

# SparseFormer: Detecting Objects in HRW Shots via Sparse Vision Transformer –Supplementary Material–

Anonymous Author(s)

This supplementary material contains additional details of the main manuscript. Section 1 presents additional details of the models and training strategies. Section 2 further explains the Algorithm 1. Section 3 evaluate the reasonableness of the keeping ratio formulation. Section 4 shows more visualization results to prove the effectiveness of SparseFormer. Section 5 provide the PyTorch-like code for SparseFormer.

## 1 DETAILS OF MODELS AND TRAINING

### 1.1 Dataset Preparation

**PANDA dataset** [4]. PAN [1] prepares the data for training by using a sliding window of 2,048×1,024 pixels to decompose the image which is downsampling with a factor of 4. In this paper, we use a multi-scale pre-process to accelerate training convergence and boost performance. Firstly, we cropped the patches according to a 16×16 grid, an 8×8 grid, and a 4×4 grid on the original image and filter the patch which has no persons. Then, all of the patches are resized to the same resolution for training. In other words, our pre-process is using the sliding window on images downsampled at three scales.

**DOTA dataset** [5]. We crop the original images into the patches of 1024×1024 with a stride of 824, which means the pixel overlap between two adjacent patches is 200. In our experiment, we do not use multi-scale training and testing and compared with the current state-of-the-art methods which have the same setting.

### 1.2 Training and Inference

The observations in [2] show that training from scratch is no worse than the fine-tuning counterpart. Thus, we train the detector with 36 epochs instead of pretraining on ImageNet. This also ensures that we have a fair comparison with Swin Transformer, the performance improvement comes from the novel design rather than better pre-train weight. Benefiting from the sparse architecture, our model can be easily trained on GeForce RTX 3090Ti GPUs.

During inference, we divided large-resolution images into patches with a resolution of 6000×3600 and an overlap of 600 pixels between patches for the PANDA dataset, and with a resolution of 1024×1024 and an overlap of 200 pixels between patches for the DOTA dataset.

## 2 ALGORITHM EXPLANATION FOR C-NMS

Suppose there are only two candidate box sets  $\mathcal{B}_1$  and  $\mathcal{B}_2$  for simplicity, indicating the predictions of the object detector on the two overlapped sliding windows. We first employ the conventional score-based NMS post-processing on both box sets  $\mathcal{B}_1$  and  $\mathcal{B}_2$  to locally eliminate redundant boxes with low confidence. This process results in two locally-suppressed outcomes:  $\mathcal{B}'_1$  and  $\mathcal{B}'_2$ . Then, for the cross-window box suppression, we unite  $\mathcal{B}'_1$  and  $\mathcal{B}'_2$  to form a union box set  $\mathcal{B}$ . In order to retain the boxes covering the complete

### Algorithm 1: Cross-window NMS (C-NMS)

---

**Variables** :  $\mathcal{B}_1, \mathcal{B}_2$  are candidate box sets from two windows,  $\tau$  is the C-NMS threshold;

**Functions** :  $\text{NMS}(\cdot)$  is the conventional NMS;  
 $\text{AREA}(\cdot)$  calculates the area of a box;

```

1  $\mathcal{B}'_1 \leftarrow \text{NMS}(\mathcal{B}_1); \mathcal{B}'_2 \leftarrow \text{NMS}(\mathcal{B}_2);$ 
2  $\mathcal{B} \leftarrow \mathcal{B}'_1 \cup \mathcal{B}'_2; \mathcal{B}' \leftarrow \emptyset;$ 
3 while  $\mathcal{B} \neq \emptyset$  do
4    $m \leftarrow \text{argmax}_i \text{AREA}(b_i), \text{ s.t. } b_i \in \mathcal{B};$ 
5    $\mathcal{B}' \leftarrow \mathcal{B}' \cup \{b_m\}; \mathcal{B} \leftarrow \mathcal{B} - \{b_m\};$ 
6   for  $b_i \in \mathcal{B}$  do
7     if  $\text{IoU}(b_i, b_m) \geq \tau$  then
8        $\mathcal{B} \leftarrow \mathcal{B} - \{b_i\};$ 
9 return  $\mathcal{B}'$ ;
```

---

Table 1: Ablation studies on sparse ratio.

Method	Ratio	GFLOPs			AP			
		Fore	Back	All	Total	Small	Medium	Large
DyHead	0.1	5.91	31.24	37.15	75.1	31.6	71.4	85.4
	0.3	5.90	34.14	40.04	75.9	34.5	72.7	85.3
	0.5	6.16	40.59	46.75	76.5	34.7	73.5	85.5
	0.7	6.19	60.87	67.06	77.1	36.4	74.0	86.3
	1.0	6.21	117.9	124.1	78.2	44.2	75.4	86.3
DINO	0.1	6.74	44.77	51.51	75.6	33.5	75.1	82.4
	0.3	6.79	48.86	54.65	76.1	41.1	76.2	81.3
	0.5	6.84	52.53	59.37	76.7	42.5	77.2	80.9
	0.7	6.90	68.81	75.71	77.3	44.3	77.5	81.3
	1.0	7.06	134.2	141.3	78.0	53.0	77.8	81.9

targets, we design to use the area of boxes to suppress boxes located at overlapping regions. As in Algorithm 1, at each iteration, the box  $b_m$  with the largest area is removed from  $\mathcal{B}$  and append to the suppressed set  $\mathcal{B}'$ . We then remove all boxes  $b_i$  from  $\mathcal{B}$ , whose IoUs with  $b_m$  are greater than a pre-defined threshold  $\tau$ . The algorithm ends when the box union set  $\mathcal{B}$  is empty. This strategy is effective in selecting optimal boxes for HRW shot object detection.

## 3 STUDY ON KEEPING RATIO.

Discarding the worthless grids is an important part of our strategy. So we conduct experiments to study the effect of the keeping ratio of the grids on the final performance. Table 1 shows the results and the keeping ratio  $k$  is formulated as  $[k, k^2, k^3, k^4]$  for each stage. As the features become more and more sparse, we can see a significant drop in FLOPs but an insignificant drop in accuracy. In order to evaluate the reasonableness of this formulation, we conduct experiments on new parameters containing  $[k, 1, 1, 1]$  denoted as “Stage-1”, and  $[1, 1, 1, k]$  denoted as “Stage-4” with  $k \in \{0.1, 0.5, 1\}$ . The results are shown in Figure 1. From the perspective of the GFLOPs, discarding

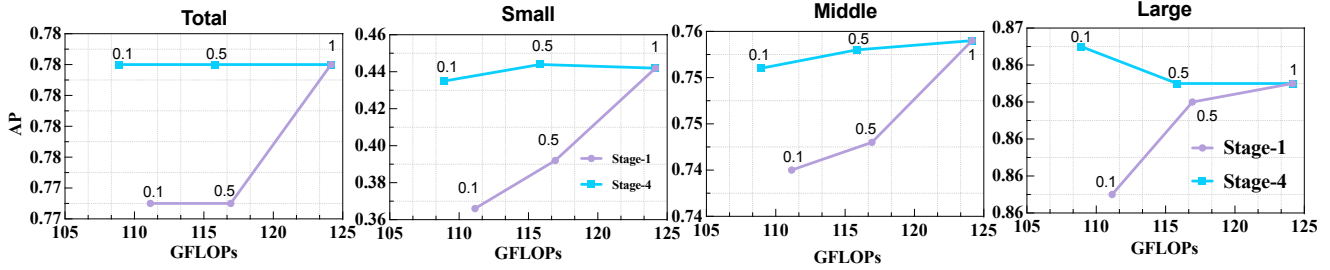


Figure 1: Effect of keeping ratio on different stage-1 and stage-4.

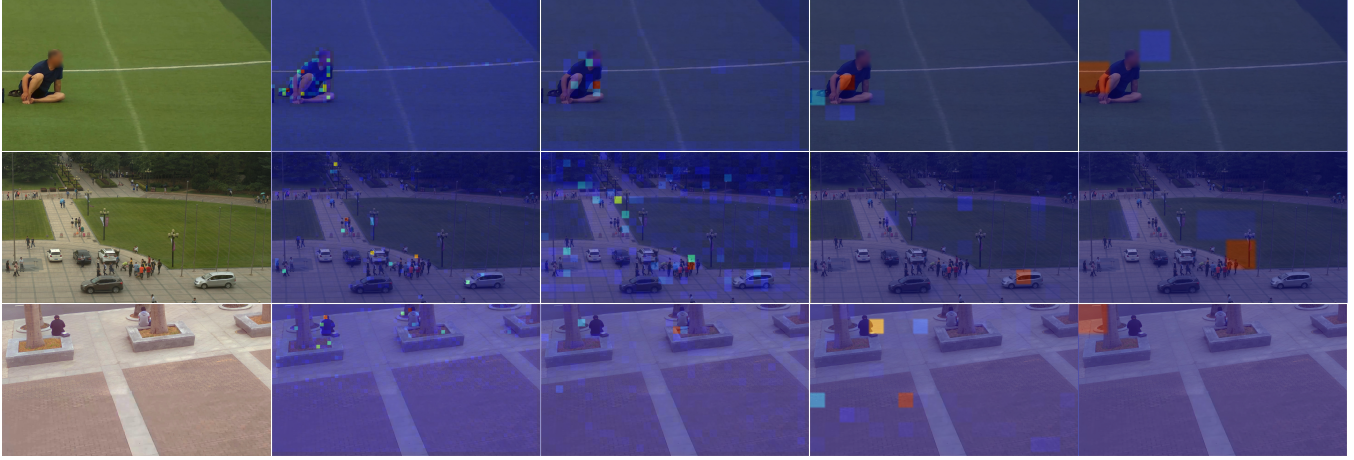


Figure 2: Visualization of the sparse window scores of four stages. The samples are from the PNADA dataset [4]. Highlighted patches mean higher scores. It is easy to observe that our model pays more attention to the details of foreground and complex regions.

the tokens in Stage-4 can save more computation than in Stage-1. It is because the most computation of window self-attention is on the MLP layer and the feature in Stage-4 has larger dimensions. To our surprise, with more sparse windows, Stage-4 can still perform well on total AP. From the perspective of objects of different scales, more sparse windows (lower  $k$ ) on Stage-4 can drop the performance on small and middle objects but improve the AP on large objects. The opposite is true for Stage-1. Therefore, The formulation of  $[k, k^2, k^3, k^4]$ , with more drop in Stage-4, seems a reasonable choice.

## 4 MODEL ANALYSIS

### 4.1 Visualization Results

To further investigate the behavior of the window sparsification, we visualize selected windows of each stage in Figure 2 to show the advantage of the reduction on the redundancy computation such as background and low-entropy foreground. The results also demonstrate that our sparse representation learning makes sense and we believe this will become a promising research direction for not only future detection but any other vision tasks. The selected windows of each stage on the DOTA dataset are shown in Figure 3. The results show that when there is more background and less foreground, our local self-attention is more concentrated

in the foreground region. When it is all background, the scores are relatively even across all regions. To our surprise, Stage-2 usually generates a more even score map which might show its importance among the four stages. This is also an important basis for further exploring how to design more efficient sparse architecture.

### 4.2 Qualitative comparison of detection results.

We present qualitative results in Figure 4 to compare the SparseFormer and Swin Transformer. We observe that through the token sparsification on the background, SparseFormer has lower false positive examples.

## 5 CODE IN PYTORCH

The local block is an essential module for reducing the computation, and it can be easily implemented by inserting a plug-in module into the original network. We provide the PyTorch-like pseudocode in Algorithm 2 and Algorithm 3 associated with the Local self-attention block. Algorithm 4 further shows the SparseFormer block built with these modules. It can be seen from the pseudocode that our sparse design is not limited to the Transformer but also could modify the convolution neural network like ResNet [3] by adding

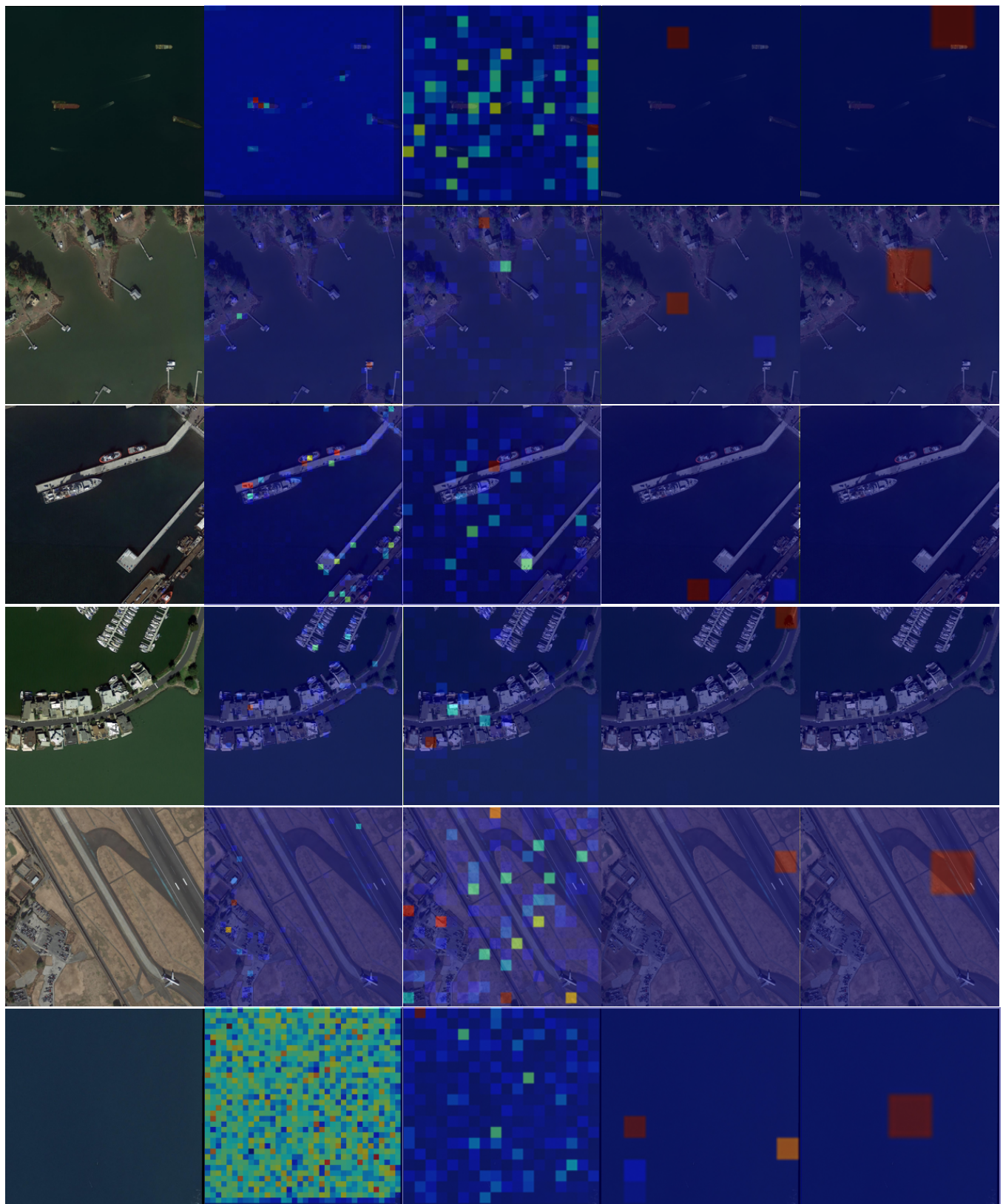


Figure 3: Visualization of the sparse window scores of each stage on DOTA dataset. The samples are from the DOTA dataset [5].





Figure 4: Qualitative results. We compare our SparseFormer to the Swin TransFormer and our model has fewer false positives thanks to a sparse sampling of the background.

windows. The whole code will be publicly released after the review process.

## REFERENCES

- [1] Jiahao Fan, Huabin Liu, Wenjie Yang, John See, Aixin Zhang, and Weiyao Lin. 2022. Speed Up Object Detection on Gigapixel-Level Images With Patch Arrangement. In *CVPR*.
- [2] Kaiming He, Ross Girshick, and Piotr Dollár. 2019. Rethinking imagenet pre-training. In *ICCV*.
- [3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*.
- [4] Xueyang Wang, Xiya Zhang, Yinheng Zhu, Yuchen Guo, Xiaoyun Yuan, Liuyu Xiang, Zerun Wang, Guiguang Ding, David Brady, Qionghai Dai, et al. 2020. Panda: A gigapixel-level human-centric video dataset. In *CVPR*.
- [5] Gui-Song Xia, Xiang Bai, Jian Ding, Zhen Zhu, Serge Belongie, Jiebo Luo, Mihai Datcu, Marcello Pelillo, and Liangpei Zhang. 2018. DOTA: A large-scale dataset for object detection in aerial images. In *CVPR*.

---

**Algorithm 2:** Sparse Window based Multi-head Self-attention, PyTorch-like Pseudocode

---

```

class SparseWindowMSA(BaseModule):

    def forward(self, query, hw_shape, keep_token_indices):
        B, L, C = query.shape
        H, W = hw_shape
        K = keep_token_indices.shape[1]
        query = query.view(B, H, W, C)

        # pad feature maps to multiples of window size
        pad_r = (self.window_size - W % self.window_size) % self.window_size
        pad_b = (self.window_size - H % self.window_size) % self.window_size
        query = F.pad(query, (0, 0, 0, pad_r, 0, pad_b))
        H_pad, W_pad = query.shape[1], query.shape[2]

        # cyclic shift
        shifted_query = shift_function(query)

        # B, nW, window_size, window_size, C
        query_windows = self.window_partition(shifted_query)

        # *****plug-in*****
        # sparse windows:nW->K (B, K, window_size, window_size, C)
        query_windows_sparse = torch.gather(query_windows, sparse_grad=True, dim=1, index=keep_token_indices)
        query_windows_sparse = query_windows_sparse.view(-1, self.window_size**2, C)
        # *****

        # W-MSA/SW-MSA (B*K, window_size*window_size, C)
        attn_windows_sparse = self.w_msa(query_windows_sparse, mask=attn_mask_sparse)

        # *****plug-in*****
        # (B, K, window_size, window_size, C)
        attn_windows_sparse = attn_windows_sparse.view(B, -1, self.window_size, self.window_size, C)
        # inverse sparse:K->nW (B, nW, window_size, window_size, C)
        attn_windows = query_windows.scatter(src=attn_windows_sparse, dim=1, index=keep_token_indices)
        # *****

        # (B, H_pad, W_pad, C)
        shifted_x = self.window_reverse(attn_windows, H_pad, W_pad)

        # reverse cyclic shift
        x = shift_function(shifted_x)

        if pad_r > 0 or pad_b:
            x = x[:, :H, :W, :].contiguous()
            x = x.view(B, H * W, C)
            x = self.drop(x)
        return x

```

---

**Algorithm 3: Sparse Feed-forward Network, PyTorch-like Pseudocode**


---

```

from module import FFN, BaseModule

class SparseFFN(BaseModule):

    def forward(self, x, hw_shape, keep_token_indices, identity):

        B, L, C = x.shape
        H, W = hw_shape
        K = keep_token_indices.shape[1]
        query = x.view(B, H, W, C)

        # pad feature maps to multiples of window size
        pad_r = (self.window_size - W % self.window_size) % self.window_size
        pad_b = (self.window_size - H % self.window_size) % self.window_size
        query = F.pad(query, (0, 0, 0, pad_r, 0, pad_b))
        H_pad = query.shape[1]
        W_pad = query.shape[2]

        # B, nW, window_size, window_size,
        query_windows = self.window_partition(query)

        # *****plug-in*****
        # sparse windows:nW->K (B, K, window_size, window_size, C)
        query_windows_sparse = torch.gather(query_windows, sparse_grad=True, dim=1, index=keep_token_indices)
        # *****

        output = self.ffn(query_windows_sparse)

        # *****plug-in*****
        # inverse sparse:K->nW (B, nW, window_size, window_size, C)
        x = query_windows_sparse.scatter(src=output, dim=1, index=keep_token_indices)
        # *****

        # merge windows
        x = x.view(-1, self.window_size, self.window_size, C)
        x = self.window_reverse(x, H_pad, W_pad)
        if pad_r > 0 or pad_b:
            x = x[:, :H, :W, :].contiguous()
        x = x.view(B, H * W, C)

        if not self.add_identity:
            return x
        return x + identity

```

---

---

**Algorithm 4: Local Self-attention Block, PyTorch-like Pseudocode**

---

```

class LocalBlock(BaseModule):
    def __init__(self,
        embed_dims,
        num_heads,
        feedforward_channels,
        window_size=7,
        shift=False,
        qkv_bias=True,
        qk_scale=None,
        drop_rate=0.,
        attn_drop_rate=0.,
        drop_path_rate=0.,
        act_cfg=dict(type='GELU'),
        norm_cfg=dict(type='LN'),
        init_cfg=None):

        super(LocalBlock, self).__init__()

        self.init_cfg = init_cfg
        self.with_cp = with_cp
        self.window_size = window_size

        self.norm1 = build_norm_layer(norm_cfg, embed_dims)[1]
        self.attn = SparseWindowMSA(
            embed_dims=embed_dims,
            num_heads=num_heads,
            window_size=window_size,
            shift_size=window_size // 2 if shift else 0,
            qkv_bias=qkv_bias,
            qk_scale=qk_scale,
            attn_drop_rate=attn_drop_rate,
            proj_drop_rate=drop_rate,
            dropout_layer=dict(type='DropPath', drop_prob=drop_path_rate),
            init_cfg=None)

        self.norm2 = build_norm_layer(norm_cfg, embed_dims)[1]
        self.ffn = SparseFFN(
            embed_dims=embed_dims,
            feedforward_channels=feedforward_channels,
            num_fcs=2,
            ffn_drop=drop_rate,
            dropout_layer=dict(type='DropPath', drop_prob=drop_path_rate),
            act_cfg=act_cfg,
            add_identity=False,
            init_cfg=None)

    def forward(self, x, hw_shape, keep_token_indices):
        identity = x
        x = self.norm1(x)
        x = self.attn(x, hw_shape, keep_token_indices)

        x = x + identity

        identity = x
        x = self.norm2(x)
        x = self.ffn(x, hw_shape, keep_token_indices, identity=identity)

        return x

```

---