CVSformer: Cross-View Synthesis Transformer for Semantic Scene Completion –Supplementary Material–

Haotian Dong[1]*, Enhui Ma[1]*, Lubo Wang[1], Miaohui Wang[2], Wuyuan Xie[2], Qing Guo[3], Ping Li[4], Lingyu Liang[5]†, Kairui Yang[6], Di Lin[1]†

[1] Tianjin University, [2] Shenzhen University, [3] The Hong Kong Polytechnic University, [4] IHPC and CFAR, Agency for Science, Technology and Research, Singapore, [5] Pazhou Lab, South China University of Technology, [6] Alibaba Damo Academy

1. Introduction

This supplementary material presents: (1) extra ablation analysis of our approach; (2) visualization results of competitive methods (i.e., DDRNet [7], AICNet [6], SISNet [2]) and CVSformer on NYU [9] and NYUCAD [5] datasets; (3) code segments of MVFS and CVTr.

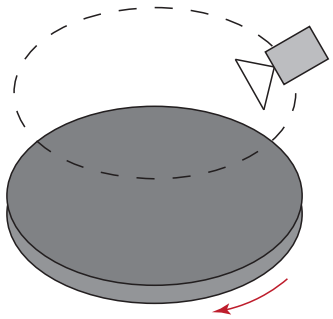


Figure 1. Experiment equipment.

MVFS	CVTr	IoU(%)	mIoU(%)
		72.9	51.3
✓		73.2	51.8
	✓	73.6	52.0
✓	✓	**73.7**	**52.6**

Table 1. Ablation study on MVFS and CVTr. The performances are evaluated on NYU dataset.

2. Extra Ablation Studies

Analysis of Synthetic View To verify that features synthesized by rotating convolutional kernels can capture multi-view information of an object, we designed the following experiments. As in Figure 1, our equipment is a high-precision rotating platform, and Azure Kinect DK camera with pre-calibrated and precise camera poses. We place a group of objects on the rotating platform and capture RGB and Depth images of the objects from different perspectives by controlling the rotation of the platform. During the experiment, we change the camera's pose by adjusting the distance and degree of the camera to the platform. The first and last images in each set are 150 degrees in-between, with 5 degrees between the two adjacent images. Specific information about the four sets of images is shown in Figure 2.

RGB images from different views, TSDF, and 2D-3D projection relationships are fed into CVSformer, which encodes the original feature map, and the feature map synthesized by MVFS, and then extracts the features of different samples from them. We use similarity matrix $\mathbf{S}$ to count the similar object features extracted from different views and different kernel rotations. We denote a set of object features with different rotations of the kernel as $\mathbb{K} = \{\mathbf{K}_i \mid i = 45°, 90°, 135°\}$ and a set of object features with different views as $\mathbb{V} = \{\mathbf{V}_j \mid j = o°, 5°, \cdots, 150°\}$. We use the Euclidean distance to calculate the similarity of the object features across the two sets. If $\mathbf{K}_i$ and $\mathbf{V}_j$ have the smallest distance, we add 1 to $\mathbf{S}_{i,j}$. With this approach, we can obtain the final similarity matrix. Figure 3 shows the visualization of the similarity matrix, where the darker color represents the higher similarity of the two features. We can find that the features obtained by rotating the convolution kernel are highly similar to the object features at a particular view degree. The camera poses influence this degree, which we believe is consistent with the realistic human understanding of multi-view information. The experiment results illustrate that the feature maps synthesized by rotating the convolution kernel can capture the object relationships for each voxel from multiple views.

Analysis of Multi-View Features In the first two rows of the Figure 4, we visualize the semantic voxels and the multi-view feature maps in the rotated kernels. The $0°$, $45°$ and

*Co-first authors. The names are listed in alphabetical order.
†Co-corresponding authors.

Figure 2. Captured images. The different color boxes represent the different objects selected.

Rotation	$\mathbf{0°}$		$\{\mathbf{0°, 15°, 30°, 45°}\}$		$\{\mathbf{0°, 30°, 60°, 90°}\}$		$\{\mathbf{0°, 45°, 90°, 135°}\}$	
	IoU(%)	**mIoU(%)**	**IoU(%)**	**mIoU(%)**	**IoU(%)**	**mIoU(%)**	**IoU(%)**	**mIoU(%)**
x	72.9	51.3	**73.8**	52.3	73.2	52.2	73.7	**52.6**
y			72.8	52.5	73.5	52.1	73.8	52.2
z			73.4	52.0	73.0	52.1	73.1	52.0
x, y, z			73.2	52.0	73.0	52.1	72.3	52.3

Table 2. Sensitivity to the interval between rotation degrees. We report the performances on NYU.

90° kernels together yield feature maps that capture the bed, floor, and partially-visible furniture. Intuitively, these feature maps hint at the furniture on the floor behind the bed. These feature maps fused by CVSformer help to reason the

Rotation	$\mathbf{0°}$		$\{\mathbf{0°, 45°}\}$		$\{\mathbf{0°, 45°, 90°}\}$		$\{\mathbf{0°, 45°, 90°, 135°}\}$	
	IoU(%)	**mIoU(%)**	**IoU(%)**	**mIoU(%)**	**IoU(%)**	**mIoU(%)**	**IoU(%)**	**mIoU(%)**
x	72.9	51.3	**73.8**	52.1	73.5	52.0	73.7	**52.6**
y			73.2	52.2	72.3	52.2	73.8	52.2
z			73.0	52.1	73.5	52.2	73.1	52.0
x, y, z			72.7	52.4	73.4	51.7	72.3	52.3

Table 3. Sensitivity to the view number. We report the performances on NYU.

Rotation	$\mathbf{0°}$		$\{\mathbf{0°, 45°}\}$		$\{\mathbf{0°, 45°, 90°}\}$		{0°,45°,90°,135°}	
	IoU(%)	**mIoU(%)**	**IoU(%)**	**mIoU(%)**	**IoU(%)**	**mIoU(%)**	**IoU(%)**	**mIoU(%)**
x	72.6	51.1	72.7	50.8	72.6	51.6	73.4	51.1
y			72.3	51.6	72.6	52.3	72.8	52.0
z			72.6	51.1	72.6	51.6	72.8	51.7
x, y, z			72.2	52.1	72.7	51.7	72.9	51.4

Table 4. Ablation study of MVFS with deformable 3D convolution on NYU.

Rotation Degrees	IoU(%)	mIoU(%)	Rotation Degrees	IoU(%)	mIoU(%)
$0°$	72.9	51.3	$0°$	72.9	51.3
$\{0°, 45°\}$	73.8	52.1	$\{-45°\ 0°\}$	73.5	52.0
$\{0°, 45°, 90°\}$	73.5	52.0	$\{-90°, -45°, 0°\}$	73.2	52.5
$\{0°, 45°, 90°, 135°\}$	73.7	52.6	$\{-135°, -90°, -45°, 0°\}$	73.4	51.7

Table 5. Study of symmetric degrees of rotation on NYU.

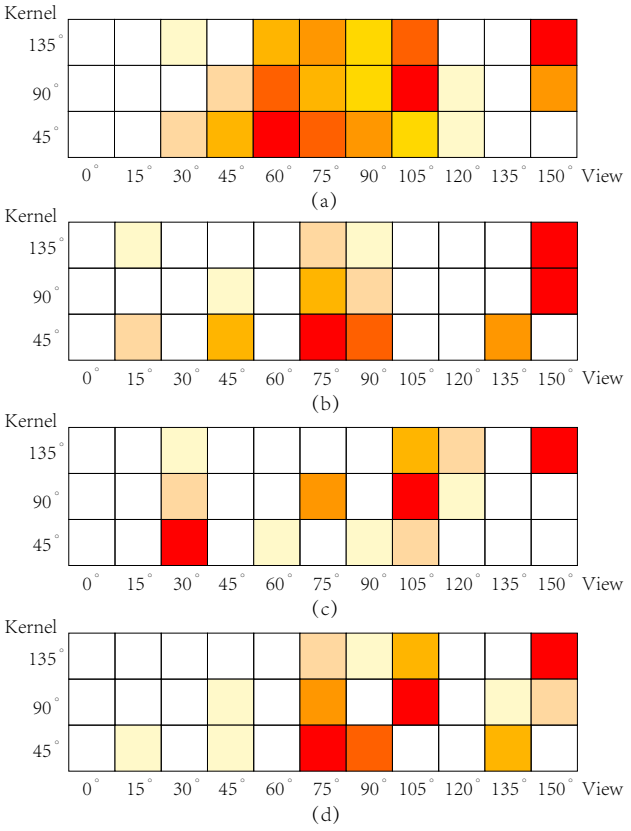


Figure 3. Similarity statistics. (a), (b), (c), (d) are the statistics results for row 1, 2, 3, 4 of Figure 2 respectively.

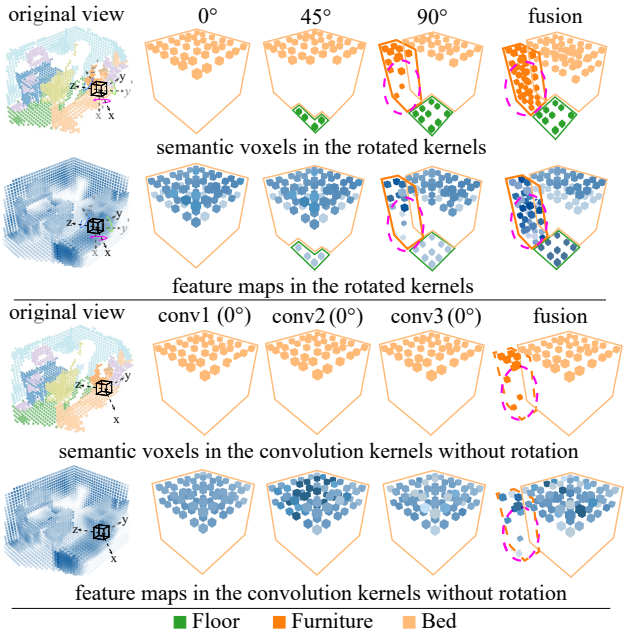


Figure 4. Semantic voxels and feature maps in the rotated kernels and convolutional kernels

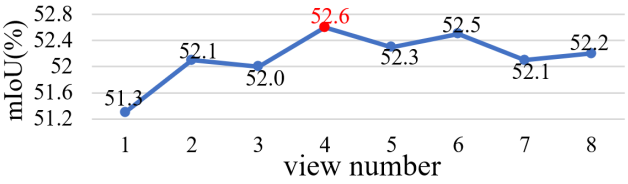


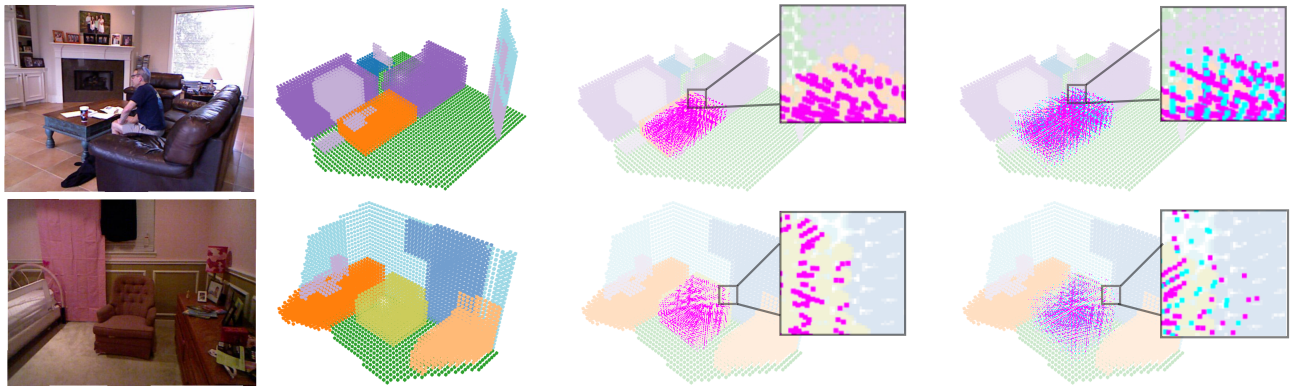
Figure 5. Various number of views. The performances are evaluated on NYU dataset.

occluded furniture (see the pink ellipse in the fusion). The last two rows show the semantic voxels and the feature maps in the separate kernels without rotation. Though the feature maps are learned from different latent spaces, they only suppress/highlight the bed structure, hardly considering richer context of objects. The fusion of these feature maps only captures the bed without reasoning the occluded furniture (see the right-most pink ellipses), which appears out of the kernels.

Tabel 1 evaluates the feature maps learned by the separate kernels without rotation (see the third row). These feature maps fused by the cross-view transformer yield lower performances than our method (see the last row). It demonstrates the effectiveness of rotating the kernels rather than learning distinct feature spaces.

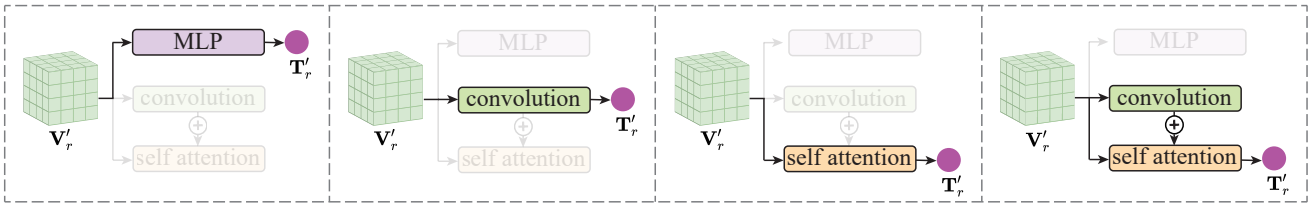
Sensitivity to Interval between Rotation Degrees We experiment with changing the combination of rotation degrees. Different combinations yield the corresponding synthetic-view feature maps. We report the performances in Tabel 2. By setting small intervals (e.g., $15°$) between the rotation degrees, the difference between synthetic views is insignificant, thus providing redundant but useless information. Larger interval (i.e., $45°$ by default) improves mIoU up to 52.6%.

Sensitivity to View Number We use MVFS to synthesize different numbers of views. Each view is associated with a rotation degree taken from the set $\{0°, 45°, 90°, 135°\}$. The results are list in Table 3. More views diversify the context of objects and lead to better performances. An empirical observation is that the rotation along the $x-$axis generally offers better results (see the first row). Furthermore, we also experiment with increasing the view number as the Figure 5.



(a) RGB (b) Scene (c) Deformable Kernels (d) Rotated Kernels

Figure 6. Deformable kernel and rotated kernel of specific category in different scenes. From left to right: (a) RGB images, (b) voxelized scenes, (c) deformable convolution kernels, (d) rotated kernels ($0°$ and $45°$). We find that deformable convolution kernels focus more on the individual object shapes, while rotated kernels cover more 3D locations for yielding richer context.



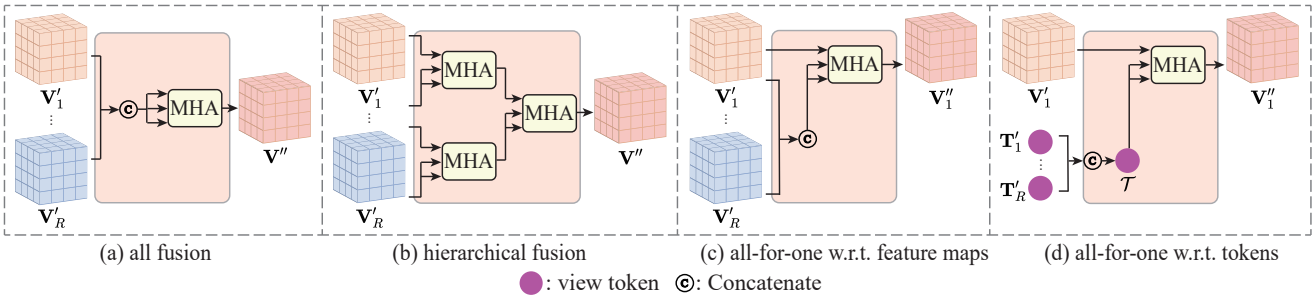
(a) MLP (b) convolution (c) self attention (d) self attention + convolution

Figure 7. Different strategies of learning view tokens in CVTr. The performances are evaluated on NYU dataset.

Empirically, four views (i.e., $0°, 45°, 90°, 135°$) lead to satisfactory results on NYU dataset. More views increase the complexity of network optimization in a reasonable training time, slightly degrading the performance.

Analysis of Deformable 3D Convolution In Table 4, we add the deformable 3D convolution module on top of MVFS, whose output (i.e., the synthetic-view feature maps) are fed into CVTr for learning the cross-view object relationships. It should be noted that the deformable 3D convolution degrades IoU and mIoU. As reported in the fourth column of the first row, mIoU even dropped to 51.1%. Note that richer context of objects is very important to the voxel-wise SSC task. However, deformable convolution focuses more on fitting the individual object shapes (see Figure 6). Thus, it is less powerful in terms of capturing the diverse object relationships. In contrast, our convolutional kernels with controllable rotations achieve more context of the surrounding objects.

Analysis of Symmetric Rotation Degrees In Table 5, we show the results of applying symmetric degrees in MVFS. The clockwise- and counterclockwise-rotations of same degrees make little difference. This may be because the sym-



(a) all fusion (b) hierarchical fusion (c) all-for-one w.r.t. feature maps (d) all-for-one w.r.t. tokens

: view token ⓒ: Concatenate

Figure 8. Illustration of different fusion schemes in CVTr. (a) all fusion, where the concatenation of all synthetic-view feature maps are fed into self-attention. (b) hierarchical fusion, where self attention is computed between two of all synthetic-view feature maps. (c) all-for-one w.r.t. feature maps, where the concatenation of all synthetic-view feature maps are the key and value of cross attention. (d) all-for-one w.r.t. tokens, where the concatenated view tokens learned by the transformer encoders are the key and value for feature enhancement.

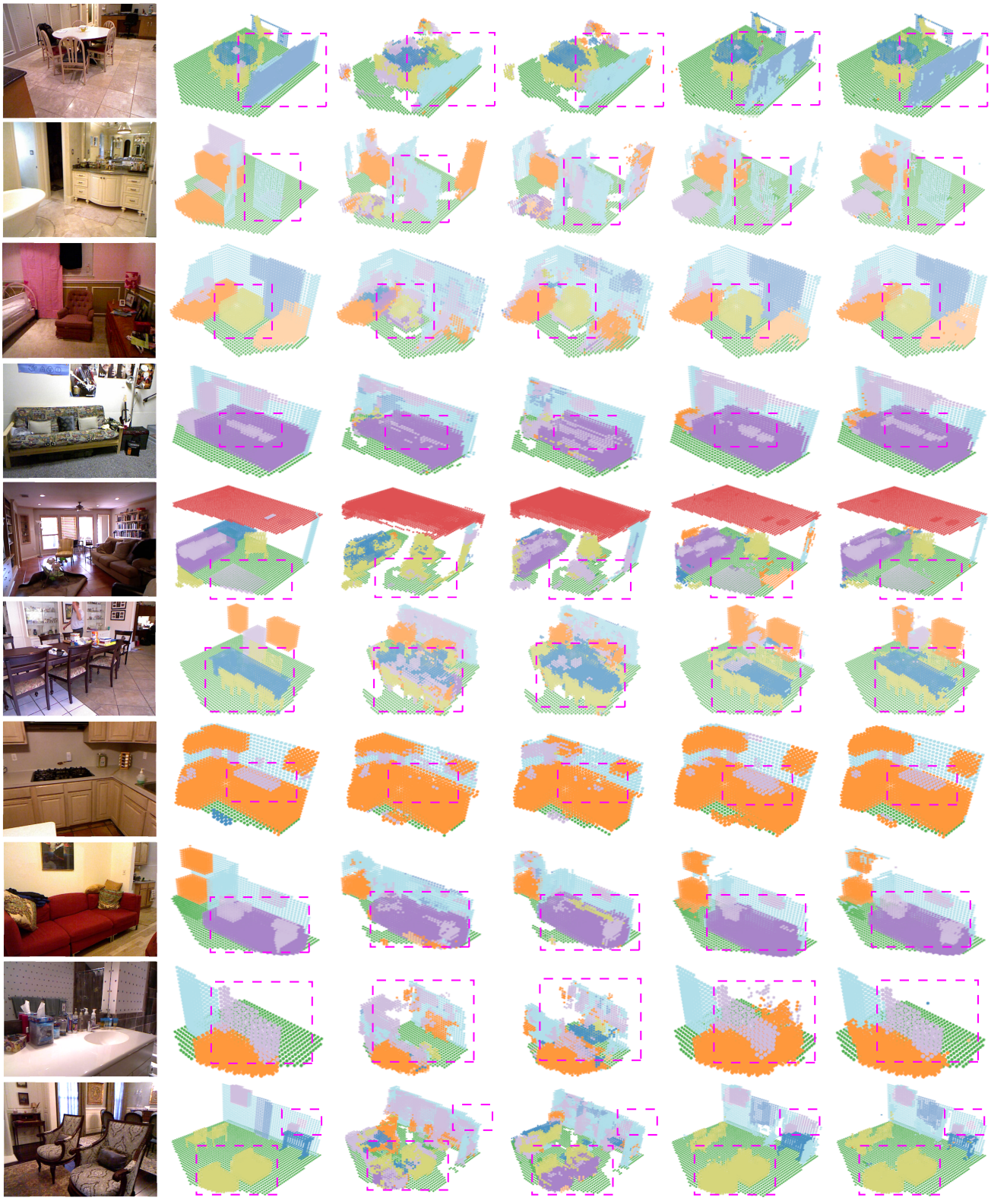


Figure 9. Completion results of different methods on NYU.

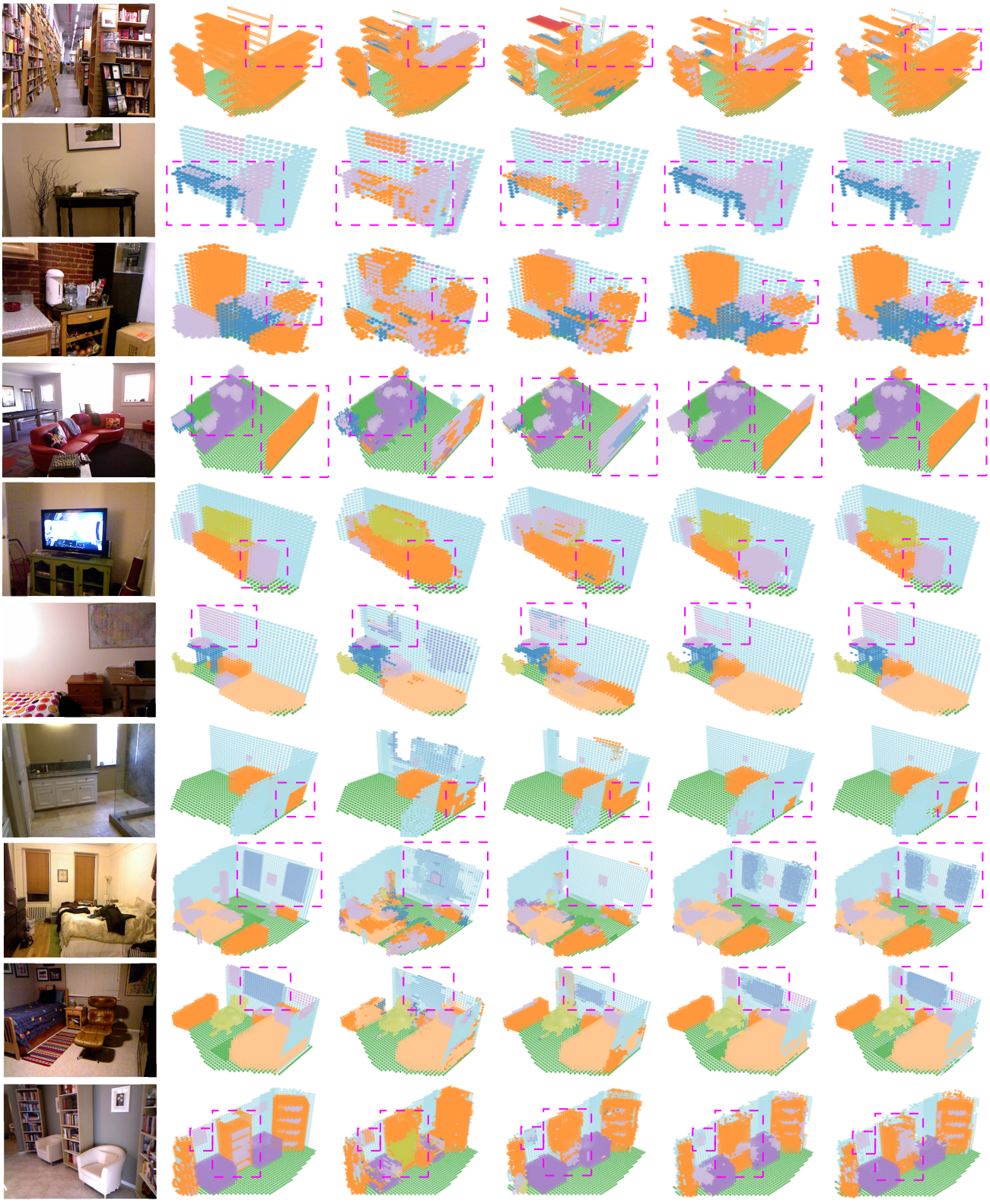


Figure 10. Completion results of different methods on NYUCAD.

metric rotations provide less novel information, given the man-made objects normally with symmetric shapes.

Rotation of Kernel, Feature Map, and Image The rotation of kernel, feature map, and image are three different fashions for capturing multi-view context. We focus on the kernel rotation, which enables an accurate control of the views for each voxel. Other fashions manipulate all voxels simultaneously in the feature map or scene. Given a degree for globally rotating the feature map or scene, the rotated view for a voxel may be within or beyond the given degree, thus yielding uncontrollable multi-view information. These schemes can incorporate. In Table 6, we investigate the rotation of kernel, feature map, and image with or without our CVSformer that controls the kernel rotation. Each scheme with CVSformer achieves better performance.

In addition, we consider the effects of random rotation, which produces unreasonable views of objects (e.g., the upside-down beds and chairs), thus harming the completion of the objects in normal views. We randomly choose three different rotation degrees from the range of [0,π]. The random rotation degrades IoU and mIoU from 73.7%, 52.6% to 73.0%, 51.5%.

Rotation	w/ CVSformer		w/o CVSformer	
	IoU(%)	mIoU(%)	IoU(%)	mIoU(%)
kernel	**73.7**	**52.6**	73.2	51.8
feature map	**73.5**	**52.1**	73.1	51.0
image	**72.4**	**46.4**	72.3	46.2

Table 6. Various rotation fashions. Performances on NYU.

Effectiveness of Position Embedding We perform experiments to show the effectiveness of position embedding in our CVSfomer. As shown in Table 7, we consider the inputs as a sequence of patches and employ single-dimensional learnable positional embedding to learn the cross-view feature maps. The effectiveness of positional embedding is evidenced by the performances improvement (0.8% IoU, 0.4% mIoU) when we use the positional embedding to provide geometric information.

Depth of Transformer Encoder In Table 8, we change the depth of transformer encoder for learning view tokens. We increase the depth from 1 to 8. A modest depth (e.g., 2) is sufficient for summarizing global semantic information from the voxelized volumes. It balances both performances and computation. In comparison, the view tokens learned by shallower encoder (e.g., when depth is set to 1) contain less semantic and geometric information. Meanwhile, it is difficult to optimize too deep encoder (e.g., 5 and 8), which degrades the performances.

Different Strategies of Learning View Tokens In Table 9, we experiment with different strategies for learning view tokens. We illustrate different strategies in Figure 7. We first employ a MLP module to learn the view tokens, achieving

Position Embedding	IoU(%)	mIoU(%)
×	72.9	52.2
✓	**73.7**	**52.6**

Table 7. Ablation study of position embedding in CVTr. The performances are evaluated on NYU.

Transformer Encoder	IoU(%)	mIoU(%)	Params
1	73.3	52	2.49G
2	**73.7**	**52.6**	2.57G
5	72.9	52.2	2.80G
8	73	52.6	2.95G

Table 8. Various depth of transformer Encoder in CVTr. The performances are evaluated on NYU.

(72.6% IoU, 51.8% mIoU). MLP loses local geometric details, which is sensitive to the voxel-wise SSC.

As reported in the second and third rows, we use the regular 3D convolution and self-attention to learn the view tokens, respectively. Self-attention slightly outperforms the regular 3D convolution, because it is good at capturing long-range dependencies of the whole scene. Yet, it eliminates the configuration of the voxelized structure, which is respected by the regular 3D convolution. The importance of respecting the voxelized format is evidenced by the performances improvement up to (73.7% IoU, 52.6% mIoU) when we combine 3D convolution and self-attention for learning the view tokens.

Different Fusion Schemes in CVTr In Table 10, we compare different schemes of fusing the synthetic-view feature maps in CVTr. Figure 8 shows the details of different schemes. We concatenate all synthetic-view feature maps, which are fed into self-attention for computing an augmented-view feature map (see "all fusion"). We further try to fuse the synthetic-view feature maps hierarchically (see "hierarchical fusion"). The hierarchical fusion takes into account the high correlation of adjacent angles but needing expensive computation.

In the third scheme (see "all-for-one w.r.t. feature maps"), we concatenate all of the synthetic-view feature maps to enhance each synthetic-view feature map. This scheme yields performances (72.9% IoU, 51.6% mIoU) worse than the first scheme. It evidences that high-dimensional synthetic-view feature maps contain redundant information for distracting the information fusion between multiple views.

Our cross-view fusion (see "all-for-one w.r.t. tokens") reduces the computational effort. The cross-view fusion refines the semantic and geometric information contained in the relatively low-dimensional view tokens, outperforming other alternatives.

Extend The Method to The Outdoor Scenarios We evaluate CVSformer on the SemanticKITTI [1] dataset in the Table 11. CVSformer outperforms the recent methods UD-

View Token	IoU(%)	mIoU(%)
MLP	72.2	51.5
convolution	72.6	51.8
self-attention	72.7	52.0
convolution + self-attention	**73.7**	**52.6**

Table 9. Various strategies for learning view tokens. The performances are evaluated on NYU.

Fusion Scheme	IoU(%)	mIoU(%)
all fusion	73.0	52.3
hierarchical fusion	73.6	52.4
all-for-one w.r.t. feature maps	72.9	51.6
all-for-one w.r.t. tokens	**73.7**	**52.6**

Table 10. Various fusion schemes in CVTr. The performances are evaluated on NYU.

Net [10], MonoScene [3], and OccDepth [8], which also rely on low-resolution voxels for semantic scene completion. From another direction of research, some methods (e.g., S3CNet [4]) leverage point clouds with more detailed 3D information to improve the completion results on the outdoor dataset. This direction motivates us to extend CVSformer in the future, allowing CVSformer to learn richer multi-view information from point clouds.

Method	MonoScene	OccDepth	UDNet	CVSformer	S3CNet
mIoU(%)	11.1	15.9	19.5	20.7	29.5

Table 11. We compare CVSformer with recent methods on the test set of SemanticKITTI.

3. Visualization Results

We provide more visualization results on both NYU [9] and NYUCAD [5] datasets in Figure 9 and Figure 10, respectively. The compared methods are DDRNet [7], AICNet [6], SISNet [2], and CVSformer. CVSformer achieves high quality of visual results on SSC.

4. Core Code of CVSformer

We show some of the core code about MVFS and CVTr in following. First, the overall framework of our CVSFormer is shown below.

```
class Network(nn.Module):
    '''
    Args:
        cls_num (int): Numer of classes.
            Default: 12
        dim (int): Number of channels.
            Default: 256
        norm_layer: Normalization layer.
            Default: torch.nn.BatchNorm3d
        bn_m: BN momentum. Default: 0.1
    '''
    def __init__(self, cls_num, dim,
        norm_layer, bn_m):
        super(Network, self).__init__()
        self.segProj = SegProjection()
        self.seg_blks = seg_Blocks()
        self.tsdf_blks = TSDF_Blocks()
        self.encoder = Encoder(dim=dim,
            norm_layer=norm_layer, bn_m=bn_m)
        self.decoder = Decoder(cls_num,
            dim=dim, norm_layer=norm_layer,
            bn_m=bn_m)
        self.view_num = 4 # view number
        self.MVSynthesis =
            MVSynthesisBlock(dim=dim,
            view_num = self.view_num)
        self.m = 75 # resolution of view
            tokens
        self.CVTransformer =
            CVTransformerBlock(dim=dim,
            view_num=self.view_num, m=self.m)

    def forward(self, seg, tsdf, mapp):
        '''
        Iuputs:
           seg: 2D Semantic Segmentation
           tsdf: voxelized TSDF volume
           mapp: 2D->3D mapping relationship
        '''
        # project 2D seg result to 3D
        segRes = self.segProj(seg, mapp)
        seg_fea = self.seg_blks(segRes)
        tsdf_fea = self.tsdf_blks(tsdf)
        seg_fea = seg_fea + tsdf_fea
        ori_fea = self.encoder(seg_fea)

        ''' Multi-View Feature Synthesis '''
        multi_fea = self.MVSynthesis(ori_fea)
        ''' Cross-View Transformer '''
        aug_fea =
            self.CVTransformer(multi_fea)

        out = self.decoder(aug_fea)
        return out
```

Second, we show the datails of MVFS below.

```
class MVSynthesisBlock(nn.Module):
    def __init__(self,
        dim=256,num_branches=4):
        super(MVSynthesisBlock,self)
        .__init__()
        self.business_layer = []
        self.rotation = Rotation()
        self.num_branches= num_branches
        self.rotationModule =
            KernelRotate()
        self.rotationConv1 =
            nn.Conv3d(dim, dim,
```

```
            kernel_size=3, padding=0,
            stride=3)
        self.rotationConv2 =
            nn.Conv3d(dim, dim,
            kernel_size=3, padding=0,
            stride=3)
        self.rotationConv3 =
            nn.Conv3d(dim, dim,
            kernel_size=3, padding=0,
            stride=3)
    def forward(self, ori_fea):
        b_s2, c_s2, d_s2, h_s2, w_s2 =
            ori_fea.shape
        points_base_45, points_base_90,
            points_base_135, point_base0 =
            self.rotation(ori_fea)
        weights_45 = compute_weights(
           points_base_45)
        weights_90 = compute_weights(
           points_base_90)
        weights_135 = compute_weights(
           points_base_135)
        ori_fea_r45 =
            self.rotationModule(ori_fea,
            points_base_45.cuda(),
            weights_45.cuda()).reshape(
           b_s2, c_s2, 3, 3, -1)
        ori_fea_r90 =
            self.rotationModule(ori_fea,
            points_base_90.cuda(),
            weights_90.cuda()).reshape(
           b_s2, c_s2, 3, 3, -1)
        ori_fea_r135 =
            self.rotationModule(ori_fea,
            points_base_135.cuda(),
            weights_135.cuda()).reshape(
           b_s2, c_s2, 3, 3, -1)
        ori_fea_r45 =
            self.rotationConv1(ori_fea_r45)
        .reshape(b_s2, c_s2, d_s2, h_s2,
            w_s2)
        ori_fea_r90 =
            self.rotationConv2(ori_fea_r90)
        .reshape(b_s2, c_s2, d_s2, h_s2,
            w_s2)
        ori_fea_r135 = self.rotationConv3(
           ori_fea_r135).reshape(b_s2,
               c_s2, d_s2, h_s2, w_s2)
        return [ori_fea,
            ori_fea_r45,ori_fea_r90,
        ori_fea_r135] # 4 view

# MVSynthesis pre before entering cuda
class Rotation(nn.Module):
    def __init__(self):
        super(Rotation, self).__init__()

    def first_neighbor(self, x_y_z_offset
, kernel_size):
        x = int(x_y_z_offset[0][0][0]
         .item())
        y = int(x_y_z_offset[0][0][1]
         .item())
        z = int(x_y_z_offset[0][0][2]
         .item())
        neighbor=torch.zeros(kernel_size*
        kernel_size*kernel_size,3)
        neighbor_num=0
        # for neighbor_num in range(27):
        for neighbor_x in range(x-1, x+2):
            for neighbor_y in range(y-1,
                y+2):
               for neighbor_z in range(z-1,
                z+2):
                  neighbor[neighbor_num]=
                  torch.tensor([neighbor_x,
                  neighbor_y, neighbor_z])
                  neighbor_num =
                  neighbor_num+1
        return neighbor

    def forward(self,featuremap):
        length =
            np.arange(featuremap.shape[2])
        width =
            np.arange(featuremap.shape[3])
        height =
            np.arange(featuremap.shape[4])
        a,b,c = np.meshgrid(length,
            width, height
        , indexing="ij")
        x_offset =
            torch.FloatTensor(a).view(-1,
            1)
        y_offset =
            torch.FloatTensor(b).view(-1,
            1)
        z_offset =
            torch.FloatTensor(c).view(-1,
            1)
        x_y_z_offset =
            torch.cat((x_offset, y_offset
        , z_offset), 1).view(-1,
            3).unsqueeze(0).
        repeat(featuremap.shape[0],1,1)
        kernel_size=3
        firstNeighbor =
        self.first_neighbor(x_y_z_offset,
            kernel_size)

        # x-axis rotation matrix
        rotation_matrix_45 =
            torch.tensor([
           [torch.cos(torch.tensor(
               np.pi/4)),
          torch.sin(torch.tensor(
```

```
            np.pi/4)),
          0],[-torch.sin(torch.tensor(
          np.pi/4)),
              torch.cos(torch.tensor(
             np.pi/4)),0],[0,0,1]])

      return neighbor_rotation_45,
          firstNeighbor

def compute_weights(points):
   # params: points 27*3
   # len_d,len_h,len_w,len_l =
       points.shape
   len_points, len_xyz = points.shape
   weights = torch.zeros([len_points,
       8], dtype=torch.float32)
   for i in range(len_points):
      x, y, z = points[i]
      fx = torch.floor(x)
      fy = torch.floor(y)
      fz = torch.floor(z)

      xd = x - fx
      yd = y - fy
      zd = z - fz

      weights[i][0] =
          (1-xd)*(1-yd)*(1-zd)
      weights[i][1] = (1-xd)*(1-yd)*zd
      weights[i][2] = (1-xd)*yd*(1-zd)
      weights[i][3] = (1-xd)*yd*zd
      weights[i][4] = xd*(1-yd)*(1-zd)
      weights[i][5] = xd*(1-yd)*zd
      weights[i][6] = xd*yd*(1-zd)
      weights[i][7] = xd*yd*zd
   return weights

class KernelRotate(nn.Module):
   def __init__(self):
      super(KernelRotate,
          self).__init__()
   def forward(self, features,
       points_base, weights):
      return KernelRotateFunction.
      apply(features, points_base,
          weights)

class KernelRotateFunction(torch.
autograd.Function):
   @staticmethod
   def forward(ctx, features,
       points_base, weights):
      features_output =
          kernel_rotation.forward(
         features, points_base, weights)
      ctx.save_for_backward(features,
          points_base
      , weights)
      return features_output

   @staticmethod
   def backward(ctx, grad_output):
      features, points_base, weights =
      ctx.saved_tensors
      grad = kernel_rotation.backward(
          features,
      grad_output, points_base, weights)
      return grad, None, None

# crucial code about MMVSynthesis in
    cuda

__global__ void
    GetRotatedFeaturesForwardKernel(
   int len_d,
   int len_h,
   int len_w,
   const float *__restrict__ features,
   const float *__restrict__ weights,
   const float *__restrict__
       points_base,
   float *__restrict__ features_rotated
   ){
   int batch_index = blockIdx.x;
   int channel_index = blockIdx.y;
   int index   = threadIdx.x;
   int stride  = blockDim.x;

   features += (batch_index * gridDim.y
       + channel_index) * len_d * len_h
       * len_w;
   features_rotated += (batch_index *
       gridDim.y + channel_index) *
       len_d * 3 * len_h * 3 * len_w * 3;

   for (int j = index; j < len_d *
       len_h * len_w; j += stride){
      // j: the index of the original
          feature
      // neightbor: <0 or >len_d *
          len_h * len_w-1 -> 0
      // neighbor feature

      //center coordinates
      int f_d = j / (len_h * len_w);
      int f_h = j % (len_h * len_w) /
          len_w;
      int f_w = j % len_w;
      //neightbors coordinates
      float * neighbors = new float[27
          * 3];
      for (int k = 0; k < 27; k ++){
         neighbors[k * 3 + 0] = f_d +
             points_base[k * 3 + 0];
         neighbors[k * 3 + 1] = f_h +
             points_base[k * 3 + 1];
```

```
            neighbors[k * 3 + 2] = f_w +
                points_base[k * 3 + 2];
        }

        int len_features = len_d * len_h
            * len_w;

        for (int m = 0; m < 27; m ++){
            int * indexs =
                vtx_indexs(neighbors[m * 3
                + 0], neighbors[m * 3 + 1],
                neighbors[m * 3 + 2],
                len_d, len_h, len_w);
            //weight interpolation
            features_rotated[j * 27 + m] =
                get_feature(features,
                indexs[0], len_features) *
                weights[m * 8 + 0]
            + get_feature(features,
                indexs[1], len_features) *
                weights[m * 8 + 1]
             + get_feature(features,
                 indexs[2], len_features) *
                 weights[m * 8 + 2]
             + get_feature(features,
                 indexs[3], len_features) *
                 weights[m * 8 + 3]
             + get_feature(features,
                 indexs[4], len_features) *
                 weights[m * 8 + 4]
             + get_feature(features,
                 indexs[5], len_features) *
                 weights[m * 8 + 5]
             + get_feature(features,
                 indexs[6], len_features) *
                 weights[m * 8 + 6]
             + get_feature(features,
                 indexs[7], len_features) *
                 weights[m * 8 + 7];
            delete indexs;
        }
        delete neighbors;
    }
}
```

Finally, we show the datails of CVTr below.

```
class CVTransformerBlock(nn.Module):
    '''
    Args:
        dim (int): Number of channels.
            Default: 256
        view_num (int): View number of
            Multi-View Feature Synthesis.
            Default: 4
        m (int): Spatial Resolution of View
            Tokens. Default: 75
    '''

    def __init__(self, dim=256, view_num=4,
        m=75):
        super(CVTransformerBlock,
            self).__init__()
        self.view_num = view_num
        self.m = m
        self.chan_pro =
            nn.Conv1d(in_channel=dim*view_num,
            out_channel=dim,
            kernel_size=3,padding=1)
        self.chan_pro_1 =
            nn.Conv3d(in_channel=dim*view_num,
            out_channel=dim,
            kernel_size=3,padding=1)

        self.sa_blocks = nn.ModuleList()
        for _ in range(self.view_num):
            tmp = [SA_Block(
                embed_dim=dim, depth_SA=2,
                    num_heads=8, qkv_bias=True,
                    m=self.m)]
            self.sa_blocks.append(
                nn.Sequential(*tmp))
        self.conv_blocks = nn.ModuleList()
        for _ in range(self.view_num):
            tmp = [Conv_Block(embed_dim=dim,
                voxel_size=15*9*15, m=self.m)]
            self.conv_blocks.append(
                nn.Sequential(*tmp))
        self.cross_attn_blks =
            nn.ModuleList()
        for _ in range(self.view_num):
            tmp =
                [NonCrossAttn_Block(embed_dim
                = dim)]
            self.cross_attn_blks.append(
                nn.Sequential(*tmp))

        self.view_token = nn.ParameterList(
            [nn.Parameter(torch.zeros(1,
                self.m, dim)) for _ in
                range(self.view_num)])
        self.pos_embed = nn.ParameterList(
            [nn.Parameter(torch.zeros(1,
                self.m+15*9*15, dim)) for _ in
                range(self.view_num)])
        for i in range(self.view_num):
            trunc_normal_(self.pos_embed[i],
                std=.02)
            trunc_normal_(self.view_token[i],
                std=.02)

    def forward(self, x):
        '''
        Iuput:
            x (list): list of all
                synthesis-view feature maps.
                Default:
```

```
            [(B,C,D,H,W),...,(B,C,D,H,W)]
        Output:
            y: the augmented-view feature
                map. Default: (B,C,D,H,W)
        '''
        B, C ,D, H, W= x[0].shape
        # concat view tokens and patchs
        feature_li = []
        x_ = [x[i].flatten(2).transpose(1,2)
            for i in range(self.view_num)]
        for i in range(self.view_num):
            tmp = x_[i]
            vts =
                self.view_token[i].expand(B,
                -1, -1)
            tmp = torch.cat((vts, tmp), dim=1)
            tmp = tmp + self.pos_embed[i]
            feature_li.append(tmp)

        # self attention to get view tokens
        fea_li = [blk(t) for t, blk in
            zip(feature_li, self.sa_blocks)]
        vt_sa = [fea_li[i][:, 0:self.m ,...]
            for i in range(self.view_num)]
        # convolution to get view tokens
        fea_li = [blk(t) for t, blk in
            zip(feature_li, self.conv_blocks)]
        vt_conv = [fea_li[i][:, 0:self.m
            ,...] for i in
            range(self.view_num)]
        vts = [vt_sa[i] + vt_conv[i] for i
            in range(self.view_num)]
        vt_g = torch.cat((vts[0], vts[1],
            vts[2], vts[3]),
            dim=2).transpose(1,2)
        vt_g =
            self.chan_pro(vt_g).transpose(1,2)

        # cross attention
        outs = []
        for i in range(self.view_num):
            tmp = self.cross_attn_blks[i](
                x_[i], vt_g)
            outs.append(tmp)
        y = torch.cat((outs[0], outs[1],
            outs[2],
            outs[3]),dim=2).transpose(1,2)
        y = self.chan_pro_1(y).reshape(
            B, C ,D, H, W)
        y = y + x[0] # skip conn
        return y
```

References

[1] Jens Behley, Martin Garbade, Andres Milioto, Jan Quenzel, Sven Behnke, Cyrill Stachniss, and Jurgen Gall. Semantickitti: A dataset for semantic scene understanding of lidar sequences. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9297–9307, 2019. 7

[2] Yingjie Cai, Xuesong Chen, Chao Zhang, Kwan-Yee Lin, Xiaogang Wang, and Hongsheng Li. Semantic scene completion via integrating instances and scene in-the-loop. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 324–333, 2021. 1, 8

[3] Anh-Quan Cao and Raoul de Charette. Monoscene: Monocular 3d semantic scene completion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3991–4001, 2022. 8

[4] Ran Cheng, Christopher Agia, Yuan Ren, Xinhai Li, and Liu Bingbing. S3cnet: A sparse semantic scene completion network for lidar point clouds. In *Conference on Robot Learning*, pages 2148–2161. PMLR, 2021. 8

[5] Yu-Xiao Guo and Xin Tong. View-volume network for semantic scene completion from a single depth image. *arXiv preprint arXiv:1806.05361*, 2018. 1, 8

[6] Jie Li, Kai Han, Peng Wang, Yu Liu, and Xia Yuan. Anisotropic convolutional networks for 3d semantic scene completion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3351–3359, 2020. 1, 8

[7] Jie Li, Yu Liu, Dong Gong, Qinfeng Shi, Xia Yuan, Chunxia Zhao, and Ian Reid. Rgbd based dimensional decomposition residual network for 3d semantic scene completion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7693–7702, 2019. 1, 8

[8] Ruihang Miao, Weizhou Liu, Mingrui Chen, Zheng Gong, Weixin Xu, Chen Hu, and Shuchang Zhou. Occdepth: A depth-aware method for 3d semantic scene completion. *arXiv preprint arXiv:2302.13540*, 2023. 8

[9] Nathan Silberman, Derek Hoiem, Pushmeet Kohli, and Rob Fergus. Indoor segmentation and support inference from rgbd images. In *European conference on computer vision*, pages 746–760. Springer, 2012. 1, 8

[10] Hao Zou, Xuemeng Yang, Tianxin Huang, Chujuan Zhang, Yong Liu, Wanlong Li, Feng Wen, and Hongbo Zhang. Up-to-down network: Fusing multi-scale context for 3d semantic scene completion. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 16–23. IEEE, 2021. 8