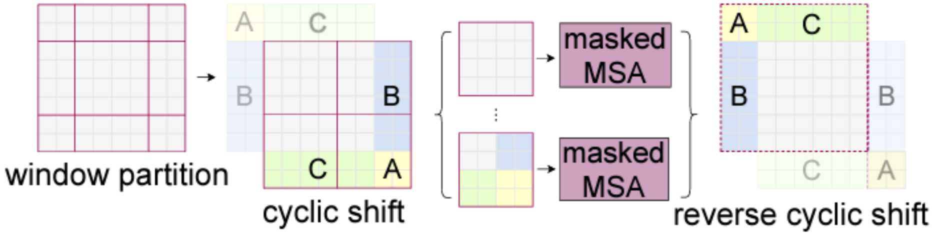
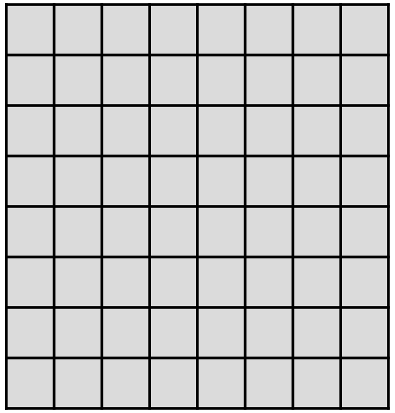
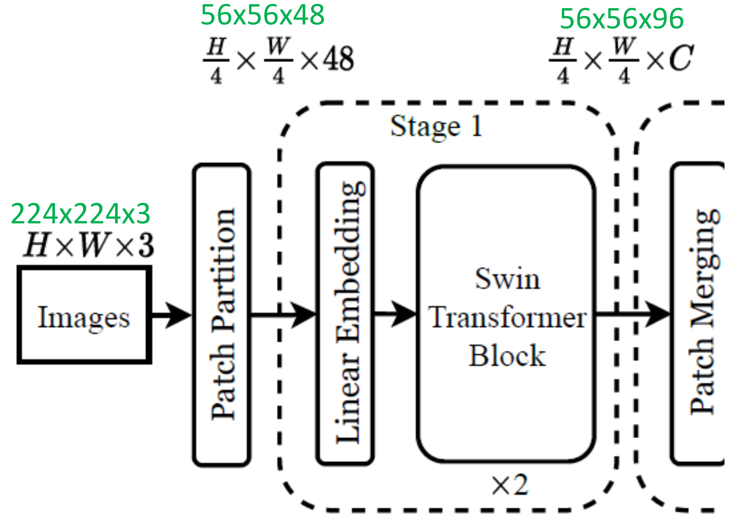


Figure 4. Illustration of an efficient batch computation approach for self-attention in shifted window partitioning.

windows = 8x8

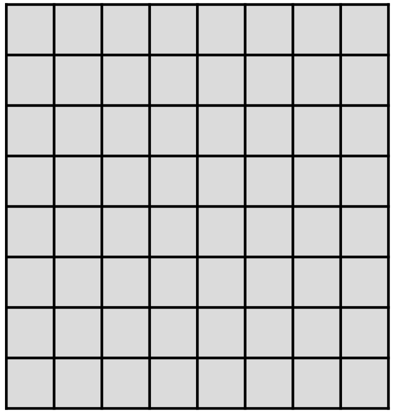


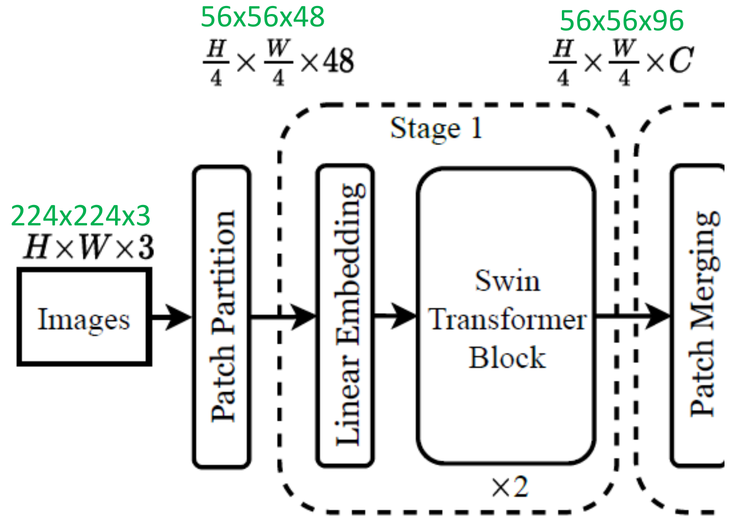


```python
class SwinTransformerBlock(nn.Module):

def forward(self, x):
    H, W = self.input_resolution 56, 56
    B, L, C = x.shape  (B, 3136, 96)
    assert L == H * W, "input feature has wrong size"

    shortcut = x (B, 3136, 96)
    x = self.norm1(x)
    x = x.view(B, H, W, C) (B, 56, 56, 96)

    # cyclic shift
    if self.shift_size > 0:
        shifted_x = torch.roll(x, shifts=(-self.shift_size, -self.shift_size), dims=(1, 2))
    else:
        shifted_x = x  (B, 56, 56, 96)

    # partition windows
    x_windows = window_partition(shifted_x, self.window_size)  # nW*B, window_size, window_size, C  (64B, 7, 7, 96)
    x_windows = x_windows.view(-1, self.window_size * self.window_size, C)  # nW*B, window_size*window_size, C (64B, 49, 96)

    # W-MSA/SW-MSA
    attn_windows = self.attn(x_windows, mask=self.attn_mask)  # nW*B, window_size*window_size, C (64B, 49, 96)

    # merge windows
    attn_windows = attn_windows.view(-1, self.window_size, self.window_size, C)  (64B, 7, 7, 96)
    shifted_x = window_reverse(attn_windows, self.window_size, H, W)  # B H' W' C  (B, 56, 56, 96)

    # reverse cyclic shift
    if self.shift_size > 0:
        x = torch.roll(shifted_x, shifts=(self.shift_size, self.shift_size), dims=(1, 2))
    else:
        x = shifted_x (B, 56, 56, 96)
    x = x.view(B, H * W, C)  (B, 56x56, 96) = (B, 3136, 96)

    # FFN
    x = shortcut + self.drop_path(x)
    x = x + self.drop_path(self.mlp(self.norm2(x)))

    return x (B, 3136, 96)
```

```python
def window_partition(x, window_size):
    """
    Args:
        x: (B, H, W, C)
        window_size (int): window size

    Returns:
        windows: (num_windows*B, window_size, window_size, C)
    """
    B, H, W, C = x.shape
    x = x.view(B, H // window_size, window_size, W // window_size, window_size, C)
    windows = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(-1, window_size, window_size, C)
    return windows
```

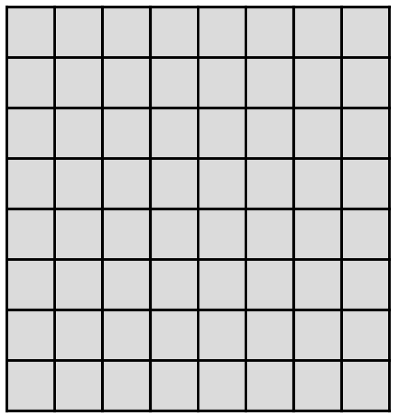
windows = 8x8

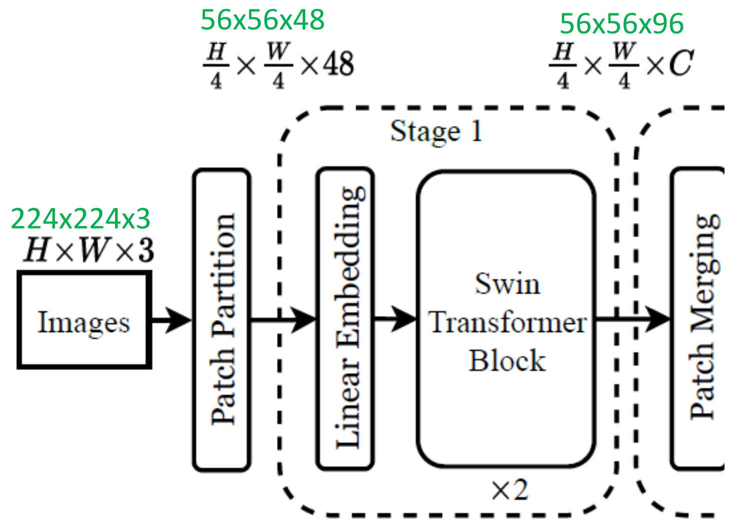




```python
class SwinTransformerBlock(nn.Module):

        # W-MSA/SW-MSA
        attn_windows = self.attn(x_windows, mask=self.attn_mask)  # nW*B, window_size*window_size, C (64B, 49, 96)
```

windows = 8x8





```
# W-MSA/SW-MSA
attn_windows = self.attn(x_windows, mask=self.attn_mask)  # nW*B, window_size*window_size, C (64B, 49, 96)
```

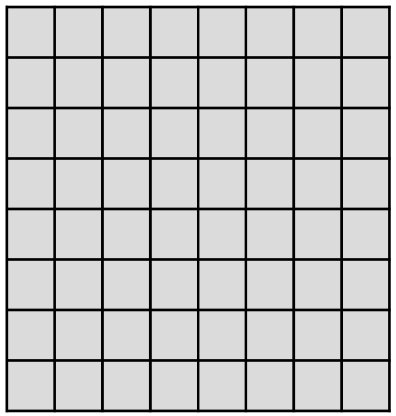
```
class WindowAttention(nn.Module):
    r""" Window based multi-head self attention (W-MSA) module with relative position bias.
    It supports both of shifted and non-shifted window.

    Args:
        dim (int): Number of input channels.
        window_size (tuple[int]): The height and width of the window.
        num_heads (int): Number of attention heads.
        qkv_bias (bool, optional):  If True, add a learnable bias to query, key, value. Default: True
        qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set
        attn_drop (float, optional): Dropout ratio of attention weight. Default: 0.0
        proj_drop (float, optional): Dropout ratio of output. Default: 0.0
    """

    def __init__(self, dim, window_size, num_heads, qkv_bias=True, qk_scale=None, attn_drop=0., proj_drop=0.):

        super().__init__()
        self.dim = dim
        self.window_size = window_size  # Wh, Ww
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim ** -0.5

        # define a parameter table of relative position bias
        self.relative_position_bias_table = nn.Parameter(
            torch.zeros((2 * window_size[0] - 1) * (2 * window_size[1] - 1), num_heads))  # 2*Wh-1 * 2*Ww-1, nH

        # get pair-wise relative position index for each token inside the window
        coords_h = torch.arange(self.window_size[0])
        coords_w = torch.arange(self.window_size[1])
        coords = torch.stack(torch.meshgrid([coords_h, coords_w]))  # 2, Wh, Ww
        coords_flatten = torch.flatten(coords, 1)  # 2, Wh*Ww
        relative_coords = coords_flatten[:, :, None] - coords_flatten[:, None, :]  # 2, Wh*Ww, Wh*Ww
        relative_coords = relative_coords.permute(1, 2, 0).contiguous()  # Wh*Ww, Wh*Ww, 2
        relative_coords[:, :, 0] += self.window_size[0] - 1  # shift to start from 0
        relative_coords[:, :, 1] += self.window_size[1] - 1
        relative_coords[:, :, 0] *= 2 * self.window_size[1] - 1
        relative_position_index = relative_coords.sum(-1)  # Wh*Ww, Wh*Ww
        self.register_buffer("relative_position_index", relative_position_index)

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

        trunc_normal_(self.relative_position_bias_table, std=.02)
        self.softmax = nn.Softmax(dim=-1)
```

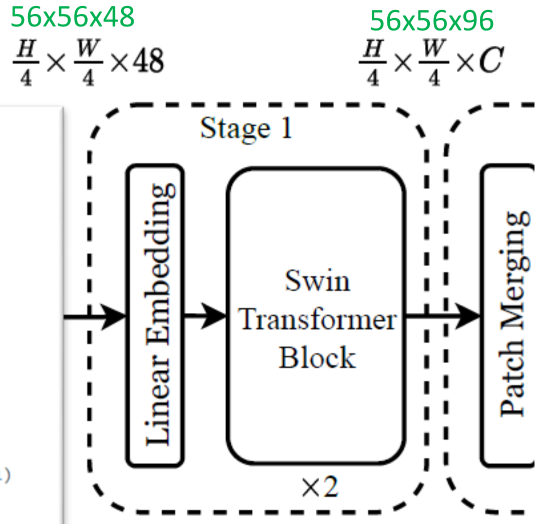
```
def forward(self, x, mask=None):
    """
    Args:
        x: input features with shape of (num_windows*B, N, C)
        mask: (0/-inf) mask with shape of (num_windows, Wh*Ww, Wh*Ww) or None
    """
    B_, N, C = x.shape   (64B, 49, 96)
    qkv = self.qkv(x).reshape(B_, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)
    q, k, v = qkv[0], qkv[1], qkv[2]  # make torchscript happy (cannot use tensor as tuple)
(64B, 3, 49, 32)          (64B, 49, 96x3)     (64B, 49, 3, 3 32)     (3,64B, 3, 49, 32)

    q = q * self.scale
    attn = (q @ k.transpose(-2, -1))
(64B, 3, 49, 49)

    relative_position_bias = self.relative_position_bias_table[self.relative_position_index.view(-1)].view(
        self.window_size[0] * self.window_size[1], self.window_size[0] * self.window_size[1], -1)  # Wh*Ww,Wh*Ww,nH
    relative_position_bias = relative_position_bias.permute(2, 0, 1).contiguous()  # nH, Wh*Ww, Wh*Ww
    attn = attn + relative_position_bias.unsqueeze(0)
(3, 49, 49)

    if mask is not None:
        nW = mask.shape[0]
        attn = attn.view(B_ // nW, nW, self.num_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)
        attn = attn.view(-1, self.num_heads, N, N)
        attn = self.softmax(attn)
    else:
        attn = self.softmax(attn)
(64B, 3, 49, 49)

    attn = self.attn_drop(attn)
                 (64B, 3, 49, 32)  (64B, 49, 3, 32)  (64B, 49, 96)
    x = (attn @ v).transpose(1, 2).reshape(B_, N, C)
    x = self.proj(x)   (64B, 49, 96)
    x = self.proj_drop(x)
    return x
```

```python
class SwinTransformerBlock(nn.Module):

def forward(self, x):
    H, W = self.input_resolution 56, 56
    B, L, C = x.shape  (B, 3136, 96)
    assert L == H * W, "input feature has wrong size"

    shortcut = x
    x = self.norm1(x)(B, 3136, 96)
    x = x.view(B, H, W, C) (B, 56, 56, 96)

    # cyclic shift
    if self.shift_size > 0:
        shifted_x = torch.roll(x, shifts=(-self.shift_size, -self.shift_size), dims=(1, 2))
    else:
        shifted_x = x  (B, 56, 56, 96)

    # partition windows
    x_windows = window_partition(shifted_x, self.window_size)  # nW*B, window_size, window_size, C  (64B, 7, 7, 96)
    x_windows = x_windows.view(-1, self.window_size * self.window_size, C)  # nW*B, window_size*window_size, C (64B, 49, 96)

    # W-MSA/SW-MSA
    attn_windows = self.attn(x_windows, mask=self.attn_mask)  # nW*B, window_size*window_size, C (64B, 49, 96)

    # merge windows
    attn_windows = attn_windows.view(-1, self.window_size, self.window_size, C)  (64B, 7, 7, 96)
    shifted_x = window_reverse(attn_windows, self.window_size, H, W)  # B H' W' C  (B, 56, 56, 96)

    # reverse cyclic shift
    if self.shift_size > 0:
        x = torch.roll(shifted_x, shifts=(self.shift_size, self.shift_size), dims=(1, 2))
    else:
        x = shifted_x (B, 56, 56, 96)
    x = x.view(B, H * W, C)  (B, 56x56, 96) = (B, 3136, 96)

    # FFN
    x = shortcut + self.drop_path(x)
    x = x + self.drop_path(self.mlp(self.norm2(x)))

    return x (B, 3136, 96)
```

```python
def window_reverse(windows, window_size, H, W):
    """
    Args:
        windows: (num_windows*B, window_size, window_size, C)
        window_size (int): Window size
        H (int): Height of image
        W (int): Width of image

    Returns:
        x: (B, H, W, C)
    """
    B = int(windows.shape[0] / (H * W / window_size / window_size))
    x = windows.view(B, H // window_size, W // window_size, window_size, window_size, -1)
    x = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(B, H, W, -1)
    return x
```

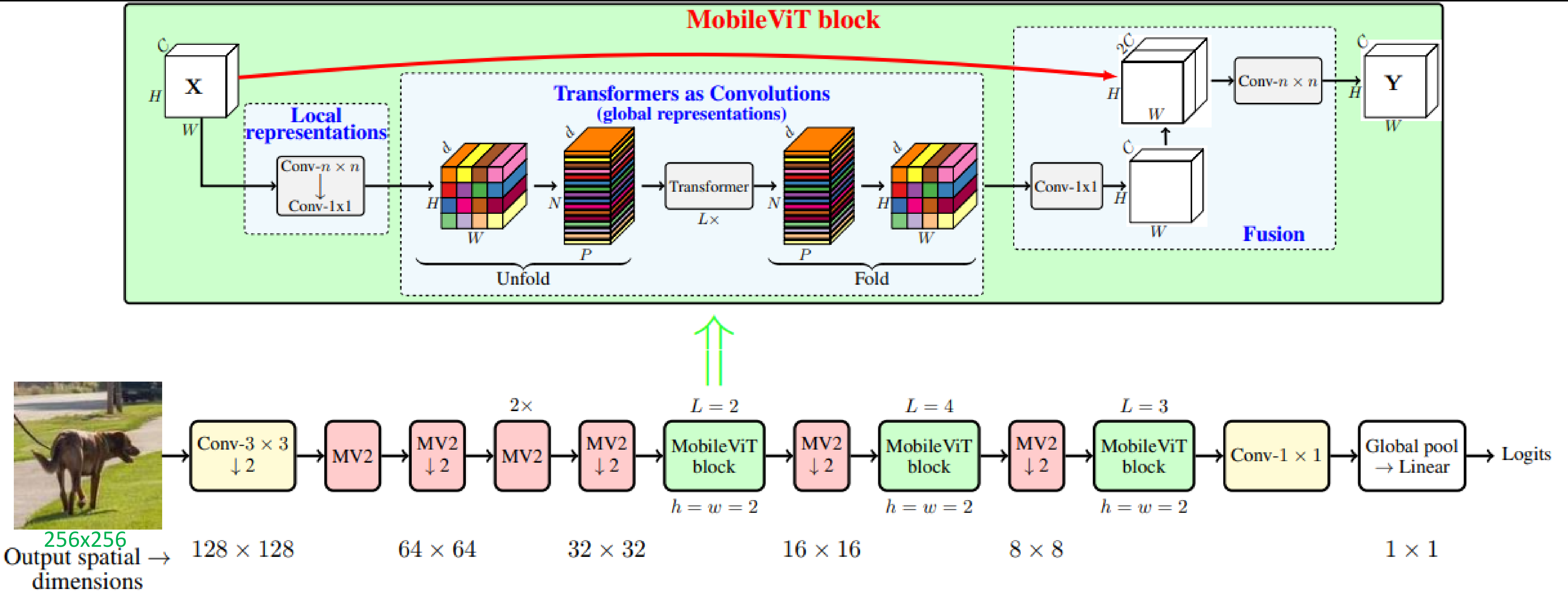
windows = 8x8





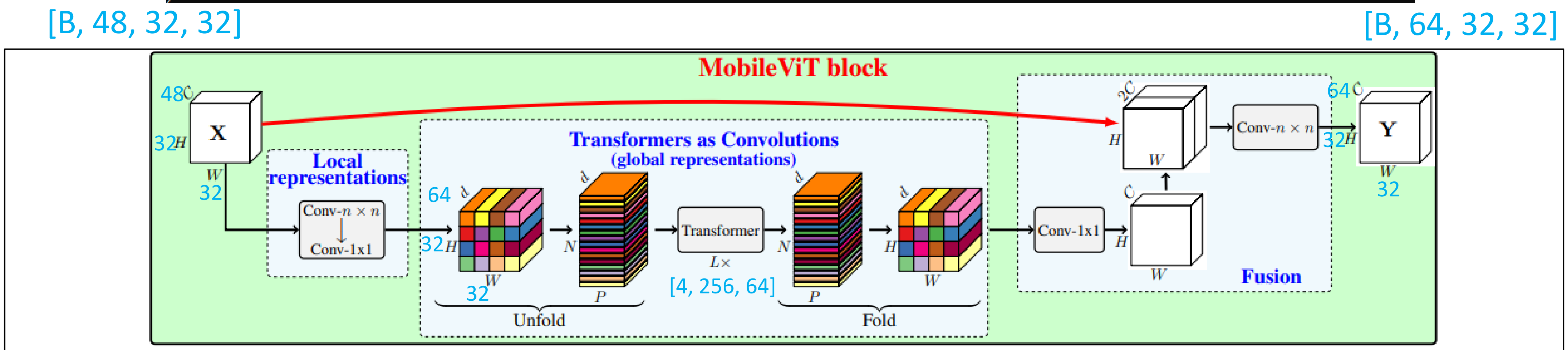
Outline

- Swin Transformer
- **MobileVit**

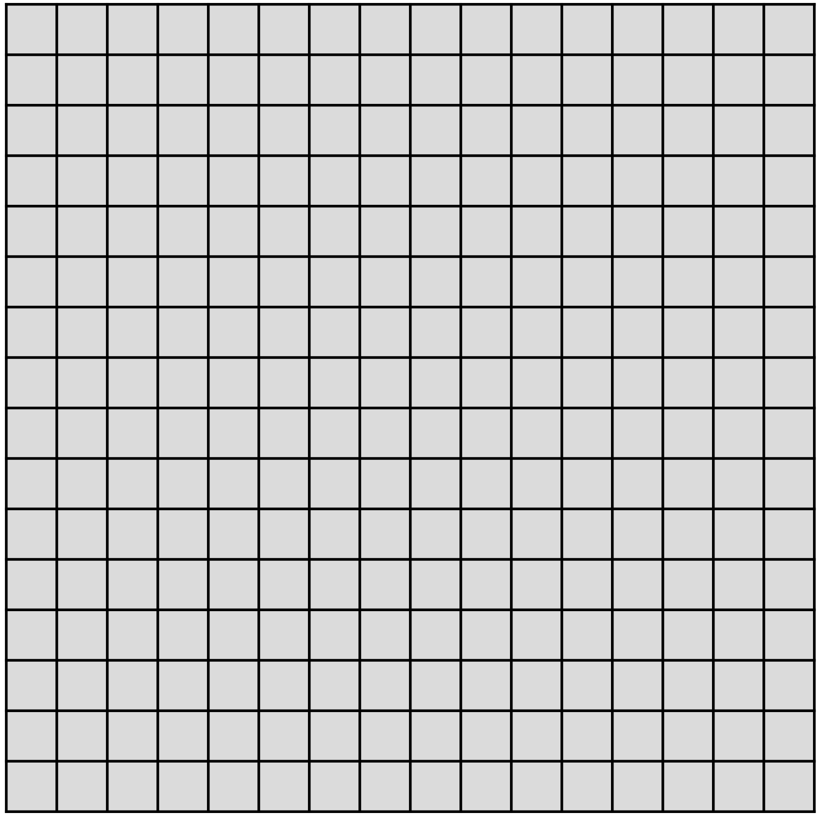


(b) **MobileViT**. Here, Conv-$n \times n$ in the MobileViT block represents a standard $n \times n$ convolution and MV2 refers to MobileNetv2 block. Blocks that perform down-sampling are marked with $\downarrow 2$.

```
MobileViT(
  (conv_1): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False, normalization=BatchNorm2d, activation=ReLU, bias=False)
  (layer_1): Sequential(
    (0): InvertedResidual(in_channels=16, out_channels=16, stride=1, exp=2, dilation=1)
  )
  (layer_2): Sequential(
    (0): InvertedResidual(in_channels=16, out_channels=24, stride=2, exp=2, dilation=1)
    (1): InvertedResidual(in_channels=24, out_channels=24, stride=1, exp=2, dilation=1)
    (2): InvertedResidual(in_channels=24, out_channels=24, stride=1, exp=2, dilation=1)
  )
  (layer_3): Sequential(
    (0): InvertedResidual(in_channels=24, out_channels=48, stride=2, exp=2, dilation=1)
    (1): MobileViTBlock(
        conv_in_dim=48, conv_out_dim=64, dilation=1, conv_ksize=3
        patch_h=2, patch_w=2
        transformer_in_dim=64, transformer_n_heads=4, transformer_ffn_dim=128, dropout=0.1, ffn_dropout=0.0, attn_dropout=0.1, blocks=2
    )
  )
  (layer_4): Sequential(
    (0): InvertedResidual(in_channels=48, out_channels=64, stride=2, exp=2, dilation=1)
    (1): MobileViTBlock(
        conv_in_dim=64, conv_out_dim=80, dilation=1, conv_ksize=3
        patch_h=2, patch_w=2
        transformer_in_dim=80, transformer_n_heads=4, transformer_ffn_dim=160, dropout=0.1, ffn_dropout=0.0, attn_dropout=0.1, blocks=4
    )
  )
  (layer_5): Sequential(
    (0): InvertedResidual(in_channels=64, out_channels=80, stride=2, exp=2, dilation=1)
    (1): MobileViTBlock(
        conv_in_dim=80, conv_out_dim=96, dilation=1, conv_ksize=3
        patch_h=2, patch_w=2
        transformer_in_dim=96, transformer_n_heads=4, transformer_ffn_dim=192, dropout=0.1, ffn_dropout=0.0, attn_dropout=0.1, blocks=3
    )
  )
  (conv_1x1_exp): Conv2d(80, 320, kernel_size=(1, 1), stride=(1, 1), bias=False, normalization=BatchNorm2d, activation=ReLU, bias=False)
  (classifier): Sequential(
    (global_pool): GlobalPool(type=mean)
    (fc): LinearLayer(in_features=320, out_features=1000, bias=True)
  )
)
```

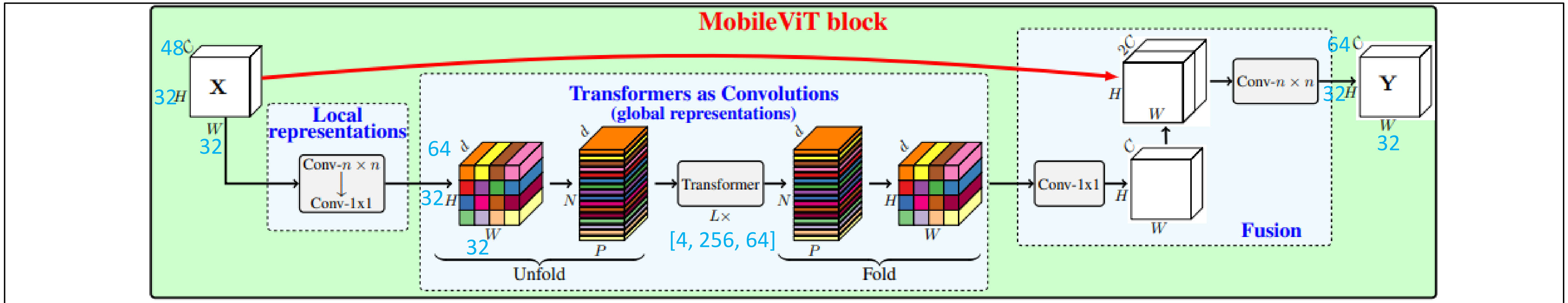


windows = 16x16, 2x2-sized window



[B, 48, 32, 32]

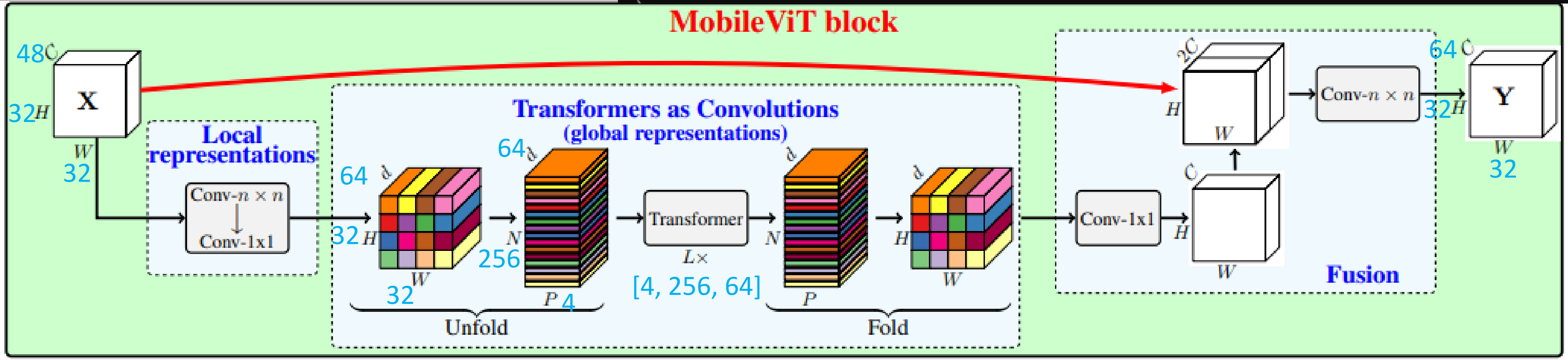
[B, 64, 32, 32]



```
def unfolding(self, feature_map: Tensor) -> Tuple[Tensor, Dict]:
    patch_w, patch_h = self.patch_w, self.patch_h
    patch_area = int(patch_w * patch_h)
    batch_size, in_channels, orig_h, orig_w = feature_map.shape [B, 64, 32, 32]

32  new_h = int(math.ceil(orig_h / self.patch_h) * self.patch_h)
32  new_w = int(math.ceil(orig_w / self.patch_w) * self.patch_w)

    interpolate = False
    if new_w != orig_w or new_h != orig_h:
        # Note: Padding can be done, but then it needs to be handled in attention function.
        feature_map = F.interpolate(feature_map, size=(new_h, new_w), mode="bilinear", align_corners=False)
        interpolate = True

    # number of patches along width and height
16  num_patch_w = new_w // patch_w # n_w
16  num_patch_h = new_h // patch_h # n_h
256 num_patches = num_patch_h * num_patch_w # N

    # [B, C, H, W] --> [B * C * n_h, p_h, n_w, p_w] [Bx64x16, 2, 16, 2]
    reshaped_fm = feature_map.reshape(batch_size * in_channels * num_patch_h, patch_h, num_patch_w, patch_w)
    # [B * C * n_h, p_h, n_w, p_w] --> [B * C * n_h, n_w, p_h, p_w] [Bx64x16, 16, 2, 2]
    transposed_fm = reshaped_fm.transpose(1, 2)
    # [B * C * n_h, n_w, p_h, p_w] --> [B, C, N, P] where P = p_h * p_w and N = n_h * n_w [B,64,256, 2x2]
    reshaped_fm = transposed_fm.reshape(batch_size, in_channels, num_patches, patch_area)
    # [B, C, N, P] --> [B, P, N, C] [B, 4, 256, 64]
    transposed_fm = reshaped_fm.transpose(1, 3)
    # [B, P, N, C] --> [BP, N, C] [4B, 256, 64]
    patches = transposed_fm.reshape(batch_size * patch_area, num_patches, -1)

    info_dict = {
        "orig_size": (orig_h, orig_w),
        "batch_size": batch_size,
        "interpolate": interpolate,
        "total_patches": num_patches,
        "num_patches_w": num_patch_w,
        "num_patches_h": num_patch_h
    }

    return patches, info_dict
```

```
def folding(self, patches: Tensor, info_dict: Dict) -> Tensor:
    n_dim = patches.dim()
    assert n_dim == 3, "Tensor should be of shape BPxNxC. Got: {}".format(patches.shape)
    # [BP, N, C] --> [B, P, N, C] [B, 4, 256, 64]
    patches = patches.contiguous().view(info_dict["batch_size"], self.patch_area, info_dict["total_patches"], -1)

    batch_size, pixels, num_patches, channels = patches.size()
    num_patch_h = info_dict["num_patches_h"]
    num_patch_w = info_dict["num_patches_w"]

    # [B, P, N, C] --> [B, C, N, P] [B, 64, 256, 4]
    patches = patches.transpose(1, 3)

    # [B, C, N, P] --> [B*C*n_h, n_w, p_h, p_w] [Bx64x16, 16, 2, 2]
    feature_map = patches.reshape(batch_size * channels * num_patch_h, num_patch_w, self.patch_h, self.patch_w)
    # [B*C*n_h, n_w, p_h, p_w] --> [B*C*n_h, p_h, n_w, p_w] [Bx64x16, 2, 16, 2]
    feature_map = feature_map.transpose(1, 2)
    # [B*C*n_h, p_h, n_w, p_w] --> [B, C, H, W] [B, 64, 32, 32]
    feature_map = feature_map.reshape(batch_size, channels, num_patch_h * self.patch_h, num_patch_w * self.patch_w)
    if info_dict["interpolate"]:
        feature_map = F.interpolate(feature_map, size=info_dict["orig_size"], mode="bilinear", align_corners=False)
    return feature_map
```

```
(1): MobileViTBlock(
    conv_in_dim=48, conv_out_dim=64, dilation=1, conv_ksize=3
    patch_h=2, patch_w=2
    transformer_in_dim=64, transformer_n_heads=4, transformer_ffn_dim=128, dropout=0.1, ffn_dropout=0.0, attn_dropout=0.1, blocks=2
)
```



Thank You